Sprawozdanie Projektowanie Efektywnych Algorytmów

Temat:
Implementacja i analiza efektywności algorytmu podziału i ograniczeń, i programowania dynamicznego

Politechnika Wrocławska
Dawid Szeląg 264008
Prowadzący: dr. inż. Jarosław Mierzwa
16.11.2023r.

Politechnika Wrocławska

Spis treści

1 Wstęp teoretyczny **2**

2 Algorytmy **2**

2.1 Przegląd zupełny 2

2.2 Podział i ograniczenia 2

2.2.1 Przykład - Branch&Bound 3

2.2.2 Implementacja B&B 8

2.3 Programowanie Dynamiczne 11

3 Plan eksperymentu **14**

4 Analiza danych pomiarowych **14**

5 Wnioski **17**

1 Wstęp teoretyczny

Problem komiwojażera (ang. The Traveling Salesman Problem (TSP)), polega na znalezieniu minimalnego cyklu Hamiltona w grafie pełnym ważonym. Wspomniany wcześniej cykl, polega na przejściu przez każdy wierzchołek w grafie oraz powrocie do wierzchołka początkowego.
W rozwiązywanym problemie, rozważamy dodatkowo asymetryczność (ATSP), co oznacza że droga z A do B $\neq$ z B do A.
Główną przeszkodą w rozwiązywaniu problemu jest spora ilość możliwych kombinacji dróg. Dla ATSP wynosi ona (n-1)!, gdzie n = ilość wierzchołków. Dla dużych instancji, czas w jakim znajduje się wszytkie kombinacje (metoda przeglądu zupełnego), jest nieakceptowalny. Dlatego też do rozwiązania owego problemu wykorzystuje się różne algorytmy.

2 Algorytmy

2.1 Przegląd zupełny

Wspomniany juz wcześniej przegląd zupełny (ang. Brute Force) polega na wygynerowaniu wszystkich możliwych rozwiązań oraz na wybraniu optymalnego rozwiązania. Jest to metoda dokładna, która daje zawsze najlepszy wynik.
Zaletą tego podejścia jest łatwa implementacja algorytmu na jego podstawie. Wadą za to jest duża złożoność obliczeniowa która wynosi: **O(n!)**
W TSP metoda ta polega na:

1. wygenerowaniu wszystkich możliwych cyklów Hamiltona
2. wyborze cyklu, która ma najmniejszą sumę wszystkich dróg między miastami w cyklu.

2.2 Podział i ograniczenia

Metoda podziału i ograniczeń (ang. Branch&Bound) dzieli się na (jak sama nazwa wskazuje) na:

- **Podział**, które polega na dzieleniu zbioru rowiązań reprezentowanego przez węzeł drzewa na rozłączne podzbiory, reprezentowane przez nastepników tego węzła.
- **Ograniczenia**, które polega na pomijaniu przeszukiwaniu wierzchołków, które nie mają juz szans na zawieranie optymalnego rozwiązania.

Przeszukiwanie całego drzewa byłoby kosztowne, dlatego też, dla każdego węzła obliczane jest ograniczenie, które pozwala na stwierdzenie czy dane rozwiązanie może być optymalne. Pozwala to na zmniejszenie ilości odwiedzanych wierzchołków i przyspieszyć czas rozwiązywania.
W obliczanym danym poddrzewie rozwiązań, poszukiwane są takie rozwiązania, które mieszczą się pomiędzy dolną, a górną granicą. Górne ograniczenie to najlepsze znalezione rozwiązanie dotychczas. Wartość ograniczenia dolnego musi obliczana w taki sposób, aby każdy potomek dla którego owe ograniczenie jest obliczane, musi być równe, bądź większe. Jest to robione, za pomocą heurustycznego oszacowania najlepszego rozwiązania, które może zostać odnalezione w przez potomków danego wierzchołka.
W projekcie, jako strategia to odwiedzania wierzchołków, została użyta strategia: LeastCost.
Złożoność obliczeniowa dla TSP, dla tej metody, jest bardzo zależna od charakterystki grafu. W najlepszym przypadku wynosi ono: **n**, a w najgorszym osiągamy metodę bruteForce, czyli: **n!**.

2.2.1 Przykład - Branch&Bound

Macierz sąsiedztwa grafu, na potrzeby przykładu:

	1	2	3	4
1	-1	7	4	10
2	3	-1	6	5
3	11	9	-1	10
4	6	2	9	-1

, gdzie -1, oznacza brak połączenia.

Krok 1: Zamień -1 na ∞ (w kodzie na INT32_MAX)

	1	2	3	4
1	∞	7	4	10
2	3	∞	6	5
3	11	9	∞	10
4	6	2	9	∞

Krok 2: Stworzenie korzenia drzewa. Będzie nim wierzchołek **1**, z którego zaczynamy drogę.
Do obliczenia dolnego ograniczenia korzenia, należy dokonać: redukcji macierzy oraz kolumn. Czyli należy wybrać z każdego wiersza najmniejszą wartość

i odjęciu jej od pozostałych. To samo z kolumnami.

	1	2	3	4	Redukcja wierszy
1	∞	7	4	10	4
2	3	∞	6	5	3
3	11	9	∞	10	9
4	6	2	9	∞	2

	1	2	3	4
1	∞	3	0	6
2	0	∞	3	2
3	2	0	∞	1
4	4	0	7	∞
Redukcja kolumn	0	0	0	1

	1	2	3	4
1	∞	3	0	5
2	0	∞	3	1
3	2	0	∞	0
4	4	0	7	∞

Lower bound = 4+3+9+2+0+0+0+1= 19

Za to upper bound na saymym poczatku wynosi ∞. Zmieniamy go na samym początku algorytmu, poprzez wyszukanie 1 dowolnego rozwiązania. W tym przypadku, jest on obliczany w sposób zachłanny: mianowicie robimy to samo co algorytmu tutaj, czyli wybieramy minimum z danego wiersz, następnie znów minimum z rozwiniętego wierzchołka itd. aż dotrzemy do liścia. Wtedy zmieniamy upper bounda na lower bound obliczonego liścia. Różnica jest taka, że w opisywanym tutaj algorytmie, bierzemy minimum z każdego obliczonego wierchołka, a przy obliczaniu upper bounda, bierze, pod uwagę tylko rozważany wiersz.
W owym przykładzie oblicznie upper bound'a, jest równe obliczeniu rozwiązania, stąd też UP=19.

Krok 3: Rozważamy wszystkie pozostałe wierzchołki, do których możemy się dostać z aktualnego rozważanego węzła, oraz te które są mniejsze, bądź

równe upper bound
Dla wierzchołków pozostałych niż korzeń, wygląda to dość podobnie:

1. Zablokowanie wiersza o numerze wierzchołka, z którego wychodzimy
2. Zablokowanie kolumny o numerze wierzchołka, do którego wchodzimy
3. Zablokowanie powrotu do wierzchołka startowego
4. Przeprowadzenie redukcji w taki sam sposób, jak w przypadku korzenia
5. Obliczenie dolnego ograniczenia za pomocą wzoru:

$$LB = LB(rodzica) + obliczonaRedukcja + tab[X, Y] \tag{1}$$

   , gdzie:
   LB(rodzica) - dolne ograniczenie rodzica
   tab[X,Y] - koszt przejścia od ojca (X) do syna (Y), pobrany z macierzy ojca (X)

Rozważamy pierwszą możliwość: 1 - 2

	1	2	3	4
1	∞	∞	∞	∞
2	∞	∞	3	1
3	2	∞	∞	0
4	4	∞	7	∞

Redukcja = (1 + 4) + (2) = 7

	1	2	3	4
1	∞	∞	∞	∞
2	∞	∞	0	0
3	0	∞	∞	0
4	0	∞	1	∞

LB = 19 + 7 + 3 = 29
LB >UP, zatem obcinamy tą gałąź.
Rozważamy drugą możliwość: 1 - 3

	1	2	3	4
1	∞	∞	∞	∞
2	0	∞	∞	1
3	∞	0	∞	0
4	4	0	∞	∞

Redukcja = (0) + (0) = 0

	1	2	3	4
1	∞	∞	∞	∞
2	0	∞	∞	1
3	∞	0	∞	0
4	4	0	∞	∞

LB = 19 + 0 + 0 = 19
LB $\not>$ UP, zatem zostawiamy wierzchołek
Rozważamy drugą możliwość: 1 - 4

	1	2	3	4
1	∞	∞	∞	∞
2	0	∞	3	∞
3	2	0	∞	∞
4	∞	0	7	∞

Redukcja = (0) + (3) = 3

	1	2	3	4
1	∞	∞	∞	∞
2	0	∞	0	∞
3	2	0	∞	∞
4	∞	0	4	∞

LB = 19 + 3 + 5 = 27
LB >UP, zatem obcinamy tą gałąź.

Krok 4. Następny wierzchołek, który rozwijamy to ten, który ma najmniejsze dolne ograniczenie, czyli wierzchołek **3**.

Rozważamy pierwszą możliwość: 1 - 3 - 2

	1	2	3	4
1	∞	∞	∞	∞
2	∞	∞	∞	1
3	∞	∞	∞	∞
4	4	∞	∞	∞

Redukcja = (1) + (4) = 5

	1	2	3	4
1	∞	∞	∞	∞
2	∞	∞	∞	0
3	∞	∞	∞	∞
4	0	∞	∞	∞

LB = 19 + 5 + 0 = 24
LB >UP, zatem obcinamy tą gałąź.

Rozważamy drugą możliwość: 1 - 3 - 4

	1	2	3	4
1	∞	∞	∞	∞
2	0	∞	∞	∞
3	∞	∞	∞	∞
4	∞	0	∞	∞

Redukcja = (0) + (0) = 0

	1	2	3	4
1	∞	∞	∞	∞
2	0	∞	∞	∞
3	∞	∞	∞	∞
4	∞	0	∞	∞

LB = 19 + 0 + 0 = 19
LB $\not>$ UP, zatem zostawiamy wierzchołek

Krok 5. Powtórz krok 4. dopóki nie dojdziesz do liścia, który będzie jednocześnie optymalnym rozwiązaniem.

Zatem, wybieramy wierzchołek 4.
Rozważamy jedyną możliwość: 1 - 3 - 4 - 2

	1	2	3	4
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	∞
4	∞	∞	∞	∞

Redukcja = (0) + (0) = 0

	1	2	3	4
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	∞
4	∞	∞	∞	∞

LB = 19 + 0 + 0 = 19

Doszliśmy do liścia, który LB $\not>$ UP, zatem jest to nasze rozwiązanie.
Ścieżka: 1 - 3 - 4 - 2 - 1
Koszt: 19

2.2.2 Implementacja B&B

Wykorzystane struktury:

- Graf przechowywany jest w dynamicznej tablicy dwuwymiarowej
- Do obliczania algorytmu, wykorzystywana jest kolejka priorytetowa:

```
std::priority_queue<NodeBB*,
    std::vector<NodeBB*>,compareLowerBound> queue;
```

```
//stworzenie struktury do implemenetacji kolejki
    priorytetowej, aby moc sortowac obiekty po zmiennych
    "lowerBound" oraz pierwszenstwo ma wierzcholek o wyzszym
    poziomie, gdy jest sytuacja rowna
struct compareLowerBound {
    bool operator()(NodeBB* const & n1, NodeBB* const & n2)
    {
        if (n1->lowerBound == n2->lowerBound)
        {
            return n1->level < n2->level;
        }
        return n1->lowerBound > n2->lowerBound;
    }
};
```

- Do kolejki dodawne są wierzchołki **NodeBB**, które posiadaja informacje o:
  - numerze wierzcołka w grafie oryginalnym
  - swoją własną, zredukowaną macierz (za pomocą vectora)
  - wskaźnik na ojca
  - dolne ograniczenie
  - poziom w drzewie

Obliczanie ograniczeń odbywa się za pomocą 2 funkcji:

- void blockMatrix(NodeBB *node, NodeBB* father)

```
    void BranchAndBound::blockMatrix(NodeBB *node, NodeBB*
        father) {
    ///BLOKOWANIE WIERSZA/KOLUMNY W MACIERZY
    for (int i = 0; i < matrix->getNodesCount(); ++i) {
        node->table[father->numberOfNode][i] = INT32_MAX;
            //Blokowanie wiersza
        node->table[i][node->numberOfNode] = INT32_MAX;
            //Blokowanie kolumny
    }
    node->table[node->numberOfNode][matrix->getStartNode()] =
        INT32_MAX; //Blokowanie powrotu do wierzcholka
        poczatkowego

}
```

- void countMatrix(NodeBB* node, NodeBB* father)

```
void BranchAndBound::countMatrix(NodeBB* node, NodeBB*
    father) {
    int *tableCountRow = new int [matrix->getNodesCount()];
    int *tableCountColumn = new int [matrix->getNodesCount()];

    ///Przeszukujemy wszystkie wiersze w poszukiwaniu stalych
        do usuniecia z macierzy
    for (int i = 0; i < matrix->getNodesCount(); ++i) {
    //przypisujemy nieskonczonosc, aby znalesc minimalna
        liczbe w wierszu
        tableCountRow[i] = INT32_MAX;
        for (int j = 0; j < matrix->getNodesCount(); ++j) {
            if (node->table[i][j] < tableCountRow[i])
            //szukanie liczby najmniejszej w wierszu
                tableCountRow[i] = node->table[i][j];
        }
        if (tableCountRow[i] != INT32_MAX && tableCountRow[i]
            != 0) {
        //jesli nie jest to nieskonczonosc i 0, to znaleziono
            stala to odjecia
            for (int j = 0; j < matrix->getNodesCount(); ++j) {
                if (node->table[i][j] != INT32_MAX)
                //w wierszu usuwamy najmniejsza liczbe
                    znaleziona w wierszu
                    node->table[i][j] -= tableCountRow[i];
            }
        } else
        //jesli po przeszukaniu, znaleziono same
            nieskonczonosci to zapisz 0
            tableCountRow[i] = 0;
    }

    ///Przeszukujemy wszystkie kolumny w poszukiwaniu stalych
        do usuniecia z macierzy
    for (int i = 0; i < matrix->getNodesCount(); ++i) {
    //przypisujemy nieskonczonosc, aby znalesc minimalna
        liczbe w wierszu
        tableCountColumn[i] = INT32_MAX;
        for (int j = 0; j < matrix->getNodesCount(); ++j) {
            if (node->table[j][i] < tableCountColumn[i])
            //szukanie liczby najmniejszej w wierszu
```

```
                tableCountColumn[i] = node->table[j][i];
        }
        if (tableCountColumn[i] != INT32_MAX &&
            tableCountColumn[i] != 0) {
            for (int j = 0;
                 j < matrix->getNodesCount(); ++j) {
    //jesli nie jest to nieskonczonosc i 0, to znaleziono
        stala to odjecia
                if (node->table[j][i] != INT32_MAX)
                //w wierszu usuwamy najmniejsza liczbe
                    znaleziona w wierszu
                    node->table[j][i] -= tableCountColumn[i];
            }
        } else
        //jesli po przeszukaniu, znaleziono same
            nieskonczonosci to zapisz 0
            tableCountColumn[i] = 0;
    }

    ///Obliczanie lower bound
    //Lower bound = lower bound(father) + costOfPath from
        Father to Son + reduction
    for (int i = 0; i < matrix->getNodesCount(); ++i) {
        node->lowerBound += tableCountRow[i] +
            tableCountColumn[i];
    }

    if (father != nullptr) { //ograniczenie dla korzenia
        node->lowerBound += father->lowerBound;
        node->lowerBound +=
            father->table[father->numberOfNode][node->numberOfNode];
    }

    delete[] tableCountColumn;
    delete[] tableCountRow;
}
```

2.3 Programowanie Dynamiczne

Kolejną metodą jest programowanie dynamiczne, które polega na rozkładaniu problemu na podproblemy, a następnie ich zapamiętywanie. Działa podobnie jak metoda dziel i zwyciężaj, lecz z tą różnicą, że zapamiętuje roz-

wiązania, aby nie rozwiązywać ich ponownie.
W problemie komiwojażera pozbywamy się powtarzania obliczania części ścieżek, które powtarzają się w innych rozwiązaniach. W ten sposób, zmniejszamy złożoność z $\mathbf{O(n!)}$, do $\mathbf{O}(2^n n^2)$.

Implementacja DP:
Wykorzystane struktury:

- Graf przechowywany jest w dynamicznej tablicy dwuwymiarowej
- Wektor dwuwymiarowy, który przechowuje obliczone już podproblemy
- Wektor dwuwymiarowy, który przechowuje wierzchołki, potrzebne do odwzorowania ścieżki

```
//stala maska, ktora ma na kazdej pozycji 1, np. dla 4
   miast: 1111
int maskAllVisited;
//koncowy wynik
int result;
//tablica o rozmiarach N x 2^N, ktora przechowuje
   rozwiazane juz podproblemy
std::vector<std::vector<int>> subProblems;
//talblica dodatkowa, ktora bedzie przechowywac nam
   wierzcholki do sciezki
std::vector<std::vector<int>> tablePath;
```

Algorytm wyglada następująco:

```
void DynamicPrograming::main() {
    result =
        DPFunction(1<<matrix->getStartNode(),matrix->getStartNode());
}

int DynamicPrograming::DPFunction(int mask, int node) {
    //jesli wszystkie miasta zostaly odwiedzone, to zwróć drogę do
        początkowego wierzchołka
    if (mask == maskAllVisited)
    {
        return matrix->getWsk()[node][0];
    }
    //jeśli istnieje już rozwiązany podproblem --> zwróć wartość
    if (subProblems[node][mask] != -1)
    {
        return subProblems[node][mask];
    }

    int subResult = INT32_MAX;

    //przeszukiwanie nieodwiedzonych miast, rozwiązywanie
        podproblemów
    for (int i = 0; i < matrix->getNodesCount(); ++i) {
        //jeśli miasto nie jest odwiedzone --> rozwiąż podproblem
        //Operacja AND na masce i 1 na pozycji danego miasta
        if ((mask&(1<<i)) == 0)
        {
            //rozwiąż kolejny podproblem
            int newResult = matrix->getWsk()[node][i] +
                DPFunction(mask|(1<<i),i);
            if (min(subResult,newResult) == newResult)
            {
                subResult = min(subResult,newResult);
                tablePath[node][mask] = i;
            }
        }
    }
    subProblems[node][mask] = subResult;

    return subResult;
}
```

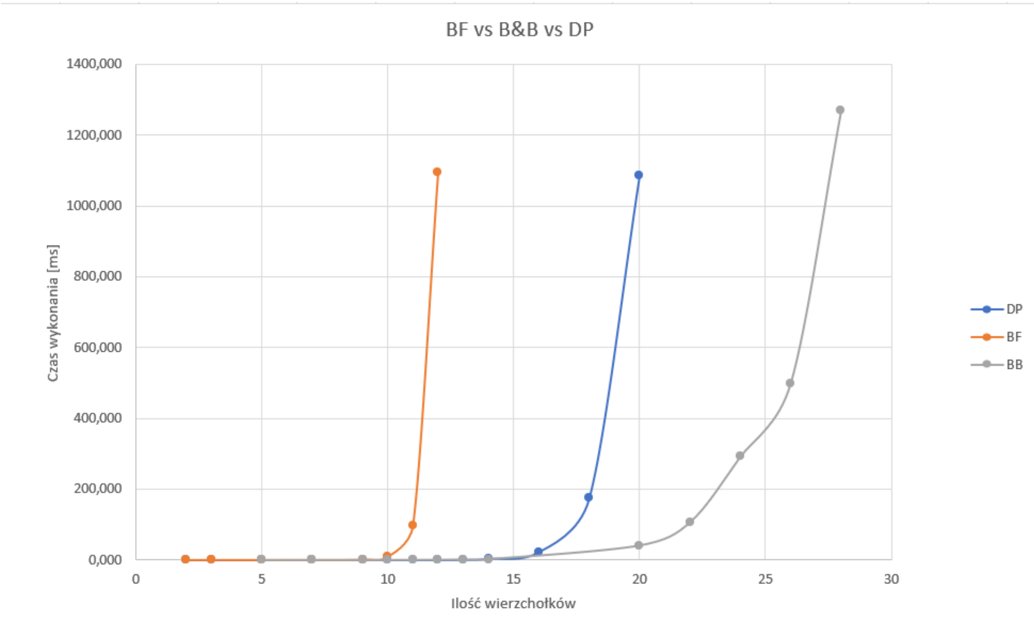
3 Plan eksperymentu

Program został napisany w języku C++ w środowisku CLion i testowany byl w wersji Release.
Grafy wejściowe były generowane jako dynamiczne tablice o wartościach z przedziału od 0 do 100. Dane na przekątnej są potraktowane jako nieskończoności (posiadają wartość = -1). Pomiar czasu został przeprowadzony przy pomocy wykorzystania zegara z biblioteki **<chrono>**: std::chrono::high_resolution_clock.

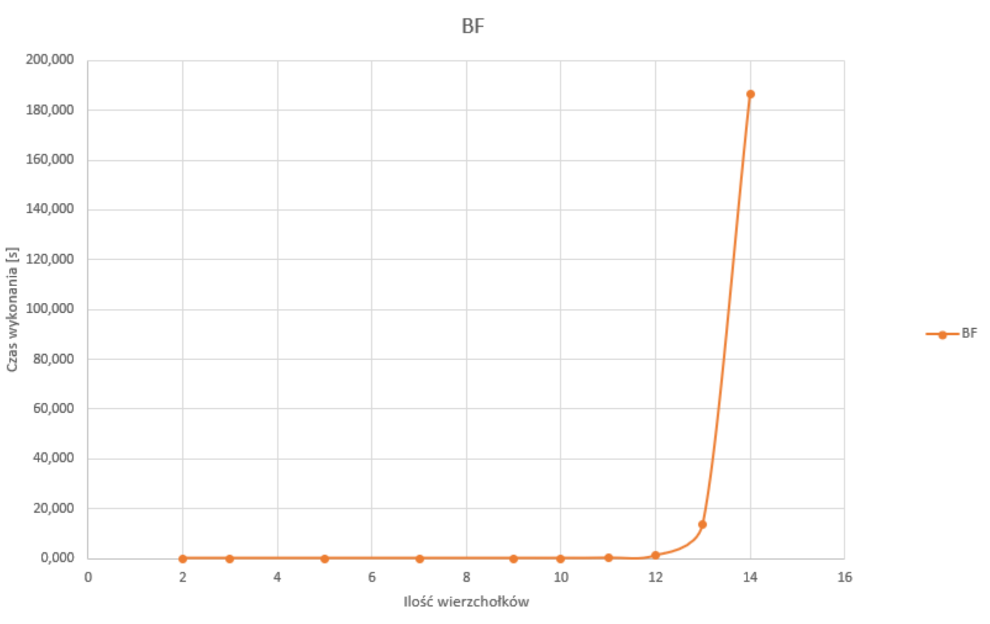
4 Analiza danych pomiarowych

	Czas wykonania dla danego algorytmu [ms]		
Rozmiar problemu	Brute Force	Branch & Bound	Dynamic Programming
2	0,001	0,024297	0,001
3	0,002	0,066525	0,001
5	0,003	0,198024	0,002
7	0,021	0,33341	0,009
9	1,052	0,531619	0,056
10	9,578	0,832298	0,136
11	97,111	1,24932	0,324
12	1094,360	2,57627	0,761
13	13808,400	41,5077	1,862
14	186536,000	107,037	4,470
16	-	292,392	23,600
18	-	500,373	175,064
20	-	1270,63	1085,540
22	-	-	6080,820
23	-	-	14077,600
24	-	-	32733,700

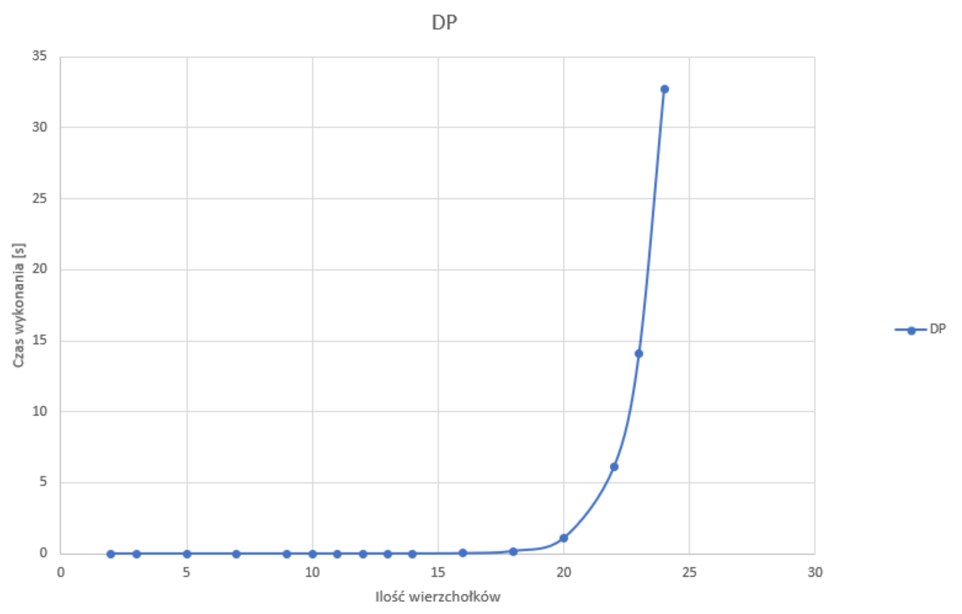
Tabela 1: Wyniki pomiarów



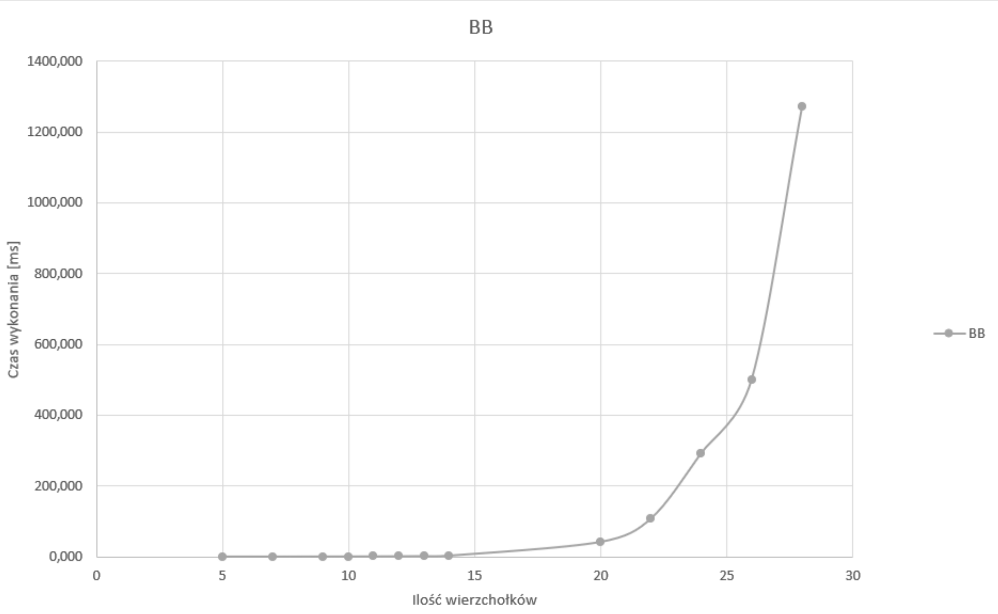
Rysunek 1: Wszystkie algorytmy



Rysunek 2: Brute force



Rysunek 3: Dynamic Programming



Rysunek 4: Branch & Bound

5 Wnioski

Analizując wszystkie wykresy oraz wyniki, można dojść do wniosku, że wyniki dla algorytmu Brute force oraz Dynamic Programming, są zgodne z przewidywaniami. Przegląd zupełny pokazuje dobrze klasę obliczeniową O(n!). Badania dla n>14, nie miały już sensu, gdyż zajęłyby one 15 razy więcej czasu niż dla n=14, czyli $15 * 168s = 46,5min$, czyli kompletnie nieakceptowalny czas.
Programowanie dynamiczne również dobrze pokazuje klase obliczeniową $\mathbf{O}(2^n n^2)$. Dzięki zapamiętywaniu podproblemów, poradził sobie znacznie lepiej niz przegląd zupełny.
Wyniki dla Branch&Bound wyszły dość niezgodnie z oczekiwaniami, gdyż B&B okazał się lepszy niz DP. Może to wynikać z różnych rzeczy.
Po pierwsze może być to kwestia losowości, gdyż algorytym ten wypada różnie w zależności od wartości na grafie.
Po drugie użycie kolejki priorytetowej do przeszukiwania grafu, mogło polepszyć efekty działania tego algorytmu.