IMPLEMENTACJA I ANALIZA EFEKTYWNOŚCI ALGORYTMU PODZIAŁU I OGRANICZEŃ LUB PROGRAMOWANIA DYNAMICZNEGO DLA WYBRANEGO PROBLEMU OPTYMALIZACJI

Projektowanie efektywnych algorytmów

AUTOR: MACIEJ BRONIKOWSKI 248838
GRUPA PROJEKTOWA: ŚRODA 13:15
PROWADZĄCY: ANTONI STERNA

1 OPIS PROBLEMU

Przedmiotem projektu było zaprojektowanie algorytmów dokładnych dla jednoprocesorowego problemu szeregowania zadań przy kryterium minimalizacji ważonej sumy opóźnień zadań. Problem przedstawia zadania opisywane przez trzy parametry:

- p_i – czas wykonywania zadania
- d_i – termin, w którym zadanie powinno być wykonane
- w_i – dodatnia waga zadania

Zadania należy ułożyć kolejno w taki sposób, aby ich suma kar zapisanych wzorem $TWT = \sum_{i=1}^{n} w_i * T_i$, gdzie $T_i = \max(0, C_i - d_i)$ jest jak najmniejsza. Zadania są przetwarzane na jednej maszynie, nie mogą być wykonywane jednocześnie oraz muszą być wykonywane bez przerwy od czasu rozpoczęcia przez czasy wykonywania. Problem ten jest NP – trudny, czyli niedający się rozwiązać w złożoności czasowej wielomianowej. W projekcie zostały zastosowane trzy algorytmy: przeglądu zupełnego, podziału i ograniczeń oraz programowania dynamicznego.

1.1 PRZYKŁAD OBLICZANIA CAŁKOWITEJ KARY ZA OPÓŹNIENIA

Nr zadania	1	2	3
p_i	4	6	2
d_i	10	8	10
w_i	3	6	4

Dla podanego przykładu zostanie obliczona sumaryczne ważone opóźnienie dla kolejności zadań {1,2,3}. Najpierw należy wyznaczyć T_i:

$T_1 = \max(0, 4 - 10) = 0$ $T_2 = \max(0, 4 + 6 - 8) = 2$ $T_3 = \max(0, 10 + 2 – 10) = 2$

Całkowity ważony czas opóźnienia dla danej kolejności będzie równy:

$$TWT = 0 * 3 + 2 * 6 + 2 * 4 = 12 + 8 = 20$$

2 ALGORYTMY WYZNACZANIA MINIMALNEGO ROZWIĄZANIA

2.1 PRZEGLĄD ZUPEŁNY

Brute force jest metodą siłową, sprawdzającą wszystkie możliwe permutacje kolejności zadań i oblicza dla nich opóźnienie ważone. Jeżeli zostanie znaleziony wynik poprawiający rozwiązanie, zostaje on zapisany do obiektu zwracanego przez algorytm. Do wygenerowania wszystkich możliwych kolejności indeksów zadań został wykorzystany algorytm Heapa. Algorytm ten używany jest przez klasę WeightedTardiness w metodzie bruteforce.

2.1.1 Generowanie wszystkich permutacji zbioru zadań

Do algorytmu brute force została wykorzystana wersja rekurencyjna algorytmu Heap'a. Algorytm można przedstawić za pomocą pseudokodu w następujący sposób:

```
procedure generujPermutacje(k ∈ C, A : tablica elementów Aᵢ ∈ C, best ∈ C):
    if k = 1 than
        loos <- value(A)
        if loos < best
            best <- loos
        end if
    else
        generujPermutacje(k – 1, A, best)
        for i in 0 .. k – 1 do
            if k % 2 = 0 than
                swap(A[i], A[k-1]
            else
                swap(A[0], A[size-1]
            end if
            generujPermutacje(k – 1, A, best)
        end for
    end if
```

Algorytm 1 Algorytm przeglądu zupełnego

Gdzie:

- A – tablica indeksów, przechowująca kolejne permutacje
- k – wielkość tablicy, dla której będą wykonywane kolejne zamiany
- best – najlepszy dotychczas znaleziony wynik
- value(A) – funckja wyliczająca całkowite ważone opóźnienie wynikające z podanego ułożenia zadań
- swap(A[i], A[j]) – zamiana elementów znajdujących się w tablicy

Algorytm dokonuje zamian na odpowiednich pozycjach tablicy A wykonując kolejne rekurencje, dopóki k!=1. k==1 oznacza, że wyznaczona została kolejna permutacja tablicy A. Przykład działania algorytmu dla tablicy 3 – elementowej:

Nr permutacji	Indeks tablicy			Opis
	A[0]	A[1]	A[2]	
	0	1	2	Początkowa zawartość tablicy A
1	**0**	**1**	**2**	**Wynik (k=1)**
2	1	0	2	Zamiana A[0] i A[1]
	1	**0**	**2**	**Wynik (k=1)**
3	2	0	1	Zamiana A[0] i A[2]
	2	**0**	**1**	**Wynik (k=1)**
4	0	2	1	Zamiana A[0] i A[1]
	0	**2**	**1**	**Wynik (k=1)**
5	1	2	0	Zamiana A[0] i A[2]
	1	**2**	**0**	**Wynik (k=1)**
6	2	1	0	Zamiana A[0] i A[1]
	2	**1**	**0**	**Wynik (k=1)**

2.1.2 Wykorzystane struktury

JobsOrder* best – Instancja przechowująca najlepszy znaleziony dotychczas wynik. Składa się z dwóch elementów:

- std::vector<size_t>* order – Wektora przechowującego kolejność najlepszej znalezionej dotychczas kolejności zadań
- unsigned int totalLoos – Całkowite ważone opóźnienie wynikające z obecnie przechowywanej kolejności

std::vector<size_t>* order – Wektor przechowujący kolejność zadań kolejnych sprawdzanych permutacji potencjalnych rozwiązań problemu

2.1.3 Teoretyczna złożoność obliczeniowa

Złożoność czasowa	Złożoność pamięciowa
n!	n

Teoretyczna złożoność czasowa wynika z tego, że należy przejść przez wszystkie możliwe permutacje problemu. Można wyróżnić dokładnie n! wszystkich możliwych ciągów zadań zbioru n – elementowego.

