Politechnika
Wrocławska,
Wydział Informatyki
i Telekomunikacji
Semestr V Rok III

Zadanie projektowe nr 1

Implementacja i analiza efektywności algorytmu podziału i ograniczeń i programowania dynamicznego.

Wykonał:
Patryk Ignasiak 263889
Gr. Piątek 11:15
Prowadzący:
dr inż. Jarosław Mierzwa
Przedmiot:
Projektowanie efektywnych
algorytmów – Projekt

Wrocław, 16 Października 2023

Spis treści

1 Wstęp

Podczas pierwszego projektu należało zaimplementować oraz przeanalizować efektywność algorytmów opartych o metody:

- Przeglądu zupełnego (ang. Brute Force)
- Podziału i ograniczeń (ang. Branch & Bound)
- Programowania dynamicznego (ang. Dynamic Programming)

dla asymetrycznego problemu komiwojażera (ATSP). Problem ten polega na odnalezieniu minimalnego cyklu Hamiltona w grafie pełnym, skierowanym o wagach krawędzi >= 0. O minimalnym cyklu Hamiltona mówimy, gdy suma wag krawędzi wykorzystanych w cyklu jest najmniejsza. Rozwiązywanie tego problemu ma wiele trudności, m.in. z racji, iż mamy do czynienia z grafem pełnym, skierowanym mamy do rozważenia V(V-1) krawędzi. A wszystkich możliwych kombinacji tworzących cykl Hamiltona jest (V-1)!. Jest to bardzo dużo danych do przeanalizowania, dlatego też efektywność algorytmów dla tego typu problemów ma ogromne znaczenie.

2 Omówienie sposobu testowania algorytmów

Dla każdej metody zostało wygenerowanie i przetestowane 100 różnych instancji dla każdego rozmiaru grafu. Testy zostały przeprowadzone dla następujących wielkości:

- Metoda Brute Force – 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
- Metoda Branch & Bound – 6, 10, 12, 14, 16, 18, 20, 22, 24, 26
- Metoda Dynamic Programming – 6, 10, 12, 14, 16, 18, 20, 22, 24

2.1 Pomiary czasu

Czas był mierzony za pomocą „chrono” i zapisywany do tablicy czasów, która na koniec testów była zapisywana do pliku tekstowego. Przykład pomiaru czasu:

Listing kodu 1 Przykład pomiaru czasu

```
auto start = chrono::steady_clock::now();
bb->findShortestPathLC(data->matrix, data->size);
auto end = chrono::steady_clock::now();

auto duration = end - start;
times[i-3] += chrono::duration_cast<chrono::microseconds>(duration).count();
```

Przed wywołaniem algorytmu pobieramy czas, następnie wywoływany jest algorytm. Po wykonaniu się algorytmu pobieramy czas ponownie i obliczoną różnicę czasów zapisujemy do tablicy.

2.2 Generowanie danych

Dane do testów były generowane za pomocą metody generateData() z klasy Data. Metoda ta tworzy macierz o zadanej wielkości, a następnie wypełnia ją losowymi danymi.

Listing kodu 2 Generowanie danych do testów

```
void Data::generateData(int min, int max, int mSize) {
	if (mSize < 2) {
		cout << "Podano za mały rozmiar!!!" << endl;
		return;
	}
	size = mSize;
	createMatrix();

	random_device rd;
	mt19937 gen(rd());
	uniform_int_distribution<> distValue(min, max);

	for (int i = 0; i < size; i++) {
		for (int j = 0; j < size; j++) {
			if (i != j) {
				matrix[i][j] = distValue(gen);
			}
			else {
				matrix[i][j] = 0;
			}
		}
	}
}
```

3 Metoda przeglądu zupełnego

3.1 Wstęp

Brute Force jest to metoda dokładna, lecz bardzo wolna z racji na naiwność algorytmu. Algorytm ten polega na wygenerowaniu wszystkich możliwych kombinacji krawędzi, które tworzą cykl Hamiltona, a następnie wybranie najkrótszego z cykli. Generowanie wszystkich możliwych kombinacji jest bardzo powolne przez co złożoność tej metody wynosi O(n!). Algorytm ten jest dobrym rozwiązaniem w momencie, gdy mamy do odwiedzenia niewiele miast. Ponieważ dla większej liczby miast wykonanie algorytmu nie odbywa się w akceptowalnym czasie.

3.2 Omówienie implementacji

Metoda ta polega na generowaniu kolejnych permutacji i obliczaniu dla nich drogi w celu wybrania najkrótszej. Do generowania permutacji wykorzystałem sposób zaprezentowany przez R. Sedgewicka na Uniwersytecie w Princeton, na jednym z jego wykładów. Jest to algorytm rekurencyjny, który zamienia ze sobą (funkcja swap()) kolejne wierzchołki z k-tym wierzchołkiem. Następnie jest wywoływany ponownie dla k-1, aż k == 0, wtedy obliczana jest długość ścieżki i jeśli jest ona krótsza od poprzedniej to zostaje ona zapamiętana jako ta najkrótsza.

Listing kodu 3 Generowanie permutacji

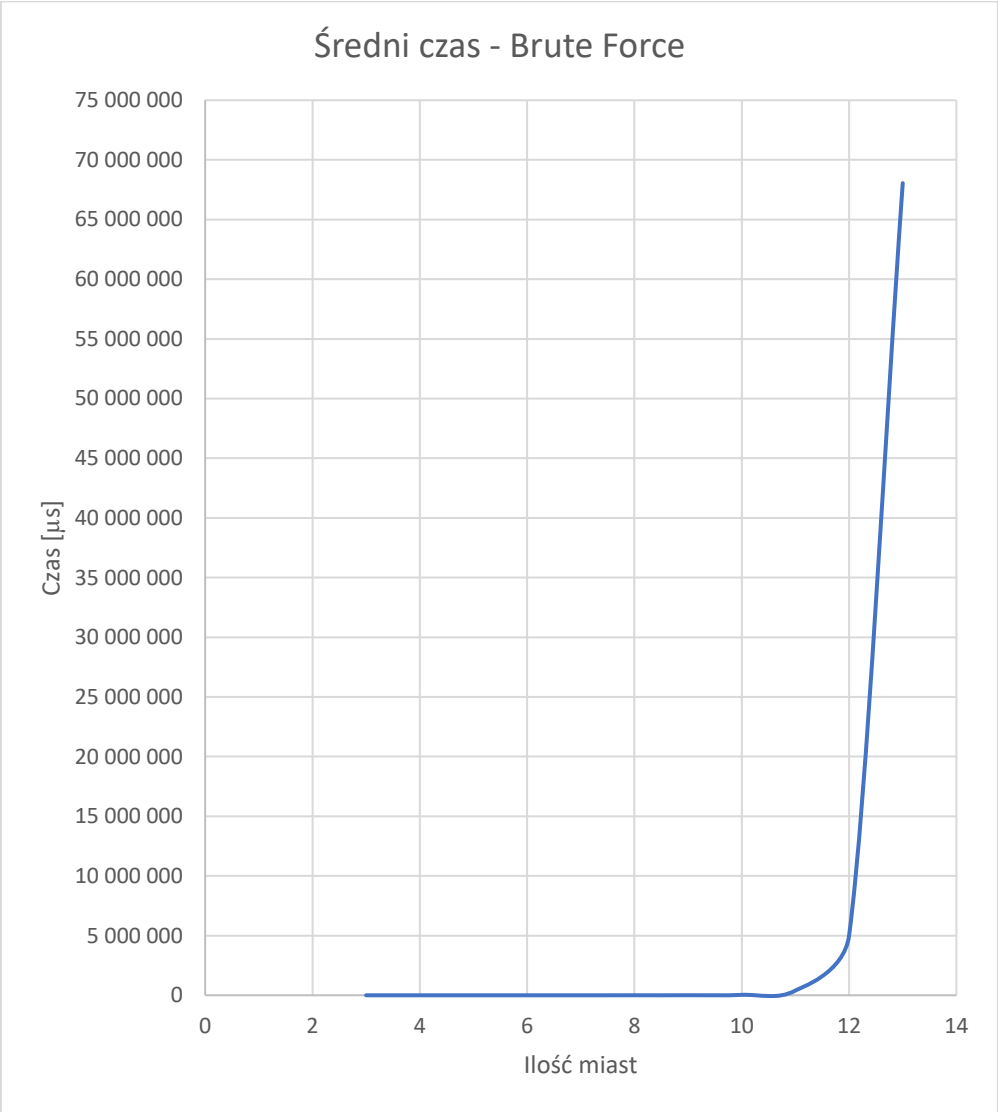
```
void BF::permutations(int k) {
	if (k == 0) {
		countPath();
	}
	else {
		for (int i = 0; i <= k; i++) {
			swap(i, k);
			permutations(k - 1);
			swap(i, k);
		}
	}

}
```

