


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 группы 1304

Преподаватель



Завражин Д.Г.

Шевелева А.М.

Санкт-Петербург
2023

Цель работы

Изучить и на практике освоить азы применения нахождения кратчайших путей посредством жадного алгоритма и алгоритма A*.

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.

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.

Задание (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.

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.

1 Используемые функции и структуры данных

Программа написана на парадигме объектно-ориентированного программирования. Основная логика заключена в классе *TripPlanner*, конструктор которого инициализирует все необходимые для поиска решения переменные, а сам поиск начинается посредством вызова метода *plan()*.

В классе *TripPlanner* имеется вспомогательный вложенный класс *Map*, реализующий интерфейс работы с графом с взвешенными дугами. В нём есть вло-

женная структура *Route*, используемая в нём для представления рёбра. У класса *Map* имеются следующие методы:

- *unsigned is_mapped(Route route) const*; – принимает возможную дугу и возвращает, есть ли оно в графе;
- *unsigned weight(Route route) const*; – принимает дугу и возвращает сопоставляемый ему вес;
- *unsigned &weight(Route route)*; – принимает возможную дугу и возвращает ссылку на сопоставляемый ей вес, также создавая ей при отсутствии таковой.

Как видно, все они являются вспомогательными.

В свою очередь, класс *TripPlanner* содержит следующие методы:

- *unsigned heuristic_distance(Map::Location location)*; – Принимает узел графа и возвращает применяющуюся в алгоритме A^* эвристическую оценку расстояние от него до представляющегося пунктом назначения узла;
- *greedy_planner(Map::Route route, bool reset = true)* – Принимает содержащую начальный и конечный узлы структуру и обозначающий рекурсивность вызова флаг и затем, опираясь на жадный подход, рекурсивно строит путь из первого узла ко второму, используя поиск с возвратом при необходимости;
- *bool a_star_planner()*; – не имеет входных параметров, осуществляет построение пути посредством алгоритма A^* . Построение выполняется в два этапа: сначала осуществляется обход графа при помощи очереди с приоритетом с сохранением необходимых для восстановления пути пометок, затем на втором этапе путь по пометкам восстанавливается в обратном порядке.
- *TripPlanner &plan(bool use_a_star)*; – Принимает на вход используемый для выбора алгоритма флаг, делегирует на его основании построение пути одному из двух применимых для этого методов, затем возвращает ссылку на хранящий построенное решение объект класса *TripPlanner*.

Полный исходный код программы представлен в Листинге 1 в Приложении А.

Выводы

Были изучены и на практике освоены азы применения нахождения кратчайших путей посредством жадного алгоритма и алгоритма A^* . При разработке жадного подхода к решению задачи были также отточены навыки реализации перебора с возвратом.

При не включённом в отчёт самостоятельном тестировании в процессе отладки разрабатываемого кода было отмечено, что жадный подход к данной задаче в весомом количестве случаев не подходит для нахождения глобально оптимального решения, из чего следует неоптимальность его применения к такому классу задач. Напротив, алгоритм A^* в рассмотренных случаях с выбранной эвристикой находит более оптимальные решения, из чего следует большая целесообразность его применения и адекватность заданной эвристической функции.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Листинг 1 — Содержащийся в файле *main.cpp* исходный код

```
1  #include <algorithm>
2  #include <iostream>
3  #include <map>
4  #include <queue>
5  #include <vector>
6
7  // 'TripPlanner' is a class that encapsulates the logic of finding a shortest
8  // path between two points given a certain map, all provided through stdin
9  class TripPlanner
10 {
11 private:
12     // An auxiliary class that represents a graph to be traversed
13     class Map
14     {
15 public:
16         // A location is simply represented as one char, as the problem suggests
17         using Location = char;
18
19         // An auxiliary struct to hold a pair of locations
20         // It is used to describe both a single graph edge and a path sought for
21         struct Route
22         { Location source;
23           Location destination;
24
25           // The comparison op. is necessary for Route to be used as a map key
26           bool operator<(const Route &route) const
27           { return this->source < route.source or
28             this->source == route.source and this->destination < route.destination;
29           };
30
31           static constexpr unsigned MAX_WEIGHT = -1; // the underflow is intentional
32
33 private:
34         std::map<Route, unsigned> routes; // holds weights of graph edges
35
36 public:
37         // 'is_mapped' tests for whether an edge is present in the graph
38         unsigned is_mapped(Route route) const
39         { return this->routes.count(route) != 0; }
40
41         // 'weight' retrieves a weight corresponding to a given edge
42         // It is also used to introduce a new one
43         unsigned weight(Route route) const
44         { return this->routes.at(route); }
45
46         unsigned &weight(Route route)
47         { return this->routes[route]; }
48     };
49
50     Map map;
51     static constexpr Map::Location FIRST_LOCATION_ID = 'a';
52     Map::Location last_location_id = FIRST_LOCATION_ID;
53     std::vector<Map::Location> trip_plan; // hopefully gets to contain a solution
54     Map::Route trip_route;
55
56     // The heuristic function for the A* method
57     unsigned heuristic_distance(Map::Location location)
58     { return this->trip_route.destination - location; }
59
60     // 'greedy_planner' recursively finds a solution using a greedy approach
61     bool greedy_planner(Map::Route route, bool reset = true)
62     { this->trip_plan.push_back(route.source);
```

```

63     if(route.source == route.destination)
64     { return true; }
65     // 'discarded_routes' holds routes which can no longer be traversed
66     static Map discarded_routes;
67     if(reset)
68     { discarded_routes = Map(); }
69     // Recursively build a path, backtrack if necessary
70     while(true)
71     { unsigned min_weight = Map::MAX_WEIGHT;
72       Map::Route next_route;
73       for(Map::Location location = FIRST_LOCATION_ID;
74         location <= this->last_location_id; ++location)
75       { Map::Route new_route{route.source, location};
76         if(not this->map.is_mapped(new_route))
77         { continue; }
78         if(discarded_routes.is_mapped(new_route))
79         { continue; }
80         if(map.weight(new_route) < min_weight)
81         { min_weight = map.weight(new_route);
82           next_route = new_route;
83         } }
84       // If there is no path forward, time to backtrack
85       if(min_weight == Map::MAX_WEIGHT)
86       { return this->trip_plan.pop_back(), false; }
87       // The chosen route is marked as discarded
88       discarded_routes.weight(next_route);
89       // If a solution is recursively discovered, return that it is valid
90       if(greedy_planner({next_route.destination, route.destination}, false))
91       { return true; }
92       // Otherwise, try another route
93     } }
94
95     // 'a_star_planner' finds a solution using the A* approach
96     bool a_star_planner()
97     // Declare containers, initialise them with a source location
98     { using Tag = std::pair<unsigned, Map::Location>;
99       std::map<Map::Location, Tag> tags;
100       std::priority_queue<Tag, std::vector<Tag>, std::greater<Tag>> location_queue;
101       tags[this->trip_route.source] = {0, 0};
102       location_queue.push({0, this->trip_route.source});
103       // Construct a path
104       while(not location_queue.empty())
105       { Map::Location source = location_queue.top().second;
106         location_queue.pop();
107         if(source == this->trip_route.destination)
108         { break; }
109         for(Map::Location destination = FIRST_LOCATION_ID;
110           destination <= this->last_location_id; ++destination)
111         { Map::Route route{source, destination};
112           if(not this->map.is_mapped(route))
113           { continue; }
114           unsigned weight = tags[source].first + this->map.weight(route);
115           if(tags.count(destination) == 0 or weight < tags[destination].first)
116           { tags[destination] = {weight, source};
117             weight += heuristic_distance(destination);
118             location_queue.push({weight, destination});
119           } } }
120       // Reconstruct a path based on saved tags
121       Map::Location destination = this->trip_route.destination;
122       while(destination != 0)
123       { this->trip_plan.push_back(destination);
124         destination = tags[destination].second;
125       }
126       return std::reverse(this->trip_plan.begin(), this->trip_plan.end()), true;
127     }
128
129 public:

```

```

130 // The constructor reads the required data from stdin and initialises the map
131 // accordingly
132 TripPlanner()
133 { std::cin >> this->trip_route.source >> this->trip_route.destination;
134
135     Map::Location route_source, route_target;
136     float weight;
137     while(std::cin >> route_source >> route_target >> weight)
138     { this->map.weight({route_source, route_target}) = weight;
139       this->last_location_id = std::max({
140         route_source, route_target,
141         this->last_location_id });
142     } }
143
144 // 'plan' selects an algorithm to be used and then calls the appropriate
145 // method
146 TripPlanner &plan(bool use_a_star)
147 {
148     if(use_a_star)
149         this->a_star_planner();
150     else
151         this->greedy_planner(this->trip_route);
152     return *this;
153 }
154
155 friend std::ostream &operator<<(std::ostream &out, const TripPlanner &tabletop);
156 };
157
158 // Outputs a solution
159 std::ostream &operator<<(std::ostream &out, const TripPlanner &planner)
160 { for(int i = 0; i < planner.trip_plan.size(); i++)
161   { out << planner.trip_plan[i]; }
162   return out;
163 }
164
165 int main()
166 { return std::cout << TripPlanner().plan(true) << std::endl, 0; }
167

```