STEROWANIE PROCESAMI CIĄGŁYMI
Problem RPQ

Prowadzący: dr inż. Paweł Drąg
Autorzy:
Emilia Szymańska 248975
Igor Zieliński 248944

21.03.2021r.

1 Opis problemu

Problem RPQ jest problemem jednomaszynowym, gdzie na wejściu podane są trzy parametry dla każdego zadania:

- r_i – termin dostępności zadania i,
- p_i – czas wykonania zadania i,
- q_i – czas dostarczenia zadania i.

Wyjściem działania algorytmu obsługującego problem RPQ jest permutacja wykonania zadań π oraz moment maksymalny z możliwych terminów dostarczenia zadań C_{max}. Problem jest dość obszerny i jest wiele algorytmów, które go rozwiązują. My zaimplementowaliśmy trzy algorytmy - sortowanie po r oraz dwa własne, by móc porównać różne metody.

2 Zaimplementowane algorytmy

2.1 Sortowanie po r

Algorytm ten jest nieskomplikowany - sprowadza się do posortowania zadań rosnąco względem wartości parametru r_i. W przypadku zadań, które mają taką samą wartość parametru r_i, wybierane jest to, które ma większą wartość parametru q_i.

Algorytm ten - zaimplementowany jak pokazano na rys. 1 - ma złożoność $O(n \cdot logn)$.

```cpp
bool cmp( const task& a, const task& b)
{
    if( a.r == b.r)
        return  a.q >= b.q;
    else
        return a.r < b.r;
}

int main()
{
    std::vector<task> tasks;
    read_data(tasks);
    auto start =std::chrono::high_resolution_clock::now();
    std::sort(tasks.begin(), tasks.end(), cmp);
    auto end = std::chrono::high_resolution_clock::now();

    double time_taken = std::chrono::duration_cast<std::chrono::microseconds>(end - start).count();


    printf("Time: %.3f us\n", time_taken);

    int Cmax = compute_cmax_for_list_of_tasks(tasks);
    printf("Permutation: ");
    for(auto it = tasks.begin(); it != tasks.end(); ++it)
        printf("%d ", it->index);
    printf("\nCmax: %d", Cmax);
    return 0;
}
```

Rysunek 1: Sortowanie po r.

2.2 Algorytm własny z wykorzystaniem elementu stochastycznego

Algorytm ma bardzo proste założenie - sprawdzenie losowych ustawień zadań i policzenie dla tych przypadków C_{max}. Sposób postępowania jest następujący:

1. oblicz C_{max} dla pierwotnego uszeregowania,
2. policz ilość podmian jako number_of_swaps $= 100 \cdot n^2$,
3. dla liczby number_of_swaps:
   (a) wylosuj dwie liczby,
   (b) odpowiadające im zadania zamień ze sobą kolejnością,
   (c) oblicz C_{max} dla tego uszeregowania,
   (d) jeśli to uszeregowanie jest lepsze niż poprzednie, to przypisz pod C_{max} nową wartość; w przeciwnym wypadku przywróć poprzednie uszeregowanie.

Wbrew pozorom, powyższy algorytm (którego implementację można zobaczyć na rys. 2) całkiem dobrze sobie radzi ze znajdowaniem dobrego lub nawet optymalnego rozwiązania, jednak ma jedną, istotną wadę - jego przybliżona złożoność obliczeniowa wynosi $100n^3 \approx O(n^3)$.

```
srand(time(NULL));
int number_of_swaps = 100 * perm.size() * perm.size();
for(int i = 0; i < number_of_swaps; i++)
{
    int a, b;

    a = rand()%perm.size();
    b = rand()%perm.size();

    std::swap(perm[a], perm[b]);
    new_cmax = compute_cmax_for_permutation(perm, r, p, q);

    if( new_cmax < curr_cmax)
        curr_cmax = new_cmax;
    else
        std::swap(perm[a], perm[b]);
}
```

Rysunek 2: Algorytm własny z wykorzystaniem elementu stochastycznego.

2.3 Algorytm własny z wykorzystaniem podmian elementów oddalonych o k

Algorytm jest podobny do algorytmu przedstawionego powyżej, jednak różni się wyznaczaniem kolejności podmian. Lista kroków wygląda następująco:

1. oblicz C_{max} dla pierwotnego uszeregowania,
2. dla każdego k od 1 do ilość_zadań-1:
   (a) dla każdego i od 0 do ilość_zadań-k,
      i. podmień zadanie i-te z zadaniem i+k-tym
      ii. jeśli to uszeregowanie jest lepsze niż poprzednie, to przypisz pod C_{max} nową wartość; w przeciwnym wypadku przywróć poprzednie uszeregowanie.

Implementację algorytmu można zobaczyć na rys. 3. W porównaniu do algorytmu poprzedniego eliminujemy element losowości, jednak przybliżona złożoność obliczeniowa programu wynosi $5n^4 \approx O(n^4)$.

```
for(int k = 1; k <= perm.size() - 1; ++k)
{
    for(int n = 0; n < perm.size() - k; ++n)
    {
        for(int i = 0; i < perm.size() - k; ++i)
        {
            std::swap(perm[i], perm[i + k]);
            new_cmax = compute_cmax_for_permutation(perm, r, p, q);
            if( new_cmax < curr_cmax)
                curr_cmax = new_cmax;
            else
                std::swap(perm[i], perm[i + k]);
        }
    }
}
```

Rysunek 3: Algorytm własny z wykorzystaniem podmian elementów oddalonych o k.

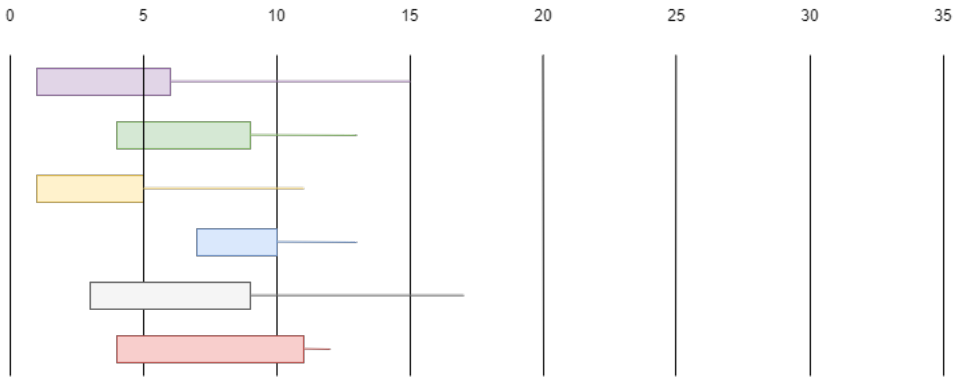
3 Wyniki i ich porównanie

Dane testowe pobrane były ze strony `http://new.zsd.iiar.pwr.edu.pl/educ.php?lid=132&zid=SCHRAGE`.

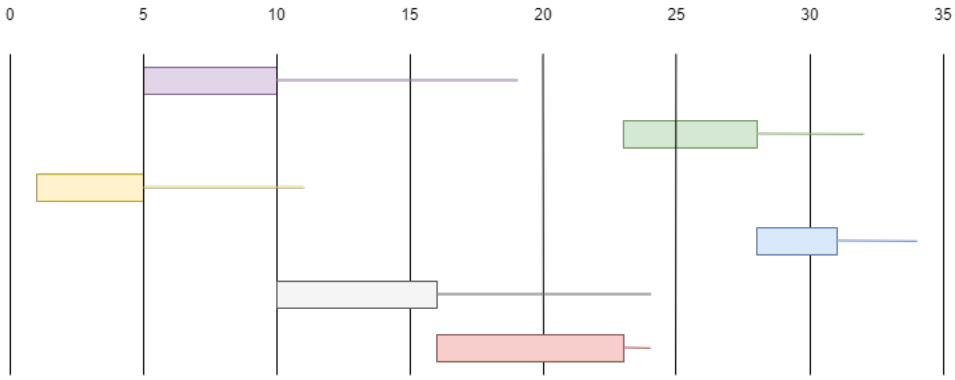
3.1 Przykładowe działanie algorytmów

Na podstawie pierwszego zestawu testowego zostanie przedstawione działanie algorytmów opisanych w sekcji 2. Dane tekstowe przedstawiają się wówczas następująco:
6
1 5 9
4 5 4
1 4 6
7 3 3
3 6 8
4 7 1

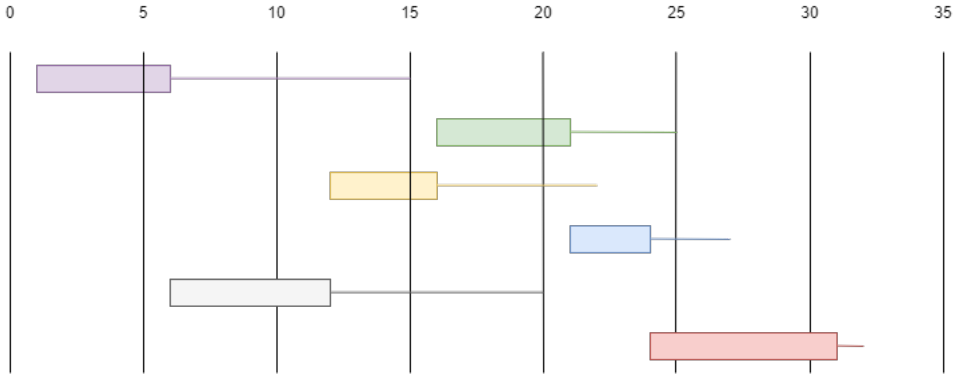
Pierwsza liczba oznacza liczbę zadań, które mamy uszeregować. Kolejne parametry dla każdego zadania odpowiadają wartościom r_i, p_i, q_i objaśnione w opisie problemu na początku sprawozdania. Dane obrazują sytuację zaprezentowane na rysunku 4. Po przepuszczeniu przez sortowanie po r, otrzymujemy poszeregowanie jak na rys. 5 odpowiadające kolejności zadań: 3, 1, 5, 6, 2, 4 oraz wartości $C_{max} = 34$. Przy uruchomieniu algorytmów własnych otrzymujemy rezultat jak na rys. 6 odpowiadający kolejności zadań: 1, 5, 3, 2, 4, 6 oraz wartości $C_{max} = 32$.



