Politechnika Wrocławska
Wydział Elektroniki
Informatyka

Projektowanie efektywnych algorytmów

Sprawozdanie

Zadanie 1. Problem komiwojażera

Autor:
Mikołaj Sawicki, 248883

Termin zajęć:
Poniedziałek 15:15-16:55

Prowadzący:
Dr inż. Jarosław Mierzwa

19 listopada 2020

Spis treści

1 Opis problemu **2**

2 Założenia projektowe **2**

3 Zaimplementowane algorytmy **3**
3.1 Przegląd zupełny . . . 3
3.1.1 Drzewo przeszukań . . . 3
3.1.2 Przegląd permutacji . . . 5
3.1.3 Złożoność obliczeniowa przeglądu zupełnego . . . 6
3.2 Algorytm podziału i ograniczeń . . . 7
3.2.1 Działanie algorytmu podziału i ograniczeń - przykład . . . 8
3.2.2 Złożoność obliczeniowa algorytmu podziału i ograniczeń . . . 10

4 Struktury danych w programie **11**
4.1 Klasa Graph . . . 11
4.2 Klasa SquareMatrix . . . 11
4.3 Klasa TSPSolver . . . 11

5 Pomiar czasu działania algorytmów **12**
5.1 Plan eksperymentu . . . 12
5.2 Wyniki eksperymentu . . . 13
5.2.1 Przegląd zupełny - permutacje . . . 13
5.2.2 Przegląd zupełny - drzewo przeszukań . . . 14
5.2.3 Algorytm podziału i ograniczeń . . . 15
5.2.4 Porównanie czasów działania wszystkich algorytmów . . . 16

6 Wnioski **17**

7 Źródła **17**

1 Opis problemu

Problem komiwojażera (ang. Travelling Salesman Problem, TSP) to przykład optymalizacyjnego problemu NP-trudnego (takiego, którego rozwiązanie potrafimy znaleźć tylko w czasie wyższym, niż wielomianowy).

Wyobraźmy sobie komiwojażera, który podróżuje pomiędzy miastami. Komiwojażer chce odwiedzić wszystkie miasta w regionie, przemierzając jak najmniejszy dystans. Każde miasto chce odwiedzić tylko raz. Musi więc znaleźć drogę biegnącą od danego miasta, przez wszystkie kolejne, aż wróci do miasta początkowego. Komiwojażer chce, żeby ta droga była najkrótszą z możliwych.

Problem ten możemy przedstawić za pomocą teorii grafów. Rozpatrujemy graf, którego wierzchołkami są rozważane przez nas miasta. Krawędzie pomiędzy wierzchołkami mają przypisane wagi (wagi te mogą odpowiadać odległościom pomiędzy miastami, czasom przejazdów pomiędzy miastami, kosztom podróży, itp.). Rozpatrujemy ścieżki, które zaczynają się w danym wierzchołku k, przechodzą dokładnie raz przez każdy z pozostałych wierzchołków grafu i wracają do wierzchołka k. Problem polega na wybraniu takiej ścieżki, w której suma wag krawędzi będzie najmniejsza. Zatem, poszukiwana przez nas trasa to cykl przechodzący przez każdy wierzchołek grafu dokładnie raz - jest to cykl Hamiltona. Szukamy cyklu Hamiltona o jak najmniejszej sumie wag krawędzi.

Problem komiwojażera jest trudny obliczeniowo. Zauważmy, że jeśli weźmiemy pod uwagę wszystkie możliwe permutacje wierzchołków (a zatem możliwe do zbudowania ścieżki), otrzymamy n! permutacji, gdzie n to liczba wierzchołków. Dla ścieżki w określonym początku mamy więc (n-1)! możliwych permutacji. My chcemy znaleźć tę jedną - optymalną.

