

Московский авиационный институт  
(национальный исследовательский университет)

Факультет информационных технологий и прикладной  
математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу "Дискретный анализ" на тему  
"Эвристический поиск в графе"

Студент: А. О. Тояков  
Преподаватель: А. Н. Ридли  
Группа: М8О-307Б-18  
Дата:  
Оценка:  
Подпись:

Москва  
2021

## Условие

Разработать программу на языке C или C++, реализующую указанный алгоритм согласно заданию.

Задана карта, состоящая из  $n \cdot m$  клеток. Необходимо найти путь из вершины с номером `start` в вершину с номером `finish` при помощи алгоритма  $A^*$ . Также на карте возможно наличие непроходимых точек.

## Метод решения

Данное задание предполагает реализацию алгоритма  $A^*$  по нахождению пути.

1. Добавить стартовую клетку в открытый список (при этом её значения  $G$ ,  $H$  и  $F$  равны 0).
2. Повторять следующие шаги:  
Ищем в открытом списке клетку с наименьшим значением величины  $F$ , делаем её текущей.  
Удаляем текущую клетку из открытого списка и помещаем в закрытый список.  
Для каждой из соседних к текущей точке клеток:  
Если клетка непроходима или находится в закрытом списке, игнорируем её.  
Если клетка не в открытом списке, то добавляем её в открытый список, при этом рассчитываем для неё значения  $G$ ,  $H$  и  $F$ , и также устанавливаем ссылку родителя на текущую клетку.  
Если клетка находится в открытом списке, то сравниваем её значение  $G$  со значением  $G$  таким, что если бы к ней пришли через соседнюю клетку. Если сохранённое в проверяемой клетке значение  $G$  больше нового, то меняем её значение  $G$  на новое, пересчитываем её значение  $F$  и изменяем указатель на родителя так, чтобы она указывала на текущую клетку.  
Останавливаемся, если:  
В открытый список добавили целевую клетку (в этом случае путь найден). Открытый список пуст (в этом случае к целевой клетке пути не существует).
3. Сохраняем путь, двигаясь назад от целевой точки, проходя по указателям на родителей до тех пор, пока не дойдём до стартовой клетки.

## Описание программы

Мой проект состоит из трёх файлов: `main.cpp`, `map.hpp`, `search.hpp`. Поиск происходит на карте с закрытыми точками. В качестве эвристики было выбрано манхэттенское расстояние, а перемещение происходит по 4 направлениям.

`map.hpp`: класс `map` представляет собой карту и содержит один основной метод: `ConstructSymbolsMap(unsigned int NoPointAmount)`. В качестве параметра указывается количество добавляемых непроходимых точек, которые необходимо ввести. Непроходимые

точки добавляются в закрытый список. Соответственно карта представляет собой двумерный массив, содержащий информацию о точке: вес, значение  $F = \text{вес} + \text{эвристика}$ , координаты родителя.

search.hpp: данный класс представляет собой реализацию эвристического поиска. Открытый список реализован с помощью приоритетной очереди, отсортированной по возрастанию. Также используется дополнительная структура для хранения актуального списка точек в открытом списке. Данный класс также содержит объект map. Соответственно соседние точки от текущей добавляются в вектор. После данные точки анализируются согласно алгоритму. Ответ формируется в виде карты, где посещённые точки отмечаются восклицательным знаком. Например:

путь (4 направления и эвристика - манхэттенское расстояние)

стартовая точка: (2,2), конечная точка: (9,9)

```
# # # # # # # # # #
# 0 ! ! ! 0 0 0 0 0 #
# 0 ! ! ! 0 0 0 0 0 #
# 0 # 0 ! 0 0 0 0 0 #
# 0 0 0 ! 0 0 0 0 0 #
# 0 0 0 ! 0 0 0 0 0 #
# 0 # 0 ! 0 0 0 # 0 #
# 0 # 0 ! 0 # # 0 0 #
# 0 0 # ! ! ! 0 # # #
# 0 0 0 # # ! ! ! ! #
# # # # # # # # # #
```

Пояснения: закрытый список - список рассмотренных вершин. Открытый список - список вершин, которые необходимо рассмотреть.

## Исходный код

```
// main.cpp

#include "source/map.hpp"
#include "source/search.hpp"

int main() {
    unsigned int n,m;
    std::cin >> n >> m;
    Map* testMap = new Map(n,m);
    unsigned int amount;
    std::cin >> amount;
    testMap->ConstructSymbolsMap(amount);
}
```

```

    Search* testSearch = new Search(testMap);
    std::cout << "Start(x,y) && Finish(x,y)" << std::endl;
    unsigned int x,y,x1,y1;
    std::cin >> x >> y >> x1 >> y1;
    auto start = std::make_pair(x,y);
    auto end = std::make_pair(x1,y1);
    testSearch->Result(start, end);
    testMap->Print();
    delete testMap;
    delete testSearch;
    return 0;
}

//map.hpp

#ifndef MAP_HPP
#define MAP_HPP
#include <iostream>
#include <vector>
#include <cmath>
#include <set>
#include <queue>

struct Information {
    std::string symbol; // 0 - true, # - false
    unsigned int allScore; // weight + heuristic(finish);
    unsigned int weight;
    std::pair<unsigned int, unsigned int> from; // parent_coordinate

    Information() {
        symbol = "0";
        allScore = 0;
        weight = 0;
        from = std::make_pair(0,0);
    }
};

class Map {
private:
    unsigned int lines;
    unsigned int columns;

```

```

std::vector<std::vector<Information>> symbolsMap;
std::set<std::pair<unsigned int, unsigned int>> noPoints; //new # in symbolsMap

public:
    friend class Search;

    Map(unsigned int lines, unsigned int columns) {
        this->lines = lines + 2;
        this->columns = columns + 2;
        symbolsMap.resize(this->lines);
    }

    void Print() {
        for(auto i = 0; i < lines;i++) {
            for(auto j = 0; j < columns;j++) {
                std::cout << symbolsMap[i][j].symbol << " ";
            }
            std::cout << std::endl;
        }
    }

    bool CheckNewPoint(int x, int y) {
        if(x > lines - 1 || y > columns - 1 || x <= 0 || y <= 0) {
            return false;
        }
        return true;
    }

    void ConstructSymbolsMap(unsigned int NoPointAmount) {
        std::string status;
        int temp = 0;
        for(auto i = 0; i < lines; i++) {
            for(auto j = 0; j < columns; j++) {
                symbolsMap[i].push_back(Information());
            }
        }
        for(auto j = 0; j < columns;j++) {
            symbolsMap[0][j].symbol = '#';
            symbolsMap[lines - 1][j].symbol = '#';
        }
    }

```

```

        noPoints.insert(std::make_pair(0, j));
        noPoints.insert(std::make_pair(lines - 1, j));
    }
    for(auto i = 0; i < lines;i++) {
        symbolsMap[i][0].symbol = '#';
        symbolsMap[i][columns - 1].symbol = '#';
        noPoints.insert(std::make_pair(i, 0));
        noPoints.insert(std::make_pair(i, columns - 1));
    }
    // add new #(noPoint)
    while(temp < NoPointAmount) {
        unsigned int pointX, pointY;
        std::cin >> pointX >> pointY;
        if(CheckNewPoint(pointX, pointY)) {
            symbolsMap[pointX][pointY].symbol = '#';
            noPoints.insert(std::make_pair(pointX, pointY));
            std::cout << "ADD" << std::endl;
        }
        temp++;
    }
    /*
    Example:
        map(3,3) and noPoint{(2,2)}
        # # # # #
        # 0 0 0 #
        # 0 # 0 #
        # 0 0 0 #
        # # # # #
        0 - =>
        Information() {
            symbol = "0";
            allScore = 0;
            weight = 0;
            from = std::make_pair(0,0);
            # -> # coordinate in "closedList" noPoints
        }
    */
    Print();
}
};

```

```

#endif

//seacrh.hpp

#ifndef SEARCH_HPP
#define SEARCH_HPP
#include "map.hpp"
#include <map>
#include <algorithm>
#define const_weight 5
using point = std::pair<unsigned int, unsigned int>;
using pointAndScore = std::pair<point, unsigned int>;

struct MyCompare {
    constexpr bool operator()(std::pair<point, unsigned int> const & a, std::pair<point,
        return a.second >= b.second;
    }
};

class Search {
public:
    Search(Map* map) {
        this->map = map;
    }

    void Result(point start, point finish) {
        if(map->CheckNewPoint(start.first,start.second) == true
            && map->CheckNewPoint(finish.first, finish.second) == true
            && AStar(start,finish) == true) {
            std::cout << "YES" << std::endl;
            ChangeSymbolMapAndPrintResult(start,finish);
        } else {
            std::cout << "ERROR" << std::endl;
        }
    }

private:
    Map* map;
    // открытый список
    std::priority_queue <pointAndScore,
        std::vector <pointAndScore>,MyCompare> openQueue;
    std::set<point> pointsInOpenList;

```

```

void ChangeSymbolMapAndPrintResult(point start, point finish) {
    point current = finish;
    int i = 1;
    while(current != start) {
        //map->symbolsMap[current.first][current.second].symbol = std::to_string(i);
        map->symbolsMap[current.first][current.second].symbol = '!';
        current = map->symbolsMap[current.first][current.second].from;
        i++;
    }
    //map->symbolsMap[start.first][start.second].symbol = std::to_string(i);
    map->symbolsMap[start.first][start.second].symbol = '!';
}

```

```

bool FindElement(std::set<point>& pointsInOpenList, point& old) {
    auto it = pointsInOpenList.find(old);
    if(it == pointsInOpenList.end()) {
        return false;
    }
    return true;
}

```

```

void Erase(std::set<point>& pointsInOpenList, point& old) {
    for(auto it = pointsInOpenList.begin(); it != pointsInOpenList.end(); it++) {
        if((*it) == old) {
            pointsInOpenList.erase(it);
            return;
        }
    }
}

```

```

unsigned int Heuristik(point start, point finish) {
    //return (std::max(abs(start.first - finish.first),abs(start.second - finish.second)));
    //return (sqrt((pow((start.first - finish.first),2) + pow((start.second - finish.second),2))));
    return ((abs(start.first - finish.first) + abs(start.second - finish.second)) * 10);
}

```

```

void GetFourVertex(std::vector<point>& adjacentVertex, point& current) {

```



```

    int i_const, j_const;
    i_const = current.first;
    j_const = current.second;
    int i_f, j_f, i_s, j_s;
    j_f = current.second - 1;
    j_s = current.second + 1;
    adjacentVertex.push_back({i_const, j_f});
    adjacentVertex.push_back({i_const, j_s});
    i_f = i_const - 1;
    i_s = i_const + 1;
    adjacentVertex.push_back({i_f, j_const});
    adjacentVertex.push_back({i_s, j_const});
}

void GetAllVertex(std::vector<point>& adjacentVertex, point& current) {
    for(auto j = current.second - 1; j <= current.second + 1; j++) {
        if(j == current.second) {
            unsigned int first = j - 1;
            unsigned int second = j + 1;
            adjacentVertex.push_back({current.first, first});
            adjacentVertex.push_back({current.first, second});
        }
        unsigned int high, low;
        high = current.first + 1;
        low = current.first - 1;
        adjacentVertex.push_back({low, j});
        adjacentVertex.push_back({high, j});
    }
}

bool AStar(point start, point finish) {
    openQueue.push({start, 0});
    pointsInOpenList.insert(start);
    while(openQueue.size() > 0) {
        pointAndScore current = openQueue.top();
        if(current.first == finish) {
            return true;
        }
        openQueue.pop();
        Erase(pointsInOpenList, current.first);
    }
}

```

```

map->noPoints.insert(current.first);
std::vector<point> adjacentVertex;
unsigned int tentativeScore =
    map->symbolsMap[current.first.first][current.first.second].weight + cons
GetFourVertex(adjacentVertex,current.first);
//GetAllVertex(adjacentVertex,current.first);
for(size_t j = 0;j < adjacentVertex.size(); j++) {
    //Если клетка непроходима или находится в закрытом списке, игнорируем её
    if(map->noPoints.find(adjacentVertex[j]) != map->noPoints.end()) {
        continue;
    }
    if(FindElement(pointsInOpenList,adjacentVertex[j])) {
        if(tentativeScore < map->symbolsMap[adjacentVertex[j].first][adjacent
            map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second
            map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second
            map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second
        }
    } else {
        map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].f
        map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].w
        map->symbolsMap[adjacentVertex[j].first][adjacentVertex[j].second].a
        //std::cout << adjacentVertex[j].first << " " << adjacentVertex[j].s
        openQueue.push({adjacentVertex[j],map->symbolsMap[adjacentVertex[j].
        pointsInOpenList.insert({adjacentVertex[j].first,adjacentVertex[j].s
    }
}
}
return false;
}
};

#endif

```

## Недочёты

Использование менее оптимальных структур данных для хранения информации.

Использование дополнительного множества для хранения актуального списка точек открытого списка.

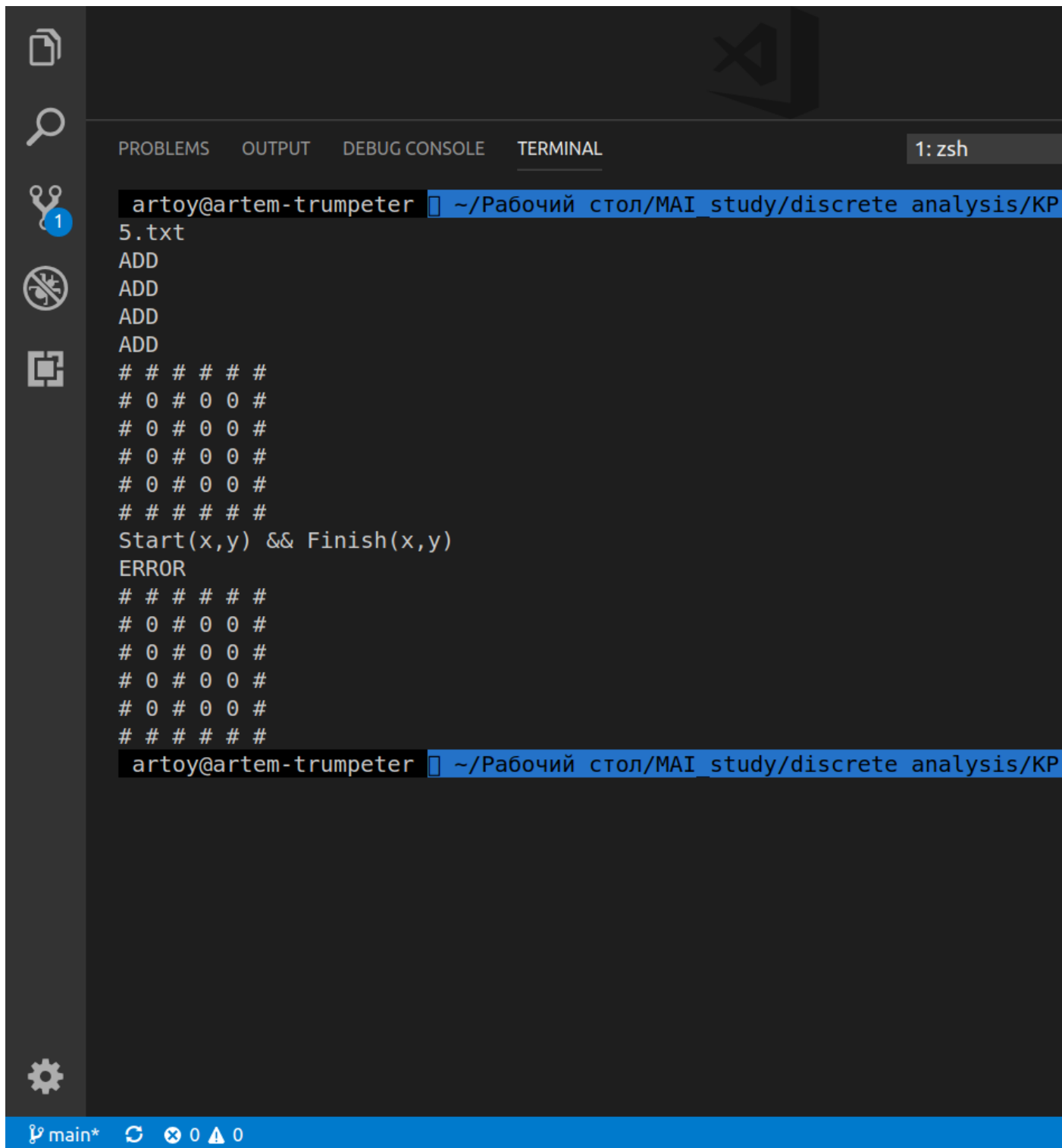
## Пример работы программы

Для проверки результата работы программы было составлено 5 тестов вручую и написан test-gen, генерирующий рандомные корректные входные данные.

```
test_gen.cpp x
1  #include <iostream>
2
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
artoy@artem-trumpeter ~/Рабочий стол/MAI study/discrete analysis/KP [main] ./a.out < t1.txt
# # # # #
# 0 0 0 0 0 #
# 0 0 0 0 0 #
# 0 0 0 0 0 #
# 0 0 0 0 0 #
# 0 0 0 0 0 #
# # # # #
Start(x,y) && Finish(x,y)
YES
# # # # #
# ! 0 0 0 0 #
# ! 0 0 0 0 #
# ! 0 0 0 0 #
# ! ! ! 0 #
# # # # #
artoy@artem-trumpeter ~/Рабочий стол/MAI study/discrete analysis/KP [main] ./a.out < t2.txt
ADD
ADD
ADD
ADD
ADD
# # # # #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# # # # #
Start(x,y) && Finish(x,y)
YES
# # # # #
# 0 ! ! ! 0 0 0 0 #
# ! ! 0 # ! 0 0 0 0 #
# 0 0 0 # ! 0 0 0 0 #
# 0 0 0 # ! 0 0 0 0 #
# 0 0 0 # ! ! ! ! #
# 0 0 0 # 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# 0 0 0 0 0 0 0 0 #
# # # # #
artoy@artem-trumpeter ~/Рабочий стол/MAI study/discrete analysis/KP [main]
```







The image shows a Visual Studio Code editor window with a terminal pane open. The terminal shows the execution of a script named 5.txt. The script contains several 'ADD' commands, a grid of numbers, and a function call 'Start(x,y) && Finish(x,y)'. The output shows an 'ERROR' message. The terminal prompt is 'artoy@artem-trumpeter' and the current directory is '~/Рабочий стол/MAI\_study/discrete analysis/KP'. The status bar at the bottom indicates 'main\*' and shows 0 errors and 0 warnings.

```
artoy@artem-trumpeter ~/Рабочий стол/MAI_study/discrete analysis/KP
5.txt
ADD
ADD
ADD
ADD
# # # # # #
# 0 # 0 0 #
# 0 # 0 0 #
# 0 # 0 0 #
# 0 # 0 0 #
# 0 # 0 0 #
# # # # # #
Start(x,y) && Finish(x,y)
ERROR
# # # # # #
# 0 # 0 0 #
# 0 # 0 0 #
# 0 # 0 0 #
# 0 # 0 0 #
# 0 # 0 0 #
# # # # # #
artoy@artem-trumpeter ~/Рабочий стол/MAI_study/discrete analysis/KP
```

main\* 0 0

## Выводы

Основное применение данного алгоритма это нахождения минимальных путей в графе. От жадного алгоритма, который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь. Таким образом алгоритм  $A^*$  в некотором роде является наследником алгоритма Дейкстры.  $A^*$  использует эвристику, с помощью которой можно отсекаать некоторые неоптимальные пути. Поведение алгоритма сильно зависит от того, какая эвристика используется. В свою очередь, выбор эвристики зависит от постановки задачи. Манхэттенское расстояние используется, когда мы можем перемещаться в четырёх направлениях. Также существуют эвристики для случаев, когда к перемещению добавляются диагонали (расстояние Чебышева) или передвижение вообще не ограничено сеткой (евклидово расстояние). В худшем случае, число вершин исследуемых алгоритмом растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда оценка  $h(x)$  не растёт быстрее, чем логарифм оптимальной эвристики.

Этот алгоритм также можно улучшить некоторыми способами в виду того, что мы используем очень много памяти для хранения и выполняем некоторые необязательные действия. Во-первых, конечно, нужно подбирать оптимальную эвристику под каждую конкретную задачу. Во-вторых можно использовать такие улучшения, как ограничение объёма памяти или итеративный поиск в глубину. Это обеспечит нам быстроедействие алгоритма, однако, таким образом, мы можем не найти оптимальный путь при слишком строгих ограничениях. В-третьих, нужно подбирать оптимальные структуры данных для хранения информации, например красно-чёрное дерево и булеву хэш таблицу для закрытого списка.

В целом применять этот алгоритм мы можем во многих жизненных задачах. Например, поиск оптимального пути к нужной точке на карте. Также  $A^*$  широко используется в играх. В жанре Tower Defense мы можем показывать вражескому AI кратчайший путь до наших укреплений. Помимо этого  $A^*$  использовался в очень популярной игре Warcraft 3 для тех же целей. В итоге этот алгоритм будет эффективен в тех случаях, когда нам известна конечная точка.