Rysunek 4: Nieposzeregowane zadania z pierwszego zbioru danych.



Rysunek 5: Uszeregowane zadania z pierwszego zbioru danych według sortowania po r.



Rysunek 6: Uszeregowane zadania z pierwszego zbioru danych według algorytmów własnych.

3.2 Wyniki na danych testowych

W tabelach 1, 2, 3 przedstawione zostały wartości otrzymanych C_{max} dla poszczególnych zestawów danych, a w tabeli 4 oraz na wykresach 7, 8 porównano czasy wykonywania się algorytmów (z dokładnością mikrosekund). Dla własnych algorytmów wywołanie programów z tysiącem zadań wiązałoby się ze zbyt długim czasem oczekiwania na wynik, dlatego zostało to pominięte.

zestaw testowy	C_{max}
1	34
2	746
3	1594
4	1576
5	4013
6	4296
7	7831
8	7973
9	86648

Tabela 1: Algorytm sortowania po r.

zestaw testowy	C_{max}
1	32
2	641
3	1267
4	1386
5	3472
6	3617
7	6922
8	7420
9	-

Tabela 2: Algorytm własny z wykorzystaniem elementu stochastycznego.

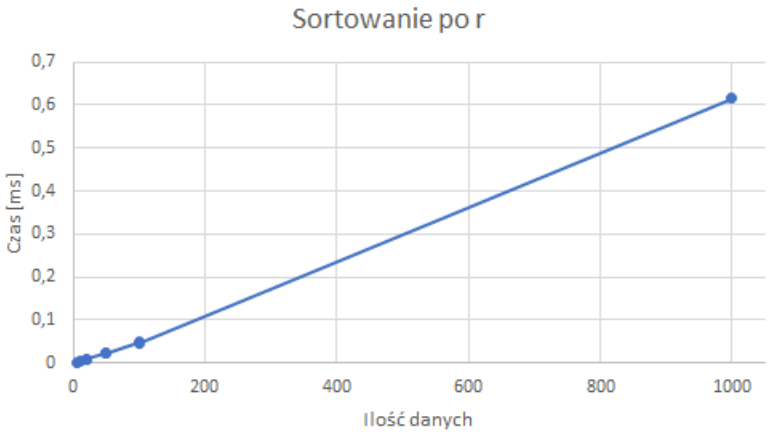
zestaw testowy	C_{max}
1	32
2	675
3	1324
4	1386
5	3585
6	3807
7	7105
8	7748
9	-

Tabela 3: Algorytm własny z wykorzystaniem podmian elementów oddalonych o k.

zestaw danych	ilość zadań w zestawie	sortowanie po r	alg. własny stoch.	alg. własny k
1	6	0.002	2	0.1
2	10	0.005	9	2
3	20	0.01	49	15
4	20	0.01	55	16
5	50	0.023	797	629
6	50	0.022	693	624
7	100	0.048	5571	9457
8	100	0.046	5726	9462
9	1000	0.616	-	-

Tabela 4: Porównanie czasów wykonywania się algorytmów w milisekundach.

Rysunek 7: Porównanie czasu działania algorytmów. Niebieski wykres odpowiada sortowaniu po r, pomarańczowy i szary własnym algorytmom.



Rysunek 8: Czas działania algorytmu sortowania po r.

3.3 Porównanie z algorytmem Jacksona

Sortowanie po r jest lepszym rozwiązaniem w porównaniu do algorytmu Jacksona ze względu na złożoność $O(n\log n)$. Ciężko jednak byłoby porównać oba algorytmy w czystych formach, gdyż w przypadku alg. Jacksona nie był podawany parametr q - należałoby je odpowiednio zmodyfikować, aby wykonywać je z tymi samymi danymi testowymi.

4 Wnioski

- Sortowanie po r okazuje się być najszybszą spośród zaprezentowanych metod, jednak nie zwraca z pewnością wyniku optymalnego.
- Algorytm własny wykorzystujący podmiany elementów oddalonych o k radzi sobie najgorzej spośród własnych algorytmów w zakresie odnajdywania najmniejszego C_{max}. W większości przypadków algorytm własny wykorzystujący losowość dawał najlepsze wyniki, jednak jego czas działania w przypadków większej ilości zadań znacznie utrudnia wykorzystanie tego rozwiązania w praktyce.