

Projektowanie efektywnych algorytmów

Jednoprocesorowy problem szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań

Zadanie projektowe nr 1:
Metoda programowania dynamicznego

Iwo Bujkiewicz (226203)

Teoria

Opis problemu

Jednoprocesorowy problem szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań (ang. *Single machine total weighted tardiness problem*, SMTWTP) zdefiniowany jest następująco:

Dany jest pewien zestaw zadań, z których każde opisane jest indywidualnym numerem, ilością jednostek czasu potrzebnych do jego wykonania, wagą (priorytetem) i oczekiwanym terminem zakończenia jego wykonywania. Każde zadanie jest dostępne do wykonywania w chwili zero. Zadanie jest spóźnione, jeśli jego wykonywanie zakończy się po oczekiwanym terminie. Miara opóźnienia zadania jest równa różnicy chwili zakończenia jego wykonywania i oczekiwanego terminu zakończenia, jednak nie mniejsza, niż 0. Zadania wykonywane są bez przerw przez pojedynczy procesor mogący wykonywać co najwyżej jedno zadanie jednocześnie. Znaleźć taką kolejność wykonywania zadań, aby zminimalizować sumę opóźnień wszystkich zadań pomnożonych przez ich wagi.

Algorytm

Wykorzystany w projekcie algorytm został skonstruowany częściowo na podstawie [2].

Algorytm rozpoczyna pracę od posortowania zadań w kolejności od najwcześniejszego do najpóźniejszego oczekiwanego terminu zakończenia. Jest to część heurystyczna i daje dobrą, ułożoną najczęściej korzystniej niż losowo, bazę do działań metaheurystycznych.

Następuje właściwe wyszukiwanie rozwiązania z uwzględnieniem funkcji celu. Początkowo algorytm rozpatruje podsekwencję zadań o długości 2, znajdującą się na początku całej sekwencji, a w kolejnych etapach bierze pod uwagę coraz dłuższe (o 1 zadanie na etap) podsekwencje. Na każdym etapie testuje wszystkie możliwości zamiany ostatniego elementu podsekwencji z jednym z poprzednich elementów, a następnie wykonuje taką zamianę, która daje najlepszy wynik funkcji celu (najniższą ważoną sumę opóźnień) dla danej podsekwencji, jeśli ten wynik jest lepszy od stanu aktualnego.

Kluczowa dla algorytmu jest jego ostatnia część, bez której byłby on raczej mało użyteczny. Za każdym razem, kiedy wykonana zostanie zamiana, algorytm powtarza wykonanie aktualnego etapu, co rekurencyjnie powoduje powtórzenie wszystkich dotychczasowych etapów. Pozwala to uniknąć sytuacji, w której 'krytyczne' zadanie zostałoby wypchnięte na późniejszy termin realizacji, znacznie zwiększając ważoną sumę opóźnień, bez możliwości poprawy.

Zadanie 'krytyczne' to takie, którego umieszczenie na dalszych pozycjach sekwencji zwiększa ważoną sumę opóźnień bardziej, niż większość pozostałych zadań. W praktyce są to zazwyczaj zadania, których waga jest stosunkowo wysoka, a dodatkowo charakteryzują się wczesnym terminem zakończenia i/lub długim czasem wykonywania. Ponowne rozpatrzenie aktualnego i poprzednich etapów sprawia, że algorytm jest w stanie dokonać poprawek w rozmieszczeniu rozważanych już wcześniej zadań, w zmienionej sytuacji. Dodatkową zaletą takiego rozwiązania jest fakt, że podczas ponownego przejścia może się okazać, że wzajemna pozycja zadań, która wcześniej nie miała znaczenia dla wyniku funkcji celu, stała się istotna i można ją poprawić. Przykładem takiej sytuacji jest wykonanie zamiany, w wyniku której z pozycji 8. na pozycję 4. przeniesione zostaje zadanie o dłuższym czasie wykonywania, niż to, które znajdowało się tam poprzednio. Jest możliwe, że przed tą zamianą zadania na pozycjach 5. i 6. nie były spóźnione, ale po zamianie są. Należy wtedy rozpatrzyć, czy zamiana zadań na pozycjach 5. i 6. nie poprawiłaby przypadkiem wyniku funkcji celu.

```
Zadanie : { numer, czas_wykonywania, waga, termin }
```

```
Dane
```

```
  $zadania : Lista (Zadanie) [1..n]
```

```
Funkcja wazona_suma_opoznień($zakres) : Liczba
```

```
  $wynik := 0
```

```
  $chwila := 0
```

```
  dla każdego $i w $zakres
```

```
    $chwila := $chwila + $zadania[$i].czas_wykonywania
```

```
    $wynik := $wynik + max(0, $zadania[$i].termin - $chwila) * $zadania[$i].waga
```

```
  zwróć $wynik
```

```
Koniec
```

```

Procedura znajdź_rozwiazanie($k)
    jeżeli $k > 2
        wykonaj znajdź_rozwiazanie($k - 1)

    $najlepsza_suma_opoznien := wazona_suma_opoznien(od 1 do $k)
    $najlepsza_pozycja := $k

    dla każdego $i od 1 do ($k - 1)
        zamień $zadania[$i] z $zadania[$k]
        jeżeli wazona_suma_opoznien(od 1 do $k) < $najlepsza_suma_opoznien
            $najlepsza_suma_opoznien := wazona_suma_opoznien(od 1 do $k)
            $najlepsza_pozycja := $i
        zamień $zadania[$i] z $zadania[$k]

    jeżeli $najlepsza_pozycja != $k
        zamień $zadania[$najlepsza_pozycja] z $zadania[$k]
        wykonaj znajdź_rozwiazanie($k)
Koniec

Start
    posortuj $zadania według Zadanie.termin rosnąco

    wykonaj znajdź_rozwiazanie(n)
Koniec

```

Pseudokod

Złożoność obliczeniowa

Złożoność czasową opisanego algorytmu można oszacować w następujący sposób.

Operacja	Przypadek średni	Przypadek najgorszy
Sortowanie heurystyczne W załączonej implementacji: pochodna <i>mergesort</i> i <i>insertion sort</i> ^[3]	$O(n \log n)$	$O(n \log n)$
Obliczenie ważonej sumy opóźnień	$O(k)$	$O(k)$
Znalezienie najlepszej zamiany w podsekwencji	$O(k^2)$	$O(k^2)$
Powtórzenie dotychczasowych etapów	$O((3k/4) * (3k/4)) = O(k^2)$	$O(k!)$
Główne wyszukiwanie	$O((n^2 + n^2) * n^2)$	$O(n!)$

Górnym ograniczeniem złożoności czasowej tego algorytmu jest liczba wszystkich permutacji sekwencji zadań, ponieważ każda zamiana zadań może jedynie poprawić wynik funkcji celu, a w najgorszym wypadku algorytm może przechodzić przez te permutacje od najgorszej do najlepszej. W średnim zaś przypadku złożoność czasowa wynosi około $O(n^4)$.

Złożoność pamięciowa algorytmu, jeśli nie liczyć stosu, wynosi $O(1)$. Wliczenie stosu (zawierającego adresy powrotne metody rekurencyjnej) zwiększa jednak tą złożoność do $O(n^3)$ w średnim przypadku i $O(n!)$ w najgorszym przypadku.

Przykład działania

Rozważmy przykładową sekwencję 4 zadań:

numer	czas_wykonywania	waga	termin
1	26	1	118
2	24	10	122
3	79	9	133
4	46	10	127

Początkowo sekwencja ma następującą postać i ważoną sumę opóźnień:

{ 1, 2, 3, 4 } 480

Po posortowaniu według najwcześniejszego terminu otrzymujemy sekwencję z mniejszą sumą opóźnień.

{ 1, 2, 4, 3 } 378

Rozpoczynamy rekurencję. Na początek rozważamy następującą podsekwencję:

{ 1, 2 } 0

Żadna zamiana nie poprawi wyniku w tej podsekwencji, przechodzimy zatem do następnej.

{ 1, 2, 4 } 0

Sytuacja się powtarza, przechodzimy znów do kolejnej podsekwencji.

{ 1, 2, 4, 3 } 378

Rozważamy po kolei możliwości zamiany miejscami zadania nr 3 z zadaniami 1, 2 i 4. Zamiana miejscami zadań 1 i 3 daje wynik 277, 2 i 3 - 770, 4 i 3 - 480. Wynik 277 jest lepszy od aktualnego (378), wykonujemy więc zamianę zadań 1 i 3.

{ 3, 2, 4, 1 } 277

Powtarzamy poprzednio wykonane etapy, tym razem na nowej sekwencji

{ 3, 2 } 0
{ 3, 2, 4 } 220

Sprawdzamy, czy możemy poprawić taką podsekwencję. Zamiana miejscami zadań 3 i 4 daje wynik 144, 2 i 4 - 270. Wynik 144 jest lepszy od aktualnego (220), wykonujemy więc zamianę zadań 3 i 4. Powtarzamy etapy.

{ 4, 2, 3 } 144
{ 4, 2 } 0

```
{ 4, 2, 3 }      144
```

Sprawdzamy, czy możemy poprawić taką podsekwencję. Zamiana miejscami zadań 4 i 3 daje wynik 220, 2 i 3 - 270. Brak możliwości poprawy.

```
{ 4, 2, 3, 1 }    201
```

Sprawdzamy, czy możemy poprawić taką podsekwencję. Zamiana miejscami zadań 4 i 1 daje wynik 378, 2 i 1 - 692, 3 i 1 - 378. Brak możliwości poprawy.

W tym miejscu wychodzimy z rekurencji i działanie algorytmu się kończy. Nasz końcowy wynik to zatem:

```
{ 4, 2, 3, 1 }    201
```

... co jest optymalną kolejnością wykonywania tych czterech zadań.

Opis implementacji

Do implementacji algorytmu wybrano język Java w wersji 8. W celu ułatwienia uruchamiania testów zaimportowano do projektu bibliotekę Reflections (`org.reflections`), jednak nie jest ona konieczna do poprawnego działania żadnych elementów projektu poza klasą `TestRunner` .

Do przechowywania sekwencji zadań wykorzystano strukturę `ArrayList` , rozszerzoną o dodatkowe metody. Samo pojedyncze zadanie było reprezentowane przez klasę `Job` , implementującą wzorzec JavaBean, posiadającą cztery atrybuty: `id` (numer), `processingTime` (czas potrzebny do wykonania), `dueTime` (oczekiwany termin zakończenia) oraz `weight` (waga).

Początkowe sortowanie heurystyczne odbywało się za pośrednictwem wbudowanej w Java API metody `ArrayList.sort()` , do której jako argument podawana była instancja specjalnego komparatora:

```
public class EarliestDueDateHeuristic implements Comparator<Job> {
    @Override
    public int compare(Job o1, Job o2) {
        return Comparator.<Job>comparingInt(Job::getDueTime).compare(o1, o2);
    }
}
```

Zamiany zadań w sekwencji wykonywane były z użyciem wbudowanej metody `Collections.swap()` .

Do wczytywania danych pobranych z [1] do testowania algorytmu użyto odpowiednio przygotowanej klasy `JobOrderLoader` .

Analiza wyników i wydajności

Działanie algorytmu testowano z użyciem zestawów danych pobranych z [1]: `wt40.txt` , `wt50.txt` oraz `wt100.txt` . Podczas analizy wyników wykorzystano udostępnione tamże pliki z optymalnymi lub najlepszymi dotychczas znalezionymi wartościami funkcji celu: `wtopt40.txt` oraz `wtbest100b.txt` .

Bibliografia

1. Beasley, J.E., 1990-2017. *OR-Library* (<http://people.brunel.ac.uk/~mastjjb/jeb/orlib/wtinfo.html>)
2. Congram, R.K., Potts, C.N., van de Velde, S.L., 1998. *An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem*

3. *Java™ Platform, Standard Edition 8 API Specification* (Oracle America, Inc., March 2015)
(<https://docs.oracle.com/javase/8/docs/api/>)