ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 2000
Seria: AUTOMATYKA z. 129 Nr kol. 1474

Piotr LIPIŃSKI, Mieczysław WODECKI
Uniwersytet Wrocławski

PROBLEM SZEREGOWANIA ZADAŃ Z LINIAMI KRYTYCZNYMI NA JEDNEJ MASZYNIE. ALGORYTM GENETYCZNY

Streszczenie. W pracy rozpatrywany jest jednomaszynowy problem minimalizacji sumy kosztów zadań opóźnionych. W literaturze jest on oznaczany przez $1||\sum w_i T_i$ i należy do klasy problemów *silnie NP-zupełnych.* Do jego rozwiązywania przedstawimy odpowiednio zaadaptowany algorytm genetyczny.

SINGLE MACHINE TARDINESS SCHEDULING PROBLEM. A GENETIC ALGORITHM

Summary. In the paper, one – machine sequencing problem is considered under condition that the total weighted tardiness cist is minimized. Some genetic algorithms and computation results are presented.

1. Wstęp

Od wielu lat procedury metaheurystyczne, tj. przeszukiwania tabu, symulowanego wyżarzania oraz algorytmu genetycznego są z powodzeniem stosowane do rozwiązywania (wyznaczania rozwiązań przybliżonych) zagadnień kombinatorycznych należących do klasy problemów *silnie NP-zupełnych.* Choć ich czas obliczeń jest zazwyczaj dosyć długi, to jednak pozwalają one na znaczną poprawę rozwiązań wyznaczonych przez specjalizowane heurystyki.

Algorytmy genetyczne (w skrócie AG) są probabilistycznymi algorytmami przeszukiwania i obejmują grupę metod obliczeniowych, których wspólną cechą jest korzystanie, przy rozwiązywaniu problemu z mechanizmu opartego na zjawisku naturalnej ewolucji gatunków [7], [9]. Są one bezpośrednią adaptacją tego zjawiska, stąd w ich opisie używa się pojęć z genetyki. W metodach tych, na wyróżnionych podzbiorach zbioru rozwiązań dopuszczalnych wykonywane są cyklicznie trzy podstawowe operacje: selekcji, krzyżowania i mutacji. Proces ten ma charakter ewolucyjny i prowadzi do wygenerowania podzbioru zawierającego „najbardziej obiecujące" rozwiązania.

W pracy przedstawiamy algorytm genetyczny rozwiązywania zagadnienia optymalizacji kolejności wykonywania zadań na jednej maszynie, w którym kryterium optymalności jest suma kosztów zadań wykonanych nieterminowo (opóźnionych). W problemie tym, każde z n

zadań (ponumerowanych liczbami *1,2, ... ,n*) należy wykonać, bez przerywania, na jednej maszynie. Maszyna ta w dowolnej chwili może wykonywać co najwyżej jedno zadanie. Dla zadania *i* (*i=1,2, ... ,n*) niech p_i, w_i, d_i będą odpowiednio: *czasem wykonywania*, *wagą funkcji kosztów* oraz *linią krytyczną*. Jeżeli ustalona jest kolejność wykonywania zadań oraz C_i jest terminem zakończenia wykonywania zadania *i* (*i=1,2, ... ,n*), to $T_i = \max\{0, C_i - d_i\}$ nazywamy *opóźnieniem*, a $f_i(C_i)=w_i \cdot T_i$ *kosztem opóźnienia* zadania. Rozważany problem polega na wyznaczeniu takiej kolejności wykonywania zadań, która minimalizuje *sumę kosztów opóźnień*, tj. $\sum w_i T_i$. Należy on do klasy problemów *silnie NP-zupełnych* (Lawler [11], Lenstra i inni [13]]).

Obecnie opisanych jest w literaturze wiele algorytmów optymalnych rozwiązywania rozpatrywanego problemu, opartych na metodzie podziału i ograniczeń ([2],[5],[14],[16]), a także na metodzie programowania dynamicznego ([12],[17]). Algorytmy te pozwalają rozwiązywać (w rozsądnym czasie) przykłady, w których liczba zadań jest nie większa niż 50. Ze względu na małą efektywność tych algorytmów w praktycznych zastosowaniach dużą rolę odgrywają algorytmy przybliżone. Nie zawsze satysfakcjonująca jakość rozwiązań wyznaczonych przez nawet najlepsze algorytmy heurystyczne ([8], [16]) oraz bardzo interesujące wyniki otrzymane dla wielu zagadnień przez zastosowanie metaheurystyk (symulowanej relaksacji, przeszukiwania tabu oraz algorytmu genetycznego), przedstawione w pracach [1], były inspiracją adaptacji rzadziej stosowanej metody algorytmu genetycznego do rozwiązywania rozpatrywanego problemu kolejnościowego.

2. Klasyczny algorytm genetyczny

Klasyczny algorytm genetyczny stosuje się do maksymalizacji funkcji celu F o wartościach rzeczywistych określonej na zbiorze ciągów binarnych o ustalonej długości *n*. Ogólny schemat takiego algorytmu można zapisać następująco:

Algorytm 2.1. Klasyczny algorytm genetyczny

$k \leftarrow 0$;
$P_k \leftarrow$ LosowaPopulacja;
repeat
  Selekcja(P_k,P'_k); {*Wybór rodziców*}
  Krzyżowanie(P'_k,P''_k); {*Generowanie potomstwa*}
  Mutacja(P''_k);
  $P_{k+1} \leftarrow P''_k$; {*Nowa populacja*}
  $k \leftarrow k + 1$;
until WarunekKońca;

Działanie algorytmu genetycznego rozpoczyna się od utworzenia populacji początkowej P_0 (przez populację rozumie się tutaj m elementową kombinację z powtórzeniami 2^n elementowego zbioru dopuszczalnych rozwiązań) składającej się z ustalonej liczby m osobników (rozwiązań dopuszczalnych) $\pi_1, \pi_2, \ldots, \pi_m$. Proces ten polega na m-krotnym losowaniu, z jednostajnym rozkładem prawdopodobieństwa, ciągu binarnego π_i długości n. Stosowanie jednostajnego rozkładu prawdopodobieństwa gwarantuje, że każdy element przestrzeni zostanie wylosowany z takim samym prawdopodobieństwem równym 2^{-n}. Następnie rozpoczyna się symulowanie ewolucji. Proces ten składa się z selekcji naturalnej i reprodukcji. W wyniku ewolucji z bieżącej populacji P_k zostanie utworzona populacja potomna P_{k+1}. Wszystkie populacje przez cały czas działania algorytmu będą miały taką samą liczbę elementów m. Każdemu osobnikowi π_i z populacji P_k można przyporządkować pewną wartość $f(\pi_i)$, zwaną przystosowaniem. Zazwyczaj jest to wartość funkcji celu, tj. $f(\pi_i)=F(\pi_i)$. Definiując przystosowanie wprowadza się często różne skalowanie funkcji celu.

Selekcja naturalna polega na wygenerowaniu z populacji P_k populacji rodziców P'_k. Proces ten polega na m-krotnym losowaniu osobnika π_i z bieżącej populacji P_k. Prawdopodobieństwo wylosowania elementu π_i jest wprost proporcjonalne do jego przystosowania $f(\pi_i)$. Dzięki temu rodzicami zostają osobniki najlepiej przystosowane. Następnie wylosowani rodzice są łączeni w pary. Każda para podlega procesowi reprodukcji, w wyniku którego powstaje para potomków zastępująca rodziców w nowej populacji.

Reprodukcja polega na wygenerowaniu z pary rodziców dwóch potomków (ciągów binarnych długości n). Proces ten składa się z krzyżowania i mutacji. Najpierw losowana jest liczba naturalna k z przedziału ($0 \le k \le n$), określająca miejsce krzyżowania. Następnie ciągi rodziców są krzyżowane, tzn. pierwsze k bitów z pierwszego rodzica jest kopiowane na początek pierwszego potomka, pierwsze k bitów z drugiego rodzica jest kopiowane na początek drugiego potomka, zaś ostatnie n-k bitów pierwszego rodzica trafia na koniec drugiego potomka oraz ostatnie n-k bitów drugiego rodzica trafia na koniec pierwszego potomka. Na zakończenie odbywa się proces mutacji polegający na negacji, z bardzo małym prawdopodobieństwem, skopiowanych bitów. W wyniku przedstawionego procesu zostaje utworzona nowa populacja P''_k, złożona z m ciągów binarnych długości n. Jest ona następnym pokoleniem i staje się populacją bieżącą w następnej iteracji. Dobry algorytm genetyczny powinien działać tak, aby z dużym prawdopodobieństwem nowe pokolenie było lepiej przystosowane.

Proces ewolucji powinien zostać przerwany w momencie, gdy od pewnego czasu populacja nie rozwija się (zbyt szybka zbieżność algorytmu) lub gdy osobniki wchodzące w skład populacji są do siebie bardzo podobne (ograniczenie poszukiwania do pewnego podzbioru). Należy także pamiętać najlepszych osobników w całej historii gatunku, gdyż może zdarzyć się, że pojawią się oni w pewnym momencie i nie przetrwają do ostatniego pokolenia.

3. Algorytmy genetyczne dla problemów kombinatorycznych

Rozpatrzmy zagadnienie optymalizacji polegające na maksymalizacji funkcji F o wartościach rzeczywistych określonej na zbiorze n-elementowych permutacji. Zastosowanie klasycznego algorytmu genetycznego okazuje się niemożliwe, gdyż metody reprodukcji, operatory krzyżowania i mutacji, mogą prowadzić do wygenerowania potomka, który nie jest permuta-cją (rozwiązaniem dopuszczalnym). Można dopuścić do występowania w populacji osobni-ków nie będących permutacjami i wprowadzić system kar i nagród, który premiuje permutacje w procesie selekcji naturalnej. Dzięki temu osobniki nie będące permutacjami szybko wyginą, ustępując miejsca permutacjom, które będą odgrywać główną rolę w procesie ewolucji. W pracy [3] przedstawiono system binarnej reprezentacji rozwiązań, dzięki któremu można bezpośrednio stosować klasyczny algorytm genetyczny do rozwiązywania problemów kombinatorycznych. Jeszcze inne podejście polega na zmianie operatorów krzyżowania i mutacji, tak aby w wyniku ich działania uzyskać potomków będących permutacjami.

W literaturze opisano kilka propozycji algorytmów krzyżowania permutacji generujących potomków, będących także permutacjami. Poniżej przedstawiamy trzy z nich.

Operator krzyżowania **PMX** (*Partial – Mapped Crossover*) został zaproponowany przez Goldberga i Lingle'a [6]. Polega on na wylosowaniu dwóch liczb naturalnych z przedziału $[1..n]$, które wyznaczają podział chromosomu na trzy części. Środkowe części są zamieniane tzn. środkowa część pierwszego rodzica trafia do drugiego potomka, środkowa część drugiego rodzica trafia do pierwszego potomka. Odwzorowanie to wyznacza funkcję między pewnymi liczbami z przedziału $[1..n]$. Pozostałe części rodziców są kopiowane do odpowiednich potomków, przy czym, jeśli kopiowana liczba burzy strukturę permutacji, to zamiast niej jest wstawiana wartość wyznaczona przez określone przyporządkowanie.

Przykład 1: Operator krzyżowania **PMX**

```
P1 = ( 1 2 3|4 5 6 7|8 9 )          {rodzic pierwszy}
P2 = ( 4 5 2|1 8 7 6|9 3 )          {rodzic drugi}

4 - 1   5 - 8   6 - 7   7 - 6
     ( . . .|1 8 7 6|. . )
     ( . . .|4 5 6 7|. . )
```

```
C1 = ( 4 2 3|1 8 7 6|5 9 )          {potomek pierwszy}
C2 = ( 1 8 2|4 5 6 7|9 3 )          {potomek drugi}
```

Operator krzyżowania **OX** (*Order Crossover*) został przedstawiony w pracy Davisa [4]. Polega on, podobnie jak operator **PMX**, na losowym podziale każdego z rodziców na trzy części. Środkowe części są wymieniane. Występowanie liczb w pierwszym z rodziców definiuje pewien ciąg. Wypisując te liczby kolejno z 3., 1. i 2. części otrzymujemy pewien ciąg liczb naturalnych. Z niego wykreślane zostają te liczby, które występują w środkowej części drugiego rodzica. Reszta jest kopiowana kolejno do 3. i 1. części pierwszego potomka. Podobnie dla drugiego rodzica.

Przykład 2: Operator krzyżowania **OX**

```
P1 = ( 1 2 3|4 5 6 7|8 9 )          {rodzic pierwszy}
P2 = ( 4 5 2|1 8 7 6|9 3 )          {rodzic drugi}

     ( . . .|4 5 6 7|. . )
     ( . . .|1 8 7 6|. . )

8 - 9 - 1 - 2 - 3 - 4 - 5 - 6 - 7
9 - 3 - 4 - 5 - 2 - 1 - 8 - 7 - 6

9 - 2 - 3 - 4 - 5
9 - 3 - 2 - 1 - 8

C1 = ( 2 1 8|4 5 6 7|9 3 )      {potomek pierwszy}
C2 = ( 3 4 5|1 8 7 6|9 2 )      {potomek drugi}
```

Operator **CX** (*Cyclic Crossover*) został przedstawiony przez Hollanda [10]. Polega on na wymianie cykli między rodzicami. Z przeprowadzonych eksperymentów obliczeniowych dla omawianego w tej pracy problemu szeregowania wynika, że jest to najlepszy operator krzyżowania. Dlatego poniżej przedstawiamy dokładny jego algorytm.

Algorytm. Algorytm krzyżowania permutacji **CX**

```
for i := 1 to n do
    C1[i] := 0;
    C2[i] := 0;
r := 1;
C1[r] := P1[r];
C2[r] := P2[r];
while (P1[r] <> P2[r])
    k := 1;
    while (P1[k] <> P2[r])
        k := k + 1;
    r := k;
    C1[r] := P1[r];
    C2[r] := P2[r];
for i := 1 to n do
    if(C1[i] = 0) C1[i] := P2[i];
    if(C2[i] = 0) C2[i] := P1[i];
```

Przykład 3: Operator krzyżowania CX

```
P1 = ( 1 2 3 4 5 6 7 8 9 )        {rodzic pierwszy}
P2 = ( 4 1 2 8 7 6 9 3 5 )        {rodzic drugi}

C1 = ( 1 2 3 4 7 6 9 8 5 )        {potomek pierwszy}
C2 = ( 4 1 2 8 5 6 7 3 9 )        {potomek drugi}
```

Najczęściej stosowanymi operatorami mutacji w algorytmach genetycznych dla rozwiązywania problemów kombinatorycznych są:

- **inwersja,** polegająca na wycięciu losowego fragmentu permutacji i wstawieniu go w to samo miejsce, ale w odwrotnej kolejności,
- **wstawienie,** polegające na wylosowaniu liczby z przedziału $[1...n]$ i wstawieniu jej w losowe miejsce permutacji. Oczywiście, liczba ta powinna zostać usunięta z miejsca, gdzie występowała przed mutacją, aby w wyniku nie zaburzyć struktury permutacji,
- **przemieszczanie,** polegające na wycięciu losowego fragmentu permutacji i wstawieniu go w innym, wylosowanym miejscu,
- **wzajemna wymiana,** polegająca na wylosowaniu dwóch elementów w permutacji i zamianie ich miejscami.

4. Algorytm genetyczny dla jednomaszynowego problemu szeregowania zadań

Niech $N=\{1,2, \ldots ,n\}$ będzie zbiorem wszystkich zadań, a Π zbiorem permutacji elementów z N. Dla permutacji $\pi \in \Pi$ przez:

$$F(\pi) = \sum_{i=1}^{n} f_{\pi(i)}(C_{\pi(i)})$$

oznaczamy *koszt permutacji* (tj. sumę kosztów opóźnień, gdy zadania są wykonywane w kolejności występowania w π). Rozważany problem sprowadza się do wyznaczenia permutacji optymalnej (o minimalnym koszcie) w zbiorze wszystkich permutacji Π.

W konstrukcji algorytmu genetycznego zastosowano "naturalną", tj. w postaci permutacji, reprezentację kolejności wykonywania zadań. Dalej przedstawiamy podstawowe modyfikacje zwiększające efektywność działania algorytmu.

Przede wszystkim zastosowano tzw. elitarną strategię ewolucji, polegającą na tym, że kolejne pokolenie zawsze zawiera (omijając proces selekcji naturalnej i reprodukcji) k najlepiej przystosowanych osobników z poprzedniej populacji. Aby zachować stałą liczbę osobników we wszystkich populacjach odrzuca się k najgorzej przystosowanych osobników potomnych z populacji P''_k powstałej w wyniku procesu reprodukcji. Dzięki użyciu tej metody gwarantuje się przetrwanie najlepiej przystosowanych osobników w całej historii gatunku. W celu ograniczenia przeszukiwania obszarów rozwiązań dopuszczalnych o dużych wartościach funkcji celu zastosowano strategię częściowej zagłady gatunku. Strategia ta polega na usunięciu części populacji i zastąpieniu jej losowo wygenerowanymi osobnikami w

przypadku, gdy cała populacja gromadzi się w niewielkim obszarze wokół minimum lokalnego. Jest to wzmocnienie operatora mutacji, który w krytycznych sytuacjach okazuje się nieskuteczny.

Istotną modyfikacją jest usprawnienie metod krzyżowania. Proces wykonywania zadań przez maszynę można podzielić na trzy fazy, dzieląc tym samym permutację na trzy części. Można przypuszczać, że w pierwszej fazie powinny być wykonywane zadania o dużej wadze i małej wartości linii krytycznej. W ostatniej fazie powinny być wykonywane zadania o małej wadze i dużej wartości linii krytycznej. W środkowej fazie będą wykonywane pozostałe zadania. Oczywiście, kolejność wykonywania zadań w pierwszej fazie nie wpływa na opóźnienie zadań wykonywanych w dwóch kolejnych fazach. Podobnie, kolejność wykonywania zadań w drugiej fazie nie wpływa na opóźnienie zadań w trzeciej fazie. Na początku działania algorytmu unika się zmiany kolejności wykonywania zadań w poszczególnych fazach, a szczególnie w trzeciej fazie. Dopiero po wygenerowaniu kilku pierwszych populacji rozpoczyna się właściwe działanie algorytmu.

Kolejna modyfikacja polega na premiowaniu zachowywania bloków zadań o ustalonej długości. W początkowej fazie działania algorytm zachowuje przy krzyżowaniu permutacji bloki zadań (długości 2 – 5, w zależności od liczby zadań). Pozwala to na szybkie wyznaczanie dobrej populacji początkowej.

Zastosowano także dodatkowe operatory mutacji. Pierwszy z nich jest modyfikacją operatora inwersji, polegającą na wycięciu losowego fragmentu permutacji i wklejeniu go w odwrotnej kolejności, o ile polepszy to jakość rozwiązania. Drugi operator, będący modyfikacją operatora wzajemnej wymiany, zamienia miejscami zadania znajdujące się na wylosowanym miejscu z zadaniem, które po umieszczeniu na tym miejscu będzie miało najmniejszy koszt wykonania. Trzeci z kolei oparty jest na operatorze przemieszczania i polega na wycięciu losowego fragmentu permutacji i wstawieniu go w innym miejscu tak, aby maksymalnie polepszyć jakość rozwiązania.

Kolejną modyfikacją jest odpowiednie skalowanie funkcji celu. W przypadku, gdy kilka osobników danej populacji ma bardzo wysoki wskaźnik przystosowania, niewspółmiernie większy niż reszta populacji, osobniki te zdominują populację rodziców. To z kolei spowoduje dominację ich potomków w następnym pokoleniu. W wyniku tego zbieżność algorytmu będzie zbyt szybka – algorytm nie przeszuka całej przestrzeni rozwiązań skupiając się tylko na małym jej fragmencie. Inne zagrożenie może wystąpić, gdy wszystkie osobniki populacji mają bardzo zbliżoną wartość funkcji przystosowania. Wówczas o ich wejściu do populacji rodziców będzie decydował w dużej mierze przypadek, a nie wskaźnik przystosowania. W celu uniknięcia powyższych problemów zastosowano skalowanie funkcji

nia. W celu uniknięcia powyższych problemów zastosowano skalowanie funkcji celu. Jeżeli w populacji występuje kilka osobników o wyjątkowo dużej wartości przystosowania, to skalowanie spowoduje znormalizowanie różnicy. Jeżeli cała populacja ma bardzo zbliżony współczynnik przystosowania, to skalowanie funkcji celu spowoduje istotne uwydatnienie różnic między poszczególnymi osobnikami. W konstrukcji algorytmów zastosowano trzy rodzaje funkcji skalującej: funkcję logarytmiczną, funkcję liniową i funkcję wykładniczą.

5. Eksperymenty obliczeniowe

Przedstawiony w poprzednich punktach pracy algorytm genetyczny testowano na wielu losowych przykładach. Sposób ich generowania dokładnie przedstawiono i przedyskutowano w pracy [14]. Testy przeprowadzono także na wybranej grupie przykładów, które okazały się szczególnie trudne dla innych algorytmów rozwiązywania omawianego problemu.

Dla każdego zadania algorytm uruchamiano kilkakrotnie z różnymi operatorami krzyżowania (**PMX, OX, CX**), z różnymi funkcjami skalującymi funkcję przystosowania i różnymi wartościami parametrów algorytmu. Poniżej przedstawiamy wartości tych parametrów:

- wielkość populacji: 100 – 1000 osobników,
- prawdopodobieństwo krzyżowania: 0.75 – 0.95,
- prawdopodobieństwo mutacji (dla każdego rodzaju mutacji osobno): 0.0001 – 0.01,
- próg wzrostu przystosowania, poniżej którego następuje częściowa zagłada: 1%,
- część populacji ulegająca zagładzie: 10% – 50%.

Jako warunek kończący działanie algorytmu przyjęto kryterium zbieżności populacji. Polega ono na sprawdzaniu wzrostu średniego lub maksymalnego przystosowania osobników w kolejnych populacjach oraz różnicy między maksymalnym i minimalnym przystosowaniem osobników. Algorytm kończył działanie, gdy w kolejnych populacjach nie uzyskiwano poprawy rozwiązania lub gdy rozwiązania w ramach danej populacji niewiele się od siebie różniły (około 1%). Świadczyło to bowiem o nikłej szansie dalszego rozwoju populacji.

W tabeli 1 przedstawiono wyniki działania algorytmu. Dla ustalonej liczby zadań (równej 40, 50, 100) wykonano testy na 100 przykładach. W tabeli zamieszczono średni i maksymalny błąd uzyskanych rozwiązań (względem rozwiązań optymalnych) oraz liczby iteracji i czas działania algorytmu. Obliczenia wykonano na komputerze z procesorem Pentium 450MHz.

Tabela 1

Wyniki obliczeń

Liczba zadań	Średni błąd (%)	Maksymalny błąd (%)	Średnia ilość iteracji	Średni czas działania (s)
40	4.05	9.41	54	41

50	2.64	5.48	81	56
100	2.01	4.87	214	83

Przeprowadzone obliczenia dla mniejszej liczby zadań (20 – 30) nie dają wyników lepszych niż inne algorytmy heurystyczne. Wynika to z faktu, że algorytm genetyczny wymaga dostatecznie dużej przestrzeni rozwiązań dopuszczalnych i wówczas widoczna jest jego zdecydowana przewaga nad innymi metodami.

Przeprowadzono także testy dla przykładów o większej liczbie zadań (200 – 500). Uzyskane wyniki (w porównaniu z innymi algorytmami przybliżonymi) były zadowalające. Niestety, z powodu braku algorytmów generujących dobre rozwiązania nie było możliwe przeprowadzenie precyzyjnych porównań.

Na podstawie eksperymentów stwierdzono także, że najlepsze wyniki otrzymuje się przy zastosowaniu operatora krzyżowania **CX** oraz logarytmicznej funkcji skalującej. Istotny wpływ na jakość uzyskiwanych rozwiązań ma wprowadzenie modyfikacji uwzględniających specyfikę problemu szeregowania, które zostały omówione we wcześniejszych rozdziałach.

6. Podsumowanie

W pracy przedstawiono zastosowanie algorytmów genetycznych do rozwiązywania problemów szeregowania zadań na jednej maszynie. Wyniki eksperymentów obliczeniowych wskazują na dużą efektywność tej metody, w szczególności w przypadkach problemów o większej liczbie zadań. W porównaniu z innymi metodami lokalnego przeszukiwania algorytm genetyczny okazuje się lepszy, o ile liczba zadań jest dostatecznie duża.

Dalsze prace nad ulepszeniem algorytmu genetycznego, głównie poprzez większe wykorzystanie specyfiki problemu, może doprowadzić do zwiększenia jego efektywności. Ponadto interesujące jest zagadnienie zastąpienia algorytmu genetycznego przez rozwijane ostatnio tzw. strategie ewolucyjne lub ewolucję różnicową.

LITERATURA

1. Aarts E., Lenstra J.K. (edited): Local Search in Combinatorial Optimization, John Wiley&Sons Ltd., 1997.
2. Adrabiński A., Grabowski J., Wodecki M.: Algorytm rozwiązywania zagadnienia kolejnościowego postaci $1||\sum w_i T_i$, Archiwum Automatyki i Telemechaniki, Tom XXXIII (1988), 623-636.
3. Crauwels H.A.J., Potts C.N., Van Wassenhove L.N.: Local Search Heuristics for the Single Machine Total Weighted Tardiness Scheduling Problem, INFORMS Journal on Computing, Vol. 10, No. 3, Summer 1998.
4. Davis L.: Genetic Algorithms and Simulated Annealing, London, Morgan Kaufmann, 1987.

5. Fisher M.L.: A Dual Algorithm for the One-Machine Scheduling Problem, Mathematical Programming, 11 (1976), 229-252.
6. Goldberg D. E.: Lingle, Alleles, Loci and the TSP, Proc. Intern. Conf. on Genetic Algorithms, Pittsburg, 1985, pp. 154-159.
7. Goldberg D. E.: Genetic Algorithms i Search, Optimization and Machine Learning, Addison-Wesley, 1989.
8. Grabowski J., Pempera J.: Algorytmy szeregowania zadań z kryterium minimalnokosztowym, Zeszyty Naukowe Politechniki Śląskiej, s. Automatyka, z. 125 Gliwice 1998, 37-45.
9. Holland J.: Adaptation in natural and artificial systems. Univ. of Michigan Press, Ann Arbor, MI, 1975.
10. Holland J, Oliver R., Smith W.: A study of permutation crossover operators on the TSP, Proc. Intern. Conf. on Genetic Algorithms, MIT, Cambridge 1987,. 224-230.
11. Lawler E.L.: A "Pseudopolynomial" Algorithm for Sequencing Jobs to Minimize Total Tardiness, Annals of Discrete Mathematics 1 (1977), 331-342.
12. Lawler E.L.: Efficient Implementation of Dynamic Programming Algorithms for Sequencing Problems, Report BW106, Mathematisch Centrum, Amsterdam (1979).
13. Lenstra, J.K. Rinnoy Kan, Brucker P.: Complexity of Machine Scheduling Problems, Annals of Discrete Mathematics 1 (1977), 343-362.
14. Potts C.N., Van Wassenhove L.N.: A Branch and Bound Algorithm for the Total Weighted Tardiness Problem, Operations Research, 33 (1985), 177-181.
15. Potts C.N., Van Wassenhove L.N.: Single Machine Tardiness Sequencing Heuristics, IIE Transactions, Volume 23, Number 4 (1991), 346-354.
16. Rinnoy Kan A.H.G.,Lageweg B.J., Lenstra J.K.: Minimizing total costs in one-machine scheduling, Operations Research, 25 (1975), 908-927.
17. Schrage L., Baker K.R.: Dynamic Programming solution of Sequencing Problems with Precedence Constraints, Operational Research, 26 (1978), 444-449.

Recenzent: Prof.dr hab.inż. A.Niederliński

Abstract

This paper deals with heuristics for the well-known single machine tardiness problem. Consider n jobs (numbered 1, ..., n) to be processed without interruption on a single machine that can handle only one job at a time. Job i (i = 1, ..., n) becomes available for processing at time zero, requires a positive processing time p_i, has a positive weight w_i, and has a due date d_i. For a given sequence of the jobs the earliest completion time C_i and the tardiness $T_i = \max\{C_i - d_i, 0\}$ of job i (i = 1, ..., n) can be computed. In the total weighted tardiness problem the objective is to find a processing order of the jobs that minimizes $\sum_{i=1}^{n} w_i T_i$. When all job weights are equal, minimizing $\sum_{i=1}^{n} T_i$ is called the total tardiness problem. In the paper, we propose several genetic algorithms. Finally, the computation results and discussion of the performance of algorithms are presented.