Teoretyczna złożoność pamięciowa wynika z tego, że należy w danej chwili pamiętać jedynie obecnie przetwarzaną permutację zbioru n - elementowego.

2.2 PROGRAMOWANIE DYNAMICZNE

Programowanie dynamiczne jest alternatywną strategią znajdującą najlepszy możliwy wynik problemu. Polega on na kolejnym rozwiązywaniu pod problemów zaczynając od najmniejszego możliwego rozbicia problemu, kończąc na zbiorze zawierającym wszystkie elementy problemu.

2.2.1 Opis algorytmu

Działanie algorytmu zostanie przedstawione na poniższym przykładzie:

Nr zadania	1	2	3
p_i	4	6	2
d_i	10	8	10
w_i	3	6	4

Opis funkcji wykorzystanych w przykładzie

- S – zbiór zadań które zostały już wykonane
- value(x, S) – najmniejsze możliwe ważone opóźnienie, jeżeli zadanie x zostało wykonane po zadaniach znajdujących się w zbiorze S.
- value(S) – najmniejsze możliwe ważone opóźnienie zbioru zadań S
- T(x, t) – wartość ważonego opóźnienia, jeżeli zadanie x zostałoby wykonane w czasie t
- k – ilość elementów zbioru S
- $t_{x|S}$ = czas, w którym zadanie x zostało wykonane zakładając, że zostało ono wykonane zaraz po zadaniach ze bioru S

Rozwiązywanie problemu należy zacząć od najmniejszego możliwego zbioru rozwiązań. W tym wypadku będzie to zbiór pusty. Oznacza to, że dla każdego z zadań 1..3 należy obliczyć całkowite ważone opóźnienie zakładając, że są one wykonywane jako pierwsze.

- k = 0:

$value(1,\emptyset) = T(1,p_1) = 3 * \max(0,4-10) = 0, \qquad t_{1|\emptyset} = 4$

$value(2,\emptyset) = T(2,p_2) = 6 * \max(0,6-8) = 0, \qquad t_{2|\emptyset} = 6$

$value(3,\emptyset) = T(3,p_3) = 4 * \max(0,2-10) = 0, \qquad t_{3|\emptyset} = 2$

W następnym etapie należy rozwiązać problem dla każdego z zadań 1..3 uznając, że zadania są wykonywane jako drugie. Jako poprzednie rozwiązanie do którego obecne się odnosi należy wziąć pod uwagę wszystkie możliwe kombinacje 1-elementowe zbioru zadań. (Należy pominąć te zadania, które zostały już wykonane w danej kombinacji zbioru S).

- k=1

$value(2,\{1\}) = T\big(2,p_2 + t_{1|\emptyset}\big) + value(1,\emptyset) = 6 * \max(0,6+4-8) + 0 = 2, \qquad t_{2|\{1\}} = 10$

$value(3,\{1\}) = T\big(3,p_3 + t_{1|\emptyset}\big) + value(1,\emptyset) = 4 * \max(0,2+4-10) + 0 = 0, \qquad t_{3|\{1\}} = 6$

$value(1,\{2\}) = T\big(1,p_1 + t_{2|\emptyset}\big) + value(2,\emptyset) = 3 * \max(0,4+6-10) + 0 = 0, \qquad t_{1|\{2\}} = 10$

$value(3,\{2\}) = T\big(3,p_3 + t_{2|\emptyset}\big) + value(2,\emptyset) = 4 * \max(0,2+6-10) + 0 = 0, \qquad t_{3|\{2\}} = 8$

$value(1,\{3\}) = T\big(1,p_1 + t_{3|\emptyset}\big) + value(3,\emptyset) = 3 * \max(0,4+2-10) + 0 = 0, \qquad t_{1|\{3\}} = 6$

$value(2,\{3\}) = T\big(2,p_2 + t_{3|\emptyset}\big) + value(3,\emptyset) = 6 * \max(0,6+2-8) + 0 = 0, \qquad t_{2|\{3\}} = 8$

W kolejnym etapie należy rozwiązać problem dla każdego z zadani 1..3 zakładając, że zadanie jest wykonane jako trzecie. Tym razem dodatkowo jednak należy wziąć pod uwagę że pod zbiorów rozwiązań S jest więcej niż 1, dlatego należy znaleźć wartość najmniejszą biorąc pod uwagę obecnie sprawdzane zadanie.

- k=2

$$value(3,\{1,2\}) = \min\Big(T\big(3,p_3 + t_{2|\{1\}}\big) + value(2,\{1\}), \qquad \boldsymbol{T\big(3,p_3 + t_{1|\{2\}}\big) + value(1,\{2\})}\Big) =$$
$$= \min(4 * \max(0,2+10-10) + 2, \qquad \mathbf{4 * \max(0,2+10-10) + 0}) =$$
$$= \min(10,\mathbf{8}) = 8, \qquad t_{3|\{1,2\}} = 2 + 10 = 12$$

$$value(2,\{1,3\}) = \min\Big(\boldsymbol{T\big(2,p_2 + t_{3|\{1\}}\big) + value(3,\{1\})}, \qquad T\big(2,p_2 + t_{1|\{3\}}\big) + value(1,\{3\})\Big) =$$
$$= \min(\mathbf{6 * \max(0,6+6-8) + 0}, \qquad 6 * \max(0,6+6-8) + 0) =$$
$$= \min(\mathbf{24},24) = 24, \quad t_{2|\{1,3\}} = 12$$

$$value(1,\{2,3\}) = \min\Big(\boldsymbol{T\big(1,p_1 + t_{3|\{2\}}\big) + value(3,\{2\})}, \qquad T\big(1,p_1 + t_{2|\{3\}}\big) + value(2,\{3\})\Big) =$$
$$= \min(\mathbf{3 * \max(0,4+8-10) + 0}, \qquad 3 * \max(0,4+8-10) + 0) =$$
$$= \min(\mathbf{6},6) = 6, \qquad t_{1|\{2,3\}} = 12$$

Kolejny etap jest już ostatni jeżeli chodzi o znalezienie minimalnej ważonej sumy opóźnień zadań. Należy znaleźć wartość minimalną z kroku poprzedniego, gdzie k=2

- k=3

$$value(\{1,2,3\}) = \min\big(value(3,\{1,2\}), value(2,\{1,3\}), \boldsymbol{value(1,\{2,3\})}\big) =$$
$$= \min(8,24,\mathbf{6}) = 6$$

Wynikiem algorytmu jest najmniejsza ważona suma opóźnień, która w tym wypadku wynosi 6. Algorytm można zapisać za pomocą pseudo kodu w następujący sposób:

```
for k in 1..n do      ;generowanie rozwiązań dla najmniejszego pod problemu
   loos[k][Ø] <- p_i
   time[k][Ø] <- d_i
end for
for k in 1..n-1 do       ;kolejne wielkości problemu
   for S ⊆ {1,2,..,n} and |S| = k do       ;kolejne kombinacje k-elementowe
      tmpMinLoos <- ∞
      tmpTime = 0
      for all x ∈ {1,2,..,n} and x ∈ S do       ;najlepszy wynik dla danego podzbioru rozwiązań
         if loos[x][S/{x}] < tmpMinLoos than
            tmpTime <- time[x][S/{y}]+p_x
            tmpMinLoos <- loos[x][S/{x}]
         end if
      end for
      for all x ∈ {1,2,..,n} and x ∉ S do       ;wyliczanie ważonego opóźnienia dla kolejnych zadań
         time[x][S] <- tmpTime + d_x
         loos[x][S] <- tmpMinLoos + T(x, time[x][S])
      end for
   end for
end for
result <- ∞
for k in 1..n do       ;znajdowanie optymalnego rozwiązania według wyliczonych danych*
   if loos[x][{1…n}] < result than
      result <- loos[x][{1…n}]
   end if
end for
```

Algorytm 2 Algorytm DP znajdujący najmniejszą ważoną sumę kary opóźnień zadań

Powyższy algorytm wymaga jeszcze drobnej zmiany. W obecnej formie jest w stanie wyznaczyć jedynie najlepsze rozwiązanie, jednak dalej nie znamy kolejności wykonywania zadań. Zamiast ostatniej pętli oznaczonej gwiazdką można zastosować dodatkowy algorytm znajdujący najlepszą kolejność zadań:

```
currentSet <- {1,2,…,n}    ;obecnie sprawdzany podzbiór w poszukiwaniu najlepszego wyniku
for k in n-1…0 do    ;dla każdej wielkości problemu zaczynając od największej
   minLoos <- ∞
   submaskBest <- Ø
   for x ∈ {1,2,..,n} and x ∈ currentSet do    ;dla każdego zadania w obecnym podzbiorze
      if loos[x][currentSet \ {x}] < minLoos than
         minLoos <- loos[x][currentSet \ {x}]
         submaskBest <- currentSet \ {x}
         bestOrder[k] <- x    ;przypisanie numeru zadania do najlepszej wartości
      end if
   end for
   currentSet <- submaskBest    ;najlepsze znalezione zadanie pomniejsza zbiór rozpatrywany
end for
bestLoos <- loos[bestOrder[n-1]][{1…n} \ bestOrder[n-1]]
```

Algorytm 3 Algorytm DP znajdujący najlepsze ułożenie zadań

Na podstawie Algorytmu 2 i przykładu z punktu 2.2.1 można uzupełnić tablicę ważonych sum opóźnień podzbiorów oraz tablicę czasów wykonywania
Tablica ważonych sum opóźnień (loos[][]):

Podzbiór / Zadanie	Ø	{1}	{2}	{3}	{1,2}	{1,3}	{2,3}
1	0	∞	0	0	∞	∞	6
2	0	2	∞	0	∞	24	∞
3	0	0	0	∞	8	∞	∞

Tablica czasów wykonywania (time[][])

Podzbiór / Zadanie	Ø	{1}	{2}	{3}	{1,2}	{1,3}	{2,3}
1	4	-	10	6	-	-	12
2	6	10	-	8	-	12	-
3	2	6	8	-	12	-	-

Algorytm 3 na podstawie tablicy ważonych sum opóźnień dla kolejnych wielkości podzbiorów zaczynając od największego znajduje kolejność zadań:

Podzbiór / Zadanie	Ø	{1}	{2}	{3}	{1,2}	{1,3}	{2,3}
1	0	∞	0	0	∞	∞	(6)
2	(0)	2	∞	0	∞	24	∞
3	0	0	(0)	∞	8	∞	∞

Przykładowo podczas pierwszej iteracji najlepsze rozwiązanie znajdowane jest w *loos[1][{2,3}]*. W kolejnej iteracji przeszukiwany będzie podzbiór wielkości 1. Należy jednak pamiętać, że zadanie 1 zostało już wykorzystane, dlatego mamy dostępne tylko do wyboru pod problemy *loos[3][{2}]* oraz *loos[2][{3}]*. W ostatniej iteracji zostaje już nam do wykorzystania jedynie zadanie nr 2.
Aby znaleźć wartość całkowitej sumy opóźnień wystarczy pobrać wartość z komórki *loos[1][{2,3}]*

2.2.2 Implementacja algorytmu

Najważniejszymi strukturami przechowującymi dane w algorytmie Programowania Dynamicznego są:

- std::vector<std::vector<DbSubset>> - dwuwymiarowy wektor obiektów przechowujących jednocześnie obliczone ważone sumy opóźnień, oraz czasy wykonywania się pod problemów.
- JobsOrder* best – Ta sama instancja obiektu która została wykorzystana w punkcie 2.1.2. Przechowuje najlepszą kolejność zadań, oraz jej całkowitą sumę ważoną opóźnień.

Zbiory zostały zaimplementowane przez operacje na liczbie binarnej. Można zauważyć, że jeżeli mamy dostępne 5 zadań oraz zbiór który aktualnie chcemy uwzględnić to {1,3,4}, może on zostać przedstawiony w postaci liczby binarnej 01101. Wartość bitowa 1 oznacza, że daną wartość uwzględniamy w rozważanym zbiorze, a wartość bitowa 0, że jej nie uwzględniamy. Dzięki temu można bardzo szybko uzyskać potrzebne podzbiory:

- Zbiór pełny: (size_t(1)<<n) – 1
- Test, czy podzbiór S należy do podzbioru W: S&W
- Odjęcie zbioru S od zbioru D: S^D

2.2.3 Teoretyczna złożoność obliczeniowa

Złożoność czasowa	Złożoność pamięciowa
n^2*2^n	n*2^n

Wartość złożoności pamięciowej wynika z tego, że należy przechowywać w pamięci wartości ważonych sum opóźnień dla każdego zadania, jeżeli zostałoby ono wykonane zaraz po wykonaniu się zadań z danego podzbioru. Ilość zadań jest równa n, a ilość dostępnych podzbiorów problemu jest równa 2^n.

Złożoność czasowa wynika z tego, że dla każdego z 2^n podzbiorów należy określić wartości dla (nie zawsze) każdego z n zadań należy przejrzeć poprzednie (nie zawsze) n podzbiorów w poszukiwania poprzedniego minimum.

2.3 Algorytm Branch and bound

Algorytm opera się o interpretowanie kolejnych rozwiązań jako drzewa, oraz odcinanie pełnego obliczania wyniku tych gałęzi, które są nieobiecujące. Można wyróżnić dwa ograniczenia:

- Górne – Może to być po prostu najlepsze dotychczas znalezione rozwiązanie. Wiemy, że najlepsze rozwiązanie nie będzie na pewno większe od tego ograniczenia.
- Dolne – Określa, że wynik końcowy nie będzie od tego ograniczenia mniejszy.

Jeżeli Dolne ograniczenie jest większe lub równe ograniczeniu górnemu, wiemy, że nie ma sensu dalej rozwijać danej gałęzi drzewa.

Problem w implementacji algorytmu wynika z tego, że nie ma jasnej zasady dobierania ograniczeń. Należy je zaprojektować w taki sposób, żeby dawały pewne wyniki, oraz były jak najlżejsze w obliczaniu.

Należy również dobrać w jaki sposób chcemy przeglądać drzewo rozwiązań. Przykładowo możemy przeglądać je w głąb, w szerz, lub zawsze biorąc pod uwagę z obecnie najmniejszym ograniczeniem dolnym (Best First). W programie został wykorzystany pierwszy ze sposobów.

2.3.1 Algorytm B&B przeglądu drzewa w głąb

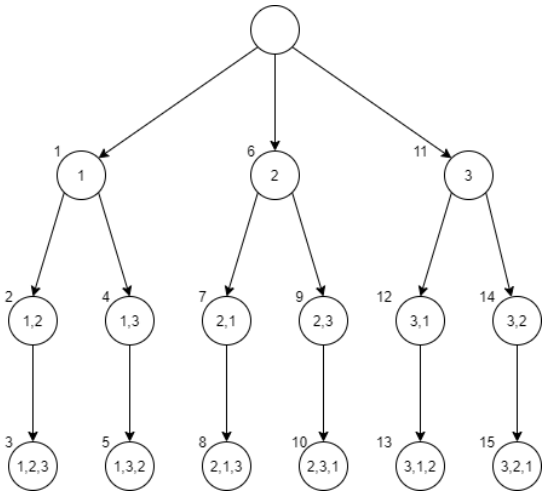
```
jobs[n] = {{p1, d1, w1},...}
procedure generujPermutacje(A : tablica elementów Ai ∈ C, left ∈ C, loos ∈ C, time ∈ C, ref best ∈ C, B :
tablica elementów Bi ∈ {true, false}) :
  if left = count(A) than    ;liść drzewa
    currentTime <- time + jobs[A[l]].p
    currentLoos <- loos + value(A[l], currentTime)
    if currentLoos < best than
      best <- currentLoos
    end if
  else
    for i in left ... count(A) do    ;kolejne zamiany indeksów
      currentTime <- time + jobs[A[i]].p
      currentLoos <- loos + value(A[i], currentTime)
      B[i] = true
      if currentLoos + lowerBound(B) < best than    ;test czy warto rozwijać gałąź
          swap(A[l], A[i])
          generujPermutacje(A, left + 1, currentLoos, currentTime, best, B);
          swap(A[l], A[i])
      end if
      B[i] = false
    end for
  end if
end procedure

best <- upperBoud()    ;pierwsza granica górna
A <- {1, 2, ..., n}
B <- {false, false, ..., false}
generujPermutacje(A, 0, 0, 0, best, B)    ;pierwsze wywołanie rekurencji
```

Algorytm 4 Generowanie kolejnych permutacji

Powyższy algorytm generuje kolejne permutacje problemu. Ważniejszymi z elementów algorytmu są:

- A[] – tablica przechowująca obecną permutację problemu
- B[] – tablica przechowująca, które z zadań w obecnym wierzchołu drzewa zostały już rozpatrzone. Dolne ograniczenie będzie wyliczane tylko dla tych elementów, które jeszcze nie zostały rozpatrzone.
- lowerBound(B) – funkcja obliczająca ograniczenie dolne na podstawie informacji, które z zadań zostały już wcześniej uwzględnione w wyniku
- upperBound() – oblicza pierwsze ograniczenie górne problemu

W najgorszym wypadku, jeżeli ograniczenia nie odetną żadnej gałęzi, dla problemu 3 zadań, generowanie kolejnych rozwiązań można zobrazować za pomocą drzewa:



Wartości znajdujące się wewnątrz okręgów obrazują zadania, których wartość w danej iteracji została już uwzględniona. Do wartości ich opóźnień zostaje dodana wartość obliczona przez funkcję lowerBound(n). Wartości poza okręgami obrazują kolejność sprawdzania kolejnych wierzchołków drzewa.

2.3.2 Ograniczenie górne

Jako pierwsze ograniczenie górne może zostać przypisana wartość nieskończoność (maksymalnie odstępna wartość). W programie jest ono ustawiona jako wartość równania:

$$upperBound = \left(\sum_{i=1}^{n} w_{A[i]} * T_{A[i]}\right) + 1$$

Gdzie A[] oznacza tablicę zadań posortowanych w kolejności od tych, których wykonanie zajmuje najmniej czasu, do tych, których wykonanie zajmuje najwięcej czasu.
Według wstępnych testów dało to nieznacznie lepsze wyniki od ustawiania początkowej wartości na równą maksymalnej dostępnej wartości.

2.3.3 Ograniczenie dolne

Zaimplementowane zostały dwa różne ograniczenia dolne

2.3.3.1 Ograniczenie dolne 1

Według pierwszego z ograniczeń dolnych zakładamy, że wszystkie z nieuwzględnionych jeszcze zadań wykonają się z maksymalnym czasem wykonywania się z pośród tych zadań. Załóżmy że tablica A zawiera n zadań. Zadania o indeksach A[1] – A[k] są zadaniami, których suma całkowitego ważonego opóźnienia została już wyliczona. W komórkach tablicy A[k+1] – A[n] znajdują się indeksy nie uwzględnionych zadań, posortowane w taki sposób, aby pierwszym z zadań było to o najkrótszym czasie, a ostatnim to o najdłuższym czasie wykonywania. W pierwszym etapie należy wyznaczyć maksymalny czas wykonywania z zadań A[k+1] – A[n], załóżmy że wartość ta zostanie oznaczona symbolem d. Wtedy minimalną wartością jaka może zostać osiągnięta jest

$$lowerBound = \left(\sum_{i=1}^{k} w_{A[i]} * T_{A[i]}\right) + \left(\sum_{i=k+1}^{n} w_{A[i]} * max(0, C_i - d)\right)$$

2.3.3.2 Ograniczenie dolne 2

Drugie z ograniczeń zostało zaprojektowane w taki sposób, aby uznać wszystkie nieuwzględnione z zadań za takie, które wykonują się w tym samym czasie od razu po tych już uwzględnionych. Załóżmy, że t jest czasem, w którym zadania A[1] – A[k] zostały wykonane. Wtedy

$$lowerBound = \left(\sum_{i=1}^{k} w_{A[i]} * T_{A[i]} \right) + \left(\sum_{i=k+1}^{n} w_{A[i]} * max(0, t - d) \right)$$

2.3.4 Implementacja

Najważniejszymi strukturami używanymi przez algorytm są

- JobsOrder* best – Ta sama instancja obiektu która została wykorzystana w punkcie 2.1.2. Przechowuje najlepszą kolejność zadań, oraz jej całkowitą sumę ważoną opóźnień.
- std::vector<size_t>* currentPermutation – przechowuje indeksy zadań obecnej sprawdzanej permutacji
- std::vector<size_t>* orderedJobs – przechowuje indeksy zadań posortowanych w kolejności od najkrótszego do najdłuższego czasu wykonywania się. Używana jest przy wyznaczaniu pierwszego ograniczenia górnego, oraz pierwszej z wersji ograniczeń dolnych

Dodatkowo metoda WeightedTardiness::BandBDFS główna algorytmu jako parametr otrzymuje wskaźnik do funkcji. Pozwala to w prosty sposób na wybór ograniczenia dolnego.

2.3.5 Teoretyczna złożoność obliczeniowa

Złożoność czasowa	Złożoność pamięciowa
n!	n

Teoretyczna złożoność czasowa algorytmu jest taka sama jak ta z algorytmu przeglądu zupełnego. Należy jednak pamiętać, że jest to najgorszy możliwy przypadek do którego może nigdy nie dojść, jeżeli w odpowiedni sposób dobierzemy ograniczenia. W praktyce algorytm ten jest znacznie szybszy od algorytmu przeglądu zupełnego.

Teoretyczna złożoność pamięciowa wynika z tego, że należy trzymać w pamięci obecną permutację zadań. Dodatkowo w przedstawionym wyżej algorytmie potrzebne jest przechowywanie zadań w odpowiedniej kolejności, aby sortowanie nie odbywało się podczas każdego testu dolnego ograniczenia, oraz informacji o tym, które z zadań są już uwzględnione w wyniku. Wielkość tych tablic wynosi zawsze n.

3 EKSPERYMENT

3.1 PLAN EKSPERYMENTU

Dla każdej z wielkości problemów wygenerowane jest 100 losowych instancji problemu. Każdy z parametrów zadań (czas wykonywania się, oczekiwany czas zakończenia zadania oraz priorytet) zostały wygenerowane losowo za pomocą rozkładu jednostajnego ciągłego. Kolejne parametry generowane były w przedziale:

- Czas wykonywania się zadania: [1; 100]
- Priorytet zadania [1; 10]
- Oczekiwany czas zakończenia [max(p_i); sum(p_i)]

Gdzie max(p_i) oznacza maksymalną wartość czasu wykonywania się spośród zadań, a sum(p_i) oznacza sumę czasów wykonywania się zadań.

