

Санкт-Петербургский политехнический университет Петра Великого  
Институт машиностроения, материалов и транспорта  
Высшая школа автоматизации и робототехники

## КУРСОВАЯ РАБОТА

### Алгоритм ALT

по дисциплине «Объектно-ориентированное программирование»

Выполнил  
студент гр. 3331506/70401

<подпись>

А.А. Ларионов

Руководитель  
преподаватель

<подпись>

М.С. Ананьевский

Санкт-Петербург

2020

## СОДЕРЖАНИЕ

Введение.....	3
1. Описание алгоритма.....	4
2. Программная реализация алгоритма.....	8
3. Анализ алгоритма.....	12
Заключение.....	15
Список использованных источников.....	16
Приложение 1: Программный код алгоритма.....	17

## ВВЕДЕНИЕ

Данная курсовая работа посвящена алгоритму поиска на графе маршрута с наименьшей стоимостью (кратчайший путь/расстояние) от начальной вершины к конечной – алгоритму *ALT*.

Данная задача берет свое начало в середине XX века, на сегодня известно множество алгоритмов ее решения. Наиболее известны из них:

- алгоритм Дейкстры;
- алгоритм  $A^*$  (*A star*).

Первый алгоритм находит кратчайшее расстояние от одной вершины графа до всех остальных. Второй – от одной вершины до другой, осуществляя поиск по первому наилучшему совпадению.

Алгоритм *ALT* был изобретен в начале XXI века специалистами Эндрю В. Голдбергом (*Andrew V. Goldberg*) и Крисом Харрельсоном (*Chris Harrelson*). Он является логическим продолжением и усовершенствованием упомянутых алгоритмов.

Задача о кратчайшем пути находит широкое применение на практике, т.к. составляющие графа – вершины и ребра – могут играть различные роли. Для дорожной навигации вершинами могут быть перекрестки дорог, а ребрами – связующие их дороги. Тогда задачей становится минимизация суммарной длины пройденных дорог между заданными перекрестками. Ребра можно также связать с временем, расходами и другими характеристиками. Минимизация этих параметров важна в географии, экономике и во многих других областях.

## 1. ОПИСАНИЕ АЛГОРИТМА

Задача поиска кратчайшего расстояния между вершинами графа может быть определена для неориентированного, ориентированного и смешанного графа. Граф называется ориентированным, если его ребра имеют направление. В дальнейшем будем рассматривать неориентированный граф.

Граф  $G(V, E)$  – совокупность множества вершин  $V$  и ребер  $E$ . Две вершины графа, соединенные ребром, называются смежными.

Пусть даны две смежные вершины графа  $v_i$  и  $v_j$ . Тогда соединяющее их ребро обозначим как  $e_{i,j}$ . Этому ребру соответствует функция  $f$ , определяющая вес этого ребра, выраженный действительным числом. Путь в графе будет представлять собой последовательность вершин  $P = (v_1, v_2, \dots, v_n)$ , где  $v_1$  – начальная вершина,  $v_n$  – конечная вершина. Причем  $v_i$  смежна с  $v_{i+1}$  для  $1 \leq i \leq n$ . Кратчайшим же путем из вершины  $v_1$  в вершину  $v_n$  будет такой путь  $P$ , который имеет минимальное значение суммы  $\sum_{i=1}^{n-1} f(e_{i,i+1})$ .

**Алгоритм Дейкстры.** Алгоритм работает только для графов без ребер с отрицательным весом. Пусть дана задача найти кратчайшие пути в графе от некоторой вершины  $s$  до всех остальных. Введем обозначения:  $m$  – метка, т.е. минимальное известное расстояние от  $s$  до произвольной вершины; расстояние от произвольной вершины до смежной с ней  $d(v_i, v_{i+1}) = m_i + f(e_{i,i+1})$ , где  $m_i$  – значение метки произвольной вершины;  $Q$  – множество непосещенных вершин;  $U$  – множество посещенных вершин. Принцип работы алгоритма заключается в следующем:

- 1) каждой вершине из  $V$  сопоставляется  $m$  равная бесконечности, метка вершины  $s$  равна 0;
- 2) все вершины принадлежат  $Q$ ;
- 3) из  $Q$  выбирается вершина с наименьшим значением  $m$  и рассматриваются все смежные с ней вершины;
- 4) рассчитывается  $d(v_i, v_{i+1})$  для каждой смежной вершины, не принадлежащей  $U$ ;

- 5) если  $d(v_i, v_{i+1}) < m_{i+1}$ , то  $m_{i+1} = d(v_i, v_{i+1})$ ;
- 6) выбранная вершина включается в  $U$  и исключается из  $Q$ ;
- 7) повторяем пункты 3-6, пока  $Q$  – не пустое.

В итоге будут просмотрены все вершины. К каждой вершине будет определен кратчайший путь  $P$  от  $s$ , метки вершин будут содержать минимальное значение суммы  $\sum_i f(e_{i,i+1})$  соответственно.

**Алгоритм  $A^*$ .** Представляет собой расширение алгоритма Дейкстры, в нем удалось достичь повышения производительности по времени за счет применения эвристики. Переход делается в ту смежную вершину, предположительный путь из которой до конечной вершины будет меньше. Поиск по первому наилучшему совпадению – подход, в котором следующая вершина выбирается на основе оценочной функции  $f(v)$ . Выбирается вершина с наименьшим значением этой функции.

Оценочная функция определяется следующим образом

$$f(v) = g(v) + h(v),$$

где  $g(v)$  – расстояние уже пройденного пути от начальной вершины до  $v$ ;  $h(v)$  – эвристическая функция, т.е. оценка кратчайшего расстояния от  $v$  до конечной вершины.

Если  $v$  – конечная вершина, то  $h(v) = 0$ .

Для оптимальности алгоритма выбранная функция  $h(v)$  должна быть допустимой, т.е. она никогда не переоценивает фактическое расстояние до конечной вершины (нижняя оценка).

При перемещении по поверхности, покрытой координатной сеткой, в качестве эвристической функции обычно используют манхэттенское расстояние  $h(v) = |x_v - x_t| + |y_v - y_t|$ , где  $x$  и  $y$  – координаты вершины;  $t$  – конечная вершина. Если передвижение не ограничено сеткой, то можно применять евклидово расстояние  $h(v) = \sqrt{(x_v - x_t)^2 + (y_v - y_t)^2}$ .

Пусть дана задача найти кратчайший путь в графе от вершины  $s$  до вершины  $t$ . Принцип работы будет отличаться от алгоритма Дейкстры дополнительным расчетом оценочной функции  $f(v)$  для вершин, где  $g(v_i) =$

$d(v_{i-1}, v_i)$ , а выбор вершины будет осуществляться по минимальному значению оценочной функции. При этом будут просмотрены только вершины на пути к конечной.

**Алгоритм *ALT*.** Аббревиатура *ALT* – алгоритм  $A^*$ , ориентиры (*landmarks*) и неравенство треугольника (*triangle inequality*). Ориентиры и неравенство треугольника используется для расчета эвристической функции.

Алгоритм – двухэтапный. Состоит из препроцессинга и поиска. Препроцессинг запускается однократно, занимает достаточно долгое время и рассчитывает вспомогательную информацию. В случае алгоритма *ALT* выбирается небольшое количество вершин-ориентиров, далее для каждой вершины графа вычисляет расстояния от и до каждого ориентира. Этап поиска может использовать полученную на этапе препроцессинга информацию и ищет кратчайшее расстояние между двумя вершинами за наименьшее время. Алгоритм *ALT* на этапе поиска использует алгоритм  $A^*$ , переход делается в ту вершину, которая находится на кратчайшем расстоянии между начальной вершиной и выбранным ориентиром.

В качестве допустимой эвристической функции используется неравенство треугольника (см. рисунок 1.1). Введем обозначения:  $L \subset V$  – множество ориентиров;  $l \in L$  – ориентир;  $dist(v, w)$  – оценка расстояния между некоторыми вершинами  $v$  и  $w$ . Пусть дана задача найти кратчайший путь в графе от вершины  $s$  до вершины  $t$ . Тогда для эвристической функции получим два выражения

$$dist(v_{i-1}, v_i) \geq dist(l, v_i) - dist(l, v_{i-1}),$$

$$dist(v_{i-1}, v_i) \geq dist(v_{i-1}, l) - dist(v_i, l);$$

$$h_t^{l-}(v) = dist(l, t) - dist(l, v),$$

$$h_t^{l+}(v) = dist(v, l) - dist(t, l).$$

Расстояние от и до ориентиров уже подсчитаны на этапе препроцессинга. В качестве эвристической функции используем максимум из двух выражений по всем  $l \in L$

$$h_t^l(v) = \max_{l \in L} \{h_t^{l+}(v), h_t^{l-}(v)\}.$$

Видно, что эвристическая функция будет наиболее эффективна, если существуют ориентиры, находящиеся до начальной вершины или за конечной.

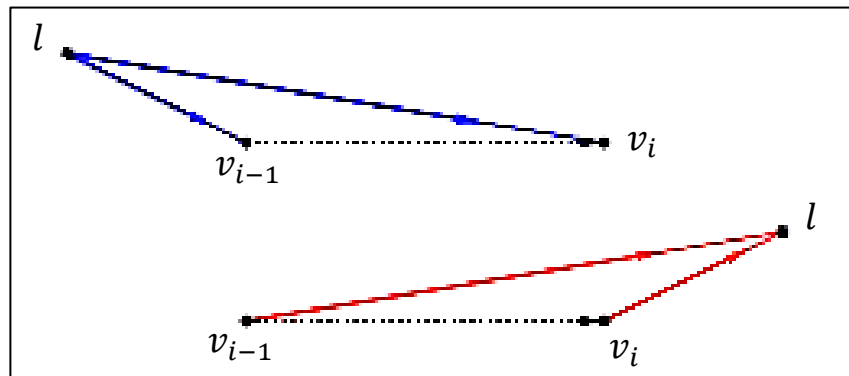


Рисунок 1.1 — Неравенство треугольника

Выбор ориентиров достаточно сложен, т.к. выбранные в качестве них вершины должны быть «неплохими» для всех возможных случаев поиска кратчайшего расстояния. Существуют несколько способов их выбора:

1. случайный выбор (*random*);
2. выбор на плоскости (*planar*);
3. избирательный выбор (*avoid*);
4. максимальное покрытие (*maxcover*).

## 2. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ АЛГОРИТМА

В качестве графа выберем квадратную координатную сетку  $9 \times 9$ , где значению 1 соответствует вершина, в которую можно перейти; значению 0 – недоступная для перехода вершина; нижнему индексу – порядковый номер вершины (см. рисунок 2.1).

		<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>
<b>0</b>	$1_{00}^0$	$1_{01}$	$1_{02}$	$1_{03}$	$1_{04}$	$1_{05}$	$1_{06}$	$1_{07}$	$1_{08}^1$	
<b>1</b>	$1_{09}$	$1_{10}$	$1_{11}$	$1_{12}$	$1_{13}$	$0_{14}$	$0_{15}$	$0_{16}$	$0_{17}$	
<b>2</b>	$1_{18}$	$1_{19}$	$0_{20}$	$1_{21}$	$1_{22}$	$1_{23}$	$1_{24}$	$1_{25}$	$1_{26}$	
<b>3</b>	$1_{27}$	$1_{28}$	$0_{29}$	$1_{30}$	$1_{31}$	$1_{32}$	$1_{33}$	$1_{34}$	$1_{35}$	
<b>4</b>	$1_{36}$	$1_{37}$	$0_{38}$	$1_{39}$	$1_{40}$	$1_{41}$	$1_{42}$	$1_{43}$	$1_{44}$	
<b>5</b>	$1_{45}$	$1_{46}$	$1_{47}$	$1_{48}$	$1_{49}$	$1_{50}$	$1_{51}$	$1_{52}$	$1_{53}$	
<b>6</b>	$1_{54}$	$1_{55}$	$1_{56}$	$1_{57}$	$0_{58}$	$1_{59}$	$1_{60}$	$1_{61}$	$1_{62}$	
<b>7</b>	$1_{63}$	$1_{64}$	$1_{65}$	$1_{66}$	$0_{67}$	$1_{68}$	$1_{69}$	$1_{70}$	$1_{71}$	
<b>8</b>	$1_{72}^2$	$1_{73}$	$1_{74}$	$1_{75}$	$0_{76}$	$1_{77}$	$1_{78}$	$1_{79}$	$1_{80}^3$	

Рисунок 2.1 — Координатная сетка

Ребра графа не показаны, вес ребер равен 1. Перемещение из каждой вершины ограничивается сеткой и возможно вверх, вниз, влево и вправо

Так как представленный граф имеет правильную форму, то воспользуемся способом выбора ориентиров на плоскости. Разделим сетку на 4 сектора одинаковой площади и возьмем наиболее удаленную от центра сетки точку в каждом из них. Это и будут ориентиры. Обозначим их с помощью верхнего индекса, значение которого равно их порядковому номеру (см. рисунок 2.1)

На этапе препроцессинга подсчитываются и сохраняются кратчайшие расстояния от каждой вершины графа до каждого ориентира. Для подсчета используется алгоритм Дейкстры. На этапе поиска применяется алгоритм  $A^*$ , эвристическая функция которого рассчитывается как  $h_t^L(v)$ .

Программный код реализованного алгоритма *ALT* написан на языке *C++* (см. приложение 1). Рассмотрим ключевые особенности и полученные



результаты решения задачи поиска кратчайшего пути между двумя произвольными вершинами на координатной сетке.

Ключевой составляющей алгоритма является класс – вершина (см. листинг 2.1).

#### Листинг 2.1 — Вершина

---

```
class Node
{
public:
    // Возможность прохода -> 1 - да; 0 - нет
    int passage;
    // Координаты вершины
    int x = 0;
    int y = 0;
    // Оценочная функция -> f = g + h
    int f = 0;
    // Расстояние от начальной вершины
    int g = 0;
    // Эвристическая функция
    int h = 0;
    // Вершина, из которой произошел переход
    Node* parent = nullptr;
    // Расстояния до ориентиров (количество - 4)
    // рассчитывается на этапе препроцессинга
    int l[4] = { 0 };
public:
    Node(const int x, const int y, const int passage)
    {
        this->x = x;
        this->y = y;
        this->passage = passage;
    };
    Node() {};
};
```

---

В ходе препроцессинга создаются вершины и заполняется массив l[.]. Результат выполнения этапа посмотрим на примере значения l[1] каждой вершины, т.е. кратчайшего расстояния до первого ориентира (символом # заменено значение 10000, что обозначает недоступную вершину)

8	7	6	5	4	3	2	1	0
9	8	7	6	5	#	#	#	#
10	9	#	7	6	7	8	9	10
11	10	#	8	7	8	9	10	11
12	11	#	9	8	9	10	11	12.
13	12	11	10	9	10	11	12	13
14	13	12	11	#	11	12	13	14
15	14	13	12	#	12	13	14	15
16	15	14	13	#	13	14	15	16

Хорошо видно, что недоступные вершины служат серьезной преградой и значительно повышают итоговый вес пути.

Также важной особенностью данной реализации алгоритма является повторение этапа препроцессинга при каждом поиске кратчайшего расстояния. Это было сделано из соображений удобства использования – вызова всего одной функции. Для алгоритма *ALT* этот этап должен вызываться всего один раз для каждого графа, что легко осуществить выносом соответствующей функции и ее отдельным использованием.

Для примера найдем с помощью алгоритма кратчайшее расстояние от вершины с порядковым номер 60 ( $x = 6, y = 6$ ) до вершины с порядковым номером 8 ( $x = 8, y = 0$ ). В результате получим координаты вершин, которые нужно пройти, причем с конца

$$(8; 0) \leftarrow (7; 0) \leftarrow (6; 0) \leftarrow (5; 0) \leftarrow (4; 0) \leftarrow (4; 1) \leftarrow (4; 2) \leftarrow (5; 2) \leftarrow (6; 2) \leftarrow (6; 3) \leftarrow (6; 4) \leftarrow (6; 5) \leftarrow (6; 6) \text{ Start.}$$

Легко заметить, что пройденное расстояние является кратчайшим, однако это не единственный возможный вариант.

Также интерес представляет количество просмотренных вершин. Для этого приведем значение  $g$  всех вершин после окончания поиска (символом # заменено значение 10000, что обозначает вершину, которая не рассматривалась)

#	#	#	9	8	9	10	11	12
#	#	#	8	7	#	#	#	#
#	#	#	7	6	5	4	5	#
#	#	#	6	5	4	3	4	#
#	#	#	5	4	3	2	3	#.
#	#	#	4	3	2	1	2	#
#	#	#	#	#	1	0	1	#
#	#	#	#	#	2	1	#	#
#	#	#	#	#	#	#	#	#

Алгоритм изучает всех соседей вершины, в которую переходит. Таким образом, всего просмотренных вершин, с учетом пройденных и исключением возможных общих соседей, должно быть 27. В результате же получили число

33. Это произошло ввиду существования нескольких вариантов кратчайших расстояний до конечной вершины.

### 3. АНАЛИЗ АЛГОРИТМА

Эффективность алгоритма *ALT* зависит как от количества и расстановки ориентиров, так и от размера и структуры графа. Поиск хороших ориентиров имеет решающее значение для общей эффективности алгоритма *ALT*.

Для поиска подходящей вершины для перехода во всех рассмотренных алгоритмах используется очередь с приоритетом. Самый простой вариант ее реализации заключается в простом поиске подходящей вершины во всем множестве (например, в массиве). Для ускорения процесса очередь часто создают на основе сортирующих деревьев.

Для оценки алгоритма воспользуемся некоторыми результатами исследований Эндрю В. Голдберга и Криса Харрельсона [1].

В работе для поиска ориентиров используется оптимизированный выбор на плоскости. Общее их количество равно 16. Очередь с приоритетом реализована на основе двоичной кучи. Выбраны графы с разным количеством вершин, представляющие собой реальные маршруты (см. таблица 3.1). Вес ребер графа – это функция от расстояния и времени. Начальная и конечная вершины выбирались случайным образом.

Таблица 3.1 — Графы

Условное имя графа	Количество вершин	Условное имя графа	Количество вершин
$M_1$	267 403	$M_7$	2 219 925
$M_2$	330 024	$M_8$	2 263 758
$M_3$	563 992	$M_9$	4 130 777
$M_4$	588 940	$M_{10}$	4 469 462
$M_5$	639 821	$M_{11}$	6 687 940
$M_6$	1 235 735		

В результатах (см. таблица 3.2) используются параметры эффективность (%) – количество просмотренных вершин, находящихся на кратчайшем

расстоянии, к общему количеству просмотренных вершин; среднее время работы алгоритма (мс). Время зависит от оборудования и реализации, но является достаточно важным для общего сравнения.

Таблица 3.2 — Результаты

Условное имя графа	Алгоритм Дейкстры	Алгоритм $A^*$ с евклидовым расстоянием	Алгоритм $ALT$
$M_1$	0,44 57,14	0,46 112,42	5,34 8,01
$M_2$	0,26 66,38	0,28 140,90	3,02 13,05
$M_3$	0,17 137,46	0,18 326,12	2,90 15,50
$M_4$	0,24 139,34	0,24 353,95	3,82 14,20
$M_5$	0,22 240,52	0,23 521,61	4,21 13,04
$M_6$	0,25 281,19	0,26 641,15	2,39 62,33
$M_7$	0,14 605,04	0,15 1252,61	3,13 40,67
$M_8$	0,15 579,59	0,16 1325,01	2,69 59,78
$M_9$	0,09 1208,30	0,10 2565,61	1,87 92,88
$M_{10}$	0,10 1249,86	0,10 2740,81	1,56 147,54
$M_{11}$	0,08 2113,80	0,08 4693,30	1,81 132,83

Алгоритм  $ALT$  показал отличные результаты по сравнению с алгоритмом Дейкстры и  $A^*$ . Это объяснимо – алгоритм Дейкстры не использует эвристику, а эвристика алгоритма  $A^*$  – евклидово расстояние – не всегда хорошо работает на используемых в работе графах. Однако не стоит забывать, что алгоритм  $ALT$  требует дополнительную память и время для расчета вспомогательных данных.

Представим полученные данные алгоритма *ALT* графически, как зависимость эффективности и времени выполнения от количества вершин графа (см. рисунок 3.1).

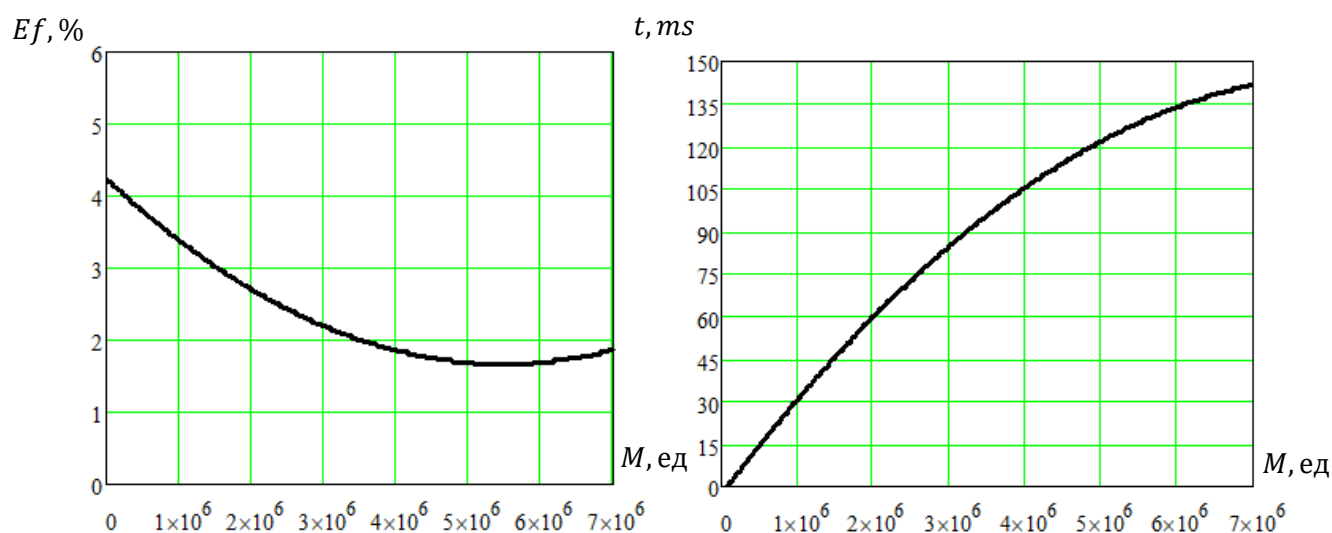


Рисунок 3.1 — Аппроксимация данных

Как можно заметить, их характер близок к линейному.

Также с увеличением количества ориентиров средняя эффективность алгоритма *ALT* повышается, что подробно описано в исследованиях.

## ЗАКЛЮЧЕНИЕ

В ходе работы были рассмотрены особенности алгоритма поиска кратчайшего расстояния – алгоритма *ALT*. Алгоритм был реализован на языке *C++*, с его помощью успешно решена задача поиска кратчайшего расстояния на координатной сетки. Проведен анализ алгоритма на основе результатов исследований его создателей Эндрю В. Голдберга и Криса Харрельсона, заключающийся в определении показателей – эффективности и времени выполнения.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A\* Search Meets Graph Theory. – Microsoft Research, 2004.*
2. *A. V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. – Microsoft Research, 2005.*
3. *Fabian Funch, Reinhard Bauer and Giacomo Nannicini. On Preprocessing the ALT-Algorithm. – Karlsruhe Institute of Technology, 2010.*



## ПРОГРАММНЫЙ КОД АЛГОРИТМА

---

```

#include <iostream>
#include <vector>

#define INFINITE 10000

class Node
{
public:
    // Возможность прохода -> 1 - да; 0 - нет
    int passage;
    // Координаты вершины
    int x = 0;
    int y = 0;
    // Оценочная функция -> f = g + h
    int f = 0;
    // Расстояние от начальной вершины
    int g = 0;
    // Эвристическая функция
    int h = 0;
    // Вершина, из которой произошел переход
    Node* parent = nullptr;
    // Расстояния до ориентиров (количество - 4)
    // рассчитывается на этапе препроцессинга
    int l[4] = { 0 };
public:
    Node(const int x, const int y, const int passage)
    {
        this->x = x;
        this->y = y;
        this->passage = passage;
    };
    Node() {};
};

Node* extract_min(std::vector<Node*>& setNode, const int numberLand)
{
    Node* buffer = nullptr;
    int index = 0;
    int minValue = INFINITE;

    for (size_t i = 0; i < setNode.size(); i++)
    {
        if (setNode[i]->l[numberLand] < minValue)
        {
            index = i;
            minValue = setNode[i]->l[numberLand];
        }
    }

    buffer = setNode[index];
    setNode.erase(setNode.begin() + index);
}

```

```

    return buffer;
}

std::vector <Node*> get_neighbour(std::vector <Node*> graph, const int lenght, const
int width, const Node* node)
{
    std::vector <Node*> buffer;

    if (((node->y + 1) >= 0 && (node->y + 1) < lenght) && (node->x >= 0 && node->x <
width))
    {
        size_t indexUp = (node->y + 1) * width + node->x;

        if (graph[indexUp]->passage == 1)
        {
            buffer.push_back(graph[indexUp]);
        }
    }

    if (((node->y - 1) >= 0 && (node->y - 1) < lenght) && (node->x >= 0 && node->x <
width))
    {
        size_t indexDown = (node->y - 1) * width + node->x;

        if (graph[indexDown]->passage == 1)
        {
            buffer.push_back(graph[indexDown]);
        }
    }

    if ((node->y >= 0 && node->y < lenght) && ((node->x + 1) >= 0 && (node->x + 1) <
width))
    {
        size_t indexRight = node->y * width + (node->x + 1);

        if (graph[indexRight]->passage == 1)
        {
            buffer.push_back(graph[indexRight]);
        }
    }

    if ((node->y >= 0 && node->y < lenght) && ((node->x - 1) >= 0 && (node->x - 1) <
width))
    {
        size_t indexLeft = node->y * width + (node->x - 1);

        if (graph[indexLeft]->passage == 1)
        {
            buffer.push_back(graph[indexLeft]);
        }
    }

    return buffer;
}

bool check_visited_node(std::vector <Node*> setNode, Node* node)
{
    bool result = false;

    for (size_t i = 0; i < setNode.size(); i++)
    {
        if (setNode[i] == node)

```

```

        {
            result = true;
            break;
        }
    }

    return result;
}

void search_way_from_landmark(std::vector<Node*>& graph, const int lenght, const int
width, const int indexLand, const int numberLand)
{
    std::vector<Node*> nextNodes;
    std::vector<Node*> visitedNodes;

    for (size_t i = 0; i < graph.size(); i++)
    {
        if (i == indexLand)
        {
            graph[i]->l[numberLand] = 0;
            continue;
        }
        graph[i]->l[numberLand] = INFINITE;
    }

    nextNodes.push_back(graph[indexLand]);

    while (nextNodes.empty() != true)
    {
        Node* current = extract_min(nextNodes, numberLand);

        visitedNodes.push_back(current);

        std::vector<Node*> neighbours = get_neighbour(graph, lenght, width, current);

        for (size_t i = 0; i < neighbours.size(); i++)
        {
            bool result = check_visited_node(visitedNodes, neighbours[i]);

            if (result == false)
            {
                int tempDistance = current->l[numberLand] + 1;

                if (tempDistance < neighbours[i]->l[numberLand])
                {
                    neighbours[i]->l[numberLand] = tempDistance;
                }

                nextNodes.push_back(neighbours[i]);
            }
        }
    }

    return;
}

void preprocess(int graph[][9], std::vector<Node*>& convertedGraph, const int
lenght, const int width)
{
    for (int i = 0; i < lenght; i++)
    {
        for (int j = 0; j < width; j++)

```

```

    {
        Node* node = nullptr;
        if (graph[i][j] == 1)
        {
            node = new Node(j, i, 1);
        }
        if (graph[i][j] == 0)
        {
            node = new Node(j, i, 0);
        }
        convertedGraph.push_back(node);
    }
}

search_way_from_landmark(convertedGraph, lenght, width, (0 * width + 0), 0);
search_way_from_landmark(convertedGraph, lenght, width, (0 * width + 8), 1);
search_way_from_landmark(convertedGraph, lenght, width, (8 * width + 0), 2);
search_way_from_landmark(convertedGraph, lenght, width, (8 * width + 8), 3);

return;
}

int max(const int oneDist, const int twoDist)
{
    int buffer = 0;

    if (oneDist > twoDist)
    {
        buffer = oneDist - twoDist;
    }
    else
    {
        buffer = twoDist - oneDist;
    }

    return buffer;
}

int dist_estimate(std::vector <Node*> graph, const Node* current, const Node*
destination)
{
    int result = 0;

    for (int i = 0; i < 4; i++)
    {
        int buffer = max(current->l[i], destination->l[i]);

        if (buffer > result)
        {
            result = buffer;
        }
    }

    return result;
}

Node* extract_next_node(std::vector <Node*>& setNode)
{
    Node* buffer = nullptr;
    int index = 0;
    int minValue = INFINITE;

```

```

for (size_t i = 0; i < setNode.size(); i++)
{
    if (setNode[i]->f < minValue)
    {
        index = i;
        minValue = setNode[i]->f;
    }
}

buffer = setNode[index];
setNode.erase(setNode.begin() + index);

return buffer;
}

void find_way_between_two_nodes(std::vector <Node*> & graph, const int lenght, const
int width, const int launch, const int destination)
{
    std::vector <Node*> nextNodes;
    std::vector <Node*> visitedNodes;

    for (size_t i = 0; i < graph.size(); i++)
    {
        if (graph[i]->passage == 1)
        {
            if (i == launch)
            {
                graph[i]->g = 0;
                graph[i]->h = dist_estimate(graph, graph[i], graph[destination]);
                graph[i]->f = graph[i]->g + graph[i]->h;
                continue;
            }
            graph[i]->g = INFINITE;
            graph[i]->h = dist_estimate(graph, graph[i], graph[destination]);
            graph[i]->f = graph[i]->g + graph[i]->h;
        }
        if (graph[i]->passage == 0)
        {
            graph[i]->g = INFINITE;
            graph[i]->h = INFINITE;
            graph[i]->f = graph[i]->g + graph[i]->h;
        }
    }

    nextNodes.push_back(graph[launch]);

    while (nextNodes.empty() != true)
    {
        Node* current = extract_next_node(nextNodes);

        if (current == graph[destination])
        {
            break;
        }

        visitedNodes.push_back(current);

        std::vector <Node*> neighbours = get_neighbour(graph, lenght, width, current);

        for (size_t i = 0; i < neighbours.size(); i++)
        {
            bool result = check_visited_node(visitedNodes, neighbours[i]);

```

```

        if (result == false)
        {
            int tempDistance = current->g + 1;

            if (tempDistance < neighbours[i]->g)
            {
                neighbours[i]->g = tempDistance;
                neighbours[i]->f = neighbours[i]->g + neighbours[i]->h;
                neighbours[i]->parent = current;
            }

            nextNodes.push_back(neighbours[i]);
        }
    }
}

return;
}

void print_path(std::vector <Node*> graph, const int destination)
{
    std::vector <Node*> path;
    Node* current = graph[destination];

    while (current != nullptr)
    {
        path.push_back(current);
        current = current->parent;
    }

    for (size_t i = 0; i < path.size(); i++)
    {
        std::cout << "x=" << path[i]->x << "," << "y=" << path[i]->y;

        if (i != (path.size() - 1))
        {
            std::cout << " <- ";
        }
    }

    std::cout << " start";

    return;
}

void search_way(int graph[][9], const int lenght, const int width, const int launch,
const int destination)
{
    std::vector <Node*> graphNodes;

    preprocess(graph, graphNodes, lenght, width);

    find_way_between_two_nodes(graphNodes, lenght, width, launch, destination);

    print_path(graphNodes, destination);

    return;
}

int main()

```

```

{
    const int lenght = 9;
    const int width = 9;

    int grid[lenght][width] =
    {
        {1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 0, 0, 0, 0},
        {1, 1, 0, 1, 1, 1, 1, 1, 1},
        {1, 1, 0, 1, 1, 1, 1, 1, 1},
        {1, 1, 0, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 0, 1, 1, 1, 1},
        {1, 1, 1, 1, 0, 1, 1, 1, 1},
        {1, 1, 1, 1, 0, 1, 1, 1, 1},
    };

    search_way(grid, lenght, width, 60, 8);

    return 0;
}

```

---