

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»**

(СПБГУТ)

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ СЕТЕЙ И СИСТЕМ (ИКСС)

**КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ (ПИ И
ВТ)**

ДИСЦИПЛИНА: «АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

ЛАБОРАТОРНАЯ РАБОТА №6.

ТЕМА: «ЭВРИСТИЧЕСКИЕ АЛГОРИТМЫ (ЛАБИРИНТЫ)»

Выполнили:

Студенты группы ИКПИ-05

Принял:

Доцент кафедры ПИиВТ

Молошников Ф.А., Мартынюк А.А.

Подпись _____

Дагаев А.В.

Подпись _____

«_____» _____ 2022

СОДЕРЖАНИЕ

1. Цель работы	3
2. Область применения	3
3. Описание Программы.....	3
3.1. Описание программы и среды разработки	3
3.2. Структура программы	4
3.3. Разработка классов и функций.....	5
3.3.1. maze	5
3.3.2. mazeWeighted	7
3.3.3. statisticsMaze	7
3.3.1. statisticsMazeSearch	8
4. Описание алгоритмов	9
4.1. Алгоритмы генерации лабиринта.....	9
4.1.1. Олдоса-Бродера.....	9
4.1.2. Уилсона.....	11
4.1.3. Уилсона (модификация).....	15
4.1.4. Двоичным деревом	16
4.2. Алгоритмы поиска пути в лабиринте.....	19
4.2.1. Подготовка лабиринта	19
4.2.2. Ли	19
4.2.3. Ли (модификация с двумя волнами)	21
4.2.4. Дейкстры.....	22
4.2.5. A* (AStar; A со звездой).....	26
5. Результаты работы	29
5.1. Графики	29
5.2. Анализ графиков	33
5.2.1. Алгоритмы генерации лабиринтов	33
5.2.2. Алгоритмы поиска в лабиринтах	33
5.2.3. Сравнение алгоритмов генерации и поиска	34
6. Выводы	34
7. Инструкция пользователя	34
7.1. Подготовка среды (Debian 11)	34
7.2. Установка проекта.....	35
7.3. Сборка и запуск.....	35
8. Список используемых источников	39

1. ЦЕЛЬ РАБОТЫ

Ознакомление с эвристическими алгоритмами поиска пути в лабиринте и методикой оценки их эффективности. Для сравнения были выбраны:

- 1) Алгоритмы генерации лабиринта:
 - 1) Уилсона
 - 2) Уилсона (модифицированный)
 - 3) Олдоса-Бродера
 - 4) Бинарного дерева
- 2) Алгоритмы поиска пути в лабиринте:
 - 1) Ли
 - 2) Ли (модификация с 2 волнами)
 - 3) Дейкстры
 - 4) A* (AStar)

Первые два алгоритма поиска пути работают на ДРП (дискретном рабочем поле), где, с точки зрения прокладки путей, все ячейки одинаковы. 3 и 4 алгоритмы работают на графах. Мы используем их модификацию для работы на ДРП (частный случай графа), где ячейки могут иметь разные веса. Вес означает путь в эту ячейку из соседней ячейки. В общем случае, вес ячейки соответствует ребру соответствующего ориентированного графа.

2. ОБЛАСТЬ ПРИМЕНЕНИЯ

Разработанная программа может использоваться в разработке игр. Ведь лабиринты — это не только самостоятельный класс игр, но и основа для создания локаций в играх других жанров: например, систем пещер, которые, в свою очередь, могут быть использованы в очень широком классе игр-бродилок. При постоянном изучении одних и тех же локаций, теряется интерес к игре. Потому алгоритмы генерации лабиринтов, рассмотренные в данной работе, смогут решить данную проблему, чтобы каждое очередное прохождение игры проходило на заново сгенерированной территории.

Проанализированные в работе алгоритмы поиска оптимального пути также могут использоваться в играх. Пример внедренных алгоритмов из данной работы в уже существующие игры: алгоритм Дейкстры – в *Might and Magic III*, A* - *Factorio*.

Однако, стоит подчеркнуть, что алгоритмы поиска оптимального пути могут быть полезны и в программном обеспечении беспилотных систем. Разработанная программа может быть внедрена как встраиваемый модуль для квадрокоптеров или дронов: с помощью нее системы смогут ориентироваться на карте местности, рассчитывать правильный пути с избеганием препятствий и столкновения с различными объектами. В наземной же робототехнике рассмотренные алгоритмы генерации лабиринтов и поиска оптимального пути могут быть также внедрены в роботов-доставщиков.

3. ОПИСАНИЕ ПРОГРАММЫ

3.1. Описание программы и среды разработки

Для исследования алгоритмов генерации лабиринтов [2] и поиска в них пути были разработаны классы `maze` и `mazeWeighted`, которые реализуют хранение лабиринтов и доступ к ним. Для сбора информации о времени генерации лабиринтов и о времени поиска в них путей были разработаны классы `StatisticsMaze` и `StatisticsMazeSearch`. В работе использовались классы `presenthandler` и `plothandler`, скрипт `PlotScript.bash`, разработанные в рамках предыдущих лабораторных работ и предназначенные для упрощения вывода промежуточных и конечных результатов работы алгоритмов. Работа также содержит вспомогательные функции для ввода и вывода. Помимо двух основных целей, которые собираются из файлов `StatisticsMaze_main.cpp` и `StatisticsMazeSearch_main.cpp` и служат для исследования времени генерации лабиринтов и

поиска путей в них соответственно, работа содержит отдельные цели для демонстрации алгоритмов, а так же для демонстрации и отладки разработанных классов.

- 1) Язык: C++
- 2) Среда: VS Code
- 3) ОС: Debian 11
- 4) Оболочка: xfce 4
- 5) Класс "priority_queue" стандартной библиотеки STL C++

3.2. Структура программы

Таким образом, работа состоит из следующих файлов:

- 1) Файлы с реализацией основных возможностей
 - Файлы системы сборки:
 - CMakeLists.txt
 - Файлы с классами и функциями над лабиринтами
 - maze.h
 - MazeGenerationAlgs.h
 - MazeSearchAlg.h
 - Файлы с реализацией сбора времени выполнения алгоритмов
 - StatisticsMaze.inl
 - StatisticsMazeSearch.inl
 - Файлы со вспомогательными функциями
 - Funcs.h
 - presenthandler.h
 - plothandler.h
 - bash-скрипты
 - PlotScript.bash
- 2) Файлы с реализацией примеров использования основных возможностей
 - maze_debug_main.cpp
 - mazeWeighted_debug_main.cpp
 - Wilson_main.cpp
 - AldousBroder_main.cpp
 - BinaryTree_main.cpp
 - Lee_main.cpp
 - Lee2Waves_main.cpp
 - Dijkstra_main.cpp
 - AStar_main.cpp
 - StatisticsMaze_main.cpp
 - StatisticsMazeSearch_main.cpp

3.3. Разработка классов и функций

Остановимся подробнее на некоторых классах и функциях, существенных в данной работе:

3.3.1. maze

Класс, реализующий лабиринт. Позволяет хранить лабиринт и обращаться к нему.

Лабиринт хранится внутри вектора BaseVector в виде символов. Нечетные строки хранят информацию о верхних и нижних стенах ячеек, четные – о боковых стенах ячеек и значения, хранящиеся в ячейках. Используются следующие символы:

- 1) # - стена между ячейками
- 2) ? – отсутствие стены между ячейками
- 3) + - угол ячейки (Служит для удобства представления. Никак не используется)
- 4) . – Значение ячейки по умолчанию.

Такой подход к хранению лабиринта нельзя назвать оптимальным по используемой памяти, т.к. треть памяти занята символами «+», которые нигде не используются (например, более оптимальным было бы хранить в массиве только нижнюю и левую стены), однако он обеспечивает довольно быстрое обращение к своим элементам, поскольку использует контейнер vector для хранения элементов.

Также он позволяет в относительно наглядном виде просматривать лабиринт (рисунок 1 и рисунок 2) и относительно просто его редактировать (для замены стены на проем или проем на стену достаточно поменять символ # на ? или ? на #).

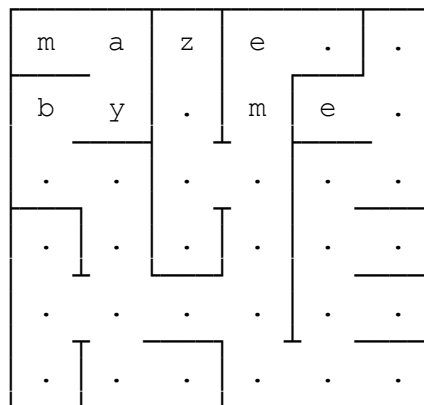


Рисунок 1 – Лабиринт и его внутреннее представление

```

+++++
#m?a#z#e?.#.#
+#+?+?+?+#+?+
#b?y#.#m#e?.#
+?+#+?+?+#+?+
#.#.#.#.#.#
+#+?+?+?+#+
#.#.#.#.#.#
+?+?+#+?+#+
#.#.#.#.#.#
+?+?+#+?+#+
#.#.#.#.#.#
+++++

```

Рисунок 2 – Внутреннее представление лабиринта

Доступ к ячейкам реализован по индексам i j . При этом нумерация ячеек аналогична нумерации элементов в матрицах: Увеличение индекса строк i идет сверху вниз, увеличение индекса столбцов j слева направо.

Вывод лабиринтов в декоративном виде осуществляется с помощью символов Unicode и доступен в двух вариантах: с мелкими и крупными ячейками. Вариант с крупными ячейками уже был показан выше (рисунок 1), вариант с мелкими ячейками показан ниже (рисунок 3).

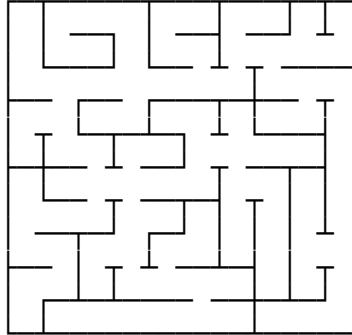


Рисунок 3 – Лабиринт с мелкими ячейками

3.3.1.2. Свойства:

<code>int n=0</code>	высота лабиринта
<code>int m=0;</code>	ширина лабиринта
<code>std::vector<std::vector<char>> BaseVector;</code>	Вектор, в котором хранится лабиринт

3.3.1.3. Методы:

<code>maze(int Nn, int Nm)</code>	Конструктор с параметрами. Создает лабиринт высотой Nn и шириной Nm , ячейки которого изолированы друг от друга стенами.
<code>void SetCellValue(int i, int j, char c)</code>	Устанавливает значение в ячейке
<code>void ResetValues()</code>	Сбрасывает значения ячеек (по умолчанию «.»)
<code>char GetCellValue(int i, int j)</code>	Возвращает значение в ячейке
<code>void SetCellWalls(int i, int j, int alpha, bool HasWall, bool Protected=false)</code>	Позволяет установить или убрать стену между ячейками. α – полярный угол, под которым радиус вектор из центра ячейки встречает соответствующую стену (измеряется в $\pi/4$). Т.е. если $\alpha=0$, то будет изменено состояние правой стены ячейки, если $\alpha=1$, то верхней стены ячейки, если $\alpha=2$, то левой стены, если $\alpha=3$, то нижней стены и т.д. Флаг <code>Protected</code> позволяет замораживать внешние стены лабиринта, не позволяя изменять их состояние.
<code>bool HasWall(int i, int j, int alpha)</code>	Позволяет определять наличие стены вокруг ячейки. α имеет тот же смысл, что и в методе <code>SetCellWalls</code>
<code>int GetNeighborI(int i, int alpha)</code> <code>int GetNeighborJ(int j, int alpha)</code>	Позволяют получить координаты (в I , J) соседней ячейки, радиус вектор до которой находится под углом α .

<code>std::string Show(char* filename=(char*)"cin")</code>	Не используется
<code>void CharFromWallFlags(std::stringstream& ss, std::string& WallFlags)</code>	Вспомогательная функция для получения символа по флагу, т.е. по его описанию.
<code>GetCellValueFromStruct</code> 3 перегрузки	Вспомогательные функции для организации вывода в декоративном виде данных внутри ячеек.
<code>std::string StrFromInt(T value)</code>	
<pre>template <typename T=const std::string> void ShowDecorate(char* filename=(char*)"cout", int Mode=0, int Scale=1, bool IsWithValues=false, T& Values=std::string("GetCellValue"), int member=0)</pre>	Шаблонный метод для вывода лабиринта в терминал или файл. Первый аргумент – имя файла для вывода (по умолчанию в терминал), второй аргумент – режим вывода 0 – внутреннее представление и декоративный вид. 1 – только декоративный вид. Scale – 1 мелкие ячейки. (высота каждой ячейки 0.5+0.5=1 строка; ширина – 0.5+1+0.5=2 символа), 2 крупные ячейки. В случае крупных ячеек возможен вывод внутри ячеек значений из внешних контейнеров различных типов. member служит для выбора поля внутри элемента контейнера для представления внутри ячейки. Для того чтобы выводить какой-либо свой тип контейнера внутри ячейки, нужно написать перегрузку функции <code>GetCellValueFromStruct</code> . В ячейку можно выводить значения длиной до 3х символов. Если выводимое значение имеет большую длину, то можно, например, выводить остаток этого значения от деления на 1000 либо отсечь часть символов.

3.3.2. mazeWeighted

Наследует класс `maze`. Позволяет создавать лабиринты с весами ячеек.

3.3.2.1. Свойства:

<code>std::vector<std::vector<int>>> Weights;</code>	Контейнер с весами ячеек
---	--------------------------

3.3.2.2. Методы:

<code>mazeWeighted(int Nn, int Nm, int FillWeight=0)</code>	Конструктор с параметрами. Создает лабиринт из изолированных друг от друга ячеек с весом по умолчанию равным нулю.
<code>void WeightsToValues()</code>	Не используется. Лучше использовать <code>ShowDecorate</code> с параметром <code>Weights</code> . Копирует веса в значения ячеек для удобного представления.

3.3.3. statisticsMaze

Класс для исследования функций, выполняющих генерацию лабиринтов.

3.3.3.1. Свойства:

<code>int CurrentN=0;</code>	Текущая высота лабиринта
------------------------------	--------------------------

<code>int CurrentM=0;</code>	Текущая ширина лабиринта
<code>int NStart=0, NEnd=0, NStep=0;</code>	Начальное, конечное значения высоты лабиринта и шаг её изменения
<code>double MRatio=1;</code>	М/Н отношение ширины к длине
<code>int NumberOfRuns=1;</code>	число прогонов для текущих значений N М

3.3.3.2. Методы:

<code>StatisticsMaze(int NNStart, int NNEnd, int NNStep, double NMRatio, int NNumberOfRuns, std::default_random_engine& generator, presenthandler& PresentHandler, std::string filename, void (*CallBackGenerate)(maze&, std::default_random_engine&, presenthandler&, CallBackParamsTail&...), CallBackParamsTail& ...callbackparamstail)</code>	Конструктор. Запускает функцию генерации лабиринта со значениями высоты от NNStart до NNEnd с шагом NNStep и отношением ширины к высоте NMRatio. Записывает время генерации в файл.
<code>void printLabel(std::string filename)</code>	Запись информации об условиях выполнения сортировки в файл
<code>void printLabel(std::string filename)</code>	Запись искомых значений времени в файл

3.3.1. statisticsMazeSearch

Аналогичен классу statisticsMaze. Следует отметить, что передача в конструктор генератора позволяет для различных алгоритмов поиска генерировать одинаковую последовательность лабиринтов, причем последовательности стартовых и финишных ячеек так же будут совпадать.

3.3.1.1. Свойства:

<code>int CurrentN=0;</code>	Текущая высота лабиринта
<code>int CurrentM=0;</code>	Текущая ширина лабиринта
<code>int NStart=0, NEnd=0, NStep=0;</code>	Начальное, конечное значения высоты лабиринта и шаг её изменения
<code>double MRatio=1;</code>	М/Н отношение ширины к длине
<code>int NumberOfRuns=1;</code>	число прогонов для текущих значений N М

3.3.1.2. Методы:

<code>void StatisticsMazeSearchFunc(int NNStart, int NNEnd, int NNStep, double NMRatio, int NNumberOfRuns, std::default_random_engine& generator, presenthandler& PresentHandler, std::string filename, void (*CallBackSearch)(T&, int starti, int startj, int finishi, int finishj, std::vector<std::pair<int,int>>& Path, presenthandler& PrHandler))</code>	Генерирует лабиринт со значениями высоты от NNStart до NNEnd с шагом NNStep и отношением ширины к высоте NMRatio. Запускает для него функцию поиска между случайно выбранными ячейками и записывает время поиска в файл
<code>void printLabel(std::string filename)</code>	Запись информации об условиях выполнения сортировки в файл

void printLabel(std::string filename)	Запись искомых значений времени в файл
---------------------------------------	--

4. ОПИСАНИЕ АЛГОРИТМОВ

4.1. Алгоритмы генерации лабиринта

В данной работе реализованы алгоритмы генерации идеальных лабиринтов [3], т.е. лабиринтов без недостижимых областей и без циклов. Генерация происходит методом вырезания проходов.

