Bartłomiej Barszczak

WEAIiIB Automatyka i Robotyka

Rok II semestr IV grupa 3

# Zadanie 1

#include <iostream>  
#include <vector>  
#include <map>  
#include <limits>  
#include <cmath>  
  
// nieskonczonosc  
int INF = std::numeric\_limits<int>::max();  
  
/\*  
 \* Funkcja liczy i zwraca odlegosc pomiedzy dwoma wierzcholkami korzystajac z normy Euklidesa  
\*/  
int calcualte\_cost\_to\_reach\_goal(std::pair<int, int> a, std::pair<int, int> b) {  
 int result = ((b.first - a.first) \* (b.first - a.first)) + ((b.second - a.second) \* (b.second - a.second));  
 return int(sqrt(result));  
}  
  
/\*  
 \* Funckja wyznacza najkrotsza sciezke z ulozonych wierzcholkow oraz zwraca wynik  
 \*/  
std::vector<int> reconstruct\_path(std::map<int, int> &previous\_nodes, int last\_node, int start\_node) {  
 std::vector<int> result = **{**last\_node**}**; // zainicjalizowanie zmiennej wynikowej wierzcholkiem docelowym  
  
 while (true) { // nieskonczona petla  
 // petla odpowiadajaca za dodanie nowego wierzcholka do kontenera wynikowego  
 for (auto &item: previous\_nodes) {  
 if (item.first == last\_node) { // warunek sprawdzajacy jaki jest poprzedni wirzecholek do obecnego  
 last\_node = previous\_nodes[item.first]; // ustwienie nowego wierzcholka  
 result.insert(result.cbegin(), last\_node); // dodanie nowego wierzcholka do kontenera wynikowego  
 }  
 }  
 if (start\_node == result[0]) // warunek sprawdzajacy czy doszlismy do wierzcholka poczatowego jesli tak to przerywa petle  
 break;  
 }  
 return result; // zwracanie wyniku  
}  
  
/\*  
 \* Funkcja ktora wyznacza najkrotsza sciezke za pomoca algorytmu A\*, zwraca pare: najkrotsza sciezka oraz dlugosc sciezki  
 \*/  
std::pair<std::vector<int>, int> A\_star(const std::map<int, std::vector<int>> &graph, const std::map<std::pair<int, int>, int> &wages,  
 const std::map<int, std::pair<int, int>> &coordinates, int start, int finish) {  
  
 std::vector<int> open\_nodes = **{**start**}**; // wierzcholki zamkniete  
 std::map<int, int> closed\_nodes = {}; // wierzcholki z ktorych przyszlismy  
 std::map<int, int> gscore = {}; // g[u] - aktualny koszt  
 std::map<int, int> fscore = {}; // f[u] = g[u] + h[u] - estymacja osigniecia kosztu z s do k przez u  
 int min\_value; // zmienna pomocnicza do wyznaczenia minimalnej wartosci  
 int new\_node; // zmienna pomocnicza do przewoania nowego wierzcholka  
 int potential\_node; // zmienna pomocnicza do wyznaczenia potencjalnego nowego wierzcholka  
  
 // ustawnienie wartosci wierzcholkow w gscore oraz fscore na nieskonczonosc  
 for (const auto &item: graph) {  
 gscore[item.first] = INF;  
 fscore[item.first] = INF;  
 }  
  
 gscore[start] = 0; // ustawnienie gscore od staru na 0  
 // obliczenie nowego potencjalnego kosztu do przejscia w lini prostej  
 fscore[start] = calcualte\_cost\_to\_reach\_goal(coordinates.find(start)->second,  
 coordinates.find(finish)->second);  
  
 // glowna petla wykonujaca sie dopoki sa wierzolki nieodwiedzone  
 while (!open\_nodes.empty()) {  
 min\_value = INF;  
 // szukanie najmniejszego elementu w fscore i wyznaczenie nowego najlepszego wierzcholka  
 for (auto u: open\_nodes) {  
 if (fscore[u] < min\_value) {  
 min\_value = fscore[u];  
 }  
 }  
 for (auto u: open\_nodes) {  
 if (min\_value == fscore[u]) {  
 new\_node = u;  
 break;  
 }  
 }  
  
 // warunek sprawdzajacy czy nowy najlepszy wierzcholek nie jest wierzcholkiem koncowym  
 if (new\_node == finish) {  
 // zwracanie wyniku  
 return {reconstruct\_path(closed\_nodes, new\_node, start), gscore[new\_node]};  
 }  
  
 // usuwane nowego najlepszego wierzcholka z kontenera wierzcholkow zamknietych (zeby nie byl brany pod uwage)  
 auto breaker = std::remove(open\_nodes.begin(), open\_nodes.end(), new\_node);  
 // warunek sprawdzajacy czy przypadkiem nie chcemy usunac elementu ktorego nie istnieje  
 if (breaker == open\_nodes.cend())  
 return {};  
  
 // petla szukajaca siadiadow nowego najlepszego wierzcholka  
 for (auto v: graph.find(new\_node)->second) {  
 // wyznaczenie kosztu nowego potnecjalnego wierzcholka  
 potential\_node = gscore[new\_node] + wages.find({new\_node, v})->second;  
 // warunek sprawdzajacy czy koszt nowego potencjalnego wierzolka jest mniejszy niz inne  
 if (potential\_node < gscore[v]) {  
 closed\_nodes[v] = new\_node; // ustawienie potencjalnego przejscia  
 gscore[v] = potential\_node; // ustawienie kosztu sasiada nowego najlepszego wierzolka na potencjalny koszt  
 // wyznaczenie nowego calkowitego kosztu  
 fscore[v] = potential\_node + calcualte\_cost\_to\_reach\_goal(coordinates.find(v)->second, coordinates.find(finish)->second);  
 // warunek sprawdzajacy czy dany sasiad nowego najlepszego wierzcholka znajduje sie w wierzolkach zamknietych jesli nie to jest dodawany  
 if (std::find(open\_nodes.cbegin(), open\_nodes.cend(), v) == open\_nodes.cend()) {  
 open\_nodes.push\_back(v);  
 }  
 }  
 }  
 }  
 return {}; // zwracanie pustej pary w przypadku gdy nie znalezlismy drogi  
}  
  
  
}

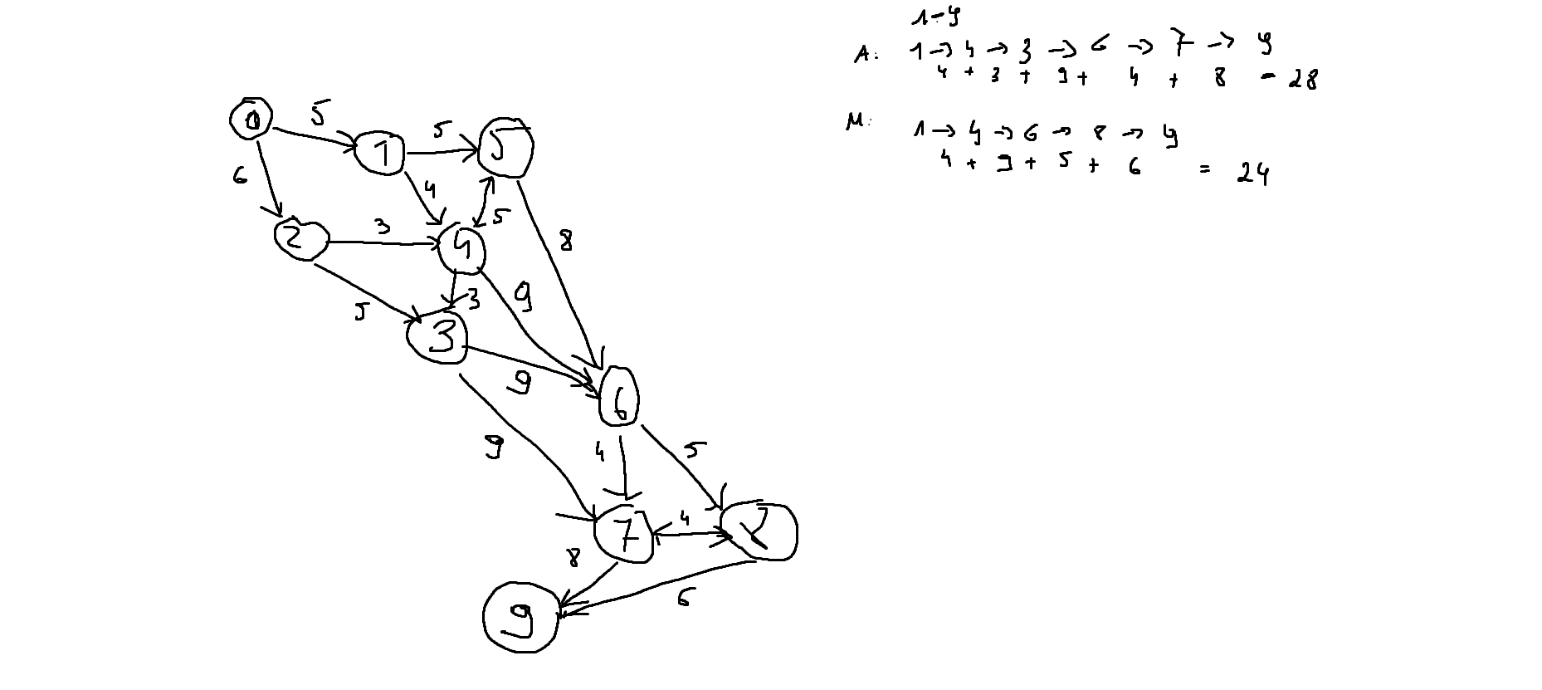
# Zadanie 2

Ważne jest, żeby graf był spójny, może być zarówno nieskierowany jak i skierowany, przy czym w przypadku skierowanego ważne jest, żeby było możliwe przejście z wierzchołka początkowego do wierzchołka końcowego. Ważna jest również funkcja heurystyki.

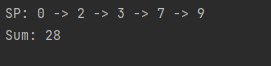
Zdefiniowane dane:

int main() {  
 // graf  
 std::map<int, std::vector<int>> g1 = **{** {0, {1, 2}},  
 {1, {4, 5}},  
 {2, {3, 4}},  
 {3, {6, 7}},  
 {4, {3, 5, 6}},  
 {5, {4, 6}},  
 {6, {7, 8}},  
 {7, {8, 9}},  
 {8, {7, 9}},  
 {9, {}}  
 **}**;  
  
 // funkcja wag  
 std::map<std::pair<int, int>, int> w1 = **{** {{0, 1}, 5}, {{0, 2}, 6},  
 {{1, 4}, 4}, {{1, 5}, 5},  
 {{2, 3}, 5}, {{2, 4}, 3},  
 {{3, 6}, 9}, {{3, 7}, 9},  
 {{4, 3}, 3}, {{4, 5}, 5}, {{4, 6}, 9},  
 {{5, 4}, 5}, {{5, 6}, 8},  
 {{6, 7}, 4}, {{6, 8}, 5},  
 {{7, 8}, 4}, {{7, 9}, 8},  
 {{8, 7}, 4}, {{8, 9}, 6}  
 **}**;  
  
 // wspolrzedne wierzcholkow  
 std::map<int, std::pair<int, int>> c1 = **{** {0, {1, 1}},  
 {1, {4, 2}},  
 {2, {2, 5}},  
 {3, {5, 8}},  
 {4, {5, 5}},  
 {5, {8, 1}},  
 {6, {13, 7}},  
 {7, {13, 11}},  
 {8, {17, 10}},  
 {9, {18, 16}}  
 **}**;  
  
 // wywolanie funkcji wykonujacej algorytm A\*  
 auto result = A\_star(g1, w1, c1, 0, 9);  
  
 // prezentowanie wynikow  
 std::cout << "SP: ";  
 for (int i = 0; i < result.first.size(); i++) {  
 if (i == result.first.size() - 1)  
 std::cout << result.first[i];  
 else  
 std::cout << result.first[i] << " -> ";  
 }  
 std::cout << "\nSum: " << result.second << "\n";  
  
 return 0;  
}

Ilustracja zdefiniowanego grafu:



Prezentacja wyniku:



# Zadanie 3

Złożoność obliczeniowa algorytmu A\* zależy w głównej mierze od funkcji heurystyki. W tym przypadku funkcja heurystyki to linia prosta pomiędzy dwoma wierzchołkami.

Złożoność czasowa algorytmu: O(log(h\*(x)) - gdzie „h\*(x)” to optymalna funkcja heurystyki

Złożoność pamięciowa algorytmu: O(ax) – gdzie „x” to długość rozwiązania, a „a” to współczynnik rozgałęzienia

Optymistyczna złożoność obliczeniowa algorytmu: O(1)

Pesymistyczna złożoność obliczeniowa algorytmu: O(ax)

Średnia złożoność obliczeniowa algorytmu: