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, 2020

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 возможные 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-> g + 1*, h = *abs(current-> x -* x_{min}*) + abs(current-> 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-> 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 зацикливания мы храним вершины, которые уже являются частью

оптимального пути, в отдельной очереди *Close Queue.*

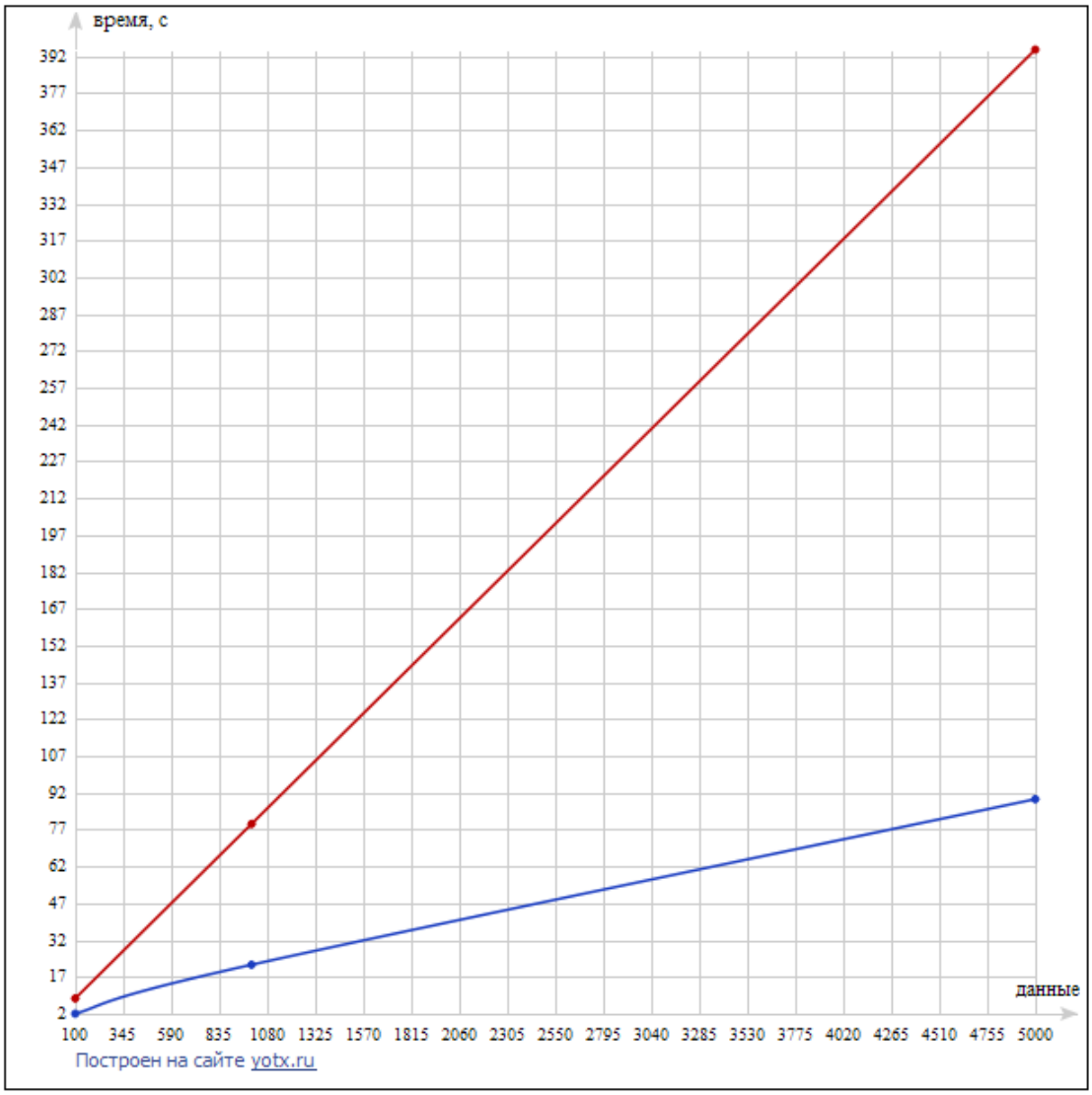
В процессе написания алгоритма я столкнулась с несколькими оптимизационными задачами.
Т.к. по заданию к каждой решётке может быть применим набор запросов, после каждого из них нужно «возвращаться» к начальным условиям — инициализировать вершины заново. Если это делать для всей решётки, затраты по времени 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 нескольку раз не добавлялись в этот вектор, будем добавлять вершину в этот вектор только при первом открытии, т.е. при условии, что $v\text{->}\ open == false$.

4 Сравнение с Дейкстрой

В качестве тестов я выбрала три карты: FloodedPlains.map, battleground.map и GladiatorPits.map.

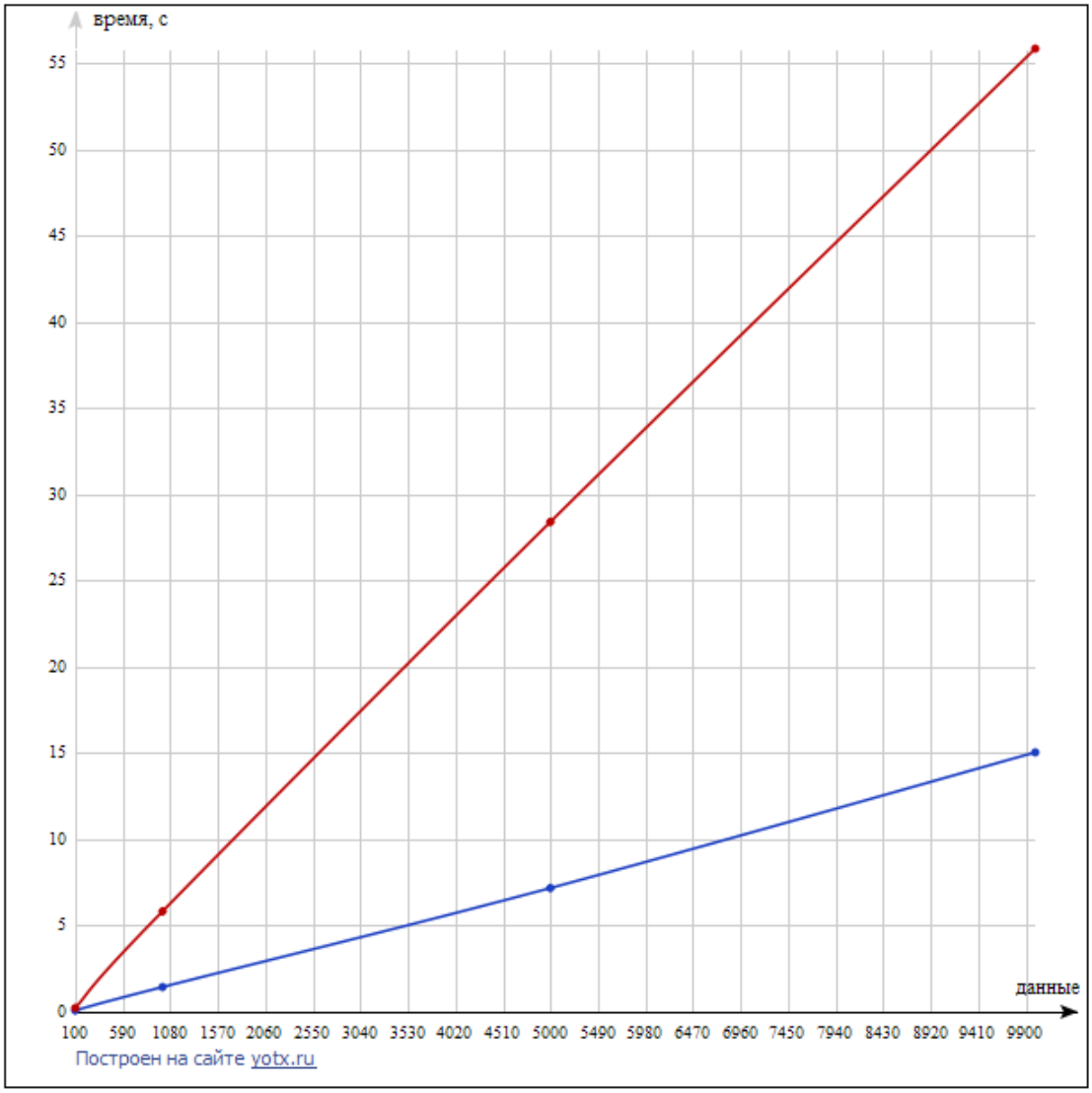
1. **FloodedPlains.map: height 768, width 768**

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



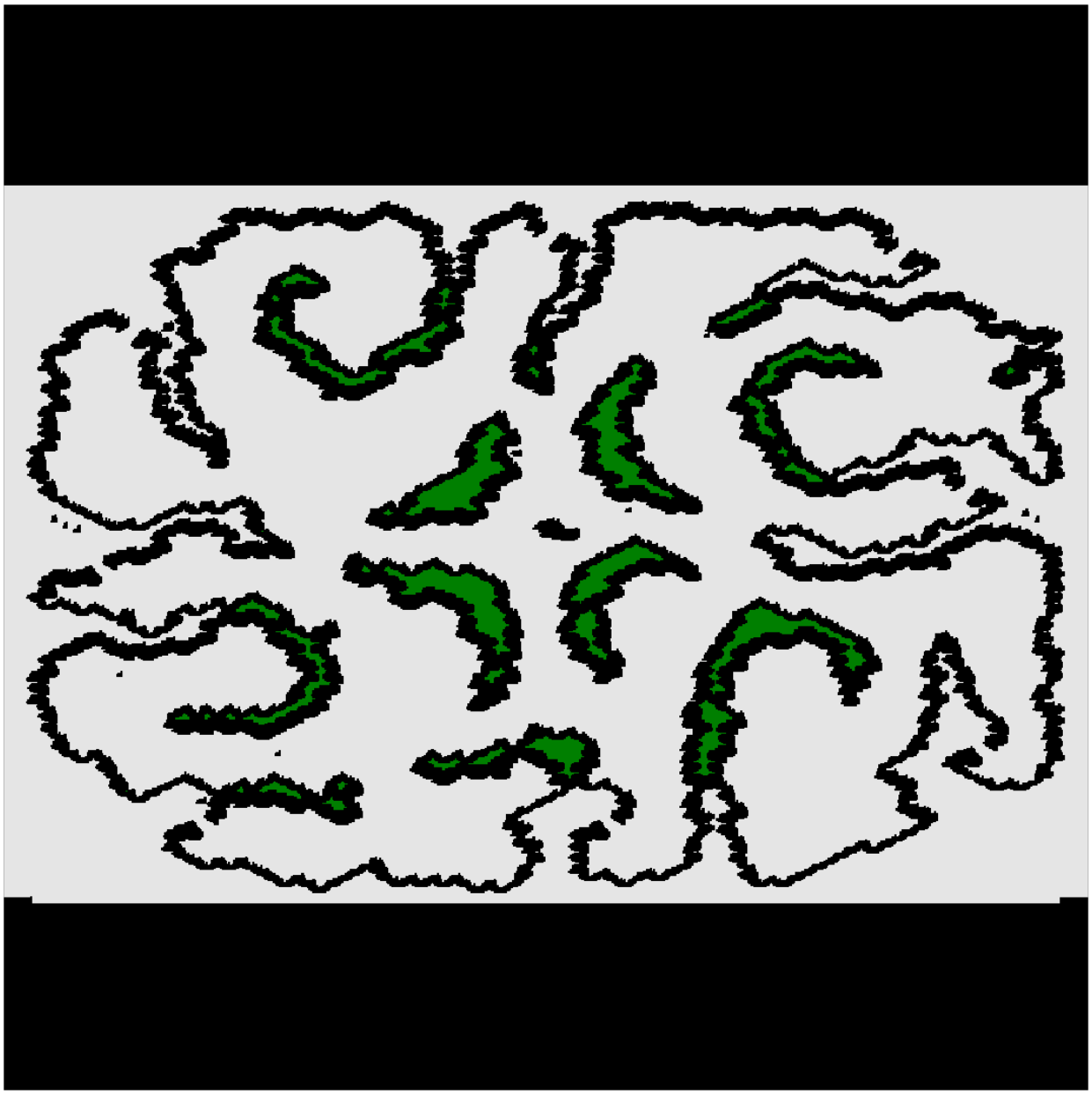
2. **battleground.map: height 512, width 512**