3.2 GENEROWANIE DANYCH LOSOWYCH

Do wygenerowania danych losowych stworzony został singleton RandomGenerator. Jego najważniejszymi elementami są

- std::random_device – niedeterministyczny generator liczb całkowitych o rozkładzie jednostajnym ciągłym
- std::mt19937 – pseudolosowy generator liczb 32-btitowych
- std::uniform_int_distribution<unsigned int> - pozwala na wytrzenie liczb pseudolosowych z podanego przedziału

3.3 SPOSÓB POMIARU CZASU

Pomiar czasu odbywa się z użyciem std::chrono::high_resolution_clock. Klasa ta pozwala na wyznaczanie bardzo małych odstępów czasu. Przykład wyznaczania odstępu czasu:

```
std::chrono::high_resolution_clock::time_point start = std::chrono::high_resolution_clock::now();
//obliczenia, których czas chcemy zmierzyć
std::chrono::high_resolution_clock::time_point end = std::chrono::high_resolution_clock::now();
long long time = std::chrono::duration_cast<std::chrono::microseconds>(end - this->start).count();
```

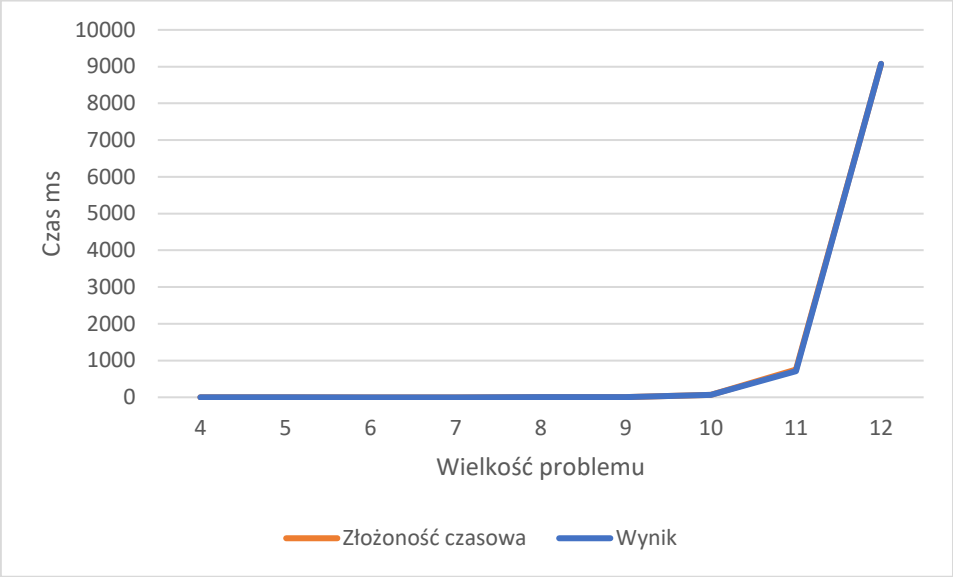
Wynikiem jest czas podany w mikrosekundach.

3.4 WYNIKI POMIARÓW

Algorytm / Rozmiar	Przegląd zupełny [ms]	Programowanie Dynamiczne [ms]	B&B Dolne ograniczenie 1 [ms]	B&B Dolne ograniczenie 2 [ms]
4	0.00023	0.00064	0.00123	0.00113
5	0.00200	0.00163	0.00542	0.00465
6	0.01129	0.00539	0.02531	0.01923
7	0.07899	0.01166	0.15058	0.07367
8	0.63947	0.02469	0.80978	0.32396
9	5.84637	0.04927	5.23334	1.93773
10	63.19853	0.14027	35.18731	9.28988
11	707.94667	0.32319	264.14211	47.67949
12	9075.89036	0.62935	1872.50998	226.57621
13	-	1.18628	17281.14987	2484.36296
14	-	2.33547	-	14896.24278
15	-	5.05423	-	-
16	-	10.59848	-	-
17	-	28.85828	-	-
18	-	80.49654	-	-
19	-	195.25306	-	-
20	-	456.96966	-	-
21	-	1035.99948	-	-
22	-	2346.81375	-	-
23	-	5322.89272	-	-

3.5 ANALIZA WYNIKÓW

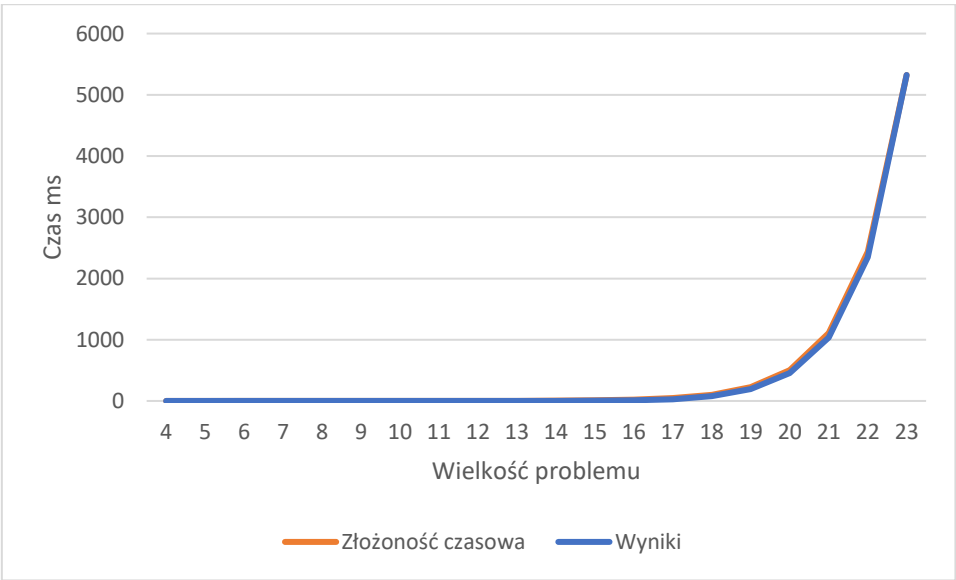
3.5.1 Przegląd zupełny



Na wykresie został przedstawiony teoretyczny czas wykonywania się zadania uwzględniając złożoność czasową O(n!) oraz oszacowany czas wykonywania się algorytm dla jednej iteracji – pomarańczowa linia i rzeczywiste wyniki eksperymentu – niebieska linia. Wykres pokazuje, że złożoność obliczeniowa zgadza się z tą teoretyczną.

Złożoność czasowa algorytmu potrafi być naprawdę duża. Dla 12 zadań średni czas wyniósł już około 9 sekund. Oznacza to, że jeżeli potrzebowalibyśmy wyznaczyć kolejność 20 zadań, potrzebowalibyśmy około 1445 lat.

3.5.2 Programowanie dynamiczne



Na powyższym wykresie zostały przedstawione dwa rodzaje danych. Pierwszy, oznaczony linią pomarańczową, jest teoretycznym czasem wykonywania się algorytmu na podstawie złożoności czasowej oraz oszacowanego czasu wykonywania się algorytmu dla jednej iteracji. Drugi, oznaczony linią niebieską jest rzeczywisty wynik eksperymentu. Wyniki są bardzo zbliżone do siebie.

3.5.3 Branch and Bound

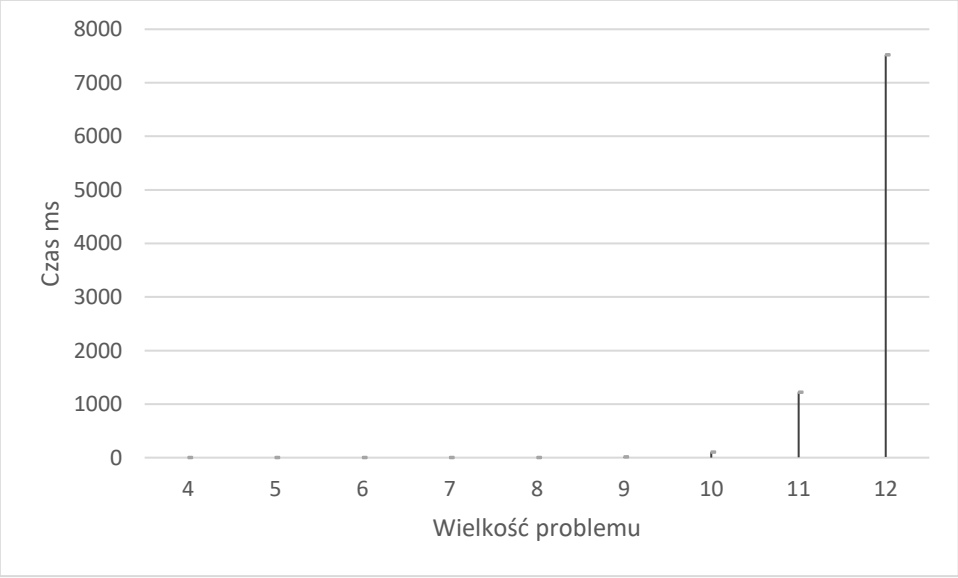
Liczne pomijanie gałęzi które rozwijają drzewo rozwiązań powoduje, że zestawienie z teoretyczną złożonością obliczeniową traci sens.

Czas działania algorytmu Branch and Bound w dużej mierze może zależeć od tego, czy wygenerowane dane sprzyjają metodzie generowania ograniczeń. Na poniższych tabela dla mniejszej wielkości instancji zostały przedstawione nie tylko średnie, ale również najmniejsze i największe czasy wykonywania się algorytmów:

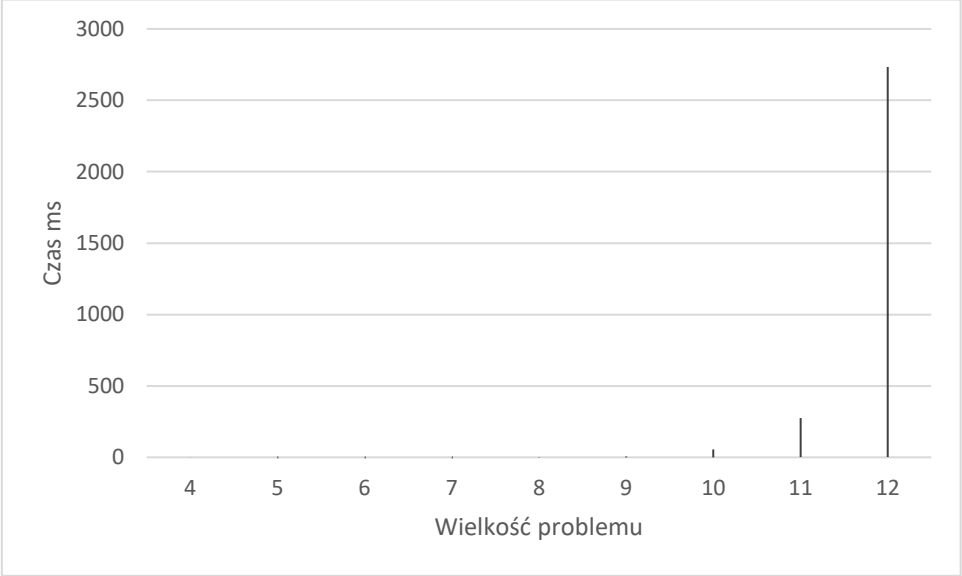
Algortym / Rozmiar	**B&B 1**			**B&B 2**		
	śr [ms]	**min [ms]**	**max [ms]**	**śr [ms]**	**min [ms]**	**max [ms]**
4	0.00127	0.00100	0.00300	0.00108	0.00000	0.00300
5	0.00577	0.00300	0.00900	0.00483	0.00100	0.00700
6	0.02625	0.00500	0.04300	0.02012	0.00100	0.03300
7	0.13418	0.00300	0.24900	0.08378	0.01900	0.21000
8	0.87212	0.00800	1.90100	0.37291	0.02800	1.45200
9	5.07455	0.32100	12.82600	1.60875	0.06000	5.86800
10	36.44355	4.08400	104.49900	7.71543	0.22200	55.13200
11	272.93537	4.53100	1221.76600	38.26759	1.22100	275.53200
12	1924.19265	11.79300	7521.58000	243.64117	1.18700	2733.19100

