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}
    };
```

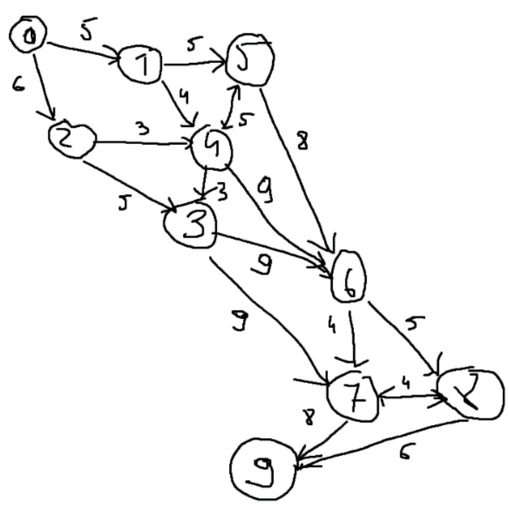
```
    // wspolrzedne wierzcholkow
    std::map<int, std::pair<int, int>> c1 = {
            {0, {1,  1}},
            {1, {4,  2}},
            {2, {2,  5}},
            {3, {5,  8}},
            {4, {5,  5}},
            {5, {8,  1}},
            {6, {13, 7}},
            {7, {13, 11}},
            {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:

```
SP: 0 -> 2 -> 3 -> 7 -> 9
Sum: 28
```

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: $O(a^x)$ – gdzie „x” to długość rozwiązania, a „a” to współczynnik rozgałęzienia

Optymistyczna złożoność obliczeniowa algorytmu: $O(1)$

Pesymistyczna złożoność obliczeniowa algorytmu: $O(a^x)$

Średnia złożoność obliczeniowa algorytmu: