Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Poniedziałek 15:25*
Temat *Metoda podziału i ograniczeń i programowanie dynamiczne*	**Problem** *TSP*
Skład grupy *241385 Radosław Lis*	**Nr grupy** *-*
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *25 listopada 2019*

Spis treści

1 Opis problemu

Problem komiwojażera (ang. **TSP** - *Travelling Salesman Problem*) to jedno z najbardziej powszechnych zagadnień z dziedziny algorytmiki. W celu zobrazowania zagadnienia, należy wyobrazić sobie komiwojażera, który podróżuje między miastami w prowincji, sprzedając swoje towary. Wyrusza ze swojego domu, po czym jego trasa przebiega dokładnie jeden raz przez każde miasto w prowincji, aż na końcu wraca do domu rodzinnego. Rozwiązanie problemu to znalezienie odpowiedniej drogi i kosztów podróży (odległość, czas itp.), która maksymalnie je zminimalizuje.

Z matematycznej perspektywy wygląda to tak, że miasta są wierzchołkami grafu, a łączace je trasy to krawędzie z odpowiednimi wagami. Jest to graf pełny, ważony oraz może być skierowany - co tworzy problem *asymetryczny*. Rozwiązanie problemu komiwojażera sprowadza się do znalezienia właściwego - o najmniejszej sumie wag krawędzi - *cyklu Hamiltona*, czyli cyklu przechodzącego przez każdy wierzchołek grafu dokładnie jeden raz. Przeszukanie wszystkich cykli (czyli zastosowanie metody *Brute Force*) nie jest optymalną metodą, jako że prowadzi do wykładniczej złożoności obliczeniowej - $O(n!)$, dla której problemy o dużym n są traktowane jako nierozwiązywalne. Klasyfikuje to problem komiwojażera jako *problem NP-trudny*, czyli niedający rozwiązania w czasie wielomianowym. To powoduje konieczność skorzystania z tzw. *algorytmów heurystycznych* bądź *metaheurystycznych* (bardziej ogólnych), a w naszym przypadku konkretnie algorytmu *Branch & Bound* oraz *programowania dynamicznego*.

2 Metoda rozwiązania

2.1 Brute Force

Brute Force to metoda sprawdzająca wszystkie permutacje i dająca gwarancję uzyskania optymalnego rozwiązania . Została zaimplementowana w dwóch wersjach - *algorytmem B.R.Heap'a* oraz *drzewem przeszukiwań*.

2.1.1 Algorytm B.R. Heap'a

Pierwszą metodą na generowanie wszystkich permutacji zbioru {1, 2, ..., n-1} jest algorytm B.R. Heap'a, który rekurencyjnie wywołuje funkcję, w której w pętli następuje *swap* odpowiednich elementów tablicy, wywołanie funkcji *permute* i późniejszy *powrotny swap* po wyliczeniu funkcji celu dla danej permutacji.

0.	0	1	2	3	4	swap indeksów [0][0]
1.	**0**	**1**	**2**	**3**	**4**	swap indeksów [2][3]
2.	**0**	**1**	**2**	**4**	**3**	powrotny swap indeksów [2][3]
3.	0	1	2	3	4	swap indeksów [1][2]
4.	**0**	**1**	**3**	**2**	**4**	swap indeksów [2][3]
5.	**0**	**1**	**3**	**4**	**2**	powrotny swap indeksów [2][3]
6.	0	1	3	2	4	powrotny swap indeksów [1][2]
7.	0	1	2	3	4	swap indeksów [1][3]
8.	**0**	**1**	**4**	**3**	**2**	swap indeksów [2][3]
9.	**0**	**1**	**4**	**2**	**3**	powrotny swap indeksów [2][3]
10.	0	1	4	3	2	powrotny swap indeksów [1][3]
11.	0	1	2	3	4	powrotny swap indeksów [0][0]
12.	0	1	2	3	4	swap indeksów [0][1]
13.	**0**	**2**	**1**	**3**	**4**	swap indeksów [2][3]
14.	**0**	**2**	**1**	**4**	**3**	powrotny swap indeksów [2][3]
15.	0	2	1	3	4	swap indeksów[1][2]
16.	**0**	**2**	**3**	**1**	**4**	swap indeksów [2][3]
17.	**0**	**2**	**3**	**4**	**1**	powrotny swap indeksów [2][3]
18.	0	2	3	1	4	powrotny swap indeksów[1][2]
19.	0	2	1	3	4	swap indeksów[1][3]
20.	**0**	**2**	**4**	**3**	**1**	swap indeksów [2][3]
21.	**0**	**2**	**4**	**1**	**3**	powrotny swap indeksów [2][3]
22.	0	2	4	3	1	powrotny swap indeksów[1][3]
23.	0	2	1	3	4	powrotny swap indeksów[0][1]
24.	0	1	2	3	4	swap indeksów[0][2]
25.	**0**	**3**	**2**	**1**	**4**	dalsza część analogicznie

Tablica 1: Schemat działania algorytmu

Po prześledzeniu wszystkich kroków generowania permutacji - które są wytłuszczone - można łatwo zobrazować sobie ideę i koncepcję algorytmu, który gwarantuje przejście po wszystkich permutacjach danego zbioru. Łączna liczba permutacji to $(n-1)!$, gdyż ustalone zostało, że wierzchołek 0 to wierzchołek startowy.

Listing 1: Generowanie permutacji

```
void permute(int *a, int *b, int l, int r, int &min, int size, int **matrix)
{
    int value;
    if (l == r) {
        value = calculate(a,size,matrix);

        if (value < min) {
            for (int i = 0; i <= r; i++)
                b[i] = a[i];
            min = value;
        }
    }
    else
    {
        for (int i = l; i <= r; i++)
        {
            swap(a[l], a[i]); //swap
            permute(a, b, l + 1, r, min, size,matrix);
            swap(a[l], a[i]); //powrotny swap
        }
    }
}
//wywolanie w main
permute(tab, bestTab, 1, sizeMatrix - 1, min, sizeMatrix, TSPMatrix);
```

Listing 2: Liczenie funkcji celu dla danej permutacji

```
int calculate(int* permut, int size, int **matrix) {
    int sum = 0;
    int i, j;
    for (int iter = 0; iter < size - 1; iter++) {
        i = permut[iter];
        j = permut[iter + 1];
        sum += matrix[i][j];
    }
    sum = sum + matrix[j][permut[0]];

    return sum;
}
```

2.1.2 Drzewo przeszukiwań

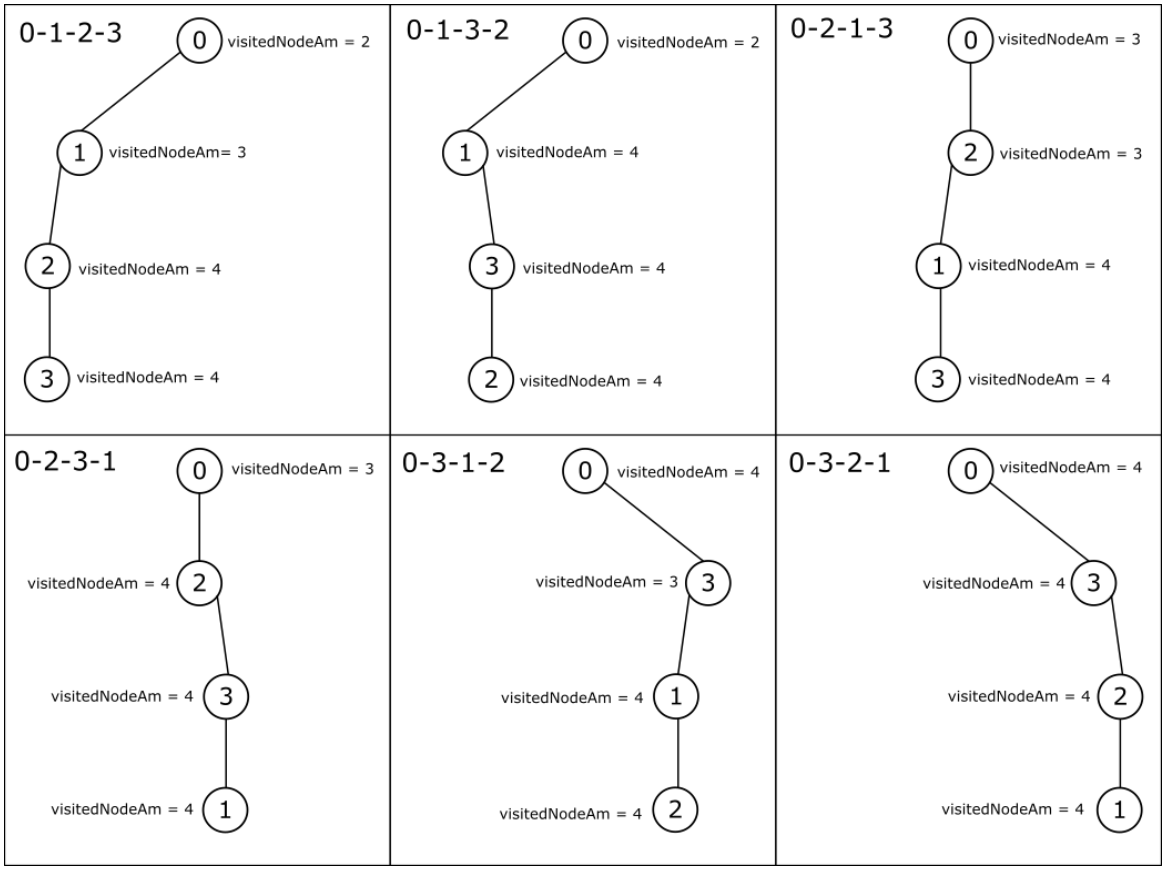
Dodatkowo została zaimplementowana metoda generująca wszystkie permutacje na podstawie drzewa rozwijanego od korzenia, którym jest wierzchołek 0. Algorytm generowania permutacji przedstawia się następująco:

1. Utwórz wierzchołek 0 z następującymi atrybutami (*n* to rozmiar macierzy reprezentującej problem komiwojażera):
   route = 00...00 (*n* zer), *visited* = 100...00 (*n-1* zer), *beforeVisited* = 100...00 (*n-1* zer), *visitedNodeAmount* = 1, *lvl* = 0, *matrixSize* = *n*

2. Zapisanie w *tempLvl* na jakim poziomie drzewa się obecnie znajdujemy i sprawdzane jest *visited* wierzcholka - odczytany jest indeks pod jakim jest pierwsze napotkane 0. Tworzony jest wierzchołek z *route* uwzględniającą przejście, z nową *visited* i taką samą *beforeVisited* uwzględniającą nowy wierzchołek i powiększonym o jeden *visitedNodeAmount* i *lvl*. Ponadto aktualizowane jest *visited* oraz *visitedNodeAmount* ojca,

3. Jeśli uprzednio zinkrementowane *tempLvl* jest mniejsze od *n* to jest powrót do kroku 2, jeśli już się zrównał to idziemy do kroku 4,

4. Wyliczana jest funkcja celu dla danej permutacji i jeśli jest mniejsza od obecnego minimum to minimum jest aktualizowane. Usuwane są po kolei od końca wszystkie wierzchołki, które sąsiadują ze sobą i mają *visitedNodeAmount* równe *n*. Jeśli wektor zawiera wierzchołki to dla ostatniego w wektorze wywoływany jestd krok 2. Jeśli wektor jest pusty to następuje koniec algorytmu.



Rysunek 1: Uproszczone zobrazowanie algorytmu

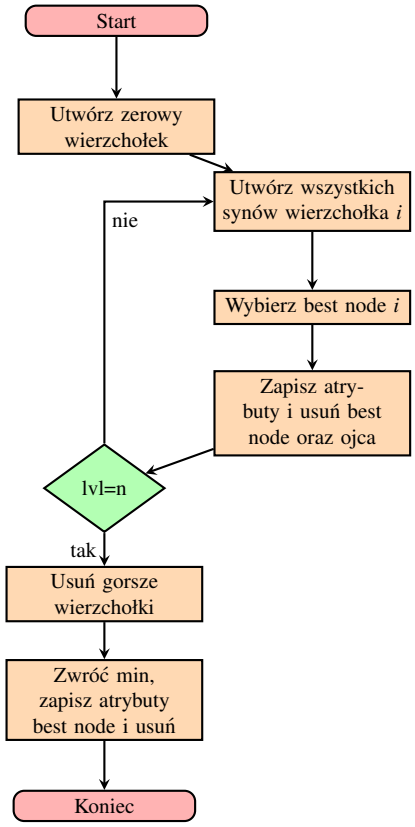
2.2 Branch & Bound

Algorytm *Branch & Bound*, czyli metoda podziału i ograniczeń opiera się na przeszukiwaniu drzewa reprezentującego przestrzeń rozwiązań problemu. Dzięki tak zwanym „odcięciom" można znacznie zredukować liczbę przeszukiwanych wierzchołków. Nazwa *Branch & Bound* wiele mówi o ogólnej koncepcji algorytmu - rozgałęzianie (ang. *branching*) tworzy następników (synów) danego wierzchołka, a ograniczanie (ang. *bounding*) odcina - czyli ściślej rzecz biorąc pomija - gałęzie, które nie doprowadzą nas do optymalnego rozwiązania.

Poszczególne odmiany *Branch & Bound* mogą różnić się się niemalże wszystkimi parametrami - poprzez różne strategie przeszukiwania, bardziej i mniej efektywne funkcje liczenia dolnego ograniczenia (ang. *lower bound*) oraz różne mechanizmy dające nam szybciej górne ograniczenie (ang. *upper bound*).

Napisany program pozwala wybrać strategie przeszukiwania, zgodnie z którą będą wybierane kolejne wierzchołki w drzewie do przeszukania. Ta bardziej efektywna - czyli *Best First* - przy każdej kolejnej iteracji wybiera wierzchołek o obecnie najlepszym dolnym ograniczeniu, co zazwyczaj powoduje przyśpieszenie otrzymania optymalnego rozwiązania. Druga metoda - czyli *Depth First* - po wybraniu wierzchołka z najlepszym dolnym ograniczeniem schodzi wgłąb aż do liścia dając nam za każdym razem końcowe dolne ograniczenie, które opcjonalnie może zastąpić obecne górne ograniczenie. W dalszej części zamiennie będą stosowane wyrażenia *lower bound* i *dolne ograniczenie* oraz *upper bound* i *górne ograniczenie*, a także *syn* i *następnik* oraz *ojciec* i *poprzednik*.

2.2.1 Część pierwsza - wyznaczenie początkowego upper bound



Rysunek 2: Schemat pierwszej części algorytmu Branch & Bound

2.2.2 Pola klasy Node

W programie, każdy utworzony wierzchołek to obiekt należacy do wektora *tree* o następujących atrybutach:

1. **Indeks** - atrybut zapewniający unikalność tworzonych wierzchołków, swoistego rodzaju *klucz główny*,
2. **Id** - atrybut określający jaki to wierzchołek, numeracja od 0 do n-1, gdzie *n* to wielkość macierzy reprezentującej problem komiwojażera,
3. **Wartość** - atrybut przechowujący dolne ograniczenie wierzchołka,
4. **Poziom** - atrybut określający na jakim poziomie drzewa znajduje się wierzchołek,
5. **Macierz** - atrybut przechowujący odpowiednio zredukowaną macierz dla danego wierzchołka,
6. **Ścieżka** - atrybut przechowujący drogę, za pomocą której doszliśmy do danego wierzchołka (włącznie z nim samym).

2.2.3 Algorytm

Zgodnie ze schematem z rysunku 1, w pierwszym kroku utworzony zostaje zerowy wierzchołek. Następnie na każdym kolejnym *i*-tym poziomie, aż do *n-1* tworzymy (*n-1-i*) następników rozwijanego wierzchołka aż do osiągnięcia liścia - czyli poziomu *n-1* - którego wartość ustawiona zostanie jako *upper bound*. Po utworzeniu wszystkich następników na danym poziomie zostaje wybierany ten z najlepszym dolnym ograniczeniem i to on jest rozwijany w kolejnej iteracji, a przy okazji - po zapisaniu jego *indeksu* - usuwany z wektora wierzchołków. Nie ma znaczenia jaki wierzchołek wybierzemy jako startowy, gdyż ostateczna ścieżka tworzy *cykl*, dla uproszczenia przyjęte zostało, że wierzchołkiem startowym (o indeksie 0) jest wierzchołek nr 0 - numeracja wierzchołków od 0 do n-1.

Algorytm obliczający pierwsze dolne ograniczenie dla wierzchołka o indeksie *0* opisana jest w następujący sposób:

1. Zredukuj każdy wiersz poprzez odjęcie minimalnej wartości od każdej komórki,
2. Zredukuj każdą kolumnę poprzez odjęcie minimalnej wartości od każdej komórki,
3. Dodaj wszystkie wartości o jakie została zredukowana macierz (*koszty redukcji*).

Z poniższej minimalizacji dla przykładowej macierzy (przypadek symetryczny) widać, że pierwsze dolne ograniczenie będzie wynosiło $18+21+14+18+14+26+0+0+0+0+0+5$, czyli **116**. Trzecia macierz - najbardziej z prawej strony - posłuży jako macierz do redukcji macierzy dla następników (synów) wierzchołka zerowego.

