

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Жадный алгоритм и A^*

Студент гр. 1304

Шаврин А.П

Преподаватель

Шевелева А.М.

Санкт-Петербург

2023

Цель работы.

Изучить жадный алгоритм и алгоритм A*. Решить задачу построения пути в ориентированном графе с наименьшим весом ребер, используя жадный алгоритм и алгоритм A*.

Задание.

1. Разработайте программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Пример входных данных:

a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0

В первой строке через пробел указываются начальная и конечная вершины. Далее в каждой строке указываются ребра графа и их вес.

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Для приведённых в примере входных данных ответом будет:

Abcde

1. Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A*. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.

Пример входных данных:

```
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0
```

В первой строке через пробел указываются начальная и конечная вершины. Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Для приведённых в примере входных данных ответом будет:

```
ade
```

Выполнение работы.

Для решения данных задач были написаны следующие блоки кода:

1. Функция `main`, не принимающая входных параметров и реализующая основную логику программы (создание объекта класса `Solver`, который будет описан ниже, вызов метода класса для решения данной задачи (`GreedyAlgorithm` / `AStarAlgorithm`) и вызов метода класса для вывода решения).

2. Был написан класс `Node` – являющийся узлом.

Он имеет в себе следующие приватные поля:

- `char name` - Имя узла

- `std::unordered_map <Node*, int> children` - Словарь дочерних узлов с весами дуг до них
- `std::vector <Node*> visited_children` - Вектор просмотренных дочерних узлов

Он имеет в себе следующие приватные методы:

- `bool isVisitedChild(Node* child_node)` - Метод, возвращающий значение, просмотрен ли уже дочерний узел или нет

Он имеет в себе следующие публичные методы:

- `Node(char name)` - Конструктор класса, реализует установку имени узла
- `char getName()` - Метод, реализующий передачу имени узла
- `void addChild(Node* child_node, int weight)` - Метод, реализующий добавление дочернего узла с весом дуги до него
- `std::unordered_map <Node*, int> getChildren()` - Метод, реализующий передачу словаря дочерних узлов и весов дуг до них
- `Node* getBestChild()` - Метод, возвращающий лучший дочерний узел, для жадного алгоритма

3. После был написан класс `Graph` – являющийся графом, реализующим хранение всех узлов, получение узлов по имени, добавление и т.п.

В классе были объявлены следующие приватные поля:

- `std::vector <Node*> nodes` - Вектор всех узлов графа

В классе были объявлены следующие приватные методы:

- `bool isPresentNode(char node_name)` - Метод, реализующий проверку наличия в графе узла по его имени

В классе были объявлены следующие публичные методы:

- Graph() - Конструктор класса
- Node* getNode(char node_name) - Метод, реализующий доступ к узлу по его имени
- void addBranch(char first_node_name, char second_node_name, int weight) - Метод, реализующий добавление в граф узлов и весов дуг между ними
- ~Graph() - Деструктор класса

4. Затем был написан класс Solver – реализующий считывание входных данных и решение задачи, обоими алгоритмами.

Класс имеет в себе следующие приватные поля:

- Graph* graph - Граф вершин и ребер
- Node* start_node - Стартовый узел
- Node* end_node - Конечный узел
- std::vector <Node*> path - Вектор вершин, составляющих путь от стартовой вершины, до конца
- std::unordered_map<Node*, Node*> came_from_dict - Словарь, где ключ - вершина, значение - вершина откуда в нее пришли
- std::unordered_map<Node*, int> path_cost_dict - Словарь, где ключ - вершина, значение - оценка сложности пути
- typedef std::pair<int, Node*> PQElement – определение типа для очереди с приоритетом
- std::priority_queue<PQElement, std::vector<PQElement>, std::greater<PQElement>> priority_queue - Очередь с приоритетом из стандартной библиотеки

Класс имеет в себе следующие приватные методы:

- inline int getHeuristic(char node_name, char end_node_name) - Метод, реализующий вычисление эвристической оценки

- void reconstructPath() - Метод, реализующий восстановление пути для алгоритма A*

Класс имеет в себе следующие публичные методы:

- Solver() - Конструктор класса
- void GreedyAlgorithm() - Метод, реализующий решение задачи жадным алгоритмом (использует GreedyAlgorithmR)
- void AStarAlgorithm() - Метод, реализующий решение задачи алгоритмом A*
- void printPath() - Метод, реализующий вывод ответа (пути)
- ~Solver() - Деструктор класса

5. Решение задачи жадным алгоритмом работает следующим образом. В конструкторе класса Solver происходит заполнение графа. В методе GreedyAlgorithm во временную переменную Node* best_node сначала помещается стартовый узел, от которого нужно строить путь, он же добавляется в вектор path. Затем пока этот best_node не станет конечным узлом происходит следующее. Сперва переменная best_node обновляется на новое значение равное лучшему дочернему узлу от текущего best_node. Далее идет проверка на то, что такой узел нашелся или же такого узла уже нет в пути. Если эта проверка не проходит, то происходит откат к предыдущему лучшему узлу. Если проверка пройдена, то узел добавляется в вектор path.

6. Решение задачи алгоритмом A* работает следующим образом. В конструкторе класса Solver происходит заполнение графа. В очередь с приоритетом 0 помещается стартовый узел, в словаре came_from_dict ему устанавливается значение узла, из которого в него пришли - сам стартовый узел. Далее идет цикл по всем вершинам в очереди. Во временную переменную Node* current_node извлекается самый приоритетный элемент очереди. Потом происходит проверка, является ли он конечным, если да – функция заканчивает

свою работу. Если нет, то идет цикл по всем дочерним узлам текущего, рассчитывается стоимость пути до каждого из них и если это значение меньше, чем было до или ранее до данной вершины не было пути, то в словарь `path_cost_dict` записывается новая стоимость пути до этой вершины и она добавляется в очередь с приоритетом равным сумме стоимости пути и эвристической оценке. Также для этой вершины в словаре `came_from_dict` указывается вершина, через которую был улучшен путь. После того как алгоритм отработал в нем вызывается метод `reconstructPath`, который по словарю `came_from_dict` восстанавливает путь до старта.

Разработанный программный код см. в приложении А.

Результаты тестирования см. в разделе Тестирование.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde	Тест для жадного алгоритма
2.	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade	Тест для алгоритма A*

Выводы.

Изучена работа жадного алгоритма и алгоритма A*. Решена задача построения пути в ориентированном графе с наименьшим весом ребер, используя жадный алгоритм и алгоритм A*. Работа прошла как собственные тесты, так и тесты на платформе Stepic. Для решения задачи было решено использовать ООП и оба алгоритма были реализованы как методы одного класса. При написании решающая роль была отведена такому типу данных как словарь и абстрактная структура данных очередь с приоритетом. В качестве эвристической функции для алгоритма A* была использована разность ASCII символов, обозначающих имя конечного и текущего узла.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: GAandAStar.cpp

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <algorithm>
#include <queue>

/*
    Класс Node реализует хранение своего имени, а также узлов, в
    которые из него можно попасть и веса дуг до этих узлов
*/
class Node{
private:
    char name; // Имя узла
    std::unordered_map <Node*, int> children; // Словарь
дочерних узлов с весами дуг до них
    std::vector <Node*> visited_children; // Вектор
просмотренных дочерних узлов

    bool isVisitedChild(Node* child_node); // Метод,
возвращающий значение, просмотрен ли уже дочерний узел или нет

public:
    Node(char name); //
Конструктор класса, реализует установку имени узла
    char getName(); // Метод,
реализующий передачу имени узла
    void addChild(Node* child_node, int weight); // Метод,
реализующий добавление дочернего узла с весом дуги до него
    std::unordered_map <Node*, int> getChildren(); // Метод,
реализующий передачу словаря дочерних узлов и весов дуг до них
    Node* getBestChild(); // Метод,
возвращающий лучший дочерний узел, для жадного алгоритма
};

/*
    Конструктор класса, устанавливающий имя узла
    Args:
        - name (char) - имя узла
*/
Node::Node(char name){
    this->name = name;
};

/*
    Метод, возвращающий значение, просмотрен ли уже дочерний узел
    или нет
*/
bool Node::isVisitedChild(Node* child_node){
    return std::binary_search(this->visited_children.begin(), this-
>visited_children.end(), child_node);
};
```

```

    }

    /*
        Метод класса, возвращающий имя узла
        Returns:
            - name (char) - имя узла
    */
    char Node::getName() {
        return this->name;
    };

    /*
        Метод, реализующий добавление дочернего узла и веса дуги до
        него в словарь
        Args:
            - child_node (Node*) - дочерний узел
            - weight (int) - вес дуги до дочернего узла
    */
    void Node::addChild(Node* child_node, int weight) {
        this->children.insert(std::pair<Node*, int>(child_node,
weight));
    };

    /*
        Метод, реализующий возвращение словаря дочерних узлов с весами
        дуг до них
        Returns:
            - children (std::unordered_map<Node*, int>) - словарь
        дочерних узлов с весами дуг до них
    */
    std::unordered_map<Node*, int> Node::getChildren() {
        return this->children;
    };

    /*
        Метод, возвращающий лучший дочерний узел, для жадного алгоритма
    */
    Node* Node::getBestChild() {
        int min_weight = 10000;
        Node* best_child = nullptr;

        for (auto i = this->children.begin(); i != this->children.end();
++i) {
            if (this->isVisitedChild(i->first) == false) {
                if (i->second < min_weight || (i->second == min_weight
&& (int)i->first->getName() < (int)best_child->getName())) {
                    min_weight = i->second;
                    best_child = i->first;
                }
            }
        }

        if (best_child != nullptr) {
            this->visited_children.emplace_back(best_child);
        }
        return best_child;
    };

```

```

/*
    Класс Graph реализует хранение всех узлов графа
*/
class Graph{
private:
    std::vector <Node*> nodes;           // Вектор всех узлов
графа

    bool isPresentNode(char node_name); // Метод, реализующий
проверку наличия в графе узла по его имени

public:
    Graph();
// Конструктор класса
    Node* getNode(char node_name);
// Метод, реализующий доступ к узлу по его имени
    void addBranch(char first_node_name, char second_node_name,
int weight); // Метод, реализующий добавление в граф узлов и весов дуг
между ними
    ~Graph();
// Деструктор класса
};

/*
    Конструктор класса, реализующий предварительное выделение
памяти под вектор узлов
*/
Graph::Graph(){
    this->nodes.reserve(10);
};

/*
    Метод, реализующий проверку наличия в графе узла по его имени
Args:
    - node_name (char) - имя искомого узла
Returns:
    - is_present (bool) - есть или нет узел в графе
*/
bool Graph::isPresentNode(char node_name){
    bool is_present = false;
    for (Node* i_node : this->nodes){
        if (node_name == i_node->getName()){
            is_present = true;
            break;
        }
    }
    return is_present;
};

/*
    Метод, возвращающий узел графа, с соответствующим именем
Args:
    - node_name (char) - имя узла
Returns:
    - node (Node*) - объект узла

```

```

*/
Node* Graph::getNode(char node_name){
    Node* node = nullptr;
    for (Node* i_node : this->nodes){
        if (node_name == i_node->getName()){
            node = i_node;
            break;
        }
    }
    return node;
};

/*
Метод, реализующий добавление в граф узлов и весов дуг между
НИМИ
Args:
    - first_node_name (char) - узел, из которого есть дуга
    - second_node_name (char) - узел в который есть дуга
    - weight (int) - вес дуги
*/
void Graph::addBranch(char first_node_name, char second_node_name,
int weight){
    Node* first_node = nullptr;
    Node* second_node = nullptr;

    // Проверка наличия первого узла в графе
    if (!this->isPresentNode(first_node_name)){
        this->nodes.emplace_back(new Node(first_node_name));
    }
    first_node = this->getNode(first_node_name);

    // Проверка наличия второго узла в графе
    if (!this->isPresentNode(second_node_name)){
        this->nodes.emplace_back(new Node(second_node_name));
    }
    second_node = this->getNode(second_node_name);

    // Добавление дочернего узла первому с весом дуги
    first_node->addChild(second_node, weight);
};

/*
Деструктор класса, реализующий удаление всех узлов из памяти
*/
Graph::~~Graph(){
    for (Node* node : this->nodes){
        delete node;
    }
};

/*
Класс Solver реализует решение исходной задачи и хранение всех
необходимых для этого полей и методов
*/
class Solver{

```

```

private:
    Graph* graph;           // Граф вершин и ребер
    Node* start_node;       // Стартовый узел
    Node* end_node;         // Конечный узел
    std::vector<Node*> path; // Вектор вершин, составляющих
    путь от стартовой вершины, до конца

    std::unordered_map<Node*, Node*> came_from_dict; //
Словарь, где ключ - вершина, значение - вершина откуда в нее пришли
    std::unordered_map<Node*, int> path_cost_dict; //
Словарь, где ключ - вершина, значение - оценка сложности пути

    typedef std::pair<int, Node*> PQElement;
    std::priority_queue<PQElement, std::vector<PQElement>,
std::greater<PQElement>> priority_queue; // Очередь с приоритетом из
стандартной библиотеки

    inline int getHeuristic(char node_name, char end_node_name);
// Метод, реализующий вычисление эвристической оценки
    void reconstructPath();
// Метод, реализующий восстановление пути для алгоритма A*

public:
    Solver(); // Конструктор класса
    void GreedyAlgorithm(); // Метод, реализующий решение
задачи жадным алгоритмом (использует GreedyAlgorithmR)
    void AStarAlgorithm(); // Метод, реализующий решение
задачи алгоритмом A*
    void printPath(); // Метод, реализующий вывод ответа
(пути)
    ~Solver(); // Деструктор класса
};

/*
    Метод, реализующий вычисление эвристической оценки
    Args:
        - node_name (char) - имя узла, эвристика которого
расчитывается
        - end_node_name (char) - имя узла, до которого
расчитывается эвристика
    Returns:
        - heuristic (int) - эвристическая оценка
*/
inline int Solver::getHeuristic(char node_name, char
end_node_name){
    int heuristic = std::abs((int)node_name - (int)end_node_name);
    return heuristic;
}

/*
    Конструктор класса, реализующий считывание входных данных,
создание и заполнение начальными значениями полей класса
*/
Solver::Solver(){
    // Считывание имен начального и конечного узла
    char start_node_name;
    char end_node_name;
    std::cin >> start_node_name >> end_node_name;

```

```

        // Создание графа
        this->graph = new Graph();

        // Заполнение графа
        char first_node_name;
        char second_node_name;
        float weight;
        while (std::cin >> first_node_name >> second_node_name >>
weight){
            this->graph->addBranch(first_node_name, second_node_name,
weight);
        }

        // Запоминаем начальный и конечный узел
        this->start_node = this->graph->getNode(start_node_name);
        this->end_node = this->graph->getNode(end_node_name);
    };

    /*
        Метод, реализующий решение задачи жадным алгоритмом
    */
    void Solver::GreedyAlgorithm() {
        Node* best_node = this->start_node;
        this->path.emplace_back(best_node);

        while (best_node != this->end_node) {
            best_node = best_node->getBestChild();
            if (best_node == nullptr || std::binary_search(this-
>path.begin(), this->path.end(), best_node)) {
                if (best_node == nullptr) {
                    this->path.pop_back();
                }
                best_node = this->path.back();
                this->path.pop_back();
            }
            this->path.emplace_back(best_node);
        }
    }

    /*
        Метод, реализующий решение задачи алгоритмом A*
    */
    void Solver::AStarAlgorithm() {
        this->priority_queue.emplace(0, this->start_node); //
Помещаем стартовую вершину в очередь с приоритетом 0
        this->came_from_dict[this->start_node] = this->start_node; //
Указываем, что пришли в эту вершину из нее самой
        this->path_cost_dict[this->start_node] = 0; //
Указываем оценку сложности пути 0

        // Цикл по всем вершинам в очереди
        while (!priority_queue.empty()) {
            Node* current_node = this->priority_queue.top().second;
            // Извлекаем самый приоритетный узел
            this->priority_queue.pop();
            // Удаляем самый приоритетный узел из очереди

```

```

        // Проверка конца алгоритма
        if (current_node == this->end_node) {
            break;
        }

        // Цикл по всем дочерним узлам текущего
        std::unordered_map <Node*, int> children = current_node-
>getChildren();
        for (auto i = children.begin(); i != children.end(); ++i) {
            Node* next_node = i->first;
            int branch_weight = i->second;
            int new_cost = this->path_cost_dict[current_node] +
branch_weight;    // Оценка сложности пути до новой вершины

            // Проверка возможности улучшить путь до этой вершины
            if (this->path_cost_dict.find(next_node) == this-
>path_cost_dict.end() || new_cost < this->path_cost_dict[next_node]){
                // Улучшение сложности пути до вершины
                this->path_cost_dict[next_node] = new_cost;

                // Помещение вершины в очередь с новым приоритетом
                int priority = new_cost + this-
>getHeuristic(next_node->getName(), this->end_node->getName());
                this->priority_queue.emplace(priority, next_node);

                // Помечаем откуда улучшили путь в вершину
                this->came_from_dict[next_node] = current_node;
            }
        }
        this->reconstructPath();
    }

    /*
    Метод, реализующий восстановление пути
    */
    void Solver::reconstructPath() {
        Node* current = this->end_node;
        // Проверка отсутствия пути
        if (this->came_from_dict.find(this->end_node) == this-
>came_from_dict.end()) {
            return;
        }

        // Восстановление пути
        while (current != this->start_node) {
            this->path.emplace_back(current);
            current = came_from_dict[current];
        }
        this->path.push_back(this->start_node);

        // Реверс пути
        std::reverse(this->path.begin(), this->path.end());
    };

    /*
    Метод, реализующий вывод пути
    */

```



```

void Solver::printPath(){
    for (Node* node : path){
        std::cout << node->getName();
    }
    std::cout << std::endl;
};

/*
    Деструктор класса, очищающий память
*/
Solver::~Solver(){
    delete this->graph;
}

/*
    Главная функция, реализующая решение задачи
*/
int main(){
    Solver solver = Solver();
    solver.GreedyAlgorithm();
    solver.printPath();
    return 0;
};

```