Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Projektowanie Efektywnych Algorytmów

Projekt – zadanie 1.

Autor:

Stanisław Strauchold 259142

Prowadzący:

dr inż. Jarosław Mierzwa

Grupa:

Środa 11:15 – 13:00 – grupa wcześniejsza

Termin oddania:

16.11.2022

1. Wstęp teoretyczny

Problemem badanym w tym zadaniu był problem komiwojażera (TSP). Jest to zagadnienie optymalizacyjne, polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Problem ten można przedstawić z punktu widzenia wędrownego sprzedawcy: dane jest n miast, które komiwojażer ma odwiedzić oraz odległość pomiędzy każdą parą miast. Celem jest znalezienie najkrótszej drogi łączącej wszystkie miasta, zaczynającej się i kończącej w tym samym punkcie. W naszym przypadku mamy do czynienia z asymetrycznym problemem komiwojażera (ATSP), co oznacza, że dla dowolnych miast A i B odległość z A do B może być różna od odległości z B do A.

- cykl Hamiltona – taki cykl w grafie, w którym każdy wierzchołek grafu odwiedzany jest dokładnie raz (oprócz pierwszego wierzchołka).
- pełny graf ważony – graf, w którym każdy wierzchołek jest połączony krawędzią ze wszystkimi pozostałymi wierzchołkami. Każda z tych krawędzi posiada wagę.

1.2. Wykorzystywane algorytmy znajdywania najkrótszego cyklu Hamiltona

1.2.1. Brute Force (przegląd zupełny)

Algorytm przeglądu zupełnego polega na znalezieniu każdego możliwego cyklu Hamiltona oraz policzeniu dla niego sumy wag krawędzi. Następnie wybierany jest ten, którego suma krawędzi jest najmniejsza. Taki algorytm posiada wykładniczą złożoność obliczeniową O(n!).

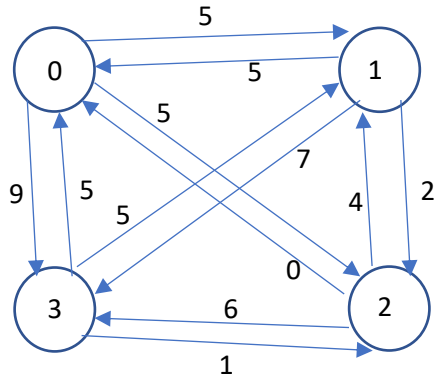
1.2.2. B&B (metoda podziału i ograniczeń)

W tym algorytmie definiujemy drzewo przestrzeni stanów, które w korzeniu zawiera listę zawierającą tylko wierzchołek początkowy. Na pierwszym poziomie znajdują się węzły zawierające listy dwóch wierzchołków – do listy dodajemy kolejny wierzchołek po początkowym. Na kolejnych poziomach węzły zawierają ścieżki dłuższe o 1 względem poprzednich poziomów, aż do liści, które zawierają listy (n-1) elementowe, gdzie n jest liczbą wierzchołków w grafie. Przeglądamy kolejne węzły obliczając ich granice. W moim przypadku zastosowana została metoda „przeszukiwanie najpierw najlepszy”, co oznacza, że po przejrzeniu synów jakiegoś węzła przeglądamy wszystkie pozostałe obiecujące węzły (takie, których granica jest mniejsza od najlepszej znalezionej) i rozwijamy węzeł o najlepszej granicy. Pesymistyczna złożoność algorytmu wynosi O(n!), zakłada ona, że każda kolejna przeszukiwana ścieżka będzie miała wartość granicy mniejszą od poprzedniej. Optymistyczna złożoność wynosi O(2n), zakłada ona, że pierwsza ścieżka będzie miała granicę o niższej wartości niż granice pozostałych synów korzenia w drzewie.

Pojęciem używanym przy okazji algorytmu B&B jest granica. Jest to liczba będąca ograniczeniem wartości rozwiązania jakie można uzyskać dzięki rozwinięciu (przejrzeniu potomków) danego węzła.

2. Przykład praktyczny

W niniejszej sekcji przedstawiony jest opis działania algorytmu B&B metodą „przeszukiwanie najpierw najlepszy” krok po kroku dla przykładowej instancji danego problemu o wartości N = 4.



Pierwszym krokiem algorytmu po inicjalizacji zmiennych pomocniczych jest utworzenie zredukowanej macierzy sąsiedztwa dla wierzchołka 0, który jest korzeniem w naszym drzewie przestrzeni stanów. Obliczany jest dla niego również koszt granicy, który jest sumą redukcji wierszy i kolumn w macierzy początkowej.

Macierz redukowana jest w następujący sposób:

- Dla każdego wiersza macierzy wybierana jest krawędź o najmniejszej wadze
- Od każdej krawędzi występującej w wierszu odejmowana jest znaleziona najmniejsza waga krawędzi
- Dla każdej kolumny macierzy wybierana jest krawędź o najmniejszej wadze
- Od każdej krawędzi występującej w kolumnie odejmowana jest znaleziona najmniejsza waga krawędzi
- Najmniejsze wartości w każdej kolumnie oraz wierszu są sumowane i w ten sposób otrzymujemy wartość granicy dla węzła

Do tego każdy węzeł przechowuje również ścieżkę reprezentującą drogę. W tym wypadku ścieżka będzie wyglądać następująco: 0.

Wyniki prezentują się następująco:

```
Wezel: 0
0  0  0  0
3  0  0  1
0  4  0  2
4  4  0  0
Koszt: 12
```

Następnie dla wszystkich synów korzenia wywoływana jest metoda, która jest odpowiedzialna za wyliczenie granicy dla każdego z nich. Ścieżka dla każdego z synów powiększana jest o odpowiedni wierzchołek. Macierz należąca do syna początkowo jest kopią macierzy należącej do rodzica. Na początku w macierzy wartości w wierszu odpowiedzialnym za ostatni element ścieżki rodzica oraz kolumnie odpowiedzialnej za ostatni element ścieżki syna wszystkie wartości zmieniane są na inf, co jest reprezentowane przez wartość -1. Potem przechodzimy do redukcji macierzy, która odbywa się na tej samej zasadzie, co redukcja opisana wyżej dla korzenia. Przy redukcji wartości -1 są pomijane. Granica dla każdego z synów obliczana jest w następujący sposób: granica rodzica + suma minimalnych wartości znalezionych przy redukcji macierzy + wartość krawędzi łączącej ostatni wierzchołek w ścieżce rodzica oraz ostatni wierzchołek w ścieżce syna.

Wyniki dla tego kroku prezentują się następująco:

Wartości wypisane przy polu węzeł oznaczają ostatni element ścieżki dla danego węzła. Przykładowo w pierwszym przykładzie ścieżka prezentuje się następująco: 0 -> 1, stąd przy polu węzeł występuje liczba 1.

```
Wezel: 1
-1  -1  -1  -1
-1  -1  0  0
0  -1  0  1
4  -1  0  0
Koszt: 13
Wezel: 2
-1  -1  -1  -1
2  0  -1  0
-1  2  -1  0
0  0  -1  0
Koszt: 19
Wezel: 3
-1  -1  -1  -1
3  0  0  -1
0  0  0  -1
-1  0  0  -1
Koszt: 16
```

Na koniec metoda sprawdza czy ścieżka przechowywana przez syna jest równa liczbie wierzchołków w grafie. Jeśli tak, to nastąpi wyjście z metody, ponieważ doszliśmy do końca cyklu Hamiltona. Zmiennej upper oznaczającej najmniejszą znalezioną granicę przypisywana jest wartość znalezionej granicy, jeżeli jest ona mniejsza od aktualnej wartości zmiennej upper. Jeżeli nie, węzeł dodawany jest do kolejki priorytetowej przechowującej węzły.

Następnie metoda licząca granicę wywoływana jest dla kolejnych węzłów w kolejce. Warto tutaj zaznaczyć, że elementy w kolejce sortowane są rosnąco według wartości obliczonej granicy.
Najmniejsza wartość granicy przechowywana jest dla pierwszego syna korzenia, którego ścieżka prezentuje się następująco: 0->1. Pierwszy syn tego węzła do ścieżki dopisze wierzchołek 2, gdyż jest on pierwszym z nieodwiedzonych jeszcze w danej ścieżce wierzchołków, a drugi z synów (będzie tylko dwóch, ponieważ do odwiedzenia pozostały nam jedynie 2 wierzchołki) dopisze wartość 3, stąd takie wartości dla obu synów występują przy polu węzeł.
Po wywołaniu funkcji liczącej granicę dla obu synów wyniki prezentują się następująco:

```
Wezel: 2
-1  -1  -1  -1
-1  -1  -1  -1
0  -1  -1  0
0  -1  -1  0
Koszt: 18
Wezel: 3
-1  -1  -1  -1
-1  -1  -1  -1
0  -1  0  -1
4  -1  0  -1
Koszt: 14
```