3.3 Wyniki testów

Tabela 1 Wyniki testów dla metody Brute Force

Liczba miast	Średni czas [µs]
3	0
4	1
5	1
6	6
7	54
8	356
9	3 202
10	33 982
11	402 141
12	5 009 098
13	68 051 226

Wykres 1 Średni czas w zależności od liczby miast dla metody Brute Force



3.4 Omówienie wyników

Przeprowadzone badania pokazują, iż algorytm faktycznie osiąga złożoność O(V!), co czyni go bardzo nieefektywnym dla dużych grafów. Jednakże jego zaletą jest 100% dokładność wyników, tzn. stosując tą metodę mamy pewność, że dostaniemy najlepsze rozwiązanie, ponieważ sprawdza on wszystkie możliwości.

4 Metoda podziału i ograniczeń

4.1 Wstęp

Metoda ta ściśle zależy od rozłożenia danych w grafie, przez co jest ona bardzo nieobliczalna. W najgorszym wypadku sprowadza się ona do przeglądu zupełnego. Metoda ta polega na szacowaniu, które węzły są obiecujące, a które nie. Następnie przegląda ona węzły obiecujące, co prowadzi do szybszego rozwiązania problemu. Najprostszymi z metod przeglądania węzłów są metody:

- Low Cost (Best First) – czyli za każdym razem wybieramy węzeł najbardziej obiecujący.

- DFS (przeszukiwanie w głąb) – przeglądamy wierzchołki obiecujące metodą przeszukiwani w głąb.

4.2 Przykład praktyczny

W przykładzie praktycznym wykorzystam metodę przeszukiwania Best First (Low Cost).

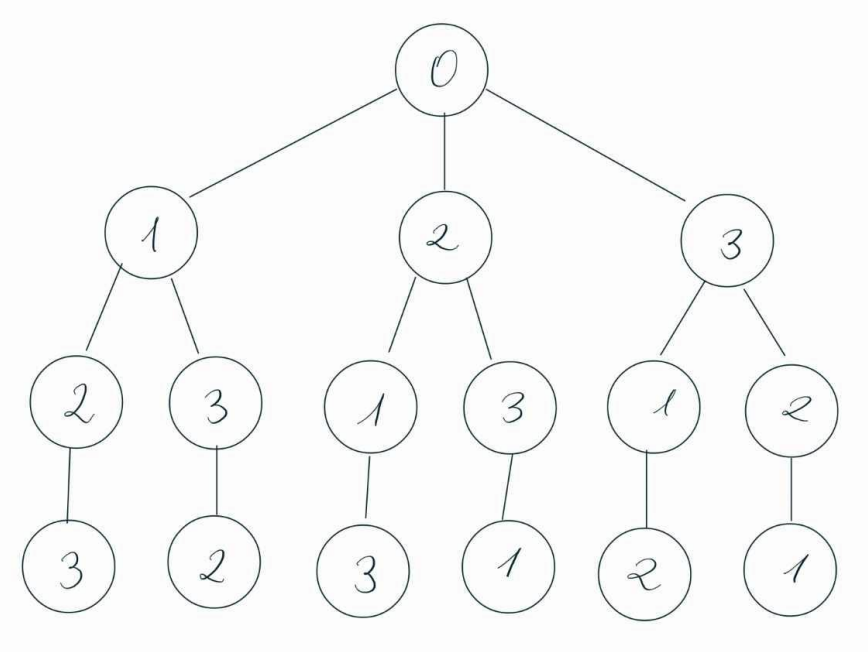
Mamy następującą macierz sąsiedztwa:

Tabela 2 Macierz sąsiedztwa dla prezentowanego przykładu

	0	1	2	3
0	x	3	1	2
1	4	x	2	1
2	3	1	x	2
3	1	2	1	x

Oraz następujące drzewo:

Rysunek 1 Drzewo prezentujące wszystkie możliwe rozwiązania



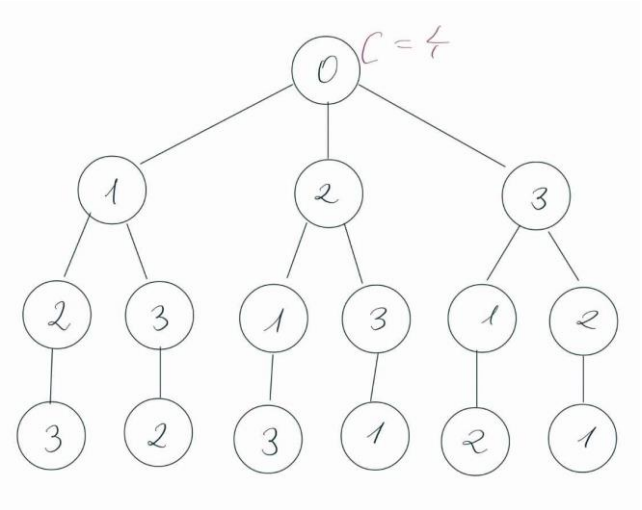
Wybieramy wierzchołek startowy 0 i przeprowadzamy dla niego redukcję wierszy.

Tabela 3 Redukcja wierszy dla węzła startowego

	0	1	2	3	
0	x	2	0	1	1
1	3	x	1	0	1
2	2	0	x	1	1
3	0	1	0	x	1

Następnie dla wszystkich niezerowych kolumn powinniśmy przeprowadzić redukcję jednak w tym wypadku takich nie mamy, więc zapisujemy koszt redukcji przy węźle startowym.

Rysunek 2 Stan drzewa po przeprowadzeniu redukcji



Następnie rozpatrujemy wierzchołki z kolejnego poziomu. Dla wierzchołka 1 mamy następującą macierz.

Tabela 4 Macierz dla wierzchołka 1

	0	1	2	3
0	x	x	x	x
1	x	x	1	0
2	2	x	x	1
3	0	x	0	x

Teraz musimy przeprowadzić redukcję.

Tabela 5 Redukcja macierzy dla wierzchołka 1

	0	1	2	3	
0	x	x	x	x	
1	x	x	1	0	0
2	1	x	x	0	1
3	0	x	0	x	0

Po przeprowadzeniu redukcji wierszy nie zostały nam kolumny niezerowe. Więc możemy obliczyć koszt dla węzła w następujący sposób.

$$C = C(0,1) + r + \hat{r}$$

Gdzie:

- C(0,1) – koszt drogi z wierzchołka 0 do 1

- r – koszt redukcji wierzchołka poprzedniego

- $\hat{r}$ - koszt redukcji wierzchołka obecnego

Zatem:

$$C = 3+4+1 = 8$$

Redukcje dla kolejnych węzłów przeprowadzamy w ten sam sposób

Węzeł dla wierzchołka 2

Tabela 6 Macierz dla wierzchołka 2

	0	1	2	3
0	x	x	x	x
1	3	x	x	0
2	x	0	x	1
3	0	1	x	x

I wiersze i kolumny są zerowe zatem nie musimy przeprowadzać redukcji.

C = 1+4+0 = 5

Węzeł dla wierzchołka 3

Tabela 7 Macierz dla wierzchołka 3

	0	1	2	3
0	x	x	x	x
1	3	x	1	x
2	2	0	x	1
3	x	1	0	x

Tutaj musimy przeprowadzić redukcję.

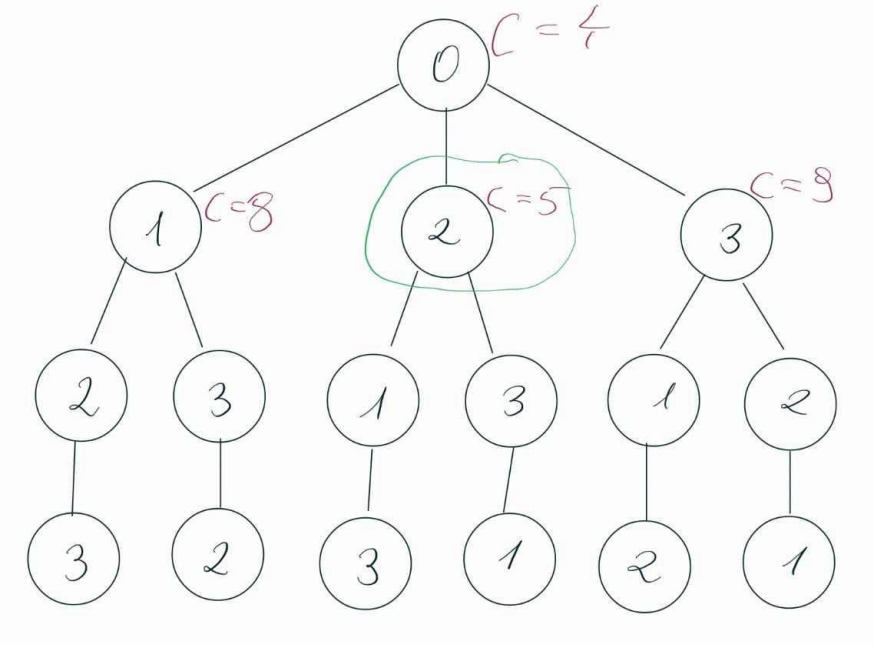
Tabela 8 Redukcja wierszy i kolumn

	0	1	2	3	
0	x	x	x	x	
1	0	x	0	x	1
2	0	0	x	1	0
3	x	1	0	x	0
	2	0	0		

C = 2 + 4 + 3 = 9

Teraz wybieramy węzeł o najmniejszej wartości.

Rysunek 3 Wybór kolejnego węzła do rozpatrzenia



Rozpatrujemy kolejne węzły.

Węzeł dla wierzchołka 1.

Tabela 9 Macierz dla wierzchołka 1

	0	1	2	3
0	x	x	x	x
1	x	x	x	0
2	x	x	x	x
3	0	x	x	x

C = 1 + 5 + 0 = 6

Węzeł dla wierzchołka 3.

Tabela 10 Macierz i redukcja dla wierzchołka 3

	0	1	2	3
0	x	x	x	x
1	0	x	x	x
2	x	x	x	x
3	x	0	x	x

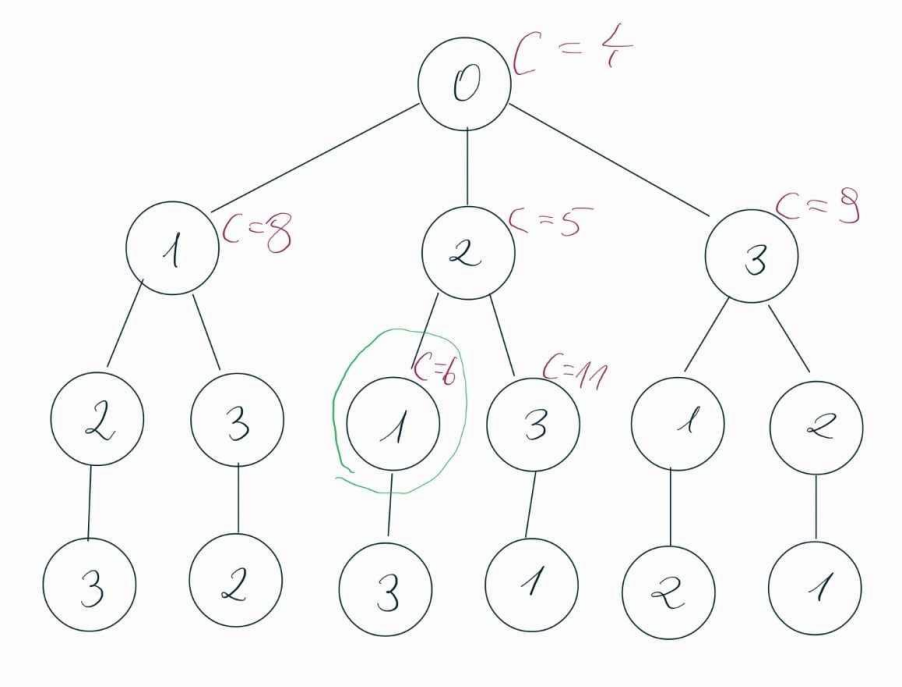
3

1

C = 2 + 5 + 4 = 11

Znowu wybieramy węzeł o najmniejszej wartości.

