Jednomaszynowy problem minimalizacji sumy kosztów zadań spóźnionych

Metody optymalizacji, temat 3, zadanie 1
Michał Zimniak, Damian Dyńdo

Sformułowanie problemu

- Dany jest zbiór N = {1,2,…,n} zadań do wykonania na pojedynczej maszynie.
- W danej chwili może być wykonywane tylko jedno zadanie bez wywłaszczania.
- Każde zadanie i ∈ N ma przyporządkowaną krotkę (p_i, w_i, d_i), gdzie
  - p – czas trwania zadania
  - w – waga funkcji kosztów (koszt spóźnienia)
  - d – wymagany termin zakończenia (linia krytyczna)
- Niech C_i będzie terminem wykonania zadania. ($C_i = \sum_{j=1}^{i} p_j$).
- Niech $U_i = \begin{cases} 1 \text{ gdy } C_i > d_i \\ 0 \qquad w\,p.p. \end{cases}$ będzie spóźnieniem.
- Wówczas problem polega na znalezieniu takiej kolejności zadań (permutacji zbioru N), która minimalizuje sumę kosztów spóźnień, tj. $\sum_{i=1}^{n} w_i U_i$.

Notacja

- N = {1,2,…,n} – zbiór zadań opisanych krotką:
  (p – czas wykonywania, w - koszt, d – linia krytyczna).
- S(N) zbiór permutacji elementów z N.
- π – permutacja zbioru. $\pi \in S(N)$.
- $\pi(\mathrm{i})$ - i-ty element permutacji π, przykładowo $\mathrm{C}_{\pi(\mathrm{i})}$- czas zakończenia wykonywania i-tego zadania w perm. π.
- C_i – termin wykonania zadania *i*. U_i – spóźnienie.
- $f_i(c) = \sum_{i=1}^{n} w_i U_i$ – koszt spóźnienia wykonywania zadania *i* rozpoczętego w czasie *c*.

Model matematyczny

- Dane
  - n – rozmiar zbioru zadań.
  - n krotek (p, w, d) opisujących te zadania
    - p – czas wykonania
    - w – waga funkcji kosztów
    - d – wymagany termin zakończenia
- Wyjście to permutacja π^*, taka że
  $F(\pi^*) = \min\{F(\delta) : \delta \in S(N)\}$,
  gdzie funkcja celu to
  $F(\pi) = \sum_{i=1}^{n} f_{\pi(i)}(C_{\pi(i)}) = \sum_{i=1}^{n} w_{\pi(i)} U_{\pi(i)}$.

Charakter problemu

- Jednomaszynowy problem minimalizacji sumy kosztów zadań spóźnionych jest oznaczany w literaturze przez $1||\sum w_i U_i$.
- Ten wariant szeregowania zadań należy do klasy problemów NP-trudnych.
- Algorytmy wyznaczania rozwiązania optymalnego mają wykładnicze zapotrzebowanie na pamięć i czas obliczeń, dlatego mogą być stosowane jedynie do rozwiązywania przykładów o niewielkich rozmiarach.
- Obecnie stosuje się niemal wyłączenie algorytmy przybliżone, które często dają bardzo zadowalające wyniki różniące się od najlepszych rozwiązań o mniej niż 1%.

Algorytmy poszukiwania zstępującego

- Są to algorytmy przybliżone.
- Polegają na generowaniu z bieżącego rozwiązania, podzbioru rozwiązań (otoczenia), z którego jest wybierany najlepszy element.
- Jest on kolejnym bieżącym rozwiązaniem.
- Postępowanie to jest kontynuowane aż do osiągnięcia minimum lokalnego.
- Sposób generowania i przeglądania otoczenia ma znaczny wpływ na złożoność obliczeniową, czas działania i jakość rozwiązań.

Algorytmy poszukiwań lokalnych

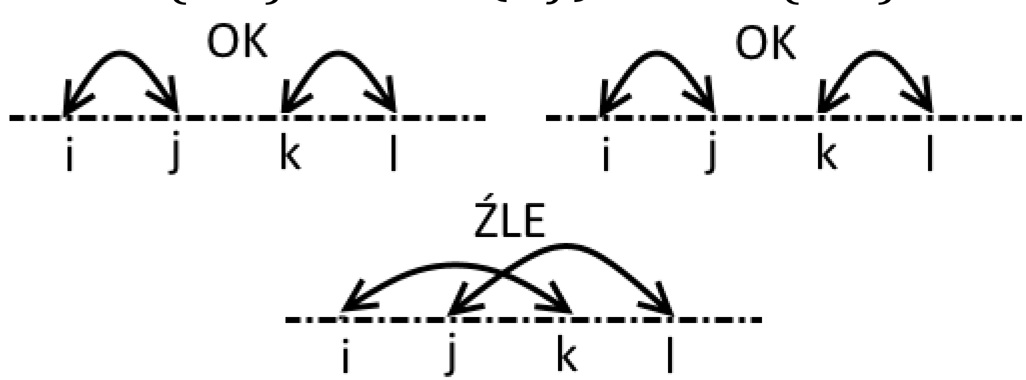
- Otoczenie jest generowane poprzez ruchy – przekształcenia pojedynczych rozwiązań.
- Mogą to być np. ruchy typu „zamień”, które zamieniają pozycjami dwa elementy w permutacji.
- Do tradycyjnych algorytmów poszukiwań lokalnych należą: *first-improve* i *best-improve*.
- Generują otoczenie rozmiaru $\binom{n}{2} = \frac{n(n-1)}{2}$ - wszystkie permutacje powstałe po wykonaniu pojedynczych ruchów. **(Rozmiar wielomianowy).**
- W *first-improve* bieżące rozwiązanie jest zastępowane przez pierwsze lepsze rozwiązanie z otoczenia, a w *best-improve* wybierane jest najlepsze rozwiązanie.
- Główną wadą tych algorytmów jest krótkowzroczność. Poszukiwanie może prowadzić do lokalnego optimum będącego znacznie gorszym od globalnego.
- Zalety to lepsza wydajność niż reszta algorytmów i prostsza implementacja. **Złożoność obliczeniowa jest wielomianowa** proporcjonalnie do rozmiaru otoczenia.

Poprawienie jakości rozwiązania – **„dynasearch”**

- W przeciwieństwie do tradycyjnych algorytmów stosujemy otoczenie **wykładniczego rozmiaru** $2^{\mathrm{n}-1} - 1$.
- Elementy otoczenia to permutacje powstałe po wykonaniu **serii** *niezależnych* ruchów typu „zamień”.
- Algorytm dynamiczny pozwala znaleźć najlepsze rozwiązanie z otoczenia w czasie **wielomianowym**.
- Można zastosować kryteria eliminacyjne pozwalające przyspieszyć obliczenia. (Relacja częściowego porządku omówiona później).

Niezależne ruchy

- Notacja
  - $\pi = (\pi(1), \pi(2), \dots, \pi(n-1), \pi(n))$.
  - $\pi_l^k = (\pi(1), \dots, \pi(k-1), \pi(l), \pi(k+1), \dots, \pi(l-1), \pi(k), \pi(l+1) \dots, \pi(n))$ – ruch tworzący permutację powstałą przez zamienienie miejscami π(k) oraz π(l).
  - Otoczenie $DM(\pi)$ zawiera wszystkie permutacje powstałe z π przez wykonanie serii parami niezależnych ruchów.
- Ruchy π_l^k, π_l^k nazywamy niezależnymi, jeżeli $\max\{i,j\} < \min\{k,l\}$ lub $\min\{i,j\} > \max\{k,l\}$.



Algorytm dynamiczny - teoria

- **Algorytm wyznacza serię ruchów niezależnych generujących z permutacji $\boldsymbol{\pi}$ najlepszy element otoczenia $\boldsymbol{DM(\pi)}$** - czyli nie przeglądamy $2^{n-1}-1$ permutacji, tylko wyznaczamy najlepszą w czasie $O(n^3)$ i pamięci $O(n)$.
- Przez π_j oznaczamy permutację częściową zadań ze zbioru $\{\pi(1), \pi(2), \dots, \pi(j)\}$, która ma minimalną wartość funkcji $F(\pi)$ celu dla tego zbioru.
- π_j można otrzymać z π_i $(0 \leq i < j)$.
- Jeśli i = j – 1, to $\pi_j = (\pi_{j-1}, \pi(j))$ oraz $F(\pi_j) = F(\pi_{j-1}) + f_{\pi(j)}(C_{\pi(j)})$.
- Dla $0 \leq i < j-1$ uzupełniamy π_i o elementy i+1, i+2,…, j-1, j, oraz zamieniamy element i+1 z j.
  Mamy $F(\pi_j) = F(\pi_i) + f_{\pi(j)}(C_{\pi(i)} + p_{\pi(j)})$
  $+ \sum_{k=i+2}^{j-1} f_{\pi(k)}(C_{\pi(k)} + p_{\pi(j)} - p_{\pi(i+1)}) + f_{\pi(i+1)}(C_{\pi(j)})$.

Algorytm dynamiczny

- Dla j = 2,3,…n z warunkami początkowymi $F(\pi_0) = 0$ oraz $F(\pi_1) = w_{\pi(1)}U_{\pi(1)}$ obliczamy wartości

$$F(\pi_j) = min \begin{cases} F(\pi_{j-1}) + f_{\pi(j)}(C_{\pi(j)}) \\ \min\limits_{0 \leq i \leq j-1} \{F(\pi_i) + f_{\pi(j)}(C_{\pi(i)} + p_{\pi(j)}) + \sum_{k=i+2}^{j-1} f_{\pi(k)}(C_{\pi(k)} + p_{\pi(j)} - p_{\pi(i+1)}) \\ \quad + f_{\pi(i+1)}(C_{\pi(j)})\} \end{cases}$$

- Otrzymujemy optymalną wartość funkcji celu $F(\pi_n)$, a odpowiadającą jej permutację można wyznaczyć przeglądając proces obliczania „wstecz”.

Ulepszenie

- Na zbiór zadań N można nałożyć relację częściowego porządku R.
- Zadanie i jest w relacji z zadaniem j wtedy i tylko wtedy, gdy istnieje permutacja optymalna, w której zadanie i poprzedza j.
- Teraz możemy nałożyć na generowane permutacje kryterium eliminacyjne – każde dwa zadania muszą być ze sobą w relacji R.
- Pozwala to zredukować ilość obliczeń drugiego członu ze wzoru rekurencyjnego na $F(\pi_j)$ - po dołączeniu do podpermutacji zadania $\pi(j)$ jest ono zamieniane z $\pi(i+1)$ tylko wówczas, gdy po zamianie jest spełniona relacja R.
- Wniosek: relacja R pozwala na eliminację wielu elementów z otoczenia, bez utraty rozwiązań optymalnych.

Twierdzenie

- Niech Γ^{+} i Γ^{-} będą odpowiednio zbiorami następników oraz poprzedników zadania i w relacji R.
- Jeżeli dla pary zadań $i, j \in N$ spełniony jest jeden z warunków
  - $p_i \leq p_j, w_i \geq w_j, d_i \leq d_j$
  - $p_i \leq p_j, w_i \geq w_j, d_i \leq \sum_{l \in \Gamma_j^{-}} p_l + p_j$
  - $w_i \geq w_j, d_i \leq d_j, d_i \geq \sum_{l \in \Gamma_j^{+}} p_l + p_j$

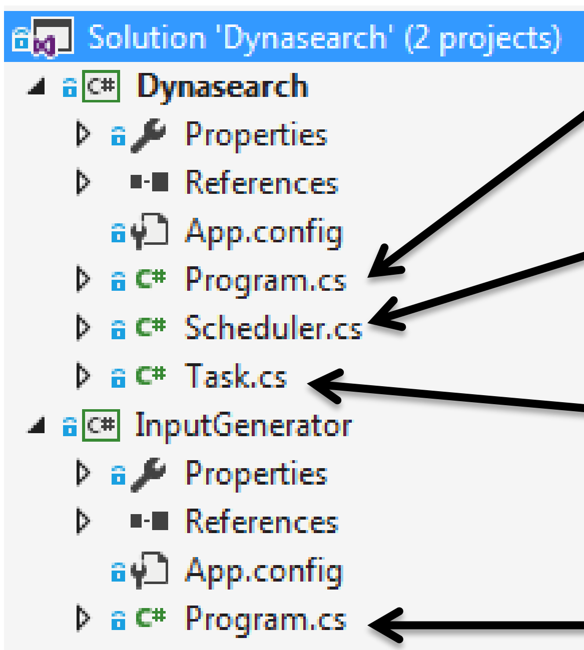
to istnieje permutacja optymalna, w której zadanie i poprzedza j.