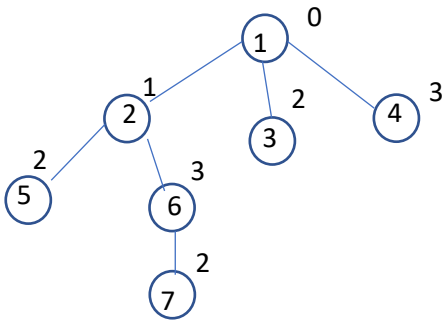
Oba węzły dodawane są do kolejki, gdyż ich ścieżki nie zawierają jeszcze 4 wierzchołków. Następnie ponownie brany jest pierwszy element kolejki. Tym razem jest to drugi z przeglądanych przed chwilą synów. Po wywołaniu dla niego metody liczącej granicę, wyniki prezentują się następująco:

```
Wezel: 2
-1  -1  -1  -1
-1  -1  -1  -1
0  -1  -1  -1
-1  -1  -1  -1
Koszt: 14
```

Ścieżka w tym przypadku zostaje już zapełniona, zatem wartość granicy porównywana jest z wartością przechowywaną przez zmienną upper. Jest ona mniejsza, zatem zmienna upper przyjmuje wartość 14, a aktualnie najkrótszy cykl Hamiltona ustawiany jest jako ścieżka przechowywana przez odwiedzany węzeł.
W kolejce priorytetowej nie znajdują się już żadne węzły o granicy mniejszej niż 14, zatem są one pomijane.
Na koniec na podstawie zapisanej najkrótszej ścieżki liczona jest jej wartość przy wykorzystaniu macierzy użytej do wczytania grafu do pamięci komputera. Ostateczne wyniki prezentują się następująco:

```
dlugosc: 13
sciezka: 0 -> 1 -> 3 -> 2 ->
```

Aby przejrzyściej opisać proces odwiedzania kolejnych węzłów poniżej pokazane jest drzewo uzyskane w trakcie działania naszego algorytmu. W środku węzłów przechowywany jest numer odpowiedzialny za kolejność jego powstania, natomiast z prawej strony przechowywany jest aktualnie dodawany do ścieżki wierzchołek grafu.



3. Opis implementacji algorytmu

3.1. Brute Force

Metoda bruteForce w programie służy do rekurencyjnego rozwiązywania problemu komiwojażera. Algorytm sprawdza po kolei wszystkie cykle Hamiltona. Pierwsze wywołanie metody następuje w metodzie main() programu, gdzie jest ona wywoływana z parametrem równym wierzchołkowi początkowemu grafu. Zmienne pomocnicze metody to:

- stackH – stos zawierający aktualnie badaną ścieżkę
- visited[] – tablica wartości logicznych reprezentująca przeszukane już wierzchołki
- A – macierz przechowująca graf
- dh – zmienna przechowująca aktualną długość ścieżki
- d – zmienna przechowująca najkrótszą znalezioną ścieżkę
- sciezka[] – tablica przechowująca ścieżkę, dla której koszt drogi jest najmniejszy dotąd znaleziony

```
stackH->push(v);
```

Na początku wierzchołek, na rzecz którego wywoływana jest algorytm jest dodawany do stosu przechowującego aktualnie badaną ścieżkę

```
if (stackH->n == n)
```

Następnie sprawdzane jest czy ścieżka jest już pełna, to znaczy czy zawiera wszystkie wierzchołki. Jeżeli tak, to do kosztu ścieżki dodawana jest waga krawędzi łączącej ostatni z wierzchołków na naszej ścieżce z wierzchołkiem 0.

```
dh = dh + A[v][0];
```

Następnie sprawdzany jest warunek, czy koszt znalezionej ścieżki jest mniejszy od najmniejszego znalezionego dotychczas kosztu.

```
if (dh < d)
{
	d = dh;
	if (sciezka != NULL)
	{
		delete[] sciezka;
		sciezka = NULL;
	}
	int pomoc = stackH->n;
	sciezka = new int[pomoc];
	for (int i = 0; i < pomoc; i++)
	{
		sciezka[i] = stackH->pop();
	}
	for (int i = pomoc - 1; i >= 0; i--)
	{
		stackH->push(sciezka[i]);
	}
}
```

Jeżeli warunek jest prawdziwy, to jeżeli istniała już jakaś znaleziona ścieżka końcowa, jest ona usuwana z pamięci. Następnie tworzona jest nowa ścieżka końcowa poprzez pobieranie kolejnych elementów ze stosu reprezentującego aktualnie przeszukiwaną ścieżkę. Następnie z racji na to, iż żeby pobrać elementy ze stosu, musieliśmy je jedocześnie z niego usunąć, to ścieżka jest z powrotem kopiowana na stos w takiej samej kolejności w jakiej wcześniej się w nim znajdowała

Następuje wyjście z warunku.

```
dh = dh - A[v][0];
```

Od aktualnie przechowywanego kosztu odejmowana jest krawędź łącząca wierzchołek początkowy z wierzchołkiem końcowym.

Wychodzimy z warunku, który sprawdzał czy nasza aktualnie przeszukiwana ścieżka zawiera już wszystkie wierzchołki.

Przechodzimy do fragmentu kodu, który wywoływany jest kiedy aktualnie przeszukiwana ścieżka nie zawiera wszystkich wierzchołków grafu.

```
visited[v] = true;
```

Przekazany jako parametr metody wierzchołek oznaczamy jako przeszukany.

```
for (int j = 0; j < n; j++)
{
	if (A[v][j] != -1)
	{
		if (visited[j] == true)
			continue;
		else
		{
			dh = dh + A[v][j];
			bruteForce(j);
			dh = dh - A[v][j];
		}
	}
}
```

Następnie dla wszystkich jego sąsiadów, którzy nie są zawarci w aktualnej ścieżce, wywołana jest metoda bruteForce, która jako parametr przyjmuje aktualnie przeszukiwanego sąsiada wierzchołka przekazanego w poprzednim jej wywołaniu. Wcześniej do aktualnego kosztu drogi dodawana jest waga krawędzi między wierzchołkiem, a jego przeszukiwanym sąsiadem.

Po powrocie z wywołanej rekurencyjnie metody następuje odjęcie od kosztu ścieżki wagi między krawędzią prowadzącą z wierzchołka z poprzedniego wywołania bruteForca do wierzchołka z wywołania bruteForce, z którego właśnie wróciliśmy.

```
visited[v] = false;
```

Po przeszukaniu wszystkich sąsiadów danego wierzchołka, następuje jego wykreślenie z tablicy reprezentującej wierzchołki już odwiedzone.

```
stackH->pop();
```

Na samym końcu metody następuje zdjęcie ostatniego wierzchołka z aktualnie przeszukiwanej ścieżki. Jest ono wywołane za każdym razem na końcu metody.

Algorytm rekurencyjny ciężko jest opisać słowami, tak aby można było go z łatwością zrozumieć. Z tego powodu na koniec warto dodać ,iż działanie algorytmu można porównać do algorytmu przeszukiwania w głąb drzewa przestrzeni stanów, w którym węzły oznaczają wierzchołki grafu, a każda droga od korzenia do liścia jest inną ścieżką zawierającą wszystkie wierzchołki.

3.2. B&B

Metoda B_B w programie jest metodą podziału i ograniczeń, kierująca się strategią „najpierw najlepszy”. W algorytmie używane są następujące zmienne pomocnicze:

- A – macierz reprezentująca graf, w którym należy znaleźć cykl Hamiltona o najmniejszym koszcie
- upper – zmienna przechowująca najmniejszą znaleziona granicę
- A1 – macierz pomocnicza służąca do przechowywania zredukowanej macierzy A
- struct wezel – struktura reprezentująca węzeł w drzewie przestrzeni stanów, zawiera następujące pola:
  - B – zredukowana macierz dla każdego węzła/stanu
  - cost – granica dla każdego węzła/stanu
  - sciezka – tablica przechowującą ścieżkę dla każdego stanu/węzła
  - parent – wskaźnik na rodzica danego węzła
  - dlugosc – zmienna przechowujaca długość ścieżki w danym stanie/węźle
- visited – tablica reprezentująca już przeszukane węzły

Pierwszym krokiem algorytmu jest skopiowanie macierzy początkowej do macierzy A1 reprezentującej macierz zredukowaną dla korzenia drzewa. Następnie następuje zredukowanie początkowej macierzy reprezentującej graf. W tym celu szukana jest minimalna krawędź w każdym wierszu, a następnie jest ona odejmowana od każdej krawędzi znajdującej się w tym wierszu. Następnie analogicznie minimalizowana jest każda kolumna.

```
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < n; j++)
		{
			if (A1[i][j] < min && i != j)
				min = A1[i][j];
		}
		for (int j = 0; j < n; j++)
		{
			if(A1[i][j] != 0)
				A1[i][j] -= min;
		}
		cost += min;
		min = 2147483647;
	}
	for (int i = 0; i < n; i++)
	{
		for (int j = 0; j < n; j++)
		{
			if(A1[j][i] < min && i != j)
				min = A1[j][i];
		}
		for (int j = 0; j < n; j++)
		{
			if(A1[j][i] != 0)
				A1[j][i] -= min;
		}
		cost += min;
		min = 2147483647;
	}
```

Następnie tworzony jest węzeł reprezentujące korzeń w drzewie stanów.

```
wezel* wezel1 = new wezel;
wezel1->B = A1;
wezel1->sciezka = new int[1];
wezel1->sciezka[0] = 0;
wezel1->dlugosc = 1;
wezel1->cost = cost;
wezel1->parent = NULL;
```

Wartość granicy dla korzenia równa się sumie znalezionych wcześniej minimalnych wartości.

Następnie dla każdego syna korzenia, czyli każdego wierzchołka, do którego można dojść z wierzchołka początkowego, wywoływana jest funkcja odpowiedzialna za stworzenie. Jest to jednocześnie funkcja licząca dla niego granicę.

```
for (int i = 1; i < n; i++)
{
	bound(wezel1, i);
}
```

W tym momencie omówiona zostanie funkcja bound().

Na początku każdemu węzłowi przypisywana jest zredukowana macierz rodzica, długość przechowywanej w tym węźle ścieżki oraz aktualnie przeszukiwana ścieżka.

```
wezel* syn = new wezel;
	syn->parent = parent;
	syn->B = copy_matrix(parent->B);
```

```
	int m = parent->dlugosc;
	syn->dlugosc = m + 1;
	int k = i;
	syn->sciezka = new int[m + 1];
	for (int j = 0; j < m; j++)
		syn->sciezka[j] = parent->sciezka[j];
	syn->sciezka[m] = i;
	syn->cost = 0;
```

Następnie zerowany jest wiersz reprezentujący poprzedni element ścieżki oraz kolumnę reprezentującą aktualnie dodany element ścieżki. Zerowana jest także krawędź reprezentująca połączenie aktualnie dodanego do ścieżki wierzchołka z wierzchołkiem początkowym.

Potem następuje redukcja macierzy identyczna jak dla korzenia, stąd nie będę jej omawiał.

```
syn->cost += A1[parent->sciezka[m - 1]][i] + parent->cost;
```

Po redukcji następuje wyliczenie granicy dla danego węzła. Jest ona granicy rodzica plus wadze krawędzi łączącej dwa ostatnie elementy w ścieżce plus sumie znalezionych w wierszach i kolumnach wartości minimalnych.

```
if (syn->dlugosc == n)
{
	if (syn->cost < upper)
	{
		upper = syn->cost;
		sciezka1 = syn->sciezka;
		return;
	}
}
queue.push(*syn);
```

Na koniec sprawdzane jest czy doszliśmy do końca ścieżki. Jeżeli tak, to jeżeli granica liścia jest mniejsza od najmniejszej dotychczas znalezionej granicy, to jest ona podmieniana. Podmienia jest też wtedy ścieżka dla najmniejszej znalezionej granicy. Następuje wyjście z metody. Jeżeli węzeł nie jest liściem, to jest on dodawany do kolejki priorytetowej.

Po stworzeniu wszystkich węzłów korzenia przechodzimy do dalszej części metody B_B.

```
wezel* aktualny = new wezel;
while (!queue.empty())
{
	*aktualny = queue.top();
	queue.pop();
	if (aktualny->cost < upper)
	{
		for (int i = 0; i < n; i++)
			visited[i] = false;

		for (int i = 0; i < aktualny->dlugosc; i++)
		{
			visited[aktualny->sciezka[i]] = true;
		}
		for (int i = 0; i < n; i++)
		{
			if (!visited[i])
			{
				bound(aktualny, i);
```

```
                                visited[i] = true;
                                //cout << "bound" << "   " << endl;
                        }
                }
        }
}
```

W powyższym fragmencie kodu następuje badanie kolejnych węzłów w kolejce posortowanych rosnąco po wartości ich granicy. Dla każdego z nich tworzeni są jego wszyscy możliwi synowie poprzez wywołanie funkcji bound(). Jeżeli pobrany z kolejki węzeł ma granicę większą niż najmniejsza dotychczas znaleziona, pobierany jest kolejny węzeł. Z racji, że w kolejce węzły posortowane są rosnąco według wartości granicy, to w takim przypadku nastąpi jedynie opróżnienie kolejki.

```
int dlugosc = 0;
for (int i = 0; i < n-1; i++)
{
	dlugosc += A[sciezka1[i]][sciezka1[i + 1]];
}
dlugosc += A[sciezka1[n - 1]][0];
cout << "dlugosc: " << dlugosc << endl;
cout << "sciezka: ";
for (int i = 0; i < n; i++)
cout << sciezka1[i] << " -> ";
```

Na sam koniec na podstawie znalezionej ścieżki obliczany jest jej ostateczny koszt, a następnie następuje wypisanie wyników końcowych. Warto tutaj zaznaczyć, że w momencie dokonywania pomiarów wszelkie akcje związane z wypisywaniem wartości zostały zakomentowane, by nie wpływać na czas działania algorytmu.

4. Plan eksperymentu

Aby zmierzyć czas wykonywania poszczególnych algorytmów w zależności od rozmiaru grafu, zostanie wybranych 7 różnych wartości N, które oznaczają ilość wierzchołków w grafie. Następnie dla każdego rozmiaru grafu problem zostanie rozwiązany dla 100 losowo wygenerowanych instancji. Podczas wykonywania algorytmu mierzony będzie czas, który następnie będzie uśredniony dla każdego N. Warto tutaj nadmienić, iż mierzony czas nie będzie zawierać czasu generowania instancji.

Zarówno dla algorytmu przeglądu zupełnego oraz metody podziału i ograniczeń przyjęte zostały następujące rozmiary struktur danych: 7, 8, 9, 10, 11, 12, 13.

4.1. Sposób generowania danych

Za generowanie losowych instancji problemu odpowiedzialna jest metoda:

```
void AdjacencyMatrix::random_problem(int N)
{
	if (n != 0)
	{
		for (int i = 0; i < N; i++)
			delete A[i];
		delete[] A;
	}
	n = N;

	A = new int* [N];
```

```
        for (int i = 0; i < N; i++)
        {
                A[i] = new int[N];
        }
        for (int i = 0; i < N; i++)
        {
                for (int j = 0; j < N; j++)
                {
                        if (i != j)
                                A[i][j] = rand() % 100 + 1;
                        else
                                A[i][j] = 0;
                }
        }

}
```

Generuje ona macierz sąsiedztwa o zadanym rozmiarze. Wagi krawędzi są generowane losowo, a ich wartości mieszczą się w zakresie 1 – 100.

4.2. Sposób mierzenia czasu

W celu zmierzenia czasu wykorzystano 2 funkcje pomocnicze:

```
void StartCounter()
{
        LARGE_INTEGER li;
        if (!QueryPerformanceFrequency(&li))
                cout << "QueryPerformanceFrequency failed!" << endl;

        PCFreq = double(li.QuadPart) / 1000000000.0;    //nanosekundy

        QueryPerformanceCounter(&li);
        CounterStart = li.QuadPart;
}


double GetCounter()
{
        LARGE_INTEGER li;
        QueryPerformanceCounter(&li);
        return double(li.QuadPart - CounterStart) / PCFreq;
}
```

Funkcja StarCounter() odpowiedzialna jest za rozpoczęcie liczenia czasu, natomiast funkcja GetCounter() zwraca aktualny czas. Czas liczony jest w nanosekundach.

4.3. Sposób dokonywanie pomiarów

Za dokonanie pomiarów odpowiedzialny jest poniższy fragment kodu:

```
        int N = 10;
        long long int suma = 0;
        for (int i = 0; i < 100; i++)
        {
                matrix->random_problem(N);
                StartCounter();
```

```
        matrix->B_B(0);
        suma += GetCounter();
}
cout << suma / 100;
```

Zmienna N odpowiedzialna jest za ilość wierzchołków w generowanym grafie, a zmienna suma za przechowywanie sumy czasów potrzebnych na wykonanie algorytmu dla 100 różnych instancji. W pętli for, która wykonuje się 100 razy na początku generujemy losowy graf dla zadanej liczby wierzchołków. Następnie uruchamiany jest licznik czasu, wywoływana jest funkcja odpowiedzialna za algorytm Brute Force lub B&B(w powyższym przykładzie wywoływany jest B&B, w celu wywołania Brute Force należałoby podmienić matrix->B&B na matrix->bruteForce). Po wykonaniu algorytmu do sumy dodawany jest aktualny stan licznika. Po zmierzeniu czasu dla 100 instancji problemu, jego suma jest dzielona na 100, tak aby uzyskać średni wynik.

5. Wyniki eksperymentów

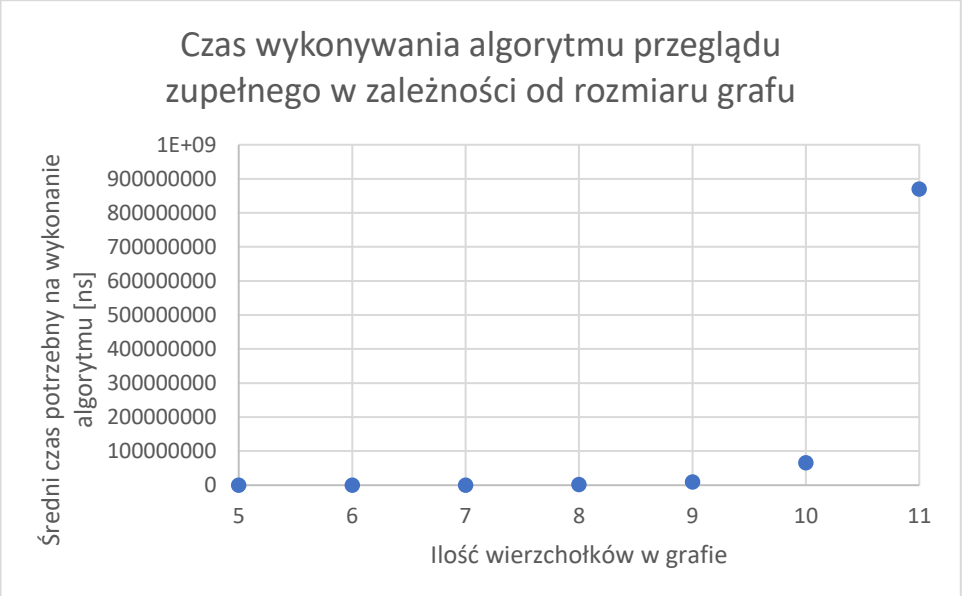
Eksperyment przeprowadzony był dla grafów, których liczba wierzchołków równa była kolejno: 5, 6, 7, 8, 9, 10, 11.

5.1. Brute Force

Dla algorytmu przeglądu zupełnego osiągnięto następujące wyniki:

Ilość wierzchołków	5	6	7	8	9	10	11
Średni czas[ns]	7580	27160	133590	1163780	9360196	65349238	869725015

Tabela 1. Średni czas algorytmu przeglądu zupełnego dla różnych rozmiarów grafów



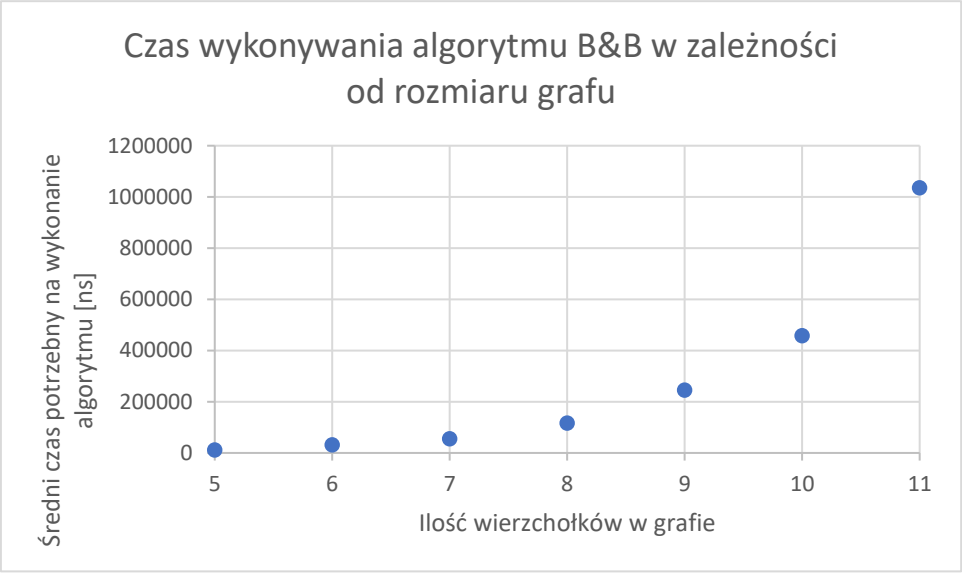
Rysunek 1. Wykres zależności czasu wykonywania algorytmu przeglądu zupełnego od ilości wierzchołków w grafie

5.2. B&B

Dla metody podziału i ograniczeń osiągnięto następujące wyniki;

Ilość wierzchołków	5	6	7	8	9	10	11
Średni czas [ns]	10487	30469	54075	115671	244607	457406	1035138

Tabela 2. Średni czas wykonywania algorytmu B&B dla różnych rozmiarów grafów

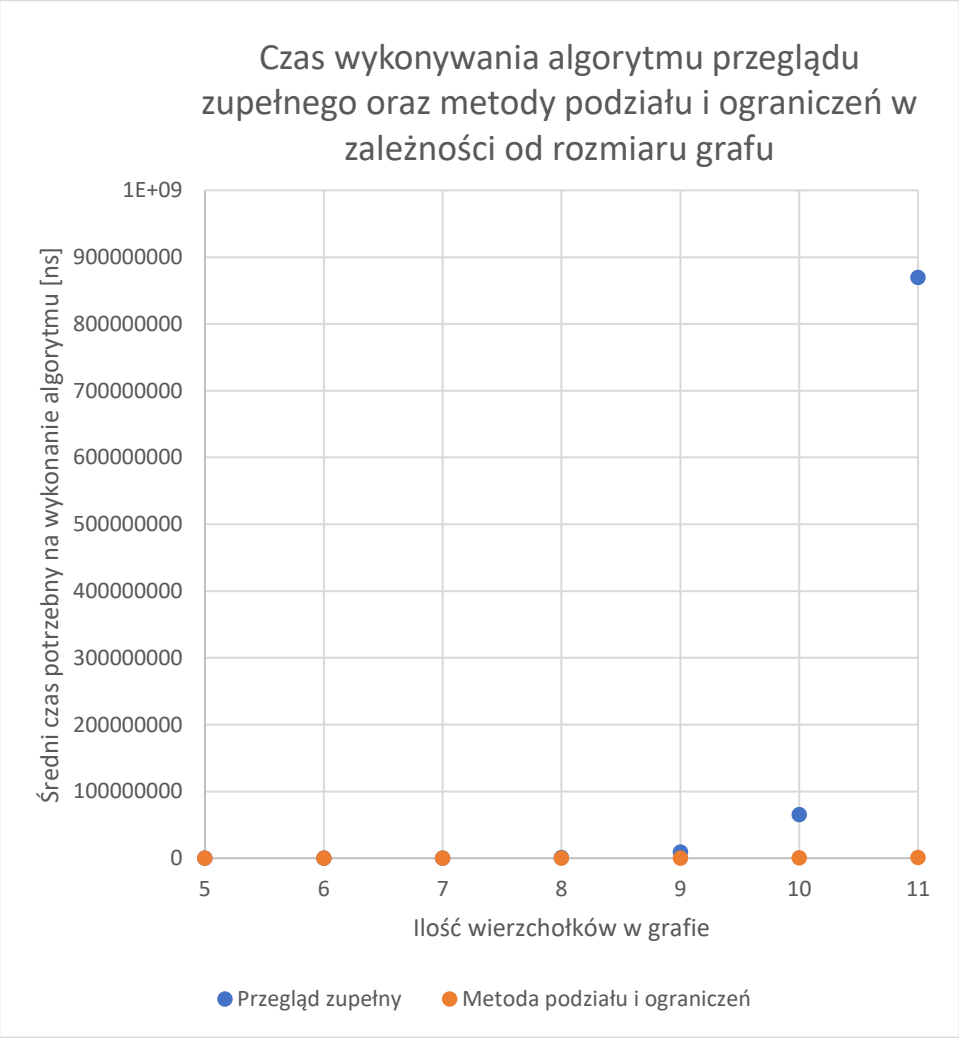


Rysunek 2. Wykres zależności czasu wykonywania algorytmu B&B od ilości wierzchołków w grafie

5.3. Porównanie obu metod

	Średni czas [ns]	
Ilość wierzchołków	**Brute Force**	**B&B**
5	7580	10487
6	27160	30469
7	133590	54075
8	1163780	115671
9	9360196	244607
10	65349238	457406
11	869725015	1035138

Tabela 3. Średni czas potrzeny na wykonanie algorytmu w zależności od rozmiaru grafu



Rysunek 3. Wykres zależności czasu wykonywania algorytmów w zależności od rozmiaru grafu

6. Wnioski

Metoda przeglądu zupełnego jest efektywna dla bardzo małych grafów. W ich przypadku jest nawet szybsza od metody podziału i ograniczeń. Niestety ze względu na złożoność obliczeniową równą O(n!), dla większych grafów czas jej wykonywania staje się bardzo duży. Metoda podziału i ograniczeń jest znacznie bardziej efektywna dla dużych grafów niż jej poprzedniczka.

Wyniki otrzymane dla metody przeglądu zupełnego są zgodne z oczekiwaną złożonością.

Wyniki otrzymane dla metody podziału i ograniczeń również znalazły się w przedziale określonym we wstępie.

7. Bibliografia

https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf

https://pl.wikipedia.org/wiki/Cykl_Hamiltona

https://pl.wikipedia.org/wiki/Problem_komiwoja%C5%BCera