∞	81	50	**18**	75	39	*18*
81	∞	76	**21**	37	26	*21*
50	76	∞	24	**14**	58	*14*
18	21	24	∞	19	58	*18*
75	37	**14**	19	∞	31	*14*
39	**26**	58	58	31	∞	*26*

→

∞	63	32	0	57	21
60	∞	55	**0**	16	**5**
36	62	∞	10	**0**	44
0	3	6	∞	1	40
61	23	**0**	5	∞	17
13	**0**	32	32	5	∞
0	*0*	*0*	*0*	*0*	*5*

→

∞	63	32	0	57	16
60	∞	55	0	16	0
36	62	∞	10	0	39
0	3	6	∞	1	35
61	23	0	5	∞	12
13	0	32	32	5	∞

Uniwersalna funkcja obliczająca dolne ograniczenie dla poziomów *n>0* wygląda identycznie, lecz przed dokonaniem redukcji należy *zablokować* odpowiedni wiersz i kolumnę. Przykładowo utworzenie wierzchołka o indeksie 5 na poziomie pierwszym, czyli o ścieżce $0 \rightarrow 4$ odbędzie się poprzez zablokowanie wiersza 0-ego i kolumny 4-ej oraz komórki o indeksie [4][0]. Zablokowanie to po prostu ustawienie danych komórek na nieskończoność.

∞	63	32	0	57	16
60	∞	55	0	16	0
36	62	∞	10	0	39
0	3	6	∞	1	35
61	23	0	5	∞	12
13	0	32	32	5	∞

$\rightarrow$

∞	∞	∞	∞	∞	∞
60	∞	55	0	∞	0
36	62	∞	10	∞	39
0	3	6	∞	∞	35
∞	23	0	5	∞	12
13	0	32	32	∞	∞

Po zablokowaniu należy zredukować macierz dokładnie tak jak we wcześniejszym przypadku. W tym przypadku nie jest potrzebna redukcja kolumn, gdyż każda z nich jest już zredukowana, więc finalne koszty redukcji to jedynie koszty redukcji wierszy, czyli **10**.

∞	∞	∞	∞	∞	∞	
60	∞	55	0	∞	**0**	*0*
36	62	∞	**10**	∞	39	*10*
0	3	6	∞	∞	35	*0*
∞	23	**0**	5	∞	12	*0*
13	**0**	32	32	∞	∞	*0*

$\rightarrow$

∞	∞	∞	∞	∞	∞
60	∞	55	**0**	∞	**0**
26	52	∞	0	∞	29
0	3	6	∞	∞	35
∞	23	**0**	5	∞	12
13	**0**	32	32	∞	∞
0	*0*	*0*	*0*		*0*

Ogólny wzór na ograniczenie dolne:

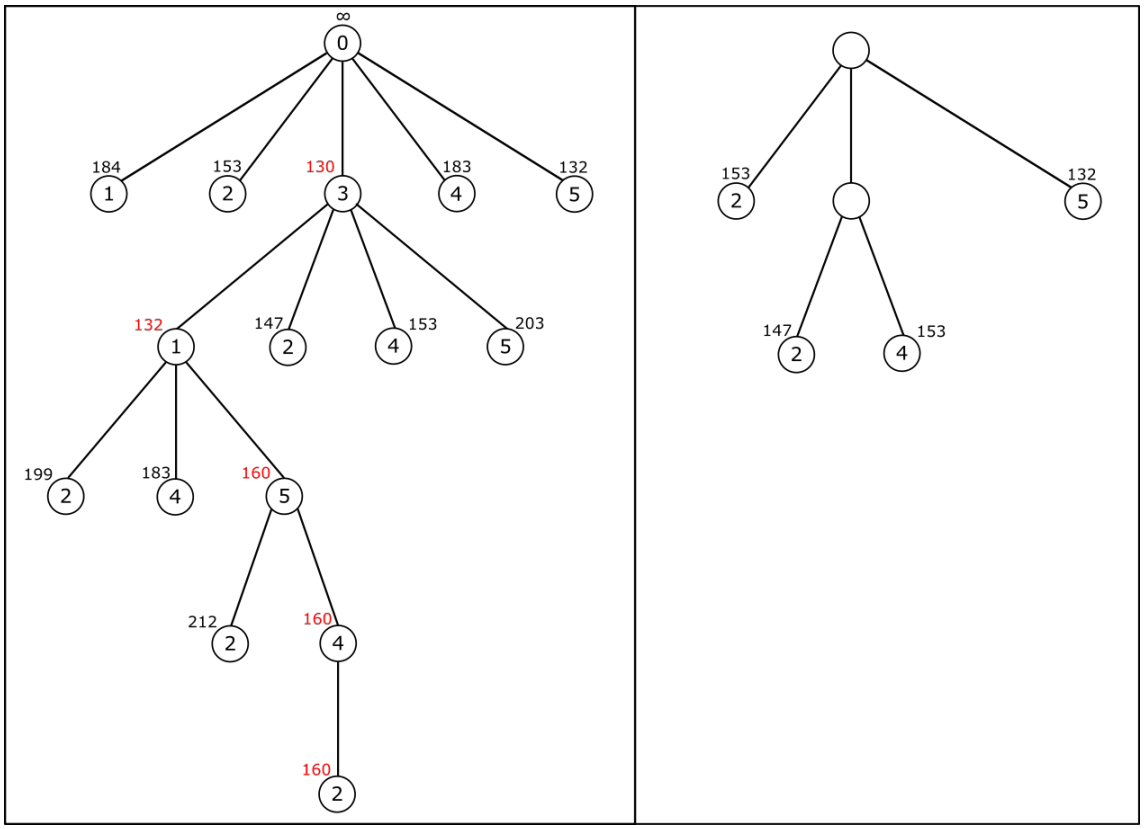
$$lowerbound = cost(\gamma) + reduction + M[A,B] \tag{1}$$

gdzie:
cost(γ) - lower bound ojca,
reduction - koszt redukcji macierzy,
M[A,B] - koszt przejścia z ojca do następnika, pobrany z macierzy ojca.

Więc dla powyższego przypadku *lower bound* = 116 + 10 + 57 = **183**. Za pomocą tej metody dochodzi się do ostatniego - najniższego - poziomu, za każdym razem rozwijając najlepszy wierzchołek (o najniższym *lower bound*) na danym poziomie. Na koniec pierwszej części usuwane są wszystkie wierzchołki o dolnym ograniczeniu większym lub równym wartości dolnego ograniczenia w najniższym poziomie, które jednocześnie stanowi *upper bound* przed rozpoczęciem drugiej części algorytmu.

2.2.4 Schemat działania części pierwszej na przykładzie

Jak widać na załączonym niżej rysunku odcięcia często prowadzą do znacznej redukcji sprawdzanych wierzchołków. Na podstawie macierzy reprezentującej problem komiwojażera zostały wygenerowane kolejne wierzchołki drzewa. Po pierwszej części algorytmu zostało uzyskane rozwiązanie **160** o ścieżce 0-3-1-5-4-2-0, a optymalnym rozwiązaniem jest **158** o ścieżce 0-3-2-4-1-5, co prowadzi do wniosku, że jeśli jest potrzeba minimalizacji czasu wykonywania algorytmu to dla większych instancji można wykonać tylko jego pierwszą część aby uzyskać przybliżone do optymalnego rozwiązanie. "Puste"wierzchołki zostały zachowane dla spójności drzewa, mimo że zostały odcięte z racji większego dolnego ograniczenia.

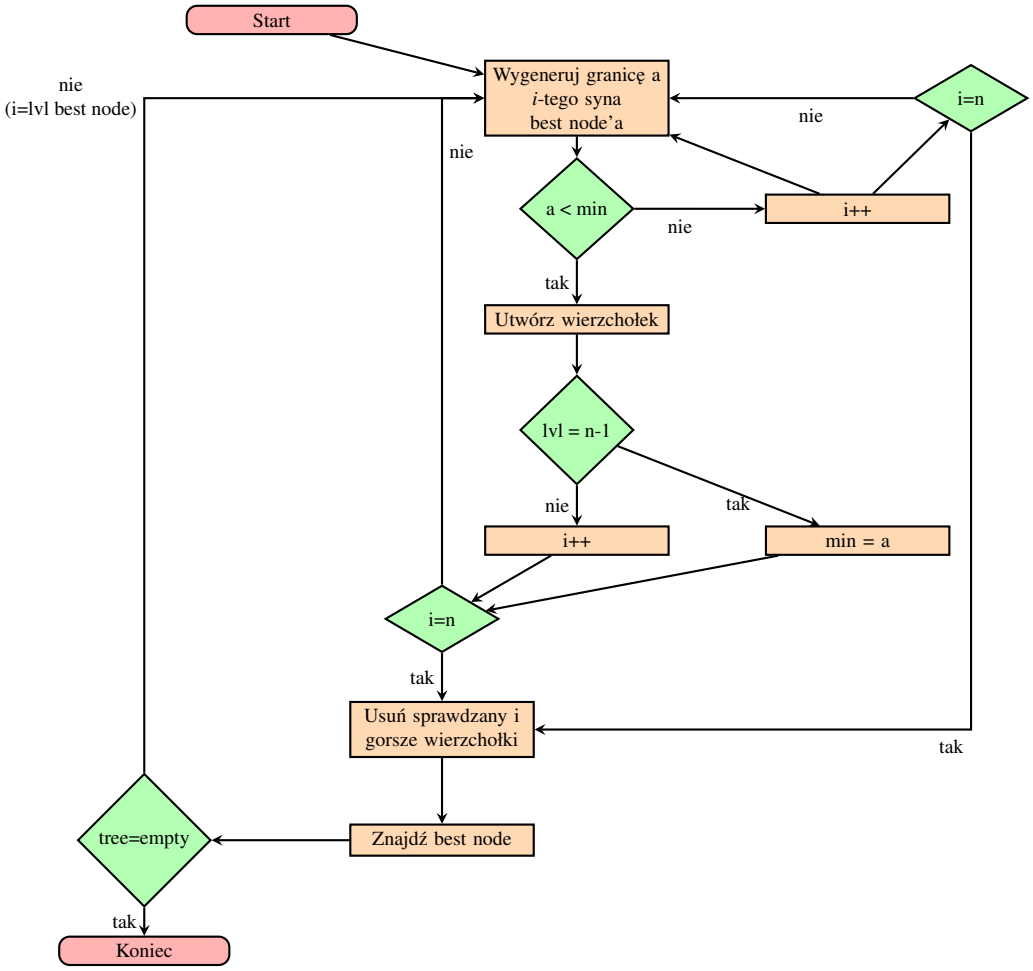


Rysunek 3: Sytuacja po pierwszej części algorytmu i odcięcie

2.2.5 Część druga - wyznaczenie optymalnego rozwiązania

Druga część algorytmu - zgodnie ze schematem na następnej stronie - rozpoczyna się od wygenerowania dolnych ograniczeń dla wszystkich następników (synów) obecnego najlepszego wierzchołka (*best node*). Jeśli dolne ograniczenie jest mniejsze od obecnego *upper bound* to syn zostaje dodany do wektora obiektów wraz z odpowiednimi parametrami. Dodatkowo jeśli został osiągnięty najniższy poziom to aktualizowana jest wartość *upper bound*. Jeśli nie to sprawdzane jest czy sprawdziło się wszystkich synów - jeśli nie to kontynuuowane jest sprawdzanie synów, zaś jeśli tak to usuwany jest ojciec oraz ewentualne gorsze wierzchołki w przypadku zaktualizowania *upper bound*. Na końcu jest znajdowany najlepszy wierzchołek (wersja BF), który będzie rozwijany w nastepnej iteracji bądź wybierany jest najlepszy syn (wersja DF - jeżeli nie istnieje to szukany jest best node) pod warunkiem, że drzewo zawiera jeszcze wierzchołki. Algorytm trwa aż do momentu, gdy w wektorze obiektów nie będzie znajdował się ani jeden wierzchołek.

Kod programu jest zbyt długi, żeby wstawiać jakiekolwiek listingi. Dobrze okomentowany kod, znajduje się w serwisie github, a link do repozytorium można znaleźć w bibliografii.



Rysunek 4: Schemat drugiej części algorytmu Branch & Bound - wersja *Best First*

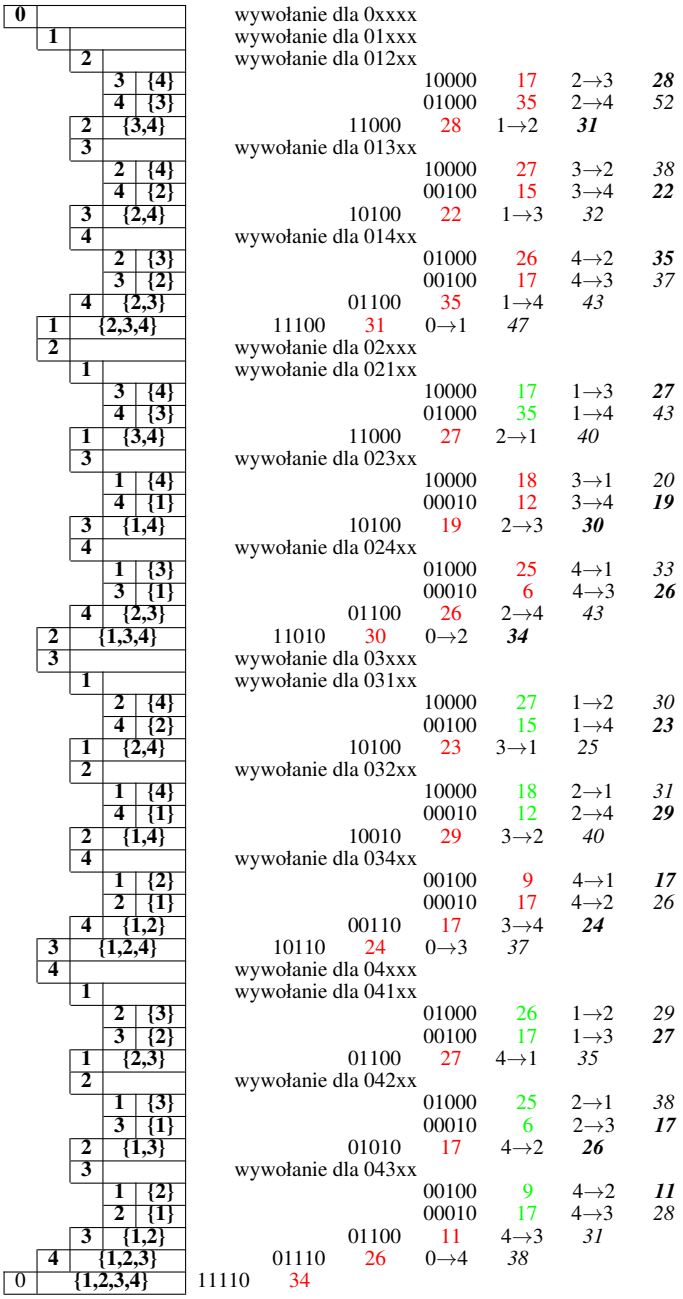
2.3 Programowanie dynamiczne

2.3.1 Macierz reprezentująca przykładowy problem komiwojażera

∞	**16**	**4**	**13**	**12**
4	∞	**3**	**10**	**8**
6	**13**	∞	**11**	**17**
15	**2**	**11**	∞	**7**
10	**8**	**9**	**20**	∞

2.3.2 Schemat działania programu

Kolejną heurystyczną metodą, za pomocą której można rozwiązać problem komiwojażera to metoda *programowania dynamicznego*. Jest to algorytm dokładny, o złożoności obliczeniowej $O(n^2 2^n)$. Koncepcja DP opiera się na uproszczeniu skomplikowanego problemu do prostszych problemów, aż do momentu gdy prostsze problemy stają się tzw. *problemami trywialnymi*, które - w przypadku problemu komiwojażera - będą bezpośrednim pobraniem wartości odległości z macierzy reprezentującej problem.



Schemat działania algorytmu na przykładzie

2.3.3 Idea masek bitowych - podzbiory

W samym algorytmie *programowania dynamicznego* zostały wykorzystane maski bitowe jako sposób na szybkie tworzenie potrzebnych dla danego problemu podzbiorów. Idea jest taka, że rozpatrywane zbiory są przedstawione za pomocą liczby całkowitej, a dokładniej rzecz biorąc jej binarnej postaci. Przykładowo gdy rozpatrywany jest zbiór {1,3,4} to dane o nim są przechowywane tylko w jednej zmiennej o wartości 26, czyli 11010 binarnie. Jeśli element należy do podzbioru to przyjmuje wartość 1 w reprezentacji binarnej na odpowiednim indeksie, a jeśli przeciwnie - to przyjmuje wartość 0. Na maskach bitowych można przeprowadzać różnego rodzaju operacje, a w szczególności jedną, która zwiększy wydajność algorytmu, a mianowicie sumę logiczną.

2.3.4 Algorytm

Jak widać na powyższym schemacie działania algorytmu na przykładowej instancji problemu, działanie programu rozpoczyna się od wywołania głównej funkcji programu dla ścieżki *0xxxx*, która zwróci najlepsze rozwiązanie zaczynające się na wierzchołku 0 i przechodzące przez wszystkie wierzchołki ze zbioru *{1,2,3,4}*. Analogicznie jest wywołyna ta sama funkcja tylko dla ścieżki *01xxx*, która zwróci najlepsze rozwiązanie zaczynające się od wierzchołka 0, następnie 1 i przez resztę wierzchołków ze zbioru *{2,3,4}*. Następnie to samo dla ścieżki *012xx* i tym razem przeszukiwane są permutacje ze zbioru *{3,4}*. Nastepnie analizowane są ścieżki *0123x*. Po tym wywołaniu działanie programu sprowadzane jest do problemów trywialnych.

Dla podwywołania *0123x* do danego wektora wektorów *e* zapisywane jest minimum kosztów z podwywołania ścieżek *0xx34*, które uwzględnia jeszcze przejście z ostatniego wierzchołka do pierwszego - z macierzy odczytywane i sumowane są wartości z komórek o indeksach *[3][4]* oraz *[4][0]*, czyli 7 i 10 co razem daje **17**. Wartość ta zapisywana jest jako najmniejszy koszt, który trzeba ponieść by dojść do wierzchołka 3 po jednoelementowym zbiorze wierzchołków {4}. Miejsce zapisu kosztu określa *następny wierzchołek*, czyli **3** oraz *bitowa reprezentacja zbioru*, w tym przypadku dla zbioru 4 to 10000, a więc dziesiętne **16**. Zapis kosztu odbywa się w komórce o indeksach *[3][16]* i następuje koniec podwywołania dla ścieżek *0123x* i w zmiennej *min* jest zapisywany tymczasowy minimalny koszt ścieżek *0x2{3,4}* równy **28** (17+11, 11 to koszt przejścia z 2 do 3), a wywoływana jest funkcja dla ścieżki *0124x*, która zapisze minimum dla ścieżek *0xx43* i ewentualnie zaktualizuje zmienną *min*, jeśli okaże się mniejsza. Wyliczony koszt dla ścieżek *0xx43* to zsumowane wartości z komórek macierzy o indeksach *[4][3]* oraz *[3][0]*, co finalnie daje **35**. Zapis odbywa się w komórce o indeksie *[4][8]*, bo reprezentacja zbioru *{3}* to 01000, czyli dziesiętnie **8**, zaś następny wierzchołek to **4**. Tu następuje koniec podwywołania dla ścieżek *0124x* i sprawdzane jest czy **52** (35+17, 17 to koszt przejścia z 2 do 4) jest mniejsze od *min*. Koszt ten nie jest mniejszy niż *min*, czyli 28 (52>28), więc pozostawiony jest minimalny koszt 28 w zmiennej *min*. Następuje koniec podwywołania dla ścieżek *012xx* i w komórce o indeksie *[2][24]*- bo reprezentacja zbioru {3,4} to 11000, czyli dziesiętnie **24**, zaś następny wierzchołek to **2** - zapisywana jest wartość *min*, czyli 28.

Tym razem w zmiennej *min* zapisywany jest tymczasowy minimalny koszt dla ścieżek 01xxx jako **31** (28+3), czyli zwrócone *min* z podwywołań plus koszt przejścia z 1 do 2. Zostanie on finalnie ustawiony i zapisany do wektora gdy zostanie zestawiony z kosztami dla ścieżek *013xx* oraz *014xx*, a następnie - po dodaniu odpowiedniej wartości z macierzy (komórka *[0][1]*) - będzie porównywany z kosztami dla ścieżek *02xxx*, *03xxx* oraz *04xxx*, co da końcowe rozwiązanie. Wytłuszczone zostały koszta, które zostały wybrane jako minimalne w obrębie podwywołania dla danego wywołania.

Cała procedura kończy się w momencie zakończenia pierwszego wywołania dla ścieżek *0xxxx*. Ważnym wnioskiem jes to, że w zakresie każdego podwywołania jest wybierane minimum z jego podwywołań, które następnie jest zapisywane do wektora *e*. W pierwszym podwywołaniu dla danego wywołania do zmiennej *min* jest zapisywane ewentualne minimum, które może być zaktualizowane po każdym podwywołaniu, a na zakończenie wszystkich podwywołań wartość z *min* jest zapisywana w pamięci jako minimalny koszt dla ścieżek z danego wywołania. Kolejne podwywołania prowadzą finalnie do problemów trywialnych, które polegają na odczytaniu danych z macierzy reprezentującej problem komiwojażera. Funkcja główna programu daje się bardzo krótko zapisać jeśli tylko wykorzysta się rekurencję. Można by się zastanawiać na czym polega ulepszenie w porównaniu do metody *Brute Force*. Cała idea *programowania dynamicznego* polega na zapisywaniu w pamięci minimów w zakresie podwywołań. Kluczem są tutaj linijki 5-7 przedstawionego niżej *Listing 3*, które upraszczają całą procedurę wyliczania i zwracają koszt jeśli został już wcześniej wyliczony. Przykładowo wyliczony został koszt dla ścieżek *0xx34*, który może zostać wyliczony tylko pierwszym

wywołaniu, czyli *01234*, a wywołanie *02134* nie potrzebuje już obliczeń. W przedstawionym schemacie działania programu na zielono zostały zaznaczone wartości, które zostały pobrane z pamięci - w przeciwieństwie do czerwonych, które zostały wyliczone i zapisane. Dla instancji tej wielkości jeszcze nie widać olbrzymich korzyści jakie niesie ze sobą wykorzystanie dynamicznego programowania, bowiem wraz ze wzrostem wielkości instancji stosunek *zielonych* do *czerwonych* jest coraz większy, a zarazem ilość wartości do obliczenia nie rośnie tak szybko.

Listing 3: Funkcja rekurencyjna

```
int getMinimum(int a, int b, int c, int **d, vector<vector<int>>&e,
                        vector<vector<int>>&f, int &g, int &h, int &j) {
        int min = INT_MAX, tempMin;
        g++;
        if (e[a][b] != -1) {
            return e[a][b];
        }
        else {
            for (int i= 0; i < g; i++) {
                h = pow(2, g) - 1 - pow(2, i);
                j = b & h;
                if (j != b) {
                    tempMin = d[a][i] + getMinimum(i,j,c,d,e,f,g,h,j);
                    if (tempMin < min) { //minimalizacja w zakresie podwywolania
                        min = tempMin;
                        f[a][b] = i;
                    }
                }
            }
        }
        e[a][b] = min;
        return min;
    }
```

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	**34**
1	**4**	∞	∞	∞	**9**	∞	∞	∞	**25**	∞	∞	∞	**27**	∞	∞	∞	**18**	∞	∞	∞	**23**	∞	∞	∞	**27**	∞	∞	∞	**31**	∞	∞
2	**6**	∞	**17**	∞	∞	∞	∞	∞	**26**	∞	**17**	∞	∞	∞	∞	∞	**27**	∞	**29**	∞	∞	∞	∞	∞	**28**	∞	**30**	∞	∞	∞	∞
3	**15**	∞	**6**	∞	**17**	∞	**11**	∞	∞	∞	∞	∞	∞	∞	∞	∞	**17**	∞	**19**	∞	**22**	∞	**24**	∞	∞	∞	∞	∞	∞	∞	∞
4	**10**	∞	**12**	∞	**15**	∞	**17**	∞	**35**	∞	**26**	∞	**35**	∞	**26**	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Tablica 2: Tabela zapisanych wartości poszczególnych zbiorów na koniec wykonania algorytmu

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*	*27*	*28*	*29*	*30*
0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	**2**
1	∞	∞	∞	∞	**2**	∞	∞	∞	**3**	∞	∞	∞	**3**	∞	∞	∞	**4**	∞	∞	∞	**4**	∞	∞	∞	**3**	∞	∞	∞	**2**	∞	∞
2	∞	∞	**1**	∞	∞	∞	∞	∞	**3**	∞	**3**	∞	∞	∞	∞	∞	**4**	∞	**4**	∞	∞	∞	∞	∞	**3**	∞	**3**	∞	∞	∞	∞
3	∞	∞	**1**	∞	**2**	∞	**1**	∞	∞	∞	∞	∞	∞	∞	∞	∞	**4**	∞	**4**	∞	**4**	∞	**4**	∞	∞	∞	∞	∞	∞	∞	∞
4	∞	∞	**1**	∞	**2**	∞	**1**	∞	**3**	∞	**3**	∞	**2**	∞	**2**	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Tablica 3: Tabela kolejnych wierzchołków na koniec wykonania algorytmu

2.3.5 Odtwarzanie ścieżki

Zaprezentowany wyżej algorytm pozwala na uzyskanie optymalnego rozwiązania, ale bez zwrócenia ścieżki prowadzącej do niego. Ponownie wykorzystując rekurencję i poruszając się po tabeli kolejnych wierzchołków z podpunktu 2.3.4 można łatwo odtworzyć ścieżkę.

Listing 4: Odtwarzanie ścieżki

```
1  void getRoute(int start, int set, int size, vector<vector<int>>&possibleRouteTab,
2              int*bestTab, int &c, int &bitMask, int &newSubset) {
3      if (possibleRouteTab[start][set] == -1) {
4          return;
5      }
6      int i = possibleRouteTab[start][set];
7      bestTab[c] = i;
8      c++;
9
10     bitMask = pow(2, size) - 1 - pow(2, i);
11     newSubset = set & bitMask;
12
13     getRoute(i, newSubset, size, possibleRouteTab,bestTab,c,bitMask,newSubset);
14 }
```

1. iteracja: Pobranie następnego wierzchołka *i* z komórki [0][30] - wartość **2** - i stworzenie maski bitowej bez tego wierzchołka (11011). Suma logiczna *bset* i *bbitMask* wytworzyła nowy bset 11010, czyli **26**,
2. iteracja: Pobranie następnego wierzchołka *i* z komórki [2][26] - wartość **3** - i stworzenie maski bitowej bez tego wierzchołka (10111). Suma logiczna *bset* i *bbitMask* wytworzyła nowy bset 10010, czyli **18**,
3. iteracja: Pobranie następnego wierzchołka *i* z komórki [3][18] - wartość **4** - i stworzenie maski bitowej bez tego wierzchołka (01111). Suma logiczna *bset* i *bbitMask* wytworzyła nowy bset 00010, czyli **2**,
4. iteracja: Pobranie następnego wierzchołka *i* z komórki [4][2] - wartość **1**. Koniec odtwarzania, w kolumnie i jest optymalna ścieżka.

start	set	bset	bitMask	bbitMask	i
0	30	11110	27	11011	2
2	26	11010	23	10111	3
3	18	10010	15	01111	4
4	2	00010	-	-	1

Tablica 4: Procedura odtwarzania na przykładzie

3 Eksperymenty obliczeniowe

Obliczenia zastały wykonane na komputerze klasy PC z procesorem i5-7400 ze zintegrowaną kartą graficzną Intel HD Graphics 630, 8GB RAM i DYSK HDD.

3.1 Średnie wyniki dla danych instancji

W ramach eksperymentów obliczeniowych zostały obliczone czasy wykonania zaimplementowanych algorytmów dla instancji podanych przez prowadzącego. Dla każdego rodzaju instancji i dla każdego algorytmu zostało wykonanych po 10 testów, z których wyniki zostały następnie uśrednione.

Instancja	Brute Force [ms]	Branch & Bound DF [ms]	Branch & Bound BF [ms]	Dynamiczne [ms]	Minimum
data10	9,351	0,6895	0,6197	1,541	212
data11	93,33	0,8002	0,7001	3,616	202
data12	996,4	0,5521	0,5011	8,805	264
data13	13450	0,7035	0,7392	20,30	269
data14	-	2,447	1,382	46,09	125
data15	-	9,088	6,382	103,7	291
data16	-	25,17	15,17	239,5	156
data17	-	11710	358,1	556,0	2085
data18	-	62,67	20,71	1336	187

Tablica 5: Tabela uśrednionych czasów wykonania zaimplementowanych algorytmów dla danych instancji

3.2 Średnie wyniki dla losowych instancji

Ponadto, w ramach eksperymentów obliczeniowych zostały za pomocą automatycznych testów zostały utworzone macierze danej wielkości reprezentujące problem komiwojażera. Za pomocą funkcji *rand()* zostały wylosowane pseudolosowe liczby całkowite z przedziału <*10,100*> i wstawione w odpowiednie miejsca w macierzy. Dla każdej wielkości (od minimum 4 do maksimum 24 w zależności od algorytmu) zostało wygenerowanych 50 macierzy (instancji problemu), z których każda została rozwiązana każdym z algorytmów, a wyniki zostały następnie uśrednione oraz zostało wyliczone odchylenie standardowe σ.

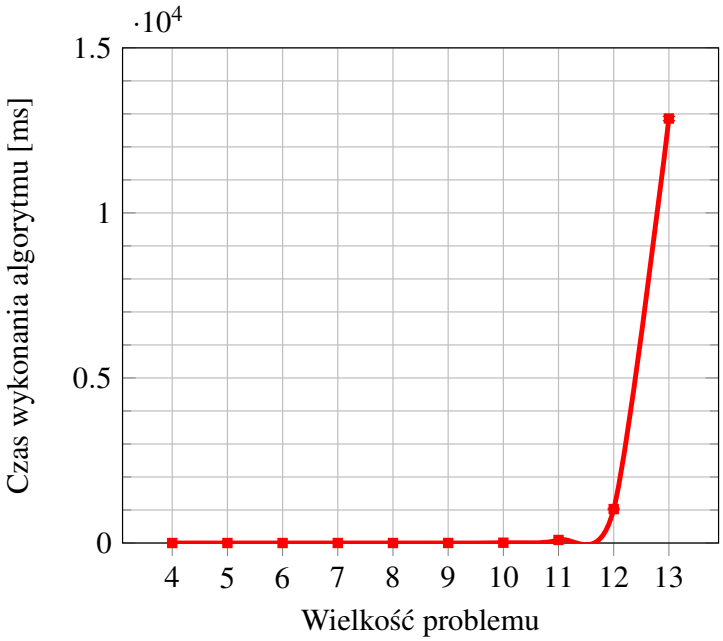
Instancja	Brute Force [ms]		Branch & Bound DF [ms]		Branch & Bound BF [ms]		Dynamiczne [ms]	
	avg	σ	avg	σ	avg	σ	avg	σ
rand4	0,0005152	0,007339	0,03054	0,02150	0,02465	0,01804	0,004402	0,008772
rand5	0,0009671	0,0001231	0,04890	0,01611	0,06919	0,02998	0,02421	0,01498
rand6	0,003498	0,001625	0,06250	0,02648	0,06329	0,02590	0,04532	0,01969
rand7	0,02363	0,006824	0,1080	0,04115	0,09319	0,02936	0,1013	0,04025
rand8	0,1366	0,02029	0,1503	0,07508	0,1438	0,05030	0,2463	0,02189
rand9	0,9690	0,04302	0,2957	0,1620	0,2433	0,08854	0,6229	0,08649
rand10	8,768	0,2230	0,4614	0,2473	0,3667	0,1690	1,514	0,1847
rand11	91,88	2,070	0,6743	0,3603	0,7042	0,4182	3,762	0,5096
rand12	1027	23,31	1,448	0,9128	1,123	0,6396	8,873	0,6679
rand13	12860	65,92	2,476	2,872	1,721	1,274	21,55	1,413
rand14	-	-	4,000	3,706	2,135	1,545	50,70	2,813
rand15	-	-	5,174	6,655	3,657	4,022	119,4	3,652
rand16	-	-	9,622	8,723	5,486	4,313	288,0	5,443
rand17	-	-	17,51	23,71	10,66	10,67	657,1	9,080
rand18	-	-	25,40	33,17	15,10	11,62	1503	26,33
rand19	-	-	55,83	68,70	24,27	20,59	3427	31,26
rand20	-	-	70,27	83,04	33,02	26,29	7776	68,85
rand21	-	-	114,8	156,1	51.98	29,03	17310	596,0
rand22	-	-	149,6	144,5	69,94	52,68	38170	791,5
rand23	-	-	289.6	462,1	74,33	52,53	81790	490,6
rand24	-	-	387.3	548,4	117,4	80,7	-	-

Tablica 6: Tabela uśrednionych czasów wykonania zaimplementowanych algorytmów dla losowych instancji

3.3 Analiza wyników

3.3.1 Brute Force

Do badań został użyty algorytm B.R.Heap'a, który - podczas testów nie zawartych w sprawozdaniu - spisywał się znacznie efektywniej niż algorytm drzewa przeszukiwań, zarówno pod względem pamięciowym jak i obliczeniowym.

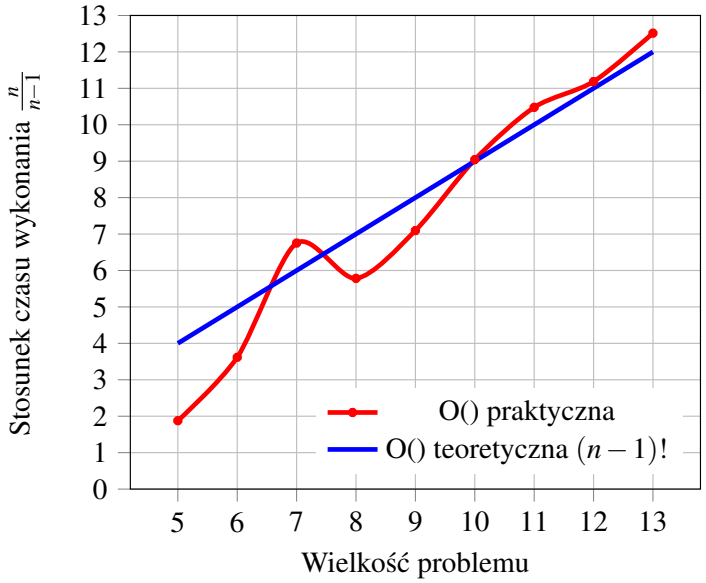


Rysunek 5: Czas wykonania algorytmu w zależności od wielkości problemu

3.3.1.1 Analiza złożoności

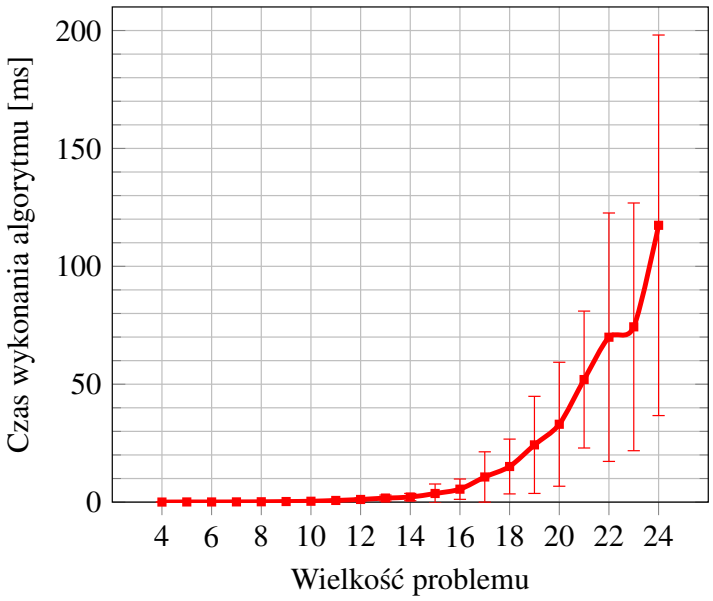
Zgodnie z teoretycznymi przewidywaniami, w algorytm okazał się mieć złożoność obliczeniową $O((n-1)!)$. Na poniższym wykresie niebieska linia reprezentuje spodziewaną, teoretyczną złożoność, czyli czas wykonania dla każdej wielkości n wzrasta *(n-1)*-krotnie w porównaniu do wielkości *(n-1)*. Czerwona linia reprentuje wyniki uzyskane w ramach eksperymentów. Łatwo zauważyć, że wraz z wzrostem wielkości instancji problemu oba wykresy się coraz bardziej zbiegają.

Badania zostały przeprowadzone do instancji wielkości *n=13*, z racji tego, że wykonanie 50-u testów dla *n=14* trwałoby około 140 minut.

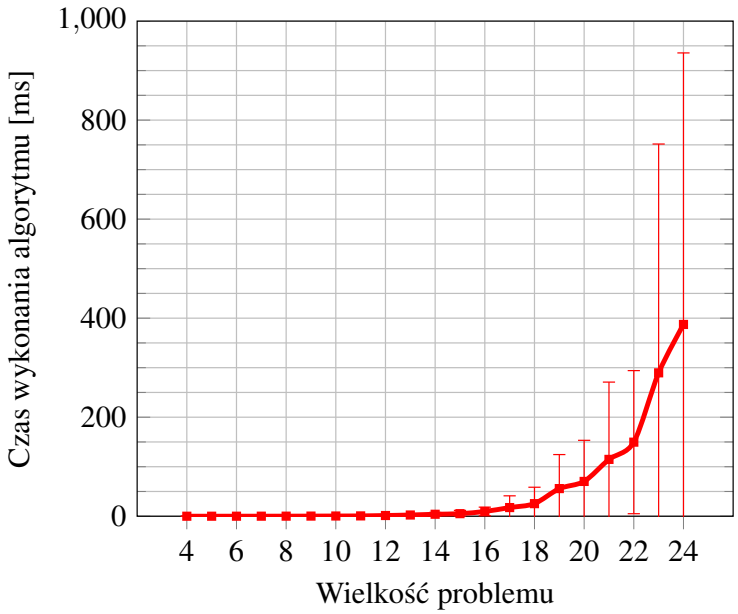


3.3.2 Branch & Bound

Badania algorytmem Branch & Bound zostały przeprowadzone w dwóch opisanych wcześniej metodach - *Best First* oraz *Depth First*. Spośród tych dwóch znacznie korzystniej wypadła pierwsza - *Best First*, która okazała się być mniej efektywna jedynie dla instancji wielkości 5 oraz 6, co zapewne spowodowane jest małą próbką obliczeń. Dla większych instancji spisywała się kilkukrotnie lepiej niż *Depth First*.



Rysunek 6: Czas wykonania algorytmu (wersja BF) w zależności od wielkości problemu



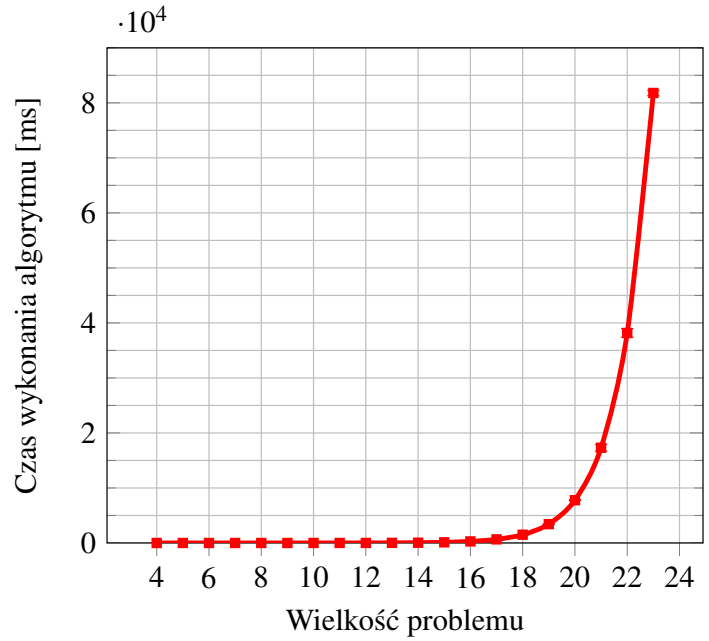
Rysunek 7: Czas wykonania algorytmu (wersja DF) w zależności od wielkości problemu

Ze względu na specyfikę algorytmu - m.in. liczne odcięcia - nie sposób ustalić złożoności obliczeniowej go reprezentującej. W celu zobrazowania tego została załączona tabela prezentująca minimalny i maksymalny czas wykonania w zakresie każdej instancji.

Instancja	**Branch & Bound DF [ms]**			**Branch & Bound BF [ms]**		
	min	avg	max	min	avg	max
rand4	0,0130	0,0305	0,0912	0,0125	0,0247	0,0841
rand5	0,0215	0,0489	0,107	0,0213	0,0692	0,182
rand6	0,0326	0,0625	0,132	0,0325	0,0633	0,145
rand7	0,0511	0,108	0,209	0,0521	0,0932	0,192
rand8	0,0723	0,150	0,386	0,0743	0,144	0,332
rand9	0,104	0,296	0,819	0,126	0,243	0,438
rand10	0,145	0,461	1,11	0,165	0,367	0,833
rand11	0,199	0,674	1,63	0,198	0,704	1,96
rand12	0,323	1,45	4,12	0,307	1,12	2,51
rand13	0,317	2,48	15,4	0,339	1,72	7,46
rand14	0,407	4,00	16,9	0,470	2,14	8,19
rand15	0,900	5,17	39,8	0,601	3,66	20,1
rand16	1,11	9,62	36,8	0,688	5,49	16,3
rand17	0,859	17,5	115	1,36	10,7	52,9
rand18	1,61	25,4	130	1,06	15,1	55,7
rand19	2,77	55,8	369	1,24	24,3	79,5
rand20	3,01	70,3	440	3,76	33,0	126
rand21	4,12	115	705	5,82	52,0	121
rand22	7,60	150	735	6,12	69,9	256
rand23	5,91	290	5009	13,1	74,3	206
rand24	16,1	387	2609	5,98	117	351

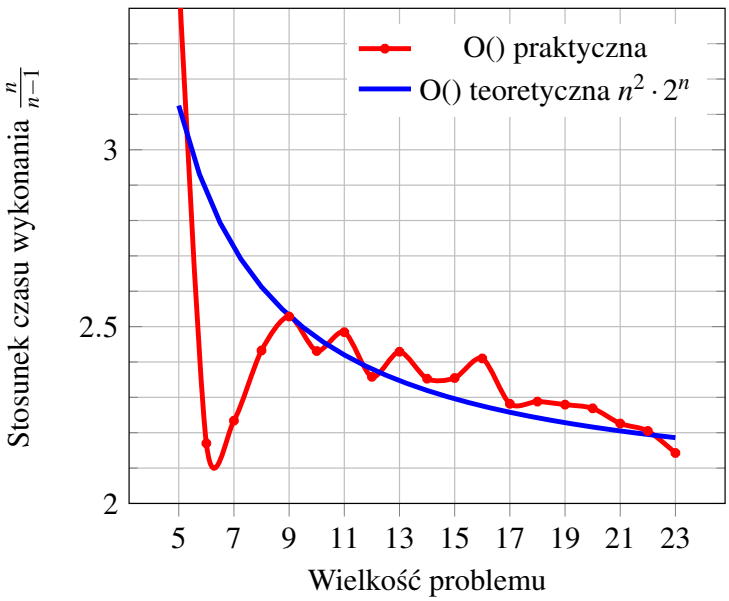
Tablica 7: Czasy wykonania algorytmu Branch & Bound

3.3.3 Programowanie dynamiczne



3.3.3.1 Analiza złożoności

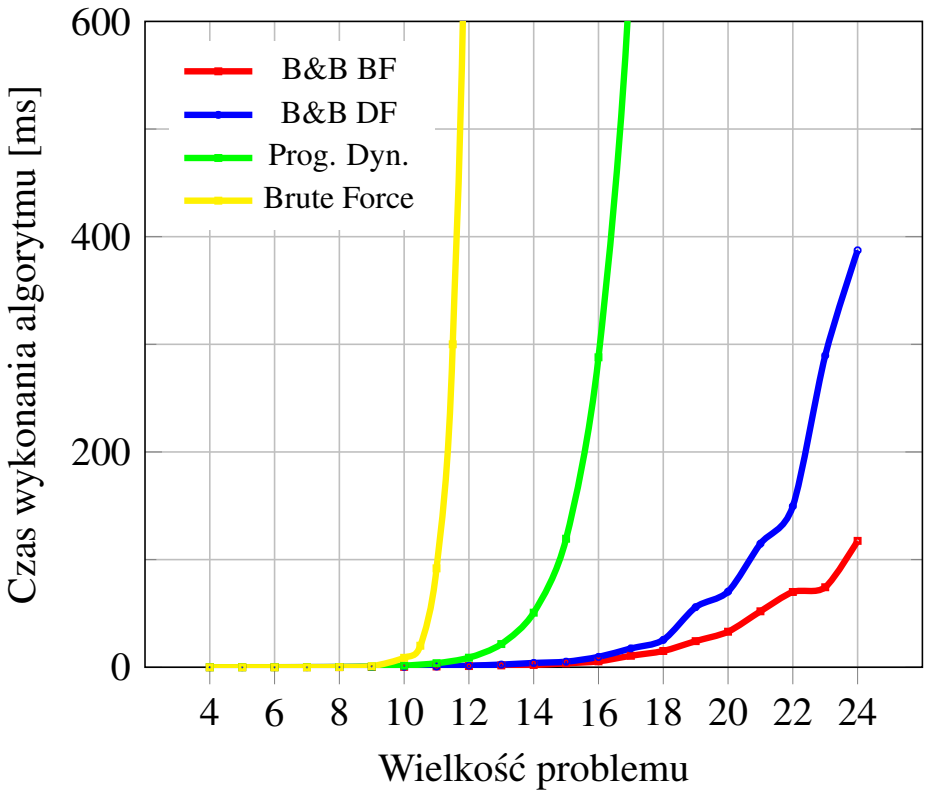
Złożoność okazała się być taka jak zakładana - $O(n^2 \cdot 2^n)$. Czerwona linia na wykresie reprezentuje wyniki uzyskane w ramach eksperymentów, a niebieska - zakładaną, teoretyczną złożoność. Czas wykonania algorytmu dla instancji n w porównaniu do n-1 wynosi $\frac{n^2 \cdot 2^n}{(n-1)^2 \cdot 2^{n-1}}$, czyli $2 \cdot (\frac{n^2}{(n-1)^2})$, a więc - teoretycznie - dla dużych n czas powinien rosnąć lekko ponad dwukrotnie. Podobnie jak w przypadku *Brute Force* dla coraz większych n wykresy zbiegają się.



4 Wnioski

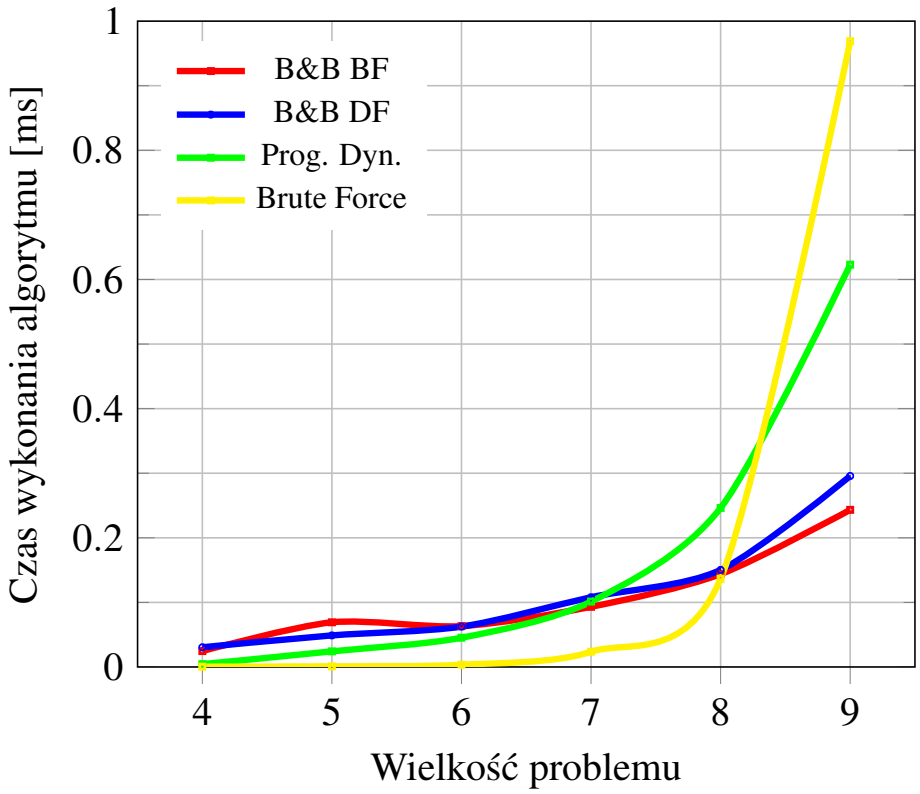
Sumarycznie - dla większych instancji - porównując wszystkie zaimplementowane algorytmy dokładne najlepiej wypadła wersja *Best First* algorytmu *Branch & Bound*. Niewiele mniej efektywna okazała się być druga wersja tego samego algorytmu - *Depth First*. Są to jednak metody o dość dużej złożoności pamięciowej, ponieważ wraz ze wzrostem wielkości problemu pamięć operacyjna jest zmuszona do przechowywania coraz większej ilości informacji o wierzchołkach i *Microsoft Visual Studio 2017* zwraca po pewnym czasie błędy w alokacji pamięci. Algorytm *Branch & Bound* może nadawać się do liczenia większych instancji problemu, gdyż powstają coraz doskonalsze funkcje liczenia dolnego ograniczenia, które znacznie zwiększają liczbę odcięć, jednak są trochę badziej skomplikowane w implementacji. Dwa pozostałe algorytmy mogą być trochę bardziej wydajne jedynie jeśli będziemy dysponować komputerem z lepszym procesorem. Jeśli chodzi o implementację to najbardziej skomplikowana okazała się być implementacja *Branch & Bound*, która pochłonęła dużo czasu na testy oraz różne poprawki, pozostałe algorytmy były dosyć proste - jedynie algorytm *programowania dynamicznego* wymagał zauważenia pewnych rzeczy, m.in. wychwycenia rekurencji tak by implementacja była jak najkrótsza, a sam algorytm jak najbardziej efektywny.

W celu porównania algorytmów, dla każdego zestawu wyników zostały wyliczone odchylenia standardowe. Dla obu wersji algorytmu *Branch & Bound* wraz ze wzrostem wielkości problemu zaczęły przerastać średnie czasy potrzebne na rozwiązanie problemu, co widać na zamieszczonych wykresach. Przykładowo w zakresie instancji *rand23* problem został najkrócej rozwiązany w czasie *0,00591s*, a najdłużej w czasie *5,009s*, co można odczytać z tabeli nr 7. Dla kontrastu, w dwóch pozostałych algorytmach odchylenia były bardzo małe - w algorytmie *programowania dynamicznego* dla większych instancji oscylowały w granicach *1%* średniego czasu wykonania, a w *Brute Force* dla *n=12* stosunek ten wynosił około *2%*, a dla *n=13* już lekko ponad *0,5%*, sukcesywnie malejąc w szybkim tempie.



Rysunek 8: Porównanie czasów wykonania algorytmów

Dla $n \leq 9$ czasy rozwiązania problemu każdym z algorytmów są podobne i zostały ukazane na poniższym wykresie. Widać, że dla $n \leq 7$ obie wersje algorytmu *Branch & Bound* wypadają mniej efektywnie niż *Brute Force* i *programowanie dynamiczne*. Jednak już dla $n=7$ czas wykonania *programowania dynamicznego* i dla $n=8$ czas wykonania *Brute Force* zrównują się z algorytmem *Branch & Bound*, by później - zgodnie z swoją złożonością obliczeniową - dynamicznie poszybować w górę.



Rysunek 9: Porównanie czasów wykonania algorytmów dla małych instancji ($n \leq 9$)

Bibliografia

[1] D. Applegate, W Cook, S. Dash, and D. Johnson. A Practical Guide to Discrete Optimization. pages 44–51, December 2014.

[2] Robert Kruse. *Data structures and program design*. Prentice-Hall, 1984.

[3] Radosław Lis. Listing kodu. `https://github.com/radosz99/PEA-etap1/`. [Online; accessed 24-November-2019].

[4] Chaitanya Pothineni. Travelling Salesman Problem using Branch and Bound Approach. pages 3–7, December 2013.