Recepta na program „zstępujące poszukiwanie z otoczeniem dynasearch”

1. Obliczenia rozpoczynamy od pewnego rozwiązania startowego $\pi^{(0)}$ wyznaczonego losowo lub za pomocą alg. konstrukcyjnego.
2. $\pi \leftarrow \pi^{(0)}$
3. Niech $\beta \leftarrow$ wynik alg. dynamicznego z początkową perm. π.
4. Jeżeli $F(\beta) < F(\pi)$ to $\pi \leftarrow \beta$ i skocz do 2.
5. wpp zwróć π

- Algorytm uruchamiamy wielokrotnie z rozwiązań otrzymywanych przez losowe zaburzenie bieżącego minimum lokalnego.

Implementacja



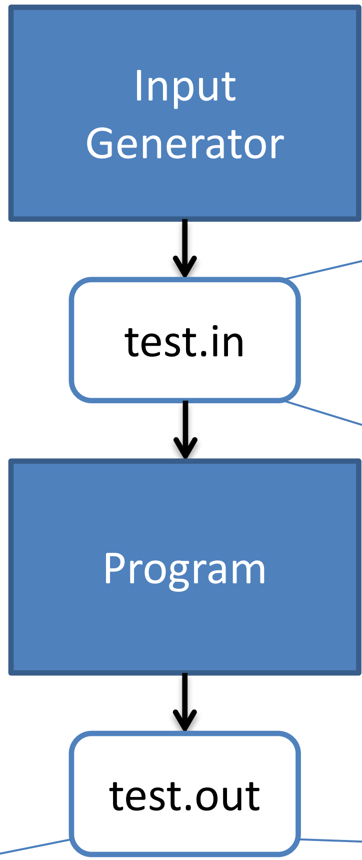
Program, który wyznacza optymalne szeregowanie dla zadanego wejścia: ilość iteracji, ilość zadań * krotka (p,w,d)

Implementuje dynasearch.

Definicja struktury trwałej, która reprezentuje szeregowane zadanie. Zawiera pola: period, weight, deadline.

Generator przykładów testowych.

Sposób uruchamiania



$ InputGenerator.exe > test.in
$ 2
$ 3

```
2 (ilość iteracji)
3 (ilość zadań)
25 3 95 (period, weight, deadline)
78 1 62
81 7 77
```

$ Program.exe < test.in > test.out

```
|          Tasks| Starting cost|Optimized cost|    Improvement|    Iterations|     Time (ms)|
|              3|            11|             8|         27.27%|             2|             4|

Optimal pi: 1, 2, 3
```

Taki koszt miało szeregowanie wyznaczone z losowej permutacji.

Generator przykładów testowych

- Poszczególne wartości krotek są losowane w podanych przedziałach:
  - p: [1, 100]
  - w: [1, 10]
  - d: [P/10, 7*P/10], gdzie P to suma wszystkich p.

```
Random rnd = new Random();

int iterMax = int.Parse(System.Console.ReadLine());
int itemCount = int.Parse(System.Console.ReadLine());

int[] periods = Enumerable.Range(1, itemCount).Select(x => rnd.Next(1, 100)).ToArray();
int[] weights = Enumerable.Range(1, itemCount).Select(x => rnd.Next(1, 10)).ToArray();
int P = periods.Sum();
int[] deadlines = Enumerable.Range(1, itemCount).Select(x => rnd.Next(P / 10, 7*P/10 )).ToArray();

Console.WriteLine(iterMax);
Console.WriteLine(itemCount);
for (int i = 0; i < itemCount; ++i)
{
    Console.WriteLine("{0} {1} {2}", periods[i], weights[i], deadlines[i]);
}
```

Struktura Task

- Bardzo prosta. Zawiera pola: Period, Weight, Deadline.
- Nie pozwala na modyfikowanie pól po inicjalizacji.

```
struct Task
{
    private readonly int period;
    private readonly int weight;
    private readonly int deadline;

    public int Period { get { return period; } }
    public int Weight { get { return weight; } }
    public int Deadline { get { return deadline; } }

    public Task(int period, int weight, int deadline)
    {
        this.period = period; this.weight = weight; this.deadline = deadline;
    }

    public override string ToString()
    {
        return string.Format("{0},{1},{2}", Period, Weight, Deadline);
    }
}
```

