

Санкт-Петербургский политехнический университет Петра Великого  
Институт машиностроения, материалов и транспорта

Курсовая работа  
по теме «А\* алгоритм»  
по дисциплине «Объектно-ориентированное программирование»

Студент группы 3331506/90401

Кильдибекова А. А.

Преподаватель

Ананьевский М. С.

«\_\_\_\_\_»\_\_\_\_\_2022 г.

Санкт-Петербург  
2022 г.

## Содержание

Введение .....	3
Описание алгоритма .....	4
Исследование алгоритма .....	8
Заключение .....	10
Список литературы .....	11
Приложение .....	12

## Введение

A\* алгоритм (A-star алгоритм) – это один из наиболее широко используемых алгоритмов поиска кратчайших путей на евклидовом графе. Его частным случаем является алгоритм Дейкстры.

A\* получил широкое распространение во многих программных системах, от машинного обучения и поисковой оптимизации до разработки игр. К конкретным примерам можно отнести маршрутизацию телефонного трафика, навигацию по лабиринту, компоновку печатных плат, механическое доказательство теорем и решение задач. Основная область применения алгоритма – нахождение кратчайшего пути на графе, который является моделью карты.

Ограничением по применению A\* является необходимость наличия евклидового графа, то есть такого графа, вершины которого являются точками на плоскости и, соответственно, обладают координатами.

A\* принято считать самостоятельным алгоритмом, но на самом деле – это семейство алгоритмов, так как он вычисляет маршрут для заданной эвристики. Выбор конкретной эвристической функции определяет конкретный алгоритм из семейства.

## Описание алгоритма

A\* объединяет два способа поиска пути - математический подход и эвристический подход. Математический подход обычно имеет дело со свойствами абстрактных графов и с алгоритмами, которые предписывают упорядоченное изучение узлов графа для определения пути с минимальными затратами. В рамках этого подхода определяется минимальное расстояние (стоимость перемещения) от исходного (стартового) узла к текущему (рассматриваемому узлу). Принято обозначать его, как  $g$ .

Эвристический подход обычно использует специальные знания о предметной области задачи, представленной графом, для повышения вычислительной эффективности решений конкретных задач поиска по графу. В рамках этого подхода используется некая эвристическая функция (её принято обозначать его, как  $h$ ), которая является оценкой стоимости оптимального пути от текущего узла до предпочтительного целевого узла. Многие задачи, которые могут быть представлены как задача нахождения пути с минимальной стоимостью на графе, содержат некоторую информацию, которая может быть использована для формирования эвристической функции. Выбор  $h = 0$  соответствует случаю отсутствия или незнания этой информации. С такой эвристической функцией A\* будет представлять собой алгоритм Дейкстры. Поскольку мы, на самом деле, обладаем большей информацией, а именно знаем, как вычисляется расстояние в евклидовом пространстве через координаты двух точек, можно рассчитывать  $h$ , как  $\sqrt{x^2 + y^2}$  (где  $x$  и  $y$  - величины различий в координатах  $x$ ,  $y$  вершин). Введение эвристической функции помогает значительно уменьшить количество рассматриваемых узлов.

Два этих подхода объединяются введением функции оценки стоимости оптимального пути  $f$ , которая вычисляется как  $f = g + h$ .

Работа алгоритма:



Перед началом работы создаются два списка open (граница) – список узлов, которые находятся в очереди на посещение и closed – список посещённых узлов.

1. Заносят начальный узел в границу (список open)
2. Для него вычисляется  $f$
3. Определяется узел с наименьшим значением  $f$  и назначается текущим
4. Проверяют, не является ли текущий узел целевым
5. Текущий узел убирают из границы и заносят его в посещённые узлы (список closed)
6. Для каждого из узлов в границе вычисляется  $f$
7. Для текущего узла находят соседей и заносят их в границу (список open)
8. Повторяют пункты 3-8

Псевдокод:

```
function A* {
    var closed = empty();
    var open = list (start);
    var from = map (null);
    g[start] = 0;
    f[start] = g[start] + h(start, end);
    while (open) {
        curr = min_f(open);
        if (curr == end) {return success;}
        remove(curr, open);
        add(curr, closed);
        for each neighbour in unclosed_neighbour(curr) {
            temp_g = g[curr] + dist(curr, neighbour);
            if (neighbour not in open or temp_g < g[neighbour]) {
                from[neighbour] = curr;
                g[neighbour] = temp_g;
                f[neighbour] = g[neighbour] + h(neighbour, end);
                if (neighbour not in open) {add(neighbour, open);}
            }
        }
    }
    return failure;
}
```

Рассмотрим пример:

Задача: построить кратчайший путь от узла  с координатами (0,0) к узлу  с координатами (4,5), стоимость перемещения между клетками – 1. Граф задан в виде сетки.

На рисунке 1 изображено начало нахождения пути. Текущий узел является начальным. Для него вычисляется оценка общего расстояния  $f = g + h = h = (4 - 0) + (5 - 0) = 9$ . Затем находятся его соседи (они заносятся в границу), для каждого из которых тоже вычисляется  $f$ . Определяется узел с минимальным значением  $f$ , он становится текущим.

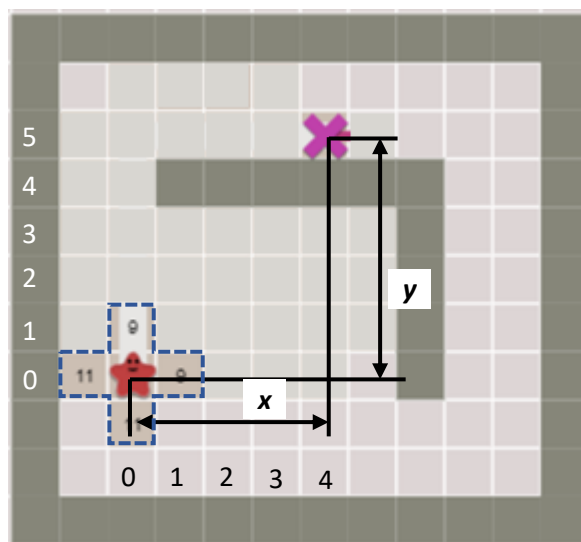


Рисунок 1 – Пример

Рассмотрим подробнее, как вычисляется оценка общего расстояния. На рисунке 2 изображена середина процесса нахождения пути. Допустим, что текущий узел задаётся координатами (1,2). Его соседний узел, занесённый в границу, задаётся координатами (2,2). Минимальное расстояние до этого узла от начального  $g = 4$  (количество пройденных клеток – фиолетовая линия). Эвристическая функция задана как сумма разности координат по  $x$  и по  $y$ . Оценка расстояния от рассматриваемого узла к целевому  $h = (4 - 2) + (5 - 2) = 5$ . Тогда оценка общего расстояния для рассматриваемого узла  $f = 4 + 5 = 9$ .

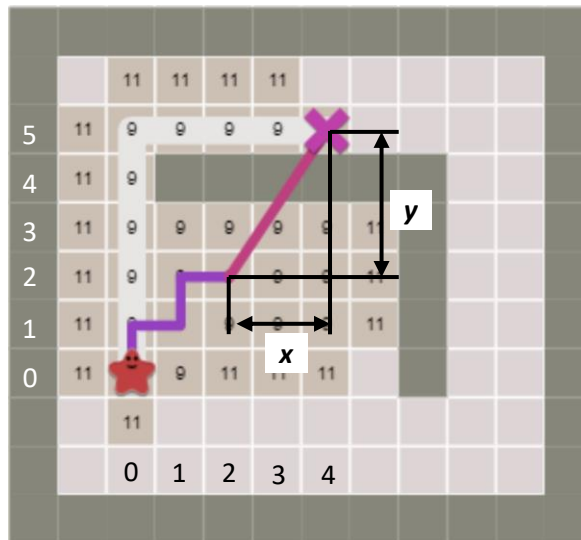


Рисунок 2 – Пример середины процесса

При написании кода был использован ориентированный взвешенный граф (так как это более общий случай), для которого был предварительно составлен массив, содержащий количество шагов между каждым из узлов.

## Исследование алгоритма

Временная сложность алгоритма  $A^*$  зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

где  $h^*$  — оптимальная эвристика, то есть точная оценка расстояния из вершины  $x$  к цели. Другими словами, ошибка  $h(x)$  не должна расти быстрее, чем логарифм от оптимальной эвристики.

Получены следующие графики временной сложности алгоритма для максимальной стоимости перемещения между узлами 1, 25 и 50 (рис. 3, 4, 5). Для сравнения также представлены данные полученные при работе алгоритма Дейкстры.

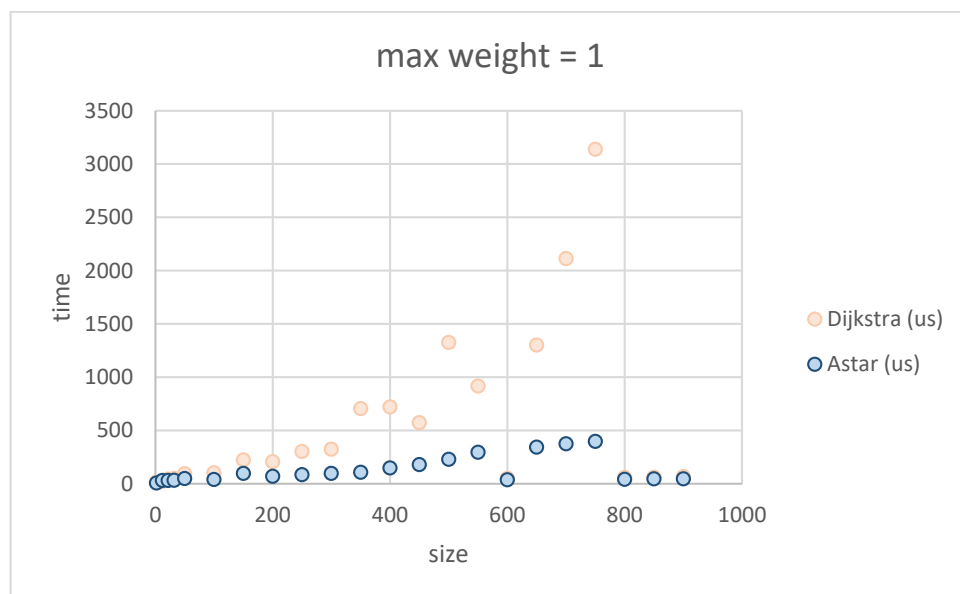


Рисунок 3 – Графики временной сложности алгоритма



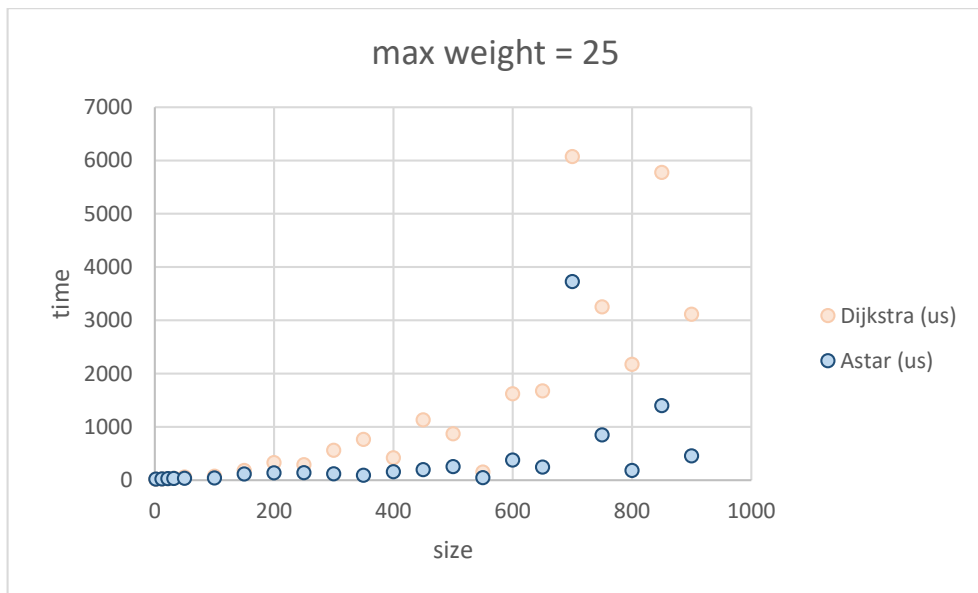


Рисунок 4 – Графики временной сложности алгоритма

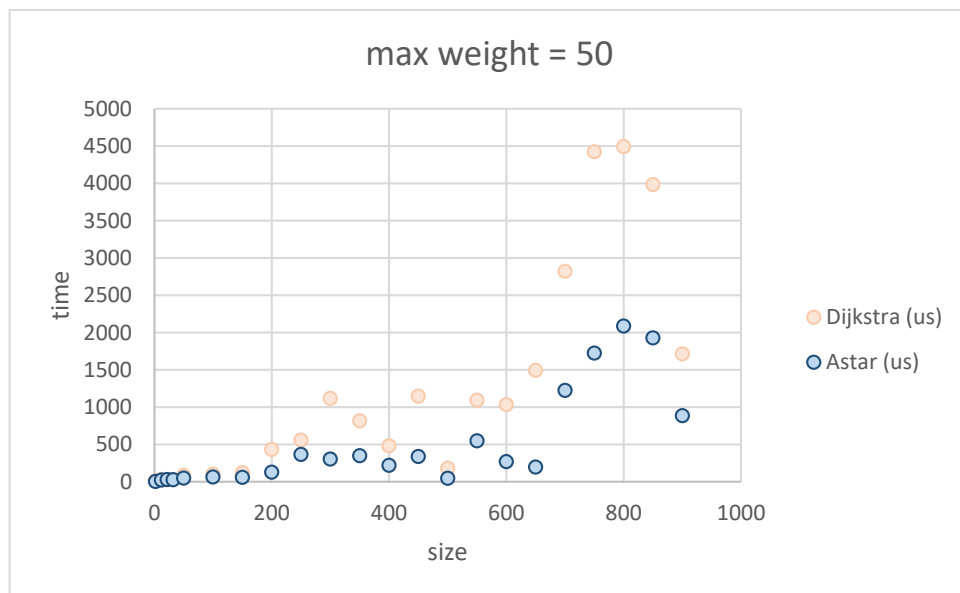


Рисунок 5 – Графики временной сложности алгоритма

## **Заключение**

A\* алгоритм хорошо подходит для определения кратчайшего пути на графе, который является моделью карты, с минимальными затратами по времени и памяти. Однако, если специальные знания о графе скудные или отсутствуют, этот алгоритм будет менее эффективен. В таком случае предлагается использовать алгоритм Дейкстры.

## Список литературы

1. Питер Э. Харт, Нильс Дж. Нильссон и Бертрам Рафаэль. A Formal Basis for the Heuristic Determination of Minimum Cost Paths - IEEE Transactions on Systems Science and Cybernetics , гл. 4, стр. 100-107, июль 1968 г.
2. Седжвик Роберт. Фундаментальные алгоритмы на С. Анализ/Структуры данных/Сортировка/Поиск/Алгоритмы на графах: Пер. с англ./Роберт Седжвик – СПб: ООО «ДиаСофтЮП», 2003. — 1136 с.
3. Люгер Джордж Ф. Искусственный интеллект: стратегии и методы решения сложных проблем, 4-е издание: Пер. с англ. – М: Издательский дом «Вильямс», 2003. – 864 с.
4. Amit Patel. Red Blob Games [Электронный ресурс] // Introduction to the A\* Algorithm. URL: <https://www.redblobgames.com/pathfinding/a-star/introduction.html> (даты обращения: 20.02.2022-20.05.2022).

## Приложение

### A-star.h

```
1  //
2  //
3  //
4  #ifndef A_STAR_ALGORITHM_A_STAR_H
5  #define A_STAR_ALGORITHM_A_STAR_H
6  #include <iostream>
7  #include <string.h>
8
9  class Graph;
10 class Front;
11
12 enum type_of_graph {NETWORK, BAD_CASE};
13 enum type_of_search {AStar, Dijkstra};
14
15 class Graph {
16 public:
17     Graph(uint size, type_of_graph, uint max_weight = 1);
18     virtual ~Graph();
19 private:
20     uint size;
21     uint **adjacency_table;
22     uint max_weight;
23 private:
24     void memory_allocation();
25     void free_memory();
26     void network_graph();
27     void bad_case_graph();
28 private:
29     uint random_weight();
30 private:
31     void get_heuristic_table();
32     void heuristic_mem_alloc();
33     void free_heuristic_mem();
34     void create_heuristic_table();
```

```

35     public:
36         void AStarSearch(uint start_node = 0, uint goal_node = 0);
37         void DijkstraSearch(uint start_node = 0, uint goal_node = 0);
38     private:
39         uint start = 0;           // Номер 1-го узла - 0
40         uint goal = 0;
41         uint **heuristic_table;
42         uint* dist_from_start_to;
43         uint* dist_estimate;
44         uint* path;
45         uint path_cost;
46         uint path_length;
47     private:
48         void find_AStar_path();
49         void AStar_mem_alloc();
50         void free_AStar_mem();
51         uint dist_to_goal_from(uint node);
52     private:
53         void find_Dijkstra_path();
54         void Dijkstra_mem_alloc();
55         void free_Dijkstra_mem();
56     private:
57         void print_path_search_res(type_of_search algorithm);
58         void get_path(const Front* currFront);
59     public:
60         void print_graph();
61     public:
62         friend class Front;
63 };

```

```

64
65 class Front {
66     private:
67         explicit Front(const Graph * graph);
68         virtual ~Front();
69     private:
70         const Graph * graph;
71         uint* front;
72         uint front_pointer;    // Указатель на верх границы
73         bool* visited;
74     private:
75         void init_front();
76         void get_front(uint current_node);    // Добавление новых узлов
77         void change_front(uint current_node); // Убирает текущий узел
78         void clean_front();
79         void memory_allocation();
80         void free_memory();
81     public:
82         friend class Graph;
83 };
84 #endif //A_STAR_ALGORITHM_A_STAR_H
85

```

## A-star.cpp

```
1      #include "A-star.h"
2
3      Graph::Graph(uint size, type_of_graph type, uint max_weigh) {
4          this->size = size;
5          this->max_weight = max_weigh;
6          memory_allocation();
7          switch(type) {
8              case NETWORK: network_graph();
9                  break;
10             case BAD_CASE: bad_case_graph();
11                 break;
12             default:
13                 std::cout<<"Некорректный тип графа"<<std::endl;
14                 free_memory();
15                 return;
16         }
17         get_heuristic_table();
18     }
19
20     Graph::~~Graph() {
21         free_memory();
22         free_heuristic_mem();
23     }
24
25     void Graph::memory_allocation() {
26         adjacency_table = new uint* [size];
27         for (uint col = 0; col < size; col++) {
28             adjacency_table[col] = new uint[size];
29         }
30     }
```

```

31
32 → void Graph::free_memory() {
33     for (uint col = 0; col < size; col++) {
34         free(adjacency_table[col]);
35     }
36     free(adjacency_table);
37     size = 0;
38     max_weight = 0;
39 }
40
41 → void Graph::network_graph() {
42     for (uint row = 0; row < size; row++) {
43         for (uint col = 0; col < size; col++) {
44             adjacency_table[row][col] = random_weight();
45         }
46     }
47     for (uint node = 0; node < size; node++) {
48         adjacency_table[node][node] = 0;
49     }
50 }
51
52 → void Graph::bad_case_graph() {
53     network_graph();
54     for (uint row = 0; row < size; row++) {
55         for (uint col = 0; col < size; col++) {
56             adjacency_table[row][col] *= rand() % (1 + 1);
57         }
58     }
59 }
60
61 → void Graph::get_heuristic_table() {
62     heuristic_mem_alloc();
63     create_heuristic_table();
64 }
65
66 → void Graph::heuristic_mem_alloc() {
67     heuristic_table = new uint* [size];
68     for (uint col = 0; col < size; col++) {
69         heuristic_table[col] = new uint[size];
70     }
71 }

```



```

72
73 → void Graph::free_heuristic_mem() {
74     for (uint col = 0; col < size; col++) {
75         free(heuristic_table[col]);
76     }
77     free(heuristic_table);
78 }
79
80 → void Graph::create_heuristic_table() {
81     for (uint row = 0; row < size; row++) {
82         for (uint col = 0; col < size; col++) {
83             heuristic_table[row][col] = 0;
84         }
85     }
86     for (uint node = 0; node < size; node++) {
87         start = node;
88         uint curr_node = node;
89         uint next_node = node;
90
91         Front hFront ( graph: this);
92         hFront.init_front();
93
94         while (hFront.front_pointer > 0) {
95             hFront.change_front(curr_node);
96             hFront.get_front(curr_node);
97             next_node = hFront.front[0];
98             for (uint i = 0; i < hFront.front_pointer; i++) {
99                 if (adjacency_table[curr_node][hFront.front[i]]) {
100                     uint temp_length = heuristic_table[node][curr_node] + 1;
101                     if (!heuristic_table[node][hFront.front[i]] ||
102                         temp_length < heuristic_table[node][hFront.front[i]]) {
103                         heuristic_table[node][hFront.front[i]] = temp_length;
104                     }
105                 }
106             }
107             curr_node = next_node;
108         }
109     }
110     start = 0;
111 }

```

```

112
113 → void Graph::random_weight() {
114     uint weight = rand() % (max_weight + 1);
115     return weight;
116 }
117
118 → void Graph::print_graph() {
119     for (uint row = 0; row < size; row++) {
120         for (uint col = 0; col < size; col++) {
121             std::cout<<adjacency_table[row][col]<<"\t";
122         }
123         std::cout<<std::endl;
124     }
125 }
126
127
128 → void Graph::print_path_search_res(type_of_search algorithm) {
129     std::string alg_name;
130     switch (algorithm) {
131         case AStar:alg_name = "A-star";
132         break;
133         case Dijkstra:alg_name = "Dijkstra";
134         break;
135         default:alg_name = "Unknown algorithm";
136     }
137
138     if (!path_length) {
139         std::cout<<alg_name<<" path no found"<<std::endl;
140         return;
141     }
142     std::cout<<alg_name<<" cost: "<<path_cost<<std::endl;
143     std::cout<<alg_name<<" length: "<<path_length<<std::endl;
144     std::cout<<alg_name<<" path: \t";
145     for (uint node = 0; node < path_length - 1; node++) {
146         std::cout<<path[node]<<" -> ";
147     }
148     std::cout<<path[path_length - 1]<<std::endl<<std::endl;
149 }

```

```

150
151
152 → void Graph::AStarSearch(uint start_node, uint goal_node) {
153     if (start_node > size - 1) {return;}
154     if (goal_node > size - 1) {return;}
155     this->start = start_node;
156     this->goal = goal_node;
157     AStar_mem_alloc();
158     find_AStar_path();
159     print_path_search_res( algorithm: AStar);
160     free_AStar_mem();
161 }
162
163 → void Graph::find_AStar_path() {
164     if (start == goal) {return;}
165
166     path_cost = 0;
167     path_length = 0;
168     for (uint node = 0; node < size; node++) {
169         dist_from_start_to[node] = 0;
170         dist_estimate[node] = 0;
171     }
172
173     uint min_dist = max_weight + size;
174     uint current_node = start;
175     uint node_to_visit = start;
176
177     Front currFront ( graph: this);
178     currFront.init_front();
179     dist_estimate[start] = dist_from_start_to[start] + dist_to_goal_from(start);
180

```

```

180
181     while (currFront.front_pointer > 0) {
182         currFront.change_front(current_node);
183         currFront.get_front(current_node);
184         node_to_visit = currFront.front[0];
185         for (uint i = 0; i < currFront.front_pointer; i++) {
186             if (adjacency_table[current_node][currFront.front[i]]) {
187                 uint temp_dist_from_start = dist_from_start_to[current_node]
188                     + adjacency_table[current_node][currFront.front[i]];
189                 if (!dist_from_start_to[currFront.front[i]] ||
190                     dist_from_start_to[currFront.front[i]] > temp_dist_from_start) {
191                     dist_from_start_to[currFront.front[i]] = temp_dist_from_start; // Если к
192                     dist_estimate[currFront.front[i]] = dist_from_start_to[currFront.front[i]]
193                         + dist_to_goal_from(currFront.front[i]);
194                 }
195             }
196             if (min_dist > dist_estimate[currFront.front[i]]) {
197                 min_dist = dist_estimate[currFront.front[i]];
198                 node_to_visit = currFront.front[i];
199             }
200         }
201         // take_next_node
202         current_node = node_to_visit;
203         if (current_node == goal) {
204             currFront.clean_front();
205             path_cost = dist_from_start_to[current_node];
206             path = new uint[size];
207             for (uint node = 0; node < size; node++) {
208                 path[node] = 0;
209             }
210             get_path(&currFront);
211             return;}
212         min_dist += max_weight + size;
213     }
214 }
215
216 uint Graph::dist_to_goal_from(uint node) {
217     return (uint)((float)heuristic_table[node][goal] * (1.0 + (float)max_weight * 0.005f));
218 }

```

```

219
220 void Graph::AStar_mem_alloc() {
221     dist_from_start_to = new uint [size];
222     dist_estimate = new uint [size];
223 }
224
225 void Graph::free_AStar_mem() {
226     start = 0;
227     goal = 0;
228     free(path);
229     free(dist_from_start_to);
230     free(dist_estimate);
231     path_cost = 0;
232     path_length = 0;
233 }
234
235 void Graph::DijkstraSearch(uint start_node, uint goal_node) {
236     if (start_node > size - 1) {return;}
237     if (goal_node > size - 1) {return;}
238     this->start = start_node;
239     this->goal = goal_node;
240     Dijkstra_mem_alloc();
241     find_Dijkstra_path();
242     print_path_search_res( algorithm: Dijkstra);
243     free_Dijkstra_mem();
244 }
245 // ^ ^ -----
246 // ='t'= < MEOW~ ~ ~ ~ |
247 // u u S -----
248 void Graph::find_Dijkstra_path() {
249     if (start == goal) {return;}
250
251     path_cost = 0;
252     path_length = 0;
253     for (uint node = 0; node < size; node++) {
254         dist_from_start_to[node] = 0;
255     }
256
257     uint min_dist = max_weight;
258     uint current_node = start;
259     uint node_to_visit = start;

```

```

260
261     Front currFront ( graph: this);
262     currFront.init_front();
263
264     while (currFront.front_pointer > 0) {
265         currFront.change_front(current_node);
266         currFront.get_front(current_node);
267         node_to_visit = currFront.front[0];
268         for (uint i = 0; i < currFront.front_pointer; i++) {
269             if (adjacency_table[current_node][currFront.front[i]]) {
270                 uint temp_dist = dist_from_start_to[current_node]
271                     + adjacency_table[current_node][currFront.front[i]];
272                 if (!dist_from_start_to[currFront.front[i]] ||
273                     dist_from_start_to[currFront.front[i]] > temp_dist) {
274                     dist_from_start_to[currFront.front[i]] = temp_dist; // Если
275                 }
276             }
277             if (min_dist > dist_from_start_to[currFront.front[i]]) {
278                 min_dist = dist_from_start_to[currFront.front[i]];
279                 node_to_visit = currFront.front[i];
280             }
281         }
282         // take_next_node
283         current_node = node_to_visit;
284         if (current_node == goal) {
285             currFront.clean_front();
286             path_cost = dist_from_start_to[current_node];
287             path = new uint [size];
288             for (uint node = 0; node < size; node++) {
289                 path[node] = 0;
290             }
291             get_path(&currFront);
292             return;}
293         min_dist += max_weight;
294     }
295 }

```

```

296
297 void Graph::get_path(const Front* currFront) {
298     uint current_node = goal;
299     path[0] = goal;
300     uint i = 0;
301     while (current_node != start) {
302         for (uint prev_node = 0; prev_node < size; prev_node++) {
303             if (currFront->visited[prev_node] && adjacency_table[prev_node][current_node]
304                 && (dist_from_start_to[prev_node] + adjacency_table[prev_node][current_node]
305                     == dist_from_start_to[current_node])) {
306                 i++;
307                 path[i] = prev_node;
308                 current_node = prev_node;
309                 prev_node = size - 1;
310             }
311         }
312     }
313     //reflect path
314     path_length = i;
315     for (; i > path_length/2; i--) {
316         uint temp = path[path_length - i];
317         path[path_length - i] = path[i];
318         path[i] = temp;
319     }
320     path_length++;
321 }
322
323 void Graph::Dijkstra_mem_alloc() {
324     dist_from_start_to = new uint [size];
325 }
326
327 void Graph::free_Dijkstra_mem() {
328     start = 0;
329     goal = 0;
330     free(path);
331     free(dist_from_start_to);
332     path_cost = 0;
333     path_length = 0;
334 }

```

```

335
336     /// class Front ///
337
338     Front::Front(const Graph * graph) {
339         this->graph = graph;
340         memory_allocation();
341     }
342
343     Front::~~Front() {
344         free_memory();
345     }
346
347     void Front::memory_allocation() {
348         front = new uint [graph->size];
349         visited = new bool [graph->size];
350         front_pointer = 0;
351     }
352
353     void Front::free_memory() {
354         free(front);
355         free(visited);
356         front_pointer = 0;
357     }
358
359     void Front::init_front() {
360         for (uint node = 0; node < graph->size; node++) {
361             visited[node] = false;
362             front[node] = 0;
363         }
364         front[0] = graph->start;
365         front_pointer = 1;
366         visited[graph->start] = true;
367     }

```



```

368
369 → void Front::get_front(uint current_node) {
370     for (uint node = 0; node < graph->size; node++) {
371         if (!visited[node] && graph->adjacency_table[current_node][node]) {
372             visited[node] = true;
373             front[front_pointer] = node;
374             front_pointer++;
375         }
376     }
377 }
378
379 → void Front::change_front(uint current_node) {
380     for (uint i = 0; i < front_pointer; i++) {
381         if (front[i] == current_node) {
382             front[i] = front[front_pointer - 1];
383             front[front_pointer - 1] = 0;
384             front_pointer--;
385             return;
386         }
387     }
388 }
389
390 → void Front::clean_front() {
391     while (front_pointer > 0) {
392         front_pointer--;
393         visited[front[front_pointer]] = false;
394         front[front_pointer] = 0;
395     }
396 }

```

## main.cpp

```
1  #include <iostream>
2  #include "A-star.h"
3  #include <chrono>
4
5  int main() {
6      uint max_weight = 25;
7      std::cout<<"-----"<<std::endl;
8      for (uint i = 2; i <= 32; i += 10) {
9          std::cout << "for size = " << i << "\t and max_weight = " << max_weight << std::endl;
10         Graph First(i, BAD_CASE, max_weight);
11         //First.print_graph();
12
13         std::chrono::high_resolution_clock::time_point Dijkstra_start_time = std::chrono::high_resolution_clock::now();
14         First.DijkstraSearch( start_node: 0, goal_node: 1);
15         std::chrono::high_resolution_clock::time_point Dijkstra_end_time = std::chrono::high_resolution_clock::now();
16
17         std::chrono::high_resolution_clock::time_point AStar_start_time = std::chrono::high_resolution_clock::now();
18         First.AStarSearch( start_node: 0, goal_node: 1);
19         std::chrono::high_resolution_clock::time_point AStar_end_time = std::chrono::high_resolution_clock::now();
20
21         auto Dijkstra_runtime = std::chrono::duration_cast<std::chrono::microseconds>(
22             Dijkstra_end_time - Dijkstra_start_time).count();
23         auto AStar_runtime = std::chrono::duration_cast<std::chrono::microseconds>(
24             AStar_end_time - AStar_start_time).count();
25
26         std::cout << "### Dijkstra runtime: " << Dijkstra_runtime << " us." << std::endl;
27         std::cout << "### AStar runtime: " << AStar_runtime << " us." << std::endl;
28         std::cout << std::endl << std::endl;
29     }
30     return 0;
31 }
```