МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

**«Дальневосточный федеральный университет»**

**ШКОЛА ЕСТЕСТВЕННЫХ НАУК**

**Кафедра прикладной математики, механики, управления и программного обеспечения**

**Отчет по лабораторной работе №1**

По дисциплине «Современные языки и системы программирования»

Специальность «Программная инженерия»

|  |
| --- |
| Выполнили студенты гр. М8119-09.04.04  \_\_\_\_\_\_\_ В. А. Титов |
| Проверил доцент кафедры ПММУиПО, к.т.н \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ С. Н. Остроухова  (зачтено/не зачтено) |

г. Владивосток

2020 г.

**Неформальная постановка задачи**

Существует лабиринт, представляемый прямоугольным полем, состоящим из проходимых и непроходимых клеток. Каждая проходимая клетка обладает разной степени проходимости. Необходимо найти кратчайший путь до выхода из лабиринта.

Лабиринт можно представить графом, исключив из него все непроходимые точки, где дугами являются затраты на перемещения из одной клетки в другую, а вершины – клетки поля. Необходимо при помощи объектно-ориентированной модели программирования:

1. Реализовать структуру поля, включающей функции:
   1. конструктор и деструктор
   2. чтение файла со структурой поля
   3. конвертация матричного представления поля в граф
2. Реализовать структуру графа, включающий функции:
3. конструктор и деструктор
4. добавление вершины и дуги
5. удаление вершины и дуги
6. поиск кратчайшего пути из точки A в точку B
7. Реализовать алгоритм A\*
8. **Алгоритм A\***

**Поиск A** — в [информатике](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) и [математике](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0), [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) поиска [по первому наилучшему совпадению](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%BF%D0%BE_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%BC%D1%83_%D0%BD%D0%B0%D0%B8%D0%BB%D1%83%D1%87%D1%88%D0%B5%D0%BC%D1%83_%D1%81%D0%BE%D0%B2%D0%BF%D0%B0%D0%B4%D0%B5%D0%BD%D0%B8%D1%8E) на [графе](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Порядок обхода вершин определяется [**эвристической функцией**](https://ru.wikipedia.org/wiki/%D0%AD%D0%B2%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B0) «расстояние + стоимость» (обычно обозначаемой как *f(x)*). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (*x*) из начальной (обычно обозначается как *g(x)* и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как *h(x)*).

Функция *h(x)* должна быть **допустимой эвристической оценкой**, то есть не должна переоценивать расстояния к целевой вершине. Например, для задачи маршрутизации *h(x)* может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.

Этот алгоритм был впервые описан в [1968 году](https://ru.wikipedia.org/wiki/1968_%D0%B3%D0%BE%D0%B4) [Питером Хартом](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B0%D1%80%D1%82,_%D0%9F%D0%B8%D1%82%D0%B5%D1%80&action=edit&redlink=1), [Нильсом Нильсоном](https://ru.wikipedia.org/w/index.php?title=%D0%9D%D0%B8%D0%BB%D1%8C%D1%81%D0%BE%D0%BD,_%D0%9D%D0%B8%D0%BB%D1%8C%D1%81&action=edit&redlink=1) и [Бертрамом Рафаэлем](https://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B0%D1%84%D0%B0%D1%8D%D0%BB%D1%8C,_%D0%91%D0%B5%D1%80%D1%82%D1%80%D0%B0%D0%BC&action=edit&redlink=1). Это по сути было расширение [алгоритма Дейкстры](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B), созданного в 1959 году. Новый алгоритм достигал более высокой производительности (по времени) с помощью эвристики. В их работе он упоминается как «алгоритм A». Но так как он вычисляет лучший маршрут для заданной эвристики, он был назван A\*.

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все [информированные алгоритмы поиска](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От [жадного алгоритма](https://ru.wikipedia.org/wiki/%D0%96%D0%B0%D0%B4%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), 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. Составляющая *g(x)* — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение *f(x)*, после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в [очереди с приоритетом](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_%D1%81_%D0%BF%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BE%D0%BC_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Приоритет пути определяется по значению *f(x) = g(x) + h(x)*. Алгоритм продолжает свою работу до тех пор, пока значение *f(x)* целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

Чем меньше эвристика *h(x)*, тем больше приоритет, поэтому для реализации очереди можно использовать [сортирующие деревья](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B5%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE).

Множество просмотренных вершин хранится в closed, а требующие рассмотрения пути — в очереди с приоритетом open. Приоритет пути вычисляется с помощью функции *f(x)* внутри реализации очереди с приоритетом.

Псевдокод:

**function** A\*(start, goal, f)

*% множество уже пройденных вершин*

var closed := the empty set

*% множество частных решений*

var open := make\_queue(f)

enqueue(open, path(start))

while open is not empty

var p := remove\_first(open)

var x := the last node of p

if x in closed

continue

if x = goal

return p

add(closed, x)

*% добавляем смежные вершины*

foreach y in successors(x)

enqueue(open, add\_to\_path(p, y))

return failure

1. **Объектно-ориентированная парадигма**
   1. **Основные понятия**

Объектно-ориентированное программирование (ООП) позволяет разбить проблему на составные части, каждая из которых становится самостоятельным объектом. Каждый из объектов содержит свой собственный код и данные, которые относятся к этому объекту. Любая программа, написанная на языке ООП, отражает в своих данных состояние физических предметов либо абстрактных понятий – объектов программирования, для работы, с которыми она предназначена.

С объектом связывается набор действий, иначе называемых методами. С точки зрения языка программирования набор действий или методов – это функции, получающие в качестве обязательного параметра указатель на объект и выполняющие определенные действия с данными объекта программирования. Технология ООП запрещает работать с объектом иначе, чем через методы, таким образом, внутренняя структура объекта скрыта от внешнего пользователя.

Описание множества однотипных объектов называется классом.

Объект – это структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии.

Класс – это описание множества объектов программирования (объектов) и выполняемых над ними действий.

Любая функция в программе представляет собой метод для объекта некоторого класса. Класс должен формироваться в программе естественным образом, как только в ней возникает необходимость описания новых объектов программирования. Каждый новый шаг в разработке алгоритма должен представлять собой разработку нового класса на основе существующих.

Основными приципами ООП являются

* инкапсуляция;
* наследование;
* полиморфизм.

Инкапсуляция данных – это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В ООП код и данные могут быть объединены вместе (в так называемый «черный ящик») при создании объекта.

Наследование. Новый, или производный класс может быть определен на основе уже имеющегося, или базового класса. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса.

Полиморфизм – это свойство, которое позволяет один и тот же идентификатор (одно и то же имя) использовать для решения двух и более схожих, но технически разных задач.

* 1. **Обоснование выбора языка реализации**

Для реализации был выбран язык С#. Несмотря на то, что C# можно назвать мультипарадигменным языком, он вобрал в себя всё лучшее из объектно-ориентированного подхода и отлично подходит для написания объектно-ориентированных программ. В С# реализованы все принципы объектно-ориентированного подхода:

* Все является объектом. Все данные программы хранятся в объектах. Каждый объект создается, существует какое-то время, потом уничтожается.
* Программа есть группа объектов, общающихся друг с другом. Кроме того, что объект хранит какие-то данные, он умеет выполнять различные операции над своими данными и возвращать результаты этих операций.
* Каждый объект имеет свою память, состоящую из других объектов и/или элементарных данных. Объект хранит некоторые данные.
* Каждый объект имеет свой тип (класс). В объектно-ориентированном подходе не рассматривается возможность создания произвольного объекта, состоящего из того, например, что мы укажем в момент его создания. Все объекты строго типизированы.
* Все объекты одного и того же типа могут получать одни и те же сообщения.
  1. **Описание классов**

**class Matrix** – класс описывающий лабиринт и содержащий методы:

* public void ReadFromFile(string fileName) – чтения структуры лабиринта из файла;
* public Graph ConvertToGraph() – конвертирует матричное представление лабиринта в графовую структуру проходимых клеток;

**class AStar** – класс реализующий алгоритм AStar

public Astar(Graph graph); - конструктор

public Stack<ILocation> FindPath(ILocation start, ILocation finish); - возвращает стэк клеток составляющих кратчайший путь от start до finish.

**public class Graph** – класс, описывающий граф и содержащий методы, которые инициируют работу .

* public void AddEdge/RemoveEdge (Edge e) – добавление/удаление рёбер графа.
* Public void AddLocation/RemoveLocation(ILocation location) – добавление/удаление вершин графа.
* Public FindPathAStar(ILocation start, ILocation finish) – инициирует объект типа IPathFindingAlgorythm объектом типа AStar и вызывает на нём метод FindPath(start, finish)

1. **Тестирование**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Тестовая ситуация | Структура лабиринта | Входные данные | Кратчайший путь |
| 1 | Выполнение алгоритма на пустом графе |  | Start: (0;0)  Finish: (0;0) |  |
| 2 | Выполнение алгоритма на графе с одной вершиной |  | Start: (0;0)  Finish: (0;0) | (0;0) |
| 3 | Выполнение алгоритма на графе начиная с «Висячей вершиной» | 1 2 3  2 1 3  -1 -1 -1  -1 -1 2 | Start (2;3)  Finish(0;0) |  |
| 4 | Выполнение алгоритма на графе заканчивая на «Висячей вершиной» | 1 2 3  2 1 3  -1 -1 -1  -1 -1 2 | Start (0;0)  Finish(2;3) |  |
| 5 | Успешное выполнение алгоритма | -1 5 8 4 2  13 -1 -1 7 9  1 3 4 11 -1  1 7 -1 -1 -1  5 3 5 3 1 |  | (0:1) -> (0:2) -> (0:3) -> (1:3) -> (2:3) -> (2:2) -> (2:1) -> (3:1) -> (4:1) -> (4:2) -> (4:3) -> (4:4) |
| 6 | Успешное выполнение алгоритма с альтернативными весами | -1 5 8 4 2  13 -1 -1 7 9  1 3 4 11 -1  1 70 -1 -1 -1  5 3 5 3 1 |  | (0:1) -> (0:2) -> (0:3) -> (1:3) -> (2:3) -> (2:2) -> (2:1) -> (2:0) -> (3:0) -> (4:0) -> (4:1) -> (4:2) -> (4:3) -> (4:4) |