Tabela 11 Wybór kolejnego węzła



Rozpatrujemy kolejne węzły.

Węzeł dla wierzchołka 3.

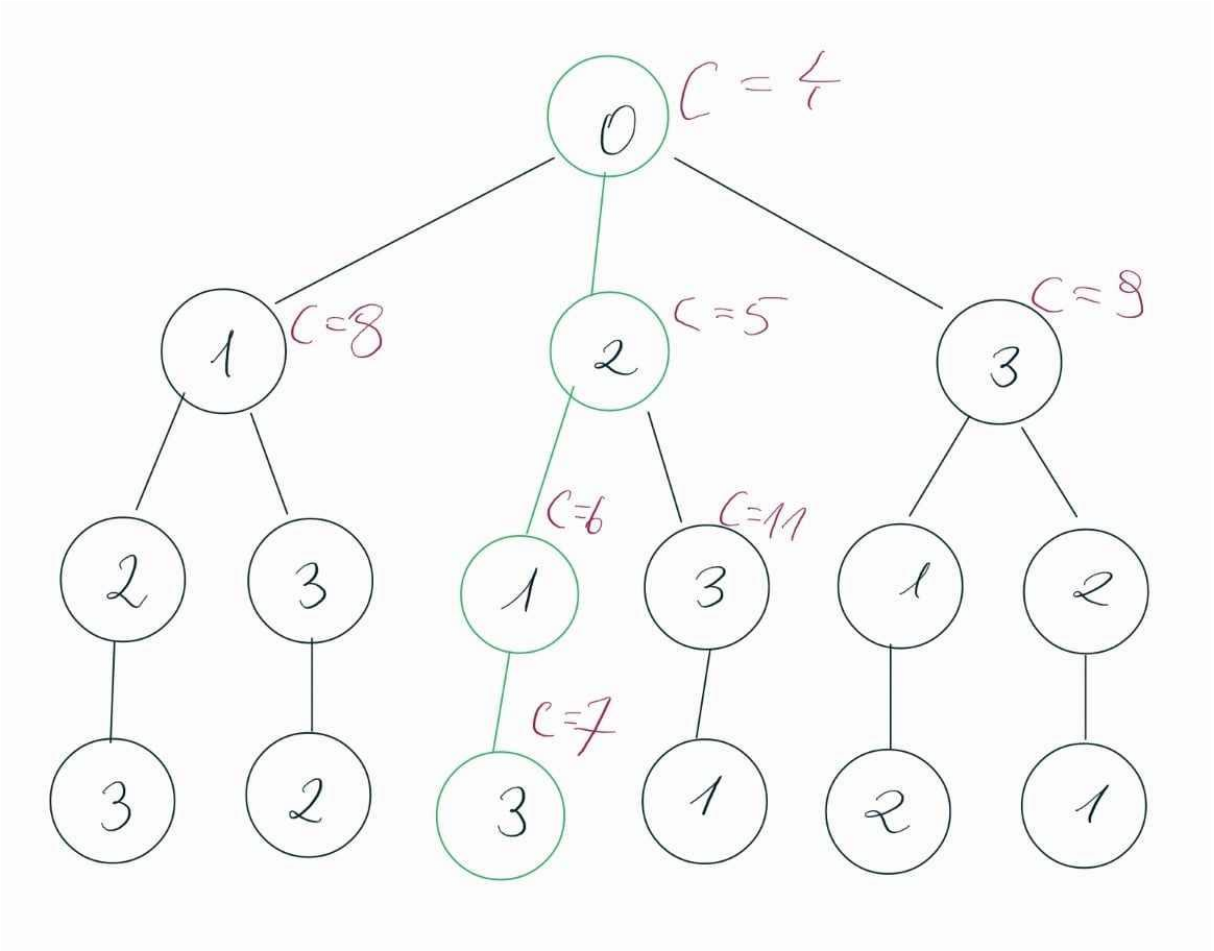
Tabela 12 Macierz dla wierzchołka 3

	0	1	2	3
0	x	x	x	x
1	x	x	x	x
2	x	x	x	x
3	x	x	x	x

C=1+6+0 = 7

Wartość ta zostaje również wartością górnej granicy, następnie sprawdzamy czy któryś z rozwiniętych wierzchołków ma mniejszą dolną granicę niż otrzymana przed chwilą górna granica. Niestety żaden z nich się nie kwalifikuje. Zatem ostateczne rozwiązanie przedstawię poniżej.

Rysunek 4 Prezentacja końcowego rozwiązania



Długość: 7

Ścieżka: 0, 2, 1, 3

4.3 Omówienie implementacji - Low Cost

Metoda ta polega na pobieraniu z kolejki priorytetowej węzła o najmniejszym koszcie. Do implementacji kolejki zastosowałem kopiec binarny, utworzony na pierwszym projekcie z przedmiotu „Struktury danych i złożoność obliczeniowa”. Pobieramy kolejne węzły z kolejki i oceniamy, czy są obiecujące, jeśli tak to je rozpatrujemy, a jeśli nie to odrzucamy. Jeśli poziom węzła jest równy rozmiarowi grafu, to porównujemy czy nowa ścieżka jest lepsza od starej.

Listing kodu 4 B&B - przeszukiwanie Low Cost

```
void BB::findShortestPathLC(int** matrix, int size) {
	BB::matrix = matrix;
	BB::size = size;
	int upperBound = INT32_MAX;
	Node node = Node(0, 0, 1, matrix, size);
	reduceRow(node);
	reduceColumn(node);
	Heap heap = Heap(size * 3);
	heap.push(node);
	while (heap.noEmpty()) {

		node = heap.pop();
		bool fulfilled = false;
		if (upperBound < INT32_MAX) {
			if (getLowerBound(node) < upperBound) {
				fulfilled = true;
			}
		}
		else {
			fulfilled = true;
		}
		if (fulfilled) {
			for (int i = 0; i < size; i++) {
				if (node.getVertice() != i &&
node.getMatrix()[node.getVertice()][i] < INT32_MAX) {
					Node childNode = Node(i, node.getValue(),
node.getLevel() + 1, node.getMatrix(), size, node.getPath());

					makeInf(node.getVertice(), i, childNode);
					reduceRow(childNode);
					reduceColumn(childNode);

					heap.push(childNode);

					if (childNode.getLevel() == size) {
						upperBound = childNode.getValue();
						best = childNode;
					}
				}
			}
		}
	}

}
```

4.4 Omówienie implementacji - DFS

W tej metodzie zastosowałem kolejkę LIFO (last in, first out) w celu uzyskania przeszukiwania w głąb. Podobnie jak w poprzedniej metodzie kolejne wierzchołki są pobierane z kolejki, oceniane oraz dla obiecujących rozważane.

Listing kodu 5 B&B - przeszukiwanie DFS

```
void BB::findShortestPathDFS(int** matrix, int size) {
	BB::matrix = matrix;
	BB::size = size;
	int upperBound = INT32_MAX;
	Node node = Node(0, 0, 1, matrix, size);
	reduceRow(node);
	reduceColumn(node);
	List list = List();
	list.pushBack(node);
	while (list.noEmpty()) {
		node = list.popBack();
		bool fulfilled = false;
		if (upperBound < INT32_MAX) {
			if (getLowerBound(node) < upperBound) {
				fulfilled = true;
			}
		}
		else {
			fulfilled = true;
		}
		if (fulfilled) {
			for (int i = 0; i < size; i++) {
				if (node.getVertice() != i &&
node.getMatrix()[node.getVertice()][i] < INT32_MAX) {
					Node childNode = Node(i, node.getValue(),
node.getLevel() + 1, node.getMatrix(), size, node.getPath());

					makeInf(node.getVertice(), i, childNode);
					reduceRow(childNode);
					reduceColumn(childNode);
					list.pushBack(childNode);

					if (childNode.getLevel() == size) {
						upperBound = childNode.getValue();
						best = childNode;
					}
				}
			}
		}
	}
}
```

4.5 Omówienie pozostałych funkcji i elementów używanych w poprzednich algorytmach

4.5.1 Węzeł (Node)

Algorytm ten nie operuje bezpośrednio na wierzchołkach, lecz na węzłach drzewa. Węzeł ma następujące pola:

- vertice – przechowuje wierzchołek przypisany dla węzła

- value – przechowuje koszt odwiedzenia węzła

- level – przechowuje poziom drzewa na którym znajduje się węzeł

- path – przechowuje ścieżkę prowadzącą do węzła

- visited – przechowuje informacje czy wierzchołek o danym indeksie został odwiedzony

- matrix – przechowuje macierz sąsiedztwa dla danego węzła, różni się ona od pierwotnej macierzy, ponieważ zawiera ona wartości zredukowane

4.5.2 getLowerBound()

Funkcja getLowerBound() zwraca wartość dolnej granicy dla danego węzła. Granica ta jest obliczana ze wzoru:

$$LB = node_{value} + \frac{min(V_{in}) + min(V_{out})}{2}$$

Gdzie:

- V_{in} - wierzchołki, z których jeszcze można dotrzeć do wierzchołka, dla którego obliczamy granicę

- V_{out} - wierzchołki, do których jeszcze można dotrzeć z wierzchołka, dla którego obliczamy granicę

Listing kodu 6 Obliczanie dolnej granicy

```
int BB::getLowerBound(Node node) {
	int** nMatrix = node.getMatrix();
	int sum = node.getValue();
	for (int i = 0; i < size; i++) {
		if (!node.getVisited()[i]) {
			int min1 = INT32_MAX, min2 = INT32_MAX;
			for (int j = 0; j < size; j++) {
				if (!node.getVisited()[i] && i != j) {

					if (min1 > matrix[i][j]) {
						min1 = matrix[i][j];
					}
					if (min2 > matrix[j][i]) {
						min2 = matrix[j][i];
					}

				}
			}
			sum += (min1 + min2) / 2;
		}
	}
	return sum;
}
```

4.5.3 makeInf()

Funkcja ta modyfikuje macierz opisującą dany węzeł zamieniając wartości wiersza dla wierzchołka „from” na Infinity (nieskończoność), oraz kolumny dla wierzchołka „to”. Dodatkowo również zamienia wartość dla powrotu z wierzchołka „to” do wierzchołka „from”

Listing kodu 7 Wstawianie Infinity

```
void BB::makeInf(int from, int to, Node& node) {
	int** nMatrix = node.getMatrix();
	node.addValue(nMatrix[from][to]);
	for (int i = 0; i < size; i++) {
		nMatrix[i][to] = INT32_MAX;
		nMatrix[from][i] = INT32_MAX;
	}
	nMatrix[to][from] = INT32_MAX;
}
```

4.5.4 reduceRow()

Funkcja tak redukuje wartości wierszy macierzy węzła o wartość minimum w danym wierszu, a następnie sumuje te wartości i zapisuje jako koszt węzła.

Listing kodu 8 Redukcja wierszy

```
void BB::reduceRow(Node& node) {
	int** nMatrix = node.getMatrix();
	int sum = 0;
	for (int i = 0; i < size; i++) {
		int min = INT32_MAX;
		for (int j = 0; j < size; j++) {
			if (nMatrix[i][j] < INT32_MAX) {
				if (min > nMatrix[i][j]) {
					min = nMatrix[i][j];
				}
			}
		}
		if (min < INT32_MAX && min > 0) {
			for (int j = 0; j < size; j++) {
				if (nMatrix[i][j] < INT32_MAX) {
					nMatrix[i][j] -= min;
				}
			}
		}
		if (min < INT32_MAX) {
			sum += min;
		}

	}
	node.addValue(sum);
}
```

4.5.5 reduceColumn()

Funkcja tak redukuje wartości kolumn macierzy węzła o wartość minimum w danej kolumnie, a następnie sumuje te wartości i zapisuje jako koszt węzła.

Listing kodu 9 Redukcja kolumn

```
void BB::reduceColumn(Node& node) {
	int** nMatrix = node.getMatrix();
	int sum = 0;
	for (int j = 0; j < size; j++) {
		int min = INT32_MAX;
		for (int i = 0; i < size; i++) {
			if (nMatrix[i][j] < INT32_MAX) {
				if (min > nMatrix[i][j]) {
					min = nMatrix[i][j];
				}
			}
		}
		if (min < INT32_MAX && min > 0) {
			for (int i = 0; i < size; i++) {
				if (nMatrix[i][j] < INT32_MAX) {
					nMatrix[i][j] -= min;
				}
			}
		}
		if (min < INT32_MAX) {
			sum += min;
		}
	}
	node.addValue(sum);
}
```

4.6 Wyniki testów

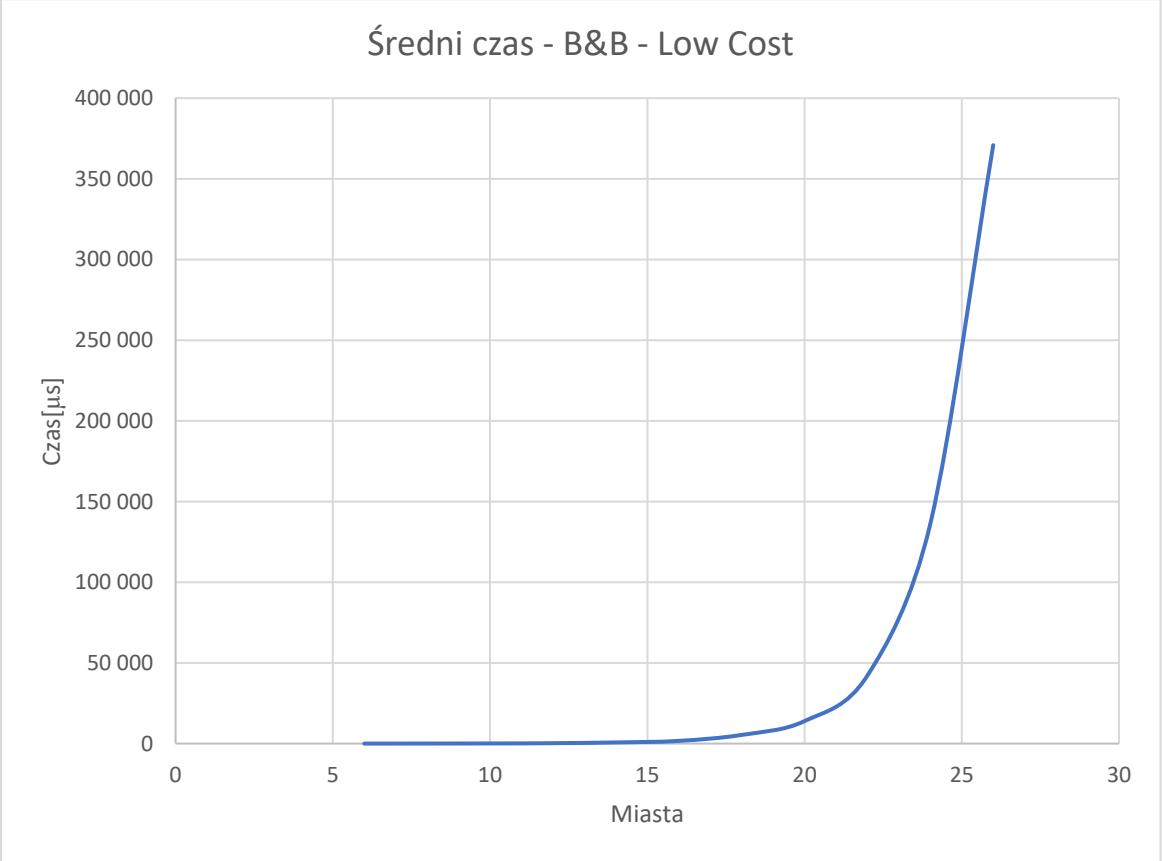
Tabela 13 Wyniki dla B&B DFS

Liczba miast	Średni czas [µs]
6	16
10	53
12	95
14	138
16	244
18	371
20	519
22	726
24	1 948
26	1 498

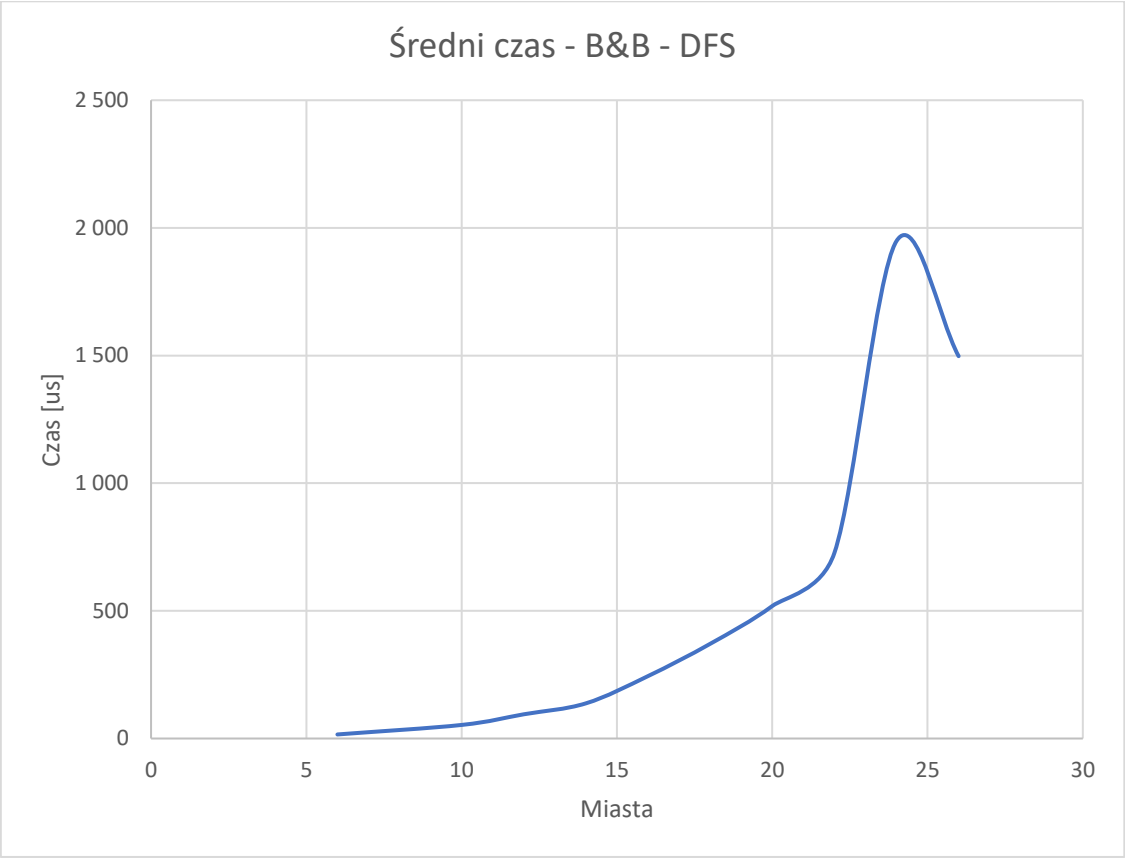
Tabela 14 Wyniki dla B&B Low Cost

Liczba miast	Średni czas [µs]
6	9
10	84
12	258
14	717
16	1 675
18	5 377
20	13 944
22	42 183
24	136 258
26	370 855

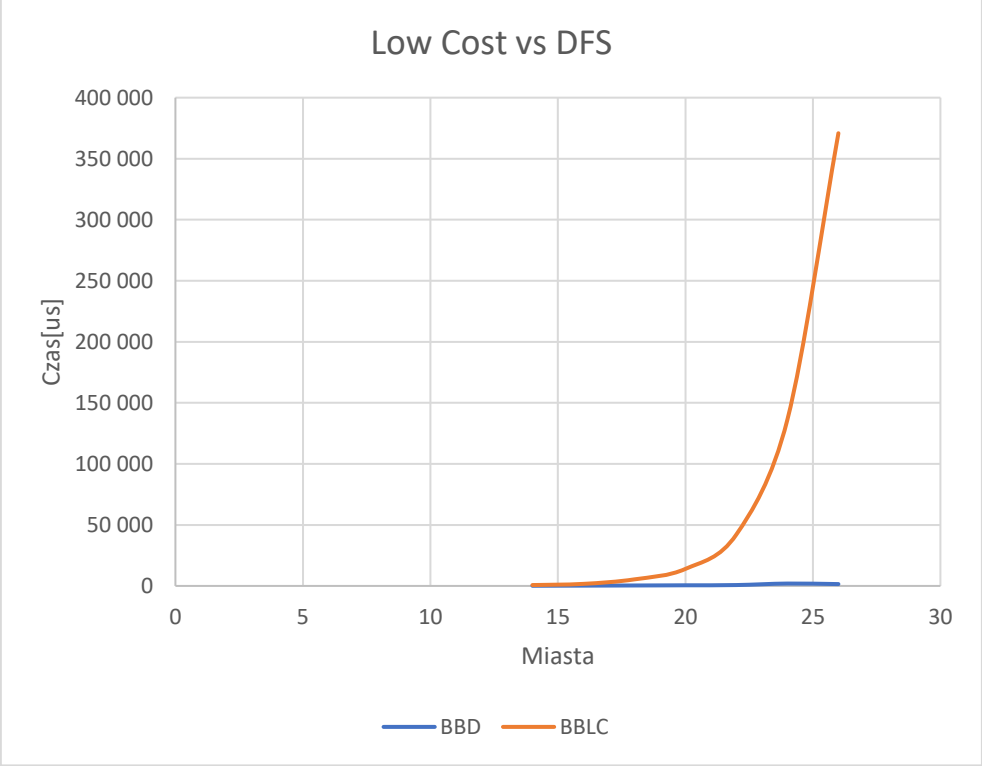
Wykres 2 Średni czas w zależności od liczby miast B&B Low Cost



Wykres 3 Średni czas w zależności od liczby miast B&B DFS



Wykres 4 Porównanie średnich czasów



4.7 Omówienie wyników

Przeprowadzone badania udowodniły, że metoda Branch & Bound nie tylko zależy od wielkości grafu, ale również od danych, które bada. Jak możemy zauważyć na drugim wykresie dla większej instancji otrzymałem mniejszy czas niż dla mniejszej. Związane to jest zapewne z tym jak w grafach ułożyły się akurat dane. Dodatkowo, jak można zauważyć na trzecim wykresie metoda Low Cost wyszła znacząco gorsza, związane jest to zapewne z wykorzystaniem kopca binarnego do kolejki priorytetowej. Przez co każde dodawanie i usuwanie elementów wymaga kopcowania, a dodatkowo przy większych instancjach problemu wymaga co jakiś czas relokacji kopca. Dlatego też ta metoda okazała się mniej wydajna, jednak dla niektórych przypadków jest lepsza jednak wartość średnia tego nie pokazuje.

5 Metoda programowania dynamicznego

5.1 Wstęp

Jest to metoda rekurencyjna, która polega na dzieleniu dużych problemów na mniejsze. Podobnie jak w metodzie „dziel i zwyciężaj", jednak tutaj pod-problemy nie muszą być rozłączne. Dzięki czemu, w odróżnieniu od metody Brute Force, sprowadza problem ponad-wielomianowy do problemu wielomianowego. Złożoność tej metody wynosi $O(V^2 \cdot 2^V)$., a metodę tę możemy opisać następującym wzorem:

$$g(i, S) = \min_{k \in S}\{C_{ik} + g(k, S - \{k\}\}$$

Gdzie:

i – wierzchołek początkowy

S – pozostałe wierzchołki

k – wierzchołek kolejny

C – koszt od i do k

5.2 Omówienie implementacji

Do rozwiązania problemu wykorzystałem algorytm w wersji rekurencyjnej z zapamiętywaniem wyników. Dzięki zapamiętywaniu wyników algorytm nie musi kilka razy obliczać tego samego pod-problemu, dzięki czemu działa znacznie szybciej. Iterując po kolejnych wierzchołkach, dla każdego nieodwiedzonego, aktualizowana jest zmienna mask poprzez dodanie nowego wierzchołka oraz zapisanie do zmiennej newMask. Następnie obliczany koszt z wierzchołka start do wierzchołka i. Gdy koszt jest mniejszy od obecnego minimum to jest on oraz wierzchołek zapamiętywany w celu późniejszego zapisania w tablicach.

Listing kodu 10 Programowanie dynamiczne

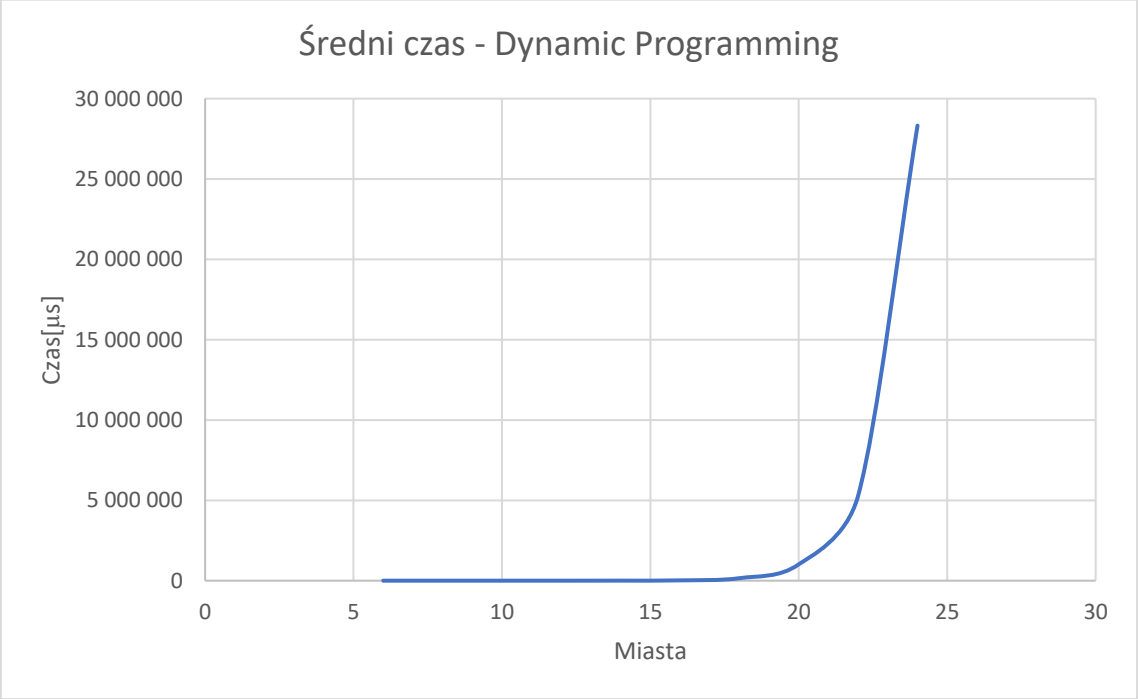
```
int DP::solve( int start, int mask, vector<vector<int>>& memory, vector<vector<int>>&
path) {

	// Jak wszystko było odwiedzone to dodajemy koszt powrotu
	if (mask == (1 << size) - 1) {
		path[start][mask] = 0;
		return matrix[start][0];
	}
	// jak bylo sprawdzane to zwrocić zapamietane
	if (memory[start][mask] != -1) {
		return memory[start][mask];
	}

	int minimum = INT_MAX;
	int nextNode = -1;

	for (int i = 0; i < size; i++) {
		if ((mask & (1 << i)) == 0) {
			int newMask = mask | (1 << i);
			int cost = matrix[start][i] + solve( i, newMask, memory, path);
			if (cost < minimum) {
				minimum = cost;
				nextNode = i;
			}
		}
	}
	//zapamietanie rozwiazania
	memory[start][mask] = minimum;

	//zapamiętanie ścieżki
	path[start][mask] = nextNode;

	return minimum;
}
```

5.3 Wyniki testów

Tabela 15 Wyniki dla metody programowania dynamicznego

Liczba miast	Średni czas [µs]
6	2
10	108
12	689
14	3 717
16	20 169
18	156 633
20	1 019 513
22	5 295 550
24	28 322 311

Wykres 5 Średni czas w zależności od liczby miast programowanie dynamiczne



5.4 Omówienie wyników

Przeprowadzone badania wykazały, że faktycznie ta metoda jest lepsza od metody Brute Force. Dzięki czemu można stosować ten algorytm dla większych instancji problemu.

6 Podsumowanie oraz porównanie wszystkich algorytmów.

6.1 Wyniki

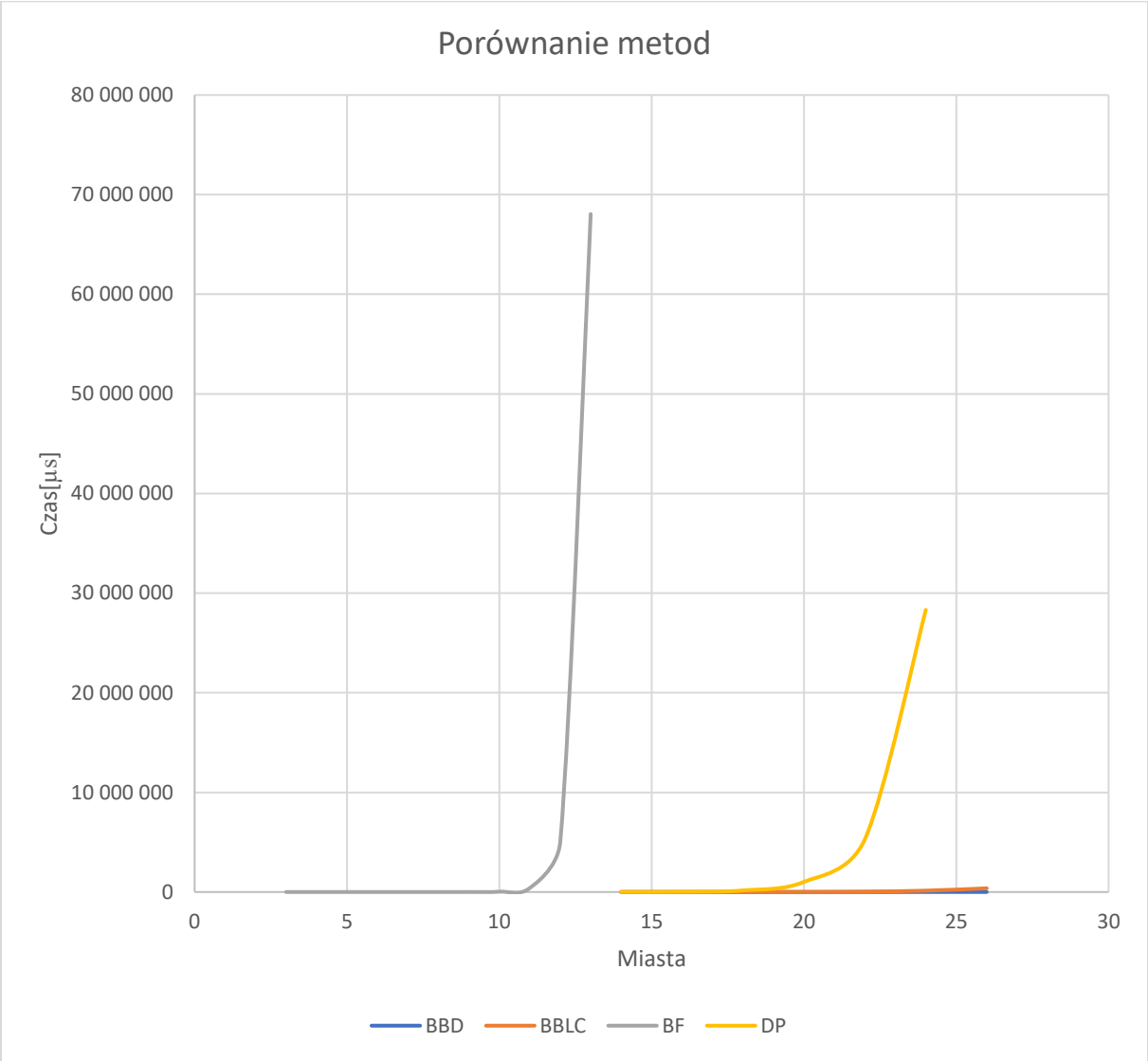
Omówienie skrótów:

- BBD – metoda Branch & Bound z algorytmem przeszukiwania w głąb (DFS)
- BBLC – metoda Branch & Bound z algorytmem przeszukiwania najpierw najlepszy (Low Cost)
- BF – metoda Brute Force
- DP – metoda Dynamic Programming

Tabela 16 Porównanie wszystkich metod

Miasta	BBD	BBLC	BF	DP
3			0	
4			1	
5			1	
6	16	9	6	2
7			54	
8			356	
9			3 202	
10	53	84	33 982	108
11			402 141	
12	95	258	5 009 098	689
13			68 051 226	
14	138	717		3 717
16	244	1 675		20 169
18	371	5 377		156 633
20	519	13 944		1 019 513
22	726	42 183		5 295 550
24	1 948	136 258		28 322 311
26	1 498	370 855		

Wykres 6 Porównanie wszystkich metod



6.2 Analiza wyników

Powyższe testy znakomicie pokazują, iż Brute Force jest najmniej wydajnym z algorytmów. Możemy również zauważyć, że najlepszymi są metody Branch & Bound, jednak to nie do końca prawda, gdyż ściśle zależą one od specyficzności danych w grafie. Gdyż w najgorszym przypadku sprowadza się ta metoda do przeglądu zupełnego. Jednak testy dla 100 losowych instancji nie były w stanie tego tak dobrze pokazać. Dlatego też metoda programowania dynamicznego wydaje się być najmniej zawodną z metod.

7 Bibliografia

Ogólne

- Wykłady dr inż. Tomasz Kapłon
- „Wprowadzenie do algorytmów” Clifford Stein, Ron Rivest i Thomas H. Cormen
- https://eduinf.waw.pl/inf/
- http://www.algorytm.org/
- https://en.cppreference.com/w/cpp/chrono/parse
- http://jaroslaw.mierzwa.staff.iiar.pwr.edu.pl/sdizo/

Brute Force

- http://algorytmika.wikidot.com/exponential-permut

Branch & Bound

- https://youtu.be/1FEP_sNb62k?si=cC6-sqslnUK_ghYW
- https://ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf
- https://cs.pwr.edu.pl/zielinski/lectures/om/mow10.pdf

Dynamic Programming

- https://youtu.be/XaXsJJh-Q5Y?si=A7usHn_KkgVE1hB_
- https://youtu.be/Q4zHb-Swzro?si=s5WGEnLsZtd_TQkC
- https://www.mimuw.edu.pl/~jrad/wpg/zajecia2.html