Adrian Frydmański 209865 Data oddania: **22 XII 2015r.**Dawid Gracek 209929 Prowadzący: **dr inż. Jarosław Mierzwa**

Projektowanie efektywnych algorytmów  
– laboratorium

**Temat: „Rozwiązywanie problemu komiwojażera metodą podziału i ograniczeń”**

# Zadanie

Należy zaimplementować algorytm metody podziału i ograniczeń dla problemu komiwojażera oraz dokonać testów polegających na pomiarze czasu działania algorytmu w zależności od wielkości instancji.

# Informacje wstępne

Problem komiwojażera opisuje znajdywanie minimalnego cyklu Hamiltona w grafie pełnym. Innymi słowy: komiwojażer ma za zadanie obejść wszystkie miasta, mając do dyspozycji odległości między nimi. Musi to zrobić jak najkrótszą drogą.

Każdy wierzchołek grafu jest miastem, przez które musi przejechać komiwojażer. Grupę *n* miast reprezentuje zbiór   
*N = {1, 2, …, n}*. Drogi pomiędzy miastami reprezentowane są przez macierz   
*D = {dij, i є N, j є N, i ≠ j}*, gdzie *dij* jest odległością pomiędzy miastem *i* i *j*.

Metoda podziału i ograniczeń polega na uporządkowanym przeszukiwaniu zbioru rozwiązań zadania optymalizacyjnego zakładając, że jest skończony. Zbiór wszystkich rozwiązań *D* dzieli się na mniejsze podzbiory, a dla każdego podzbioru obliczany jest kres dolny czyli oszacowanie z dołu wartości funkcji celu. W kolejnych krokach odbywa się podział zbiorów z najmniejszą wartością kresu dolnego. Podział postępuje do chwili znalezienia dopuszczalnego rozwiązania, dla którego wartość funkcji celu jest mniejsza bądź równa najmniejszemu dolnemu ograniczeniu wszystkich niepodzielonych podzbiorów. W najgorszym przypadku konieczne jest sprawdzenie wszystkich rozwiązań, co przesądza o wykładniczej złożoności obliczeniowej tej metody.

Dla wyliczenia kresów dolnych należy zdefiniować funkcję ograniczającą *w*. Musi ona spełniać następujące warunki:



Efektywność tej metody zależy od jakości oszacowań kresów dla każdego podzbioru. Im są one dokładniejsze, tym mniej podziałów potrzeba do uzyskania wyniku. Idealna funkcja ograniczająca jest następująca:



Złożoność obliczeniowa w najgorszym wypadku wynosi: O()

# Opis działania algorytmu

Algorytm korzystający z metody podziału i ograniczeń został podany przez Johna Little’a.

Każdy ze zbiorów rozwiązań, któremu odpowiada wierzchołek w drzewie poszukiwań, jest rozbijany na dwa podzbiory:

* zawierający łuk *<i,j>*
* niezawierający łuku *<i,j>*

Po wykonaniu podziału liczone są kresy dolne poprzez redukcję macierzy kosztów przejść. Podzbiory rozwiązań mające wartości kresów dolnych niemniejsze, niż długości najkrótszego z dotychczas znalezionych rozwiązań są pomijane, co ogranicza przestrzeń poszukiwań.

# Przykład działania algorytmu

Dana jest macierz kosztów C:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *C* | *1* | *2* | *3* | *4* |  |
| *1* | ∞ | 2 | 7 | 3 |  |
| *2* | 7 | ∞ | 8 | 5 |  |
| *3* | 9 | 4 | ∞ | 6 |  |
| *4* | 3 | 8 | 5 | ∞ |  |

Niech G0 oznacza początkowy zbiór rozwiązań. W celu wyznaczenia dolnej granicy dla G0 redukujemy macierz C przez znalezienie elementu minimalnego w każdym z wierszy i odjęcie go od wszystkich elementów danego wiersza oraz wykonanie tego samego procesu dla kolumn.

Po redukcji macierzy, w każdym wierszu i każdej kolumnie znajduje się element zerowy. Suma odjętych wielkości jest dodawana do dotychczasowej wartości dolnego ograniczenia *LB*, które na początku wynosi 0.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *C0* | *1* | *2* | *3* | *4* |  |
| *1* | ∞ | 0 | 3 | 1 | -2 |
| *2* | 2 | ∞ | 1 | 0 | -5 |
| *3* | 5 | 0 | ∞ | 2 | -4 |
| *4* | 0 | 5 | 0 | ∞ | -3 |
|  |  |  | -2 |  | LB=16 |

Po redukcji LB = 16 (dolne ograniczenie). Kandydatami do podziału są odcinki: <1,2>, <2,4>, <3,2>, <4,1> i <4,3>. Mają one w zredukowanej macierzy koszt równy 0. Wybieramy ten, który posiada najwyższy optymistyczny koszt wyłączenia. Wyłączając odcinek <i,j>, z miasta „i” należy wyruszyć do innego miasta, niż „j”, a do miasta „j” z miasta różnego od „i”. Rozważmy dla przykładu odcinek <2,4> – jeżeli go wyłączymy, to z miasta 2 musimy wyruszyć do miasta różnego od 4, a minimalny koszt z tym związany jest równy 1 (minimalna wartość w wierszu 2 różna od tej na pozycji 2,4). Z kolei do miasta 4 trzeba dotrzeć z miasta różnego od 2 i minimalny koszt z tym związany jest równy 1 (minimalna wartość w kolumnie 4 różna od tej na pozycji 2,4). Suma tych wartości (2) jest optymistycznym kosztem który na pewno musimy ponieść wyłączając odcinek <2,4>. Wartości tychże kosztów podane są w postaci górnych indeksów:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *C0* | *1* | *2* | *3* | *4* |  |
| *1* | ∞ | 01 | 3 | 1 | -2 |
| *2* | 2 | ∞ | 1 | 02 | -5 |
| *3* | 5 | 02 | ∞ | 2 | -4 |
| *4* | 02 | 5 | 01 | ∞ | -3 |
|  |  |  | -2 |  | LB=16 |

W tym wypadku trzy odcinki mają optymistyczny koszt wyłączenia równy 2. Wybieramy jeden z nich – odcinek <2,4> tworząc zbiory G1 i G2 – zawierające lub niezawierające go.

G0 (16)

Weźmy zbiór G1. Tworzymy macierz C1 poprzez usunięcie z macierzy C0 wiersza 2 i kolumny 4, i przyjmując, że c42=∞. Ma to na celu blokadę drogi przeciwnej. Zmodyfikowana i zredukowana macierz C1 wygląda następująco:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *C1* | *1* | *2* | | *3* | |  | |
| *1* | ∞ | 03 | | 3 | |  | |
| *3* | 5 | 05 | | ∞ | |  | |
| *4* | 05 | ∞ | | 03 | |  | |
|  |  | |  | |  | | LB=0 |

Weźmy teraz zbiór G2. Tworzymy macierz C2 z macierzy C0 przyjmując c24=∞. Macierz C2 po modyfikacji i redukcji znajduje się poniżej:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *C2* | *1* | *2* | | *3* | | *4* |  | |
| *1* | ∞ | 00 | | 3 | | 01 |  | |
| *2* | 1 | ∞ | | 01 | | ∞ | -1 | |
| *3* | 5 | 01 | | ∞ | | 1 |  | |
| *4* | 01 | 5 | | 00 | | ∞ |  | |
|  |  | |  | | -1 | | LB = 2 |  | |

G1 (16)

<2,4>

G2 (18)

~<2,4>

G0 (16)

Do dalszego podziału wybieramy wierzchołek o najmniejszej wartości dolnego ograniczenia G1. Z macierzy C1 wynika, że mogą być rozpatrywane odcinki <3,2> i <4,1> (zerowy koszt, maksymalna optymistyczna wartość kosztu wyłączenia 5). Arbitralnie wybieramy <3,2>, tworząc zbiory G3 i G4.

Dla zbioru G3 tworzymy macierz C3, usuwając z macierzy C1 wiersz 3 i kolumnę 2. Nie dokonujemy podstawienia c32=∞, gdyż ten element został już wyeliminowany. Zmodyfikowana i zredukowana macierz C3 wygląda tak:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *C3* | *1* | *3* | |  | |
| *1* | ∞ | 0∞ | | -3 | |
| *4* | 0∞ | 00 | |  | |
|  |  | | LB=3 | |  | |

Dla zbioru G4 tworzymy macierz C4 z macierzy C1 przyjmując c32=∞. Zmodyfikowana i zredukowana macierz C4 wygląda następująco:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *C4* | *1* | *2* | | *3* | |  | |
| *1* | ∞ | 0∞ | | 3 | |  | |
| *3* | 0∞ | ∞ | | ∞ | | -5 | |
| *4* | 00 | ∞ | | 03 | |  | |
|  |  | |  | | LB=5 | |  |

G1 (16)

<2,4>

G2 (18)

~<2,4>

G3 (19)

<3,2>

G4 (21)

~<3,2>

G0 (16)

Do dalszego podziału wybieramy wierzchołek z najmniejszą wartością kresu dolnego G2. Z macierzy C2 wynika, że tylko odcinki <1,4>, <2,3> i <3,2> mogą być rozpatrywane (zerowy koszt, maksymalna optymistyczna wartość kosztu wyłączenia 1). Arbitralnie wybieramy <1,4> tworząc zbiory G5 i G6.

W zbiorze G5 tworzymy macierz C5 usuwając z macierzy C2 1 wiersz i 4 kolumnę. Dokonujemy podstawienia c41=∞. Zmodyfikowana i zredukowana macierz C5 znajduje się poniżej:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *C5* | *1* | *2* | | *3* | |  | | |
| *2* | 04 | ∞ | | 00 | |  | |
| *3* | 4 | 09 | | ∞ | |  | |
| *4* | ∞ | 5 | | 05 | |  | |
|  | -1 | |  | |  | | LB=1 | |

W G6 tworzymy macierz C6 z macierzy C2 przyjmując c14=∞. Zmodyfikowana i zredukowana macierz C6 wygląda tak:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *C6* | *1* | *2* | | *3* | | *4* |  | | |
| *1* | ∞ | 03 | | 3 | | ∞ |  | | |
| *2* | 1 | ∞ | | 01 | | ∞ |  | | |
| *3* | 5 | 00 | | ∞ | | 0∞ |  | | |
| *4* | 01 | 5 | | 00 | | ∞ |  | | |
|  |  | |  | | -1 | | |  | LB=1 |

G3 (19)

<3,2>

G4 (21)

~<3,2>

G5 (19)

<1,4>

G6 (19)

~<1,4>

G1 (16)

<2,4>

G2 (18)

~<2,4>

G0 (16)

Do dalszego podziału wybieramy wierzchołek o najmniejszej wartości dolnej granicy w G3. Z tablicy C3 wynika, że tylko odcinki <1,3> i <4,1> mogą być rozpatrywane przez ich zerowy koszt i maksymalną optymistyczna wartość kosztu wyłączenia (∞). Arbitralnie wybieramy <1,3> tworząc zbiory G7 i G8.

W zbiorze G7 tworzymy macierz C7 usuwając z macierzy C3 wiersz 1 i kolumnę 3. Macierz redukuje się do jednego elementu. Zmodyfikowana i zredukowana macierz C7 znajduje się poniżej:

|  |  |  |
| --- | --- | --- |
| *C7* | *1* |  |
| *4* | 0 |  |
|  |  | LB=0 |

Z tej macierzy wynika, że dołączając do wierzchołka G7 jedyną możliwą trasę <4,1> otrzymujemy rozwiązanie końcowe, dla którego wartość funkcji kosztu równa się kresowi dolnemu równemu 19. Oznaczmy to rozwiązanie przez wierzchołek G9.

W zbiorze G8 tworzymy macierz C8 z macierzy C3, podstawiając c13=∞. Zmodyfikowana macierz C8 znajduje się poniżej:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *C8* | *1* | *3* | |  | |
| *1* | ∞ | ∞ | |  | |
| *4* | 0∞ | 00 | |  | |
|  |  | |  | R = ∞ (!) |

Koszt redukcji wynosi ∞ (wiersz 1). Kres dolny byłby równy także ∞. Oznacza to, że wierzchołek ten nie będzie dalej dzielony. Pomijamy więc w dalszych rozważaniach ten wierzchołek.

G7 (19)

<1,3>

G8 (\*)

~<1,3>

G9 (19)

<4,1>

G1 (16)

<2,4>

G2 (18)

~<2,4>

G3 (19)

<3,2>

G4 (21)

~<3,2>

G0 (16)

G5 (19)

<1,4>

G6 (19)

~<1,4>

Podział pozostałych zbiorów nie da lepszego rozwiązania, gdyż otrzymane jest optymalne, choć niekoniecznie jedyne. W celu stwierdzenia czy istnieją inne, należy dokonać podziału zbiorów G5 i G6.

# Implementacja algorytmu

Program został napisany w języku C++ za pomocą środowiska Visual Studio 2015. Poniżej przedstawiona została funkcja znajdująca optymalną drogę za pomocą powyższego algorytmu.

Klasa *Edge* reprezentuje krawędź grafu. *Item* przedmiot listy. *MatrixGraph* to graf z odpowiednimi elementami takimi, jak lista odwiedzonych wierzchołków, czy macierz kosztów. *BnbMatrix* jest klasą stworzoną do rozwiązywania problemu metodą podziału i ograniczeń, reprezentującą pojedynczą macierz, która ulega podziałom.

class Edge

{

public:

int source;

int dest;

float weight;

Edge(int source, int dest, float weight);

};

class Item

{

public:

float value;

Item\* next;

Item();

};

class MatrixGraph {

vector<bool> visited; // tablica odwiedzonych

protected:

vector<vector<float>> nodes; // wierzchołki grafu

public:

void Generate(int N); // generacja grafu

void Load(ifstream &file); // wczytanie grafu z pliku

void LoadTSP(ifstream &file); // wczytanie grafu z pliku

void LoadATSP(ifstream &file); // wczytanie grafu z pliku

void Display(); // wyświetlenie grafu

float DoesEdgeExist(int node1\_index, int node2\_index); // zwraca informację,

// czy krawędź pomiędzy wierzchołkami istnieje

int Density(); // zwraca rzeczywistą gęstość grafu

int GetSize(); // zwraca długość boku macierzy

vector<Edge> EdgesOf(int node\_index); // zwraca krawędzie danego wierzchołka

vector<Edge> Edges(); // zwraca wszystkie krawędzie w grafie

float CheckMaxNodeValue();

vector<vector<float>> GetNodes();

void DisplayMatrix();

};

class BnbMatrix {

vector<vector<float>> nodes; // wierzchołki grafu

EdgePtr edges; // krawędzie grafu

vector<float> minrow; // minimalne wartości wiersza

vector<float> mincol; // minimalne wartości kolumny

vector<unsigned char> rowid; // indeksy wiersza

vector<unsigned char> colid; // indeksy kolumny

pair<unsigned char, unsigned char> minpos; // pozycja minimum

float limit; // granica

float number; // numer

void Reduce(); // funkcja do redukcji

void FindZero(); // szukanie zera

void CutOut(); // odcięcie

void InsertInf(); // wstawienie nieskończoności

void InsertInfRev(); // to samo, ale dla <j,i>, nie <i,j>

public: // poniżej konstruktory i getery

BnbMatrix(MatrixGraph \_graph);

BnbMatrix(BnbMatrix &matrix, bool cutout, float \_number);

unsigned char size;

pair<unsigned char, unsigned char> GetMinPos();

vector<vector<float>> GetNodes();

vector<unsigned char> GetRowID();

vector<unsigned char> GetColID();

float GetLimit();

pair<unsigned char, unsigned char> EdgeOf(unsigned char x, unsigned char y);

EdgePtr GetEdges();

float GetNumber();

void Clean();

};

Poniżej przedstawiona jest funkcja znajdująca optymalną drogę za pomocą metody podziału i ograniczeń.

vector<unsigned char> KomiwojazerBranchAndBound(MatrixGraph graph)

{

Wartość zwracana jest vectorem charów, który mówi o kolejnych miastach do odwiedzenia na najlepszej z możliwych tras. Zastosowany typ zmiennej wynika z optymalizacji pamięci (char zajmuje tylko jeden bajt).

BnbMatrix\* matrix = new BnbMatrix(graph); // rodzic

BnbMatrix\* child1; // lewy syn (z krawędzią)

BnbMatrix\* child2; // prawy syn (bez krawędzi)

vector<unsigned char> solution; // rozwiązanie ostateczne

EdgePtr edges; // wskaźnik na rozwiązanie

// przechowywane w danej macierzy

float matrixcount = 1; // numer rodzica

priority\_queue <P, vector<P>, Order> limits; // kolejka priorytetowa granic macierzy

bool goodsolution = false; // czy dobre rozwiązanie?

Wśród powyższych deklaracji zmiennych jest wskaźnik na *matrix*, która początkowo jest kopią macierzy kosztów z grafu, oraz na macierze dwojga jej synów. Solution jest wcześniej wspomnianym rozwiązaniem – to on zostaje ostatecznie zwrócony. *EdgePtr edges* to tak zwany shared pointer, vector par, opisujących maksymalne minimum jakie istnieje w danej macierzy – to są współrzędne komórki do usunięcia, zastąpienia nieskończonością. *Matrixcount* jest licznikiem stworzonych macierzy. *Limits* jest kolejką priorytetową zawierającą wartości ograniczeń macierzy, *goodsolution* określa poprawność rozwiązania. Każda macierz przechowuje dotychczasowe wierzchołki, które zostały usunięte, np. według poniższego rysunku macierz 5 musi pamiętać krawędzie, które wzięła w macierzy 2 i 1. Natomiast 6 pamięta tylko krawędzie wyliczone w 1, bo cokolwiek było w 3, było bez danej krawędzi.

Rysunek 1 przykładowe drzewo

Główna pętla wykonuje się aż do znalezienia rozwiązania.

while (true) // bo rekurencję zamieniono na iteracje

{

if (matrix->size == 1 && matrix->GetLimit() < FLT\_MAX && matrix->GetNodes()[0][0] == 0)

// sprawdzenie, czy już nie znaleziono rozwiązania - jeśli granica macierzy mniejsza od

nieskończoności i zawartość macierzy = 0, to mozna rozpatrywać, czy to jest dobre rozwiązanie

{

goodsolution = true; // zakładamy, ze dobre, żeby potem ewentualnie zmienić

edges = matrix->GetEdges(); // przypisanie danego rozwiązania

solution.push\_back(0);

for (int j = 1; j <= edges->size(); j++) // generowanie rozwiązania,

//szukanie cyklu Hamiltona

{

for (int k = 0; k < edges->size(); k++)

{

if (edges->at(k).first == solution[j - 1])

// jeśli nie ma drogi wychodzącej z wierzchołka,

// do którego weszliśmy, rozwiązanie błędne

{

solution.push\_back(edges->at(k).second);

break;

}

}

if (solution.size() <= j)

{

goodsolution = false;

break;

}

}

if (goodsolution) // jeżeli do tej pory było ok...

{

for (int k = 1; k < edges->size(); k++) // czy dobre rozwiązanie?

{

if (k < solution.size() - 1 && solution[k] == 0)

// jeżeli rozwiązanie, które sprawdzam jest mniejsze od pełnego rozwiązania,

// to złe; jeśli napotka 0 na wcześniejszej pozycji, to znaczy,

// że za mały cykl - tej macierzy już nie pamiętamy, bo jest do dupy (resize)

{

goodsolution = false;

edges->resize(0);

break;

}

}

unsigned char node;

for (int k = 0; k < solution.size(); k++)

{

if (goodsolution) // jeżeli do tej pory było ok...

{

if (solution[k] != 0)

// sprawdza, czy wierzchołek nie jest 0

{

node = 0;

for (int j = 0; j < solution.size(); j++)

// poszukiwanie wierzchołka w rozwiązaniu

{

if (solution[k] == solution[j])

node++;

if (node > 1)

// jeśli wystąpił więcej, niż raz,

// to rozwiązanie błędne

{

goodsolution = false;

break;

}

}

}

}

else

break;

}

}

if (goodsolution) // jeżeli do tej pory było ok, to znaczy, ze dobre rozwiązanie

{

while (!limits.empty()) // czyszczenie kolejki priorytetowej

{

delete limits.top().second; // usuń wskaźnik na macierz

limits.pop(); // zdejmij element z kolejki priorytetowej

}

return solution; // zwróć poprawny wynik

}

Else // jeśli rozwiązanie złe, to rozwiązanie znów wielkości 0,

// żeby od nowa pushbackować

solution.resize(0);

}

if (matrix->size > 1) // jeśli wielkość macierzy > 1 (jeśli można jeszcze zredukować)

{

matrixcount++; //zwiększ licznik łączny wszystkich macierzy

child1 = new BnbMatrix(\*matrix, true, matrixcount); // utworzenie lewego syna

// z usunętym wierszem i kolumną (matrixcount mówi, który wiersz i kolumna,

// matrix to wkaźnik na rodzica)

child1->Clean(); // usuwa zawartość minimalnych wartości wierszy i kolumn

limits.push(make\_pair(child1->GetLimit(), child1));

// dodaj lewego syna do kolejki priorytetowej

matrixcount++; // znalogicznie dla prawego syna

child2 = new BnbMatrix(\*matrix, false, matrixcount);

// prawy nie wykreśla wiersza i kolumny, a bierze nieskończoność

child2->Clean();

limits.push(make\_pair(child2->GetLimit(), child2));

}

if (matrix->GetNumber() > 0) // jeśli numer macierzy większy od 0

{

delete limits.top().second; // usuwa wskaźnik na następną macierz

// z kolejki priorytetowej (bo 0 macierz nie była w kolejce priorytetowej)

limits.pop(); // zdejmujemy wartość z kolejki, bo będzie teraz rozpatrywana

}

matrix = limits.top().second; // następna macierz bierze wskaźnik z kolejki priorytetowej

}

return vector<unsigned char>(1, 0); // zwrócenie wektora {0}, żeby Visual się nie czepiał

}

Konstruktor użyty przy tworzeniu potomków wygląda następująco:

BnbMatrix::BnbMatrix(BnbMatrix &matrix, bool cutout, float \_number)

{

nodes = matrix.GetNodes(); // wypełnienie macierzy z macierzy rodzica (wierzchołki)

size = (unsigned char)nodes.size(); // liczba wierzchołków

minrow.resize(size, FLT\_MAX); // tablica z minimalnymi wartościami wierszy

mincol.resize(size, FLT\_MAX); // i kolumn

rowid = matrix.GetRowID(); // po usunięciu zmieniają się indeksy,

// rowid przechowuje oryginalne indeksy wierszy

colid = matrix.GetColID(); // colid - kolumn

limit = matrix.GetLimit(); // granica

minpos = matrix.GetMinPos(); // znaleziona komórka z maksymalnym minimum

number = \_number; // numer macierzy

if (cutout) // czy wycinać kolumnę, czy nie?

{

CutOut(); // obetnij

edges = EdgePtr(new vector<pair<unsigned char, unsigned char>>); //wskaźnik na nowe rozwiąz.

edges->resize(matrix.edges->size()); // zwiększ do wielkości poprzedniego rozwiazania

for (auto i = 0; i < edges->size(); i++) // i przepisz

(\*edges)[i] = (\*matrix.edges)[i];

edges->push\_back(pair<unsigned char, unsigned char>(minpos.first, minpos.second));

// komórkę znalezioną dodaj do rozwiązania

}

else

{

InsertInf(); // ustaw nieskończoność

edges = matrix.edges;

}

Reduce(); // redukuj macierz

FindZero(); // znajdź pozycję 0, dla którego minimum kolumny i wiersza jest najwyższe

if (nodes.size() == 1) // jeśli wielkość macierzy = 1 syn nie będzie stworzony,

// więc dodaj pozycję zera do rozwiązań już teraz.

edges->push\_back(pair<unsigned char, unsigned char>(minpos.first, minpos.second));

}

Funkcje w nim użyte:

void BnbMatrix::FindZero()

{

double tmp\_min\_row\_val, tmp\_min\_col\_val, tmp\_min = 0;

for (auto i = 0; i < size; i++) // wiersz

for (auto j = 0; j < size; j++) // kolumna

{

tmp\_min\_row\_val = FLT\_MAX;

tmp\_min\_col\_val = FLT\_MAX;

if (nodes[i][j] == 0) // jeżeli komórka jest 0

{

for (auto k = 0; k < size; k++)

{

if (k != j && nodes[i][k] < tmp\_min\_row\_val)

// jeżeli droga jest mniejsza od min\_row

tmp\_min\_row\_val = nodes[i][k]; // nadpisz min\_row

if (k != i && nodes[k][j] < tmp\_min\_col\_val)

tmp\_min\_col\_val = nodes[k][j]; // nadpisz min\_col

}

if (tmp\_min\_col\_val + tmp\_min\_row\_val >= tmp\_min)

{

tmp\_min = tmp\_min\_col\_val + tmp\_min\_row\_val;

minpos.first = rowid[i];

minpos.second = colid[j];

}

}

}

}

void BnbMatrix::Reduce() {

// znalezienie min w wierszach, odjęcie minimów od wierszy

// znalezienie kolumn w wierszach, odjęcie minimów od kolumn

// sprawdzanie minimów po wierszach

for (auto i = 0; i < size; i++) // wiersz

for (auto j = 0; j < size; j++) // kolumna

if (minrow[i] > nodes[i][j])

minrow[i] = nodes[i][j];

// odejmowanie minimów od wierszy

for (auto i = 0; i < size; i++) // wiersz

{

for (auto j = 0; j < size; j++) // kolumna

{

if (nodes[i][j] != FLT\_MAX)

nodes[i][j] -= minrow[i];

if (nodes[i][j] < 0.0) // jeżeli poniżej 0, to INF

nodes[i][j] = FLT\_MAX;

}

limit += minrow[i];

}

// sprawdzanie minimów po kolumnach

for (auto i = 0; i < size; i++) //kolumna

for (auto j = 0; j < size; j++) //wiersz

if (mincol[i] >nodes[j][i])

mincol[i] = nodes[j][i];

// odejmowanie minimów od kolumn

for (auto i = 0; i < size; i++) // kolumna

{

for (auto j = 0; j < size; j++) // wiersz

{

if (nodes[j][i] != FLT\_MAX)

nodes[j][i] -= mincol[i];

if (nodes[j][i] < 0) // jeżeli poniżej 0, to INF

nodes[j][i] = FLT\_MAX;

}

limit += mincol[i];

}

}

void BnbMatrix::CutOut() // obcina macierz

{

pair<unsigned char, unsigned char> pos = EdgeOf(minpos.first, minpos.second);

nodes.erase(nodes.begin() + pos.first);

rowid.erase(rowid.begin() + pos.first);

colid.erase(colid.begin() + pos.second);

minrow.erase(minrow.begin() + pos.first);

mincol.erase(mincol.begin() + pos.second);

size--;

for (auto i = 0; i < size; i++)

nodes[i].erase(nodes[i].begin() + pos.second);

InsertInfRev();

}

void BnbMatrix::InsertInf() // wstawia nieskończoność

{

pair<unsigned char, unsigned char> infpos = { EdgeOf(minpos.first, minpos.second) };

nodes[infpos.first][infpos.second] = numeric\_limits<float>::infinity();

}

void BnbMatrix::InsertInfRev() // wstawia w przeciwny wierz i kolumnę inf (nie <i, j>, a <j, i>)

{

pair<unsigned char, unsigned char> infpos = { EdgeOf(minpos.second, minpos.first) };

if (infpos.first == UCHAR\_MAX && infpos.second == UCHAR\_MAX)

return;

else

nodes[infpos.first][infpos.second] = numeric\_limits<float>::infinity();

}

void BnbMatrix::Clean() // czyszczenie

{

minrow.clear();

minrow.shrink\_to\_fit();

mincol.clear();

mincol.shrink\_to\_fit();

}

# Testowanie

Pliki wejściowe dla programu mogą być w dwóch postaciach:

* liczba miast, a następnie kolejne wartości z macierzy odległości między nimi;
* plik TSP ze współrzędnymi miast – liczba miast podana po „DIMENSION:” (lub „DIMENSION :”), współrzędne i numery wierzchołków podawane trójkami po „NODE\_COORD\_SECTION”.

Do zbadania czasu działania algorytm był wykonywany 100 razy (dla losowo generowanych instancji o zadanym rozmiarze), a następnie jego wyniki zostały uśrednione. Rozmiary testowanych grafów to wielokrotności czwórki, nie większe niż 32. Dla porównania wykonano przegląd zupełny dla 4 i 8 wierzchołkowych grafów, resztę czasów policzono dzięki znanej złożoności czasowej przeglądu zupełnego: O(n!).

Czas wykonywania się algorytmu był mierzony dzięki własnej klasie *MyTimer*. Zlicza ona takty zegara procesora i dzieli tę liczbę przez jego częstotliwość, a następnie mnoży przez 1000 aby wynik był w *ms.*

# Wyniki

Poniżej przedstawiono wyniki pomiarów.

Tabela 1 Pomiar czasów wykonywania się algorytmów (na szaro czas prognozowany)

|  |  |  |
| --- | --- | --- |
| Liczba miast | Przegląd zupełny | Metoda podziału i ograniczeń |
| [ms] | |
| 4 | 7,4309E-02 | 9,0317E-02 |
| 8 | 3,5143E+02 | 8,7697E-01 |
| 12 | 6,8669E+06 | 3,4433E+00 |
| 16 | 4,1753E+11 | 1,1609E+01 |
| 20 | 6,2223E+16 | 6,9398E+01 |
| 24 | 1,9355E+22 | 1,4127E+02 |
| 28 | 1,1225E+28 | 2,1900E+02 |
| 32 | 1,1166E+34 | 7,1468E+02 |

Wykres 1 Zależność czasu od rozmiaru instancji w metodzie podziału i ograniczeń (skala liniowa)

Wykres 2 Porównanie czasów wykonania się przeglądu zupełnego i metody podziału i ograniczeń (skala logarytmiczna)

Jedynie dla cztero- i ośmiowierzchołkowych grafów wykonywany był przegląd zupełny – pozostałe dane zostały zaprognozowane na podstawie wyników z mniejszych instancji przez wyliczenie stałej *c* przy założeniu złożoności obliczeniowej *O(c\*n!)*.

Na samym początku, przy 4 wierzchołkach grafu widać, że czas dla przeglądu zupełnego jest nieznacznie mniejszy. Jednakże wraz ze wzrostem rozmiaru instancji czas ten rośnie zdecydowanie szybciej, niż w metodzie podziału i ograniczeń.

# Podsumowanie i wnioski

W przypadku pesymistycznym oba algorytmy rozwiążą się w tym samym czasie, gdyż podczas przeszukiwania drzewa będzie trzeba wejść do wszystkich wierzchołków. Jednakże rzadko takowy przypadek się trafia i najczęściej widoczna jest diametralna różnica, taka, jak na wykresie 2.

# Bibliografia

* Maciej Komosiński, „Optymalizacja. Algorytmy dokładne”, <http://www.cs.put.poznan.pl/mkomosinski/materialy/optymalizacja/BB_DP.pdf>
* Mateusz Łyczek, „Metoda podziału i ograniczeń”, <https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf>
* „Rozwiązanie problemu komiwojażera metodą podziału i ograniczeń”, <http://www.gryf-elektryk.pl/sprawozdania/Rozwiazenie-problemu-komiwojazera-metoda-podzialu-i-ograniczen.pdf>
* „Badania operacyjne - ćwiczenia”, http://umcs.net.pl