Scheduler

- Konstruktor przyjmuje zestaw zadań do uszeregowania.
- Klasa udostępnia metodę *FindLocalMinimum(int[] pi)*, która znajduje najoptymalniejsze szeregowanie w otoczeniu Dynasearch.

```
public Tuple<int[], int> FindLocalMinimum(int[] pi)
{
    this.pi = (int[])pi.Clone();

    DpInit();
    DpAlgorithm();
    DpCreatePi();

    return new Tuple<int[], int>(this.pi, dp[n]);
}
```

Zeruje tablice pomocnicze i wykonuje preprocessing obliczający czasy zakończenia wykonywania zadań.

Implementacja algorytmu dynamicznego. Tworzy tablicę kosztów (dp) i pszesunięć (dpSwaps).

Opt. wart. znajduje się na końcu tablicy.

Odtwarza optymalną permutację na podstawie dpSwaps.

Algorytm dynamiczny

Zasada działania została wyjaśniona na poprzednich slajdach.

```
private void DpAlgorithm()
{
    for (int j = 2; j <= n; ++j)
    {
        int min = dp[j - 1] + Cost(Pi(j), completionTimes[j]);
        int swap = j;

        for (int i = 0; i <= j - 2; ++i)
        {
            int val = dp[i] + Cost(Pi(j), completionTimes[i] + Period(Pi(j)))
                // sum for k from i+2 to j-1 (there should be total j-i-2 elements in the sum)
                + Enumerable.Range(i + 2, j - i - 2).Sum(k => Cost(Pi(k), completionTimes[k] + Period(Pi(j)) - Period(Pi(i + 1))))
                + Cost(Pi(i + 1), completionTimes[j]);

            if(val < min)
            {
                min = val;
                swap = i + 1;
            }
        }

        dp[j] = min;
        dpSwaps[j] = swap;
    }
}
```

Odtworzenie optymalnej permutacji

```
private void DpCreatePi()
{
    for (int i = n; i >= 1; --i)
    {
        if (dpSwaps[i] != i)
        {
            int tmp = pi[i - 1];
            pi[i - 1] = pi[dpSwaps[i] - 1];
            pi[dpSwaps[i] - 1] = tmp;
            i = dpSwaps[i];
        }
    }
}
```

Program testowy

- Program przyjmuje wejście, tworzy instancję Scheduler’a, a następnie wywołuje metodę *FindLocalMinimum(int[] pi)* dopóki jest w stanie poprawić wynik.

```
int  min    = int.MaxValue;
bool repeat = true;

while (repeat)
{
    var beta = scheduler.FindLocalMinimum(piCurrent);
    if (beta.Item2 >= min)
    {
        repeat = false;
    }
    else
    {
        min = beta.Item2;
        piCurrent = beta.Item1;
    }
}
```

- Aby zwiększyć szanse na optymalizację wyniku procedura jest powtarzana dla kilku losowych, startowych permutacji.

```
int[]  piCurrent    = randomPi(itemCount);
```

Wyniki obliczeniowe

- Rozwiązania startowe zostały wyznaczone losowo.
- Poniższa tabelka przedstawia średnią poprawę rozwiązań algorytmu losowego.

Liczba zadań n	Algorytm dynasearch		
	Poprawa	Liczba iteracji	Czas [ms]
40	79,41%	5	20
50	79.03%	5	49
100	76.17%	3	328
średnio	78,20%		

Literatura

- W. Bożejko, M. Wodecki, **Algorytmy lokalnych poszukiwań z otoczeniami o wykładniczej liczbie elementów**, *Komputerowo Zintegrowane Zarządzanie,* WNT, ISBN 83-204-3080-1, Warszawa (2005), 130-137.
- Richard K. Congram, Chris N. Potts, and Steef L. van de Velde. 2002. **An Iterated Dynasearch Algorithm for the Single-Machine Total Weighted Tardiness Scheduling Problem**. *INFORMS J. on Computing* 14, 1 (January 2002), 52-67. DOI=http://dx.doi.org/10.1287/ijoc.14.1.52.7712