

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Курсовой проект по теме «Эвристический поиск на решётках»

Студент: С. М. Власова
Преподаватель: А. А. Журавлёв
Группа: М8О-206Б
Дата:
Оценка:
Подпись:

Москва, 2019

Задание курсовой работы

Эвристический поиск на решётках.

1 Описание

1

1000

Данные

Реализуйте алгоритм A^* для графа на решётке.

Первые четыре строки входного файла выглядят следующим образом:

```
type octile  
height <x>  
width <y>  
map
```

Где "*x*" и "*y*" — высота и ширина карты соответственно.

Далее в *x* строках задана сама карта в виде решётки символов, в которой символы '.' и 'G' обозначают проходимые клетки. Переход между ячейками возможен только по сторонам.

Далее дано число *q* и в следующих *q* строках даны запросы в виде четвёрок чисел на поиск кратчайшего пути между двумя позициями в решётке.

В ответ на каждый запрос выведите единственное число — расстояние между ячейками из запроса.

2 Определение

Эвристический алгоритм — это алгоритм решения задачи, правильность которого для всех возможных случаев не доказана, но про который известно, что он даёт достаточно хорошее решение в большинстве случаев. В действительности может быть даже известно (то есть доказано) то, что эвристический алгоритм формально неверен. Его всё равно можно применять, если при этом он даёт неверный результат только в отдельных, достаточно редких и хорошо выделяемых случаях или же даёт неточный, но всё же приемлемый результат.

Проще говоря, эвристика — это не полностью математически обоснованный (или даже «не совсем корректный»), но при этом *практически полезный алгоритм*.

Важно понимать, что эвристика, в отличие от корректного алгоритма решения задачи, обладает *следующими особенностями*:

1. Она не гарантирует нахождение лучшего решения;
2. Она не гарантирует нахождение решения, даже если оно заведомо существует (возможен «пропуск цели»);
3. Она может дать неверное решение в некоторых случаях.

Поиск A^* — алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Порядок обхода вершин определяется **эвристической функцией** $f(x)$, определенной, как сумма двух других: $g(x)$ — функции стоимости достижения текущей вершины *current* из начальной, и $h(x)$ — функции эвристической оценки расстояния от рассматриваемой вершины к конечной.

Функция $h(x)$ должна быть допустимой эвристической оценкой, то есть не должна переоценивать расстояния к целевой вершине.

Алгоритм, по сути, является расширением алгоритма Дейкстры — благодаря введенной эвристике он быстрее находит кратчайший путь от одной вершины к другой.

3 Алгоритм

Реализация алгоритма A^* на решётках не сильно отличается от его реализации на графе — лишь допустимыми шагами и способом хранения вершин.

Каждая вершина имеет следующие атрибуты:

1. идентификатор, определяющий ее проходимость (если значение «.» или «G», то вершина считается проходимой);
2. значения функций h , f и g ;
3. ее координаты на решетке x и y ;
4. значения, определяющие степень ее «открытия» и «обработки» — $open$, $close$;
5. указатель на родительский узел (может меняться в процессе поиска оптимального пути).

Каждая вершина графа инициализируется начальными значениями $\{id, 0, 0, 0, x, y, false, false, NULL\}$.

Возможные ходы: т.к. граф на решётке, есть четыре возможных шага — вверх, вниз, вправо, влево (на 1). Стоит помнить, что не все клетки проходимые.

Принцип поиска: на каждом шаге алгоритма мы выбираем самую выгодную вершину v_{min} с минимальной оценкой f , достижимую из текущей вершины $current$. Значение этой функции рассчитывается следующим образом: $f = h + g$, $g = current \rightarrow g + 1$, $h = abs(current \rightarrow x - x_{min}) + abs(current \rightarrow y - y_{min})$. В случае, если у двух разных вершин значение функции f одинаковое, мы выбираем ту из них, расстояние от которой до конечной вершины меньше, т.е. сравниваем по функции h . Вершина v_{min} становится текущей, а остальные достижимые вершины (при условии, что они еще не были открыты) добавляются в очередь с приоритетами *OpenQueue*.

Такой способ хранения вершин удобен тем, что вершины с минимальным «весом» обрабатываются быстрее. Таким образом, вершины, которые кажутся наименее приоритетными, будут обрабатываться в последнюю очередь, если эффективный путь еще не будет найден. Подход, основанный на определенном предположении об оптимальном пути, позволяет избежать рассмотрения «ненужных» вершин.

Если вершина уже была «открыта», и мы опять находимся в радиусе ее доступности, то необходимо сделать переоценку ее функции f — если значение ее функции g окажется больше, чем $current \rightarrow g + 1$, необходимо изменить ее родителя на текущую вершину, значение g на выше упомянутое и переоценить значение функции $f = g + h$. Так мы найдем более оптимальный путь в эту вершину.

Алгоритм завершит свою работу, когда вершина v_{final} , в которую мы идем, будет открыта.

Для избежания заикливания мы храним вершины, которые уже являются частью

оптимального пути, в отдельной очереди *Close Queue*.

В процессе написания алгоритма я столкнулась с несколькими оптимизационными задачами.

Т.к. по заданию к каждой решётке может быть применен набор запросов, после каждого из них нужно «возвращаться» к начальным условиям — инициализировать вершины заново. Если это делать для всей решётки, затраты по времени окажутся большими. Чтобы избежать повторной инициализации вершин, которые не были обработаны в процессе предыдущего решения, можно хранить указатели на «затронутые алгоритмом» вершины в отдельном векторе. Так, после нахождения минимального расстояния, только вершины из этого вектора будут инициализированы заново.

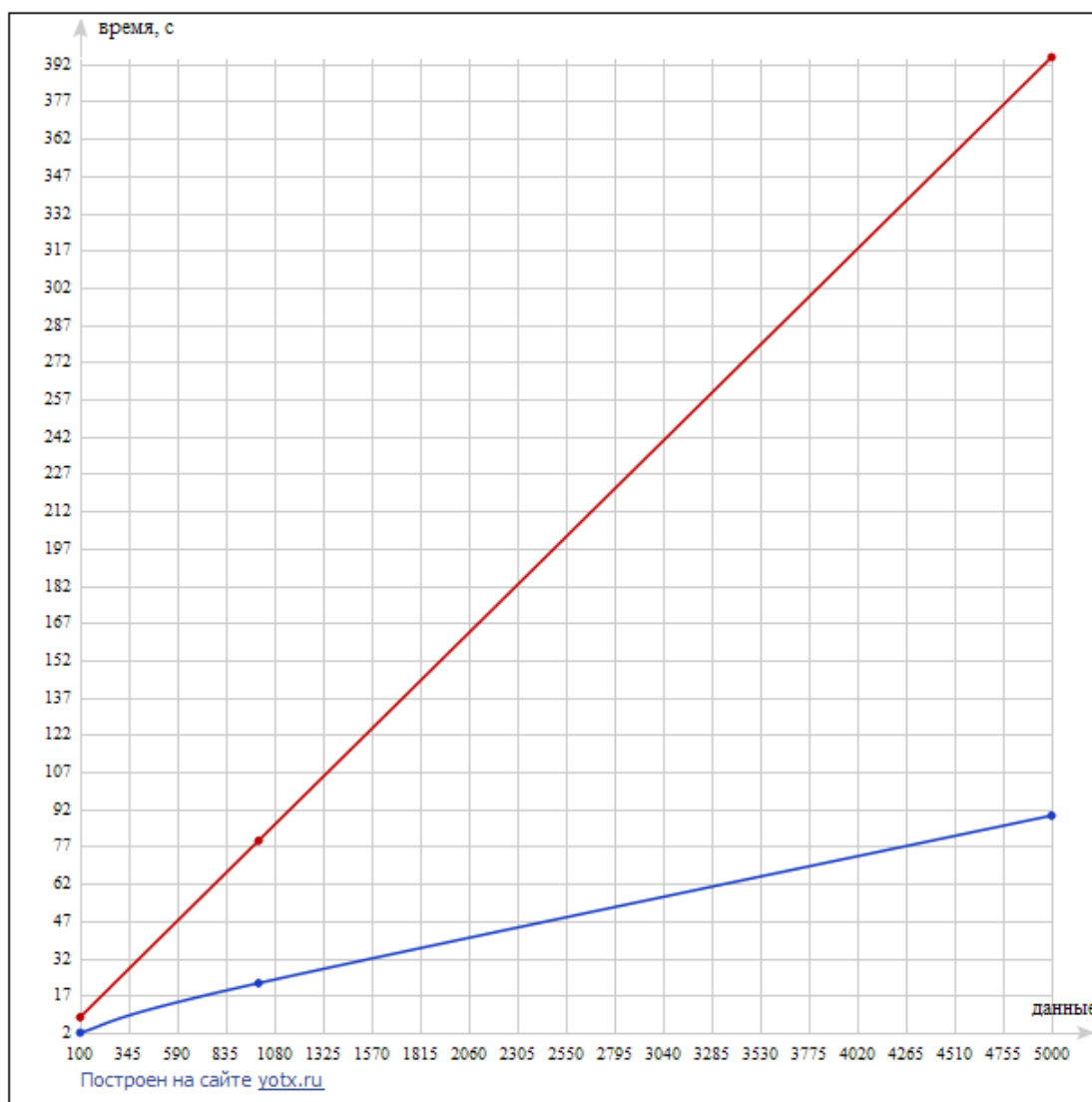
Чтобы в процессе поиска оптимального пути одни и те же вершины по несколько раз не добавлялись в этот вектор, будем добавлять вершину в этот вектор только при первом открытии, т.е. при условии, что $v \rightarrow open == false$.

4 Сравнение с Дейкстрой

В качестве тестов я выбрала две карты: FloodedPlains.map и battleground.map.

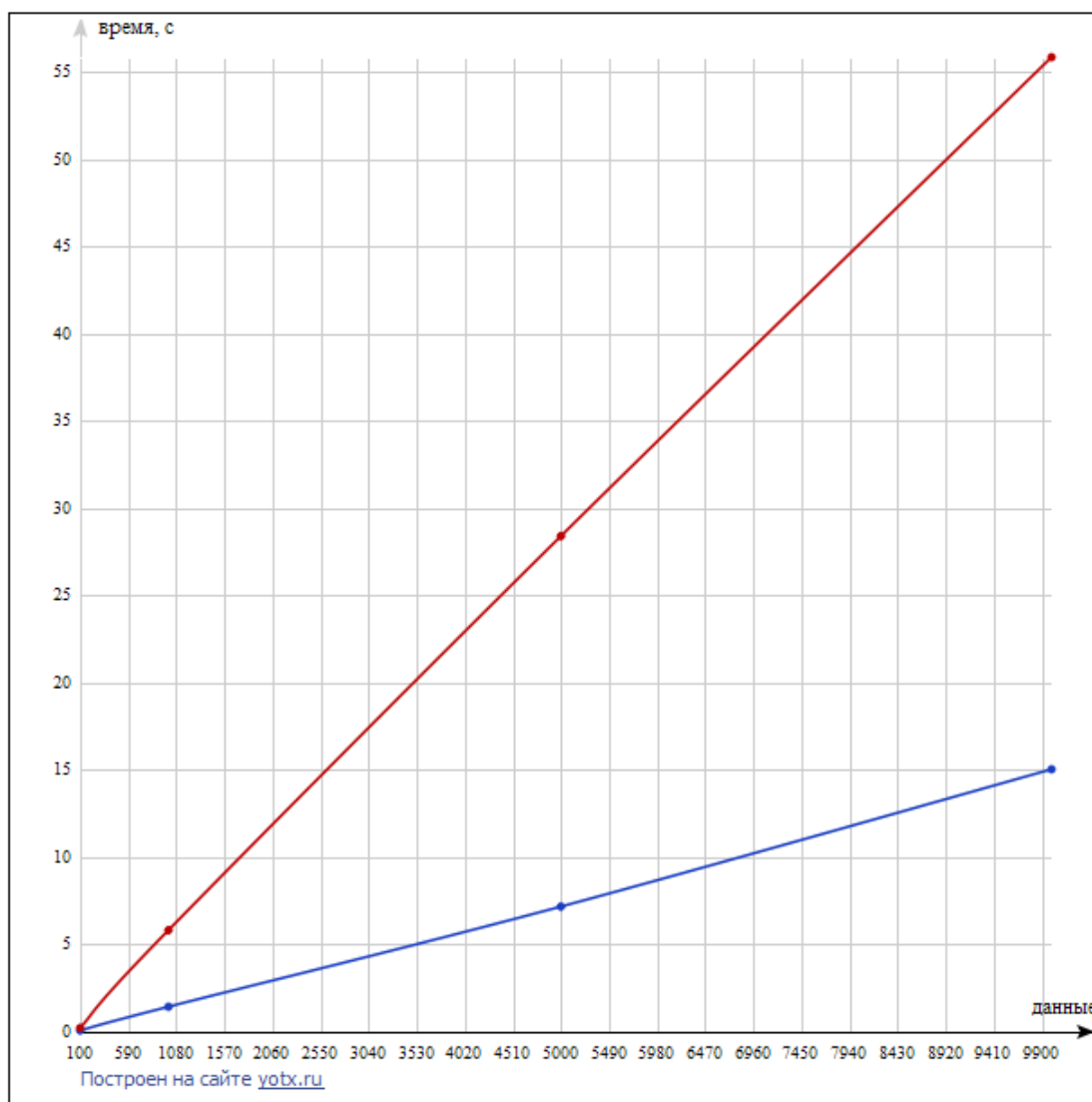
1. FloodedPlains.map: height 768, width 768

Количество запросов	Время A*	Время Дейкстры
100	1.63788	7.86195
1000	21.5844	78.9286
5000	89.0385	394.31



2. battleground.map: height 512, width 512

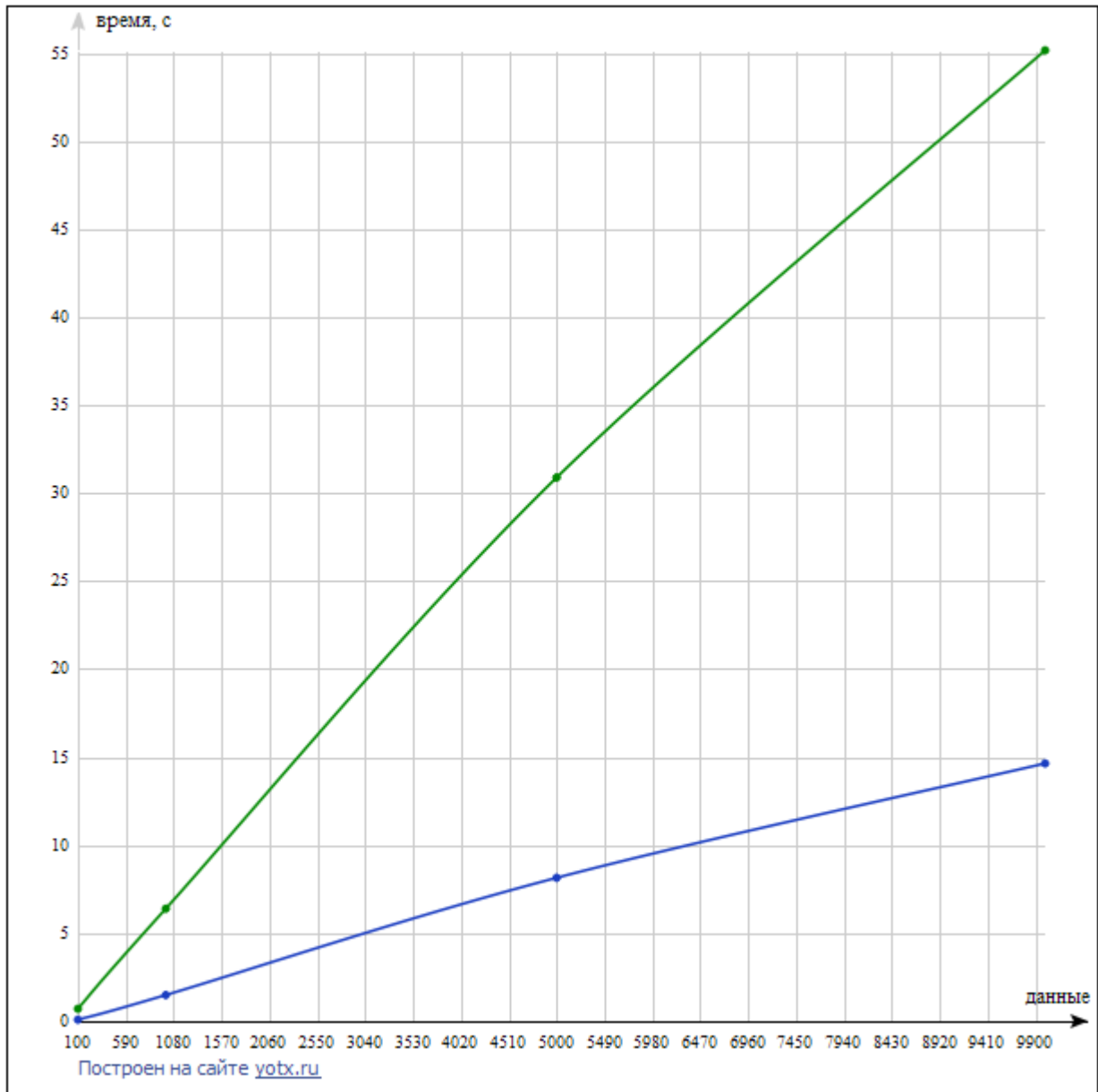
Количество запросов	Время A*	Время Дейкстры
100	0.077701	0.217977
1000	1.43707	5.82507
5000	7.17788	28.4098
10000	15.0465	55.8555



По графикам видно, что время работы алгоритма Дейкстры в несколько раз превышает время работы A*.

5 Сравнение с алгоритмом без доп.векторов

График хорошо иллюстрирует результаты оптимизационной задачи — производительность на больших данных улучшилась в несколько раз.



6 Исходный код

Листинг 1: Алгоритм A* на решётках

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <vector>
4 #include <string>
5 #include <queue>
6 #include <cmath>
7 #include <ctime>
8
9 #define PASS 1
10 #define UNPASS 0
11
12 typedef struct Cell{
13     int id;
14     int h;
15     int g;
16     int f;
17     int x, y;
18     bool open;
19     bool close;
20     struct Cell * parent_cell;
21 } Cell;
22
23 struct GreaterThanByF{
24     bool operator()(const Cell* left, const Cell* right) const{
25         if(left->f == right->f)
26             return (left->g > right->g);
27         return (left->f > right->f);
28     }
29 };
30
31 void A_Help_Search(std::vector<std::vector<Cell>>& Grill, std::vector<Cell*>& Changed, std:::
    priority_queue<Cell*, std::vector<Cell*>, GreaterThanByF>& OpenQueue, Cell * current, int
    x1, int y1, int x2, int y2){
32     if(Grill[x1][y1].id == PASS && Grill[x1][y1].close == false){
33         if(Grill[x1][y1].open == false){
34             Grill[x1][y1].open = true;
35             Grill[x1][y1].parent_cell = current;
36             Grill[x1][y1].g = current->g + 1;
37             Grill[x1][y1].h = abs(x2 - x1) + abs(y2 - y1);
```

```

38         Grill[x1][y1].f = Grill[x1][y1].g + Grill[x1][y1].h;
39         OpenQueue.push(&Grill[x1][y1]);
40         Changed.push_back(&Grill[x1][y1]);
41     }
42     else if(Grill[x1][y1].g > current->g + 1){
43         Grill[x1][y1].g = current->g + 1;
44         Grill[x1][y1].f = Grill[x1][y1].g + Grill[x1][y1].h;
45         Grill[x1][y1].parent_cell = current;
46         Cell * check = OpenQueue.top();
47         OpenQueue.push(check);
48     }
49 }
50 }
51
52 int A_Search(std::vector<std::vector<Cell>>& Grill, std::vector<int>& Num, std::vector<Cell
53 *>& Changed){
54     int x1 = Num[0], y1 = Num[1], x2 = Num[2], y2 = Num[3];
55     int path = 0;
56
57     if( Grill[x1][y1].id == UNPASS || Grill[x2][y2].id == UNPASS || x1 > (int)Grill.size() || x2 > (
58         int)Grill.size() || y1 > (int)Grill[0].size() || y2 > (int)Grill[0].size() || x1 < 0 || x2 < 0 ||
59         y1 < 0 || y2 < 0)
60         return -1;
61
62     std::priority_queue<Cell*, std::vector<Cell*>, GreaterThanByF> OpenQueue;
63     std::queue<Cell*> CloseQueue;
64     Cell * current = NULL;
65
66     current = &Grill[x1][y1];
67     OpenQueue.push(current);
68     current->open = true;
69     Changed.push_back(&Grill[x1][y1]);
70
71     while(Grill[x2][y2].open == false){
72         if(OpenQueue.empty())
73             return -1;
74         CloseQueue.push(current);
75         current->close = true;
76         OpenQueue.pop();
77
78         if(current->x - 1 >= 0)
79             A_Help_Search(Grill, Changed, OpenQueue, current, current->x - 1, current->y,
80                 x2, y2);
81         if(current->x + 1 < Grill.size())

```

```

78         A_Help_Search(Grill, Changed, OpenQueue, current, current->x + 1, current->y,
79                        x2, y2);
80     if(current->y - 1 >= 0)
81         A_Help_Search(Grill, Changed, OpenQueue, current, current->x, current->y - 1,
82                        x2, y2);
83     if(current->y + 1 < Grill[0].size())
84         A_Help_Search(Grill, Changed, OpenQueue, current, current->x, current->y + 1,
85                        x2, y2);
86
87     current = OpenQueue.top();
88 }
89 current = &Grill[x2][y2];
90 while(current->parent_cell!= NULL){
91     path++;
92     current = current->parent_cell;
93 }
94 return path;
95 }
96
97 void InitGrill(Cell * element){
98     element->f = 0;
99     element->h = 0;
100    element->g = 0;
101    element->open = false;
102    element->close = false;
103    element->parent_cell = NULL;
104 }
105
106 int main(){
107     time_t start = clock();
108     std::string str;
109     std::vector<int> Num(4);
110     std::vector<Cell*> Changed;
111     Cell element;
112     int height, width, q;
113
114     std::getline(std::cin, str);
115     std::getline(std::cin, str);
116
117     for( int i{1}; i < str.size(); i++ ){
118         try{
119             height = std::stoi( str.substr(i));
120             break;
121         }

```

```

119         catch(...)
120         {}
121     }
122     std::vector< std::vector<Cell> > Grill(height);
123     std::getline(std::cin, str);
124
125     for(int i{1}; i < str.size(); i++){
126         try{
127             width = std::stoi( str.substr(i) );
128             break;
129         }
130         catch(...)
131         {}
132     }
133     std::getline(std::cin, str);
134     for( int i = 0; i < height; i++){
135         std::getline(std::cin, str);
136         for( int j = 0; j < width; j++){
137             if( str[j] == '.' || str[j] == 'G' )
138                 element.id = PASS;
139             else element.id = UNPASS;
140             InitGrill(&element);
141             element.x = i;
142             element.y = j;
143             Grill[i].push_back(element);
144         }
145     }
146     std::cin >> q;
147     while(q > 0){
148         for(int i = 0; i < 4; i++)
149             std::cin >> Num[i];
150         std::cout << A_Search(Grill, Num, Changed) << std::endl;
151         while(!Changed.empty()){
152             InitGrill(Changed.back());
153             Changed.pop_back();
154         }
155         q--;
156     }
157     time_t end = clock();
158     std::cout << "TIME: " << (double)(end - start) / CLOCKS_PER_SEC << " seconds"
159         << std::endl;
160     return 0;
}

```

7 Выводы

Выполнив эту курсовую работу, я познакомилась с таким понятием, как эвристика, и реализовала простой эвристический алгоритм A^* на решётке, позволяющий найти кратчайший путь достаточно быстро. Эвристический алгоритм опирается на определенные предположения, и, исходя из этого, выстраивает более или менее приоритетные пути решения. В моем алгоритме эвристической функцией является сумма катетов прямоугольного треугольника, гипотенузой которого являются соединенные между собой текущая и конечная вершина.

Я провела сравнительный анализ, наглядно показав, что эвристический алгоритм значительно опережает алгоритм Дейкстры по производительности.