STUDIA INFORMATICA

Formerly: *Zeszyty Naukowe Politechniki Śląskiej, seria INFORMATYKA*
Quarterly

Volume 31, Number 4B (93)

Sebastian DEOROWICZ

SEKWENCYJNE I RÓWNOLEGŁE ALGORYTMY ZNAJDOWANIA PODCIĄGÓW

Silesian University of Technology Press
Gliwice 2010

STUDIA INFORMATICA **Volume 31, Number 4B (93)**
Formerly: *Zeszyty Naukowe Politechniki Śląskiej, seria INFORMATYKA* Nr kol. 1835

Editor in Chief

Dr. Marcin SKOWRONEK
Silesian University of Technology
Gliwice, Poland

Editorial Board

Dr. Mauro CISLAGHI
Project Automation
Monza, Italy

Prof. Bernard COURTOIS
Lab. TIMA
Grenoble, France

Prof. Tadeusz CZACHÓRSKI
Silesian University of Technology
Gliwice, Poland

Prof. Jean-Michel FOURNEAU
Université de Versailles - St. Quentin
Versailles, France

Prof. Jurij KOROSTIL
IPME NAN Ukraina
Kiev, Ukraine

Dr. George P. KOWALCZYK
Networks Integrators Associates, President
Parkland, USA

Prof. Stanisław KOZIELSKI
Silesian University of Technology
Gliwice, Poland

Prof. Peter NEUMANN
Otto-von-Guericke Universität
Barleben, Germany

Prof. Olgierd A. PALUSINSKI
University of Arizona
Tucson, USA

Prof. Svetlana V. PROKOPCHINA
Scientific Research Institute BITIS
Sankt-Petersburg, Russia

Prof. Karl REISS
Universität Karlsruhe
Karlsruhe, Germany

Prof. Jean-Marc TOULOTTE
Université des Sciences et Technologies de Lille
Villeneuve d'Ascq, France

Prof. Sarma B. K. VRUDHULA
University of Arizona
Tucson, USA

Prof. Hamid VAKILZADIAN
University of Nebraska-Lincoln
Lincoln, USA

Prof. Stefan WĘGRZYN
Silesian University of Technology
Gliwice, Poland

Prof. Adam WOLISZ
Technical University of Berlin
Berlin, Germany

STUDIA INFORMATICA is indexed in INSPEC/IEE (London, United Kingdom)

© Copyright by Silesian University of Technology Press, Gliwice 2010
PL ISSN 0208-7286, QUARTERLY
Printed in Poland

ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ

KOLEGIUM REDAKCYJNE
REDAKTOR NACZELNY – Prof. dr hab. inż. Andrzej Buchacz
REDAKTOR DZIAŁU – Dr inż. Marcin Skowronek
SEKRETARZ REDAKCJI – Mgr Elżbieta Leśko

Monice, Pawłowi

Spis treści

Wykaz najważniejszych symboli i oznaczeń 13

Wykaz algorytmów omawianych w pracy 15

Wstęp 21

1. Pojęcia wstępne 27

1.1. Definicje . . . 27

1.2. Model obliczeniowy . . . 28

1.3. Obliczenia ogólnego przeznaczenia na procesorach graficznych . . . 29

CZĘŚĆ I. PODCIĄGI ROSNĄCE 33

2. Najdłuższy podciąg rosnący 35

2.1. Wprowadzenie . . . 35

2.2. Dodatkowe definicje i przyjęte założenia . . . 36

2.3. Długość najdłuższego podciągu rosnącego . . . 36

2.4. Algorytmy . . . 36

2.4.1. Tableau Younga . . . 36

2.4.2. Pokrycie zachłanne . . . 38

2.5. Podsumowanie . . . 40

3. Minimalny i maksymalny najdłuższy podciąg rosnący 42

3.1. Wprowadzenie . . . 42

3.2. Wspólne idee algorytmów . . . 43

3.3. Wariant o minimalnej wysokości . . . 44

3.4. Wariant o maksymalnej wysokości . . . 47

3.5. Wariant o minimalnej szerokości . . . 49

3.6. Wariant o maksymalnej szerokości . . . 50

3.7. Wariant o minimalnej sumie . . . 52

3.8. Wariant o maksymalnej sumie . . . 54

3.9. Analiza złożoności czasowych i pamięciowych . . . 55

3.10. Podsumowanie . . . 56

4. Najdłuższy podciąg rosnący o zadanym pochyleniu **58**
4.1. Wprowadzenie 58
4.2. Algorytm 59
4.3. Podsumowanie 65

5. Najdłuższy cykliczny podciąg rosnący **66**
5.1. Wprowadzenie 66
5.2. Algorytmy 67
5.2.1. Podstawowe koncepcje 67
5.2.2. Reprezentacja pokrycia za pomocą list 72
5.2.3. Reprezentacja pokrycia za pomocą drzew zrównoważonych 73
5.2.4. Reprezentacja pokrycia za pomocą list drzew zrównoważonych 75
5.3. Podsumowanie 77

6. Najdłuższy podciąg rosnący w przesuwającym się oknie **78**
6.1. Wprowadzenie 78
6.2. Algorytmy 79
6.2.1. Podstawowe koncepcje 79
6.2.2. Reprezentacja pokrycia za pomocą list 83
6.2.3. Reprezentacja pokrycia za pomocą drzew zrównoważonych 84
6.2.4. Reprezentacja pokrycia za pomocą list drzew zrównoważonych 84
6.3. Podsumowanie 86

CZĘŚĆ II. WSPÓLNE PODCIĄGI **87**

7. Najdłuższy wspólny podciąg **89**
7.1. Wprowadzenie 89
7.2. Dodatkowe definicje 91
7.3. Algorytmy sekwencyjne i równoległości bitowej 92
7.3.1. Programowanie dynamiczne 92
7.3.2. Algorytm Hunta–Szymanskiego 93
7.3.3. Algorytm Hirschberga 96
7.3.4. Algorytm równoległości bitowej 98
7.3.5. Algorytm równoległości bitowej – wariant Hirschberga 101
7.3.6. Związek z problemem najdłuższego podciągu rosnącego 104
7.3.7. Inne wybrane algorytmy 105
7.4. Uogólnienie na przypadek wielu ciągów 107

7.5. Algorytmy równoległe . . . 109
7.6. Algorytmy dla procesorów graficznych . . . 110
7.6.1. Wprowadzenie . . . 110
7.6.2. Podstawowe idee . . . 112
7.6.3. Algorytm programowania dynamicznego . . . 114
7.6.4. Algorytm równoległości bitowej . . . 118
7.6.5. Wyniki eksperymentalne . . . 122
7.7. Podsumowanie . . . 126

8. Najdłuższy wspólny podciąg niezmienniczy względem transpozycji 128
8.1. Wprowadzenie . . . 128
8.2. Algorytm pudełkowy . . . 129
8.3. Algorytm oparty na metodzie Hunta–Szymanskiego . . . 132
8.3.1. Idea podstawowa . . . 132
8.3.2. Algorytm . . . 135
8.3.3. Wyniki eksperymentalne . . . 139
8.4. Algorytm hybrydowy . . . 141
8.4.1. Algorytm . . . 141
8.4.2. Wyniki eksperymentalne . . . 143
8.5. Algorytm równoległy dla procesorów graficznych . . . 147
8.5.1. Algorytm . . . 147
8.5.2. Wyniki eksperymentalne . . . 149
8.6. Podsumowanie . . . 150

9. Najdłuższy ukierunkowany wspólny podciąg 152
9.1. Wprowadzenie . . . 152
9.2. Algorytm oparty na metodzie Hunta–Szymanskiego . . . 155
9.2.1. Podstawowa idea . . . 155
9.2.2. Szczegóły implementacyjne i analiza złożoności . . . 158
9.2.3. Zastosowanie techniki punktów wejścia-wyjścia . . . 161
9.2.4. Wyniki eksperymentalne . . . 161
9.3. Algorytm równoległości bitowej . . . 167
9.3.1. Podstawy . . . 167
9.3.2. Algorytm paskowy . . . 169
9.3.3. Wyznaczanie ciągu wynikowego . . . 177
9.3.4. Szczegóły implementacyjne i analiza złożoności . . . 178
9.3.5. Wyniki eksperymentalne . . . 182

9.4. Algorytm równoległy dla procesorów graficznych 184
9.4.1. Algorytm 184
9.4.2. Wyniki eksperymentalne 189
9.5. Podsumowanie 190

10. Najdłuższy scalony wspólny podciąg **192**
10.1. Wprowadzenie 192
10.2. Algorytm równoległości bitowej 195
10.2.1. Podstawowe idee 195
10.2.2. Algorytm 197
10.2.3. Szczegóły implementacyjne 200
10.3. Wyniki eksperymentalne 201
10.4. Podsumowanie 203

Wnioski końcowe **207**

Bibliografia **210**

Spis rysunków **226**

Spis tabel **233**

Streszczenie **234**

Skorowidz **236**

CONTENTS

List of Most Important Symbols and Abbreviations 13

List of Algorithms Discussed in This Work 15

Preface 21

1. Basic concepts 27
1.1. Definitions 27
1.2. Computational model 28
1.3. General processing on graphical processing units 29

PART I. INCREASING SUBSEQUENCES 33

2. Longest increasing subsequence 35
2.1. Introduction 35
2.2. Additional definitions and made assumptions 36
2.3. Longest increasing subsequence length 36
2.4. Algorithms 36
2.4.1. Young tableaux 36
2.4.2. Greedy cover 38
2.5. Summary 40

3. Minimal and maximal longest increasing subsequence 42
3.1. Introduction 42
3.2. Common ideas of the algorithms 43
3.3. Minimal height variant 44
3.4. Maximal height variant 47
3.5. Minimal width variant 49
3.6. Maximal width variant 50
3.7. Minimal sum variant 52
3.8. Maximal sum variant 54
3.9. Time and space complexity analysis 55
3.10. Summary 56

4. Slope-constrained longest increasing subsequence **58**

4.1. Introduction . . . 58
4.2. An algorithm . . . 59
4.3. Summary . . . 65

5. Longest increasing cyclic subsequence **66**

5.1. Introduction . . . 66
5.2. Algorithms . . . 67
5.2.1. Basic concepts . . . 67
5.2.2. List-based cover representation . . . 72
5.2.3. Balanced trees-based cover representation . . . 73
5.2.4. List of balanced trees-based cover representation . . . 75
5.3. Summary . . . 77

6. Longest increasing subsequence in sliding window **78**

6.1. Introduction . . . 78
6.2. Algorithms . . . 79
6.2.1. Basic concepts . . . 79
6.2.2. List-based cover representation . . . 83
6.2.3. Balanced trees-based cover representation . . . 84
6.2.4. List of balanced trees-based cover representation . . . 84
6.3. Summary . . . 86

PART II. COMMON SUBSEQUENCES **87**

7. Longest common subsequence **89**

7.1. Introduction . . . 89
7.2. Additional definitions . . . 91
7.3. Sequential and bit-parallel algorithms . . . 92
7.3.1. Dynamic programming . . . 92
7.3.2. Hunt–Szymanski algorithm . . . 93
7.3.3. Hirschberg algorithm . . . 96
7.3.4. Bit-parallel algorithm . . . 98
7.3.5. Bit-parallel algorithm – Hirschberg variant . . . 101
7.3.6. Relation with longest increasing subsequence problem . . . 104
7.3.7. Other selected algorithms . . . 105
7.4. Generalisation for many sequences . . . 107

7.5. Parallel algorithms . . . 109
7.6. Algorithms for graphical processors . . . 110
7.6.1. Introduction . . . 110
7.6.2. Main ideas . . . 112
7.6.3. Dynamic programming algorithm . . . 114
7.6.4. Bit-parallel algorithm . . . 118
7.6.5. Experimental results . . . 122
7.7. Summary . . . 126

8. Longest transposition-invariant common subsequence 128
8.1. Introduction . . . 128
8.2. Box algorithm . . . 129
8.3. Algorithm based on Hunt–Szymanski method . . . 132
8.3.1. Main idea . . . 132
8.3.2. Algorithm . . . 135
8.3.3. Experimental results . . . 139
8.4. Hybrid algorithm . . . 141
8.4.1. Algorithm . . . 141
8.4.2. Experimental results . . . 143
8.5. Parallel algorithm for graphical processors . . . 147
8.5.1. Algorithm . . . 147
8.5.2. Experimental results . . . 149
8.6. Summary . . . 150

9. Constrained longest common subsequence 152
9.1. Introduction . . . 152
9.2. Algorithm based on Hunt–Szymanski method . . . 155
9.2.1. Main idea . . . 155
9.2.2. Implementation details and complexity analysis . . . 158
9.2.3. Application of the entry-exit points technique . . . 161
9.2.4. Experimental results . . . 161
9.3. Bit-parallel algorithm . . . 167
9.3.1. Basics . . . 167
9.3.2. Strip algorithm . . . 169
9.3.3. Computation of the output subsequence . . . 177
9.3.4. Implementation details and complexity analysis . . . 178
9.3.5. Experimental results . . . 182

9.4. Parallel algorithm for graphical processors 184
9.4.1. Algorithm 184
9.4.2. Experimental results 189
9.5. Summary 190

10. Merged longest common subsequence **192**
10.1. Introduction 192
10.2. Bit-parallel algorithm 195
10.2.1. Main ideas 195
10.2.2. Algorithm 197
10.2.3. Implementation details 200
10.3. Experimental results 201
10.4. Summary 203

Conclusions **207**

List of Figures **230**

Abstract **235**

Index **236**

Wykaz najważniejszych symboli i oznaczeń

Symbol/ Oznaczenie	Opis	Definicja (str.)
A	ciąg wejściowy	
a_i	symbole ciągu A	
B	ciąg wejściowy	
b_j	symbole ciągu B	
b_{h}	wysokość pudełka w algorytmach równoległych dla procesorów GPU	109
$b_{\mathrm{h}}^{\mathrm{cpu}}$	wysokość pudełka w algorytmach równoległych dla procesorów CPU	124
b_{w}	szerokość pudełka w algorytmach równoległych dla procesorów GPU	109
$b_{\mathrm{w}}^{\mathrm{cpu}}$	szerokość pudełka w algorytmach równoległych dla procesorów CPU	124
Γ	pokrycie zachłanne ciągu	38
Γ^*	pokrycie zachłanne rozszerzone ciągu	43
C_x	licznik bitów o wartości 1 w odpowiednim wektorze V_x (problem CLCS)	170
D	liczba dopasowań dominujących dla A i B	135
d	liczba dopasowań dla A i B	95
D^*	liczba dopasowań dominujących dla $A+t$ i B dla wszystkich transpozycji t	135
D^{sm}	sumaryczna liczba dopasowań silnych (problem CLCS)	180
d_k^{sm}	liczba dopasowań silnych dla ustalonego k (problem CLCS)	179
E	odczyt według pokrycia (problemy LICS, LISW)	67
η_1	liczba multiprocesorów składających się na procesor GPU	114
η_2	liczba rdzeni wchodzących w skład multiprocesora (GPU)	114
G	macierz wektorów zmian (problem MerLCS)	195
g	współczynnik pochylenia (problem SLIS)	58
h	ranga elementu	196
i, j, k	indeksy elementów w ciągach	
J^0	wektor zawierający liczbę wystąpień każdego symbolu alfabetu w ciągu A	158
J^1	macierz zawierająca listy wystąpień każdego symbolu alfabetu w ciągu A	158
ℓ	długość podciągu wynikowego S (wszystkie problemy)	
M	macierz programowania dynamicznego	92
m	długość ciągu B	
n	długość ciągu A	
n^{stop}	liczba punktów stopu (problemy LICS, LISW)	72
n_i^{stop}	liczba punktów stopu w i-tym oknie rozmiaru u (problem LISW)	83
P	ciąg ukierunkowujący (problem CLCS)	
p_i	symbol ciągu P	

Symbol/ Oznaczenie	Opis	Definicja (str.)
Ψ	projekcja punktu lub elementu	60
Q	struktura rozwiązująca problem poprzednika	94
r	długość ciągu P	
S	ciąg wynikowy (różne algorytmy)	
s	dowolny symbol alfabetu	
s_i	symbol w ciągu S	
Σ	alfabet	27
σ	rozmiar alfabetu	27
t	transpozycja (problem LCTS)	128
U, V, W	wektory bitowe	99
w	długość słowa komputerowego	27
w_c, w_g	długość słowa komputerowego procesorów CPU, GPU	27
Y	wektory masek bitowych w algorytmach równoległości bitowej	
Z	ciąg wejściowy dla problemu MerLCS	192
z_i	symbol w ciągu Z	
BP	metoda równoległości bitowej	
CLCS	najdłuższy ukierunkowany wspólny podciąg	152
DP	metoda programowania dynamicznego	
HS	metoda Hunta–Szymanskiego	
LCS	najdłuższy wspólny podciąg	92
LCTS	najdłuższy wspólny podciąg niezmienniczy względem transpozycji	128
LICS	najdłuższy cykliczny podciąg rosnący	66
LIS	najdłuższy podciąg rosnący	35
LISW	najdłuższy podciąg rosnący w przesuwającym się oknie	78
MerLCS	najdłuższy scalony wspólny podciąg	192

Wykaz algorytmów omawianych w pracy

W zamieszczonym poniżej wykazie umieszczono wszystkie algorytmy autorskie oraz algorytmy znane z literatury, których pseudokody znajdują się w niniejszej pracy.

Algorytmy autorskie

1. CLCS-BP-Length (str. 175)

Algorytm równoległości bitowej wyznaczający długość podciągu CLCS.

Złożoność czasowa (przypadek pesymistyczny): $O\left(\sqrt{nmrD^{\mathrm{sm}}} + n\left\lceil\frac{m}{w}\right\rceil r\right)$.

Złożoność czasowa (przypadek średni): $O\left(n\left\lceil\frac{m}{\min(\sigma,w)}\right\rceil r\right)$.

Złożoność pamięciowa: $O(\sigma\left\lceil\frac{m}{w}\right\rceil + \beta' + d_*^{\mathrm{sm}})$ słów.

Opublikowany w: *Fundamenta Informaticae* (2010) 99(4):409–433 [69].

2. CLCS-BP-Sequence (str. 178)

Algorytm równoległości bitowej wyznaczający podciąg CLCS na podstawie struktur danych utworzonych przez rozszerzony algorytm CLCS-BP-Length.

Złożoność czasowa: $\Theta(n)$.

Złożoność pamięciowa: $\Theta(n\left\lceil\frac{m}{w}\right\rceil r)$ słów.

Opublikowany w: *Fundamenta Informaticae* (2010) 99(4):409–433 [69].

3. CLCS-CUDA (str. 184)

Algorytm równoległy wyznaczający długość podciągu LCTS dla procesorów GPU.

Złożoność czasowa (w zależności od pewnych parametrów – szczegóły w tekście):

- $\Theta\left(nmr/\max\left(\eta_2, \min\left(\eta_1\eta_2, \frac{nmr}{D^{\mathrm{sm}}}, \eta_2 b_{\mathrm{h}}\right)\right)\right)$ lub
- $\Theta\left(nmr/\left(\max\left(\eta_2, \min\left(\frac{\eta_2 mr}{b_{\mathrm{h}}}, \frac{\eta_2 nr}{b_{\mathrm{w}}}, \frac{\eta_2 nm}{b_{\mathrm{w}}b_{\mathrm{h}}}, \frac{nmr}{D^{\mathrm{sm}}}, \eta_2 b_{\mathrm{h}}\right)\right)\right)\right)$.

Złożoność pamięciowa: $\Theta(nr + b_{\mathrm{w}}mr)$ słów.

Opublikowany w: *Software—Practice and Experience* (2010) 40(8):673–700 [71].

4. CLCS-HS-Length (str. 160)

Algorytm oparty na metodzie Hunta–Szymanskiego wyznaczający długość podciągu CLCS.

Złożoność czasowa: $O((m\ell + d)r + n)$.

Złożoność pamięciowa: $\Theta(d + \max(n, \sigma))$ słów.

Opublikowany w: *Theoretical and Applied Informatics* (2007) 19(2):91–102 [64].

5. CLCS-HS-Sequence (str. 161)

Algorytm oparty na metodzie Hunta-Szymanskiego wyznaczający podciąg CLCS.

Złożoność czasowa: $O((m\ell + d)r + n)$.

Złożoność pamięciowa: $\Theta(rd + \max(n, \sigma))$ słów.

Opublikowany w: *Theoretical and Applied Informatics* (2007) 19(2):91–102 [64].

6. LCS-BP-CUDA (str. 120)

Algorytm równoległy wyznaczający długość podciągu LCS metodą równoległości bitowej dla procesorów GPU.

Złożoność czasowa:

- wersja ze wstępnym wczytaniem wektorów masek: $\Theta\left(\frac{nm}{w_g} \times \frac{1}{\eta_2 \min\left(\min(n', m', \eta_1), \frac{b_h}{w_g}, \frac{b_w}{\sigma}\right)}\right)$,
- wersja z tablicą pośredniczącą: $\Theta\left(\frac{nm}{\eta_2 w_g}\right)$.

Złożoność pamięciowa: $\Theta\left(n + \sigma\left\lceil\frac{m}{w_g}\right\rceil\right)$ słów.

Opublikowany w: *Software—Practice and Experience* (2010) 40(8):673–700 [71].

7. LCS-DP-CUDA (str. 117)

Algorytm równoległy wyznaczający długość podciągu LCTS metodą programowania dynamicznego dla procesorów GPU.

Złożoność czasowa: $\Theta\left(\frac{nm}{\eta_2 \min\left(\frac{n}{b_w}, \frac{m}{b_h}, \eta_1, b_h\right)}\right)$.

Złożoność pamięciowa: $\Theta(n)$ słów.

Opublikowany w: *Software—Practice and Experience* (2010) 40(8):673–700 [71].

8. LCTS-CUDA (str. 148)

Algorytm równoległy wyznaczający długość podciągu LCTS dla procesorów GPU.

Złożoność czasowa: $\Theta\left(\frac{nm\sigma}{\eta_2 w_g}\right)$.

Złożoność pamięciowa: $\Theta\left(\sigma\left(n + \left\lceil\frac{m}{w_g}\right\rceil\right)\right)$ słów.

Opublikowany w: *Advances in Intelligent and Soft Computing* (2009) 551–559 [66]; wersja rozszerzona w: *Software—Practice and Experience* (2010) 40(8):673–700 [71].

9. LCTS-HS-*

Algorytm wyznaczający długość podciągu LCTS oparty na metodzie Hunta–Szymanskiego (cztery warianty różniące się wewnętrzną organizacją danych).

Złożoność czasowa:

- LCTS-HS-1 (str. 136): $O(D^* \log\log\sigma + (n+\sigma)m)$,
- LCTS-HS-2 (str. 137): $O(D^* \log\log\sigma + nm)$,

- LCTS-HS-3 (str. 138): $O\left(D^* \left\lceil \frac{\log\sigma}{\log w} \right\rceil + (n+\sigma)m\right)$,
- LCTS-HS-4 (str. 138): $O\left(nm + \left\lceil \frac{n}{w} \right\rceil m\sigma\right)$.

Złożoność pamięciowa:

- LCTS-HS-1 (str. 136): $\Theta(n+\sigma)$ słów,
- LCTS-HS-2 (str. 137): $\Theta\left(\left\lceil \frac{n}{w} \right\rceil \sigma\right)$ słów,
- LCTS-HS-3 (str. 138): $\Theta\left(\left\lceil \frac{n}{w} \right\rceil \sigma\right)$ słów,
- LCTS-HS-4 (str. 138): $\Theta\left(\left\lceil \frac{n}{w} \right\rceil \sigma\right)$ słów.

Opublikowany w: *Information Processing Letters* (2006) 100(1):14–20 [63].

10. LCTS-NGMD (str. 131)

Algorytm pudełkowy wyznaczający długość podciągu LCTS.
Złożoność czasowa: $O(n\sigma + nm\log\log\sigma)$.
Złożoność pamięciowa: $O(n\sigma + \sigma^2)$ słów.
Opublikowany jako: Technical Report (2005) [163].

11. LCTS-HYBRID (str. 141)

Algorytm hybrydowy wyznaczający długość podciągu LCTS.
Złożoność czasowa: $O\left(\sigma n \left\lceil \frac{m}{w} \right\rceil + nm \left\lceil \frac{\log\sigma}{\log w} \right\rceil\right)$.
Złożoność pamięciowa: $O\left(\left\lceil \frac{n}{w} \right\rceil \sigma\right)$ słów.
Opublikowany w: *Computing and Informatics* (2009) 28(5):729–744 [73].

12. LICS (str. 71)

Algorytm wyznaczający długość oraz miejsce wystąpienia podciągu LICS (3 warianty różniące się wewnętrzną reprezentacją pokrycia zachłannego ciągu).
Złożoność czasowa:

- reprezentacja za pomocą list: $O(n\ell)$,
- reprezentacja za pomocą drzew zrównoważonych: $O(n\log n + \min(n\ell, \ell^3)\log n)$,
- reprezentacja za pomocą list drzew zrównoważonych: $O\left(n\log\log n + \min(n\ell, n+\ell^3)\log\left\lceil \frac{n}{\ell^2} \right\rceil\right)$.

Złożoność pamięciowa: $\Theta(n)$ słów.
Opublikowany w: *Information Processing Letters* (2009) 109(12):630–634 [65].

13. LISW (str. 82)

Algorytm wyznaczający długość oraz miejsce wystąpienia podciągu LISW (3 warianty różniące się wewnętrzną reprezentacją pokrycia zachłannego ciągu).
Złożoność czasowa:

- reprezentacja za pomocą list: $O(n\ell)$,
- reprezentacja za pomocą drzew zrównoważonych: $O\left(\min\left(n\ell, n\left\lceil\frac{\ell^3}{u}\right\rceil\right)\log u\right)$,
- reprezentacja za pomocą list drzew zrównoważonych: $O\left(n\log\log n+\min\left(n\ell, n\left\lceil\frac{\ell^3}{u}\right\rceil\right)\log\left\lceil\frac{u}{\ell^2}\right\rceil\right)$.

Złożoność pamięciowa: $\Theta(n)$ słów.

Zgłoszony do publikacji [68].

14. MaxHLIS (str. 48)

Algorytm wyznaczający podciąg LIS o maksymalnej wysokości.

Złożoność czasowa: $O(n\log\ell)$ lub $O(n\log\log\sigma)$.

Złożoność pamięciowa: $\Theta(n)$ słów lub $\Theta(n)$ słów $+\Theta(\sigma)$ bitów.

Opublikowany w: *Theoretical and Applied Informatics* (2009) 21(3–4):135–148 [67].

15. MaxSLIS (str. 54)

Algorytm wyznaczający podciąg LIS o maksymalnej sumie.

Złożoność czasowa: $O(n\log\ell)$ lub $O(n\log\log\sigma)$.

Złożoność pamięciowa: $\Theta(n)$ słów lub $\Theta(n)$ słów $+\Theta(\sigma)$ bitów.

Opublikowany w: *Theoretical and Applied Informatics* (2009) 21(3–4):135–148 [67].

16. MaxWLIS (str. 51)

Algorytm wyznaczający podciąg LIS o maksymalnej szerokości.

Złożoność czasowa: $O(n\log\ell)$ lub $O(n\log\log\sigma)$.

Złożoność pamięciowa: $\Theta(n)$ słów lub $\Theta(n)$ słów $+\Theta(\sigma)$ bitów.

Opublikowany w: *Theoretical and Applied Informatics* (2009) 21(3–4):135–148 [67].

17. MerLCS-BP (str. 200)

Algorytm równoległości bitowej wyznaczający długość podciągu MerLCS.

Złożoność czasowa: $\Theta\left(nm\left\lceil\frac{r}{w}\right\rceil\log w\right)$.

Złożoność pamięciowa: $\Theta\left(m\left\lceil\frac{r}{w}\right\rceil\right)$ słów.

Zgłoszony do publikacji [70].

18. MinHLIS (str. 45)

Algorytm wyznaczający podciąg LIS o minimalnej wysokości.

Złożoność czasowa: $O(n\log\ell)$ lub $O(n\log\log\sigma)$.

Złożoność pamięciowa: $\Theta(n)$ słów lub $\Theta(n)$ słów $+\Theta(\sigma)$ bitów.

Opublikowany w: *Theoretical and Applied Informatics* (2009) 21(3–4):135–148 [67].

19. MinSLIS (str. 53)

Algorytm wyznaczający podciąg LIS o minimalnej sumie.

Złożoność czasowa: $O(n \log \ell)$ lub $O(n \log\log \sigma)$.

Złożoność pamięciowa: $\Theta(n)$ słów lub $\Theta(n)$ słów $+\Theta(\sigma)$ bitów.

Opublikowany w: *Theoretical and Applied Informatics* (2009) 21(3–4):135–148 [67].

20. MinWLIS (str. 49)

Algorytm wyznaczający podciąg LIS o minimalnej szerokości.

Złożoność czasowa: $O(n \log \ell)$ lub $O(n \log\log \sigma)$.

Złożoność pamięciowa: $\Theta(n)$ słów lub $\Theta(n)$ słów $+\Theta(\sigma)$ bitów.

Opublikowany w: *Theoretical and Applied Informatics* (2009) 21(3–4):135–148 [67].

21. SLIS (str. 64)

Algorytm wyznaczania podciągu LIS o zadanym pochyleniu.

Złożoność czasowa: $O\left(n \min\left(\sqrt{\frac{\log \ell}{\log\log \ell}}, \log\log n\right)\right)$.

Złożoność pamięciowa: $\Theta(n)$ słów.

Opublikowany w: *Advances in Intelligent and Soft Computing* (2009) 541–549 [74].

Algorytmy znane z literatury

22. Cover-Make (str. 39)

Algorytm tworzenia pokrycia zachłannego ciągu.

Złożoność czasowa: $O(n \log \ell)$ lub $O(n \log\log n)$.

Złożoność pamięciowa: $\Theta(n)$ słów.

23. LCS-BP-Length (str. 99)

Algorytm równoległości bitowej wyznaczający długość podciągu LCS.

Złożoność czasowa: $\Theta\left(n \left\lceil \frac{m}{w} \right\rceil\right)$.

Złożoność pamięciowa: $\Theta\left(\left\lceil \frac{m}{w} \right\rceil \sigma\right)$ słów.

24. LCS-BP-Sequence (str. 102)

Algorytm równoległości bitowej wyznaczający podciąg LCS na podstawie struktur danych utworzonych przez rozszerzoną wersję algorytmu LCS-BP-Length.

Złożoność czasowa: $\Theta(n)$.

Złożoność pamięciowa: $\Theta\left(n \left\lceil \frac{m}{w} \right\rceil\right)$ słów.

25. LCS-HS (str. 94)

Algorytm Hunta–Szymanskiego wyznaczający podciąg LCS.

Złożoność czasowa: $O(n \log n + d \log \ell)$ lub $O(n \log n + d \log\log n)$.

Złożoność pamięciowa: $\Theta(n+d)$ słów.

26. LIS-Read (str. 40)

Algorytm wyznaczający podciąg LIS na podstawie pokrycia zachłannego ciągu.

Złożoność czasowa: $O(n)$.

Złożoność pamięciowa: $\Theta(n)$ słów.

27. Tableau-Insert (str. 37)

Algorytm wstawiania elementu do tableau Younga.

Wstęp

Współcześnie gromadzi się oraz przetwarza duże ilości danych. Samo gromadzenie danych nie ma jednak większego znaczenia, jeśli nie wiadomo, jak wśród zgromadzonych danych wyszukać *istotne* informacje. Trudno także stwierdzić, jakie informacje są *istotne*, gdyż zależy to od wielu czynników. Ważny jest również sposób przechowywania danych. Często dane przechowywane są w postaci ciągów składających się z symboli o znaczeniu zależnym od tego, co dany ciąg reprezentuje. Symbolami mogą być liczby (np. całkowite, rzeczywiste), litery należące do różnych alfabetów języków naturalnych, litery reprezentujące zasady budujące łańcuchy DNA bądź aminokwasy budujące białka, wiersze tekstów źródłowych programów itd.

Celem niniejszej pracy jest omówienie istniejących bądź zaproponowanie nowych algorytmów sekwencyjnych rozwiązujących problemy, w których danymi wejściowymi są ciągi, a żądanym wynikiem jest pewien ich podciąg. Dla niektórych problemów będą dyskutowane także algorytmy równoległe, które są ważne z praktycznego punktu widzenia. Należy bowiem wziąć pod uwagę to, jak szybko w ostatnich latach wzrasta moc obliczeniowa produkowanych procesorów i jak zmienia się ich architektura. Z uwagi na ograniczenia fizyczne, częstotliwość taktowania zegarów procesorów w zasadzie pozostaje niezmieniona, a postęp bierze się z wprowadzania coraz to większej liczby równolegle pracujących rdzeni. Obliczenia równoległe nie są niczym nowym, ale, z wyjątkiem ostatnich lat, stosowane były głównie w superkomputerach. Obecnie obliczenia tego typu wkraczają z konieczności na rynek masowy. Jedną z ciekawych alternatyw, która pojawiła się w kończącej się dekadzie, a która intensywnie się rozwija, jest stosowanie wysoce równoległych procesorów graficznych (ang. *graphical processing unit*, GPU) do przeprowadzania obliczeń ogólnego przeznaczenia. Moc obliczeniowa procesorów graficznych jest często o rząd wielkości większa od mocy obliczeniowej procesorów centralnych (ang. *central processing unit*, CPU). Tendencję do stosowania procesorów graficznych daje się także zauważyć przy projektowaniu superkomputerów – w rankingu Top500 najszybszych superkomputerów z listopada 2010 r. na pierwszym, trzecim oraz czwartym miejscu znajdują się superkomputery, których większość mocy obliczeniowej pochodzi z układów GPU [201].

Specyficzną odmianą obliczeń równoległych jest równoległość bitowa [76, 26]. Polega ona na wykonywaniu obliczeń na pojedynczych bitach słowa komputerowego o długości w, przy czym równolegle wykonywanych jest $O(w)$ operacji. Oczywiście za pomocą pojedynczego bitu nie można reprezentować zbyt wielu informacji, więc obliczenia muszą być stosunkowo proste. Niemniej równoległość bitowa okazuje się zadziwiająco skuteczna w praktyce i dla wielu problemów algorytmy opracowane z jej zastosowaniem są znacznie szybsze od algorytmów sekwencyjnych.

Dysponując reprezentacją danych w postaci ciągu symboli, można formułować różne zadania obliczeniowe. Przykładowo, jeśli ciąg reprezentuje tekst w języku naturalnym, to ważne mogą być algorytmy wyszukiwania fragmentów identycznych z fragmentem podanym jako wzorzec. Jeśli ciąg reprezentuje łańcuch DNA, to w algorytmach wyszukiwania zadanego wzorca powinny być dopuszczalne pewne niezgodności, które są charakterystyczne dla łańcuchów DNA, będących efektem wielokrotnego kopiowania i krzyżowania, nie zawsze w 100% dokładnego.

Często ważna może być odpowiedź na pytanie o podobieństwo dwu ciągów. Przykładów w tym względzie jest wiele, od tak prostych jak wyszukiwanie prawdopodobnych podpowiedzi przez program korygujący pisownię, przez znajdowanie plagiatów (tekstów, utworów muzycznych), ustalanie ojcostwa za pomocą analizy łańcuchów DNA, badanie pokrewieństwa gatunków. Za każdym razem przy wyznaczaniu podobieństwa dwóch ciągów trzeba zdefiniować sposób jego pomiaru. Jedną z najprostszych możliwości jest wyznaczenie tzw. odległości Levenshteina [142], która jest minimalną liczbą koniecznych do wykonania operacji edycyjnych, takich jak wstawienie, usunięcie, zmiana symboli, aby przekształcić jeden ciąg w drugi. O ile miara taka jest dość naturalna przy porównywaniu tekstów wpisywanych przez człowieka, bo odzwierciedla część możliwych do popełnienia pomyłek, o tyle nie zawsze sprawdza się w bardziej skompilowanych sytuacjach. Przykładowo, trudno przypuszczać, że komórka wykonująca kopię łańcucha DNA będzie się „zachowywać" podobnie do osoby wpisującej tekst na klawiaturze. Często błędy mogą być zupełnie innego rodzaju, np. wstawienie długiego fragmentu DNA wyciętego z innego miejsca. Podobnie przy porównywaniu dwóch utworów muzycznych w zapisie nutowym, poleganie na odległości Levenshteina może być zawodne, gdyż dwie wersje tego samego utworu zapisane w innej tonacji zostaną ocenione jako istotnie różne.

Inną interesującą miarą podobieństwa jest długość najdłuższego wspólnego podciągu[1]. Najdłuższy wspólny podciąg (ang. *longest common subsequence*, LCS) dwu ciągów jest ciągiem o maksymalnej możliwej długości spośród podciągów obu ciągów. W tym przypadku im długość podciągu LCS jest większa (przy ustalonej długości porównywanych ciągów), tym te ciągi uznawane są za bardziej podobne. W niniejszej pracy przedstawiono najważniejsze algorytmy sekwencyjne oraz bitowo-równoległe dla rozwiązania problemu znajdowania podciągu LCS (w skrócie: dla problemu LCS). Opisano także dwa autorskie algorytmy równoległe dedykowane dla procesorów graficznych, które zostały opublikowane w [71]. Oba algorytmy opierają się na zaproponowanym ogólnym schemacie obliczania macierzy programowania dynamicznego, w którym w każdym z multiprocesorów przetwarzany jest pewien prostokątny obszar tej macierzy. Pierwszy z algorytmów jest równoległą wersją klasycznego, sekwencyjnego al-

[1] Podciąg otrzymuje się z ciągu przez usunięcie zera lub większej liczby symboli.

gorytmu programowania dynamicznego (podrozdz. 7.6.3). Drugi z nich to równoległa wersja algorytmu równoległości bitowej Hyyrö [120] (podrozdz. 7.6.4)[2].

Poleganie na długości podciągu LCS ma podobne wady co poleganie na odległości Levenshteina przy określaniu podobieństwa ciągów. Z tego powodu w literaturze dyskutowanych jest wiele uogólnień tej miary, dostosowanych do różnych przypadków. W problemie najdłuższego wspólnego podciągu niezmienniczego względem transpozycji (ang. *longest common transposition-invariant subsequence*, LCTS) dopuszcza się, aby do wszystkich symboli jednego z ciągów wejściowych była dodana pewna dowolna, stała wartość. Uzasadnieniem takiej modyfikacji jest porównywanie ciągów reprezentujących dane muzyczne i chęć znalezienia podobieństwa pomiędzy utworami zapisanymi w innej tonacji.

Dla rozwiązania problemu LCTS zaproponowano w niniejszej pracy kilka autorskich algorytmów. Pierwszy z nich, algorytm pudełkowy [163] (podrozdz. 8.2), polega na podziale macierzy programowania dynamicznego na prostokątne fragmenty i wyznaczaniu wartości tych fragmentów macierzy w pewien szczególny sposób. Drugi z algorytmów przedstawionych w pracy oparty jest na metodzie Hunta–Szymanskiego znanej z problemu wyznaczania podciągu LCS. Algorytm ten, opublikowany w [63], został opracowany w czterech wariantach, różniących się metodą reprezentacji wewnętrznej danych, z czego wynikają różne złożoności czasowe i pamięciowe algorytmu (podrozdz. 8.3). Złożoność czasowa dwóch z tych wariantów jest lepsza niż złożoności czasowe innych algorytmów znanych z literatury. Dla typowych danych muzycznych algorytmy te są również zdecydowanie szybsze od innych znanych algorytmów. Kolejną propozycją jest algorytm hybrydowy, opublikowany w [73], łączący jeden z wariantów algorytmów opartych na metodzie Hunta–Szymanskiego z algorytmem równoległości bitowej (podrozdz. 8.4.1). Okazuje się on czasami nawet dwukrotnie szybszy od szybszego z algorytmów składowych. Ostatnim algorytmem zaproponowanym przez autora niniejszej pracy dla rozwiązywania problemu LCTS jest algorytm równoległy przeznaczony dla procesorów graficznych, opublikowany w [71] oraz wcześniej w nieco uproszczonej formie w [66] (podrozdz. 8.5). Algorytm ten jest równoległą wersją algorytmu równoległości bitowej. Wyniki eksperymentalne pokazują, że wersja ta jest kilkakrotnie szybsza od algorytmu, który został zrównoleglony.

W przypadku przetwarzania danych biologicznych (np. reprezentujących łańcuchy DNA, białka) badacze często dysponują wiedzą dziedzinową, np. dotyczącą tego, że szukany najdłuższy wspólny podciąg musi zawierać pewne symbole w ściśle określonym porządku. Wprowadza się wówczas trzeci podciąg, który ukierunkowuje poszukiwania podciągu LCS, a problem nazywa się problemem ukierunkowanego najdłuższego wspólnego podciągu (ang. *constrained longest common subsequence*, CLCS).

[2] W niektórych przypadkach korzystne jest porównywanie więcej niż dwu ciągów. Taka sytuacja ma miejsce np. w bioinformatyce. W podrozdziale 7.4 kwestia ta będzie krótko dyskutowana, jednak generalnie niniejsza praca ogranicza się do przypadku dwu ciągów.

W niniejszej pracy dla problemu CLCS przedstawiono algorytm sekwencyjny oparty na metodzie Hunta–Szymanskiego, opublikowany w [64] (podrozdz. 9.2.1). Algorytm ten charakteryzuje się lepszą złożonością czasową od algorytmów znanych z literatury. Eksperymenty praktyczne potwierdziły dużą szybkość jego działania. Ulepszona wersja tego algorytmu została następnie opublikowana w [75] (podrozdz. 9.2.3). Kolejnym algorytmem zaproponowanym przez autora dla tego problemu jest algorytm równoległości bitowej, opublikowany w [69] (podrozdz. 9.3). Jego złożoność czasowa w przypadku średnim jest istotnie lepsza od złożoności czasowych algorytmów znanych z literatury. Eksperymenty pokazały przewagę tego algorytmu nad algorytmami literaturowymi oraz algorytmem opartym na metodzie Hunta–Szymanskiego. Ostatnim algorytmem zaproponowanym dla tego problemu jest algorytm równoległy dla procesorów graficznych, opublikowany w [71] (podrozdz. 9.4). Jest to równoległa wersja algorytmu sekwencyjnego China i in. [42] opartego na programowaniu dynamicznym. Przeprowadzone badania eksperymentalne dowiodły, że metoda ta jest zwykle dziesiątki razy szybsza od algorytmu, który był podstawą zrównoleglania. Algorytm ten jest także wielokrotnie szybszy od innych, opracowanych przez autora niniejszej pracy, algorytmów sekwencyjnych oraz algorytmów równoległości bitowej.

W zastosowaniach bioinformatycznych został sformułowany również problem najdłuższego scalonego wspólnego podciągu (ang. *merged longest common subsequence*, MerLCS). Danymi wejściowymi są w nim trzy ciągi, a żądanym wynikiem jest najdłuższy taki podciąg pierwszego z ciągów wejściowych, który można rozdzielić na dwa ciągi, z których każdy jest podciągiem jednego z pozostałych ciągów wejściowych. Problem ten został sformułowany przy okazji weryfikacji hipotezy duplikacji całego genomu, po której następuje masowa utrata genów. Autor niniejszej pracy zaproponował dla problemu MerLCS algorytm równoległości bitowej [70] (podrozdz. 10.2). Wyniki eksperymentów pokazują, że zwykle jest on szybszy od algorytmów znanych z literatury ponadpięćdziesięciokrotnie. Również złożoność czasowa tego algorytmu jest istotnie lepsza.

Czasami zdarza się tak, że wiele cennych informacji można uzyskać analizując tylko jeden ciąg i wyszukując w nim najdłuższy podciąg rosnący (ang. *longest increasing subsequence*, LIS). W takim przypadku ciąg wejściowy składa się zazwyczaj z liczb, a dla podciągu wynikowego stawia się wymaganie, aby jego symbole były uporządkowane rosnąco. Problem LIS początkowo był czysto teoretycznym zagadnieniem kombinatorycznym. Znalazł jednak interesujące zastosowania m.in. w porównywaniu całych genomów organizmów żywych (projekt MUMmer [57, 58, 137]), tworzeniu map genowych [84] czy przy odkrywaniu nowych genów [223]. Jednym ze sposobów wyznaczania podciągu LIS, omówionym w pracy, jest budowa pokrycia zachłannego ciągu, na podstawie którego można wyznaczyć ciąg wynikowy.

Problem LIS doczekał się wielu odmian opracowanych z myślą o specyficznych zastosowaniach. W niektórych sytuacjach nakłada się pewne ograniczenia na podciąg wynikowy. Mogą one być dwojakiego rodzaju. W pierwszym przypadku, spośród wielu podciągów identycznej długości, z których każdy może być rozwiązaniem problemu LIS, wybiera się takie, które spełniają dodatkowe warunki. W niniejszej pracy rozpatrywanych jest sześć takich wariantów, w których żądanym wynikiem jest podciąg LIS o: minimalnej/maksymalnej szerokości/wysokości/sumie. Przez szerokość podciągu rozumie się różnicę indeksów jego skrajnych symboli, a przez wysokość – różnicę skrajnych symboli. Dla każdego z tych wariantów w pracy zaproponowano algorytm oparty na wprowadzonej koncepcji pokrycia rozszerzonego (podrozdziały 3.3–3.8). Złożoności czasowe tych algorytmów, opublikowanych w [67], są takie same jak złożoność czasowa algorytmu wyznaczania podciągu LIS opartego na idei pokrycia zachłannego.

Do drugiej kategorii ograniczeń narzucanych na wyznaczany podciągu rosnący należą takie, które powodują, że otrzymany wynik może być krótszy niż podciąg LIS. Jednym z takich problemów jest najdłuższy podciąg rosnący o zadanym pochyleniu (ang. *slope-constrained longest increasing subsequence*, SLIS). Przez pochylenie podciągu rozumie się najmniejszy z ilorazów wartości różnicy dwóch symboli oraz różnicy indeksów tych symboli, a więc stawia się wymaganie, by symbole rosły dostatecznie szybko. Dla problemu SLIS autor zaproponował algorytm, opublikowany w [74] (podrozdz. 4.2), którego idea polega na pewnym przekształceniu ciągu wejściowego oraz zastosowaniu odpowiednio dobranych struktur danych dla problemu wyznaczania poprzednika elementu. Dzięki temu opracowany algorytm ma lepszą złożoność czasową niż algorytm znany z literatury.

Problem najdłuższego cyklicznego podciągu rosnącego (ang. *longest increasing cyclic subsequence*, LICS) jest uogólnieniem problemu LIS, w którym szuka się najdłuższego podciągu LIS wśród wszystkich rotacji ciągu wejściowego. Dla tego problemu zaproponowano algorytm wykorzystujący autorską technikę łączenia pokryć reprezentujących fragmenty ciągu. Pokrycie ciągu może być wewnętrznie reprezentowane na różne sposoby, co skutkuje różnymi wariantami proponowanego algorytmu. Dwa z zaproponowanych wariantów zostały opublikowane w [65] (podrozdziały 5.2.2, 5.2.3), a wariant trzeci będący ich ulepszeniem w [74] (podrozdz. 5.2.4). Złożoności czasowe zaproponowanych algorytmów są lepsze od złożoności czasowych najlepszych algorytmów znanych z literatury dla pewnej kategorii ciągów wejściowych.

W problemie najdłuższego podciągu rosnącego w przesuwającym się oknie (ang. *longest increasing subsequence in sliding window*, LISW) szuka się najdłuższego podciągu LIS w każdym spójnym podciągu ciągu wejściowego określonego rozmiaru[3]. Stosując autorską technikę

[3]Spójny podciąg otrzymuje się z ciągu przez usunięcie zera lub większej liczby początkowych i końcowych symboli.

łączenia pokryć zaproponowaną dla problemu LICS uzyskano trzy warianty algorytmu wyznaczania podciągu LISW [68] (podrozdziały 6.2.2–6.2.4). Złożoność czasowa najlepszego z zaproponowanych wariantów jest lepsza od złożoności czasowych najlepszych algorytmów znanych z literatury dla pewnej kategorii ciągów wejściowych.

W niniejszej pracy dla każdego z opracowanych algorytmów przeprowadzono analizę złożoności czasowej w przypadku pesymistycznym, a dla niektórych także w przypadku średnim. Wyznaczona jest także złożoność pamięciowa algorytmów w przypadku pesymistycznym. Większość zaproponowanych algorytmów była badana eksperymentalnie pod kątem szybkości działania, a wyniki eksperymentów zamieszczone są w tabelach oraz zilustrowane na rysunkach.

Praca składa się z dziesięciu rozdziałów. W rozdziale 1. zdefiniowano najważniejsze terminy i oznaczenia, przeprowadzono dyskusję modeli obliczeniowych stosowanych przy analizie istniejących i proponowanych algorytmów oraz dokonano krótkiego wprowadzenia do obliczeń ogólnego przeznaczenia z zastosowaniem procesorów graficznych. Pozostałe rozdziały tworzą dwie części, dotyczące odpowiednio: wyznaczania podciągów rosnących w jednym ciągu oraz wyznaczania wspólnych podciągów dla dwu ciągów. W rozdziale 2. omawiany jest problem wyznaczania najdłuższego podciągu rosnącego (LIS). Rozdział 3. traktuje o podciągach rosnących o minimalnej/maksymalnej szerokości/wysokości/sumie. W rozdziale 4. rozważany jest wariant problemu LIS, w którym na podciąg wynikowy nałożone są ograniczenia, dotyczące tego, jak szybko jego symbole powinny rosnąć. Rozdział 5. dotyczy wariantu problemu LIS, w którym ciąg wejściowy traktowany jest cyklicznie. W rozdziale 6. dyskutowane jest wyznaczanie podciągu LIS w dowolnym oknie ustalonego rozmiaru ciągu wejściowego. Rozdział 7., rozpoczynający drugą część pracy, dotyczy problemu najdłuższego wspólnego podciągu (LCS). W rozdziale 8. dyskutowany jest wariant problemu LCS, w którym wartości symboli jednego z ciągów mogą być zwiększone o pewną stałą wartość. Rozdział 9. dotyczy problemu najdłuższego ukierunkowanego wspólnego podciągu, w którym nakładane jest pewne ograniczenie na to, jakie symbole musi zawierać podciąg wynikowy. W rozdziale 10. dyskutowany jest problem najdłuższego scalonego wspólnego podciągu. Praca kończy się podsumowaniem, po którym znajduje się spis pozycji literaturowych oraz spisy tabel i rysunków.

Badania, których omówienie zawiera niniejsza praca, były częściowo realizowane w ramach grantów Ministerstwa Nauki i Szkolnictwa Wyższego nr 3177/B/T02/2008/35 oraz nr N N516 441938.

Zabrze, listopad 2010 r.

1. Pojęcia wstępne

1.1. Definicje

Danymi wejściowymi algorytmów omawianych w niniejszej pracy są *ciągi* symboli zwanych także elementami. Każdy *symbol* (*element*) należy do alfabetu Σ, który dla prostoty analizy będzie podzbiorem zbioru liczb całkowitych, a więc $\Sigma \subset \mathbb{Z}$. *Rozmiarem* alfabetu jest liczba symboli, które alfabet zawiera. Przez *długość* lub *rozmiar* ciągu będzie rozumiana liczba symboli, z których ten ciąg się składa.

Podciąg otrzymuje się z ciągu przez usunięcie zera lub większej liczby elementów, tj. dla dowolnego ciągu $X = x_1x_2 \ldots x_n$ ciąg $X' = x'_1x'_2 \ldots x'_{n'}$ będzie podciągiem ciągu X, jeśli istnieje taki ciąg indeksów $1 \leq i_1 < i_2 < \cdots < i_{n'} \leq n$, że $x_{i_1}x_{i_2} \ldots x_{i_{n'}} = X'$. Podciąg będzie nazywany *spójnym*, jeśli dodatkowo $i_{n'} - i_1 = n' - 1$. Dla dowolnego ciągu X przez $X_{i,j}$ będzie oznaczany podciąg spójny $x_ix_{i+1} \ldots x_j$, a przez X_i^j – podciąg spójny ciągu X traktowanego w sposób cykliczny, tzn.:

- jeśli $f(i) \leq f(j)$, to $X_i^j = x_{f(i)} \ldots x_{f(j)}$,
- jeśli $f(i) > f(j)$, to $X_i^j = x_{f(i)} \ldots x_nx_1 \ldots x_{f(j)}$,

gdzie $f(x) = ((x-1) \bmod n) + 1)$. Zatem, X_i^{i-1} oznacza rotację ciągu X rozpoczynającą się na pozycji i, gdzie $1 \leq i \leq n$. Warto zwrócić uwagę, że notacje $X_{i,j}$ oraz X_i^j są równoważne, jeśli $1 \leq i \leq j \leq n$. Dla dowolnego ciągu X notacje X_i, X_{-i} oznaczają odpowiednio $X_{1,i}$ oraz $X_{i,|X|}$, gdzie $|X|$ oznacza długość ciągu X.

W niektórych algorytmach wykorzystywane będą następujące operacje bitowe na słowach komputerowych: & (iloczyn bitowy), | (suma bitowa), $\sim$ (negacja wszystkich bitów), $\oplus$ (różnica symetryczna), $\ll$ (przesunięcie bitowe w lewo), $\gg$ (przesunięcie bitowe w prawo). Rozmiar słowa komputerowego będzie oznaczany przez w. Przejęte zostanie, że w jest całkowitą potęgą 2, co nie jest istotnym ograniczeniem z praktycznego punktu widzenia, ponieważ tak w rzeczywistości jest w produkowanych procesorach. Jeśli to założenie nie jest spełnione, to algorytm można zapisać w taki sposób, aby uwzględniał tylko $2^{\lfloor \log_2 w \rfloor}$ najmłodszych bitów z każdego słowa. W sytuacjach, w których istotne będzie rozróżnienie pomiędzy rozmiarem słowa procesora centralnego (ang. *central processing unit*, CPU) oraz procesora graficznego (ang. *graphical processing unit*, GPU), będą używane oznaczenia odpowiednio w_{c} oraz w_{g}. Instrukcja przypisania $=_w$ oznacza przypisanie w najmłodszych bitów. Zapisy 0^i oraz 1^i oznaczają odpowiednio i bitów o wartości 0 oraz i bitów o wartości 1.

W niniejszej pracy dla zwięzłości zapisu przez złożoność czasową będzie rozumiana zawsze złożoność czasowa w przypadku pesymistycznym, chyba że będzie wyraźnie zaznaczone ina-

czej. Podobnie, złożoność pamięciowa będzie rozpatrywana dla przypadku pesymistycznego. Z uwagi na to, że niektóre z rozważanych algorytmów opierają się na metodzie równoległości bitowej, czyli podstawową jednostką danych jest dla nich bit, zawsze przy specyfikowaniu złożoności pamięciowej będzie jawnie podane, czy jest ona wyrażana w słowach komputerowych czy w bitach.

Część algorytmów omawianych w ramach niniejszej pracy stanowią algorytmy równoległe będące równoległymi wersjami algorytmów sekwencyjnych. W pracy będzie w związku z tym wyznaczane także przyspieszenie uzyskane dzięki zrównoleglaniu, które jest zdefiniowane następująco:

Definicja 1.1. *Przyspieszeniem nazywany jest stosunek czasu wykonania algorytmu sekwencyjnego oraz czasu wykonania algorytmu równoległego.*

1.2. Model obliczeniowy

Istnieje wiele modeli obliczeniowych różniących się pomiędzy sobą m.in. zbiorem operacji podstawowych i założeniami dotyczącymi organizacji pamięci. W niniejszej pracy rozważania prowadzone będą przy założeniu modelu Word RAM, który stosunkowo dobrze oddaje cechy współczesnych komputerów. Model ten jest rozwinięciem modelu RAM (*Random Access Machine*) zaproponowanego w 1973 przez Cooka i Reckhowa [47]. W modelu Word RAM zwykle zakłada się, że pamięć operacyjna o dostępie swobodnym składa się z w-bitowych słów, gdzie $w \geq \log n$, a n jest rozmiarem danych[1]. Założenie to powoduje, że każdy wskaźnik lub indeks do danych można zapisać na pojedynczym słowie komputerowym. Każda z operacji podstawowych, do których należą: dodawanie, odejmowanie, mnożenie[2], operacje bitowe ($|$ – suma, & – iloczyn, $\oplus$ – różnica symetryczna, $\ll$, $\gg$ – przesunięcia, $\sim$ – negacja wartości wszystkich bitów), odczyt i zapis z/do pamięci wykonuje się w czasie $O(1)$ i działa na $O(1)$ komórkach pamięci [177].

Czasami stosowany jest także nieco bardziej rygorystyczny model AC^0 RAM, w którym można wykonywać tylko takie operacje podstawowe, które można fizycznie zrealizować za pomocą układu bramek logicznych o nieograniczonej liczbie wejść z zachowaniem stałej głębokości układu. W praktyce powoduje to, że dopuszczalny jest zbiór operacji z modelu Word RAM z wyjątkiem mnożenia. Przyjęte założenie dotyczące modelu ma oczywiście konsekwencje, jeśli chodzi o złożoności czasowe algorytmów. Jednym z przykładów może być struktura danych

[1] W tekście pracy przyjęto, że log oznacza $\log_2$.

[2] Niektórzy autorzy, np. Papadimitriou [169], nie zaliczają mnożenia do operacji podstawowych w modelu Word RAM. Inni autorzy, np. Cormen i in. [48], do zbioru operacji podstawowych modelu Word RAM zaliczają też dzielenie.

van Emde Boasa [209, 210] stosowana często w algorytmach proponowanych w niniejszej pracy. Otwartym problemem pozostaje, czy możliwa jest jej implementacja w modelu AC^0 RAM przy takich samych złożonościach czasowych jak w modelu Word RAM. Dobry przegląd, także w ujęciu historycznym, modeli obliczeniowych można znaleźć w pracy van Emde Boasa [208] oraz w artykule Allendera [7]. Ciekawą dyskusję, dotyczącą wpływu wyboru modelu obliczeniowego na złożoność czasową operacji wyznaczania poprzednika elementu w uporządkowanym zbiorze, która to operacja (właściwie operacja do niej symetryczna: wyznaczanie następnika) będzie często wykorzystywana w algorytmach niniejszej pracy, można znaleźć w [177].

W pracy będą także rozważane algorytmy równoległe, dla których istnieje kilka modeli obliczeniowych i trudno wskazać jeden dominujący model. Wynika to z faktu, że w praktyce korzysta się z zasadniczo różnych architektur równoległych, które modelowane są w różny sposób. Jednym z popularniejszych modeli jest model PRAM (*Parallel Random Access Machine*) [187], w którym zakłada się, że wiele procesorów współpracuje ze sobą za pomocą wspólnej pamięci. W jego ramach wyróżnia się kilka klas w zależności od sposobu dostępu do pamięci. Model ten jest stosunkowo dobrym przybliżeniem współczesnych systemów komputerowych, w których procesor centralny zawiera wiele rdzeni pracujących ze wspólną pamięcią operacyjną. Innym interesującym modelem jest model BSP (*Bulk-Synchronous Parallelism*), w którym p identycznych par procesor-pamięć komunikuje się za pomocą sieci komputerowej [207]. Model ten stosunkowo dobrze opisuje sytuację, w której komputery tworzące klaster obliczeniowy łączone są w sieć. Również architekturę wielordzeniową można opisywać za pomocą modelu BSP. W tej sytuacji wspólna pamięć operacyjna pełni rolę sieci łączącej poszczególne rdzenie.

Głównym tematem niniejszej pracy są algorytmy sekwencyjne oraz algorytmy oparte na równoległości bitowej. Rozważane są także ich wersje równoległe dla specyficznej architektury procesorów graficznych. Tym samym, podczas analizy algorytmów nie będą wykorzystane wymienione wcześniej modele obliczeń równoległych. O modelach tych wspomniano dlatego, że część istniejących algorytmów równoległych służących do rozwiązywania problemów omawianych w niniejszej pracy zostało zaproponowanych dla takich właśnie modeli, a niektóre z nich były badane teoretycznie przy założeniu pewnych parametrów modeli. Algorytmy te stanowiły inspirację dla tworzenia algorytmów przeznaczonych do wykonywania z zastosowaniem procesorów graficznych.

1.3. Obliczenia ogólnego przeznaczenia na procesorach graficznych

Architektura współczesnych procesorów centralnych może być scharakteryzowana w następujący sposób. Procesor składa się z kilku niezależnych rdzeni (najczęściej od 2 do 6). Każdy z rdzeni jest niezależną jednostką obliczeniową wyposażoną w pamięć podręczną (ang. *cache*

memory) zorganizowaną w kilka poziomów (na niektórych poziomach pamięć ta może być wspólna dla rdzeni). Rozmiar tej szybkiej pamięci podręcznej to aktualnie kilka megabajtów. Dostępna jest także pamięć RAM (ang. *Random Access Memory*) wspólna dla wszystkich rdzeni. Jej rozmiar jest rzędu pojedynczych gigabajtów.

Architektura procesorów graficznych jest zupełnie inna. Opis architektury procesorów GPU zostanie przedstawiony na przykładzie procesorów firmy NVidia, z uwagi na to, że dla tych procesorów wykonano implementacje algorytmów, jednak architektura procesorów graficznych, drugiej wiodącej na tym rynku, firmy AMD/ATI jest podobna. Pojedynczy procesor GPU składa się z dziesiątek *multiprocesorów* (30 w modelu GTX 280, 15 w modelu GTX 480). Każdy z multiprocesorów zawiera od kilku do kilkudziesięciu rdzeni (8 w modelu GTX 280, 32 w modelu GTX 480), które realizują obliczenia zgodnie z modelem SIMT (ang. *single instruction multiple thread*), podobnym do modelu SIMD (ang. *single instruction multiple data*) w taksonomii Flynna [90]. Każdy multiprocesor wyposażony jest w niewielkich rozmiarów *pamięć wspólną* (ang. *shared memory*) (16 KB w GTX 280), z którą jest związana pamięć podręczna. *Pamięć globalna* (ang. *global memory*), wspólna dla wszystkich multiprocesorów jest rozmiaru pojedynczych gigabajtów i nie jest wyposażona w pamięć podręczną (w starszych modelach, do GTX 285 włącznie), przez co dostęp do niej jest stosunkowo wolny. W najnowszych modelach procesorów firmy NVidia, z rodziny Fermi (np. GTX 480) pamięć ta ma pamięć podręczną, której rozmiar wynosi 768 KB. Inne rodzaje pamięci dostępne w procesorach GPU, a niewystępujące w procesorach CPU to:

- *pamięć lokalna* (ang. *local memory*) – niezależna dla każdego multiprocesora, bez pamięci podręcznej, rozmiar dziesiątek kilobajtów,
- *pamięć stała* (ang. *constant memory*) – podobna do pamięci lokalnej, ale o innej charakterystyce dostępu, z pamięcią podręczną,
- *pamięć tekstur* (ang. *texture memory*) – nie jest to osobny obszar pamięci, a raczej inny sposób dostępu do pamięci globalnej, w którym dzięki wykorzystaniu pamięci podręcznej tekstur możliwy jest nieco szybszy dostęp do pamięci globalnej, przy założeniu że żądane dane w pamięci globalnej ułożone są w specyficzny sposób.

Liczba dostępnych rejestrów w pojedynczym multiprocesorze (16 384 w GTX 280, 32 768 w GTX 480) jest znacznie większa niż w procesorach CPU. Dla efektywnego wykorzystania procesora GPU kluczowy jest właściwy dobór rodzaju pamięci dla każdej struktury danych, jak również sposobu dostępu do pamięci przez równolegle pracujące wątki. Niezadbanie o to może spowodować serializację wątków bądź wykorzystanie tylko niewielkiego ułamka maksymalnej przepustowości pamięci globalnej. Generalnie, pamięć globalna powinna być używana najrzadziej jak to możliwe, gdyż pojedyncze odwołanie do niej zajmuje 400–800 cykli procesora. Co więcej, kolejne wątki powinny odwoływać się do kolejnych słów pamięci globalnej, co daje lep-

Tabela 1.1

Charakterystyka procesorów CPU i GPU

	Zalety	Wady
CPU	• łatwa komunikacja pomiędzy wątkami • duża, wielopoziomowa pamięć podręczna • niezależne rdzenie (architektura MIMD) • wiele języków programowania • duże doświadczenie w programowaniu CPU	• niewielka liczba rdzeni (do 6) • umiarkowana przepustowość pamięci • niewielka liczba rejestrów
GPU	• duża liczba rdzeni • duża przepustowość pamięci • duża liczba rejestrów	• rdzenie zależne (architektura SIMT) • pamięć globalna bez pamięci podręcznej • mała pamięć wspólna • ograniczona komunikacja pomiędzy wątkami • konieczność dbania o efektywność użycia pamięci • specjalizowane języki programowania • niewielkie doświadczenie w programowaniu GPU

sze wykorzystanie przepustowości pamięci, dzięki tzw. dostępowi połączonemu (ang. *coalesced access*). Odwołania do pamięci wspólnej w ramach multiprocesora powinny mieć miejsce bez powodowania konfliktów dostępu do banków pamięci podręcznej. Szczegóły dotyczące efektywnego dostępu do pamięci w procesorach GPU można znaleźć w [167].

W procesorach firmy NVidia wątki grupowane są na kilku poziomach. Na poziomie najniższym, 32 wątki grupowane są w *wiązkę* (ang. *warp*). Kilka wiązek tworzy *blok* (ang. *block*). Każdy blok wątków wykonywany jest w pojedynczym multiprocesorze, a wątki należące do jednego bloku mogą być synchronizowane, dzięki czemu mogą one łatwo wymieniać się danymi. Na najwyższym poziomie, wiele bloków zgrupowanych jest w pojedyncze *wywołanie jądra* (ang. *kernel execution*). Liczba bloków powinna być co najmniej równa liczbie multiprocesorów, gdyż w przeciwnym razie niektóre multiprocesory będą niewykorzystane. Aby zredukować opóźnienia w dostępie do pamięci globalnej, rejestrów oraz pamięci wspólnej, zalecane jest, by liczba bloków przekraczała kilkakrotnie liczbę multiprocesorów. Podsumowując, liczba wątków w pojedynczym wywołaniu jądra jest zwykle rzędu dziesiątek tysięcy. Jak widać z powyższej charakterystyki, programowanie dla procesorów CPU oraz dla procesorów GPU różni się w sposób zasadniczy. Dobre wprowadzenie do programowania dla procesorów GPU można znaleźć w serii artykułów [85] oraz w książce [127]. Krótkie podsumowanie cech procesorów CPU i GPU znajduje się w tabeli 1.1.

Nie każdy algorytm sekwencyjny nadaje się do zrównoleglania przy użyciu procesorów CPU, w których liczba rdzeni jest niewielka. Znacznie trudniejsza sytuacja ma miejsce, w przypadku zrównoleglania z zastosowaniem procesorów GPU, ponieważ liczba współpracujących ze sobą wątków powinna być rzędu tysięcy lub dziesiątek tysięcy. W ogólności, najlepszymi kandydatami do równoległego wykonywania w procesorach GPU są algorytmy, które:

- mogą być łatwo zdekomponowane na wiele niezależnych zadań – wątki należące do różnych bloków nie mogą ze sobą współpracować, wobec czego jeśli synchronizacja pomiędzy nimi

jest konieczna, to musi być zrealizowana przez kolejne wywołania jądra, co jest czasochłonne,

- wykonują znacznie więcej obliczeń niż odwołań do pamięci – dostępy do pamięci globalnej są kosztowne i ich liczba powinna być redukowana do niezbędnego minimum,
- używają niewielkiej przestrzeni pamięci dla każdego wątku – najlepiej jeśli algorytm najpierw kopiuje dane z pamięci globalnej do pamięci wspólnej lub rejestrów, następnie wykonuje na tych danych niezbędne obliczenia i wreszcie zapisuje wyniki do pamięci globalnej.

Współcześnie, jedną z głównych wad procesorów GPU jest niewielka ilość szybkiej pamięci wspólnej, w związku z tym, mimo iż np. procesor GTX 280 ma 240 rdzeni, to nie należy oczekiwać 240-krotnego przyspieszenia w stosunku do wersji sekwencyjnej wykonywanej w procesorze CPU. Innymi powodami, dla których takie przyspieszenie jest nieosiągalne, są: około dwukrotnie niższa częstotliwość taktowania zegara procesora, a także krótsze rejestry (32-bitowe), co w niektórych zastosowaniach ma znaczenie.

Wykorzystanie procesorów graficznych do obliczeń ogólnego przeznaczenia (ang. *General-Purpose computation on Graphics Processing Units*, GPGPU) staje się coraz częstsze, głównie dzięki aktywnej promocji architektury CUDA firmy NVidia [167, 166] oraz powstaniu języka OpenCL [126, 202], który został stworzony z myślą o obliczeniach w środowiskach heterogenicznych, m.in. z wykorzystaniem procesorów graficznych. O rosnącym znaczeniu tego kierunku rozwoju świadczyć może także fakt, że na temat architektury CUDA prowadzone są wykłady w wielu uniwersytetach na całym świecie [166]. Znamienne jest też to, że w zestawieniu najszybszych superkomputerów Top500.Org [201] z listopada 2010 r. na pierwszym, trzecim oraz czwartym miejscu pod względem mocy obliczeniowej zmierzonej testem LINPACK znajdują się komputery, w których większość mocy obliczeniowej pochodzi z zastosowanych procesorów graficznych firmy NVidia.

Na podstawie publikacji dostępnych w literaturze można zauważyć, że osiągane przyspieszenia dzięki zastosowaniu procesorów GPU zależą silnie od badanego problemu. Do najbardziej udanych przykładów zastosowania procesorów graficznych do obliczeń o charakterze ogólnym można zaliczyć, m.in. rozwiązywanie problemu N ciał w astrofizyce [189], rozwiązywanie równań dynamicznych [93], symulacje Monte Carlo migracji fotonów [6], komputerowe generowanie hologramów [191]. Bogaty katalog rozwiązań wykorzystujących architekturę CUDA można znaleźć w [166].

Część I

PODCIĄGI ROSNĄCE

2. Najdłuższy podciąg rosnący

2.1. Wprowadzenie

Problem wyznaczania najdłuższego podciągu rosnącego (ang. *longest increasing subsequence*, LIS) w ciągu liczbowym zajmował uwagę matematyków i informatyków od początku XX w. Początkowo rozważano go w kategoriach czysto teoretycznych, ale w ostatnich latach znalazł zastosowanie w wielu dziedzinach, o czym będzie mowa w dalszej części niniejszego rozdziału. Problem ten można zdefiniować następująco:

Problem 2.1 (Najdłuższy podciąg rosnący, LIS). *Dla skończonego ciągu liczbowego* $A = a_1 a_2 \ldots a_n$ *należy znaleźć podciąg* $a_{i_1} a_{i_2} \ldots a_{i_\ell}$ *o maksymalnej długości, taki że* $1 \le i_1 < i_2 < \cdots < i_\ell \le n$ *oraz* $a_{i_1} < a_{i_2} < \cdots < a_{i_\ell}$.

Przykład 2.1 (Najdłuższy podciąg rosnący, LIS). *Dla ciągu liczbowego* $A = 4\ 6\ \underline{2}\ 8\ 1\ \underline{3}\ 12\ 9\ \underline{5}\ \underline{7}\ 11\ \underline{10}$ *najdłuższym podciągiem rosnącym jest m.in.* $A' = 2\ 3\ 5\ 7\ 10$. *W ciągu A podkreślono symbole tworzące podciąg LIS.*

Zgodnie z powyższą definicją, szukany podciąg musi być ściśle rosnący. Istnieje także odmiana tego problemu, w którym wymagane jest tylko, aby otrzymany podciąg był niemalejący. Definicja tego problemu podana jest poniżej.

Problem 2.2 (Najdłuższy podciąg niemalejący). *Dla skończonego ciągu liczbowego* $A = a_1 a_2 \ldots a_n$ *należy znaleźć podciąg* $a_{i_1} a_{i_2} \ldots a_{i_\ell}$ *o maksymalnej długości, taki że* $1 \le i_1 < i_2 < \cdots < i_\ell \le n$ *oraz* $a_{i_1} \le a_{i_2} \le \cdots \le a_{i_\ell}$.

Przykład 2.2 (Najdłuższy podciąg niemalejący). *Dla ciągu liczbowego* $A = 4\ 6\ \underline{2}\ 8\ 1\ \underline{3}\ \underline{3}\ 12\ 9\ 9\ \underline{5}\ \underline{7}\ 11\ \underline{10}$ *najdłuższym podciągiem rosnącym jest m.in.* $A' = 2\ 3\ 3\ 5\ 7\ 10$. *W ciągu A podkreślono symbole tworzące podciąg LIS.*

Jeśli symbole ciągu są unikalne, to oczywiście obie powyższe definicje są równoważne. Ponadto, pomiędzy obiema wersjami tego problemu zachodzi ścisły związek, tj. dysponując algorytmem rozwiązywania ściśle rosnącego wariantu problemu LIS, można rozwiązać drugi wariant przekształcając ciąg wejściowy w następujący sposób:

1. Posortuj stabilnym algorytmem sortowania elementy ciągu wejściowego.
2. Zastąp każdy element ciągu wejściowego jego indeksem w ciągu posortowanym.

Powyższe przekształcenie powoduje, że identyczne elementy ciągu wejściowego zostaną zastąpione przez kolejne liczby całkowite. W bardzo podobny sposób można przekształcić ciąg

wejściowy, aby do wyznaczenia ściśle rosnącego podciągu LIS móc zastosować algorytm dla wariantu nieściśle rosnącego. Złożoność czasowa przekształcenia ciągu wejściowego zależy od użytego algorytmu sortowania i zwykle jest $O(n \log n)$. W dalszej części niniejszej pracy rozważana będzie tylko wersja ściśle rosnąca problemu LIS.

2.2. Dodatkowe definicje i przyjęte założenia

W rozdziałach 2–6 będzie przyjmowane, że ciągiem wejściowym jest $A = a_1 a_2 \ldots a_n$. Elementy (symbole) tego ciągu należą do alfabetu Σ, który jest podzbiorem zbioru liczb całkowitych. W niektórych sytuacjach dogodne jest przyjęcie dodatkowego założenia, że alfabet jest ograniczony, tj. $\Sigma = \{0, \ldots, \sigma - 1\}$. W tej sytuacji σ jest *rozmiarem* alfabetu. Przez *rangę* elementu a_i, oznaczaną przez $h(a_i)$, będzie rozumiana długość najdłuższego rosnącego podciągu kończącego się na a_i. Długość ciągu wynikowego dla każdego problemu rozpatrywanego w niniejszej części będzie oznaczana przez ℓ.

2.3. Długość najdłuższego podciągu rosnącego

Pierwsze oszacowania długości podciągu LIS można znaleźć w pracy Erdösa i Szekeresa [83] z 1935 roku, którzy rozważali pewien problem geometryczny, a jednym z wyników pośrednich było wykazanie, że w dowolnym ciągu liczbowym występuje rosnący bądź malejący podciąg o długości co najmniej $\sqrt{n}$ (patrz również [193]), z czego natychmiast wynika, że oczekiwana długość podciągu LIS w ciągu będącym losową permutacją zbioru n liczb całkowitych to co najmniej $\frac{1}{2}\sqrt{n-1}$. Późniejsze symulacje numeryczne [24] z 1968 roku wskazywały, że wartość oczekiwana tej długości jest bliska $2\sqrt{n}$. W 1972 roku Hammersley [107] wykazał, że istnieje pewna stała c taka, że oczekiwana długość podciągu LIS to $c\sqrt{n} + o(\sqrt{n})$, a Vershik i Kerov [211] wykazali, że $c = 2$. Pełny rozkład prawdopodobieństwa wartości oczekiwanej długości podciągu LIS wyznaczyli Baik i in. [27] w 1999 roku. Wyniki, które uzyskali, mówią m.in., że wartość oczekiwana tej długości to $2n^{1/2} - \Theta(n^{1/6})$, podczas gdy odchylenie standardowe jest $\Theta(n^{1/6})$. Ciekawym artykułem przeglądowym, który podsumowuje osiągnięcia w tej dziedzinie, jest [5].

2.4. Algorytmy

2.4.1. Tableau Younga

Istnieje wiele algorytmów wyznaczania podciągu LIS. Historycznie pierwszy taki algorytm został przedstawiony przez Schensteda [188]. Idea jego działania opiera się na konstrukcji ta-

```
TABLEAU-INSERT(T, x)
Wejście: T – tableau postaci (n_1, n_2, ..., n_N)
         x – liczba wstawiana do T
Wyjście: tableau po wstawieniu do niego x

1  for i ← 1 to N do
2      y ← najmniejsza liczba większa od x znajdująca się w wierszu i
3      if y istnieje then
4          Zamień y w wierszu i z x
5      else
6          Wstaw x na końcu wiersza i
7          return T
8  Rozszerz tableau T przez wstawienie wiersza zawierającego x
9  return T
```

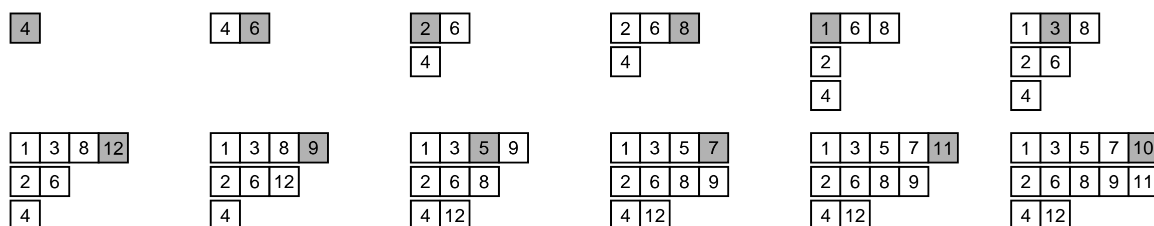
Rys. 2.1. Algorytm wstawiania liczby do tableau Younga
Fig. 2.1. Algorithm inserting an integer to Young tableau

bleau Younga [222], która to struktura zostanie poniżej przybliżona. Tableau Younga zostało zaproponowane jako metoda badania macierzowych reprezentacji permutacji. Jest ono pewnym sposobem rozmieszczenia n liczb całkowitych w tabeli składającej się z wierszy różnej długości.

Definicja 2.1. *Tableau Younga postaci* $(n_1, n_2, \ldots, n_N)$ *jest metodą reprezentacji* $n_1 + n_2 + \cdots + n_N = n$ *liczb całkowitych, przy czym* $n_1 \geq n_2 \geq \cdots \geq n_N$. *Składa się ono z* N *wierszy o długościach* $n_1, n_2, \ldots, n_N$. *Elementy w każdym wierszu uporządkowane są rosnąco od lewej do prawej strony. Każda kolumna zawiera elementy uporządkowane rosnąco od góry do dołu.*

Tableau dla ciągu tworzy się rozpoczynając od pustej struktury, do której wstawia się kolejne elementy za pomocą algorytmu TABLEAU-INSERT (rys. 2.1). Algorytm ten wyszukuje w pierwszym wierszu najmniejszą liczbę większą niż liczba wstawiana i zamienia te liczby ze sobą. Następnie wykonuje to samo w kolejnych wierszach. Jeśli w którymkolwiek kroku liczba większa niż bieżąca nie istnieje, to algorytm wstawia bieżącą liczbę na koniec bieżącego wiersza i kończy swoje działanie. Przykład działania tego algorytmu został przedstawiony na rys. 2.2. Jak wykazał Schensted [188], długość pierwszego wiersza, n_1, jest jednocześnie długością podciągu LIS, a z samej tej struktury można w odpowiedni sposób odczytać podciąg LIS. Czytelnik zainteresowany szczegółami działania tego algorytmu może znaleźć dokładny opis w [5, 188]. Złożoność czasowa algorytmu wyznaczania podciągu LIS za pomocą tableau Younga jest $O(n \log n)$.

Więcej informacji na temat samego tableau Younga i jego właściwości oraz opis innych operacji na nim, nieistotnych dla niniejszej pracy, można znaleźć m.in. w [94, 130, 95, 171].



Rys. 2.2. Ilustracja działania algorytmu tworzenia tableau Younga dla $A = 4\ 6\ 2\ 8\ 1\ 3\ 12\ 9\ 5\ 7\ 11\ 10$. Aktualnie wstawiane elementy są zaznaczone na szaro

Fig. 2.2. Example of the algorithm building Young tableau for $A = 4\ 6\ 2\ 8\ 1\ 3\ 12\ 9\ 5\ 7\ 11\ 10$. The just placed elements are in gray

2.4.2. Pokrycie zachłanne

Gusfield w książce [106] przedstawia algorytm wyznaczania podciągu LIS oparty na idei reprezentacji ciągu w postaci pokrycia zachłannego (ang. *greedy cover*). Koncepcja pokrycia ciągu jest podstawą rozważań prowadzonych w kilku kolejnych rozdziałach.

Definicja 2.2. *Pokryciem ciągu A jest uporządkowany zbiór list, z których każda zawiera malejący podciąg ciągu A. Każdy element ciągu A należy do dokładnie jednej z list.*

Definicja 2.3. *Rozmiarem pokrycia ciągu jest liczba list wchodzących w jego skład.*

Definicja 2.4. *Pokryciem zachłannym ciągu nazywane jest pokrycie, w którym dla każdego symbolu ciągu zachodzi, że indeks listy zawierającej ten symbol jest długością podciągu LIS kończącego się na tym symbolu.*

Pokrycie zachłanne ma kilka interesujących właściwości, których znajomość będzie pomocna w kolejnych rozdziałach:

1. Dla każdego elementu pozycja listy w pokryciu zachłannym, do której ten element należy, jest długością podciągu LIS kończącego się na tym elemencie.
2. Pokrycie zachłanne jest unikalne, tzn. nie istnieje inne pokrycie o własności 1.
3. Pokrycie zachłanne ma minimalny rozmiar, tzn. nie da się zbudować pokrycia o mniejszej liczbie list.
4. Rozmiar pokrycia zachłannego jest długością podciągu LIS dla całego ciągu.

Pokrycie zachłanne ciągu A będzie oznaczane przez $\Gamma(A)$ lub krócej przez Γ, jeśli z kontekstu będzie oczywiste, o pokryciu którego ciągu mowa. Listy należące do pokrycia będą numerowane począwszy od 1, a $\Gamma(A)[k]$ będzie oznaczało k-tą listę pokrycia $\Gamma(A)$. W dalszej części pracy wielokrotnie będą stosowane pokrycia zachłanne ciągów, wobec czego dla zwięzłości opisu za każdym razem, gdy pojawi się termin „pokrycie" ciągu, należy rozumieć przez nie „pokrycie zachłanne".

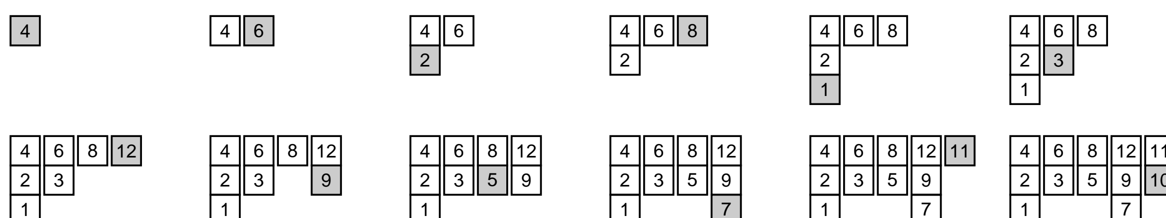
Algorytm Cover-Make tworzący pokrycie ciągu przedstawiony jest na rys. 2.3, a ilustrację jego działania można zobaczyć na rys. 2.4. Algorytm ten przetwarza kolejno symbole ciągu

COVER-MAKE(A)

Wejście: *A – ciąg, dla którego budowane jest pokrycie*
Wyjście: *pokrycie zachłanne dla ciągu A*

1 $\Gamma \leftarrow$ pusta lista przechowująca listy uporządkowane malejąco
2 **for** $i \leftarrow 1$ **to** n **do**
3     $k \leftarrow$ najmniejszy numer listy z Γ, której element końcowy jest większy niż a_i
4   **if** k istnieje **then**
5   Wstaw a_i na koniec listy $\Gamma[k]$
6   **else**
7   Utwórz nową listę zawierającą a_i i dołącz ją na koniec Γ
8 **return** Γ

Rys. 2.3. Algorytm tworzenia pokrycia zachłannego ciągu
Fig. 2.3. Greedy cover making algorithm of the sequence



Rys. 2.4. Przykład działania algorytmu COVER-MAKE wyznaczającego pokrycie zachłanne ciągu $A = 4\ 6\ 2\ 8\ 1\ 3\ 12\ 9\ 5\ 7\ 11\ 10$. Ostatnio wstawione elementy są zaznaczone na szaro. Listy ułożone są pionowo
Fig. 2.4. Example of the algorithm COVER-MAKE finding greedy cover of $A = 4\ 6\ 2\ 8\ 1\ 3\ 12\ 9\ 5\ 7\ 11\ 10$. The just placed elements are gray. The lists are vertical

i dla każdego z nich wyszukuje w dotychczas utworzonym pokryciu pierwszą taką listę, której element końcowy jest większy niż element bieżący. Element bieżący jest następnie dołączany do tej listy, dzięki czemu każda z list pokrycia zawiera elementy w porządku malejącym.

Z tak utworzonego pokrycia można w prosty sposób odczytać podciąg LIS (rys. 2.5). Należy w tym celu rozpocząć od ostatniej listy, wybrać dowolny element i przesuwać się w kolejnych iteracjach po każdej liście od przedostatniej do pierwszej wybierając zawsze największy z elementów mniejszych niż ostatnio wybrany. Można wykazać [106], że w taki sposób zawsze z poprzedniej listy wybierany jest element mniejszy niż bieżący, który już się w niej znajdował (a więc musiał wystąpić w ciągu wejściowym wcześniej niż wybrany element z bieżącej listy).

Dowód, że otrzymane w ten sposób pokrycie jest pokryciem zachłannym, można znaleźć w [106]. Łatwo można zauważyć, że w każdym momencie działania algorytmu COVER-MAKE elementy kończące kolejne listy uporządkowane są rosnąco, co pozwala wyszukiwać odpowiednią listę w wierszu 3. tego algorytmu w szybki sposób. Jeśli nie zakłada się nic o elementach oprócz tego, że można je porównywać ze sobą, to zastosowanie wyszukiwania binarnego bądź zrównoważonych drzew poszukiwań binarnych przechowujących elementy kończące listy pozwala na osiągnięcie, dla całego algorytmu, złożoności czasowej $O(n \log \ell)$ w przypadku pesy-

LIS-READ(Γ)

Wejście: Γ – *pokrycie zachłanne ciągu A*
Wyjście: *najdłuższy podciąg rosnący w ciągu A*

```
1  ℓ ← |Γ|
2  s_ℓ ← dowolny element z Γ[ℓ]
3  for i ← |Γ| − 1 downto 1 do
4      s_i ← największy z elementów z Γ[i] mniejszy niż s_{i+1}
5  return s_1 s_2 ... s_ℓ
```

Rys. 2.5. Algorytm odczytywania podciągu LIS z pokrycia utworzonego algorytmem COVER-MAKE
Fig. 2.5. An algorithm reading LIS from the cover produced by COVER-MAKE algorithm

mistycznym, gdzie ℓ jest długością podciągu LIS dla rozpatrywanego ciągu. W przypadku gdy elementy są liczbami całkowitymi z zakresu $[0, \sigma - 1]$, jak zakładane jest w niniejszej pracy (w szczególnym przypadku ciąg może być permutacją liczb z przedziału $[1, n]$), możliwe jest zastosowanie drzew van Emde Boasa [209, 210] i osiągnięcie tym samym złożoności czasowej $O(n \log\log \sigma)$ [119, 33].

Łatwo można zauważyć, że z samego pokrycia ciągu nie można w jednoznaczny sposób odtworzyć samego ciągu. Prostym przykładem są tu ciągi $A' = 2\ 3\ 1$ oraz $A'' = 2\ 1\ 3$, których pokrycia są identyczne. Nie zmienia to jednak faktu, że pokrycie ciągu niesie z sobą wiele istotnych informacji, które mogą być wykorzystane w konstrukcji różnych algorytmów.

Warto również zaznaczyć, że Fredman [91] wykazał, że złożoność czasowa $O(n \log \ell)$ jest optymalna w modelu obliczeniowym opartym na porównywaniu elementów (zwanym też modelem drzew decyzyjnych). Bespamyatnikh i Segal pokazali [33], jak wyznaczyć wszystkie najdłuższe podciągi rosnące.

2.5. Podsumowanie

Z ciekawszych przykładów zastosowań algorytmów rozwiązywania problemu LIS można wymienić projekt MUMmer [57, 58, 137], w którym problem LIS jest rozwiązywany w celu uliniowienia całych genomów organizmów żywych. W [84] problem LIS był wykorzystany do tworzenia map genowych. Zhang [223] omawia zastosowanie tego problemu przy odkrywaniu nowych genów w projektach Celera Genomics. Lin i in. [144] dyskutują zastosowania problemu LIS w projektowaniu próbek identyfikujących wirusy. Golab i in. [98] omawiają jego zastosowania do wyznaczania zależności w bazach danych. Algorytm wyznaczania podciągu LIS może być także użyty jako element składowy algorytmu wyznaczania najdłuższego wspólnego podciągu (ang. *longest common subsequence*, LCS), który to problem będzie dyskutowany w dalszych rozdziałach niniejszej pracy (podrozdz. 7.3.6). Kolejnym przykładem zastosowań jest wyznaczanie klik w grafach permutacji [100, str. 159].

Istnieje wiele wariantów problemu LIS i niektóre z nich będą szczegółowo rozważane w kolejnych rozdziałach. Zaproponowanie części z tych wariantów miało podłoże praktyczne, tj. okazało się, że bądź to podstawowy problem LIS, bądź pewne jego odmiany mogą zostać w ciekawy sposób wykorzystane do rozwiązania innych problemów.

3. Minimalny i maksymalny najdłuższy podciąg rosnący

3.1. Wprowadzenie

W klasycznym problemie najdłuższego podciągu rosnącego (LIS) wymaga się jedynie, aby długość wyniku była jak największa. Nie nakłada się żadnych ograniczeń na sam wynikowy podciąg, przez co zwykle istnieje wiele poprawnych rozwiązań. W niektórych sytuacjach pożądane może być jednak, aby szukany najdłuższy podciąg rosnący spełniał pewne dodatkowe warunki. Ograniczenia te mogą być dwojakiego rodzaju. Do pierwszej kategorii należą takie, dla których spośród najdłuższych podciągów rosnących należy wybrać podciągi o pewnych szczególnych cechach, a więc długość wyniku jest tu zawsze identyczna z długością podciągu LIS. Do drugiej kategorii należą ograniczenia, które powodują, że szukane rozwiązanie może mieć długość mniejszą niż długość podciągu LIS.

Niniejszy rozdział dotyczy ograniczeń z pierwszej z tych kategorii. Przedstawione zostaną w nim następujące warianty problemu LIS:

- LIS o minimalnej wysokości (MinHLIS) – różnica pomiędzy wartościami skrajnych elementów powinna być minimalna,
- LIS o maksymalnej wysokości (MaxHLIS) – różnica pomiędzy wartościami skrajnych elementów powinna być maksymalna,
- LIS o minimalnej szerokości (MinWLIS) – różnica pomiędzy indeksami skrajnych elementów powinna być minimalna,
- LIS o maksymalnej szerokości (MinHLIS) – różnica pomiędzy indeksami skrajnych elementów powinna być maksymalna,
- LIS o minimalnej sumie (MinSLIS) – suma wartości wszystkich elementów powinna być minimalna,
- LIS o maksymalnej sumie (MaxSLIS) – suma wartości wszystkich elementów powinna być maksymalna.

Problemy poruszane w niniejszym rozdziale nie były do tej pory badane w literaturze. Wyjątkiem jest problem MinHLIS, zdefiniowany przez Tsenga i in. [204], którzy zaprezentowali dla niego algorytm o złożoności czasowej $O(n \log \ell)$ wymagający $\Theta(n)$ słów pamięci. Poniżej przedstawiona zostanie skrótowo idea działania tego algorytmu.

Główną strukturą danych używaną w algorytmie jest las $\mathcal{K}$ zrównoważonych drzew poszukiwań binarnych $K_1, K_2, \ldots, K_\ell$ zawierających podzbiory (parami rozłączne) elementów z ciągu A. Dodatkowo elementy minimalne z każdego drzewa K_h przechowywane są w zrównoważonym drzewie poszukiwań binarnych $K_{\min}$. Z każdym elementem należącym do $K_{\min}$ skojarzony

jest indeks h drzewa K_h, do którego ten element należy. Algorytm przetwarza kolejne symbole ciągu A począwszy od a_1. W kolejnych krokach algorytmu, każde drzewo K_h, dla dowolnego $1 \leq h \leq \ell$, zawiera wszystkie dotychczas przetworzone symbole, których ranga (długość najdłuższego podciągu rosnącego kończącego się na danym symbolu) wynosi h.

Przetwarzanie każdego kolejnego symbolu a_i przebiega następująco:

1. Wyznacz następnik a_i' symbolu a_i w $K_{\min}$.
2. Jeśli a_i' istnieje, to rangi symboli a_i oraz a_i' muszą być takie same, a więc a_i jest wstawiany do drzewa, do którego należy a_i' oraz a_i' jest zastępowany w $K_{\min}$ przez a_i.
3. Jeśli a_i' nie istnieje, to rangą symbolu a_i jest rozmiar drzewa $K_{\min}$ powiększony o 1, a do lasu $\mathcal{K}$ wstawiane jest nowe drzewo $K_{|K_{\min}|+1}$ zawierające a_i oraz a_i jest wstawiany do $K_{\min}$.

Po każdorazowym wstawieniu symbolu a_i do drzewa K_h wyznaczany jest poprzednik a_i w K_{h-1} (o ile $h > 1$) i łącze do niego jest zapisywane razem z a_i do późniejszego wykorzystania. Po przetworzeniu wszystkich elementów z A, dzięki łączom do elementów z sąsiednich drzew uzyskiwany jest szukany podciąg.

3.2. Wspólne idee algorytmów

W tym i kolejnych podrozdziałach zostaną zaproponowane algorytmy rozwiązywania problemów minimalnych i maksymalnych LIS. Algorytmy te zostały opublikowane przez autora w [67]. Opierają się one na idei pokrycia zachłannego ciągu omawianej w rozdz. 2. Sama struktura pokrycia nie jest jednak wystarczająca do uzyskania algorytmów rozwiązujących badane problemy, więc została ona rozszerzona o pewne dodatkowe informacje.

Definicja 3.1. *Pokryciem rozszerzonym Γ^* ciągu A będzie nazywane pokrycie $\Gamma(A)$, w którym każdy element należący do k-tej listy pokrycia jest czwórką: $\langle a_i, i, \pi_i, v_i \rangle$, gdzie π_i jest wskaźnikiem do pewnego mniejszego symbolu należącego do $\Gamma^*[k-1]$ (jeśli $k > 1$), lub pustym wskaźnikiem (jeśli $k = 1$), a v_i jest pewnym dodatkowym polem danych, którego znaczenie zależy od rozpatrywanego wariantu problemu. Elementy w pokryciu rozszerzonym uporządkowane są według pola zawierającego a_i.*

Dla większej zwięzłości opisu w dalszej części niniejszego rozdziału przez „pokrycie” będzie każdorazowo rozumiane „pokrycie rozszerzone”. Dodatkowe pola znajdujące się w pokryciu rozszerzonym wykorzystywane są w sposób zależny od rozpatrywanego w danym momencie wariantu problemu. Wszystkie algorytmy dyskutowane w niniejszym rozdziale opisuje pewien ogólny schemat (rys. 3.1). Poniżej wprowadzonych zostanie kilka przydatnych definicji.

M***LIS(A)

Wejście: *A – ciąg, dla którego wyznaczany będzie podciąg LIS o odpowiednich własnościach*
Wyjście: *podciąg LIS o określonych dodatkowych własnościach*

```
1  Q ← pusta struktura rozwiązująca problem poprzednika
2  Γ* ← puste pokrycie rozszerzone
3  for i ← 1 to n do
4      (x,k) ← Q.successor(a_i)      {x – następnik a_i, k – numer listy z Γ* zawierającej x}
5      if następnik (x,k) nie istnieje then k ← |Γ*| + 1
6      else Q.remove(x)
7      Q.insert(⟨a_i,k⟩)
8      if k = 1 then
9          π ← nil
10         Wyznacz w pewien (zależny od algorytmu) sposób v
11     else
12         Wyznacz w pewien (zależny od algorytmu) sposób π
13         v ← π ↑ .v     {v zawiera wartość pola v czwórki wskazywanej przez π}
14     Γ*[k].append(⟨a_i,i,π,v⟩)
15 return Wynik końcowy wyznaczony na podstawie Γ*
```

Rys. 3.1. Ogólny schemat algorytmów rozwiązujących problemy MinHLIS, MaxHLIS, MinWLIS, MaxWLIS, MinSLIS, MaxSLIS
Fig. 3.1. A general scheme of the algorithms solving MinHLIS, MaxHLIS, MinWLIS, MaxWLIS, MinSLIS, MaxSLIS problems

Definicja 3.2. *Wysokością ciągu rosnącego $a'_1a'_2\dots a'_k$ jest różnica $a'_k - a'_1$. W szczególności wysokością ciągu jednoelementowego jest 0.*

Definicja 3.3. *Szerokością podciągu $a_{i_1}a_{i_2}\dots a_{i_k}$ ciągu $a_1a_2\dots a_n$ ($1 \le i_1 < i_2 < \dots < i_k \le n$) jest różnica indeksów skrajnych elementów, tj. $i_k - i_1$. W szczególności szerokością ciągu jednoelementowego jest 0.*

Definicja 3.4. *Sumą ciągu $a'_1a'_2\dots a'_k$ jest $a'_1 + a'_2 + \dots + a'_k$.*

Definicja 3.5. *Struktura danych rozwiązująca problem poprzednika (por. np. [177]) jest to struktura, na której można wykonywać następujące operacje: wstawiania, usuwania, wyszukiwania następnika oraz wyszukiwania poprzednika.*

3.3. Wariant o minimalnej wysokości

Problem 3.1 (Najdłuższy podciąg rosnący o minimalnej wysokości, MinHLIS)**.** *Dla ciągu A znaleźć najdłuższy podciąg rosnący o minimalnej wysokości.*

Przykład 3.1 (Najdłuższy podciąg rosnący o minimalnej wysokości, MinHLIS)**.** *Dla ciągu liczbowego $A = \underline{4}\ \underline{6}\ 2\ \underline{8}\ 1\ 3\ 12\ \underline{9}\ 5\ 7\ 11\ \underline{10}$ najdłuższym podciągiem rosnącym o minimalnej wyso-*

```
MinHLIS(A)
Wejście: A – ciąg, dla którego wyznaczany będzie podciąg LIS o minimalnej wysokości
Wyjście: najdłuższy podciąg rosnący o minimalnej wysokości w ciągu A

    {Wiersze 1–7 identyczne jak na rys. 3.1}
 8        if k = 1 then
 9            π ← nil
10            v ← a_i
11        else
12            π ← wskaźnik to największego symbolu z Γ*[k − 1] mniejszego niż a_i
13            v ← π ↑ .v      {v zawiera wartość pola v czwórki wskazywanej przez π}
14        Γ*[k].append(⟨a_i, i, π, v⟩)
15    return Ciąg symboli połączonych wskaźnikami π od symbolu z Γ*[|Γ*|] o min. wartości a_i − v_i
```

Rys. 3.2. Algorytm rozwiązujący problem MinHLIS
Fig. 3.2. An algorithm solving MinHLIS problem

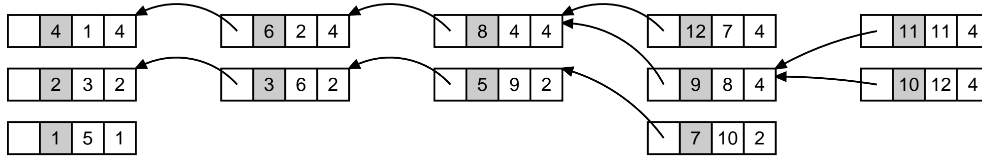
kości jest $A' = 4\ 6\ 8\ 9\ 10$. *Jego wysokość wynosi 6. W ciągu A podkreślono symbole tworzące podciąg MinHLIS.*

W algorytmie wyznaczającym najdłuższy podciąg rosnący o minimalnej wysokości (MinHLIS) pole danych v czwórki przechowywanej w pierwszej liście pokrycia Γ^* wypełniane jest bieżącym symbolem (rys. 3.2, wiersz 10). Wskaźnik π wskazuje największy symbol z $\Gamma^*[k-1]$ mniejszy niż a_i (rys. 3.2, wiersz 12). W celu znalezienia tego elementu należy przeglądnąć listę $\Gamma^*[k-1]$. Dzięki temu, że jest ona uporządkowana malejąco, to przeszukiwanie można zakończyć po znalezieniu pierwszego elementu mniejszego niż a_i. Jako że kolejny element, który zostanie dodany do $\Gamma^*[k]$, będzie na pewno mniejszy niż a_i, to dla niego przeszukiwanie listy $\Gamma^*[k-1]$ będzie można rozpocząć od elementu znalezionego w tym momencie. Można zauważyć, że dzięki temu przy przeszukiwaniu każdej listy $\Gamma^*[k-1]$ algorytm nie będzie się cofał.

Wyznaczenie rozwiązania polega na znalezieniu w ostatniej liście pokrycia Γ^*, tj. $\Gamma[|\Gamma|]$ takiej czwórki $\langle a_i, i, \pi_i, v_i \rangle$, dla której wartość $a_i - v_i$ jest minimalna. Wskaźniki obecne w każdej czwórce w pokryciu Γ^* pozwalają otrzymać oczekiwany podciąg, podczas gdy rozmiar pokrycia informuje o jego długości. Przykład działania algorytmu przedstawiony jest na rys. 3.3. W ostatnim etapie algorytmu (wiersz 15) znajdowana jest czwórka związana z symbolem 10, ponieważ dla niej różnica $a_i - v_i = 10 - 4 = 6$ jest minimalna. Podążając za wskaźnikami odczytywane jest rozwiązanie $A' = 4\ 6\ 8\ 9\ 10$. Poniżej zostanie udowodniona poprawność tego algorytmu.

Lemat 3.1. *Pola danych v_i w każdej liście pokrycia Γ^* zawierają wartości uporządkowane nierosnąco w przypadku użycia algorytmu* MinHLIS.

Dowód. Dowód indukcyjny dla k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$, $v_i = a_i$, a ponieważ lista jest uporządkowana malejąco względem a_i, to musi też być uporządkowana malejąco względem v_i. Dla $k > 1$ niech pewien symbol a_i będzie właśnie dodawany do $\Gamma^*[k]$.



Rys. 3.3. Pokrycie zachłanne rozszerzone wygenerowane przez algorytm MinHLIS dla ciągu $A = 4\ 6\ 2\ 8\ 1\ 3\ 12\ 9\ 5\ 7\ 11\ 10$. Pola czwórek prezentowane są w kolejności: wskaźnik π_i, symbol a_i, indeks i, dane dodatkowe v_i

Fig. 3.3. Annotated greedy cover produced by MinHLIS algorithm for $A = 4\ 6\ 2\ 8\ 1\ 3\ 12\ 9\ 5\ 7\ 11\ 10$. The fields are shown in order: pointer π, symbol a_i, index i, data v_i

Pole danych związane z symbolem a_j kończącym do tej pory listę $\Gamma^*[k]$ jest równe polu danych największego symbolu z $\Gamma^*[k-1]$ mniejszego niż a_j. Ponieważ $a_i < a_j$, więc największy symbol należący do $\Gamma^*[k-1]$ mniejszy niż a_i nie może być większy niż symbol wskazywany przez π_j. Z tego, że pola danych listy $\Gamma^*[k-1]$ są uporządkowane nierosnąco, wynika, że $v_i \leq v_j$. ■

Lemat 3.2. *Dla każdej czwórki $\langle a_i, i, \pi_i, v_i \rangle$, różnica $a_i - v_i$ jest wysokością podciągu MinHLIS kończącego się na a_i.*

Dowód. Dowód indukcyjny dla k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$, $a_i - v_i = 0$, a wysokość każdego ciągu jednoelementowego to 0. Dla $k > 1$, niech $\langle a_i, i, \pi_i, v_i \rangle$ będzie pewną czwórką należącą do $\Gamma^*[k]$. Spośród wszystkich czwórek $\langle a_{j'}, j', \pi_{j'}, v_{j'} \rangle$ należących do $\Gamma^*[k-1]$ takich, że $a_{j'} < a_i$ wartość $a_i - v_{j'}$ jest minimalna, jeśli $a_{j'}$ jest największy możliwy (lemat 3.1). Niech ten element będzie oznaczony przez a_j. Wynika z tego, że a_j jest wskazywany przez wskaźnik π_i, a więc $v_i = v_j$. Co więcej, ponieważ π_i wskazuje a_j, więc $j < i$, czyli aby otrzymać podciąg MinHLIS długości k kończący się na a_i, należy wziąć podciąg MinHLIS o długości $k-1$ kończący się na a_j i dołączyć do niego a_i. ■

Twierdzenie 3.1. *Algorytm* MinHLIS *wyznacza podciąg MinHLIS.*

Dowód. Wszystkie symbole z ciągu A o rangach równych długości podciągu LIS znajdują się w $\Gamma^*[|\Gamma^*|]$. Wobec tego wystarczy znaleźć wśród nich symbol, dla którego wysokość podciągu LIS jest minimalna i wyznaczyć żądany podciąg podążając za wskaźnikami począwszy od tego symbolu. ■

Złożoność czasowa zaprezentowanego algorytmu zależy od złożoności czasowych operacji na strukturze rozwiązującej problem poprzednika (ang. *predecessor problem*) oraz operacji na pokryciu. Na strukturze rozwiązującej problem poprzednika wykonywanych jest n operacji każdego z rodzajów: wstawienia, usunięcia, wyszukiwania następnika. Kwestia wyboru struktury rozwiązującej problem poprzednika i wpływu tego wyboru na złożoność czasową

algorytmu będzie omawiana później (podrozdz. 3.9), a na razie sumaryczna złożoność czasowa wszystkich operacji na strukturze rozwiązującej problem poprzednika będzie oznaczona przez τ_{PP}.

Każda z n operacji wstawiania symbolu do listy w pokryciu wymaga wykonania stałej liczby operacji (przy założeniu że dzięki strukturze rozwiązującej problem poprzednika lista, która ma być rozszerzona, została już zlokalizowana). Istotna dla wyznaczenia złożoności czasowej jest jeszcze operacja wyszukiwania elementów wskazywanych przez wskaźniki będące elementami czwórek (rys. 3.2, wiersz 12). Jak już zostało wcześniej powiedziane, przy przeglądaniu list w poszukiwaniu wskazywanego elementu nigdy na żadnej liście przeszukiwanie nie będzie się odbywało w kierunku początku listy, a ponieważ łączna liczba przeszukiwań jest $\Theta(n)$, a sumaryczna liczba elementów w pokryciu także jest $\Theta(n)$, więc całkowita złożoność czasowa wyszukiwania tych elementów jest $\Theta(n)$. Pozostałe operacje w głównej pętli algorytmu wnoszą do złożoności czasowej składnik $\Theta(n)$. Samo uzyskanie wyniku na podstawie pokrycia wymaga wykonania $O(n)$ operacji dzięki temu, że wystarczy przeglądnąć ostatnią listę, a następnie podążać za wskaźnikami znajdującymi się w elementach pokrycia. Wobec tego, złożoność czasowa zaprezentowanego algorytmu to $\tau_{\mathrm{PP}} + \Theta(n)$.

3.4. Wariant o maksymalnej wysokości

Problem 3.2 (Najdłuższy podciąg rosnący o maksymalnej wysokości, MaxHLIS)**.** *Dla ciągu A znaleźć najdłuższy podciąg rosnący o maksymalnej wysokości.*

Przykład 3.2 (Najdłuższy podciąg rosnący o maksymalnej wysokości, MaxHLIS)**.** *Dla ciągu liczbowego* $A = 4\ 6\ 2\ 8\ \underline{1}\ \underline{3}\ 12\ 9\ \underline{5}\ \underline{7}\ \underline{11}\ 10$ *najdłuższym podciągiem rosnącym o maksymalnej wysokości jest* $A' = 1\ 3\ 5\ 7\ 11$. *Jego wysokość wynosi 10. W ciągu A podkreślono symbole tworzące podciąg MaxLIS.*

W algorytmie rozwiązującym problem najdłuższego podciągu rosnącego o maksymalnej wysokości (MaxHLIS) pole danych v z czwórki przechowywanej w pierwszej liście pokrycia Γ^* wypełniane jest bieżącym symbolem (rys. 3.4, wiersz 10). Wskaźnik π wskazuje symbol kończący listę $\Gamma^*[k-1]$ w chwili przetwarzania symbolu a_i (rys. 3.4, wiersz 12).

W celu wyznaczenia żądanego podciągu należy odnaleźć w ostatniej liście pokrycia Γ^* czwórkę $\langle a_i, i, \pi_i, v_i \rangle$ o maksymalnej wartości $a_i - v_i$. Wskaźniki obecne w każdej czwórce pokrycia Γ^* pozwalają otrzymać podciąg wynikowy, którego wysokość określona jest przez różnicę $a_i - v_i$. Poniżej zostanie udowodniona poprawność tego algorytmu.

Lemat 3.3. *Pola danych v_i w każdej liście pokrycia Γ^* zawierają wartości uporządkowane niemalejąco w przypadku użycia algorytmu* MaxHLIS.

```
MaxHLIS(A)
Wejście: A – ciąg, dla którego wyznaczany będzie podciąg LIS o maksymalnej wysokości
Wyjście: najdłuższy podciąg rosnący o maksymalnej wysokości w ciągu A

   {Wiersze 1–7 identyczne jak na rys. 3.1}
 8       if k = 1 then
 9           π ← nil
10           v ← a_i
11       else
12           π ← wskaźnik do elementu kończącego Γ*[k − 1]
13           v ← π ↑ .v      {v zawiera wartość pola v czwórki wskazywanej przez π}
14       Γ*[k].append(⟨a_i, i, π, v⟩)
15   return Ciąg symboli połączonych wskaźnikami π od symbolu z Γ*[|Γ*|] o maks. wartości a_i − v_i
```

Rys. 3.4. Algorytm rozwiązujący problem MaxHLIS
Fig. 3.4. An algorithm solving MaxHLIS problem

Dowód. Dowód indukcyjny dla k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$, $v_i = a_i$, a ponieważ lista jest uporządkowana malejąco względem a_i, więc musi też być uporządkowana malejąco względem v_i. Dla $k > 1$ niech pewien symbol a_i będzie właśnie dodawany do $\Gamma^*[k]$. Pole wskaźnika związane z symbolem a_j kończącym do tej pory listę $\Gamma^*[k]$ wskazuje na pewien symbol w $\Gamma^*[k-1]$, który nie może być mniejszy niż symbol aktualnie kończący $\Gamma^*[k-1]$ (listy uporządkowane są malejąco). Z tego wynika, że $v_i \leq v_j$. ■

Lemat 3.4. *Dla każdej czwórki $\langle a_i, i, \pi_i, v_i \rangle$, różnica $a_i - v_i$ jest wysokością podciągu MaxHLIS kończącego się na a_i.*

Dowód. Dowód indukcyjny dla k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$, $a_i - v_i = 0$, a wysokość każdego ciągu jednoelementowego to 0. Dla $k > 1$, niech $\langle a_i, i, \pi_i, v_i \rangle$ będzie pewną czwórką należącą do $\Gamma^*[k]$. Spośród wszystkich czwórek $\langle a_{j'}, j', \pi_{j'}, v_{j'} \rangle$ należących do $\Gamma^*[k-1]$ takich, że $j' < i$ wartość $a_i - v_{j'}$ jest maksymalna, jeśli $a_{j'}$ jest najmniejsze możliwe (lemat 3.3). Niech ten element będzie oznaczony przez a_j. Wynika z tego, że a_j jest wskazywany przez wskaźnik π_i, a więc $v_i = v_j$. Co więcej, ponieważ π_i wskazuje a_j, więc $j < i$, czyli aby otrzymać podciąg MaxHLIS długości k kończący się na a_i, należy wziąć podciąg MaxHLIS o długości $k-1$ kończący się na a_j i dołączyć do niego a_i. ■

Twierdzenie 3.2. *Algorytm* MaxHLIS *(rys. 3.4) wyznacza podciąg MaxHLIS.*

Dowód. Wszystkie symbole z ciągu A o rangach równych długości podciągu LIS znajdują się w $\Gamma^*[|\Gamma^*|]$. Wobec tego wystarczy znaleźć wśród nich symbol, dla którego wysokość podciągu LIS jest maksymalna i wyznaczyć żądany podciąg podążając za wskaźnikami począwszy od tego symbolu. ■

```
MinWLIS(A)
Wejście: A – ciąg, dla którego wyznaczany będzie podciąg LIS o minimalnej szerokości
Wyjście: najdłuższy podciąg rosnący o minimalnej szerokości w ciągu A
    {Wiersze 1–7 identyczne jak na rys. 3.1}
 8        if k = 1 then
 9            π ← nil
10            v ← i
11        else
12            π ← wskaźnik do elementu kończącego Γ*[k − 1]
13            v ← π ↑ .v      {v zawiera wartość pola v czwórki wskazywanej przez π}
14        Γ*[k].append(⟨a_i, i, π, v⟩)
15    return Ciąg symboli połączonych wskaźnikami π od symbolu z Γ*[|Γ*|] o min. wartości a_i − v_i
```

Rys. 3.5. Algorytm rozwiązujący problem MinWLIS
Fig. 3.5. An algorithm solving MinWLIS problem

Analiza złożoności czasowej algorytmu MaxHLIS w przypadku pesymistycznym jest bardzo podobna do analizy przeprowadzonej w podrozdz. 3.3 dla algorytmu MinHLIS. Jedyną różnicą pomiędzy tymi algorytmami jest sposób wyznaczania wartości wskaźnika (wiersz 12 pseudokodu). W bieżącym przypadku złożoność czasowa tej operacji jest zawsze $O(1)$. Z powyższego wynika złożoność czasowa algorytmu wyznaczania podciągu MaxHLIS wynosząca $\tau_{PP} + \Theta(n)$.

3.5. Wariant o minimalnej szerokości

Problem 3.3 (Najdłuższy podciąg rosnący o minimalnej szerokości, MinWLIS)**.** *Dla ciągu A znaleźć najdłuższy podciąg rosnący o minimalnej szerokości.*

Przykład 3.3 (Najdłuższy podciąg rosnący o minimalnej szerokości, MinWLIS)**.** *Dla ciągu liczbowego* $A = 4\ 6\ 2\ 8\ \underline{1}\ \underline{3}\ 12\ 9\ \underline{5}\ \underline{7}\ \underline{11}\ 10$ *najdłuższym podciągiem rosnącym o minimalnej szerokości jest* $A' = 1\ 3\ 5\ 7\ 11$*. Jego szerokość wynosi 6. W ciągu A podkreślono symbole tworzące podciąg MinWLIS.*

W algorytmie rozwiązującym problem najdłuższego podciągu rosnącego o minimalnej szerokości (MinWLIS) pole danych v z czwórki przechowywanej w pierwszej liście pokrycia Γ^* wypełniane jest indeksem bieżącego elementu (rys. 3.5, wiersz 10). Wskaźnik π wskazuje symbol kończący listę $\Gamma^*[k-1]$ w chwili przetwarzania symbolu a_i (rys. 3.5, wiersz 12).

W celu wyznaczenia żądanego podciągu należy odnaleźć w ostatniej liście pokrycia Γ^* czwórkę $\langle a_i, i, \pi_i, v_i \rangle$ o minimalnej wartości $i - v_i$ (rys. 3.5, wiersz 15). Wskaźniki obecne w każdej czwórce pokrycia Γ^* pozwalają uzyskać żądany podciąg, którego szerokość określona jest przez różnicę $i - v_i$. Poniżej zostanie udowodniona poprawność tego algorytmu.

Lemat 3.5. *Pola danych v w każdej liście pokrycia Γ^* zawierają wartości uporządkowane niemalejąco w przypadku użycia algorytmu* MinWLIS.

Dowód. Dowód indukcyjny dla k, będącej numerem listy pokrycia Γ^*. Dla $k = 1$, $v_i = i$, a ponieważ elementy dodane do listy później musiały wystąpić na dalszych pozycjach w ciągu, więc pola danych pierwszej listy Γ^* są uporządkowane rosnąco. Dla $k > 1$, niech pewien symbol będzie właśnie wstawiany do $\Gamma^*[k]$. Pole wskaźnika związane z symbolem a_j kończącym do tej pory listę $\Gamma^*[k]$ wskazuje na pewien symbol w $\Gamma^*[k-1]$, który musiał zostać wstawiony do tej listy nie później niż symbol aktualnie kończący $\Gamma^*[k-1]$. Z powyższego wynika, że $v_i \geq v_j$, a więc pola danych w liście $\Gamma^*[k]$ muszą być uporządkowane niemalejąco. ∎

Lemat 3.6. *Dla każdej czwórki $\langle a_i, i, \pi_i, v_i \rangle$ różnica $i - v_i$ jest szerokością podciągu MinWLIS kończącego się na a_i.*

Dowód. Dowód indukcyjny dla k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$, $i - v_i = 0$, a szerokość każdego ciągu jednoelementowego to 0. Dla $k > 1$, niech $\langle a_i, i, \pi_i, v_i \rangle$ będzie pewną czwórką z $\Gamma^*[k]$. Spośród wszystkich czwórek $\langle a_{j'}, j', \pi_{j'}, v_{j'} \rangle$ należących do $\Gamma^*[k-1]$ takich, że $j' < i$, wartość $i - v_{j'}$ jest minimalna, jeśli j' jest największe możliwe (lemat 3.5). Niech ten indeks będzie oznaczony przez j. Wynika z tego, że a_j musiał znajdować się na końcu listy $\Gamma^*[k-1]$, kiedy a_i było wstawiane do pokrycia, a więc π_i wskazuje a_j, czyli $v_i = v_j$. Co więcej, ponieważ π_i wskazuje a_j, więc $j < i$, czyli aby otrzymać podciąg MinWLIS długości k kończący się na a_i, należy wziąć podciąg MinWLIS o długości $k-1$ kończący się na a_j i dołączyć do niego a_i. ∎

Twierdzenie 3.3. *Algorytm* MinWLIS *(rys. 3.5) wyznacza podciąg MinWLIS.*

Dowód. Wszystkie symbole z ciągu A o rangach równych długości podciągu LIS znajdują się w $\Gamma^*[|\Gamma^*|]$. Wobec tego wystarczy znaleźć wśród nich symbol, dla którego szerokość podciągu LIS jest minimalna i wyznaczyć żądany podciąg podążając za wskaźnikami począwszy od tego symbolu. ∎

Analiza złożoności czasowej tego algorytmu jest identyczna z analizą algorytmu MaxHLIS (podrozdz. 3.4). Tak więc złożoność czasowa algorytmu MinWLIS wynosi $\tau_{PP} + \Theta(n)$.

3.6. Wariant o maksymalnej szerokości

Problem 3.4 (Najdłuższy podciąg rosnący o maksymalnej szerokości, MaxWLIS). *Dla ciągu A znaleźć najdłuższy podciąg rosnący o maksymalnej szerokości.*

```
MAXWLIS(A)
Wejście: A – ciąg, dla którego wyznaczany będzie podciąg LIS o maksymalnej szerokości
Wyjście: najdłuższy podciąg rosnący o maksymalnej szerokości w ciągu A

   {Wiersze 1–7 identyczne jak na rys. 3.1}
 8       if k = 1 then
 9           π ← nil
10           v ← i
11       else
12           π ← wskaźnik do największego symbolu z Γ*[k − 1] mniejszego niż a_i
13           v ← π ↑ .v      {v zawiera wartość pola v czwórki wskazywanej przez π}
14       Γ*[k].append(⟨a_i, i, π, v⟩)
15   return Ciąg symboli połączonych wskaźnikami π od symbolu z Γ*[|Γ*|] o maks. wartości a_i − v_i
```

Rys. 3.6. Algorytm rozwiązujący problem MaxWLIS
Fig. 3.6. An algorithm solving MaxWLIS problem

Przykład 3.4 (Najdłuższy podciąg rosnący o maksymalnej szerokości, MaxWLIS)**.** *Dla ciągu liczbowego* $A = \underline{4}\ \underline{6}\ 2\ \underline{8}\ 1\ 3\ 12\ \underline{9}\ 5\ 7\ 11\ \underline{10}$ *najdłuższym podciągiem rosnącym o maksymalnej szerokości jest* $A' = 4\ 6\ 8\ 9\ 10$*. Jego szerokość wynosi 11. W ciągu A podkreślono symbole tworzące podciąg MaxWLIS.*

W algorytmie rozwiązującym problem najdłuższego podciągu rosnącego o maksymalnej szerokości (MaxWLIS) pole danych v czwórki przechowywanej w pierwszej liście pokrycia Γ wypełniane jest indeksem bieżącego symbolu (rys. 3.6, wiersz 10). Wskaźnik π wskazuje największy symbol z $\Gamma^*[k-1]$ mniejszy niż a_i (rys. 3.6, wiersz 12). W celu znalezienia tego elementu należy przeglądnąć listę $\Gamma^*[k-1]$. Dzięki temu, że jest ona uporządkowana malejąco, to przeszukiwanie można zakończyć po znalezieniu pierwszego elementu mniejszego niż a_i. Jako że kolejny element, który zostanie wstawiony do $\Gamma^*[k]$, będzie na pewno mniejszy niż a_i, to dla niego przeszukiwanie listy $\Gamma^*[k-1]$ będzie można rozpocząć od elementu znalezionego w tym momencie. Można zauważyć, że dzięki temu przy przeszukiwaniu każdej listy $\Gamma^*[k]$ algorytm nigdy nie będzie musiał się cofać, a jedyną sytuacją, w której przeszukiwanie na pewno rozpocznie się od początku listy, będzie moment, w którym element bieżący tworzy nową listę.

Wyznaczenie rozwiązania polega na znalezieniu w ostatniej liście pokrycia Γ^*, tj. $\Gamma^*[|\Gamma^*|]$ takiej czwórki $\langle a_i, i, \pi_i, v_i \rangle$, dla której wartość $i - v_i$ jest maksymalna. Wskaźniki obecne w każdej czwórce w pokryciu Γ^* pozwalają otrzymać oczekiwany podciąg, podczas gdy różnica $i - v_i$ informuje o jego szerokości. Poniżej zostanie udowodniona poprawność tego algorytmu.

Lemat 3.7. *Pola danych v w każdej liście pokrycia* Γ^* *zawierają wartości uporządkowane niemalejąco w przypadku użycia algorytmu* MAXWLIS.

Dowód. Dowód indukcyjny dla k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$, $v_i = i$, a ponieważ elementy wstawione do listy później musiały wystąpić na dalszych pozycjach ciągu

wejściowego, więc pola danych pierwszej listy Γ^* są uporządkowane rosnąco. Dla $k > 1$, niech pewien symbol będzie właśnie wstawiany do $\Gamma^*[k]$. Pole danych związane z symbolem a_j kończącym do tej pory listę $\Gamma^*[k]$ jest równe polu danych największego symbolu $\Gamma^*[k-1]$ mniejszego niż a_j. Ponieważ $a_i < a_j$, więc największy symbol należący do $\Gamma^*[k-1]$ mniejszy niż a_i nie może być większy niż symbol wskazywany π_j. Z tego, że pola danych listy $\Gamma^*[k-1]$ są uporządkowane niemalejąco, wynika, iż $v_i \geq v_j$. ■

Lemat 3.8. *Dla każdej czwórki $\langle a_i, i, \pi_i, v_i \rangle$ różnica $i - v_i$ jest szerokością podciągu MaxWLIS kończącego się na a_i.*

Dowód. Dowód indukcyjny dla k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$, $i - v_i = 0$, a szerokość każdego ciągu jednoelementowego to 0. Dla $k > 1$, niech $\langle a_i, i, \pi_i, v_i \rangle$ będzie pewną czwórką należącą do $\Gamma^*[k]$. Spośród wszystkich czwórek $\langle a_{j'}, j', \pi_{j'}, v_{j'} \rangle$ należących do $\Gamma^*[k-1]$ takich, że $a_{j'} < a_i$, wartość $i - v_{j'}$ jest maksymalna, jeśli $a_{j'}$ jest największe możliwe (lemat 3.7). Niech ten element będzie oznaczony przez a_j. Wynika z tego, że a_j jest wskazywane przez π_i, a więc $v_i = v_j$. Co więcej, ponieważ π_i wskazuje a_j, więc $j < i$, czyli aby otrzymać podciąg MaxWLIS długości k kończący się na a_i, należy wziąć podciąg MaxWLIS długości $k-1$ kończący się na a_j i dołączyć do niego a_i. ■

Twierdzenie 3.4. *Algorytm* MaxWLIS *(rys. 3.6) wyznacza podciąg MaxWLIS.*

Dowód. Wszystkie symbole z ciągu A o rangach równych długości podciągu LIS znajdują się w $\Gamma^*[|\Gamma^*|]$. Wobec tego wystarczy znaleźć wśród nich symbol, dla którego szerokość podciągu LIS jest maksymalna i wyznaczyć żądany podciąg podążając za wskaźnikami począwszy od tego symbolu. ■

Analiza złożoności czasowej tego algorytmu jest identyczna z analizą algorytmu MinHLIS (podrozdz. 3.3). Zatem, złożoność czasowa algorytmu MaxWLIS wynosi $\tau_{PP} + \Theta(n)$.

3.7. Wariant o minimalnej sumie

Problem 3.5 (Najdłuższy podciąg rosnący o minimalnej sumie, MinSLIS)**.** *DLa ciągu A znaleźć najdłuższy podciąg rosnący o minimalnej sumie.*

Przykład 3.5 (Najdłuższy podciąg rosnący o minimalnej sumie, MinSLIS)**.** *Dla ciągu liczbowego* $A =$ 4 6 2 8 $\underline{1}$ $\underline{3}$ 12 9 $\underline{5}$ $\underline{7}$ 11 $\underline{10}$ *najdłuższym podciągiem rosnącym o minimalnej sumie jest* $A' =$ 1 3 5 7 10. *Jego suma wynosi 26. W ciągu A podkreślono symbole tworzące podciąg MinSLIS.*

MinSLIS(A)

Wejście: *A – ciąg, dla którego wyznaczany będzie podciąg LIS o minimalnej sumie*
Wyjście: *najdłuższy podciąg rosnący o minimalnej sumie w ciągu A*

{*Wiersze 1–14 identyczne jak na rys. 3.4*}
15 **return** Ciąg symboli połączonych wskaźnikami π począwszy od symbolu kończącego $\Gamma^*[|\Gamma^*|]$

Rys. 3.7. Algorytm rozwiązujący problem MinSLIS
Fig. 3.7. An algorithm solving MinSLIS problem

Pokrycie wyznaczone przez algorytm MaxHLIS (rys. 3.4) może być wykorzystane także do innych celów. Na rys. 3.7 przedstawiono algorytm wyznaczający najdłuższy podciąg rosnący o maksymalnej sumie elementów. Jak można zauważyć, różnica pomiędzy algorytmami MaxHLIS i MinSLIS sprowadza się do innego sposobu znalezienia symbolu, od którego należy rozpocząć odczytywanie wyniku.

Lemat 3.9. *Dla dowolnych czwórek $\langle a_i, i, \pi_i, v_i \rangle$ oraz $\langle a_j, j, \pi_j, v_j \rangle$ należących do tej samej listy pokrycia $\Gamma^*[k]$, takich że $a_i < a_j$ suma elementów podciągu rosnącego (otrzymywanego przez podążanie za wskaźnikami) kończącego się na a_i jest mniejsza niż suma elementów podciągu rosnącego kończącego się na a_j (otrzymywanego w analogiczny sposób).*

Dowód. Niech ζ_x oznacza sumę elementów podciągu rosnącego ciągu A kończącego się na a_x otrzymanego przez podążanie za wskaźnikami począwszy od a_x. Dowód zostanie przeprowadzony przez indukcję względem k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$ jedyne elementy należące do dowolnych podciągów jednoelementowych o elementach rangi 1 to a_i oraz a_j. Ponieważ $\zeta_i = a_i$ a $\zeta_j = a_j$, więc z faktu, że $a_i < a_j$, wynika $\zeta_i < \zeta_j$. Niech teraz $k > 1$. Ponieważ $a_i < a_j$, a oba elementy należą do tej samej listy pokrycia Γ^*, więc $i > j$, tzn. a_i został wstawiony do tej listy później niż a_j. Wskaźnik π_j wskazuje element $a_{j'}$ będący najmniejszym elementem listy $\Gamma^*[k-1]$ takim, że $j' < j$. Podobnie, wskaźnik π_i wskazuje element $a_{i'}$ będący najmniejszym elementem listy $\Gamma^*[k-1]$ takim, że $i' < i$. Z tego wynika, że $a_{i'} \leq a_{j'}$ i $i' \geq j'$. Ponieważ $\zeta_{i'} \leq \zeta_{j'}$ i $a_i < a_j$, więc $\zeta_i < \zeta_j$. ■

Lemat 3.10. *Najdłuższy podciąg rosnący o minimalnej sumie kończący się na a_i może zostać odczytany z pokrycia przez podążanie począwszy od a_i za wskaźnikami znajdującymi się w czwórkach.*

Dowód. Dowód przez indukcję względem k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$ twierdzenie jest oczywiste. Niech teraz $k > 1$. Z lematu 3.9 wiadomo, że sumy podciągów rosnących, które mogą być otrzymane przez podążanie za wskaźnikami z każdego symbolu a_j należącego do $\Gamma^*[k-1]$, są uporządkowane w kolejności malejącej. Z tego wynika, że podciąg

---

MaxSLIS(A)

Wejście: *A – ciąg, dla którego wyznaczany będzie podciąg LIS o maksymalnej sumie*
Wyjście: *najdłuższy podciąg rosnący o maksymalnej sumie w ciągu A*

{*Wiersze 1–14 identyczne jak na rys. 3.2*}
15 **return** Ciąg symboli połączonych wskaźnikami π począwszy od symbolu rozpocz. $\Gamma^*[|\Gamma^*|]$

Rys. 3.8. Algorytm rozwiązujący problem MaxSLIS
Fig. 3.8. An algorithm solving MaxSLIS problem

MinSLIS kończący się na najmniejszym elemencie a_j listy $\Gamma^*[k-1]$ takim, że $j < i$ rozszerzony o symbol a_i jest podciągiem MinSLIS kończącym się na elemencie a_i, a symbol a_j jest wskazywany przez π_i. ■

Twierdzenie 3.5. *Z lematów 3.9 oraz 3.10 wynika, że podciąg MinSLIS może być otrzymany przez podążanie za wskaźnikami począwszy od symbolu kończącego* $\Gamma^*[|\Gamma^*|]$.

Analiza złożoności czasowej tego algorytmu jest identyczna z analizą algorytmu MaxHLIS przedstawioną w podrozdz. 3.4.

3.8. Wariant o maksymalnej sumie

Problem 3.6 (Najdłuższy podciąg rosnący o maksymalnej sumie, MaxSLIS)**.** *Dla ciągu A znaleźć najdłuższy podciąg rosnący o maksymalnej sumie.*

Przykład 3.6 (Najdłuższy podciąg rosnący o maksymalnej sumie, MaxSLIS)**.** *Dla ciągu liczbowego* $A = \underline{4}\ \underline{6}\ 2\ \underline{8}\ 1\ 3\ 12\ \underline{9}\ 5\ 7\ \underline{11}\ 10$ *najdłuższym podciągiem rosnącym o maksymalnej sumie jest* $A' = 4\ 6\ 8\ 9\ 11$. *Jego suma wynosi 39. W ciągu A podkreślono symbole tworzące podciąg MaxSLIS.*

Pokrycie uzyskane po zastosowaniu algorytmu MinHLIS (rys. 3.2) może być użyte także do innych celów. Na rys. 3.8 przedstawiono algorytm wyznaczający najdłuższy podciąg rosnący o maksymalnej sumie elementów. Jak można łatwo zauważyć, różnica pomiędzy algorytmami MinHLIS i MaxSLIS sprowadza się do innego sposobu znalezienia symbolu, od którego należy rozpocząć odczytywanie wyniku.

Lemat 3.11. *Dla dowolnych czwórek* $\langle a_i, i, \pi_i, v_i\rangle$ *oraz* $\langle a_j, j, \pi_j, v_j\rangle$ *należących do tej samej listy pokrycia* $\Gamma^*[k]$, *takich że* $a_i < a_j$ *suma elementów podciągu rosnącego (otrzymywanego przez podążanie za wskaźnikami) kończącego się na* a_i *jest mniejsza niż suma elementów podciągu rosnącego kończącego się na* a_j *(otrzymywanego w analogiczny sposób).*

Dowód. Niech ζ_x oznacza sumę elementów podciągu rosnącego ciągu A kończącego się na a_x otrzymanego przez podążanie za wskaźnikami począwszy od a_x. Dowód zostanie przeprowadzony przez indukcję względem k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$, jedyne elementy należące do dowolnych podciągów jednoelementowych o elementach rangi 1 to a_i oraz a_j. Ponieważ $\zeta_i = a_i$ a $\zeta_j = a_j$, więc z faktu, że $a_i < a_j$, wynika $\zeta_i < \zeta_j$. Niech teraz $k > 1$. Ponieważ $a_i < a_j$, a oba elementy należą do tej samej listy pokrycia Γ^*, więc $i > j$, tzn. a_i został wstawiony do tej listy później niż a_j. Wskaźnik π_j wskazuje największy element $a_{j'}$ mniejszy niż a_j należący do listy $\Gamma^*[k-1]$ taki, że $j' < j$. Podobnie, wskaźnik π_i wskazuje największy element $a_{i'}$ mniejszy niż a_i należący do listy $\Gamma^*[k-1]$ taki, że $i' < i$. Z tego wynika, że $a_{i'} \leq a_{j'}$ i $i' \geq j'$. Ponieważ $\zeta_{i'} \leq \zeta_{j'}$ i $a_i < a_j$, więc $\zeta_i < \zeta_j$. ■

Lemat 3.12. *Najdłuższy podciąg rosnący o maksymalnej sumie kończący się na a_i może zostać z pokrycia odczytany przez podążanie począwszy od a_i za wskaźnikami znajdującymi się w czwórkach.*

Dowód. Dowód przez indukcję względem k, będącego numerem listy pokrycia Γ^*. Dla $k = 1$ twierdzenie jest oczywiste. Niech teraz $k > 1$. Z lematu 3.11 wiadomo, że sumy podciągów rosnących, które mogą być otrzymane przez podążanie za wskaźnikami z każdego symbolu a_j należącego do $\Gamma[k-1]$, są uporządkowane w kolejności malejącej. Z tego wynika, że podciąg MaxSLIS kończący się na największym elemencie a_j listy $\Gamma^*[k-1]$ takim, że $a_j < a_i$ rozszerzony o symbol a_i jest podciągiem MaxSLIS kończącym się na elemencie a_i, a symbol a_j jest wskazywany przez π_i. ■

Twierdzenie 3.6. *Z lematów 3.11 oraz 3.12 bezpośrednio wynika, że podciąg MaxSLIS może być otrzymany przez podążanie za wskaźnikami począwszy od symbolu rozpoczynającego* $\Gamma^*[|\Gamma^*|]$.

Analiza złożoności czasowej tego algorytmu jest identyczna z analizą algorytmu MinHLIS przedstawioną w podrozdz. 3.3.

3.9. Analiza złożoności czasowych i pamięciowych

Złożoność czasowa każdego z algorytmów zaproponowanych w niniejszym rozdziale jest identyczna i wynosi $\tau_{PP} + \Theta(n)$, gdzie τ_{PP} jest sumaryczną złożonością czasową następujących operacji na strukturze rozwiązującej problem poprzednika, wykonywanych dla każdego z $\Theta(n)$ elementów: wstawienia, usunięcia, wyszukiwania następnika. Ponieważ drugi składnik złożoności czasowej jest liniowy, więc wybór struktury rozwiązującej problem poprzednika ma kluczowe znaczenie.

Istnieje wiele struktur danych, które mogą zostać użyte w proponowanych algorytmach. Przykładowo, w zrównoważonym drzewie poszukiwań binarnych (np. drzewie czerwono-czarnym [29, 105, 190], drzewie AVL [1]) złożoność czasowa każdej z trzech wspomnianych operacji jest $O(\log n)$, co może również zostać zapisane jako $O(\log \ell)$, jeśli wyrazić złożoność czasową w zależności od cech ciągu wynikowego. W tym przypadku $\tau_{PP} = O(n \log \ell)$. Jest to złożoność optymalna w modelu obliczeniowym opartym na porównywaniu elementów. W modelu Word RAM, jeśli alfabet dopuszczalnych symboli jest ograniczony z góry przez σ lub ciąg A jest permutacją liczb całkowitych z przedziału $[1, n]$, można osiągnąć lepsze wyniki. Stosując drzewa van Emde Boasa [209, 210], dla których złożoność czasowa każdej z potrzebnych operacji jest $O(\log \log \sigma)$, otrzymuje się $\tau_{PP} = O(n \log \log \sigma)$.

Wniosek 3.1. *Złożoność czasowa algorytmów* MinHLIS, MaxHLIS, MinWLIS, MaxWLIS, MinSLIS, MaxSLIS, *proponowanych w niniejszym rozdziale jest*

$$O(n \log \ell) \qquad \textit{lub} \qquad O(n \log \log \sigma).$$

Jeśli chodzi o złożoność pamięciową, to potrzebnych jest $\Theta(n)$ słów na reprezentację pokrycia oraz pewna ilość miejsca na strukturę rozwiązującą problem poprzednika. W przypadku drzew czerwono-czarnych będzie to $O(n)$ słów. Drzewa van Emde Boasa wymagają natomiast $\Theta(\sigma)$ bitów. Wynika z tego następujący wniosek:

Wniosek 3.2. *Złożoność pamięciowa algorytmów* MinHLIS, MaxHLIS, MinWLIS, MaxWLIS, MinSLIS, MaxSLIS, *jest*

$$\Theta(n) \textit{ słów} \qquad \textit{lub} \qquad \Theta(n) \textit{ słów } + \Theta(\sigma) \textit{ bitów}.$$

Złożoność czasowa jedynego znanego w literaturze algorytmu rozwiązującego problem wyznaczania MinHLIS jest $O(n \log \ell)$, co jest gorszym wynikiem niż uzyskany dla proponowanych algorytmów. Złożoności pamięciowe są za to jednakowe.

3.10. Podsumowanie

W niniejszym rozdziale zaproponowane zostały algorytmy rozwiązujące sześć wariantów problemu najdłuższego podciągu rosnącego. W literaturze znany jest sposób rozwiązywania tylko jednego z tych wariantów, ale zaproponowany tu algorytm cechuje się lepszą złożonością czasową w przypadku pesymistycznym. Złożoności czasowe w przypadku pesymistycznym dla dyskutowanych problemów są takie same jak dolne ograniczenie złożoności czasowej dla pro-

blemu najdłuższego podciągu rosnącego. Ponieważ we wszystkich rozważanych w niniejszym rozdziale problemach jest wyznaczany podciąg LIS, oczywiste jest, że kres dolny złożoności czasowej dla dowolnego z tych problemów nie może być mniejszy od kresu dolnego złożoności czasowej dla problemu LIS.

4. Najdłuższy podciąg rosnący o zadanym pochyleniu

4.1. Wprowadzenie

W niniejszym rozdziale rozważany będzie problem najdłuższego podciągu rosnącego o zadanym pochyleniu (ang. *slope-constrained longest increasing subsequence*, SLIS). W problemie tym nakładany jest dodatkowy warunek na to, jak szybko muszą rosnąć elementy szukanego podciągu. W przeciwieństwie do problemów rozważanych w rozdziale 3. wynik uzyskany w tym problemie może być krótszy niż podciąg LIS dla ciągu wejściowego. W niniejszym rozdziale zakłada się, że alfabet jest podzbiorem zbioru liczb całkowitych z zakresu $[1,\sigma]$.

Definicja 4.1. *Pochyleniem podciągu rosnącego* $A' = a_{i_1}a_{i_2}\dots a_{i_\ell}$ *ciągu* $A = a_1a_2\dots a_n$*, takiego że* $1 \le i_1 < i_2 < \dots < i_\ell \le n$ *oraz* $a_{i_1} < a_{i_2} < \dots < a_{i_\ell}$ *nazywana jest największa taka liczba rzeczywista dodatnia* g*, że dla dowolnego indeksu* $1 \le k < \ell$ *zachodzi:*

$$\frac{a_{i_{k+1}} - a_{i_k}}{i_{k+1} - i_k} \ge g.$$

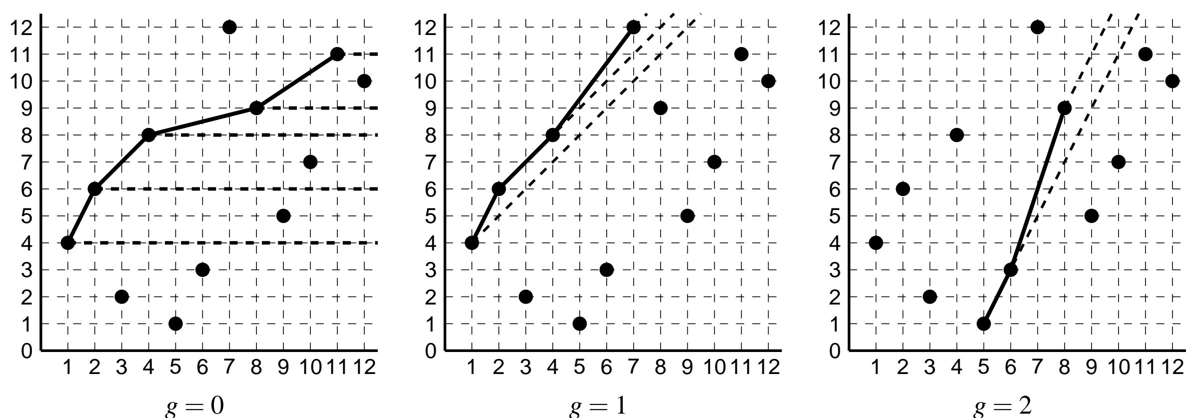
Problem 4.1 (Najdłuższy podciąg rosnący o zadanym pochyleniu, SLIS)**.** *Dla ciągu liczbowego* $A = a_1a_2\dots a_n$ *i zadanego pochylenia* $g \ge 0$ *znaleźć najdłuższy podciąg rosnący, którego pochylenie wynosi co najmniej* g*.*

Przykład 4.1 (Najdłuższy podciąg rosnący o zadanym pochyleniu, SLIS)**.** *Dla ciągu liczbowego* $A =$ $\underline{4}$ $\underline{6}$ 2 $\underline{8}$ 1 3 $\underline{12}$ 9 5 7 11 10 *najdłuższym podciągiem rosnącym o pochyleniu co najmniej* $g = 1$ *jest* $A' =$ 4 6 8 12*. W ciągu* A *podkreślono symbole tworzące podciąg SLIS.*

Rysunek 4.1 pokazuje, jak istotny może być wpływ pochylenia na uzyskiwany wynik. Ponieważ interpretacja warunku pochylenia jest geometryczna, więc przedstawiono na tym rysunku punkty o współrzędnych (i, a_i). W przypadku gdy $g = 0$, wynikiem jest jeden z najdłuższych podciągów rosnących (tu: 4 6 8 9 11). Linie przerywane ilustrują ograniczenia na kolejne punkty, które mogą zostać dołączone do wyniku, w zależności od wartości parametru pochylenia.

Problem SLIS został zdefiniowany przez Yanga i Chena [220]. Autorzy zaproponowali również dla niego algorytm o złożoności czasowej $O(n \log \ell)$ oraz złożoności pamięciowej $O(n)$, gdzie ℓ jest długością uzyskanego wyniku. Idea działania tego algorytmu opiera się na koncepcji tzw. *elementów krytycznych*, która zostanie przybliżona poniżej.

Dla dowolnej pary indeksów $i < j$ notacja $a_i \prec_g a_j$ oznacza, że $\frac{a_j - a_i}{j-i} \ge g$. W takiej sytuacji o a_i będzie się mówić, że *dominuje* a_j. Łatwo można zauważyć, że relacja $\prec_g$ jest przechodnia. Podobnie jak w poprzednich rozdziałach *rangą* elementu a_i będzie nazywana długość podciągu



Rys. 4.1. Przykład wpływu pochylenia na podciąg SLIS dla $A = 4\ 6\ 2\ 8\ 1\ 3\ 12\ 9\ 5\ 7\ 11\ 10$
Fig. 4.1. Example of the influence of the slope on an SLIS for $A = 4\ 6\ 2\ 8\ 1\ 3\ 12\ 9\ 5\ 7\ 11\ 10$

SLIS kończącego się na a_i. W algorytmie Yanga i Chena elementy przetwarzane są kolejno i dla każdego z nich wyznaczana jest ranga. Ciąg A jest dzielony na rozłączne grupy $K_1, K_2, \ldots, K_\ell$ zgodnie z rangą elementów. Autorzy wykazali, że dla stwierdzenia, czy bieżący element a_i jest rangi co najmniej h, wystarczy porównać go z ostatnio wstawionym elementem do K_{h-1} (w oczywisty sposób każdy element jest rangi co najmniej 1).

Definicja 4.2. *Na każdym etapie algorytmu elementami krytycznymi nazywane są elementy ostatnio wstawione do każdej z grup K_i.*

Z przechodniości relacji $\prec_g$ wynika, że jeśli dowolny element $a \in K_h$ dominuje a_i w i-tej iteracji algorytmu, to wszystkie elementy krytyczne w K_j, $\forall j < h$, dominują a_i. Wobec tego, aby wyznaczyć rangę bieżącego elementu, można zastosować wyszukiwanie binarne wśród elementów krytycznych. Po zakończeniu algorytmu liczba grup K_i jest długością podciągu SLIS, a dzięki dodatkowym wskaźnikom związanym ze wszystkimi elementami, które wskazują na pewne elementy z poprzedniej grupy, możliwe jest uzyskanie rozwiązania. Złożoność pamięciowa algorytmu jest $O(n)$ słów, podczas gdy złożoność czasowa determinowana jest przez wyszukiwanie binarne wykonywane dla każdego elementu i jest $O(n \log \ell)$. Wyznaczenie podciągu SLIS na podstawie utworzonych struktur danych nie wpływa w żaden sposób na te złożoności.

4.2. Algorytm

W niniejszym podrozdziale zostanie zaprezentowany algorytm rozwiązywania problemu SLIS zaproponowany przez autora w [74]. Dla tego algorytmu istotne jest nałożenie pewnych ograniczeń na rozmiar alfabetu, i tak $\sigma = \lceil n^c \rceil$ dla pewnej wartości $c \geq 0$. O stałej c zakłada się, że spełnia warunek $w = O(c \log n)$, gdzie w jest długością słowa komputerowego. Warto zauważyć, że założenie to mówi jedynie tyle, że każdą wartość alfabetu można zapisać na co najwyżej stałej liczbie słów komputerowych, a więc w praktyce nie jest ono zbyt rygorystyczne.

Proponowany algorytm opiera się na zbliżonej idei do algorytmu Yanga i Chena, ale w miejsce wyszukiwania binarnego zostanie zastosowana bardziej wyrafinowana struktura danych pozwalająca na efektywną realizację zapytań o następnik elementu. Najpierw konieczne będzie jednak wprowadzenie poniższych terminów.

Definicja 4.3. *Projekcją $\Psi((x,y))$ punktu (x,y) na oś rzędnych układu współrzędnych kartezjańskich przy pochyleniu g nazywana jest współrzędna rzędnych punktu przecięcia prostej o współczynniku kierunkowym g przechodzącej przez punkt (x,y) z osią rzędnych układu współrzędnych kartezjańskich, a więc $\Psi((x,y)) = y - gx$.*

Definicja 4.4. *Projekcją $\Psi(a_i)$ elementu a_i przy pochyleniu g nazywana jest projekcja punktu (i,a_i) przy pochyleniu g.*

Rysunek 4.2a przedstawia ilustrację geometryczną wyznaczania projekcji elementów. Poniższy lemat zawiera obserwację kluczową dla wykazania poprawności proponowanego algorytmu.

Lemat 4.1. *Po każdej iteracji algorytmu uporządkowane rosnąco projekcje wszystkich elementów krytycznych odpowiadają rosnącym rangom elementów, będących elementami krytycznymi. Elementy, których projekcje są identyczne, uporządkowane są według indeksów tych elementów w ciągu wejściowym.*

Dowód. Dowód będzie przeprowadzony przez zaprzeczenie. Niech w dowolnej iteracji projekcją elementu krytycznego a_i o randze $h > 1$ będzie ψ_i, a lemat będzie nieprawdziwy. Wynika z tego, że istnieje element krytyczny a_j o randze $h' < h$, którego projekcja ψ_j jest większa niż projekcja ψ_i. Możliwe są teraz dwa przypadki: $j > i$ lub $j < i$.

W pierwszym przypadku

$$\frac{a_j - a_i}{j - i} \geq g,$$

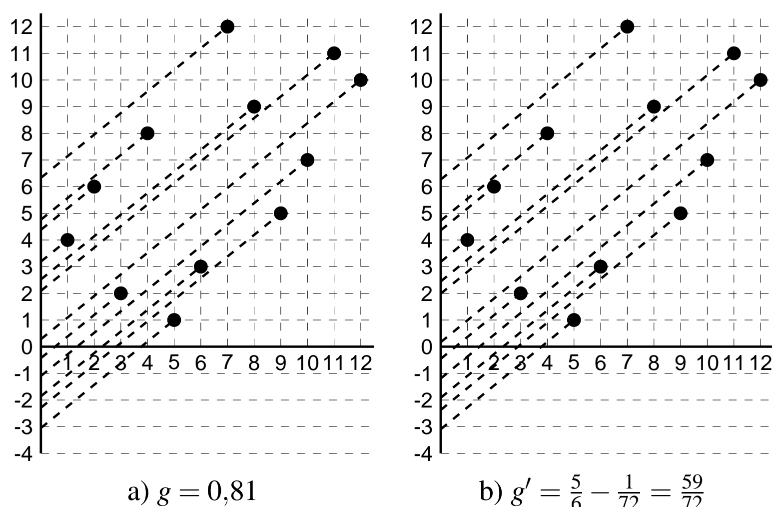
jednakże wówczas a_i dominuje a_j, a więc ranga elementu a_j musi być większa niż h, co jest sprzeczne z założeniem.

W drugim przypadku element a_j nie może należeć do tego samego podciągu SLIS co a_i, ponieważ

$$\frac{a_i - a_j}{i - j} < g$$

i podciąg ten nie miałby pochylenia g. W dowolnym podciągu SLIS kończącym się na a_i, na pozycji h' znajduje się pewien element $a_{j'} \neq a_j$, o randze h'. Z definicji punktu krytycznego wiadomo, że $j' < j$. Jednak w takim przypadku

$$\frac{a_i - a_{j'}}{i - j'} \geq g,$$



a) $g = 0{,}81$ b) $g' = \frac{5}{6} - \frac{1}{72} = \frac{59}{72}$

Rys. 4.2. Przykład wyznaczania projekcji elementów w problemie SLIS dla A = 4 6 2 8 1 3 12 9 5 7 11 10. Projekcje kolejnych elementów dla $g' = \frac{5}{6} - \frac{1}{72} = \frac{59}{72}$ wynoszą $\frac{227}{72}, \frac{310}{72}, \frac{-39}{72}, \frac{332}{72}, \frac{-233}{72}, \frac{-150}{72}, \frac{437}{72}, \frac{160}{72}, \frac{-189}{72}, \frac{-106}{72}, \frac{121}{72}, \frac{-12}{72}$

Fig. 4.2. Example of the computation of projections of symbols for the SLIS problem for A = 4 6 2 8 1 3 12 9 5 7 11 10. Projections for the successive elements for $g' = \frac{5}{6} - \frac{1}{72} = \frac{59}{72}$ are $\frac{227}{72}, \frac{310}{72}, \frac{-39}{72}, \frac{332}{72}, \frac{-233}{72}, \frac{-150}{72}, \frac{437}{72}, \frac{160}{72}, \frac{-189}{72}, \frac{-106}{72}, \frac{121}{72}, \frac{-12}{72}$

podczas gdy

$$\frac{a_i - a_j}{i - j} < g.$$

Z tego wynika, że

$$\frac{a_j - a_{j'}}{j - j'} \geq g,$$

a więc $a_{j'}$ dominuje a_j oraz ranga elementu $a_{j'}$ jest mniejsza niż ranga elementu a_j, co jest sprzeczne z założeniem. ■

Z lematu 4.1 wynika, że zamiast dla każdego kolejnego elementu a_i wyszukiwać element krytyczny o maksymalnej randze, który dominuje a_i, można wyszukiwać element krytyczny a_j o minimalnej projekcji, która jest większa niż projekcja a_i. Jeśli taki element istnieje, to ranga elementu a_i będzie taka sama jak ranga elementu a_j, wobec czego w zbiorze elementów krytycznych należy wymienić a_j na a_i, ponieważ jego projekcja jest mniejsza, a ranga ta sama. W przeciwnym przypadku ranga elementu a_i jest o jeden większa niż aktualna liczba elementów krytycznych i a_i należy wstawić do zbioru elementów krytycznych. Istnieje wiele struktur danych, które mogą zostać zastosowane do przechowywania elementów krytycznych uporządkowanych według ich projekcji, jednak kwestia wyboru jednej z nich zostanie omówiona nieco później.

Na podstawie założenia dotyczącego alfabetu możliwe jest nałożenie pewnych ograniczeń na istotne wartości g. Po pierwsze, g jest ograniczone od góry przez $\lceil n^c \rceil - 1$, ponieważ dla

każdego większego g problem staje się trywialny i odpowiedzią na pytanie o podciąg SLIS jest dowolny jednoelementowy podciąg ciągu A. Po drugie, bez utraty ogólności można zamienić g na $g' \geq g$, jeśli tylko projekcje wszystkich punktów należących do zbioru $\{(x,y) : 1 \leq x \leq n, 1 \leq y \leq \lceil n^c \rceil\}$ są uporządkowane w takiej samej kolejności przy pochyleniu g' jak przy pochyleniu g. Dzięki tym dwóm obserwacjom liczba istotnych do rozważenia pochyleń jest $O(n^{c+1})$ (w sumie, dla wszystkich możliwych wartości parametru g). Wszystkie one są zdefiniowane przez współczynniki kierunkowe prostych przechodzących przez punkt $(1,1)$ oraz każdy z punktów zbioru $\{(x,y) : 2 \leq x \leq n, 1 \leq y \leq \lceil n^c \rceil\}$. Wartości pochylenia są liczbami wymiernymi o postaci p/q, gdzie $0 \leq p < \lceil n^c \rceil - 1$ i $0 < q < n$. W celu znalezienia pochylenia g' przy znanym pochyleniu g należy zminimalizować wartość $p/q - g$, przy założeniu że $p/q \geq g$.

Lemat 4.2. *Uporządkowania elementów ciągu według projekcji dla pochylenia g oraz dla pochylenia $g' = p/q$, gdzie $0 \leq p < \lceil n^c \rceil$, $0 < q < n$ oraz p/q jest najmniejszym takim ułamkiem, że $p/q \geq g$ są identyczne.*

Dowód. Niech dane będą dwa punkty (x_1,y_1) oraz (x_2,y_2) takie, że $x_1 < x_2$. Jeśli teraz przy pochyleniu g projekcja punktu (x_1,y_1) jest większa niż projekcja punktu (x_2,y_2), to ponieważ $g' \geq g$ przy pochyleniu g' projekcje te będą uporządkowane tak samo.

Druga część dowodu dla przypadku, w którym dla pochylenia g projekcja punktu (x_1,y_1) jest niemniejsza niż projekcja punktu (x_2,y_2), zostanie przeprowadzona przez zaprzeczenie. Niech dla pochylenia $g' = p/q \geq g$ takiego, że $g' - g$ jest najmniejsze spośród wszystkich możliwych dla $0 \leq p < \lceil n^c \rceil$ oraz $0 < q < n$, projekcja punktu (x_1,y_1) będzie mniejsza niż projekcja punktu (x_2,y_2). W takim przypadku musi istnieć pochylenie $g \leq \frac{y_2-y_1}{x_2-x_1} < g'$ takie, że dla niego projekcje punktów (x_1,y_1) i (x_2,y_2) są równe, co stoi w sprzeczności z założeniem, że g' jest najmniejszym możliwym ułamkiem p/q nie mniejszym niż g. ∎

Lemat 4.3. *Uporządkowania elementów ciągu według projekcji dla pochylenia g oraz dla pochylenia $g'' = p/q - 1/(qn)$, gdzie $0 \leq p < \lceil n^c \rceil$, $0 < q < n$ oraz p/q jest najmniejszym takim ułamkiem, że $p/q \geq g$ są identyczne.*

Dowód. W oczywisty sposób projekcje elementów przy pochyleniu $g' = p/q$ są o postaci x/q, gdzie x jest pewną liczbą całkowitą. Projekcje przy pochyleniu g'' różnią się od projekcji przy pochyleniu g' o addytywny składnik $i/(qn)$, gdzie i jest indeksem elementu w ciągu. Ponieważ $1 \leq i \leq n$, więc składnik ten zawsze mieści się w zakresie $[1/(qn), 1/q]$. Jeśli więc dla dowolnych dwóch elementów a_i, a_j projekcje przy pochyleniu g' były różne, to ich względne uporządkowanie musi być takie samo przy pochyleniach g' oraz g''.

Jeśli jednak projekcje elementów a_i oraz a_j, dla $i < j$ są identyczne przy pochyleniu g', to zgodnie z lematem 4.1 uporządkowanie elementów jest takie, że a_j jest traktowany jako większy niż a_i. Przy zastosowaniu pochylenia g'' projekcja elementu a_j będzie większa niż projekcja elementu a_i, dzięki dodatkowym składnikom $i/(qn)$ oraz $j/(qn)$. ■

Pochylenia, które należy rozważyć dla danego g, aby wyznaczyć $g' = p/q$, odpowiadają współczynnikom kierunkowym prostych przechodzących przez punkt $(1,1)$ oraz $(q+1, p+1) = (i, 1+\lceil (i-1)g \rceil)$ dla wszystkich $i \in \{2,\dots,n\}$. Ponieważ punktów tych jest $\Theta(n)$, a wyznaczenie wartości $p/q - g$ wymaga wykonania stałej liczby operacji, więc złożoność czasowa operacji wyznaczania g' jest $\Theta(n)$.

Skoro $g'' = p/q - 1/(qn) = (pn-1)/(qn)$, to projekcją $\Psi(a_i)$ przy pochyleniu g'' jest $a_i - i(pn-1)/(qn)$. (Ilustracja projekcji elementów dla pochylenia g'' pokazana jest na rys. 4.2b). Skrajne dopuszczalne wartości projekcji definiowane są przez sytuacje, w których $a_n = 1$ oraz $a_1 = \lceil n^c \rceil$. Wartości projekcji dla nich to odpowiednio

$$1 - \frac{n(pn-1)}{qn} \qquad \text{oraz} \qquad \lceil n^c \rceil - \frac{pn-1}{qn}.$$

Wszystkie projekcje można przeskalować na liczby całkowite przez pomnożenie ich przez qn, a otrzymane w ten sposób wartości będą nazywane *projekcjami całkowitymi* elementów. Każda z projekcji całkowitych musi być liczbą całkowitą z przedziału $[qn - n(pn-1), \lceil n^c \rceil qn - pn + 1]$, a projekcją całkowitą elementu a_i jest $a_i qn - i(pn-1)$. Warto zauważyć, że dla ustalonego g liczba różnych możliwych projekcji całkowitych jest $O(n^{c+2})$.

W tak przekształconym problemie dziedzina projekcji całkowitych jest rozmiaru $O(n^{O(1)})$, a możliwych wartości występujących równocześnie w strukturze danych przechowującej elementy krytyczne jest $O(\ell)$. Co więcej, wszystkie projekcje, które równocześnie mogą wystąpić, są parami różne. Zadaniem, od którego zależy złożoność czasowa całego algorytmu, jest wykonanie na strukturze danych zawierającej elementy krytyczne $O(n)$: zapytań o następnik projekcji całkowitej elementu, usunięć projekcji całkowitej, wstawień projekcji całkowitej.

Pseudokod proponowanego algorytmu przedstawiony jest na rys. 4.3. Po wyznaczeniu współczynnika pochylenia $g'' = p/q - 1/(qn)$ algorytm oblicza projekcje całkowite wszystkich elementów ciągu wejściowego. Następnie stosuje strukturę danych Q przechowującą elementy krytyczne do wyszukiwania następnika. Każda zmienna l_i przechowuje wskaźnik do elementu o randze o jeden mniejszej niż ranga elementu a_i, który dominuje a_i. Dzięki tym wskaźnikom możliwe jest uzyskanie szukanego podciągu.

Kluczową kwestią dla złożoności czasowej proponowanego algorytmu jest dobór odpowiedniej struktury danych Q. Zastosowanie wykładniczych drzew poszukiwań (ang. *exponential search trees*) wprowadzonych przez Anderssona i Thorupa [11] gwarantuje, że złożoność czasowa

SLIS(A, g)

Wejście: *A – ciąg, dla którego wyznaczany będzie podciąg LIS o zadanym pochyleniu*
g – pochylenie
Wyjście: *najdłuższy podciąg rosnący o o zadanym pochyleniu w ciągu A*

```
   {Wyznaczanie projekcji całkowitych}
 1 Wyznacz współczynnik pochylenia g'' = p/q − 1/(qn)
 2 for i ← 1 to n do ψ_i ← a_i qn − i(pn − 1)
   {Obliczenia właściwe}
 3 Q ← pusta struktura rozwiązująca problem poprzednika
 4 ℓ ← 0;   e ← 0
 5 for i ← 1 to n do
 6       s ← Q.successor(ψ_i)
 7       if s istnieje then Q.remove(s)
 8       else ℓ ← ℓ + 1;   e ← i
 9       Q.insert(ψ_i)
10       l_i ← Q.predecessor(ψ_i)
   {Wyznaczanie wyniku}
11 for i ← ℓ downto 1 do
12       s_i ← a_e
13       e ← l_e
14 return s_1 s_2 ... s_ℓ
```

Rys. 4.3. Algorytm rozwiązujący problem SLIS
Fig. 4.3. Algorithm solving SLIS problem

każdej z podstawowych operacji wykonywanych na Q jest $O(\sqrt{\log \ell / \log\log \ell})$, a cała struktura zajmuje $\Theta(\ell)$ słów pamięci. Inną możliwością jest użycie drzew van Emde Boasa [209, 210], dla których potrzebnych jest $O(n^{c+2})$ bitów pamięci, a złożoność czasowa każdej ze stosowanych operacji jest $O(\log\log n)$. Niestety, z uwagi na złożoność pamięciową struktura ta nie może być użyta bezpośrednio. Ponieważ w algorytmie istotna jest tylko względna kolejność projekcji elementów, więc można zamiast operować na tych wartościach wprost, posortować projekcje całkowite wyznaczone dla elementów ciągu A i używać ich indeksów w tablicy posortowanych projekcji całkowitych. Dzięki temu, że różnych kluczy do sortowania jest $\Theta(n)$, a ich uniwersum jest $O(n^{c+2}) = O(n^{O(1)})$, to algorytm sortowania pozycyjnego (ang. *radix sort*) pozwala posortować je w czasie $O(nc)$. Dla dużej wartości parametru c można również użyć algorytmu z [128] o złożoności czasowej $O(n \log c)$.

Ponieważ zaproponowano dwa konkurencyjne do siebie algorytmy, więc istotne jest wskazanie sposobu, który gwarantuje użycie algorytmu szybszego. W tym celu należy rozpocząć wykonywanie algorytmu z wykorzystaniem struktury Anderssona–Thorupa i przetwarzać kolejne elementy tak długo, jak sekwencja wynikowa jest dostatecznie krótka, tj. $\sqrt{\log \ell / \log\log \ell} < \log\log n$. W momencie, w którym warunek ten nie będzie spełniony (jeśli takie coś nastąpi), należy algorytm przerwać i rozpocząć go od nowa w wersji z drzewami van Emde Boasa. Wobec tego można sformułować następujący wniosek:

Wniosek 4.1. *Złożoność czasowa zaproponowanego algorytmu dla problemu SLIS jest*

$$O\left(n\min\left(\sqrt{\frac{\log\ell}{\log\log\ell}},\log\log n\right)\right).$$

4.3. Podsumowanie

Zaproponowany w niniejszym rozdziale algorytm rozwiązywania problemu SLIS wymaga $\Theta(n)$ słów pamięci, a jego złożoność czasowa jest $O(n\min(\sqrt{\log\ell/\log\log\ell},\log\log n))$, a więc mniej niż najlepszego algorytmu znanego z literatury, którego złożoność czasowa jest $O(n\log\ell)$. W celu osiągnięcia tego wyniku konieczne było poczynienie pewnych założeń na dane wejściowe. Założenia te nie stanowią jednak w praktyce większego problemu, bo można je streścić stwierdzeniem, że każdy symbol alfabetu powinien być liczbą całkowitą dającą się reprezentować na stałej liczbie słów komputerowych.

5. Najdłuższy cykliczny podciąg rosnący

5.1. Wprowadzenie

W niniejszym rozdziale rozważany będzie problem najdłuższego cyklicznego podciągu rosnącego (ang. *longest increasing cyclic subsequence*, LICS), tzn. żądanym wynikiem jest najdłuższy podciąg rosnący występujący w dowolnej z rotacji ciągu wejściowego. Przykładowo, dla ciągu $A = 2\ 5\ 4\ 3\ 6\ 1$ można znaleźć podciąg rosnący $2\ 4\ 6$, jednak wykonując na tym ciągu rotację o 1 symbol w prawo, uzyskuje się ciąg: $A^* = 1\ 2\ 5\ 4\ 3\ 6$, w którym najdłuższym podciągiem rosnącym jest m.in. $1\ 2\ 4\ 6$.

Łatwo można wykazać, że długość podciągu LICS nie może przekroczyć podwojonej długości podciągu LIS dla dowolnej rotacji ciągu oryginalnego. Wystarczy w tym celu zauważyć, że elementy podciągu LICS po dowolnej rotacji rozbijane są na dwa ciągi, z których choć jeden musi mieć długość równą co najmniej połowie długości podciągu LICS. Z drugiej strony, łatwo można wykazać, że oszacowanie to jest ostre, ponieważ dla ciągu parzystej długości o postaci: $(n/2+1)(n/2+2)\dots(n)(1)(2)\dots(n/2)$ długością podciągu LIS jest $n/2$, podczas gdy długością podciągu LICS jest n. Wartość oczekiwana długości podciągu LICS w ciągu będącym losową permutacją jest asymptotycznie podobna jak dla problemu LIS, tj. wynosi $2n^{1/2} + o(n^{1/2})$ [3]. Problem LICS można formalnie zdefiniować następująco:

Problem 5.1 (Najdłuższy cykliczny podciąg rosnący, LICS)**.** *Dla ciągu $A = a_1a_2\dots a_n$ znaleźć najdłuższy spośród podciągów rosnących dla wszystkich ciągów A_i^{i-1}, gdzie $1 \le i \le n$.*

Przykład 5.1 (Najdłuższy cykliczny podciąg rosnący, LICS)**.** *Dla ciągu liczbowego $A = 4\ \underline{6}\ 2\ \underline{8}\ \underline{1}\ \underline{3}\ 12\ 9\ \underline{5}\ 7\ 11\ 10$ najdłuższym cyklicznym podciągiem rosnącym jest $A' = 1\ 3\ 5\ 6\ 8$. W ciągu A podkreślono symbole tworzące podciąg LICS.*

Na potrzeby niniejszego rozdziału przyjęte zostanie założenie, że ciąg A jest permutacją liczb całkowitych z zakresu $[1,n]$. Jeśli to założenie nie jest spełnione, to ciąg wejściowy należy przekształcić zamieniając każdy element na jego indeks w ciągu posortowanym, przy czym elementy identyczne sortowane są w taki sposób, że element znajdujący się na dalszej pozycji w ciągu sortowanym traktowany jest jako mniejszy. Przykładowo, ciąg $A = 5\ 15\ 4\ 1\ 23\ 51\ 4\ 5\ 13\ 18$ jest przekształcany w ciąg $5\ 7\ 3\ 1\ 9\ 10\ 2\ 4\ 6\ 8$. Takie przekształcenie wymaga czasu $O(n \log n)$ i w dalszej części niniejszego rozdziału składnik ten nie będzie brany pod uwagę, gdyż inne składniki mają większy wkład do złożoności czasowej.

Najprostszym sposobem wyznaczenia podciągu LICS jest użycie szybkiego algorytmu wyznaczania podciągu LIS dla każdej rotacji: A_1^n, A_2^1, …, A_n^{n-1}. Złożoność czasowa w tym przy-

padku jest $O(n^2 \log\log n)$. Albert i in. [3] zaproponowali kilka szybszych algorytmów. Złożoność czasowa pierwszego z nich jest $O(n\ell \log n)$, gdzie ℓ to długość podciągu LICS. Jak można zauważyć, algorytm ten w szczególnym przypadku, gdy $\ell = \Theta(n)$, jest nawet asymptotycznie wolniejszy niż podejście naiwne. Ponieważ oczekiwaną wartością ℓ jest $2\sqrt{n} + o(\sqrt{n})$, więc stanowi on jednak istotny postęp. Drugi z algorytmów zaproponowanych w [3] ma złożoność czasową $O(n^{3/2} \log n)$, ale otrzymany wynik jest poprawny tylko z pewnym dużym prawdopodobieństwem.

Inną możliwością jest wykorzystanie algorytmów rozwiązujących problem najdłuższego podciągu rosnącego w przesuwającym się oknie (ang. *longest increasing subsequence in sliding window*, LISW) (patrz rozdz. 6). Zastosowanie takich algorytmów dla ciągu będącego konkatenacją dwóch kopii ciągu A przy założonej długości okna n jest w istocie rozwiązywaniem problemu LICS. Użycie algorytmu zaproponowanego przez Alberta i in. [4] pozwala osiągnąć złożoność czasową $O(n \log\log n + n\ell)$. Chen i in. [40] ulepszyli jeszcze ten wynik do $O(n\ell)$.

W ostatnim czasie Tiskin [199] zaproponował nowatorskie podejście do rozwiązywania problemu najdłuższego wspólnego podciągu (ang. *longest common subsequence*, LCS), o którym będzie jeszcze mowa w rozdziale 7. Podejście to opiera się na acyklicznych grafach skierowanych i tzw. macierzach najwyższych wyników (ang. *highest-score matrices*), które reprezentują macierze przetwarzane w metodzie programowania dynamicznego. W [199] dzięki zastosowaniu tych technik do problemu LICS Tiskin uzyskał złożoność czasową $O(n^{3/2})$, a w kolejnej pracy [200] wynik ten został jeszcze poprawiony do $O(n \log^2 n)$.

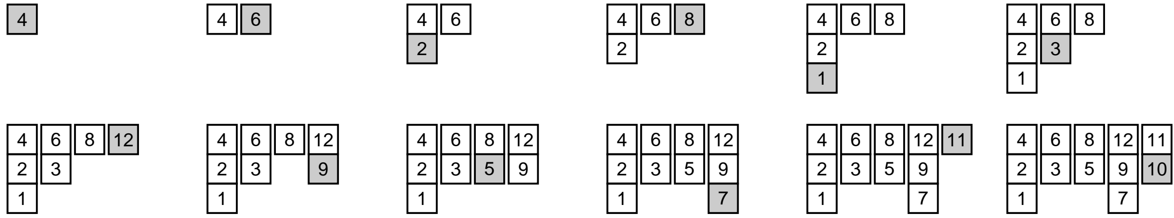
5.2. Algorytmy

5.2.1. Podstawowe koncepcje

W tym i kolejnych podrozdziałach zostaną zaprezentowane algorytmy rozwiązywania problemu LICS zaproponowane przez autora w [65, 74]. W algorytmach tych ciąg A reprezentowany jest za pomocą pokrycia zachłannego $\Gamma(A)$ (podrozdz. 2.1). Kluczowe dla tych algorytmów jest wykazanie, jak dysponując dla ciągu $A = A'A''$ pokryciami $\Gamma(A')$ oraz $\Gamma(A'')$ wyznaczyć pokrycie $\Gamma(A)$.

Definicja 5.1. *Odczytem według pokrycia dla ciągu A, oznaczanym przez $E(A)$, jest konkatenacja kolejnych malejących list tworzących pokrycie* $\Gamma(A)$.

Przykład pokrycia dla ciągu $A =$ 4 6 2 8 1 3 12 9 5 7 11 10 pokazany jest na rys. 5.1. (Dodatkowo, dla przypomnienia pokazany jest tu cały proces tworzenia pokrycia). Dla tego przykładu odczyt według pokrycia to $E(A) =$ 4 2 1 6 3 8 5 12 9 7 11 10.



Rys. 5.1. Przykład działania algorytmu Cover-Make wyznaczającego zachłanne pokrycie ciągu $A =$ 4 6 2 8 1 3 12 9 5 7 11 10. Elementy wstawiane w kolejnych krokach są zaznaczone na szaro. Listy ułożone są pionowo

Fig. 5.1. Example of the algorithm Cover-Make finding greedy cover of $A =$ 4 6 2 8 1 3 12 9 5 7 11 10. The just placed elements are gray. The lists are vertical

W celu wykazania poprawności ogólnego schematu algorytmów proponowanych w niniejszym rozdziale dowiedzione zostaną poniższe lematy i twierdzenia.

Lemat 5.1. *Niech ciąg A będzie konkatenacją ciągów A' oraz A'', tj. $A = A'A''$. Pokrycie ciągu $A'A''$ jest identyczne do pokryć ciągów $E(A')E(A'')$ oraz $A'E(A'')$.*

Dowód. Algorytm tworzenia pokrycia Cover-Make (rys. 2.3) przetwarza kolejne symbole ciągu A, a więc tworzy najpierw pokrycie $\Gamma(A')$, które w oczywisty sposób jest identyczne z pokryciem $\Gamma(E(A'))$. Dalszy ciąg dowodu zostanie przeprowadzony przez indukcję względem długości ciągu A'' oznaczanej przez n''. Dla $n'' = 1$, $A'' = E(A'')$, a więc twierdzenie jest prawdziwe.

Niech teraz $n'' > 1$, a drugi ciąg niech będzie oznaczony przez $A''x$, gdzie x jest pewnym symbolem. Jeśli x jest ostatnim symbolem $E(A''x)$, to lemat jest prawdziwy. Niech zatem x nie będzie ostatnim symbolem $E(A''x)$. Wiadomo wówczas, że

$$E(A'') = \Gamma(A'')[1]\ldots\Gamma(A'')[p]\Gamma(A'')[p+1]\ldots\Gamma(A'')[k],$$

oraz

$$E(A''x) = \Gamma(A'')[1]\ldots\Gamma(A'')[p]x\Gamma(A'')[p+1]\ldots\Gamma(A'')[k].$$

Symbol s kończący listę $\Gamma(A'')[p]$ jest większy niż x, a wszystkie symbole z list $\Gamma(A'')[p+1],\ldots,\Gamma(A'')[k]$ są większe niż s. Dlatego też, kiedy algorytm Cover-Make przetwarza ciąg $E(A')E(A''x)$ i napotyka symbol x, to umieszcza go na końcu pewnej listy znajdującej się nie na prawo od listy zawierającej s. Następnie algorytm Cover-Make przetwarza pozostałe symbole z list $\Gamma(A'')[p+1],\ldots,\Gamma(A'')[k]$, ale ponieważ wszystkie one są większe niż s, to na pewno będą w jednej z list znajdujących się na prawo od listy zawierającej s. Z powyższego wynika, że ich pozycje w pokryciu w żaden sposób nie mogą się zmienić z powodu wcześniejszego wstawienia do niego symbolu x. Wobec tego pokrycia $\Gamma(E(A')E(A'')x)$ oraz $\Gamma(E(A')E(A''x))$ są identyczne, a więc przetwarzanie ciągu A'' zgodnie z odczytem według pokrycia zamiast przetwarzania bezpośrednio A'' nie ma wpływu na uzyskane pokrycie wynikowe. ■

Ponieważ w problemie LICS rozważane są wszystkie rotacje, więc bez utraty ogólności można założyć, że a_1 jest największym symbolem w ciągu A i takie założenie będzie przyjmowane do końca niniejszego rozdziału. (Złożoność czasowa wstępnej rotacji koniecznej do spełnienia tego warunku jest $\Theta(n)$). Długość podciągu LIS dla tego wstępnie zmodyfikowanego ciągu będzie oznaczana przez ℓ', podczas gdy długość LICS – przez ℓ.

Definicja 5.2. *Punktami stopu dla ciągu A będą nazywane symbole, które znajdują się na początkach list pokrycia* $\Gamma(A)$.

Dla przykładu, z rys. 5.1 punktami stopu będą $4, 6, 8, 12, 11$.

Lemat 5.2. *Jeśli ciąg* A_1^n *zostanie cyklicznie przesunięty w lewo o* $1 \leq i < n$ *symboli, to element znajdujący się na początku ostatniej listy pokrycia* $\Gamma(A_{n+1-i}^{n-i})$ *jest punktem stopu.*

Dowód. Z lematu 5.1 wiadomo, że

$$\Gamma(A_{n+1-i}^{n-i}) = \Gamma(A_{n+1-i}^{n} E(A_1^{n-i})).$$

Algorytm Cover-Make tworzy najpierw pokrycie Γ dla A_{n+1-i}^n. Następnie rozszerza je elementami z kolejnych list $\Gamma(A_1^{n-i})$. Element rozpoczynający każdą z tych list jest punktem stopu i może być dołączony do jakiejś listy w Γ bądź rozpocząć nową listę w Γ. Ponieważ symbole w listach dołączanego pokrycia są uporządkowane malejąco, więc tylko punkt stopu może być dołączony do tworzonego pokrycia na początku nowej listy. Z tego, że a_1 jest elementem maksymalnym w A wynika, że przynajmniej ten element musi znaleźć się na początku nowej listy dołączanej do Γ, a więc ostatnia lista pokrycia $\Gamma(A_{n+1-i}^{n-i})$ musi zaczynać się od punktu stopu. ■

Lemat 5.3. *Długość podciągu LICS dla ciągu A jest równa największej z długości podciągu LIS dla wszystkich rotacji A kończących się punktem stopu.*

Dowód. Z lematu 5.2 wynika, że na początku ostatniej listy pokrycia dla dowolnej rotacji A_i^{i-1} zawsze znajduje się jakiś punkt stopu a_k. Oznacza to, że długość podciągu LIS kończącego się na elemencie a_k w tej rotacji jest równa długości podciągu LIS dla rotacji A_i^{i-1}. Dla dowolnego punktu stopu a_k najdłuższy podciąg rosnący kończący się na a_k wśród wszystkich możliwych rotacji można znaleźć w A_{k+1}^k. Z powyższego wynika, że długość podciągu LICS dla ciągu A jest równa długości najdłuższego podciągu LIS wśród wszystkich rotacji kończących się punktami stopu. ■

Niech teraz $A = A'A''$ oraz $\Gamma' = \Gamma(A')$ i $\Gamma'' = \Gamma(A'')$ będą znane, a celem będzie wyznaczenie $\Gamma = \Gamma(A)$. Zgodnie z lematem 5.1 algorytm Cover-Make może do pokrycia Γ' wstawiać elementy z A'' w kolejności, w jakiej występują one w odczycie według pokrycia.

COVER-MERGE(Γ', Γ'')

Wejście: Γ', Γ'' – *łączone pokrycia*
Wyjście: *pokrycie będące wynikiem połączenia*

```
 1  for i ← 1 to |Γ''| do
 2      if element początkowy Γ''[i] jest większy niż element końcowy Γ'[|Γ'|] then
 3          Dodaj do Γ' pustą listę
 4      j ← |Γ'|
 5      while Γ''[i] nie jest pusta and j > 1 do
 6          Znajdź największy symbol s w Γ''[i] mniejszy niż element końcowy Γ'[j − 1]
 7          Przenieś symbole większe niż s z Γ''[i] do Γ'[j]
 8          j ← j − 1
 9      Dołącz pozostałą część listy Γ''[i] (jeśli nie jest pusta) do Γ'[1]
10  return Γ'
```

Rys. 5.2. Ogólny schemat algorytmów łączących pokrycia. (Jeżeli w wierszu 6. nie zostanie znaleziony symbol s, to żadne symbole nie są przenoszone w wierszu 7.)
Fig. 5.2. A general scheme of the cover merging algorithm. (If there is no such a symbol s in line 6, no symbols are moved in line 7.)

Twierdzenie 5.1. *Algorytm* COVER-MERGE *łączący pokrycia (rys. 5.2) jest poprawny.*

Dowód. Niezmiennikiem pętli zewnętrznej algorytmu jest: Γ' *jest pokryciem ciągu* $A'\Gamma(A'')[1] \ldots \Gamma(A'')[i-1]$. Z definicji $\Gamma(A'')[1] \ldots \Gamma(A'')[0]$ oznacza ciąg pusty.

Przed rozpoczęciem wykonywania pętli zewnętrznej, Γ' zawiera pokrycie ciągu A', a więc niezmiennik jest spełniony. Niech teraz dla ustalonego i, Γ' zawiera pokrycie dla $A'\Gamma(A'')[1] \ldots \Gamma(A'')[i-1]$. Lista $\Gamma''[i]$, uporządkowana malejąco, jest przetwarzana jak poniżej. Elementy $\Gamma''[i]$ większe niż symbol kończący ostatnią listę Γ' są przenoszone do Γ' jako nowa lista. Dla wszystkich $1 < j \leq |\Gamma'|$, symbole większe niż element kończący $\Gamma'[j-1]$, ale jednocześnie mniejsze niż element kończący $\Gamma'[j]$ są przenoszone do $\Gamma'[j]$. Ewentualne pozostałe symbole muszą być mniejsze niż element kończący $\Gamma'[1]$ i są przenoszone do $\Gamma'[1]$. Procedura ta umieszcza symbole dokładnie tak samo, jak robi to algorytm COVER-MAKE, a więc po inkrementacji i, Γ' zawiera pokrycie dla $A'\Gamma(A'')[1] \ldots \Gamma(A'')[i-1]$ i niezmiennik pętli jest spełniony.

Po zakończeniu wykonywania pętli zewnętrznej, Γ' zawiera pokrycie dla $A'E(A'')$, a więc na podstawie lematu 5.1 twierdzenie jest prawdziwe. ■

Lemat 5.4. *Podczas łączenia pokrycia* Γ'*, zawierającego na wejściu* x *list, i* Γ''*, elementy każdej z list* Γ'' *są porównywane z elementami kończącymi najwyżej* $x+1$ *list* Γ'.

Dowód. Dowód przez indukcję względem i, które jest numerem listy z Γ''. Dla $i = 1$ pokrycie Γ' zawiera x list, więc lemat jest w oczywisty sposób prawdziwy. Dla $i > 1$ założone zostanie, że element kończący $\Gamma''[i-1]$ był umieszczony na końcu pewnej k-tej listy Γ', ale nie dalej niż

LICS(A)

Wejście: *A – ciąg, w którym wyznaczany jest podciąg LICS*
Wyjście: *długość podciągu LICS oraz indeks elementu będącego ostatnim w rotacji, w której długość podciągu LIS równa jest długości podciągu LICS*

```
1  Przesuń cyklicznie A, aby a_1 był maksymalnym elementem w A
2  Γ'' ← Cover-Make(A_1^n);  j ← n
3  if a_n jest punktem stopu then ℓ ← |Γ''|; i ← n else ℓ ← 0; i ← 0
4  Usuń a_n z Γ''
5  for k ← n − 1 downto 1 do
6      if a_k jest punktem stopu then
7          Γ' ← Cover-Make(A_{k+1}^j)
8          Γ'' ← Cover-Merge(Γ', Γ'')        {Wyznacza Γ(A_{k+1}^k) przez łączenie Γ', Γ''}
9          if ℓ < |Γ''| then ℓ ← |Γ''|; i ← k
10         j ← k
11     Usuń a_k z Γ''
12 return ℓ, i
```

Rys. 5.3. Ogólny schemat algorytmów wyznaczania długości podciągu LICS
Fig. 5.3. A general scheme of the algorithm computing LICS length

$x-1$ list od końca. Ostatnie porównanie było wykonane z elementem kończącym listę $\Gamma'[k-1]$ (co najwyżej x list od końca Γ'). Element kończący listę $\Gamma''[i]$ jest większy niż element kończący $\Gamma''[i-1]$, a więc ostatnie porównanie przy przetwarzaniu listy $\Gamma''[i+1]$ będzie wykonane co najwyżej z elementem kończącym listę $\Gamma'[k]$. Lista ta znajduje się na pozycji o jeden w prawo od listy, z którą był porównywany element kończący $\Gamma''[i-1]$. Podczas przetwarzania jednej listy z Γ'' rozmiar pokrycia Γ' może wzrosnąć najwyżej o 1. Z powyższego, podczas przetwarzania i-tej listy Γ'' będzie analizowanych co najwyżej $x+1$ list Γ'. ■

Ogólny schemat proponowanego algorytmu rozwiązywania problemu LICS przedstawiony jest na rys. 5.3. Ciąg A jest rotowany n razy po jednym symbolu i dla niektórych (w najgorszym przypadku dla wszystkich) rotacji wyznaczane jest pokrycie. Algorytm zwraca długość podciągu LICS oraz indeks punktu stopu, dla którego został on znaleziony, dzięki czemu, jeśli to konieczne, w prosty sposób można wyznaczyć żądany podciąg.

Twierdzenie 5.2. *Algorytm* LICS *wyznacza długość podciągu LICS oraz indeks ostatniego elementu, który należy do podciągu LICS.*

Dowód. Niech niezmiennikiem pętli będzie: Γ'' *przechowuje pokrycie* A_{j+1}^{k}, *a* ℓ *jest największą z długości podciągu LIS dla wszystkich rotacji kończących się punktami stopu o indeksach z zakresu* $[k+1,n]$.

Przed rozpoczęciem wykonywania pętli Γ'' przechowuje pokrycie dla $A_{n+1}^{n-1} = A_1^{n-1}$, a ℓ – długość podciągu LIS dla A_1^n, jeśli a_n jest punktem stopu, wobec czego niezmiennik jest spełniony. Niech teraz niezmiennik będzie prawdziwy dla pewnego k. Jeśli a_k nie jest punktem

stopu, to a_k jest usuwany z Γ'', a ℓ się nie zmienia, więc dekrementacja k powoduje, że niezmiennik pętli dalej jest spełniony. Niech teraz a_k będzie punktem stopu. W wierszu 7. wyznaczane jest pokrycie Γ' dla A^j_{k+1}, a Γ'' przechowuje w tym czasie pokrycie dla A^k_{j+1}. Następnie, w wierszu 8., wyznaczane jest pokrycie dla A^k_{k+1}. W wierszu 9. wartością ℓ staje się długość najdłuższego podciągu LIS spośród wszystkich rotacji kończących się punktami stopu o indeksach z zakresu $[k,n]$. W wierszu 10. do j wpisywane jest k, a więc Γ'' przechowuje pokrycie dla A^k_{j+1}. Następnie a_k jest usuwane z Γ'' i dekrementacja k powoduje, że niezmiennik pętli dalej jest spełniony. Po zakończeniu wykonywania pętli ℓ przechowuje maksymalną wartość długości podciągu LIS dla wszystkich rotacji kończących się punktami stopu. Z lematu 5.3 wartość ta jest długością podciągu LICS dla całego ciągu. ■

Jako ciekawostkę można zauważyć, że ponieważ punktów stopu jest $\Theta(\ell)$, więc algorytm wyznaczający pokrycie algorytmem COVER-MAKE tylko dla rotacji kończących się punktami stopu, a więc budowania go za każdym razem na podstawie samego ciągu ma złożoność czasową $O(n\ell\log\log n)$. Poniżej zostaną przedstawione szybsze algorytmy, w których istotna będzie znajomość pokryć cząstkowych. Kluczową kwestią w tych algorytmach jest wybór odpowiednich struktur danych do reprezentacji pokryć.

5.2.2. Reprezentacja pokrycia za pomocą list

Najprostszym sposobem reprezentacji pokrycia jest zastosowanie uporządkowanego zbioru list uporządkowanych malejąco. Pierwszym etapem działania algorytmu jest wykonanie wstępnej rotacji (złożoność czasowa $O(n)$), po której następuje wyznaczenie pokrycia dla ciągu A. Złożoność czasowa tej fazy algorytmu to $t^{\text{init}} = O(n\log\ell)$. Wybierając do budowy początkowego pokrycia algorytm stosujący drzewa van Emde Boasa [209, 210], dostaje się $\tau^{\text{init}} = O(n\log\log n)$.

W kolejnych iteracjach z pokrycia usuwane są poszczególne elementy ciągu począwszy od ostatniego. Kluczowe znaczenie dla złożoności czasowej tej czynności mają sposób znalezienia elementu do usunięcia w pokryciu oraz samo usunięcie. Tworząc na etapie inicjalizacji tablicę zawierającą wskaźniki do wszystkich elementów, można każdy z nich znaleźć w pokryciu w czasie stałym. Usuwanie elementu z listy wymaga wykonania $\tau^{\text{del}} = O(1)$ operacji.

W niektórych przebiegach pętli (definiowanych przez pozycje punktów stopu) wykonywana jest budowa pokrycia dla fragmentu przesuniętego cyklicznie ciągu (wiersz 7). Przyjmując, że liczba elementów pomiędzy $(i-1)$-szym a i-tym punktem stopu (z definicji element a_n jest zerowym punktem stopu) to n_i, złożoność czasowa tej operacji wynosi $\tau^{\text{build}}(n_i) = O(n_i\log\ell)$. Punktów stopu jest $n^{\text{stop}} = \Theta(\ell)$ i wiadomo, że $\sum_{i=1}^{n^{\text{stop}}} n_i = O(n)$.

W wierszu 8. wykonywane jest łączenie pokryć. Ponieważ pokrycie Γ' zawiera n_i elementów, więc składa się z $O(\min(n_i, \ell))$ list. Z lematu 5.4 wynika, że dla ustalonego punktu stopu sumaryczna liczba przebiegów wewnętrznej pętli algorytmu COVER-MERGE jest $O(\min(n_i, \ell)\ell)$. W każdym przebiegu tej pętli wyznaczany jest symbol s, co wymaga przeglądnięcia aktualnie przetwarzanej listy pokrycia Γ''. Lista ta jest przeglądana począwszy od elementu największego. W momencie znalezienia elementu spełniającego określony w algorytmie warunek, wszystkie większe od niego elementy są przenoszone do pokrycia Γ'. Ponieważ sumaryczna liczba elementów w pokryciu Γ'' jest $O(n)$, więc sumaryczna złożoność czasowa operacji znajdowania elementów podziału dla jednego łączenia pokryć wynosi $\tau^{\text{find}}(n_i) = O(\min(n_i, \ell)\ell + n)$. W wierszu 9. algorytmu COVER-MERGE lista jest rozcinana (złożoność czasowa $\tau^{\text{split}} = O(1)$) oraz sklejana z inną listą (złożoność czasowa $\tau^{\text{join}} = O(1)$). Dysponując wszystkimi składnikami złożoności czasowej, można zapisać wyrażenie na sumaryczną złożoność czasową algorytmu LICS.

Wniosek 5.1. *Złożoność czasowa algorytmu* LICS *wynosi:*

$$\tau^{\text{init}} + n\tau^{\text{del}} + \sum_{i=1}^{n^{\text{stop}}} \left(\tau^{\text{build}}(n_i) + \tau^{\text{find}}(n_i) + O(\min(n_i, \ell)\ell) \left(\tau^{\text{split}} + \tau^{\text{join}}\right)\right). \tag{5.1}$$

Podstawienie do (5.1) złożoności czasowych poszczególnych składników daje:

$$\begin{aligned}
&O(n\log\ell) + nO(1) + \\
&+ \sum_{i=1}^{n^{\text{stop}}} \Big(O(n_i\log\ell) + O(\min(n_i,\ell)\ell + n) + \min(n_i,\ell)\ell(O(1) + O(1))\Big) = \\
&= O(n\log\ell) + O(n) + O(n\log\ell) + O(n\ell) + O\left(\min\left(n\ell, \ell^3\right)\right) = O(n\ell),
\end{aligned}$$

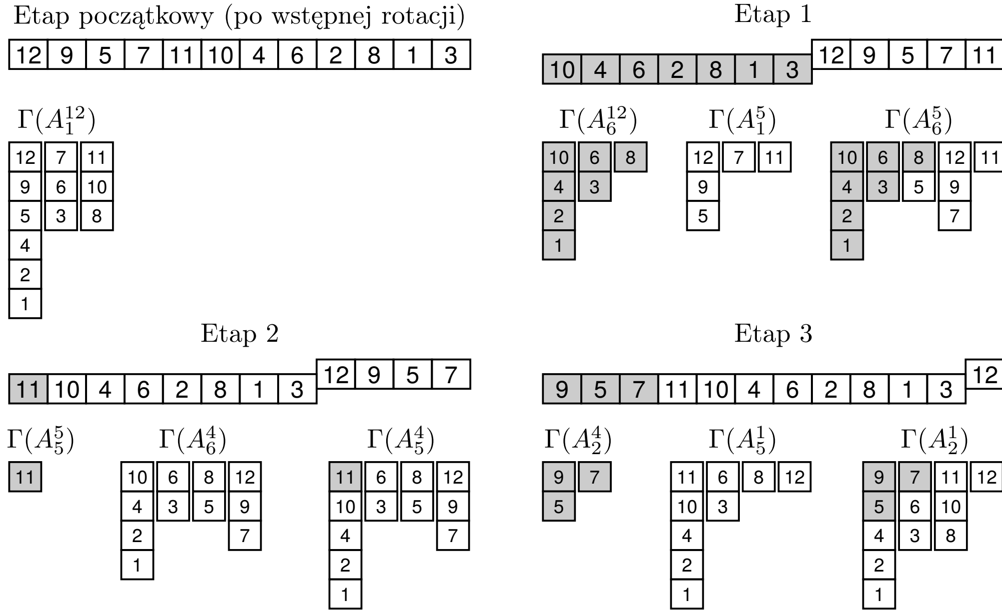
co można podsumować w postaci wniosku:

Wniosek 5.2. *Złożoność czasowa algorytmu* LICS *dla reprezentacji listowej pokrycia jest*

$$O(n\ell). \tag{5.2}$$

5.2.3. Reprezentacja pokrycia za pomocą drzew zrównoważonych

Złożoność czasowa powyższego algorytmu może być poprawiona przez zastosowanie nieco bardziej wyrafinowanej struktury danych do reprezentacji pokrycia. Jednym z możliwych rozwiązań jest reprezentowanie każdej z list pokrycia jako zrównoważonego drzewa poszukiwań binarnych, np. drzewa czerwono-czarnego [29, 105, 190]. (Innymi możliwościami są np. drzewa AVL [1], drzewa typu splay [194, 198]). Zaletą tego podejścia jest potencjalnie szybszy sposób wyznaczania miejsca podziału listy pokrycia, gdyż w tym przypadku $\tau^{\text{find}}(n_i) =$



Rys. 5.4. Przykład działania algorytmu wyznaczania podciągu LICS opartego na reprezentacji listowej pokrycia dla ciągu A = 4 6 2 8 1 3 12 9 5 7 11 10. Komórki zaznaczone na szaro oznaczają elementy, które zostały zrotowane w kolejnym etapie. Niektóre elementy są opuszczone, aby wskazać miejsce, w którym zaczyna się oryginalny ciąg A

Fig. 5.4. Example of the LICS-computing algorithm with list-based cover representation in work for sequence A = 4 6 2 8 1 3 12 9 5 7 11 10. The greyed cells denote the cells which are rotated in each stage. Some cells are pushed down to indicate the place where the original sequence A starts

$O(\min(n_i,\ell)\ell\log n)$. Wadą jest za to dłuższy czas podziału i łączenia tych list, gdyż $\tau^{\text{split}} = \tau^{\text{join}} = O(\log n)$ [35, 130]. Także usuwanie elementu trwa nieco dłużej niż dla reprezentacji listowej – $\tau^{\text{del}} = O(\log n)$. Tworzenie pierwszego pokrycia można wykonać za pomocą algorytmu klasycznego, co daje $\tau^{\text{init}} = O(n\log\ell)$, ale można także zastosować tu drzewa van Emde Boasa [209, 210] (por. podrozdz. 2.4.2), dzięki którym możliwe jest uzyskanie pokrycia w reprezentacji listowej w czasie $O(n\log\log n)$. Dla każdej posortowanej listy pokrycia można (znając jej rozmiar) utworzyć drzewo czerwono-czarne w czasie liniowym w zależności od rozmiaru tej listy. Dzięki temu złożoność czasowa budowy pokrycia dla reprezentacji za pomocą drzew czerwono-czarnych to $\tau^{\text{init}}(n_i) = O(n\log\log n)$. Warto również zauważyć, że w taki sam sposób można budować pokrycia dla elementów przesuniętych cyklicznie, a więc $\tau^{\text{build}}(n_i) = O(n_i\log\log n)$. Podstawiając te złożoności czasowe składników algorytmu do (5.1), otrzymuje się złożoność czasową algorytmu LICS dla reprezentacji pokrycia za pomocą drzew czerwono-czarnych:

$$
\begin{aligned}
&O(n\log\log n)+nO(\log n)+\\
&+\sum_{i=1}^{n^{\mathrm{stop}}}\left(O(n_i\log\log n)+O\left(\min\left(n_i,\ell\right)\ell\log n\right)+O\left(\min\left(n_i,\ell\right)\ell\right)O(\log n)\right)=\\
&O(n\log\log n)+O(n\log n)+O(n\log\log n)+O\left(\min\left(n\ell,\ell^3\right)\log n\right)+\\
&+O\left(\min\left(n\ell,\ell^3\right)\log n\right)=\\
&=O\left(n\log n+\min\left(n\ell,\ell^3\right)\log n\right).
\end{aligned}
$$

Wniosek 5.3. *Złożoność czasowa algorytmu* LICS *dla reprezentacji pokrycia za pomocą drzew zrównoważonych jest*

$$O\left(n\log n+\min\left(n\ell,\ell^3\right)\log n\right). \tag{5.3}$$

5.2.4. Reprezentacja pokrycia za pomocą list drzew zrównoważonych

To, dla której z reprezentacji pokrycia omówionych w poprzednich podrozdziałach złożoność czasowa jest lepsza, zależy od długości podciągu wyjściowego. Dzięki temu, że możliwe jest oszacowanie długości rozwiązania z dokładnością do czynnika 2 na podstawie rozmiaru pokrycia dla ciągu wejściowego można zaproponować algorytm hybrydowy, w którym najpierw wyznaczane jest pokrycie za pomocą reprezentacji listowej, a następnie podejmowana jest decyzja o tym, której reprezentacji pokrycia użyć. Możliwe jest jednak zaproponowanie jeszcze innego sposobu reprezentacji, który będzie łączył zalety obu wcześniejszych w jednej strukturze danych, oferując złożoność czasową nie gorszą niż każdy z omówionych do tej pory sposobów.

Idea tego rozwiązania opiera się na reprezentacji każdej z list pokrycia jako uporządkowanej listy drzew czerwono-czarnych o pewnym rozmiarze $\Theta(e)$, który zostanie wyznaczony nieco później. Ponadto, z każdym z tych drzew czerwono-czarnych związana jest pewna dodatkowa zmienna przechowująca wartość minimalną w drzewie, aby była ona dostępna w czasie stałym. W tym przypadku złożoności czasowe składowych algorytmu to: $\tau^{\mathrm{init}}=O(n\log\log n)$, $\tau^{\mathrm{del}}=O(\log e)$, $\tau^{\mathrm{build}}(n_i)=O(n_i\log\log n)$, $\tau^{\mathrm{split}}=\tau^{\mathrm{join}}=O(\log e)$. Złożoność czasowa operacji wyszukiwania punktu podziału listy zależy od rozmiaru drzewa na dwa sposoby. Po pierwsze, należy przeglądnąć listę drzew w celu znalezienia drzewa zawierającego miejsce podziału. Po drugie, w drzewie rozmiaru $\Theta(e)$ należy znaleźć tę wartość. W związku z tym, że w pokryciu Γ'' liczba drzew zrównoważonych (łącznie we wszystkich listach) jest $O(n/e+\ell)$, to:

$$\tau^{\mathrm{find}}(n_i)=O\left(\min(n_i,\ell)\ell\log e+\frac{n}{e}+\ell\right).$$

Podstawienie złożoności składowych do (5.1) daje

$$
\begin{aligned}
&O(n\log\log n) + nO(\log e) + \\
&+\sum_{i=1}^{n^{\mathrm{stop}}} \Big(O(n_i \log\log n) + O(\min(n_i,\ell)\ell\log e + \frac{n}{e} + \ell) + O(\min(n_i,\ell)\ell) O(\log e) \Big) = \\
&= O(n\log\log n) + O(n\log e) + \\
&+ O(n\log\log n) + O\left(\min\left(n\ell,\ell^3\right)\log e + \frac{n\ell}{e} + \ell^2 \right) + O\left(\min\left(n\ell,\ell^3\right)\log e\right) = \\
&O\left(n\log\log n + \min(n\ell, n+\ell^3)\log e + \frac{n\ell}{e} \right).
\end{aligned}
$$

Łatwo można stwierdzić, że powyższe wyrażenie ma wartość minimalną dla $e = \Theta(\lceil n/\ell^2 \rceil)$, wobec czego złożoność czasowa tej wersji algorytmu jest

$$
O\left(n\log\log n + \min\left(n\ell, n+\ell^3\right)\log\left\lceil \frac{n}{\ell^2} \right\rceil \right).
$$

Wartość e można oszacować z dokładnością do czynnika 4 po wykonaniu budowy pierwszego pokrycia na podstawie długości podciągu LIS w początkowej rotacji.

Ostatnią kwestią, którą należy rozważyć, jest to, czy listy drzew czerwono-czarnych mogą być efektywnie aktualizowane, bez negatywnego wpływu na złożoność czasową algorytmu. W szczególności chodzi o to, aby po każdej operacji (podziale, połączeniu, usunięciu elementu) na drzewie czerwono-czarnym można było szybko zagwarantować, że przechowuje ono $\Theta(e)$ elementów. W tym celu na drzewa nakładane jest ograniczenie, że ich rozmiar zawsze musi być w zakresie $[e, 2e]$. Jedynym wyjątkiem jest sytuacja, w której lista pokrycia ma mniej niż e elementów. (Wtedy lista zawiera tylko jedno drzewo). Po każdej z operacji usuwania elementu z drzewa (złożoność czasowa tej operacji jest $O(\log e)$) sprawdzane jest, czy rozmiar drzewa zmalał poniżej e. Jeśli takie coś miało miejsce, to drzewo jest łączone z drzewem sąsiednim w liście pokrycia (złożoność czasowa tej operacji jest $O(\log e)$) i otrzymywane jest drzewo, którego rozmiar mieści się w przedziale $[e, 3e]$. Następnie, jeśli to konieczne, tj. rozmiar uzyskanego drzewa jest większy niż $2e$, drzewo to jest dzielone na możliwie równe części, tak aby rozmiar każdego z otrzymanych drzew mieścił się w przedziale $[e, 2e]$. Do wykonania tego podziału konieczne jest znalezienie mediany wśród elementów należących do drzewa. Aby móc znajdować medianę szybko, należy rozszerzyć wszystkie węzły drzewa czerwono-czarnego o dodatkowe pole przechowujące liczbę węzłów należących do poddrzewa, którego korzeniem jest dany węzeł. Informacje te można aktualizować w trakcie wykonywania operacji na drzewie bez pogarszania ich złożoności czasowej pozostałych operacji [130, 35, 156]. Złożoność czasowa procesu normalizacji rozmiaru drzewa, który został opisany powyżej, jest $O(\log e)$, czyli tyle samo co złożoność czasowa operacji usuwania elementu z drzewa, a więc nie wpływa negatywnie na złożoność czasową całego algorytmu.

W podobny sposób rozmiar drzewa normalizuje się po podziale i łączeniu drzew. Po każdym podziale otrzymywane są dwa drzewa, z których jedno przenoszone jest do innej listy pokrycia. W obu tych listach pokrycia wykonywane są sprawdzenia, czy pozostałe (przeniesione) drzewo jest poprawnego rozmiaru, a jeśli nie, to dokonywane jest jego połączenie z drzewem sąsiednim i ewentualny podział otrzymanego drzewa na dwa mniejsze. Złożoność czasowa tego procesu normalizacji rozmiaru drzewa także w tym przypadku jest $O(\log e)$, a więc nie więcej niż złożoność czasowa operacji podziału i łączenia drzew z pokrycia. Z powyższego wynika, że możliwe jest utrzymywanie rozmiaru drzew w zakresie $[e, 2e]$ bez wpływu na złożoność czasową całego algorytmu. Wobec tego można sformułować wniosek:

Wniosek 5.4. *Złożoność czasowa algorytmu* LICS *dla reprezentacji pokrycia za pomocą list drzew zrównoważonych jest*

$$O\left(n \log\log n + \min\left(n\ell, n+\ell^3\right) \log\left\lceil \frac{n}{\ell^2} \right\rceil\right). \tag{5.4}$$

5.3. Podsumowanie

W niniejszym rozdziale rozważany był problem najdłuższego cyklicznego podciągu rosnącego (LICS). W momencie proponowania algorytmów autorskich złożoność czasowa w przypadku pesymistycznym dla najlepszych znanych metod była: $O(n\ell)$ [40] i $O(n^{3/2})$ [199]. Powodowało to, że ostatni z proponowanych wariantów miał najlepszą złożoność czasową dla

$$\ell = \omega(\log n) \qquad \text{oraz} \qquad \ell = o\left(\frac{n^{1/2}}{\log^{1/2} n}\right),$$

a nie gorszą niż te algorytmy dla

$$\ell = O(n^{1/2}).$$

W związku z ostatnią pracą Tiskina [200], który ulepszył algorytm z [199] osiągając złożoność czasową $O(n \log^2 n)$ algorytm proponowany oferuje najlepszą złożoność czasową dla

$$\ell = \omega(\log n) \qquad \text{oraz} \qquad \ell = o\left((n \log n)^{1/3}\right).$$

Kwestią otwartą pozostaje odpowiedź na pytanie, jaki jest kres dolny złożoności czasowej dla problemu LICS. Do tej pory wiadomo jedynie, że nie może on być niższy od kresu dolnego złożoności czasowej dla problemu LIS, gdyż gdyby było inaczej, to za pomocą algorytmu dla problemu LICS można by rozwiązać problem LIS w asymptotycznie takim samym czasie. W tym celu wystarczyłoby w ciągu $A = a_1 a_2 \ldots a_n$ znaleźć maksymalny element x, skonstruować ciąg $A^* = a_1 a_2 \ldots a_n (x+1) \ldots (x+n)$ i wyznaczyć dla niego podciąg LICS. W tak skonstruowanym ciągu A^* podciągi LICS i LIS są identyczne, a więc wystarczy z wyniku odrzucić sufiks $(x+1) \ldots (x+n)$, aby otrzymać podciąg LIS dla ciągu A.

6. Najdłuższy podciąg rosnący w przesuwającym się oknie

6.1. Wprowadzenie

Problem najdłuższego podciągu rosnącego w przesuwającym się oknie (ang. *longest increasing subsequence in sliding window*, LISW) jest uogólnieniem problemu najdłuższego podciągu rosnącego (LIS). W problemie tym rozważa się wszystkie spójne podciągi ciągu wejściowego (okna) określonego rozmiaru i wyszukuje najdłuższy podciąg LIS spośród istniejących w tych oknach. Można to sformułować formalnie w następujący sposób:

Problem 6.1 (Najdłuższy podciąg rosnący w przesuwającym się oknie, LISW)**.** *Dla ciągu* $A = a_1a_2 \ldots a_n$ *oraz rozmiaru okna u znaleźć najdłuższy spośród podciągów rosnących dla wszystkich ciągów* A_i^{i+u-1}*, gdzie* $1 \leq i \leq n-u+1$.[1]

Przykład 6.1 (Najdłuższy podciąg rosnący w przesuwającym się oknie, LISW)**.** *Dla ciągu liczbowego* $A =$ 4 6 $\underline{2}$ $\underline{8}$ 1 3 $\underline{12}$ 9 5 7 11 10 *najdłuższym podciągiem rosnącym w przesuwającym się oknie rozmiaru* $u = 5$ *jest m.in.* $A' =$ 2 8 12*. W ciągu A podkreślono symbole tworzące podciąg LISW.*

Problem LISW zdefiniowali po raz pierwszy Albert i in. [4] w kilku wersjach, m.in. w wersji podanej powyżej. W swojej pracy zaproponowali także algorytm rozwiązujący najogólniejszy z tych wariantów, zdefiniowany następująco:

Problem 6.2 (Najdłuższy podciąg rosnący w przesuwającym się oknie – wersja lokalna)**.** *Dla ciągu* $A = a_1a_2 \ldots a_n$ *oraz rozmiaru okna u znaleźć najdłuższy podciąg rosnący dla każdego ciągu* A_i^{i+u-1}*, gdzie* $1 \leq i \leq n-u+1$.

Algorytmy zaproponowane dla wersji problemu według def. 6.2 mogą być także zastosowane do rozwiązywania problemu w wersji z def. 6.1 przy zachowaniu takich samych złożoności czasowych. W niniejszym rozdziale rozważany będzie problem LISW, zgodnie z def. 6.1. Przyjęte jest również założenie (takie samo jak u Alberta i in. [4]), że ciąg A jest permutacją liczb całkowitych z zakresu $[1, n]$. Jeśli tak nie jest, to podobnie jak w przypadku problemu LICS można taki ciąg sprowadzić do ciągu spełniającego to założenie w czasie $O(n \log n)$ (por. podrozdz. 5.1).

---

[1] Notacja A_i^j będzie stosowana w niniejszym rozdziale w miejsce bardziej naturalnej notacji $A_{i,j}$, ponieważ, z uwagi na zakres rozpatrywanych indeksów, obie te notacje w tym rozdziale są równoważne, a pierwsza z nich jest istotnie krótsza, co ma znaczenie z uwagi na dość skomplikowane wyrażenia określające zakresy podciągów.

Albert i in. [4] wprowadzili w swojej pracy nową strukturę danych reprezentującą pewną część tableau Younga (por. podrozdz. 2.4.1) dla okna rozmiaru u. Pokazali również, jak tę strukturę zbudować dla ciągu A_1^u, a następnie uaktualniać ją wykonując iteracyjnie dla każdego okna A_i^{i+u-1} usunięcie z niej symbolu a_i oraz wstawienie symbolu a_{i+u}, dzięki czemu otrzymuje się strukturę reprezentującą fragment tableau Younga dla kolejnego okna. Na podstawie tej struktury danych można odczytać zarówno podciąg LIS w bieżącym oknie, jak i jego długość. Złożoność czasowa algorytmu Alberta i in. jest $O(n \log\log n + n\ell)$, gdzie ℓ oznacza długość podciągu LISW.

Inny algorytm dla problemu LISW został zaproponowany w pracy Chena i in. [40]. Autorzy zastosowali w nim reprezentację ciągu w każdym oknie za pomocą podziału ciągu na tzw. antyłańcuchy. Pokazali również, jak tę reprezentację modyfikować przy przesuwaniu okna o kolejne symbole. Złożoność czasowa tego algorytmu jest $O(n\ell)$.

Tiskin w [200] badał problem nieco ogólniejszy, w którym przedmiotem zainteresowania są wszystkie okna dowolnych rozmiarów. Algorytm zaproponowany przez niego można łatwo zastosować do problemu LISW, otrzymując złożoność czasową $O(n \log^2 n)$.

Problem LISW można też traktować jako uogólnienie problemu LICS, ponieważ problem LICS można sformułować następująco:

Problem 6.3 (Najdłuższy cykliczny podciąg rosnący, LICS – sformułowanie oparte na problemie LISW)**.** *Dla danego ciągu $A = a_1a_2 \ldots a_n$ znaleźć podciąg LISW w ciągu AA, przy założeniu że długość okna to n.*

6.2. Algorytmy

6.2.1. Podstawowe koncepcje

W tym i kolejnych podrozdziałach zostaną zaprezentowane algorytmy rozwiązywania problemu LISW zaproponowane przez autora w [68]. Algorytmy te oparte są na podobnych ideach co algorytmy zaproponowane dla problemu LICS w rozdz. 5. Podstawowa koncepcja polega na reprezentacji spójnego podciągu z bieżącego okna za pomocą pokrycia zachłannego (por. podrozdz. 2.4.2). W kolejnych podrozdziałach będzie wyznaczany właściwie nie sam podciąg LISW, a jego długość oraz to gdzie on się znajduje, jednak dzięki wiedzy o miejscu jego położenia łatwo można w czasie $O(u \log\log n)$ wyznaczyć podciąg LISW.

Dla łatwiejszego wykazania poprawności proponowanego algorytmu wygodnie jest założyć, że w pierwszym etapie algorytmu wyznaczane są pokrycia zachłanne dla następujących spójnych podciągów: $A_1^u, A_{u+1}^{u+u}, \ldots, A_{iu+1}^{iu+u}, \ldots, A_{\lceil n/u \rceil u-u+1}^{\lceil n/u \rceil u}$ (dla uproszczenia wywodu zakłada

się tu, że n jest całkowitą wielokrotnością u). Długość podciągu LIS w ciągu A_{iu+1}^{iu+u} oznaczana będzie przez ℓ_i.

Lemat 6.1. *Długość podciągu LIS w dowolnym oknie rozmiaru u podciągu spójnego A_{iu+1}^{iu+2u} jest nie większa niż $\ell_i + \ell_{i+1}$.*

Dowód. Dowód jest natychmiastowy, wystarczy bowiem zauważyć, że długość podciągu LIS w ciągu A_{iu+1}^{iu+2u} nie może być większa niż $\ell_i + \ell_{i+1}$, wobec czego długość podciągu LIS w żadnym podciągu spójnym tego ciągu nie może być większa. ■

Lemat 6.2. *Niech dany będzie podciąg A_{iu+1}^{iu+2u} szerokości dwa razy większej niż rozmiar okna, u. Dla każdego okna A_{iu+1+k}^{iu+u+k} rozmiaru u w nim, gdzie $0 \leq k \leq u$, symbol znajdujący się na początku ostatniej listy pokrycia dla tego okna jest albo punktem stopu dla prawego skrajnego okna ciągu A_{iu+1}^{iu+2u}, tj. $A_{(i+1)u+1}^{(i+1)u+u}$ albo symbolem z lewego skrajnego okna, A_{iu+1}^{iu+u}.*

Dowód. Z lematu 5.1 wiadomo, że

$$\Gamma(A_{iu+1+k}^{iu+u+k}) = \Gamma(A_{iu+1+k}^{iu+u} E(A_{iu+u+1}^{iu+u+k})).$$

Algorytm Cover-Make tworzący pokrycie dla ciągu A_{iu+1+k}^{iu+u+k}, tworzy najpierw pokrycie Γ dla A_{iu+1+k}^{iu+u}. Następnie rozszerza je elementami z kolejnych list $\Gamma(A_{iu+u+1}^{iu+u+k})$. Element rozpoczynający każdą z tych list jest punktem stopu dla $A_{(i+1)u+1}^{(i+1)u+u}$ i może być dołączony do jakiejś listy w Γ bądź rozpocząć nową listę w Γ. Ponieważ symbole w listach dołączanego pokrycia są uporządkowane malejąco, więc tylko punkt stopu może być dołączony do tworzonego pokrycia na początku nowej listy. Z tego wynika, że ostatnia lista w tworzonym pokryciu musi się zaczynać albo od jakiegoś punktu stopu dla $A_{(i+1)u+1}^{(i+1)u+u}$, albo też od symbolu z A_{iu+1+k}^{iu+u}. ■

Lemat 6.3. *Długość podciągu LISW dla ciągu A_{iu+1}^{iu+2u} przy rozmiarze okna u jest równa największej z: długości podciągu LIS dla wszystkich okien A_{iu+1+k}^{iu+u+k}, gdzie $1 \leq k \leq u$ kończących się punktem stopu z $A_{(i+1)u+1}^{(i+1)u+u}$, bądź długości podciągu LIS dla A_{iu+1}^{iu+u}.*

Dowód. Z lematu 6.2 wynika, że na początku ostatniej listy pokrycia dla dowolnego okna ciągu A_{iu+1}^{iu+2u} znajduje się albo punkt stopu, albo też symbol z lewego skrajnego okna, A_{iu+1}^{iu+u}. Jeśli jest to symbol z A_{iu+1}^{iu+u}, oznacza to, że podciąg LIS w tym oknie kończący się na tym symbolu składa się tylko z symboli z A_{iu+1}^{iu+u}, a najdłuższy podciąg LIS składający się z takich symboli to podciąg LIS dla A_{iu+1}^{iu+u}.

Jeśli jednak symbolem rozpoczynającym ostatnią listę w pokryciu dla danego okna jest punkt stopu dla $A_{(i+1)u+1}^{(i+1)u+u}$, to jest on ostatnim symbolem podciągu LIS w tym oknie. Dla dowolnego takiego punktu stopu a_j najdłuższy podciąg rosnący kończący się na a_j spośród wszystkich możliwych okien rozmiaru u można znaleźć w podciągu A_{j-u+1}^{j}.

LISW-RANGE(A, i, Γ^{left}, Γ^{right})

Wejście: A – *ciąg, w którym wyznaczany jest podciąg LISW*
Γ^{left}, Γ^{right} – *pokrycia odpowiednio dla* A_{iu+1}^{iu+u} *oraz* $A_{(i+1)u+1}^{(i+1)u+u}$

Wyjście: *długość podciągu LISW i indeks jego ostatniego elementu w dowolnym oknie rozmiaru u spójnego podciągu* $A_{iu+1}^{(i+1)u+u}$

```
 1  Γ″ ← Γ^right;  j ← (i+1)u+u
 2  if a_j jest punktem stopu dla A_{(i+1)u+1}^{(i+1)u+u} then ℓ ← |Γ″|; s ← j else ℓ ← 0; s ← 0
 3  Usuń a_j z Γ″
 4  for k ← (i+1)u+u−1 downto (i+1)u+1 do
 5      if a_k jest punktem stopu then
 6          Γ′ ← COVER-MAKE(A_{k+1−u}^{j−u})
 7          Γ″ ← COVER-MERGE(Γ′,Γ″)        {Wyznacza Γ(A_{k+1−u}^{k}) przez łączenie Γ′,Γ″}
 8          if ℓ < |Γ″| then ℓ ← |Γ″|; s ← k
 9          j ← k
10      Usuń a_k z Γ″
11  if ℓ < |Γ^left| then ℓ ← |Γ^left|; s ← (i+1)u
12  return ℓ, s
```

Rys. 6.1. Ogólny schemat algorytmów wyznaczania podciągu LISW w przedziale szerokości podwójnego rozmiaru okna

Fig. 6.1. A general scheme of the algorithm computing LISW in a range of width twice as large as window size

Z powyższego wynika, że długość podciągu LISW w ciągu A_{iu+1}^{iu+2u} przy rozmiarze okna u jest największą z wartości: długości podciągu LIS dla A_{iu+1}^{iu+u} lub długości podciągu LIS dla wszystkich okien kończących się punktami stopu z $A_{(i+1)u+1}^{(i+1)u+u}$. ■

Na podstawie powyższych lematów można skonstruować algorytm wyznaczający podciąg LISW w podciągu spójnym A_{iu+1}^{iu+2u} dla rozmiaru okna wynoszącego u (rys. 6.1). Algorytm ten jest podobny do algorytmu LICS. Rozpoczyna on swoje działanie od pokrycia $\Gamma(A_{(i+1)u+1}^{(i+1)u+u})$, z którego usuwane są końcowe symbole ciągu $A_{(i+1)u+1}^{(i+1)u+u}$, aż do osiągnięcia jakiegokolwiek punktu stopu. Następnie tworzone jest pokrycie Γ', które jest łączone z pokryciem Γ'' za pomocą algorytmu COVER-MERGE. W ten sposób otrzymywane jest pokrycie dla okna kończącego się punktem stopu. W pokryciu tym wyznaczana jest długość podciągu LIS i sprawdzane jest, czy jest ona większa niż największa z dotychczas znalezionych wartości długości podciągu LIS. Procedura ta jest powtarzana w pętli, aż zostaną znalezione pokrycia dla wszystkich okien kończących się punktami stopu z $A_{(i+1)u+1}^{(i+1)u+u}$. Na koniec weryfikowane jest, czy jeszcze dłuższy podciąg LIS nie znajduje się w lewym skrajnym oknie, A_{iu+1}^{iu+u}. Opierając się na lemacie 6.3, można sformułować wniosek.

Wniosek 6.1. *Algorytm* LISW-RANGE *(rys. 6.1) poprawnie wyznacza długość podciągu LISW i indeks jego ostatniego elementu w podciągu spójnym ciągu A o rozmiarze podwójnej szerokości okna.*

LISW(A)

Wejście: *A – ciąg, w którym wyznaczany jest podciąg LISW*
Wyjście: *długość podciągu LISW i indeks jego ostatniego elementu*

```
1  ℓ ← 0; s ← 0
2  Γ^right ← Cover-Make(A^n_{n−u+1})
3  for i ← ⌈n/u⌉ − 2 downto 0 do
4      Γ^left ← Cover-Make(A^{iu+u}_{iu+1})
5      ℓ′, s′ ← LISW-Range(i, Γ^left, Γ^right)
6      if ℓ′ > ℓ then ℓ ← ℓ′; s ← s′
7      Γ^right ← Γ^left
8  return ℓ, s
```

Rys. 6.2. Ogólny schemat algorytmów wyznaczania podciągu LISW
Fig. 6.2. A general scheme of the algorithm computing LISW

Wykonując algorytm LISW-Range dla każdego $i = 0, 1, \ldots, \lceil n/u \rceil - 2$ i wybierając największą ze znalezionych długości znajdowana jest długość LISW dla ciagu A oraz indeks wystąpienia ostatniego symbolu tego podciagu.

Twierdzenie 6.1. *Algorytm* LISW *(rys. 6.2) wyznacza długość podciągu LISW dla ciągu wejściowego $A = a_1 \ldots a_n$ przy rozmiarze okna u.*

Dowód. Na podstawie lematu 6.3 wiadomo, że algorytm LISW-Range wyznacza długość podciągu LISW i indeks wystąpienia ostatniego symbolu LISW dla ciągu A^{iu+2u}_{iu+1}. Algorytm ten jest wykonywany iteracyjnie dla wszystkich $0 \le i \le \lceil n/u \rceil - 2$, co powoduje, że każde okno rozmiaru u w A^n_1 jest rozważane w jednym wywołaniu algorytmu LISW-Range. Z otrzymanych wyników wybierany jest największy, który jest wobec tego długością podciągu LISW dla A^n_1. Ponadto, znajdowany jest indeks wystąpienia ostatniego symbolu podciągu LISW. ■

Dla złożoności czasowej algorytmu LISW-Range kluczowe znaczenie ma sposób reprezentacji pokrycia. Posługując się podobnymi argumentami jak w podrozdz. 5.2.2, można wykazać, że złożoność ta dla zakresu indeksów $[iu, iu + 2u]$ jest:

$$\tau_i = u\tau^{\mathrm{del}} + \sum_{j=1}^{n_i^{\mathrm{stop}}} \left(\tau^{\mathrm{build}}(n_j) + \tau^{\mathrm{find}}(n_j) + O(\min(n_j, \ell)\ell) \left(\tau^{\mathrm{split}} + \tau^{\mathrm{join}}\right)\right), \quad (6.1)$$

gdzie poszczególne oznaczenia to:

- n_i^{stop} – liczba punktów stopu dla $A^{(i+1)u+u}_{(i+1)u+1}$, która jest ograniczona z góry przez ℓ_{i+1},
- n_j – liczba elementów pomiędzy j-tym a $(j+1)$-szym punktem stopu dla ciągu $A^{(i+1)u+u}_{(i+1)u+1}$; z definicji $(n_i^{\mathrm{stop}} + 1)$-szym punktem stopu jest ostatni symbol ciągu $A^{(i+1)u+u}_{(i+1)u+1}$,
- τ^{del} – złożoność czasowa usuwania pojedynczego elementu z reprezentacji pokrycia,

- $\tau^{\text{build}}(n_j)$ – złożoność czasowa budowania pokrycia dla elementów, o które okno się przesunęło od ostatniego łączenia pokryć,
- $\tau^{\text{find}}(n_j)$ – złożoność czasowa znajdowania punktów podziału list pokryć,
- τ^{split} – złożoność czasowa podziału listy pokrycia,
- τ^{join} – złożoność czasowa łączenia list pokrycia.

Uzasadnienie wyrażenia 6.1 jest następujące. Dokładnie u razy usuwany jest jeden symbol z pokrycia (czas τ^{del} na każdy symbol). Algorytm LISW-Range wykonuje algorytm Cover-Merge $\Theta(n_i^{\text{stop}})$ razy. Przed każdym łączeniem pokryć tworzone jest pokrycie C' zawierające n_j symboli. Przed każdym podziałem listy pokrycia konieczne jest znalezienie punktu podziału – sumaryczny czas τ^{find} na jedno wykonanie algorytmu Cover-Merge. Następnie listy pokrycia są dzielone i łączone, a sumaryczna liczba podziałów i połączeń jest iloczynem rozmiarów łączonych pokryć, $|\Gamma'| \times |\Gamma''|$ (lemat 5.4).

Rozważając złożoność czasową algorytmu LISW, należy jeszcze wziąć pod uwagę, że $\lceil n/u \rceil$ razy wykonywane jest tworzenie pokryć dla ciągów A_{iu+1}^{iu+u}, gdzie $0 \le i < \lceil n/u \rceil$. Jeśli złożoność czasowa tworzenia jednego takiego pokrycia oznaczona będzie przez τ^{init}, to wyrażenie na sumaryczną złożoność czasową algorytmu LISW przyjmuje postać:

$$\sum_{i=0}^{\lceil n/u \rceil - 1} \left(\tau^{\text{init}} + \tau_i \right). \tag{6.2}$$

Ponieważ algorytm wyznaczania podciągu LISW w przedziale jest podobny do algorytmu wyznaczania podciągu LICS, więc także złożoności czasowe składowych tych algorytmów są często analogiczne. W związku z tym poniżej dokładniejsze rozważania zostaną przeprowadzone jedynie tam, gdzie jest to konieczne, a w pozostałych przypadkach należy przyjąć, że znajdują zastosowanie te same argumenty, jakie były przytaczane przy okazji wyznaczania złożoności czasowych algorytmów dla różnych reprezentacji pokrycia dyskutowanych w rozdziale 5.

6.2.2. Reprezentacja pokrycia za pomocą list

W listowej reprezentacji pokrycia złożoność czasowa usunięcia elementu z pokrycia Γ'' wynosi $\tau^{\text{del}} = O(1)$, dzięki dodatkowej tablicy wskaźników wskazujących elementy na listach. Tablica ta ma takie samo znaczenie jak w algorytmach wyznaczania podciągu LICS dyskutowanych w rozdziale 5. Złożoność czasową budowy małych pokryć można oszacować analogicznie, jak to było dla problemu LICS, tj. $\tau^{\text{build}}(n_j) = O(n_j \log \ell)$, gdzie n_j oznacza liczbę elementów pomiędzy kolejnymi punktami stopu. Złożoność czasowa znajdowania punktów podziału list pokryć wynosi $\tau^{\text{find}}(n_j) = O(\min(n_j, \ell)\ell + u)$. Z oczywistych względów $\sum_{j=1}^{n_i^{\text{stop}}} n_j \le u$. Ponadto, $\tau^{\text{split}} = \tau^{\text{join}} = O(1)$ oraz $\tau^{\text{init}} = O(u \log \ell)$. Podstawiając te wartości do (6.1), otrzymuje się:

$$\tau_i = uO(1) + \sum_{j=1}^{n_i^{\text{stop}}} \Big(O(n_j \log \ell) + O(\min(n_j, \ell)\ell + u) + O(\min(n_j, \ell)\ell)(O(1) + O(1)) \Big) =$$
$$= O(u) + O(u \log \ell) + O(\min(u\ell, \ell^3)) + O(u\ell) + O(\min(u\ell, \ell^3)) = O(u\ell). \tag{6.3}$$

Na podstawie (6.2) wiadomo natomiast, że:

Wniosek 6.2. *Złożoność czasowa algorytmu* LISW *dla listowej reprezentacji pokrycia jest:*

$$\sum_{i=0}^{\lceil n/u \rceil - 1} (O(u \log \ell) + O(u\ell)) = O(n\ell). \tag{6.4}$$

6.2.3. Reprezentacja pokrycia za pomocą drzew zrównoważonych

W wersji z reprezentacją pokrycia za pomocą drzew zrównoważonych złożoności czasowe składowych części algorytmu można wyznaczyć analogicznie jak dla problemu LICS. Zatem:

$$\tau^{\text{del}} = \tau^{\text{split}} = \tau^{\text{join}} = O(\log u), \qquad \tau^{\text{build}}(n_j) = O(n_j \log \ell),$$
$$\tau^{\text{find}}(n_j) = O(\min(n_j, \ell)\ell \log u), \qquad \tau^{\text{init}} = O(u \log \ell).$$

Po podstawieniu tych wartości do (6.1) oraz (6.2) otrzymuje się:

$$\tau_i = uO(\log u) +$$
$$\sum_{j=1}^{n_i^{\text{stop}}} \Big(O(n_j \log \ell) + O(\min(n_j, \ell)\ell \log u) + O(\min(n_j, \ell)\ell)(O(\log u) + O(\log u)) \Big) =$$
$$= O(u \log u) + O(u \log \ell) + O\left(\min\left(u\ell, \ell^3\right) \log u\right) + O\left(\min\left(u\ell, \ell^3\right) \log u\right) =$$
$$= O\left(u \log u + \min\left(u\ell, \ell^3\right) \log u\right) =$$
$$= O\left(\min\left(u\ell, u + \ell^3\right) \log u\right). \tag{6.5}$$

Na podstawie (6.2) można sformułować wniosek:

Wniosek 6.3. *Złożoność czasowa algorytmu* LISW *dla reprezentacji pokrycia za pomocą drzew zrównoważonych jest:*

$$\sum_{i=0}^{\lceil n/u \rceil - 1} \left(O(u \log \ell) + O\left(\min\left(u\ell, u + \ell^3\right) \log u\right)\right) = O\left(\min\left(n\ell, n\left\lceil \frac{\ell^3}{u} \right\rceil\right) \log u\right). \tag{6.6}$$

6.2.4. Reprezentacja pokrycia za pomocą list drzew zrównoważonych

W reprezentacji pokrycia za pomocą list drzew zrównoważonych rozmiaru $\Theta(e)$, dla pewnego e, które zostanie ustalone później, złożoności czasowe poszczególnych składowych algorytmu wynoszą:

$$\tau^{\text{del}} = \tau^{\text{split}} = \tau^{\text{join}} = O(\log e), \qquad \tau^{\text{build}}(n_j) = O(n_j \log \log n),$$

$$\tau^{\text{find}}(n_j) = O\left(\min\left(n_j,\ell\right)\ell\log e + \frac{u}{e} + \ell\right), \qquad \tau^{\text{init}} = O(u\log\log n).$$

Po podstawieniu tych wartości do (6.1) oraz (6.2) otrzymuje się:

$$\begin{aligned}
\tau_i = uO(\log e) + \sum_{j=1}^{n_i^{\text{stop}}} \big(O(n_j\log\log n) +& \\
O\left(\min\left(n_j,\ell\right)\ell\log e + \frac{u}{e} + \ell\right) + O(\min(n_j,\ell)\ell)(O(\log e) + O(\log e)\Big) &= \\
= O(u\log e) + O(u\log\log n) + O\left(\min\left(u\ell,\ell^3\right)\log e + \frac{u\ell}{e} + \ell^2\right) +& \\
O(\min(u\ell,\ell^3)\log e) &= \\
= O\left(u\log\log n + \min(u\ell, u+\ell^3)\log e + \frac{u\ell}{e}\right)& \qquad (6.7)
\end{aligned}$$

oraz złożoność czasową całego algorytmu:

$$\begin{aligned}
\sum_{i=0}^{\lceil n/u\rceil-1}\left(O(u\log\log n) + O\left(u\log\log n + \min(u\ell, u+\ell^3)\log e + \frac{u\ell}{e}\right)\right) &= \\
O\left(n\log\log n + \min\left(n\ell, n\left\lceil\frac{\ell^3}{u}\right\rceil\right)\log e + \frac{n\ell}{e}\right).& \qquad (6.8)
\end{aligned}$$

Łatwo można stwierdzić, że powyższa wartość jest minimalna dla $e = \Theta(\lceil u/\ell^2\rceil)$. Otrzymuje się wówczas złożoność czasową

$$O\left(n\log\log n + \min\left(n\ell, n\left\lceil\frac{\ell^3}{u}\right\rceil\right)\log\left\lceil\frac{u}{\ell^2}\right\rceil\right). \qquad (6.9)$$

Wartość e można oszacować z dokładnością do czynnika 4, opierając się na lemacie 6.1. W tym celu na etapie przetwarzania wstępnego należy wyznaczyć największą z długości LIS dla każdego okna A_{iu+1}^{iu+u}, gdzie $0 \leq i \leq \lceil n/u\rceil - 1$. Etap ten nie ma wpływu na całkowitą złożoność czasową algorytmu LISW.

Rozważania dotyczące normalizacji rozmiarów drzew przeprowadzone w podrozdz. 5.2.4 są aktualne także w tym miejscu i złożoność czasowa tych operacji nie ma wpływu na sumaryczną złożoność całego algorytmu, w związku z czym można sformułować wniosek:

Wniosek 6.4. *Złożoność czasowa algorytmu* LISW *dla reprezentacji pokrycia za pomocą list drzew zrównoważonych jest*

$$O\left(n\log\log n + \min\left(n\ell, n\left\lceil\frac{\ell^3}{u}\right\rceil\right)\log\left\lceil\frac{u}{\ell^2}\right\rceil\right). \qquad (6.10)$$

6.3. Podsumowanie

W niniejszym rozdziale przedstawiony został problem najdłuższego podciągu rosnącego w oknie ustalonej szerokości (LISW). Rozważania prowadzone były dla wariantu problemu, w którym szukana jest długość podciągu LISW, ale dzięki temu, że razem z nią wyznaczana była też pozycja tego najdłuższego podciągu, bardzo łatwo można na tej podstawie wyznaczyć podciąg LISW przy takiej samej złożoności czasowej.

Złożoności czasowe algorytmów znanych z literatury dla problemu LISW są $O(n \log\log n + n\ell)$ [4], $O(n\ell)$ [40] oraz $O(n \log^2 n)$ [200]. Złożoność czasowa wariantu z reprezentacją listową, $O(n\ell)$ jest taka sama jak drugiego z wymienionych algorytmów, a w przypadku wariantu z wykorzystaniem list drzew zrównoważonych proponowany algorytm jest od niej szybszy, o ile

$$\ell = o\left(\frac{u^{1/2}}{\log^{1/2} u}\right),$$

a nie gorszy, jeśli

$$\ell = O\left(u^{1/2}\right).$$

W porównaniu do algorytmu Tiskina [200] proponowany algorytm z reprezentacją pokrycia za pomocą list drzew zrównoważonych o złożoności czasowej $O(n \log\log n + \min(n\ell, n\lceil \ell^3/u \rceil) \times \log\lceil u/\ell^2 \rceil)$ jest lepszy dla

$$\ell = o\left(u^{1/3} \frac{\log^{2/3} n}{\log^{1/3} u}\right). \tag{6.11}$$

Kwestią otwartą pozostaje odpowiedź na pytanie, jaki jest kres dolny złożoności czasowej dla problemu LISW. Do tej pory wiadomo jedynie, że nie jest on niższy niż kres dolny złożoności czasowej dla problemu LIS. W przeciwnym bowiem przypadku byłoby możliwe zastosowanie algorytmu osiągającego ten kres dla problemu LISW do problemu LIS po przyjęciu założenia, że $u = n$.

Część II

WSPÓLNE PODCIĄGI

7. Najdłuższy wspólny podciąg

7.1. Wprowadzenie

W wielu sytuacjach zachodzi konieczność porównania ciągów symboli, którymi mogą być liczby całkowite, symbole oznaczające zasady (w łańcuchach DNA), aminokwasy (w łańcuchach białkowych), litery (w tekstach) itp. W zależności od okoliczności stosowane są różne miary podobieństwa ciągów, które zostały tak zdefiniowane, aby możliwie najlepiej nadawały się do konkretnych sytuacji. Znając podobieństwo dwu lub większej liczby ciągów, można m.in.:

- tworzyć drzewo filogenetyczne (np. [77, 59, 106, 159]) – ciągami są łańcuchy DNA lub białkowe,
- wykonywać kompresję danych (np. [30, 41, 185]) – ciągami są teksty,
- korygować błędy pisowni (np. [142, 134, 157]) – ciągami są słowa,
- wyszukiwać plagiaty, podobne dokumenty (np. [118]) – ciągami są teksty, teksty źródłowe programów, transkrypcje muzyczne,
- wyszukiwać w bazach danych informacje podane w przybliżeniu – ciągami są np. teksty nieco zniekształcone.

Istnieje wiele miar podobieństwa ciągów. Jedną z najprostszych i często stosowanych jest *odległość Levenshteina* [142, 186]. Odległość ta jest liczbą podstawowych operacji edycyjnych, tj. *wstawienia* symbolu, *usunięcia* symbolu, *zamiany* symbolu na inny, które są konieczne do przekształcenia jednego ciągu w drugi. Jej uogólnieniem jest *odległość edycyjna*, w której z każdą operacją edycyjną związany jest pewien koszt. Koszt operacji zamiany symbolu na inny symbol zależy od tego, jakie symbole są zamieniane. W literaturze można znaleźć bardzo wiele prac dotyczących tej miary podobieństwa, wśród których warto wymienić m.in. przeglądowy artykuł Navarro [161], książki Crochemore'a i Ryttera [55], Crochemore'a i in. [50], Gusfielda [106].

Najbardziej typowym przykładem zastosowania odległości edycyjnej jest korekcja pisowni (ang. *spelling correction*). Zwykle aplikacja sprawdzająca poprawność pisowni porównuje każdy kolejny wyraz tekstu ze słownikiem i jeśli nie znajdzie w nim bieżącego słowa, sygnalizuje to użytkownikowi podając też kilka proponowanych poprawnych słów. Ponieważ typowe słowniki zawierają setki tysięcy lub nawet miliony słów, więc konieczne jest, aby lista kandydatów na poprawną wersję błędnie zapisanego słowa była znaleziona szybko, a podpowiedzi były dla użytkownika pomocne. Idealne by było, gdyby pierwsze proponowane słowo było tym, co użytkownik miał zamiar napisać [12]. Oczywiście, zadanie to nie jest łatwe i w wielu badaniach z tej dziedziny zaproponowano różne metody. Zwykle ich idee opierają się na tym, aby proponować takie słowa, które z największym prawdopodobieństwem mogły zostać przekształcone do słowa

Odległość Levenshteina

Słowa:	SŁOWO SOWA
Operacje:	usuń(Ł), zamień(O–A)
Odległość:	2

Odległość Damerau

Słowa:	SŁOWO SOWA
Operacje:	usuń(Ł), zamień(O–A)
Odległość:	2

Odległość indel

Słowa:	SŁOWO SOWA
Operacje:	usuń(Ł), usuń(O), wstaw(A)
Odległość:	3

Odległość Hamminga

Słowa:	SŁOWO SOWA
Niezgodności:	Ł–O, O–W, W–A, O–
Odległość:	4

Najdłuższy wspólny podciąg

Słowa:	SŁOWO SOWA
LCS:	SOW
Długość:	3

Uliniowienie ciągów

Słowa:	SŁOWO SOWA
Słowo 1:	SŁOWO
Słowo 2:	S-OWA

Rys. 7.1. Przykład różnych miar odległości podobieństwa ciągów
Fig. 7.1. Example of various distance measures for sequence similarity

błędnego, np. z powodu wciśnięcia niepoprawnego klawisza na klawiaturze. Zastosowanie odległości edycyjnej w tym przypadku jest o tyle uzasadnione, że z dużym prawdopodobieństwem można zakładać, że im słowo poprawne ze słownika jest bardziej odległe od słowa błędnego występującego w tekście, tym mniejsza szansa, że to właśnie słowo użytkownik miał na myśli.

Odległość edycyjna, mimo iż koncepcyjnie prosta i elegancka, ma jednak swoje ograniczenia. Przykładowo, dla języków naturalnych czasami ze sporym prawdopodobieństwem można oczekiwać pomyłki na większej liczbie liter, np. w języku polskim błąd Ż–RZ będzie policzony jako 2 niezależne błędy według odległości edycyjnej, podczas gdy jest to zwykle pojedynczy błąd ortograficzny. Przegląd wielu metod sprawdzania pisowni (ang. *spellchecking*) i korygowania błędów pisowni (ang. *spelling correction*) można znaleźć m.in. w [134, 157, 46, 72, 158]. Przykłady dyskutowanych w tym podrozdziale miar odległości można zobaczyć na rys. 7.1.

Odmianą odległości Levenshteina jest *odległość Damerau* [56], w której oprócz trzech operacji edycyjnych wymienionych wcześniej dopuszcza się zamianę kolejności sąsiednich liter. Rozszerzenie to wynika z faktu, że często takie właśnie pomyłki popełniane są przez piszących na klawiaturze komputerowej. Inną modyfikacją odległości edycyjnej jest dopuszczenie tylko dwu operacji edycyjnych: *wstawienia* i *usunięcia* symbolu. Od angielskich nazw tych operacji miara ta nazywana jest *odległością indel* [212]. Kolejną miarą jest *odległość Hamminga* [108], która jest liczbą par niezgodnych symboli na kolejnych pozycjach dwu ciągów.

Wymienione miary odległości mierzą liczbę lub koszt „błędów”, czyli im wartość jest mniejsza, tym bardziej podobne do siebie są dwa ciągi. Przeciwieństwem tego podejścia są dwie kolejne miary. Długość *najdłuższego wspólnego podciągu* (ang. *longest common subsequence*, LCS) jest długością najdłuższego ciągu, który jest podciągiem obu ciągów. Oczekuje się, że im większa jest ta wartość (przy ustalonej długości porównywanych ciągów), tym bardziej podobne do siebie są te ciągi. Dla ciągów A i B zachodzi ponadto:

$$\mathrm{LLCS}(A,B) = |A| + |B| - d_{\mathrm{id}}(A,B),$$

gdzie d_{id} oznacza odległość indel, a LLCS – długość podciągu LCS. Zastosowania tej miary są bardzo liczne i obejmują m.in. wyszukiwanie podobieństw w plikach tekstowych, np. tekstach źródłowych programów komputerowych na potrzeby systemów kontroli wersji [118], porównywanie łańcuchów DNA [48, 170], strukturalne uliniowienie łańcuchów RNA [31], klasyfikację użytkowników odwiedzających strony [28]. Dobre omówienie tej miary można znaleźć w [16, 17].

Jeszcze inną miarą jest tzw. *uliniowienie ciągów* (ang. *sequence alignment*) stosowane przede wszystkim w bioinformatyce, gdzie porównywane ciągi reprezentują DNA, białka itp. W obrazowy sposób uliniowienie dwu ciągów można przedstawić jako podpisanie ciągów jeden pod drugim i takie wstawienie przerw (ang. *gaps*), długości jednego lub więcej symbolu do każdego z ciągów, aby sumaryczny wynik uliniowienia był maksymalny. Wynik ten wyznacza się jako sumę wyników dla par odpowiadających sobie symboli oraz kosztów przerw [164]. U podłoża wprowadzenia takiej miary leżały obserwacje biologów, że podmiana jednego aminokwasu na drugi w białku może skutkować bardzo drobną modyfikacją funkcjonowania tego białka, podczas gdy podmiana na inny aminokwas może mieć bardzo duże znaczenie. W pierwszym przypadku, mimo iż na jednej pozycji dwa białka się różnią, to są one do siebie bardzo podobne, podczas gdy w drugim przypadku tak już nie jest. Idea wstawiania przerw wzięła się z obserwacji, że w trakcie ewolucji w DNA pojawiały się wtrącenia różnego materiału genetycznego. Więcej informacji o uliniowieniu ciągów i zastosowaniach w bioinformatyce można znaleźć m.in. w książkach Mounta [159], Jonesa i Pevznera [124], Polańskiego i Kimmela [176], artykule przeglądowym Haque'a i in. [109].

7.2. Dodatkowe definicje

W rozdziałach bieżącej części niniejszej pracy będzie przyjmowane, że ciągami wejściowymi są $A = a_1a_2\ldots a_n$ oraz $B = b_1b_2\ldots b_m$. Elementy (symbole) tych ciągów należą do alfabetu $\Sigma \subset \mathbb{Z}$. Czasami dogodne jest przyjęcie dodatkowego założenia, że alfabet jest ograniczony, tj. $\Sigma = \{0,\ldots,\sigma-1\}$. W tej sytuacji σ będzie nazywane *rozmiarem* alfabetu. *Dopasowaniem*

dla pary (i, j) nazywana jest sytuacja, w której $a_i = b_j$. *Rangą* pary (i, j) dla ciągów A i B nazywana jest długość podciągu LCS dla A_i, B_j.[1] Dopasowanie (i, j) rangi h jest nazywane *dominującym*, jeśli nie istnieje inne dopasowanie (i', j') o takiej samej randze h, dla którego zachodzi $i' \leq i$ oraz $j' \leq j$. Długość wyniku w każdym problemie rozpatrywanym w rozdziałach 7–10 jest oznaczana przez ℓ. Ponieważ problemy dyskutowane w tej części są symetryczne ze względu na A i B, więc bez utraty ogólności będzie zakładane, że $m \leq n$.

Niniejszy rozdział dotyczy problemu, który można zdefiniować następująco:

Problem 7.1 (Najdłuższy wspólny podciąg, LCS)**.** *Dla ciągów $A = a_1a_2 \dots a_n$ i $B = b_1b_2 \dots b_m$ znaleźć najdłuższy ciąg $S = s_1s_2 \dots s_\ell$ będący jednocześnie podciągiem A i B.*

Przykład 7.1 (Najdłuższy wspólny podciąg, LCS)**.** *Dla ciągów $A =$ A<u>BAAD</u>A<u>CB</u>A<u>A</u>BC oraz $B =$ C<u>B</u>CBD<u>AADC</u>D<u>BA</u> najdłuższym wspólnym podciągiem jest $S =$ BAADCBA. W ciągach A oraz B podkreślono symbole tworzące podciąg LCS.*

7.3. Algorytmy sekwencyjne i równoległości bitowej

7.3.1. Programowanie dynamiczne

Klasyczną metodą rozwiązywania problemu LCS jest użycie programowania dynamicznego, w którym prostokątna macierz rozmiarów $(n+1) \times (m+1)$ jest wypełniana zgodnie z następującą zależnością:

$$M(i,j) = \begin{cases} 0, & \text{jeśli } i = 0 \vee j = 0, \\ \max(M(i-1,j), M(i,j-1)), & \text{jeśli } 0 < i \leq n, 0 < j \leq m, a_i \neq b_j, \\ M(i-1,j-1)+1, & \text{jeśli } 0 < i \leq n, 0 < j \leq m, a_i = b_j. \end{cases} \tag{7.1}$$

Wartość $M(i, j)$ jest długością podciągu LCS dla ciągów A_i oraz B_j, a więc w szczególności $M(n,m)$ jest długością podciągu LCS dla A i B. W celu uzyskania podciągu wynikowego (a nie tylko jego długości) należy przejść otrzymaną macierz począwszy od komórki $M(n,m)$ do $M(1,1)$ przesuwając się każdorazowo do komórki, której wartość była użyta do obliczenia komórki bieżącej. Dla każdej komórki reprezentującej dopasowanie należy dołączyć do wyniku symbol bieżący z dowolnego z ciągów, przy czym podciąg wynikowy konstruowany jest od końca (rys. 7.2).

Zajętość pamięciowa tego algorytmu jest $\Theta(nm)$ słów, a jego złożoność czasowa jest $\Theta(nm)$. Wyniki te biorą się wprost z liczby obliczanych komórek macierzy oraz z konieczności przechowywania całej macierzy, aby później móc ją przejść. Jak wykazali Aho i in. [2], w modelu

[1] W kolejnych rozdziałach definicje dopasowania i rangi będą nieco inne, co ma związek ze specyfiką rozważanych tam problemów.

j \ i		0	1	2	3	4	5	6	7	8	9	10	11	12
			A	B	A	A	D	A	C	B	A	A	B	C
0		0	0	0	0	0	0	0	0	0	0	0	0	0
1	C	0	0	0	0	0	0	0	1	1	1	1	1	1
2	B	0	0	1	1	1	1	1	1	2	2	2	2	2
3	C	0	0	1	1	1	1	1	2	2	2	2	2	3
4	B	0	0	1	1	1	1	1	2	3	3	3	3	3
5	D	0	0	1	1	1	2	2	2	3	3	3	3	3
6	A	0	1	1	2	2	2	3	3	3	4	4	4	4
7	A	0	1	1	2	3	3	3	3	3	4	5	5	5
8	D	0	1	1	2	3	4	4	4	4	4	5	5	5
9	C	0	1	1	2	3	4	4	5	5	5	5	5	6
10	D	0	1	1	2	3	4	4	5	5	5	5	5	6
11	B	0	1	2	2	3	4	4	5	6	6	6	6	6
12	A	0	1	2	3	3	4	5	5	6	7	7	7	7

j \ i		0	1	2	3	4	5	6	7	8	9	10	11	12
			A	B	A	A	D	A	C	B	A	A	B	C
0		0	0	0	0	0	0	0	0	0	0	0	0	0
1	C	0	0	0	0	0	0	0	1	1	1	1	1	1
2	B	0	0	1	1	1	1	1	1	2	2	2	2	2
3	C	0	0	1	1	1	1	1	2	2	2	2	2	3
4	B	0	0	1	1	1	1	1	2	3	3	3	3	3
5	D	0	0	1	1	1	2	2	2	3	3	3	3	3
6	A	0	1	1	2	2	2	3	3	3	4	4	4	4
7	A	0	1	1	2	3	3	3	3	3	4	5	5	5
8	D	0	1	1	2	3	4	4	4	4	4	5	5	5
9	C	0	1	1	2	3	4	4	5	5	5	5	5	6
10	D	0	1	1	2	3	4	4	5	5	5	5	5	6
11	B	0	1	2	2	3	4	4	5	6	6	6	6	6
12	A	0	1	2	3	3	4	5	5	6	7	7	7	7

Rys. 7.2. Przykład działania klasycznego algorytmu wyznaczania podciągu LCS opartego na programowaniu dynamicznym dla ciagów: ABAADACBAABC, CBCBDAADCDBA. Strona lewa: macierz programowania dynamicznego z zaznaczonymi dopasowaniami. Strona prawa: wyznaczanie podciągu LCS; ciemnoszare komórki reprezentują dopasowania, na podstawie których tworzony jest wynik BAADCBA

Fig. 7.2. An example of the classical dynamic programming algorithm computing an LCS for the sequences: ABAADACBAABC, CBCBDAADCDBA. Left subfigure: the DP matrix with marked matches. Right subfigure: the traceback; the dark gray cells represents the matches taken to the result: BAADCBA

obliczeniowym opartym na porównaniach elementów jest to kres dolny złożoności czasowej dla tego zadania, a więc algorytm ten jest optymalny w sensie asymptotycznym. Podobny wynik osiągnęli Wong i Chandra [219].

W modelu obliczeniowym Word RAM, zakładanym jako podstawowy model obliczeniowy w niniejszej pracy, najlepszym znanym oszacowaniem kresu dolnego złożoności czasowej jest $\Omega(n \log n)$ dla alfabetu nieograniczonego i $\Omega(n)$ dla alfabetu ograniczonego [114]. W dalszej części niniejszego rozdziału zostaną omówione m.in. trzy bardzo istotne sposoby rozwiązywania problemu LCS, dzięki którym możliwe jest: znaczne przyspieszenie obliczeń w przypadku niewielkiej liczby dopasowań (podrozdz. 7.3.2), zredukowanie złożoności pamięciowej (podrozdz. 7.3.3), prowadzenie równoległych obliczeń na pojedynczych bitach w słowie komputerowym (podrozdz. 7.3.4). Dla wielu wariantów problemu LCS właśnie te sposoby pozwalają na uzyskanie najszybszych, bądź najoszczędniejszych pamięciowo algorytmów, stąd ich prezentacja będzie możliwie szczegółowa. Ponadto, wiele z algorytmów proponowanych w kolejnych rozdziałach opiera się na ideach podobnych do stosowanych w powyżej wymienionych algorytmach.

7.3.2. Algorytm Hunta–Szymanskiego

Algorytm Hunta i Szymanskiego [119] rozpoczyna swoje działanie od przetwarzania wstępnego, w którym tworzona jest tablica L rozmiaru σ zawierająca dla każdego symbolu alfabetu

LCS-HS(A, B)

Wejście: *A, B – ciągi, dla których wyznaczany jest podciąg LCS*
Wyjście: *znaleziony podciąg LCS*

```
   {Przetwarzanie wstępne}
 1 for i ← 1 to n do L[a_i].append(i)
   {Obliczenia właściwe}
 2 ℓ ← 0;  Q ← pusta struktura rozwiązująca problem poprzednika
 3 for j ← 1 to m do
 4     for each i z listy L[b_j] czytanej od końca do
 5         x ← Q.successor(i − 1)
 6         if x istnieje then Q.remove(x)
 7         else ℓ ← ℓ + 1
 8         Q.insert(⟨i, j⟩)
 9         M(i, j) ← Q.predecessor(i)
   {Wyznaczenie wyniku}
10 x ← Q.predecessor(n + 1)
11 for k ← ℓ downto 1 do
12     s_k ← a_{x.i};  x ← M(x)
13 return s_1 s_2 ... s_ℓ
```

Rys. 7.3. Algorytm Hunta–Szymanskiego rozwiązujący problem LCS
Fig. 7.3. Hunt-Szymanski algorithm for the LCS problem

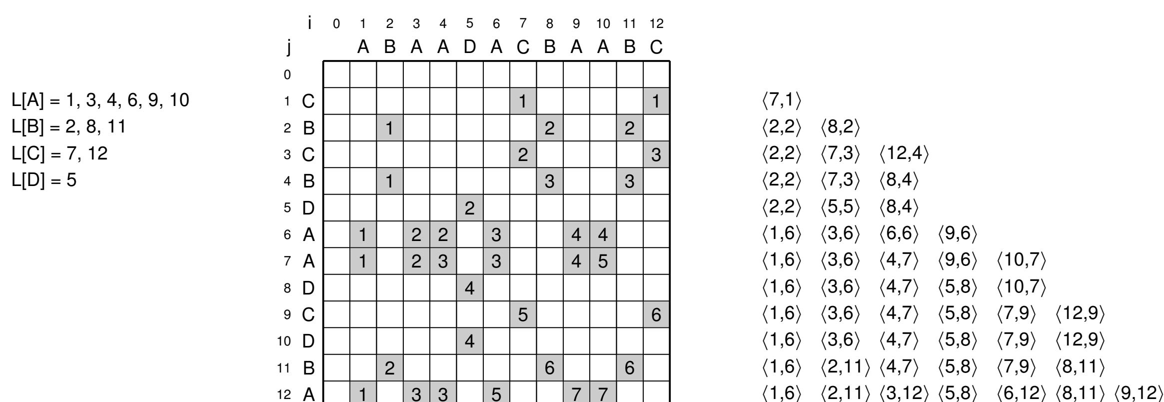
listę jego wystąpień w ciągu A (rys. 7.3).[2] Następnie rozpoczynają się obliczenia właściwe, w których główną strukturą danych jest struktura Q rozwiązująca problem poprzednika (ang. *predecessor problem*), udostępniająca operacje wstawiania, usuwania i wyszukiwania następnika oraz poprzednika. W strukturze tej przechowywane są pary wartości, przy czym porządek jest wyznaczany tylko według pierwszej składowej, a druga składowa pary zawiera dane dodatkowe. Po przetworzeniu j-tego wiersza macierzy, w stukturze Q znajdują się dla każdej rangi występującej we fragmencie macierzy $M(1..n, 1..j)$ pary (i', j') o najmniejszych indeksach kolumn i' dopasowań o tych rangach. Na rys. 7.4 przedstawiono ilustrację działania tego algorytmu. W szczególności pokazano zawartość struktury Q po wykonaniu każdego przebiegu pętli zewnętrznej oraz zawartość tablicy list L dla każdego symbolu alfabetu.

W algorytmie Hunta i Szymanskiego wykorzystuje się równoważną do (7.1) regułę wyznaczania macierzy M:

$$M(i, j) = \max_{1 \le i' < i,\, 1 \le j' < j,\, a_{i'} = b_{j'}} (M(i', j')) + 1. \tag{7.2}$$

Funkcja max z definicji zwraca 0, jeśli nie ma żadnych argumentów. Wartości tej macierzy wyznaczane są tylko dla dopasowań, a wynikiem końcowym jest największa z wartości znajdujących się w macierzy. W celu znalezienia podciągu LCS, a nie tylko jego długości, należy

[2]Opis algorytmu prezentowany w tym miejscu różni się nieco od opisu oryginalnego [119], ale istota jego działania oraz złożoności czasowa i pamięciowa są takie same.



Rys. 7.4. Przykład działania algorytmu Hunta–Szymanskiego wyznaczania podciągu LCS dla ciągów: ABAADACBAABC, CBCBDAADCDBA. Zaznaczone komórki oznaczają dopasowania. Strona lewa: listy wystąpień symboli w ciągu A. Środek: macierz DP z zaznaczonymi dopasowaniami (tylko dla porównania). Strona prawa: stan struktury Q po przetworzeniu każdego wiersza

Fig. 7.4. An example of the Hunt–Szymanski algorithm computing the LCS for the sequences: ABAADACBAABC, CBCBDAADCDBA. The marked cells denote matches. Left: lists of symbol appearances in A. Middle: the DP matrix with marked matches (just for comparison). Right: state of Q structure after computing each row

przechowywać w komórkach macierzy reprezentujących dopasowania wskaźnik do komórki, która posłużyła do wyznaczenia wartości danej komórki, a następnie przejść tę macierz zgodnie z tymi wskaźnikami odczytując podciąg wyjściowy, który utworzą elementy ciągów z kolumn (bądź wierszy) odwiedzanych dopasowań.

Na złożoność czasową algorytmu składa się czas przetwarzania wstępnego, $O(\sigma + n)$ oraz czas obliczeń właściwych, $O(m + d\tau_Q)$, gdzie d to liczba dopasowań w całej macierzy, a τ_Q to najdłuższy z czasów wykonywania operacji wstawiania, usuwania, szukania następnika i poprzednika w strukturze Q. Sumaryczny czas działania algorytmu jest $O(\sigma + n + m + d\tau_Q)$. Przyjmując, często uzasadnione, założenie, że $\sigma = O(n)$, otrzymuje się złożoność czasową $O(n + d\tau_Q)$, która zależy od wyboru struktury danych Q. Użycie zrównoważonych drzew poszukiwań binarnych (np. drzew czerwono-czarnych [29, 105, 190]), dla których $\tau_Q = O(\log \ell)$, daje algorytm o złożoności czasowej

$$O(n + d \log \ell). \tag{7.3}$$

Można też zauważyć, że element wstawiany do Q ma zawsze albo taką samą rangę jak element chwilę wcześniej usunięty, albo o 1 większą niż najwyższa z rang znajdujących się w Q. Dzięki temu jako struktury Q można użyć posortowanej tablicy i przeszukiwać ją w sposób binarny, co daje dokładnie taką samą złożoność czasową jak w (7.3). Kolejną możliwością jest użycie drzew van Emde Boasa [209, 210], dla których $\tau_Q = O(\log \log n)$, a złożoność czasowa algorytmu LCS-HS jest

$$O(n + d \log \log n). \tag{7.4}$$

W obu przypadkach czas dodatkowych operacji na strukturze M w celu uzyskania podciągu wynikowego jest $O(d)$, a więc nie wpływa on na złożoność czasową całego algorytmu.

Pozostaje jeszcze do rozważenia przypadek, w którym $\sigma = \omega(n)$, wtedy znaczący wpływ na złożoność czasową algorytmu zaczyna mieć faza przetwarzania wstępnego. Aby zachować złożoność pamięciową na akceptowalnym w praktyce poziomie, można w miejsce tablicy L zastosować np. zrównoważone drzewo poszukiwań binarnych indeksowane symbolem alfabetu, w węzłach którego będą znajdowały się listy indeksów dla każdego symbolu alfabetu, wtedy czas przetwarzania wstępnego jest $O(n \log n)$. Ponadto, podczas rozpoczynania przetwarzania każdego wiersza należy znaleźć w tym drzewie listę dopasowań dla symbolu opisującego ten wiersz. Sumarycznie wnosi to do złożoności czasowej składnik $O(m \log n)$. Wobec tego sumaryczna złożoność czasowa jest w zależności od użytej struktury danych do reprezentacji Q:

$$O(n \log n + d \log \ell) \qquad \text{lub} \qquad O(n \log n + d \log \log n). \tag{7.5}$$

Złożoność pamięciowa tego algorytmu zależy od wyboru reprezentacji macierzy M. Najprostszym sposobem jest użycie listy przeszukiwanej od końca w wierszu 12. algorytmu LCS-HS. Zajmie ona $\Theta(d)$ słów. Do tego należy dodać jeszcze zajętość pamięci pozostałych struktur danych, czyli $\Theta(n)$ słów. Sumarycznie złożoność pamięciowa wynosi zatem $\Theta(n+d)$ słów.

Algorytm Hunta–Szymanskiego sprawdza się najlepiej w sytuacjach, w których liczba dopasowań, d, jest istotnie mniejsza niż nm. Sytuacje takie mają miejsce często. W przypadku pesymistycznym jednak $d = \Theta(nm)$, co powoduje, że złożoność czasowa tego algorytmu może być gorsza niż dla algorytmu programowania dynamicznego.

Przetwarzanie tylko komórek reprezentujących dopasowania i wyznaczanie dla nich rang jest popularnym sposobem, na którym opiera się wiele bardzo wydajnych algorytmów wyznaczania podciągu LCS, np. algorytm Kuo–Crossa [135], który okazał się najszybszy w testach porównawczych przeprowadzonych przez Bergrotha i in. [32]. Idea ta jest stosowana także w algorytmach dla problemów pokrewnych problemowi LCS. Często podejście takie jest nazywane *metodą* Hunta–Szymanskiego.

7.3.3. Algorytm Hirschberga

W poprzednim podrozdziale dyskutowana była kwestia osiągnięcia lepszej złożoności czasowej dla problemu LCS. Niestety, złożoność pamięciowa wynosiła przynajmniej $\Theta(n+d)$ słów, co w pesymistycznym przypadku jest $\Theta(nm)$. Właśnie złożoność pamięciowa może w praktyce dyskwalifikować powyżej opisane algorytmy dla długich ciągów. Rozwiązać ten problem można na różne sposoby. Najprościej jest w sytuacji, w której żądanym wynikiem jest tylko długość podciągu LCS, a nie sam ciąg. Wtedy wystarczy w algorytmie programowania dynamicznego przetwarzać macierz wierszami, dzięki czemu konieczne jest przechowywanie jedynie

2 wierszy macierzy M i złożoność pamięciowa jest $\Theta(n)$ słów, przy niezmienionej złożoności czasowej. Algorytm Hunta–Szymanskiego można zmodyfikować w podobny sposób. W tym celu należy zrezygnować ze struktury M i w miejsce wierszy 10–13 algorytmu LCS-HS wstawić instrukcję zwracania wartości zmiennej ℓ. Złożoność pamięciowa spada wtedy do $O(n)$ słów przy zachowanych złożonościach czasowych.

Powyższe podejścia są jednak nieskuteczne, jeśli wynikiem ma być także sam podciąg LCS. Historycznie pierwszy algorytm pozwalający na uzyskanie podciągu LCS w czasie $O(nm)$ przy złożoności pamięciowej $\Theta(n)$ słów zaproponował Hirschberg [112]. Algorytm ten został skonstruowany zgodnie z paradygmatem „dziel-i-zwyciężaj” (ang. *divide and conquer*). Macierz programowania dynamicznego dzielona jest na dwie części: $M(1..\lfloor n/2\rfloor, 1..m)$ oraz $M(\lfloor n/2\rfloor + 1..n, 1..m)$, z których pierwsza jest obliczana zgodnie z równaniem (7.1), a druga począwszy od $M(n,m)$ na podstawie zależności:

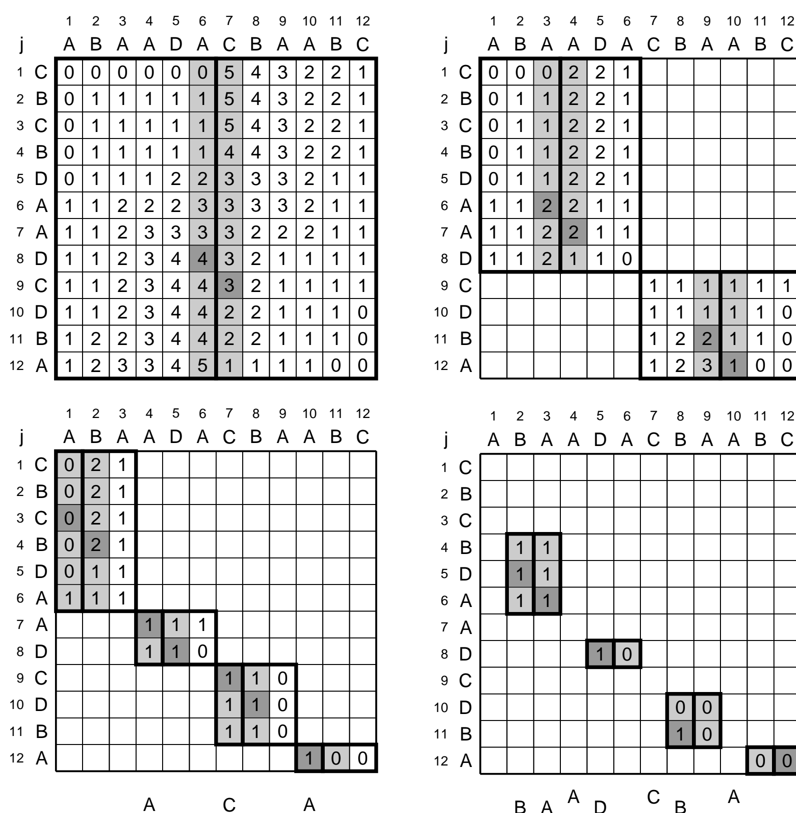
$$M(i,j) = \begin{cases} 0, & \text{jeśli } i = n+1 \vee j = m+1, \\ \max(M(i+1,j), M(i,j+1)), & \text{jeśli } 0 < i \leq n, 0 < j \leq m, a_i \neq b_j, \\ M(i+1,j+1)+1, & \text{jeśli } 0 < i \leq n, 0 < j \leq m, a_i = b_j. \end{cases} \tag{7.6}$$

Każdą z części oblicza się przechowując po 2 kolumny macierzy, dzięki czemu złożoność pamięciowa jest $\Theta(n)$ słów. Otrzymuje się sąsiednie kolumny $M(\lfloor n/2\rfloor, 1..m)$ oraz $M(\lfloor n/2\rfloor + 1, 1..m)$, pierwszą na podstawie (7.1), a drugą na podstawie (7.6). Komórki w tych kolumnach przechowują informację o długości podciągu LCS dla pewnych prefiksów lub sufiksów obu ciągów. Kluczowe jest teraz znalezienie takiej pary punktów $(\lfloor n/2\rfloor, j)$, $(\lfloor n/2\rfloor + 1, j+1)$, dla których konkatenacja podciągów LCS dla prefiksów i sufiksów ciągów da ciąg maksymalnej długości, a więc wyznacza się

$$\operatorname*{argmax}_j \left(M\left(\left\lfloor \frac{n}{2} \right\rfloor, j\right) + M\left(\left\lfloor \frac{n}{2} \right\rfloor + 1, j+1\right)\right). \tag{7.7}$$

Następnie należy wykonać rekurencyjnie tę samą procedurę dla prefiksów $A_{1,\lfloor n/2\rfloor}$ i $B_{1,j}$ oraz sufiksów $A_{\lfloor n/2\rfloor+1,n}$ i $B_{j+1,m}$ otrzymując fragmenty wyniku S' oraz S'' Wynikiem końcowym jest konkatenacja podciągów LCS znalezionych w wywołaniach rekurencyjnych: $S'S''$. W oczywisty sposób, jeśli długość fragmentu ciągu A wynosi 1, to otrzymuje się wtedy zero lub jeden symbol w ciągu będącym wynikiem wywołania rekurencyjnego algorytmu. Dzięki temu, że na każdym kolejnym poziomie rekurencji wyznacza się jedynie około połowę komórek macierzy obliczonych na poziomie wyższym, sumaryczna złożoność czasowa jest $\Theta(nm)$, a złożoność pamięciowa $\Theta(n)$. Przykład działania algorytmu Hirschberga przedstawiony jest na rys. 7.5.

Opisany wyżej sposób zredukowania złożoności pamięciowej przy jednoczesnym utrzymaniu złożoności czasowej jest często stosowany w algorytmach rozwiązujących problemy pokrewne do problemu LCS. Nazywa się go wtedy *metodą* Hirschberga.



Rys. 7.5. Przykład działania algorytmu Hirschberga o liniowej złożoności pamięciowej wyznaczającego podciąg LCS dla ciągów: ABAADACBAABC, CBCBDAADCDBA. Na kolejnych rysunkach przedstawiono poziomy rekurencji. Kolumny zaznaczone na szaro pokazują przebieg podziału każdej (pod)macierzy. Pod macierzami wskazano, które symbole są wyznaczane na danym poziomie rekurencji. Wynik ostateczny to BAADCBA

Fig. 7.5. An example of the Hirschberg linear-space algorithm computing the LCS for the sequences: ABAADACBAABC, CBCBDAADCDBA. Figures show recurrence levels. Greyed columns show (sub)matrix divisions. Below two bottom matrices the computed symbols are given. They form the result: BAADCBA

7.3.4. Algorytm równoległości bitowej

Jednym z wydajniejszych podejść w projektowaniu szybkich algorytmów w dziedzinie przetwarzania tekstów i ciągów jest użycie równoległości bitowej. Autorem tego podejścia jest Dömölki [76], jednak później było one odkrywane wielokrotnie, a największy wpływ na współczesne prace miały artykuł Baeza-Yatesa i Gonneta [26] oraz wcześniejsza rozprawa doktorska Baeza-Yatesa [25]. Podstawowa koncepcja tego podejścia opiera się na dwóch prostych obserwacjach. Po pierwsze, do przechowywania wyniku niektórych obliczeń wystarczy jeden bit. Przykładem może tu być porównywanie liter w problemie wyszukiwania wystąpienia wzorca w tekście, gdzie wynikiem jest prawda lub fałsz. Po drugie, niektóre obliczenia mogą być wykonywane równolegle na wielu bitach. Ponieważ dzisiejsze komputery dysponują procesorami o długości słowa komputerowego wynoszącej 32 lub 64 bity, więc potencjalne przyspieszenie jest duże. Oczywiście, nie zawsze podejście to może być zastosowane, gdyż jeden bit w wie-

LCS-BP-LENGTH(A, B)

Wejście: *A, B – ciągi, dla których wyznaczana jest długość podciągu LCS*
Wyjście: *znaleziona długość podciągu LCS*

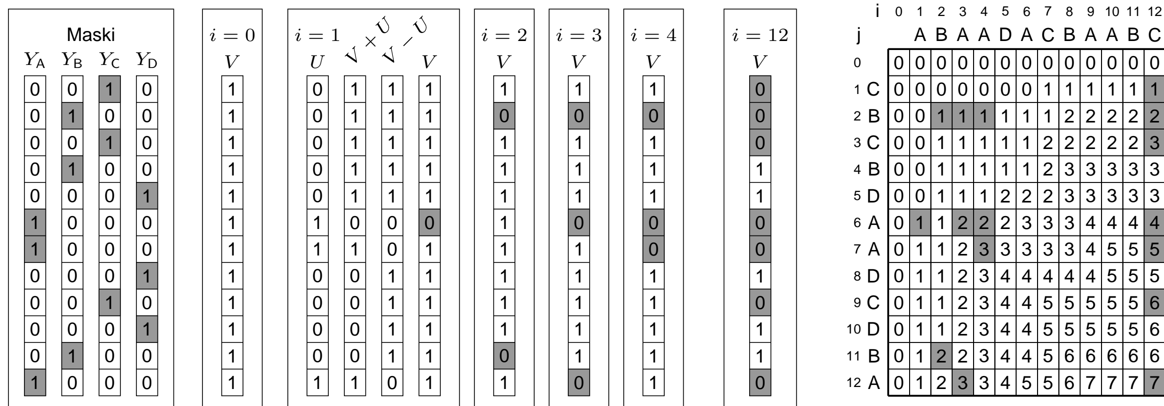
```
    {Przetwarzanie wstępne}
 1  for x ∈ Σ do Y_x ← 0^m
 2  for j ← 1 to m do Y_{b_j} ← Y_{b_j} | 0^{m−j}10^j
    {Obliczenia właściwe}
 3  V ← 1^m
 4  for i ← 1 to n do
 5      U ← V & Y_{a_i}
 6      V ← (V + U) | (V − U)
    {Wyznaczanie liczby bitów o wartości 0 w V – długości podciągu LCS}
 7  ℓ ← 0;  V ← ~V
 8  while V ≠ 0^m do
 9      ℓ ← ℓ + 1;  V ← V & (V − 1)
10  return ℓ
```

Rys. 7.6. Bitowo-równoległy algorytm Hyyrö wyznaczający długość podciągu LCS [120]
Fig. 7.6. Bit-parallel algorithm computing the LCS length by Hyyrö [120]

lu przypadkach nie wystarcza do reprezentacji wyniku obliczeń, a także często niemożliwe jest wykonywanie równoległych obliczeń na wielu bitach tego samego słowa komputerowego. W związku z tym zwykle opracowanie algorytmu równoległości bitowej dla konkretnego problemu nie należy do zadań łatwych. Jako zgrubną regułę można przyjąć, że najlepszymi kandydatami spośród klasycznych algorytmów do przekształcenia ich w algorytmy równoległości bitowej są algorytmy najprostsze. Przykładowo, algorytm równoległości bitowej Shift-Or [106] wyszukujący wystąpienia wzorca w tekście jest odmianą algorytmu naiwnego wyszukiwania wzorca.

Analizując macierz programowania dynamicznego dla problemu LCS (rys. 7.2), można zaobserwować, że różnice pomiędzy sąsiednimi komórkami wynoszą 0 lub 1. Obserwacja ta, której dowód można znaleźć m.in. w [120], jest bardzo pomocna w konstrukcji algorytmu opartego na równoległości bitowej (BP), ponieważ alternatywnym sposobem reprezentacji tej macierzy jest przechowywanie w poszczególnych komórkach tylko bitowej informacji, czy wartość w danej komórce jest równa wartości z komórki znajdującej się powyżej, czy jest od niej większa. Co więcej, można pokazać, że przy takiej reprezentacji wartości w poszczególnych wierszach można obliczać równolegle. Istnieją przynajmniej trzy algorytmy równoległości bitowej dla problemu LCS zaproponowane przez Allison i Dix [8], Crochemore'a i in. [52], Hyyrö [120]. Ostatni z tych algorytmów (rys. 7.6) jest zwykle w praktyce najszybszy, dlatego poniżej właśnie on zostanie przedstawiony. Pozostałe algorytmy są do niego podobne, a dyskusję różnic można znaleźć w [120].

Rysunek 7.7 ilustruje wyznaczanie długości podciągu LCS za pomocą tego algorytmu. Macierz programowania dynamicznego przedstawiona w prawej części rysunku została pokazana



Rys. 7.7. Przykład działania bitowo-równoległego algorytmu Hyyrö dla problemu LCS
Fig. 7.7. An example of the bit-parallel algorithm by Hyyrö for the LCS problem

tylko dla zilustrowania wykonywanych operacji i nie jest ona wprost obliczana w algorytmie LCS-BP-Length. Algorytm działa na wektorach bitowych. Dla prostoty opisu założone zostanie na razie, że każdy z wektorów bitowych mieści się w pojedynczym słowie komputerowym. Ramka Maski zawiera σ wektorów bitowych, po jednym na każdy symbol alfabetu. Wektory te obliczane są w etapie przetwarzania wstępnego (wiersze 1–2 algorytmu). Bit o indeksie j wektora związanego z symbolem $x \in \Sigma$ ma wartość 1 wtedy i tylko wtedy, gdy $b_j = x$. Głównym wektorem, na którym prowadzone są obliczenia, jest V. Odzwierciedla on różnice w wartościach poszczególnych komórek w kolumnie macierzy programowania dynamicznego, tj. j-ty bit V ma wartość 0 wtedy i tylko wtedy, gdy $M(i, j) - M(i, j-1) = 1$, dla aktualnej i-tej kolumny. Alternatywnie można stwierdzić, że indeks j bitu o wartości 0 w V po przetworzeniu i-tej kolumny jest najmniejszym indeksem wiersza takim, że (i', j) jest dopasowaniem pewnej rangi h i $i' \leq i$. Kolejne ramki rysunku ($i = 0$, $i = 1$, …, $i = 4$, $i = 12$) przedstawiają zawartość wektora V po przetworzeniu i-tej kolumny. Dla $i = 1$ przedstawionych jest nieco więcej szczegółów, które ilustrują obliczenia wykonywane w wierszach 5–6 algorytmu. Bity o wartościach 0 w wektorze V i komórki macierzy M, których wartość jest większa niż wartość górnego sąsiada, zostały wyczernione na rysunku, aby podkreślić związek pomiędzy kolumnami macierzy M i wektorem V. W wierszach 7–9 wyznaczana jest liczba bitów o wartości 0 w ostatnim wektorze V, która jest długością podciągu LCS.

W przypadku gdy $m > w$, konieczne jest reprezentowanie wektorów bitowych jako tablic słów komputerowych. Nie stanowi to jednak większego problemu, gdyż wszystkie potrzebne operacje bitowe można wykonywać na poszczególnych słowach niezależnie. Podobnie przy odejmowaniu wektorów bitowych nie może w tym algorytmie powstać przeniesienie. Jedynie przy dodawaniu powstaje konieczność obsługi przeniesienia, ale jest to proste do zrealizowania.

Główna pętla algorytmu wykonywana jest n razy a w każdym jej przebiegu ma miejsce tylko kilka operacji bitowych i arytmetycznych. Z uwagi na to, że wektory bitowe emulowane są

przez tablice słów komputerowych o rozmiarach $\lceil m/w \rceil$ słów, złożoność czasowa przedstawianego algorytmu jest $\Theta(n\lceil m/w \rceil)$. Złożoność pamięciowa tego algorytmu jest $\Theta(\lceil m/w \rceil)$ słów na wektor V oraz $\Theta(\sigma\lceil m/w \rceil)$ słów na wektory masek. Do tego należy jeszcze doliczyć rozmiar danych wejściowych, co sumarycznie daje $\Theta(n+\sigma\lceil m/w \rceil)$. W przypadku gdy nie można założyć, że $\sigma = O(n)$, możliwe jest (podobnie jak dla algorytmu Hunta–Szymanskiego) przechowywanie wektorów masek w drzewie poszukiwań binarnych. Daje to złożoność pamięciową $\Theta(n\lceil m/w \rceil)$ słów przy złożoności czasowej $\Theta(n\lceil m/w \rceil + n\log n)$.

W celu wyznaczenia podciągu LCS długości ℓ konieczne jest wykonanie przejścia po wygenerowanej macierzy bitowej algorytmem LCS-BP-Sequence (rys. 7.8).[3] W tym przypadku należy też tak zmodyfikować algorytm LCS-BP-Length, aby zapisywał zawartość wektora V po przetworzeniu każdej kolumny. Niech zatem $V(i)$ zawiera tę wartość dla i-tej kolumny. Wartości bitów znajdujących się w $V(i)$ będą służyły jako swego rodzaju „drogowskazy” wskazujące, w którym kierunku należy przesuwać się po macierzy bitowej. Jeśli j-ty bit wektora $V(i)$ zawiera 1, to oznacza to, że $M(i,j) = M(i,j-1)$, a więc możliwe jest przejście z komórki (i,j) do $(i,j-1)$, ponieważ rangi obu tych par są takie same. W przeciwnym przypadku przejście następuje do komórki $(i-1,j)$, ponieważ ranga tej pary jest nie mniejsza niż ranga pary $(i,j-1)$. Oczywiście, jeśli (i,j) jest dopasowaniem, to przejście odbywa się do $(i-1,j-1)$, a symbol a_i dołączany jest do ciągu wynikowego. Złożoność czasowa tego algorytmu konstrukcji wyniku na podstawie znanych wektorów bitowych V jest $\Theta(m+n) = \Theta(n)$, ponieważ w każdym przebiegu pętli zmniejszana jest wartość co najmniej jednego z pary indeksów i, j, których początkowe wartości wynoszą odpowiednio n, m. Złożoność pamięciowa tego algorytmu jest $\Theta(n\lceil m/w \rceil)$ słów niezależnie od tego czy $\sigma = O(n)$, czy też nie.

7.3.5. Algorytm równoległości bitowej – wariant Hirschberga

Algorytm LCS-BP-Sequence do wyznaczenia podciągu LCS wymaga $\Theta(n\lceil m/w \rceil)$ słów pamięci. Możliwe jest jednak zastosowanie do niego metody Hirschberga w celu zmniejszenia złożoności pamięciowej bez zmiany złożoności czasowej. Modyfikacja ta jest stosunkowo prosta [51]. Pewnej uwagi wymagają jednak dwie kwestie, tj. obsługa małych podmacierzy (takich, których wysokość jest nie większa niż w) oraz problem znalezienia wiersza dzielącego macierz na dwie mniejsze podmacierze, które to problemy niestety nie zostały przedyskutowane w [51].

Ogólna idea jest jednak taka sama jak w oryginalnym algorytmie Hirschberga. Macierz dzielona jest na dwie części i każda z nich jest obliczana algorytmem BP niezależnie, z tym że w czę-

[3] Prezentowany tu algorytm wyznaczania podciągu LCS odnosi się do omawianego algorytmu Hyyrö. W [121] pokazano algorytm wyznaczania podciągu LCS oparty na algorytmie równoległości bitowej Crochemore’a i in. [52].

LCS-BP-SEQUENCE(A, B, V, ℓ)

Wejście: *A, B – ciągi, dla których wyznaczany jest podciąg LCS*
V – tablica wektorów bitowych V wyznaczanych algorytmem LCS-BP-LENGTH
ℓ – długość podciągu LCS (wynik działania algorytmu LCS-BP-LENGTH*(A, B))*
Wyjście: *znaleziony podciąg LCS*

1 $\ell^* \leftarrow \ell;\quad i \leftarrow n;\quad j \leftarrow m$
2 **while** $\ell^* > 0$ **do**
3   **if** $a_i = b_j$ **then**
4         $s_{\ell^*} \leftarrow a_i;\quad \ell^* \leftarrow \ell^* - 1;\quad i \leftarrow i - 1;\quad j \leftarrow j - 1$
5   **else**
6   **if** j-ty bit $V(i)$ ma wartość 1 **then** $j \leftarrow j - 1$
7   **else** $i \leftarrow i - 1$
8 **return** $s_1 s_2 \ldots s_\ell$

Rys. 7.8. Algorytm równoległości bitowej wyznaczania podciągu LCS z macierzy wygenerowanej algorytmem LCS-BP-LENGTH

Fig. 7.8. Bit-parallel algorithm recovering an LCS from the matrix produced by LCS-BP-LENGTH algorithm

j \ i		0	1 A	2 B	3 A	4 A	5 D	6 A	7 C	8 B	9 A	10 A	11 B	12 C
0		0	0	0	0	0	0	0	0	0	0	0	0	0
1	C	0	0	0	0	0	0	0	1	1	1	1	1	1
2	B	0	0	1	1	1	1	1	1	2	2	2	2	2
3	C	0	0	1	1	1	1	1	2	2	2	2	2	3
4	B	0	0	1	1	1	1	1	2	3	3	3	3	3
5	D	0	0	1	1	1	2	2	2	3	3	3	3	3
6	A	0	1	1	2	2	2	3	3	3	4	4	4	4
7	A	0	1	1	2	3	3	3	3	3	4	5	5	5
8	D	0	1	1	2	3	4	4	4	4	4	5	5	5
9	C	0	1	1	2	3	4	4	5	5	5	5	5	6
10	D	0	1	1	2	3	4	4	5	5	5	5	5	6
11	B	0	1	2	2	3	4	4	5	6	6	6	6	6
12	A	0	1	2	3	3	4	5	5	6	7	7	7	7

j \ i		0	1 A	2 B	3 A	4 A	5 D	6 A	7 C	8 B	9 A	10 A	11 B	12 C
0														
1	C	1	1	1	1	1	1	1	0	0	0	0	0	0
2	B	1	1	0	0	0	0	0	1	0	0	0	0	0
3	C	1	1	1	1	1	1	1	0	1	1	1	1	0
4	B	1	1	1	1	1	1	1	1	0	0	0	0	1
5	D	1	1	1	1	1	0	0	1	1	1	1	1	1
6	A	1	0	1	0	0	1	0	0	1	0	0	0	0
7	A	1	1	1	1	0	0	1	1	1	1	0	0	0
8	D	1	1	1	1	1	0	0	0	0	1	1	1	1
9	C	1	1	1	1	1	1	1	0	0	0	1	1	0
10	D	1	1	1	1	1	1	1	1	1	1	1	1	1
11	B	1	1	0	1	1	1	1	1	0	0	0	0	1
12	A	1	1	1	0	1	1	0	1	1	0	0	0	0

Rys. 7.9. Przykład działania algorytmu równoległości bitowej wyznaczania podciągu LCS. Ciągiem wynikowym jest BAADCBA. Lewa część: macierz programowania dynamicznego (tylko dla porównania). Prawa część: przejście po macierzy bitowej. Dla prostoty prezentacji bity w słowach numerowane są od 1

Fig. 7.9. An example of the bit-parallel algorithm by Hyyrö for the LCS problem. The output sequence is BAADCBA. Left: DP matrix just for comparison of what is going on. Right: backtracking on bit matrix. Note that for simplicity of presentation the bits in Hyyrö's algorithm are numbered from 1

ści prawej ciągi są odwrócone (oczywiście wymaga to, aby istniały dwie tablice wektorów masek: dla ciągu B i odwróconego ciągu B). Następnie dla każdej części macierzy algorytm wykonywany jest rekurencyjnie, aż dojdzie do sytuacji, w której wysokość podmacierzy nie przekracza w. Dla takiej podmacierzy wykonuje się zwykły algorytm równoległości bitowej, w którym przechowuje się całą podmacierz reprezentowaną na wektorach bitowych. Z uwagi na niewielką wysokość tej podmacierzy, każdy wektor bitowy mieści się w pojedynczym słowie komputerowym, a całkowita zajętość pamięciowa tej bitowej podmacierzy to liczba słów równa jej szerokości. Małe podmacierze pojawiają się w końcowym etapie rekurencji i nie mogą obejmować

tych samych kolumn, a więc sumaryczna złożoność pamięciowa dla nich wszystkich jest $O(n)$ słów, a obliczane są w czasie $O(n)$. Ponieważ na każdym poziomie rekurencji szerokość macierzy zmniejsza się o połowę, więc maksymalna liczba poziomów rekurencji wynosi $\lceil \log_2 n \rceil$.

Pozostaje teraz do oszacowania liczba operacji na słowach komputerowych na kolejnych poziomach rekurencji. Na każdym poziomie macierz dzielona jest na podmacierze o rozłącznych zakresach wierszy i kolumn. Niech wysokości wszystkich podmacierzy, które są większe niż w, wynoszą $m_1, m_2, \ldots, m_k$ dla pewnego $k < \lceil m/w \rceil$, a szerokości tych podmacierzy na j-tym poziomie rekurencji wynoszą nie więcej niż $\lceil n/2^j \rceil$. Liczba operacji na słowach komputerowych na tym poziomie rekurencji dla takich podmacierzy wynosi wtedy:

$$\sum_{i=1}^{k} \left\lceil \frac{m_i}{w} \right\rceil \left\lceil \frac{n}{2^j} \right\rceil = \left\lceil \frac{n}{2^j} \right\rceil \sum_{i=1}^{k} \left\lceil \frac{m_i}{w} \right\rceil \leq \left\lceil \frac{n}{2^j} \right\rceil 2 \left\lceil \frac{m}{w} \right\rceil. \tag{7.8}$$

Sumując po wszystkich poziomach rekurencji, otrzymuje się

$$O\left(n \left\lceil \frac{m}{w} \right\rceil\right). \tag{7.9}$$

Wynik ten nie uwzględnia jednak problemu wyznaczenia indeksów wierszy dzielących macierze. W tym celu należy przeglądnąć wektory bitowe sąsiednich kolumn, zliczając liczbę bitów o wartości 0. Ponieważ na każdym z $\lceil \log_2 n \rceil$ poziomów rekurencji sumaryczna długość wszystkich wektorów bitowych znajdujących się na granicach podziału podmacierzy jest $O(m)$ bitów, więc sumarycznie czas potrzebny na te operacje w całym algorytmie jest

$$O(m \log n), \tag{7.10}$$

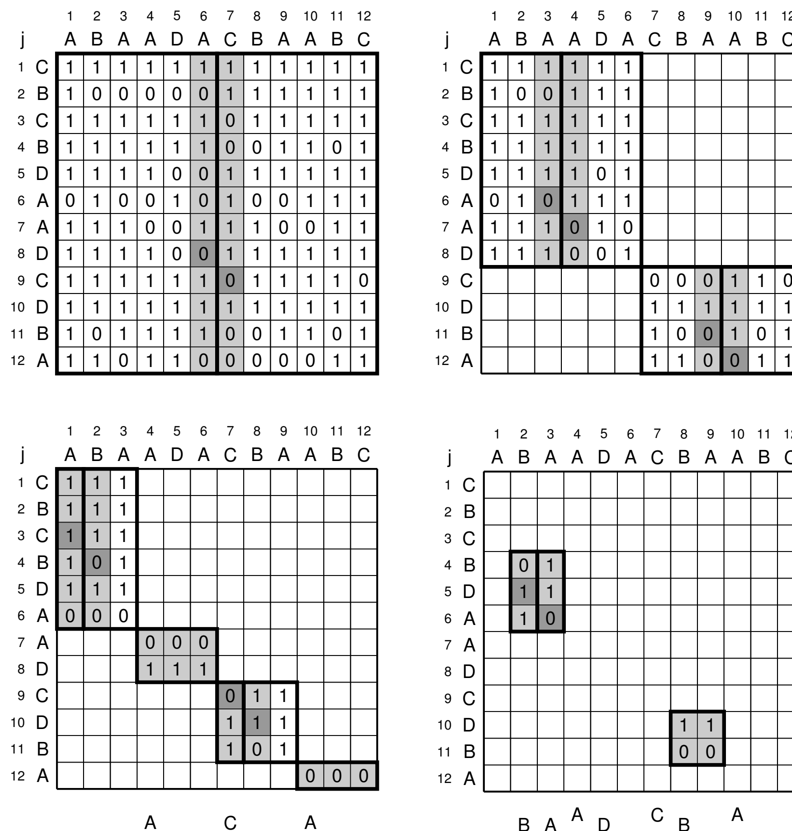
a więc całkowita złożoność czasowa tego algorytmu to suma (7.9) i (7.10):

$$O\left(n \left\lceil \frac{m}{w} \right\rceil + m \log n\right). \tag{7.11}$$

Jeśli $m = O(w)$, to złożoność pamięciowa zwykłego algorytmu równoległości bitowej, w którym przechowywane są wszystkie wektory bitowe kolumn, jest $\Theta(n)$ słów, a więc stosowanie wariantu Hirschberga nie ma sensu. Jeśli $n = O(w \log w)$, to złożoność pamięciowa zwykłego algorytmu równoległości bitowej jest $O(w \log w \lceil w \log w/w \rceil) = O(w \log^2 w)$ słów, co w praktyce jest wielkością tak małą, że również wariant Hirschberga nie powinien być stosowany. Niech zatem $m = \omega(w)$ oraz $n = \omega(w \log w)$, wtedy $\log n = O(n/w)$ i złożoność czasową (7.11) można wyrazić jako

$$\Theta\left(n \left\lceil \frac{m}{w} \right\rceil\right), \tag{7.12}$$

a więc tak samo jak w przypadku zwykłego algorytmu równoległości bitowej, podczas gdy złożoność pamięciowa jest $\Theta(n)$ słów. Na rys. 7.10 zilustrowano działanie tego algorytmu dla przypadku, gdy $w = 2$.



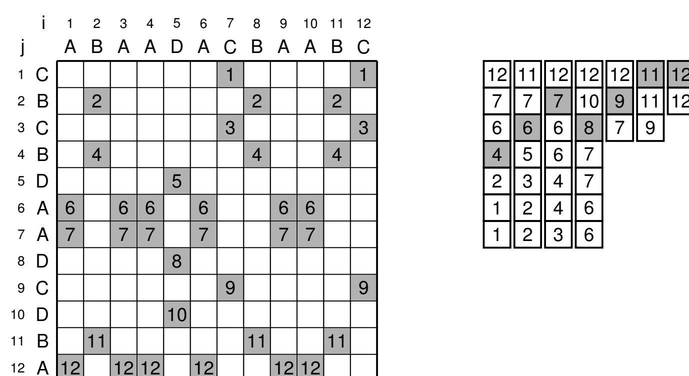
Rys. 7.10. Przykład działania algorytmu wyznaczania podciągu LCS opartego na metodzie równoległości bitowej z wykorzystaniem metody Hirschberga o liniowej złożoności pamięciowej dla ciągów: ABAADACBAABC, CBCBDAADCDBA dla $w = 2$. Na kolejnych rysunkach przedstawiono poziomy rekurencji. Kolumny zaznaczone na szaro pokazują przebieg podziału każdej (pod)macierzy. Pod macierzami wskazano, które symbole na danym poziomie rekurencji są wyznaczane. Wynik ostateczny to BAADCBA

Fig. 7.10. An example of the bit-parallel LCS computing algorithm with Hirschberg linear-space method computing the LCS for the sequences: ABAADACBAABC, CBCBDAADCDBA for $w = 2$. Figures show recurrence levels. Greyed columns show (sub)matrix divisions. Below two bottom matrices the computed symbols are given. They form the result: BAADCBA

Algorytmy równoległości bitowej zarówno dla problemu LCS, jak i dla innych problemów wykorzystują pewne techniki bitowe, które są często mało oczywiste bez gruntownego zrozumienia specyfiki reprezentacji liczb całkowitych i rzeczywistych w komputerach. Doskonałe wprowadzenie do dziedziny operacji bitowych stanowią książki Warrena [215] oraz Knutha [131]. Interesująca może też być książka Korena [132].

7.3.6. Związek z problemem najdłuższego podciągu rosnącego

Jak wspomniano w podrozdz. 2.5, algorytmy rozwiązywania problemu LIS mogą posłużyć do rozwiązania problemu LCS. W tym celu należy odpowiednio przekształcić dane wejściowe problemu LCS, którymi są dwa ciągi na dane dla problemu LIS, w którym potrzebny jest tylko jeden ciąg liczbowy. Podstawowa idea opiera się na wyznaczeniu wszystkich do-



X = 12 7 6 11 4 2 12 7 6 12 7 6 10 8 5 12 7 6 9 3 1 11 4 2 12 7 6 12 7 6 11 4 2 9 3 1
LIS: 4 6 7 8 9 11 12
LCS: BAADCBA

Rys. 7.11. Ilustracja działania algorytmu wyznaczania podciągu LCS opartego na algorytmie wyznaczania podciągu LIS. Po lewej stronie macierz programowania dynamicznego z zaznaczonymi dopasowaniami i liczbami, które je reprezentują. Poniżej ciąg liczbowy X. Po prawej stronie pokrycie utworzone dla ciągu X. Na dole rysunku znaleziony podciąg LIS na podstawie pokrycia oraz odpowiadający mu podciąg LCS

Fig. 7.11. Example of the LIS-based algorithm for the LCS problem. Left: dynamic programming matrix with marked matches and numbers representing that matches. Below: sequence X. Right: greedy cover for sequence X. Bottom: LIS for X and LCS obtained on that LIS

pasowań pomiędzy ciągami A i B oraz odpowiednim ich ponumerowaniu [106]. Sposób numerowania polega na wyznaczeniu tablicy zawierającej pod indeksem $x \in \Sigma$ listę indeksów wszystkich wystąpień symbolu x w ciągu B. Następnie w pętli przechodzi się ciąg A i dla każdego kolejnego symbolu a_i wpisuje się do ciągu liczbowego X indeksy wszystkich wystąpień a_i w ciągu B w kolejności malejącej. Złożoność czasowa tego przetwarzania wstępnego jest $O(\sigma + m) + O(n + d) = O(n + \sigma + d)$, gdzie d to liczba dopasowań. W ciągu liczbowym X wyznacza się następnie najdłuższy podciąg rosnący (LIS) w czasie $O(d \log\log m)$. Warto zauważyć, że występuje tu wariant problemu LIS, w którym symbole mogą się powtarzać (rys. 7.11). Elementy wyznaczonego podciągu są indeksami symboli w ciągu B, które tworzą podciąg LCS dla A i B. Całkowita złożoność czasowa tego algorytmu jest $O(\sigma + n + d \log\log m)$.

Związek pomiędzy problemami LIS i LCS został odnotowany w pracach Apostolico i Guerra [15, 19], Jacobsona i Vo [123] oraz Pevznera i Watermana [175]. Rozwinięcie tych idei znajduje się w pracach Eppsteina i in. [81, 82], w których zaproponowano algorytm o złożoności czasowej $O(n + D \log\log \min(D, nm/D))$, gdzie D jest liczbą dopasowań dominujących.

7.3.7. Inne wybrane algorytmy

Historycznie pierwszy algorytm dla problemu LCS o złożoności subkwadratowej zaproponowali Masek i Paterson [152]. Jego złożoność czasowa jest $O(nm/\log n)$ dla alfabetu ograni-

czonego oraz $O(nm \log\log n / \log n)$ dla alfabetu nieograniczonego. Opiera się on na tzw. metodzie czterech Rosjan (ang. *4-Russian method*) [21]. W skrócie, algorytm ten polega na podziale macierzy programowania dynamicznego na bloki niewielkich rozmiarów $e \times e$. Następnie na etapie przetwarzania wstępnego wyznacza się wszystkie możliwe macierze $e \times e$ dla możliwych ciągów je opisujących oraz wartości w komórkach sąsiednich do tego bloku. W ostatnim etapie wyznacza się całą macierz blok po bloku. Dobry opis tego algorytmu można znaleźć m.in. w [106].

W niektórych sytuacjach wyznaczanie podciągu LCS można przyspieszyć, jeśli ciągi wejściowe podda się wcześniej kompresji. Jednym z możliwych sposobów jest *kodowanie długości serii* (ang. *run length encoding*) [99]), dla którego zostały zaproponowane różne algorytmy wyznaczania podciągu LCS, m.in. [37, 20, 146, 13]. Złożoność czasowa najlepszego z nich [13] jest $O(\widetilde{n}\widetilde{m} \min(\gamma_1, \gamma_2))$, gdzie $\widetilde{n}$ oraz $\widetilde{m}$ są długościami ciągów A oraz B po kompresji algorytmem RLE, a γ_1 oraz γ_2 są liczbami, odpowiednio, elementów w dolnych i prawych krawędziach bloków, które są dopasowaniami (dopasowanie dla pary bloków definiowane jest analogicznie jak dla pary elementów). W praktyce, algorytmy te jednak rzadko znajdują zastosowanie, gdyż kompresja RLE jest stosunkowo mało skuteczna.

Zdecydowanie lepszy współczynnik kompresji osiągany jest za pomocą algorytmów Ziva–Lempela [224, 225, 218, 196]. Algorytmy te należą do kategorii algorytmów słownikowych, które najkrócej można scharakteryzować tak, że przetwarzając ciąg wejściowy zamiast symboli zapisują informację o tym, gdzie w przeszłości (w już przetworzonym fragmencie ciągu) znajduje się identyczny fragment. Tego typu redundancja jest znacznie częstsza w danych spotykanych w praktyce niż powtórzenia tego samego symbolu. Zgrubnie można stwierdzić, że algorytmy wyznaczania podciągu LCS stosujące tę metodę kompresji opierają się na tym, że jeśli jakiś fragment ciągu się powtarza, to podobnie do bieżącego może wyglądać jakiś przeszły fragment macierzy programowania dynamicznego. Algorytm Crochemore'a i in. [53, 45] dla alfabetu stałego rozmiaru ma złożoność czasową $O(Hn^2 / \log n)$, gdzie $0 \le H \le 1$ jest entropią[4] ciągu.

Inne sposoby przyspieszenia wyznaczania podciągu LCS opierają się na zastosowaniu kompresji gramatycznej (ang. *grammar-based compression*) i reprezentacji ciągów za pomocą gramatyk bezkontekstowych o jedynie dwóch typach produkcji, generujących dokładnie jeden ciąg [111] oraz algorytmu Byte-Pair [96]. Rytter pokazał w [182], jak szybko uzyskać taką gramatykę dysponując ciągiem skompresowanym algorytmami Ziva–Lempela [224, 225]. Wyczerpujący opis algorytmów o złożonościach $o(nm)$ dla problemu LCS można znaleźć m.in. w [216].

[4] Mówiąc nieco nieprecyzyjnie, entropia jest miarą, jak silnie dany ciąg da się skompresować. Dokładniejsze omówienie pojęcia entropii można znaleźć np. w [30, 54].

Jak już wspominano wcześniej, problemami pokrewnymi do problemu LCS są problemy odległości edycyjnej i uliniowienia ciągów. Problemy te można traktować jako swego rodzaju uogólnienie problemu LCS. Zwłaszcza problem uliniowienia ciągów ma duże znaczenie w praktyce. Z uwagi na bardzo duże rozmiary danych biologicznych, w których wyznacza się uliniowienie lokalne, w praktyce zamiast algorytmu Smitha–Watermana [195], którego czas działania dla tych danych jest zbyt długi,wykorzystuje się algorytmy heurystyczne, m.in., BLAST [9, 10]. Mimo iż podejścia te sprawdzają się w praktyce bardzo dobrze, to nie gwarantują znalezienia wszystkich istotnych biologicznie uliniowień lokalnych. Ciekawe rozwiązanie tego problemu zostało przedstawione w pracy Lama i in. [138]. Autorzy zaproponowali wstępne zbudowanie drzewa sufiksów dla długiego ciągu, którym może być np. ludzki genom. Następnie uliniowienie jest wyznaczane dla drugiego ciągu oraz tego drzewa. Dzięki temu, że w drzewie sufiksów [217, 153, 206, 136] powtarzające się podsłowa są reprezentowane przez jedną ścieżkę, to uliniowienie jest wyznaczane tylko jednokrotnie dla wszystkich powtarzających się podsłów.

Drzewa sufiksów cechują się, niestety, relatywnie dużą złożonością pamięciową wynoszącą $7n$ słów. Jednym z możliwych rozwiązań jest wykorzystanie zamiast nich indeksów FM [88, 89]. Opierają się one na transformacie Burrowsa–Wheelera [38], która jest także podstawą wydajnych algorytmów kompresji danych (m.in. [87, 60, 61, 62]). Indeksy te pozwalają na wykonywanie części operacji, możliwych do wykonania na drzewach sufiksów, jednak podzbiór ten wystarcza do efektywnego uliniawiania ciągów, a złożoność pamięciowa tych indeksów może być zbliżona do entropii ciągu. Wyniki eksperymentalne z [138] wskazują, że dzięki zastosowaniu tego podejścia przyspieszenie uzyskane w stosunku do algorytmu Smitha–Watermana wynosi ponad 1000.

Z innych ciekawych algorytmów warto wymienić algorytmy Hirschberga [113] zaprojektowane specjalnie do sytuacji, w której długość ciągu wynikowego jest znacznie mniejsza niż m lub bliska m.

7.4. Uogólnienie na przypadek wielu ciągów

W niektórych przypadkach porównywanie tylko 2 ciągów jest niewystarczające i dane wejściowe dla problemu stanowi zbiór $\mathcal{A} = \{A^1, A^2, \ldots, A^N\}$, składający się z N ciągów, dla których należy znaleźć najdłuższy wspólny podciąg. Taka sytuacja często występuje w bioinformatyce (głównie dla problemu uliniawiania ciągów). Stosunkowo prostym do zrealizowania rozwiązaniem jest uogólnienie algorytmu programowania dynamicznego. W celu znalezienia podciągu LCS dla wielu ciągów należy wyznaczyć N-wymiarową macierz M o rozmiarach $(|A^1|+1) \times (|A^2|+1) \times \cdots \times (|A^N|+1)$ zgodnie z następującą zależnością:

$$M([i_1,\ldots,i_N]^T)=\begin{cases}0 & \text{dla } i_k=0,\ \text{gdzie } 1\le k\le N,\\ \max(M([i_1-\delta_1,\ldots,i_N-\delta_N]^T)) & \text{dla } 0<i_k\le|A^k|, 0\le\delta_k\le 1,\\ & 0<\sum_k\delta_k<N,\ \text{gdzie}\\ & 1\le k\le N,\\ M([i_1-1,\ldots,i_N-1]^T)+1 & \text{dla } 0<i_k\le|A^k|, a^1_{i_1}=\cdots=a^N_{i_N},\\ & \text{gdzie } 1\le k\le N.\end{cases} \tag{7.13}$$

Złożoność czasowa tego algorytmu jest $O(2^N\Pi_{k=1}^N|A^k|)$. Drugi człon wynika z liczby komórek w macierzy, a pierwszy odzwierciedla fakt, że do policzenia pojedynczej komórki może być potrzebne znalezienie maksimum z 2^N-2 jej sąsiadów. Jeśli wartość N nie jest ograniczona, to złożoność czasowa tego algorytmu jest wykładnicza, czyli nawet dla kilku ciągów o praktycznej długości może on być zupełnie niemożliwy do zastosowania. Jednym z najszybszych sposobów rozwiązywania tego wariantu problemu LCS jest algorytm wykorzystujący rzadkość macierzy programowania dynamicznego [213]. Jego złożoność czasowa w przypadku pesymistycznym jednak także jest wykładnicza. O samym problemie wiadomo, że jest NP-trudny [149].

Ponieważ problemy uliniowienia ciągów, zarówno dwu (ang. *pairwise sequence alignment*, PSA), jak i wielu (ang. *multiple sequence alignment*, MSA), są bardzo podobne do problemów LCS dla dwu lub wielu ciągów (ang. *multiple longest common subsequence*, MLCS), to istniejące algorytmy wyznaczania PSA/MSA łatwo zaadaptować do problemów LCS/MLCS. Często podobną adaptację można wykonać w drugą stronę, np. algorytm Hirschberga wyznaczania podciągu LCS o złożoności pamięciowej $\Theta(n)$ został później zastosowany w problemie PSA [160]. Ponadto, te dwie rodziny problemów są równoważne, o ile wartości parametrów w PSA/MSA zostaną wybrane w określony sposób, np. wynikiem dla par identycznych symboli jest 1, a wynikiem wstawienia i usunięcia jest 0. Dokładne algorytmy znajdowania MSA wyznaczają taką samą N-wymiarową macierz, obliczaną na podstawie nieco tylko innej reguły rekurencyjnej.

W związku z tym, że w praktyce niemożliwe jest rozwiązywanie problemów MLCS oraz MSA w sposób dokładny, zaproponowanych zostało dla nich wiele heurystyk. Pierwszym etapem tych heurystyk jest zwykle wyznaczenie podciągu LCS (PSA) dla wszystkich $\binom{N}{2}$ par ciągów ze zbioru $\mathcal{A}$. Dalsze etapy są już różne w różnych algorytmach. Jedną z możliwości jest zamiana dwu najbardziej podobnych ciągów, tzn. ciągów, których podciąg LCS jest najdłuższy (wynik uliniowienia jest największy), przez ich podciąg LCS (uliniowienie) i powtórzenie tego procesu $N-2$ razy otrzymując pojedynczy podciąg (uliniowienie).[5] Postępowanie to nie gwarantuje, że otrzymany podciąg jest najdłuższy z możliwych (ma najlepszy możliwy wynik),

[5]Dla problemu MSA stosowane są także inne algorytmy wyboru dwu ciągów (uliniowień), por. np. [77, 183].

ale często uzyskane rezultaty są zadowalające. Przegląd istniejących metod wyznaczania MSA można znaleźć w [101] a te same idee są możliwe do zastosowania w problemie MLCS. Dla problemu MLCS stosowane są często także ogólne metody optymalizacji kombinatorycznej (m.in. [34, 78, 79, 192]).

7.5. Algorytmy równoległe

Algorytmy rozwiązywania problemu LCS zostały zaproponowane dla wielu architektur równoległych. Poniżej wymieniono niektóre z tych rozwiązań. Lipton i Lopresti [145] zaproponowali algorytm o złożoności czasowej $O(n)$ dla macierzy systolicznej z n procesorami. Algorytm Ranka–Sahni [179] ma złożoność czasową $O(\sqrt{n}\log n)$ w modelu hiperkostki z $\Theta(n^2)$ procesorami. Dla modelu PRAM CRCW (ang. *Concurent Read Concurent Write*) zaproponowano algorytmy: Apostolico i in. [18] o złożoności czasowej $O(\log n(\log\log m)^2)$ dla $\Theta(nm/\log\log m)$ procesorów, Babu i Saxena [23] o złożonościach czasowych $O(\log m)$ oraz $O(\log^2 m)$ dla odpowiednio $\Theta(nm)$ oraz $\Theta(nm\log m)$ procesorów. Algorytm Rajko–Aluru [178] charakteryzuje się złożonością czasową $O(nm/p)$ dla p równolegle pracujących komputerów (implementacja w C++ z wykorzystaniem MPI). Dla modelu SIMD Edimston i Wagner [80] zaproponowali algorytm o złożoności czasowej $O(m+n)$ dla $\Theta(m)$ procesorów.

Jedną z ostatnich publikacji dotyczących równoległego rozwiązywania problemu LCS jest praca Kursche i Tiskina [133], w której zaprezentowano algorytm dla modelu BSP (por. podrozdz. 1.2). W algorytmie tym macierz programowania dynamicznego dzielona jest na prostokątne obszary, tzw. pudełka (ang. *boxes*) rozmiaru $b_{\mathrm{w}} \times b_{\mathrm{h}}$ komórek (rys. 7.12). Wnętrze każdego pudełka obliczane jest sekwencyjnie, ale poszczególne pudełka obliczane są równolegle. Zależności występujące w macierzy programowania dynamicznego dla problemu LCS są lokalne, tzn. aby wyznaczyć wartość jednej komórki, konieczna jest znajomość komórek sąsiednich: lewej, górnej i lewej-górnej. Wynika z tego, że przed obliczeniem pudełka muszą być znane wartości wszystkich komórek z sąsiednich pudełek: lewego, górnego i lewego-górnego. Jednym z możliwych schematów jest równoległe obliczanie pudełek w grupach: $\{(0,0)\}$, $\{(0,1),(1,0)\}$, $\{(0,2),(1,1),(2,0)\}$, $\{(0,3),(1,2),(2,1),(3,0)\}$, $\{(1,3),(2,2),(3,1)\}$, $\{(2,3),(3,2)\}$, $\{(3,3)\}$.

Krusche i Tiskin ocenili możliwości zrównoleglenia w ten sposób zarówno algorytmu programowania dynamicznego, jak i algorytmu równoległości bitowej Crochemore'a i in. [52], zakładając pewne parametry modelu BSP. Mimo braku implementacji tych algorytmów przewidywane wartości przyspieszenia dla wersji równoległej są obiecujące.

(0,0) [1]	(0,1) [2]	(0,2) [3]	(0,3) [4]
(1,0) [2]	(1,1) [3]	(1,2) [4]	(1,3) [5]
(2,0) [3]	(2,1) [4]	(2,2) [5]	(2,3) [6]
(3,0) [4]	(3,1) [5]	(3,2) [6]	(3,3) [7]

Rys. 7.12. Podstawowa idea dekompozycji macierzy programowania dynamicznego dla problemu LCS na pudełka. Pary liczb są identyfikatorami pudełek. Liczby w nawiasach kwadratowych oznaczają numer etapu przetwarzania pudełka. Kolory zostały wprowadzone jedynie, aby podkreślić kolejność przetwarzania pudełek

Fig. 7.12. A general scheme of decomposing the computation of the DP matrix for the LCS problem into boxes. Pairs of numbers denote the box id's. The number in square brackets is the stage number. The colors only emphasise the antidiagonal computation of boxes

7.6. Algorytmy dla procesorów graficznych

7.6.1. Wprowadzenie

Ponieważ architektura procesorów GPU jest znacząco różna od modeli wymienionych wyżej, to żaden ze wspomnianych wyżej algorytmów nie może być łatwo zaadaptowany do uruchomienia w procesorze GPU. W literaturze istnieje kilka prac dotyczących wykorzystania procesorów GPU do równoległego rozwiązywania problemu LCS lub problemów pokrewnych. Jak zostało wspomniane w podrozdz. 1.3, w procesorze GPU synchronizować pomiędzy sobą można tylko wątki należące do jednego bloku. Konieczność synchronizacji wszystkich wątków za pomocą procesora CPU i kolejnych wywołań jądra rodzi duże problemy z wydajnością. Większość autorów omija ten problem i zamiast równoległego algorytmu wyznaczania podciągu LCS proponuje, aby każdy blok wątków wyznaczał podobieństwo innej pary ciągów. Takie częściowo równoległe podejście jest jednak uzasadnione w niektórych przypadkach. Za przykład może posłużyć sytuacja, w której pojawia się ciąg zawierający sekwencję białkową i należy porównać go ze wszystkimi sekwencjami białkowymi znajdującymi się w bazie danych. W takim przypadku nie ma znaczenia, czy w danym momencie algorytm będzie porównywał jedną parę, czy każdy blok inną parę. Dzięki temu, że w tym drugim przypadku nie występują takie problemy z synchronizacją, można oczekiwać dla niego większych wartości przyspieszenia.

Podejście to zostało zastosowane przez Liu i in. [147] dla problemu uliniowienia ciągów. Autorzy wykonali implementację w języku OpenGL Shading Language, a eksperymenty przeprowadzili na procesorze NVidia GeForce 7800 GTX. Jako odniesienie wybrana została implementacja sekwencyjna działająca na procesorze Intel Pentium4 (zegar 3000 MHz). Uzyskane

wyniki pokazują, że algorytm ten był szybszy od 4,5 do 15 razy (im dłuższe były ciągi, tym większe było przyspieszenie) od algorytmu sekwencyjnego. Podobne podejście przyjęli Manavski i Valle [151]. Ich implementacja została wykonana w jezyku C z użyciem biblioteki CUDA [167], a eksperymenty przeprowadzono z użyciem procesora NVidia GeForce 8800 GTX. Niestety, wyniki eksperymentalne nie zawierają czasów działania wersji sekwencyjnej dla procesora CPU. Z wyników wynika jedynie, że algorytm ten jest 3,3–9,4 razy szybszy (konfiguracja z jedną kartą graficzną) lub 6,3–18 razy szybszy (konfiguracja z dwoma kartami graficznymi) niż implementacja Liu i in. Najlepsze wyniki dla problemu uliniowienia ciągów zostały uzyskane przez Ligowskiego i Rudnickiego [143]. Implementacja różni się szczegółami, ale idea jest taka sama jak w propozycji Manavskiego–Valle'a i jest od niej szybsza około 3 razy (NVidia 9800 GX2). Aktualnie najszybszym algorytmem dla problemu uliniowienia ciągów dla procesorów CPU jest algorytm Farrara [86], który wykorzystuje rozszerzenia SSE2 dostępne w procesorach firmy Intel. Algorytm Ligowskiego–Rudnickiego (wykonywany na pojedynczym procesorze GPU) jest ponad 8 razy szybszy niż algorytm Farrara wykonywany na jednym rdzeniu procesora CPU.

Prawdziwie równoległy algorytm wyznaczania podciągu LCS za pomocą procesorów GPU został zaproponowany przez Kloetzliego i in. [129]. Podejście to staje się praktyczne dla dostatecznie długich ciągów wejściowych (rzędu milionów symboli). Dla takich ciągów rozmiar macierzy programowania dynamicznego byłby rzędu terabajta, a więc konieczne jest zastosowanie algorytmu o niskiej złożoności pamięciowej, np. algorytmu Hirschberga. W rzeczywistości, autorzy zastosowali algorytm Chowhury'ego–Ramachandrana [44], który dobrze nadaje się do implementacji dla procesorów GPU i jest wariantem algorytmu Hirschberga. W algorytmie Kloetzliego i in. problem dzieli się na podproblemy metodą dziel-i-zwyciężaj. Każdy podproblem rozwiązywany jest na osobnym multiprocesorze. W celu uzyskania maksymalnego przyspieszenia autorzy stworzyli algorytm hybrydowy działający równocześnie na procesorach CPU i GPU. Idea ta wzięła się z obserwacji następujących cech procesorów GPU:

- rozmiar szybkiej pamięci w procesorach GPU jest bardzo mały,
- wywołanie funkcji w procesorze GPU zajmuje znacznie więcej czasu niż w procesorze CPU, a więc zysk z obliczeń równoległych w procesorach GPU może być stracony przez wolną komunikację.

Z powyższych powodów autorzy przeznaczyli do obliczeń w procesorach GPU tylko podmacierze o średnich rozmiarach. Eksperymentalnie znaleziono, że najlepsze wyniki uzyskiwane są dla następującego przydziału zadań w zależności od długości ciągów:

- $(0, 2^{11})$ – klasyczny algorytm programowania dynamicznego dla procesora CPU,
- $[2^{11}, 2^{16}]$ – algorytm rekurencyjny Hirschberga dla procesorów GPU,
- $(2^{16}, n)$ – algorytm rekurencyjny Hirschberga dla procesora CPU.

Eksperymenty przeprowadzone na procesorze AMD Athlon 64 i karcie grafiki z rodziny NVidia G80 GTX pokazały, że ten algorytm jest około 10 razy szybszy niż algorytm Hirschberga dla procesora CPU oraz około 5 razy szybszy niż algorytm Chowhury'ego–Ramachandrana [44] dla procesora CPU.

7.6.2. Podstawowe idee

W niniejszym podrozdziale omawiane są dwa algorytmy równoległe dla procesorów GPU wyznaczające długość podciągu LCS.[6] Algorytmy te zostały zaproponowane przez autora w [71, 66]. Są one równoległymi wersjami klasycznego algorytmu programowania dynamicznego (podrozdz. 7.3.1) oraz algorytmu równoległości bitowej (podrozdz. 7.3.4). W obu algorytmach wyznaczana jest dwuwymiarowa macierz, zgodnie z pewną zależnością rekurencyjną. Macierz ta dzielona jest na pudełka, tak jak w [133], które są obliczane zgodnie z tzw. *zasadą drugiej przekątnej*.[7] Każde pudełko jest obliczane przez jeden blok wątków.

Pseudokod obu tych algorytmów przedstawiony jest na rys. 7.13. (Pseudokod ten jest także wspólny dla algorytmów rozwiązujących problemy dyskutowane w dwóch kolejnych rozdziałach). Na etapie przetwarzania wstępnego macierz dzielona jest na pudełka. Wysokość pudełka jest równa liczbie wątków należących do jednego bloku, a więc, z uwagi na ograniczenia architektury CUDA, $32 \leq b_\mathrm{h} \leq 512$ oraz b_h jest wielokrotnością 32. (Dla prostoty opisu algorytmów zakłada się, że n oraz m są zawsze potęgami 2. Adaptacja prezentowanych algorytmów dla przypadku ogólnego jest jednak natychmiastowa). Szerokość pudełka, b_w, dla prostoty implementacji, jest całkowitą wielokrotnością b_h. Wartości liczbowe parametrów b_h, b_w zależą od długości ciągów i będą podane przy omawianiu wyników eksperymentalnych.

Dla algorytmu równoległości bitowej konieczne jest także wyznaczenie wektorów masek, co jest wykonywane w procesorze CPU. Następnie prowadzone są właściwe obliczenia w procesorze GPU. Dla wygody prezentacji założone zostanie na moment, że macierz programowania dynamicznego, którą należy obliczyć, składa się z 16 pudełek (rys. 7.12). W pierwszej iteracji, zbiór równolegle obliczanych pudełek zawiera tylko jeden element: $\mathcal{S} = \{(0,0)\}$, ponieważ tylko dla niego znane są wartości komórek sąsiednich. Po wyznaczeniu wartości w tym pudełku wartości z prawego i dolnego brzegu są kopiowane do pamięci globalnej (nie są one przesyłane do pamięci RAM). W drugiej iteracji, w procesorze GPU równolegle obliczane są pudełka $\mathcal{S} = \{(0,1),(1,0)\}$. W trakcie uruchamiania kolejnych obliczeń w procesorze GPU przesyłane są tylko indeksy pudełek do obliczenia oraz wskaźniki do struktur danych w pamięci globalnej procesora GPU, dzięki czemu ilość przesyłanych danych jest znikoma. W kolejnych iteracjach: $\mathcal{S} = \{(0,2),(1,1),(2,0)\}$, $\mathcal{S} = \{(0,3),(1,2),(2,1),(3,0)\}$, $\mathcal{S} = \{(1,3),(2,2),(3,1)\}$,

[6]Wszystkie implementacje dla procesorów GPU wykonane zostały dla architektury CUDA firmy NVidia.
[7]Równocześnie obliczane są pudełka, których suma indeksów wiersza i kolumny jest identyczna.

```
LCS*-CUDA(...)
Wejście: dwa lub trzy ciągi
Wyjście: wyznaczony podciąg
   {Przetwarzanie wstępne}
1  Podziel macierz(e) programowania dynamicznego na pudełka o wysokości b_h i szerokości b_w
2  Jeśli to konieczne, wykonaj przetwarzanie wstępne (np. wyznacz wektory bitowe masek)
3  Przydziel pamięć na karcie graficznej i skopiuj konieczne dane do pamięci procesora GPU
   {Obliczenia właściwe}
4  while not wszystkie pudełka obliczone do
5        Wyznacz zbiór S zawierający pudełka, które mogą być obliczane
6        Wylicz pudełka z S w sposób równoległy w procesorze GPU – każdy blok
         oblicza jedno pudełko (pojedyncze wykonanie kodu jądra)
   {Przetwarzanie końcowe}
7  Wyznacz wynik końcowy na podstawie wyników obliczonych przez bloki w procesorze GPU
```

Rys. 7.13. Ogólny schemat algorytmów rozwiązujących problem LCS i jego warianty w modelu programowania CUDA

Fig. 7.13. A general scheme of the algorithms solving the LCS, and related problems in the CUDA programming model

$\mathcal{S} = \{(2,3),(3,2)\}$, $\mathcal{S} = \{(3,3)\}$. Na samym końcu wynik kopiowany jest z pamięci globalnej procesora GPU do pamięci RAM.

Komórki należące do każdego pudełka obliczane są zgodnie z zasadą drugiej przekątnej, ponieważ występują dla nich takie same zależności jak pomiędzy pudełkami na wyższym poziomie. Ilustrację sposobu obliczania pudełka pokazuje rys. 7.14 dla $b_h = 4$ i $b_w = 12$. Liczby na rysunku pokazują kolejność w jakiej komórki są wczytywane z pamięci globalnej/obliczane/zapisywane do pamięci globalnej. Kolejne etapy to:

- 1 – wartości komórek z górnej krawędzi dla b_h najbardziej na lewo położonych kolumn wczytywane są z pamięci globalnej do pamięci wspólnej,
- 2 – wartości komórek z lewej krawędzi wczytywane są z pamięci globalnej do rejestrów (każdy wątek przechowuje wartość lewej sąsiedniej komórki w rejestrze),
- 3 – wartości komórek lewych-górnych sąsiadów wczytywane są z pamięci globalnej do pamięci wspólnej,
- $4_1,\ldots,4_4$ – cztery etapy, w których obliczane są wartości komórek macierzy,
- 5 – wartości komórek z górnej krawędzi z b_h kolumn wczytywane są z pamięci globalnej do pamięci wspólnej,
- $6_1,\ldots,6_4$ – cztery etapy, w których obliczane są wartości komórek macierzy,
- 7 – wartości komórek z dolnej krawędzi zapisywane są w pamięci globalnej (będą potrzebne do wyznaczania pudełka leżącego poniżej pudełka bieżącego),
- 8 – wartości komórek z górnej krawędzi z b_h kolumn wczytywane są z pamięci globalnej do pamięci wspólnej,
- $9_1,\ldots,9_4$ – cztery etapy, w których obliczane są wartości komórek macierzy,

3	1	1	1	1	5	5	5	5	8	8	8	8
2/3	4_1	4_2	4_3	4_4	6_1	6_2	6_3	6_4	9_1	9_2	9_3	$9_4/13$
2/3	4_2	4_3	4_4	6_1	6_2	6_3	6_4	9_1	9_2	9_3	9_4	$11_1/13$
2/3	4_3	4_4	6_1	6_2	6_3	6_4	9_1	9_2	9_3	9_4	11_1	$11_2/13$
2	$4_4/7$	$6_1/7$	$6_2/7$	$6_3/7$	$6_4/10$	$9_1/10$	$9_2/10$	$9_3/10$	$9_4/12$	$11_1/12$	$11_2/12$	$11_3/12/13$

Rys. 7.14. Kolejność obliczeń w pudełku w algorytmie rozwiązywania problemu LCS dla procesora GPU. Komórki zawierające więcej niż jedną liczbę (rozdzielone ukośnikiem) przetwarzane są w więcej niż jednym etapie. Kolory jedynie poprawiają czytelność rysunku. Ramka pokazuje wnętrze pudełka. Komórki poza ramką należą do sąsiednich pudełek

Fig. 7.14. The order of computing a box in GPU algorithm for the LCS problem. If some cell contains more than one number (separated by a slash), it is processed in more than one stage. Colors are only for clarity of the figure. The frame shows the interior of the box. The cells outside the frame belong to the neighbouring boxes

- 10 – wartości komórek z dolnej krawędzi zapisywane są w pamięci globalnej,
- $11_1, \ldots, 11_3$ – trzy etapy, w których obliczane są wartości komórek macierzy,
- 12 – wartości komórek z dolnej krawędzi zapisywane są w pamięci globalnej,
- 13 – wartości komórek z prawej krawędzi zapisywane są w pamięci globalnej (będą potrzebne do wyznaczania pudełka leżącego z prawej strony pudełka bieżącego).

W dalszej części wyznaczone zostaną teoretyczne przyspieszenia proponowanych algorytmów w stosunku do algorytmu sekwencyjnego dla procesora CPU. W tym celu przyjęty zostanie uproszczony model obliczeniowy dla procesorów GPU, w którym procesor GPU składa się z η_1 multiprocesorów, a każdy z nich zawiera η_2 rdzeni pracujących według zasady SIMD. Dla prostoty obliczeń w analizie pominięta zostanie kwestia przepustowości pamięci. We wzorach przyjęte zostaną następujące oznaczenia: $n' = n/b_\mathrm{w}$, $m' = m/b_\mathrm{h}$. Ponadto, przyjęte zostaną założenia, że $b_\mathrm{w} \geq b_\mathrm{h} \geq 32$ (dla algorytmu opartego na programowaniu dynamicznym) oraz $b_\mathrm{w} \geq b_\mathrm{h}/w_\mathrm{g} \geq 32$ (dla algorytmu opartego na równoległości bitowej), $m \geq \max(w_\mathrm{c}, w_\mathrm{g})$.

7.6.3. Algorytm programowania dynamicznego

Przekształcenie sekwencyjnego algorytmu programowania dynamicznego (DP) dla problemu LCS w algorytm równoległy dla procesorów GPU, zgodnie ze schematem zaproponowanym powyżej, jest stosunkowo proste. Przetwarzanie wstępne (rys. 7.13, wiersz 2) nie jest tu wykonywane. Struktury danych znajdujące się w pamięci globalnej procesora GPU to:

- odpowiednio n oraz m słów dla ciągów A oraz B,
- tablica n słów dla komórek z górnej krawędzi (b_w dla każdego równolegle przetwarzanego pudełka),
- tablica m słów dla komórek z lewej krawędzi (b_h dla każdego równolegle przetwarzanego pudełka),
- tablica $3m/b_\mathrm{h}$ słów dla komórek zawierających wartość prawej dolnej komórki lewego-górnego sąsiedniego pudełka do pudełka bieżącego.

Z powyższego wynika, że całkowita zajętość pamięci globalnej w tym algorytmie jest $\Theta(n+m) = \Theta(n)$ słów. Pamięć dla lewych i górnych krawędzi może być liniowa dzięki temu, że każde równolegle obliczane pudełko wykorzystuje inny obszar tych tablic. Wymaganych jest 3 razy więcej słów w tablicy przechowującej wartości prawych dolnych komórek pudełek, ponieważ wartości z dwóch ostatnio przetwarzanych pudełek z bieżącego wiersza muszą być znane, a także gdzieś musi być możliwość przechowania wartości analogicznej komórki z bieżącego pudełka. Wynik końcowy znajduje się w prawej dolnej komórce prawego dolnego pudełka, a więc tylko ta liczba musi być skopiowana do pamięci RAM i żadne inne obliczenia nie są konieczne.

Część kodu tego algorytmu pokazana jest na rys. 7.15. Parametrami wywołania funkcji są:

- block_x – maksymalny indeks poziomy pudełka wyznaczanego w bieżącym wykonaniu jądra,
- block_y – indeks pionowy pudełka wskazanego powyżej,
- g_A – wskaźnik do tablicy (umieszczonej w pamięci globalnej) rozmiaru n zawierającej w i-tej komórce wartość a_i oraz $M(i, j')$, gdzie j' jest największym takim indeksem j, dla którego wartość $M(i, j)$ została już obliczona,
- g_B – wskaźnik do tablicy (umieszczonej w pamięci globalnej) rozmiaru m zawierającej w j-tej komórce wartość b_j oraz $M(i', j)$, gdzie i' jest największym takim indeksem i, dla którego wartość $M(i, j)$ została już obliczona,
- g_res – wskaźnik do tablicy (umieszczonej w pamięci globalnej) rozmiaru $3\lceil m/w \rceil$ zawierającej wartości prawych dolnych komórek ostatnio obliczanych pudełek.

Symbole ciągu B wczytywane są z pamięci globalnej do rejestrów sB (wiersze 17–28). Ciąg A wczytywany jest do tablicy s_A (znajdującej się w pamięci wspólnej) w partiach (np. wiersze 15–16, 28–29), ponieważ s_A funkcjonuje jako bufor cykliczny. Za każdym razem, kiedy wątek oblicza wartość pojedynczej komórki, znane są dla niej wartości:

- komórki będącej górnym sąsiadem – wartość w tablicy s_u (z pamięci wspólnej),
- komórki będącej lewym sąsiadem – wartość w zmiennej rejestrowej v,
- komórki będącej lewym-górnym sąsiadem – wartość w zmiennej rejestrowej v_upp.

Opierając się na tych wartościach oraz na symbolach ciągów A, B, wyznaczana jest wartość bieżącej komórki, która jest następnie umieszczana w tablicy s_u (wiersze 33–34). Za każdym razem kiedy wyznaczona jest połowa wartości znajdujących się w tablicy s_u, wartości te kopiowane są do tablicy g_A umieszczone w pamięci globalnej (wiersze 38, 44). Po zakończeniu przetwarzania pudełka wartości komórek z prawej krawędzi są zapisywane do tablicy g_B (linia 43). Ponadto, wartość prawej-dolnej komórki jest umieszczana w tablicy g_res.

Dla przejrzystości prezentacji, pokazany kod dotyczy tylko wyznaczania środkowej części pudełka, tj. odpowiadającej etapom 6_1–6_4 oraz 9_1–9_4 z rys. 7.14. Kod dla etapów 4_1–4_4 oraz 11_1–11_3 jest podobny, z tą różnicą, że niektóre wątki nie wykonują obliczeń.

```
1  __global__ void Kernel_LCS_DP(const unsigned int block_x,
2    const unsigned int block_y, uint2 *g_A, uint2 *g_B, unsigned int *g_res)
3  {
4    __shared__ unsigned int s_A[2*BLK_HEIGHT];
5    __shared__ unsigned int s_u[2*BLK_HEIGHT];
6    unsigned int sB;
7    int i;
8    unsigned int v, v_upp;
9
10   const unsigned int num_threads = BLK_HEIGHT;
11   const unsigned int tid = threadIdx.x;
12   const unsigned int bx0 = block_x - blockIdx.x;
13   const unsigned int by0 = block_y + blockIdx.x;
14   ...
15   uint2 tmp = g_A[orig_x + i + tid];
16   s_A[(tid + i) & BLK_MASK] = tmp.x;   s_u[(tid + i) & BLK_MASK] = tmp.y;
17   tmp = g_B[orig_y + tid];
18   sB = tmp.x;  v  = tmp.y;
19
20   // Left-upper
21   if(tid > 0)            v_upp = g_B[orig_y + tid - 1].y;
22   else if(orig_y == 0)   v_upp = 0;
23   else                   v_upp = g_res[off_res_read + by0 - 1];
24   ...
25
26   for(; i < BLK_WIDTH; ) {
27     int i_max = i + num_threads;
28     uint2 tmp = g_A[orig_x + i + tid];
29     s_A[(tid + i) & BLK_MASK] = tmp.x;   s_u[(tid + i) & BLK_MASK] = tmp.y;
30
31     for(; i < i_max; ++i) {
32       int n_i = (i - tid) & BLK_MASK;
33       v = max(v, v_upp + (sB == s_A[n_i]));    v_upp    = s_u[n_i];
34       v = max(v_upp, v);                       s_u[n_i] = v;
35       __syncthreads();
36     }
37
38     g_A[orig_x+tid+i-2*num_threads].y = s_u[(tid+i-2*num_threads) & BLK_MASK];
39     __syncthreads();
40   }
41   ...
42
43   g_B[orig_y+tid].y = v;
44   g_A[orig_x+tid+i-2*num_threads].y = s_u[(tid+i-2*num_threads) & BLK_MASK];
45
46   if(tid == BLK_HEIGHT - 1)
47     g_res[off_res_write+by0] = v;
48
49   __syncthreads();
50 }
```

Rys. 7.15. Część kodu jądra algorytmu LCS-DP-CUDA rozwiązującego problem LCS metodą DP w procesorze GPU
Fig. 7.15. A part of the kernel code for the LCS DP algorithm (LCS-DP-CUDA) at GPU

Poniżej rozważone zostanie teoretyczne przyspieszenie tej wersji algorytmu w stosunku do algorytmu sekwencyjnego, działającego w czasie $\Theta(nm)$. Do obliczenia pojedynczego pudełka potrzeba $\Theta(b_{\mathrm{w}}/\eta_2)$ połączonych (ang. *coalesced*) dostępów do pamięci globalnej, aby wczytać związaną z tym pudełkiem część ciągu A. Podobnie potrzeba $\Theta(b_{\mathrm{h}}/\eta_2)$ połączonych dostępów do pamięci globalnej, aby wczytać części ciągu B. Taka sama liczba połączonych dostępów do pamięci globalnej jest wymagana, aby wczytać oraz zapisać wartości na krawędziach pu-

dełka. Z tego wynika, że sumaryczny czas dostępu do pamięci globalnej dla jednego pudełka jest $\Theta((b_\mathrm{w}+b_\mathrm{h})/\eta_2)=\Theta(b_\mathrm{w}/\eta_2)$. Dostępy do pamięci globalnej przez różne multiprocesory wykonywane są sekwencyjnie, co zostanie wzięte pod uwagę nieco później.

Każdy multiprocesor wyposażony jest w osobną pamięć wspólną i nie występuje serializacja wątków operujących na pamięci wspólnej, jeśli należą one do różnych bloków. Wszystkie dostępy do pamięci wspólnej wykonywane są bez konfliktów banków (ang. *bank conflicts*), a więc równolegle. Liczba przebiegów pętli w ramach jednego pudełka jest $\Theta(b_\mathrm{w}+b_\mathrm{h})=\Theta(b_\mathrm{w})$, z czego liczba operacji wykonywanych dla wyliczenia jednego pudełka to:

$$\Theta\left(\frac{b_\mathrm{w}b_\mathrm{h}}{\eta_2}\right). \tag{7.14}$$

Rozważając teraz wyznaczanie kolejnych pudełek, można zauważyć, że w pierwszym uruchomieniu kodu jądra, przetwarzane jest tylko 1 pudełko, w drugim uruchomieniu kodu jądra – 2 pudełka itd. Maksymalna liczba pudełek, które mogą być przetwarzane w jednym uruchomieniu kodu jądra to $\min(n',m')$. Złożoność czasowa pojedynczego wykonania kodu jądra zależy liniowo od maksymalnej liczby pudełek przydzielonych do jednego multiprocesora, tzn. e pudełek wyznaczanych jest w $\lceil e/\eta_2\rceil$ krokach. Z tego, sumaryczna liczba kroków (dla wszystkich wykonań kodu jądra) jest

$$\Theta(\max(n',m'))+\Theta\left(\frac{n'm'}{\min(n',m',\eta_1)}\right)=\Theta\left(\frac{n'm'}{\min(n',m',\eta_1)}\right). \tag{7.15}$$

Jako że pojedyncze pudełko obliczane jest w czasie (7.14), sumaryczna złożoność czasowa obliczania pudełek jest

$$\Theta\left(\frac{n'm'}{\min(n',m',\eta_1)}\right)\times\Theta\left(\frac{b_\mathrm{w}b_\mathrm{h}}{\eta_2}\right)=\Theta\left(\frac{nm}{\min(n',m',\eta_1)\,\eta_2}\right). \tag{7.16}$$

Łącznie istnieje $\Theta(n'm')$ pudełek, przez co sumaryczny czas dostępu do pamięci globalnej jest

$$\Theta\left(\frac{n'm'b_\mathrm{w}}{\eta_2}\right). \tag{7.17}$$

Wniosek 7.1. *Złożoność czasowa zaprezentowanego algorytmu wyznaczania podciągu LCS (*LCS-DP-CUDA*) jest sumą (7.16) i (7.17):*

$$\Theta\left(\frac{nm}{\min(n',m',\eta_1)\eta_2}+\frac{nm}{\eta_2 b_\mathrm{h}}\right)=\Theta\left(\frac{nm}{\eta_2\min(\frac{n}{b_\mathrm{w}},\frac{m}{b_\mathrm{h}},\eta_1,b_\mathrm{h})}\right). \tag{7.18}$$

Wniosek 7.2. *Przyspieszenie w stosunku do wersji sekwencyjnej dla procesora CPU jest*

$$\Theta\left(\eta_2\min\left(\frac{n}{b_\mathrm{w}},\frac{m}{b_\mathrm{h}},\eta_1,b_\mathrm{h}\right)\right). \tag{7.19}$$

7.6.4. Algorytm równoległości bitowej

W algorytmie równoległości bitowej (BP) dla problemu LCS (podrozdz. 7.3.4) na etapie przetwarzania wstępnego wyznacza się wektory masek Y_x. W wersji równoległej algorytmu wektory te wyznaczane są w procesorze CPU, a następnie przesyłane są do pamięci globalnej procesora GPU. Wektory te mogą być używane przez algorytm równoległy na kilka różnych sposobów. Najszybszym sposobem jest wykonanie ich kopii w pamięci wspólnej. Niestety, jest to możliwe tylko dla małych alfabetów, np. dla $b_\mathrm{h} = 32$ i 16 KB (4096 słów) pamięci wspólnej oczywistym limitem na rozmiar alfabetu jest $\sigma = 4096/b_\mathrm{h} = 128$. Pamięć wspólna używana jest jednak także do innych celów, m.in. do przechowywania fragmentów ciągów A i B, wymiany informacji o przeniesieniach przy dodawaniu wektorów bitowych. Ponadto, aby wykorzystać w pełni możliwości procesorów GPU, liczba bloków powinna być kilkakrotnie większa niż liczba multiprocesorów. Wszytko to powoduje, że maksymalny rozmiar alfabetu, dla którego możliwe jest wczytanie wszystkich wektorów masek do pamięci wspólnej, jest znacznie mniejszy i w praktyce wynosi 20–32.

Alternatywą jest wykorzystywanie wektorów masek bitowych znajdujących się w pamięci globalnej. Niestety, przy takim podejściu opóźnienia spowodowane dostępem do tej pamięci będą bardzo duże i w dalszych badaniach nie brano tego wariantu pod uwagę.

Trzecie podejście jest połączeniem dwóch powyższych, w którym starano się połączyć zalety każdego z opisanych podejść przy jednoczesnej minimalizacji ich wad. Najszybszym sposobem dostępu do pamięci globalnej jest odczyt połączony, tzn. kolejne wątki operują na kolejnych słowach z pamięci globalnej. Niestety, wątki wyznaczające macierz muszą to robić metodą drugiej przekątnej, co powoduje, że każdy kolejny wątek może potrzebować fragmentu wektora maski, dotyczącego innego symbolu alfabetu (takiego, jaki opisuje odpowiednią kolumnę macierzy). Dlatego też nie jest możliwe, aby dostęp do wektorów masek przez kolejne wątki był połączony. Wstępne eksperymenty pokazały, że dobrym kompromisem jest wykonywanie 128-bitowego dostępu do pamięci globalnej przez co czwarty wątek. Oznacza to, że fragment wektora masek dla 4 sąsiednich wątków ($w_\mathrm{g} = 32$) odczytywany jest w sposób połączony. Wartości te umieszczane są w tablicy pomocniczej znajdującej się w pamięci wspólnej, a każdy wątek pobiera z tej tablicy fragment wektora masek, kiedy go potrzebuje. Podejście to redukuje liczbę dostępów do pamięci globalnej o czynnik 4 w stosunku do wariantu, w którym każdy wątek bezpośrednio czyta z pamięci globalnej, a przy tym nie obciąża tak mocno pamięci wspólnej jak wariant z wykonaniem kopii całej tablicy wektorów masek.

Struktury danych znajdujące się w pamięci globalnej procesora GPU to:

- tablica n słów dla ciągu A,
- tablica $\sigma\lceil m/w_\mathrm{g}\rceil$ słów dla wektorów masek bitowych,

- tablica n słów dla przeniesień z górnej krawędzi,
- tablica $\lceil m/w_\mathrm{g}\rceil$ słów dla wektora bitowego z lewej krawędzi.

Sumaryczna zajętość pamięci globalnej jest wobec tego $\Theta(n+\sigma\lceil m/w_\mathrm{g}\rceil)$ słów.

Po wykonaniu obliczeń dla wszystkich pudełek do pamięci RAM transferowana jest zawartość prawej kolumny wszystkich pudełek z prawej krawędzi, dzięki czemu w procesorze CPU można wyznaczyć liczbę bitów o wartości 0, która jest wynikiem działania algorytmu.

Część kodu jądra dla tego algorytmu pokazana jest na rys. 7.16. Parametrami wywołania funkcji są:

- block_x, block_y – znaczenie podobne jak w algorytmie DP,
- g_A – wskaźnik do tablicy zawierającej ciąg A,
- g_masks – wskaźnik do tablicy wektorów masek bitowych,
- g_lr_b – wskaźnik do tablicy zawierającej części wektora V (prawa krawędź sąsiedniego z lewej strony pudełka),
- g_tb_b – wskaźnik do tablicy zawierającej przeniesienia powstałe w trakcie obliczania górnego sąsiedniego pudełka,
- max_block_x – dodatkowa wartość pomocna do stwierdzenia, czy bieżące pudełko jest ostatnie w poziomie; jeśli tak, to zamiast prawej krawędzi zapisywana jest w g_lr_b liczba zer w V, co przyspiesza wyznaczenie końcowego wyniku.

Przedstawiony kod zawiera dwa warianty (o wyborze decyduje obecność makrodefinicji SMALL_ALPH). Jeśli alfabet jest mały, to wszystkie wektory masek bitowych wczytywane są do pamięci wspólnej (wiersze 18–19), skąd są pobierane, kiedy jest to potrzebne (wiersz 32). W przeciwnym przypadku wektory masek bitowych są pobierane z pamięci globalnej do tablicy pośredniej p_masks (wiersze 35–37) i są stamtąd odczytywane (wiersze 37, 39).

Analiza teoretycznego przyspieszenia algorytmu proponowanego w stosunku do algorytmu sekwencyjnego dla procesora CPU jest podobna do analizy przeprowadzonej dla algorytmu programowania dynamicznego, z tą różnicą, że w_g wierszy przetwarzanych jest na raz, co daje całkowity czas wykonywanych obliczeń

$$\Theta\left(\frac{nm}{\min(n',m',\eta_1)\eta_2 w_\mathrm{g}}\right). \tag{7.20}$$

Dostęp do pamięci wspólnej w obu wariantach algorytmu równoległości bitowej odbywa się bez konfliktów banków, a więc nie pojawia się tu serializacja wątków. Czas dostępu do pamięci globalnej jest jednak znacznie większy niż dla algorytmu opartego na programowaniu dynamicznym. W obu wersjach algorytmu równoległości bitowej (ze wstępnym wczytaniem wektorów masek oraz z wczytywaniem ich do tablicy pośredniej) występuje następująca liczba połączonych dostępów do pamięci globalnej dla każdego pudełka: $\Theta(b_\mathrm{w}/\eta_2)$ (ładowanie części ciągu A oraz wczytywanie/zapisywanie górnej/dolnej krawędzi, $\Theta(b_\mathrm{h}/(\eta_2 w_\mathrm{g}))$ (wczytywanie-/zapisywanie lewej i prawej krawędzi).

```
1  __global__ void Kernel_LCS_BP(unsigned int block_x, unsigned int block_y,
2    unsigned int *g_A, unsigned int *g_masks,
3    unsigned int *g_lr_b, unsigned int *g_tb_b, int max_block_x)
4  {
5    __shared__ unsigned int sB[2*BLK_HEIGHT];
6    __shared__ unsigned int s_A[2*BLK_HEIGHT];
7  #ifdef SMALL_ALPH
8    __shared__ unsigned int p_masks[ALPH_SIZE][BLK_HEIGHT];
9  #else
10   __shared__ unsigned int p_masks[4][BLK_HEIGHT];
11 #endif
12   unsigned int V, V2, tB;
13   const unsigned int num_threads = blockDim.x;
14   const unsigned int tid = threadIdx.x;
15   uint4 *u4g_masks = (uint4 *) g_masks;
16   ...
17 #ifdef SMALL_ALPH
18   for(i = 0; i < ALPH_SIZE; ++i)
19     p_masks[i][tid] = g_masks[(i << SHIFT_MASKS) + orig_y + tid];
20 #endif
21   ...
22   for(; i < BLK_WIDTH; ) {
23     int i_max = i + num_threads;
24     load_A(s_A, g_A, orig_x, tid, i, orig_y);
25     load_sB(sB, g_tb_b, orig_x, tid, i);
26     __syncthreads();
27
28     for(; i < i_max; ++i) {
29       int n_i = (i - tid) & BLK_MASK;
30       unsigned int m;
31 #ifdef SMALL_ALPH
32       m = p_masks[s_A[n_i]][tid];
33 #else
34       if((tid & 3) == 0) {
35         uint4 mm = u4g_masks[s_A[n_i] + (tid >> 2)];
36         p_masks[n_i & 3][tid+1] = mm.y;    p_masks[n_i & 3][tid+2] = mm.z;
37         p_masks[n_i & 3][tid+3] = mm.w;    m = mm.x;
38       }
39       else    m = p_masks[n_i & 3][tid];
40 #endif
41       tB = V & m;           V2 = V + sB[n_i];     V2 += tB;
42       sB[n_i] = V2 < V;     V = (V2 | (V - tB));
43       __syncthreads();
44     }
45     g_tb_b[orig_x+tid+i-2*BLK_HEIGHT] = sB[(tid+i-2*BLK_HEIGHT) & BLK_MASK];
46   }
47   ...
48   if(max_block_x != block_x-blockIdx.x)  g_lr_b[orig_y + tid] = V;
49   else                                   g_lr_b[orig_y + tid] = __popc(~V);
50 }
```

Rys. 7.16. Część programu jądra dla algorytmu LCS-BP-CUDA rozwiązującego problem LCS metodą równoległości bitowej w procesorze GPU

Fig. 7.16. A part of the kernel code for the LCS BP algorithm (LCS-BP-CUDA) at GPU

Wersja ze wstępnym wczytaniem wektorów masek

W wersji ze wstępnym wczytaniem wektorów masek występuje konieczność wczytania całych wektorów masek dla części ciągu B, co wymaga $\Theta(\sigma b_{\mathrm{h}}/(\eta_2 w_{\mathrm{g}}))$ połączonych dostępów do pamięci. Ponieważ $\sigma \leq b_{\mathrm{w}}$ (jeśli to nie jest spełnione, to wystarczy wczytać te wektory masek, dla których występują symbole w odpowiedniej części ciągu A), całkowity czas dostępów do pamięci globalnej jest

$$\Theta\left(\frac{b_{\mathrm{w}}}{\eta_2} + \frac{b_{\mathrm{h}}}{\eta_2 w_{\mathrm{g}}} + \frac{b_{\mathrm{h}}\sigma}{\eta_2 w_{\mathrm{g}}}\right) n'm' = \Theta\left(\frac{b_{\mathrm{w}} w_{\mathrm{g}} + b_{\mathrm{h}}\sigma}{\eta_2 w_{\mathrm{g}}} n'm'\right) = \Theta\left(\frac{\max(nm'w_{\mathrm{g}}, n'm\sigma)}{\eta_2 w_{\mathrm{g}}}\right). \tag{7.21}$$

Złożoność czasowa tego algorytmu jest sumą (7.20) oraz (7.21):

$$\begin{aligned}
&\Theta\left(\frac{nm}{\min(n',m',\eta_1)\eta_2 w_{\mathrm{g}}} + \frac{\max(w_{\mathrm{g}} n \frac{m}{b_{\mathrm{h}}}, \sigma \frac{n}{b_{\mathrm{w}}} m)}{\eta_2 w_{\mathrm{g}}}\right) = \\
&\Theta\left(\frac{nm}{\eta_2 w_{\mathrm{g}}} \times \max\left(\frac{1}{\min(n',m',\eta_1)}, \frac{w_{\mathrm{g}}}{b_{\mathrm{h}}}, \frac{\sigma}{b_{\mathrm{w}}}\right)\right) = \\
&\Theta\left(\frac{nm}{w_{\mathrm{g}}} \times \frac{1}{\eta_2 \min\left(\min(n',m',\eta_1), \frac{b_{\mathrm{h}}}{w_{\mathrm{g}}}, \frac{b_{\mathrm{w}}}{\sigma}\right)}\right).
\end{aligned} \tag{7.22}$$

Złożoność czasowa wersji sekwencyjnej dla procesora CPU jest $\Theta(nm/w_{\mathrm{c}})$, a więc:

Wniosek 7.3. *Przyspieszenie wersji ze wstępnym wczytaniem wektorów masek w stosunku do wersji sekwencyjnej jest*

$$\Theta\left(\frac{w_{\mathrm{g}}}{w_{\mathrm{c}}} \eta_2 \min\left(\eta_1, \frac{n}{b_{\mathrm{w}}}, \frac{m}{b_{\mathrm{h}}}, \frac{b_{\mathrm{h}}}{w_{\mathrm{g}}}, \frac{b_{\mathrm{w}}}{\sigma}\right)\right). \tag{7.23}$$

Wersja z tablicą pośredniczącą

W wersji z tablicą pośredniczącą występuje $\Theta(b_{\mathrm{w}} b_{\mathrm{h}}/(\eta_2 w_{\mathrm{g}}))$ dostępów do pamięci globalnej dla każdego pudełka w celu wczytania wektorów masek.[8] Wobec tego całkowity czas dostępów do pamięci globalnej w tym algorytmie to

$$\begin{aligned}
&\Theta\left(\left(\frac{b_{\mathrm{w}}}{\eta_2} + \frac{b_{\mathrm{h}}}{\eta_2 w_{\mathrm{g}}} + \frac{b_{\mathrm{w}} b_{\mathrm{h}}}{\eta_2 w_{\mathrm{g}}}\right) n'm'\right) = \Theta\left(\frac{b_{\mathrm{w}} w_{\mathrm{g}} + b_{\mathrm{w}} b_{\mathrm{h}}}{\eta_2 w_{\mathrm{g}}} n'm'\right) = \\
&\Theta\left(\frac{b_{\mathrm{w}} b_{\mathrm{h}}}{\eta_2 w_{\mathrm{g}}} \times \frac{n}{b_{\mathrm{w}}} \times \frac{m}{b_{\mathrm{h}}}\right) = \Theta\left(\frac{nm}{\eta_2 w_{\mathrm{g}}}\right).
\end{aligned} \tag{7.24}$$

Złożoność czasowa tego algorytmu jest sumą (7.20) oraz (7.24):

$$\Theta\left(\frac{nm}{\min(n',m',\eta_1)\eta_2 w_{\mathrm{g}}} + \frac{nm}{\eta_2 w_{\mathrm{g}}}\right) = \Theta\left(\frac{nm}{\eta_2 w_{\mathrm{g}}}\right). \tag{7.25}$$

[8] Przyjmuje się tu, że każdy dostęp połączony obejmuje $\Theta(\eta_2)$ słów.

Wniosek 7.4. *Przyspieszenie wersji z tablicą pośredniczącą w stosunku do wersji sekwencyjnej dla procesora CPU jest*

$$\Theta\left(\frac{\eta_2 w_g}{w_c}\right). \tag{7.26}$$

7.6.5. Wyniki eksperymentalne

Dla oceny zaproponowanych algorytmów równoległych przeprowadzone zostały eksperymenty z użyciem następującego zestawu komputerowego:

- CPU: procesor AMD Phenom II X4 810 taktowany zegarem 2600 MHz z 4 MB pamięci podręcznej trzeciego poziomu, 4096 MB RAM (taktowanej 1033 MHz),
- GPU: procesor NVidia GTX 260 taktowany następująco: 696 MHz (jądro), 1501 MHz (multiprocesory), 896 MB pamięci globalnej (taktowanej 1100 MHz) z 27 multiprocesorami (każdy multiprocesor zawiera 8 rdzeni).

Zestaw komputerowy został wybrany w taki sposób, aby wartości rynkowe procesorów CPU i GPU były podobne (jeśli wziąć pod uwagę, że karta graficzna zwiera płytę oraz pamięć globalną, to w zasadzie powinno się uwzględnić sumaryczny koszt procesora CPU, płyty głównej oraz pamięci RAM, który jest około dwukrotnie większy niż koszt samego CPU).

Trudno jest zdefiniować jeden sposób porównania wyników uzyskiwanych dzięki zrównolegleniu, który byłby akceptowalny przez wszystkich, ponieważ istnieje kilka alternatywnych możliwości, m.in.

- algorytm sekwencyjny dla procesora CPU i algorytm równoległy dla procesora GPU przy podobnej cenie sprzętu – pokazuje jak dobrze algorytmy dla danego problemu zrównoleglają się dla procesora GPU,
- algorytm równoległy dla wielordzeniowego procesora CPU i algorytm równoległy dla procesora GPU przy podobnej cenie sprzętu – pokazuje, dzięki której architekturze (wiele rdzeni w procesorze CPU czy procesorze GPU) możliwe jest uzyskanie lepszych przyspieszeń,
- algorytm sekwencyjny dla procesora CPU i algorytm hybrydowy dla CPU+GPU używający zarówno dostępnych rdzeni procesora CPU, jak i multiprocesorów GPU – pokazuje, jak wiele można zyskać dzięki zastosowaniu procesora GPU jako akceleratora obliczeń prowadzonych z użyciem pełnej mocy obliczeniowej procesora CPU.

Przeprowadzone eksperymenty pozwalają ocenić przyspieszenie dla dwu pierwszych przypadków. Implementacja algorytmów hybrydowych (trzeci przypadek) jest zwykle trudna, ponieważ architektury procesorów CPU oraz GPU różnią się w sposób zasadniczy, a przesyły danych oraz synchronizacja pomiędzy CPU i GPU są stosunkowo wolne. Co więcej, zwykle algorytmy dla procesorów GPU działają znacznie szybciej niż równoległe algorytmy dla procesorów CPU i potencjalny zysk ze stworzenia algorytmu hybrydowego nie jest duży. Dlatego

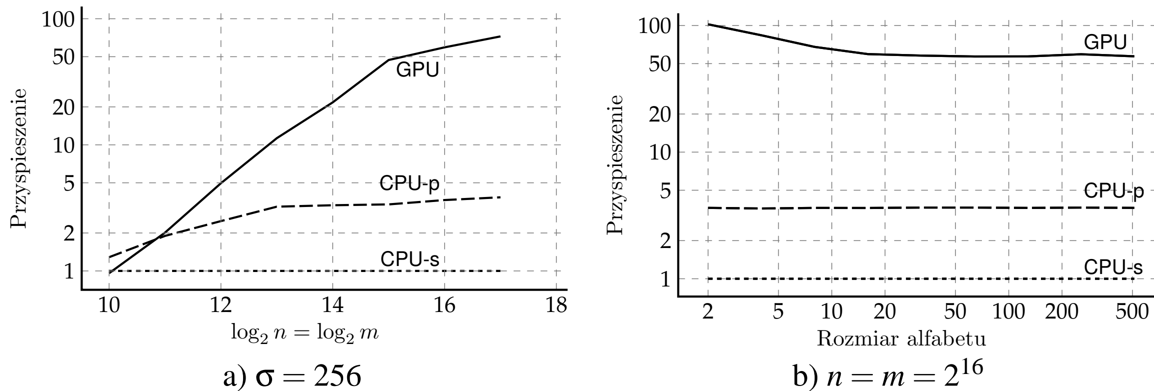
też, na podstawie uzyskanych wyników, jedynie dyskutowane są potencjalne korzyści z użycia algorytmów hybrydowych. Równoległe algorytmy dla procesorów CPU zostały zaprojektowane w podobny sposób jak algorytmy dla procesorów GPU, tzn. obliczenia wykonywane są w pudełkach przetwarzanych zgodnie z zasadą drugiej przekątnej.

Badane algorytmy zostały zaimplementowane w języku C++ z użyciem biblioteki CUDA 2.3. Do kompilacji użyto MS Visual C++ 2008 z maksymalną optymalizacją pod kątem prędkości. Ciągi testowe zostały wygenerowane z użyciem generatora liczb pseudolosowych o rozkładzie równomiernym. Wszystkie czasy są medianami ze 101 wykonań. Zamiast czasów absolutnych na wykresach pokazane jest przyspieszenie w stosunku do odpowiednich algorytmów sekwencyjnych dla procesorów CPU. Algorytmy na wykresach oznaczane są następująco:

- CPU-s – algorytm sekwencyjny dla procesorów CPU,
- CPU-p – algorytm równoległy dla procesorów CPU (zastosowany procesor CPU zawiera 4 rdzenie, jednak wstępne eksperymenty pokazały, że lepiej jest uruchamiać na raz 32 wątki),
- GPU – algorytm równoległy, oparty na programowaniu dynamicznym, dla procesorów GPU,
- prefetch – algorytm równoległy, oparty na metodzie równoległości bitowej, dla procesorów GPU ze wstępnym wczytaniem wektorów masek,
- preload – algorytm równoległy, oparty na metodzie równoległości bitowej, dla procesorów GPU z tablicą pośredniczącą.

Algorytm oparty na programowaniu dynamicznym

W pierwszym eksperymencie badano algorytm oparty na metodzie programowania dynamicznego (rys. 7.15). Ponieważ algorytm sekwencyjny równoległości bitowej dla problemu LCS jest około 50 razy szybszy niż klasyczny algorytm programowania dynamicznego (dla $w = 64$), eksperyment ten przeprowadzony został głównie w celu sprawdzenia potencjalnych możliwości przyspieszenia wyznaczania macierzy programowania dynamicznego za pomocą metody drugiej przekątnej, w której wykonywanych jest wiele wywołań kodu jądra. Na rys. 7.17a można zaobserwować, że aby korzyści z użycia procesora GPU były znaczące, ciągi powinny być odpowiednio długie. Dla krótkich ciągów liczba pudełek obliczanych równolegle jest niewielka i niektóre multiprocesory są nieobciążone. Co więcej, dla dłuższych ciągów rozmiary pudełek mogą być większe, dzięki czemu czas obliczania pojedynczego pudełka jest dłuższy i relatywny koszt kolejnych uruchomień kodu jądra jest mniejszy. Przyspieszenie dla algorytmu GPU osiąga wartość 60–70 dla $n = m = 10^5$, podczas gdy przyspieszenie dla algorytmu CPU-p – wartość 3,8 (bliską teoretycznej wartości 4), czyli ponaddziesięciokrotnie mniej. Tak więc ewentualny algorytm hybrydowy (CPU+GPU) nie działałby znacząco szybciej niż GPU. Rozmiary pudełek dobrano, w trakcie wstępnych eksperymentów, w zależności



Rys. 7.17. Porównanie przyspieszenia równoległego algorytmu wyznaczania LCS dla procesorów GPU opartego na programowaniu dynamicznym

Fig. 7.17. Comparison of the speedup of the GPU parallel algorithm based on the classical LCS dynamic programming algorithm

od długości ciągów następująco:

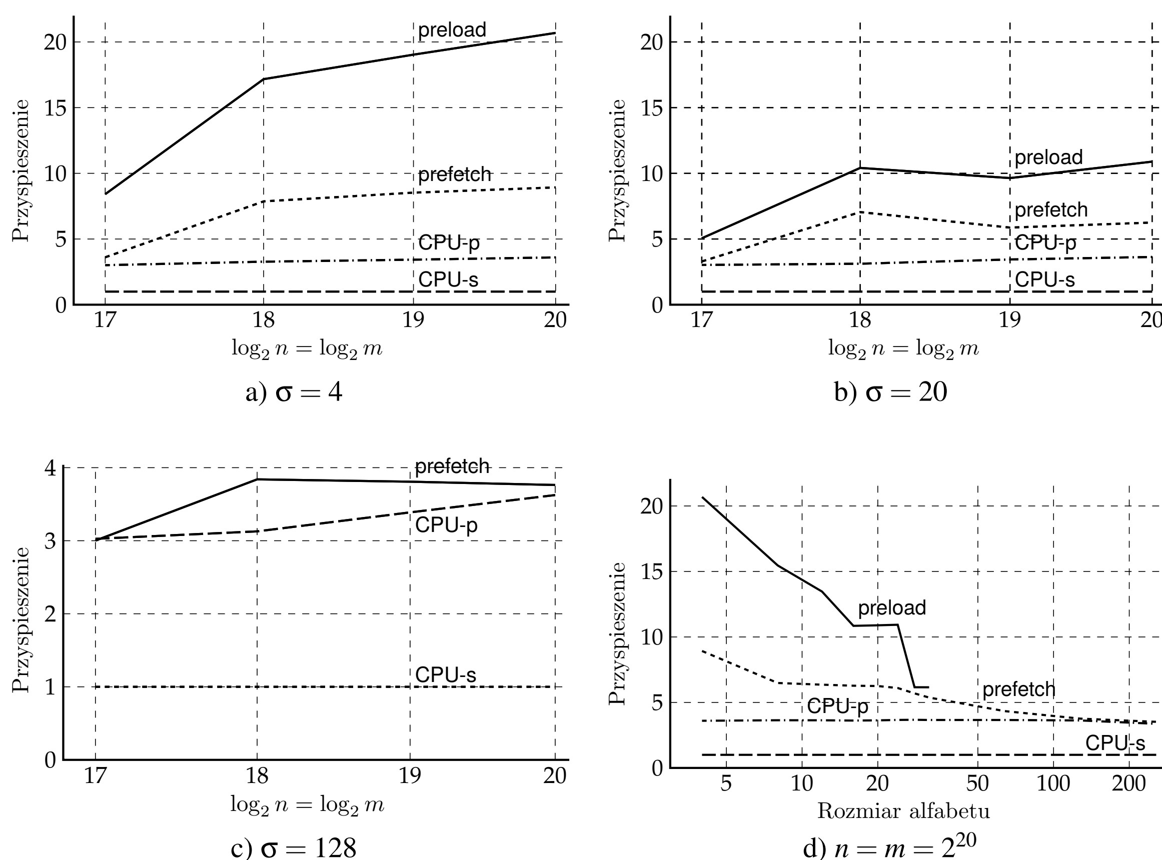
$$b_{\mathrm{w}} = \max\left(\frac{n}{2^7}, 2^6\right), \qquad b_{\mathrm{h}} = \min\left(\max\left(\frac{m}{2^8}, 2^5\right), 2^7\right), \qquad b_{\mathrm{w}}^{\mathrm{cpu}}, b_{\mathrm{h}}^{\mathrm{cpu}} \in [2^8, 2^{11}],$$

gdzie $b_{\mathrm{w}}^{\mathrm{cpu}}$ oraz $b_{\mathrm{h}}^{\mathrm{cpu}}$ są odpowiednio długością oraz szerokością pudełka w równoległej implementacji dla procesorów CPU. Ustalenia powyższe gwarantują na tyle dużą liczbę pudełek, że multiprocesory są w miarę równomiernie obciążone, a pudełka nie są zbyt małe (a także nie są zbyt duże z powodu ograniczeń architektury CUDA), co redukuje koszty wielu uruchomień jądra.

Na rys. 7.17b można zauważyć, że dla $n = m = 2^{16}$, przyspieszenia dla dostatecznie dużych alfabetów ($\sigma \geq 20$) są zbliżone do siebie i wynoszą ok. 65. Dla mniejszych alfabetów przyspieszenia są jeszcze większe, z uwagi na stosukowo dużą liczbę dopasowań, co powoduje częstsze wejście do wnętrza instrukcji warunkowej w kodzie dla procesora CPU. Dla procesora GPU spowolnienie spowodowane dużą liczbą dopasowań jest znikome.

Algorytmy równoległości bitowej

W drugim eksperymencie oceniano algorytmy równoległości bitowej (rys. 7.18). Ponieważ algorytm BP dla problemu LCS jest zwykle najszybszym sposobem wyznaczania podciągu LCS w procesorze CPU, uzyskane wyniki mają znacznie większe znaczenie praktyczne niż wyniki dla algorytmu programowania dynamicznego. Eksperymenty przeprowadzono dla znacznie dłuższych ciągów, ponieważ algorytmy równoległości bitowej w procesorach GPU przetwarzają naraz w_{g} wierszy z odpowiedniej macierzy algorytmu DP. Długości ciągów wejściowych ustalono na $n = m = 2^{17}, 2^{18}, 2^{19}, 2^{20}$ dla trzech rozmiarów alfabetu: $\sigma = 4, 20, 128$ (rysunki 7.18a–c). Dla małych alfabetów zbadano oba warianty algorytmu równoległości bitowej dla procesorów GPU. Rozmiary pudełek dobrano, w trakcie wstępnych eksperymentów,



Rys. 7.18. Porównanie przyspieszenia algorytmu równoległego wyznaczania długości podciągu LCS dla procesorów GPU opartego na równoległości bitowej

Fig. 7.18. Comparison of the speedup of the LCS-length-computing GPU parallel algorithm based on the bit-parallel algorithm

w zależności od długości ciągów następująco:

$$b_{\mathrm{w}} = \max\left(\frac{n}{2^{10}}, 2^8\right), \quad b_{\mathrm{h}} = \min\left(\max\left(\frac{m}{2^7}, 2^{10}\right), 2^{11}\right),$$
$$b_{\mathrm{w}}^{\mathrm{cpu}} \in [2^{13}, 2^{14}], \quad b_{\mathrm{h}}^{\mathrm{cpu}} \in [2^{14}, 2^{15}].$$

Dla małych alfabetów algorytm preload okazał się ponad dwa razy szybszy niż algorytm prefetch. Wskazuje to, że w algorytmie prefetch nie było możliwe ukrycie w pełni wysokich kosztów dostępu do pamięci globalnej. Dla $\sigma = 4$ przyspieszenie algorytmu preload w stosunku do CPU-s wynosi około 20 i rośnie wraz ze wzrostem długości ciągów. Dla $\sigma = 20$ przyspieszenie jest mniejsze, głównie z powodu częstszych chybień w pamięci podręcznej (wektory masek znajdują się w pamięci wspólnej, ale częściej są wymiatane ze związanej z nią pamięci podręcznej). Co więcej, algorytm preload wymaga znacznie więcej pamięci wspólnej na wektory masek i mniej pudełek może być równocześnie obliczanych w tym samym multiprocesorze, co utrudnia ukrywanie kosztów dostępu do pamięci globalnej. Tym niemniej, przyspieszenia są i tak znacznie większe niż dla $\sigma = 128$, gdzie mógł zostać użyty tylko algorytm prefetch. Wartości przyspieszeń około 4 nie są imponujące, jeśli weźmie się pod uwagę, że procesor GPU

dysponuje 27 multiprocesorami (216 rdzeniami), ale należy pamiętać, że zegar procesora GPU jest około dwukrotnie wolniejszy niż zegar procesora CPU, a także długość słowa komputerowego procesora GPU jest dwa razy mniejsza niż długość słowa komputerowego procesora CPU. Nawet jednak mimo to można zaobserwować, że koszt dostępów do pamięci globalnej jest duży, bo uzyskiwane przyspieszenia są umiarkowane. Przyspieszenia algorytmu CPU-p są bliskie wartości teoretycznej, ale są one porównywalne z przyspieszeniami algorytmów dla procesora GPU tylko dla dużych alfabetów.

Na rys. 7.18d można zaobserwować, jak przyspieszenie maleje wraz ze wzrostem rozmiaru alfabetu. Dla dużych alfabetów czasy działania algorytmów prefetch oraz CPU-p są zbliżone, co powoduje, że uzasadniona może być w takiej sytuacji decyzja o stworzeniu algorytmu hybrydowego (CPU+GPU). Niestety, ilość danych, które muszą być wymieniane między procesorami CPU a GPU jest duża, przez co stworzenie wydajnego algorytmu hybrydowego istotnie szybszego od algorytmów prefetch oraz CPU-p jest trudne.

7.7. Podsumowanie

W literaturze można znaleźć wiele prac poświęconych problemowi LCS, co wynika m.in. z jego istotnych zastosowań praktycznych. W niniejszym rozdziale omówiono tylko najważniejsze z istniejących algorytmów, które wyróżniają się prostotą (programowanie dynamiczne), dużą szybkością (algorytmy równoległości bitowej, Hunta–Szymanskiego), małą złożonością pamięciową (algorytm Hirschberga), czy też pokazują związek tego problemu z problemem LIS.

Oprócz wielu algorytmów sekwencyjnych dla problemu LCS istnieją także algorytmy równoległe. Z uwagi na dużą różnorodność równoległych modeli obliczeniowych wymieniono tylko niektóre z nich, a skupiono się na algorytmach dla procesorów graficznych (GPU). Dziedzina wykorzystywania procesorów GPU do obliczeń ogólnego przeznaczenia w ostatnich latach bardzo intensywnie się rozwija. W niniejszej pracy zaproponowano (podrozdz. 7.6) dwa algorytmy wyznaczania długości podciągu LCS dla procesorów GPU. Pierwszy z tych algorytmów opiera się na metodzie programowania dynamicznego. Z uwagi na dużą prostotę algorytmu sekwencyjnego oraz niewielkie ilości danych wymienianych z pamięcią globalną procesora GPU przyspieszenia uzyskane dla tego algorytmu w stosunku do wersji sekwencyjnej wyniosły ok. 60–70. Drugim algorytmem zrównoleglonym dla procesora GPU był algorytm równoległości bitowej. Charakteryzuje się on znacznie większą ilością danych wymienianych z pamięcią globalną procesora GPU, przez co osiągane przyspieszenia nie były aż tak dobre i zwykle należały do przedziału 5–20. Dla obu tych proponowanych algorytmów przeprowadzono także analizę złożoności czasowej i pamięciowej, dzięki której było możliwe wyznaczenie dla nich teoretycznych wartości przyspieszenia.

Oprócz podstawowej wersji problemu LCS sporą popularnością cieszą się w literaturze różne jego modyfikacje. W kolejnych rozdziałach poświęcono większą uwagę wybranym wariantom, a poniżej jedynie wymieniono z nazwy niektóre inne, interesujące, a nieomawiane dalej. Efektywny algorytm wyznaczania wszystkich podciągów LCS podał Rick w [180]. W problemie wyznaczania najdłuższego wspólnego rosnącego podciągu (ang. *longest common increasing subsequence*) oczekiwanym wynikiem jest najdłuższy podciąg dwu lub więcej ciągów, którego wyrazy uporządkowane są rosnąco (ew. niemalejąco) [221, 184, 36, 39]. W problemie najdłuższego mozaikowego wspólnego podciągu (ang. *mosaic longest common subsequence*) [117] danymi wejściowymi są ciąg A, zbiór ciągów $\mathcal{Z} = \{Z^1, Z^2, \ldots Z^N\}$ oraz liczba k. Celem jest znalezienie takiej konkatenacji ciągów $B = B^1 B^2 \ldots B^k$, gdzie $B^i \in \mathcal{Z}$ dla $1 \leq i \leq k$, aby długość podciągu LCS dla A oraz B była maksymalna. Kolejnymi wariantami są problemy, w których jeden z ciągów jest traktowany cyklicznie i/lub może być odczytywany z obu kierunków [165].

8. Najdłuższy wspólny podciąg niezmienniczy względem transpozycji

8.1. Wprowadzenie

Jedną z dziedzin, w której porównywanie ciągów za pomocą ich podciągu LCS nie sprawdza się zbyt dobrze, jest przetwarzanie danych muzycznych (ang. *musical information retrieval*, MIR) [168, 97, 154, 205]. Danymi w MIR są często melodie zapisane w formacie MIDI [181], w którym każdy dźwięk opisany jest przez parę: wysokość, czas trwania. Teoretycznie w celu sprawdzenia podobieństwa dwu utworów możliwe jest (podobnie jak dla tekstów) wyznaczenie ich podciągu LCS i porównanie go z długościami ciągów wejściowych. Praktyka pokazała jednak, że percepcja muzyki i tekstu przez ludzi bardzo się różni. Po pierwsze, aby wyznaczyć podobieństwo dwu utworów, często można zignorować czasy trwania dźwięków, bo ludzkie ucho mimo różnic w nich będzie w stanie zauważyć podobieństwo pomiędzy utworami. Po drugie, ludzie rozpoznają utwory, nawet jeśli zostaną one zagrane w innej tonacji, czyli w praktyce wszystkie parami odpowiadające sobie dźwięki różnią się o pewną stałą wartość. Z powyższych powodów wprowadzony został wariant miary LCS, który jest niezmienniczy względem transpozycji i w którym analiza podobieństwa ciągów dotyczy tylko wysokości dźwięków [49, 140, 141]. Problem *najdłuższego niezmienniczego względem transpozycji wspólnego podciągu* (ang. *longest common transposition-invariant subsequence*, LCTS) znajduje także zastosowanie przy porównywaniu szeregów czasowych, obrazów oraz w innych dziedzinach [163].

Definicja 8.1. *Podciągiem transponowanym o* $t \in \mathbb{Z}$ *ciągu* $A = a_1 a_2 \dots a_n$ *jest ciąg* $A + t = (a_1 + t)(a_2 + t) \dots (a_n + t)$.

Problem 8.1 (Najdłuższy niezmienniczy względem transpozycji wspólny podciąg, LCTS)**.** *Dla ciągów* $A = a_1 a_2 \dots a_n$ *oraz* $B = b_1 b_2 \dots b_m$*, których symbole należą do alfabetu* $\Sigma = \{0, \dots, \sigma - 1\}$*, znaleźć najdłuższy spośród wspólnych podciągów dla wszystkich par* $A + t$ *oraz* B*, dla dowolnego* t*.*

Przykład 8.1 (Najdłuższy wspólny podciąg, LCS)**.** *Dla ciągów* $A = 1\ \underline{2}\ \underline{1}\ \underline{1}\ \underline{4}\ 1\ \underline{3}\ \underline{2}\ 1\ \underline{1}\ 2\ 3$ *oraz* $B = 4\ \underline{3}\ 4\ 3\ 5\ \underline{2}\ \underline{2}\ \underline{5}\ \underline{4}\ 5\ \underline{3}\ \underline{2}$ *najdłuższym niezmienniczym względem transpozycji wspólnym podciągiem jest* $S = 2\ 1\ 1\ 4\ 3\ 2\ 1$*. W ciągach A oraz B podkreślono symbole tworzące podciąg LCTS. W tym przypadku* $t = 1$*.*

Najprostszym sposobem znalezienia podciągu LCTS jest wykonanie dowolnego algorytmu wyznaczającego podciąg LCS dla każdej z $2\sigma - 1$ transpozycji t (istotne wartości t na-

leżą do przedziału $[-\sigma+1,\sigma-1]$) i wzięcie najdłuższego z otrzymanych wyników. Złożoność czasowa w takim przypadku jest $O(\sigma)$ razy większa niż złożoność czasowa użytego algorytmu dla problemu LCS. W szczególności, zastosowanie algorytmu programowania dynamicznego (podrozdz. 7.3.1) skutkuje złożonością czasową $O(nm\sigma)$. Już jednak użycie algorytmu równoległości bitowej (podrozdz. 7.3.4) pozwala na osiągnięcie złożoności czasowej $O(n\lceil m/w\rceil\sigma)$.

Istnieją także algorytmy specjalizowane dla problemu LCTS. W pracy [139] zaproponowano algorytm oparty na binarnej metodzie podziału i ograniczeń (ang. *branch and bound*), którego złożoność czasowa jest $O((nm+\log\sigma)\sigma)$ w przypadku pesymistycznym, a więc gorzej niż algorytmu opartego na programowaniu dynamicznym. Przypadek pesymistyczny jest jednak bardzo mało prawdopodobny, a w przypadku optymistycznym złożoność czasowa tego algorytmu jest $O((nm+\log\log\sigma)\log\sigma)$. W tej samej pracy zaproponowano także algorytm oparty na k-arnej metodzie podziału i ograniczeń, którego złożoność czasowa jest $O((nm+\log(\sigma k/(k-1)))\sigma k/(k-1))$ w przypadku pesymistycznym oraz $O((nm+\log(k\log_k\sigma))k\log_k\sigma)$ w przypadku optymistycznym. Eksperymenty praktyczne pokazały, że algorytm ten jest najszybszy dla $k=3$. Algorytm wykorzystujący rzadkość macierzy programowania dynamicznego [150] pozwala osiągnąć złożoność czasową $O(nm\log m)$ dla przypadku pesymistycznego. Wariant algorytmu Crochemore'a i in. [52] pozwala osiągnąć złożoność czasową $O(nm\lceil\sigma/w\rceil)$. Algorytm Grabowskiego i Navarro [104] działa w czasie $O(nm\log\sigma)$.

8.2. Algorytm pudełkowy

W niniejszym podrozdziale zaprezentowany zostanie algorytm rozwiązywania problemu LCTS zaproponowany przez autora w [163]. W podrozdz. 7.3.2 przedstawiono algorytm Hunta–Szymanskiego dla problemu LCS. Może on zostać wykonany niezależnie dla każdej transpozycji, a ponieważ łącznie dopasowań dla wszystkich transpozycji jest nm, da to złożoność czasową $O(n\sigma+nm\log\log m)$. Wynik ten można poprawić dzieląc macierz DP na prostokątne obszary, pudełka, o precyzyjnie dobranym rozmiarze $e\times e$ komórek. Dla ustalenia uwagi rozważmy podmacierz o indeksach (i^*,j^*), dla pewnej transpozycji t, reprezentującą komórki $M(ei^*+1..e(i^*+1),ej^*+1..e(j^*+1))$. Danymi wejściowymi dla algorytmu wyznaczania każdej podmacierzy są wartości komórek będących lewymi i górnymi sąsiadami tej podmacierzy, tj. $M(ei^*,ej^*..e(j^*+1))$ oraz $M(ei^*..e(i^*+1),ej^*)$. Na podstawie wartości w komórkach z górnego wiersza wejściowego tworzona jest struktura Q. Dla każdej wartości występującej w $M(ei^*..e(i^*+1),ej^*)$ do Q wpisywany jest minimalny indeks kolumny zawierającej tę wartość w $M(ei^*..e(i^*+1),ej^*)$. Algorytm przetwarza następnie wszystkie dopasowania wiersz po wierszu od góry do dołu, a w każdym wierszu od prawej do lewej strony w sposób analogicz-

LCTS-NGMD-One-Box(A, B, i^*, j^*, e, M)

Wejście: *A, B – ciągi, dla których wyznaczany jest podciąg LCTS*
i^*, j^* – indeksy pudełka w macierzy programowania dynamicznego
e – rozmiar pudełka
M – macierz programowania dynamicznego
Wyjście: *M – macierz programowania dynamicznego z wyznaczonymi wartościami danego pudełka*

```
   {Przetwarzanie wstępne}
 1 Q ← Puste drzewo van Emde Boasa
 2 for t ← −σ+1 to σ−1 do
 3     Q[t].insert(⟨ei*, M(ei*, ej*)⟩)
 4     for i ← ei*+1 to e(i*+1) do
 5         if M(t, i, ej*) > M(t, i−1, ej*) then
 6             Q.insert(⟨i, M(t, i, ej*)⟩)
   {Obliczenia właściwe}
 7 for j ← ej*+1 to e(j*+1) do
 8     for all dopasowania (i, j) w wierszu j od największego indeksu kolumny do
 9     for i ← (e+1)i* downto ei*+1 do
10         t ← b_j − a_i
11         ⟨i′, h⟩ ← Q[t].successor(i−1)
12         Q[t].remove(⟨i′, h⟩);   Q[t].insert(⟨i, h⟩)
13     for t ← −σ+1 to σ−1 do
14         ⟨i′, h⟩ ← Q[t].minimum()
15         if M(t, ei*, j) ≥ r and i′ > ei* then
16             Q[t].remove(⟨ei*, h⟩);   Q[t].insert(⟨ei*, h⟩)
17         ⟨i′, h⟩ ← Q[t].maximum()
18         M(t, (e+1)i*, j) ← h
   {Wypełnienie dolnej krawędzi macierzy}
19 for t ← −σ+1 to σ−1 do
20     for i ← ei* to e(i*+1) do
21         ⟨i′, h⟩ ← Q[t].predecessor(i+1)
22         M(t, i, e(j*+1)) ← h
```

Rys. 8.1. Algorytm wyznaczający jedno pudełko macierzy programowania dynamicznego dla algorytmu z rys. 8.2
Fig. 8.1. An algorithm computing one box for the algorithm given at Fig. 8.2

ny do algorytmu Hunta–Szymanskiego dla problemu LCS. Dokładniejszy opis działania tego algorytmu można znaleźć w [163].

Pseudokod na rys. 8.1 przedstawia algorytm LCTS-NGMD-One-Box wyznaczający jeden fragment macierzy M. W wierszu 9. obliczana jest transpozycja, dla której bieżąca para indeksów (i, j) stanowi dopasowanie. Jako strukturę Q dogodnie wybrać jest drzewo van Emde Boasa [209, 210], gdyż możliwe jest dzięki temu wykonywanie wszystkich potrzebnych na niej operacji w czasie $O(\log\log e)$. Złożoność czasowa tego algorytmu jest: $O(\sigma e \log\log e)$ na etapie przetwarzania wstępnego, $O(\sigma e \log\log e)$ na etapie przetwarzania końcowego oraz $O(e^2 \log\log e)$ w przetwarzaniu właściwym, w związku z czym:

```
LCTS-NGMD(A, B, σ, e)
Wejście: A, B – ciągi, dla których wyznaczany jest podciąg LCTS
         e – rozmiar pudełka
         σ – rozmiar alfabetu
Wyjście: długość podciągu LCTS

   {Inicjalizacja}
 1 for t ← −σ+1 to σ−1 do
 2     for i ← 0 to n do M(t,i,0) ← 0
 3     for j ← 0 to m do M(t,0,j) ← 0
   {Obliczenia właściwe}
 4 for i* ← 0 to ⌈n/e⌉ − 1 do
 5     for j* ← 0 to ⌈m/e⌉ − 1 do
 6         LCTS-NGMD-One-Box(A, B, i*, j*, e, M)
   {Wyznaczanie wyniku}
 7 ℓ ← 0
 8 for t ← −σ+1 to σ−1 do
 9     ℓ ← max(ℓ, M(t,n,m))
10 return ℓ
```

Rys. 8.2. Algorytm wyznaczający długość podciągu LCTS metodą kolejnych pudełek
Fig. 8.2. Box-based algorithm solving the LCTS problem

Wniosek 8.1. *Złożoność czasowa algorytmu* LCTS-NGMD-One-Box *jest*

$$O((\sigma e + e^2)\log\log\sigma). \tag{8.1}$$

Kompletny algorytm LCTS-NGMD wyznaczania długości podciągu LCTS, stosujący algorytm LCTS-NGMD-One-Box obliczający jedno pudełko, przedstawiony jest na rys. 8.2. Po inicjalizacji, w której wypełniana jest górna i lewa krawędź macierzy, następuje wyznaczanie kolejnych fragmentów. Złożoność czasowa całego algorytmu wyznaczania długości podciągu LCTS jest

$$O\left((n+m)\sigma + \left\lceil\frac{n}{e}\right\rceil \left\lceil\frac{m}{e}\right\rceil (\sigma e + e^2)\log\log\sigma\right). \tag{8.2}$$

Wartość ta jest minimalna dla $e = \Theta(\sigma)$.

Wniosek 8.2. *Złożoność czasowa algorytmu* LCTS-NGMD *wyznaczającego długość podciągu LCTS jest*

$$O(n\sigma + nm\log\log\sigma). \tag{8.3}$$

Wniosek 8.3. *Złożoność pamięciowa algorytmu* LCTS-NGMD *jest*

$$O(n\sigma + \sigma^2) \text{ słów}. \tag{8.4}$$

8.3. Algorytm oparty na metodzie Hunta–Szymanskiego

8.3.1. Idea podstawowa

Niniejszy podrozdział zawiera omówienie oryginalnego algorytmu wyznaczania długości podciągu LCTS, który został zaproponowany przez autora w [63]. Podstawową jego ideą jest wyznaczanie długości podciągu LCS dla każdej transpozycji z wykorzystaniem rzadkości macierzy programowania dynamicznego. W szczególności można zauważyć, że dowolna para indeksów (i, j) jest dopasowaniem dla dokładnie jednej z możliwych transpozycji.

Na początku omówione zostanie działanie algorytmu dla pojedynczej transpozycji, a w dalszym ciągu niniejszego podrozdziału pokazane zostanie, jak na jego bazie uzyskać szybki algorytm dla problemu LCTS. Bez utraty ogólności, a dla uproszczenia opisu, przyjęte zostanie założenie, że rozpatrywaną transpozycją jest $t = 0$. Aby znaleźć dopasowanie (i', j') o największej randze h, we fragmencie macierzy programowania dynamicznego $M(1..i-1, 1..j-1)$ wystarczy znać dla każdej z występujących rang dopasowanie o najmniejszym indeksie kolumny. W tym celu dopasowania takie będą przechowywane w strukturze Q w postaci trójek: $\langle i', j', h\rangle$. Łatwo można zauważyć, że takie dopasowania są *dominujące*. Co więcej, indeksy kolumn w tej strukturze rosną wraz ze wzrostem rang. Na razie przyjęte będzie tylko, że struktura Q pozwala na wykonywanie operacji wstawiania, usuwania, wyszukiwania elementu i wyszukiwania następnika. Parametrem trzech ostatnich z tych operacji jest numer kolumny. Dyskusja możliwych do zastosowania struktur danych i wynikającej z tego złożoności czasowej zostanie przeprowadzona pod koniec niniejszego podrozdziału.

Przetwarzanie j-tego wiersza macierzy programowania dynamicznego można streścić następująco. Dla kolejnych dopasowań (i, j) o rosnących indeksach kolumn wykonuje się:

1. Jeśli w Q znajduje się dopasowanie o indeksie kolumny i, to nic więcej nie jest robione.
2. W przeciwnym przypadku wyszukiwane jest w Q dopasowanie o najmniejszym indeksie kolumny $i' > i$, a następnie:
   a) Dopasowanie o indeksie kolumny i' jest usuwane z Q, a dopasowanie (i, j) jest do Q wstawiane z taką samą rangą jak dopasowanie właśnie usunięte.
   b) Pomijane są w bieżącym wierszu wszystkie dopasowania o indeksach kolumn nie większych niż i'.
3. Jeśli nie istnieje dopasowanie w Q o indeksie kolumny większym niż i, to dopasowanie (i, j) wstawiane jest do Q z rangą o jeden większą niż maksymalna ranga dopasowania w Q.

Lemat 8.1. *Po przetworzeniu każdego j-tego wiersza macierzy programowania dynamicznego, zgodnie z powyżej opisaną metodą, w strukturze Q znajdują się dopasowania o wszystkich istniejących rangach we fragmencie macierzy DP: $M(1..n, 1..j)$, przy czym dla każdej rangi znajdujące się w Q dopasowanie ma najmniejszy możliwy indeks kolumny.*

Dowód. Na początku działania algorytmu struktura Q jest pusta, a więc teza jest w oczywisty sposób prawdziwa.

Niech teraz teza będzie prawdziwa dla $j-1$, gdzie $j \geq 1$. Ponieważ problemem rozwiązywanym dla pojedynczej transpozycji jest w zasadzie problem LCS, więc przydatne będzie tu sformułowanie reguły wyznaczania wartości macierzy programowania dynamicznego z (7.2). Łatwo można zauważyć, że zaproponowany powyżej algorytm dla każdego dopasowania (i, j) z wiersza o indeksie j wyznacza jego rangę i jeśli dla dopasowania (i, j) rangi h indeks i jest większy niż indeks kolumny dopasowania tej samej rangi z Q, to wymienia w Q istniejące dopasowanie rangi h na (i, j). Co za tym idzie, po przetworzeniu j-tego wiersza w Q znajdują się dopasowania wszystkich występujących rang w $M(t, 1..n, 1..j)$ i są to dopasowania o najmniejszych indeksach kolumn. ■

Wniosek 8.4. *Po zakończeniu przetwarzania całej macierzy największa ranga z Q oznacza długość podciągu LCS.*

Zaproponowany algorytm jest wariantem metody Hunta–Szymanskiego (podrozdz. 7.3.2), w której dopasowania przetwarzane są w ramach wiersza od lewej do prawej strony, a nie odwrotnie jak w oryginalnym algorytmie Hunta–Szymanskiego. Zmiana kolejności przetwarzania powoduje, że dla części dopasowań zamiast wyznaczania następnika, wykonywane jest sprawdzanie, czy dopasowanie o danym indeksie kolumny znajduje się w Q. Taka modyfikacja daje pewne korzyści w przypadku algorytmu wyznaczania podciągu LCTS.

Na rys. 8.3 zilustrowane jest działanie proponowanego algorytmu. Przykładowo, po przetworzeniu wiersza o indeksie 3 struktura Q zawiera następujące trójki: $\langle 1,2,1\rangle, \langle 3,3,2\rangle, \langle 12,3,3\rangle$. Odpowiadają one dopasowaniom o rangach 1, 2, 3 znajdujących się w kolumnach o najmniejszych indeksach. Następnie przetwarzany jest wiersz o indeksie 4. Pierwszym dopasowaniem jest w nim $(4,1)$, ale jest ono pomijane, ponieważ w Q znajduje się już dopasowanie o indeksie kolumny 1. Następne dopasowanie to $(4,6)$, które jest przetwarzane, ponieważ w Q brak dopasowania o indeksie kolumny 6. Wobec tego w Q wyszukiwane jest dopasowanie o najmniejszym indeksie kolumny większym niż 6 – wynikiem jest $\langle 12,3,3\rangle$. Jest ono usuwane z Q, a wstawiane jest $\langle 6,4,3\rangle$. Pozostałe dopasowania, $(4,6)$ oraz $(4,10)$, z bieżącego wiersza są pomijane, ponieważ indeks kolumny każdego z nich jest nie większy niż 12 (indeks kolumny właśnie usuniętego dopasowania).

Ponieważ rangi trójek z Q rosną o 1, więc wystarczy przechowywać pary $\langle i, j\rangle$ posortowane według indeksu kolumny i, a jeśli konieczne jest wyznaczenie rangi elementu, należy sprawdzić pozycję bieżącej pary w strukturze Q. Co więcej, indeksy wierszy są nieistotne, jeśli wynikiem ma być jedynie długość podciągu LCTS.

i \ j		1 A	2 B	3 C	4 C	5 B	6 A	7 A	8 B	9 D	10 A	11 B	12 C	
1	B		●			•			•			•		⟨2,1,1⟩
2	A	●					●	•			•			⟨1,2,1⟩ ⟨6,2,2⟩
3	C			●	•								●	⟨1,2,1⟩ ⟨3,3,2⟩ ⟨12,3,3⟩
4	A	•					●	•			•			⟨1,2,1⟩ ⟨3,3,2⟩ ⟨6,4,3⟩
5	B		●			●			●			•		⟨1,2,1⟩ ⟨2,5,2⟩ ⟨5,5,3⟩ ⟨8,5,4⟩
6	D									●				⟨1,2,1⟩ ⟨2,5,2⟩ ⟨5,5,3⟩ ⟨8,5,4⟩ ⟨9,6,5⟩
7	B		•			•			•			●		⟨1,2,1⟩ ⟨2,5,2⟩ ⟨5,5,3⟩ ⟨8,5,4⟩ ⟨9,6,5⟩ ⟨11,7,6⟩
8	A	•					●	•			●			⟨1,2,1⟩ ⟨2,5,2⟩ ⟨5,5,3⟩ ⟨6,8,4⟩ ⟨9,6,5⟩ ⟨10,8,6⟩

Rys. 8.3. Przykład działania algorytmu wyznaczania podciągu LCTS. Lewa strona: macierz programowania dynamicznego z zaznaczonymi dopasowaniami. Prawa strona: zawartość struktury Q po przetworzeniu kolejnych wierszy. Duże koła oznaczają dopasowania dominujące

Fig. 8.3. Example of the LCTS computing algorithm in work. The left side shows the matrix with marked matches, while the right side presents the contents of the Q data structure after processing each row. Large circles denote dominant matches

Złożoność czasowa tego algorytmu zależy w największym stopniu od wyboru struktury danych Q. Kwestia znalezienia wszystkich dopasowań zostanie rozważona nieco później, a na razie tylko przyjęte zostanie, że ich liczba to d. Dla każdego z d dopasowań (i,j) należy wykonać kilka operacji na strukturze Q. Co najwyżej raz należy: wyszukać trójkę o indeksie kolumny i, znaleźć następnik i, usunąć dopasowanie, wstawić dopasowanie. Istnieje wiele struktur danych, które mogą zostać użyte do reprezentacji Q. Zastosowanie zrównoważonego drzewa poszukiwań binarnych (np. drzewa czerwono-czarnego [29, 105, 190]) powoduje, że złożoność każdej z tych operacji jest $O(\log \ell)$, gdzie ℓ to długość podciągu wynikowego. Prowadzi to do algorytmu o złożoności czasowej $O(d \log \ell)$ (por. rozdz. 7). Ponieważ dopasowania w Q posortowane są według indeksu kolumny, więc można zastosować drzewa van Emde Boasa (vEB) [209, 210], dla których złożoność każdej z koniecznych operacji jest $O(\log\log x)$, gdzie x to maksymalna wartość elementu, który może kiedykolwiek być wstawiony do tej struktury. Ponieważ indeksy kolumn są z zakresu $[1,n]$, więc w analizowanym przypadku użycie vEB prowadzi do algorytmu o złożoności czasowej $O(d \log\log n)$.

W ramach każdego wiersza zapytania do Q wykonywane są dla monotonicznie rosnących indeksów kolumn. Umożliwia to reprezentowanie Q jako rodziny $\lceil (n+1)/e \rceil$ drzew vEB o rozmiarze $e \leq n$ każde (wartość e zostanie ustalona nieco dalej), w taki sposób, że drzewo pierwsze przechowuje dopasowania o indeksach kolumn z zakresu $[0, e-1]$, drzewo drugie – $[e, 2e-1]$ itd. (Zakłada się, że $e \leq n$, jeśli $e > n$, to występuje tylko jedno drzewo). Za każdy razem, kiedy wykonywane jest wstawienie, usunięcie, wyszukanie dopasowania o indeksie kolumny i, używane jest $\lfloor i/e \rfloor$-te drzewo vEB, a czas wykonania każdej z tych operacji jest $O(\log\log e)$. Wyszukiwanie następnika dla kolumny o indeksie i jest nieco bardziej skomplikowane, ponieważ jeśli drzewo o indeksie $\lfloor i/e \rfloor$ go nie zawiera, konieczne jest przeszukiwanie kolejnych drzew. Jednak dzięki temu, że w ramach wiersza wyszukiwane będą kolejno zawsze następniki coraz

to większych indeksów kolumn, to tylko raz dla każdego z tych drzew (w pojedynczym wierszu) będzie stwierdzone, że brak w nim następnika i należy przejść do drzewa następnego. Czas wyszukiwania następnika w drzewie jest $O(\log\log e)$, a dodatkowy czas wynikający z konieczności przejścia do kolejnego drzewa jest $O(n/e)$ na cały wiersz. Wobec tego całkowity czas działania algorytmu jest $O(d\log\log e + nm/e)$.

Ostatnią rzeczą, którą należy wziąć pod uwagę, jest czas sprawdzania, czy dopasowanie o danym numerze kolumny znajduje się w Q. Można z tej operacji w ogóle zrezygnować, a algorytm pozostanie poprawny. Można jednak rozszerzyć każde z drzew vEB o dodatkowy wektor bitowy rozmiaru e zawierający informacje, czy wartość o danym indeksie występuje w Q. Zarządzanie tym dodatkowym wektorem nie ma wpływu na złożoność czasową operacji na vEB, a dzięki niemu czas potrzebny na sprawdzenie, czy element występuje w drzewie, jest $O(1)$. Warto zauważyć, że operacja sprawdzania, czy element występuje w drzewie, jest najczęściej wykonywaną operacją na drzewach vEB – co najwyżej raz na każde dopasowanie. Każda z pozostałych operacji wykonywana jest tylko dla dopasowań dominujących. Dzięki takiemu usprawnieniu:

Wniosek 8.5. *Złożoność czasowa powyższego algorytmu* LCTS-One-Trans *jest* $O(D\log\log e + d + nm/e)$, *gdzie* D *to liczba dopasowań dominujących.*

8.3.2. Algorytm

Po takim zmodyfikowaniu algorytmu Hunta–Szymanskiego dla problemu LCS, jakie pokazano w poprzednim podrozdziale, można na jego podstawie zaproponować efektywny algorytm dla problemu wyznaczania długości LCTS.[1] Można to zrobić na dwa sposoby. Pierwszy sposób polega na niezależnym rozwiązaniu problemu LCS dla każdej transpozycji. W tej sytuacji omówiony powyżej algorytm wykonywany jest $\Theta(\sigma)$ razy, a sumaryczna liczba dopasowań wynosi dokładnie nm. Wobec tego otrzymuje się algorytm dla problemu LCTS o złożoności czasowej $O(D^*\log\log e + nm + nm\sigma/e)$, gdzie D^* jest sumaryczną liczbą dopasowań dominujących dla wszystkich transpozycji. Po przyjęciu $e = \Theta(\sigma)$ złożoność czasowa wynosi $O(D^*\log\log\sigma + nm)$.

Powyższa analiza nie uwzględnia faktu, że do wygenerowania zbioru dopasowań dla każdej z transpozycji konieczny jest pewien czas. Można to zrobić w taki sposób, że dla każdego symbolu alfabetu tworzona jest lista indeksów, na których występuje on w ciągu A. Wymaga to czasu $\Theta(n+\sigma)$. Dzięki temu przy przechodzeniu macierzy wierszami zawsze do dyspozycji jest

[1] W niniejszym podrozdziale dyskutowany jest algorytm wyznaczania tylko długości podciągu LCTS, ale w prosty sposób, analogicznie jak to było zrobione dla algorytmu LCS-HS (podrozdz. 7.3.2), można zmodyfikować ten algorytm, aby wyznaczał także podciąg LCTS. Wiąże się to oczywiście z większą złożonością pamięciową.

lista wszystkich dopasowań z bieżącego wiersza. Ponieważ macierz jest przechodzona dla każdej transpozycji, wnosi to dodatkowy koszt czasowy $\Theta(m\sigma)$, jako że dla każdego wiersza trzeba przynajmniej sprawdzić, czy dopasowania dla bieżącej transpozycji w nim występują czy nie.[2] Jeśli $\sigma = O(n)$, to ten dodatkowy składnik nie ma wpływu na złożoność czasową. W ogólności, złożoność czasowa powinna jednak uwzględniać tę możliwość:

Wniosek 8.6. *Złożoność czasowa algorytmu* LCTS-HS-1 *wykonującego niezależnie algorytm* LCTS-ONE-TRANS *dla każdej transpozycji jest* $O(D^* \log\log\sigma + m(n+\sigma))$, *gdzie* D^* *jest sumaryczną liczbą dopasowań dominujących dla wszystkich transpozycji.*

Wersja ta jest bardzo oszczędna pamięciowo, ponieważ potrzebne jest tylko $\Theta(\sigma)$ bitów pamięci dla każdego z $\Theta(n/\sigma)$ drzew vEB oraz $\Theta(n+\sigma)$ słów na inne zmienne. Wobec tego:

Wniosek 8.7. *Złożoność pamięciowa algorytmu* LCTS-HS-1 *jest* $\Theta(n+\sigma)$ *słów.*

Drugim sposobem zastosowania algorytmu LCTS-ONE-TRANS jest założenie, że macierz programowania dynamicznego przechodzona jest dokładnie raz, a ponieważ każda para (i, j) jest dopasowaniem dla dokładnie jednej transpozycji, więc wystarczy „przełączyć się” na struktury danych dla niej. Innymi słowy, $\Theta(\sigma)$ problemów LCS jest rozwiązywanych równolegle. W tym przypadku nie występują struktury danych zawierające listy dopasowań.

Wniosek 8.8. *Złożoność czasowa algorytmu* LCTS-HS-2 *polegającego na jednokrotnym przejściu przez całą macierz programowania dynamicznego jest* $O(D^* \log\log\sigma + nm)$.

Wniosek 8.9. *Złożoność pamięciowa algorytmu* LCTS-HS-2 *jest* $\Theta(\lceil n/w \rceil \sigma)$ *słów.*

Pseudokod algorytmu LCTS-HS-2 pokazany jest na rys. 8.4. Dla prostoty zakłada się w nim, że $e = \sigma$. Algorytm ten wyznacza długości podciągów LCS dla każdej transpozycji i przechowuje je w tablicy $\ell[-\sigma+1..\sigma-1]$. Tablica $\Upsilon[-\sigma+1..\sigma-1]$ zawiera numery indeksów kolumn ostatnio znalezionego dopasowania w Q, które zostało zastąpione dopasowaniem o mniejszym indeksie kolumny. W rzeczywistości, w Υ przechowywane są indeksy kolumn powiększone o iloczyn indeksu wiersza i długości ciągu A, dzięki czemu struktura ta nie musi być czyszczona po przetworzeniu każdego wiersza, ponieważ każda z wartości $(i + jn)$ z nowego wiersza jest większa niż dowolna z wartości występujących w Υ.

Przedstawiony algorytm ma lepszą złożoność czasową niż algorytm zaproponowany w podrozdz. 8.2, a ponadto jest istotnie prostszy. Ponieważ drzewa van Emde Boasa w praktyce są stosunkowo wolne, więc poniżej zaproponowana zostanie inna struktura danych, która może je

[2] Możliwe jest również przygotowanie w czasie $O(nm)$ list dopasowań dla każdej z transpozycji i używanie ich w algorytmie, dzięki czemu koszt $O(m\sigma)$ się nie pojawia, ale ponieważ w rozważanej wersji algorytmu chodzi o minimalizację pamięci, więc ten wariant nie będzie rozważany, gdyż same te listy wymagają $\Theta(nm)$ pamięci.

```
LCTS-HS-2(A, B, σ)
Wejście: A, B – ciągi, dla których wyznaczany jest podciąg LCTS
         σ – rozmiar alfabetu
Wyjście: długość podciągu LCTS

   {Przetwarzanie wstępne}
 1 for t ← −σ+1 to σ−1 do
 2     ℓ[t] ← 0;   ϒ[t] ← −1
 3     for i ← 0 to ⌊(n+σ−1)/σ⌋−1 do Q[t][i].init(σ)
   {Obliczenia właściwe}
 4 for j ← 0 to m−1 do
 5     for i ← 0 to n−1 do
 6         t ← a_i − b_j
 7         if i+j×n > ϒ[t] and not Q[t][⌊i/σ⌋].find(i mod σ) then
 8             x ← Q[t][⌊i/σ⌋].successor(i mod σ)
 9             if s < 0 then
10                 for k ← ⌊i/σ⌋+1 to ⌊(n+σ−1)/σ⌋−1 do
11                     if Q[t][k].minimum() ≥ 0 then
12                         x ← k×σ+Q[t][k].minimum();   break
13             else x ← x+i−i mod σ
14             Q[t][⌊i/σ⌋].insert(i mod σ)
15             if x ≥ 0 then ϒ[t] ← x+j×n;   Q[t][⌊x/σ⌋].remove(x mod σ)
16             else ϒ[t] ← n−1+j×n;   ℓ[t] ← ℓ[t]+1
   {Wyznaczanie wyniku}
17 ℓ ← 0;
18 for t ← −σ+1 to σ−1 do ℓ ← max(ℓ, ℓ[t])
19 return ℓ
```

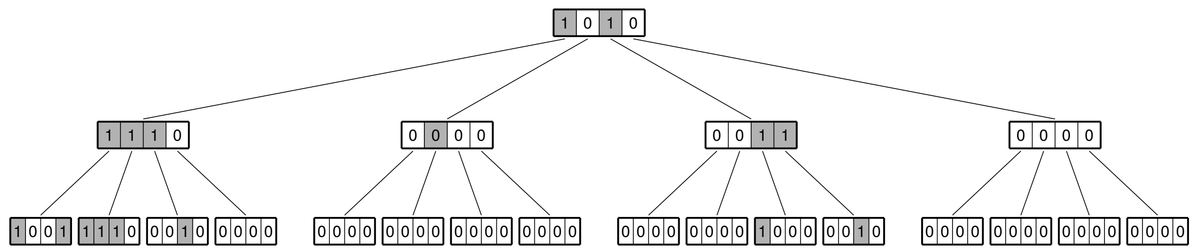
Rys. 8.4. Algorytm wyznaczający długość podciągu LCTS o złożoności czasowej $O(D^* \log\log\sigma + nm)$. (Operacje successor i minimum zwracają wartość -1, jeśli nie istnieje następnik lub wartość minimalna)

Fig. 8.4. An $O(D^* \log\log\sigma + nm)$-time algorithm computing the LCTS length. (The operations successor and minimum return -1 when there is no successor or minimal value)

zastąpić w tym algorytmie. Dogodnym kandydatem jest tu drzewo w-arne, gdzie w to długość słowa komputerowego w bitach. Każdy z węzłów wewnętrznych zawiera do w bitów. Bit ma wartość 1 wtedy i tylko wtedy, gdy odpowiadające mu dziecko zawiera co najmniej jeden bit o wartości 1. Dane przechowywane są w liściach – bit ma wartość 1, gdy odpowiadająca mu liczba całkowita jest przechowywana w drzewie. Przykład tej struktury danych pokazany jest na rys. 8.5.

Przy założeniu że operacja wyznaczania liczby zer wiodących (ang. *number of leading zeros*, NLZ) dla typów całkowitych jest obliczana przez procesor w czasie stałym (co jest prawdą dla współczesnych architektur), można zaimplementować każdą z potrzebnych operacji na tym drzewie w czasie $O(\lceil \log\sigma / \log w \rceil)$.[3] W celu sprawdzenia, czy wartość x występuje w drze-

[3]Możliwe jest także stworzenie, na etapie przetwarzania wstępnego, tablicy przeglądowej (ang. *lookup table*) zawierającej wartość funkcji NLZ dla każdej liczby całkowitej z zakresu $[0, 2^{\lceil w/2 \rceil} - 1]$ w czasie $O(2^{\lceil w/2 \rceil})$. Dzięki



Rys. 8.5. Przykładowe drzewo w-arne ($w = 4$) zawierające następujące liczby całkowite z zakresu $[0, 63]$: $0, 3, 4, 5, 6, 10, 40, 46$

Fig. 8.5. Sample tree of arity 4 storing integers from the range $[0, 63]$ (the integers $0, 3, 4, 5, 6, 10, 40, 46$ are stored in leaves)

wie w-arnym, wystarczy sprawdzić wartości bitu $(x \bmod w)$ w liściu o indeksie $\lfloor x/w \rfloor$. Wstawienie lub usunięcie wartości wymaga ustawienia na 1 lub 0 odpowiedniego bitu w odpowiednim liściu i zaktualizowania informacji na ścieżce w górę drzewa. Operacja wyszukiwania następnika jest nieco bardziej skomplikowana. Wymaga ona sprawdzenia czy bit o indeksie większym niż $(x \bmod w)$ w liściu o indeksie $\lfloor x/w \rfloor$ ma wartość 1 . Jeśli nie, to należy przejść w górę drzewa i postępować analogicznie, aż taki bit zostanie znaleziony. Wówczas należy zejść w dół drzewa i znaleźć minimum w odpowiednim poddrzewie. Wszystkie te operacje na drzewie w-arnym mają złożoność czasową $O(\log e / \log w)$, gdzie e jest zakresem przechowywanych wartości. Przyjmując $e = \Theta(\sigma)$ otrzymuje się po zastosowaniu tej struktury w miejsce każdego z drzew vEB algorytmy o złożonościach czasowych $O(D^* \lceil \log \sigma / \log w \rceil + nm)$ oraz $O(D^* \lceil \log \sigma / \log w \rceil + m(n + \sigma))$ (algorytm LCTS-HS-3). Złożoności pamięciowe pozostają bez zmian.

Jeszcze inną możliwością (algorytm LCTS-HS-4) jest przyjęcie $e = w$ i zorganizowanie Q jako tablicy $\lceil (n+1)/w \rceil$ liczb całkowitych bez znaku. W celu sprawdzenia, czy liczba x występuje w Q, wystarczy sprawdzić bit o indeksie $(x \bmod w)$ w $\lfloor x/w \rfloor$-tym słowie, co wymaga czasu $O(1)$. Wstawienie bądź usunięcie liczby jest tak samo łatwe. Znalezienie następnika $(x \bmod w)$ w $\lfloor x/w \rfloor$-tym słowie także wymaga stałego czasu (przy założeniu dostępności operacji NLZ), ale oczywiście jeśli następnik znajduje się w jednym z kolejnych słów, to należy te słowa przeglądnąć. Podejście takie nie ma wpływu na złożoność pamięciową, a sumarycznie na każdy wiersz wnosi do złożoności czasowej składnik $O(\lceil n/w \rceil)$. Można zatem sformułować wniosek:

Wniosek 8.10. *Złożoność czasowa algorytmu* LCTS-HS-4 *jest* $O(nm + \lceil n/w \rceil m\sigma)$.

tej tablicy wartość funkcji NLZ dla dowolnej liczby całkowitej rozmiaru w bitów może być wyznaczona w dwu zapytaniach do tablicy przeglądowej. Metoda ta może być stosowana tak długo, jak $2^{\lceil w/2 \rceil} = O(D \log \sigma / \log w + nm)$.

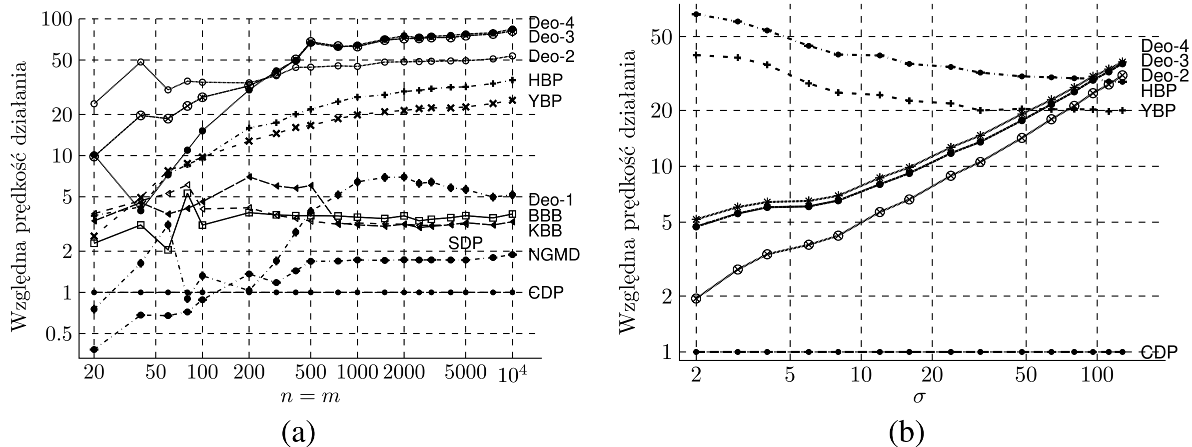
8.3.3. Wyniki eksperymentalne

W celu porównania zaproponowanych algorytmów z algorytmami istniejącymi przeprowadzono dwa eksperymenty. W pierwszym z nich mierzony był czas działania algorytmów dla ciągów zawierających wysokości dźwięków z rzeczywistych plików formatu MIDI ($\sigma = 128$).[4] Założono, że długości porównywanych ciągów są równe ($n = m$). Testy przeprowadzono dla następujących algorytmów:

- BBB – algorytm oparty na binarnej metodzie podziału i ograniczeń o złożoności czasowej $O((nm + \log\sigma)\sigma)$ w przypadku pesymistycznym i $O((nm + \log\log\sigma)\log\sigma)$ w przypadku optymistycznym [139],
- CDP – klasyczny algorytm programowania dynamicznego dla problemu LCS powtórzona dla każdej transpozycji $t \in (-\sigma, \sigma)$ o złożoności czasowej $O(nm\sigma)$ [141],
- KBB – algorytm oparty na k-arnej metodzie podziału i ograniczeń o złożoności czasowej $O((nm + \log(\sigma k/(k-1)))\sigma k/(k-1))$ w przypadku pesymistycznym oraz $O((nm + \log(k\log_k\sigma)) \times k\log_k\sigma)$ w przypadku optymistycznym [139]; w eksperymentach przyjęto $k = 3$, ponieważ zarówno w [139], jak i we wstępnych eksperymentach dla takiej wartości parametru algorytm okazał się najszybszy,
- SDP – algorytm oparty na idei programowania dynamicznego dla macierzy rzadkich o złożoności czasowej $O(nm\log m)$ [150],
- YBP – algorytm równoległości bitowej o złożoności czasowej $O(nm\lceil\sigma/w\rceil)$ [51],
- HBP – algorytm równoległości bitowej dla problemu LCS wykonany dla każdej z transpozycji $t \in (-\sigma, \sigma)$ o złożoności czasowej $O(\lceil n/w\rceil m\sigma)$ [120],
- NGMD – algorytm zaproponowany w podrozdziale 8.2 o złożoności czasowej $O(nm \times \log\log\sigma)$,
- Deo-1 – algorytm proponowany LCTS-HS-1 z drzewami van Emde Boasa o złożoności czasowej $O(D^*\log\log\sigma + m(n+\sigma))$,
- Deo-2 – algorytm proponowany LCTS-HS-2 z drzewami w-arnymi o złożoności czasowej $O(D^*\lceil\log\sigma/\log w\rceil + nm)$,
- Deo-3 – algorytm LCTS-HS-3, w którym transpozycje przetwarzane są kolejno, a nie wszystkie równocześnie, dzięki czemu potrzebna pamięć jest znacznie mniejsza; złożoność czasowa tego algorytmu jest $O(D^*\lceil\log\sigma/\log w\rceil + m(n+\sigma))$,
- Deo-4 – algorytm proponowany LCTS-HS-4 z pojedynczymi słowami komputerowymi zamiast drzew vEB; złożoność czasowa tego algorytmu jest $O(m\sigma\lceil n/w\rceil + nm)$.

Wszystkie eksperymenty przeprowadzono na komputerze wyposażonym w procesor AMD Athlon 2500 XP+ (zegar 1800 MHz) i 512 MB RAM. Kody źródłowe algorytmów BBB, CDP,

[4] Do wygenerowania danych rzeczywistych użyto 7543 utworów muzycznych zapisanych w formacie MIDI [92], z których wzięto tylko informacje o wysokości dźwięku. Sumaryczna długość tego ciągu wynosiła 1 828 089 symboli. Alfabet w tym przypadku miał rozmiar 128, co daje 255 możliwych transpozycji.



Rys. 8.6. Relatywna prędkość działania algorytmów rozwiązujących problem LCTS, pokazująca ile razy badany algorytm jest szybszy niż CDP. (a) Zależność od długości ciągów. (Algorytm SDP nie był oceniany dla długich ciągów z uwagi na bardzo duże zapotrzebowanie na pamięć operacyjną.) (b) Zależność od rozmiaru alfabetu dla $n = m = 1000$

Fig. 8.6. Relative speed of algorithms solving the LCTS problem specifying how many times evaluated algorithm is faster than CDP. (a) The dependency on the sequences length. (The SDP algorithm was not evaluated for long sequences due to its huge space consumption.) (b) The dependency on the alphabet size for $n = m = 1000$

KBB, SDP, YBP, HBP zostały udostępnione przez autorów. Implementacje wykonano w języku C++. Z uwagi na wykorzystany procesor długość słowa komputerowego wynosiła $w = 32$. Wszystkie implementacje zoptymalizowano pod kątem szybkości w taki sposób, aby osiągnąć podobny stopień optymalizacji. Wyniki eksperymentów stanowią mediany z 501 uruchomień każdego z algorytmów dla różnych ciągów wejściowych. Na rys. 8.6a przedstawiono czasy działania poszczególnych algorytmów. Dla większej przejrzystości zamiast bezwzględnych czasów działania przedstawione jest, ile razy każdy z algorytmów jest szybszy od algorytmu CDP.

Jak można zauważyć, algorytm HBP jest najszybszy spośród znanych w literaturze. Algorytmy NGMD oraz Deo-1 są relatywnie wolne mimo obiecującej złożoności czasowej. Jest to wynik zastosowania drzew van Emde Boasa, które w praktyce nie należą do szybkich rozwiązań. Algorytm Deo-2 działa kilka razy szybciej dla krótkich ciągów ($0 < n, m < 500$) i około 50% szybciej dla długich ciągów ($1000 < n, m \leq 10000$) niż algorytm HBP. Jeszcze szybsze (dwa razy szybsze niż algorytm HBP) dla ciągów dłuższych niż 300 okazały się algorytmy Deo-3 oraz Deo-4. Ich złożoność czasowa jest gorsza niż złożoność czasowa algorytmu Deo-1, ale w praktyce okazują się szybsze głównie z uwagi na prostotę implementacji. Co ciekawe, dla $n = m = 10^4$ algorytmy te potrzebują jedynie 3 razy więcej czasu na rozwiązanie problemu LCTS niż algorytm programowania dynamicznego na rozwiązanie problemu LCS.

W drugim eksperymencie badano wpływ rozmiaru alfabetu na czasy działania algorytmów (rys. 8.6b). Dla większej przejrzystości wybrano do tego testu tylko najszybsze algorytmy z po-

przedniego eksperymentu. Ciągi zostały wygenerowane z użyciem generatora liczb pseudolosowych o rozkładzie równomiernym. Algorytmy zaproponowane w niniejszym podrozdziale okazują się znacznie wolniejsze od algorytmu HBP dla małych alfabetów oraz szybsze dla alfabetów relatywnie dużych. Dla $\sigma = 128$ różnica w prędkości działania pomiędzy algorytmami Deo-4 oraz HBP jest mniejsza niż w poprzednim eksperymencie, co wynika prawdopodobnie ze specyfiki danych wejściowych. W danych rzeczywistych rozkład częstości występowania symboli daleki jest bowiem od równomiernego.

8.4. Algorytm hybrydowy

8.4.1. Algorytm

W niniejszym podrozdziale przedstawiony zostanie algorytm hybrydowy LCTS-HYBRID zaproponowany przez autora w [103, 73].

Wyniki eksperymentalne dla problemu LCTS pokazują, że dla stosunkowo dużych alfabetów zaproponowany algorytm oparty na metodzie Hunta–Szymanskiego (LCTS-HS-3) jest szybszy niż algorytm oparty na metodzie równoległości bitowej, w którym dla każdej transpozycji podciąg LCS wyznaczany jest za pomocą algorytmu LCS-BP-LENGTH (podrozdz. 7.3.4). Oba te algorytmy wyznaczają jednak w zasadzie podciągi LCS dla różnych transpozycji. Analizując czasy działania tych algorytmów dla poszczególnych transpozycji, można zauważyć, że algorytm LCTS-HS-3 jest szybszy od algorytmu LCS-BP-LENGTH, jeśli liczba dopasowań jest znacznie mniejsza niż nm. W przeciwnym przypadku szybszy jest algorytm równoległości bitowej. Ponieważ w problemie LCTS liczba dopasowań silnie zależy od transpozycji (w przybliżeniu im wartość bezwzględna transpozycji większa, tym mniej dopasowań), więc można zaproponować podejście hybrydowe, w którym transpozycje z małą liczbą dopasowań będą rozwiązywane metodą HS (algorytm LCTS-HS-3), a te z dużą liczbą dopasowań metodą BP (algorytm LCS-BP-LENGTH).

W proponowanym algorytmie hybrydowym,[5] na etapie przetwarzania wstępnego wyznaczana jest liczba dopasowań dla każdej z transpozycji. Złożoność czasowa tego etapu jest $O(n+m+\sigma^2)$. Następnie transpozycje są sortowane według liczby dopasowań w czasie $O(\sigma\log\sigma)$ lub $O(\sigma\log_\sigma n)$ (z użyciem sortowania pozycyjnego).[6]

[5] Algorytm ten jest opisany również w [102], będącej rozprawą habilitacyjną drugiego autora tego algorytmu.

[6] Etap zliczania liczby dopasowań dla transpozycji może być także wykonany w czasie $O(n+m+\sigma\log\sigma)$ z użyciem szybkiej transformaty Fouriera (FFT), jeśli tylko dopuści się nieco bardziej ogólny model obliczeniowy, w którym wśród operacji podstawowych są operacje mnożenia i dzielenia liczb zespolonych. W tym celu należy utworzyć dwie tablice N^A oraz N^B zawierające $2\sigma-1$ elementów. Tablica N^A zawiera liczbę wystąpień symboli alfabetu w A, a N^B w B na pozycjach $0..\sigma-1$. Pozostałe elementy tych tablic są wypełnione zerami. Następnie wyznaczana jest cykliczna korelacja dyskretnych ciągów N^A oraz N^B, co może być wykonane z użyciem FFT w czasie $O(\sigma\log\sigma)$. Otrzymane $2\sigma-1$ współczynników zawiera liczbę dopasowań dla każdej z $2\sigma-1$ transpozycji.

W dalszej części opisu algorytmu stosowana będzie notacja $R(t)$, która oznacza dla transpozycji t jej rangę w tablicy transpozycji posortowanej według liczby dopasowań. Zachodzi przy tym $0 \leq R(t) < 2\sigma - 1$, a więc $R(t) = 0$, jeśli t jest transpozycją, dla której liczba dopasowań jest maksymalna (jeśli istnieje wiele takich transpozycji, to wybierana jest którakolwiek z nich). Przez $d(t)$ będzie oznaczana liczba dopasowań dla transpozycji t. Następnie wykonywane jest przetwarzanie wstępne dla algorytmów LCTS-HS-3 i LCS-BP-Lenght w sposób podany poniżej.

Jeśli dla każdej transpozycji przetwarzanie wstępne w algorytmie LCS-BP-Length wykonywane jest niezależnie, to jego sumaryczna złożoność czasowa jest $O(\lceil m/w \rceil \sigma^2 + m\sigma)$. Można ten wynik bardzo prosto poprawić do $O(\lceil m/w \rceil \sigma + m)$. W tym celu wystarczy utworzyć listy wystąpień każdego symbolu alfabetu w każdym ciągu oraz wektory masek bitowych dla ciągu B i przy przetwarzaniu transpozycji t oraz kolumny opisanej symbolem a_i wziąć wektor masek bitowych dla symbolu $(a_i - t)$. Ta prosta idea pozwala wyeliminować niepotrzebne, wielokrotne wykonywanie przetwarzania wstępnego, a przeprowadzone eksperymenty pokazują, że uzyskiwana oszczędność czasu to ok. 20%. Można ponadto zauważyć, że niezerowych wektorów masek bitowych może być co najwyżej $\min(m, \sigma)$, wobec czego pamięć na pozostałe wektory nie musi być alokowana. Dodatkowy wkład do złożoności czasowej tej modyfikacji jest $\Theta(m + \sigma)$.

Czas przetwarzania wstępnego można zredukować jeszcze bardziej, w nieco mniej oczywisty sposób, do $O(m)$. Wymaga to zwiększenia rozmiaru zajmowanej pamięci o stały czynnik. Zysk czasowy jest osiągany przez powstrzymanie się od zerowania wektorów bitowych na etapie ich inicjalizacji. W tym celu należy wykorzystać metodę inicjalizacji tablic [155, rozdz. III 8.1], która wymaga zastosowania dodatkowych tablic, przez co zajętość pamięci rośnie co najwyżej trzykrotnie. Navarro [162] zaprezentował ulepszenie tej metody, dzięki czemu dodatkowa pamięć wymagana dla tablicy rozmiaru N to jedynie $N + o(N)$ bitów. Przekłada się to na $\lceil m/w \rceil + o(\lceil m/w \rceil)$ dodatkowych bitów w rozważanym problemie. Wadą tej metody jest jednak konieczność trzykrotnego dostępu do pamięci, aby uzyskać informację o wektorze masek bitowych, co przy relatywnie niedużej zajętości pamięci przez te wektory powoduje, że ta optymalizacja pamięciowa ma znaczenie raczej teoretyczne.

Po przetwarzaniu wstępnym transpozycje dzielone są pomiędzy algorytmy LCTS-HS-3 oraz LCS-BP-Length. Ponieważ algorytm LCTS-HS-3 lepiej nadaje się do transpozycji z małą liczbą dopasowań, a algorytm LCS-BP-Length z dużą ich liczbą, więc podział polega na znalezieniu takiej granicznej liczby dopasowań, że wszystkie transpozycje o mniejszej ich liczbie będą przetwarzane algorytmem LCTS-HS-3, a pozostałe algorytmem LCS-BP-Length. Kryterium podziału jest wyznaczane na podstawie czasów obliczeń dla kilku wstępnie wybranych transpozycji. W szczególności: dla transpozycji t_1 rangi 0 i t_2 rangi $\lceil (2\sigma - 1)/3 \rceil - 1$ wykonywany jest

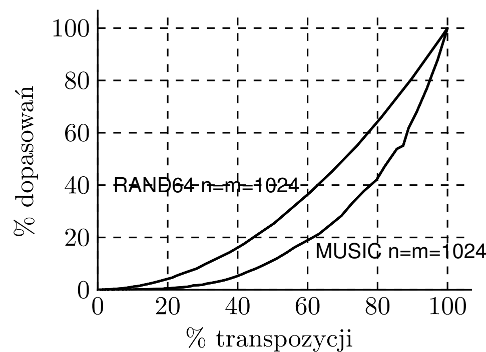
algorytm LCS-BP-LENGTH, a czasy działania to odpowiednio: $\tau(t_1)$ oraz $\tau(t_2)$. Transpozycje te są następnie usuwane ze zbioru wszystkich transpozycji (dla nich długość podciągów LCS została właśnie policzona). Teraz wybierane są transpozycje t_3 oraz t_4 o rangach odpowiednio: $\lceil(2\sigma - 3)/3\rceil - 1$ oraz $2\sigma - 4$ (najrzadsza transpozycja) i dla nich wykonywany jest algorytm LCTS-HS-3. Uzyskane czasy to odpowiednio: $\tau(t_3)$ oraz $\tau(t_4)$.

Istotą działania algorytmu hybrydowego jest założenie, że dla każdego z algorytmów, LCTS-HS-3 i LCS-BP-LENGTH, czas działania z dobrym przybliżeniem zależy liniowo od liczby dopasowań. Oczywiście założenie to nie jest spełniane dokładnie, ale wstępne eksperymenty pokazały, że przyjęta strategia sprawdza się całkiem dobrze w praktyce. Zależność czasu działania algorytmu LCTS-HS-3 od liczby dopasowań jest dość oczywista. W przypadku algorytmu LCS-BP-LENGTH już tak nie jest, ale eksperymenty pokazują, że prawdopodobnie z powodów zależnych od sprzętu też daje się taką zależność zauważyć, choć jest ona słaba. Otrzymane 4 czasy dla 4 transpozycji są wystarczające, aby poprowadzić dwie proste i na podstawie ich punktu przecięcia wyznaczyć zakresy transpozycji do przetwarzania przez poszczególne algorytmy składowe. W szczególności pierwsza prosta prowadzona jest przez punkty: $(d(t_1), \tau(t_1))$ i $(d(t_2), \tau(t_2))$, a druga przez punkty: $(d(t_3), \tau(t_3))$ i $(d(t_4), \tau(t_4))$. Transpozycje o liczbie dopasowań większej niż pierwsza współrzędna otrzymanego punktu przecięcia przetwarzane są algorytmem LCS-BP-LENGTH, a pozostałe algorytmem LCTS-HS-3. Oczywiście cztery wcześniej policzone transpozycje nie są ponownie przetwarzane. Kolejnym ulepszeniem algorytmu LCTS-HS-3 jest przedwczesne jego przerwanie, jeśli na którymś etapie wiadomo już, że dla bieżącej transpozycji niemożliwe jest poprawienie najlepszego uzyskanego wyniku, nawet gdyby dla każdego kolejnego wiersza długość podciągu LCS wzrastała o 1. Ponieważ, statystycznie rzecz biorąc, dla gęstszych transpozycji jest większa szansa otrzymania dłuższych wspólnych podciągów, więc najpierw uruchamiany jest algorytm LCS-BP-LENGTH, a po nim dopiero algorytm LCTS-HS-3, dzięki czemu ta idea przedwczesnego zakończenia przetwarzania w algorytmie LCTS-HS-3 jest częściej stosowana.

Na koniec warto odnotować, że sumaryczna złożoność czasowa proponowanego algorytmu jest $O(n\lceil m/w\rceil\sigma + nm\lceil\log\sigma/\log w\rceil)$, wliczając czas przetwarzania wstępnego dla obu składowych algorytmów w najgorszym przypadku (przy założeniu że $w = O(\log^{O(1)} n)$.

8.4.2. Wyniki eksperymentalne

W celu porównania algorytmu zaproponowanego w podrozdz. 8.4.1 z innymi algorytmami dla problemu LCTS przeprowadzono serię eksperymentów podobnych do tych opisanych w podrozdz. 8.3.3. W testach zastosowano komputer wyposażony w procesor AMD Athlon64 5000+ (zegar 2600 MHz) i 2048 MB RAM, działający pod kontrolą systemu operacyjnego MS Vista64. Wszystkie algorytmy zostały zaimplementowane w języku C++, a do kompilacji użyto



Rys. 8.7. Skumulowana procentowa liczba dopasowań (począwszy od transpozycji najrzadszych) w zależności od rangi transpozycji (problem LCTS)
Fig. 8.7. Cumulative % matches vs. % transpositions counted from the most rare transpositions (LCTS problem)

MS Visual C++ 2005 z opcją -Ox (maksymalna optymalizacja pod kątem szybkości działania). Eksperymenty przeprowadzono zarówno dla $w = 32$, jak i $w = 64$.

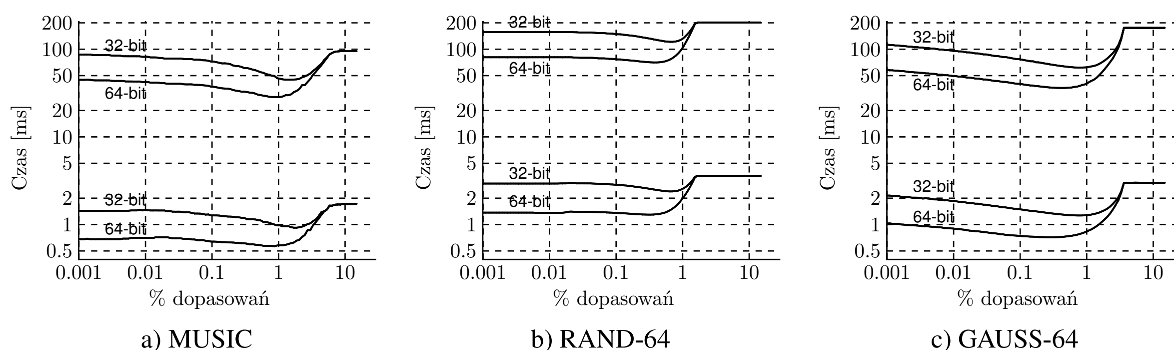
Dane użyte do testów były trzech rodzajów:

- MUSIC: dane rzeczywiste z plików MIDI (te same, których użyto w podrozdz. 8.3.3),
- RAND: ciągi wygenerowane za pomocą generatora liczb pseudolosowych o rozkładzie równomiernym,
- GAUSS: ciągi wygenerowane za pomocą generatora liczb pseudolosowych o rozkładzie Gaussa (średnia w połowie alfabetu, a odchylenie standardowe – $\sigma/8$).

Z wyjątkiem pierwszego rodzaju danych ciągi były generowane dla różnych rozmiarów alfabetu.

W pierwszym eksperymencie, dla zestawu MUSIC, ciągi do testów zostały utworzone jako losowo wybrane fragmenty tego ciągu bazowego. Dla każdej długości ciągów ($n = m$) wygenerowano 101 par, a otrzymane czasy są medianą czasów wykonań dla tych par. Rysunek 8.7 pokazuje zależność pomiędzy procentową liczbą transpozycji z najmniejszą liczbą dopasowań a sumaryczną procentową liczbą dopasowań, które te transpozycje zawierają. Można z niego odczytać, że, dla danych rzeczywistych (MUSIC), 20% najczęstszych transpozycji zawiera ponad 50% wszystkich dopasowań, a 60% istniejących transpozycji zawiera ponad 90% dopasowań. Dla danych losowych krzywa ta jest znacznie bardziej płaska.

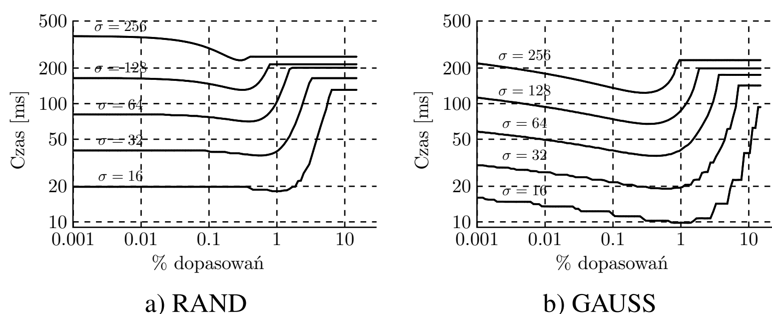
Na rys. 8.8 pokazano sumaryczny czas działania proponowanego algorytmu hybrydowego w zależności od maksymalnej liczby dopasowań w ramach transpozycji przetwarzanych przez algorytm składowy LCTS-HS-3. Analogiczną zależność przedstawiono na rys. 8.9, z tą różnicą, że tu pokazano wykresy dla alfabetów różnych rozmiarów. (Skrajne wartości progów dotyczą sytuacji, w której wszystkie transpozycje przetwarzane są przez ten sam algorytm składowy). W większości przypadków, dla alfabetów do rozmiaru 64 lub 128 (w zależności od zestawu danych i wersji, 32- lub 64-bitowej), szybszym z dwóch algorytmów składowych okazywał się LCS-BP-Length. Jedynie dla większych alfabetów wygrywała składowa HS. Można rów-



a) MUSIC b) RAND-64 c) GAUSS-64

Rys. 8.8. Czasy działania algorytmu LCTS-HYBRID dla problemu LCTS dla zmieniającego się progu podziału maksymalnej liczby dopasowań w transpozycji przetwarzanej przez składową LCTS-HS-3. Górne pary krzywych dla $n = m = 4096$, dolne pary dla $n = m = 512$

Fig. 8.8. Processing time of LCTS-HYBRID algorithm with varying threshold of the maximal number of matches in transpositions handled by the LCTS-HS-3 component. Top pairs of curves for $n = m = 4096$, bottom pairs for $n = m = 512$



a) RAND b) GAUSS

Rys. 8.9. Czas działania algorytmu LCTS-HYBRID dla problemu LCTS dla zmiennego progu podziału maksymalnej liczby dopasowań w transpozycji przetwarzanej przez składową LCTS-HS-3. $n = m = 4096$, $\sigma = 16, \ldots, 256$, $w = 64$

Fig. 8.9. Processing time of LCTS-HYBRID algorithm with varying threshold of maximal number of matches in transpositions handled by the LCTS-HS-3 component. $n = m = 4096$, $\sigma = 16, \ldots, 256$, $w = 64$

nież zauważyć, że w przypadku danych gaussowskich czas działania algorytmu bardziej zależy od drobnych zmian progu podziału transpozycji pomiędzy składowymi LCTS-HS-3 i LCS-BP-LENGTH.

Z testów wynika, że najlepszym punktem podziału dla danych rzeczywistych jest przetwarzanie około 80–90% dopasowań (z najczęstszych transpozycji) przez algorytm LCS-BP-LENGTH, a pozostałych 10–20% przez wariant algorytmu LCTS-HS-3 (tabele 8.1–8.4). Innymi słowy (rys. 8.7), mniej niż 40% najczęstszych transpozycji powinno być przetwarzanych przez składową LCS-BP-LENGTH. Można również zauważyć, że składowa LCS-BP-LENGTH przetwarzająca wszystkie transpozycje jest szybsza o około 25% (implementacja 32-bitowa) lub 100% (implementacja 64-bitowa) niż składowa LCTS-HS-3 przetwarzająca wszystkie transpozycje.

W celu określenia, jak dobrze wykonywany jest podział transpozycji pomiędzy obie składowe, wyznaczony został także najlepszy możliwy próg (kolumna „Czas najl.”). Jak można zauwa-

Tabela 8.1
Czasy działania algorytmu LCTS-Hybrid ($w = 32$) dla problemu LCTS, dane: MUSIC

n=m	Czas BP [ms]	Czas HS [ms]	Czas najl. [ms]	Czas hyb. [ms]	Próg najl. [%]	Próg hyb. [%]	Trans. BP [%]	Dopas. BP [%]
256	0,3157	0,5167	0,2686	0,2768	1,6928	1,3930	45,4	83,8
512	1,4366	1,7222	0,9210	0,9686	1,8621	1,5752	42,1	80,5
1024	5,5727	6,3598	3,0630	3,1441	1,6928	1,4759	40,2	81,5
2048	23,2568	25,5058	13,3131	13,5623	1,5389	1,4969	34,8	78,3
4096	91,7169	95,0223	44,9423	45,6610	1,3990	1,4930	34,7	80,9
8192	381,6627	381,3309	185,5993	188,4778	1,5389	1,5642	33,6	79,9

Tabela 8.2
Czasy działania algorytmu LCTS-Hybrid ($w = 64$) dla problemu LCTS, dane: MUSIC

n=m	Czas BP [ms]	Czas HS [ms]	Czas najl. [ms]	Czas hyb. [ms]	Próg najl. [%]	Próg hyb. [%]	Trans. BP [%]	Dopas. BP [%]
256	0,1842	0,5183	0,1842	0,2151	0,0000	0,7371	61,9	92,4
512	0,6811	1,7288	0,5669	0,5868	0,8687	0,7022	61,2	93,1
1024	3,0936	6,3303	2,0560	2,1032	1,1562	0,8202	55,0	91,1
2048	12,2291	25,5285	8,3442	8,6279	0,8687	0,7895	52,1	90,1
4096	47,1900	95,0745	28,4840	29,0666	0,7897	0,7683	48,9	91,8
8192	193,8347	380,7648	112,5319	113,1798	0,8687	0,7952	43,7	91,2

Tabela 8.3
Czasy działania algorytmu LCTS-Hybrid ($w = 64$) dla problemu LCTS, dane: RAND-128

n=m	Czas BP [ms]	Czas HS [ms]	Czas najl. [ms]	Czas hyb. [ms]	Próg najl. [%]	Próg hyb. [%]	Trans. BP [%]	Dopas. BP [%]
256	0,7996	1,4635	0,7677	0,7771	0,1890	0,1898	75,6	93,6
512	2,7441	4,5429	2,4480	2,4640	0,2287	0,2248	71,3	91,2
1024	11,8452	15,4458	9,6989	9,7430	0,3044	0,2938	62,3	85,3
2048	42,9331	56,6781	34,5518	34,7321	0,3044	0,2914	62,3	85,6
4096	164,4345	215,2981	130,7266	131,3415	0,3044	0,2933	62,3	85,3
8192	729,7012	877,4197	572,9071	576,5836	0,3349	0,3295	57,6	81,8

żyć, zaproponowany sposób podziału transpozycji oparty na idei znalezienia punktu przecięcia prostych zachowuje się całkiem dobrze, ponieważ strata w stosunku do najlepszego możliwego do uzyskania wyniku wynosi zwykle nie więcej niż 2% (najgorszy przypadek to $n = m = 256$ w tabeli 8.2, gdzie wynosi ona 6%).

Szczegółowe czasy działania algorytmów: LCTS-HS-3, LCS-BP-Length i proponowanej wersji hybrydowej podane są w tabelach zarówno dla wersji 32-, jak i 64-bitowej. Kolejne wiersze tabeli zawierają dane dla różnych długości ciągów wejściowych od $n = m = 256$ do $n = m = 8192$. Krótkiego wyjaśnienia wymaga prawa część kolumn tabel. Kolumna „Próg najl." określa minimalną procentową liczbę dopasowań w transpozycji, dla której algorytm LCS-BP-Length działa szybciej niż algorytm LCTS-HS-3. Kolumna „Próg hyb." zawiera podobną informację, z tą różnicą, że wartości w niej umieszczone zostały wyznaczone przez proponowany algorytm hybrydowy. Zgrubnie można stwierdzić, że im bliższe wartości w tych dwu kolum-

Tabela 8.4
Czasy działania algorytmu LCTS-Hybrid ($w = 64$) dla problemu LCTS, dane: GAUSS-128

n=m	Czas BP [ms]	Czas HS [ms]	Czas najl. [ms]	Czas hyb. [ms]	Próg najl. [%]	Próg hyb. [%]	Trans. BP [%]	Dopas. BP [%]
256	0,5463	1,1105	0,4193	0,4316	0,1420	0,2572	51,1	94,3
512	1,9991	3,7308	1,3030	1,3254	0,1890	0,2741	46,7	94,0
1024	9,2762	13,3282	5,0989	5,1376	0,3349	0,3455	40,3	92,3
2048	35,9229	50,6291	18,0331	18,1210	0,3349	0,3336	38,0	92,6
4096	144,2099	198,6377	67,3530	67,4932	0,3044	0,3349	36,1	92,7
8192	597,2110	778,5215	263,3859	263,5459	0,3684	0,3506	34,3	92,3

nach, tym algorytm hybrydowy powinien działać lepiej.[7] W kolumnie „Trans. BP” znajduje się informacja o procentowej liczbie transpozycji przetwarzanych przez składową LCS-BP-Length w algorytmie hybrydowym. Kolumna „Dopas. BP” informuje o procentowej liczbie dopasowań przetwarzanych przez składową LCS-BP-Length w algorytmie hybrydowym.

Dla danych muzycznych algorytm hybrydowy jest szybszy od szybszego z algorytmów składowych od 1,36 ($n = 256$) do 1,97 ($n = 8192$) razy w implementacji 32-bitowej i od 1,11 ($n = 256$) do 1,71 ($n = 8192$) razy w implementacji 64-bitowej.

Dla danych losowych (RAND) sytuacja jest nieco inna. Dla alfabetów małych i średnich (do około $\sigma = 64$) algorytm LCS-BP-Length jest znacznie szybszy niż algorytm LCTS-HS-3. Nawet w implementacji 32-bitowej jest on czterokrotnie szybszy dla $\sigma = 16$ i $n = 4096$. Sytuacja zmienia się dla $\sigma = 128$ i $\sigma = 256$. Co ciekawe, dla małych alfabetów składowa LCTS-HS-3 okazuje się szybsza od składowej LCS-BP-Length dla kilku najrzadszych transpozycji, dzięki czemu algorytm hybrydowy okazuje się nieznacznie szybszy (do 10%). Dla dużych alfabetów składowa LCS-BP-Length jest wolniejsza od składowej LCTS-HS-3 dla niemal wszystkich transpozycji (lub wręcz dla wszystkich). Wobec tego algorytm hybrydowy degeneruje się do algorytmu LCTS-HS-3. Przypadkiem granicznym jest $\sigma = 128$, w którym algorytm hybrydowy jest szybszy niż szybsza ze składowych (LCTS-HS-3) o 12–24%.

8.5. Algorytm równoległy dla procesorów graficznych

8.5.1. Algorytm

Niniejszy podrozdział zawiera opis algorytmu równoległego dla procesorów GPU wyznaczania długości podciągu LCTS, który został zaproponowany przez autora w [66, 71]. W dostępnej literaturze nie ma, jak dotąd, podobnych algorytmów. Idea działania tego algorytmu opiera się na ogólnym schemacie działania algorytmów dla procesorów GPU zaproponowanym w podrozdz. 7.6.2. Zastosowany zostanie ten sam ogólny schemat algorytmu LCS*-CUDA przedstawiony w pseudokodzie z rys. 7.13. Zasadniczą różnicą pomiędzy problemami LCTS a LCS,

[7] Dokładniejszą dyskusję wyników w podanych tabelach można znaleźć w [73].

z punktu widzenia zrównoleglania, jest to, że problem LCTS można rozdzielić na niezależne podproblemy, tzn. $2\sigma - 1$ niezależnych zadań wyznaczenia podciągu LCS dla każdej możliwej transpozycji, z których następnie zostanie wybrany najdłuższy wspólny podciąg. W celu osiągnięcia jak największej szybkości działania punktem wyjścia do zrównoleglania jest algorytm równoległości bitowej dla problemu LCS (podrozdz. 7.3.4). Ponieważ wśród głównych zastosowań problemu LCTS znajduje się porównywanie sekwencji muzycznych, w których rozmiar alfabetu, σ, jest znacznie większy niż 32, więc poniżej użyty zostanie tylko algorytm z tablicą pośredniczącą (por. podrozdz. 7.6.4).

Na etapie przetwarzania wstępnego (rys. 7.13, wiersz 1), trójwymiarowa macierz dzielona jest na dwuwymiarowe pudełka. Ponieważ pomiędzy poziomami tej trójwymiarowej macierzy nie występują bezpośrednie zależności, więc liczba pudełek, które mogą być obliczane równolegle, jest duża (co najmniej $2\sigma - 1$).

Wektory masek bitowych dla dowolnych dwu transpozycji t_1 oraz t_2 takich, że $t_1 - t_2 = t_\Delta$ są przesunięte względem siebie o t_Δ, co pozwala na zapisanie wszystkich wektorów masek bitowych (dla wszystkich transpozycji) z użyciem tylko $3\sigma\lceil m/w_g\rceil$ słów pamięci (a nie $(2\sigma - 1)\sigma\lceil m/w\rceil$, jak mogłoby się wydawać, gdyby użyć naiwnej reprezentacji, w której dla każdej transpozycji tworzone są niezależne wektory masek). Struktury danych przechowywane w pamięci globalnej dla tego algorytmu to:

- tablica n słów na ciąg A,
- tablica $3\lceil m/w_g\rceil\sigma$ słów na wektory masek bitowych,
- tablica $(n(2\sigma - 1))$ słów na przeniesienia z górnych krawędzi,
- tablica $(\lceil m/w_g\rceil(2\sigma - 1))$ słów na wektory bitowe z lewej krawędzi.

Wniosek 8.11. *Złożoność pamięciowa algorytmu* LCTS-CUDA *jest*

$$\Theta\left(\left(n + \left\lceil \frac{m}{w_g} \right\rceil\right)\sigma\right) \textit{ słów.}$$

Kod jądra tego algorytmu jest bardzo podobny do kodu jądra dla dużych alfabetów algorytmu równoległości bitowej dla problemu LCS (rys. 7.16), dlatego też zostanie tu pominięty.

Rozważając złożoność czasową proponowanego algorytmu, można zauważyć, że w algorytmie z tablicą pośredniczącą dla problemu LCS (podrozdz. 7.6.4) składnikiem dominującym był czas potrzebny do wczytania do tej tablicy wektorów masek. Łatwo można sprawdzić, że w algorytmie dla problemu LCTS dominujący będzie ten sam składnik. W związku z tym, że w tym problemie wykonywanych jest $\Theta(\sigma)$ razy więcej obliczeń niż w problemie LCS:

Wniosek 8.12. *Złożoność czasowa proponowanego algorytmu* LCTS-CUDA *jest*

$$\Theta\left(\frac{nm\sigma}{\eta_2 w_g}\right). \tag{8.5}$$

Wniosek 8.13. *Przyspieszenie w stosunku do algorytmu sekwencyjnego dla procesora CPU wykonującego* $\Theta(nm\sigma/w_c)$ *operacji wynosi*

$$\Theta\left(\frac{\eta_2 w_g}{w_c}\right). \tag{8.6}$$

8.5.2. Wyniki eksperymentalne

Dla oceny zaproponowanego algorytmu wykonane zostały eksperymenty z użyciem następującego zestawu komputerowego:

- CPU: procesor AMD Phenom II X4 810 taktowany zegarem 2600 MHz z 4 MB pamięci podręcznej trzeciego poziomu, 4096 MB RAM (taktowanej 1033 MHz),
- GPU: procesor NVidia GTX 260 taktowany następująco: 696 MHz (jądro), 1501 MHz (multiprocesory), 896 MB pamięci globalnej (taktowanej 1100 MHz) z 27 multiprocesorami (każdy multiprocesor zawiera 8 rdzeni).

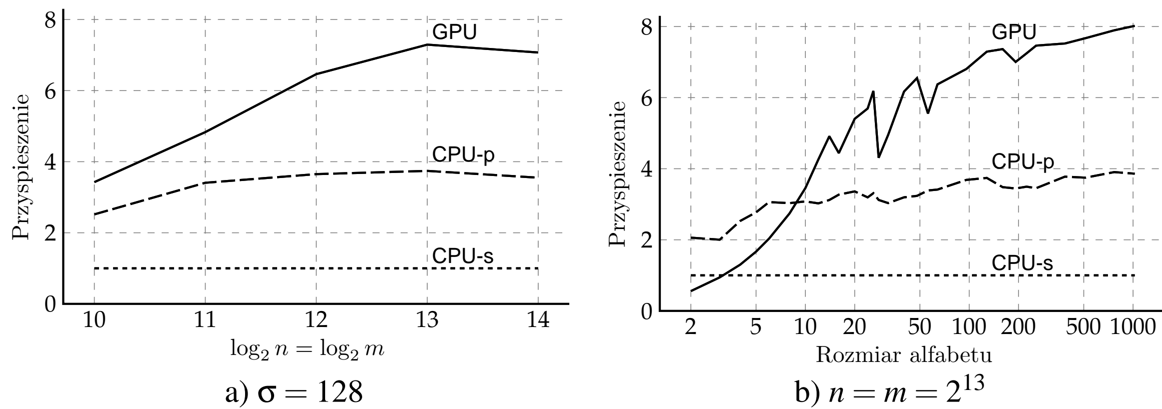
Przyspieszenie mierzono w taki sam sposób jak w podrozdz. 7.6.5. Użyto takiego samego kompilatora i tej samej wersji biblioteki CUDA.

Wszystkie czasy są medianami ze 101 wykonań. Zamiast czasów absolutnych, na wykresach pokazane jest przyspieszenie w stosunku do odpowiedniego algorytmu sekwencyjnego (działającego na jednym rdzeniu procesora CPU). Algorytmy na wykresach oznaczane są następująco:

- CPU-s – algorytm sekwencyjny dla procesorów CPU,
- CPU-p – algorytm równoległy dla procesorów CPU (wykorzystany CPU zawiera 4 rdzenie, ale wstępne eksperymenty pokazały, że lepiej jest uruchamiać naraz 32 wątki),
- GPU – algorytm równoległy dla procesorów GPU z tablicą pośredniczącą.

W problemie tym, dla ciągów krótkich, $m < 2^{14}$, pojedynczy blok może obliczyć całą dwuwymiarową macierz dla jednej transpozycji. Dzięki temu występuje tylko jedno wywołanie kodu jądra, co redukuje do minimum czas komunikacji CPU–GPU. Dla dłuższych ciągów wywołań kodu jądra jest oczywiście więcej. Wyniki dla $\sigma = 128$ (rys. 8.10a), typowego rozmiaru alfabetu dla sekwencji MIDI, pokazują, że maksymalne przyspieszenie osiąga wartość 7. Eksperymenty dla różnych rozmiarów alfabetów (rys. 8.10b) pokazują powolny wzrost przyspieszenia dla dużych alfabetów, ale nawet dla $\sigma = 1024$ przyspieszenie osiąga wartość jedynie 8.

Niemonotoniczny wzrost przyspieszenia, występujący w okolicach $\sigma = 32$, $\sigma = 64$, $\sigma = 192$, wynika z architektury procesora GPU użytego w eksperymentach. Przykładowo, dla $\sigma = 32$ istnieją $2\sigma - 1 = 63$ transpozycje i taka sama liczba bloków. Ponieważ procesor GPU zawiera 27 multiprocesorów, więc zgrubnie rzecz ujmując 9 z nich oblicza 3 bloki, a 18 oblicza 2 bloki. Całkowity czas obliczeń jest zdeterminowany przez obliczenia wykonywane przez 9 najbardziej obciążonych multiprocesorów. Dla $\sigma = 24$ jest więc 47 bloków i na żaden multiprocesor nie



Rys. 8.10. Porównanie przyspieszenia algorytmu równoległości bitowej dla procesora GPU w stosunku dla algorytmu LCTS-BP-Length dla procesora CPU. Rozmiary pudełek zostały dostosowane do długości ciągów w następujący sposób: $b_{\mathrm{w}} = b_{\mathrm{w}}^{\mathrm{cpu}} = \min\left(n, 2^{14}\right), b_{\mathrm{h}} = b_{\mathrm{h}}^{\mathrm{cpu}} = \min\left(m, 2^{14}\right)$

Fig. 8.10. Comparison of the speedup for the bit-parallel LCTS algorithm for GPU over LCTS-BP-Length algorithm for CPU. The box sizes were adjusted to the sequences lengths as follows: $b_{\mathrm{w}} = b_{\mathrm{w}}^{\mathrm{cpu}} = \min\left(n, 2^{14}\right), b_{\mathrm{h}} = b_{\mathrm{h}}^{\mathrm{cpu}} = \min\left(m, 2^{14}\right)$

przypadają 3 bloki. Wreszcie, dla $\sigma = 40$ występuje 79 bloków, a multiprocesory obciążone są następująco: 25 obsługuje 3 bloki, a 2 obsługują 2 bloki. Dlatego też czasy działania algorytmu dla $\sigma = 32$ oraz $\sigma = 40$ są mniej więcej takie same, a czas działania dla $\sigma = 24$ jest znacznie krótszy. Czas działania algorytmu sekwencyjnego dla procesora CPU zależy liniowo od σ. Przyspieszenie algorytmu CPU-p jest bliskie wartości teoretycznej dla długich ciągów, ale mniejsze dla ciągów krótkich. Dla małych alfabetów przyspieszenie CPU-p wynosi połowę wartości teoretycznej, ponieważ rdzenie procesora CPU nie są równomiernie obciążone. W problemie tym algorytm hybrydowy (CPU+GPU) byłby stosunkowo łatwy do implementacji, ponieważ problem LCTS można zdekomponować na niezależne podproblemy, a koszt komunikacji jest niewielki. Wymagany jest jedynie odpowiedni podział podproblemów pomiędzy procesory CPU i GPU. Można jednak oczekiwać, że hybrydowy algorytm osiągnąłby przyspieszenie co najwyżej 12, a więc o 50% więcej od przyspieszenia algorytmu GPU.

8.6. Podsumowanie

W niniejszym rozdziale zaproponowano kilka różnych algorytmów wyznaczania podciągu LCTS. Pierwsze dwa z nich (LCTS-NGMD oraz LCTS-HS) wykorzystują fakt, że sumaryczna liczba dopasowań dla wszystkich możliwych transpozycji wynosi tylko nm, co powoduje, że macierze programowania dynamicznego dla tego problemu są rzadkie. Algorytm LCTS-HS został zaproponowany w kilku wariantach, w zależności od rodzaju struktury danych wybranej do reprezentowania dopasowań dominujących o minimalnych rangach na każdym etapie algorytmu. Wyniki przeprowadzonych eksperymentów pokazują wyraźną przewagę tego algorytmu w stosunku do innych algorytmów znanych z literatury. Kolejnym zaproponowanym

algorytmem jest algorytm hybrydowy LCTS-Hybrid łączący algorytmy LCTS-HS oraz LCS-BP-Length. Pokazane zostało, jak za pomocą prostej heurystyki można połączyć zalety każdego z nich i uzyskać dzięki temu nawet dwukrotne przyspieszenie obliczeń w stosunku do szybszego z algorytmów składowych.

Ostatnią propozycją jest algorytm równoległy dla procesorów GPU oparty na algorytmie równoległości bitowej LCS-BP-Length. Po zastosowaniu tego samego ogólnego schematu obliczeń w procesorach GPU, który został zaproponowany w podrozdz. 7.6, stworzono algorytm, który w eksperymentach okazał się 6–8 razy szybszy od algorytmu LCS-BP-Length wykonywanego niezależnie dla każdej transpozycji. W oczywisty sposób algorytm ten jest także kilkakrotnie szybszy od sekwencyjnego algorytmu hybrydowego. W dostępnej literaturze nie były opisywane do tej pory próby stworzenia równoległego algorytmu dla problemu LCTS ani w wersji dla procesorów CPU, ani dla procesorów GPU.

9. Najdłuższy ukierunkowany wspólny podciąg

9.1. Wprowadzenie

Problem *najdłuższego ukierunkowanego wspólnego podciągu* (ang. *constrained longest common subsequence*, CLCS) został zdefiniowany w [203]. Jest on uogólnieniem problemu LCS lepiej sprawdzającym się w porównywaniu niektórych sekwencji biologicznych. W problemie CLCS występuje trzeci ciąg, który narzuca pewne ograniczenia na ciąg wynikowy.

Problem 9.1 (Najdłuższy ukierunkowany wspólny podciąg, CLCS)**.** *Dla ciągów $A = a_1a_2\ldots a_n$, $B = b_1b_2\ldots b_m$ oraz $P = p_1p_2\ldots p_r$ znaleźć najdłuższy ciąg S, który jest podciągiem zarówno A, jak i B, a jednocześnie P jest podciągiem S.*

Przykład 9.1 (Najdłuższy ukierunkowany wspólny podciąg, CLCS)**.** *Dla ciągów* $A = \mathsf{A\underline{B}AADA\underline{CBAAB}C}$, $B = \mathsf{C\underline{BCB}D\underline{AA}DCD\underline{B}A}$ *oraz* $P = \mathsf{CBB}$ *najdłuższym ukierunkowanym wspólnym podciągiem jest* $S = \mathsf{B\underline{CB}AA\underline{B}}$. *W ciągach A oraz B podkreślono symbole tworzące podciąg CLCS, a w ciągu S podkreślono symbole z ciągu P.*

Jednym z zastosowań problemu CLCS opisanym przez Tanga i in. [197] jest porównywanie sekwencji RNase, o których wiadomo, że zawierają trzy aktywne residua, His(H), Lyn(K), His(H), kluczowe dla degradacji RNA. Z tego powodu dla biologów interesujące są tylko wspólne podciągi, zawierające te trzy residua w takiej kolejności. O ciągu HKH mówi się, że „ukierunkowuje” poszukiwania wspólnych podciągów.

Problem CLCS jest silnie spokrewniony z problemem *ukierunkowanego uliniawiania ciągów* (ang. *constrained sequence alignment*, CSA) [197, 174, 110]. W problemie CSA poszukuje się takiego uliniowienia ciągów, w którym występują, w odpowiedniej kolejności, kolumny zawierające kolejne symbole ciągu ukierunkowującego. Możliwość nałożenia takiego ograniczenia jest kluczowa w sytuacjach, w których badacz (np. biolog) ma pewną wiedzę o tym, jak powinno wyglądać szukane uliniowienie. Dzięki temu może on z góry wykluczyć uliniowienia o dobrym wyniku, które jednak biologicznie nie mają znaczenia.

Podobnie jak dla poprzednich problemów, bez utraty ogólności założymy, że $m \le n$. Z oczywistych powodów musi zachodzić $r \le m$. *Dopasowaniem* będzie nazywana para (i, j), jeśli $a_i = b_j$. *Dopasowaniem silnym* będzie nazywana trójka (i, j, k) taka, że $a_i = b_j$ oraz $a_i = p_k$. *Rangą* trójki (i, j, k) będzie nazywana długość podciągu CLCS dla A_i, B_j, P_k. Ciągi A oraz B nazywane są ciągami *głównymi*, a ciąg P – ciągiem *ukierunkowującym*.

Dla problemu CLCS zostało zaproponowanych wiele algorytmów. Historycznie pierwszy z nich [203] jest całkowicie niepraktyczny z uwagi na ogromną złożoność zarówno czasową,

$O(n^2m^2r)$, jak i pamięciową. Bardzo szybko po nim pojawiła się jednak cała grupa algorytmów opartych na programowaniu dynamicznym, których złożoność czasowa jest $O(nmr)$, a pamięciowa – $O(nmr)$ lub lepsza. Najprostszym z nich jest algorytm China i in. [42]. Wyznacza się w nim macierz programowania dynamicznego M o wymiarach $(n+1)\times(m+1)\times(r+1)$ za pomocą następującej zależności:

$$M(i,j,k) = \begin{cases} M(i-1,j-1,k-1)+1, & \text{jeśli } i,j,k>0 \land a_i = b_j = p_k, \\ M(i-1,j-1,k)+1, & \text{jeśli } i,j>0, a_i = b_j \land \\ & \quad (k=0 \lor a_i \neq p_k), \\ \max(M(i-1,j,k), M(i,j-1,k)), & \text{jeśli } i,j>0 \land a_i \neq b_j. \end{cases} \tag{9.1}$$

Warunki brzegowe zdefiniowane są następująco:

$$\begin{aligned} M(i,0,0) &= 0, && \text{dla } 0 \le i \le n, \\ M(0,j,0) &= 0, && \text{dla } 0 \le j \le m, \\ M(i,0,k) &= -\infty, && \text{dla } 0 \le i \le n, 1 \le k \le r, \\ M(0,j,k) &= -\infty, && \text{dla } 0 \le j \le m, 1 \le k \le r. \end{aligned} \tag{9.2}$$

Ilustrację działania tego algorytmu przedstawia rys. 9.1. Macierz M, która jest tu wyznaczana, pojawia się także w niektórych innych algorytmach, w szczególności w algorytmach proponowanych w niniejszym rozdziale. Poniżej krótko zostaną przedstawione istniejące algorytmy rozwiązywania problemu CLCS. Nieco dokładniej zostaną pokazane te algorytmy, których poznanie będzie przydatne do zrozumienia algorytmów proponowanych w kolejnych podrozdziałach. Dokładniejsze omówienie pozostałych propozycji można znaleźć w [75].

Algorytm Penga [172] został zaproponowany niezależnie, ale wyznacza w zasadzie tę samą macierz programowania dynamicznego co algorytm China i in. [42]. Podobnie rzecz się ma z algorytmem Arslana–Eğecioğlu [22], w którym wprowadzono aż 4 macierze, ale istota obliczeń jest ta sama. Algorytm znacząco różniący się od wymienionych powyżej zaproponowali Peng i Ting [174]. Opiera się on na schemacie dziel i zwyciężaj. Obliczanie macierzy M wykonywane jest przez podział zadania na mniejsze części i następnie scalenie uzyskanych podmacierzy. Podejście to przypomina algorytm Hirschberga dla problemu LCS (por. podrozdz. 7.3.3). Algorytm Wanga [214] jest o tyle interesujący, że na etapie przetwarzania wstępnego analizuje się, które fragmenty trójwymiarowej macierzy programowania dynamicznego nie muszą być obliczane, bo i tak nie mogą wpłynąć w żaden sposób na końcowy wynik. W dalszej części, dzięki tym informacjom, redukuje się obszar koniecznych obliczeń.

Rozwinięciem algorytmu Arslana–Eğecioğlu jest algorytm Iliopoulosa–Rahmana [122]. Jego autorzy oparli się na tych samych zależnościach rekurencyjnych, ale zmodyfikowali sposób wyznaczania wartości komórek macierzy M w następujący sposób:

k=0

j \ i	0 A	1 B	2 A	3 A	4 D	5 A	6 C	7 B	8 A	9 A	10 B	11 C
0 C	0	0	0	0	0	0	1	1	1	1	1	1
1 B	0	1	1	1	1	1	1	2	2	2	2	2
2 C	0	1	1	1	1	1	2	2	2	2	2	3
3 B	0	1	1	1	1	1	2	3	3	3	3	3
4 D	0	1	1	1	2	2	2	3	3	3	3	3
5 A	1	1	2	2	2	3	3	3	4	4	4	4
6 A	1	1	2	3	3	3	3	3	4	5	5	5
7 D	1	1	2	3	4	4	4	4	4	5	5	5
8 C	1	1	2	3	4	4	5	5	5	5	5	6
9 D	1	1	2	3	4	4	5	5	5	5	5	6
10 B	1	2	2	3	4	4	5	6	6	6	6	6
11 A	1	2	3	3	4	5	5	6	7	7	7	7

k=1

j \ i	0 A	1 B	2 A	3 A	4 D	5 A	6 C	7 B	8 A	9 A	10 B	11 C
0 C	–	–	–	–	–	–	1	1	1	1	1	1
1 B	–	–	–	–	–	–	1	2	2	2	2	2
2 C	–	–	–	–	–	–	2	2	2	2	2	3
3 B	–	–	–	–	–	–	2	3	3	3	3	3
4 D	–	–	–	–	–	–	2	3	3	3	3	3
5 A	–	–	–	–	–	–	2	3	4	4	4	4
6 A	–	–	–	–	–	–	2	3	4	5	5	5
7 D	–	–	–	–	–	–	2	3	4	5	5	5
8 C	–	–	–	–	–	–	5	5	5	5	5	6
9 D	–	–	–	–	–	–	5	5	5	5	5	6
10 B	–	–	–	–	–	–	5	6	6	6	6	6
11 A	–	–	–	–	–	–	5	6	7	7	7	7

k=2

j \ i	0 A	1 B	2 A	3 A	4 D	5 A	6 C	7 B	8 A	9 A	10 B	11 C
0 C	–	–	–	–	–	–	–	–	–	–	–	–
1 B	–	–	–	–	–	–	–	2	2	2	2	2
2 C	–	–	–	–	–	–	–	2	2	2	2	3
3 B	–	–	–	–	–	–	–	3	3	3	3	3
4 D	–	–	–	–	–	–	–	3	3	3	3	3
5 A	–	–	–	–	–	–	–	3	4	4	4	4
6 A	–	–	–	–	–	–	–	3	4	5	5	5
7 D	–	–	–	–	–	–	–	3	4	5	5	5
8 C	–	–	–	–	–	–	–	3	4	5	5	6
9 D	–	–	–	–	–	–	–	3	4	5	5	6
10 B	–	–	–	–	–	–	–	6	6	6	6	6
11 A	–	–	–	–	–	–	–	6	7	7	7	7

k=3

j \ i	0 A	1 B	2 A	3 A	4 D	5 A	6 C	7 B	8 A	9 A	10 B	11 C
0 C	–	–	–	–	–	–	–	–	–	–	–	–
1 B	–	–	–	–	–	–	–	–	–	–	–	–
2 C	–	–	–	–	–	–	–	–	–	–	–	–
3 B	–	–	–	–	–	–	–	–	–	–	3	3
4 D	–	–	–	–	–	–	–	–	–	–	3	3
5 A	–	–	–	–	–	–	–	–	–	–	3	3
6 A	–	–	–	–	–	–	–	–	–	–	3	3
7 D	–	–	–	–	–	–	–	–	–	–	3	3
8 C	–	–	–	–	–	–	–	–	–	–	3	4
9 D	–	–	–	–	–	–	–	–	–	–	3	4
10 B	–	–	–	–	–	–	–	–	–	–	6	6
11 A	–	–	–	–	–	–	–	–	–	–	6	6

Rys. 9.1. Przykład działania algorytmu China i in. dla $A = \mathsf{ABAADACBAABC}$, $B = \mathsf{CBCBDAADCDBA}$, $P = \mathsf{CBB}$. Komórki zaznaczone na szaro oznaczają dopasowania silne, a symbol „–” oznacza wartość $-\infty$

Fig. 9.1. Example of the algorithm by Chin *et al.* for $A = \mathsf{ABAADACBAABC}$, $B = \mathsf{CBCBDAADCDBA}$, $P = \mathsf{CBB}$. Grayed cells denote strong matches and '–' symbols denote $-\infty$ values

$$M(i,j,k) = \max(M'(i,j,k), M''(i,j,k), M(i-1,j,k), M(i,j-1,k)), \tag{9.3}$$

$$V_1 = \max_{1 \le i' < i,\, 1 \le j' < j,\, a_{i'} = b_{j'}} (M(i',j',k-1)),$$

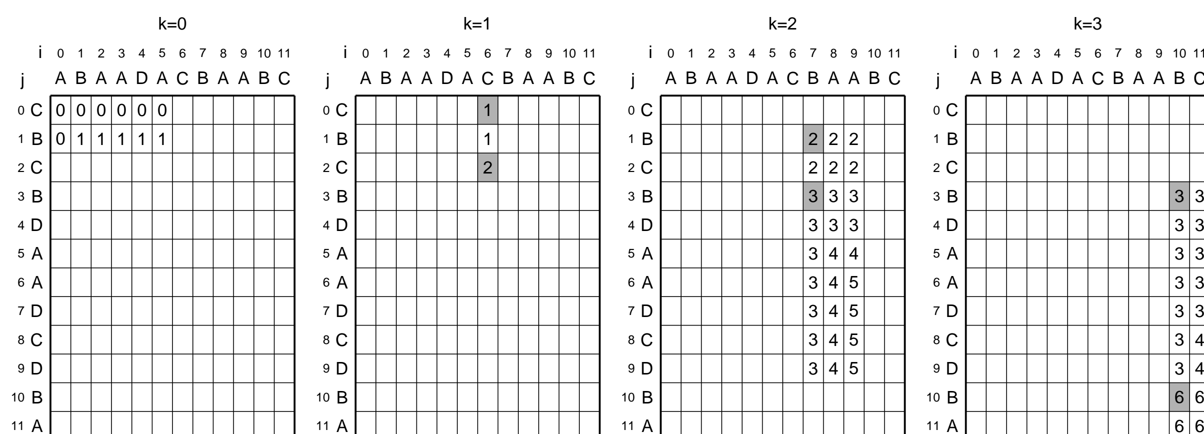
$$V_2 = \max_{1 \le i' < i,\, 1 \le j' < j,\, a_{i'} = b_{j'}} (M(i',j',k)),$$

$$M'(i,j,k) = \begin{cases} V_1 + 1, & \text{jeśli } (k = 1 \vee (k > 1 \wedge V_1 > 0)) \wedge a_i = b_j = p_k, \\ 0 & \text{w przeciwnym przypadku,} \end{cases}$$

$$M''(i,j,k) = \begin{cases} 1, & \text{jeśli } (i = 0 \vee j = 0) \wedge (a_i = b_j), \\ V_2 + 1, & \text{jeśli } (k = 0 \vee V_2 > 0) \wedge a_i = b_j, \\ 0 & \text{w przeciwnym przypadku.} \end{cases}$$

W celu wyznaczania wartości V_1 i V_2 autorzy wykorzystali strukturę danych *ograniczonego kopca* (ang. *bounded heap*) zaproponowaną przez Brodala i in. [36]. Dzięki temu czas wyznaczenia V_1 oraz V_2 jest $O(\log\log n)$. Ponadto, każda konieczna operacja modyfikacji tego kopca wykonywana jest w zamortyzowanym czasie $O(\log\log n)$ dla każdego dopasowania. Biorąc pod uwagę, że dopasowań jest $O(rd)$, sumaryczna złożoność czasowa tego algorytmu jest $O(rd \log\log n)$. W przypadku gdy rd jest istotnie mniejsze niż nmr, wynik ten jest interesujący, niestety w przypadku pesymistycznym złożoność czasowa tego algorytmu jest $\omega(nmr)$.

Idea podobna do tej, na której oparł się Wang [214], została zastosowana przez He i Arslana [110] dla pokrewnego problemu CSA. Poniżej metoda ta zostanie omówiona w wersji zaadaptowanej do algorytmu China i in. Macierz M obliczana jest zgodnie z równaniem (9.1), ale pomija się wyznaczanie wartości tych komórek, które nie mogą wpłynąć na końcowy wynik. W szczególności jeśli P_k nie jest podciągiem A_i, to $M(i,j,k) = -\infty$ dla dowolnego j. Co więcej, jeśli P_{-k} nie jest podciągiem A_{-i}, to wszystkie komórki $M(i-1,j,k-1)$ nie mają żadnego



Rys. 9.2. Przykład działania algorytmu China i in. z okrojonymi poziomami macierzy dla $A =$ ABAADACBAABC, $B =$ CBCBDAADCDBA, $P =$ CBB. (Komórki zaznaczone na szaro oznaczają dopasowania silne. Tylko okrojone fragmenty są wypełnione liczbami)

Fig. 9.2. Example of the algorithm by Chin *et al.* with trimmed levels for $A =$ ABAADACBAABC, $B =$ CBCBDAADCDBA, $P =$ CBB. (Grayed cells denote strong matches. Only the trimmed parts are filled with numbers)

wpływu na $M(n,m,r)$ (jest to dokładnie ten sam przypadek co poprzedni, w którym wszystkie ciągi zostały zapisane od końca). Podobną obserwację można poczynić dla ciągu B. Wobec tego na etapie przetwarzania wstępnego wyznacza się obszary macierzy, które można wykluczyć z obliczeń. Ilustrację tego podejścia można zobaczyć na rys. 9.2.

Dopasowania silne reprezentują jedyne komórki macierzy, których wartość zależy bezpośrednio od wartości komórek z innego poziomu macierzy. Po usunięciu ze zbioru wszystkich dopasowań silnych tych, które nie należą do okrojonych fragmentów, otrzymuje się zbiór tzw. *punktów wejścia-wyjścia* (ang. *entry-exit points*, EEP). He i Arslan pokazali [110], jak efektywnie wyznaczyć zbiór EEP i jak dzięki niemu zmniejszyć wymogi pamięciowe dla algorytmu rozwiązującego problem CSA. Oparli się tu na obserwacji, że jedynymi komórkami poprzedniego poziomu macierzy, których wartości są konieczne, aby wyznaczyć wartości komórek na bieżącym poziomie macierzy, są wartości dla komórek sąsiednich do EEP z bieżącego poziomu macierzy M. Technika EEP nie była do tej pory stosowana dla problemu CLCS.

9.2. Algorytm oparty na metodzie Hunta–Szymanskiego

9.2.1. Podstawowa idea

W niniejszym podrozdziale zaprezentowany zostanie opis algorytmu opartego na metodzie Hunta–Szymanskiego, który został zaproponowany przez autora w [64]. Algorytmy oparte na metodzie Hunta–Szymanskiego są jednymi z najefektywniejszych metod dla problemu LCS (por. podrozdz. 7.3.2). Sprawdzają się one szczególnie dobrze w sytuacji, w której liczba dopasowań jest stosunkowo niewielka, a więc macierz programowania dynamicznego jest rzad-

ka. Niestety, z uwagi na dopasowania silne i wprowadzane przez nie zależności pomiędzy komórkami z sąsiednich poziomów macierzy niemożliwe jest zastosowanie tej metody wprost do problemu CLCS. Jedną z adaptacji tej metody dla problemu CLCS jest algorytm Iliopoulosa–Rahmana [122] omówiony w poprzednim podrozdziale. Historycznie pierwsze zastosowanie tej metody Hunta–Szymanskiego zostanie jednak zaprezentowane poniżej.

Punktem wyjścia do rozważań jest algorytm China i in., a w szczególności zależności (9.1)–(9.2). Każda komórka $M(i,j,k)$ przechowuje długość podciągu CLCS dla A_i, B_j, P_k, a wartości komórek M nie maleją dla rosnących wartości indeksów. Pozwala to zapisać zależność (9.1) w następujący, równoważny, sposób przy zachowaniu niezmienionych warunków brzegowych (9.2):

$$M(i,j,k) = \begin{cases} \max\limits_{0 \le i' < i,\, 0 \le j' < j} M(i',j',k-1)+1, & \text{jeśli } i,j,k > 0 \wedge a_i = b_j = p_k, \\ \max\limits_{0 \le i' < i,\, 0 \le j' < j} M(i',j',k)+1, & \text{jeśli } i,j > 0, a_i = b_j \wedge (k = 0 \vee a_i \neq p_k), \\ \max\limits_{\substack{0 \le i' \le i,\, 0 \le j' \le j, \\ (i' \neq i \vee j' \neq j)}} M(i',j',k), & \text{jeśli } i,j > 0 \wedge a_i \neq b_j. \end{cases} \tag{9.4}$$

Przy wyznaczaniu podciągu CLCS można ograniczyć się tylko do komórek reprezentujących dopasowania, ponieważ dla niedopasowań wartość komórki jest tylko kopią wartości jednej z sąsiednich komórek. Wobec tego zależność (9.4) można przedstawić następująco:

$$M(i,j,k) = \begin{cases} \max\limits_{\substack{0 \le i' < i,\, 0 \le j' < j \\ a_{i'} = b_{j'}}} M(i',j',k-1)+1, & \text{jeśli } i,j,k > 0 \wedge a_i = b_j = p_k, \\ \max\limits_{\substack{0 \le i' < i,\, 0 \le j' < j \\ a_{i'} = b_{j'}}} M(i',j',k)+1, & \text{jeśli } i,j > 0, a_i = b_j \wedge (k = 0 \vee a_i \neq p_k). \end{cases} \tag{9.5}$$

Ponieważ dla $i = 0$ lub $j = 0$ nie ma dopasowań, konieczne jest zdefiniowanie w nieco inny sposób warunków brzegowych. Robi się to wprowadzając pseudodopasowanie $(0,0)$ (przyjmując, że $a_0 = b_0$), dla którego wartości komórek macierzy M zdefiniowane są następująco:

$$M(0,0,0) = 0 \quad \text{oraz} \quad M(0,0,k) = -\infty \quad \text{dla} \quad 1 \le k \le r. \tag{9.6}$$

Z powodów analogicznych jak dla algorytmu Hunta–Szymanskiego dla problemu LCS zachodzi

Lemat 9.1. *Wartości wszystkich komórek macierzy M reprezentujących dopasowania wyznaczone za pomocą zależności (9.5) oraz (9.6) są identyczne jak wyznaczone algorytmem China i in. – zależności (9.1) oraz (9.2).*

Przetwarzanie dopasowań odbywa się poziomami, a na każdym poziomie wierszami. Wprowadzona zostanie ponadto macierz T, obliczana równolegle z macierzą M, zgodnie z zależnością

$$T(i,j,k) = \max_{\substack{0 \le i' < i,\, 0 \le j' < j \\ a_{i'} = b_{j'}}} M(i',j',k) + 1 \tag{9.7}$$

dla wszystkich dopasowań. Wobec tego zależność (9.5) można zapisać:

$$M(i,j,k) = \begin{cases} T(i,j,k-1), & \text{jeśli } i,j,k > 0 \wedge a_i = b_j = p_k, \\ T(i,j,k), & \text{jeśli } i,j > 0, a_i = b_j \wedge (k = 0 \vee a_i \neq p_k). \end{cases} \tag{9.8}$$

W celu obliczenia wartości komórek macierzy T stosowany jest wariant algorytmu Hunta–Szymanskiego dla problemu LCS. Pojedynczy poziom macierzy przetwarzany jest wierszami od góry do dołu, a w ramach każdego wiersza od lewej do prawej strony. Dla bieżącego poziomu k (dla wszystkich k kolejno od 0 do r) i bieżącego wiersza j w liście L przechowywane są pary $\langle i',h\rangle$ (dla wszystkich poprawnych wartości h), gdzie i' jest najmniejszym takim indeksem kolumny, że przy ustalonych j, k długość podciągu CLCS dla $A_{i'}$, B_j, P_k wynosi h. Jest to więc najmniejsza taka wartość indeksu kolumny, że we fragmencie macierzy $M(0..i',j,k)$ występuje wartość h. Łatwo można zauważyć, że lista L odpowiada strukturze Q w algorytmie Hunta–Szymanskiego dla problemu LCS. (Rysunek 9.3 przedstawia zawartość listy L w czasie przetwarzania poziomu 0 macierzy M wiersz po wierszu). Po przetworzeniu wszystkich wierszy dla bieżącego poziomu k wartości macierzy T obliczone są w analogiczny sposób, jak robi to algorytm Hunta–Szymanskiego dla problemu LCS. Wobec tego:

Wniosek 9.1. *Wartości macierzy M dla dopasowań wyznaczone według zależności (9.8) są identyczne jak wyznaczone według zależności (9.5) oraz (9.6).*

Po wypełnieniu całej macierzy M wartość maksymalna w niej jest długością podciągu CLCS. W celu znalezienia podciągu CLCS potrzebna jest dodatkowo trójwymiarowa macierz F, w której dla każdej komórki (i,j,k) reprezentującej dopasowanie przechowywany jest wskaźnik do komórki macierzy M użytej do obliczenia wartości $M(i,j,k)$. Przejście po komórkach począwszy od tej z wartością maksymalną, zgodnie ze wskaźnikami znajdującymi się w macierzy F, pozwala odczytać podciąg CLCS, gdyż wartości a_i odwiedzanych komórek tworzą ciąg wynikowy czytany od końca.

Wniosek 9.2. *Powyżej opisany algorytm wyznacza poprawnie długość podciągu CLCS lub także podciąg CLCS, jeśli wypełniana jest macierz F.*

j \ i		1 A	2 B	3 A	4 A	5 D	6 A	7 C	8 B	9 A	10 A	11 B	12 C	
1	C							1					1	L = ⟨7, 1⟩
2	B		1						2			2		L = ⟨2, 1⟩, ⟨8, 2⟩
3	C							2					3	L = ⟨2, 1⟩, ⟨7, 2⟩, ⟨12, 3⟩
4	B		1						3			3		L = ⟨2, 1⟩, ⟨7, 2⟩, ⟨8, 3⟩
5	D					2								L = ⟨2, 1⟩, ⟨5, 2⟩, ⟨8, 3⟩
6	A	1		2	2		3			4	4			L = ⟨1, 1⟩, ⟨3, 2⟩, ⟨6, 3⟩, ⟨9, 4⟩
7	A	1		2	3		3			4	5			L = ⟨1, 1⟩, ⟨3, 2⟩, ⟨4, 3⟩, ⟨9, 4⟩, ⟨10, 5⟩
8	D					4								L = ⟨1, 1⟩, ⟨3, 2⟩, ⟨4, 3⟩, ⟨5, 4⟩, ⟨10, 5⟩
9	C							5					6	L = ⟨1, 1⟩, ⟨3, 2⟩, ⟨4, 3⟩, ⟨5, 4⟩, ⟨7, 5⟩, ⟨12, 6⟩
10	D					4								L = ⟨1, 1⟩, ⟨3, 2⟩, ⟨4, 3⟩, ⟨5, 4⟩, ⟨7, 5⟩, ⟨12, 6⟩
11	B		2						6			6		L = ⟨1, 1⟩, ⟨2, 2⟩, ⟨4, 3⟩, ⟨5, 4⟩, ⟨7, 5⟩, ⟨8, 6⟩
12	A	1		3	3		5			7	7			L = ⟨1, 1⟩, ⟨2, 2⟩, ⟨3, 3⟩, ⟨5, 4⟩, ⟨6, 5⟩, ⟨8, 6⟩, ⟨9, 7⟩

Rys. 9.3. Przykład działania algorytmu wyznaczającego macierz M dla problem CLCS dla $A =$ ABAADACBAABC, $B =$ CBCBDAADCDBA i $k = 0$. Aktualna zawartość listy L pokazana jest po przetworzeniu każdego wiersza

Fig. 9.3. Example of the algorithm calculating M matrix for the CLCS problem for $A =$ ABAADACBAABC, $B =$ CBCBDAADCDBA, and $k = 0$. The current content of the list L after finishing each row is also presented

9.2.2. Szczegóły implementacyjne i analiza złożoności

W opisie algorytmu występują trzy trójwymiarowe macierze oraz pewna liczba dodatkowych struktur (np. lista L) i zmiennych. Większość z nich została jednak wprowadzona tylko dla wygodniejszego opisu idei algorytmu. Najłatwiej zredukować pamięć dla macierzy T, można bowiem zauważyć, że zawsze wystarczy przechowywać tylko bieżący (k-ty) i poprzedni ($(k-1)$-szy) poziom tej macierzy. Ponadto, lista L oraz poprzedni poziom macierzy T wystarczają do wyznaczenia wartości bieżącego poziomu zarówno macierzy M, jak i T. Wartość maksymalną w całej macierzy M można otrzymać z ostatniej listy L po zakończeniu działania całego algorytmu. Dlatego też przechowywanie macierzy M jest zbyteczne. Wszystkie dopasowania dla A i B mogą być znalezione szybko: wystarczy tylko wyznaczenie wektora J^0 przechowującego liczbę wystąpień każdego symbolu alfabetu w A oraz macierzy J^1 zawierającej pozycje tych wystąpień. Wektor J^0 oraz macierz J^1 można zaimplementować z użyciem $\Theta(\max(n,\sigma))$ słów pamięci. Macierz T może być zaimplementowana z użyciem $\Theta(d)$ słów pamięci, gdzie d jest liczbą dopasowań dla A i B, ponieważ tylko d komórek tej macierzy jest wyznaczanych na jednym poziomie macierzy. Lista L zajmuje $O(\ell)$ słów pamięci, gdzie ℓ jest długością podciągu LCS dla A i B. Dla wariantu problemu, w którym wynikiem jest długość podciągu CLCS, są to wszystkie potrzebne struktury danych, a jeżeli wynikiem ma być także podciąg CLCS, to potrzeba $\Theta(rd)$ słów pamięci na macierz F. Wobec tego:

Wniosek 9.3. *Złożoność pamięciowa algorytmu opartego na matodzie Hunta–Szymanskiego wyznaczającego długość podciągu CLCS jest* $\Theta(d + \max(n,\sigma))$ *słów. Złożoność pamięciowa tego algorytmu w przypadku wyznaczania także podciągu CLCS jest* $\Theta(rd + \max(n,\sigma))$ *słów.*

Inicjalizacja wektora J^0 oraz macierzy J^1 wymaga $\Theta(\max(n,\sigma))$ czasu, co jest zaniedbywalne, jeśli $\sigma = O(n)$. (Przypadek $\sigma = \omega(n)$ będzie dyskutowany nieco niżej). Przetwarzanie każdego poziomu wymaga $O(m\ell + d)$ czasu, ponieważ w każdym wierszu przeglądana jest od lewej do prawej cała lista L, a dla każdego dopasowania wykonywany jest co najmniej jeden dostęp do listy L, ale nie więcej niż $(\ell +$ liczba dopasowań w wierszu$)$ dla wszystkich dopasowań w wierszu, ponieważ algorytm nigdy nie przesuwa się na liście w tył. Lista musi być także aktualizowana, ale wykonuje się to w prostym, liniowym przejściu po niej. Z powyższego wynika, że zarządzanie listą L wymaga sumarycznie (w całym algorytmie) $O((m\ell + d)r)$ czasu. Każdy dostęp do macierzy M, T, F wymaga czasu stałego. Ponadto, czas inicjalizacji J^0 i J^1 jest $O(n)$. Wobec powyższego można sformułować następujący wniosek:

Wniosek 9.4. *Złożoność czasowa algorytmu wyznaczania podciągu CLCS opartego na metodzie Hunta–Szymanskiego jest $O((m\ell + d)r + n)$.*

Jeśli $\sigma = \omega(n)$, to należy poczynić następujące modyfikacje. Ciąg A jest sortowany i usuwane są powtórzenia, przez co otrzymywany jest ciąg A^*. Następnie każdy symbol z A, B, P jest zastępowany swoim indeksem w A^*. Elementy ciągu B, które nie występują w A^*, zostają usunięte (nie wystąpią dla nich dopasowania). Takie przekształcenie ciągów wejściowych wymaga czasu $O(n\log n + m) = O(n\log n)$, a więc złożoność czasowa algorytmu w tym przypadku jest $O((ml + d)r + n\log n)$.

Pełny pseudokod algorytmu wyznaczającego długość podciągu CLCS pokazany jest na rys. 9.4. W wierszach 1–2 wyznaczane są pozycje wystąpień wszystkich symboli alfabetu w ciągu A. Ponieważ jest tylko n takich pozycji, więc macierz J^1 może być zaimplementowana z wykorzystaniem $\Theta(n)$ słów pamięci. Wiersze 3–31 zawierają główną pętlę algorytmu, w której macierz przechodzona jest poziomami. Inicjalizacja macierzy M jest tu uproszczona i sprowadza się do ustalenia jej wartości dla pseudodopasowań $(0,0,k)$ dla każdego k na: 0 $(k = 0)$ lub $-\infty$ $(k > 0)$. W pętli znajdującej się w wierszach 10–31 bieżący poziom przechodzony jest wierszami. Dla każdego wiersza, po skopiowaniu pseudodopasowania, przetwarzane są wszystkie dopasowania w nim się znajdujące. W wierszach 13–19 wykonywana jest kopia części listy L^0 do L^1. Następnie wstawiane jest do L^1 nowe dopasowanie (wiersze 20–26). W wierszach 27–30 wykonywana jest kopia pozostałej części listy L^0 do L^1. Wreszcie, listy L^0 i L^1 są zamieniane, dzięki czemu po zakończeniu przetwarzania bieżącego wiersza lista L^0 zawiera poprawne wartości rang dla dopasowań z przetworzonej części bieżącego poziomu. Lista L^1 jest następnie kasowana. Rysunek 9.5 zawiera pseudokod algorytmu znajdującego podciąg CLCS. Parametrami tego algorytmu są m.in. struktury L^0, F oraz $n_0 = |L^0|$ wyznaczone przez algorytm CLCS-HS-Length.

CLCS-HS-LENGTH(A, B, P)

Wejście: *A, B – ciągi główne, dla których wyznaczany jest podciąg CLCS*
P – ciąg ukierunkowujący
Wyjście: *długość podciągu CLCS*

```
   {Przetwarzanie wstępne}
 1 for i ← 0 to σ − 1 do J⁰[i] ← 0
 2 for i ← 1 to n do J¹[a_i, J⁰[a_i]] ← i;   J⁰[a_i] ← J⁰[a_i] + 1
   {Obliczenia właściwe}
 3 for k ← 0 to r do
 4     if k = 0 then
 5         L⁰[0].ℓ ← 0
 6     else
 7         L⁰[0].ℓ ← −∞
 8     L⁰[0].i ← −∞;   L⁰[0].j ← −∞
 9     n_0 ← 1;   n_1 ← 0
10     for j ← 1 to m do
11         L¹[n_1] ← L⁰[0];   n_1 ← n_1 + 1;   p ← 1
12         for x ← 0 to J⁰[b_j] − 1 do
13             i ← J¹[b_j][x]
14             while p < n_0 do
15                 if L⁰[p].i ≥ i then
16                     break
17                 else if L¹[n_1 − 1].ℓ < L⁰[p].ℓ and L⁰[p].ℓ > 0 then
18                     L¹[n_1] ← L⁰[p];   n_1 ← n_1 + 1
19                 p ← p + 1
20             if k > 0 and b_j = p_k then
21                 v ← T[i, j];   T[i, j] ← L⁰[p − 1].ℓ + 1
22             else
23                 v ← L⁰[p − 1].ℓ + 1;   T[i, j] ← v
24             F[i, j, k].i ← L⁰[p − 1].i;   F[i, j, k].j ← L⁰[p − 1].j
25             if L¹[n_1 − 1].ℓ < v then
26                 L¹[n_1].i ← i;   L¹[n_1].j ← j;   L¹[n_1].ℓ ← v;   n_1 ← n_1 + 1
27         while p < n_0 and L¹[n_1 − 1].ℓ ≥ L⁰[p].ℓ do
28             p ← p + 1
29         while p < n_0 do
30             L¹[n_1] ← L⁰[p];   n_1 ← n_1 + 1;   p ← p + 1
31     L⁰ ← L¹;   n_0 ← n_1;   n_1 ← 0
32 return L⁰[n_0 − 1].ℓ
```

Rys. 9.4. Algorytm wyznaczania długości podciągu CLCS oparty na metodzie Hunta–Szymanskiego
Fig. 9.4. Algorithm computing the length of the CLCS based on Hunt–Szymanski method

CLCS-HS-Sequence(A, B, P, L^0, n_0, F)

Wejście: *A, B – ciągi główne, dla których wyznaczany jest podciąg CLCS*
P – ciąg ukierunkowujący
L^0 *– wektor wyznaczony w algorytmie* CLCS-HS-Length
n_0 *– rozmiar wektora* L^0
F – macierz wyznaczona w algorytmie CLCS-HS-Length
Wyjście: *podciąg CLCS*

```
1  i ← L^0[n_0 − 1].i;   j ← L^0[n_0 − 1].j;   k ← r
2  for ℓ ← L^0[n_0 − 1].ℓ downto 1 do
3      s_ℓ ← b_j
4      if k > 0 and b_j = p_k then
5          k ← k − 1
6      i' ← F[i, j, k].i;   j' ← F[i, j, k].j
7      i ← i';   j ← j'
8  return s_1 s_2 ... s_{L^0[n_0−1].ℓ}
```

Rys. 9.5. Algorytm wyznaczania podciągu CLCS oparty na metodzie Hunta–Szymanskiego
Fig. 9.5. Algorithm computing the CLCS based on Hunt–Szymanski method

9.2.3. Zastosowanie techniki punktów wejścia-wyjścia

W [75] autor zaproponował algorytm adaptujący technikę EEP do algorytmu opartego na metodzie Hunta–Szymanskiego. Propozycja ta zostanie opisana w niniejszym podrozdziale. Ponieważ algorytm ten przetwarza macierz M poziomami, więc na etapie przetwarzania wstępnego (w czasie $O(n)$) wystarczy wyznaczyć, dla których dopasowań wyznaczanie wartości $M(i, j, k)$ jest zbędne. Złożoność czasowa zmodyfikowanego algorytmu nie zmienia się, bo w pesymistycznym przypadku redukcja rozmiaru macierzy M będzie znikoma.[1]

9.2.4. Wyniki eksperymentalne

Szczegóły implementacyjne i środowisko testowe

Implementując tak dużą liczbę algorytmów do praktycznych testów, istotne jest, aby były one zaprogramowane z podobną dbałością o szczegóły, które mogą mieć istotny wpływ na czas działania. W przypadku problemu CLCS szczególnie istotne jest efektywne zarządzanie pamięcią, gdyż dla ciągów o typowych rozmiarach zajętość pamięci może być bardzo duża. Przykładowo, w opisie algorytmu China i in. nie jest określone, w jaki sposób należy reprezentować

[1] Technika EEP nie została zastosowana dla pozostałych istniejących algorytmów wyznaczania podciągu CLCS z poniższych powodów. Algorytmy Penga i Arslan-Eğecioğlu stosują bardzo podobną zasadę obliczeń jak algorytm China i in., który we wstępnych eksperymentach okazał się szybszy, wobec czego także po zastosowaniu techniki EEP zmodyfikowany algorytm China i in. powinien być szybszy niż te dwa wspomniane i zmodyfikowane algorytmy. Algorytm Penga–Tinga opiera się na metodzie dziel-i-zwyciężaj i zastosowanie do niego techniki EEP nie jest łatwe. Algorytm Wanga wykorzystuje podobną optymalizację do techniki EEP. Algorytm Iliopoulosa–Rahmana oblicza macierz programowania dynamicznego w taki sposób, że nieistotne dopasowania są w nim traktowane specjalnie i zastosowanie techniki EEP nie przyniosłoby większych efektów, a byłoby skomplikowane.

macierz M i w jakiej kolejności ją obliczać. Dogodnie jest opisywać ten algorytm w taki sposób, że kolejne poziomy obliczane są jeden po drugim. Sugeruje to taką samą organizację macierzy M w pamięci operacyjnej. W praktyce rodzi to spore problemy związane z wymiataniem pamięci podręcznej (ang. *cache memory*). Jeśli (i, j, k) jest dopasowaniem silnym, to do obliczenia $M(i, j, k)$ konieczne będzie odczytanie $M(i-1, j-1, k-1)$, która to komórka w pamięci operacyjnej znajduje się około $4nm$ bajtów[2] wcześniej, co nawet dla ciągów o umiarkowanej długości może oznaczać megabajty. W tej sytuacji można mieć w zasadzie pewność, że wartości $M(i-1, j-1, k-1)$ już w pamięci podręcznej nie będzie.

Alternatywnym sposobem jest wirtualne przetransponowanie macierzy, tak aby wspomniane komórki znajdowały się w odległości $4rm$ bajtów, co w praktyce znacząco zwiększa szanse na szybszy dostęp do wartości $M(i-1, j-1, k-1)$, ponieważ r jest zwykle bardzo małą liczbą całkowitą. Wstępne eksperymenty pokazały, że taka zmiana organizacji macierzy w pamięci i co za tym idzie zmiana kolejności obliczeń powoduje ok. 20% redukcję czasu działania, wobec czego w dalszych eksperymentach użyto już tylko tej zmodyfikowanej wersji algorytmu.

Wszystkie algorytmy zostały zaimplementowane w języku C++. Do kompilacji użyto MS Visual C++ 2008 z opcją kompilacji -Ox (maksymalna optymalizacja ze względu na czas działania). Eksperymenty wykonano na komputerze wyposażonym w procesor AMD Phenom II X4 810 (zegar 2600 MHz) i 4096 MB RAM.

Zestawy danych i metodologia testów

W celu gruntownego porównania algorytmów, eksperymenty przeprowadzone zostały zarówno na danych rzeczywistych, jak i losowych.[3] Dane losowe zostały zastosowane, aby porównać zachowanie algorytmów dla różnych: długości głównych ciągów, długości ciągu ukierunkowującego, rozmiaru alfabetu. Dane zostały wygenerowane z zastosowaniem generatora liczb pseudolosowych o rozkładzie równomiernym. Do testów na danych rzeczywistych stworzony został korpus ciągów zaproponowanych w literaturze dla oceny algorytmów rozwiązywania pokrewnych problemów, takich jak CPSA i CMSA. Chin i in. w eksperymentach dla problemu CMSA [43] zaproponowali cztery zestawy testowe, zawierające sekwencje RNase o długościach z zakresu $[111, 327]$. Wszystkie te dane zostały włączone do stworzonego w niniejszym rozdziale korpusu. Włączono do niego także dane zaproponowane przez Lu i Huanga [148], którzy wykonywali eksperymenty także dla problemu CMSA. Dane te zawierają sekwencje z rodziny proteaz asparaginowych. Krótka charakterystyka stworzonego korpusu przedstawiona jest w tabeli 9.1[4]. Entropia rzędu 0 została wyznaczona dla każdego zestawu ciągów, aby

[2] Zakłada się tu, że pojedyncza komórka macierzy M jest przechowywana na 4 bajtach.
[3] Opisane tu eksperymenty zostały przeprowadzone w ramach pracy autora [75].
[4] Dane dostępne pod adresem `http://sun.aei.polsl.pl/~sdeor/pub/do09-ds.zip`

Tabela 9.1

Zestawy testowe z danymi rzeczywistymi użyte w eksperymentach dla problemu CLCS. Kolumna H_0 zawiera entropię rzędu 0 ciągów w bitach (entropia ciągu dla rozkładu równomiernego przy rozmiarze alfabetu 20 wynosi około 4,322 bita)

Zestaw danych	Liczba ciągów	Zawartość	Długość ciągów (min., med., maks.)	Rozm. alfab.	H_0	Źródło
ds0	7	H-RNase3, H-RNase2, BP-RNaseA, BS-RNase, H-RNaseA, H-RNase4, RC-RNase	(111, 124, 134)	20	4,172	[43]
ds1	6	gi|119124|sp|P12724|ecp_human, gi|2500564|sp|P70709|ecp_rat, gi|13400006|pdb|ldyt|, gi|20930966|ref|xp_142859.1|, gi|20873960|ref|xp_127690.1|, gi|20930966|ref|xp_142859.1|	(124, 149, 185)	20	4,246	[43]
ds2	6	gi|20930966|ref|XP_142859.1|, gi|119124|sp|P12724|ECP_HUMAN, gi|2500564|sp|P70709|ECP_RAT, gi|13400006|pdb, gi|20930966|ref|XP_142859.1|, gi|20873960|ref|XP_127690.1|	(131, 142, 160)	20	4,189	[43]
ds3	5	gi|10068295|gb|AAE40716.1|, gi|17549935|ref|NP_510780.1|, gi|28509297|ref|XP_282983.1|, gi|28499937|ref|XP_204162.2|, gi|4902995|dbj|BAA77929.1	(189, 277, 327)	20	4,191	[43]
ds4	6	Protease: HTLV-1, RSV, HIV-1, SRV-1, CaMV, 17.6	(98, 114, 123)	20	4,111	[148]

sprawdzić, jak bardzo częstość występowania symboli jest podobna do częstości występowania symboli w sekwencjach wygenerowanych przez generator o rozkładzie równomiernym.

W testach na danych losowych, przygotowano w każdym eksperymencie 201 trójek ciągów o założonej długości i rozmiarze alfabetu. Następnie jako czas działania każdego z algorytmów wzięto medianę z czasów działania dla tych 201 próbek. W testach na danych rzeczywistych każdy algorytm był wykonywany na każdej możliwej parze różnych ciągów należących do jednego zbioru danych przy założonym ciągu ukierunkowującym. Następnie wyznaczony został sumaryczny czas obliczeń dla każdego zestawu danych. Eksperymenty te zostały również powtórzone 201 razy i do wyznaczenia sumy czasów działania dla każdego zestawu danych wzięto mediany z czasów tych wykonań.

Algorytmy, które testowano, oznaczono na wykresach i w tabelach przez:

- AE – algorytm Arslan–Eğecioğlu [22],
- Chin – algorytm China i in. [42],
- ChinEE – algorytm Chin i in. z techniką EEP (podrozdz. 9.1),
- Deo – algorytm zaproponowany w podrozdz. 9.2.1,
- DeoEE – algorytm Deo z techniką EEP (podrozdz. 9.2.3),
- IR – algorytm Iliopoulosa–Rahmana [122],

- PT – algorytm Penga–Tinga [174],
- Wang – algorytm Wanga [214].

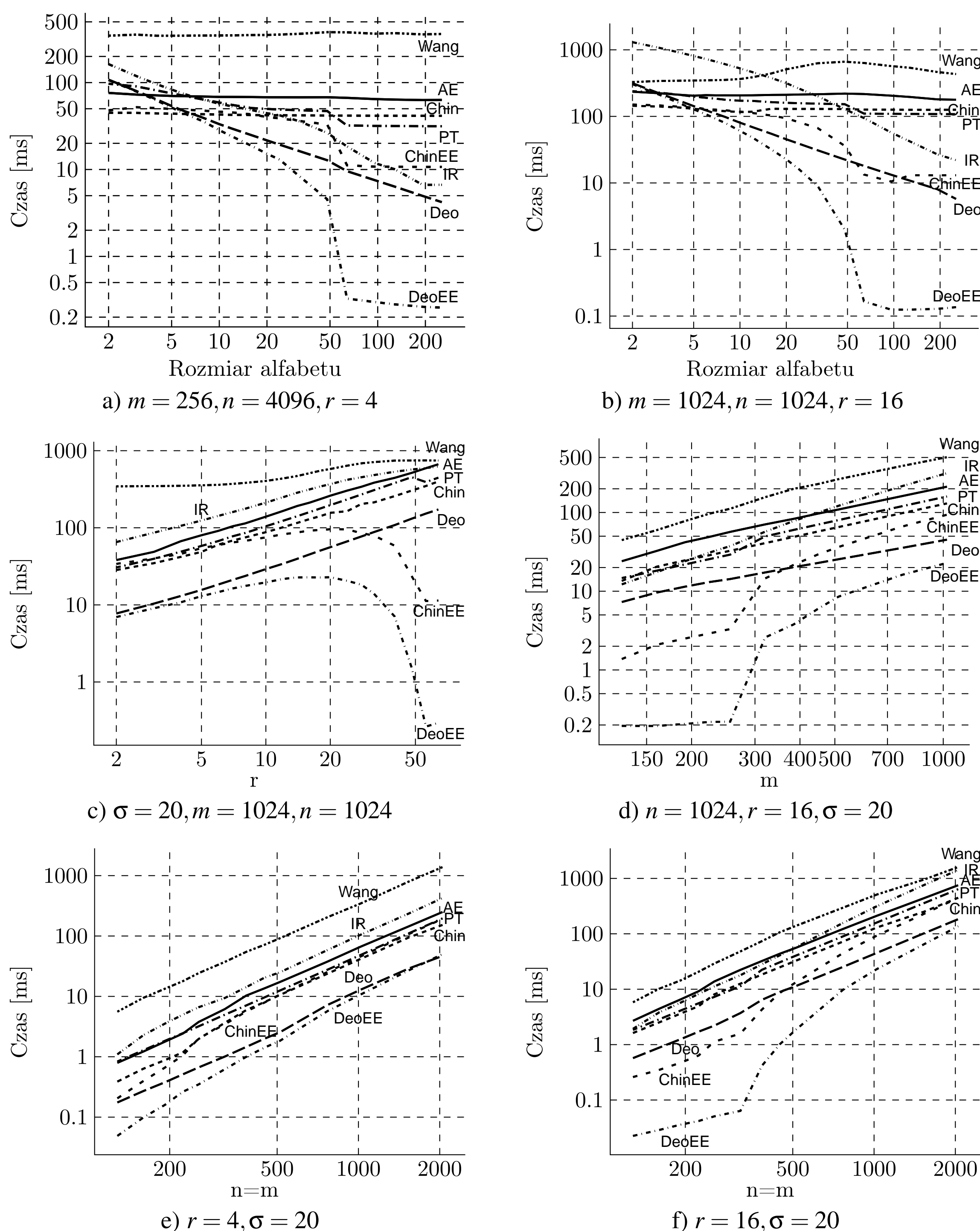
Wyznaczanie podciągu CLCS

W pierwszym eksperymencie, na danych losowych badano wpływ rozmiaru alfabetu na czas działania algorytmów. Przeprowadzono w tym celu testy dla różnych długości ciągów wejściowych (rys. 9.6a–b). Wprawdzie rozmiar alfabetu nie występuje wprost w złożonościach czasowych algorytmów, ale wpływa na czas ich działania pośrednio, ponieważ, zwłaszcza dla danych losowych, od rozmiaru alfabetu zależy liczba dopasowań. W przypadku algorytmów Deo, DeoEE, IR liczba dopasowań występuje jawnie we wzorze na złożoność czasową, a im większy rozmiar alfabetu, tym mniej dopasowań. Co więcej, rozmiar alfabetu wpływa na czas działania algorytmów stosujących technikę EEP (ChinEE, DeoEE) także w inny sposób. Większy rozmiar alfabetu oznacza bowiem krótszy ciąg wynikowy, mniejsze fragmenty macierzy do policzenia oraz mniej punktów wejścia-wyjścia (oczywiście jest to tylko wniosek przybliżony).

Dla alfabetów małych rozmiarów ($\sigma \leq 4$) najszybszy okazał się algorytm China i in. Wyznacza on zawsze całą trójwymiarową macierz programowania dynamicznego, ale robi to w sposób bardzo prosty. Bardziej wyrafinowane algorytmy, np. Deo, DeoEE, wykonują o wiele więcej obliczeń dla każdego dopasowania, a ponieważ dopasowań jest dużo, to ich czas działania jest gorszy niż czas działania algorytmu Chin. Dla dużych alfabetów, algorytmy Deo i DeoEE wygrywają w sposób zdecydowany. Również algorytmy IR i ChinEE okazują się być stosunkowo szybkie w tej sytuacji, dzięki wyznaczaniu tylko fragmentu macierzy (ChinEE) lub niewielkiej liczbie dopasowań (IR).

Głównym obszarem zastosowań problemu CLCS jest bioinformatyka, w której danymi wejściowymi są m.in. sekwencje białkowe i RNase, dla których $\sigma = 20$. Częstość występowania symboli w rzeczywistych danych nie jest równomierna, ale nie różni się od niej w sposób bardzo znaczący, więc w kolejnych eksperymentach użyto losowych sekwencji o rozmiarze alfabetu 20. Na rys. 9.6 przedstawiono wpływ: (c) długości ciągu ukierunkowującego, (d) długości krótszego głównego ciągu, (e)–(f) długości obu ciągów głównych na czasy działania algorytmów.

Jak można było oczekiwać, algorytmy niestosujące techniki EEP działają coraz dłużej, w miarę jak wydłuża się ciąg ukierunkowujący, co wynika z proporcjonalności czasów ich działania do długości tego ciągu. Czasy działania algorytmów ChinEE oraz DeoEE też początkowo rosną wraz ze wzrostem długości ciągu ukierunkowującego, ale później zdecydowanie maleją, ponieważ długi ciąg ukierunkowujący znacząco redukuje obszar macierzy do obliczenia. W przypadku ciągów głównych jednakowej długości zdecydowanie najszybsze okazały się algorytmy Deo



a) $m = 256, n = 4096, r = 4$

b) $m = 1024, n = 1024, r = 16$

c) $\sigma = 20, m = 1024, n = 1024$

d) $n = 1024, r = 16, \sigma = 20$

e) $r = 4, \sigma = 20$

f) $r = 16, \sigma = 20$

Rys. 9.6. Porównanie czasów działania algorytmów wyznaczających podciąg CLCS dla różnych rozmiarów alfabetu

Fig. 9.6. Comparison of the CLCS computing time for changing alphabet size

Tabela 9.2
Porównanie czasów działania (w ms) algorytmów wyznaczania podciągu CLCS dla danych rzeczywistych. W nawiasach: ile razy algorytm jest szybszy od algorytmu Chin

Zestaw	*P*	AE	Chin	ChinEE	Deo	DeoEE	IR	PT	Wang
ds0	HKH	10,827 (0,55)	5,969 (1,00)	3,871 (1,54)	3,012 (1,98)	**1,117** (**5,34**)	22,084 (0,27)	12,504 (0,48)	116,018 (0,05)
ds1	HKH	11,964 (0,56)	6,662 (1,00)	5,442 (1,22)	3,227 (2,06)	**1,567** (**4,25**)	23,247 (0,29)	13,557 (0,49)	137,817 (0,05)
ds1	HKSH	16,055 (0,51)	8,258 (1,00)	5,030 (1,64)	4,012 (2,06)	**1,423** (**5,81**)	26,603 (0,31)	16,584 (0,50)	137,823 (0,06)
ds1	HKSTH	19,760 (0,50)	9,818 (1,00)	4,831 (2,03)	4,801 (2,04)	**1,393** (**7,05**)	30,574 (0,32)	19,360 (0,51)	143,683 (0,07)
ds2	HKSH	13,631 (0,52)	7,047 (1,00)	3,745 (1,88)	3,534 (1,99)	**1,070** (**6,59**)	24,701 (0,29)	14,447 (0,49)	110,691 (0,06)
ds2	HKSTH	16,770 (0,50)	8,386 (1,00)	3,310 (2,53)	4,207 (1,99)	**0,915** (**9,16**)	27,752 (0,30)	16,964 (0,49)	113,112 (0,07)
ds3	HKH	27,313 (0,63)	17,220 (1,00)	15,676 (1,10)	5,868 (2,93)	**2,878** (**5,98**)	43,931 (0,39)	24,778 (0,69)	280,309 (0,06)
ds4	DGGG	8,461 (0,52)	4,376 (1,00)	2,502 (1,75)	2,329 (1,88)	**0,791** (**5,53**)	14,014 (0,31)	9,178 (0,48)	69,436 (0,06)

oraz DeoEE, a różnica pomiędzy nimi maleje wraz ze wzrostem długości ciągów głównych. Wynika to z proporcjonalnie mniejszej redukcji obszarów obliczanych macierzy po zastosowaniu techniki EEP.

Wyniki eksperymentalne dla danych rzeczywistych przedstawione są w tabeli 9.2. Potwierdzają one wnioski wyciągnięte przy symulacji danych biologicznych ciągami losowymi dla rozmiaru alfabetu 20. Najszybszy algorytm, DeoEE, jest 2,0–4,7 razy szybszy niż Deo i 3,2–5,4 razy szybszy niż ChinEE. Pozostałe algorytmy są znacznie wolniejsze. Jak można zaobserwować, dla danych z zestawu ds1 względna prędkość działania algorytmu DeoEE w stosunku do algorytmu Chin wzrasta wraz ze wzrostem długości ciągu ukierunkowującego. Ponadto, czasy działania algorytmów stosujących technikę EEP (ChinEE i DeoEE) są często mniejsze dla dłuższych ciągów ukierunkowujących, dzięki większej redukcji obszaru macierzy, który należy obliczyć.

Wyznaczanie długości podciągu CLCS

Jeśli wyznaczanie podciągu CLCS jest zbędne i wystarcza znajomość jego długości, algorytmy mogą działać znacznie szybciej, głównie dzięki lepszej organizacji pamięci. Przykładowo, w algorytmie China i in. wystarczy przechowywać tylko dwa poziomy macierzy M (bieżący i poprzedni). Powoduje to m.in. lepsze wykorzystanie pamięci podręcznej. Ponadto, aby wyznaczyć podciąg CLCS, konieczne są pewne struktury danych, dzięki którym możliwe jest przejście po tej macierzy i odczytanie ciągu wynikowego (np. macierz F w algorytmie Deo). Struktury te

Tabela 9.3
Porównanie czasów działania (w ms) algorytmów wyznaczania długości podciągu CLCS dla danych rzeczywistych. W nawiasach: ile razy algorytm jest szybszy od algorytmu Chin

Zestaw	*P*	AE	Chin	ChinEE	Deo	DeoEE	IR	PT	Wang
ds0	HKH	9,922 (0,55)	5,446 (1,00)	3,078 (1,77)	2,625 (2,07)	**0,922** (**5,90**)	20,403 (0,27)	12,935 (0,42)	17,995 (0,30)
ds1	HKH	10,822 (0,56)	6,036 (1,00)	4,333 (1,39)	2,810 (2,15)	**1,310** (**4,61**)	21,375 (0,28)	14,190 (0,43)	20,686 (0,29)
ds1	HKSH	16,407 (0,46)	7,565 (1,00)	3,917 (1,93)	3,499 (2,16)	**1,159** (**6,53**)	24,405 (0,31)	17,278 (0,44)	20,620 (0,37)
ds1	HKSTH	19,270 (0,47)	9,051 (1,00)	3,734 (2,42)	4,193 (2,16)	**1,128** (**8,03**)	28,003 (0,32)	20,488 (0,44)	21,539 (0,42)
ds2	HKSH	13,659 (0,47)	6,486 (1,00)	2,940 (2,21)	3,068 (2,11)	**0,869** (**7,46**)	22,703 (0,29)	15,016 (0,43)	16,434 (0,39)
ds2	HKSTH	16,037 (0,48)	7,739 (1,00)	2,526 (3,06)	3,666 (2,11)	**0,720** (**10,75**)	25,416 (0,30)	17,717 (0,44)	16,809 (0,46)
ds3	HKH	20,497 (0,54)	10,986 (1,00)	8,328 (1,32)	5,020 (2,19)	**2,389** (**4,60**)	40,547 (0,27)	25,859 (0,42)	38,216 (0,29)
ds4	DGGG	8,858 (0,46)	4,040 (1,00)	1,986 (2,03)	2,019 (2,00)	**0,645** (**6,26**)	12,860 (0,31)	9,447 (0,43)	10,614 (0,38)

są zbyteczne, jeśli interesuje nas tylko długość podciągu CLCS, co zmniejsza liczbę dostępów do pamięci (nie trzeba tych struktur wypełniać ani odczytywać).

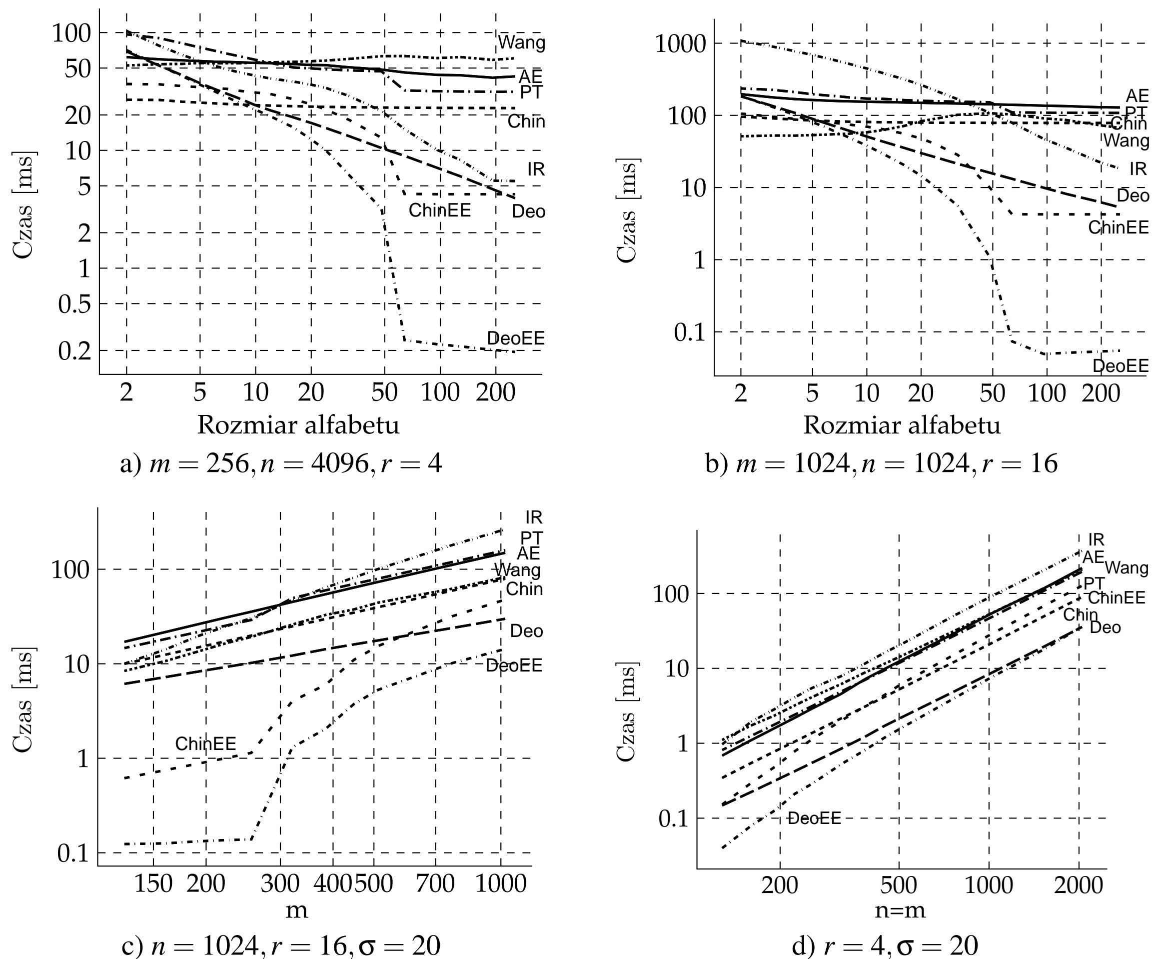
Także w tym przypadku dla danych rzeczywistych (tab. 9.3) najszybszym algorytmem okazał się DeoEE, który jest szybszy od algorytmu Deo od 2,1- do 5,1-krotnie, a od algorytmu ChinEE od 3,1- do 3,5-krotnie. Pozostałe algorytmy są znacznie wolniejsze. Warto również odnotować, że większość algorytmów wyznaczania długości podciągów CLCS jest szybsza od swoich odpowiedników wyznaczających podciągi CLCS o około 20%. Dla danych losowych (rys. 9.7) algorytmy dla tej łatwiejszej wersji problemu są szybsze od 1,5 (Deo) do 6,0 (Wang) razy. Niezmiennie jednak dla typowego rozmiaru alfabetu ($\sigma = 20$) oraz alfabetów jeszcze większych najszybszymi algorytmami pozostają DeoEE, Deo i ChinEE.

9.3. Algorytm równoległości bitowej

9.3.1. Podstawy

Niniejszy podrozdział zawiera omówienie algorytmu równoległości bitowej zaproponowanego przez autora w [69]. Jest to, jak do tej pory, jedyny taki algorytm dla tego problemu.

Różnice pomiędzy problemami CLCS i LCS są na tyle duże, że nie jest możliwe proste uogólnienie algorytmu równoległości bitowej dla problemu LCS (podrozdz. 7.3.4) do problemu CLCS. Podobnie jak w przypadku problemu LCS, punktem wyjścia do zrównoleglania



Rys. 9.7. Porównanie czasów wyznaczania długości podciągów CLCS dla różnych zestawów parametrów

Fig. 9.7. Comparison of the CLCS length computing time for various parameters

bitowego będzie, koncepcyjnie najprostszy, algorytm China i in. [42]. Na podstawie zależności (9.1) i (9.2) wyznacza się w nim trójwymiarową macierz M.

Najważniejsze trudności przy zrównoleglaniu bitowym, które tu występują to:

- Różnice pomiędzy sąsiednimi komórkami macierzy M mogą być większe niż 1 (w przeciwieństwie do problemu LCS), a więc, w ogólności, nie jest możliwe przechowywanie tej różnicy za pomocą pojedynczego bitu.
- Macierz M składa się z $r+1$ poziomów, które nie są niezależne, ponieważ wartość komórki macierzy M dla dopasowania silnego zależy od wartości komórki z poprzedniego poziomu.

Powyższe problemy są konsekwencją występowania dopasowań silnych. Jeśli (i, j, k) jest dopasowaniem silnym, to wartość $M(i, j, k)$ jest o 1 większa od długości podciągu CLCS dla $A_{i-1}, B_{j-1}, P_{k-1}$, podczas gdy wartość $M(i, j-1, k)$ jest długością podciągu CLCS dla A_i, B_{j-1}, P_k. Dlatego też różnica pomiędzy $M(i, j, k)$ a $M(i, j-1, k)$ może być duża. Ciąg ukierunkowujący ma duży wpływ na długość podciągu CLCS, o czym można się przekonać porównując wartości $M(i, j, k)$ dla ustalonych i, j oraz różnego k na rys. 9.1.

Jeśli (i, j, k) nie jest dopasowaniem silnym, to $M(i, j, k) - M(i, j-1, k) \in \{0, 1\}$, jeśli $b_j \neq p_k$ i $M(i, j, k) - M(i-1, j, k) \in \{0, 1\}$, jeśli $a_i \neq p_k$. Wynika to z faktu, że dla wszystkich wymienionych komórek ciąg ukierunkowujący jest identyczny, a wówczas skrócenie jednego z ciągów A lub B o ostatni symbol nie może spowodować zmiany długości podciągu CLCS dla nich o więcej niż 1.

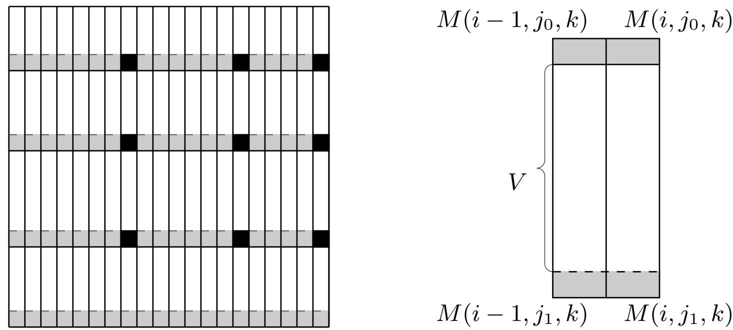
Podstawową ideą algorytmu proponowanego w niniejszym podrozdziale jest wyznaczanie wprost wartości tylko tych komórek macierzy M, które mogą się różnić o więcej niż 1 od wartości komórki będącej górnym sąsiadem oraz zastosowanie szybkiego algorytmu równoległości bitowej do wyznaczania pozostałej części macierzy. Warto tu zaznaczyć, że komórki, dla których wartość musi być wyznaczona wprost, należą do wierszy zawierających dopasowania silne.

9.3.2. Algorytm paskowy

Proponowany algorytm wyznacza długość podciągu CLCS w podobny sposób jak algorytm LCS-BP-Length wyznacza długość podciągu LCS (podrozdz. 7.3.4), choć różnice pomiędzy tymi algorytmami są istotne. Tworzona jest tu macierz M, w której przechowywane są wprost wartości niektórych komórek. W dalszej części podrozdziału zostanie pokazane, jak tę trójwymiarową macierz można implementować za pomocą dwóch kolejek typu FIFO.

Macierz M przetwarzana jest poziomami. Na początku zostanie rozważony przypadek pojedynczego poziomu dla $k > 0$. Algorytm dzieli nieujemną część tej dwuwymiarowej macierzy na *paski* (ang. *strips*). Pasek jest prostokątnym obszarem poziomu macierzy M o szerokości pojedynczej kolumny i pewnej wysokości. Jest on definiowany przez czwórkę: $\langle i, j_0, j_1, k\rangle$. Wertykalne ułożenie pasków jest determinowane przez pozycje dopasowań silnych. Każdy pasek kończy się na wierszu opisanym przez symbol p_k (tzn. dla takiego j_1, że $b_{j_1} = p_k$) i zaczyna zaraz za poprzednim takim wierszem o indeksie j_0. Dla kompletności podziału niektóre paski kończą się na dolnym końcu macierzy M, nawet jeśli $b_m \neq p_k$. Komórki znajdujące się w pierwszym wierszu zawierającym wartości nieujemne nie należą do żadnego paska. Zgodnie z definicją paska, jedynym miejscem w nim, w którym może wystąpić dopasowanie silne, jest jego koniec. Wobec tego możliwe jest reprezentowanie za pomocą wektora bitowego $V_{j_0}(i)$ różnic pomiędzy wartościami $M(i, j_0, k), \ldots, M(i, j_1 - 1, k)$. Bity o wartości 0 w $V_{j_0}(i)$ reprezentują (podobnie jak w algorytmie LCS-BP-Length) najmniejsze możliwe indeksy wierszy dla dopasowań o kolejnych rangach w przetworzonej części macierzy M. Znalezienie fragmentu macierzy M, w którym wszystkie komórki zawierają wartość $-\infty$ jest stosunkowo proste. Wykonuje się to z użyciem procedury CLCS-BP-Start-Point (rys. 9.9), a znaleziony fragment macierzy jest wykluczany z dalszych obliczeń.

Wyznaczenie każdego poziomu macierzy wykonuje się kolumna po kolumnie, a w ramach każdej kolumny pasek po pasku od góry do dołu. Kluczowym elementem całego algorytmu jest



Rys. 9.8. Idea działania algorytmu paskowego równoległości bitowej dla problemu CLCS. Wiersze zaznaczone na szaro zawierają dopasowania silne (czarne kwadraty). Wartości dla tych wierszy są wyznaczane wprost. Wertykalnie ułożone paski kończą się na wierszach zaznaczonych na szaro. Ich wnętrze wyznaczane jest w sposób bitowo-równoległy

Fig. 9.8. An idea of the strip-based bit-parallel algorithm for the CLCS problem. The grayed rows contain strong matches (black boxes) and the values for these rows are computed explicitly. The vertical strips ending at grayed rows are computed in a bit-parallel way

CLCS-BP-Start-Point(A, B, P, k)

Wejście: *A, B – ciągi główne, dla których wyznaczany jest podciąg CLCS*
P – ciąg ukierunkowujący
k – indeks poziomu macierzy programowania dynamicznego
Wyjście: *indeksy punktu startowego na poziomie o indeksie k macierzy programowania dynamicznego*

```
1   i* ← 1;   k* ← 1
2   while i* ≤ n and k* ≤ k do
3       if a_{i*} = p_{k*} then k* ← k* + 1
4       i* ← i* + 1
5   if k* ≤ k then
6       return n + 1, m + 1
7   j* ← 1;   k* ← 1
8   while j* ≤ m and k* ≤ k do
9       if b_{j*} = p_{k*} then k* ← k* + 1
10      j* ← j* + 1
11  if k* ≤ k then
12      return n + 1, m + 1
13  return i* − 1, j* − 1
```

Rys. 9.9. Algorytm znajdujący punkt startowy dla poziomu k (problem CLCS, algorytm równoległości bitowej)

Fig. 9.9. Finding starting point for the level k (CLCS problem, bit-parallel algorithm)

wykonanie obliczeń dla pojedynczego paska. Danymi wejściowymi są m.in. obliczone wcześniej paski górny i lewy. Dla każdego paska $\langle i, j_0, j_1, k\rangle$ aktualizowane są następujące wartości:

- $V_{j_0}(i)$ – wektor bitowy reprezentujący na kolejnych bitach informację, czy $M(i, j, k) = M(i, j-1, k)$ (bit o wartości 1), czy też $M(i, j, k) \neq M(i, j-1, k)$ (bit o wartości 0) dla $j_0 < j < j_1$,
- $C_{j_0}(i)$ – liczba bitów wartości 0 w wektorze $V_{j_0}(i)$,
- $M(i, j_1, k)$ – długość podciągu CLCS dla ciągów A_i, B_{j_1}, P_k.

CLCS-BP-STRIP($A, B, P, M, C, V, i, j_0, j_1, k$)

Wejście: *A, B – ciągi główne, dla których wyznaczany jest podciąg CLCS*
P – ciąg ukierunkowujący
M – macierz programowania dynamicznego
C – macierz liczników bitów o wartości 0 w odpowiednim wektorze bitowym w macierzy V
V – macierz wektorów bitowych
i – indeks bieżącej kolumny
j_0, j_1 *– zakres indeksów określających pojedynczy pasek*
k – indeks poziomu macierzy programowania dynamicznego
Wyjście: *zaktualizowane wartości w macierzach M, C, V*

1 $U \leftarrow V_{j_0}(i-1)$ & Y_{a_i,j_0}
2 $V_{j_0}(i) \leftarrow (V_{j_0}(i-1)+U) \mid (V_{j_0}(i-1)-U)$
3 $C_{j_0}(i) \leftarrow C_{j_0}(i-1)+$ w_poprzeniej_linii_pojawiło_się_nowe_0
4 Usuń $M(i,j_0,k)-M(i-1,j_0,k)$ najmniej znaczących bitów 0 z $V_{j_0}(i)$
oraz zaktualizuj odpowiednio $C_{j_0}(i)$
5 **if** (i,j_1,k) jest dopasowaniem silnym **then**
6 $M(i,j_1,k) \leftarrow M(i-1,j_1-1,k-1)+1$
7 **else**
8 $M(i,j_1,k) \leftarrow \max(M(i,j_0,k)+C_{j_0}(i), M(i-1,j_1,k))$
9 **for each** dopasowanie silne $(i+1,j+1,k+1)$ gdzie $j \in [j_0,j_1]$ **do**
10 Wylicz $M(i,j,k)$ na podstawie $M(i,j_0,k)$ i $V_{j_0}(i)$

Rys. 9.10. Algorytm równoległości bitowej wyliczający pojedynczy pasek (problem CLCS)
Fig. 9.10. Bit-parallel strip computing algorithm (CLCS problem)

Lemat 9.2. *Algorytm* CLCS-BP-STRIP *(rys. 9.10) prawidłowo wylicza* $V_{j_0}(i)$, $C_{j_0}(i)$, $M(i,j_1,k)$ *oraz wartości* $M(i,j,k)$ *dla wszystkich* $j_0 < j \leq j_1$ *takich, że* $(i+1,j+1,k+1)$ *jest dopasowaniem silnym, jeśli znane są wartości:* $M(i-1,j_0,k)$, $M(i,j_0,k)$, $V_{j_0}(i-1)$, $C_{j_0}(i-1)$ *i* $M(i-1,j_1-1,k-1)$, *jeśli* (i,j_1,k) *jest dopasowaniem silnym.*

Dowód. Wartość $M(i-1,j_0,k)$ jest największa spośród rang dopasowań (i',j',k) dla $0 < i' \leq i-1$, $0 < j' \leq j_0$. Każdy s-ty bit o wartości 0, dla wszystkich poprawnych s, w wektorze $V_{j_0}(i-1)$ na pozycji bitowej j (licząc od j_0) jest najmniejszym indeksem wiersza zawierającego dopasowanie rangi $M(i-1,j_0,k)+s$ dla dopasowań (i',j',k), gdy $0 < i' \leq i-1$ oraz $0 < j' \leq j_1$.

Bit o indeksie j w wektorze Y_{a_i} (dla $j_0 < j \leq j_1$) ma wartość 1 wtedy i tylko wtedy, gdy $a_i = b_j$. Przetwarzanie bitowe w wierszach 1–2 jest identyczne jak w algorytmie Hyyrö [120] (podrozdz. 7.3.4). W wierszu 3. wyznaczana jest liczba bitów o wartości 0 w wektorze $V_{j_0}(i)$ (może ona być równa $C_{j_0}(i-1)$ lub większa o 1, jeśli właśnie pojawiło się nowe 0 w wektorze $V_{j_0}(i)$). W związku z tym, po wykonaniu instrukcji z tych wierszy, bity wartości 0 w $V_{j_0}(i)$ reprezentują indeksy wierszy dopasowań o rangach począwszy od $M(i-1,j_0,k)$. $M(i,j_0,k)$ (największa spośród rang dopasowań (i',j',k) dla $0 < i' \leq i$ oraz $0 < j' \leq j_0$) może jednak być większa niż $M(i-1,j_0,k)$. Przykładowo, jeśli (i,j_0,k) jest dopasowaniem silnym, to $M(i,j_0,k)-M(i-1,j_0,k)$ może być nawet większe niż 1. Dlatego też $M(i,j_0,k)-M(i-1,j_0,k)$ najmniej znaczących bi-

tów wartości 0s jest usuwanych z $V_{j_0}(i)$, aby zapewnić, że dopasowania z $V_{j_0}(i)$ odzwierciedlają rangi począwszy od $M(i, j_0, k)$ (wiersz 4.). Odpowiednio aktualizowana jest też wartość $C_{j_0}(i)$.

Następnie wyznaczana jest wartość $M(i, j_1, k)$. Jeśli (i, j_1, k) nie jest dopasowaniem silnym, to wartość $M(i, j_1, k)$ jest obliczana jako suma $M(i, j_0, k)$ i liczby bitów o wartości 0 w bieżącym wektorze V. W przeciwnym przypadku jest ona obliczana na podstawie wartości $M(i-1, j-1, k-1)$ z poprzedniego poziomu (wiersze 5–8).

W etapie końcowym (wiersze 9–10) wyznaczane są wartości $M(i, j, k)$, dla $j_0 < j \leq j_1$, na postawie $M(i, j_0, k)$ i bieżącej zawartości wektora V, jeśli $(i+1, j+1, k+1)$ są dopasowaniami silnymi (te wartości macierzy M będą musiały być znane wprost do wyznaczenia obliczenia poziomu macierzy). ■

Wyznaczanie długości podciągu CLCS odbywa się poziom po poziomie. Dysponując procedurą obliczania pojedynczego paska, można teraz rozważyć kwestię obliczenia całego poziomu, przy założeniu że $k \geq 1$. Przypadek $k = 0$ będzie dyskutowany osobno.

Lemat 9.3. *Jeśli dla wszystkich dopasowań silnych (i, j, k) wartości $M(i-1, j-1, k-1)$ są poprawnie wyznaczone, to algorytm* CLCS-BP-Level *(rys. 9.11) obliczający pojedynczy poziom macierzy M dla $k > 0$ poprawnie wyznacza wartości $M(i, j, k)$ dla wszystkich takich trójek (i, j, k), że $(i+1, j+1, k+1)$ jest dopasowaniem silnym.*

Dowód. Niech niezmiennikiem głównej pętli algorytmu (wiersze 18–20) będzie:

- $M(i, j, k)$ zawiera największą z rang dopasowań (i', j', k) takich, że $0 < i' \leq i$, $0 < j' \leq j$ oraz $b_j = p_k$ lub $j = m$.

Z zależności (9.1)–(9.2) wiadomo, że wszystkie komórki macierzy $M(i', j', k)$ dla $i' < i^*$ lub $j' < j^*$ mają wartość ujemną. Dla każdej trójki (i', j^*, k) takiej, że $i^* < i' \leq n$ zachodzi:

- jeśli trójka jest dopasowaniem silnym, to wartość $M(i', j^*, k)$ została wyznaczona przy obliczaniu poprzedniego poziomu,
- w przeciwnym przypadku, wartość $M(i', j^*, k)$ jest wyznaczana jako $\max(M(i'-1, j^*, k), M(i', j^*-1, k)) = M(i'-1, j^*, k)$ dokładnie tak samo jak w zależności (9.1).

Wobec tego wartości macierzy M na górnej krawędzi są wyznaczone poprawnie (wiersze 5–9).

Wartości komórek macierzy reprezentujących dopasowania silne, które znajdują się na lewej krawędzi, tj. (i^*, j, k) obliczane są na podstawie $(k-1)$-szego poziomu M (wiersze 10–17). Podobnie jak dla górnej krawędzi, wszystkie wartości $M(i^*, j', k)$ dla niedopasowań są równe $M(i^*, j_0, k)$ dla największego takiego $j_0 < j'$, że (i^*, j_0, k) jest dopasowaniem silnym (w rzeczywistości te komórki nie są obliczane, choć znane są ich wartości).

Dla każdego j_0 takiego, że (i^*, j_0, k) jest dopasowaniem silnym i najmniejszego $j_1 > j_0$ takiego, że (i, j_1, k) również jest dopasowaniem silnym lub $j_1 = m$, czwórka $\langle i, j_0, j_1, k \rangle$ definiuje

CLCS-BP-LEVEL(A, B, P, M, C, V, k)

Wejście: *A, B – ciągi główne, dla których wyznaczany jest podciąg CLCS*
P – ciąg ukierunkowujący
M – macierz programowania dynamicznego
C – macierz liczników bitów o wartości 0 w odpowiednim wektorze bitowym w macierzy V
V – macierz wektorów bitowych
k – indeks poziomu macierzy programowania dynamicznego

Wyjście: *informacja, czy na poziomie k-tym w macierzy programowania dynamicznego jest jakakolwiek wartość dodatnia*
zaktualizowane wartości w macierzach M, C, V

```
   {Wyznaczanie pierwszej nieujemnej komórki na k-tym poziomie}
 1 i*, j* ← CLCS-BP-START-POINT(A, B, P, k)
 2 if i* > n then
 3     return false
 4 Wyznacz maski bitowe dla poziomu k
   {Wyznaczanie górnej krawędzi}
 5 for i ← i* to n do
 6     if a_i = p_k then
 7         M(i, j*, k) ← M(i − 1, j* − 1, k − 1) + 1
 8     else
 9         M(i, j*, k) ← M(i − 1, j*, k)
   {Wyznaczanie lewej krawędzi}
10 y ← 0;   j_0 ← j*;   strips ← ⟨⟩
11 for j ← j* + 1 to m do
12     y ← (y ≪ 1) | 1
13     if b_j = p_k or j = m then
14         strips ← strips + ⟨j_0, j⟩
15         V_{j_0}(i*) ← y;   C_{j_0}(i*) ← 0
16         M(i*, j_0, k) ← M(i* − 1, j_0 − 1, k − 1) + 1
17         j_0 ← j;   y ← 0
   {Obliczenia właściwe}
18 for i ← i* + 1 to n do
19     for wszystkich pasków ⟨j_0, j_1⟩ w strips do
20         CLCS-BP-STRIP(A, B, P, M, C, V, i, j_0, j_1, k)
21 return true
```

Rys. 9.11. Algorytm równoległości bitowej wyznaczający pojedynczy poziom (problem CLCS)
Fig. 9.11. Bit-parallel level-computing algorithm for the CLCS problem

pasek. Wektor $V_{j_0}(i^*)$ nie zawiera bitów o wartości 0, ponieważ wszystkie wartości M dla paska (z wyjątkiem j_1) są równe.

Przed rozpoczęciem wykonywania głównej pętli (wiersze 18–20) niezmiennik pętli jest prawdziwy dla $i = i^*$ oraz $j = j^*$ (początkowe wartości i oraz j). Przy założeniu że $M(i-1, j_0, k)$, $M(i, j_0, k)$, $M(i-1, j_1, k)$, $V_{j_0}(i-1)$ mają poprawną wartość, algorytm CLCS-BP-STRIP poprawnie oblicza $M(i, j_1, k)$ oraz $V_{j_0}(i)$. Oblicza on także wartości komórek $M(i, j, k)$ takich, że $(i+1, j+1, k+1)$ jest dopasowaniem silnym. ■

CLCS-BP-LEVEL-O(A, B, P, M, C, V)

Wejście: *A, B – ciągi główne, dla których wyznaczany jest podciąg CLCS*
P – ciąg ukierunkowujący
M – macierz programowania dynamicznego
C – macierz liczników bitów o wartości 0 w odpowiednim wektorze bitowym w macierzy V
V – macierz wektorów bitowych
Wyjście: *zaktualizowane wartości w macierzach M, C, V*

```
  {Przetwarzanie wstępne}
1 for i ← 0 to n do M(i,0,0) ← 0
2 for j ← 1 to m do M(0, j,0) ← 0
3 V_1(0) ← 1^m
  {Obliczenia właściwe}
4 for i ← 1 to n do
5     CLCS-BP-STRIP(A, B, P, M, C, V, i, 1, m, 0)
```

Rys. 9.12. Algorytm równoległości bitowej wyznaczający poziom 0 macierzy *M* dla problemu CLCS
Fig. 9.12. Bit-parallel level-0 computing algorithm for the CLCS problem

Wyznaczanie poziomu $k = 0$ jest przypadkiem specjalnym, ponieważ nie występują tu dopasowania silne. Wobec tego, w zasadzie może tu być zastosowany algorytm LCS-BP-LENGTH z taką modyfikacją, że wartości komórek $M(i, j, 0)$ muszą być obliczone wprost, jeśli $(i+1, j+1, 1)$ jest dopasowaniem silnym. Do wyznaczenia tego poziomu zastosowany zostanie jednak algorytm CLCS-BP-STRIP przy dodatkowym założeniu, że żadna trójka $(i, j, 0)$ nie jest dopasowaniem silnym.

Lemat 9.4. *Algorytm* CLCS-BP-LEVEL-O *(rys. 9.12) wyznaczający poziom* $k = 0$ *macierzy M poprawnie wyznacza wartości* $M(i, j, 0)$*, jeśli* $(i+1, j+1, 1)$ *jest dopasowaniem silnym.*

Dowód. Zgodnie z warunkami brzegowymi rekurencji China i in. (9.2) komórki $M(i,0,0)$ dla $0 \le i \le n$ oraz $M(0, j, 0)$ dla $0 \le j \le m$ otrzymują wartość 0. Ponieważ dla $k = 0$ nie występują dopasowania silne, więc paski zdefiniowane są przez czwórki: $\langle i, 0, m, 0\rangle$. Wektor $V_1(0)$ zawiera tylko bity o wartościach 1, ponieważ wszystkie wartości macierzy *M* na lewej krawędzi (kolumna 0) są identyczne. Dalszy dowód jest analogiczny jak w przypadku lematu 9.3. ■

Po złożeniu wszystkich powyżej wprowadzonych algorytmów składowych w całość otrzymuje się algorytm CLCS-BP-LENGTH przedstawiony na rys. 9.13.