Różnice szczególnie widać na wykresach:

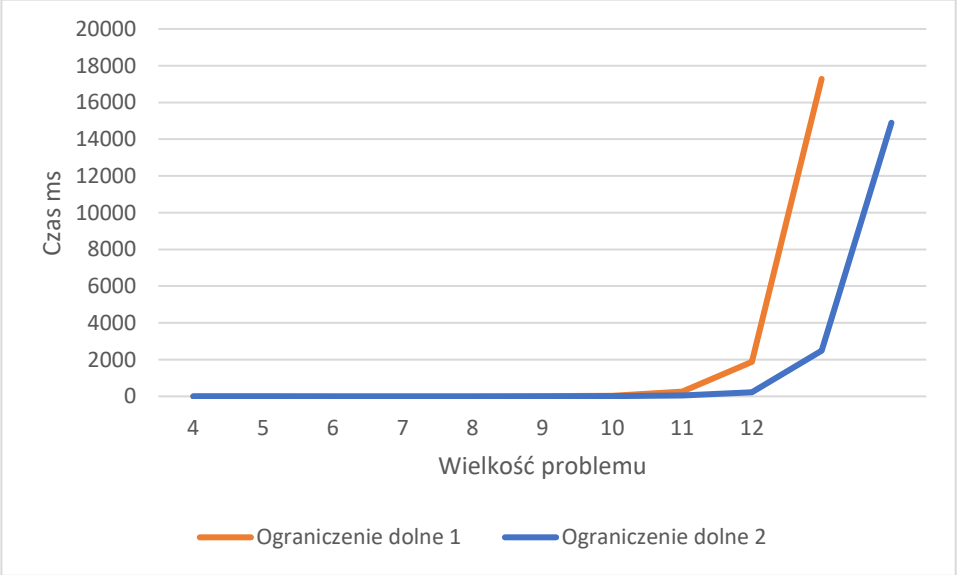
- Pierwsze ograniczenie dolne:



- Drugie ograniczenie dolne:



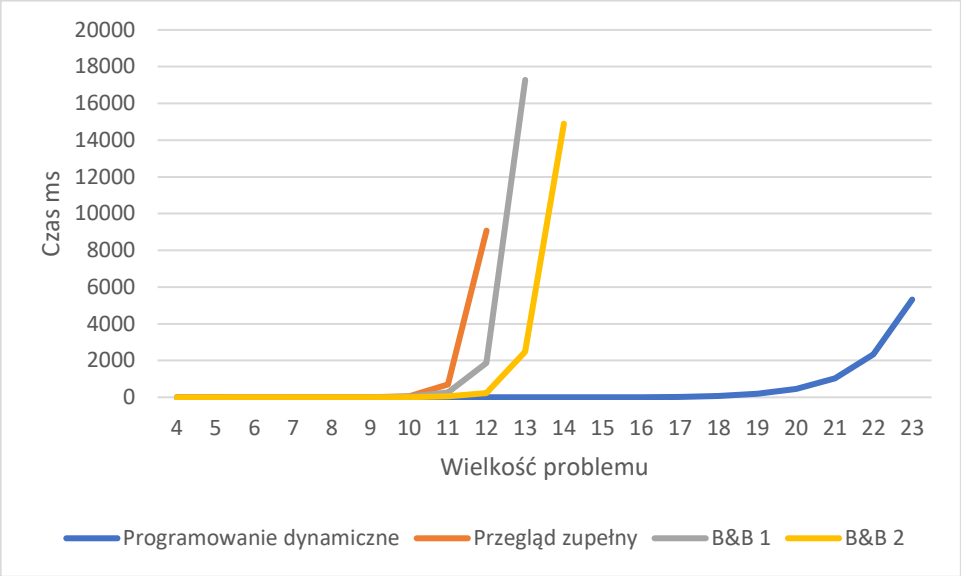
W porównaniu między pierwszą a drugą funkcją ograniczenia dolnego, lepiej wypada ta druga. Można zobaczyć to na wykresie:



Jednak różnica ta może wynikać ze sposobu generowania instancji. Różnica mogłaby się zatrzeć, a nawet przeważyć na rzecz ograniczenia dolnego 1, jeżeli przewidywany czas zakończenia zadań byłby mniej rozbity.

4 WNIOSKI

W projekcie zostały przedstawione trzy algorytmy. Przegląd zupełny, Programowanie Dynamiczne oraz Branch and Bound. Zestawienie wyników wszystkich trzech algorytmów wraz z wyróżnieniem dwóch metod wyznaczania dolnego ograniczenia dla algorytmu B&B zostało przedstawione na wykresie poniżej:



Z wykresu wynika, że najbardziej efektywnym algorytmem jest Programowanie Dynamiczne. Jest ono jednak również algorytmem, który zużywa najwięcej pamięci, w porównaniu pozostałych algorytmów, dla których wymogiem jest jedynie przechowywania kilku tablic wielkości problemu. Dla problemu wielkości 26 elementów ilość zajmowanej pamięci przez program wyniosła ok 10 GB.

Dla algorytmu B&B prędkość wyznaczenia najmniejszej ważonej sumy opóźnień w dużej mierze zależy od tego, czy dane problemu są rozłożone w taki sposób, że funkcja wyliczająca ograniczenie spowoduje szybkie ograniczenie problemu. W tabeli przedstawiającej maksymalne czasy wykonywania się algorytmu B&B można zobaczyć, że w najgorszym wypadku zbliża się on czasem wykonywania do algorytmu Przeglądu Zupełnego. Jednak zaletą algorytmu B&B w porównaniu do algorytmu Przeglądu Zupełnego jest bardzo małe zużycie zasobów pamięci urządzenia.