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

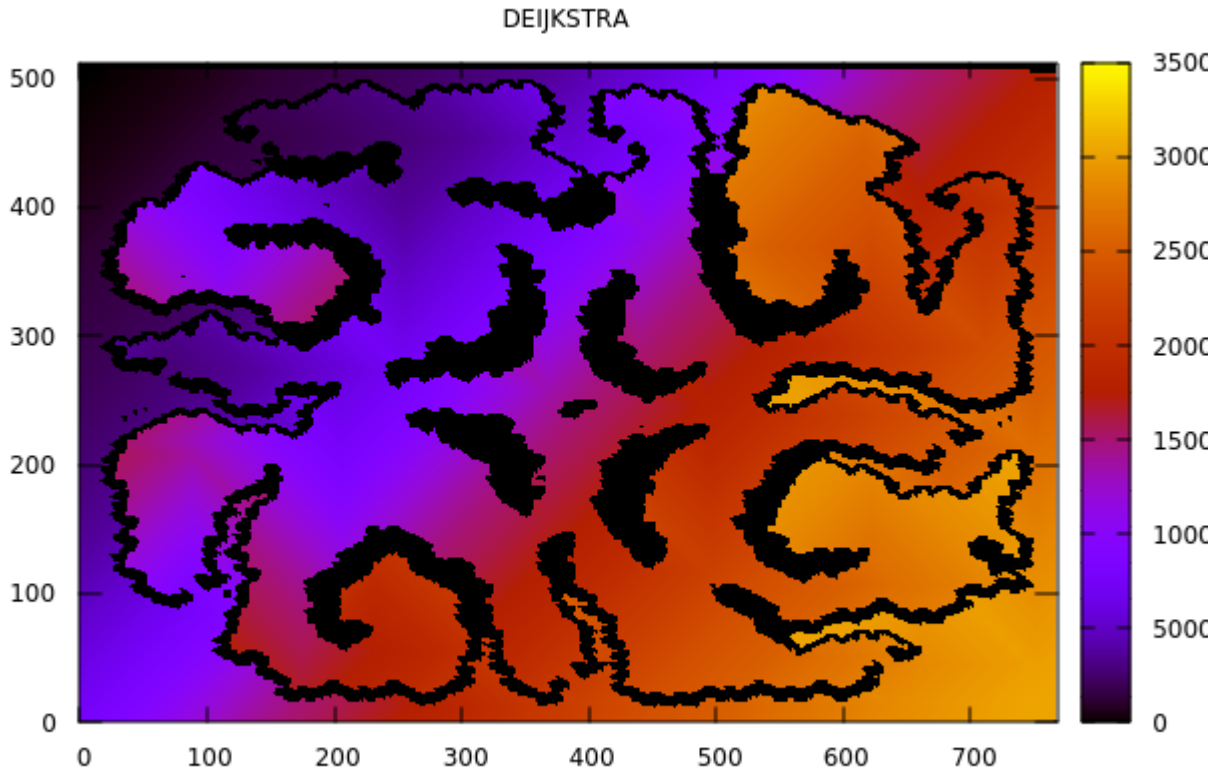


По графикам видно, что время работы алгоритма Дейкстры в несколько раз превышает время работы A*.

Сравним алгоритмы с помощью построения тепловой карты для каждого из них. Тепловая карта будет строиться по одному запросу — поиску расстояния между двумя угловыми точками карты.
В качестве исходной карты я взяла GladiatorPits.map(512x768):

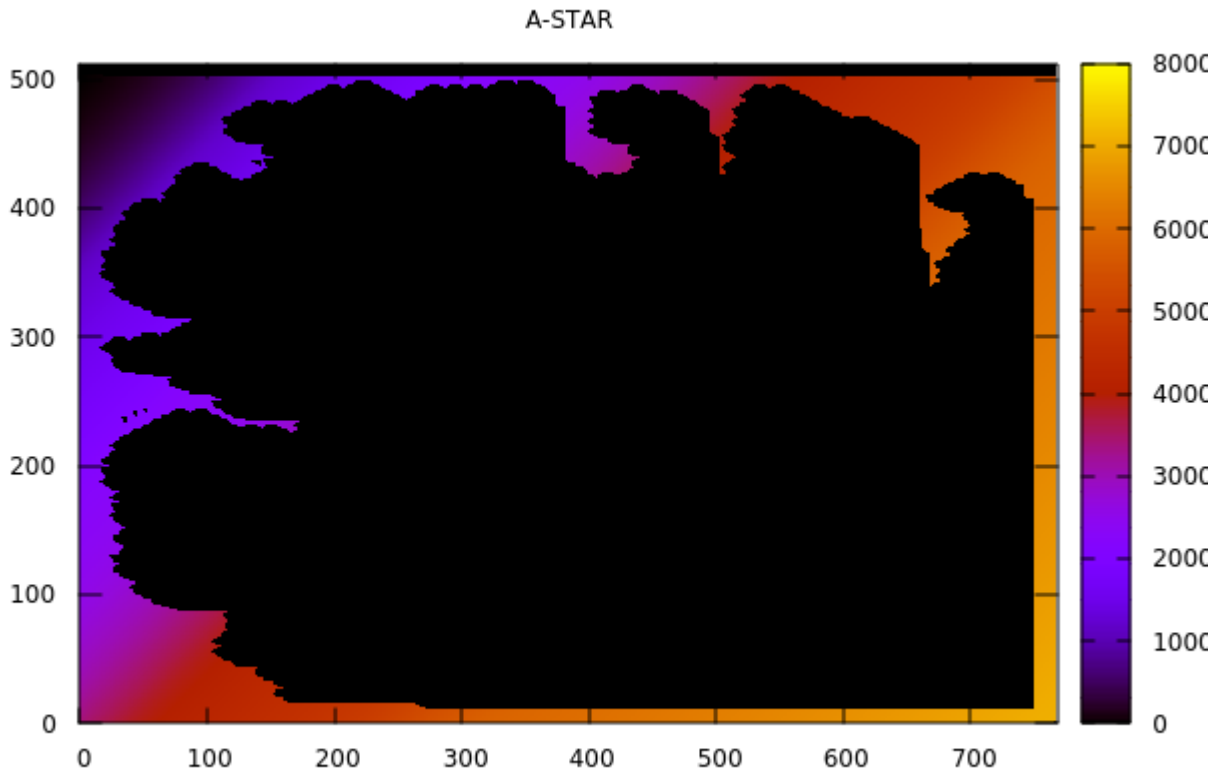


Тепловая карта работы алгоритма Дейкстры:



На данной карте расстояние искалось между точками (0, 503) и (767, 0).
Шкала варьируется от черного цвета, означающего, что вершина не была посещена (значение ноль), до желтого цвета, означающего, что вершина была посещена позже всех — т.е. цвета пикселей соответствуют моментам времени, в которые они были посещены. Чем темнее цвет, тем раньше вершина была достигнута, и наоборот.
По тепловой карте работы алгоритма Дейкстры можно сделать вывод о том, что была посещена каждая доступная вершина, т.е. поиск пути происходил во всех направлениях, равномерно.

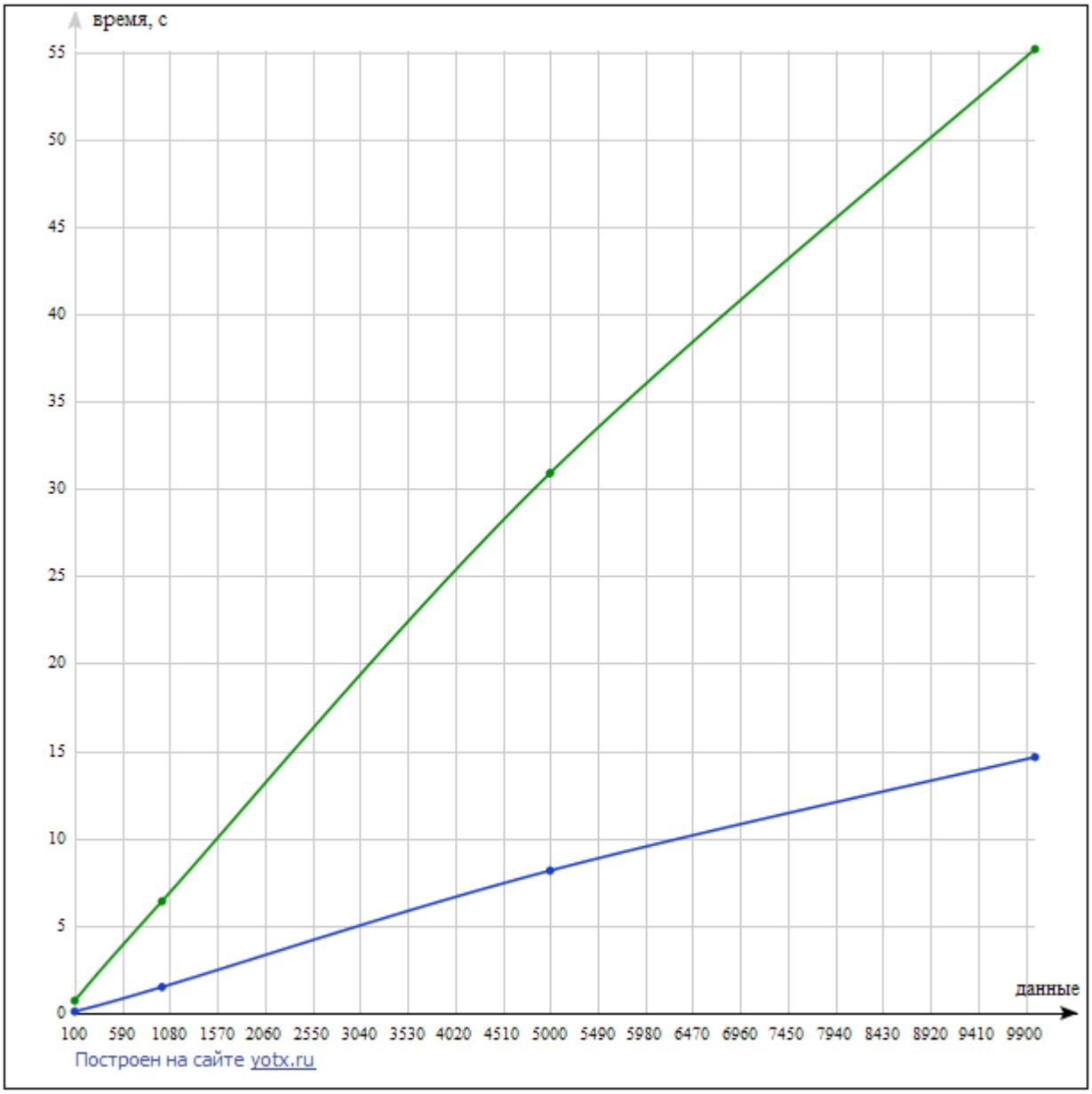
Тепловая карта алгоритма A_Star:



По тепловой карте алгоритма A* видно, что поиск происходил более сконцентрировано и оптимально. Большая часть вершин карты была справедливо проигнорирована. Также, можно заметить, что переходы между цветами менее плавные, что говорит о более быстром поиске — число вершин одного и того же цвета меньше, чем в случае алгоритма Дейкстры.

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

```
#include <iostream>
#include <cstdlib>
#include <vector>
#include <string>
#include <queue>
#include <cmath>
#include <ctime>

#define PASS 1
#define UNPASS 0

typedef struct Cell{
    int id;
    int h;
    int g;
    int f;
    int x, y;
    bool open;
    bool close;
    struct Cell * parent_cell;
} Cell;

struct GreaterThanByF{
  bool operator()(const Cell* left, const Cell* right) const{
    if(left-> f == right-> f)
        return (left-> g > right-> g);
    return (left-> f > right-> f);
  }
};

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){
    if(Grill[x1][y1].id == PASS && Grill[x1][y1].close == false){
        if(Grill[x1][y1].open == false){
            Grill[x1][y1].open = true;
            Grill[x1][y1].parent_cell = current;
            Grill[x1][y1].g = current-> g + 1;
            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
          *>& Changed){
53        int x1 = Num[0], y1 = Num[1], x2 = Num[2], y2 = Num[3];
54        int path = 0;
55
56        if( Grill[x1][y1].id == UNPASS || Grill[x2][y2].id == UNPASS || x1 > (int)Grill.size() || x2 > (
              int)Grill.size() || y1 > (int)Grill[0].size() || y2 > (int)Grill[0].size() || x1 < 0 || x2 < 0 ||
              y1 < 0 || y2 < 0)
57            return -1;
58
59        std::priority_queue<Cell*, std::vector<Cell*>, GreaterThanByF> OpenQueue;
60        std::queue<Cell*> CloseQueue;
61        Cell * current = NULL;
62
63        current = &Grill[x1][y1];
64        OpenQueue.push(current);
65        current-> open = true;
66        Changed.push_back(&Grill[x1][y1]);
67
68        while(Grill[x2][y2].open == false){
69            if(OpenQueue.empty())
70                return -1;
71            CloseQueue.push(current);
72            current-> close = true;
73            OpenQueue.pop();
74
75            if(current-> x - 1 >= 0)
76                A_Help_Search(Grill, Changed, OpenQueue, current, current-> x - 1, current-> y,
                      x2, y2);
77            if(current-> x + 1 < Grill.size())
```

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

```
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"
        << std::endl;
159     return 0;
160 }
```

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.