Problem jednomaszynowy RPQ

13 kwietnia 2025

Automatyka i Robotyka PWR	SPD	Problem $1\|r_j, q_j\|C_{max}$
Wiktor Kwiatkowski Piotr Siembab	Poniedziałek 09:15	dr inż. Agnieszka Wielgus

Spis treści

1 Opis problemu

Dana jest pojedyncza maszyna oraz zbiór zadań $J = \{1, 2, \ldots, N\}$, gdzie każde zadanie j charakteryzuje się trzema parametrami:

- p_j – czas trwania zadania (czas potrzebny na jego wykonanie)
- r_j – czas dostępności zadania (moment, od którego zadanie może być rozpoczęte)
- q_j – czas stygnięcia zadania (czas potrzebny po zakończeniu zadania, zanim można uznać je za zakończone)

1.1 Formalizacja matematyczna

Celem jest znalezienie permutacji zadań $\pi = (\pi(1), \pi(2), \ldots, \pi(N))$ minimalizującej maksymalny czas zakończenia C_{max}:

$$C_{\max} = \max_{1 \leqslant j \leqslant N} \left(C_{\pi(j)} + q_{\pi(j)} \right) \tag{1}$$

gdzie czas zakończenia $C_{\pi(j)}$ obliczamy jako:

$$C_{\pi(1)} = r_{\pi(1)} + p_{\pi(1)} \tag{2}$$

$$C_{\pi(j)} = \max \left(r_{\pi(j)}, C_{\pi(j-1)} \right) + p_{\pi(j)} \quad \text{dla} \quad j > 1 \tag{3}$$

1.2 Przykład

Dla trzech zadań o parametrach:

Zadanie	p_j	r_j	q_j
1	2	0	3
2	1	1	2
3	3	2	1

Optymalna kolejność $\pi = (1, 2, 3)$ daje:

$$\begin{aligned}
C_{\pi(1)} &= 0 + 2 = 2 \\
C_{\pi(2)} &= \max(1, 2) + 1 = 3 \\
C_{\pi(3)} &= \max(2, 3) + 3 = 6 \\
C_{\max} &= \max(2 + 3, 3 + 2, 6 + 1) = 7
\end{aligned}$$

1.3 Własności problemu

- Problem jest **NP-trudny** – dokładne rozwiązania możliwe tylko dla małych instancji
- W praktyce stosuje się algorytmy przybliżone i heurystyki
- Typowe zastosowania: systemy produkcyjne, przetwarzanie wsadowe, systemy czasu rzeczywistego

2 Algorytmy

2.1 Sortowanie po r

Idea algorytmu: Zadania są wykonywane w kolejności rosnących czasów dostępności r_j. Prosta heurystyka, ale często dająca suboptymalne wyniki.

Pseudokod:

```
function SortR(zadania)
    perm ← kopia listy zadania
    Posortuj perm rosnąco względem r
    C ← 0
    actual_time ← 0
    for all zadanie ∈ perm do
        actual_time ← max(actual_time, zadanie.r) + zadanie.p
        C ← max(C, actual_time + zadanie.q)
    end for
    return (C, czas działania w nanosekundach)
end function
```

2.2 Sortowanie po q

Idea algorytmu: Zadania są wykonywane w kolejności malejących czasów dostarczenia q_j. Lepsza jakość rozwiązań niż przy sortowaniu po r_j.

Pseudokod:

```
function SortQ(zadania)
    perm ← kopia listy zadania
    Posortuj perm malejąco względem q
    C ← 0
    actual_time ← 0
    for all zadanie ∈ perm do
        actual_time ← max(actual_time, zadanie.r) + zadanie.p
        C ← max(C, actual_time + zadanie.q)
    end for
    return (C, czas działania w nanosekundach)
end function
```

2.3 Przegląd zupełny

Idea algorytmu: Przegląd wszystkich możliwych permutacji zadań i wybór tej z minimalnym C_{max}. Gwarantuje znalezienie optymalnego rozwiązania, ale ma złożoność $O(N!)$.
Pseudokod:

```
function BRUTE(zadania)
    minC ← ∞, bestOrder ← pusta lista, perm ← kopia listy zadania
    Posortuj perm po indeksie j
    repeat
        C ← 0, actual_time ← 0
        for all zadanie ∈ perm do
            actual_time ← max(actual_time, zadanie.r) + zadanie.p
            C ← max(C, actual_time + zadanie.q)
        end for
        if C < minC then
            minC ← C
            bestOrder ← perm
        end if
    until nie ma więcej permutacji perm (używając next_permutation)
    return (minC, czas działania w milisekundach)
end function
```

