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: ПОИСК ПУТИ В ГРАФЕ

Студент гр. 9382 _____ Кодуков А.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы:

Изучить и использовать на практике алгоритмы поиска пути в графк

Задание:

Жадный алгоритм:

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

Алгоритм A*:

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

Вар. 2:

В A* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных

Описание алгоритма:

Жадный алгоритм:

На каждой итерации выбирается последняя посещенная вершина, рассматриваются все смежные ей вершины, из которых выбирается не посещённая ранее с наименьшим весом ребра, затем алгоритм повторяется для

нее. Текущий путь хранится в стеке и, при невозможности пройти далее из рассматриваемой вершины, достается последняя вершина из стека. Алгоритм заканчивает работу при переходе к финишной вершине.

Алгоритм A*:

Данный алгоритм основан на поиске в ширину с использованием эвристик вершин. Каждая вершина добавляет в очередь все смежные ей, а очередь сортируется по значению суммы стоимости пути до этой вершины и модуля разности вершины и финиша. Таким образом, следующей рассматривается вершина из очереди с наименьшим значением данной суммы. Алгоритм прекращает работу при рассмотрении финишной вершины.

Функции и структуры данных:

Структуры данных:

`typedef std::pair<type, double> path_to` – вершина и число (длина ребра для жадного поиска и путь + эвристика для A*)

`typedef std::priority_queue<path_to, std::vector<path_to>, comparator<type, double>> edges_end` – очередь из вершин, отсортированная по соответствующим алгоритму величинам

`typedef std::map<type, edges_end> edges_type` – ребра

`typedef std::vector<type> path_stack` – стек для текущего пути

`std::set<type> visited` – посещенные вершины

Реализованные функции:

Инициализация графа

Сигнатура: `path_finder(std::istream &input, bool input_mode)`

Аргументы:

- input – поток ввода

Алгоритм:

- Считать начало и конец искомого пути
- Считать эвристики вершин
- Считать ребра

Вывод ребер, исходящих из одной вершины

Сигнатура: `void print_edges_vert(const edges_end &q, const type &vert)`

Аргументы:

- q – очередь смежных вершин
- vert – первая вершина ребра

Вывод всех ребер графа

Сигнатура: `void print_all_edges(const edges_type &e)`

Аргументы:

- e – ребра

*Вывод фронта для алгоритма A**

Сигнатура: `void print_frontier(const edges_end &f)`

Аргументы:

- f – фронт

Вывод всего графа

Сигнатура: `void print_graph()`

Жадный поиск

Сигнатура: `void greedy_find()`

Алгоритм:

- Инициализировать текущую вершину стартовой вершиной
- Пока путь не найден (алгоритм перешел на финишную вершину)
    - Рассмотреть все ребра, исходящие из текущей вершины
    - Если все ребра посещены, вернуться назад по стеку
    - Иначе перейти по самому короткому не посещенному ребру, пометив прошлую вершину как посещенную

*Алгоритм A**

Сигнатура: `void astar_find()`

Алгоритм:

- Инициализировать текущую вершину стартовой вершиной
- Пока путь не найден (алгоритм перешел на финишную вершину)
    - Рассмотреть все ребра, исходящие из текущей вершины
    - Если путь до каких-то из смежных вершин оказался короче, чем записанный ранее, обновить его

- Рассчитать для каждой обновленной вершины сумму текущего пути до нее и эвристики в этой вершине и добавить их во фронт, отсортированный по данной сумме
- Перейти на вершину с наименьшей суммой из фронта

Тестирование:

№	Входные данные	Жадный поиск	Алгоритм A*
1	a g a 6 b 5 c 4 d 3 e 2 f 1 g 0 + a b 3 a c 1 b d 2 b e 3 d e 4 e a 3 e f 2 a g 8 f g 1 +	abdefg	ag
2	a e a 4 b 3 c 2 d 1 e 0 + a b 7 a c 3 b c 1 c d 8 b e 4 +	abe	abe
3	a g a 6 b 5 c 4 d 3 e 2 f 1 g 0 m 6 n 7 + a b 3 a c 1 b d 2 b e 3 d e 4	abdefg	ag

	e a 3 e f 2 a g 8 f g 1 c m 1 m n 1 +		
4	a d a 3 b 2 c 1 d 0 e 1 + a b 1 b c 9 c d 3 a d 9 a e 1 e d 3 +	aed	aed
5	a d a 3 b 2 c 1 d 0 + a b 1 b c 1 c a 1 a d 8 +	ad	ad
6	a d a 3 b 2 c 1 d 0 e 1 + a b 1 b c 9 c d 3 a d 9 a e 1 e d 3 +	aed	aed
7	a f a 5 b 4 c 3 d 2 e 1	acdf	acdf

	f 0 + a c 1 a b 1 c d 2 b e 2 d f 3 e f 3 +		
8	b e a 4 b 3 c 2 d 1 e 0 g 2 + a b 1 a c 2 b d 7 b e 8 a g 2 b g 6 c e 4 d e 4 g e 1 +	Bge	bge

