ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1999
Seria: INFORMATYKA z. 37 Nr kol. 1421

Tomasz PODESZWA
Politechnika Śląska, Instytut Informatyki

ALGORYTMY REKURENCYJNE - PRZYKŁAD PRAKTYCZNEGO WYKORZYSTANIA

Streszczenie. W artykule przedstawiono podstawowe problemy występujące podczas programowania z wykorzystaniem algorytmów rekurencyjnych z powrotami oraz przykładowe sposoby ich rozwiązania na podstawie procedury wypełniającej wzorcową krzyżówkę hasłami ze wstępnie przygotowanego słownika.

RECURSIVE ALGORITHMS - EXAMPLE OF PRACTICAL USE

Summary. The article presents basic problems which appears when recursive backtracking algorithms are used and methods to solve them. The sample procedure to fill pattern crossword using prepared dictionary is presented.

1. Wprowadzenie

Spośród wielu różnych technik programowania stosowanych do rozwiązywania typowych problemów informatycznych zdecydowanie najrzadziej wykorzystywana jest rekurencja. Spowodowane jest to prawdopodobnie niepełnym omówieniem tego sposobu programowania w podręcznikach oraz niezbyt trafnie dobranym przykładom przedstawiającym, na czym rekurencja w programowaniu polega. W przypadku niektórych problemów jest ona wręcz niezastąpiona znacząco upraszczając ich rozwiązanie.

Niestety, „nie ma róży bez kolców" i w przypadku wielu innych problemów zastosowanie procedur rekurencyjnych nie daje takich uproszczeń. Podstawowymi problemami skłaniającymi do rezygnacji z rekurencji są:

a) mała efektywność w przypadku niektórych problemów, szczególnie gdy dają się one łatwo zdefiniować w sposób nierekurencyjny;

b) znaczne wykorzystanie stosu systemowego i związana z tym możliwość stracenia wyników dłuższych obliczeń w wyniku wystąpienia błędu przepełnienia stosu;

c) bardzo trudna realizacja operacji przerwania rekurencji z zachowaniem dotychczasowego stanu pracy oraz późniejsze wznowienie obliczeń.

W niniejszej pracy przedstawiono praktyczny sposób rozwiązania problemów zawartych w punktach b) i c) dla procedur dokonujących automatycznego wypełniania krzyżówki na podstawie wbudowanego słownika haseł. Prezentowane rozwiązania można z powodzeniem wykorzystać w innych podobnych problemach rekurencyjnych.

2. Opis problemu

Jak wspomniano we wstępie, w niniejszej pracy zostanie przedstawiony przykładowy sposób usunięcia niedogodności funkcji rekurencyjnych na podstawie modułu programu wspomagającego układanie krzyżówek, a dokładnie jego części wypełniającej zaprojektowany wzór hasłami ze słownika.

Istotne problemy występujące podczas korzystania z procedur rekurencyjnych i ich źródła zostaną przedstawione w punktach 2.1, 2.2 i 2.3.

2.1. Mała efektywność

Ten argument jest wysuwany jako jeden z podstawowych problemów występujących podczas tworzenia funkcji rekurencyjnych. Wynika to przede wszystkim z błędnego przeświadczenia, że funkcja rekurencyjna jest zawsze wolniejsza od analogicznej funkcji nierekurencyjnej, popartego klasycznym przykładem obliczania wartości funkcji n! Niestety, ten prosty przykład pokazuje właśnie, że nie każdy problem przedstawiony w postaci rekurencyjnej należy w tej właśnie postaci kodować w programie. W przypadku funkcji n! istnieje bowiem może mniej elegancka w zapisie matematycznym, ale bardzo prosta do realizacji programowej postać nierekurencyjna (z wykorzystaniem pętli for).

Istnieją jednak problemy, których rozwiązanie w postaci rekurencyjnej jest zdecydowanie najprostszym lub wręcz jedynym z możliwych. Przykładem takiego problemu jest zadanie ustawienia ośmiu hetmanów na szachownicy, tak by żaden nie szachował innego [2], czy też inne zadania, gdzie nie ma analitycznego rozwiązania problemu i należy stosować metodę prób i błędów.

2.2. Obciążenie stosu systemowego

Ten problem jest niestety decydujący w przypadku części zadań, których rozwiązanie metodą rekurencyjną jest najprostsze. Ponieważ każde wywołanie procedury powoduje pozostawienie śladu (adresu spod którego procedura została wywołana) na stosie oraz rezerwację miejsca na zmienne lokalne oraz parametry wywołania funkcji, więc w krytycznych przypadkach już kilkakrotne wywołanie procedury może spowodować wystąpienie błędu przepełnienia stosu. Oszacowanie maksymalnej wielkości potrzebnego miejsca na stosie jest więc jedną z najważniejszych czynności, które należy wykonać poza zapisaniem algorytmu w wybranym języku programowania.

2.3. Przerwanie i wznowienie rekurencji

Problem przerywania i wznawiania wykonywania procedur rekurencyjnych jest również dość ważny w praktycznych zastosowaniach. Ponieważ większość problemów do rozwiązania, w których stosuje się rekurencję, nie ma analitycznego rozwiązania, więc najczęściej stosuje się do nich metodę sprawdzania kolejnych rozwiązań i odrzucania tych, które nie dają poprawnego rozwiązania. Rekurencja z powrotami (bo o niej tu mowa) pozwala na znalezienie dla takich zadań jednego (o ile istnieje) lub też wszystkich rozwiązań, ma jednak podstawową wadę (jak wszystkie metody brute force): dużą złożoność obliczeniową, która powoduje, że nawet podczas korzystania z szybkiego komputera czas do znalezienia rozwiązania (lub rozwiązań) jest dość długi.

W takich przypadkach należałoby zapewnić możliwość przerwania algorytmu z zapamiętaniem bieżącego stanu i ewentualną kontynuacją przerwanej pracy. Może to mieć znaczenie również w przypadku poszukiwania najlepszego z rozwiązań, w którym po każdym znalezionym rozwiązaniu następuje jego wartościowanie (albo według zadanego kryterium, albo na podstawie odpowiedzi udzielonej przez człowieka) i ewentualna kontynuacja lub przerwanie dalszych poszukiwań.

3. Przykładowe zastosowanie rekurencji

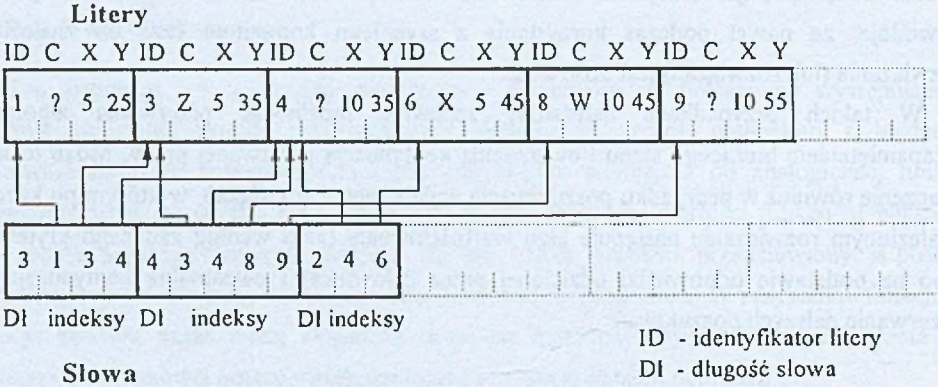
Aby pokazać sposób rozwiązania przykładowych problemów występujących podczas korzystania z rekurencji, przedstawione zostanie zadanie układania krzyżówki z wykorzystaniem zdefiniowanego wcześniej słownika haseł.

3.1. Opis zadania

Mając zdefiniowaną postać krzyżówki (położenie poszczególnych haseł i ich wzajemne powiązania) oraz podany słownik haseł, należy uzupełnić brakujące hasła krzyżówki hasłami ze słownika. W przypadku typowych zadań tworzenia krzyżówek dochodzi często założenie częściowego wypełnienia wzorca krzyżówki pojedynczymi literami albo hasłami. Należy więc uwzględnić to założenie przy projektowaniu algorytmu wyszukującego hasła do wypełnienia.

3.2. Struktury danych wykorzystane do rozwiązania zadania

Jednym z głównych elementów każdego programu są, oprócz algorytmu, także struktury danych, na których ten algorytm będzie działał. Ponieważ założono, że program powinien być jak najmniej zależny od samego kształtu (wyglądu) krzyżówki, najbardziej optymalnym sposobem opisu powiązań pomiędzy hasłami krzyżówki (litery wspólne) oraz ich położeniem na rysunku okazał się format przedstawiony na rys. 1. Jak łatwo zauważyć, opis krzyżówki jest zawarty w dwóch tablicach. Pierwsza z nich służy do przechowywania poszczególnych liter krzyżówki i zawiera oprócz samej litery także jej położenie. Każda z liter jest rozróżniana na podstawie pola ID (które w ogólnym przypadku nie jest równoważne jej indeksowi w tablicy).



Rys. 1. Sposób powiązania haseł i liter krzyżówki
Fig. 1. Relation between words and letters of the crossword

Druga tablica jest zbiorem haseł zawierającym odwołania do elementów tablicy liter. Ponieważ hasła z reguły są różnej długości, więc dodatkowo jest przechowywana długość danego hasła. Jak widać to na powyższym rysunku, każda litera może należeć do wielu haseł. Ponadto, jeżeli dla jednego hasła ustalimy literę, zostanie ona również uwzględniona we wszystkich hasłach, w których występuje.

Oprócz przedstawionej struktury danych odwzorowującej postać krzyżówki, wykorzystano również typową relacyjną bazę danych jako słownik haseł. Ponieważ każde z haseł może posiadać wiele różnych znaczeń, jak też pojedynczy opis może odpowiadać wielu różnym hasłom, więc słownik ma typową organizację M:N.

Dodatkowo został utworzony programowy stos, który zastąpił stos systemowy po zmodyfikowaniu procedury rekurencyjnej. Jest on zrealizowany jako dynamiczna tablica ze wskaźnikiem na aktualny wierzchołek, zatem operacje związane z tym stosem są identyczne z operacjami na stosie systemowym (włóż na stos/zdejmij ze stosu).

3.3. Opis algorytmu

Jak wspomniano wcześniej, do wypełniania przygotowanego wzorca krzyżówki użyto procedury rekurencyjnej, która metodą prób i błędów (rekurencja z powrotami) szuka rozwiązania (wypełnienia), korzystając z podanego słownika.

Aby pokazać siłę programowania z wykorzystaniem rekurencji, na rys. 2. przedstawiono

```
BOOL Szukaj(int nr_wyrazu)
{
  STAN stan;

  if (nr_wyrazu <= ilość_wyrazów)
    if (!ZnajdźPierwszeHasło(nr_wyrazu, stan))
      return FALSE;
    else
      while (!Szukaj(nr_wyrazu+1))
        if (!ZnajdźNastępneHasło(nr_wyrazu, stan))
          return FALSE;
  return TRUE;
};
```

Rys. 2. Schemat rekurencyjnego algorytmu poszukiwania wypełnienia krzyżówki
Fig. 2. Recursively search algorithm scheme

postać pierwotną (przed modyfikacjami) procedury wypełniającej (w postaci funkcji języka C).

Funkcje ZnajdźPierwszeHasło() oraz ZnajdźNastepneHasło() szukają hasła ze słownika pasującego do wyrazu o indeksie *nr_wyrazu*, oraz w przypadku znalezienia takiego wpisują je do wzorca krzyżówki. Zmienna *stan* służy do przechowywania poprzedniej zawartości hasła (przed wstawieniem nowego słowa). W przypadku porażki (nie udało się znaleźć słowa pasującego w dane miejsce krzyżówki) funkcja zwraca wartość *FALSE*. Pozytywne wypełnienie krzyżówki kończy się po znalezieniu wszystkich haseł (funkcja zwraca wartość *TRUE*).

Z analizy wydruku przedstawionego na rys. 2 widać, że funkcja szukająca jest bardzo prosta i chociaż posiada typowe wady funkcji rekurencyjnych (przedstawionych w rozdziale 2), to w wielu przypadkach może zostać użyta w przedstawionej postaci po dopisaniu brakujących funkcji.

3.4. Modyfikacje procedury szukającej

Mając najprostszą postać funkcji wypełniającej krzyżówkę, spróbujmy teraz tak ją zmodyfikować, by usunąć przynajmniej część niedogodności, które ma w obecnej postaci.

3.4.1. Obciążenie stosu

Zacznijmy od obciążenia stosu, a właściwie zabezpieczenia się przed możliwością jego przepełnienia. Zauważmy, że dla n wyrazów (haseł) do uzupełnienia krzyżówki funkcja Szukaj() jest wywoływana rekurencyjnie $n+1$ razy (jedno wywołanie dla każdego wyrazu oraz jedno wywołanie kończące rekurencję). Za każdym razem na stosie systemowym oprócz adresu powrotu są również przechowywane zmienne lokalne (zmienna *stan*) oraz argumenty funkcji. Przyjmując, że w zmiennej *stan* oprócz poprzedniej wartości słowa krzyżówki należy przechowywać także stan słownika, który także jest modyfikowany (gdyż hasła użyte są odpowiednio zaznaczane), może się okazać, że pojedyncze wywołanie funkcji szukaj wymaga kilkuset bajtów stosu. Dla 100 wyrazów i przy obciążeniu stosu 100 B na każde wywołanie potrzebujemy prawie 10 KB wolnego miejsca na stosie. Z podanego przykładu wynika, że przekroczenie granicy 64 KB (maksymalna wartość stosu dla procesorów 80x86 przy programach 16-bitowych) nie jest takie trudne. Ponadto wielkość tego stosu musi zostać ustalona jeszcze na etapie kompilacji.

```
BOOL Szukaj(int nr_wyrazu)
{
  STAN * pstan = new STAN;

  if (nr_wyrazu <= ilość_wyrazów)
    if (!ZnajdźPierwszeHasło(nr_wyrazu, *pstan))
      {
        delete stan;
        return FALSE;
      }
    else
      while (!Szukaj(nr_wyrazu+1))
        if (!ZnajdźNastepneHasło(nr_wyrazu, *pstan))
          {
            delete stan;
            return FALSE;
          }
  delete stan;
  return TRUE;
};
```

Rys. 3. Wersja poprawiona algorytmu - użycie zmiennych dynamicznych
Fig. 3. Modified version of algorithm - dynamic data use

Najprostszym sposobem ograniczenia wielkości stosu wymaganego podczas wywołania procedury jest zastąpienie zmiennej lokalnej *stan* wskaźnikiem do dynamicznie tworzonej na stercie zmiennej *pstan*. Zmienna ta (jak to przedstawiono na rys. 3) jest tworzona operatorem new() na początku procedury, a usuwana przy jej zakończeniu (przed instrukcją return)

operatorem delete(). Tym razem zamiast kilkusetbajtowej struktury *stan* na stosie systemowym jest przechowywany kilkubajtowy wskaźnik *pstan*, natomiast sama struktura jest przechowywana na praktycznie nielimitowanej stercie. Pomimo widocznych ulepszeń dalej jednak pozostaje problem wstępnego ustalenia wielkości stosu na poziomie niepowodującym jego przepełnienia.

Usunięcie tej niedogodności jest możliwe dzięki zrezygnowaniu z klasycznej rekurencji, w której funkcja wywołuje samą siebie (bezpośrednio czy też pośrednio) procedurą nierekurencyjną z zaimplementowanym stosem programowym i pętlą działającą podobnie do mechanizmu rekurencji. Przykład takiej funkcji został przedstawiony na rys. 4.

```
STOS * stos;          // zmienna globalna, wykorzystana także przez funkcje
                      // ZnajdźPierwszeHasło() i ZnajdźNastepneHasło()
BOOL Szukaj()
{
  stos = new STOS();
  stos->zeruj();
  int Stop = 0;
  // poszukiwanie rozwiązania...
  while (!Stop)
    {
      //*****
      int nr_wyrazu = stos->IlośćElementów();
      if (nr_wyrazu == ilość_wyrazów)
        {
          Stop = 2;
          break;                                  // rozwiązanie znalezione...
        }
      stos->wstaw(* new STAN);
      if (!ZnajdźPierwszeHasło())
        {
          delete &stos->zdejmij();
          while (!stos->CzyPusty() && !ZnajdźNastepneHasło())
            {
              // *****
              delete & stos->zdejmij();           // wyjdź o jeden poziom wyżej
            }
          if (stos->CzyPusty())                   // koniec poszukiwań
            Stop = 1;
        }
    }
  while (stos->CzyPusty())                        // wyczyszczenie stosu
    delete & stos->zdejmij();
  delete stos;
  return (Stop == 2) ? TRUE : FALSE;
}
```

Rys. 4. Wersja nierekurencyjna algorytmu
Fig. 4. Non-recursive algorithm version

Zmienna globalna stos jest zmienną obiektową, dla której zdefiniowano operacje zeruj(), wstaw(), zdejmij(), IlośćElementów() oraz funkcję CzyPusty(). Wskaźnikiem poziomu rekurencji jest tutaj aktualna wielkość (głębokość) stosu. Jak łatwo zauważyć, poniższa

```
STOS * stos;        // zmienna globalna, wykorzystana także przez funkcje
                    // ZnajdźPierwszeHasło() i ZnajdźNastępneHasło()
BOOL Szukaj(BOOL OdPoczątku)
{
  if (OdPoczątku)
    {
      ......... // zerowanie stosu i
      ......... // odtworzenie zawartości słownika na podstawie zawartości stosu
    }
  int Stop = 0;
  // jeżli stos pełny, to znajdź następne rozwiązanie
  int nr_wyrazu = stos->IlośćElementów();
  if (nr_wyrazu == ilość_wyrazów)
    {
      BOOL rezultat;
      Stop = 3;
      while (!stos->CzyPusty() && !(rezultat = ZnajdźNastępneHasło()))
        {
          delete &stos->zdejmij();
          Stop = 0;                                  // zacznij od drugiej fazy
        }
      if (!rezultat)                                 // brak następnych rozwiązań
        Stop = 2;
    }
  // faza druga - szukaj następnego rozwiązania
  while (!Stop)
    {
      nr_wyrazu = stos->IlośćElementów();
      if (nr_wyrazu == ilość_wyrazów)
        {
          Stop = 3;
          break;                                     // rozwiązanie znalezione
        }
      stos->wstaw(* new STAN);
      if (!ZnajdźPierwszeHasło())
        {
          delete & stos->zdejmij();
          while (!stos->CzyPusty() && !ZnajdźNastępneHasło())
            {
              // !!!!!
              delete & stos->zdejmij();              // szukaj dalej
            }
          if (stos->CzyPusty())                      // brak rozwiązań...
            Stop = 2;
        }
      // ***** sprawdzenie przerwania przez użytkownika
      //       (i ewentualne ustawienie Stop = 1)
    }
  return (Stop == 3) ? TRUE : FALSE;
}
```

Rys. 5. Wersja nierekurencyjna algorytmu z możliwością wznowienia działania
Fig. 5. Non-recursive algorithm version with resume option

procedura jest wywoływana tylko raz, czyli praktycznie nie obciąża stosu. Ponadto wszystkie zmienne potrzebne do działania algorytmu są przechowywane na stercie (w tym również zmienna stos).

Dodatkowo, wstawiając w miejsce gwiazdek sprawdzenie warunku przerwania (np. naciśnięcia kombinacji klawiszy) i odpowiedniego ustawienia zmiennej *Stop*, można łatwo przerwać działanie algorytmu.

3.4.2. Wznowienie przerwanej rekurencji

Problem wznowienia działania przerwanej procedury rekurencyjnej związany jest przede wszystkim z koniecznością zapamiętania bieżącego stanu rekurencji w momencie przerywania oraz z koniecznością odtworzenia tego stanu w chwili wznawiania.

W przypadku tradycyjnej rekurencji należałoby przenieść zawartość stosu systemowego do zmiennej pomocniczej. Zawartość tej zmiennej byłaby następnie wykorzystywana podczas fazy odtwarzania stanu stosu.

Jak łatwo się domyślić, w przypadku gdy korzysta się ze stosu programowego, zadanie jest znacznie ułatwione, gdyż nie trzeba usuwać i odtwarzać stanu stosu systemowego. Należy jedynie w odpowiednim miejscu wznowić działanie procedury szukającej.

Na rys. 5. przedstawiono tak zmodyfikowaną procedurę, która dodatkowo pozwala na znalezienie kolejnych rozwiązań.

Jak łatwo zauważyć, główną zmianą jest wprowadzony fragment umożliwiający wznowienie rekurencji przerwanej w wyniku znalezienia rozwiązania. Poza tym nie można badać warunku przerwania przez użytkownika w wewnętrznej pętli **while** (w miejscu oznaczonym wykrzyknikami), gdyż spowodowałoby to błędne działanie algorytmu. Należy także dodać, że utworzenie i usunięcie zmiennej *stos* należy wykonać w procedurze wywołującej funkcję szukaj(), gdyż właśnie tam jest podejmowana decyzja, czy rekurencja ma być wznawiana oraz od jakiego punktu (od początku, czy też od ostatniego miejsca przerwania).

3.4.3. Efektywność procedury rekurencyjnej

W przeciwieństwie do poprzednio rozpatrywanych problemów występujących podczas używania procedur rekurencyjnych uzyskanie poprawy efektywności jest bardzo trudne. Wynika to głównie z tego, że w zależności od samego problemu rozwiązywanego algorytmem rekurencyjnym różnie się go realizuje. W przedstawionym zadaniu należy zadbać przede wszystkim o to, by szukanie haseł w słowniku (realizowane przez funkcje ZnajdźPierwszeHasło() i ZnajdźNastepneHasło()) było możliwie efektywne, gdyż głównie od tej operacji zależy czas poszukiwania rozwiązań.

Ponieważ jest to zadanie związane zarówno z odpowiednią strukturą słownika jako struktury danych (zastosowanie indeksacji), jak też z odpowiednią kolejnością szukania haseł poniżej zostaną przedstawione skrótowo te rozwiązania.

a) Budowa słownika

Słownik został zaprojektowany jako uproszczona relacyjna baza danych o dwóch tabelach (hasła oraz objaśnienia) z powiązaniami N:M, w której jedno hasło może mieć wiele objaśnień, jak również jedno objaśnienie może mieć przypisanych sobie wiele haseł. W celu zapewnienia szybkości szukania tabela z hasłami jest posortowana alfabetycznie (ma to znaczenie, jeżeli szukane hasło ma pierwszą literę znaną). W przypadku poszukiwania hasła, którego pierwsza litera nie jest znana, wykorzystywane jest proste przeszukiwanie liniowe.

b) Kolejność poszukiwanych haseł

Ze względu na specyfikę budowy słownika oraz ilość możliwych do znalezienia w nim haseł, w procesie poprzedzającym szukanie rozwiązań zastosowano prosty algorytm zmieniający kolejność wyszukiwania poszczególnych haseł. Najpierw wybierane są te wyrazy, których pierwsze litery są znane. Jeśli takich nie ma, wybierany jest wyraz posiadający największą ilość znanych liter lub (w ostateczności) najdłuższy. Po wybraniu kolejnego wyrazu zapamiętywane są wszystkie litery, z których on się składa, a następnie wybierany jest kolejny wyraz zgodnie z przedstawionym powyżej algorytmem.

Taki sposób sortowania wyrazów powoduje, że ewentualne kombinacje wyrazów nie prowadzące do rozwiązania są stosunkowo szybko eliminowane w procesie szukania.

4. Podsumowanie

Jak to zostało przedstawione w poprzednim rozdziale, programowanie z wykorzystaniem rekurencji może być bardzo efektywnym sposobem rozwiązania pewnych problemów, których rozwiązanie analityczne nie jest znane lub których algorytm rozwiązania jest na tyle skomplikowany, że staje się nieopłacalny do zakodowania.

Podstawowe problemy (przedstawione w rozdziale 2) można stosunkowo niewielkim kosztem rozbudowy algorytmu usunąć bez większego wpływu na jego efektywność, zastępując stos systemowy odpowiednio zaimplementowanym stosem programowym, który w większości przypadków nakłada dużo mniejsze ograniczenia, dodatkowo umożliwiając odtwarzanie przerwanego procesu szukania w zasadzie od dowolnego miejsca. W przypadku wersji nierekurencyjnej operacją decydującą o czasie wykonania procedury jest i tak pewna funkcja szukająca kolejnego kierunku ruchu (w przykładzie powyżej będzie nim szukanie kolejnego hasła, które można wpisać w pola krzyżówki).

Podsumowując, przy wyborze rekurencji jako metody realizacji algorytmu należy przeanalizować również (o ile istnieją) sposoby nierekurencyjne rozwiązania problemu i dokonać wyboru uwzględniającego wszystkie plusy i minusy każdego z tych rozwiązań.

odpowiednio ważąc ich znaczenie. Przykładowo, jeżeli okaże się, że decydująca jest szybkość algorytmu, a wersja rekurencyjna jest tylko kilka procent wolniejsza od rozwiązania nierekurencyjnego, za to znacznie czytelniejsza, należałoby raczej skorzystać z tej pierwszej metody, gdyż zaoszczędzony w drugiej wersji czas może być niewspółmierny do czasu potrzebnego dla pielęgnacji (poprawiania błędów i ulepszania) programu. Warto więc poświęcić więcej czasu na optymalne przygotowanie danych wykorzystywanych w procesie szukania, zamiast poszukiwania wymyślnego nierekurencyjnego algorytmu, który w wersji ostatecznej może się okazać niezbyt udany.

LITERATURA

1. Stroustrup B.: Język C++. Wydawnictwa Naukowo-Techniczne, Warszawa 1994.
2. Wirth N.: Algorytmy + struktury danych = programy. Wydawnictwa Naukowo-Techniczne, Warszawa 1989.

Recenzent: Dr inż. Maciej Bargielski

Wpłynęło do Redakcji 7 maja 1998 r.

Abstract

Recursive backtracking algorithm is powerful tool for solve some specific computer science problems which solution without this algorithms is too difficult. But recursive algorithm makes some traps for programmers. Because typical recursive function or procedure calls itself many times while program works, this can bring to stack overflow (and abnormal program termination) or long work time (what can be very irritating for user).

This paper presents some suggestions for programmers who wants to protect owns programs from this disadvantage. The simple example presents how system stack can be easy changed to program stack, what makes possible extending stack space to avoid stack overflow. It also allows to use pause and resume option inside running function so user can watch work progress. Basic version of algorithm is presented on fig. 2. The second algorithm (fig. 3) uses variables allocated on heap instead of stack. This causes less stack utilization. The third algorithm (fig. 4) uses program stack and can be easily added pause and resume option (fig. 5).