4.1.1. Олдоса-Бродера

4.1.1.1. Теоретические сведения

Тип: алгоритм на основе дерева

Фокус: присутствует возможность вырезать/добавлять стены

Смещенность: отсутствует. Алгоритм по-разному воспринимает направления и стороны лабиринта.

Однородность: присутствует. Алгоритм генерирует все возможные лабиринты с равной вероятностью

Объем дополнительной памяти для реализации алгоритма: 0

4.1.1.2. Алгоритм

- 1) Лабиринт состоит из изолированных стенами ячеек
- 2) Выбирается случайная ячейка и отмечается как посещенная
- 3) Счетчик посещенных вершин k устанавливается равным 1
- 4) Пока k не равен числу ячеек в лабиринте
 - 1) Выбирается случайное направление движения
 - 2) Если это направление не направлено в стену, то
 - 1) проверяем, была ли уже посещена эта ячейка. Если нет, то прорезаем стену к ней, переходим к ней и помечаем её как посещенную
 - 3) В противном случае просто переходим к ней

4.1.1.3. Пример работы

На рисунке 4 приведен пример работы алгоритма Олдоса-Бродера [3].

Символом «1» функция помечает посещенные ячейки, «.» - непосещенные.

Также будем обозначать символом «*» текущую ячейку.

Система координат: (i, j) i увеличивается сверху вниз. j -слева направо. Счет с нуля.

- 1) Начальная ячейка была выбрана с координатами (1, 1)
- 2) Случайным образом было выбрано направление в правую сторону. Поскольку ячейка справа ещё не была посещена, то стена к ней прорезается. Происходит переход к этой ячейке (1, 2)
- 3) Аналогично проходится ячейка (0,2)
- 4) Выбирается направление вниз. Так как нижняя ячейка уже была посещена, то стена к ней не прорезается(в случае, если бы она там была, это было бы важно) и происходит переход к этой ячейке
- 5) Аналогично 4) возвращаемся к ячейке (0, 2)
- 6) Аналогично шагу 2) шаги 6)-11)

11) После 11 шага работа алгоритма завершается, т.к. количество посещенных ячеек стало равно количеству ячеек в лабиринте.

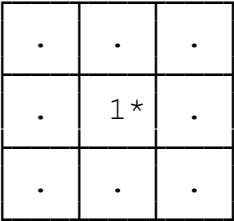
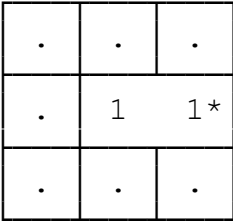
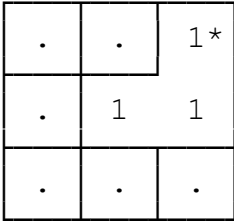
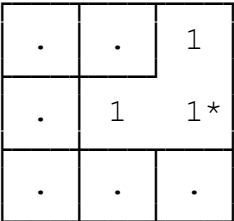
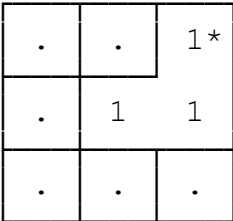
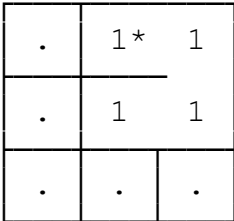
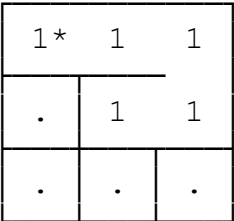
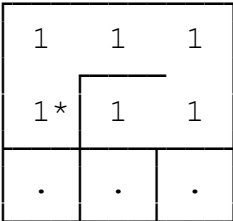
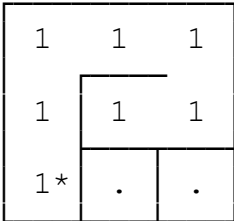
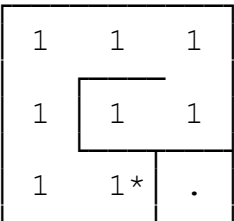
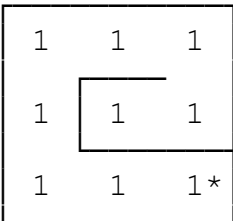
1					
4					
7					
10		11			

Рисунок 4 – Пример работы алгоритма Олдоса-Бродера

4.1.1.4. Примеры лабиринтов

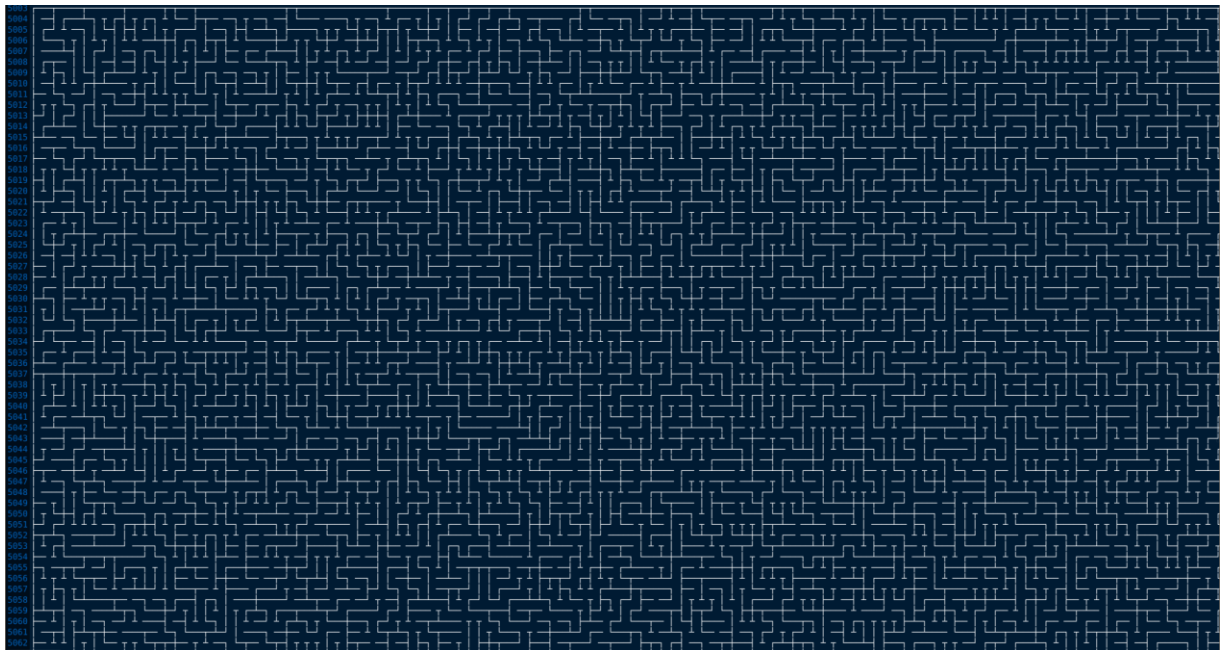


Рисунок 5 – Фрагмент лабиринта, сгенерированного алгоритмом Олдоса-Бродера (2500x2500 ячеек)

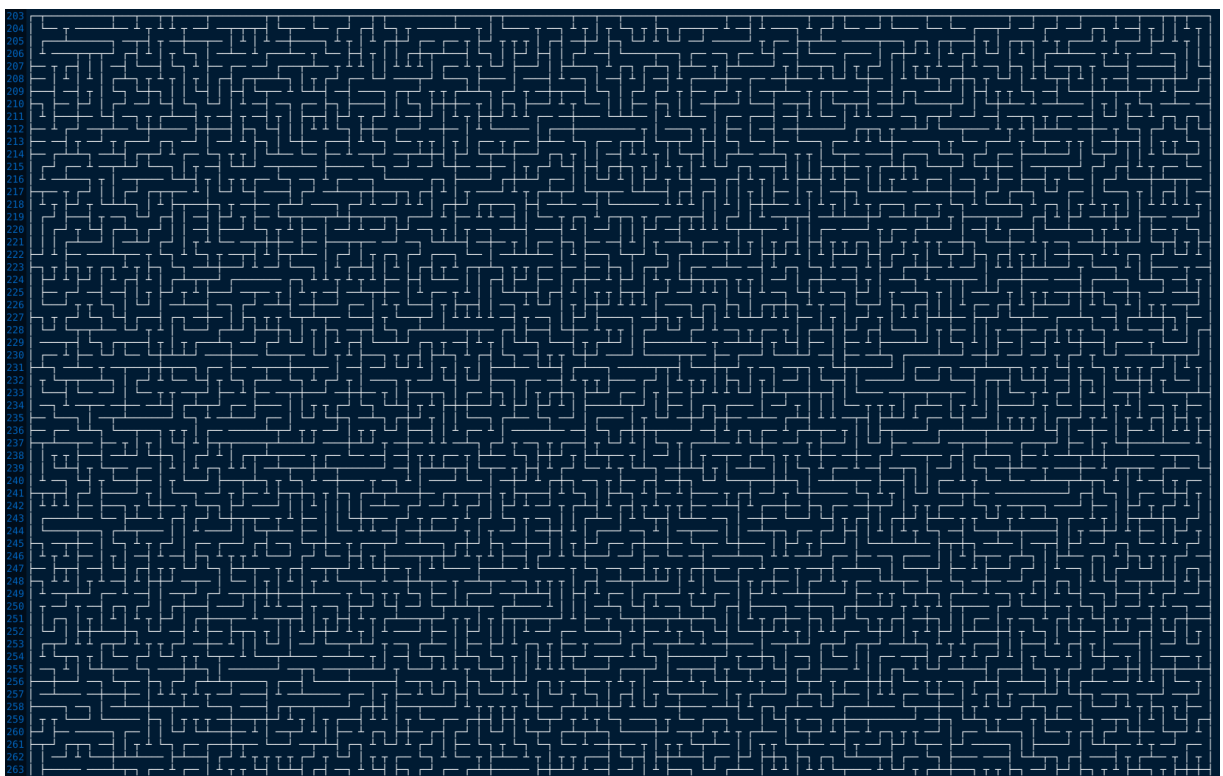


Рисунок 6 – лабиринт, сгенерированный алгоритмом Олдоса-Бродера (100x100 ячеек)

4.1.2. Уилсона

4.1.2.1. Теоретические сведения

Тип: алгоритм на основе дерева

Фокус: присутствует возможность вырезать/добавлять стены

Смещенность: отсутствует. Алгоритм по-разному воспринимает направления и стороны лабиринта.

Однородность: Да. Алгоритм генерирует все возможные лабиринты с равной вероятностью

Объём дополнительной памяти для реализации алгоритма: N^2 (пропорционально количеству ячеек)

4.1.2.2. Алгоритм

- 1) Лабиринт состоит из изолированных стенами ячеек
- 2) Выбрать случайную ячейку и добавить её в множество UST (uniform spanning trees — однородные остовные деревья).
- 3) Пока есть ячейки, не входящие в UST, выбрать случайную ячейку и совершить от неё случайную прогулку, пока не будет встречена какая-нибудь ячейка из UST. Во время прогулки в ячейки записываются путевые координаты. Причем если во время такой прогулки мы наткнемся на ячейку, в которой уже были, т.е. сделаем петлю, то её путевые координаты перезапишутся, т.е. эта петля не попадет в итоговый путь (по-видимому, это одна из причин, по которой этот алгоритм такой медленный)
- 4) Проходим по путевым координатам от ячейки, с которой мы начали прогулку, до ячейки из UST по путевым координатам, прорезая стены (сразу это нельзя сделать из-за петель, которые срезает алгоритм во время первого прохода).

4.1.2.3. Пример работы

Приведем пример работы рассматриваемого алгоритма (рисунок 7). Символом «U» будем помечать ячейки в UST, «>»^»«<»«v» - путевые координаты. «.» - непосещенные ячейки. Также будем обозначать символом «*» текущую ячейку

Система координат: (i, j) i увеличивается сверху вниз. j-слева направо. Счет с нуля.

- 1) Начальная ячейка была выбрана с координатами (1, 2). Добавляем её в UST.
- 2) Выберем случайную ячейку, которо Начнем блуждание, пока не дойдем до ячеек, которые есть в UST
- 3) Начнем блуждание, пока не дойдем до ячеек, которые есть в UST
- 8) Путевую координату ячейки (1, 0) перезаписали, т.к. мы случайным образом выбрали направление вниз, а не вверх.
- 9) Путевую координату ячейки (2, 0) перезаписали, т.к. мы случайным образом выбрали направление вправо, а не вверх.
- 10) Продолжаем блуждание
- 12) Мы дошли до ячейки в UST.
- 13) Прорезаем путь от исходной ячейки до ячейки в UST. Добавляя ячейки в UST.
- 16) Путь прорезан. Путевые координаты ячеек, которые были в петлях, оставляем. На работу алгоритма никакого влияния они не окажут.
- 17) Аналогично включаем в UST оставшиеся ячейки (17 – 30)

1	<table><tr><td>.</td><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td><td>\cup^*</td></tr><tr><td>.</td><td>.</td><td>.</td></tr></table>	\cup^*	.	.	.	2	<table><tr><td>.</td><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td><td>\cup^*</td></tr><tr><td>*</td><td>.</td><td>.</td></tr></table>	\cup^*	*	.	.	3	<table><tr><td>.</td><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td><td>\cup</td></tr><tr><td>\wedge^*</td><td>.</td><td>.</td></tr></table>	\cup	\wedge^*	.	.
.	.	.																														
.	.	\cup^*																														
.	.	.																														
.	.	.																														
.	.	\cup^*																														
*	.	.																														
.	.	.																														
.	.	\cup																														
\wedge^*	.	.																														

4	<table><tr><td>.</td><td>.</td><td>.</td></tr><tr><td>^*</td><td>.</td><td>U</td></tr><tr><td>^</td><td>.</td><td>.</td></tr></table>	.	.	.	^*	.	U	^	.	.	5	<table><tr><td>>*</td><td>.</td><td>.</td></tr><tr><td>^</td><td>.</td><td>U</td></tr><tr><td>^</td><td>.</td><td>.</td></tr></table>	>*	.	.	^	.	U	^	.	.	6	<table><tr><td>></td><td>v*</td><td>.</td></tr><tr><td>^</td><td>.</td><td>U</td></tr><tr><td>^</td><td>.</td><td>.</td></tr></table>	>	v*	.	^	.	U	^	.	.
.	.	.																														
^*	.	U																														
^	.	.																														
>*	.	.																														
^	.	U																														
^	.	.																														
>	v*	.																														
^	.	U																														
^	.	.																														
7	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>^</td><td><*</td><td>U</td></tr><tr><td>^</td><td>.</td><td>.</td></tr></table>	>	v	.	^	<*	U	^	.	.	8	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>v*</td><td><</td><td>U</td></tr><tr><td>^</td><td>.</td><td>.</td></tr></table>	>	v	.	v*	<	U	^	.	.	9	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>v</td><td><</td><td>U</td></tr><tr><td>>*</td><td>.</td><td>.</td></tr></table>	>	v	.	v	<	U	>*	.	.
>	v	.																														
^	<*	U																														
^	.	.																														
>	v	.																														
v*	<	U																														
^	.	.																														
>	v	.																														
v	<	U																														
>*	.	.																														
10	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>v</td><td><</td><td>U</td></tr><tr><td>></td><td>^*</td><td>.</td></tr></table>	>	v	.	v	<	U	>	^*	.	11	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>v</td><td>>*</td><td>U</td></tr><tr><td>></td><td>^</td><td>.</td></tr></table>	>	v	.	v	>*	U	>	^	.	12	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>v</td><td>></td><td>U*</td></tr><tr><td>></td><td>^</td><td>.</td></tr></table>	>	v	.	v	>	U*	>	^	.
>	v	.																														
v	<	U																														
>	^*	.																														
>	v	.																														
v	>*	U																														
>	^	.																														
>	v	.																														
v	>	U*																														
>	^	.																														
13	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>v</td><td>></td><td>U</td></tr><tr><td>U*</td><td>^</td><td>.</td></tr></table>	>	v	.	v	>	U	U*	^	.	14	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>v</td><td>></td><td>U</td></tr><tr><td>U</td><td>U*</td><td>.</td></tr></table>	>	v	.	v	>	U	U	U*	.	15	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>v</td><td>U*</td><td>U</td></tr><tr><td>U</td><td>U</td><td>.</td></tr></table>	>	v	.	v	U*	U	U	U	.
>	v	.																														
v	>	U																														
U*	^	.																														
>	v	.																														
v	>	U																														
U	U*	.																														
>	v	.																														
v	U*	U																														
U	U	.																														
16	<table><tr><td>></td><td>v</td><td>.</td></tr><tr><td>v</td><td>U</td><td>U*</td></tr><tr><td>U</td><td>U</td><td>.</td></tr></table>	>	v	.	v	U	U*	U	U	.	17	<table><tr><td>v*</td><td>v</td><td>.</td></tr><tr><td>v</td><td>U</td><td>U</td></tr><tr><td>U</td><td>U</td><td>.</td></tr></table>	v*	v	.	v	U	U	U	U	.	18	<table><tr><td>v</td><td>v</td><td>.</td></tr><tr><td>>*</td><td>U</td><td>U</td></tr><tr><td>U</td><td>U</td><td>.</td></tr></table>	v	v	.	>*	U	U	U	U	.
>	v	.																														
v	U	U*																														
U	U	.																														
v*	v	.																														
v	U	U																														
U	U	.																														
v	v	.																														
>*	U	U																														
U	U	.																														
19	<table><tr><td>U*</td><td>v</td><td>.</td></tr><tr><td>></td><td>U</td><td>U</td></tr><tr><td>U</td><td>U</td><td>.</td></tr></table>	U*	v	.	>	U	U	U	U	.	20	<table><tr><td>U</td><td>v</td><td>.</td></tr><tr><td>U*</td><td>U</td><td>U</td></tr><tr><td>U</td><td>U</td><td>.</td></tr></table>	U	v	.	U*	U	U	U	U	.	21	<table><tr><td>U</td><td>v</td><td>.</td></tr><tr><td>U</td><td>U*</td><td>U</td></tr><tr><td>U</td><td>U</td><td>.</td></tr></table>	U	v	.	U	U*	U	U	U	.
U*	v	.																														
>	U	U																														
U	U	.																														
U	v	.																														
U*	U	U																														
U	U	.																														
U	v	.																														
U	U*	U																														
U	U	.																														

22		23		24	
25		26		27	
28		29		30	

Рисунок 7 – Пример работы алгоритма Уилсона

4.1.2.4. Примеры лабиринтов

Лабиринты, генерируемые алгоритмом Уилсона (рисунок 8-9), по характеру не отличимы от лабиринтов, генерируемых алгоритмом Олдоса-Бродера.

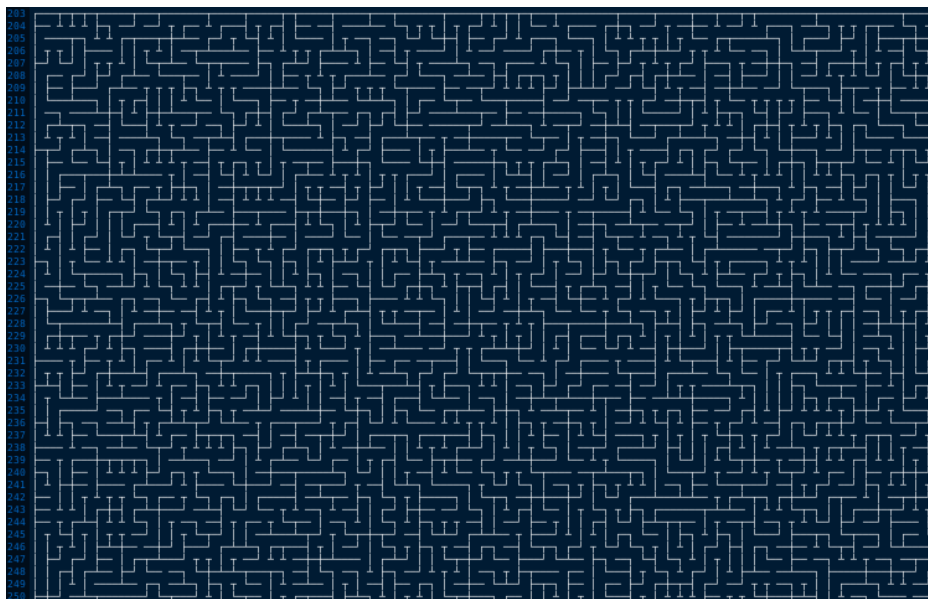


Рисунок 8 – Лабиринт, сгенерированный алгоритмом Уилсона. 100x100 ячеек

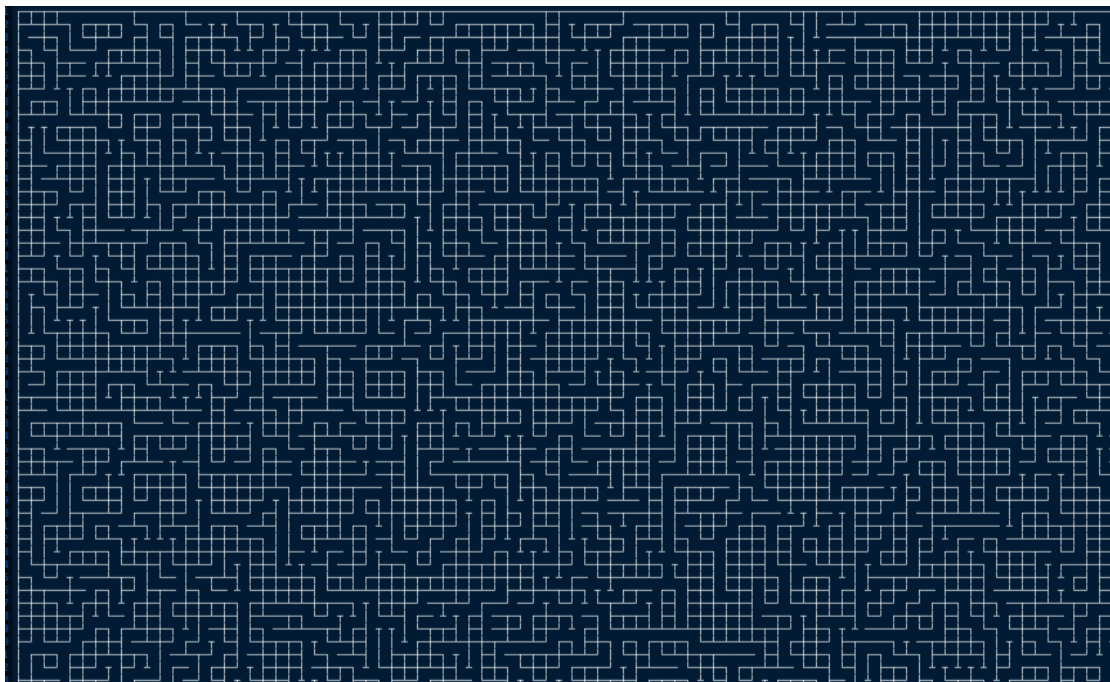


Рисунок 9 – Фрагмент лабиринта во время генерации алгоритмом Уилсона. 1000x1000 ячеек

4.1.3. Уилсона (модификация)

Данный алгоритм отличается от предыдущего методом выбора ячейки, которой ещё нет в множестве UST. В отличие от предыдущего, здесь (рисунок 10) ячейки выбираются последовательно от начала лабиринта. Отличие наглядно показано на рисунке:

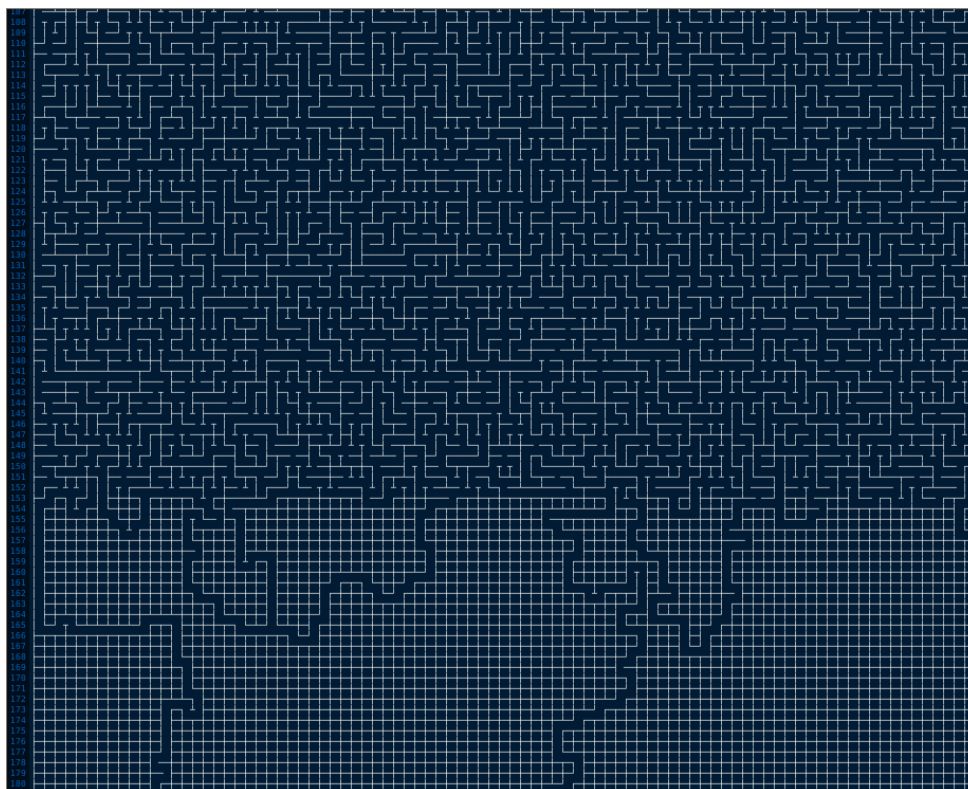


Рисунок 10 – Фрагмент лабиринта во время генерации алгоритмом Уилсона (модификация). 500x500 ячеек

4.1.4. Двоичным деревом

4.1.4.1. Теоретические сведения

Тип: алгоритм на основе множеств

Фокус: присутствует возможность вырезать/добавлять стены

Смещенность: присутствует. Алгоритм одинаково воспринимает направления и стороны лабиринта

Однородность: алгоритм никогда не генерирует лабиринты с равной вероятностью

Объём дополнительной памяти для реализации алгоритма: 0

4.1.4.2. Алгоритм

1) Лабиринт состоит из изолированных стенами ячеек

2) Для каждой ячейки в сетке случайным образом разделяем проход на север или запад (север или восток / юг или запад / юг или восток) двигаясь построчно. В данной реализации сразу вырезается один из столбцов (такой же результат можно было бы получить, двигаясь построчно, пробегая все столбцы).

4.1.4.3. Пример работы

Продemonстрируем генерацию для выбранного направления север-восток (рисунок 11)

Вырезается первая/последняя строка (при выбранном направлении север/юг). Далее вырезается левый/правый столбец, согласно выбранному направлению (запад/восток). И потом – построчно для данных направлений сверху-вниз.

1) Изначальное состояние поля

2) Вырезается первая/последняя строка (при выбранном направлении север/юг)

4) Вырезается левый/правый столбец, согласно выбранному направлению (запад/восток)

6) Далее построчно (для данных направлений построчно сверху вниз)

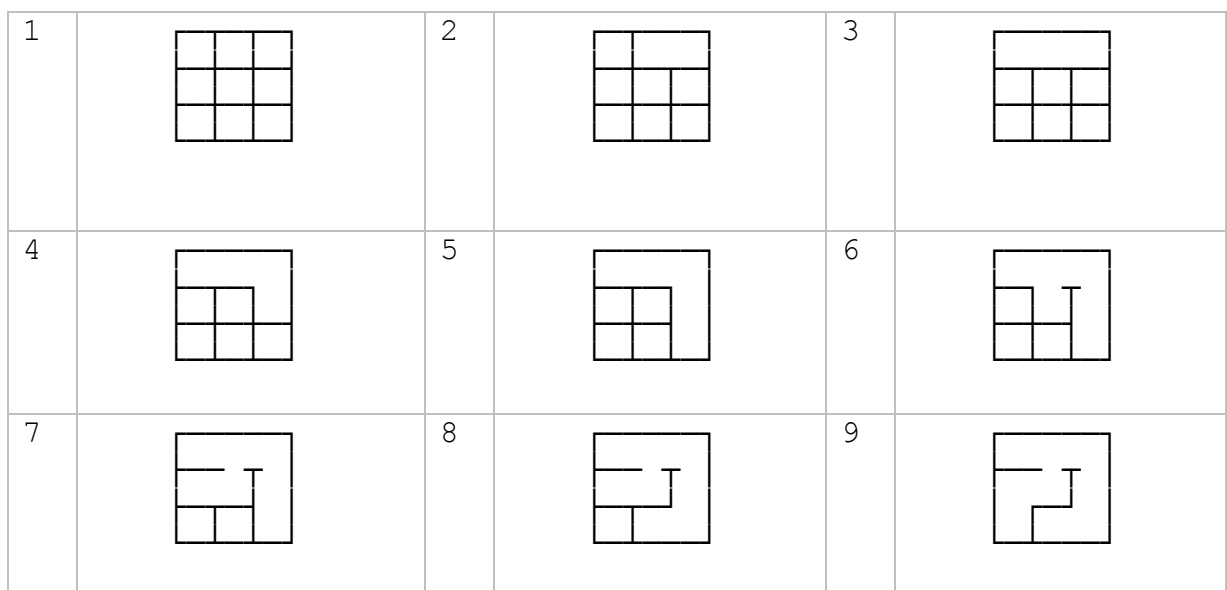


Рисунок 11 – Пример работы алгоритма Двоичное дерево

4.1.4.4. Примеры лабиринтов

Лабиринты, сгенерированные этим алгоритмом, имеют заметную на глаз смещенность.

Приведем сгенерированные лабиринты (рисунок 12-15) для различных выбранных направлений:

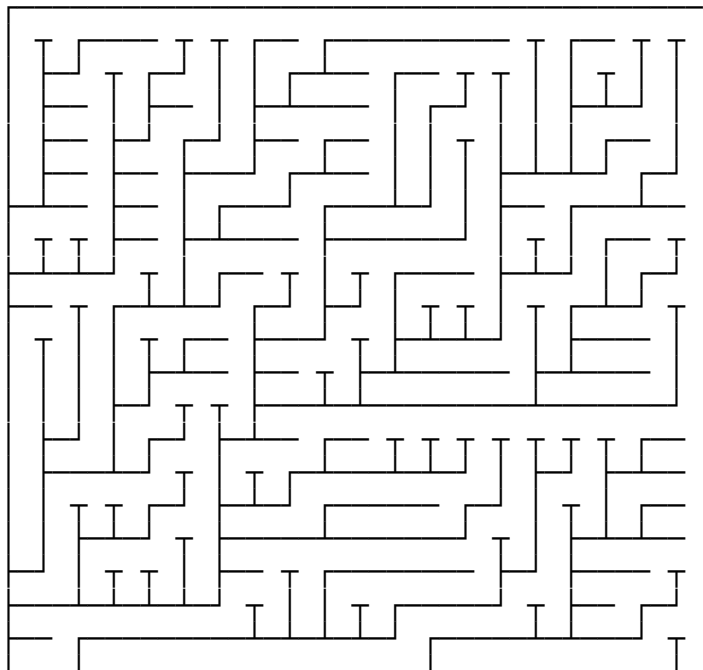


Рисунок 12 – Северо-Восток

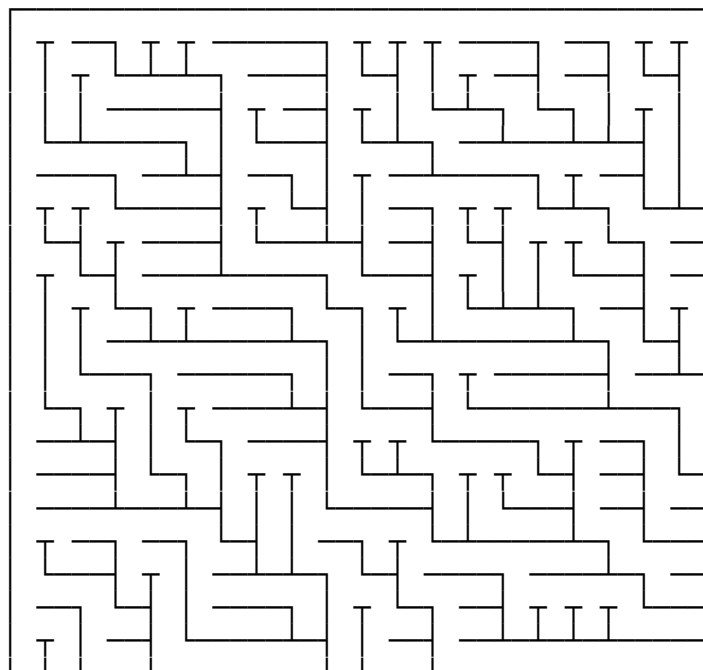


Рисунок 13 – Северо-Запад

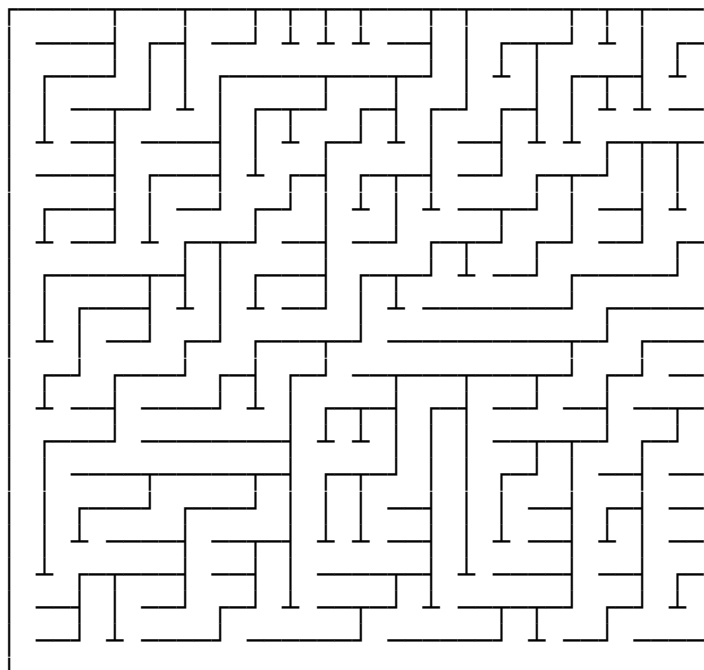


Рисунок 14 – Юго-Запад

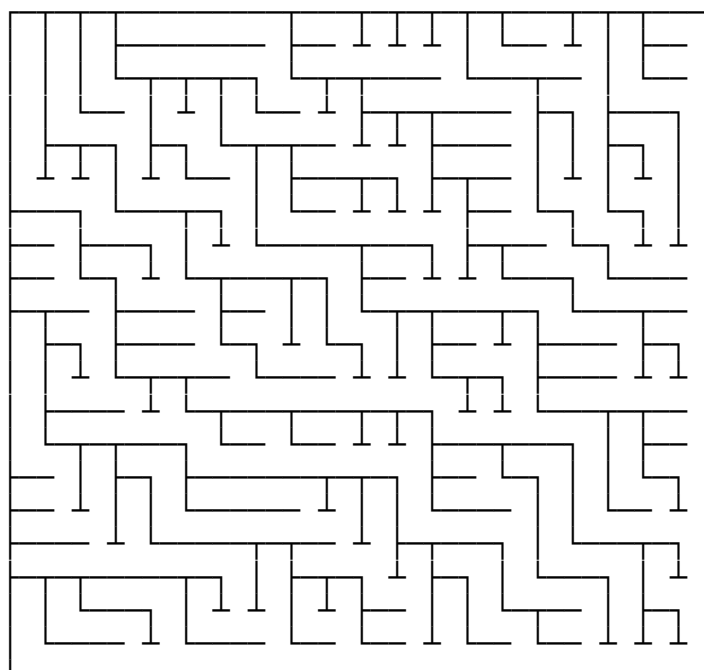


Рисунок 15 – Юго-Восток

4.2. Алгоритмы поиска пути в лабиринте

4.2.1. Подготовка лабиринта

Алгоритмы генерации, реализованные ранее, создают идеальные лабиринты. Это значит, что между любыми двумя ячейками существует путь, причем единственный. При поиске путей более интересной, на мой взгляд, является ситуация, когда между ячейками могут существовать альтернативные пути, причем стоимость (вес) пути между различными ячейками может быть различным.

Поэтому был создан класс `mazeWeighted` и написаны следующие функции:

`WallsReduce` – для разряжения лабиринта, т.е. случайного удаления его стен с заданной вероятностью.

```
void RandomCircles(mazeWeighted& MazeWeighted, std::default_random_engine&
generator, int minWeight, int maxWeight, double probability, int meanRadius,
double stddevRadiusRatio)
```

– для генерации весов лабиринта в виде кругов со случайным радиусом (имеющим нормальное распределение вероятности) и находящихся в случайных ячейках. Математическое ожидание радиуса задается с помощью `meanRadius`, `stddevRadiusRatio` – среднеквадратическое отклонение радиуса круга, деленное на математическое ожидание круга. Примеры получившихся лабиринтов можно видеть в примерах алгоритмов поиска.

4.2.2. Ли

4.2.2.1. Теоретические сведения

Алгоритм волновой трассировки (волновой алгоритм, алгоритм Ли) [4] — алгоритм поиска пути, алгоритм поиска кратчайшего пути на планарном графе. Принадлежит к алгоритмам, основанным на методах поиска в ширину.

В основном используется при компьютерной трассировке (разводке) печатных плат, соединительных проводников на поверхности микросхем. Другое применение волнового алгоритма — поиск кратчайшего расстояния на карте в компьютерных стратегических играх.

Волновой алгоритм в контексте поиска пути в лабиринте был предложен Э. Ф. Муром. Ли независимо открыл этот же алгоритм при формализации алгоритмов трассировки печатных плат в 1961 году.

Алгоритм работает на дискретном рабочем поле (ДРП), представляющем собой ограниченную замкнутой линией фигуру, не обязательно прямоугольную, разбитую на прямоугольные ячейки, в частном случае — квадратные. Множество всех ячеек ДРП разбивается на подмножества: «проходимые» (свободные), т. е. при поиске пути их можно проходить, «непроходимые» (препятствия), путь через эту ячейку запрещен, стартовая ячейка (источник) и финишная (приемник). Назначение стартовой и финишной ячеек условно, достаточно — указание пары ячеек, между которыми нужно найти кратчайший путь.

Алгоритм предназначен для поиска кратчайшего пути от стартовой ячейки к конечной ячейке, если это возможно, либо, при отсутствии пути, выдать сообщение о непроходимости.

Работа алгоритма включает в себя три этапа: инициализацию, распространение волны и восстановление пути.

4.2.2.2. Алгоритм

В данной работе дискретное поле не содержит непроходимых ячеек, однако между ячейками на этом поле могут быть стены. Стоит отметить, что в данной реализации не используется очередь, характерная для алгоритмов поиска в ширину. Что, предположительно, может являться причиной того, что этот алгоритм более быстрый, чем алгоритм Дейкстры (который, будучи примененным для невзвешенных полей, должен давать аналогичный результат за, возможно, чуть большее время) на полях небольших размеров и значительно более медленный на полях больших размеров. (Вместо извлечения элементов из очереди алгоритм пробегает ячейки в прямоугольной области, ограниченной ячейками фронта волны)

- 1) Вводится матрица весов путей (стоимости путей от стартовой ячейки до текущей)
- 2) Матрица так же используется для отличия посещенных ячеек от непосещенных (для непосещенных ячеек устанавливается вес, максимальный для int на данной платформе)
- 3) Начинается распространение волны от стартовой ячейки: ей присваивается нулевой вес.
- 4) Далее идет распространение волны: для каждой ячейки фронта волны просматриваются соседи. Если соседи ещё не были посещены, то им присваивается вес. Также отслеживается, достигнута ли финишная ячейка и удалось ли продвинуться фронту волны.
- 5) Если финишная ячейка достигнута, либо же фронт волны больше не может распространяться, то происходит завершение распространения волны. Если завершение было связано с достижением финишной ячейки, то начинается восстановление пути, в противном случае выдается сообщение о несуществовании пути, и происходит выход из функции.
- 6) Восстановление пути идет из финишной в начальную ячейку. Для того, чтобы восстановить путь нужно двигаться из финишной в начальную ячейку так, чтобы при движении вес пути уменьшался на единицу.

4.2.2.3. Пример работы

Приведем пример (рисунок 16), когда путь существует. (цифры – веса пути; ~ - непосещенные ячейки)

Сначала идет поиск пути из (0, 1) в (1, 1)

1) Потом волна распространяется от ячейки (0, 1).

2) И затем происходит восстановление пути с конца: (1, 1) -> (1, 0) (1, 1) -> (0, 0) (1, 0) (1, 1) -> (0, 1) (0, 0) (1, 0) (1, 1)

(0, 1) (0, 0) (1, 0) (1, 1) – искомый путь

1		2		3	
4		5		6	
7		8			

Рисунок 16 – Пример работы алгоритма Ли

4.2.3. Ли (модификация с двумя волнами)

Более сложная реализация алгоритма Ли [4]. Отличие заключается в том, что помимо волны, распространяющейся от стартовой ячейки, создается волна, распространяющаяся от финишной ячейки. Фронты волн продвигаются по дискретному полю поочередно, пока волны не встретятся, или пока одна из волн не сможет продвигаться дальше. В этой реализации помимо матрицы весов путей вводится матрица идентификаторов волн.

Если волны встретились, то путь восстанавливается в два этапа: сначала от стартовой ячейки до ячейки пересечения волн, затем от ячейки пересечения волн до финишной ячейки.

Данная реализация позволяет посещать примерно в два раза меньшее количество ячеек, чем вариант с одной волной.

4.2.3.1. Пример работы

Пример работы рассматриваемого алгоритма приведен на рисунке 17.

1) Происходит поиск пути от (0, 1) до (2, 4). Зеленым цветом обозначена волна от стартовой ячейки, голубой – от финишной.

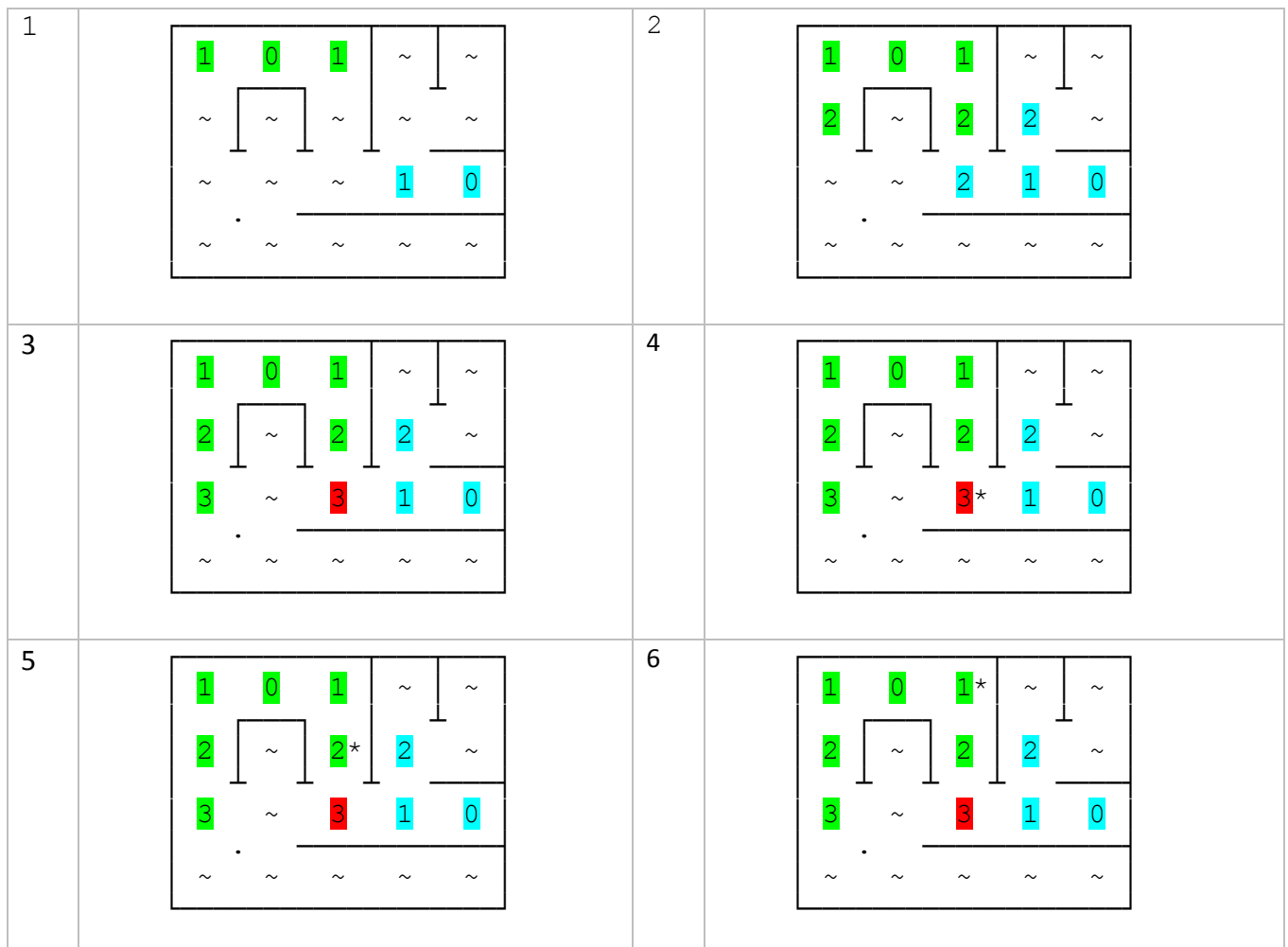
3) Как мы видим, волны пересеклись в ячейке (2, 2). Обозначим её красным цветом.

4) Затем восстанавливаем путь от стартовой ячейки до ячейки пересечения:

(2, 2) -> (1, 2) (2, 2) -> (0, 2) (1, 2) (2, 2) -> (0, 1) (0, 2) (1, 2) (2, 2)

8) Далее восстанавливаем путь от ячейки пересечения до финишной ячейки, находя искомый путь:

(0, 1)(0, 2)(1, 2)(2, 2)(2, 3) -> (0, 1)(0, 2)(1, 2)(2, 2) (2, 3) (2, 4) – наш искомый путь



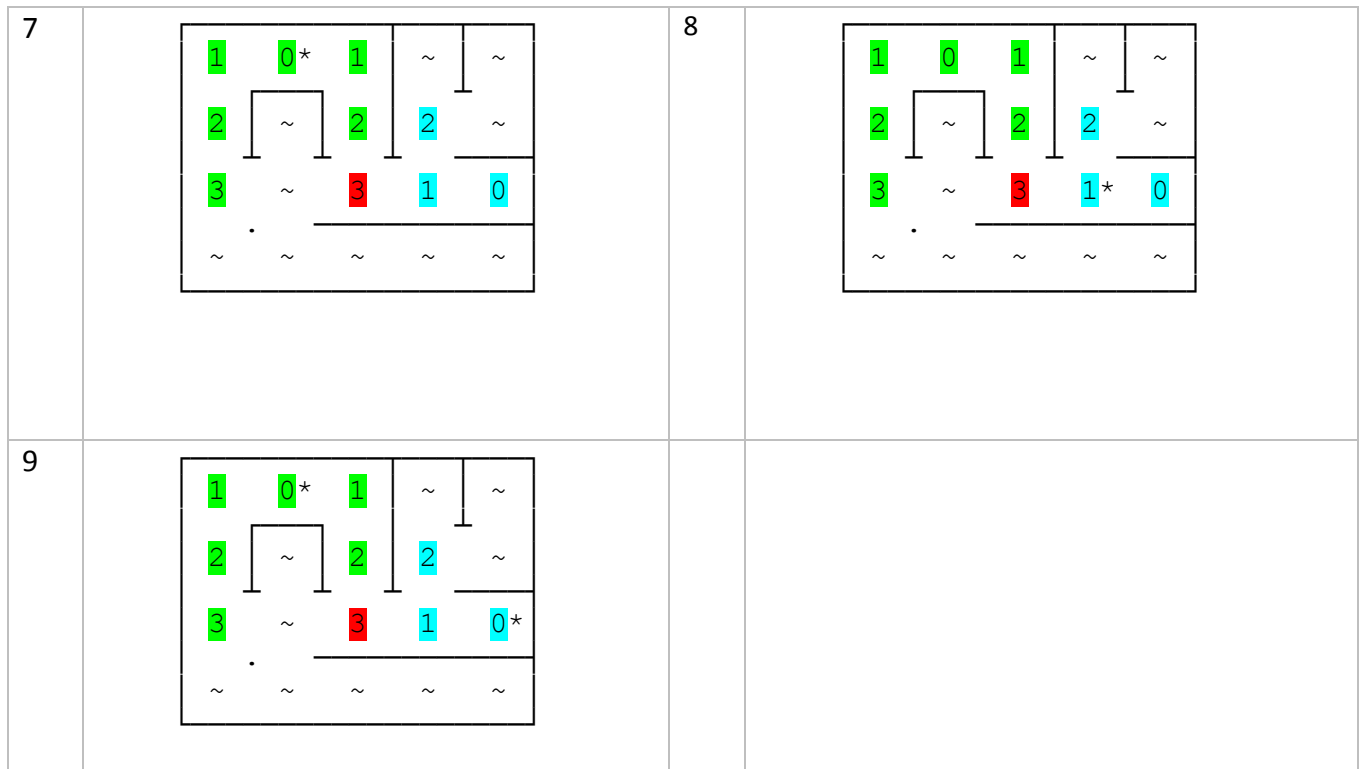


Рисунок 17 – Пример работы модифицированного алгоритма Ли с двумя волнами

4.2.4. Дейкстры

4.2.4.1. Алгоритм

Алгоритм Дейкстры – это алгоритм на графе. На ДРП этот алгоритм применяется, когда веса (стоимости) перехода (не путать с путевыми весами) из одной ячейки в другую могут быть различны. Если все веса одинаковы, то алгоритм Дейкстры будет давать тот же результат, как и поиск в ширину.

Основное отличие алгоритма Дейкстры от алгоритма поиска в ширину – использование очереди с приоритетами.

Итак, при работе алгоритма будут введены: PathCoordWeights – матрица, каждый элемент которой – это путевые координаты ячейки и её путевой вес.

PriorQueue – очередь с приоритетами, меньший вес – больший приоритет. Будет хранить в себе координаты ячеек.

- 1) Устанавливаем путевой вес стартовой ячейки равным 0.
- 2) Помещаем стартовую ячейку в очередь
- 3) Извлекаем ячейку из очереди
- 4) Проверяем всех её соседей (всего 4 соседа): Если сосед ни разу не был посещен, то устанавливаем для него путевые координаты (с какого направления к нему пришли) и путевые веса. Если сосед уже был посещен, то находим для него новый путевой вес и сравниваем с текущим. Если новый вес оказался не меньше текущего, то не делаем ничего, если же новый вес оказался меньше текущего, то посещенному соседу присваиваем новый вес, новые путевые координаты и помещаем в очередь (он оказывается в ней уже повторно)
- 5) Продолжаем выполнять пункты 3-4, пока извлеченный элемент не окажется финишной ячейкой, либо пока в очереди не останется элементов.
- 6) Узнав причину остановки, делаем вывод о том, существует ли путь или нет.

7) Если путь существует, то восстанавливаем его, двигаясь по путевым координатам от финиша к старту.

4.2.4.2. Пример работы

Приведем исходные данные для лабиринта (рисунок 18).

Цифры означают веса ячеек, т.е. стоимость при перемещении в данную ячейку из соседней. Найдем путь из ячейки (0, 0) в ячейку (2, 2). Они отмечены желтым.

1) Выставим атрибуты первой ячейки и поместим её в очередь.

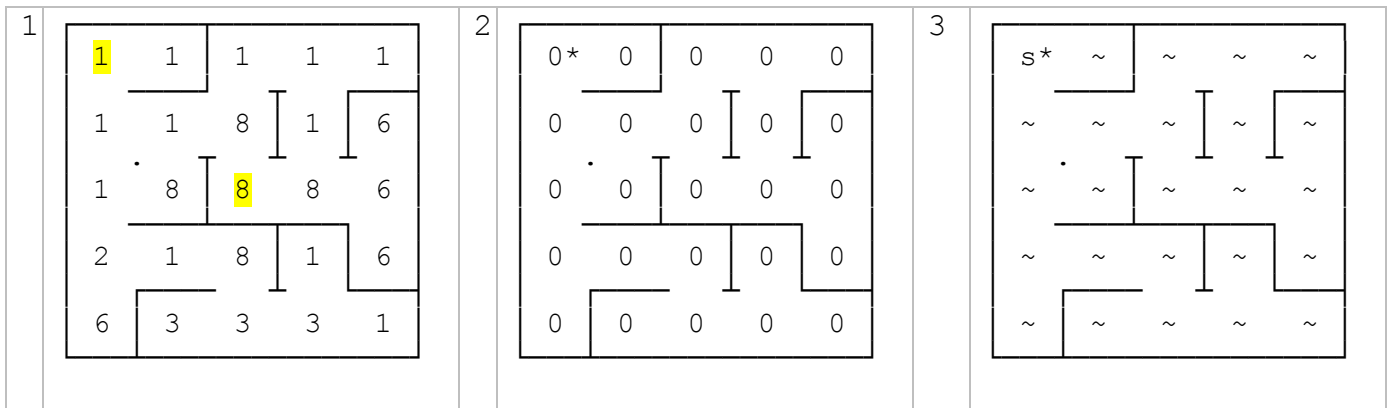


Рисунок 18 – Пример работы алгоритма Дейкстры:

исходный лабиринт, путевые веса и путевые координаты

2) Извлечем эту ячейку из очереди и пройдемся по её соседям. Вначале посмотрим левого соседа (рисунок 19.1-19.2), а потом нижнего (рисунок 19.3-19.4):

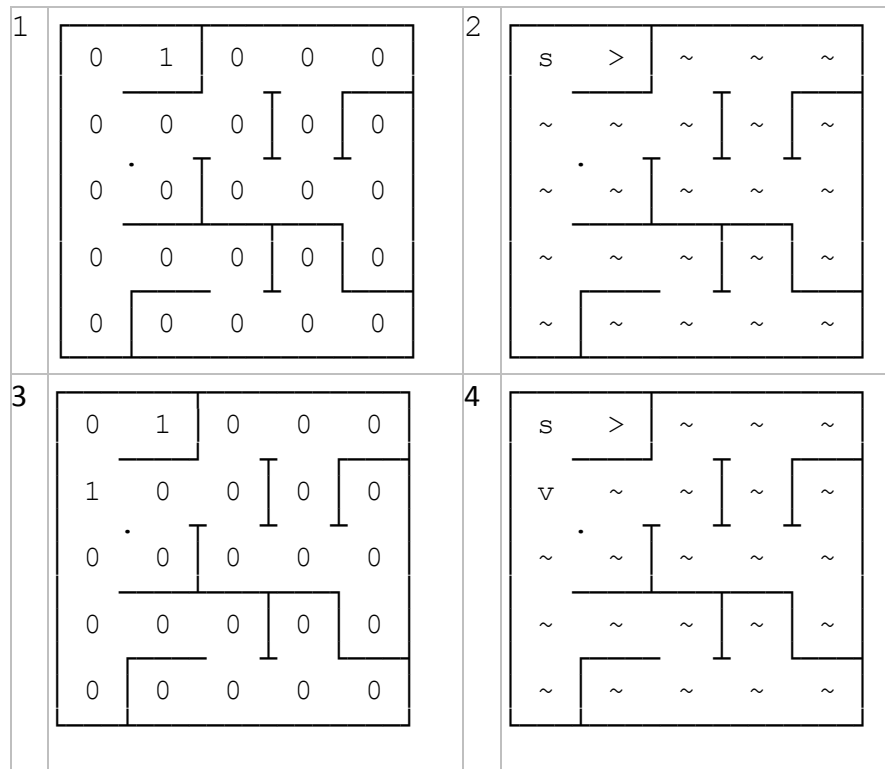


Рисунок 19 – Пример работы алгоритма Дейкстры:

извлечение стартовой ячейки из очереди

3) Теперь в очереди находятся 2 ячейки с одинаковым приоритетом 1. Это ячейки (0, 1) и (1, 0). Извлечем из очереди ячейку (0, 1). Она имеет единственного соседа, неотделенного стеной – это ячейка (0, 0). Найдем новый вес для неё (рисунок 20.1-20.2): новый путевой вес

$(0, 0) = \text{путевой вес}(0, 1) + \text{вес перемещения}(0, 0) = 1 + 1 = 2$ Это больше, чем 0. Значит, с ячейкой $(0, 0)$ мы ничего не делаем.

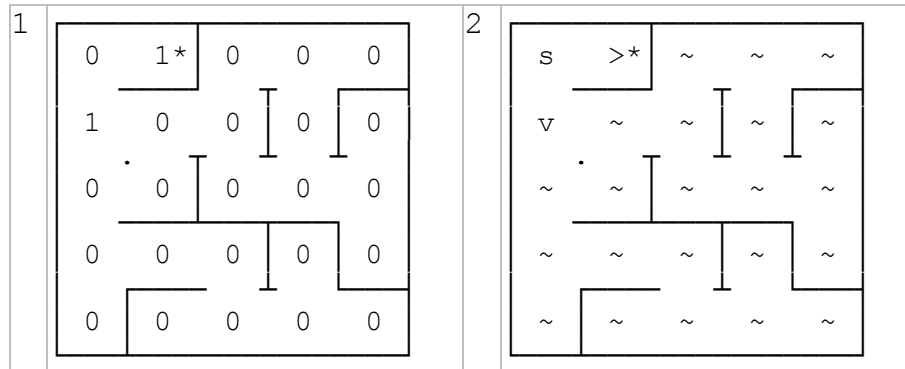


Рисунок 20 – Пример работы алгоритма Дейкстры: работа с ячейкой $(0,1)$

4) Теперь в очереди находится единственная ячейка $(1, 0)$. Она имеет трех соседей. Обойдем их последовательно. У нас получились следующие путевые веса и координаты (рисунок 21):

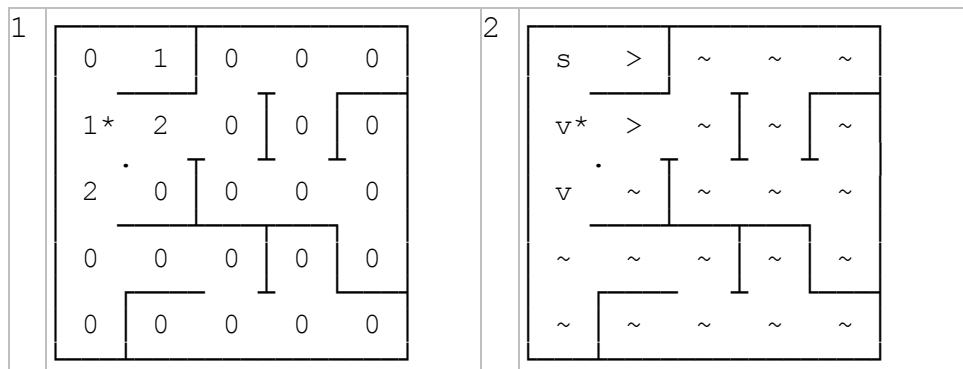


Рисунок 21 – Пример работы алгоритма Дейкстры: работа с ячейкой $(1,0)$

5) В очереди теперь две ячейки $(1, 1)$ и $(2, 0)$. Обе эти ячейки имеют приоритет 2. Извлечем из очереди ячейку $(1, 1)$ и аналогично пройдемся по её соседям (рисунок 22).

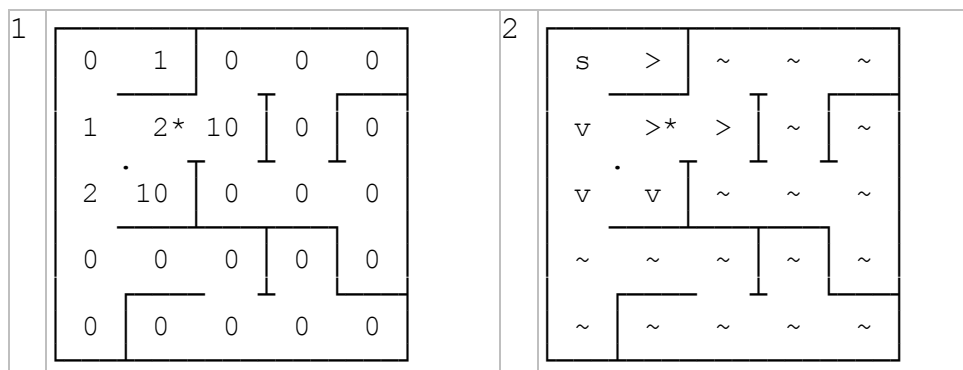


Рисунок 22 – Пример работы алгоритма Дейкстры: работа с ячейкой $(1,1)$

6) Теперь в очереди лежит по прежнему ячейка $(2, 0)$ с приоритетом 2 и лежат ячейки $(1, 2)$ и $(2,1)$ с приоритетами 10. Выберем наиболее приоритетную ячейку, т.е. ячейку $(2, 0)$. Проходимся по её соседям. В итоге у нас получаются следующие путевые веса и координаты (рисунок 23):

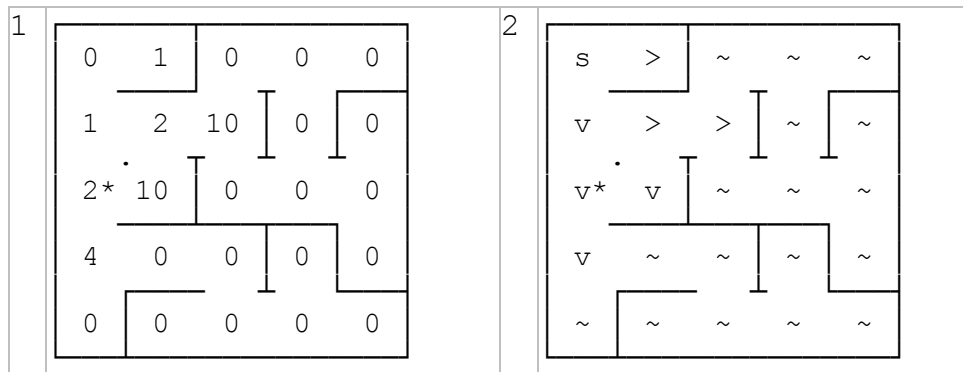


Рисунок 23 – Пример работы алгоритма Дейкстры:

получившиеся путевые веса и координаты после извлечения ячейки (2,0) из очереди

7) На этом шаге мы уже добрались до финишной ячейки (рисунок 24). Но она не самая приоритетная в очереди, поэтому поиск продолжается (существуют случаи, когда удастся найти путь короче, чем тот, который бы мы получили, сразу остановив поиск. Более того, когда мы дождемся, когда «фронт волны» полностью прокатится через полученный вес, то найденный путь окажется наикратчайшим из возможных. Т.е. алгоритм Дейкстры (в данной интерпретации, т.е. когда мы можем многократно ставить в очередь уже просмотренные элементы) гарантированно находит наикратчайший путь.). Продолжаем далее аналогично вытаскивать ячейки из очереди и просматривать их соседей, пока не найдем приоритетную финишную ячейку в очереди (рисунок 24.1-24.2).

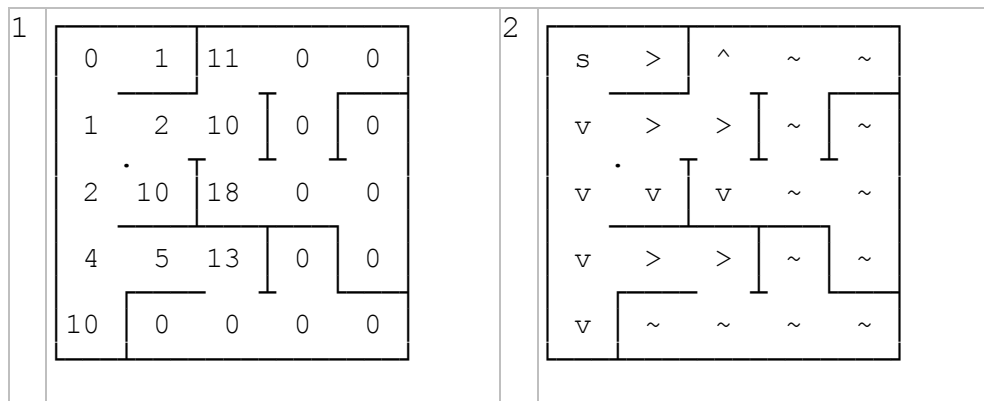


Рисунок 24 – Нахождение приоритетной финишной ячейки в очереди

8) Наконец, очередной элемент, извлеченный из очереди, оказывается финишной ячейкой. На этом поиск прекращается (рисунок 25.1-25.2), и по путевым координатам восстанавливается путь из конца в начало (рисунок 26):

(2, 2) -> (1, 2)(2, 2) -> (1, 1)(1, 2)(2, 2) -> (1, 0)(1, 1)(1, 2)(2, 2) -> (0, 0)(1, 0)(1, 1)(1, 2)(2, 2)

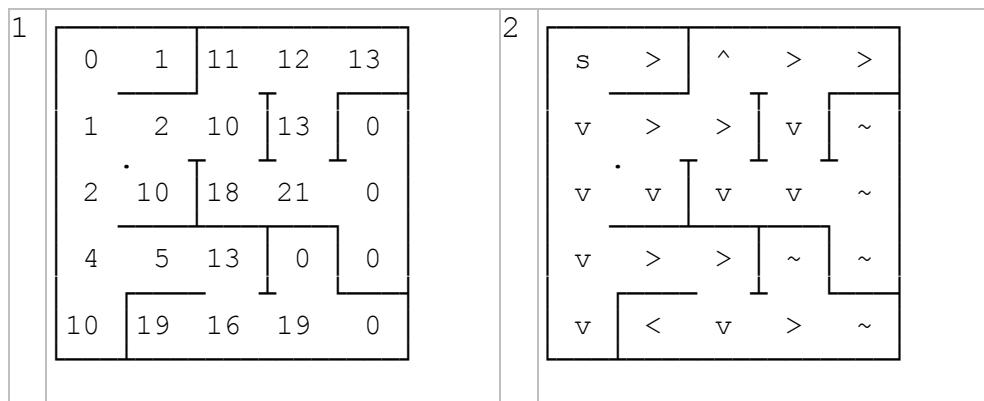


Рисунок 25 – Остановка поиска после извлечения из очереди финишной ячейки

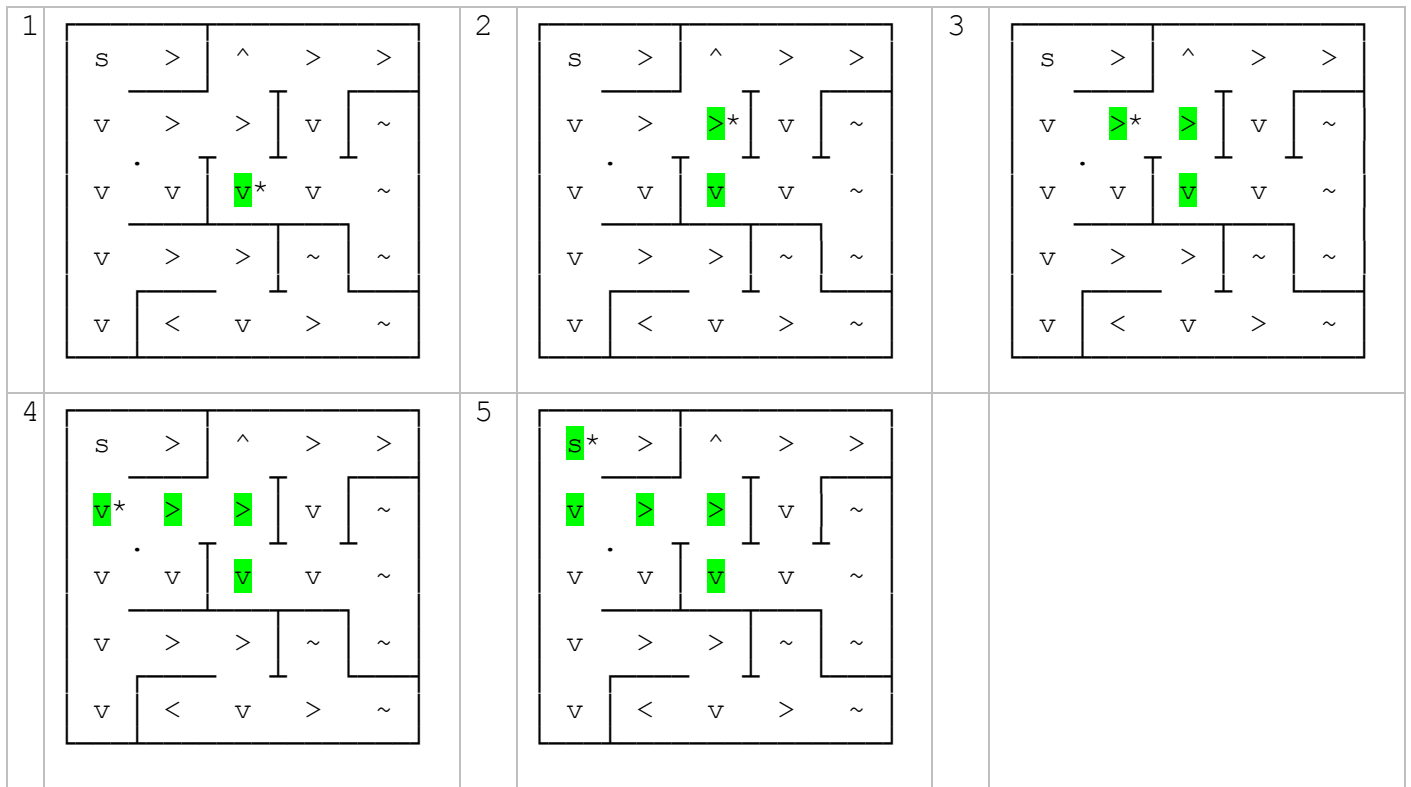


Рисунок 26 – Пример работы алгоритма Дейкстры:
восстановление пути из финишной ячейки в начало

Вершина очереди при «распространении волны» на каждой итерации выглядит так (приоритет_он_же_путевой_вес, (l, j)):

```
PriorQueue all items:
PriorQueue.top()=(0, (0, 0))
PriorQueue.top()=(1, (0, 1))
PriorQueue.top()=(1, (1, 0))
PriorQueue.top()=(2, (1, 1))
PriorQueue.top()=(2, (2, 0))
PriorQueue.top()=(4, (3, 0))
PriorQueue.top()=(5, (3, 1))
PriorQueue.top()=(10, (1, 2))
PriorQueue.top()=(10, (2, 1))
PriorQueue.top()=(10, (4, 0))
PriorQueue.top()=(11, (0, 2))
PriorQueue.top()=(12, (0, 3))
PriorQueue.top()=(13, (0, 4))
PriorQueue.top()=(13, (1, 3))
PriorQueue.top()=(13, (3, 2))
PriorQueue.top()=(16, (4, 2))
PriorQueue.top()=(18, (2, 2))
```

4.2.5. A* (AStar; A со звездой)

4.2.5.1. Теоретические сведения

Поиск A* (произносится «А звезда» или «А стар», от англ. A star) [1] — в информатике и математике, 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 «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 Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем. Это по сути было расширение алгоритма Дейкстры, созданного в 1959 году. Новый алгоритм достигал более высокой производительности (по времени) с помощью эвристики. В их работе он упоминается как «алгоритм А». Но так как он вычисляет лучший маршрут для заданной эвристики, он был назван A^* .

Обобщением для него является двунаправленный эвристический алгоритм поиска.

4.2.5.2. Алгоритм

Реализация данного алгоритма мало отличается от реализации алгоритма Дейкстры.

Для оценки расстояния от текущей ячейки до финишной вводится эвристическая функция, состоящая из одного оператора:

```
return abs(finishi-Currenti)+abs(finishj-Currentj);
```

Отличие от алгоритма Дейкстры в том, что в качестве приоритета в очереди берутся не путевые веса, а сумма путевого веса и эвристической оценки расстояния до финишной ячейки.

4.2.5.3. Пример работы

Покажем на примере исходного лабиринта (рисунок 27) работу алгоритма.

1	1	1	1	1
1	1	8	1	6
1	8	8	8	6
2	1	8	1	6
6	3	3	3	1

Рисунок 27 – Пример работы алгоритма A^* : исходный лабиринт с весами ячеек

Вершина очереди на каждой итерации:

```
PriorQueue all items:
PriorQueue.top()=(4, (0, 0))
PriorQueue.top()=(4, (0, 1))
PriorQueue.top()=(4, (1, 0))
PriorQueue.top()=(4, (1, 1))
PriorQueue.top()=(4, (2, 0))
PriorQueue.top()=(7, (3, 0))
PriorQueue.top()=(7, (3, 1))
PriorQueue.top()=(11, (1, 2))
PriorQueue.top()=(11, (2, 1))
PriorQueue.top()=(13, (0, 2))
PriorQueue.top()=(14, (3, 2))
PriorQueue.top()=(14, (4, 0))
```

```

PriorQueue.top()=(15, (0, 3))
PriorQueue.top()=(15, (1, 3))
PriorQueue.top()=(17, (0, 4))
PriorQueue.top()=(18, (2, 2))

```

Как видим, путевые веса (рисунок 28) аналогичны путевым весам в алгоритме Дейкстры, но приоритеты внутри очереди стали другими. Например, ячейка (4, 2) имела в очереди приоритет 18, т.е. такой же, как и у финишной ячейки (в вершину очереди эта ячейка не попала). При сравнении с алгоритмом Дейкстры можно увидеть, что при работе A* из очереди было извлечено число ячеек на один меньше, чем при работе алгоритма Дейкстры.

0	1	11	12	13
1	2	10	13	0
2	10	18	21	0
4	5	13	0	0
10	0	16	0	0

Рисунок 28 – Пример работы алгоритма A*: путевые веса

Построение пути (рисунок 29) для A* аналогично построению пути в алгоритме Дейкстры.

s	>	^	>	>
v	>	>	v	~
v	v	v	v	~
v	>	>	~	~
v	~	v	~	~

Рисунок 29 – Пример работы алгоритма A*: путевые координаты

Достигнув финишной ячейки, мы находим искомый путь:

Искомый путь = (0,0)(1,0)(1, 1) (1, 2) (2, 2)

5. РЕЗУЛЬТАТЫ РАБОТЫ

5.1. Графики

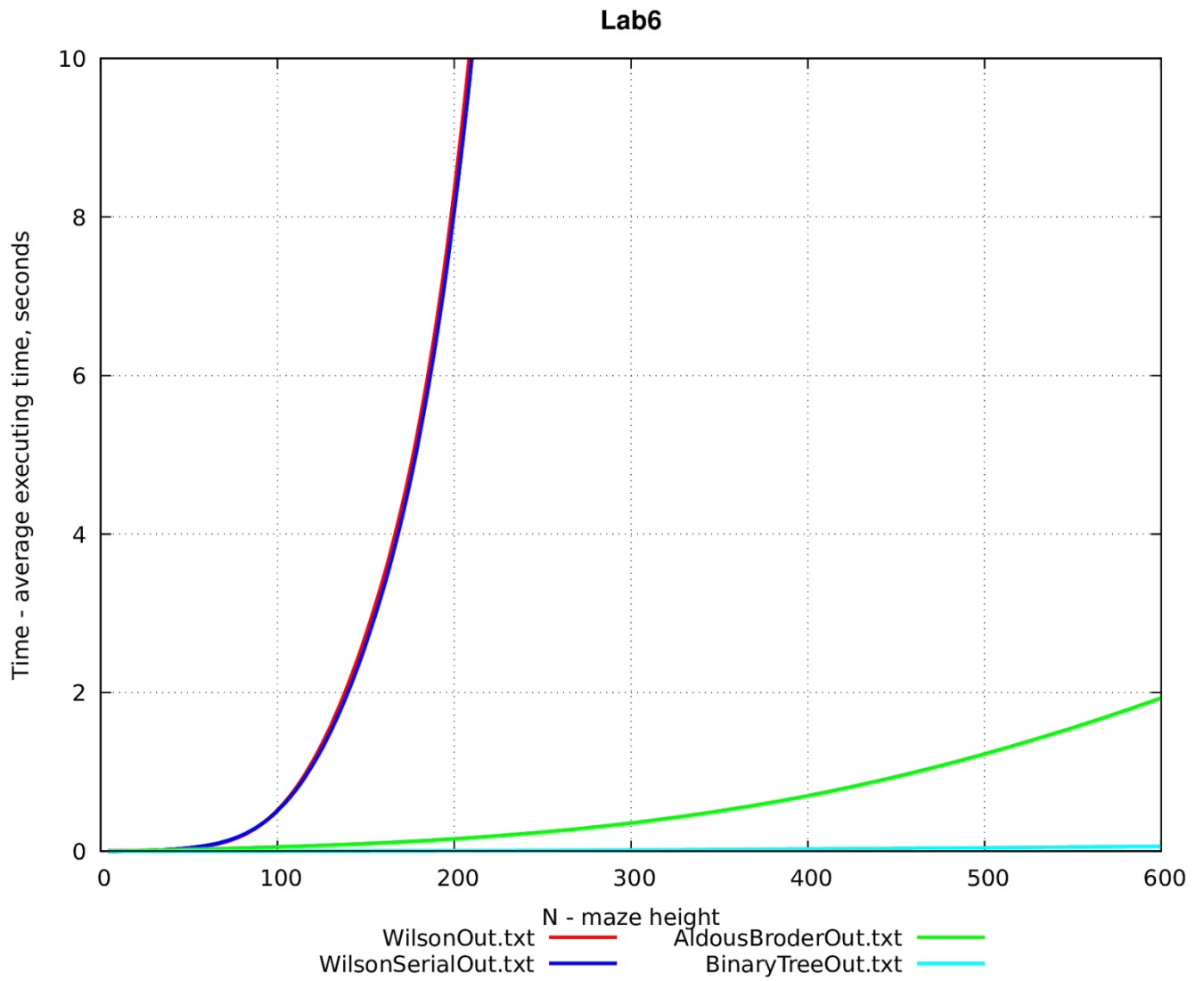


Рисунок 30 – Время генерации лабиринта от высоты лабиринта

Lab6

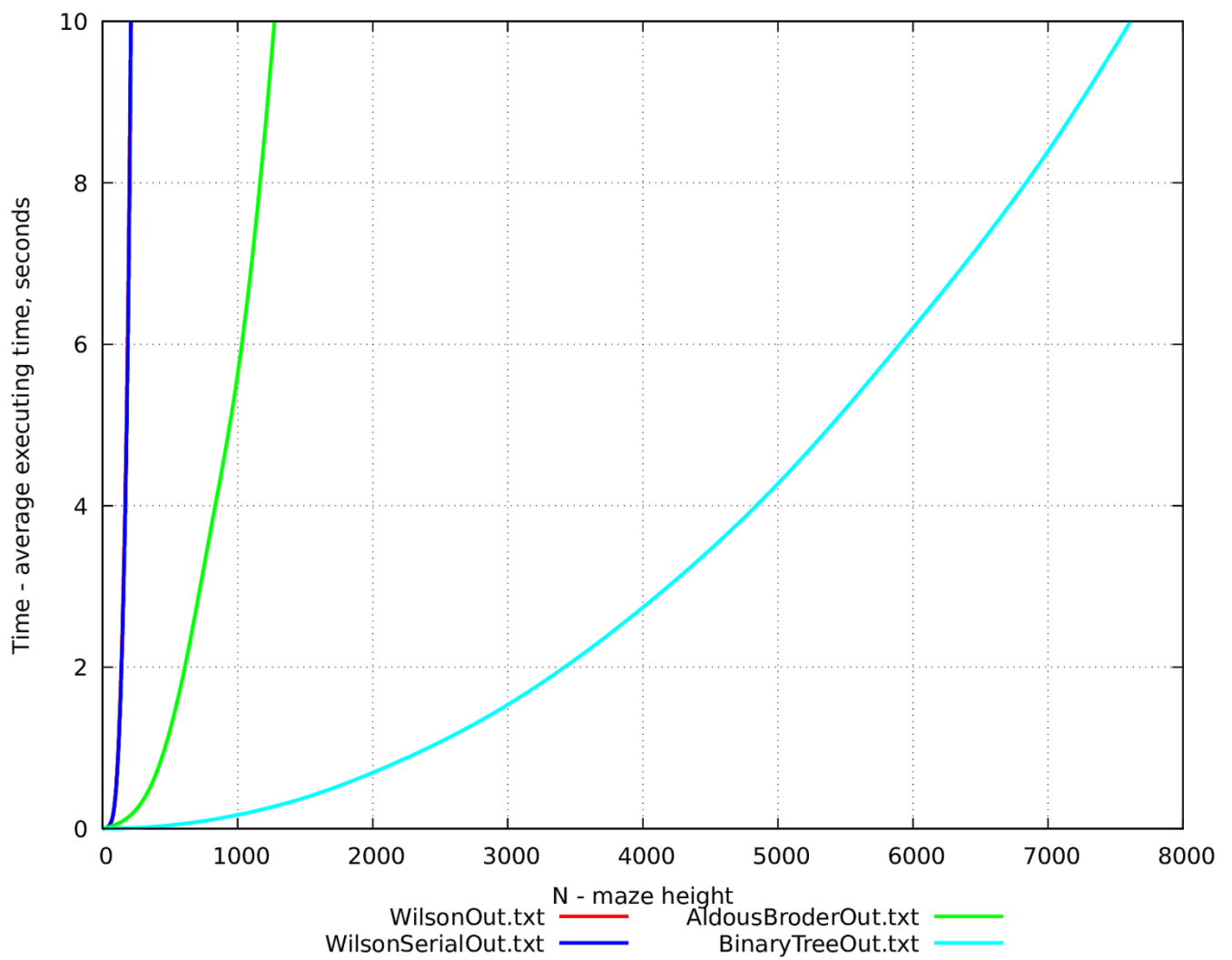


Рисунок 31 – Время генерации лабиринта от высоты лабиринта

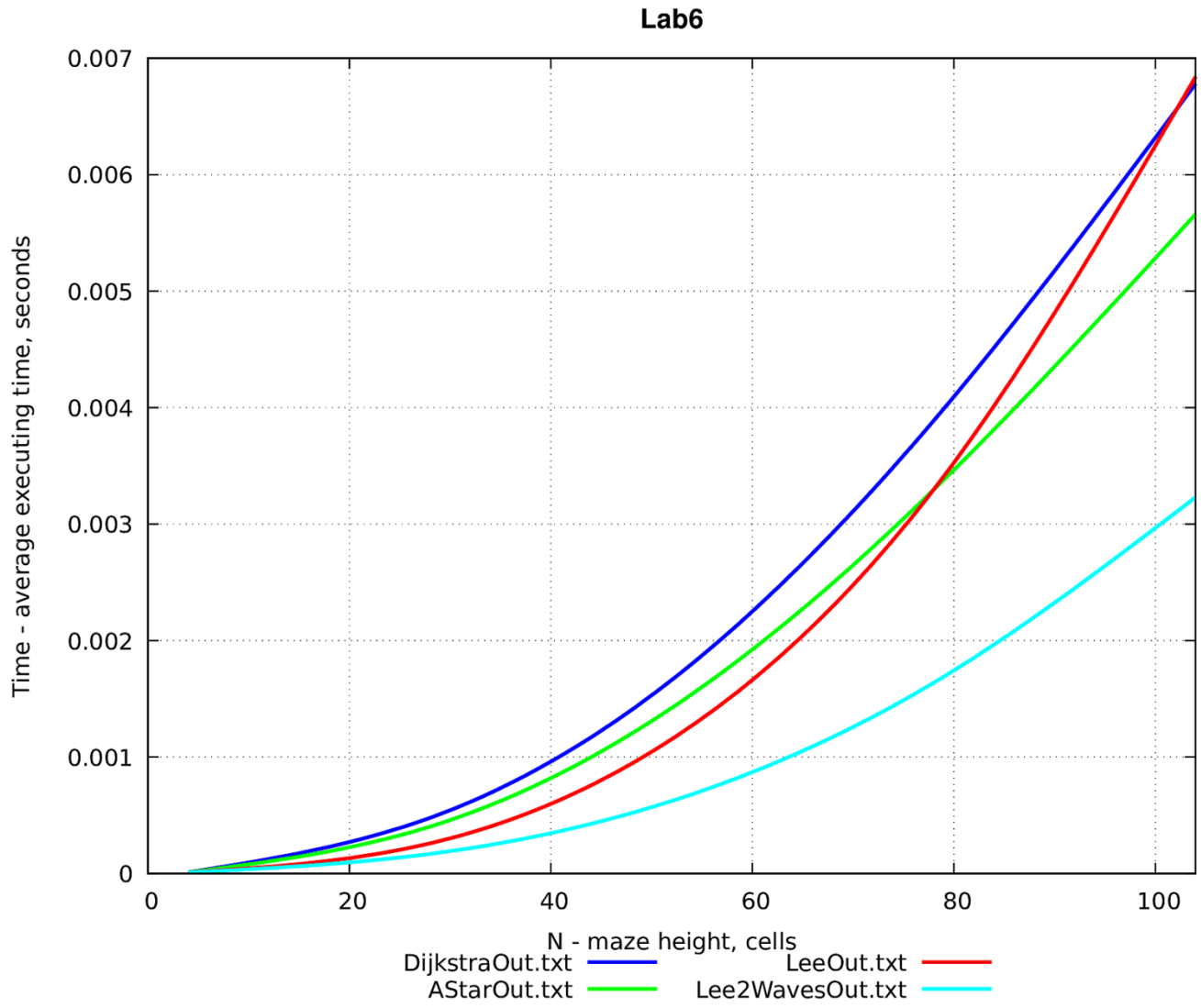


Рисунок 32 – Время поиска в лабиринте от высоты лабиринта

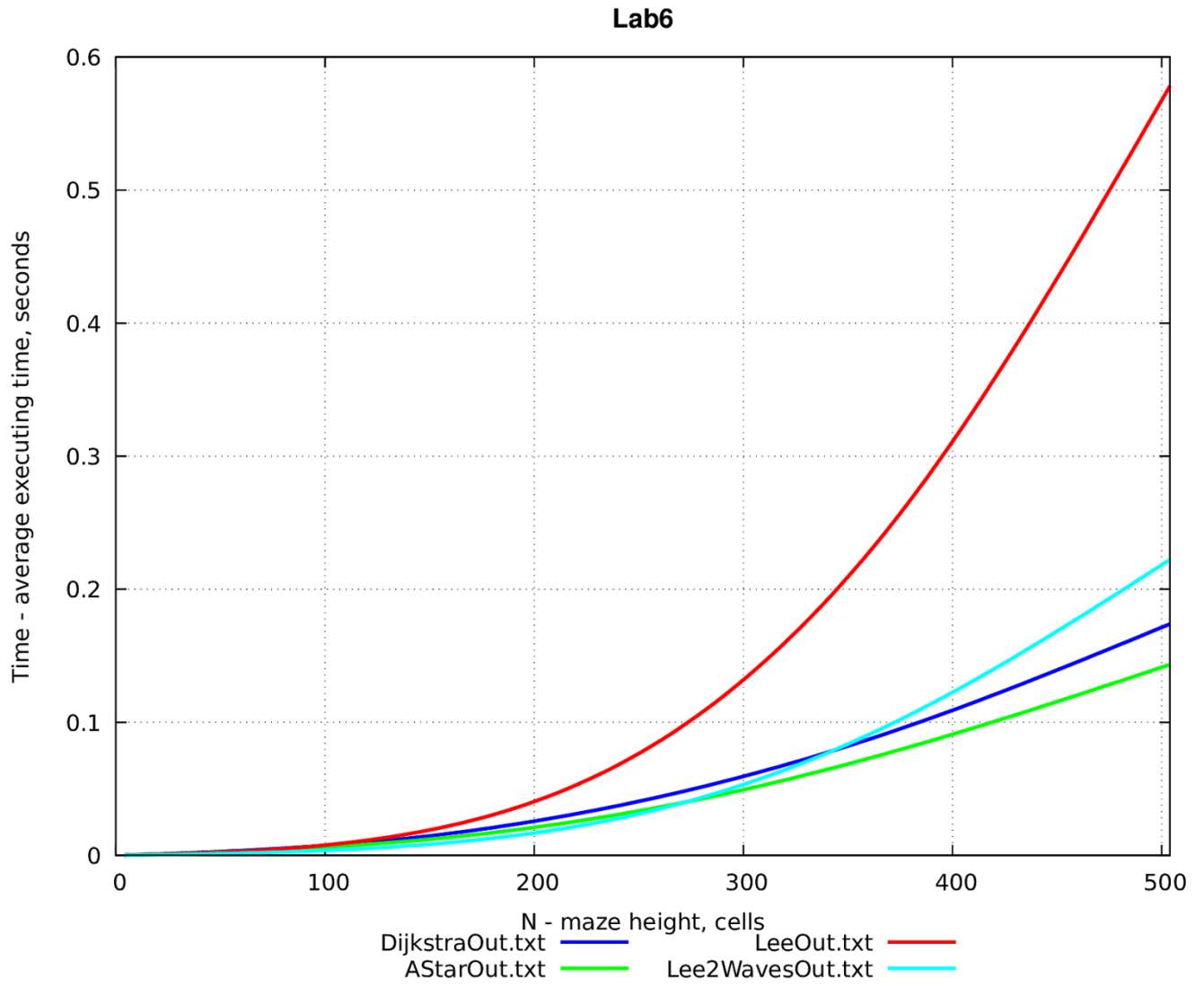


Рисунок 33 – Время поиска в лабиринте от высоты лабиринта

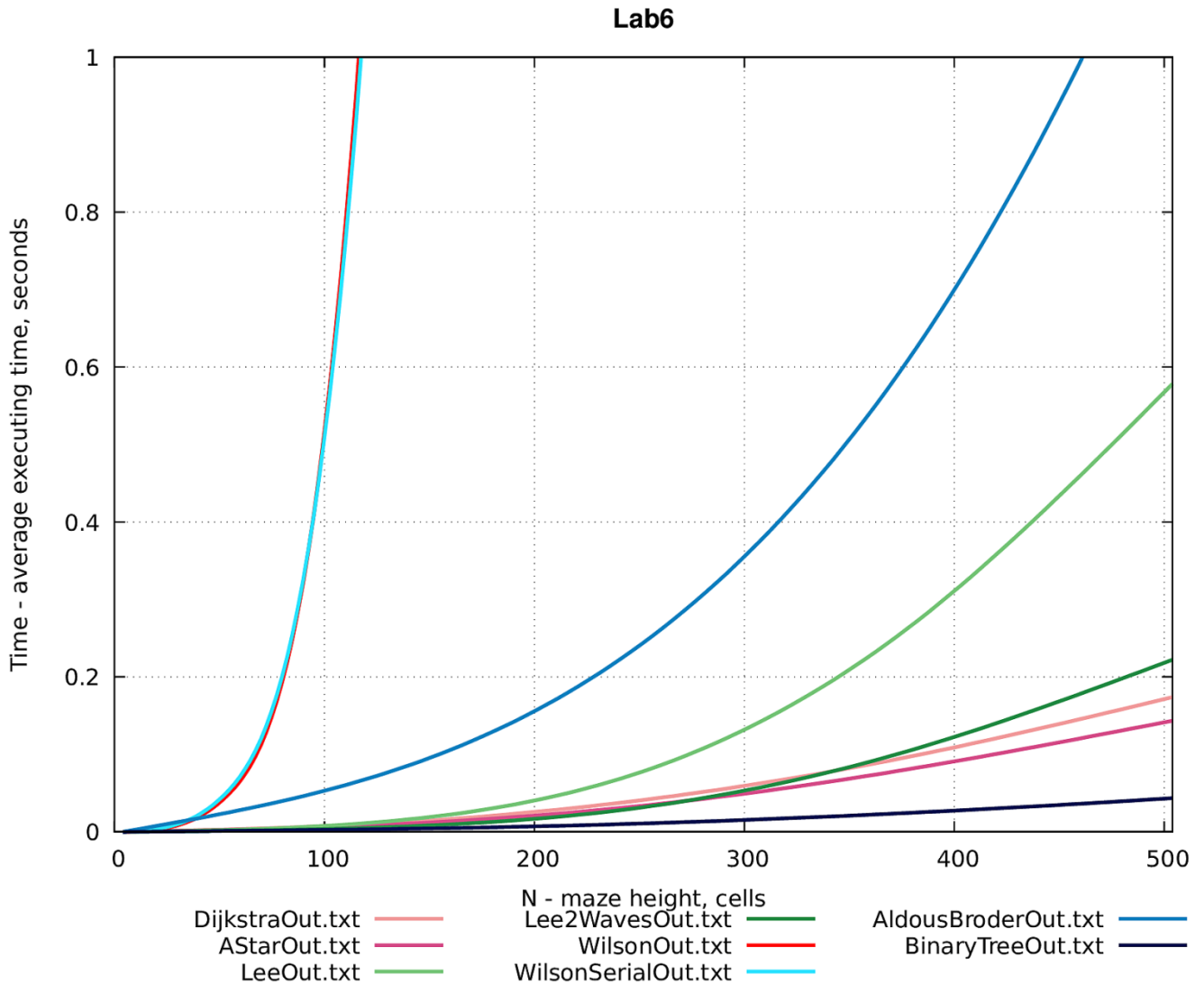


Рисунок 34 – Сравнение времени генерации и времени поиска в лабиринтах

5.2. Анализ графиков

5.2.1. Алгоритмы генерации лабиринтов

Как видно из рисунков 30 и 31, Алгоритм Уилсона оказался самым медленным из всех, сильно уступая даже алгоритму Олдоса-Бродера, который является более «глупым» (за 10 секунд алгоритм Олдоса-Бродера успевает сгенерировать примерно в 25 раз больше ячеек, чем алгоритм Уилсона). Это, по-видимому, связано с тем, что срезая петли, алгоритм многократно проходится по одним и тем же ячейкам, в отличие от Олдоса-Бродера, который добавляет в лабиринт все ячейки, на которые наткнется.

Его модификация с последовательным выбором ячеек, не входящих в UST, как оказалось, практически не влияет на время его работы.

Алгоритм Двоичного дерева оказался намного быстрее алгоритмов Уилсона (за 10 секунд он успевает сгенерировать примерно в 900 раз больше ячеек!) и Олдоса-Бродера (за 10 секунд в 36 раз больше ячеек). Но, как известно, лабиринты, сгенерированные им, имеют сильную смещенность и неоднородность, что заметно невооруженным взглядом.

5.2.2. Алгоритмы поиска в лабиринтах

Как видно из рис 32-33, Модификация Ли с двумя волнами работает примерно в 2 раза быстрее, чем вариант с одной волной. Причем при больших значениях лабиринта выигрыш во времени увеличивается.

Алгоритм А* быстрее алгоритма Дейкстры в 1,19 раз для высоты лабиринта в 104 ячейки и в 1,21 раз для высоты лабиринта в 504 ячейки.

Алгоритм Ли быстрее алгоритма А* на лабиринтах с высотой до 80 ячеек и быстрее алгоритма Дейкстры на лабиринтах с высотой до 100 ячеек. Причем при высоте в 504 ячейки алгоритм Ли становится медленнее в 3,32 раза.

Модификация Ли с двумя волнами оказывается быстрее алгоритма Дейкстры (приблизительно до 340 ячеек по высоте).

Это связано с отличиями реализации этих алгоритмов. Алгоритм Дейкстры использует очередь с приоритетами, работа с которой требует затрат времени. Очереди – обычный подход к реализации алгоритмов, основанных на поиске в ширину. В этой работе алгоритм Ли реализован без использования очередей. При каждой итерации распространения фронта волны ячейки фронта волны ищутся в прямоугольной области поля лабиринта. Эта область ограничена наиболее удаленными друг от друга ячейками фронта волны. Такой подход оказывается быстрее для относительно небольших лабиринтов (до 640 ячеек для Ли и А*) и значительно более медленным при больших лабиринтах. В прочем, для сравнения нужна реализация алгоритма Ли с очередью. Может оказаться, что очередь без приоритетов, которая нужна для реализации Ли окажется быстрее очереди с приоритетами настолько, что реализация без очередей не будет иметь преимуществ.

5.2.3. Сравнение алгоритмов генерации и поиска

Скорость генерации и поиска сопоставимы друг с другом (рисунок 34). Из рассмотренных алгоритмов генерации быстрее алгоритмов поиска оказался только алгоритм генерации Двоичным деревом.

Так, алгоритм Олдоса-Бродера медленнее алгоритма Ли примерно в 2 раза для лабиринта с высотой 400 ячеек.

6. ВЫВОДЫ

- 1) В ходе поделанной работы были изучены алгоритмы Уилсона, Уилсона(модификация), Олдоса-Бродера, Бинарного дерева для генерации лабиринтов, алгоритмы Ли, Ли с двумя волнами, Дейкстры, А* для поиска пути в лабиринтах.
- 2) Был освоен класс-шаблон адаптера контейнера `priority_queue` в стандартной библиотеке шаблонов (STL) языка C++, способы вывода данных с использованием псевдографики Unicode.
- 3) Были разработаны классы для хранения, доступа и изменения лабиринтов без весов ячеек и с весами ячеек. Реализован вывод лабиринтов в пригодном для редактирования виде, а так же в декоративном виде в двух вариантах: с мелкими и крупными ячейками. Были разработаны классы для сбора времени выполнения алгоритмов генерации и поиска в лабиринтах
- 4) Анализ показал, что алгоритмы поиска на лабиринтах по времени сопоставимы с алгоритмами генерации лабиринтов. Самым медленным является алгоритм Уилсона, самым быстрым – алгоритм Бинарного дерева. Использование эвристики уменьшает время поиска пути в лабиринте приблизительно в 1,2 раз.

7. ИНСТРУКЦИЯ ПОЛЬЗОВАТЕЛЯ

Для начала работы с проектом может быть предложен следующий набор действий:

7.1. Подготовка среды (Debian 11)

- 5) Установить среду разработки : VS Code
- 6) Установить расширения для VS Code:
 - 1) C/C++
 - 2) C/C++ Include Guard

- 3) CMake
- 4) CMake Tools
- 7) Установить gnuplot
- 8) Установить gawk
- 9) Установить ImageMagick

7.2. Установка проекта

7.2.1. Установка с исходным кодом

10) Порядок действий показан на рисунке ниже: необходимо выбрать цель `copyfile` и нажать **Build**. После этого в папку `build` установятся скрипты и текстовые файлы. **После каждого редактирования скриптов и текстовых файлов в папке исходного кода необходимо выполнять запуск цели `copyfile`!**

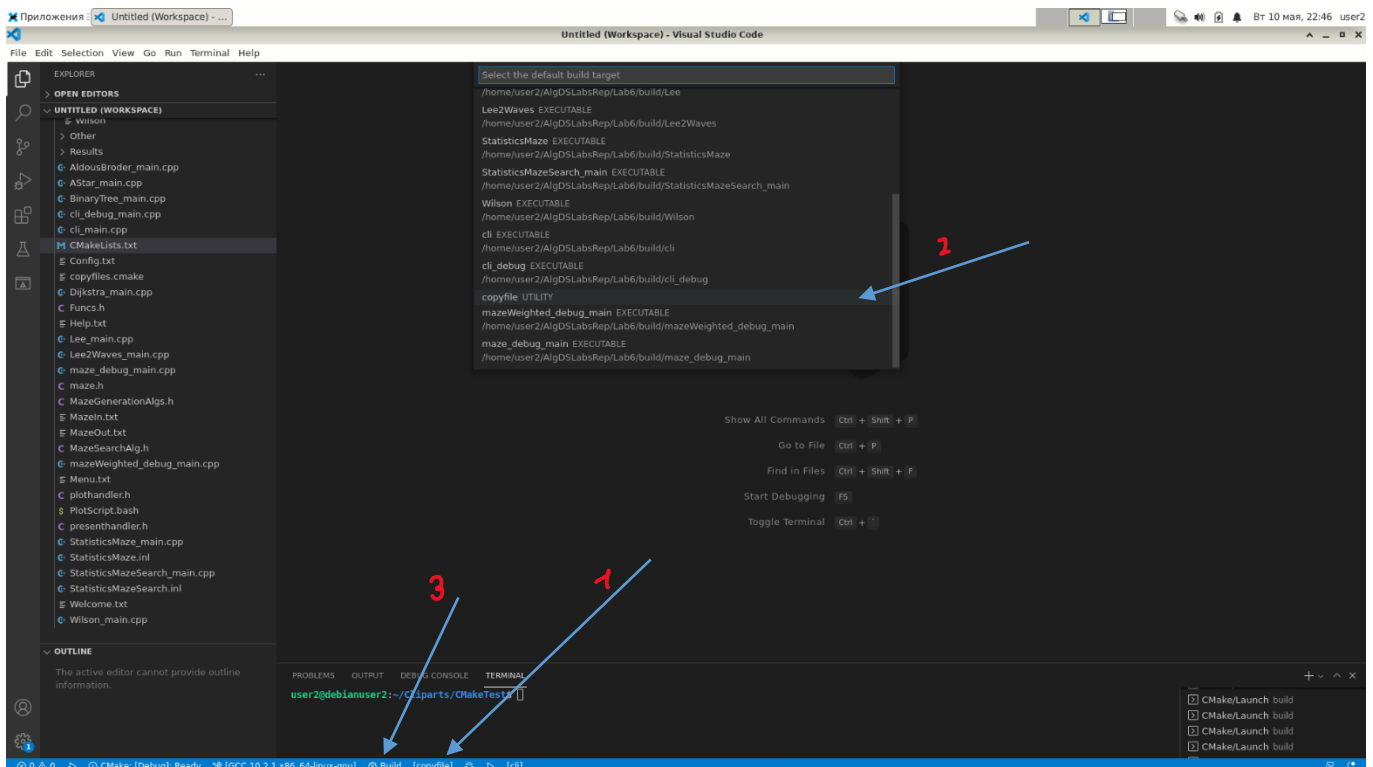


Рисунок 35 – Установка скриптов и текстовых файлов

7.2.2. Сборка и запуск

Для установки, сборки и запуска проекта используется система сборки CMake.

- 11) Выбор цели осуществляется с помощью панели расширения CMake Tools внизу окна:

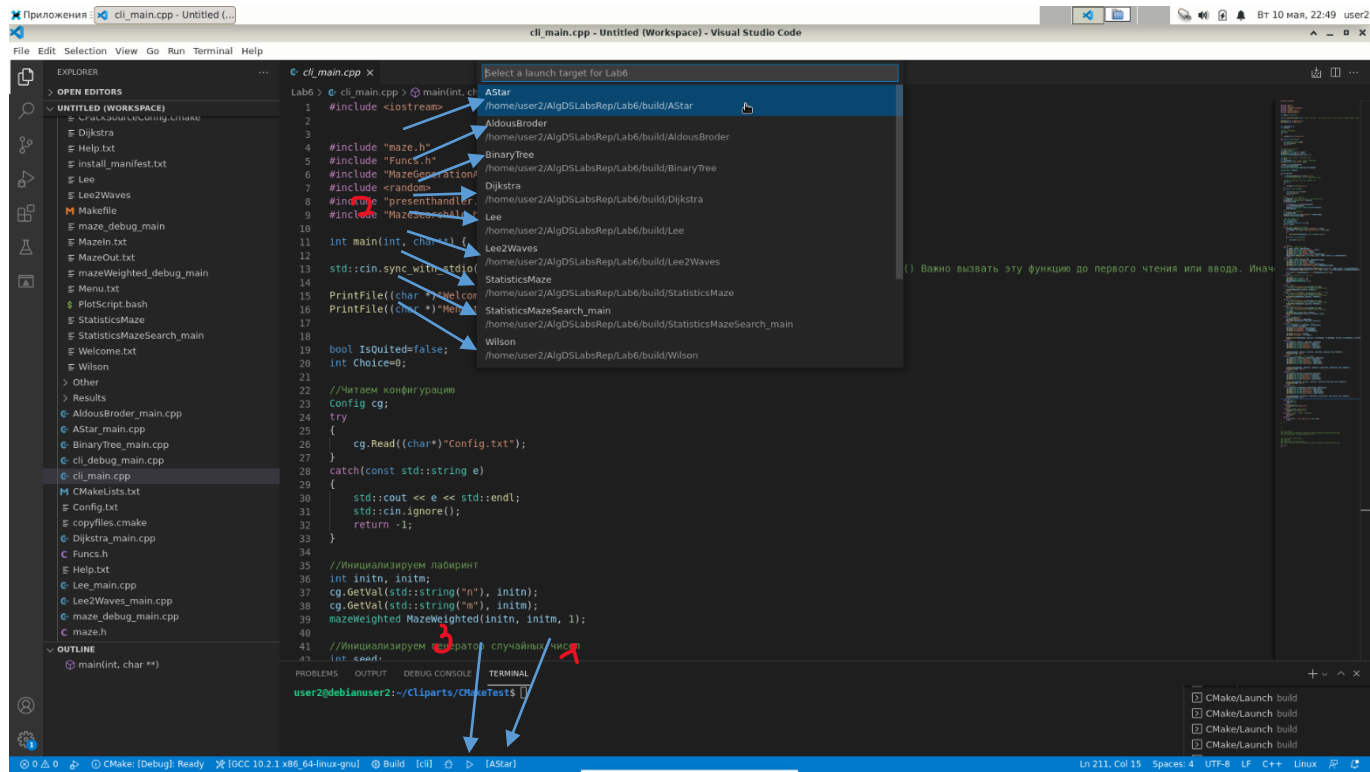


Рисунок 36 – Сборка и запуск

12) Создание и изменение существующих целей осуществляется с помощью редактирования файла CMakeLists.txt

7.2.3. Запуск с CLI

Управление программой осуществляется с помощью файлов конфигурации. Каждый раз при выборе блок-меню пользователем происходит чтение файла конфигурации, параметры которого применяются в программе.

Название	Размер	Тип	Дата изменения ▲
Config.txt	1,9 КиБ	Текстовый документ	сегодня
MazeOut.txt	1,1 КиБ	Текстовый документ	сегодня
Welcome.txt	455 байт	Текстовый документ	сегодня
cli	1,1 МиБ	Разделяемая библиотека	сегодня
Help.txt	953 байта	Текстовый документ	сегодня
Menu.txt	823 байта	Текстовый документ	сегодня
Mazeln.txt	462 байта	Текстовый документ	вчера

Рисунок 37 – Исполняемый файл и файлы, необходимые для его запуска

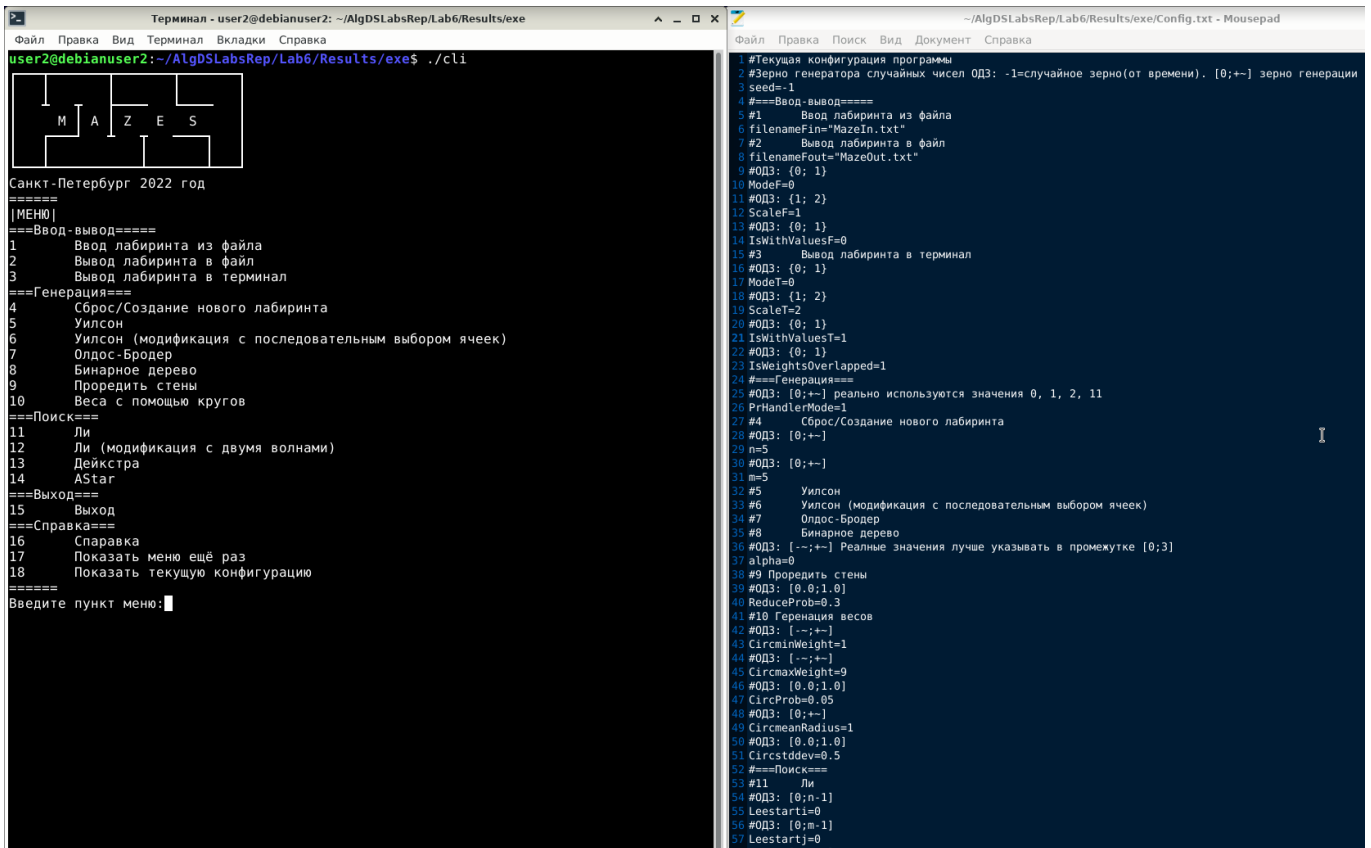


Рисунок 38 – Демонстрация работы программы: запущенная программа и файл конфигурации

7.3. Демонстрация работы

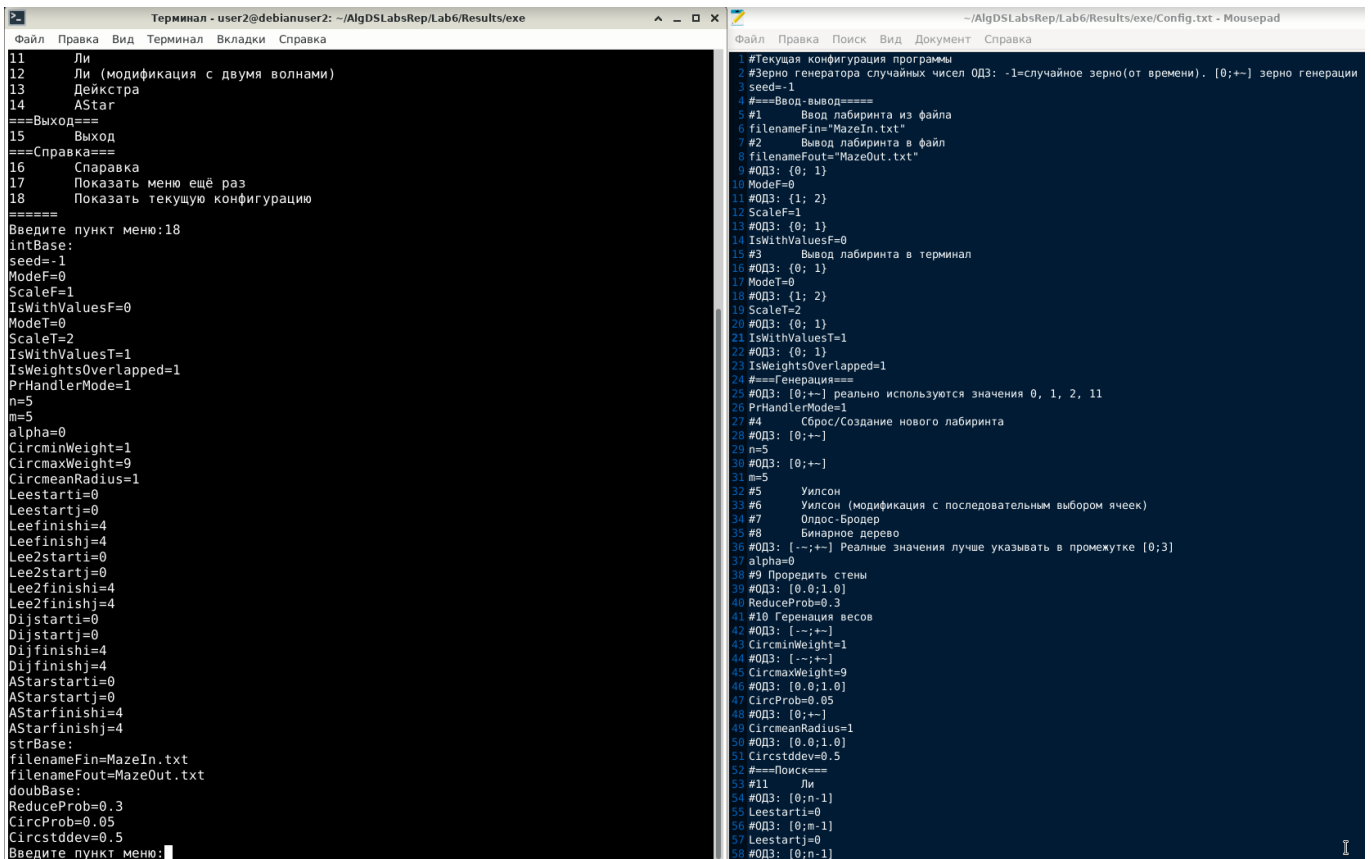


Рисунок 39 – Демонстрация текущей конфигурации

```

>- Терминал - u ~\AlgDSLabsRep\Lab6\Results\exe\Con
Файл Правка Вид Терминал В Файл Правка Поиск Вид Документ Справка
ReduceProb=0.3
CircProb=0.05
Circstddev=0.5
Введите пункт меню:5
CurrentSeed=1652212347
Введите пункт меню:3
#####
#.#.#.?.#.#
+?+?+#+?+?+
#.#.?.?.#
+?+#+?+?+#+
#.#.#.#.?.#
+?+?+#+?+?+
#.#.?.#.#.?.#
+?+#+?+?+?+
#.#.?.#.#.#
#####
-----
  1  1  1  1  1
  1  1  1  1  1
  1  1  1  1  1
  1  1  1  1  1
  1  1  1  1  1
n=5 m=5
Введите пункт меню:9
Введите пункт меню:10
Введите пункт меню:3
#####
#.#.#.?.#
+?+?+#+?+?+
#.#.?.?.#
+?+#+?+?+#+
#.#.?.#.#.?.#
+?+?+?+?+#+
#.#.?.#.#.?.#
+?+#+?+?+?+
#.#.?.#.#.#
#####
-----
  9  5  5  5  5
  5  5  5  5  1
  5  8  5  3  1
  8  8  3  7  5
  8  8  8  5  5
n=5 m=5
Введите пункт меню:
25 #0Д3: [0;+~] реально используются значения 0, 1, 2, 11
26 PrHandlerMode=1
27 #4      Сброс/Создание нового лабиринта
28 #0Д3: [0;+~]
29 n=5
30 #0Д3: [0;+~]
31 m=5
32 #5      Уилсон
33 #6      Уилсон (модификация с последовательным выбором ячеек)
34 #7      Олдос-Бродер
35 #8      Бинарное дерево
36 #0Д3: [--;+~] Реальные значения лучше указывать в промежутке [0;3]
37 alpha=0
38 #9 Проредить стены
39 #0Д3: [0.0;1.0]
40 ReduceProb=0.3
41 #10 Геренация весов
42 #0Д3: [--;+~]
43 CircminWeight=1
44 #0Д3: [--;+~]
45 CircmaxWeight=9
46 #0Д3: [0.0;1.0]
47 CircProb=0.05
48 #0Д3: [0;+~]
49 CircmeanRadius=1
50 #0Д3: [0.0;1.0]
51 Circstddev=0.5
52 #===Поиск===
53 #11      Ли
54 #0Д3: [0;n-1]
55 Leestarti=0
56 #0Д3: [0;m-1]
57 Leestartj=0
58 #0Д3: [0;n-1]
59 Leefinishi=4
60 #0Д3: [0;m-1]
61 Leefinishj=4
62 #12      Ли (модификация с двумя волнами)
63 Lee2starti=0
64 Lee2startj=0
65 Lee2finishi=4
66 Lee2finishj=4
67 #13      Дейкстра
68 Dijstarti=0
69 Dijstartj=0
70 Dijfinishi=4
71 Dijfinishj=4
72 #14      AStar
73 AStarstarti=0
74 AStarstartj=0
75 AStarfinishi=4
76 AStarfinishj=4
77 #===Выход===
78 #15
79 #===Справка===
80 #16      Справка
81 #=====
82
```

Рисунок 40 – Демонстрация работы алгоритма Уилсона

```

n=5 m=5
Введите пункт меню:16
Справка о программе
PrHandlerMode отвечает за режим представления работы алгоритмов
0 не влияет на выполнение
1 выводит промежуточные шаги минимально (например, в алг генерации пишет в файл каждые 30 секунд)
2 выводит промежуточные шаги подробно (выполнение шаг за шагом)

10-20 используются для пошагового вывода, когда исследуется время работы алгоритмов.

В Config жизненно важно:
-целые значения писать без посторонних символов (допустим только минус)
-строковые значения писать в кавычках
-дробные значения должны содержать точку
Введите пункт меню:

```

Рисунок 41 – Справка программы

8. СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

- 1) Введение в алгоритм A*. URL: <https://habr.com/ru/post/331192/> (дата обращения 2022 г.)
- 2) Неприлично простая реализация неприлично простого алгоритма генерации лабиринта. URL: <https://habr.com/ru/post/319532/> (дата обращения 2022 г.)
- 3) Лабиринты: классификация, генерирование, поиск решений. URL: <https://habr.com/ru/post/445378/> (дата обращения 2022 г.)
- 4) Овчинников В.А., Васильев А.Н., Лебедев В.В // Проектирование печатных плат: учебное пособие - 1-е изд. Тверь: ТГТУ, 2005. 116 с