**Федеральное агентство связи**

**Ордена Трудового Красного Знамени**

**Федеральное государственное бюджетное образовательное учреждение**

**Высшего образования**

**«Московский технический университет связи и информатики»**

Кафедра Математической кибернетики и информационных технологий

**Отчет по лабораторной работе №7**

По дисциплине «СиАОД»

На тему:

«Сетевые алгоритмы. Эвристические алгоритмы поиска путей»

Выполнил: студент группы БВТ1801

Задоркин Максим Александрович

Руководитель:

Кутейников Иван Алексеевич

Москва 2020

**Цель работы:**

Реализовать алгоритм поиска кратчайшего расстояния между двумя клетками в соответствии с вариантом и алгоритм A\*. Сравнить результаты работы данного алгоритма с алгоритмом A\*.

Алгоритм: Волновой алгоритм. Дополнительно выбран жадный алгоритм.   
Будут рассматриваться алгоритмы с учетом диагональных путей.

**Программа:**

Программа реализована при помощи языка javascript, domAPI. Так как использованы последние стандарты ecmascript рекомендуется запускать приложение в последних версиях браузера. Тестировано на браузере Google Chrome.

Все файлы кода можно просмотреть и скачать с моего github:

<https://github.com/MaximZar/SaDPA/tree/master/searchRoadOnMatrix>

Опробовать приложение можно прямо по следующей ссылке (для быстрого старта есть шаблон):

<https://maximzar.github.io/SaDPA/searchRoadOnMatrix/src/>

**Код программы**

Код содержит несколько абстракций:

Код алгоритма a\* (остальной код лежит на github):

**const** aStar = () **=>** {

**if** (!document.querySelector('table')) **return**;

**const** diagonal = document.querySelector('#diagonal').checked;

**const** matrix = readTable();

addDirections(matrix, diagonal);

addPosition(matrix);

**const** startCoord = document.querySelector('#startCoord').value.split(' ');

**const** endCoord = document.querySelector('#endCoord').value.split(' ');

**const** startNode = matrix[startCoord[0] - 1][startCoord[1] - 1];

**const** finishNode = matrix[endCoord[0] - 1][endCoord[1] - 1];

startNode.g = 0;

startNode.h = getH(startNode.position, finishNode.position);

startNode.f = startNode.g + startNode.h;

**let** lastMinimal = startNode;

**let** pendingReview = [];

**const** viewed = [];

**while** (lastMinimal.type !== 'end') {

**const** node = lastMinimal;

**const** nodeDirections = Object.values(node.dirs)

.filter((dir) **=>** dir !== **null**) *// убираем преграды из направлений*

.map((dir) **=>** matrix[dir.x][dir.y]) *// находим все эти позиции в матрице*

.filter((dir) **=>** !pendingReview.includes(dir)) *// удаляем те, которые уже стоят в очереди на просмотр*

.filter((dir) **=>** !viewed.includes(dir)); *// удаляем те, которые уже просмотрели*

nodeDirections.forEach((dir) **=>** setGHFFrom(dir, node, finishNode)); *// устанавливаем длины путей*

nodeDirections.forEach((dir) **=>** pendingReview.push(dir)); *// отправляем в очередь для обработки полученные направления*

pendingReview = pendingReview.sort((dir1, dir2) **=>** dir1.f > dir2.f ? 1 : -1); *// сортируем ожидающие направления*

**if** (pendingReview.length === 0 && lastMinimal.type !== 'end') {

alert('Такого пути не существует');

clearTableResult();

**return**;

}

lastMinimal = pendingReview.shift(); *// берем самый маленький и удаляем из очереди*

viewed.push(node); *// добавляем текущий узел в просмотренные*

}

*// находим пройденный путь*

**let** step = matrix[finishNode.position.x][finishNode.position.y];

**const** road = [step];

**while** (road[0].type !== 'start') {

**const** previous = step.from;

step = matrix[previous.x][previous.y];

road.unshift(step);

}

*// вес пути*

**const** weightOfRoad = matrix[finishNode.position.x][finishNode.position.y].g;

drawRoad(road, weightOfRoad);

};

**Тестирование** **программы и скриншоты работы:**

**Вывод:** построил регулярную сеть в виде клеток с заданной стоимостью прохождения. Реализовал алгоритм поиска кратчайшего расстояния между двумя клетками в соответствии с вариантом и алгоритм A\*. При сравнении скорости работы алгоритмов, можно сделать вывод, что алгоритм A\* лучше справляется со своей задачей, но затрачивает больше времени, а волновой алгоритм быстрее, но не учитывает стоимость.