Tomasz Jankowiak 249006 7 IV 2021

Sterowanie procesami dyskretnymi

Laboratorium 2

$Algorytm\ PD\ dla\ 1||\sum(w_i T_i)$

Prowadzący:	Dr inż. Mariusz Makuchowski
Termin zajęć:	Piątek, 11.15
Numer grupy laboratoryjnej:	E12-30a

1. Opis problemu

Jednomaszynowy problemem minimalizacji kosztu spóźnienia (w_iT_i) dotyczy szeregowania zadań produkcyjnych na pojedynczej maszynie. Mamy do wykonania *n* zadań, przy czym jednocześnie może być wykonywane co najwyżej jedno zadanie, musi być ono też wykonywane nieprzerwanie przez p_i czasu. Jeśli zadanie jest spóźnione (czas jego zakończenia jest większy od planowanego), naliczana jest kara. Chcemy zminimalizować sumę kar. Dla każdego zadania *i* mamy dane następujące parametry:

- p_i – czas wykonywania na maszynie,
- w_i – waga spóźnienia,
- d_i – pożądany czas zakończenia wykonywania zadania.

Szukamy takiego uszeregowania, aby zminimalizować sumę ważonych spóźnień:

$$\sum_{i=1}^{n} w_i T_i$$

, gdzie *T* oznacza spóźnienie:

$$T_i = \begin{cases} 0 & dla\ \ C_i \leq d_i \\ C_i - d_i & dla\ \ C_i > d_i \end{cases}$$

, gdzie C_i oznacza czas zakończenia i-tego zadania.

2. Opis algorytmu

Opisany wyżej problem jest problemem NP-trudnym. Stąd do jego rozwiązywania zaproponowano użycie wykładniczego schematu programowania dynamicznego (PD). Złożoność obliczeniowa algorytmu to $O(n2^n)$, a pamięciowa $O(2^n)$. Można podać bardziej oszczędną implementację algorytmu wymagającą tylko $\binom{n}{a}, a = [n/2],$ komórek pamięci na przechowywanie częściowych wyników, jednakże prowadzi to do znacznego skomplikowania struktury programu, nie uzyskując przy tym znaczącej poprawy ogólnych cech numerycznych programu.

Algorytm przeprowadza rekurencję po podzbiorach, wykorzystując fakt, że ostatnie zadanie zawsze kończy się w czasie $C(I) = \sum_{j \epsilon I} p_j$. Każdy podzbiór otrzymuje etykietę w postaci liczby binarnej, która jednoznacznie go identyfikuje. Korzystamy z następującej zależności:

$$F(I) = \min_{k \varepsilon I} \{F(I \setminus \{k\}) + K_k(C(I))\}$$

, gdzie: *I* – zbiór zadań, *F(I)* – kara optymalnego uszeregowania zadań *I*, $K_k(t)$ – kara zadania *k* zakończonego w czasie *t*, *C(I)* – długość uszeregowania zadań *I*.

Przykładowo kara optymalnego uszeregowania dziesięciu zadań jest równa minimalnej wartości z (kara optymalnego uszeregowania każdych dziewięciu zadań + kara dziesiątego zadania, które zostało). Oznacza to, że musimy to policzyć dziewięć razy, a dodatkowo w każdym kroku analogicznie liczyć optymalne kary coraz mniejszych podzbiorów zadań. W programie trzeba pamiętać, żeby przypisać zerową wartość kary dla uszeregowania bez zadań, a następnie liczyć kolejne podzbiory identyfikowane przez liczby binarne (1 – pierwsze zadanie, 10 – drugie zadanie, 11 – pierwsze i drugie, 100 – trzecie, 101 – pierwsze i trzecie, itd.) aż do uzyskania kompletnej liczby zadań, które należało uszeregować. Ponieważ do konstrukcji harmonogramu optymalnego potrzebujemy tylko optymalne uszeregowania poszczególnych podzbiorów zadań, rezygnujemy z pamiętania wszystkich czasów, zachowując jedynie optymalne.

3. Kod programu

```
#include <iostream>
#include <fstream>
//#include <ctime>

using namespace std;
int main ()
{
 //   clock_t start, stop; // do sprawdzenia czasu dzialania algorytmu
 //   float diff,t=0,totaltime; // j.w.
    int n,P[100],W[100],D[100];
    ifstream plik("dane/data.10.txt");
    plik >> n; for(int i=0;i<n;i++) plik >> P[i]>>W[i]>>D[i];
    plik.close();

    /*for(int i=0;i<n;i++)
        cout<<P[i]<<" "<<W[i]<<" "<<D[i]<<endl;*/

//    for(int zz=0;zz<100;zz++){ // wielokrotnie wykonujemy ten sam algorytm dla zmierzenia czasu
//        start = clock(); // poczatek dzialania algorytmu

        int N=1<<n, *F=new int[N]; F[0]=0;

        for(int i=1;i<N;i++)
        {
            int c=0;
            for(int k=0,b=1;k<n;k++,b*=2) if(i&b) c+=P[k];
            F[i]=9999999;
            for(int k=0,b=1;k<n;k++,b*=2) if(i&b){
                F[i] = min(F[i], F[i-b] + W[k]*max(c-D[k],0));
            }
        }

  //      stop = clock(); // koniec dzialania algorytmu
  //      diff = stop - start; // czas dzialania algorytmu (roznica miedzy poczatkiem i koncem)
  //      t=t+diff;


        cout<<F[N-1]<<endl;
  //  }
  //  totaltime = ((float)t)/CLOCKS_PER_SEC; // przeliczenie liczby taktow zegara na sekundy
  //  cout << "Czas dzialania algorytmu: " << totaltime << " sekund.\n";

    cin.get();
    return 0;
}
```

4. Działanie programu

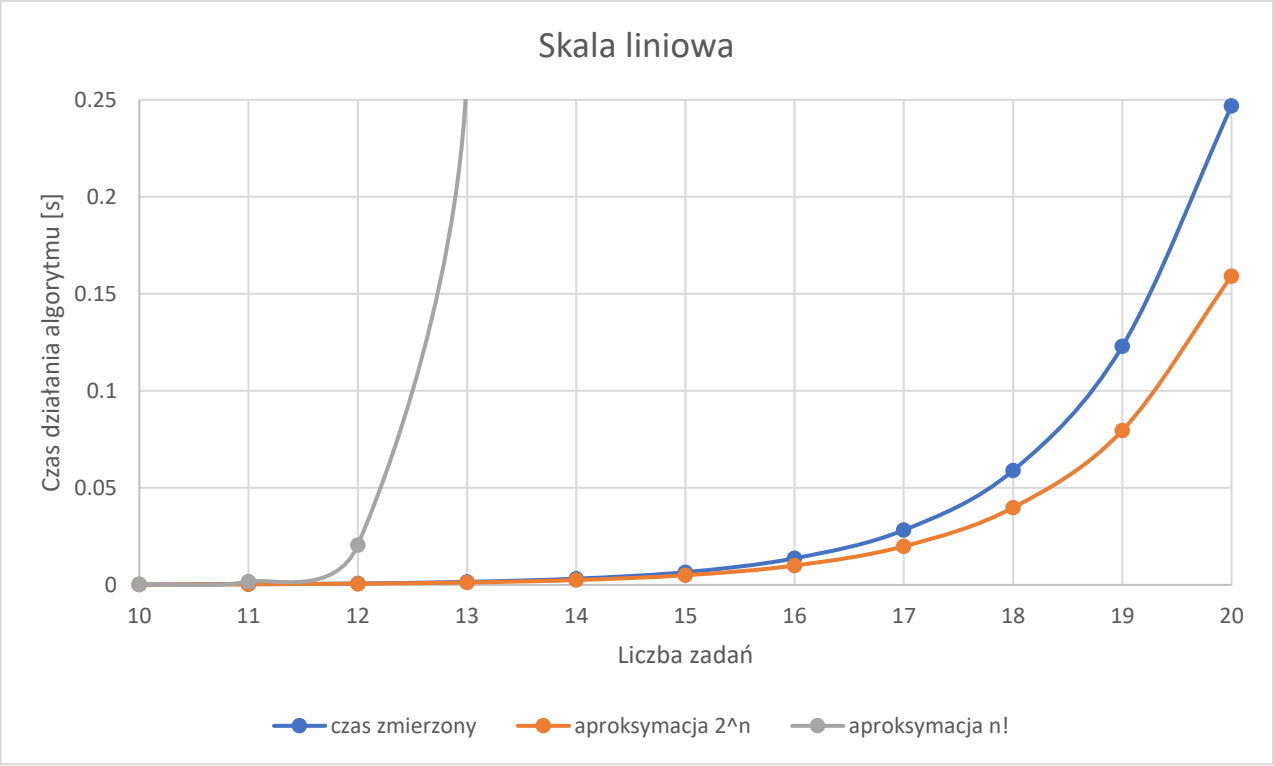
Program wczytuje liczbę zadań oraz parametry poszczególnych zadań z pliku do trzech stuelementowych tablic *P*, *W*, *D*. Format pliku powinien być następujący: pierwsza linia zawiera liczbę naturalną *n* oznaczającą liczbę zadań, a każda z kolejnych n linii zawiera trzy liczby naturalne będące parametrami danego zadania (p. opis problemu). Następnie tworzona jest dynamiczna tablica *F* o rozmiarze 2^n. Jej elementy to długości poszczególnych częściowych uszeregowań zgodnie z zasadą działania algorytmu (p. opis algorytmu). Ostatni element tablicy *F* to ostateczna suma najlepszego uszeregowania wszystkich zadań, więc wypisujemy go na wyjście.

5. Złożoność obliczeniowa

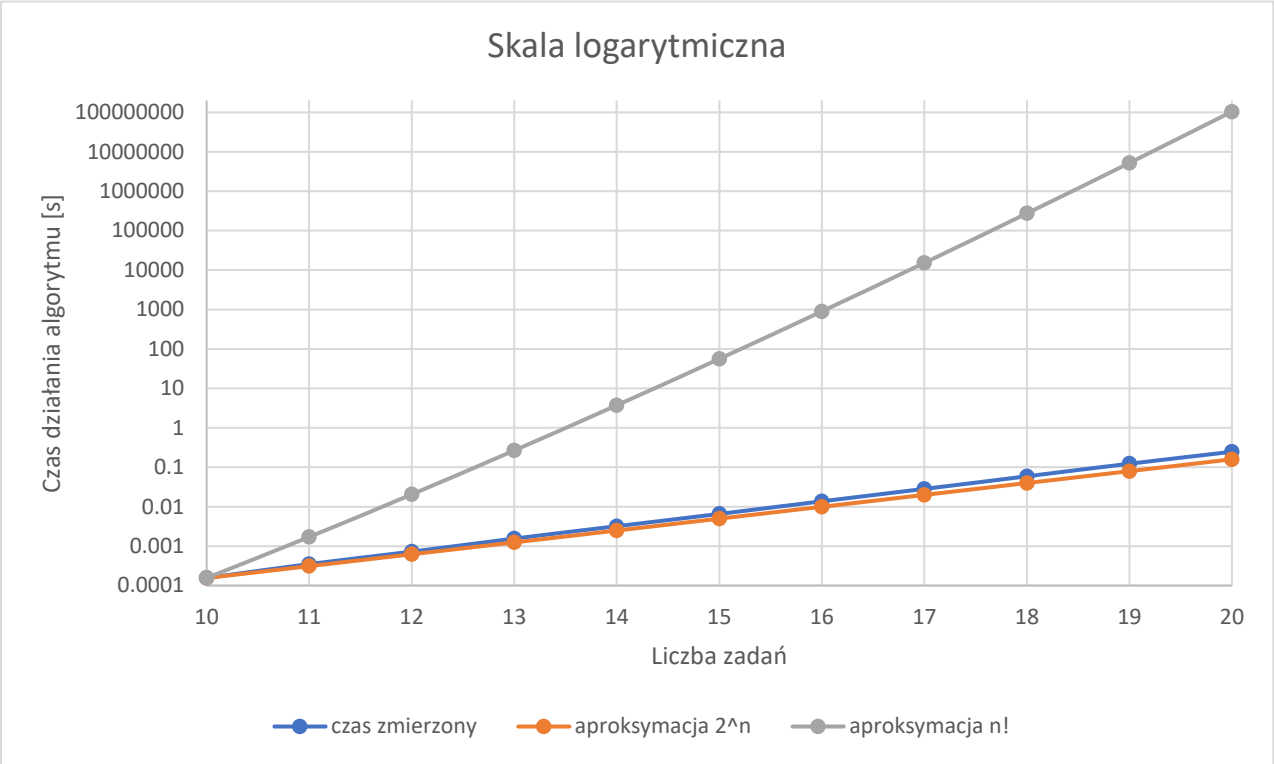
W tabeli przedstawiono zmierzone czasy działania algorytmu dla danych wejściowych z określoną liczbą zadań w przedziale <10,20>. Czas podany jest w sekundach i dotyczy algorytmu zwracającego sumę witi (bez permutacji i wczytywania danych z pliku).

l. zadań	czas zmierzony	2^n/skala	n!/skala	2^n	n!
10	0.0001554	0.0001554	0.0001554	1024	3628800
11	0.0003508	0.0003108	0.0017094	2048	39916800
12	0.000724	0.0006216	0.0205128	4096	4.79E+08
13	0.001553	0.0012432	0.2666664	8192	6.23E+09
14	0.0032	0.0024864	3.7333296	16384	8.72E+10
15	0.00655375	0.0049728	55.999944	32768	1.31E+12
16	0.0137	0.0099456	895.999104	65536	2.09E+13
17	0.02824	0.0198912	15231.98477	131072	3.56E+14
18	0.05892	0.0397824	274175.7258	262144	6.4E+15
19	0.12298	0.0795648	5209338.791	524288	1.22E+17
20	0.2468	0.1591296	104186775.8	1048576	2.43E+18

W celu porównania wykresów 2^n i n! z czasem zmierzonym przeskalowano wartości 2^n i n! odpowiednio przez $2^{10}/czas(10)$ oraz $10!/czas(10)$. Wyniki z tabeli przedstawiono na poniższych wykresach.



Z wykresu wyraźnie widać, że funkcja jest wykładnicza, a zmierzony czas działania algorytmu bardziej odpowiada aproksymacji 2^n. Późniejsza delikatna rozbieżność częściowo może wynikać z minimalnych błędów (zaokrągleń) pomiaru czasu działania algorytmu, ale przede wszystkim należy pamiętać, że dane wejściowe nie były losowe i doświadczenie przeprowadzono tylko na jednych, konkretnych danych dla każdej liczby zadań. W idealnym przypadku dla każdej liczby zadań powinniśmy mieć wiele losowo wygenerowanych danych wejściowych i wyciągnąć średnią czasu. Zależność jest też dobrze widoczna na poniższym wykresie, na którym zastosowano skalę logarytmiczną w celu zobaczenia całego wykresu aproksymacji n!.



6. Podsumowanie i wnioski

Mimo, iż początkowo może się wydawać, że złożoność obliczeniowa będzie bardzo duża - O(n!), okazuje się, że algorytm PD dla $1||\sum(w_i T_i)$ posiada złożoność $O(2^n)$. Jest to bardzo dobra wiadomość, która sprawia, że śmiało możemy go wykorzystywać do jednomaszynowego problemu szeregowania zadań ze znanymi następującymi parametrami i-tego zadania: czasem trwania, wagą spóźnienia, pożądanym czasem zakończenia.

7. Bibliografia

- Smutnicki Cz. – „Algorytmy szeregowania zadań”, Oficyna Wydawnicza PWr, Wrocław 2012
- Sawik T. – „Optymalizacja dyskretna w elastycznych systemach produkcyjnych”, WNT, Warszawa 1992
- http://mariusz.makuchowski.staff.iiar.pwr.wroc.pl/download/courses/sterowanie.procesami.dyskretnymi/lab.instrukcje/lab02.witi/witi.literatura/SPD_WiTi.pdf
- http://andrzej.gnatowski.staff.iiar.pwr.wroc.pl/SterowanieProcesamiDyskretnymi/lab04_witi/instrukcja/lab04.pdf