Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Badania operacyjne i optymalizacja dyskretna

Wtorek, 13:15-15:00

Metoda Ścieżki Krytycznej - CPM

Autorzy:

Aleksandra Rozmus, 252954

Paweł Gołębiowski, 243710

Prowadzący:

dr inż. Mariusz Makuchowski

7 listopada 2023

Politechnika
Wrocławska

Spis treści

1 Wstęp

Metoda ścieżki krytycznej jest jednym z kluczowych narzędzi w zarządzaniu projektami, umożliwiającym skuteczne planowanie, harmonogramowanie i kontrolowanie działań w ramach projektu [1]. Jest to niezwykle ważne narzędzie, które pozwala na identyfikację najważniejszych zadań w projekcie, czyli tych, które mają największy wpływ na termin zakończenia całego przedsięwzięcia. W naszym sprawozdaniu skupimy się na omówieniu dwóch stworzonych przez nas algorytmów wykorzystujących metodę ścieżki krytycznej w kontekście zarządzania projektami [2].

2 Cel ćwiczenia

Celem ćwiczenia jest implementacja dwóch rodzajów algorytmów rozwiązujących problem ścieżki krytycznej, przeprowadzenie analizy porównawczej, badań i ocena ich działania.

3 Dane wejściowe

Jako dane wejściowe wykorzystane przy sprawdzeniu poprawności działania stworzonych algorytmów wykorzystano dane dostępne w pliku *data20.txt* na stronie mariusz.makuchowski.staff.iiar.pwr.wroc.pl.

```
20 40
93 14 53 38 7 68 1 52 4 46 1 67 5 75 6 53 68 22 38 83
1 4  1 5  1 7  1 16  2 9  2 18  2 19  3 5  3 13  3 18  4 12  4 19  6 9  6 12  6 15  7 5  8 7
↪  8 12  8 19  10 3  10 8  10 17  10 18  11 9  11 12  11 18  13 4  13 15  14 12  14 15  15
↪  5  15 16  16 19  17 4  17 5  18 6  18 19  20 7  20 12  20 16
```

Listing 1: Dane wejściowe

gdzie:

- Pierwsza linia zawiera N liczbę zadań i M liczbę połączeń.
- W drugiej linii jest N czasów trwania kolejnych zadań.
- Trzecia linia zawiera M zależności między zadaniami.

4 Zaimplementowane algorytmy

Podczas laboratorium zaimplementowano dwa algorytmy. Dla obu algorytmów dane były wczytywane z pliku i zapisywane do pliku wyjściowego.

4.1 Algorytm nr 1

Pierwszy algorytm wykorzystuje przetwarzanie w przód i przetwarzanie w tył. W pierwszym etapie obliczane są najwcześniejsze możliwe czasy rozpoczęcia (ES) i zakończenia (EF) zadań. Algorytm iteruje przez zależności między zadaniami i oblicza te czasy, uwzględniając czas trwania zadań i zależności. Czas trwania całego projektu jest obliczany jako maksymalny czas spośród obliczonych czasów zakończenia zadań. W drugim etapie obliczane są najpóźniejsze możliwe czasy rozpoczęcia (LS) i zakończenia (LF) zadań. Ponownie iterujemy przez zależności, tym razem wstecz, aby obliczyć te czasy. Ostatecznie, algorytm znajduje zadania na ścieżce krytycznej, czyli te, których czasy ES i LS są sobie równe oraz EF i LF także są sobie równe. Te zadania są oznaczane jako krytyczne i zapisywane razem z czasami w pliku wyjściowym. Algorytm sprawdza także, czy w grafie istnieje cykl. Jeśli tak, to oznacza, że projekt nie może zostać zakończony i tę informację zapisuje w pliku wyjściowym.

```
#  przetwarzanie w przód: obliczenie ES, EF
    for _ in range(N):
        for a, b in dependencies:
            a, b = a-1, b-1
            earlyStart[b] = max(earlyStart[b], earlyFinish[a])
        for i in range(N):
            earlyFinish[i] = earlyStart[i] + durations[i]

    projectTime = max(earlyFinish)

    # przetwarzanie wstecz: obliczenie LS i LF
    for i in range(N):
        lateFinish[i] = projectTime
    for _ in range(N):
        for a, b in dependencies:
            a, b = a-1, b-1
            lateFinish[a] = min(lateFinish[a], lateStart[b])
        for i in range(N):
            lateStart[i] = lateFinish[i] - durations[i]

    criticalPath = []
    for i in range(N):
        if earlyStart[i] == lateStart[i] and earlyFinish[i] == lateFinish[i]:
            criticalPath.append((i + 1, earlyStart[i], earlyFinish[i]))

    criticalPath.sort(key=lambda x: x[1])
```

Listing 2: Obliczanie w przód i wstecz oraz wyliczanie ścieżki krytycznej dla algorytmu nr 1

4.2 Algorytm nr 2

W drugim algorytmie wykorzystano algorytm DFS. Algorytm DFS (Depth-First Search), czyli przeszukiwanie w głąb, jest jednym z podstawowych algorytmów przeszukiwania grafów. Jego głównym celem jest eksploracja wszystkich wierzchołków grafu, przeszukując go jak najgłębiej przed powrotem do wcześniejszych wierzchołków. Algorytm ten działa rekurencyjnie, w przeciwieństwie do poprzednio zaimplementowanego algorytmu. Na podstawie danych wejściowych program buduje graf zależności między zadaniami. Graf ten reprezentowany jest jako lista sąsiedztwa, gdzie każdy wierzchołek grafu odpowiada zadaniu, a krawędzie reprezentują zależności między zadaniami. Przed rozpoczęciem obliczeń, program sprawdza, czy graf zależności jest acykliczny. Wykorzystuje algorytm DFS do wykrywania cykli w grafie. Jeśli graf jest acykliczny, algorytm kontynuuje działanie. W przeciwnym razie zatrzymuje się i informuje o istnieniu cykli w grafie. Algorytm oblicza najwcześniejsze możliwe czasy rozpoczęcia i zakończenia dla każdego zadania. Wykorzystuje DFS, aby przeszukiwać graf w głąb, uwzględniając zależności między zadaniami. Czas rozpoczęcia zadania jest maksymalnym czasem zakończenia

zadań z nim zależnych. Czas zakończenia zadania to suma czasu rozpoczęcia i czasu trwania zadania. Algorytm następnie oblicza najpóźniejsze możliwe czasy zakończenia i rozpoczęcia dla każdego zadania. Na podstawie obliczonych czasów ES, EF, LS i LF algorytm określa ścieżkę krytyczną. Zadania, których czasy ES są równe LS i czasy EF są równe ich LF należą do ścieżki krytycznej.

```
def is_cyclic(graph, node, visited, rec_stack):
    visited[node] = True
    rec_stack[node] = True

    for neighbor in graph[node]:
        if not visited[neighbor]:
            if is_cyclic(graph, neighbor, visited, rec_stack):
                return True
        elif rec_stack[neighbor]:
            return True

    rec_stack[node] = False
    return False

def is_acyclic(graph, N):
    visited = [False] * (N + 1)
    rec_stack = [False] * (N + 1)

    for node in range(1, N + 1):
        if not visited[node]:
            if is_cyclic(graph, node, visited, rec_stack):
                return False

    return True
```

Listing 3: Funkcje wykorzystywane przy sprawdzaniu czy graf jest acykliczny dla algorytmu nr 2

Funkcja $is\_acyclic$ jest punktem wejścia do algorytmu sprawdzania acykliczności grafu. Przyjmuje graf $graph$ i ilość wierzchołków N. Funkcja is_cyclic przyjmuje graf (reprezentowany jako lista sąsiedztwa), bieżący wierzchołek ($node$), listę odwiedzonych wierzchołków ($visited$) oraz stos rekurencyjny (rec_stack), który śledzi wierzchołki w trakcie rekurencyjnego przeszukiwania.

```
def dfs_early(node, graph, durations, earlyStart, earlyFinish):
    for dependent in graph[node]:
        dfs_early(dependent, graph, durations, earlyStart, earlyFinish)
        earlyStart[node] = max(earlyStart[node], earlyFinish[dependent])
    earlyFinish[node] = earlyStart[node] + durations[node-1]

def dfs_late(node, graph, durations, lateStart, lateFinish):
    for dependent in [i for i, x in enumerate(graph) if node in x]:
        dfs_late(dependent, graph, durations, lateStart, lateFinish)
        lateFinish[node] = min(lateFinish[node], lateStart[dependent])
    lateStart[node] = lateFinish[node] - durations[node-1]
```

Listing 4: Obliczanie czasów rozpoczęcia i czasów zakończenia zadań dla algorytmu nr 2

Funkcja dfs_early oblicza najwcześniejsze możliwe czasy rozpoczęcia i zakończenia zadań w grafie. Funkcja rozpoczyna od bieżącego wierzchołka i rekurencyjnie przeszukuje wszystkie zależne zadania (sąsiednie wierzchołki) w grafie. Funkcja dfs_late oblicza najpóźniejsze możliwe czasy zakończenia ($lateFinish$) i rozpoczęcia ($lateStart$) dla zadań w grafie. Przyjmuje te same argumenty co dfs_early, ale operuje w przeciwnym kierunku (od zadań końcowych wstecz). Wyznaczanie ścieżki krytycznej odbywa się w ten sam sposób, co w algorytmie nr 1.

5 Dane wyjściowe

Poniżej przedstawiono uzyskane dane wyjściowe dla pliku *data20.txt* dla obu algorytmów. Dane te pokrywają się z oczekiwanymi.

```
process time:
286
earlyStart earlyFinish lateStart lateFinish:
0 93 88 181
0 14 85 99
46 99 46 99
114 152 181 219
195 202 279 286
121 189 121 189
98 99 278 279
46 98 167 219
189 193 282 286
0 46 0 46
0 1 98 99
189 256 219 286
99 104 176 181
0 75 114 189
189 195 189 195
195 248 195 248
46 114 113 181
99 121 99 121
248 286 248 286
0 83 112 195
critical path:
10 0 46
3 46 99
18 99 121
6 121 189
15 189 195
16 195 248
19 248 286
```

Listing 5: Dane wyjściowe

Czas obliczeń dla poszczególnych algorytmów:

- algorytm nr 1: 0.00223s,
- algorytm nr 2: 0.00123s.

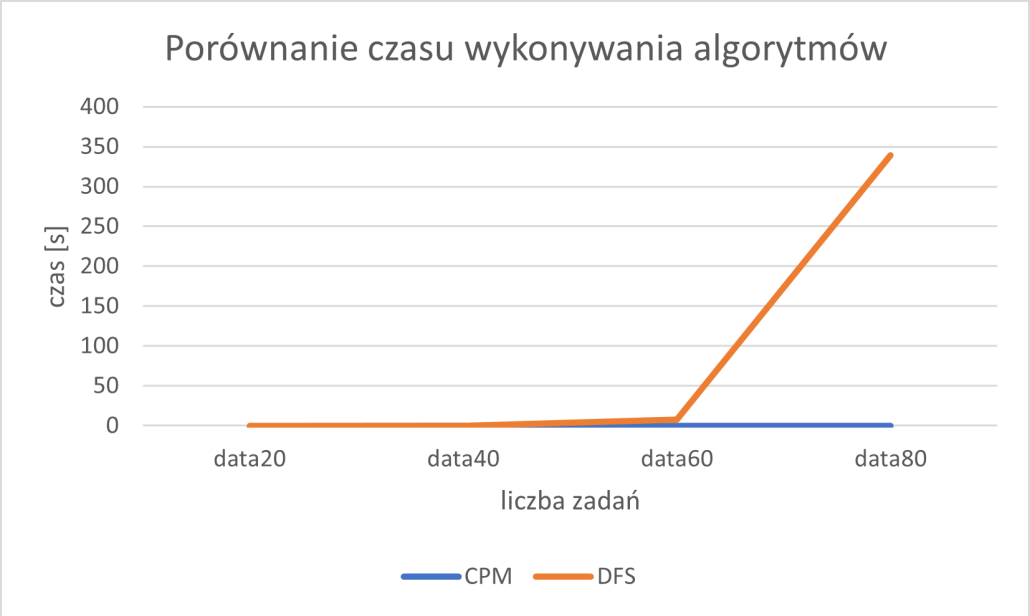
6 Badania

Aby sprawdzić wydajność stworzonych algorytmów postanowiono przeprowadzić dwa badania: pierwsze dla danych dostarczonych przez prowadzącego zajęcia laboratoryjne, a drugie dla liczby próbek $N \in [100, 1000]$ z krokiem 100 oraz liczby połączeń M równej około $M = 2N$, wygenerowanych losowo.

6.1 Badanie nr 1

W pierwszym badaniu sprawdzono jak zaimplementowane algorytmy radzą sobie z obliczeniami dla zbioru danych: *data20*, *data40*, *data60*, *data80*.

Na rysunku 6.1 widać, iż dla małych danych rzędu 20 - 60 próbek, algorytmy zachowują się podobnie. Natomiast dla 80 próbek i 640 połączeń algorytm DFS wykonuje się setki razy wolniej niż tradycyjny. Krzywą dla algorytmy DFS można by interpolować przez krzywą wykładniczą.

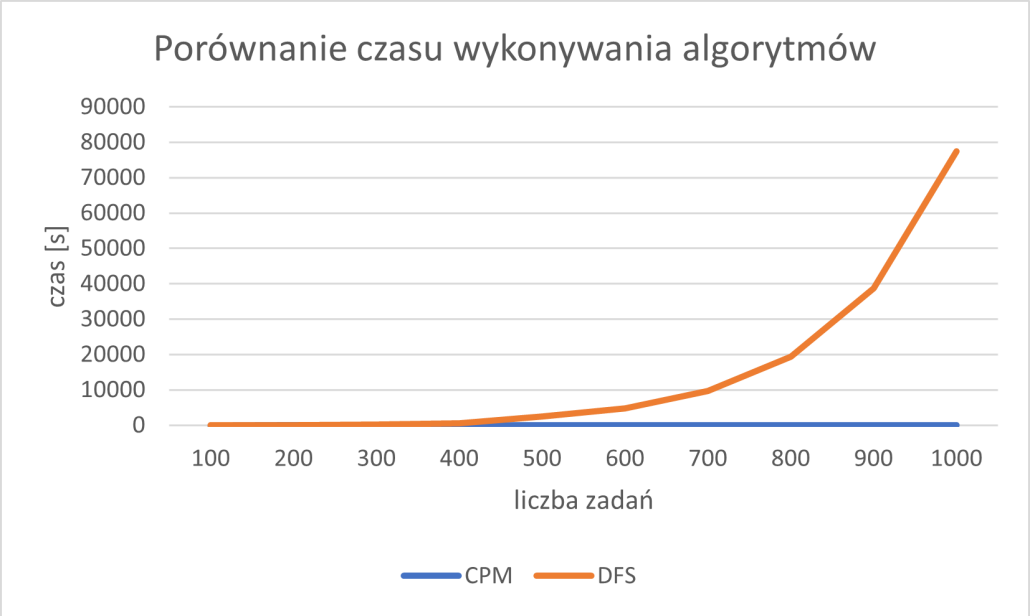


Rysunek 6.1: Porównanie czasu wykonywania algorytmu

6.2 Badanie nr 2

W drugim badaniu dane zostały wygenerowane przez program, który losowo generował czas trwania zadań oraz połączenia. Przyjęto, że liczba próbek to $N \in [100, 1000]$, natomiast liczba połączeń to $M = 2N$. Program ponadto podczas generowania danych dbał o to by nie generowały się przebiegi cykliczne.

Jak widać na rysunku 6.2 algorytm DFS mocno odbiega od tradycyjnego algorytmu przetwarzania w przód i w tył pod względem czasu wykonywania. Lepiej niż na rysunku 6.1 widać tu zależność wykładniczą czasu wykonania od ilości próbek dla algorytmu DFS, podczas gdy czas wykonywania tradycyjnego algorytmu rósł liniowo.



Rysunek 6.2: Porównanie czasu wykonywania algorytmu

Warto sprostować jedną rzecz, mianowicie czasy wykonywania algorytmu DFS dla ponad 600 próbek zostały ekstrapolowane przy użyciu czasów uzyskanych dla mniejszej ilości próbek, z racji tego, iż sprzęt którym dysponowaliśmy miał swoje ograniczenia.

7 Wnioski

- Udało się osiągnąć zamierzony cel ćwiczenia. Zaimplementowano dwa rodzaje algorytmów rozwiązujących problem ścieżki krytycznej. Przeprowadzono także badania, analizę porównawczą i wyciągnięto wnioski.
- CPM umożliwia dokładne określenie czasu trwania projektu, uwzględniając wszystkie zależności między zadaniami
- Dzięki tej metodzie można ocenić, które zadania można przyspieszyć lub opóźnić, aby zoptymalizować projekt pod kątem czasu i zasobów.
- Oba zaimplementowane algorytmy są skuteczne w identyfikacji ścieżki krytycznej.
- Rozwiązanie rekurencyjne okazało się znacznie wolniejsze. Znaczną różnicę w czasie obliczeń można zauważyć już przy ilości próbek 80.
- Czas obliczeń mocno zależy od ilości połączeń pomiędzy zadaniami.

- W implementacji rekurencyjnej algorytmu DFS, stos wywołań rekurencyjnych odgrywa kluczową rolę.
- Optymalnym rozwiązaniem okazała się metoda obliczania w przód i wstecz.

8 Bibliografia

[1] Ferdinand K Levy, Gerald L Thompson, and Jerome D Wiest. *The ABCs of the critical path method*. Harvard University Graduate School of Business Administration, 1963.

[2] dr inż. Mariusz Makuchowski. Metoda cpm/pert. http://mariusz.makuchowski.staff.iiar.pwr.wroc.pl/download/courses/badania.operacyjne/cpmpert.pdf. Data dostępu: 23.10.2022.