2.4 Algorytm Schrage

Idea algorytmu: Algorytm wykorzystujący dwie kolejki: - N - zadania niegotowe (sortowane po r_j) - G - zadania gotowe (sortowane po q_j) W każdej iteracji wybiera zadanie o najwyższym q_j spośród dostępnych. **Pseudokod**:

```
function SCHRAGE(zadania)
    Sortowanie wstępne kolejki priorytetowej N według r (min-heap)
    Sortowanie kolejki priorytetowej G według q (max-heap)    ▷ Początkowo pusta
    t ← 0, Cmax ← 0
    while N ≠ ∅ lub G ≠ ∅ do
        while N ≠ ∅ i r ⩽ t do
            Dodaj zadanie z N do G
            Usuń zadanie z N
        end while
        if G = ∅ then
            t ← przeskocz do czasu dostępności pierwszego zadania w N
        else
            Pobierz zadanie z szczytu G
            Zaktualizuj czas t ← t + p
            Cmax ← max(Cmax, t + q)
        end if
    end while
    return (Cmax, czas)
end function
```

2.5 Algorytm Schrage z podziałem

Idea algorytmu: Rozszerzenie algorytmu Schrage pozwalające na tymczasowe przerwanie zadania, gdy pojawi się zadanie o wyższym priorytecie (q_j).

Pseudokod:

```
function SCHRAGE Z PODZIALEM(zadania)
    Sortowanie wstępne kolejki priorytetowej N według r (min-heap)
    Sortowanie kolejki priorytetowej G według q (max-heap)
    t ← 0, Cmax ← 0
    aktualneZadanie.p ← 0, aktualneZadanie.q ← ∞
    while N ≠ ∅ lub G ≠ ∅ do
        while N ≠ ∅ i r ⩽ t do
            j ← zadanie z wierzchołka N
            Usuń j z N, dodaj do G
            if j.q > aktualneZadanie.q then
                aktualneZadanie.p ← t − j.r
                t ← j.r
                if aktualneZadanie.p > 0 then
                    Dodaj aktualneZadanie z powrotem do G
                end if
            end if
        end while
        if G = ∅ then
            t ← czas dostępności pierwszego zadania w N
        else
            j ← zadanie z wierzchołka G
            Usuń j z G
            aktualneZadanie ← j
            t ← t + j.p
            Cmax ← max(Cmax, t + j.q)
        end if
    end while
    return (Cmax, czas)
end function
```

2.6 Algorytm własny

Idea algorytmu: Algorytm priorytetowy z wagami dla parametrów zadań. Priorytet obliczany jako kombinacja liniowa:

$$\text{Priority}(j) = w_r \cdot r_j + w_q \cdot q_j + w_p \cdot p_j$$

gdzie $w_r = 0.3$, $w_q = 0.7$, $w_p = -0.2$.

Pseudokod:

```
function WLASNY(zadania)
    Ustal wagi: w_r ← 0.3, w_q ← 0.7, w_p ← −0.2
    R ← zadania                                  ▷ Zadania pozostałe do zaplanowania
    G ← ∅                                        ▷ Zadania gotowe do wykonania
    t ← 0, C_max ← 0
    while R ≠ ∅ do
        G ← ∅
        for all z ∈ R do
            if r_z ⩽ t then
                pri_z ← w_r · r_z + w_q · q_z + w_p · p_z
                Dodaj (pri_z, z) do G
            end if
        end for
        if G = ∅ then
            t ← min_{z∈R} r_z
        else
            Wybierz zadanie z ∈ G z minimalnym pri
            Usuń z z R
            t ← max(t, r_z) + p_z
            C_max ← max(C_max, t + q_z)
        end if
    end while
    return (C_max, czas)
end function
```

3 Przebieg przeprowadzonego eksperymentu

3.1 Specyfikacja komputera

Testy zostały przeprowadzone na komputerze z procesorem Intel Core i5-12500H (12 rdzeni, 16 wątków, 4.5GHz, 18MB cache) oraz 16GB pamięci RAM.

3.2 Wyniki

Wszystkie algorytmy zostały przetestowane dla następujących ilości zadań [6,7,8,9,10,11,12,20,50]. Ze względu na charakterystykę czasową algorytm Przeglądu zupełnego nie został wywołany dla przypadków z 20 oraz 50 zadaniami.

3.2.1 Pojedyncza instancja

W poniższej tabeli zebrane zostały wyniki C_{max} oraz czasy działania wszystkich algorytmów dla pojedynczej instacji o danym rozmiarze.

Data size / Algorithm	6	7	8	9	10	11	12	20	50
SortR Cmax [%]	34 [6%]	173 [9%]	249 [2%]	168 [10%]	329 [6%]	232 [12%]	254 [0%]	527 [0%]	1296 [1%]
SortR Time [ns]	548	157	134	159	458	1714	665	743	1040
SortQ Cmax [%]	32 [0%]	158 [0%]	252 [3%]	154 [1%]	326 [5%]	220 [6%]	263 [4%]	528 [1%]	1293 [0%]
SortQ Time [ns]	419	201	204	177	538	1516	416	682	1206
Schrage Cmax [%]	32 [0%]	164 [4%]	245 [0%]	153 [0%]	310 [0%]	213 [2%]	253 [0%]	525 [0%]	1289 [0%]
Schrage Time [ns]	1299	838	1137	1297	1612	2024	1505	2199	4900
SchrageDiv Cmax [%]	32 [-]	157 [-]	245 [-]	153 [-]	310 [-]	205 [-]	253 [-]	525 [-]	1289 [-]
SchrageDiv Time [ns]	691	833	912	1061	1293	2194	1278	1780	4056
BruteSearch Cmax [%]	32 [0%]	158 [0%]	245 [0%]	153 [0%]	310 [0%]	208 [0%]	253 [0%]	-	-
BruteSearch Time [ms]	0	0	0	5	44	369	4241	-	-
Custom Cmax [%]	40 [25%]	182 [15%]	268 [9%]	177 [16%]	327 [5%]	232 [12%]	280 [11%]	548 [4%]	1314 [2%]
Custom Time [ns]	1466	1284	1409	11293	3884	2418	2721	2564	9757

Tabela 1: Porównanie algorytmów harmonogramowania dla różnych rozmiarów danych

3.2.2 Średnie wyniki dla większej ilości instancji

W tabeli przedstawione zostały średnie wartości czasów i błędów dla 50 losowo wygenerowanych instancji danego rozmiaru.

Data size / Algorithm	6	7	8	9	10	11	12	20	50
AvgSortR Time [ns]	173.68	131.2	189.02	185.48	391.04	501.76	543.48	393.3	1111.46
AvgSortR Error [%]	9.92	14.85	13.53	13.62	13.6	11.84	10.15	5.65	1.95
AvgSortQ Time [ns]	133.12	116.86	183.98	152.66	300.72	412.44	455.4	378.6	1139.22
AvgSortQ Error [%]	13.1	18.1	14.88	14.17	14.86	8.33	7.78	4.66	1.55
AvgSchrage Time [ns]	595.42	521.52	794.94	732.42	1094.58	1515.82	1526.62	1492.04	4045.68
AvgSchrage Error [%]	3.38	2.19	1.13	0.35	0.15	0.08	0.4	0	0
AvgSchrageDiv Time [ns]	543.68	509.64	719.04	684.98	970.88	1268.5	1312.14	1367.16	3697.12
AvgSchrageDiv Error [%]	-	-	-	-	-	-	-	-	-
AvgBrute Time [ms]	0	0	0	2.5	28.54	325.78	4321.54	-	-
AvgBrute Error [%]	0	0	0	0	0	0	0	-	-
AvgCustom Time [ns]	744.68	661.24	1082.1	1947.5	1787.28	2829.18	3100.2	2061.5	8806.78
AvgCustom Error [%]	13.85	24.04	22.79	22.68	20.89	20	18.11	10.16	3.46

Tabela 2: Średnie czasy i błędy algorytmów dla różnych rozmiarów danych. Wszystkie błędy procentowe są liczone względem BruteSearch (6-12) lub Schrage (20,50).

Data size / Algorithm	6	7	8	9	10	11	12	20	50
AvgSortR Error [%]	43.4	33.78	32	32.58	28.4	22.54	23.62	10.97	4.12
AvgSortQ Error [%]	35	46.15	32.91	33.33	30.11	17.15	17.15	10.28	3.38
AvgSchrage Error [%]	28.57	16.67	16	6.35	3.7	2.01	7.02	0	0
AvgSchrageDiv Error [%]	-	-	-	-	-	-	-	-	-
AvgBrute Error [%]	0	0	0	0	0	0	0	-	-
AvgCustom Error [%]	48.98	42.59	39.66	40	35.53	29.28	29.48	15.01	4.18

Tabela 3: Maksymalne błędy algorytmów względem BruteSearch (6-12) lub Schrage (20,50) dla różnych rozmiarów danych.

4 Wnioski

- Wybór algorytmu zależy od rozmiaru instancji i dostępnego czasu obliczeniowego.
- Dla małych instancji można stosować algorytm Przeglądu Zupełnego, dla większych – algorytmy heurystyczne lub Schrage

4.1 Algorytm Przeglądu Zupełnego

- Gwarantuje znalezienie optymalnego rozwiązania, ale jego zastosowanie jest ograniczone do małych instancji z powodu wysokiej złożoności obliczeniowej (O(N!)).
- Czas działania rośnie bardzo szybko wraz z liczbą zadań.

4.2 Algorytmy Heurystyczne (Sortowanie po r, Sortowanie po q)

- Są szybkie i proste w implementacji, ale często dają suboptymalne wyniki.
- Sortowanie po q daje lepsze wyniki niż sortowanie po r, co wskazuje na ważną rolę czasu stygnięcia w optymalizacji

4.3 Algorytm Schrage

- Jego wersja z podziałem (SchrageDiv) może poprawić efektywność w niektórych przypadkach.