Lemat 9.5. *Algorytm* CLCS-BP-LENGTH *(rys. 9.13) wyznaczający długość podciągu CLCS jest poprawny.*

Dowód. Dowód indukcyjny dla k, będącego numerem poziomu macierzy. Algorytm CLCS--BP-LEVEL-O wyznaczający poziom 0 macierzy jest poprawny (lemat 9.4). Zakładając, że poziom $k-1$ jest poprawnie wyliczony, algorytm CLCS-BP-LEVEL poprawnie wylicza poziom k

CLCS-BP-LENGTH(A, B, P)

Wejście: *A, B – ciągi główne, dla których wyznaczany jest podciąg CLCS*
P – ciąg ukierunkowujący
Wyjście: *długość podciągu CLCS*

```
1  CLCS-BP-LEVEL-0(A, B, P, M, C, V)
2  for k ← 1 to r do
3      if CLCS-BP-LEVEL(A,B,P,M,C,V,k) = false then
4          return 0
5  if b_m = p_r then
6      return M(n,m,r)
7  else
8      j' ← argmax_j(b_j = p_r ∧ 1 ≤ j ≤ m)
9      return M(n, j', r) + C_j'(n)
```

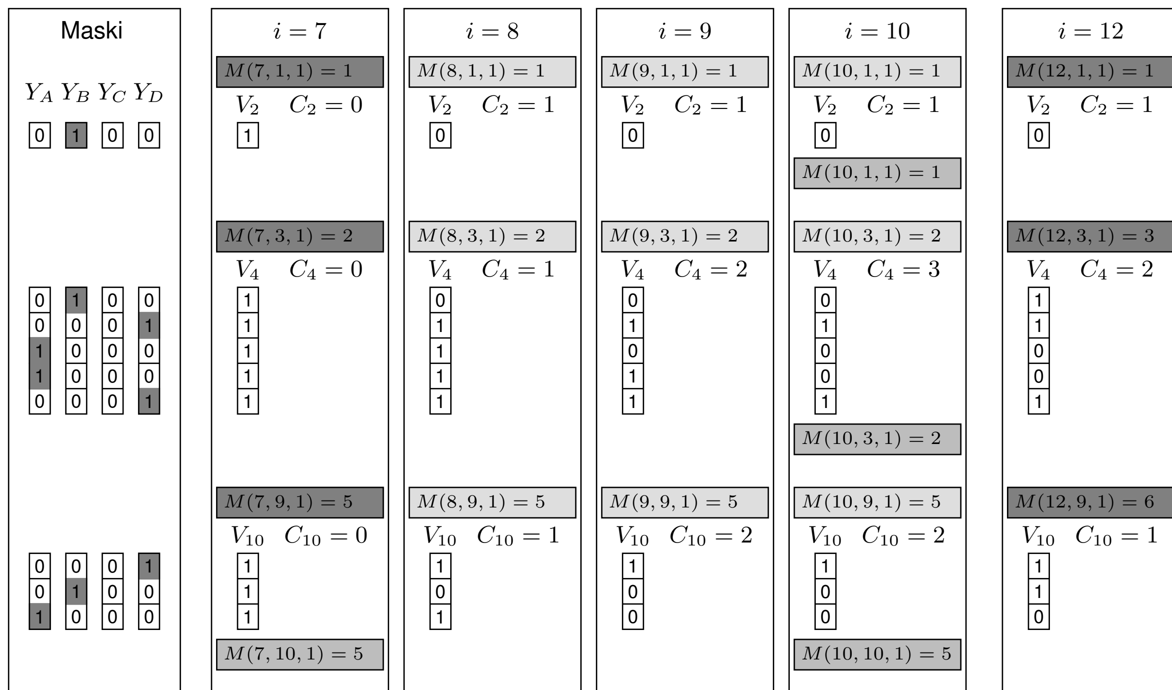
Rys. 9.13. Algorytm równoległości bitowej wyznaczający długość podciągu CLCS
Fig. 9.13. Bit-parallel CLCS length computing algorithm

(lemat 9.3). Z powyższego, komórka $M(n,m,r)$ przechowuje długość podciągu CLCS dla A, B, P, jeśli $b_m = p_r$. Jeśli ostatni wiersz macierzy M na poziomie r nie zawiera dopasowań silnych, to wynik końcowy jest wyznaczany jako suma największej wartości w macierzy M na poziomie r i liczby bitów o wartości 0 w ostatnim pasku, tj. $\langle n, j', m, k \rangle$. Jeśli nie istnieje ciąg wynikowy, to algorytm zwraca 0. ■

Przykład działania tego algorytmu pokazany jest na rys. 9.14. Przedstawione są obliczenia wykonywane dla $k = 1$. Górna część rysunku przedstawia macierz M, która w rzeczywistości nie jest tworzona w trakcie działania algorytmu, bo potrzebne (i obliczane) są tylko wartości komórek zaznaczonych na szaro. Oprócz poziomu $k = 1$ pokazano również poziomy $k = 0$ (wartości dla dopasowań silnych na poziomie $k = 1$ są obliczane na podstawie komórek z tego poziomu) oraz $k = 2$ (niektóre komórki na poziomie $k = 1$ muszą zostać obliczone wprost, bo będą potrzebne do wyznaczenia wartości dla dopasowań silnych na poziomie $k = 2$). W rzeczywistej implementacji w algorytmie używane są tylko struktury danych, które pokazane są w dolnej części rysunku.

Ramka Maski przedstawia wartości wektorów bitowych dla wierszy pasków. Przetwarzanie poziomu rozpoczyna się od punktu $(7,1)$. Ramka $i = 7$ pokazuje obliczenia wykonywane podczas przetwarzania kolumny o indeksie 7. Ponieważ jest to lewa krawędź tego poziomu macierzy, więc wykonywane są tu tylko: wyznaczanie wartości dla dopasowań silnych (komórki $M(7,1,1)$, $M(7,3,1)$, $M(7,9,1)$) oraz inicjalizacja wektorów V. Obliczana jest także wartość $M(7,10,1)$, która będzie potrzebna przy przetwarzaniu dopasowania silnego $(8,11,2)$.

Ramka $i = 8$ pokazuje następujące obliczenia. $M(8,1,1)$ obliczane jest na wstępnym etapie obliczeń górnej krawędzi – $(8,1,1)$ nie jest dopasowaniem silnym, więc wartość $M(8,1,1)$ jest kopią lewego sąsiada, tj. $M(7,1,1)$. Wartość wektora $V_2(8)$ wyznaczana jest w sposób bitowo-



Rys. 9.14. Przykład działania algorytmu równoległości bitowej wyznaczającego długość podciągu CLCS dla $k = 1$. Ciągi wejściowe to: $A =$ ABAADACBAABC, $B =$ CBCBDAADCDBA, $P =$ CBB. Górna część pokazuje zawartość macierzy M. (Macierz M pokazana jest tylko dla łatwiejszego zrozumienia całości, bo nie jest w pełni obliczana przez algorytm.) Ramki w dolnej części pokazują wykonywane obliczenia. Najbardziej szare komórki (dolna część) oznaczają dopasowania silne. Jasnoszare komórki oznaczają komórki na granicach pasków obliczane wprost. Ciemnoszare komórki oznaczają komórki obliczane wprost, bo są potrzebne dla następnego poziomu

Fig. 9.14. Example of the bit-parallel method for computation of the CLCS length for $k = 1$. The sequences are: $A =$ ABAADACBAABC, $B =$ CBCBDAADCDBA, $P =$ CBB. Upper part shows the contents of matrix M. (This is only for comparison, since a complete matrix M is not computed by the algorithm.) The frames in bottom part show the actually made computations. The darkest boxes (bottom part) denote strong matches. The light gray boxes denote the cells at boundaries of strips computed explicitly. The dark gray boxes mean the cells computed explicitly for the next level

Rozszerzenie algorytmu CLCS-BP-STRIP

```
11   T(i,k) ← T(i,k) | (V_{j0} ≪ j0)
12   if M(i−1, j1,k) = M(i, j1,k) then
13        T(i,k) ← T(i,k) & ∼(1 ≪ j1)
```

Rys. 9.15. Rozszerzenie algorytmu CLCS-BP-STRIP pozwalające na wyznaczanie podciągu CLCS
Fig. 9.15. Extension to the CLCS-BP-STRIP algorithm for computing the CLCS

Rozszerzenie algorytmu CLCS-BP-LEVEL

```
18.1      T(i,k) ← T(i,k) & ∼(1 ≪ j*)
```

Rys. 9.16. Rozszerzenie algorytmu CLCS-BP-LEVEL umożliwiające wyznaczanie podciągu CLCS
Fig. 9.16. Extension to the CLCS-BP-LEVEL algorithm for computing the CLCS

równoległy (rys. 9.10, wiersze 1–4) na podstawie $M(8,1,1)$ i $V_2(7)$. Ponieważ $(8,3,1)$ nie jest dopasowaniem, więc jego wartość wyznaczana jest jako maksimum z $M(7,3,1)$ i sumy $M(8,1,1)+C_2(8)$. Podobnie $V_4(8)$ wyznaczane jest w sposób bitowo-równoległy na podstawie $M(8,3,1)$ i $V_4(7)$. Następnie obliczane jest $M(8,9,1)$. Wreszcie, wyznaczane jest $V_{10}(8)$. W tej kolumnie nie ma potrzeby wyznaczania wartości innych komórek macierzy M, ponieważ na poziomie $k=2$ nie występują dopasowania silne w kolumnie o indeksie 9.

W podobny sposób wykonywane są obliczenia przedstawione w pozostałych ramkach.

9.3.3. Wyznaczanie ciągu wynikowego

Algorytm CLCS-BP-LENGTH umożliwia wyznaczenie tylko długości podciągu CLCS, co czasami jest niewystarczające. Uzyskanie także podciągu CLCS jest jednak stosunkowo proste i opiera się na podobnej idei jak w przypadku problemu LCS (por. podrozdz. 7.3.4).

Oprócz struktur dotychczas opisanych, tworzona jest trójwymiarowa macierz binarna T o wymiarach $(n+1)\times(m+1)\times(r+1)$. Dla wygody prezentacji pominięte tu zostaną szczegóły implementacyjne i zaznaczone zostanie tylko, że w praktyce jest to dwuwymiarowa macierz o wymiarach $(n+1)\times(r+1)$ zawierająca w każdej komórce tablicę $\lceil(m+1)/w\rceil$ słów komputerowych. Macierz T jest wypełniana przez rozszerzone wersje algorytmów CLCS-BP-STRIP (rys. 9.15) i CLCS-BP-LEVEL (rys. 9.16). Ogólna idea polega na przechowywaniu w $T(i,j,k)$ wartości bitowej pełniącej rolę „drogowskazu", do której z sąsiednich komórek należy przejść przy wyznaczaniu podciągu CLCS. Jeśli $T(i,j,k)=0$, to drogowskaz wskazuje lewą komórkę ($T(i-1,j,k)$), w przeciwnym przypadku górną ($T(i,j-1,k)$). Wartość tego bitu nie ma znaczenia, jeśli bieżąca komórka reprezentuje dopasowanie, ponieważ wiadomo, że dla dopasowania silnego należy przejść do $T(i-1,j-1,k-1)$, a dla dopasowania niesilnego do $T(i-1,j-1,k)$.

Dopasowania obsługiwane są przez algorytm konstruujący wynik w sposób szczególny i w tym akapicie pominięta zostanie ta kwestia, a uwaga zostanie skupiona na komórkach nie-

CLCS-BP-Sequence(A, B, P, M, T, ℓ)

Wejście: *A, B – ciągi główne, dla których wyznaczany jest podciąg CLCS*
P – ciąg ukierunkowujący
M – macierz programowania dynamicznego
T – macierz bitowa zawierająca informacje o sposobie przechodzenia macierzy programowania dynamicznego
ℓ – długość podciągu CLCS
Wyjście: *podciąg CLCS*

1 $\ell^* \leftarrow \ell$; $k \leftarrow r$; $i \leftarrow m$; $j \leftarrow n$
2 **while** $\ell > 0$ **do**
3 **if** $a_i = b_j$ **then**
4 **if** $k > 0$ **and** $a_i = p_k$ **then** $k \leftarrow k - 1$
5 $s_{\ell^*} \leftarrow a_i$; $\ell^* \leftarrow \ell^* - 1$
6 $i \leftarrow i - 1$; $j \leftarrow j - 1$
7 **else**
8 **if** j-ty bit $T(i,k)$ jest równy 1 **then** $j \leftarrow j - 1$
9 **else** $i \leftarrow i - 1$
10 **return** $s_1 s_2 \ldots s_\ell$

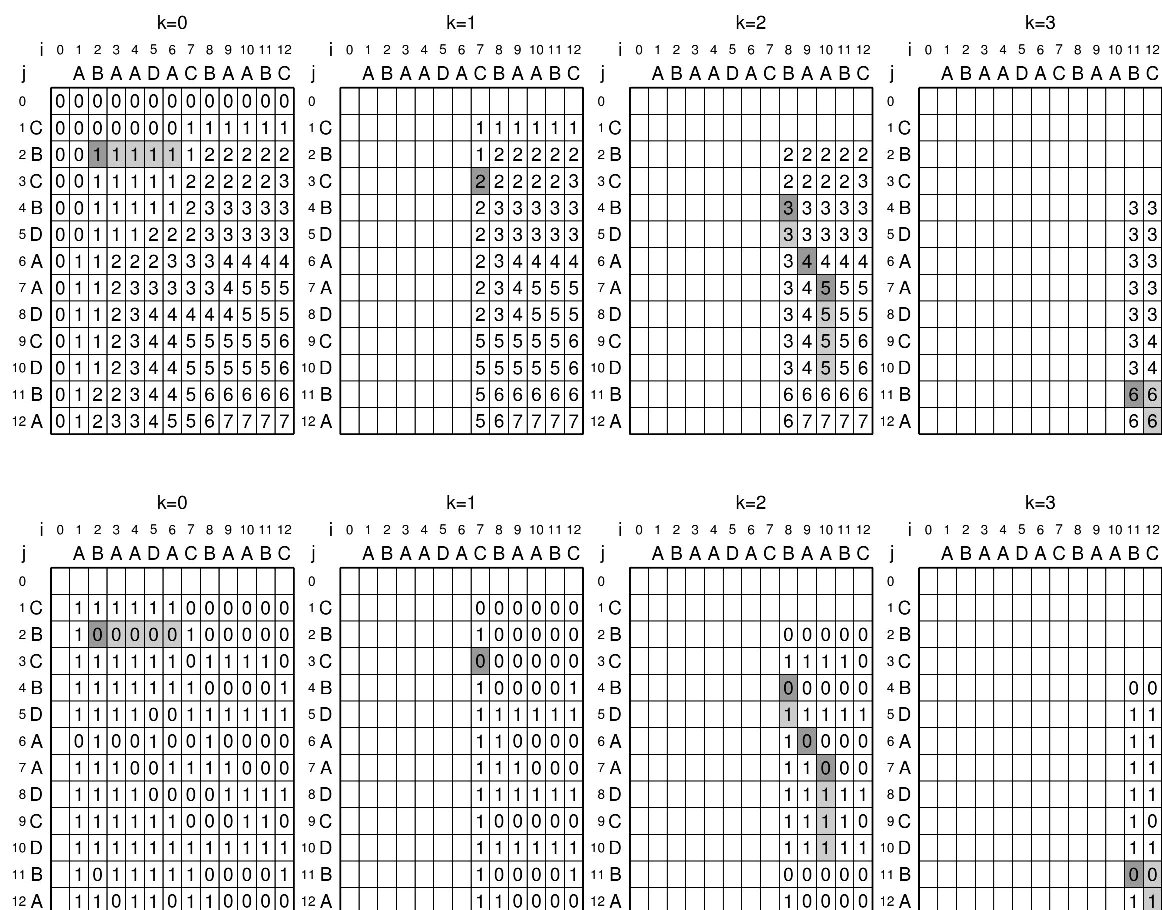
Rys. 9.17. Algorytm wyznaczania podciągu CLCS
Fig. 9.17. Traceback procedure to obtain an CLCS

reprezentujących dopasowań. Algorytm obliczający pasek zapisuje zawartość wektora V do T (rys. 9.15, wiersz 11), ponieważ jeśli $T(i,j,k) = 0$, to wiadomo (jeśli j nie jest granicą paska), że $M(i,j,k) > M(i,j-1,k)$, a więc należy przejść w lewo do wartości równej $M(i,j,k)$. W przeciwnym przypadku należy przejść w górę, ponieważ $M(i,j-1,k) = M(i,j,k)$. Dla pozycji będącej dolną krawędzią paska wymagane jest nieco inne podejście, ponieważ różnica pomiędzy komórkami z wertykalnie sąsiednich pasków nie jest przechowywana w V. Zamiast tego można jednak skorzystać z macierzy M, ponieważ dla dolnego wiersza paska wartości M są policzone wprost, a więc porównując $M(i-1, j_1, k)$ z $M(i, j_1, k)$, można określić, jaką wartość ma mieć drogowskaz $T(i, j_1, k)$. Jeśli wskazane wartości są równe, to drogowskaz może wskazywać komórkę lewą, a w przeciwnym przypadku górną, ponieważ $M(i, j_1 - 1, k)$ musi być wtedy równe $M(i, j_1, k)$.

Dla dopasowań wartości bitów z T są niepoprawne, ale nie ma to żadnego znaczenia, ponieważ o tym, czy aktualnie występuje dopasowanie czy nie, można się dowiedzieć porównując symbole ciągu. Algorytm wyznaczający podciąg CLCS działający w sposób opisany powyżej przedstawiony jest na rys. 9.17. Ilustracja jego działania pokazana jest na rys. 9.18.

9.3.4. Szczegóły implementacyjne i analiza złożoności

Przedstawiony opis algorytmu wyznaczania długości podciągu CLCS sugeruje, że złożoność pamięciowa jest $O(nmr)$ słów z powodu konieczności reprezentowania macierzy M. Ponieważ w tym algorytmie wynikiem jest długość podciągu CLCS, więc wystarcza przechowywanie



Rys. 9.18. Przykład działania algorytmu wyznaczającego podciąg CLCS. Ciągi wejściowe to: $A=$ ABAADACBAABC, $B=$ CBCBDAADCDBA, $P=$ CBB. Górna część rysunku (pokazana tylko dla porównania) przedstawia zawartość macierzy M. Dolna część rysunku przedstawia zawartość macierzy T. Ciemnoszare komórki oznaczają dopasowania brane do wyniku. Jasnoszare komórki oznaczają komórki odwiedzane przez algorytm. Ciąg wynikowy to B<u>CB</u>AA<u>B</u> (podkreślone symbole tworzą ciąg ukierunkowujący)

Fig. 9.18. Example of the traceback procedure. The sequences are: $A=$ ABAADACBAABC, $B=$ CBCBDAADCDBA, $P=$ CBB. Upper part shows (only for comparison) the contents of the matrix M. The bottom part shows the contents of the traceback array T. The dark gray boxes denote matches taken to the result. The light gray boxes denote the visited. The output sequence is B<u>CB</u>AA<u>B</u> (underlined symbols form constrained sequence)

tylko dwóch poziomów macierzy M, k-tego i $(k-1)$-szego. Co więcej, z poziomu $(k-1)$-szego potrzebne są wartości tylko tych komórek $M(i-1, j-1, k-1)$, dla których (i, j, k) jest dopasowaniem silnym. Niech liczba dopasowań silnych na poziomie k będzie oznaczona przez d_k^{sm} (jeśli $k \leq 0$ lub $k > r$, to z definicji $d_k^{\mathrm{sm}} = 0$). Wartości komórek dla tych dopasowań mogą być przechowywane w kolejce FIFO, ponieważ są one potrzebne przy wyznaczaniu poziomu k-ego w tej samej kolejności, w jakiej były obliczane na poziomie $(k-1)$-szym. Ponadto, ponieważ na każdym poziomie kolumny przetwarzane są kolejno od lewej do prawej, więc wystarcza przechowywanie tylko dwóch wektorów bitowych V dla kolumn o indeksach i oraz $i-1$.

Dodatkowo, należy przechowywać wektory masek bitowych dla wszystkich symboli alfabetu w ramach jednego poziomu. Z powyższego, sumaryczna złożoność pamięciowa tego algorytmu jest

$$O\left(\left\lceil \frac{m}{w} \right\rceil \sigma\right) + O\left(\left\lceil \frac{m}{w} \right\rceil + \beta_k\right) + O\left(d_{k-1}^{\text{sm}} + d_k^{\text{sm}} + d_{k+1}^{\text{sm}}\right) = O\left(\left\lceil \frac{m}{w} \right\rceil \sigma + \beta' + d_*^{\text{sm}}\right) \quad (9.9)$$

słów, gdzie $\beta_k = |\{b_j : b_j = p_k, 0 < j \leq m\}|$ na poziomie k (z definicji $\beta_0 = \beta_{k+1} = 0$), $\beta' = \max_{0 \leq k \leq r} \beta_k$, $d_*^{\text{sm}} = \max_{0 \leq k \leq r} d_k^{\text{sm}}$).

Rozważona teraz zostanie złożoność czasowa tego algorytmu. Liczba pasków w pojedynczej kolumnie na poziomie k jest $\Theta(\beta_k)$, ale paski rozmiaru większego niż w są emulowane przez tablice słów komputerowych, wobec czego sumaryczna liczba słów komputerowych koniecznych do reprezentowania wektora V jest $O(\beta_k + \lceil m/w \rceil)$. Dla każdego słowa komputerowego zawierającego pasek lub jego fragment czas operacji bitowych na nim jest $O(1)$ na każdą kolumnę. Ponadto, istnieje $\Theta(\alpha_{k+1})$ kolumn, dla których zachodzi konieczność wyznaczenia wartości M dla dopasowań silnych na wyższym poziomie ($\alpha_k = |\{a_i : a_i = p_k, 0 < i \leq n\}|$, z definicji $\alpha_0 = \alpha_{r+1} = 0$). Obliczenia te zajmują czas $O(m)$ na każdą kolumnę. Istnieje także konieczność usunięcia pewnej liczby najmniej znaczących wartości 0 z wektorów V w kolumnach zawierających dopasowania silne. Dla każdej z α_k takich kolumn czas tej operacji jest $O(m)$. Z powyższego, sumaryczna złożoność czasowa proponowanego algorytmu jest:

$$\sum_{k=0}^{r} O\left(n\left(\beta_k + \left\lceil \frac{m}{w} \right\rceil\right) + m\alpha_k + m\alpha_{k+1}\right) = \sum_{k=0}^{r} O\left(n\beta_k + n\left\lceil \frac{m}{w} \right\rceil + m\alpha_k\right). \quad (9.10)$$

W najgorszym przypadku $\alpha_k = n$, $\beta_k = m$ dla każdego $0 < k \leq r$, a więc złożoność czasowa jest $O(nmr)$. Złożoność czasowa w przypadku pesymistycznym może być jednak wyrażona także w zależności od sumarycznej liczby dopasowań silnych, $D^{\text{sm}} = \sum_{k=0}^{r} d_k^{\text{sm}}$. Wtedy $\alpha_k \beta_k = d_k^{\text{sm}}$, a więc złożoność czasowa to:

$$\sum_{k=0}^{r} O\left(n\beta_k + n\left\lceil \frac{m}{w} \right\rceil + m\frac{d_k^{\text{sm}}}{\beta_k}\right), \quad (9.11)$$

co jest maksymalne dla $\beta_k = \Theta(\sqrt{m/n}\sqrt{d_k^{\text{sm}}})$ dla każdego poprawnego k. Otrzymuje się więc następujące oszacowanie złożoności czasowej:

$$\sum_{k=0}^{r} O\left(m\frac{d_k^{\text{sm}}}{\sqrt{m/n}\sqrt{d_k^{\text{sm}}}} + n\sqrt{m/n}\sqrt{d_k^{\text{sm}}} + n\left\lceil \frac{m}{w} \right\rceil\right) =$$

$$\sum_{k=0}^{r} O\left(\sqrt{nm}\sqrt{d_k^{\text{sm}}} + \sqrt{nm}\sqrt{d_k^{\text{sm}}} + n\left\lceil \frac{m}{w} \right\rceil\right) =$$

$$O\left(\sqrt{nm}\sum_{k=0}^{r}\sqrt{d_k^{\text{sm}}} + n\left\lceil \frac{m}{w} \right\rceil r\right).$$

Dla ustalonego D^{sm} suma $\sum_{k=0}^{r}\sqrt{d_k^{\mathrm{sm}}}$ ma wartość maksymalną, jeśli wszystkie d_k^{sm} są równe dla $0 < k \leq r$. Wobec tego można sformułować wniosek:

Wniosek 9.5. *Złożoność czasowa algorytmu równoległości bitowej dla problemu CLCS wynosi w przypadku pesymistycznym:*

$$O\left(\sqrt{nm}\sum_{k=0}^{r}\sqrt{D^{\mathrm{sm}}/r}+n\left\lceil\frac{m}{w}\right\rceil r\right)=O\left(\sqrt{nmrD^{\mathrm{sm}}}+n\left\lceil\frac{m}{w}\right\rceil r\right). \tag{9.12}$$

Sumaryczna liczba dopasowań silnych może być co najwyżej $D^{\mathrm{sm}} = nmr$, ale w praktyce jest znacznie mniejsza.

Do wyznaczenia złożoności czasowej w przypadku średnim konieczne jest przyjęcie pewnego założenia dotyczącego zawartości ciągów wejściowych (A, B, P). Przyjęte zostanie, że zostały one wygenerowane przez generator liczb pseudolosowych o rozkładzie równomiernym dla rozmiaru alfabetu σ. Wyznaczanie złożoności czasowej w tym przypadku rozpocznie się od wzoru (9.10). Ponieważ ciągi są niezależnie, więc nie występują zależności pomiędzy liczbą dopasowań silnych na różnych poziomach. Co więcej, nie ma także zależności pomiędzy α_k i β_k na pojedynczym poziomie. Dlatego też, aby otrzymać złożoność czasową w przypadku średnim, należy wyznaczyć wartość oczekiwaną każdego składnika sumy.

Najpierw zostanie rozważony pierwszy składnik, $m\alpha_k$, dla $0 < k < r$. Wartość α_k jest liczbą symboli ciągu A równych p_k. Oczywiście prawdopodobieństwo, że $a_i = p_k$ wynosi $1/\sigma$ i oczekiwana wartość α_k to n/σ. Podobnie, oczekiwana wartość β_k to m/σ. Wobec tego:

Wniosek 9.6. *Złożoność czasowa w przypadku średnim algorytmu równoległości bitowej dla problemu CLCS jest:*

$$O\left(n\left\lceil\frac{m}{w}\right\rceil\right)+\sum_{k=1}^{r}O\left(\frac{nm}{\sigma}+\frac{nm}{\sigma}+n\left\lceil\frac{m}{w}\right\rceil\right)=O\left(nmr\left\lceil\frac{m}{\min(\sigma,w)}\right\rceil\right). \tag{9.13}$$

Jeśli żądanym wynikiem jest podciąg CLCS, a nie tylko jego długość, to konieczne jest przechowywanie macierzy T, która zajmuje $O(n\lceil m/w\rceil r)$ słów komputerowych. Czas jej wyznaczenia jest taki sam jak czas działania algorytmu wyznaczającego długość podciągu CLCS, a więc nie wpływa na złożoność czasową algorytmu CLCS-BP-Length. Złożoność czasowa wyznaczania podciągu CLCS na podstawie tej macierzy jest niewielka, ponieważ w każdej iteracji pętli w CLCS-BP-Sequence zmniejszana jest wartość przynajmniej jednego indeksu, a więc:

Wniosek 9.7. *Złożoność czasowa algorytmu* CLCS-BP-Sequence *jest*

$$\Theta(r+n+m)=\Theta(n). \tag{9.14}$$

9.3.5. Wyniki eksperymentalne

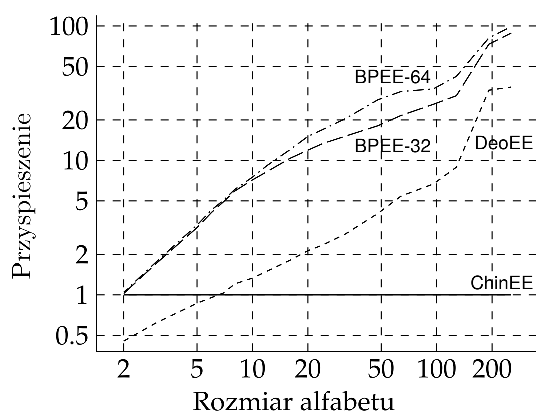
W celu porównania zaproponowanego algorytmu równoległości bitowej z innymi algorytmami przeprowadzono dwie serie eksperymentów. Prezentowane czasy stanowią medianę z czasów 501 wykonań algorytmów. Komputer testowy wyposażony był w procesor AMD Phenom II X4 810 (zegar 2600 MHz). Algorytmy zaimplementowano w języku C++ i skompilowano za pomocą MS Visual C++ 2008 z maksymalną optymalizacją pod kątem prędkości (opcja -Ox).

Z eksperymentów opisanych w podrozdz. 9.2.4 wynika, że najszybszymi algorytmami sekwencyjnymi dla problemu CLCS są algorytm China i in. [42] oraz algorytm zaproponowany w podrozdz. 9.2, w których zastosowano technikę punktów wejścia-wyjścia (podrozdz. 9.2.3). Poniżej opisano wyniki podobnych eksperymentów dla tego samego zestawu danych, wobec czego dla większej przejrzystości zrezygnowano tu z badania pozostałych algorytmów. Omawiane algorytmy oznaczone są na wykresach i w tabelach następująco:

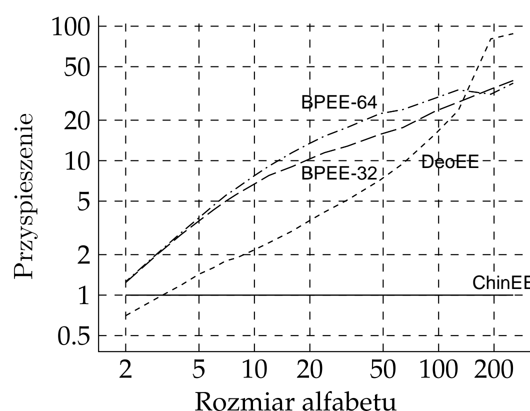
- ChinEE – wersja algorytmu China i in. [42] zmodyfikowana przez użycie techniki punktów wejścia-wyjścia (podrozdz. 9.2.3),
- DeoEE – wersja algorytmu zaproponowanego w podrozdz. 9.2.1 zmodyfikowana przez użycie techniki punktów wejścia-wyjścia (podrozdz. 9.2.3),
- BPEE-32 – algorytm równoległości bitowej z techniką punktów wejścia-wyjścia zaproponowany w podrozdz. 9.3 przy $w = 32$,
- BPEE-64 – algorytm równoległości bitowej z techniką punktów wejścia-wyjścia zaproponowany w podrozdz. 9.3 przy $w = 64$,

W pierwszej serii eksperymentów (rys. 9.19) porównano algorytmy wyznaczania podciągu CLCS i długości podciągu CLCS dla danych losowych wygenerowanych z użyciem generatora liczb pseudolosowych o rozkładzie równomiernym. Dla większej przejrzystości, zamiast bezwzględnych czasów wykonania podano na nich, ile razy dany algorytm jest szybszy niż algorytm ChinEE.

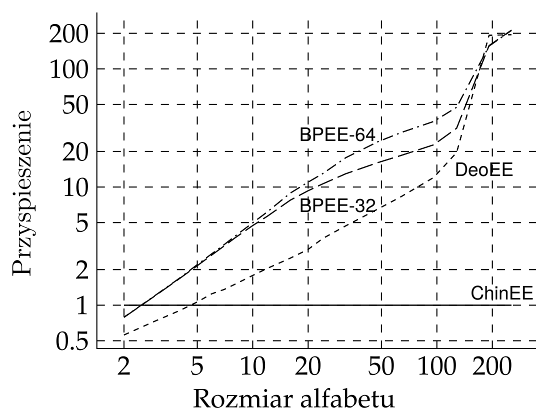
W pierwszym teście tej serii eksperymentów zbadano wpływ rozmiaru alfabetu na czas wykonywania algorytmów (rysunki 9.19a–d). Większy rozmiar alfabetu oznacza zwykle mniejszą liczbę dopasowań i dłuższe paski. Oczywiście, rozmiar słowa komputerowego stanowi swego rodzaju barierę w zwiększaniu efektywnej długości paska, bo paski dłuższe niż w są emulowane przez tablice słów komputerowych. Kiedy długości ciągów głównych były znacząco różne, a ciąg ukierunkowujący był krótki, algorytm równoległości bitowej okazał się znacząco szybszy od pozostałych algorytmów zarówno wtedy, gdy wyznaczano tylko długość podciągu CLCS (9.19a), jak i sam podciąg CLCS (9.19b). W obu przypadkach dla najmniejszego możliwego alfabetu ($\sigma = 2$) przewaga była pomijalna, ale wraz ze wzrostem rozmiaru alfabetu stawała się coraz bardziej znacząca – algorytm ChinEE był 13–16 razy wolniejszy dla $\sigma = 20$. Dla małych



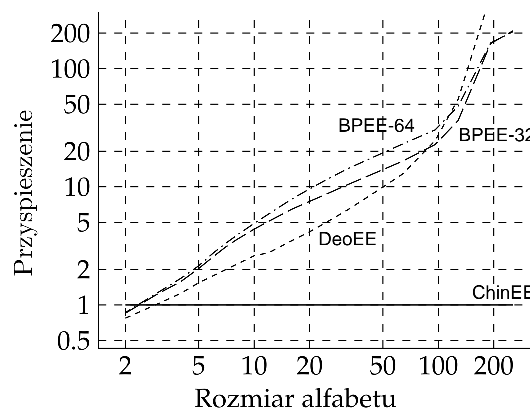
(a) $m = 512, n = 4096, r = 4$, długość CLCS



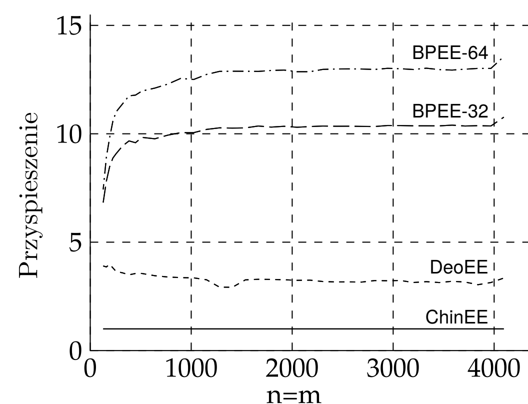
(b) $m = 512, n = 4096, r = 4$, CLCS



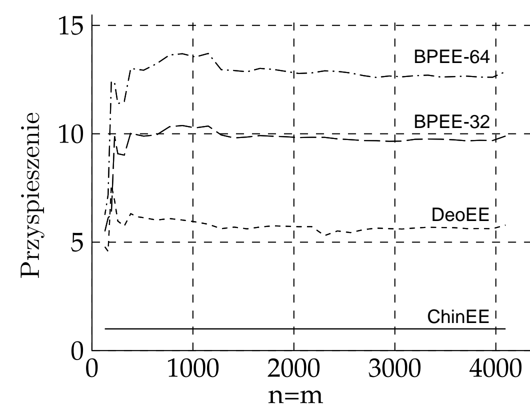
(c) $m = 2048, n = 2048, r = 12$, długość CLCS



(d) $m = 2048, n = 2048, r = 12$, CLCS



(e) $r = 4, \sigma = 20$, długość CLCS



(f) $r = 4, \sigma = 20$, CLCS

Rys. 9.19. Porównanie przyspieszenia uzyskiwanego przez algorytmy równoległości bitowej dla problemu CLCS w stosunku do ulepszonego algorytmu China i in. Kolumna lewa: wyznaczanie długości podciągu CLCS. Kolumna prawa: wyznaczanie podciągu CLCS

Fig. 9.19. Comparison of the speedup of bit-parallel algorithms for the CLCS problem. Left column: computation of CLCS length. Right column: computation of CLCS

alfabetów czasy działania algorytmów BPEE-32 i BPEE-64 były podobne, co wynika z faktu, że paski rzadko nie mieściły się w 32-bitowym słowie. Dla alfabetów o średnim rozmiarze ($20 \leq \sigma \leq 100$) wersja 64-bitowa była zauważalnie szybsza (o około 35%). Co ciekawe, dla dużych alfabetów algorytmy te stawały się jednakowo szybkie. Było to spowodowane tym, że dominującą rolę odgrywało tu bardzo duże zawężenie obliczanych obszarów macierzy bądź też już po etapie przetwarzania wstępnego wiadomo było, że szukany podciąg CLCS nie istnieje. Porównując czasy działania algorytmów BPEE-64 i DeoEE, można zauważyć, że proponowany algorytm równoległości bitowej jest kilkakrotnie szybszy.

Na rysunkach 9.19c–d pokazane są wyniki dla przypadku, w którym ciąg ukierunkowujący był dłuższy (12 symboli), a ciągi główne – równej długości. Wyniki tych eksperymentów są bardzo podobne, choć względna prędkość działania algorytmu równoległości bitowej jest nieznacznie mniejsza. Wyniki ostatnich testów w tej serii (rysunki 9.19e–f) pokazują, jak zmienia się względna prędkość algorytmów w zależności od długości obu ciągów głównych ($\sigma = 20$, $r = 4$). Dla bardzo krótkich ciągów głównych algorytm proponowany jest mniej niż 10-krotnie szybszy, ale wraz ze wzrostem długości ciągów staje się szybszy 15-krotnie dla wersji 64-bitowej i 12-krotnie dla wersji 32-bitowej w obu wariantach problemu (wyznaczanie podciągu CLCS lub tylko jego długości).

W drugiej serii eksperymentów algorytmy oceniano na danych rzeczywistych reprezentujących głównie sekwencje RNase. Były to te same dane, które wykorzystano w testach algorytmu zaproponowanego w podrozdz. 9.2.4 (tabela 9.1). Rozmiar alfabetu dla nich to $\sigma = 20$. Również metodologia przeprowadzenia tych testów była taka sama, a więc prezentowane są sumaryczne czasy wykonania algorytmów dla wszystkich par ciągów w ramach każdego zbioru.

Wyniki eksperymentów przedstawione są w tabeli 9.4. Jak można zauważyć, algorytm BPEE-64 okazał się najszybszy we wszystkich przypadkach. Jest on szybszy od algorytmu ChinEE od 5,6 do 16,7 razy. Od algorytmu DeoEE jest natomiast szybszy od 2 do 4 razy. Wyniki te pokazują, że stosując proponowany algorytm równoległości bitowej dla problemu CLCS można znacząco przyspieszyć jego rozwiązywanie.

9.4. Algorytm równoległy dla procesorów graficznych

9.4.1. Algorytm

W niniejszym podrozdziale omówiony będzie algorytm równoległy dla procesorów GPU zaproponowany przez autora w [71]. Punktem wyjścia do stworzenia algorytmu CLCS-CUDA dla procesorów GPU będzie algorytm China i in. [42]. Wyznacza on trójwymiarową macierz programowania dynamicznego w prosty sposób, dzięki czemu można go zrównoleglić stosując podejście podobne do zaproponowanego w podrozdz. 7.6.2. Algorytmy oparte na idei Hunta–

Tabela 9.4
Czasy działania (w ms) algorytmów wyznaczania podciągu CLCS i długości podciągu CLCS dla rzeczywistych danych. W nawiasach: ile razy dany algorytm jest szybszy niż ChinEE

Zestaw danych	Ciąg ukierunk.	Długość CLCS				CLCS			
		ChinEE	DeoEE	BPEE-32	BPEE-64	ChinEE	DeoEE	BPEE-32	BPEE-64
ds0	HKH	3,133 (1,00)	0,968 (3,24)	0,409 (7,66)	**0,358** (**8,75**)	3,904 (1,00)	0,910 (4,29)	0,615 (6,35)	**0,538** (**7,26**)
ds1	HKH	4,436 (1,00)	1,420 (3,12)	0,570 (7,78)	**0,468** (**9,48**)	7,370 (1,00)	2,380 (3,10)	0,872 (8,45)	**0,723** (**10,19**)
ds1	HKSH	3,962 (1,00)	1,176 (3,37)	0,558 (7,10)	**0,484** (**8,19**)	6,048 (1,00)	1,679 (3,60)	0,870 (6,95)	**0,748** (**8,09**)
ds1	HKSTH	3,845 (1,00)	1,307 (2,94)	0,564 (6,82)	**0,507** (**7,58**)	6,026 (1,00)	1,678 (3,59)	0,880 (6,85)	**0,776** (**7,77**)
ds2	HKSH	3,040 (1,00)	0,950 (3,20)	0,380 (8,00)	**0,351** (**8,66**)	5,773 (1,00)	1,934 (2,99)	0,595 (9,70)	**0,522** (**11,06**)
ds2	HKSTH	2,681 (1,00)	0,943 (2,84)	0,345 (7,77)	**0,313** (**8,57**)	5,647 (1,00)	1,687 (3,35)	0,575 (9,82)	**0,508** (**11,12**)
ds3	HKH	8,436 (1,00)	2,512 (3,36)	0,765 (11,03)	**0,597** (**14,13**)	15,089 (1,00)	2,901 (5,20)	1,299 (11,62)	**0,946** (**15,95**)
ds4	DGGG	1,994 (1,00)	0,630 (3,17)	0,353 (5,65)	**0,323** (**6,17**)	3,061 (1,00)	0,906 (3,38)	0,559 (5,48)	**0,488** (**6,27**)

Szymanskiego (podrozdziały 9.2, 9.2.3) są szybsze od algorytmu China i in. o czynnik około 5, ale znacznie trudniej je zrównoleglić. Równie trudno zrównoleglić zaproponowany w poprzednim podrozdziale algorytm równoległości bitowej.

Kluczowa różnica pomiędzy problem CLCS a problemami LCS i LCTS, dla których zaproponowano algorytmy równoległe dla procesorów GPU, polega na tym, że w macierzy programowania dynamicznego dla problemu CLCS występują zależności pomiędzy poziomami. Macierz dekomponowana jest na pudełka, ale danymi wejściowymi do obliczania każdego pudełka są: lewa i górna krawędź, prawa-dolna komórka lewego-górnego sąsiedniego pudełka i pudełko znajdujące się poziom niżej (dla $k > 0$). Dlatego też wyznaczanie wartości poziomów nie jest niezależne i przetwarzanie oparte na zasadzie drugiej przekątnej dla wyższego poziomu musi się rozpocząć o krok później. Dla przykładu, w drugim etapie, pudełka, które mogą być obliczane to: $\mathcal{S} = \{(0,0,1),(0,1,0),(1,0,0)\}$. Oznacza to, że w porównaniu do problemu LCS może być równolegle wyznaczanych znacznie więcej pudełek (nawet $r+1$ razy więcej).

Podstawową wadą tego algorytmu są duże wymagania pamięciowe, ponieważ potrzebne mogą być nie tylko dwie krawędzie, ale także całe pudełko znajdujące się poziom niżej. Struktury danych, które znajdują się w pamięci globalnej procesora GPU to:

- tablice n, m i r słów na ciągi A, B i P,
- tablica $n(r+1)$ słów dla górnej krawędzi wszystkich poziomów,
- tablica $m(r+1)$ słów dla lewej krawędzi wszystkich poziomów,

- tablica $3\lceil m/b_\mathrm{h}\rceil(r+1)$ słów dla prawych-dolnych komórek lewych-górnych sąsiednich pudełek do bieżących,
- tablica $2b_\mathrm{w}m(r+1)$ słów dla pudełek znajdujących się poziom niżej niż aktualnie obliczane.

Wniosek 9.8. *Sumaryczna złożoność pamięciowa algorytmu* CLCS-CUDA *jest:*

$\Theta(nr+b_\mathrm{w}mr)$ *słów.*

Obliczenia wewnątrz pudełka wykonywane są prawie identycznie jak w problemie LCS. Główną różnicę stanowi obsługa dopasowań silnych. Dla każdego napotkanego dopasowania silnego wczytywana jest wartość z poziomu leżącego poniżej (znajdująca się w pamięci globalnej). Jeśli dopasowanie silne wystąpi na poziomie o jeden wyższym niż bieżący, to odpowiednia wartość z poziomu bieżącego zapisywana jest do pamięci globalnej. Wartości żadnych innych komórek nie są zapisywane wprost w tablicy przechowującej wartości pudełek poniższych znajdującej się w pamięci globalnej.

Usprawnienie polegające na redukcji rozmiaru przetwarzanej macierzy dzięki wyznaczeniu punktów wejścia-wyjścia [110] również zostało zastosowane. W tym celu, na etapie przetwarzania wstępnego, z obliczeń wykluczane są pudełka, o których wiadomo, że zawierają tylko wartości $-\infty$. W eksperymentach badano algorytm tylko z tym usprawnieniem. Wynik końcowy odczytywany jest z prawej-dolnej komórki prawego-dolnego pudełka.

Kod jądra dla tego algorytmu jest podobny do kodu jądra algorytmu wyznaczania podciągu LCS za pomocą programowania dynamicznego, wobec czego zostanie on tu pominięty. Główne różnice to: inne wyznaczanie zbioru równolegle obliczanych pudełek oraz przechowywanie wartości niektórych komórek z poziomów niższych (dla obsługi dopasowań silnych).

Poniżej zostanie rozważone, jakie jest teoretyczne przyspieszenie proponowanego algorytmu równoległego w stosunku do algorytmu sekwencyjnego dla procesora CPU. Obliczenia wykonywane wewnątrz pojedynczego pudełka są bardzo podobne do tych wykonywanych w równoległym algorytmie programowania dynamicznego dla problemu LCS (7.6) z jednym wyjątkiem – przetwarzaniem dopasowań silnych. Sumaryczna liczba dopasowań silnych będzie w dalszej części tego podrozdziału oznaczana przez D^sm. Całkowita liczba połączonych dostępów do pamięci w celu obsługi dopasowań silnych to co najwyżej $\Theta(\min(D^\mathrm{sm}, nmr/\eta_2))$.

W całym algorytmie ma miejsce $\Theta(n'+m'+r)$ wywołań kodu jądra, w których obliczanych jest łącznie $\Theta(n'm'r)$ pudełek. W jednym wywołaniu kodu jądra obsługiwanych jest co najwyżej $\Theta(n'm'r/\max(n',m',r))$ pudełek, a więc sumaryczna liczba kroków jest

$$\Theta\left(\max(n',m',r)+\frac{n'm'r}{\eta_1}\right)=\Theta\left(\frac{n'm'r}{\eta_1}\right),\ \text{jeśli}\ \eta_1\le\frac{n'm'r}{\max(n',m',r)}, \tag{9.15}$$

$$\Theta(\max(n',m',r)),\ \text{jeśli}\ \eta_1>\frac{n'm'r}{\max(n',m',r)}. \tag{9.16}$$

Przypadek $\eta_1 \leq \frac{n'm'r}{\max(n',m',r)}$:

Biorąc pod uwagę czas obliczania pojedynczego pudełka (7.14), sumaryczna złożoność czasowa tego algorytmu (wyłączając dostępy do pamięci globalnej) dla tego przypadku wynosi

$$\Theta\left(\frac{n'm'r}{\eta_1} \times \frac{b_{\mathrm{w}}b_{\mathrm{h}}}{\eta_2}\right) = \Theta\left(\frac{nmr}{\eta_1\eta_2}\right). \tag{9.17}$$

Całkowita liczba dostępów do pamięci globalnej jest

$$\Theta\left(\frac{b_{\mathrm{w}}}{\eta_2}n'm'r + \min\left(D^{\mathrm{sm}}, \frac{nmr}{\eta_2}\right)\right) = \Theta\left(\min\left(D^{\mathrm{sm}} + \frac{nmr}{\eta_2 b_{\mathrm{h}}}, \frac{nmr}{\eta_2}\right)\right). \tag{9.18}$$

Z powyższego, sumaryczna złożoność czasowa tego algorytmu wynosi

$$\Theta\left(\frac{nmr}{\eta_1\eta_2} + \min\left(D^{\mathrm{sm}} + \frac{nmr}{\eta_2 b_{\mathrm{h}}}, \frac{nmr}{\eta_2}\right)\right). \tag{9.19}$$

Jeśli $D^{\mathrm{sm}} < \frac{nmr}{\eta_2}$, to

$$\Theta\left(\frac{nmrb_{\mathrm{h}} + D^{\mathrm{sm}}\eta_1 p_2 b_{\mathrm{h}} + \eta_1 nmr}{\eta_1\eta_2 b_{\mathrm{h}}}\right) = \Theta\left(nmr\max\left(\frac{1}{\eta_1\eta_2}, \frac{D^{\mathrm{sm}}}{nmr}, \frac{1}{\eta_2 b_{\mathrm{h}}}\right)\right). \tag{9.20}$$

W związku z tym przyspieszenie wynosi

$$\min\left(\eta_1\eta_2, \frac{nmr}{D^{\mathrm{sm}}}, \eta_2 b_{\mathrm{h}}\right). \tag{9.21}$$

Jeśli $D^{\mathrm{sm}} \geq \frac{nmr}{\eta_2}$, to

$$\Theta\left(\frac{nmr}{\eta_1\eta_2} + \frac{nmr}{\eta_2}\right) = \Theta\left(\frac{nmr}{\eta_2}\right), \tag{9.22}$$

a więc przyspieszenie jest

$$\Theta(\eta_2). \tag{9.23}$$

Wyniki (9.21) i (9.23) mogą być zapisane razem, co daje:

Wniosek 9.9. *Przyspieszenie algorytmu* CLCS-CUDA *w stosunku to algorytmu sekwencyjnego dla procesora CPU wynosi*

$$\max\left(\eta_2, \min\left(\eta_1\eta_2, \frac{nmr}{D^{\mathrm{sm}}}, \eta_2 b_{\mathrm{h}}\right)\right). \tag{9.24}$$

Przypadek $\eta_1 > \frac{n'm'r}{\max(n',m',r)}$:

Całkowita złożoność czasowa tego algorytmu (wyłączając dostępy do pamięci globalnej) dla tego przypadku jest

$$\Theta\left(\frac{\max(n',m',r)\times b_\mathrm{w}b_\mathrm{h}}{\eta_2}\right). \tag{9.25}$$

Koszt dostępów do pamięci globalnej wyznaczony jest w (9.18), z czego sumaryczna złożoność czasowa algorytmu jest

$$\Theta\left(\frac{\max(n',m',r)b_\mathrm{w}b_\mathrm{h}}{\eta_2}+\min\left(D^\mathrm{sm}+\frac{nmr}{\eta_2 b_\mathrm{h}},\frac{nmr}{\eta_2}\right)\right)=$$
$$\Theta\left(\frac{\max(nb_\mathrm{h},mb_\mathrm{w},rb_\mathrm{w}b_\mathrm{h})}{\eta_2}+\min\left(D^\mathrm{sm}+\frac{nmr}{\eta_2 b_\mathrm{h}},\frac{nmr}{\eta_2}\right)\right). \tag{9.26}$$

Jeśli $D^\mathrm{sm} < \frac{nmr}{\eta_2}$, to

$$\Theta\left(\frac{\max(nb_\mathrm{h}^2,mb_\mathrm{w}b_\mathrm{h},rb_\mathrm{w}b_\mathrm{h}^2,D^\mathrm{sm}\eta_2 b_\mathrm{h},nmr)}{\eta_2 b_\mathrm{h}}\right)=$$
$$\Theta\left(nmr\max\left(\frac{b_\mathrm{h}}{\eta_2 mr},\frac{b_\mathrm{w}}{\eta_2 nr},\frac{b_\mathrm{w}b_\mathrm{h}}{\eta_2 nm},\frac{D^\mathrm{sm}}{nmr},\frac{1}{\eta_2 b_\mathrm{h}}\right)\right), \tag{9.27}$$

a więc przyspieszenie wynosi

$$\Theta\left(\min\left(\frac{\eta_2 mr}{b_\mathrm{h}},\frac{\eta_2 nr}{b_\mathrm{w}},\frac{\eta_2 nm}{b_\mathrm{w}b_\mathrm{h}},\frac{nmr}{D^\mathrm{sm}},\eta_2 b_\mathrm{h}\right)\right). \tag{9.28}$$

Jeśli $D^\mathrm{sm} \geq \frac{nmr}{\eta_2}$, to (9.26) można wyrazić następująco:

$$\Theta\left(\frac{\max(nb_\mathrm{h},mb_\mathrm{w},rb_\mathrm{w}b_\mathrm{h},nmr)}{\eta_2}\right)=\Theta\left(\frac{nmr}{\eta_2}\right), \tag{9.29}$$

a więc przyspieszenie jest

$$\Theta(\eta_2). \tag{9.30}$$

Wyniki (9.28) i (9.30) mogą być zapisane łącznie, co można podsumować następująco:

Wniosek 9.10. *Przyspieszenie algorytmu równoległego* CLCS-CUDA *w stosunku do algorytmu sekwencyjnego dla procesora CPU jest*

$$\Theta\left(\max\left(\eta_2,\min\left(\frac{\eta_2 mr}{b_\mathrm{h}},\frac{\eta_2 nr}{b_\mathrm{w}},\frac{\eta_2 nm}{b_\mathrm{w}b_\mathrm{h}},\frac{nmr}{D^\mathrm{sm}},\eta_2 b_\mathrm{h}\right)\right)\right). \tag{9.31}$$

9.4.2. Wyniki eksperymentalne

W celu porównania zaproponowanego algorytmu do algorytmów dla procesorów CPU wykonane zostały eksperymenty z użyciem takiego samego następującego zestawu komputerowego, kompilatora i biblioteki CUDA jak w podrozdz. 7.6.5.

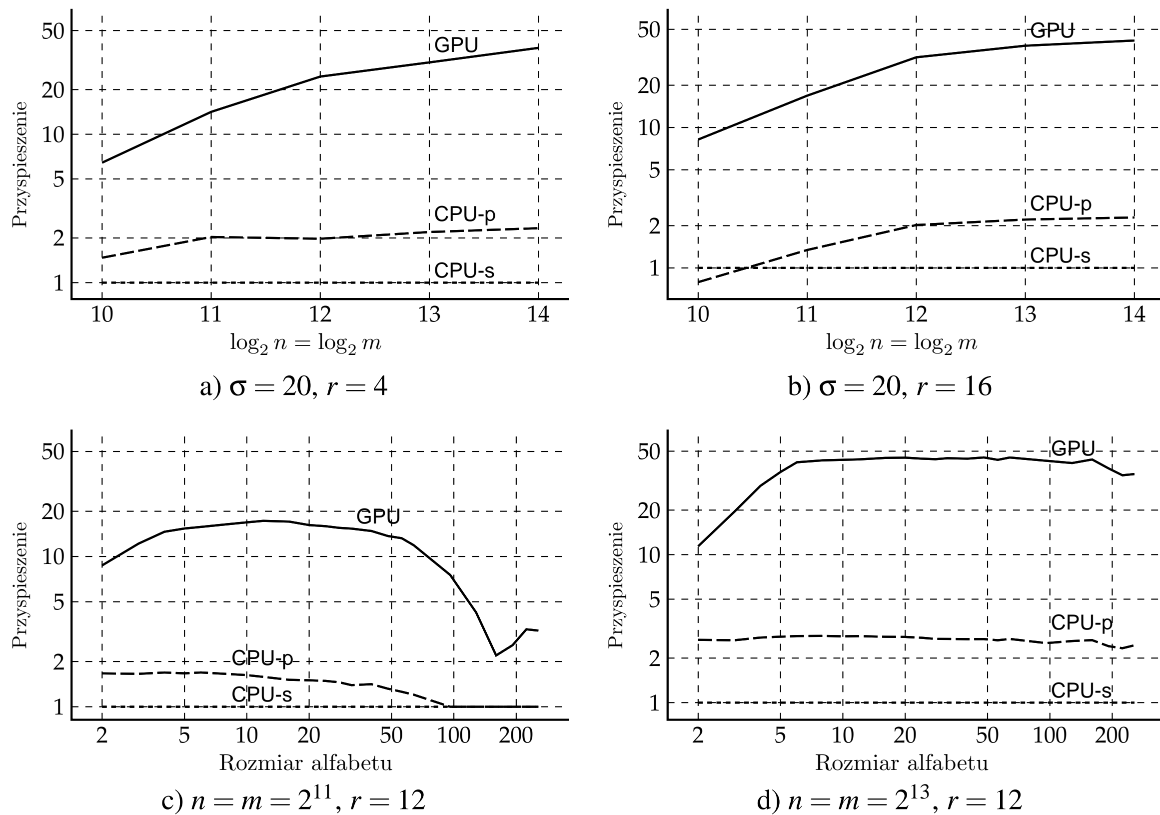
Wszystkie czasy są medianami ze 101 wykonań. Zamiast czasów absolutnych na wykresach pokazane jest przyspieszenie w stosunku do odpowiednich algorytmów sekwencyjnych (działających na jednym rdzeniu procesora CPU). Algorytmy na wykresach oznaczane są następująco:

- CPU-s – algorytm sekwencyjny dla procesorów CPU,
- CPU-p – algorytm równoległy dla procesorów CPU (zastosowany procesor CPU zawiera 4 rdzenie, ale wstępne eksperymenty pokazały, że lepiej jest uruchamiać naraz 32 wątki),
- GPU – algorytm równoległy dla procesorów GPU z tablicą pośredniczącą.

Ponieważ problem CLCS pojawił się w bioinformatyce i dotyczył porównywania łańcuchów białkowych, w eksperymentach, których wyniki pokazane są na rys. 9.20, ustalono $\sigma = 20$. Dla krótkich i średniej długości ciągów ukierunkowujących oraz dla dostatecznie długich ciągów głównych algorytm GPU osiąga przyspieszenie 40–50 w stosunku do ulepszonego algorytmu sekwencyjnego China i in.

Dla krótkich ciągów głównych ($n = m = 2^{10}$) przyspieszenie jest znacznie mniejsze, ponieważ: (i) pudełka są małych rozmiarów, (ii) liczba pudełek jest mała i nie wszystkie multiprocesory są obciążone, albo obciążone są w niewielkim stopniu i obciążenie nie jest zbalansowane. Algorytm CPU-p jest tylko ok. 2,5 razy szybszy niż CPU-s, głównie z uwagi na duży rozmiar przetwarzanych danych, co prawdopodobnie powoduje większą liczbę chybień w pamięci podręcznej, jeśli naraz wiele wątków przetwarza te dane.

Na rysunkach 9.20c–d pokazana jest zależność przyspieszenia od rozmiaru alfabetu dla ciągów głównych różnych długości. Dla relatywnie krótkich ciągów głównych ($n = m = 2^{11}$) przyspieszenie wynosi ok. 12 dla małych alfabetów i maleje do ok. 3 dla alfabetów dużych. Spowodowane to jest faktem, że dla dużych alfabetów i krótkich ciągów głównych zastosowanie techniki punktów wejścia-wyjścia powoduje redukcję przetwarzanej macierzy do bardzo małego jej fragmentu. Często już na etapie przetwarzania wstępnego (po wyznaczeniu punktów wejścia-wyjścia) wiadomo, że szukany podciąg nie istnieje. Dla dłuższych ciągów głównych ($n = m = 2^{13}$) przyspieszenie jest znacznie lepsze (bliskie 50) i nie maleje tak szybko, nawet dla $\sigma = 256$. Przyspieszenia algorytmów GPU i CPU-p są na tyle różne, że ewentualny algorytm hybrydowy CPU+GPU, który byłoby bardzo trudno zaproponować z uwagi na duże ilości danych przesyłanych do/z pamięci i problemy synchronizacyjne, mógłby być szybszy od algorytmu GPU tylko nieznacznie.



Rys. 9.20. Porównanie przyspieszeń uzyskiwanych przez równoległy algorytm programowania dynamicznego dla procesorów GPU (problemu CLCS). Rozmiary pudełek: $b_{\mathrm{w}} = \min(\max(m/2^5, 2^6), 2^8)$, $b_{\mathrm{h}} = \min(\max(m/2^6, 2^5), 2^6)$, $b_{\mathrm{h}}^{\mathrm{cpu}}, b_{\mathrm{w}}^{\mathrm{cpu}} \in [2^8, 2^9]$

Fig. 9.20. Comparison of the speedup for the classical CLCS dynamic programming algorithm. Box sizes: $b_{\mathrm{w}} = \min(\max(m/2^5, 2^6), 2^8)$, $b_{\mathrm{h}} = \min(\max(m/2^6, 2^5), 2^6)$, $b_{\mathrm{h}}^{\mathrm{cpu}}, b_{\mathrm{w}}^{\mathrm{cpu}} \in [2^8, 2^9]$

9.5. Podsumowanie

Problem najdłuższego ukierunkowanego wspólnego podciągu (CLCS) był rozważany w latach 2003–2010 w co najmniej kilkunastu publikacjach. Główne jego zastosowania znajdują się przede wszystkim w bioinformatyce. W niniejszym rozdziale przedstawiono krótko istniejące dla tego problemu algorytmy oraz omówiono nieco dokładniej algorytm China i in. [42], który był punktem wyjścia do części badań autora.

W dalszej części rozdziału zaproponowano kilka algorytmów rozwiązywania tego problemu. Historycznie pierwszy był algorytm dedykowany dla niewielkiej liczby dopasowań oparty na metodzie Hunta–Szymanskiego (podrozdz. 9.2) znanej z problemu LCS. Był to pierwszy algorytm, którego złożoność czasowa, jeśli wyrazić ją w zależności od długości ciągu wejściowego, była niższa niż iloczyn długości wszystkich trzech ciągów wejściowych. W praktyce ten algorytm okazuje się zwykle kilkakrotnie szybszy od algorytmów znanych wcześniej. Algorytm ten został następnie ulepszony przez zastosowanie techniki punktów wejścia-wyjścia, znanej z literatury dla problemu uliniawiania ukierunkowanego ciągów (CPSA). Kolejną propozycją

(podrozdz. 9.3) był pierwszy algorytm równoległości bitowej do problemu CLCS. Eksperymenty pokazały, że algorytm ten jest kilkunastokrotnie szybszy od algorytmu China i in. oraz kilkakrotnie szybszy od zaproponowanego algorytmu opartego na metodzie Hunta–Szymanskiego.

W ostatniej części niniejszego rozdziału zaproponowano algorytm równoległy dla procesorów graficznych (GPU). Algorytm ten jest zrównolegloną wersją algorytmu China i in., która została przeniesiona na procesory GPU, zgodnie ze schematem przedstawionym w podrozdziale 7.6.2. W przeprowadzonych eksperymentach okazało się, że uzyskane przyspieszenie w stosunku do algorytmu China i in. wynosiło często ok. 40–50, co jest wynikiem bardzo dobrym i pozwala mieć nadzieję, że przyspieszenie będzie jeszcze większe dla nowszych procesorów graficznych.

Jedną z najważniejszych otwartych kwestii dotyczących problemu CLCS pozostaje pytanie, czy możliwe jest skonstruowanie algorytmu o złożoności czasowej $o(nmr)$ (niezależnie od długości ciągu wynikowego). Również w algorytmie równoległości bitowej pojawia się nierozwiązany problem, czy możliwe jest uzyskanie przyspieszenia liniowo zależnego od długości słowa komputerowego. Aktualna wersja tego algorytmu cechuje się przyspieszeniem nieco mniejszym.

Z ciekawych prac teoretycznych warto wymienić ostatnio uzyskany wynik [14], dotyczący algorytmu dla problemu CLCS, w którym ciągi wejściowe zostały skompresowane za pomocą algorytmu kodowania długości serii (RLE). Niestety, z uwagi na charakter typowych danych biologicznych taki algorytm nie oferuje znaczącej kompresji. Nie są, jak na razie, znane algorytmy dla tego problemu wykorzystujące inne, bardziej zaawansowane metody kompresji, które były stosowane w problemie LCS.

10. Najdłuższy scalony wspólny podciąg

10.1. Wprowadzenie

Problem wyznaczania *najdłuższego scalonego wspólnego podciągu* (ang. *merged longest common subsequence*, MerLCS) [116], podobnie do problemu CLCS rozważanego w poprzednim rozdziale, był inspirowany badaniami prowadzonymi w bioinformatyce. Danymi wejściowymi są tu trzy ciągi: Z, A, B, a oczekiwanym wynikiem jest najdłuższy ciąg S będący podciągiem Z, który może zostać podzielony na dwa podciągi: S' i S'' takie, że S' jest podciągiem A, a S'' – podciągiem B. Sformułowanie tego problemu wiąże się z faktem, że znajdowanie *przeplatających się związków* (ang. *interleaving relationship*) pomiędzy ciągami bywa pomocne dla weryfikacji pewnych hipotez biologicznych. Jedną z nich jest *duplikacja całego genomu* (ang. *whole genome duplication*, WGD), po której występuje masowa utrata genów. W [125] Kellis i in. pokazali, że gatunek *Saccharomyces cerevisiae* powstał przez duplikację ośmiu rodzicielskich chromosomów, po której wystąpiła masowa utrata genów, co jest dowodem na występowanie WGD w trakcie ewolucji organizmów żywych. Utrata genów wystąpiła w regionach sparowanych, przez co została zachowana przynajmniej jedna kopia każdego genu. Dowód na wystąpienie tego zjawiska opiera się na porównaniu DNA dwóch gatunków drożdży: *Saccharomyces cerevisiae* i *Kluyveromyces waltii*, które pochodzą od wspólnego przodka, a rozdzieliły się w trakcie ewolucji przed wystąpieniem WGD. Oba te gatunki są związane relacją mapowania 1:2, spełniającą kilka kryteriów (szczegóły w [125]), m.in., każdy z siostrzanych regionów w genomie *S. cerevisiae* zawiera uporządkowany podciąg genów w odpowiadającym mu regionie genomu *K. waltii*. Ponadto, oba siostrzane podciągi odpowiednio scalone zawierają prawie wszystkie geny *K. waltii*. Rozwiązując problem MerLCS dla tych trzech ciągów (dwa regiony w genomie jednego gatunku i jeden region w genomie drugiego, bliskiego gatunku), można sprawdzić, czy po rozdzieleniu się tych gatunków miała miejsce WGD.

Peng i in. [173] pokazują także możliwe zastosowania problemu MerLCS w innych dziedzinach, np. przy porównywaniu sygnałów, kiedy do dyspozycji są trzy ciągi: jeden kompletny i dwa zniekształcone (zaszumione, niekompletne itp.) i należy zweryfikować, czy zniekształcone ciągi mogły powstać przez zniekształcenie ciągu kompletnego.

W problemie MerLCS danymi wejściowymi są ciągi: $Z = z_1z_2 \dots z_r$, $A = a_1a_2 \dots a_n$, $B = b_1b_2 \dots b_m$, składające się z symboli alfabetu $\Sigma \in \mathbb{Z}$. Podobnie jak w innych rozdziałach tej części, bez utraty ogólności można założyć, że $m \leq n$.

Problem 10.1 (Najdłuższy scalony wspólny podciąg, MerLCS)**.** *Dla ciągów Z, A, B znaleźć taki ciąg* $S = s_1s_2 \dots s_\ell$*, o maksymalnej długości, będący podciągiem Z, którego podciąg* $S' =$

$s_{i_1}s_{i_2}\dots s_{i_k}$, gdzie $1 \le i_1 < i_2 < \dots < i_k \le \ell$ jest podciągiem ciągu A, a ciąg S'' otrzymany z S przez usunięcie symboli z pozycji $i_1, i_2, \dots, i_k$ jest podciągiem ciągu B.

Powyższa definicja oznacza, że podciąg MerLCS dla Z, A, B jest najdłuższym wspólnym podciągiem Z oraz dowolnego ciągu, który można otrzymać scalając ciągi A i B.

Przykład 10.1 (Najdłuższy scalony wspólny podciąg, MerLCS)**.** *Dla ciągów $Z =$ $\underline{\mathsf{ABA}}\mathsf{D}$, $A =$ $\mathsf{DD}\underline{\mathsf{A}}$ oraz $B =$ $\underline{\mathsf{BA}}\mathsf{C}$ najdłuższym ukierunkowanym wspólnym podciągiem jest m.in. $S =$ ABA. W ciągach Z, A, B podkreślono symbole tworzące podciąg MerLCS.*

W niniejszym rozdziale używane są takie same operacje bitowe, jakie były stosowane w poprzednich rozdziałach. Ponadto, dla dowolnego wektora bitowego W przez $W^{[i,j]}$ będzie oznaczany ciąg bitów z W rozpoczynający się na pozycji i-tej, a kończący na pozycji j-tej, a przez $W^{[i]}$ – i-ty bit wektora bitowego W. Notacja ta będzie stosowana do specyfikowania pojedynczego (lub jego części) słowa komputerowego w tablicy emulującej długi wektor bitowy.

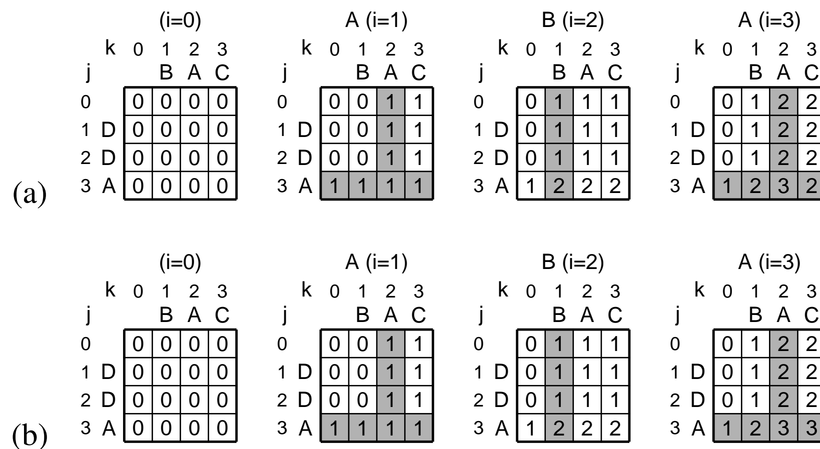
W literaturze znane są dwa algorytmy rozwiązywania problemu MerLCS. Algorytm Huanga i in. [116] opiera się na metodzie programowania dynamicznego i charakteryzuje się złożonością czasową $\Theta(nmr)$. Wymaga on, w zależności od implementacji, $\Theta(nmr)$ lub $\Theta(nm)$ słów pamięci. W [116] dyskutowany jest także problem, będący uogólnieniem problemu MerLCS. Nakłada się w nim pewne dodatkowe ograniczenia na to, jak można scalać ciągi A i B. Problem ten (Block-MerLCS) nie będzie jednak przedmiotem rozważań w niniejszym rozdziale. Algorytm Huanga i in. dla problemu MerLCS sprowadza się do następującej formuły rekurencyjnej:

$$M(i,j,k) = \max \begin{cases} M(i-1,j-1,k)+1, & \text{jeśli } z_i = a_j, \\ M(i-1,j,k-1)+1, & \text{jeśli } z_i = b_k, \\ \max \begin{cases} M(i-1,j,k), \\ M(i,j-1,k), \\ M(i,j,k-1), \end{cases} & \text{jeśli } (z_i \neq a_j) \wedge (z_i \neq b_k), \end{cases} \tag{10.1}$$

z warunkami brzegowymi określonymi następująco:

$$\begin{aligned} M(0,j,k) &= 0, \\ M(i,0,0) &= 0, \\ M(i,j,0) &= \mathrm{LLCS}(Z_i,A_j), \\ M(i,0,k) &= \mathrm{LLCS}(Z_i,B_k), \end{aligned} \tag{10.2}$$

dla wszystkich poprawnych wartości i, j, k. Przez LLCS oznaczana jest długość podciągu LCS. Po zakończeniu obliczeń w $M(r,n,m)$ znajduje się długość ciągu wynikowego, a sam ciąg można uzyskać odpowiednio przechodząc macierz M począwszy od $M(r,n,m)$.



Rys. 10.1. Przykład obliczeń dla problemu MerLCS i ciągów $Z = \mathsf{ABA}$, $A = \mathsf{DDA}$, $B = \mathsf{BAC}$ wykonywanych zgodnie z (a) zależnością oryginalną (10.1), (b) zależnością poprawioną (10.3). Wyszarzone komórki oznaczają dopasowania

Fig. 10.1. Example of computation of the MerLCS for $Z = \mathsf{ABA}$, $A = \mathsf{DDA}$, $B = \mathsf{BAC}$ according to (a) original formula (10.1), (b) fixed formula (10.3). Gray cells denote matches

Równanie (10.1) zawiera jednak pewien błąd, co można zaobserwować na rys. 10.1a, gdzie znajduje się przykład macierzy uzyskanej według tej zależności. Według algorytmu Penga i in. wynikiem jest podciąg długości 2, ponieważ $M(3,3,3) = 2$, podczas gdy poprawnym wynikiem jest podciąg długości 3, gdyż ABA jest podciągiem $Z = \mathsf{ABA}$, który może zostać rozdzielony na A (podciąg $A = \mathsf{DDA}$) oraz BA (podciąg $B = \mathsf{BAC}$). Na szczęście błąd ten łatwo skorygować otrzymując zależność:

$$M(i,j,k) = \max \begin{cases} M(i-1,j-1,k)+1, & \text{jeśli } z_i = a_j, \\ M(i-1,j,k-1)+1, & \text{jeśli } z_i = b_k, \\ M(i-1,j,k), \\ M(i,j-1,k), \\ M(i,j,k-1). \end{cases} \tag{10.3}$$

Warunki brzegowe pozostają bez zmian.

Lemat 10.1. *Zależność (10.3) z warunkami brzegowymi (10.2) poprawnie wyznacza podciąg MerLCS dla ciągów Z, A, B.*

Dowód. Dowód jest identyczny jak dowód twierdzenia 1 z [116] z jednym wyjątkiem. Dla dopasowania (sytuacji, w której $z_i = a_j$ lub $z_i = b_k$) nie można zapomnieć o możliwości, że skrócenie jednego z ciągów o jeden symbol może doprowadzić do lepszego wyniku niż podążenie za dopasowaniem i skrócenie dwu ciągów. ■

Podstawową ideą algorytmu Penga i in. [173] jest ograniczenie obliczeń tylko do komórek macierzy M reprezentujących dopasowania, których zwykle jest znacznie mniej niż nmr.

Zwłaszcza w przypadku dużych alfabetów macierz programowania dynamicznego jest rzadka. Czas potrzebny na obliczenie pojedynczej komórki macierzy może być wprawdzie większy niż $O(1)$, ale sumarycznie dla całego algorytmu złożoność czasowa jest $O(\ell mr)$, gdzie ℓ jest długością ciągu wynikowego. Zwykle, dla dużych alfabetów ℓ jest znacznie mniejsze niż n, dzięki czemu algorytm ten może być istotnie szybszy.

10.2. Algorytm równoległości bitowej

10.2.1. Podstawowe idee

Niniejszy podrozdział zawiera opis algorytmu równoległości bitowej, który został zaproponowany przez autora w [70]. Zanim jednak zostanie przedstawiony ten algorytm, konieczne jest dowiedzenie kilku lematów.

Lemat 10.2. *Wartość komórki $M(i,j,k)$ jest równa bądź większa o 1 niż wartość dowolnego z następujących jej sąsiadów: $M(i-1,j,k)$, $M(i,j-1,k)$, $M(i,j,k-1)$.*

Dowód. $M(i,j,k)$ jest długością podciągu MerLCS dla Z_i, A_j, B_k. Wymienione komórki sąsiednie zawierają długości podciągu MerLCS dla tych samych ciągów, przy czym jeden z nich jest okrojony o ostatni symbol. Niemożliwe jest, aby okrojenie jednego ciągu o ostatni symbol spowodowało zmniejszenie długości podciągu MerLCS o więcej niż 1, ponieważ usunięcie ostatniego symbolu z jednego ciągu może co najwyżej spowodować, że ten symbol nie znajdzie się w ciągu wynikowym. Podobnie, nie jest możliwe, żeby okrojenie jednego z ciągów wejściowych spowodowało, że długość podciągu MerLCS wzrośnie. ■

Na podstawie lematu 10.2 możliwa jest reprezentacja trójwymiarowej macierzy M przez dwuwymiarową macierz G, zawierającą *wektory zmian* będące wektorami liczb całkowitych z zakresu $[1,r]$. Wektory te zdefiniowane są następująco:

$$i \in G(j,k) \iff M(i,j,k) - M(i-1,j,k) = 1 \text{ dla } 1 \le i \le r. \tag{10.4}$$

Równoważność pomiędzy macierzami M i G wynika z faktu, że:

$$M(i,j,k) = |\{x : x \in G(j,k) \wedge x \le i\}|. \tag{10.5}$$

Łatwo można zauważyć, że definicja wektorów zmian jest analogiczna do sposobu reprezentacji zmian w macierzy programowania dynamicznego dla problemu LCS przedstawionej w [120].

Zanim zostanie sformułowany bezpośredni algorytm wyznaczania $G(j,k)$, dogodne jest zapisanie zależności (10.3) w następujący sposób:

$$M'(i,j-1,k) = \max\begin{cases} M'(i-1,j-1,k), & \\ M(i-1,j-1,k)+1, & \text{jeśli } z_i = a_j, \\ M(i,j-1,k), & \end{cases} \tag{10.6}$$

$$M''(i,j,k-1) = \max\begin{cases} M''(i-1,j,k-1), & \\ M(i-1,j,k-1)+1, & \text{jeśli } z_i = b_k, \\ M(i,j,k-1), & \end{cases} \tag{10.7}$$

$$M(i,j,k) = \max\begin{cases} M'(i,j-1,k), \\ M''(i,j,k-1). \end{cases} \tag{10.8}$$

Można zauważyć, że pierwsze składniki funkcji maksimum w (10.6)–(10.7) oznaczają tylko tyle, że wartości M', M'', M nie maleją przy wzroście i (por. trzeci składnik funkcji maksimum w (10.3)). Wymaganie to jest inherentne dla definicji G.

Wektor zmian $G'(j-1,k)$ dla $M'(i,j-1,k)$ może być wyznaczony na podstawie $G(j-1,k)$ następująco:

1. $G'(j-1,k) \leftarrow G(j-1,k)$.
2. Dla każdego i od r do 1 takiego, że $z_i = a_j$ wykonaj:
   a) jeśli $i \in G'(j-1,k)$ nie rób nic,
   b) w przeciwnym przypadku wstaw i do $G'(j-1,k)$ i usuń z $G'(j-1,k)$ najmniejszą wartość całkowitą (jeśli taka istnieje) większą niż i.

W podobny sposób obliczany jest wektor $G''(j,k-1)$.

Lemat 10.3. *Powyższy algorytm poprawnie oblicza wektory zmian $G'(j-1,k)$ i $G''(j,k-1)$ odpowiadające $M'(i,j-1,k)$ i $M''(i,j,k-1)$ dla wszystkich poprawnych wartości i.*

Dowód. *Rangą* $h(x)$ elementu x w wektorze zmian będzie nazywany indeks tego elementu w posortowanym zbiorze wartości znajdujących się w wektorze zmian. Można zauważyć, że dla każdego i w $G'(j-1,k)$ o randze $h(i)$ zachodzi:

- $M(i,j-1,k) = h(i)$, a więc $i \in G(j-1,k)$ lub
- $M(i-1,j-1,k) = h(i)-1$ i $z_i = a_j$.

Wobec tego i należy do $G'(j-1,k)$ wtedy i tylko wtedy, gdy:

- $i \in G(j-1,k)$ i nie istnieje $i' < i$ takie, że $M(i'-1,j-1,k) = h(i')-1$ i $z_{i'} = a_j$ lub
- i jest najmniejszym indeksem takim, że $M(i-1,j-1,k) = h(i)-1$ i $z_i = a_j$.

Dowód dla $G''(j,k-1)$ jest analogiczny. ∎

Łatwo można zauważyć, że powyższy sposób wyznaczania wektorów zmian jest dokładnie taki sam jak pokazany w [120, Observation 4] dla algorytmu LCS-BP-LENGTH rozwiązującego problem LCS.

Dysponując wektorami $G'(j-1,k)$, $G''(j,k-1)$, można wyznaczyć $G(j,k)$. Na podstawie (10.5) oraz (10.8) wiadomo, że:

$$M(i,j,k) = \max \begin{cases} |\{x : x \in G'(j-1,k) \wedge x \leq i\}| \\ |\{x : x \in G''(j,k-1) \wedge x \leq i\}| \end{cases}. \tag{10.9}$$

W związku z tym, aby obliczyć $G(j,k)$, należy postępować jak poniżej:

1. Dla każdej pary liczb całkowitych z $G'(j-1,k)$ i $G''(j,k-1)$ jednakowej rangi weź mniejszą z nich i wstaw do $G(j,k)$.
2. Jeśli wśród elementów w $G'(j-1,k)$ i $G''(j,k-1)$ występuje element o unikalnej randze, to wstaw go do $G(j,k)$.

10.2.2. Algorytm

Do efektywnej reprezentacji wektorów zmian mogą być zastosowane wektory bitowe. Ponieważ wszystkie wektory zmian w proponowanym algorytmie zawierają liczby całkowite z zakresu $[1,r]$, to długość każdego z używanych wektorów bitowych wynosi r. (Jedynie dla wygody prezentacji bity w wektorach bitowych będą numerowane od 1). Dla każdego wektora zmian $G(j,k), G'(j-1,k), G''(j,k-1)$ istnieje odpowiedni wektor bitowy $W(j,k), W'(j-1,k)$, $W''(j,k-1)$ zdefiniowany następująco: wszystkie bity mają wartość 1, z wyjątkiem tych, których indeksy występują w odpowiednim wektorze zmian.

Używana będzie także tablica wektorów bitowych Y_x, dla każdego $x \in \Sigma$. Każdy wektor Y_x zdefiniowany jest następująco: wszystkie bity mają wartość 0, z wyjątkiem tych, które odzwierciedlają pozycję symbolu x w T. Te wektory bitowe reprezentują pozycje, na których występują dopasowania przy porównywaniu ciągu T z konkretnym symbolem z A lub B. Jest to dokładnie ta sama reprezentacja jak stosowana w algorytmie LCS-BP-LENGTH (por. podrozdz. 7.3.4).

W celu obliczenia $W'(j-1,k)$ na podstawie $W(j-1,k)$ używa się wektora Y_{a_j}, jako że reprezentuje on pozycje, na których w Z występuje a_j. Ponieważ operacje wymagane w tym miejscu są identyczne do wykonywanych w algorytmie LCS-BP-LENGTH, więc zostanie użyta następująca sekwencja operacji bitowych z [120]:

$$W'(j-1,k) \leftarrow W(j-1,k) \,\&\, Y_{a_j},$$
$$W'(j-1,k) \leftarrow (W(j-1,k) + W'(j-1,k)) \mid (W(j-1,k) - W'(j-1,k)).$$

Analogiczną sekwencję operacji stosuje się do wyznaczenia $W''(j,k-1)$.

Obliczenie $W(j,k)$ na podstawie $W'(j-1,k)$ i $W''(j,k-1)$ jest nieco bardziej skomplikowane. Przydatne okażą się tu poniższe lematy.

Lemat 10.4. *Dla dowolnego* $x \in G'(j-1,k)$ *o randze* $h(x)$ *i* $y \in G''(j,k-1)$ *o randze* $h(y)$ *zachodzi:*

a) jeśli $x > y$*, to* $h(x) \geq h(y)$*,*

b) jeśli $x < y$*, to* $h(x) \leq h(y)$*,*

c) jeśli $x = y$*, to* $|h(x) - h(y)| \leq 1$.

Dowód. Dowód zostanie przeprowadzony przez zaprzeczenie.

Przypadek a: Niech $x > y$ i $h(x) < h(y)$. Z (10.6) i (10.7) wiadomo, że

$$h(x) = M'(x, j-1, k) = M(x-1, j-1, k) + 1, \tag{10.10}$$

$$h(y) = M''(y, j, k-1) = M(y-1, j, k-1) + 1. \tag{10.11}$$

Z założenia $h(x) < h(y)$ wynika, że

$$M'(x, j-1, k) < M''(y, j, k-1). \tag{10.12}$$

Na podstawie (10.8) wiadomo

$$M(y, j, k) \geq M''(y, j, k-1), \tag{10.13}$$

a z $x > y$ wynika, że

$$M(x-1, j-1, k) \geq M(y, j-1, k). \tag{10.14}$$

Łącząc wyniki z (10.10)–(10.14), otrzymuje się:

$$M(y, j, k) \geq M''(y, j, k-1) > M'(x, j-1, k) > M(x-1, j-1, k) \geq M(y, j-1, k), \tag{10.15}$$

a więc

$$M(y, j, k) - M(y, j-1, k) > 1, \tag{10.16}$$

co jest niemożliwe z uwagi na lemat 10.2.

Przypadek b: Dowód jest analogiczny jak dla przypadku a.

Przypadek c: Niech $|h(x) - h(y)| > 1$. Z (10.10) i (10.11) wynika, że

$$|M(x-1, j-1, k) - M(x-1, j, k-1)| > 1. \tag{10.17}$$

Ponieważ zarówno $M(x-1, j-1, k)$, jak i $M(x-1, j-1, k)$ są sąsiadami $M(x-1, j, k)$, więc na podstawie lematu 10.2 wartość powyższej różnicy nie może być większa niż 1. ■

Lemat 10.5. *W celu obliczenia $G(j,k)$ na podstawie $G'(j-1,k)$ i $G''(j,k-1)$ wystarczy:*
1. *Znaleźć sumę zbiorów $G'(j-1,k)$ i $G''(j,k-1)$ otrzymując wielozbiór $G^*(j,k)$, w którym niektóre liczby całkowite mogą występować dwukrotnie.*
2. *Usunąć z $G^*(j,k)$ liczby całkowite o parzystych rangach otrzymując zbiór $G(j,k)$.*

Dowód. Dowód indukcyjny względem rangi x z $G(j,k)$. Dla $x = 1$ najmniejsza liczba całkowita w wielozbiorze $G^*(j,k)$ zostaje przepisana do $G(j,k)$ (jest to element o randze 1 w $G'(j-1,k)$ lub $G''(j,k-1)$). Kolejny najmniejszy element z $G^*(j,k)$ (rangi 2) musi być elementem o randze 1 odpowiednio w $G''(j,k-1)$ lub $G'(j-1,k)$ (lemat 10.4), a więc w $G(j,k)$ znajdzie się mniejszy z elementów o randze 1 spośród wejściowych zbiorów.

Niech teraz $x > 1$ i dla elementów o rangach $2x-3$ i $2x-2$ z $G^*(j,k)$ jeden z nich jest elementem o randze $x-1$ w $G'(j-1,k)$, a drugi elementem o randze $x-1$ w $G''(j,k-1)$. Dla elementów o rangach $2x-1$ i $2x$ w $G^*(j,k)$, jeden z nich musi być elementem o randze x w $G'(j-1,k)$, a drugi elementem o randze x w $G''(j,k-1)$ (lemat 10.4). Mniejszy z tych elementów jest wstawiany do $G(j,k)$. Oczywiście, jeśli w $G^*(j,k)$ znajdują się dwa identyczne elementy, ich rangi różnią się parzystością, więc dokładnie jeden z nich jest wstawiany do $G(j,k)$, dzięki czemu $G(j,k)$ jest zbiorem. ■

Ponieważ liczby całkowite występujące w obu wektorach, $G'(j-1,k)$ i $G''(j,k-1)$, muszą należeć do $G(j,k)$, więc można usunąć je z $G^*(j,k)$ po pierwszym kroku i dodać na samym końcu (tylko po jednej kopii), po usunięciu elementów o parzystych rangach, do $G(j,k)$. Można to wykonać za pomocą operacji różnicy symetrycznej:

$$W(j,k) \leftarrow W'(j-1,k) \oplus W''(j,k-1).$$

Kluczowym etapem algorytmu wyznaczania $W(j,k)$ na podstawie $W'(j-1,k)$ i $W''(j,k-1)$ jest wpisanie wartości 0 do wszystkich bitów wartości 1 o parzystych rangach. W [215, strony 74–77] przedstawiony jest następujący algorytm wyznaczający „parzystość” w 32-bitowym słowie komputerowym x (jego adaptacja dla innych wartości w jest natychmiastowa):

$$y \leftarrow x \oplus (x \ll 1), \qquad y \leftarrow y \oplus (y \ll 2), \qquad y \leftarrow y \oplus (y \ll 4),$$
$$y \leftarrow y \oplus (y \ll 8), \qquad y \leftarrow y \oplus (y \ll 16).$$

Bit o indeksie i w y ma wartość 1 wtedy i tylko wtedy, gdy liczba bitów o wartościach 1 na pozycjach od 0 do i w x jest parzysta. Dzięki temu, aby usunąć bity 1 o parzystych rangach z wektora bitowego $W(j,k)$, należy wyznaczyć parzystość tego wektora i wykonać maskowanie bitów 1 o parzystych rangach.

Kompletny pseudokod algorytmu rozwiązującego problem MerLCS przedstawiony jest na rys. 10.2. W pierwszej części tego algorytmu (wiersze 1–7) rozwiązywany jest problem LCS dla

```
MerLCS-BP(Z, A, B)
Wejście: Z, A, B – podciągi, dla których wyznaczany jest podciąg MerLCS
Wyjście: długość podciągu MerLCS
   {Wyznaczanie warunków brzegowych}
 1  W(0,0) ← 0^r
 2  for k ← 1 to m do
 3      U ← W(0,k−1) & Y_{b_k}
 4      W(0,k) ← (W(0,k−1)+U) | (W(0,k−1)−U)
 5  for j ← 1 to n do
 6      U ← W(j−1,0) & Y_{a_j}
 7      W(j,0) ← (W(j−1,0)+U) | (W(j−1,0)−U)
   {Obliczenia właściwe}
 8  for j ← 1 to n do
 9      for k ← 1 to m do
10          U' ← W(j−1,k) & Y_{a_j}
11          W' ← (W(j−1,k)+U') | (W(j−1,k)−U')
12          U'' ← W(j,k−1) & Y_{b_j}
13          W'' ← (W(j,k−1)+U'') | (W(j,k−1)−U'')
14          U ← W' & W''
15          W ← W' ⊕ W''
16          V ← W
17          for i ← 0 to ⌈log_2 r⌉ − 1 do
18              V ← V ⊕ (V≪ (1≪ i))
19          W(j,k) ← (W & V) | U
   {Wyznaczanie wyniku}
20  ℓ ← 0;      V ← ∼W(n,m)
21  while V ≠ 0^r do
22      V ← V & (V − 1);   ℓ ← ℓ+1
23  return ℓ
```

Rys. 10.2. Algorytm równoległości bitowej wyznaczający długość podciągu MerLCS
Fig. 10.2. Bit-parallel algorithm computing MerLCS length

par ciągów Z i A oraz Z i B, w celu wyznaczenia warunków brzegowych (10.2). W wierszach 8–19 wykonywane są obliczenia właściwe, tj. dla każdej pary (j,k) wyznaczany jest wektor bitowy $W(j,k)$. Końcowa część algorytmu (wiersze 20–23) to zliczenie bitów o wartości 0 w wektorze $W(n,m)$, dzięki czemu otrzymuje się długość podciągu MerLCS.

10.2.3. Szczegóły implementacyjne

W algorytmie zaprezentowanym powyżej zakłada się, że $r \leq w$, tzn. każdy wektor bitowy mieści się w pojedynczym słowie komputerowym, co rzadko jest spełnione. Emulacja wektorów bitowych jako tablic rozmiaru $\lceil r/w \rceil$ słów komputerowych nie jest trudna. Uważnej implementacji wymagają jednak niektóre operacje bitowe i arytmetyczne. Wszystkie operacje odejmowania ($-$), bitowe ($|$, & , $\oplus$) w wierszach 3–4, 6–7, 10–15 mogą być łatwo zaimplementowane na tablicach słów komputerowych, ponieważ nie występują tu żadne przeniesienia pomiędzy

Wiersze 17–19 algorytmu MerLCS-BP na tablicy słów komputerowych

```
1  f ← false
2  for i' ← 0 to ⌈r/w⌉ − 1 do
3      V ← W^[i'w, i'w+w−1]
4      for i ← 0 to ⌈log_2 w⌉ − 1 do
5          V ← V ⊕ (V ≪ (1 ≪ i))
6      if f then V ← ~V
7      if V^[w−1] = 1 then f ← true
8      else f ← false
9      W(j,k)^[i'w, i'w+w−1] ← (W^[i'w, i'w+w−1] & V) | U^[i'w, i'w+w−1]
```

Rys. 10.3. Algorytm symulujący usuwanie bitów 1 o parzystych rangach (wiersze 17–19 z rys. 10.2) w tablicy słów komputerowych (składowa algorytmu MerLCS-BitPar)

Fig. 10.3. Algorithm simulating removal of even rank bits (lines 17–19 in Figure 10.2) in array of computer words (part of the MerLCS-BP algorithm)

kolejnymi słowami. Wyjątek stanowią operacje dodawania w wierszach 4, 7, 11, 13, w których przeniesienie może się pojawić i musi zostać prawidłowo obsłużone. Jest to jednak łatwe do zrealizowania i nie ma wpływu na złożoność czasową algorytmu. Trudniejszy przypadek stanowią wiersze 17–18. W rzeczywistości, pętla ta wykonuje się $\lceil \log_2 w \rceil$ razy dla każdego słowa z tablicy emulującej wektor bitowy, a informacja o parzystości najbardziej znaczącego bitu jest traktowana jako przeniesienie. Podczas wykonywania kolejnego przebiegu tej samej pętli dla kolejnego słowa komputerowego, wartości wszystkich bitów są zamieniane na przeciwne, jeśli to konieczne, co jest sprawdzane na podstawie przeniesienia. Na rys. 10.3 pokazano rzeczywistą implementację wierszy 17–19 z rys. 10.2.

Wniosek 10.1. *Całkowita złożoność czasowa proponowanego algorytmu jest zdeterminowana przez wyznaczanie $W(j,k)$, które wymaga czasu $\Theta(\lceil r/w \rceil \log w)$. W związku z tym złożoność czasowa algorytmu* MerLCS-BP *jest $\Theta(nm\lceil r/w \rceil \log w)$.*

Jest to złożoność lepsza o czynnik $\Theta(w/\log w)$ niż złożoność czasowa algorytmu programowania dynamicznego dla problemu MerLCS [116].

10.3. Wyniki eksperymentalne

W badaniach eksperymentach porównano zaproponowany algorytm z algorytmami znanymi z literatury. W tabelach i na rysunkach algorytmy oznaczone są następująco:

- H08 – algorytm Huanga i in. [116],
- P10 – algorytm Penga i in. [173],
- Our – algorytm zaproponowany w niniejszym rozdziale.

Oryginalny opis algorytmów H08 i P10 dopuszcza różne implementacje. W przypadku gdy żądana jest nie tylko długość podciągu wynikowego, ale też on sam, algorytmy te implemento-

Tabela 10.1

Dane testowe wykorzystane w eksperymentach dla problemu MerLCS

Zestaw	$\|Z\|$	$\|A\|$	$\|B\|$	Opis
dodA	1629	687	942	6-eksonowa sekwencja DNA *Amanita muscaria dodA* gen (gi:2072623) Z – cała sekwencja A – konkatenacja części zawierających eksony B – konkatenacja części zawierających introny
p&d	6000	2480	1756	Sekwencja DNA *Drosophila melanogaster* Z – część chromosomu 2R (gi:73917619) od pozycji 70001 do 76000 A – odwrócone dopełnienie sekwencji pita (gi:24762318) B – death caspase-1 (gi:24762322)

wane są z wykorzystaniem metody Hirschberga (por. podrozdz. 7.3.3), dzięki czemu złożoność pamięciowa nie stanowi problemu, ale czas obliczeń jest istotnie dłuższy niż w przypadku, w którym wynikiem jest tylko długość podciągu MerLCS. Zaproponowany algorytm również może być zaimplementowany z zastosowaniem metody Hirschberga. Eksperymenty przeprowadzone zostały jednak tylko dla przypadku problemu, w którym wynikiem jest długość podciągu MerLCS.

Wszystkie implementacje zostały wykonane w języku C++. Kompilację wykonano za pomocą MS Visual C++ 2008 z opcją kompilacji -Ox, a więc maksymalną optymalizacją ze względu na czas wykonania. Programy były uruchamiane na komputerze wyposażonym w procesor AMD Phenom II X4 810 (zegar 2600 MHz) i 4096 MB pamięci RAM. Każdy eksperyment był powtórzony 201 razy i jako wynik końcowy brano medianę uzyskanych czasów.

W pierwszym eksperymencie zastosowano zestawy danych testowych z [115]. Krótka ich charakterystyka przedstawiona jest w tabeli 10.1. Wyniki eksperymentów zawarte są w tabeli 10.2. Rozmiar alfabetu jest tu niewielki ($\sigma = 4$), co powoduje, że liczba dopasowań jest stosunkowo duża. W związku z tym nie jest niespodzianką, że algorytm P10 dedykowany do przypadku niewielkiej liczby dopasowań działa wolniej niż H08. Algorytm równoległości bitowej zaproponowany w niniejszym rozdziale okazuje się zdecydowanie najszybszy, szybszy około 70 razy od algorytmu H08. Jest to wynikiem zastosowania szybkich operacji bitowych i reprezentacji bitowej, która pozwala na bardziej zwarte przechowywanie odpowiednika macierzy programowania dynamicznego. Co więcej, czynnik $\log w$ występujący w złożoności czasowej pojawił się tam z powodu konieczności wyznaczania w pętli parzystości bitów w słowie komputerowym. Pętla ta jednak ma tylko kilka przebiegów na każde słowo komputerowe i sumaryczna liczba operacji wykonywanych w niej jest mniejsza niż liczba pozostałych operacji bitowych dla typowych wartości w. Wpływ tego czynnika na czas działania algorytmu nie jest więc tak mocno zauważalny, jak sugeruje to oszacowanie złożoności czasowej.

Tabela 10.2

Wyniki eksperymentalne dla danych rzeczywistych dla problemu MerLCS

Zestaw	H08 [ms]	P10 [ms]	Our [ms]	Czas Our / Czas H08	Czas Our / Czas P10
dodA	12 958,42	19 937,84	192,68	67,25	103,48
p&d	321 930,30	645 508,34	4595,00	70,06	140,48

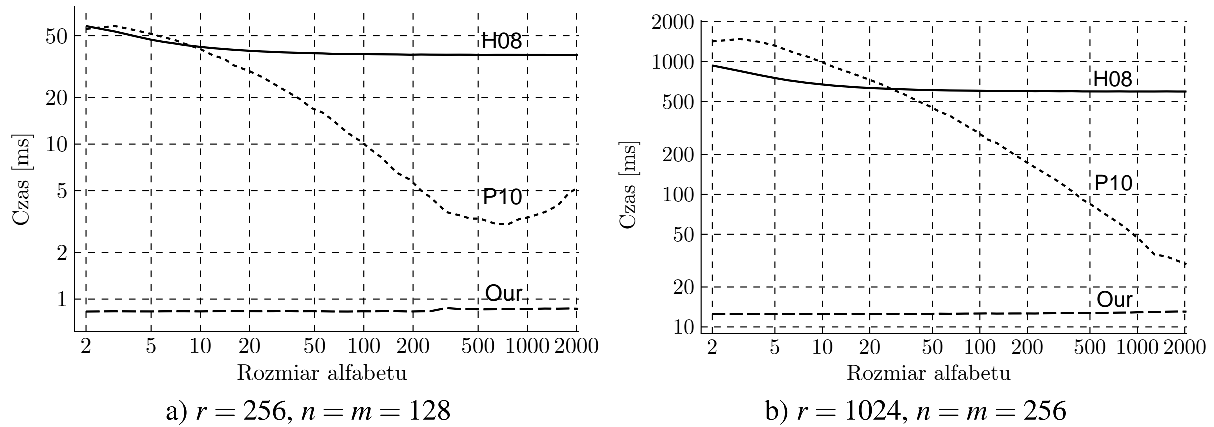
Kolejne eksperymenty zostały przeprowadzone w celu porównania badanych algorytmów dla ciągów różnej długości oraz alfabetów różnych rozmiarów. Z tego powodu przygotowano dane wejściowe z użyciem generatora liczb pseudolosowych o rozkładzie równomiernym. W eksperymentach dane były w każdym wykonaniu inne. Dla ustalonych długości ciągów wyznaczano czasy działania algorytmów w zależności od rozmiaru alfabetu. Wyniki przedstawione na rys. 10.4 pokazują, że algorytm P10 jest szybszy od algorytmu H08 dla alfabetów większych niż ok. 10–25 symboli. Dokładna wartość tego progu zależy od długości ciągów. Algorytm proponowany jest od 41- do 74-krotnie szybszy niż algorytm H08 dla obu przypadków (a i b), a od algorytmu P10 jest szybszy od 2 do 112 razy (silnie zależy to od rozmiaru alfabetu).

W trzecim eksperymencie rozmiar alfabetu został ustalony na dwie wartości: $\sigma = 4$ oraz $\sigma = 128$ i badany był wpływ długości ciągów na czas działania algorytmów. Wyniki przedstawione są na rys. 10.5. Dla relatywnie krótkich ciągów (przypadki a i b) proponowany algorytm jest szybszy od algorytmu H08 od 45 do 68 razy, a od algorytmu P10 od 35 do 159 razy. Podobne wyniki otrzymano dla dłuższych ciągów (przypadki c i d).

Podsumowując wyniki eksperymentów, można stwierdzić, że proponowany algorytm równoległości bitowej jest szybszy od algorytmu programowania dynamicznego, H08 [116] od 40 do 70 razy. Jest to więcej niż można się spodziewać, biorąc pod uwagę, że teoretycznie wyznaczona złożoność czasowa jest lepsza o czynnik $\Theta(\frac{\log w}{w})$. Porównanie z algorytmem specjalizowanym do danych z niewielką liczbą dopasowań, P10 [173], pokazuje, że względna szybkość działania tych algorytmów silnie zależy od liczby dopasowań, która z kolei zależy od rozmiaru alfabetu. We wszystkich eksperymentach proponowany algorytm był co najmniej dwukrotnie, a zwykle 20-krotnie i więcej razy szybszy.

10.4. Podsumowanie

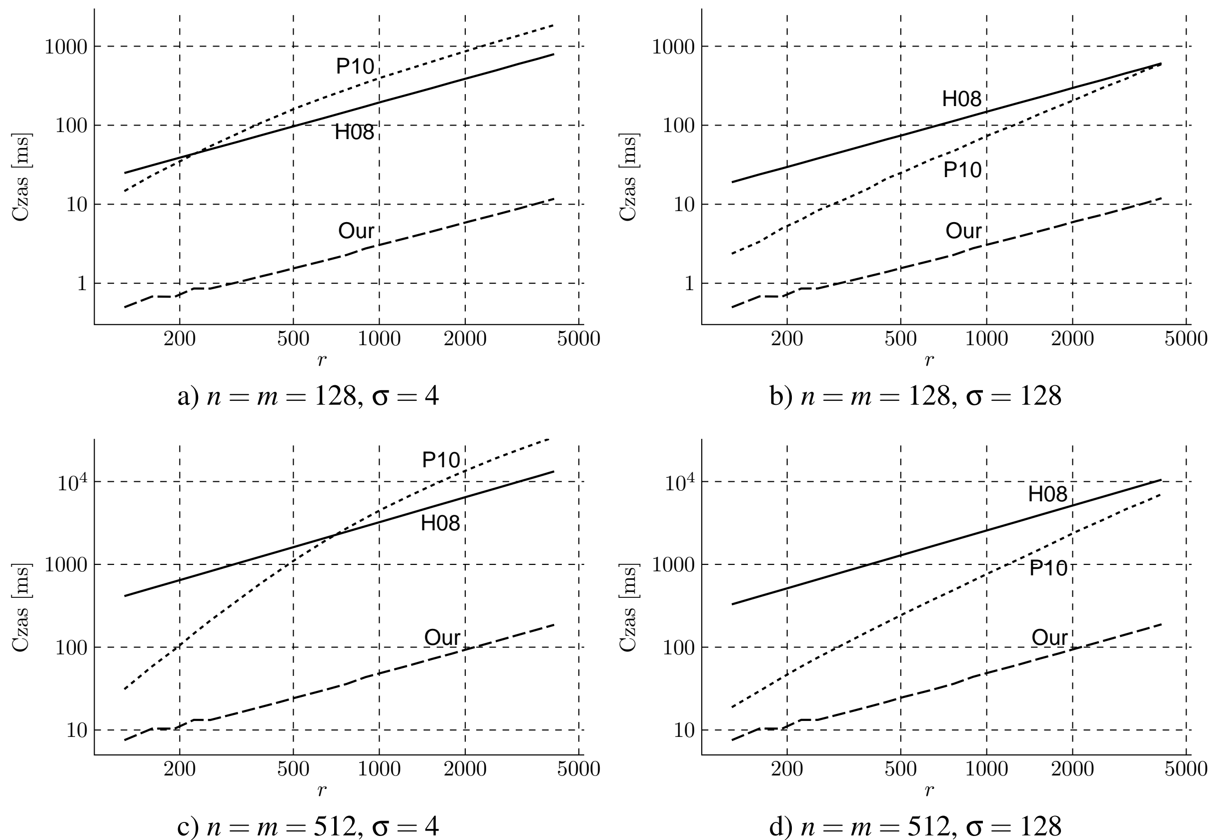
W niniejszym rozdziale omówiono problem wyznaczania najdłuższego scalonego wspólnego podciągu (MerLCS), a także podano jego zastosowania w bioinformatyce. Krótko scharakteryzowano także dwa istniejące algorytmy rozwiązywania tego problemu. Główną część rozdziału stanowi opis proponowanego algorytmu równoległości bitowej dla rozwiązywania problemu MerLCS. Złożoność czasowa tego algorytmu jest $\Theta(nm\lceil r/w \rceil \log w)$. Jak pokazały



a) $r = 256$, $n = m = 128$ b) $r = 1024$, $n = m = 256$

Rys. 10.4. Eksperymentalne porównanie algorytmów wyznaczających długość podciągu MerLCS dla różnych rozmiarów alfabetów

Fig. 10.4. Experimental comparison of the MerLCS length computing algorithms for various alphabet sizes



a) $n = m = 128$, $\sigma = 4$ b) $n = m = 128$, $\sigma = 128$

c) $n = m = 512$, $\sigma = 4$ d) $n = m = 512$, $\sigma = 128$

Rys. 10.5. Eksperymentalne porównanie czasów działania algorytmów wyznaczających długość podciągu MerLCS dla ciągów różnej długości

Fig. 10.5. Experimental comparison of the MerLCS length computing algorithms for various sequence lengths

eksperymenty na danych zarówno rzeczywistych, jak i wygenerowanych sztucznie, proponowany algorytm jest 40–70 razy szybszy od algorytmu opartego na programowaniu dynamicznym [116], którego złożoność czasowa jest $\Theta(nmr)$. Proponowany algorytm jest także zdecydowanie szybszy od algorytmu specjalizowanego dla rzadkich macierzy programowania dynamicznego [173], którego złożoność czasowa jest $O(\ell mr)$. Wyniki te pokazują, jak potężnym podejściem może być równoległość bitowa. Niestety, podejście to ma spore ograniczenia i nie zawsze może być zastosowane, a wymyślenie algorytmu opartego na nim dla konkretnego problemu może być bardzo trudne. W szczególności, pozostaje otwartą kwestią, czy możliwe jest zaproponowanie podobnie szybkiego algorytmu równoległości bitowej dla problemu będącego uogólnieniem problemu MerLCS, w którym na możliwe miejsca scalania ciągów zostały nałożone pewne dodatkowe ograniczenia [116].

Niniejszy rozdział zawiera opis algorytmu wyznaczającego długość podciągu MerLCS. Uogólnienie tego algorytmu na problem wyznaczania także samego podciągu MerLCS jest jednak bezpośrednie. Wystarczy w tym celu zastosować metodę Hirschberga (podrozdz. 7.3.3) zaproponowaną dla problemu LCS dokładnie w taki sposób, jak to zrobiono w [116] i [173] (por. również podrozdz. 7.3.5) oraz na podstawie bitów o wartości 0 w wektorach bitowych reprezentujących wektory zmian odczytać podciąg MerLCS w sposób analogiczny, jak się to robi w algorytmie równoległości bitowej dla problemu LCS (por. podrozdz. 7.3.4).

Interesującą kwestią, która pozostaje otwarta, jest odpowiedź na pytanie o kres dolny złożoności czasowej algorytmów dla problemu MerLCS. Złożoności czasowe wszystkich znanych algorytmów w przypadku pesymistycznym są $O(nmr)$ (jeśli pominięta zostanie zależność od długości słowa komputerowego), ale otwartą kwestią pozostaje, czy możliwe jest skonstruowanie algorytmów o złożoności czasowej $o(nmr)$. Z oczywistych powodów kres dolny tej złożoności nie może być lepszy niż kres dolny złożoności dla problemu LCS, bowiem jeśli jeden z ciągów A lub B jest pusty, to problem MerLCS redukuje się do problemu LCS.

Wnioski końcowe

Niniejsza praca stanowi podsumowanie wieloletnich badań autora, dotyczących wyznaczania podciągów rosnących i wspólnych. W rozdziałach 2–6 rozważany jest problem najdłuższego podciągu rosnącego (LIS) oraz jego warianty. Problem LIS jest klasycznym problemem kombinatorycznym, który znalazł ostatnio zastosowania m.in. w przetwarzaniu danych biologicznych. W szczególności pojawił się jako element składowy algorytmów uliniawiania całych genomów, wyszukiwania nowych genów, tworzenia map genowych itp. Złożoność czasowa najlepszych algorytmów jego rozwiązywania jest $O(n \log \ell)$ (w ogólnym przypadku) lub $O(n \log \log n)$ (dla permutacji liczb z zakresu $[1,n]$). Rozważane w pracy warianty problemu LIS zostały zaproponowane stosunkowo niedawno i stanowią różne uogólnienia problemu podstawowego. Dla problemów LIS o minimalnej/maksymalnej szerokości/wysokości/sumie zaproponowano w niniejszej pracy algorytmy oparte na autorskiej koncepcji rozszerzonego pokrycia ciągu (rozdz. 3). Ich złożoności czasowe są takie same jak złożoność czasowa najlepszych algorytmów dla problemu LIS.

W problemie LIS o zadanym pochyleniu (SLIS) wymaga się, aby elementy podciągu wynikowego rosły dostatecznie szybko. Dla tego problemu zaproponowano przekształcenie ciągu wejściowego w taki sposób, aby jego symbole były różnymi liczbami całkowitymi z dobrze określonego przedziału. Przekształcenie to nie wpływa na długość ciągu będącego rozwiązaniem, a dzięki niemu możliwe jest zastosowanie m.in. drzew van Emde Boasa lub struktury Anderssona–Thorupa do przechowywania tzw. elementów krytycznych. Daje to złożoność czasową proponowanego algorytmu $O(n \min(\sqrt{\log \ell / \log \log \ell}, \log \log n))$, a więc lepszą niż złożoność czasowa algorytmu znanego z literatury.

Kolejnym rozważanym problemem jest wyznaczanie cyklicznego podciągu LIS (LICS). W pracy zaproponowano oryginalną ideę łączenia pokryć zachłannych reprezentujących fragmenty ciągu wejściowego. Wykazano, że aby znaleźć podciąg LICS, wystarczy wyznaczyć podciąg LIS tylko w niektórych rotacjach ciągu wejściowego. Łączenie pokryć pozwala na szybsze uzyskanie pokrycia dla danej rotacji, niż gdyby pokrycie budować za każdym razem od początku. Ponieważ na podstawie pokrycia ciągu można łatwo wyznaczyć podciąg LIS w danej rotacji, więc zaproponowany algorytm LICS charakteryzuje się dobrą złożonością czasową. Dla najlepszego z trzech rozważanych wariantów, w którym pokrycie jest reprezentowane za pomocą listy uporządkowanych list drzew zrównoważonych, złożoność czasowa jest $O(n \log \log n + \min(n\ell, n + \ell^3) \log \lceil n/\ell^2 \rceil)$. Jest ona lepsza od złożoności algorytmów znanych z literatury, jeśli tylko długość ciągu wynikowego, ℓ, mieści się w pewnym określonym przedziale.

Ostatnim uogólnieniem problemu LIS rozważanym w pracy jest problem wyznaczania najdłuższego spośród podciągów rosnących w każdym oknie rozmiaru u (LISW). Algorytm autorski LISW dla tego problemu opiera się na łączeniu pokryć, podobnie jak w problemie LICS. Złożoność najlepszego z zaproponowanych wariantów tego algorytmu, w którym pokrycia reprezentowane są w pamięci jako listy uporządkowane list drzew zrównoważonych, jest $O(n \log\log n + \min(n\ell, n\lceil \ell^3/u \rceil) \log\lceil u/\ell^2 \rceil)$.

Druga część niniejszej pracy (rozdziały 7–10) dotyczy problemu wyznaczania najdłuższego wspólnego podciągu dwu ciągów (LCS) oraz kilku jego uogólnień. Dla problemu LCS przedstawione są najważniejsze istniejące algorytmy jego rozwiązywania. Omówione są także dwa autorskie algorytmy równoległe przeznaczone dla procesorów graficznych. Pierwszy z nich jest zrównolegloną wersją klasycznego algorytmu programowania dynamicznego, a drugi – algorytmu równoległości bitowej. Drugi z tych algorytmów opracowano w dwóch wariantach różniących się szybkością działania i zakresem stosowalności. Przyspieszenie tych algorytmów w stosunku do ich pierwowzorów wynosi od kilku do kilkudziesięciu.

Dla problemu LCS niezmienniczego względem transpozycji (LCTS) zaproponowano kilka algorytmów. Pierwszy z nich, LCTS-NGMD, jest algorytmem sekwencyjnym o złożoności czasowej $O(n\sigma + nm \log\log\sigma)$. Drugi algorytm przedstawiono w czterech wariantach, różniących się zastosowanymi strukturami danych i złożonościami czasowymi. Najlepsza z uzyskanych złożoności czasowych (dla algorytmu LCTS-HS-2) jest $O(D^* \log\log\sigma + nm)$, gdzie D^* jest liczbą dopasowań dominujących pomiędzy ciągami wejściowymi dla wszystkich możliwych transpozycji. Jest to najlepsza ze złożoności czasowych osiągniętych dla tego problemu wśród algorytmów sekwencyjnych. Łącząc ten algorytm z algorytmem równoległości bitowej, zaproponowano algorytm hybrydowy, którego zmierzona szybkość działania jest istotnie lepsza od szybkości innych algorytmów znanych z literatury. Ostatnią propozycją dla tego problemu jest algorytm równoległy dla procesorów GPU (LCTS-CUDA). W praktyce, okazał się on przynajmniej kilkakrotnie szybszy od wyżej wymienionych algorytmów.

Problem ukierunkowanego LCS (CLCS) jest kolejnym wariantem problemu LCS rozważanym w niniejszej pracy. Problem CLCS był inspirowany badaniami w bioinformatyce. Zaproponowano dla niego kilka algorytmów. Pierwszy z nich to algorytm CLCS-HS-Length oparty na metodzie Hunta–Szymanskiego. Charakteryzuje się on złożonością czasową $O(r(m\ell + d) + n)$, która jest lepsza od złożoności czasowych algorytmów sekwencyjnych znanych z literatury. Badania eksperymentalne pokazały, że algorytm ten jest kilkakrotnie szybszy od rozwiązań konkurencyjnych. Kolejną propozycją jest algorytm równoległości bitowej CLCS-BP-Length. Jego złożoność czasowa w przypadku pesymistycznym jest $O(\sqrt{rnmD^{\mathrm{sm}}} + rn\lceil m/w \rceil)$. Złożoność w przypadku średnim jest lepsza o czynnik $(1/\min(\sigma, w))$ od złożoności znanych algorytmów wyznaczających podciąg CLCS metodą programowania dynamicznego. Praktyczne ekspery-

menty potwierdziły, że algorytm ten jest kilkunastokrotnie szybszy od algorytmów znanych z literatury oraz kilkakrotnie szybszy od algorytmu CLCS-HS-Length. Ostatnią propozycją jest równoległy algorytm CLCS-CUDA opracowany dla procesorów graficznych. Szybkość jego działania jest kilkadziesiąt razy lepsza od algorytmów literaturowych oraz kilkakrotnie lepsza od algorytmu równoległości bitowej.

Ostatnim wariantem problemu LCS rozważanym w niniejszej pracy jest problem scalonego LCS (MerLCS). Zaproponowano dla niego algorytm równoległości bitowej MerLCS-BP, którego złożoność czasowa jest lepsza o czynnik $(\log w/w)$ od złożoności czasowej znanego dla tego problemu algorytmu programowania dynamicznego, gdzie w jest długością słowa komputerowego. Eksperymenty pokazały, że proponowany algorytm jest zwykle ponadpięćdziesięciokrotnie szybszy od algorytmów znanych z literatury.

Do najważniejszych nierozwiązanych kwestii dotyczących omawianych wariantów problemów LIS oraz LCS należy znalezienie kresów dolnych złożoności czasowej w przypadku pesymistycznym. Można przypuszczać, że jest to zadanie trudne, ponieważ nawet dla klasycznego problemu LCS badanego w literaturze od kilkudziesięciu lat, kres dolny złożoności czasowej w modelu Word RAM (stosowanym jako podstawowy model obliczeniowy w niniejszej pracy) różni się znacznie od złożoności czasowej najlepszych znanych algorytmów.

Dla wszystkich algorytmów proponowanych w niniejszej pracy przeprowadzono analizę złożoności czasowej i pamięciowej w przypadku pesymistycznym, a dla niektórych także w przypadku średnim. Dzięki temu często można było wykazać, że proponowane algorytmy są najszybsze w sensie asymptotycznym.

Wśród potencjalnych kierunków dalszych badań na pewno można wymienić adaptację proponowanych algorytmów dla procesorów GPU do innych problemów pokrewnych problemowi LCS. Badania pokazały, że mimo konieczności stosunkowo częstej komunikacji pomiędzy procesorami CPU oraz GPU możliwe jest osiągnięcie dużych przyspieszeń dla tych problemów. Bardzo interesującą, acz trudną kwestią, jest opracowanie algorytmów równoległości bitowej dla innych problemów będących wariantami problemu LCS. Z uwagi na duże ograniczenia dotyczące rodzaju wykonywanych operacji, projektowanie algorytmów równoległości bitowej jest zadaniem stosunkowo trudnym, jednak uzyskiwane przyspieszenia w stosunku do klasycznych algorytmów sekwencyjnych zależne często liniowo od długości słowa komputerowego mogą być dużą zachętą do tych poszukiwań. Zaproponowana metoda łączenia pokryć zachłannych dla problemów będących wariantami problemu LIS powinna być możliwa do zastosowania także w innych wariantach tego problemu.

Bibliografia

1. G.M. Adel'son-Vel'skiĭ, E.M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

2. A.V. Aho, D.S. Hirschberg, J.D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23(1):1–12, 1976.

3. M.H. Albert, M.D. Atkinson, D. Nussbaum, J.-R. Sack, N. Santoro. On the longest increasing subsequence of a circular list. *Information Processing Letters*, 101:55–59, 2007.

4. M.H. Albert, A. Golynski, A.M. Hamel, A. López-Ortiz, S.S. Rao, M.A. Safari. Longest increasing subsequences in sliding windows. *Theoretical Computer Science*, 321:405–414, 2004.

5. D. Aldous, P. Diaconis. Longest increasing subsequences: from patience sorting to the Baik–Deift–Johansson theorem. *Bulletin of the AMS*, 36:413–432, 1999.

6. E. Alerstam, T. Svensson, S. Andersson-Engels. Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. *Journal of Biomedical Optics*, 13(6):060504, 2008.

7. E. Allender. Circuit complexity before the dawn of the new millennium. *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, strony 1–18, 1996.

8. L. Allison, T.L. Dix. A bit-string longest common subsequence algorithm. *Information Processing Letters*, 23:305–310, 1986.

9. S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

10. S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, D.J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, 1997.

11. A. Andersson, M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54(3):Article No. 13, 2007.

12. E. Anirre, K. Gojenola, K. Sarasola, A. Voutilainen. Towards a single proposal in spelling correction. *Proceedings of the 17th international conference on Computational linguistics*, strony 22–28, Montreal, Quebec, Canada, 1998.

13. H.-Y. Ann, C.-B. Yang, C.-T. Tseng, C.-Y. Hor. A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings. *Information Processing Letters*, 108(6):360–364, 2008.

14. H.-Y. Ann, C.-B. Yang, C.-T. Tseng, C.-Y. Hor. Fast algorithms for computing the constrained lcs of run-length encoded strings. *Proceedings of the 2009 International Conference on Bioinformatics and Computational Biology*, tom 2, strony 646–649, 2009.

15. A. Apostolico. Improving the worst-case performance of Hunt–Szymanski strategy for the longest common subsequences of two strings. *Information Processing Letters*, 23:63–69, 1986.

16. A. Apostolico. String editing and longest common subsequences. G. Rozenberg, A. Salomaa, redaktorzy, *Handbook of formal languages*, tom 2: linear modeling: background and application, rozdział 8, strony 361–398. Springer-Verlag, 1997.

17. A. Apostolico. General pattern matchings. M.J. Atallah, redaktor, *Handbook of Algorithms and Theory of Computation*, rozdział 13. CRC-Press, 1998.

18. A. Apostolico, M.J. Atallah, L.L. Larmore, S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal of Computing*, 19(5):968–998, 1990.

19. A. Apostolico, C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

20. A. Apostolico, G.M. Landau, S. Skiena. Matching for run-length encoded strings. *Journal of Complexity*, 15(1):4–16, 1999.

21. V.K. Arlazarov, E.A. Dinic, M.A. Kronrod, I.A. Faradzev. On economic construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11:1209–1210, 1975.

22. A.N. Arslan, O. Eğecioğlu. Algorithms for the constrained longest common subsequence problems. *International Journal of Foundations of Computer Science*, 16(6):1099–1109, 2005.

23. K.N. Babu, S. Saxena. Parallel algorithms for the longest common subsequence problem. *Proceedings of the 4th International Conference on High-Performance Computing*, strony 120–125, Washington, DC, USA, December 1997. IEEE Computer Society.

24. R.M. Baer, P. Brock. Natural sorting over permutation spaces. *Mathematics of Computation*, 22:385–410, 1968.

25. R.A. Baeza-Yates. *Efficient text searching*. Rozprawa doktorska, Department of Computer Science, University of Waterloo, Ontario, Canada, 1989.

26. R.A. Baeza-Yates, Gonnet G.H. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

27. J. Baik, P. Deift, K. Johansson. On the distribution of the length of the longest increasing subsequence in random permutations. *Journal of American Mathematical Society*, 12:1119–1178, 1999.

28. A. Banerjee, J. Ghosh. Clickstream clustering using weighted longest common subsequences. *Proceedings of the Web Mining Workshop at the 1st SIAM Conference on Data Mining*, strony 33–40, Chicago, 2001.

29. R. Bayer. Symmetric binary B-trees. *Acta Informatica*, 1:290–306, 1972.

30. T. C. Bell, J. G. Cleary, I. H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.

31. S. Bereg, M. Kubica, T. Waleń, B. Zhu. Rna multiple structural alignment with longest common subsequences. *Journal of Combinatorial Optimization*, 13(2):179–188, 2007.

32. L. Bergroth, H. Hakonen, T. Raita. A survey of longest common subsequence algorithms. *Proceedings of 7th International Symposium on String Processing Information Retrieval (SPIRE)*, strony 39–48, Curuña, Spain, 2000.

33. S. Bespamyatnikh, M. Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76(1–2):7–11, 2000.

34. Ch. Blum, M.J. Blesa, M.López-Ibáñez. Beam search for the longest common subsequence problem. *Computers and Operations Research*, 36(12):3178–3186, 2009.

35. P. Brass. *Advanced Data Structures*. Cambridge University Press, 2008.

36. G.S. Brodal, K. Kaligosi, I. Katriel, M. Kutz. Faster algorithms for computing longest common increasing subsequences. *Lecture Notes in Computer Science*, 4009:330–341, 2006.

37. H. Bunke, J. Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Information Processing Letters*, 54(2):93–96, 1995.

38. M. Burrows, D.J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, California, USA, 1994.

39. W.-T. Chan, Y. Zhang, S.P.Y. Fung, D. Ye, H. Zhu. Efficient algorithms for finding a longest common increasing subsequence. *Journal of Combinatorial Optimization*, 13(3):277–288, 2007.

40. E. Chen, L. Yang, H. Yuan. Longest increasing subsequences in windows based on canonical antichain partition. *Theoretical Computer Science*, 378(3):223–236, 2007.

41. X. Chen, M. Li, B. Ma, J. Tromp. DNACompress: fast and effective DNA sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002.

42. F.Y.L. Chin, A. De Santis, A.L. Ferrara, N.L. Ho, S.K. Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters*, 90:175–179, 2004.

43. F.Y.L. Chin, N.L. Ho, T.W. Lam, P.W.H. Wong. Efficient constrained multiple sequence alignment with performance guarantee. *Journal of Bioinformatics and Computational Biology*, 3(1):1–18, 2005.

44. R.A. Chowdhury, V. Ramachandran. Cache-oblivious dynamic programming. *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, strony 591–600, New York, NY, US, 2006. ACM Press.

45. M. Chrochemore, G.M. Landau, M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. *SIAM Journal of Computing*, 32(6):1654–1673, 2003.

46. M.G. Ciura, S. Deorowicz. How to squeeze a lexicon. *Software—Practice and Experience*, 31(11):1077–1090, 2001.

47. S.A. Cook, R.A. Reckhow. Time-bounded random access machines. *Journal Computer and System Sciences*, 7:354–375, 1973.

48. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. *Introduction to Algorithms*. McGraw-Hill, wydanie 2nd, 2003.

49. T. Crawford, C. Iliopoulos, R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:71–100, 1998.

50. M. Crochemore, Ch. Hancart, T. Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, USA, 2007.

51. M. Crochemore, C.S. Iliopoulos, Y.J. Pinzon. Speeding-up Hirschberg and Hunt–Szymanski LCS algorithms. *Fundamenta Informaticae*, strony 89–103, 2003.

52. M. Crochemore, C.S. Iliopoulos, Y.J. Pinzon, J.F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80:279–285, 2001.

53. M. Crochemore, G.M. Landau, M. Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, strony 679–688, San Francisco, California, 2002. Society for Industrial and Applied Mathematics.

54. M. Crochemore, W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

55. M. Crochemore, W. Rytter. *Jewels of Stringology*. World Scientific Publishing Company, 2003.

56. F. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

57. A.L. Delcher, S. Kasif, R.D. Fleischmann, J. Peterson, O. White, S.L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

58. J.S. Deogun, J. Yang, F. Ma. Emagen: An efficient approach to multiple whole genome alignment. *Proceedings of the Second Asia-Pacific Bioinformatics Conference*, strony 113–122, 2004.

59. R.C. Deonier, S. Tavaré, M.S. Waterman. *Computational Genome Analysis: An Introduction*. Springer, 2005.

60. S. Deorowicz. Improvements to Burrows–Wheeler compression algorithm. *Software—Practice and Experience*, 30(13):1465–1483, 2000.

61. S. Deorowicz. Second step algorithms in the Burrows–Wheeler compression algorithm. *Software—Practice and Experience*, 32(2):99–111, 2002.

62. S. Deorowicz. Context exhumation after the burrows–wheeler transform. *Information Processing Letters*, 95(1):313–320, 2005.

63. S. Deorowicz. Speeding up transposition-invariant string matching. *Information Processing Letters*, 100(1):14–20, 2006.

64. S. Deorowicz. Fast algorithm for constrained longest common subsequence problem. *Theoretical and Applied Informatics*, 19(2):91–102, 2007.

65. S. Deorowicz. An algorithm for solving the longest increasing circular subsequence problem. *Information Processing Letters*, 109(12):630–634, 2009.

66. S. Deorowicz. Computing the longest common transposition-invariant subsequence with gpu. K.A. Cyran, S. Kozielski, Peters J.F., U. Stańczyk, A. Wakulicz-Deja, redaktorzy, *Man-Machine Interactions, Series Advances in Intelligent and Soft Computing*, strony 551–559. Springer-Verlag, Berlin Heidelberg, 2009.

67. S. Deorowicz. On some variants of the longest increasing subsequence problem. *Theoretical and Applied Informatics*, 21(3–4):135–148, 2009.

68. S. Deorowicz. An algorithm for the longest increasing subsequence in a sliding window problem. 2010. W recenzji (`http://sun.aei.polsl.pl/~sdeor/pub/Deo2010d-draft.pdf`).

69. S. Deorowicz. Bit-parallel algorithm for the constrained longest common subsequence problem. *Fundamenta Informaticae*, 99(4):409–433, 2010.

70. S. Deorowicz. Bit-parallel algorithm for the merged longest common subsequence problem. 2010. (zgłoszony do publikacji).

71. S. Deorowicz. Longest common subsequence and related problem solved at graphical processing units. *Software—Practice and Experience*, 40(8):673–700, 2010.

72. S. Deorowicz, M.G. Ciura. Correcting spelling errors by modelling their causes. *International Journal of Applied Mathematics and Computer Science*, 15(2):275–285, 2005.

73. S. Deorowicz, Sz. Grabowski. A hybrid algorithm for the longest common transposition-invariant subsequence problem. *Computing and Informatics*, 28(5):729–744, 2009.

74. S. Deorowicz, Sz. Grabowski. On two variants of the longest increasing subsequence problem. K.A. Cyran, S. Kozielski, Peters J.F., U. Stańczyk, A. Wakulicz-Deja, redaktorzy, *Man-Machine Interactions, Series Advances in Intelligent and Soft Computing*, strony 541–549. Springer-Verlag, Berlin Heidelberg, 2009.

75. S. Deorowicz, J. Obstój. Constrained longest common subsequence computing algorithms in practice. *Computing and Informatics*, 29(3):427–445, 2010.

76. B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, 3:29–46, 1964.

77. R. Durbin, S.R. Eddy, A. Krogh, G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999.

78. T. Easton, A. Singireddy. A specialized branching and fathoming technique for the longest common subsequence problem. *International Journal of Operations Research*, 4(2):98–104, 2007.

79. T. Easton, A. Singireddy. A large neighborhood search heuristic for the longest common subsequence problem. *Journal of Heuristics*, 14(3):271–283, 2008.

80. E. Edimiston, R.A. Wagner. Parallelization of the dynamic programming algorithm for comparison of sequences. *International Conference on Parallel Processing*, strony 78–90, 1987.

81. D. Eppstein, Z. Galil, R. Giancarlo, G.F. Italiano. Sparse dynamic programming I: linear cost functions. *Journal of the ACM*, 39(3):519–545, 1992.

82. D. Eppstein, Z. Galil, R. Giancarlo, G.F. Italiano. Sparse dynamic programming II: convex and concave cost functions. *Journal of the ACM*, 39(3):546–567, 1992.

83. P. Erdös, G. Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

84. T. Faraut, S. de Givry, P. Chabrier, T. Derrien, F. Galibert, C. Hitte, T. Schiex. A comparative genome approach to marker ordering. *Bioinformatics*, 23(2):e50–e56, 2007.

85. R. Farber. CUDA, supercomputing for the masses: Parts 1–15. *Dr. Dobb's Journal*, 2008–2010. `http://www.ddj.com/architect/207200659`.

86. M. Farrar. Striped Smith–Waterman speeds database searches six times over other SIMD implementations. *Bioinformatics*, 23(2):156–161, 2007.

87. P. Fenwick. The Burrows–Wheeler transform for block sorting text compression: Principles and improvements. *The Computer Journal*, 39(9):731–740, 1996.

88. P. Ferragina, G. Manzini. Opportunistic data structures with applications. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, strony 390–398, 2000.

89. P. Ferragina, G. Manzini. An experimental study of a compressed index. *Proceedings 12th ACM-SIAM Symposium on Discrete Algorithms*, strony 269–278, 2001.

90. M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

91. M.L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.

92. K. Fredriksson, V. Mäkinen, G. Navarro. Flexible music retrieval in sublinear time. *International Journal of Foundations of Computer Science*, 17(6):1345–1364, 2006.

93. A. Frezzotti, G.P. Ghiroldi, L. Gibelli. Solving kinetic equations on GPUs I: Model kinetic equations. arXiv:0903.4044v1 [physics.comp-ph], 2009. `http://arxiv.org/abs/0903.4044`.

94. W. Fulton. *Young Tableaux. With Applications to Representation Theory and Geometry*, tom 35 serii *Longon mathematical Society Student Texts*. Cambridge University Press, 1997.

95. W. Fulton, J. Harris. *Representation Theory. A First Course*, tom 129 serii *Graduate Texts in Mathematics. Readings in Mathematics*. Springer-Verlag, 1991.

96. P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.

97. A. Ghias, J. Logan, D. Chamberlin, B.C. Smith. Query by humming—musical information retrieval in an audio database. *Proceedings of the third ACM international conference on Multimedia*, strony 231–236, San Francisco, CA, 1995.

98. L. Golab, H.J. Karloff, F. Korn, A. Saha, D. Srivastava. Sequential dependencies. *Proceedings of the Very Large Database Conference*, tom 2, strony 574–585, 2009.

99. S. Golomb. Runlength encodings. *IEEE Transactions on Information Theory*, IT-12(3):399–401, July 1966.

100. M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*, tom 57 serii *Annals of Discrete Mathematics*. Academic Press, wydanie 2nd, 2004.

101. O. Gotoh, S. Yamada, T. Yada. Multiple sequence alignment. S. Aluru, redaktor, *Handbook of Computational Molecular Biology*, rozdział 3. Chapman & Hall/CRC, 2006.

102. Sz. Grabowski. *New Algorithms for Exact and Approximate Text Matching*, tom 1051 serii *Zeszyty Naukowe Politechniki Łódzkiej*. Wydawnictwo Politechniki Łódzkiej, Łódź, 2009.

103. Sz. Grabowski, S. Deorowicz. Nice to be a chimera: A hybrid algorithm for the longest common transposition-invariant subsequence problem. *Proceedings of 9th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET'2008)*, strony 50–54, 2008.

104. Sz. Grabowski, G. Navarro. $O(mn\log\sigma)$ time transposition invariant LCS computation. Technical Report TR/DCC-2004-6, University of Chile, Department of Computer Science, September 2004. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/transpszymon.ps.gz`.

105. L.J. Guibas, R. Sedgewick. A dichromatic framework for balanced trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, strony 8–21. IEEE Computer Society, 1978.

106. D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

107. J.M. Hammersley. A few seedlings of research. *Proceedings of the Sixth Berkley Symposium on Mathematical Statistics and Probability*, tom 1, strony 345–394. University of California Press, 1972.

108. R.W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.

109. W. Haque, A. Aravind, B. Reddy. Pairwise sequence alignment algorithms: a survey. *Proceedings of the 2009 conference on Information Science, Technology and Applications*, strony 96–103, Kuwait, Kuwait, 2009.

110. D. He, A.N. Arslan. A space-efficient algorithm for the constrained pairwise sequence alignment problem. *Genome Informatics*, 16(2):237–246, 2005.

111. D. Hermelin, G.M. Landau, S. Landau, O. Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, strony 529–540, 2009.

112. D.S. Hirschberg. A linear space algorithm for computing maximal common subsequence. *Communications of the ACM*, 18(6):341–343, 1975.

113. D.S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24:664–675, 1977.

114. D.S. Hirschberg. An information-theoretic lower bound for the longest common subsequence problem. *Information Processing Letters*, 7(1):40–41, 1978.

115. K.-S. Huang. *Some Common Subsequence Problems of Multiple Sequences and Their Applications*. Rozprawa doktorska, National Sun Yat-sen University, Taiwan, 2006.

116. K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, Y.-H. Peng. Efficient algorithm for finding interleaving relationship between sequences. *Information Processing Letters*, 105(5):188–193, 2008.

117. K.-S. Huang, C.-B. Yang, K.-T. Tseng, Y.-H. Peng, H.-Y. Ann. Dynamic programming algorithms for the mosaic longest common subsequence problem. *Information Processing Letters*, 102(2–3):99–103, 2007.

118. J.W. Hunt, M.D. McIlroy. An algorithm for differential file comparison. Computing Science Technical Report 41, Bell Laboratories, 1976. `http://www.cs.dartmouth.edu/~doug/diff.ps`.

119. J.W. Hunt, T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

120. H. Hyyrö. Bit-parallel LCS-length computation revisited. *Proceedings of the 15th Australian Workshop on Combinatorial Algorithms (AWOCA)*, strony 16–27, Australia, 2004. University of Sydney.

121. C.S. Iliopoulos, Y.J. Pinzon. Recovering an LCS in $O(n^2/w)$ time and space. *Proceedings 12th Australasian Workshop on Combinatorial Algorithms*, strony 106–117, 2001.

122. C.S. Iliopoulos, M.S. Rahman. New efficient algorithms for LCS and constrained LCS problem. *Information Processing Letters*, 106(1):13–18, 2008.

123. G. Jacobson, K.-P. Vo. Heaviest increasing/common subsequence problems. *Lecture Notes in Computer Science*, 644:52–66, 1992.

124. N.C. Jones, P.A. Pevzner. *An Introduction to Bioinformatics Algorithms*. Massachusetts Institute of Technology, 2004.

125. M. Kellis, B.W. Birren, E.S. Lander. Proof and evolutionary analysis of ancient genome duplication in the yeast saccharomyces cerevisiae. *Nature*, 428(6983):617–624, 2004.

126. Khronos OpenCL Working Group. The OpenCL specification. `http://www.khronos.org/registry/cl/specs/opencl-1.0.43.pdf`, May, 16 2009. Version 1.0, Document Revision 43.

127. D.B. Kirk, W. m.W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.

128. D.G. Kirkpatrick, S. Reisch. Upper bounds for sorting integers on random access machines. *Theoretical Computer Science*, 28:263–276, 1984.

129. J. Kloetzli, B. Strege, J. Decker, M. Olano. Parallel longest common subsequence using graphics hardware. J. Favre, K.L. Ma, D. Weiskopf, redaktorzy, *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, strony 14–15, 2008. `http://www.cs.umbc.edu/~olano/papers/cudaLCS.pdf`.

130. D.E. Knuth. *The Art of Computer Programming*, tom 3: Sorting and Searching. Addison Wesley Longman, wydanie 2nd, 1998.

131. D.E. Knuth. *The Art of Computer Programming*, tom 4, zeszyt 1: Bitwise Tricks & Techniques; Binary Decision. Addison Wesley Professional, 2009.

132. I. Koren. *Computer Arithmetic Algorithms*. A K Peters, Ltd, 2002.

133. P. Krusche, A. Tiskin. Efficient longest common subsequence computation using bulk-synchronous parallelism. *Lecture Notes in Computer Science*, 3984:165–174, 2006.

134. K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

135. S. Kuo, G.R. Cross. An improved algorithm to find the length of the longest common subsequence of two strings. *ACM SIGIR Forum*, 23(3-4):89–99, 1989.

136. S. Kurtz. Reducing the space requirement of suffix trees. *Software—Practice and Experience*, 29:1149–1171, 1999.

137. S. Kurtz, A. Phillippy, A.L. Delcher, M. Smoot, M. Shumway, C. Antonescu, S.L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12.1–R12.9, 2004.

138. T.W. Lam, W.K. Sung, S.L. Tam, C.K. Wong, S.M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2009.

139. K. Lemström, G. Navarro, Y. Pinzon. Practical algorithms for transposition-invariant string-matching. *Journal of Discrete Algorithms*, 3(2–4):267–292, 2005.

140. K. Lemström, J. Tarhio. Searching monophonic patterns within polyphonic sources. *Proceedings of Content-Based Multimedia Information Access Conference (RIAO)*, strony 1261–1279, 2000.

141. K. Lemström, E. Ukkonen. Including interval encoding into edit distance based music comparison and retrieval. *Proceedings of AISB'2000 Symposium on Creative & Cultural Aspects and Applications of AI & Cognitive Science*, strony 53–60, 2000.

142. V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Problems in Information Transmission*, 1:8–17, 1965.

143. Ł. Ligowski, W. Rudnicki. An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, strony 1–8, 2009.

144. F.-M. Lin, H.-D. Huang, Y.-C. Chang, A.-P. Tsou, P.-L. Chan, L.-C. Wu, M.-F. Tsai, J.-T. Horng. Database to dynamically aid probe design for virus identification. *IEEE Transactions on Information Technology in Biomedicine*, 10(4):705–713, 2006.

145. R.J. Lipton, D. Lopresti. A systolic array for rapid string comparison. *Proceedings of the Chapel Hill Conference on VLSI*, strony 363–376, 1985.

146. J.J. Liu, Y.L. Wang, R.C.T. Lee. Finsign a longest common subsequence between a run-length-encoded string and an uncompressed string. *Journal of Complexity*, 24(2):173–187, 2008.

147. W. Liu, B. Schmidt, G. Voss, A. Schroder, W. Muller-Wittig. Bio-sequence database scanning on a GPU. *Proceedings of the 20nd International Parallel and Distributed Processing Symposium*, strony 274–281, 2006.

148. C.L. Lu, Y.P. Huang. A memory-efficient algorithm for multiple sequence alignment with constraints. *Bioinformatics*, 21(1):20–30, 2005.

149. D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

150. V. Mäkinen, G. Navarro, E. Ukkonen. Transposition invariant string matching. *Journal of Algorithms*, 56(2):124–153, 2005.

151. S.A. Manavski, G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith–Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2):S10, 2008. `http://www.biomedcentral.com/1471-2105/9/S2/S10`.

152. W.J. Masek, M.S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer System Science*, 20(1):18–31, 1980.

153. E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

154. R.J. McNab, L.A. Smith, D. Bainbridge, I.H. Witten. The New Zealand digital library MELody inDEX. *D-Lib Magazine*, May 1997.

155. K. Mehlhorn. *Data structures and algorithms 1: sorting and searching*. Springer–Verlag, 1984.

156. K. Mehlhorn, P. Sanders. *Algorithms and Data Structures. The Basic Toolbox*. Springer-Verlag, Berlin Heidelberg, 2008.

157. R. Mitton. Spellchecking by computer. *Journal of Simplified Spelling Society*, 20(1):4–11, 1996.

158. R. Mitton. Ordering the suggestions of a spellchecker without using context. *Natural Language Engineering*, 15(2):173–192, 2009.

159. D.W. Mount. *Bioinformatics: Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, wydanie 2nd, 2004.

160. E.W. Myers, W. Miller. Optimal alignments in linear space. *Computer Applications in the Biosciences*, 4(1):11–17, 1988.

161. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

162. G. Navarro. Dynamic dictionaries in constant worst-case time. Technical Report TR/DCC-2007-11, University of Chile, Department of Computer Science, October 2007. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/consthash.ps.gz`.

163. G. Navarro, Sz. Grabowski, V. Mäkinen, S. Deorowicz. Improved time and space complexities for transposition invariant string matching. Technical Report TR/DCC-2005-4, University of Chile, Department of Computer Science, 2005. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/mnloglogs.ps.gz`.

164. S.B. Needleman, C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

165. F. Nicolas, E. Rivals. Longest common subsequence problem for unoriented and cyclic strings. *Theoretical Computer Science*, 370(1-18), 2007.

166. NVidia Corporation. CUDA zone—the resource for CUDA developers. `http://www.nvidia.com/object/cuda_home.html`, February, 2 2010.

167. NVidia Corporation. NVidia CUDA™programming guide, version 3.0. `http://www.nvidia.com/object/cuda_get.html`, February, 2 2010.

168. N. Orio. Music retrieval: a tutorial and review. *Foundations and Trends in Information Retrieval*, 1(1):1–96, 2006.

169. C.H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.

170. E. Parvinnia, M. Taheri, K. Ziarati. An improved longest common subsequence algorithm for reducing memory complexity in global alignment of DNA sequences. *Proceedings of the International Conference on BioMedical Engineering and Informatics*, tom 1, strony 57–61. IEEE Computer Society, 2008.

171. S. Pemmaraju, S. Skiena. *Computational Discrete Mathematics. Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, 2003.

172. Ch.-L. Peng. An approach for solving the constrained longest common subsequence problem. Praca magisterska, Department of Computer Science and Engineering, National Sun Yat-sen University, Taiwan, 2003. `http://etd.lib.nsysu.edu.tw/ETD-db/ETD-search/getfile?URN=etd-0828103-125439&filename=etd-0828103-125439.pdf`.

173. Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, C.-Y. Hor. Efficient sparse dynamic programming for the merged LCS problem with block constraints. *International Journal of Innovative Computing, Information and Control*, 6(4):1935–1947, 2010.

174. Z.S. Peng, H.F. Ting. Time and space efficient algorithms for constrained sequence alignment. *Proceedings of the Implementation and Application of Automata, 9th International Conference (CIAA)*, strony 237–246, 2004.

175. P. Pevzner, M. Waterman. Matrix longest common subsequence problem. *Lecture Notes in Computer Science*, 644:79–89, 1992.

176. A. Polański, M. Kimmel. *Bioinformatics*. Springer, 2007.

177. M. Pătraşcu, M. Thorup. Predecessor search. Ming-Yang Kao, redaktor, *Encyclopedia of Algorithms*, strony 661–664. SpringerScience+BuisinessMedia, LLC., 2008.

178. S. Rajko, S. Aluru. Space and time optimal parallel sequence alignments. *IEEE Transactions on Parallel and Distributed Systems*, 15(12):1070–1081, 2004.

179. S. Ranka, S. Sahni. String editing on an SIMD hypercube multicomputer. *Journal of Parallel and Distributed Computing*, 9:411–418, 1990.

180. C. Rick. Efficient computation of all longest common subsequences. *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT'00)*, strony 407–418, 2000.

181. J. Rothstein. *Midi: A Comprehensive Introduction*. A-R Editions, wydanie 2, 1995.

182. W. Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

183. N. Saitou, M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 1987.

184. Y. Sakai. A linear space algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 99:203–207, 2006.

185. D. Salomon. *Data Compression: The Complete Reference*. Springer, wydanie 4th, 2006.

186. D. Sankoff, J. Kruskal, redaktorzy. *Time Warps, String Edits, and Macromolecules. The Theory and Practice of Sequence Comparison*. Addison–Wesley, 1983.

187. W.J. Savitch, M.J. Stimson. Time bounded random access machines with parallel processing. *Journal of the ACM*, 26(1):103–118, 1979.

188. C. Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

189. H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, T. Chiueh. Graphic-card cluster for astrophysics (GraCCA) — performance tests. arXiv:0707.2991v2 [astro-ph], 2008. `http://arxiv.org/abs/0707.2991`.

190. R. Sedgewick. Left-leaning red-black trees. Talk at Combinatorics and Probability seminar at University of Pennsylvania, October 2008. `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf`.

191. A. Shiraki, N. Takada, M. Niwa, Y. Ichihashi, T. Shimobaba, N. Masuda, T. Ito. Simplified electro-holographic color reconstruction system using graphics processing unit and liquid crystal display projector. *Optics Express*, 17(8):16038–16045, 2009.

192. S.J. Shyu, C.-Y. Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Computers and Operations Research*, 36(1):73–91, 2009.

193. S. Skiena. Longest increasing subsequences. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, rozdział 2.3.6, strony 73–75. Reading, MA: Addison-Wesley, 1990.

194. D.S. Sleator, R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

195. T.F. Smith, M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

196. J. Storer, T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29:928–951, 1982.

197. C.Y. Tank, C.L. Lu, M.D.-T. Chang, Y.-T. Tsai, Y.-J. Sun, K.-M. Chao, J.-M. Chang, Y.-H. Chiou, C.-M. Wu, H.-T. Chang, W.-I. Chou. Constrained multiple sequence alignment tool development and its application to RNase family alignment. *Proceedings of the first IEEE Computer Society Bioinformatics Conference (CSB 2002)*, strony 127–137, 2002.

198. R.E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

199. A. Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008.

200. A. Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *arXiv*, strony 1–115, 2010. arXiv:0707.3619v15.

201. TOP500.Org. Top 500 supercomputer sites. `http://www.top500.org/`, June 2010.

202. N. Trevett. OpenCL. The open standard for heterogeneous parallel programming. `http://www.khronos.org/developers/library/overview/opencl_overview.pdf`, June, 18 2009.

203. Y.-T. Tsai. The constrained common subsequence problem. *Information Processing Letters*, 88:173–176, 2003.

204. C.-T. Tseng, C.-B. Yang, H.-Y. Ann. Minimal height and sequence constrained longest increasing subsequence. *Journal of Internet Technology*, 10(2):173–178, 2009.

205. A. Uitdenbogerd, J. Zobel. Melodic matching techniques for large music databases. *Proceedings of the seventh ACM international conference on Multimedia (Part 1)*, strony 57–66, Orlando, USA, 1999.

206. E. Ukkonen. On-line construction of suffix tress. *Algorithmica*, 14(3):249–260, 1995.

207. L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

208. P. van Emde Boas. Machine models and simulations. J. van Leeuwen, redaktor, *Handbook of Theoretical Computer Science. Algorithms and Complexity*, rozdział 1, strony 1–66. Elsevier Science Publishers B.V., 1990.

209. P. van Emde Boas, R. Kaas, E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

210. P. van Emde Boas, R. Kaas, E. Zijlstra. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

211. A.M. Vershik, Kerov S.V. Asymptotics of the plancherel measure of the symmetric group and the limiting form of Young tables. *Soviet Mathematics Doklady*, 18:527–531, 1977.

212. R.A. Wagner, M.J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

213. Q. Wang, D. Korkin, Y. Shang. Efficient dominant point algorithms for the multiple longest common subsequence (MLCS) problem. H. Kitano, redaktor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, strony 1494–1499, Pasadena, California, USA, 2009.

214. W.-L. Wang. Longest common subsequence with constraints. Praca magisterska, Department of Computer Science and Information Engineering National Chi-NanUniversity, R. O. C., June 2006. `http://alg.csie.ncnu.edu.tw/or/WLWang2006.pdf`.

215. H.S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2002.

216. O. Weimann. *Accelerating Dynamic Programming*. Rozprawa doktorska, Massachusetts Institute of Technology, June 2009. `http://erikdemaine.org/theses/oweimann.pdf`.

217. P. Weiner. Linear pattern matching algorithms. *Proceedings of the* 14*th IEEE Annual Symposium on Switching and Automata Theory, The University of Iowa*, strony 1–11, 1973.

218. T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

219. C.K. Wong, A.K. Chandra. Bounds for the string editing problem. *Journal of the ACM*, 23(1):13–16, 1976.

220. I.-H. Yang, Y.-C. Chen. Fast algorithms for the constrained longest increasing subsequence problems. *Proceedings of the 25th Workshop on Combinatorial Mathematics and Computation Theory*, strony 226–231, Chung Hua University, Hsinchu Hsien, Taiwan, April 25–26 2008.

221. I-H. Yang, C.-P. Huang, K.-M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93:249–253, 2005.

222. A. Young. On quantitative substitutional analysis. *Proceedings of the London Mathematical Society*, s2-28(1):255–292, 1928.

223. H. Zhang. Alignment of BLAST high-scoring segment pairs based on the longest increasing subsequence algorithm. *Bioinformatics*, 19(11):1391–1396, 2003.

224. J. Ziv, A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23:337–343, 1977.

225. J. Ziv, A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24:530–536, 1978.

Spis rysunków

2.1. Algorytm wstawiania liczby do tableau Younga 37
2.2. Ilustracja działania algorytmu tworzenia tableau Younga 38
2.3. Algorytm tworzenia pokrycia zachłannego ciągu 39
2.4. Przykład działania algorytmu Cover-Make wyznaczającego pokrycie zachłanne ciągu 39
2.5. Algorytm odczytywania podciągu LIS z pokrycia utworzonego algorytmem Cover-Make 40

3.1. Ogólny schemat algorytmów rozwiązujących problemy MinHLIS, MaxHLIS, MinWLIS, MaxWLIS, MinSLIS, MaxSLIS 44
3.2. Algorytm rozwiązujący problem MinHLIS 45
3.3. Pokrycie zachłanne rozszerzone wygenerowane przez algorytm MinHLIS 46
3.4. Algorytm rozwiązujący problem MaxHLIS 48
3.5. Algorytm rozwiązujący problem MinWLIS 49
3.6. Algorytm rozwiązujący problem MaxWLIS 51
3.7. Algorytm rozwiązujący problem MinSLIS 53
3.8. Algorytm rozwiązujący problem MaxSLIS 54

4.1. Przykład wpływu pochylenia na podciąg SLIS 59
4.2. Przykład wyznaczania projekcji elementów w problemie SLIS 61
4.3. Algorytm rozwiązujący problem SLIS 64

5.1. Przykład działania algorytmu Cover-Make wyznaczającego zachłanne pokrycie ciągu 68
5.2. Ogólny schemat algorytmów łączących pokrycia 70
5.3. Ogólny schemat algorytmów wyznaczania długości podciągu LICS 71
5.4. Przykład działania algorytmu wyznaczania podciągu LICS opartego na reprezentacji listowej pokrycia dla ciągu 74

6.1. Ogólny schemat algorytmów wyznaczania podciągu LISW w przedziale szerokości podwójnego rozmiaru okna 81
6.2. Ogólny schemat algorytmów wyznaczania podciągu LISW 82

7.1. Przykład różnych miar odległości podobieństwa ciągów 90

7.2. Przykład działania klasycznego algorytmu wyznaczania podciągu LCS opartego na programowaniu dynamicznym . . . 93
7.3. Algorytm Hunta–Szymanskiego rozwiązujący problem LCS . . . 94
7.4. Przykład działania algorytmu Hunta–Szymanskiego wyznaczania podciągu LCS . . 95
7.5. Przykład działania algorytmu Hirschberga o liniowej złożoności pamięciowej wyznaczającego podciąg LCS . . . 98
7.6. Bitowo-równoległy algorytm Hyyrö wyznaczający długość podciągu LCS [120] . . 99
7.7. Przykład działania bitowo-równoległego algorytmu Hyyrö dla problemu LCS . . . 100
7.8. Algorytm równoległości bitowej wyznaczania podciągu LCS z macierzy wygenerowanej algorytmem LCS-BP-Length . . . 102
7.9. Przykład działania algorytmu równoległości bitowej wyznaczania podciągu LCS . 102
7.10. Przykład działania algorytmu wyznaczania podciągu LCS opartego na metodzie równoległości bitowej z wykorzystaniem metody Hirschberga o liniowej złożoności pamięciowej . . . 104
7.11. Ilustracja działania algorytmu wyznaczania podciągu LCS opartego na algorytmie wyznaczania podciągu LIS . . . 105
7.12. Podstawowa idea dekompozycji macierzy programowania dynamicznego dla problemu LCS na pudełka . . . 110
7.13. Ogólny schemat algorytmów rozwiązujących problem LCS i jego warianty w modelu programowania CUDA . . . 113
7.14. Kolejność obliczeń w pudełku w algorytmie rozwiązywania problemu LCS dla procesora GPU . . . 114
7.15. Część kodu jądra algorytmu LCS-DP-CUDA rozwiązującego problem LCS metodą DP w procesorze GPU . . . 116
7.16. Część programu jądra dla algorytmu LCS-BP-CUDA rozwiązującego problem LCS metodą równoległości bitowej w procesorze GPU . . . 120
7.17. Porównanie przyspieszenia równoległego algorytmu wyznaczania podciągu LCS dla procesorów GPU opartego na programowaniu dynamicznym . . . 124
7.18. Porównanie przyspieszenia algorytmu równoległego wyznaczania długości podciągu LCS dla procesorów GPU opartego na równoległości bitowej . . . 125

8.1. Algorytm wyznaczający jedno pudełko macierzy programowania dynamicznego dla algorytmu z rys. 8.2 . . . 130
8.2. Algorytm wyznaczający długość podciągu LCTS metodą kolejnych pudełek 131
8.3. Przykład działania algorytmu wyznaczania podciągu LCTS . . . 134

8.4. Algorytm wyznaczający długość podciągu LCTS o złożoności czasowej $O(D^* \log\log\sigma + nm)$ . . . 137
8.5. Przykładowe drzewo w-arne . . . 138
8.6. Relatywna prędkość działania algorytmów rozwiązujących problem LCTS . . . 140
8.7. Skumulowana procentowa liczba dopasowań (począwszy od transpozycji najrzadszych) w zależności od rangi transpozycji (problem LCTS) . . . 144
8.8. Czasy działania algorytmu LCTS-Hybrid dla problemu LCTS dla zmieniającego się progu podziału maksymalnej liczby dopasowań w transpozycji przetwarzanej przez składową LCTS-HS-3. . . . 145
8.9. Czas działania algorytmu LCTS-Hybrid dla problemu LCTS dla zmiennego progu podziału maksymalnej liczby dopasowań w transpozycji przetwarzanej przez składową LCTS-HS-3 . . . 145
8.10. Porównanie przyspieszenia algorytmu równoległości bitowej dla procesora GPU . . 150

9.1. Przykład działania algorytmu China i in. . . . 154
9.2. Przykład działania algorytmu China i in. z okrojonymi poziomami macierzy . . . 155
9.3. Przykład działania algorytmu wyznaczającego macierz M dla problemu CLCS . . . 158
9.4. Algorytm wyznaczania długości podciągu CLCS oparty na metodzie Hunta–Szymanskiego . . . 160
9.5. Algorytm wyznaczania podciągu CLCS oparty na metodzie Hunta–Szymanskiego . 161
9.6. Porównanie czasów działania algorytmów wyznaczających podciąg CLCS dla różnych rozmiarów alfabetu . . . 165
9.7. Porównanie czasów wyznaczania długości podciągów CLCS dla różnych zestawów parametrów . . . 168
9.8. Idea działania algorytmu paskowego równoległości bitowej dla problemu CLCS. . 170
9.9. Algorytm znajdujący punkt startowy dla poziomu k (problem CLCS, algorytm równoległości bitowej) . . . 170
9.10. Algorytm równoległości bitowej wyliczający pojedynczy pasek (problem CLCS) . 171
9.11. Algorytm równoległości bitowej wyznaczający pojedynczy poziom (problem CLCS) 173
9.12. Algorytm równoległości bitowej wyznaczający poziom 0 macierzy M dla problemu CLCS . . . 174
9.13. Algorytm równoległości bitowej wyznaczający długość podciągu CLCS . . . 175
9.14. Przykład działania algorytmu równoległości bitowej wyznaczającego długość podciągu CLCS dla $k = 1$. . . . 176
9.15. Rozszerzenie algorytmu CLCS-BP-Strip pozwalające na wyznaczanie podciągu CLCS . . . 177

9.16. Rozszerzenie algorytmu CLCS-BP-Level umożliwiające wyznaczanie podciągu CLCS 177
9.17. Algorytm wyznaczania podciągu CLCS 178
9.18. Ilustracja działania algorytmu wyznaczającego podciąg CLCS 179
9.19. Porównanie przyspieszenia uzyskiwanego przez algorytmy równoległości bitowej dla problemu CLCS w stosunku do ulepszonego algorytmu China i in. 183
9.20. Porównanie przyspieszeń uzyskiwanych przez równoległy algorytm programowania dynamicznego dla procesorów GPU (problemu CLCS) 190

10.1. Przykład obliczeń dla problemu MerLCS 194
10.2. Algorytm równoległości bitowej wyznaczający długość podciągu MerLCS 200
10.3. Algorytm symulujący usuwanie bitów 1 o parzystych rangach (składowa algorytmu MerLCS-BitPar) 201
10.4. Eksperymentalne porównanie algorytmów wyznaczających długość podciągu MerLCS dla różnych rozmiarów alfabetów 204
10.5. Eksperymentalne porównanie czasów działania algorytmów wyznaczających długość podciągu MerLCS dla ciągów różnej długości 204

List of Figures

2.1. Algorithm inserting an integer to Young tableau 37
2.2. Example of the algorithm building Young tableau 38
2.3. Greedy cover making algorithm of the sequence 39
2.4. Example of the algorithm Cover-Make finding greedy cover of a sequence 39
2.5. An algorithm reading LIS from the cover produced by Cover-Make algorithm . . 40
3.1. A general scheme of the algorithms solving MinHLIS, MaxHLIS, MinWLIS, MaxWLIS, MinSLIS, MaxSLIS problems 44
3.2. An algorithm solving MinHLIS problem 45
3.3. Annotated greedy cover produced by MinHLIS algorithm 46
3.4. An algorithm solving MaxHLIS problem 48
3.5. An algorithm solving MinWLIS problem 49
3.6. An algorithm solving MaxWLIS problem 51
3.7. An algorithm solving MinSLIS problem 53
3.8. An algorithm solving MaxSLIS problem 54
4.1. Example of the influence of the slope on an SLIS 59
4.2. Example of the computation of projections of symbols for the SLIS problem 61
4.3. Algorithm solving SLIS problem 64
5.1. Example of the algorithm Cover-Make finding greedy cover of a sequence 68
5.2. A general scheme of the cover merging algorithm 70
5.3. A general scheme of the algorithm computing LICS length 71
5.4. Example of the list-based algorithm in work for a sequence 74
6.1. A general scheme of the algorithm computing LISW in a range of width twice as large as window size 81
6.2. A general scheme of the algorithm computing LISW 82
7.1. Example of various distance measures for sequence similarity 90
7.2. An example of the classical dynamic programming algorithm computing an LCS . 93
7.3. Hunt-Szymanski algorithm for the LCS problem 94
7.4. An example of the Hunt–Szymanski algorithm computing the LCS 95
7.5. An example of the Hirschberg linear-space algorithm computing the LCS 98
7.6. Bit-parallel algorithm computing the LCS length by Hyyrö [120] 99
7.7. An example of the bit-parallel algorithm by Hyyrö for the LCS problem 100
7.8. Bit-parallel algorithm recovering an LCS from the matrix produced by LCS-BP-Length algorithm 102

7.9. An example of the bit-parallel algorithm by Hyyrö for the LCS problem 102
7.10. An example of the bit-parallel LCS computing algorithm with Hirschberg linear-space method computing the LCS 104
7.11. Example of the LIS-based algorithm for the LCS problem 105
7.12. A general scheme of decomposing the computation of the DP matrix for the LCS problem into boxes 110
7.13. A general scheme of the algorithms solving the LCS, and related problems in the CUDA programming model 113
7.14. The order of computing a box in GPU algorithm for the LCS problem 114
7.15. A part of the kernel code for the LCS DP algorithm (LCS-DP-CUDA) at GPU . . . 116
7.16. A part of the kernel code for the LCS BP algorithm (LCS-BP-CUDA) at GPU . . . 120
7.17. Comparison of the speedup of the GPU parallel algorithm based on the classical LCS dynamic programming algorithm 124
7.18. Comparison of the speedup of the LCS-length-computing GPU parallel algorithm based on the bit-parallel algorithm 125
8.1. An algorithm computing one box for the algorithm given at Fig. 8.2 130
8.2. Box-based algorithm solving the LCTS problem 131
8.3. Example of the LCTS computing algorithm in work 134
8.4. An $O(D^* \log\log\sigma + nm)$-time algorithm computing the LCTS length 137
8.5. Sample tree of arity 4 138
8.6. Relative speed of algorithms solving the LCTS problem 140
8.7. Cumulative % matches vs. % transpositions counted from the most rare transpositions (LCTS problem) 144
8.8. Processing time of LCTS-Hybrid algorithm with varying threshold of the maximal number of matches in transpositions handled by the LCTS-HS-3 component. 145
8.9. Processing time of LCTS-Hybrid algorithm with varying threshold of maximal number of matches in transpositions handled by the LCTS-HS-3 component 145
8.10. Comparison of the speedup for the bit-parallel LCTS algorithm for GPU 150
9.1. Example of the algorithm by Chin *et al.* 154
9.2. Example of the algorithm by Chin *et al.* with trimmed levels 155
9.3. Example of the algorithm calculating M matrix for the CLCS problem 158
9.4. Algorithm computing the length of the CLCS based on Hunt–Szymanski method . 160
9.5. Algorithm computing the CLCS based on Hunt–Szymanski method 161
9.6. Comparison of the CLCS computing time for changing alphabet size 165
9.7. Comparison of the CLCS length computing time for various parameters 168
9.8. An idea of the strip-based bit-parallel algorithm for the CLCS problem. 170

9.9. Finding starting point for the level k (CLCS problem, bit-parallel algorithm) 170
9.10. Bit-parallel strip computing algorithm (CLCS problem) 171
9.11. Bit-parallel level-computing algorithm for the CLCS problem 173
9.12. Bit-parallel level-0 computing algorithm for the CLCS problem 174
9.13. Bit-parallel CLCS length computing algorithm 175
9.14. Example of the bit-parallel method for computation of the CLCS length for $k = 1$. 176
9.15. Extension to the CLCS-BP-Strip algorithm for computing the CLCS 177
9.16. Extension to the CLCS-BP-Level algorithm for computing the CLCS 177
9.17. Traceback procedure to obtain an CLCS 178
9.18. Example of the CLCS computing procedure 179
9.19. Comparison of the speedup of bit-parallel algorithms for the CLCS problem 183
9.20. Comparison of the speedup for the classical CLCS dynamic programming algorithm 190
10.1. Example of computation of the MerLCS 194
10.2. Bit-parallel algorithm computing MerLCS length 200
10.3. Algorithm simulating removal of even rank bits (part of the MerLCS-BP algorithm) 201
10.4. Experimental comparison of the MerLCS length computing algorithms for various alphabet sizes 204
10.5. Experimental comparison of the MerLCS length computing algorithms for various sequence lengths 204

Spis tabel

1.1. Charakterystyka procesorów CPU i GPU 31

8.1. Czasy działania algorytmu LCTS-Hybrid ($w = 32$) dla problemu LCTS, dane: MUSIC 146
8.2. Czasy działania algorytmu LCTS-Hybrid ($w = 64$) dla problemu LCTS, dane: MUSIC 146
8.3. Czasy działania algorytmu LCTS-Hybrid ($w = 64$) dla problemu LCTS, dane: RAND-128 146
8.4. Czasy działania algorytmu LCTS-Hybrid ($w = 64$) dla problemu LCTS, dane: GAUSS-128 147

9.1. Zestawy testowe z danymi rzeczywistymi użyte w eksperymentach dla problemu CLCS. Kolumna H_0 zawiera entropię rzędu 0 ciągów w bitach (entropia ciągu dla rozkładu równomiernego przy rozmiarze alfabetu 20 wynosi około 4,322 bita) . . 163
9.2. Porównanie czasów działania (w ms) algorytmów wyznaczania podciągu CLCS dla danych rzeczywistych. W nawiasach: ile razy algorytm jest szybszy od algorytmu Chin 166
9.3. Porównanie czasów działania (w ms) algorytmów wyznaczania długości podciągu CLCS dla danych rzeczywistych. W nawiasach: ile razy algorytm jest szybszy od algorytmu Chin 167
9.4. Czasy działania (w ms) algorytmów wyznaczania podciągu CLCS i długości podciągu CLCS dla rzeczywistych danych. W nawiasach: ile razy dany algorytm jest szybszy niż ChinEE 185

10.1. Dane testowe wykorzystane w eksperymentach dla problemu MerLCS 202
10.2. Wyniki eksperymentalne dla danych rzeczywistych dla problemu MerLCS 203

Sekwencyjne i równoległe algorytmy znajdowania podciągów

Streszczenie

Pierwsza części niniejszej pracy poświęcona jest problemowi najdłuższego podciągu rosnącego (LIS) oraz jego wariantom (podciąg otrzymuje się z ciągu przez usunięcie zera bądź większej liczby symboli). Problem ten znajduje zastosowania m.in. w bioinformatyce do uliniawiania genomów, wyszukiwania nowych genów. Pierwszym z wariantów problemu LIS rozważanym w niniejszej pracy jest problem podciągów rosnących, które są pod pewnymi względami ekstremalne. Kolejnym wariantem jest problem podciągu rosnącego o zadanym pochyleniu. Dalsze dwa warianty to problemy cyklicznych podciągów rosnących oraz podciągów rosnących w oknie ustalonego rozmiaru ciągu wejściowego. Dla tych ostatnich wariantów zaproponowano w pracy wykorzystanie reprezentacji ciągu za pomocą pokrycia zachłannego oraz opracowano wydajne algorytmy łączenia takich pokryć. Algorytmy te są kluczowe do efektywnego rozwiązywania wspomnianych problemów.

Druga część pracy dotyczy problemu najdłuższego wspólnego podciągu i jego wariantów. Zastosowania tych problemów są bardzo liczne i dotyczą przede wszystkim porównywania ciągów w celu oceny ich podobieństwa. Dla problemu LCS niezmienniczego względem transpozycji (LCTS) zaproponowano kilka algorytmów sekwencyjnych, które, jak wynika z eksperymentów praktycznych, okazały się znacznie szybsze od algorytmów istniejących. Dla problemu ukierunkowanego LCS (CLCS) zaproponowano algorytmy sekwencyjne, również szybsze od dotychczas istniejących. Ponadto, zaproponowano dla tego problemu pierwszy algorytm równoległości bitowej. Dla problemu scalonego LCS (MerLCS) zaproponowano pierwszy algorytm równoległości bitowej, który w eksperymentach praktycznych okazał się kilkudziesięciokrotnie szybszy od znanych algorytmów. Dla problemów LCS, LCTS, CLCS zaproponowano także algorytmy równoległe przeznaczone do wykonywania w procesorach graficznych.

Dla wszystkich algorytmów proponowanych w niniejszej pracy przeprowadzono analizę złożoności czasowej i pamięciowej w przypadku pesymistycznym (dla niektórych także w przypadku średnim). Dzięki temu często można było wykazać, że proponowane algorytmy są także najszybsze w sensie asymptotycznym.

Serial and Parallel Subsequence Finding Algorithms

Abstract

The first part of this work is on the longest increasing subsequence problem (LIS) and its variants (a subsequence can be obtained from a sequence by removing zero or more symbols). The problem has applications in bioinformatics, e.g., in sequence alignment, searching new genes. The first variant of the LIS problem, which is considered in this work, is a problem of longest increasing subsequences that are extremal from some point of view. Next variant is a slope-constrained longest increasing subsequence problem. The last two discussed variants of the LIS problem are a longest increasing cyclic subsequence problem (LICS) and a longest increasing subsequence in a sliding window problem (LISW). The algorithms for the recent two problems use cover representation of a sequence. Original algorithms for cover merging are crucial to the proposed algorithms for the LICS and LISW problems.

The second part of this work is on the longest common subsequence problem (LCS) and its variants. The applications of these problems are numerous and concentrate mainly on the sequence comparison. For the transposition-invariant LCS problem (LCTS), a few sequential algorithms were proposed. Experiments show that they are much faster than the existing algorithms. For the constrained LCS problem (CLCS), a few sequential algorithms were also proposed. They are faster than the known algorithms. Moreover, for the CLCS problem, the first bit-parallel algorithm was invented. For the merged LCS problem (MerLCS), a bit-parallel algorithm, tens times faster than the existing algorithms was proposed. For the LCS, LCTS, CLCS problems also algorithms for graphical processors were invented.

All the proposed algorithms were analysed and their time and space complexities in the worst case were determined. For some algorithms the average case was also analysed. Obtained time complexities allow to show that the proposed algorithms are usually faster than the existing algorithms also in an asymptotic sense.

Skorowidz

alfabet, 27, 36, 56, 91, 118, 182, 184, 189, 192, 195, 202–204
    rozmiar, 27, 36, 91
algorytm
    CLCS-BP-Length, 15, 174, 175, 177, 181, 208
    CLCS-BP-Level-0, 174, 175
    CLCS-BP-Level, 172–175, 177
    CLCS-BP-Sequence, 15, 178, 181
    CLCS-BP-Start-Point, 169, 170, 173
    CLCS-BP-Strip, 171, 173, 174, 177
    CLCS-CUDA, 15, 184, 186–188, 209
    CLCS-HS-Length, 15, 159–161, 208, 209
    CLCS-HS-Sequence, 16, 161
    CLCS-BP-Level-0, 174
    Cover-Make, 19, 38–40, 68–72, 80–82
    Cover-Merge, 70, 71, 73, 81, 83
    LCS*-CUDA, 113, 147
    LCS-BP-CUDA, 16, 120
    LCS-BP-Lenght, 142
    LCS-BP-Length, 19, 99–102, 141–147, 151, 169, 174, 197
    LCS-BP-Sequence, 19, 101, 102
    LCS-DP-CUDA, 16, 116, 117
    LCS-HS, 19, 94–97, 135
    LCTS-BP-Length, 150
    LCTS-CUDA, 16, 148, 208
    LCTS-HS-*, 16
    LCTS-HS-1, 16, 17, 136, 139
    LCTS-HS-2, 16, 17, 136, 137, 139, 208
    LCTS-HS-3, 17, 138, 139, 141–147
    LCTS-HS-4, 17, 138, 139
    LCTS-HS, 150, 151
    LCTS-Hybrid, 17, 141, 145–147, 151
    LCTS-NGMD-One-Box, 130, 131
    LCTS-NGMD, 17, 131, 150, 208
    LCTS-One-Trans, 135, 136
    LICS, 17, 71, 73–75, 77, 81, 207
    LIS-Read, 20, 40
    LISW-Range, 81–83
    LISW, 17, 82–85, 208
    M***LIS, 44
    MaxHLIS, 18, 47–50, 53, 54, 56
    MaxSLIS, 18, 54, 56
    MaxWLIS, 18, 51, 52, 56
    MerLCS-BitPar, 201
    MerLCS-BP, 18, 200, 201, 209
    MinHLIS, 18, 45, 46, 49, 52, 54–56
    MinSLIS, 18, 53, 56
    MinWLIS, 19, 49, 50, 56
    SLIS, 19, 64
    Tableau-Insert, 20, 37
    BLAST, 107
    Hirschberga, 96, 101, 108, 111, 112, 126, 153, 202, 205
    Hunta–Szymanskiego, 93, 96, 97, 101, 126, 129, 130, 133, 141, 143–147, 155, 157, 160, 161, 185, 190, 191
    Kuo–Crossa, 96
    Maseka–Patersona, 105
    równoległy, 109, 122
    sekwencyjny, 31
    Shift-Or, 99
    Smitha–Watermana, 107
    Ziva–Lempela, 106
AMD/ATI, 30
architektura
    MIMD, 31
    SIMD, 30, 114
    SIMT, 30, 31

białko, 89, 91, 110, 164
bioinformatyka, 91, 107, 164, 190, 192, 203

ciąg, 26, 27, 89
    antyłańcuch, 79
    długość, 27
    główny, 152, 164, 166, 184, 189
    prefiks, 97
    rotacja, 66
    rozmiar, 27
    sufiks, 97
    suma, 44, 52
    szerokość, 44
    ukierunkowujący, 152, 164, 166, 169, 179, 182, 184, 189
    wysokość, 44, 46–48
CPU, 27, 29–32, 110–112, 114, 117–119, 121–124, 126, 149–151, 186, 189
    rdzeń, 29, 31
CUDA, 32, 111–113, 123, 124, 149, 189

DNA, 89, 91, 192
dopasowanie, 91, 92, 95, 100, 132, 134, 135, 144, 145, 152, 156–159, 164, 169, 171, 178, 182, 194, 197, 202, 203

dominujące, 92, 105, 132, 135, 136, 150
pseudo, 159
ranga, 150
silne, 152, 155, 156, 162, 168–177, 179, 180, 186
drzewo
w-arne, 137–139
AVL, 56, 73
czerwono-czarne, 73, 75, 134
łączenie, 76, 77
mediana, 76
podział, 75–77
usuwanie, 76
filogenetyczne, 89
poszukiwań
wykładnicze, 63
poszukiwań binarnych, 42, 56, 73, 101
zrównoważone, 39, 42, 96, 134
splay, 73
sufiksów, 107
van Emde Boasa, 29, 40, 56, 64, 72, 74, 130, 134–136, 139, 140
dziel i zwyciężaj, 97

element, 36, 91
krytyczny, 58–61, 63
ranga, 36, 59, 61, 63, 95, 199
maksymalna, 61

Fouriera transformata, 141

genom, 40, 107
duplikacja całego, hipoteza, 192
GPGPU, 32
GPU, 26, 27, 30–32, 110–114, 118, 120, 122–126, 147, 149–151, 184, 189, 191
Fermi, 30
multiprocesor, 30, 31, 111
rdzeń, 30, 31
graf
permutacji, 40
gramatyka bezkontekstowa, 106
produkcje, 106

heurystyka, 108

klaster obliczeniowy, 29
kodowanie długości serii, 106, 191
kolejka FIFO, 169, 179
kompresja, 89, 191
kopiec ograniczony, 154
korekcja pisowni, 89

LINPACK, 32

MIDI, 128, 139, 144
model obliczeniowy, 26, 28
AC^0 RAM, 28, 29
BSP, 29, 109
operacje podstawowe, 28
PRAM, 29
RAM, 28
Word RAM, 28, 29
MPI, 109
multiprocesor, 114, 117, 118, 122–125, 149, 189
MUMmer, 40

NVidia, 30–32, 110–112, 122, 149

odległość
Damerau, 90
edycyjna, 89, 90, 107
Hamminga, 90
indel, 90, 91
Levenshteina, 89
OpenCL, 32
operacja
arytmetyczna, 100
bitowa, 27, 100
iloczyn, 27
negacja, 27
przesunięcie
w lewo, 27
w prawo, 27
różnica symetryczna, 27
suma, 27
edycyjna, 89
usunięcie, 89, 90
wstawienie, 89, 90
zamiana, 89
następnik, 44, 94
poprzednik, 44, 94
usuwanie, 44, 94
wstawianie, 44, 94

pamięć
dostęp połączony, 31, 116, 118
globalna, 30–32, 112–119, 121, 122, 125, 126, 186–188
lokalna, 30
operacyjna

wspólna, 29
podręczna, 29, 125, 162, 166
RAM, 30, 112, 113, 115, 119, 122
stała, 30
tekstur, 30
wspólna, 30–32, 115, 117, 118, 125
pamięć wspólna, 119
plagiat, 89
podciąg, 27
niemalejący, 35, 36
pochylenie, 58, 60
rosnący, 26, 36, 80
długość, 36
najdłuższy, 26, 35–42, 44, 56, 66, 77, 78, 104
długość, 36, 43
suma, 53, 55
maksymalna, 42, 54–55
minimalna, 42, 52–54
szerokość
maksymalna, 42, 50–52
minimalna, 42, 49–50
ściśle, 35
w oknie, 67, 86
najdłuższy, 78–86
wysokość
maksymalna, 42, 47–49
minimalna, 42, 44–47
zadane pochylenie, 58
najdłuższy, 58–65
spójny, 27
szerokość, 50, 52
wspólny, 26
mozaikowy, 127
najdłuższy, 26, 40, 67, 92–128, 143, 167, 169, 190, 205
długość, 91, 100, 108
kompresja, 106
wszystkie, 127
niezmienniczy
najdłuższy, 128–151
rosnący
najdłuższy, 127
scalony, 26
najdłuższy, 192–205
ukierunkowany, 26
najdłuższy, 152, 159, 166, 167, 190
pokrycie ciągu, 38, 39, 43, 46, 47, 53, 67, 68, 71, 72, 81, 83
element
usunięcie, 72
usuwanie, 82, 83
wyszukanie, 72
lista, 39, 44–55, 67–73, 76
ostatnia, 45, 47, 49, 51, 69, 70, 80
podział, 83
złączenie, 83
łączenie, 70
odczyt, 67–69
reprezentacja, 82
drzewa, 73, 74, 84
listy, 72, 74, 86
listy drzew, 75, 76, 84, 86
rozmiar, 38
rozszerzone, 43
tworzenie, 83
zachłanne, 38, 43, 79
problem
CLCS, 152
LCS, 105
LCTS, 128
LICS, 66
LIS, 35, 105
LISW, 78
MaxHLIS, 47
MaxSLIS, 54
MaxWLIS, 50
MerLCS, 193
MinHLIS, 44
MinLIS, 52
MinWLIS, 49
poprzednika, 44, 46, 47, 55
następnik, 46, 55
usuwanie, 46, 55
wstawianie, 46, 55
SLIS, 58
procesor
zegar, 32
programowanie dynamiczne, 67, 92, 107, 112, 114, 119, 123, 124, 126, 129, 140, 153, 203, 205
macierz, 92, 97, 99, 108, 109, 112, 134, 136, 153, 155, 157, 158, 161, 162, 164, 166, 172, 174, 178, 181, 184, 189, 193, 195, 202
blok, 130
krawędź
dolna, 114, 119
górna, 113, 115, 119, 148, 175
lewa, 113–115, 119, 148
prawa, 114, 119
pasek, 169, 174–176, 178, 180, 182
pudełko, 109, 112–117, 119, 123, 124, 148, 185, 186, 189
rzadka, 129, 132, 150
projekcja
elementu, 60–63

całkowita, 63, 64
punktu, 60, 62
punkt stopu, 69, 71, 72, 80–83

ranga, 94, 100
rejestr, 31, 32, 115
RNA, 152
RNase, 152, 162, 164, 184
równoległość bitowa, 98–100, 102, 103, 114, 118, 120, 124, 126, 129, 139, 141–143, 145–147, 151, 168, 169, 173–175, 182, 191, 195, 197, 200, 203, 205
wariant Hirschberga, 103

sieć komputerowa, 29
słownik, 89, 90
słowo komputerowe, 28, 98, 101, 137, 180–182, 200–202
rozmiar, 27
sortowanie pozycyjne, 64
superkomputer, 32
symbol, 27, 36, 91
ranga, 48, 50
szereg czasowy, 128

tableau Younga, 37, 79
wstawianie, 37
taksonomia Flynna, 30
tekstura, 30
teksty, 89
transpozycja, 132, 136, 141, 142, 145, 148

uliniowienie ciągów, 91, 107, 108, 110, 152
przerwa, 91
ukierunkowane, 152, 155, 162, 190
wynik, 91, 108

wątek, 30, 31, 118
blok, 31, 112
liczba, 31
serializacja, 30, 119
wiązka, 31
wywołanie jądra, 31, 32, 113, 115, 186
wywołanie kodu jądra, 117, 123
wejścia-wyjścia punkty, 155, 161, 164, 166, 186, 189, 190
wektor
bitowy, 100, 102, 118, 142, 169, 170, 172, 179, 193, 197, 200
masek, 101, 118, 119, 121, 148, 173, 175, 180
zmian, 195, 197
wyszukiwanie
binarne, 59, 95
wzorca w tekście, 99

zero wiodące
liczba, 137