Вывод:

В результате выполнения работы были изучены и реализованы алгоритмы жадного поиска и A*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <sstream>
#include <string>
#include <vector>

#define FILE_MODE 0

typedef char type;

typedef std::pair<type, double> path_to;
template <typename K, typename V>
struct comparator {
  bool operator()(const path_to &lhs, const path_to &rhs) {
    return lhs.second == rhs.second ? lhs.first < rhs.first
                                    : lhs.second > rhs.second;
  }
};

typedef std::priority_queue<path_to, std::vector<path_to>,
                            comparator<type, double>> edges_end;
typedef std::map<type, edges_end> edges_type;
typedef std::vector<type> path_stack;

class path_finder {
 private:
  edges_type edges;
  std::map<type, double> heuristic;
  type start, end;
  std::set<type> visited;

 public:
  path_finder(std::istream &input, bool input_mode) {  // 0 - handle, 1 - file
    if (input_mode == 0) {
      std::cout
          << "Input two vertices to find a path between (<start> <finish>):\n";
    }
    input >> start >> end;
    if (input_mode == 0)
      std::cout << "Input heuristics for vertices (<name> <heuristic>), Enter "
                   "to stop:\n";

    type elem;
    double value;

    while (input >> elem && elem != '+') {
      input >> value;
      heuristic[elem] = value;
    }
    if (input_mode == 0)
      std::cout << "Input edges (<first> <second> <weight>), Enter to stop:\n";
    type first, second;
    double weight;
    while (input >> first && first != '+') {
      input >> second >> weight;

      if (edges.find(first) == edges.end())
```

```
      edges.emplace(std::make_pair(first, edges_end()));
    edges[first].push(std::make_pair(second, weight));
  }
}

void print_edges_vert(const edges_end &q, const type &vert) {
  edges_end tmp = q;
  std::vector<path_to> arr;
  while (!tmp.empty()) {
    arr.push_back(tmp.top());
    tmp.pop();
  }
  for (auto &el : arr)
    std::cout << vert << "-" << el.first << ": " << el.second << "\n";
}

void print_frontier(const edges_end &f) {
  edges_end tmp = f;
  std::vector<path_to> arr;
  while (!tmp.empty()) {
    arr.push_back(tmp.top());
    tmp.pop();
  }
  for (auto &el : arr) std::cout << el.first << ": " << el.second << " ";
  std::cout << "\n";
}

void print_all_edges(const edges_type &e) {
  for (auto &el : e) {
    print_edges_vert(el.second, el.first);
  }
}

void print_graph() {
  for (int i = 0; i < 15; i++) std::cout << "*";
  std::cout << "\n";
  std::cout << "Graph:\n";
  std::cout << "Vertices and heuristics:\n";
  for (auto &h : heuristic) std::cout << h.first << ": " << h.second << " ";
  std::cout << "\nStart: " << start << "\nEnd :" << end;
  std::cout << "\nEdges:\n";
  print_all_edges(edges);
  for (int i = 0; i < 15; i++) std::cout << "*";
  std::cout << "\n";
}

void greedy_find() {
  bool path_found = false;
  type cur = start;
  edges_end cur_pathes;
  path_stack stack;

  std::cout << "\n\nStarting greedy search...\n\n";

  visited.clear();
  while (!path_found) {
    std::cout << "Current vertex: " << cur << "\n";
    bool no_way =
        false;  // true - impossible to go furter from current vertex
    path_to res;
    auto edge_iter = edges.find(cur);
    std::cout << "Current edges: \n";
    if (edge_iter != edges.end()) {
      cur_pathes = edge_iter->second;
      print_edges_vert(cur_pathes, cur);
    } else {
      cur_pathes = edges_end();
```

```
        std::cout << "No edges\n";
      }
      auto iter_visited = visited.end();
      do {
        iter_visited = visited.end();
        if (!cur_pathes.empty()) {
          res = cur_pathes.top();  // possible next vertex in path and weight of
                                   // its edge
          std::cout << "Checking path " << cur << "-" << res.first << "\n";
          cur_pathes.pop();
          if ((iter_visited = visited.find(res.first)) != visited.end())
            std::cout << "  It was visited earlier\n";
          else
            std::cout << "  It wasn't visited earlier, go there\n";
        }
        else {
          no_way = true;
          break;
        }
        if (iter_visited == visited.end() && res.first == '\0') {
          no_way = true;
        }
      } while (iter_visited != visited.end());  // continue if this vertex was
visited earlier
      visited.emplace(cur);     // visit vertex
      if (cur == end) {         // path found
        std::cout << "Current vertex is finish, path was found!\n\n";
        path_found = true;
        stack.push_back(cur);
        continue;
      }
      if (no_way) {  // go back
        std::cout << "  Cant go further from this vertex, return to previous\n";
        cur = stack.back();
        stack.pop_back();
        continue;
      }
      stack.push_back(cur);
      std::cout << "Current stack: ";
      for (auto &ch : stack)  // print path
        std::cout << ch << " ";
      std::cout << "\n";
      cur = res.first;
    }
    std::cout << "Path: ";
    for (auto &ch : stack)  // print path
      std::cout << ch;
    std::cout << "\n";
  }

  void astar_find() {
    bool path_found = false;
    type cur = start;
    edges_end cur_pathes;

    visited.clear();
    edges_end frontier;              // unvisited vertices with heuristic
    std::map<type, double> cost_to;  // cost of path to vertices
    std::map<type, type>
        came_from;  // key - vertex, value - previous vertex on path

    std::cout << "\n\nStarting A* search...\n\n";

    int cur_length = 0;
    while (!path_found) {
      std::cout << "Current vertex: " << cur << "\n";
      std::cout << "Current edges: \n";
```

```
      if (edges.find(cur) != edges.end()) {
        cur_pathes = edges.find(cur)->second;
        print_edges_vert(cur_pathes, cur);
      } else {
        std::cout << "No edges\n";
      }
      auto iter_visited = visited.end();
      int num = (int)cur_pathes.size();
      // add all unvisited neighbours to frontier
      for (size_t i = 0; i < num; i++) {
        // get one neighbour
        path_to vert = cur_pathes.top();
        cur_pathes.pop();
        std::cout << "Check neighbour " << vert.first << "\n";
        // compute length of path to neighbour
        double new_cost = cost_to[cur] + vert.second;
        if (cost_to.find(vert.first) == cost_to.end() ||
            new_cost < cost_to[vert.first]) {
          // compute heuristic for neighbour
          std::cout << "  Vertex wasnt visited or new path is better\n";
          std::cout << "  New path to it: " << new_cost
                    << ", Heuristic: " << heuristic[vert.first] << "\n";
          cost_to[vert.first] = new_cost;
          frontier.emplace(
              std::make_pair(vert.first, new_cost + heuristic[vert.first]));
          came_from[vert.first] = cur;
        } else {
          std::cout
              << "  Current path to this vertex is better, no changes needed\n";
        }
      }
      std::cout << "Current frontier (<vertex> - <path + heuristic>):\n  ";
      print_frontier(frontier);
      std::cout << "Current pathes:\n  ";
      for (auto &p : cost_to) std::cout << p.first << ": " << p.second << " ";
      std::cout << "\n";
      cur = frontier.top().first;
      frontier.pop();
      if (cur == end) {
        std::cout << "Current vertex is finish, path was found!\n\n";
        path_found = true;
      }
      if (frontier.empty() && !path_found) {
        std::cout << "No path\n";
        break;
      }
    }
    std::cout << "Path: ";
    type tracker = end;
    path_stack path;
    while (tracker != start) {
      path.push_back(tracker);
      tracker = came_from[tracker];
    }
    path.push_back(start);
    std::reverse(path.begin(), path.end());
    for (auto &v : path) std::cout << v;
    std::cout << "\n";
  }
};

int main() {
  bool mode = 1, finish = false;
  std::string fn;
  std::ifstream file;

  if (FILE_MODE) {
```

```
    std::cin >> fn;
    file.open(fn);
    if (!file.is_open()) {
      std::cout << "Impossible to open file!\n";
      return 0;
    }
  }
  path_finder pf((FILE_MODE ? file : std::cin), mode);
  if (file.is_open()) file.close();
  pf.print_graph();
  pf.greedy_find();
  pf.astar_find();
  return 0;
}
```