Instancje tego problemu mogą oczywiście różnić się parametrami. Po pierwsze, możemy minimalizować lub maksymalizować sumę wag krawędzi. Wagom krawędzi mogą odpowiadać odległości pomiędzy miastami, czasy przejazdów, koszty podróży. Problem może być symetryczny (gdy waga krawędzi od p do q jest taka sama, jak waga krawędzi od q do p) lub asymetryczny (gdy krawędzie są skierowane - wagi p->q i q-p mogą się wówczas różnić). Problem komiwojażera nie musi obrazować podróży pomiędzy miastami. Wierzchołkami grafu mogą być np. rozplanowane na płytce drukowanej otwory - naszym zadaniem jest wówczas znaleźć najkrótszą trasę dla maszyny wykonującej otwory. Przykładów rzeczywistego występowania TSP oczywiście jest więcej.

2 Założenia projektowe

1. Mamy do czynienia z asymerycznym problemem komiwojażera.
2. Program tworzony jest w języku C++, zgodnie z obiektowym paradygmatem programowania.
3. Zaimplementowane zostają następujące algorytmy:
   (a) Przegląd zupełny przy użyciu drzewa przeszukań (rekurencyjny)
   (b) Przegląd zupełny przy użyciu permutacji (iteracyjny)
   (c) Algorytm podziału i ograniczeń (Branch Bound)

3 Zaimplementowane algorytmy

3.1 Przegląd zupełny

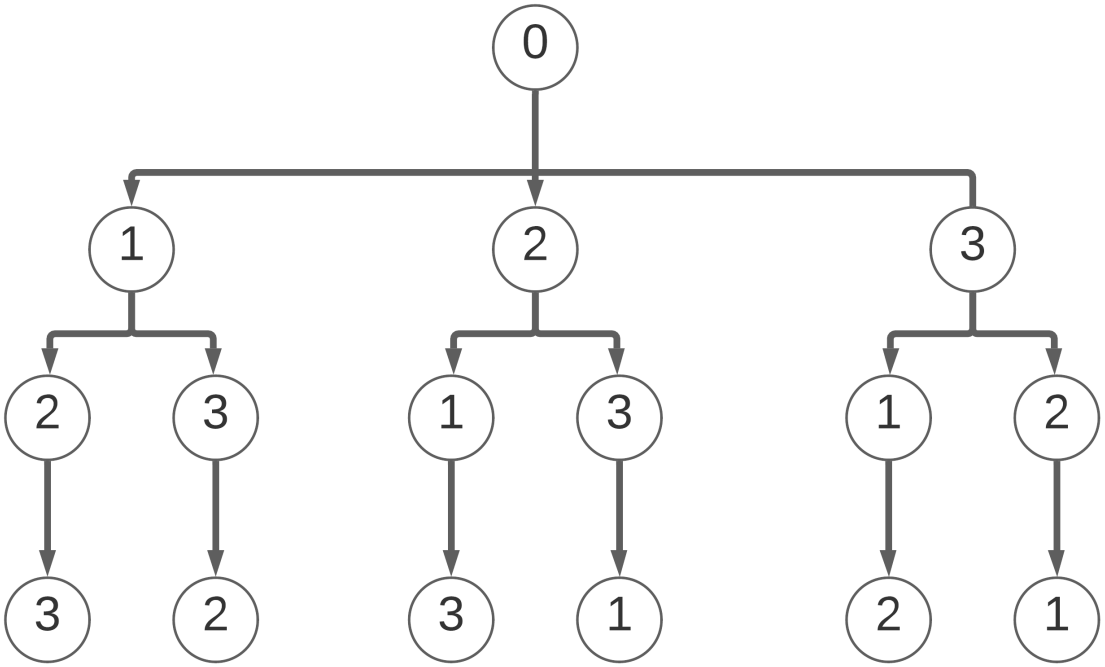
Przegląd zupełny (często nazywany metodą Brute Force) to najbardziej oczywiste rozwiązanie problemu. Polega ono na obliczeniu wszystkich możliwych ścieżek spełniających zadane warunki i wybranie tej optymalnej - a więc tej o najmniejszej sumie wag krawędzi. Niżej przedstawione zostaną dwa algorytmy przeglądu zupełnego:

– drzewo przeszukań (rekurencyjny),

– przegląd permutacji (iteracyjny).

3.1.1 Drzewo przeszukań

Algorytm polega na rekurencyjnym sprawdzeniu wszystkich możliwych ścieżek w drzewie przeszukań i wybraniu ścieżki optymalnej.



Rysunek 1: Drzewo przeszukań

Na początku tworzymy wektor z rozpatrywaną ścieżką - wchodząc do kolejnych wywołań funkcji checkAllPaths, rekurencyjnie rozbudowujemy wektor o nowe węzły. Gdy dotrzemy do liścia drzewa decyzyjnego (czyli do ostatniego dostępnego - nieużytego jeszcze - węzła), wówczas porównujemy obliczoną sumę wag z dotychczasowo najlepszą. Następnie, poprzez powrót do macierzystego wywołania, wracamy na wyższe poziomy naszego drzewa decyzyjnego. Dodajemy do ścieżki ich kolejne węzły-dzieci. Sprawdzamy w ten sposób wszystkie możliwości.

```
void TSPSolver::solveBruteForceSearchTree()
{
  std::vector<int> path;
  path.push_back(startNode);

  bestPathWeight = INT_MAX;

  memset(visited, false, (graph->getNodesCount() + 1) * sizeof(bool
   ));

  visited[startNode] = true;
  //

  solveBruteForceSearchTreeRecursive(path);
}
```

```
void TSPSolver::solveBruteForceSearchTreeRecursive(std::vector<int>
    path)
{
  SquareMatrix* matrix = graph->getAdjacencyMatrix();

  if (path.size() < graph->getNodesCount())
  {
    for (int i = 0; i < graph->getNodesCount(); i++)
    {
      if (!visited[i])
      {
        path.push_back(i);
        visited[i] = true;

        solveBruteForceSearchTreeRecursive(path);

        visited[i] = false;
        path.pop_back();
      }
    }
  }
  else if ((*matrix)[path.back()][path.front()] > 0) {
    path.push_back(path.front());

    int path_weight = 0;

    for (int i = 0; i < path.size() - 1; i++)
    {
      path_weight += (*matrix)[path[i]][path[i + 1]];
    }

    if (bestPathWeight == INT_MAX || path_weight < bestPathWeight)
    {
      for (int i = 0; i < matrix->getSize(); i++)
        bestPath[i] = path[i];

      bestPathWeight = path_weight;
    }

    path.pop_back();

    return;
  }
}
```

3.1.2 Przegląd permutacji

Algorytm bazuje na przeglądzie generowanych iteracyjnie kolejnych permutacji wierzchołków grafu. Rozwiązanie to, jako że jest implementowane iteracyjnie, jest w praktyce szybsze niż to rekurencyjne.
W programie skorzystano z funkcji next_permutation z biblioteki algorithm.
Bazuje ona na zamianie kolejnych elementów miejscami. Działa ona na zasadzie budowy liczb z cyfr w systemach liczbowych. Dokładniej: kiedy wszystkie liczby na prawo od rozważanej przez nas liczby n w danym porządku są w kolejności nierosnącej, wówczas znajdujemy największą liczbę m w tym porządku. Następnie, przemieszczamy n na prawą od m.

...
1 5 3 2
2 1 3 5
...
2 5 3 1
3 1 2 5
...

Rysunek 2: Przykładowe permutacje

Poniżej przedstawiono listing kodu funkcji next_permutation.

```
template<typename It>
bool next_permutation(It begin, It end)
{
        if (begin == end)
                return false;

        It i = begin;
        ++i;
        if (i == end)
                return false;

        i = end;
        --i;

        while (true)
        {
                It j = i;
                --i;

                if (*i < *j)
                {
                        It k = end;

                        while (!(*i < *--k))

                        iter_swap(i, k);
                        reverse(j, end);
                        return true;
                }



```

```
                if (i == begin)
                {
                        reverse(begin, end);
                        return false;
                }
        }
}
```

Kod programu wykorzystujący funkcję next_permutation:

```
void TSPSolver::solveBruteForce()
{
  int weight;
  int nodes_count = this->graph->getNodesCount();

  int* path = new int[nodes_count + 1];
  setFirstPermutation(path);

  do
  {
    weight = 0;

    for (int i = 0; i < nodes_count; i++)
    {
      weight += graph->getWeight(path[i], path[i+1]);
    }

    if (bestPathWeight == -1 || weight < bestPathWeight)
    {
      std::copy(path, path + nodes_count + 1, bestPath);
      bestPathWeight = weight;
    }
  } while (std::next_permutation(path + 1, path + nodes_count));

  delete[] path;
}
```

3.1.3 Złożoność obliczeniowa przeglądu zupełnego

W algorytmach przeglądu zupełnego sprawdzamy kolejno wszystkie możliwe kombinacje wierzchołków grafu (lub inaczej - wszystkie możliwe ścieżki w drzewie przeszukań). Dla zadanego wierzchołka generujemy zatem (n-1)! możliwych wychodzących z niego ścieżek.
Właściwie nie rozróżniamy tu nawet przypadków optymistycznych i pesymistycznych.
Złożoność algorytmu przeglądu zupełnego wynosi zatem zawsze O(n!).

3.2 Algorytm podziału i ograniczeń

Algorytm Branch & Bound, czyli metoda podziału i ograniczeń opiera się na przeszukiwaniu drzewa reprezentującego przestrzeń rozwiązań problemu. W tym wypadku nie przeszukujemy jednak wszystkich możliwych rozwiązań.
Rekurencyjnie konstruujemy ścieżkę rozwiązania. Na każdym poziomie drzewa przeszukań wyznaczamy ograniczenie (bound). Ograniczenie ma tę własność, że jeśli zostanie przekroczone, znaczy to że nie musimy dalej konstruować aktualnej ścieżki. Wówczas wiemy bowiem, że aktualna ścieżka i tak nie będzie optymalna.
Stosujemy więc odcięcia - porzucamy całe gałęzie drzewa przeszukań, które i tak nie doprowadziłyby nas do optymalnego rozwiązania.
Następnie, zgodnie z pewną strategią wybieramy kolejne drogi - stąd "podział" ("branch" - rozgałęzianie) w nazwie algorytmu. Dla kolejnych rozważanych dróg znów wyliczamy ograniczenie i sprawdzamy, czy nie zostało ono przekroczone.
Poszczególne instancje algorytmu Branch & Bound mogą się od siebie różnić parametrami. Mogą zawierać odmienne funkcje liczące ograniczenie, mogą również bazować na odmiennych strategiach podziału. W praktyce, zawsze dobieramy odpowiednie parametry do naszej instancji problemu. Zwykle staramy się wówczas wykorzystać dodatkową wiedzę o naszej instancji problemu, tak by zoptymalizować algorytm. W naszym przypadku bazować będziemy na danych generowanych losowo - użyjemy więc optymalnej dla takiej sytuacji strategii.

Nasza funkcja obliczająca dolne ograniczenie (lower bound) ma następującą postać:

$$\frac{\sum_{n=0}^{n} min_1(i) + min_2(i)}{2}$$

min_1 to funkcja obliczająca najmniejszą wagę krawędzi wychodzącej z wierzchołka i.

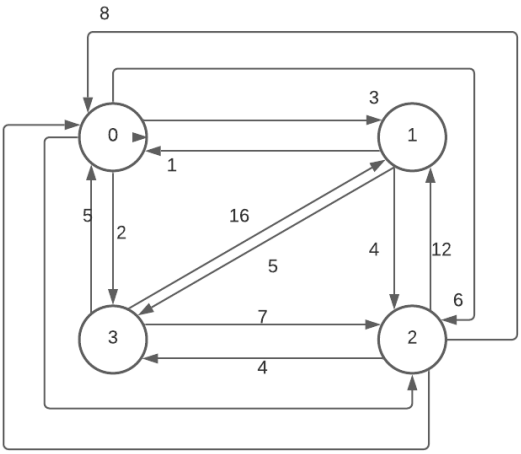
min_2 to funkcja obliczająca drugą najmniejszą wagę krawędzi wychodzącej z wierzchołka i.

Korzystamy tutaj z faktu, że suma wag dwóch dowolnych krawędzi wychodzących z danego wierzchołka jest na pewno większa lub równa sumie dwóch minimalnych wag krawędzi wychodzących z tego wierzchołka.

Suma wag wszystkich krawędzi w grafie spełnia dzięki temu podobną, opisaną wymienionym wyżej wzorem zależność.

3.2.1 Działanie algorytmu podziału i ograniczeń - przykład

Przyjmijmy, że wierzchołek startowy to 0. Szukamy optymalnego rozwiązania problemu komiwojażera dla następującego grafu:

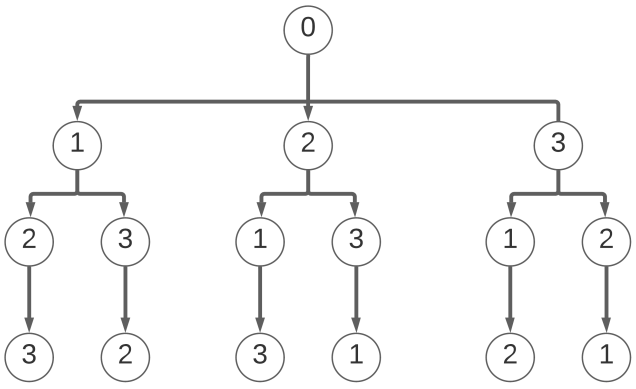


Rysunek 3: Rozpatrywany graf

$$\begin{bmatrix} -1 & 3 & 6 & 2 \\ 1 & -1 & 4 & 5 \\ 8 & 12 & -1 & 4 \\ 5 & 16 & 7 & -1 \end{bmatrix}$$

Rysunek 4: Macierz incydencji rozpatrywanego grafu

Przypomnijmy sobie również, jak wygląda drzewo przeszukań takiego grafu:



Rysunek 5: Drzewo przeszukań

1. Obliczamy ograniczenie dla korzenia. Dla każdego wierzchołka znajdujemy dwie najkrótsze krawędzie z niego wychodzące. Sumujemy ich wagi. Sumujemy obliczone sumy i wynik dzielimy przez 2. Wiemy, że dowolna ścieżka w grafie będzie miała sumę wag większą lub równą tak obliczonemu ograniczeniu.

$$\begin{bmatrix} -1 & \mathbf{3} & 6 & \mathbf{2} \\ \mathbf{1} & -1 & \mathbf{4} & 5 \\ \mathbf{8} & 12 & -1 & \mathbf{4} \\ \mathbf{5} & 16 & \mathbf{7} & -1 \end{bmatrix}$$

Rysunek 6: Minimum dla wierzchołka 0 to 3 i 2, itd.

level = 0
weight = 0
bound = (3+2) + (1+4) + (8+4) + (5+7) = 34
path = [0]

2. Przechodzimy do poziomu 1 drzewa przeszukań. Obliczamy ograniczenie dla węzła 1. Uwzględniamy nową krawędź 0 -> 1. Do starego ograniczenia dodajemy więc wagę tej krawędzi.
level = 1
weight = 3
bound = 34 + 3 = 37
Uwzględniona przez nas krawędź na pewno ma wagę mniejszą lub równą $\frac{min_1(0)+min_1(1)}{2}$. Możemy zatem odjąć tę wartość od ograniczenia (tak, by uwzględnić wartość, jaką wcześniej ta krawędź wygenerowała).
bound = 37 - (1+2)/2 = 35,5 $\approx$ 36

Sprawdzamy, czy ograniczenie nie zostało przekroczone.
$weight + bound = 40 < \infty$
Nie zostało - możemy więc budować ścieżkę dalej.

path = [0, 1]

3. Przechodzimy do poziomu 2 drzewa przeszukań.
Uwzględniamy nową krawędź 1 -> 2.
level = 2
weight = 3 + 4 = 7
bound = 36 + 4 = 40
Minimalna waga krawędzi dla wierzchołka 1 została już odjęta od ograniczenia w poprzednim kroku. Tym razem (na każdym kolejnym poziomie analogicznie) odejmować więc będziemy drugą najmniejszą wagę krawędzi dla tego wierzchołka i pierwszą najmniejszą dla kolejnego wierzchołka.
bound = 40 - $min_2(1) - min_1(2)$ = 40 - 4 - 4 = 32

$weight + bound = 47 < \infty$
path = [0, 1, 2]

Dalej postępujemy analogicznie.

Przykład ten wyjaśnia sposób odcinania gałęzi w drzewie decyzyjnym. Dalej postępować będziemy analogicznie. Gdy dotrzemy do liścia drzewa decyzyjnego, wówczas ustawimy wartość optymalnej ścieżki na obliczoną. W dalszych krokach będziemy porównywać otrzymane ścieżki z otrzymanymi wcześniej.

3.2.2 Złożoność obliczeniowa algorytmu podziału i ograniczeń

Tym razem również przeglądamy możliwe do utworzenia ścieżki drzewa przeszukań. Złożoność pesymistyczna wyniesie więc O(n!).
Znaczną część z tych ścieżek jesteśmy jednak w stanie pominąć dzięki obliczonemu przez nas ograniczeniu. Ograniczenie to pozwala nam bowiem na odcinanie całych gałęzi drzewa przeszukań. Średnia złożoność obliczeniowa, szczególnie dla dużych grafów, będzie więc w tym wypadku mniejsza, niż przy przeglądzie zupełnym.

4 Struktury danych w programie

4.1 Klasa Graph

Klasa Graph przechowuje dane wczytanego do programu grafu.

Atrybuty:

1. SquareMatrix* adjacencyMatrix
2. nodesCount

4.2 Klasa SquareMatrix

Klasa SquareMatrix przechowuje dynamicznie alokowaną tablicę dwuwymiarową.

Atrybuty:

1. **ptr
2. int size

4.3 Klasa TSPSolver

Klasa ta implementuje algorytmy rozwiązujące rozważany przez nas problem komiwojażera.

Atrybuty:

1. Graph* graph
2. int startNode
3. int* bestPath
4. int bestPathWeight
5. bool* visited

Istotne dla algorytmów metody:

1. void solveBranchAndBound()
2. void branchAndBoundRecursive(int level, int* path, int weight, int bound, int* min_1, int* min_2)
3. int startBound(int* min_1, int* min_2)
4. void fillMin(int* min_1, int* min_2)
5. void setFirstPermutation(int* considered_path)
6. void solveBruteForce();
7. void solveBruteForceSearchTree();
8. void solveBruteForceSearchTreeRecursive(std::vector<int> path)

5 Pomiar czasu działania algorytmów

5.1 Plan eksperymentu

Dla każdego algorytmu zmierzony został czas jego wykonania.
Założenia eksperymentu:

1. Do generacji pseudolosowych danych została wykorzystana biblioteka <random>.

2. Dane były generowane przy użyciu rozkładu równomiernego.

3. Dla algorytmów przeglądu zupełnego rozmiar grafu pozostał rzędu od 4 do 11 wierzchołków. Dla algorytmu Branch Bound rozmiar ten wynosił od 4 do 24 wierzchołków.

4. Test dla każdego rozmiaru problemu został powtórzony 100 razy, a wyniki zostały uśrednione. Zapisano również wyniki minimalny i maksymalny, tak by uzyskać odchylenie od średniej dla każdego rozmiaru problemu.

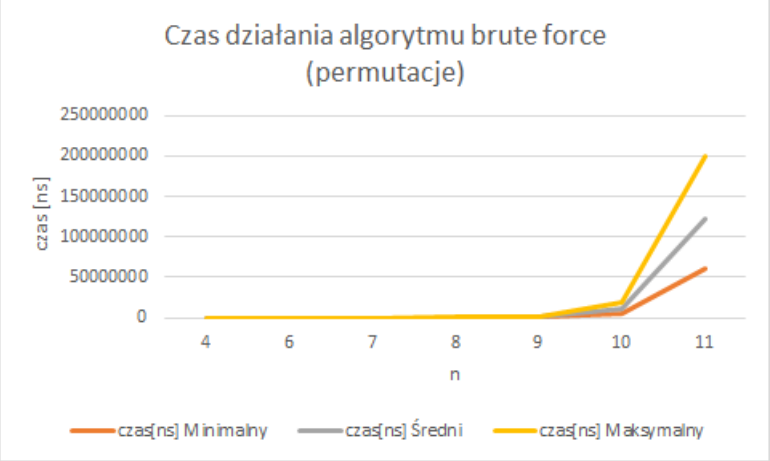
5. Testy przeprowadzone zostały w systemie Windows 10, na procesorze Intel(R) Core(TM) i7-7500U CPU 2.70GHz 2.90GHz

5.2 Wyniki eksperymentu

5.2.1 Przegląd zupełny - permutacje

n	czas [ns]		
	Minimalny	Średni	Maksymalny
4	300	402	1300
6	2100	2334	3600
7	11500	12813	18000
8	82700	83739	137600
9	594700	634055	906200
10	5605800	5813535	8113800
11	59701200	61570171	77412900

Rysunek 7: Tabela przedstawiająca wyniki pomiarów



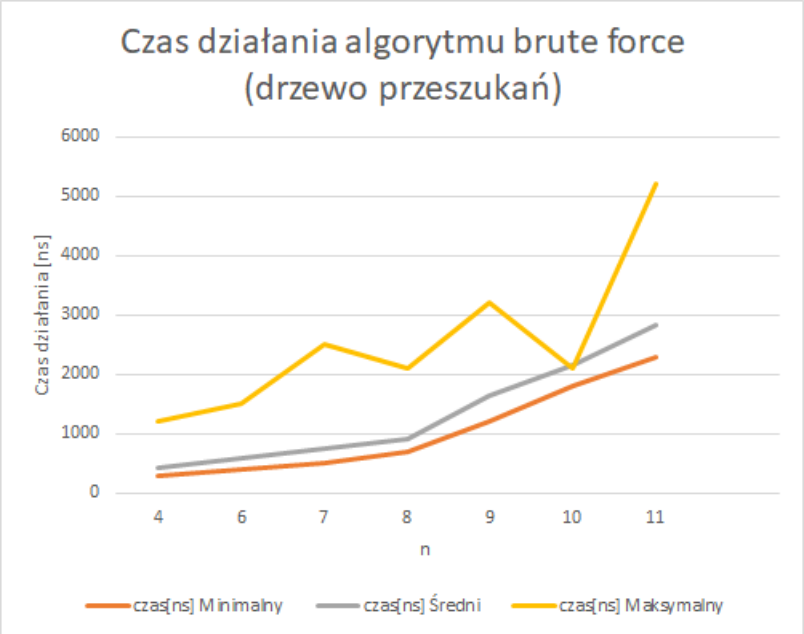
Rysunek 8: Wykres przedstawiający wyniki pomiarów

Dla wartości większych, niż 11, czas otrzymania 100 wyników przekraczał 20min - stąd uwzględniono tylko wartości dla n równego maksymalnie 11. Widzimy, że czas obliczeń rośnie bardzo szybko - o ile dla n równego 4 jest jeszcze stosunkowo mały, to dla n równego 12 jest już nieporównywalnie większy. Jest to zgodne ze złożonością wynikającą z teorii, czyli złożonością O(n!).

5.2.2 Przegląd zupełny - drzewo przeszukań

n	czas [ns]		
	Minimalny	Średni	Maksymalny
4	300	409	1200
6	400	588	1500
7	500	744	2500
8	700	906	2100
9	1200	1638	3200
10	1800	2151	2100
11	2300	2824	5200

Rysunek 9: Tabela przedstawiająca wyniki pomiarów



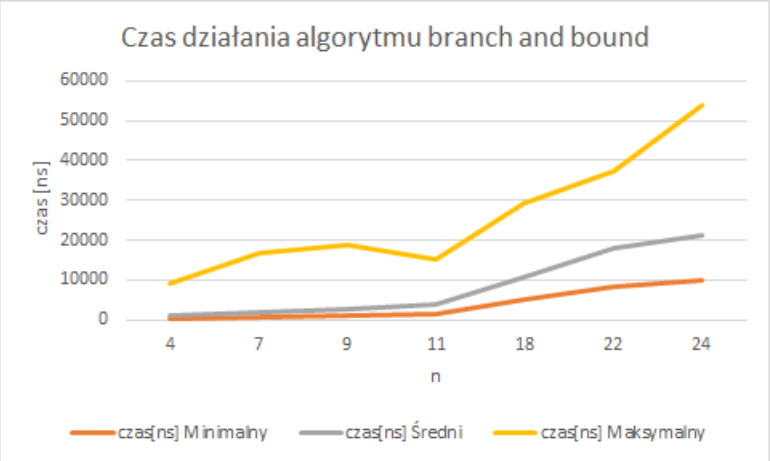
Rysunek 10: Wykres przedstawiający wyniki pomiarów

