Bartłomiej Barszczak

WEAIiIB Automatyka i Robotyka

Rok II semestr IV grupa 3

# Zadanie 1

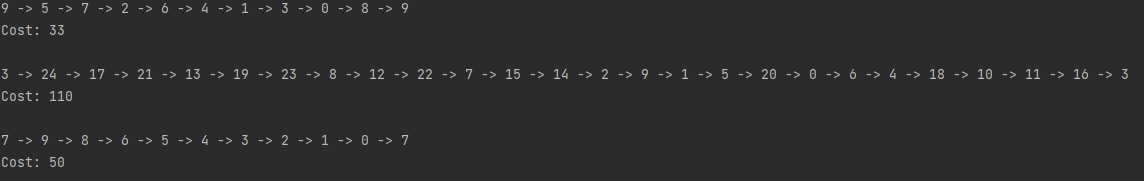
#include <iostream>  
#include <utility>  
#include <vector>  
#include <limits>  
#include <map>  
#include <random>  
  
int INF = std::numeric\_limits<int>::max(); // zmienna symulujaca nieskonczonosc  
  
/\*\*  
\* Struktura reprezentujaca graf w postaci listy sasiedztwa z wagami  
\*/  
struct my\_graph {  
 std::map<int, std::vector<int>> adj\_list;  
 std::map<std::pair<int, int>, int> wages;  
};  
  
/\*\*  
 \* Funkcja tworzy macierz kwadratowa wypelniona jedna wartościa  
 \* @param size Rozmiar macierzy  
 \* @param value Wartość elementu  
 \* @return Nowo utworzona macierz kwadratowa  
 \*/  
std::vector<std::vector<int>> identity\_matrix(int size, int value) {  
 std::vector<std::vector<int>> result = {};  
 std::vector<int> helper = {};  
  
 for (int i = 0; i < size; i++) {  
 helper.clear();  
 for (int j = 0; j < size; j++) {  
 if (i == j)  
 helper.push\_back(INF);  
 else  
 helper.push\_back(value);  
 }  
 result.push\_back(helper);  
 }  
 return result;  
}  
  
/\*\*  
 \* Funckja tworzy macierz symetryczna o podanym rozmiarze  
 \* @param size Rozmiar macierzy  
 \* @return Nowo utworzona macierz symetryczna z INF na diagonali  
 \*/  
std::vector<std::vector<int>> create\_matrix(int size) {  
 std::random\_device dev;  
 std::mt19937 rng(dev());  
 std::uniform\_int\_distribution<std::mt19937::result\_type> random\_value(1, 50);  
 std::vector<std::vector<int>> matrix = {};  
 std::vector<std::vector<int>> result = {};  
 std::vector<int> helper = {};  
  
 for (int i = 0; i < size; i++) {  
 helper.clear();  
 for (int j = 0; j < size; j++) {  
 helper.push\_back((int) random\_value(rng));  
 }  
 matrix.push\_back(helper);  
 }  
  
 for (int i = 0; i < size; i++) {  
 helper.clear();  
 for (int j = 0; j < size; j++) {  
 if (i == j)  
 helper.push\_back(INF);  
 else  
 helper.push\_back((matrix[i][j] + matrix[j][i]) / 5);  
 }  
 result.push\_back(helper);  
 }  
  
 return result;  
}  
  
/\*\*  
 \* Funkcja prezentujaca dane  
 \* @param data Dane do prezentowania w postaci pary wektora i typu calkowitego  
 \*/  
void do\_presentation(const std::pair<std::vector<int>, int> &data) {  
 for (int i = 0; i < data.first.size(); i++) {  
 if (i == data.first.size() - 1)  
 printf("%d", data.first[i]);  
 else  
 printf("%d -> ", data.first[i]);  
 }  
 printf("\nCost: %d\n\n", data.second);  
}  
  
/\*\*  
 \* Funckja zmieniajaca reprezentacje grafu z macierzy sasiedztwa na liste sasiedztwa  
 \* @param matrix Macierz sasiedztwa  
 \* @return Nowo utworzony obiekt typu my\_graph  
 \*/  
my\_graph am2al(const std::vector<std::vector<int>> &matrix) {  
 std::map<std::pair<int, int>, int> wages = {};  
 std::map<int, std::vector<int>> al = {};  
 std::vector<int> helper = {};  
  
 my\_graph result;  
  
 for (int node = 0; node < matrix.size(); node++) {  
 helper.clear();  
 for (int neighbour = 0; neighbour < matrix[node].size(); neighbour++) {  
 if (node != neighbour) {  
 helper.insert(helper.cend(), neighbour);  
 }  
 }  
 al.insert(al.cend(), {node, helper});  
 }  
  
 for (int row = 0; row < matrix.size(); row++) {  
 for (int col = 0; col < matrix[row].size(); col++) {  
 if (matrix[row][col] != INF)  
 wages.insert(wages.cend(), {{row, col}, matrix[row][col]});  
 }  
 }  
 result.adj\_list = al;  
 result.wages = wages;  
  
 return result;  
}  
  
/\*\*  
 \* Funkcja sluzaca do znalezienia najkrotszej sciezki w grafie za pomoca algorytmu NEAREST INSERTION  
 \* @param graph Graf reprezentowny jako lista sasiedztwa  
 \* @param wages Wagi grafu  
 \* @returns Nowo utowrzony obiekt typu pair zawierajacy znaleziona najkrotsza sciezke typu vector oraz koszt drogi typu calkowitego  
 \*/  
std::pair<std::vector<int>, int>  
near\_in(const std::map<int, std::vector<int>> &graph, const std::map<std::pair<int, int>, int> &wages) {  
 // losowanie wierzcholka  
 std::random\_device dev;  
 std::mt19937 rng(dev());  
 std::uniform\_int\_distribution<std::mt19937::result\_type> random\_node(0, graph.size() -  
 1);  
  
 int current = (int) random\_node(rng); // pierwszy losowo wybrany wierzolek grafu  
 std::vector<int> result = **{**current, current**}**; // najkrotsza sciezka  
 std::vector<int> nodes = {}; // wektor wszystkich wierzcholkow  
 int min\_distance; // najmniejsza odleglosc  
 int neighbour; // nowo wybrany wierzolek dla ktorego odleglosc od obecnego jest najmniejsza  
 int total\_cost = 0; // calkowity koszt sciezki  
 std::vector<int> costs = {}; // wektor kosztow przejscia  
  
 nodes.reserve(graph.size()); // dodanie do wektora wierzcholkow wszystkich wierzolkow  
 for (const auto &item: graph) {  
 nodes.push\_back(item.first);  
 }  
 nodes.erase(std::find(nodes.cbegin(), nodes.cend(), current)); // usuniecie pierwszego wierzcholka  
  
 while (!nodes.empty()) { // glowna petla, wykonuje sie dopoki sa jeszcze nieodwiedzone wierzcholki  
 // szukanie najmniejszej odlegolosci  
 min\_distance = INF;  
 for (auto node: nodes) {  
 if (wages.find({current, node})->second < min\_distance) {  
 min\_distance = wages.find({current, node})->second;  
 }  
 }  
 for (auto node: nodes) {  
 if (wages.find({current, node})->second == min\_distance) {  
 neighbour = node;  
 break;  
 }  
 }  
  
 costs.clear(); // czyszczenie wektora kosztow oraz dodanie nowych kosztow  
 costs.reserve(result.size());  
 for (int i = 1; i < result.size(); i++) {  
 costs.push\_back(wages.find({current, neighbour})->second);  
 }  
  
 // wstawianie nowo wybranego wierzcholka w najtanszym miejscu sciezki  
 int min\_cost = INF;  
 for (auto cost: costs)  
 if (cost < min\_cost)  
 min\_cost = cost;  
  
 for (int i = 0; i < costs.size(); i++) {  
 if (costs[i] == min\_cost) {  
 result.insert(std::next(result.cbegin(), i + 1), neighbour);  
 break;  
 }  
 }  
  
 // usuniecie nowego wierzcholka z wektora wierzcholkow, to oznacza ze dany wierzcholek juz odwiedzilismy  
 nodes.erase(std::find(nodes.cbegin(), nodes.cend(), neighbour));  
 total\_cost += wages.find({neighbour, current})->second; // zaktualizowanie calkowitego kosztu  
 current = neighbour;  
  
 }  
 // dodanie ostatniego kosztu przejscia do calkowitego kosztu  
 total\_cost += wages.find({neighbour, result[0]})->second;  
  
 return {result, total\_cost};  
}

# Zadanie 2

Istotne dla algorytmu jest to jaki wierzchołek początkowy zostanie wybrany, ponieważ w zależności od tego jaki wylosujemy uzyskamy różne ścieżki najczęściej z różnymi kosztami. Wagi ujemne nie przeszkadzają w wykonaniu się algorytmu.

Zdefiniowane dane:

int main() {  
 // losowe dane  
 std::vector<std::vector<int>> m1 = **{** {INF, 12, 10, 1, 8, 4, 10, 12, 6, 3},  
 {12, INF, 3, 2, 2, 9, 11, 5, 7, 3},  
 {10, 3, INF, 3, 20, 6, 2, 4, 14, 7},  
 {1, 2, 3, INF, 4, 5, 14, 12, 13, 6},  
 {8, 2, 20, 4, INF, 5, 1, 6, 13, 9},  
 {4, 12, 3, 3, 5, INF, 4, 9, 11, 5},  
 {10, 11, 2, 14, 1, 4, INF, 11, 26, 5},  
 {12, 5, 4, 12, 6, 9, 11, INF, 4, 7},  
 {6, 7, 14, 13, 13, 11, 26, 14, INF, 1},  
 {3, 3, 7, 6, 9, 5, 5, 7, 1, INF}  
 **}**;  
  
 auto g1 = am2al(m1); // zmienienie reprezentacji grafu z macierzy sasiedztwa na liste sasiedztwa  
 auto result1 = near\_in(g1.adj\_list, g1.wages); // wykonanie algorytmu  
 do\_presentation(result1); // prezentowanie wynikow dla losowych danych  
  
 // losowo wygenerowane dane  
 std::vector<std::vector<int>> m2 = create\_matrix(25); // stworzenie macierzy o rozmiarze 25  
 auto g2 = am2al(m2);  
 auto result2 = near\_in(g2.adj\_list, g2.wages);  
 do\_presentation(result2); // prezentowanie wynikow dla losowo wygenerowanych danych  
  
 // macierz "jednostkowa"  
 std::vector<std::vector<int>> m3 = identity\_matrix(10, 5);  
 auto g3 = am2al(m3);  
 auto result3 = near\_in(g3.adj\_list, g3.wages);  
 do\_presentation(result3); // prezentowanie wynikow dla macierzy "jednostkowej"  
  
 return 0;  
}



Rysunek 1 Prezentacja wyników odpowiednio dla pierwszego, drugiego i trzeciego grafu

# Zadanie 3

Złożoność obliczeniowa algorytmu: O(n2)

|  |  |
| --- | --- |
| Różnica między algorytmem NEARIN a pozostałymi | |
| Algorytm najbliższego sąsiada | Algorytm również wybiera najbliższego sąsiada, ale umieszcza go na koniec ścieżki, a nie jak w przypadku algorytmu NEARIN szuka najtańszego wstawienia |
| Algorytm G-TSP | Algorytm najpierw sortuje wagi w niemalejący ciąg i do ścieżki dodaje ten o najmniejszej wadze uważając na to czy nie tworzy się podcykl. |
| Algorytm FARIN | Różnica polega na tym, że jest wybierany nie najbliższy a najdalszy sąsiad |
| Algorytm k-Opt | Algorytm poprawia wyznaczoną wcześniej ścieżkę który zastępuje stare, gorsze krawędzie lepszymi, podczas całego wykonywania działa na początkowej ścieżce a nie na poprawianej. |
| Algorytm Christofidesa | Algorytm najpierw tworzy minimalne drzewo rozpinające, następnie tworzony jest na jego podstawie multigraf w którym wyznaczany jest cykl Eulera z którego wyznaczmy cykl Hamiltona. |