O ile przedstawione wyniki są o wiele lepsze, niż te uzyskane przy przeglądzie permutacji, o tyle znów dla grafu o 12 wierzchołkach obliczenia trwały zbyt długo. Znów wskazuje to na trafność oceny złożoności O(n!).

5.2.3 Algorytm podziału i ograniczeń

n	czas [ns]		
	Minimalny	Średni	Maksymalny
4	300	644	8000
7	600	1190	14800
9	1000	1644	16200
11	1500	2234	11400
18	5100	5706	18300
22	8200	9820	19500
24	10100	11183	32500

Rysunek 11: Tabela przedstawiająca wyniki pomiarów



Rysunek 12: Wykres przedstawiający wyniki pomiarów

Algorytm podziału i ograniczeń radzi sobie z naszymi instancjami problemu znacznie lepiej. Nawet dla rozmiaru 24 uzyskiwane czasy są jeszcze rozsądne. Dla większych wartości jest jednak trudno uzyskać wyniki. Ciągle więc pozostaje przed nami ograniczenie co do wielkości problemu.

5.2.4 Porównanie czasów działania wszystkich algorytmów

n	czas [ns]		
	Minimalny	Średni	Maksymalny
4	409	402	8000
6	588	2334	
7	744	12813	14800
8	906	83739	
9	1638	634055	16200
10	2151	5813535	9820
11	2824	61570171	11400

Rysunek 13: Tabela przedstawiająca wyniki pomiarów

Rysunek 14: Wykres przedstawiający wyniki pomiarów

Widzimy, że - zgodnie z oczekiwaniami - algorytm podziału i ograniczeń ma nieporównywalną przewagę nad algorytmami przeglądu zupełnego.

6 Wnioski

Uzyskane w eksperymencie wyniki zgodne są z naszymi oczekiwaniami. Ponadto, pokazują że w rzeczywistości algorytm podziału i ograniczeń ma ogromną przewagę nad algorytmami przeglądu zupełnego. Jak widać dzięki przyjętemu przez nas ograniczeniu, w warunkach rzeczywistych odcięcia w drzewie przeszukań istotnie wykonywane są często. Daje nam to znacznie lepszy czas działania - a co za tym idzie, możliwość radzenia sobie z instancjami problemu komiwojażera dla większych grafów, niż dla tych traktowanych algorytmem przeglądu zupełnego.
Wadą algorytmu podziału i ograniczeń mogłaby się tutaj okazać trudność w implementacji. Drugą wadą mogłoby być większe zużycie zasobów pamięciowych.
Warto powtórzyć, że algorytm Branch & Bound zaimplementowany może zostać przy pomocy różnych parametrów - ich dobór odpowiedni do sytuacji (do instancji problemu komiwojażera, z którymi w rzeczywistości chcemy sobie poradzić) może znacznie polepszyć efektywność.

7 Źródła

1. Fast Branch and Bound Algorithm for the Travelling Salesman Problem, Radosław Grymin, Szymon Jagiełło
   https://hal.inria.fr/hal-01637523/document

2. Wykłady o sztucznej inteligencji, dr hab. Maciej Komosiński
   https://www.youtube.com/playlist?list=PLl33p3GENNQGGW-H5pJ5afLMTVR18simL

3. Wykład o problemie komiwojażera, dr hab. Maciej Komosiński
   https://www.youtube.com/watch?v=ichMb7j0YKY

4. Seminarium: Algorytmy heurystyczne, Metoda podziału i ograniczeń, Mateusz Łyczek,
   https://www.ii.uni.wroc.pl/ prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf