

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Сибирский государственный университет науки и технологий
имени академика М.Ф. Решетнева»

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

КУРСОВАЯ РАБОТА

Программная реализация поиска пути в лабиринте

Руководитель

29.12.23

подпись, дата

В.В. Тынченко

инициалы, фамилия

Обучающийся

БПИ22-02, 221219040

номер группы, зачетной книжки

29.12.23

подпись, дата

К.В. Трифонов

инициалы, фамилия

Красноярск 2023

Институт информатики и телекоммуникаций
Кафедра информатики и вычислительной техники

ЗАДАНИЕ

на курсовую работу по дисциплине «Алгоритмы и структуры данных»
студенту Трифонову Кириллу Вячеславовичу

Группа БПИ22-02

Форма обучения очная

1. Тема работы: Программная реализация поиска пути в лабиринте

2. Срок сдачи студентом работы: 30.12.23

3. Перечень вопросов, подлежащих разработке при написании теоретической части:

Алгоритм Дейкстры

Алгоритм A*

Алгоритм BFS

Алгоритм DFS

Алгоритм JPS

4. Перечень вопросов, подлежащих разработке при написании практической части (либо указать номер варианта задания)

Разработка алгоритма BFS поиска

Разработка алгоритма DFS поиска

Разработка алгоритма A* поиска

5. Дата выдачи задания: 08.09.23

Руководитель: Тынченко В.В.



Подпись

Задание принял к исполнению (дата): 08.09.23



(подпись студента)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	4
1 Анализ предметной области	5
1.1 Алгоритм Дейкстры.....	5
1.2 Алгоритм A*	5
1.3 Поиск в ширину (BFS).....	6
1.4 Поиск в глубину (DFS)	8
1.5 Jump Point Search (JPS).....	8
1.6 Вывод	10
2 Описание программы.....	11
2.1 Общая характеристика программы	11
2.2 Структура программы и данных	11
2.3 Интерфейс программы	12
2.4 Тестирование программы.....	15
2.5 Вывод	18
3 Использование программы	19
3.1 Постановка задачи	19
3.2 Реализация программы.....	19
3.3 Сравнительное тестирование.....	20
3.4 Вывод	23
ЗАКЛЮЧЕНИЕ	24
Список использованных источников	25
Приложение А Листинг программы	26
Приложение Б Проверка на оригинальность	37

ВВЕДЕНИЕ

В современном мире область исследования алгоритмов поиска пути приобретает особую актуальность, находя применение в широком спектре задач, от робототехники до компьютерных игр. Программная реализация эффективных и точных методов поиска пути в лабиринте имеет важное значение в контексте разработки автономных систем, планирования маршрутов и создания интерактивных игровых сценариев.

Данная курсовая работа посвящена исследованию и программной реализации различных алгоритмов поиска пути в лабиринте. В ходе работы рассмотрены и проанализированы такие важные методы, как алгоритм Дейкстры, A*, поиск в ширину (BFS), поиск в глубину (DFS) и Jump Point Search (JPS). Каждый из этих алгоритмов рассматривается с точки зрения своей эффективности, точности и применимости в различных сценариях.

Цель работы заключается в разработке программных реализаций выбранных алгоритмов и в последующем тестировании их производительности в условиях лабиринта. Такой подход позволяет выявить особенности работы каждого алгоритма, а также определить, какие из них наиболее эффективны в различных контекстах использования.

Задачи:

- изучить теоретические основы алгоритмов создания лабиринтов;
- изучить теоретические основы алгоритмов поиска путей;
- разработать программный код алгоритмов;
- провести тестирование программы.

1 Анализ предметной области

Программная реализация поиска пути в лабиринте — это интересная и важная задача, которая используется во многих областях, таких как навигация, планирование транспортных маршрутов, сортировка, робототехника и искусственный интеллект. Существует множество алгоритмов, которые можно использовать для решения этой задачи. Наиболее популярные из них приведены ниже.

1.1 Алгоритм Дейкстры

Алгоритм назван в честь голландского ученого Эдсгера Дейкстры, который разработал его в 1956 году. За основу алгоритма берется принцип жадного выбора: на каждом шаге выбирается вершина с наименьшим известным расстоянием от источника и проверяются все её соседние вершины. Если расстояние до соседней вершины через текущую вершину оказывается короче, то расстояние обновляется.

Однако этот алгоритм подходит только для графов без отрицательных ребер, так как при наличии отрицательных циклов может возникнуть бесконечный цикл обновления расстояний.

Алгоритм описывается следующим образом:

- в начале алгоритма расстояние для начальной вершины полагается равным нулю, а все остальные расстояния заполняются большим положительным числом (бóльшим максимального возможного пути в графе);
- массив флагов заполняется нулями. Затем запускается основной цикл;
- на каждом шаге цикла мы ищем вершину v с минимальным расстоянием и флагом равным нулю. Затем мы устанавливаем в ней флаг в 1 и проверяем все соседние с ней вершины u . Если в них (в u) расстояние больше, чем сумма расстояния до текущей вершины и длины ребра, то уменьшаем его;
- цикл завершается, когда флаги всех вершин становятся равны 1, либо когда у всех вершин с флагом 0 Последний случай возможен тогда и только тогда, когда граф G несвязный.

Работа алгоритма показана на блок-схеме (рисунок 1).

1.2 Алгоритм A^*

Является модификацией алгоритма Дейкстры, который использует эвристику для ускорения процесса поиска. Он работает путем оценки стоимости каждого возможного шага и выбора того, который, как предполагается, приведет к наиболее эффективному пути [2].

Основа алгоритма - это эвристическая функция “расстояние + стоимость” (обычно обозначается как $f(x)$), которая является суммой двух компонентов: стоимости достижения текущей вершины (x) из начальной (обычно обозначается как $g(x)$ и может быть как эвристической, так и нет), и эвристической оценки расстояния от текущей вершины до конечной (обозначается как $h(x)$). Эта функция определяет порядок обхода вершин [3].



Рисунок 1 – Блок-схема алгоритма Дейкстры

Алгоритм описывается следующим образом:

- начинаем с начальной вершины и устанавливаем ее оценочную стоимость равной эвристической оценке расстояния до цели;
- пока есть не посещенные вершины выбираем вершину с наименьшей оценочной стоимостью и пометьте ее как посещенную. Если это целевая вершина, то путь найден;
- в противном случае для каждой соседней вершины обновляем ее оценочную стоимость, если текущая оценка больше суммы стоимости пути до выбранной вершины, веса ребра между ними и эвристической оценки до цели.

Работа алгоритма показана на блок-схеме (рисунок 2).

1.3 Поиск в ширину (BFS)

Начинается с корневого узла и исследует все соседние узлы на данном уровне перед переходом к узлам следующего уровня. Ключевая идея заключается в том, что мы отслеживаем состояние расширяющегося кольца, которое называется границей. В сетке этот процесс иногда называется заливкой (flood fill), но та же техника применима и для карт без сеток [4].

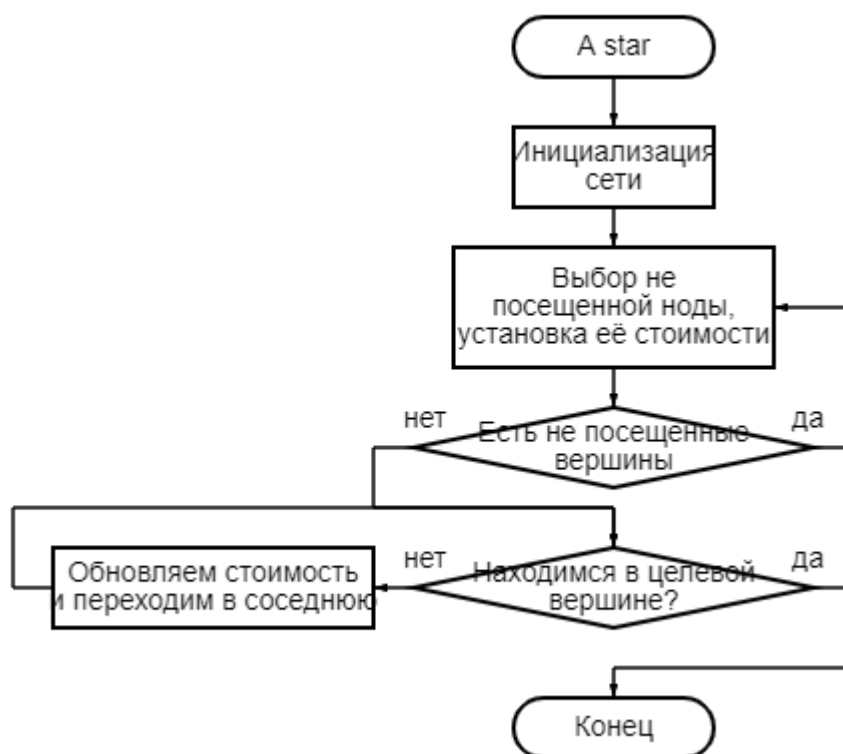


Рисунок 2 – Блок-схема алгоритма A*

Алгоритм описывается следующим образом:

- берётся первый узел из очереди и помечается как посещенный;
- если он целевой, то завершаем работу программы, иначе проверяем преемников этого узла;
- если все узлы просмотрены, а целевой найден не был, то он недостижим из начального.

работа алгоритма показана на блок-схеме (рисунок 3).



Рисунок 3 – Блок-схема BFS алгоритма

1.4 Поиск в глубину (DFS)

Основан на схожем принципе, что и поиск в ширину, однако распространяется в самую глубь графа, пока не достигнет конца, а затем возвращается назад, чтобы исследовать другие ветви. Возврат обратно к высшей вершине происходит только когда в выбранной вершине были рассмотрены все рёбра, в которых не осталось нерассмотренных вершин [5].

Алгоритм описывается следующим образом:

- начинаем с корневого узла и добавляем его в стек;
- пока стек не пуст извлекаем узел из стека и помечаем его как посещенный;
- добавляем все не посещенные соседние узлы в стек;
- если узел целевой или узлов не осталось (решение невозможно), то цикл заканчивается с соответствующим результатом.

Работа алгоритма показана на блок-схеме (рисунок 4).

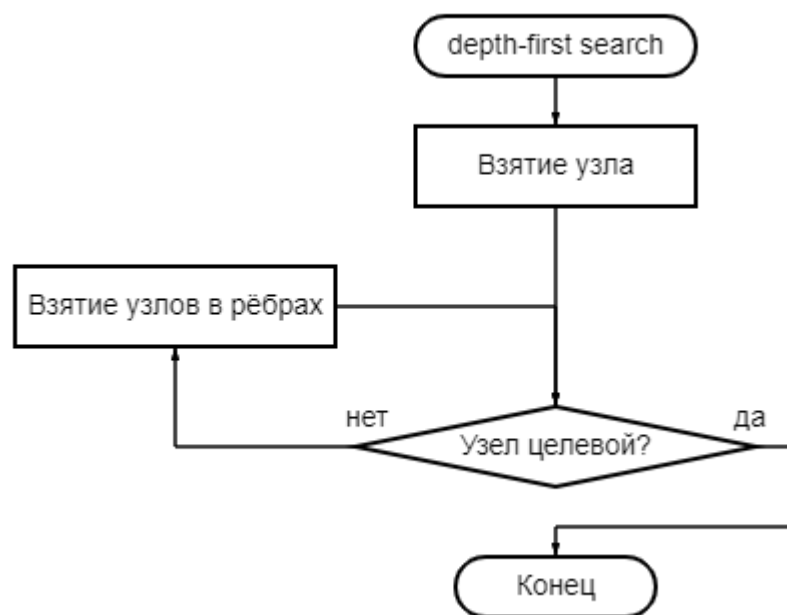


Рисунок 4 – Блок-схема DFS алгоритма

1.5 Jump Point Search (JPS)

Является оптимизацией алгоритма поиска A^* для сеток с равномерной стоимостью. Он уменьшает симметрию в процедуре поиска путем обрезки графа, устраняя определенные узлы в сетке на основе предположений, которые можно сделать о соседях текущего узла, при условии, что выполняются определенные условия, относящиеся к сетке. Преимуществом JPS является то, что он не требует предобработки и поэтому тратит меньше памяти, также он является самым современным из перечисленных – он получил широкую огласку в 2012 году, а последние его крупные улучшения были опубликованы в 2014.

Алгоритм описывается следующим образом:

- берём корневой узел;
- если узел не целевой, то выбирается точка (узел графа), в которую совершится прыжок по прямой (рисунок 5) или по диагонали (рисунок 6);

– если узел целевой или узлов не осталось (решение невозможно), то цикл заканчивается с соответствующим результатом.

Работа алгоритма показана на блок-схеме (рисунок 7).

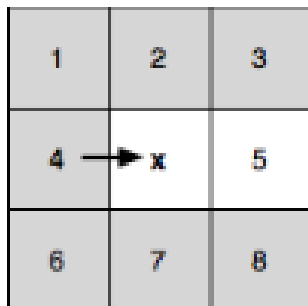


Рисунок 5 – Прямолинейный прыжок JPS алгоритма

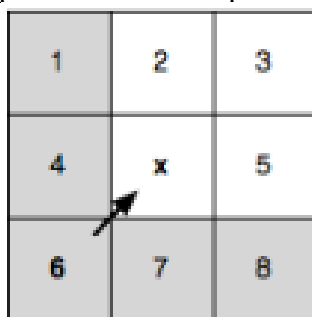


Рисунок 6 – Диагональный прыжок JPS алгоритма

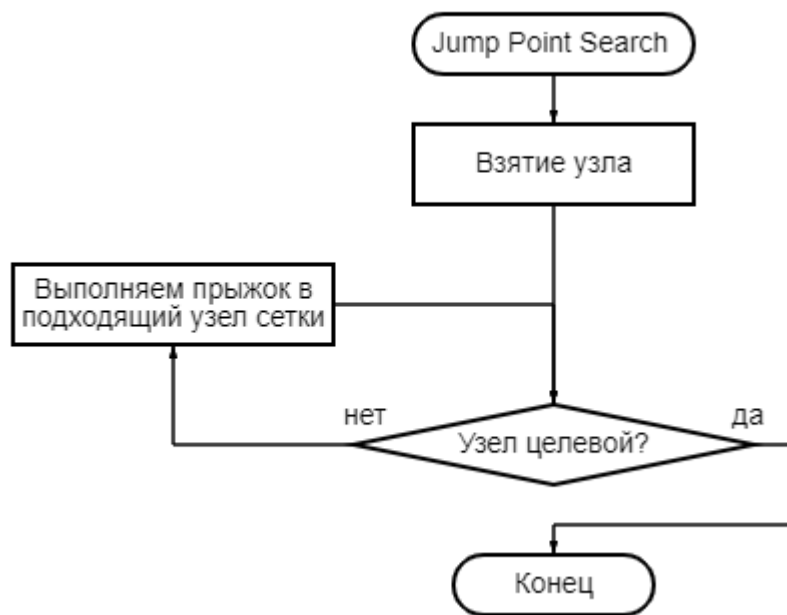


Рисунок 7 – Блок-схема JPS алгоритма

Пусть u является целевой (посадочной) точкой точки x , в направлении d , если u минимизирует значение k по закону: $y = x + kd$, и будет выполнено любое из условий:

Точка u – точка посадки.

У точки u есть хотя бы один сосед, у которого более 1 соседа.

d – движение по диагонали и существует точка $z = y + ki d_i$, которая лежит в k_i шагах в направлении $d_i \in \{d_1, d_2\}$, таких что z – точка прыжка из y при условии 1 или 2.

Рассмотрим пример одного из прыжков (рисунок 8). Пусть мы начинаем в точке x и заканчиваем движение по диагонали, пока не наткнёмся на точку y . Из y в точку z можно попасть с k_i по горизонтали. Таким образом, z является приемником точки для прыжка x , а это в свою очередь определяет y как приемник для прыжка точки x .

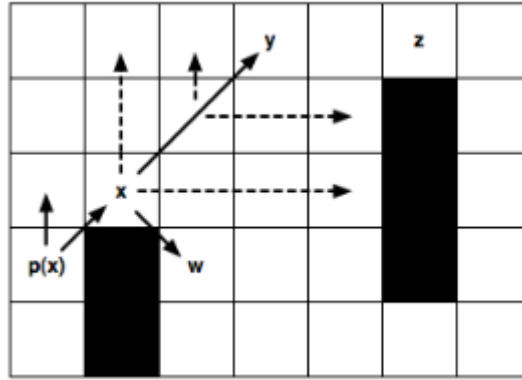


Рисунок 8 – Пример прыжка JPS алгоритма

1.6 Вывод

Представляя лабиринт в виде неориентированного графа, можно использовать множество разнообразных алгоритмов, которые могут быть эффективны в конкретном спектре задач. Большинство из них используют схожий принцип последовательного исследования графа с проверкой текущей вершины (точки лабиринта) на то, является ли она целевой (выходом из лабиринта). Однако они кардинально различаются в подходе к порядку выбора рассматриваемых вершин.

2 Описание программы

2.1 Общая характеристика программы

Название программы - «Изучение быстродействия методов поиска путей в лабиринте».

Программа, написанная в ходе выполнения курсовой работы направлена на изучение быстродействия и особенностей отдельных алгоритмов поиска путей в лабиринтах. Программа способна строить лабиринты тремя методами, находить в них путь и выводить лабиринты на экран с применением форматирования (отрисовкой стен).

Для написания программы был выбран C++, поскольку он имеет более высокое быстродействие по сравнению с python.

Минимальные системные требования:

- процессор: 2 ядра и частотой от 2 ГГц;
- оперативная память: 4 Гб;
- свободное дисковое пространство: 5Мб;
- операционная система: Windows 10 со средой разработки Visual Studio

2.2 Структура программы и данных

В разработке программы применялся метод ООП и был разработан класс «Лабиринт» с полями матрицы лабиринта и посещённых клеток и класс «Точка» с полями координат для заполнения списков очереди в алгоритмах поиска. В каждом из классов были реализованы конструкторы (по умолчанию, с параметрами, копий) и деструкторы.

В классе «Лабиринт» также были созданы методы отрисовки лабиринта на матрице с помощью трёх методов (Алгоритмом северо-восточного смещения, комбинированным алгоритмом северо-восточного и северо-западного смещения и алгоритмом «Sidewinder») и три метода поиска путей (BFS, DFS и A star).

Блок схема представлена на рисунке (Рисунок 9).

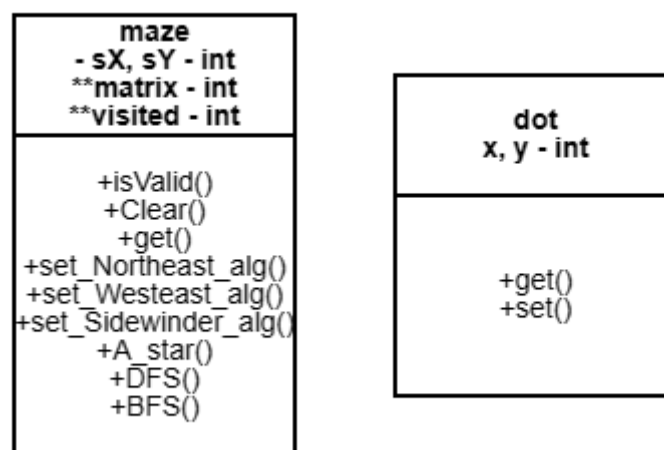


Рисунок 9 – Блок схема классов программы

2.3 Интерфейс программы

Использование консоли в качестве взаимодействия с пользователем накладывает некоторые ограничения в отрисовке изображений на экране, однако лабиринты всё равно можно вывести с помощью символов. Для отрисовки стен применять, например, крупные объемные буквы, а для пустых мест ставить пробелы или точки.

Для вывода стен-границ использован символ “Н”, для вывода внутренних стен символ “Х”, а для проходов - пробелы. Для обозначения стартовой позиции используется символ “S”, а для финиша используется символ “Е”. По умолчанию их расположение в левом верхнем и правом нижнем углах соответственно. Последняя цифра пройденного расстояния пишется в тех клетках, в которых алгоритм побывал, итоговый путь обозначается символом “\$”.

Пример вывода лабиринта, заполненного алгоритмом северо-восточного смещения размером 31 x 31 на рисунке (рисунок 10).

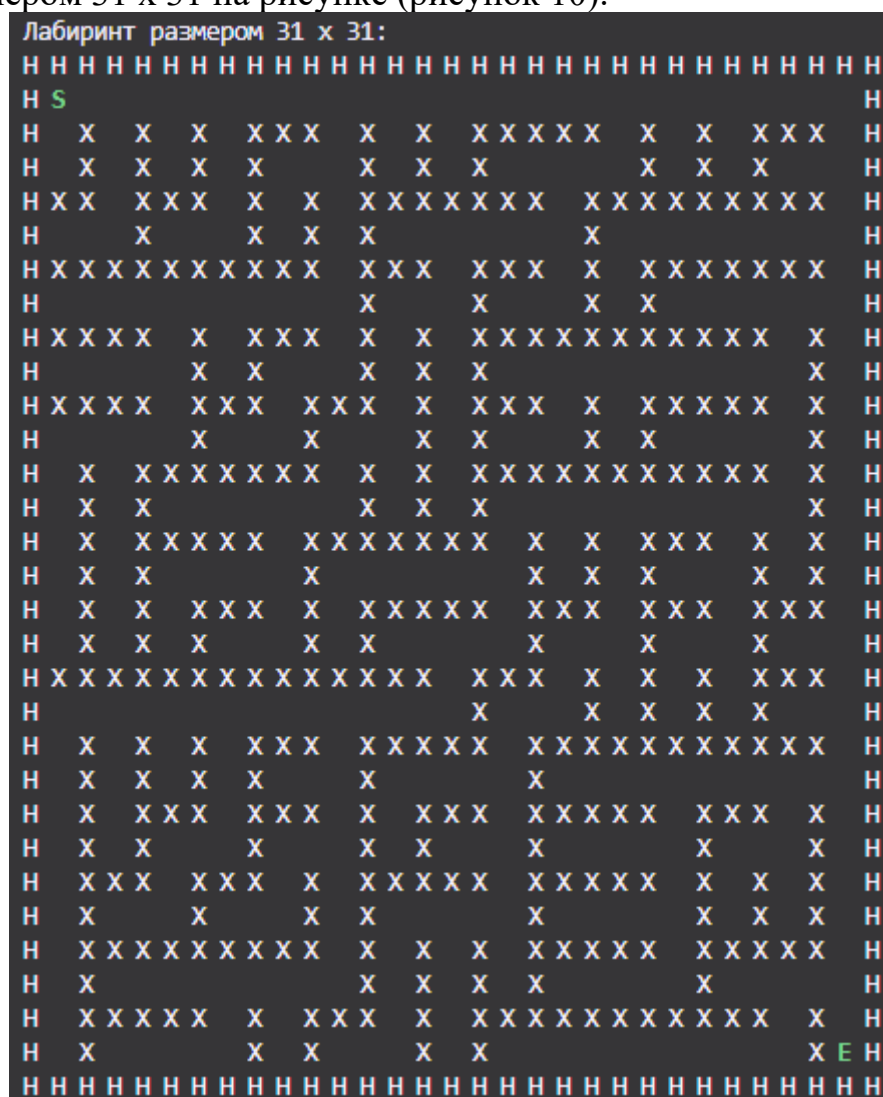


Рисунок 10 – Работа комбинированного алгоритма северо-восточного смещения

Пример вывода лабиринта, заполненного комбинированным алгоритмом северо-восточного смещения размером 31 x 31 на рисунке (рисунок 11).

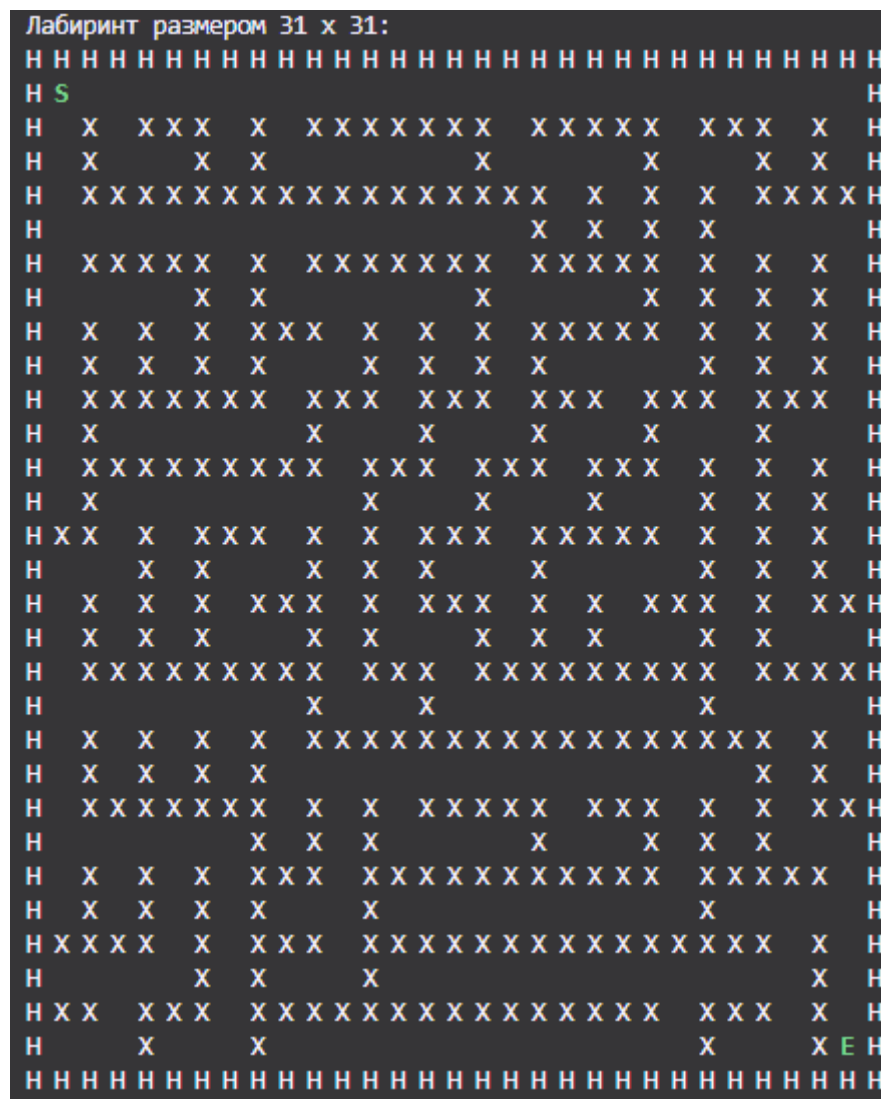


Рисунок 11 – Работа комбинированного алгоритма северо-восточного смещения

Пример вывода лабиринта, заполненного алгоритмом Sidewinder размером 31 x 31 на рисунке (рисунок 12).

Алгоритм северо-восточного смещения получилось реализовать стабильным, он всегда генерирует решаемый лабиринт с одним решением, поэтому дальнейшее тестирование будет проводиться на лабиринтах, сгенерированных этим алгоритмом.

Описание меню программы:

Визуально, меню отображается в виде пронумерованных пунктов с описанием каждой опции. Это обеспечивает понятность и удобство взаимодействия с пользователем. Каждый пункт меню отображается с новой строки, что улучшает читаемость и структурированность вывода, делая его более интуитивно понятным для пользователя.

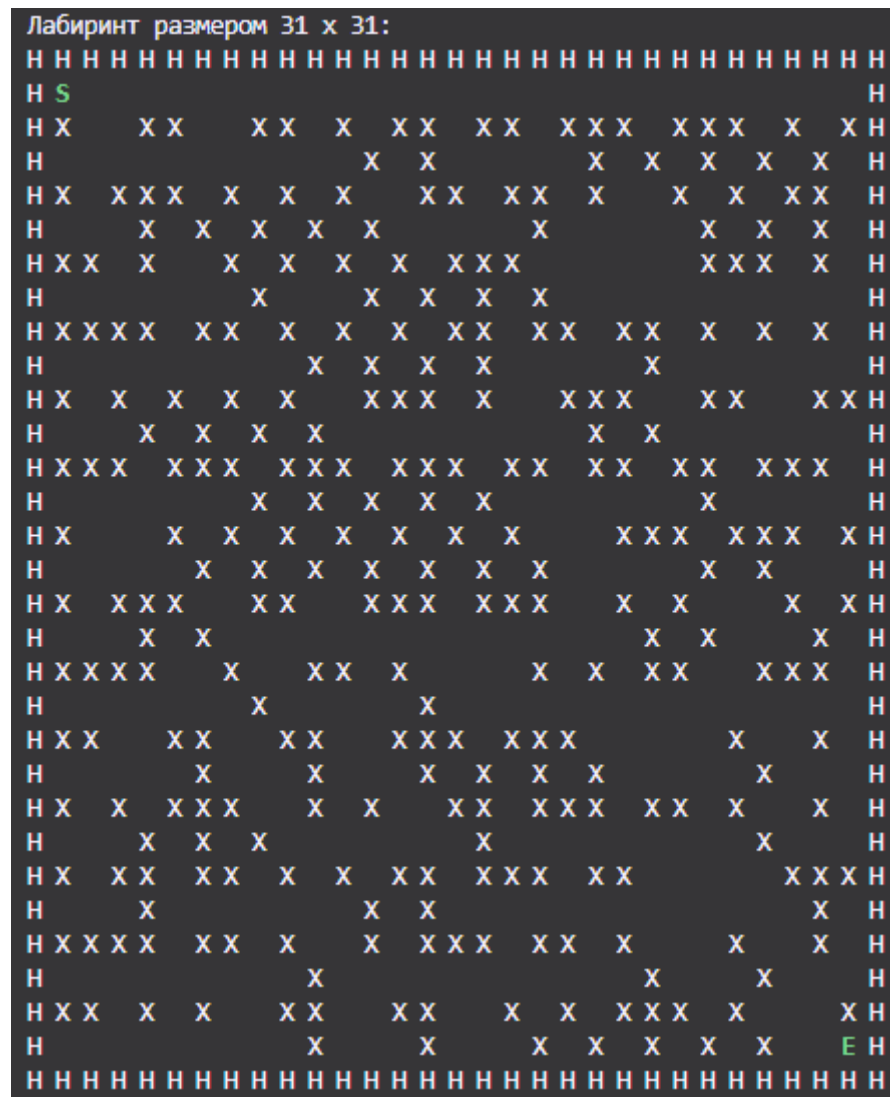


Рисунок 12 – Пример работы алгоритма Sidewinder

Меню представлено на рисунке (рисунок 13).

```

Главное меню
1) Работа с одним лабиринтом
2) Накопление статистических данных
0) <<<< Выход
  
```

Рисунок 13 – Вид главного меню

Выбор режима: позволяет выбрать режим работы (работа с одним лабиринтом, накопление статистических данных).

Режим работы с одним лабиринтом: пользователь вводит с консоли размеры исследуемого лабиринта, далее открывается меню функций по работе с одним лабиринтом: построение (по выбору), решение (с выбором метода) и вывод на экран.

Режим работы с накоплением статистических данных: пользователь вводит диапазон тестирования и шаг, далее выбирается метод заполнения и тестируемый метод решения лабиринта. По окончании решения каждого из N лабиринтов результаты выводятся на экран и файл results.txt.

UML – диаграмма прецедентов (рисунок 14).

UML – диаграмма интерфейса представлена на рисунке (рисунок 16).

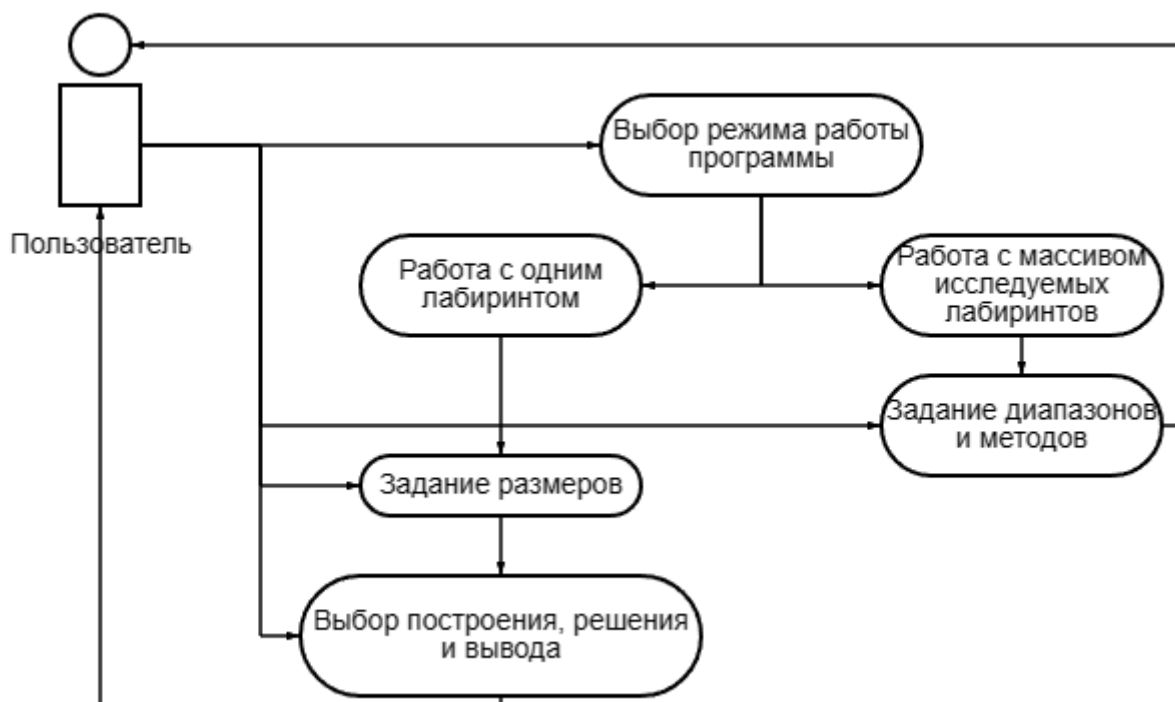


Рисунок 14 – UML диаграмма прецедентов

2.4 Тестирование программы

Для проверки работы алгоритмов решения лабиринтов создавался алгоритмом северо-восточного смещения лабиринт размером 15 x 15. Работа алгоритма начиналась с клетки “S” (1,1) левого верхнего угла и заканчивалась, когда финиш “E” становился текущей клеткой.

Для тестирования в первом режиме работы программы «Работа с одним лабиринтом» был сгенерирован лабиринт 15 x 15.

Тестируемый лабиринт изображен на рисунке (рисунок 15).

Лабиринт размером 15 x 15:

```

Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н
Н S Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н
Н Х Х Х Х Х Х Х Х Х Н Н Н Н Н Н
Н Х Н Н Н Н Н Н Н Н Н Н Н Н Н Н
Н Х Х Х Х Х Х Х Х Х Х Н Н Н Н Н
Н Н Х Н Н Н Н Н Н Н Н Н Н Н Н Н
Н Х Х Х Х Х Х Х Х Х Х Н Н Н Н Н
Н Х Х Н Н Н Н Н Н Н Н Н Н Н Н Н
Н Х Х Х Х Х Х Х Х Х Х Н Н Н Н Н
Н Н Х Н Н Н Н Н Н Н Н Н Н Н Н Н
Н Х Х Х Х Х Х Х Х Х Х Н Н Н Н Н
Н Х Х Н Н Н Н Н Н Н Н Н Н Н Н Н
Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н
  
```

Рисунок 15 – Тестируемый лабиринт

DFS алгоритм (поиск в глубину) по итогам тестирования оказался самым эффективным, т.к. финиш находится в самой отдалённой части лабиринта. Было посещено наименьшее количество точек.

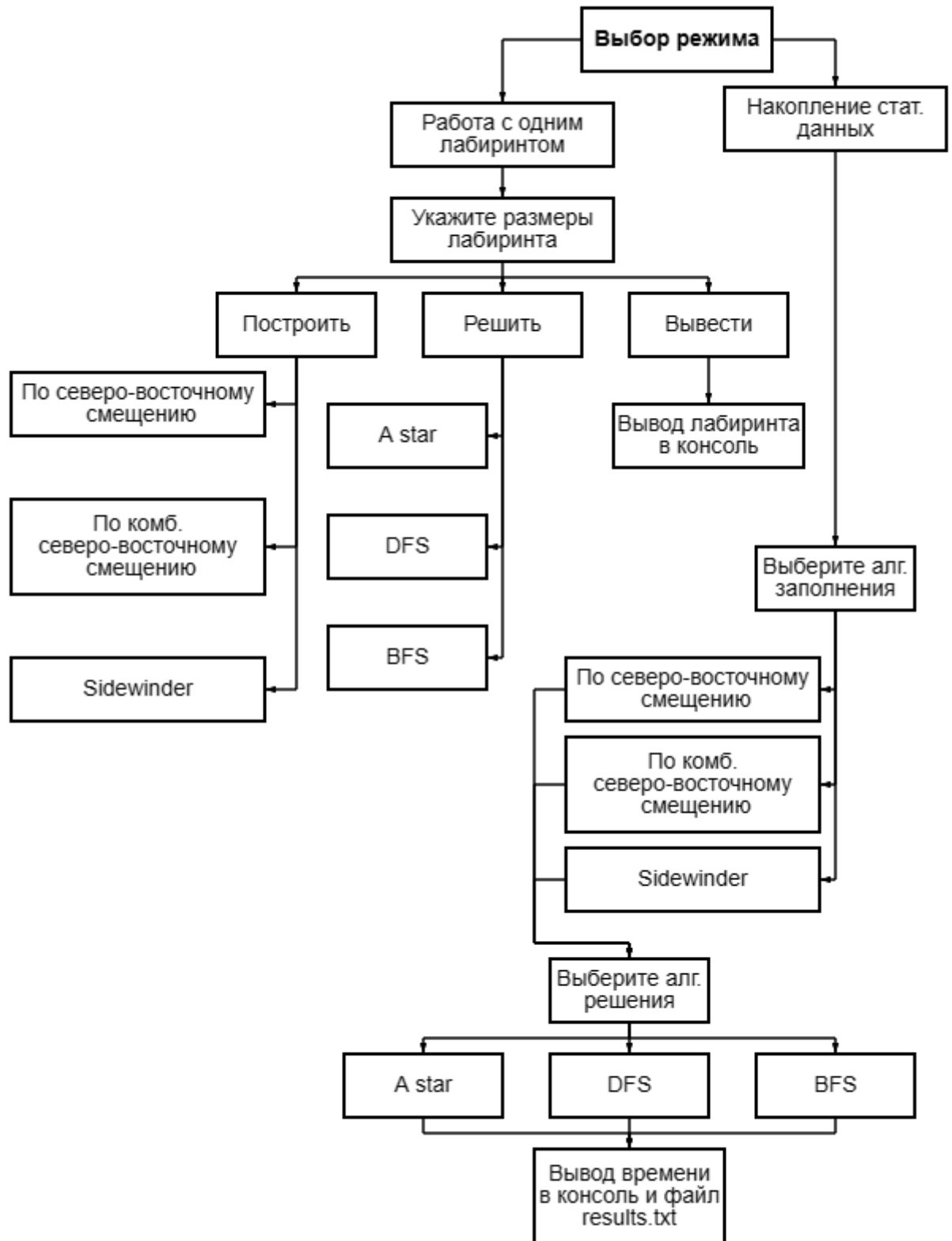


Рисунок 16 – UML диаграмма интерфейса программы

Результат работы DFS алгоритма представлен на рисунке (рисунок 17).

```

Посещено: 53
Завершено за 0ms
Лабиринт размером 15 x 15:
Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н
Н S $ $ 3 4 5 6 7 8 9 0 1 2 Н
Н Х $ Х Х Х Х Х Х Х Х Х 3 Н
Н Х $ $ $ $ $ 9 0 1 2 Х 4 Н
Н Х Х Х Х Х $ Х Х Х Х Х Н
Н          Х $ $ $ $ $ $ $ Н
Н Х Х Х Х Х Х Х Х Х $ Н
Н Х Х          Х $ Н
Н Х Х Х Х Х Х Х Х Х $ Н
Н          Х Х Х Х Х $ Н
Н Х Х Х Х Х Х Х Х Х $ Н
Н Х Х Х Х Х Х Х Х Х $ Н
Н Х Х Х Х Х Х Х Х Х $ Н
Н Х Х Х Х Х Х Х Х Х $ Н
Н Х Х Х Х Х Х Х Х Х $ Н
Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н

```

Рисунок 17 – Результат работы DFS алгоритма

BFS алгоритм (поиск в ширину) по итогам тестирования показал себя хуже всех, поскольку целевая точка находится в самом дальнем углу. Этот метод может показать себя эффективнее, чем DSF, если известно, что целевая точка находится не “на глубине”. Было посещено наибольшее количество точек.

Результат работы BFS алгоритма представлен на рисунке (рисунок 18).

```

Посещено: 96
Завершено за 1ms
Лабиринт размером 15 x 15:
Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н
Н S $ $ 3 4 5 6 7 8 9 0 1 2 Н
Н 1 Х $ Х Х Х Х Х Х Х Х Х 3 Н
Н 2 Х $ $ $ $ $ 9 0 1 2 Х 4 Н
Н 3 Х Х Х Х Х $ Х Х Х Х Х Н
Н 4 5 6 7 8 Х $ $ $ $ $ $ $ Н
Н 5 Х 7 Х 9 Х Х Х Х Х Х Х $ Н
Н 6 Х 8 Х 0 1 2 3 4 5 6 Х $ Н
Н Х Х Х Х 1 Х Х Х 5 Х Х Х $ Н
Н 6 5 4 3 2 Х 8 7 6 Х 2 1 $ Н
Н Х Х 5 Х Х Х Х Х 7 Х Х Х $ Н
Н 8 7 6 Х 2 1 0 9 8 Х 4 3 $ Н
Н 9 Х 7 Х Х Х Х Х Х Х 5 Х $ Н
Н 0 Х 8 Х 2 1 0 9 8 7 6 Х Е Н
Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н

```

Рисунок 18 – Результат работы BFS алгоритма

Алгоритм A star показывает средний результат, не заходит в заведомо более далёкие от цели пути. Его логика основывается на эвристике: у него присутствует, помимо анализа стоимости длины $g(n)$, ещё и функция $h(n)$, которая вычисляет удалённость точки до цели. Благодаря этому, на развилках, A star выберет дорогу согласно её направлению к цели. Если расположение точки будет

хаотично и не обязательно в глубине алгоритма, то A star покажет наилучшие результаты.

Результат работы A star алгоритма представлен на рисунке (рисунок 19).

```

Посещено: 66
Завершено за 0ms
Лабиринт размером 15 x 15:
Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н
Н 5 $ $ 3 4 5 6 7 8 9 0 1 2 Н
Н 1 X $ X X X X X X X X 3 Н
Н 2 X $ $ $ $ $ 9 0 1 2 X 4 Н
Н 3 X X X X X $ X X X X X Н
Н 4 5 6 7 8 X $ $ $ $ $ $ Н
Н 5 X 7 X 9 X X X X X X $ Н
Н 6 X 8 X 0 1 2 3 4 5 6 X $ Н
Н X X X X 1 X X X 5 X X X $ Н
Н      2 X      6 X      $ Н
Н X X   X X X X X 7 X X X $ Н
Н      X      8 X      $ Н
Н   X   X X X X X X X   X $ Н
Н   X   X      X      X E Н
Н Н Н Н Н Н Н Н Н Н Н Н Н Н Н

```

Рисунок 19 – Результат работы A star алгоритма

2.5 Вывод

В данном разделе проведен обзор программы, ее структуры, интерфейса и результатов тестирования. Выявлено, что использование алгоритма A* в программе обеспечивает более эффективный поиск пути в сравнении с методами BFS и DFS, что подтверждает не только теоретические предположения, но и результаты практических испытаний.

3 Использование программы

3.1 Постановка задачи

Программа должна создавать различные лабиринты, находить в них путь от начальной точки до конечной различными алгоритмами с подсчетом времени и посещенных точек для сравнения различных методов поиска пути.

3.2 Реализация программы

Для создания лабиринта были реализованы 3 алгоритма:

- алгоритм северо-восточного смещения;
- комбинированный алгоритм северо-восточного смещения;
- алгоритм Sidewinder.

Алгоритм северо-восточного смещения создаёт простой лабиринт, который имеет два пустых коридора вверху и справа в лабиринте, а также сильное диагональное смещение в направлении от левого нижнего до правого верхнего угла.

Комбинированный алгоритм северо-восточного смещения представляет собой смесь из северо-восточного и юго-западного смещения. Первые 8 клеток лабиринта обрабатываются одним, следующие 8 вторым и т.д. Не имеет пустого коридора справа, а также явное смещение в каком-либо направлении.

Алгоритм Sidewinder создаёт пустые горизонтальные коридоры случайной длины (от 1 до 8) и соединяет их пробелами в случайном месте. Имеет только верхний коридор. Реализовать стабильно его не получилось т.к. иногда возможны ситуации, когда он создаёт замкнутые области, в том числе и вокруг финиша [1].

Для поиска пути в созданном лабиринте были реализованы 3 алгоритма:

- BFS;
- DFS;
- A star.

Алгоритм BFS создаёт две дополнительные матрицы – булеву **visited** хранящую посещённые точки и целых чисел **distance** хранящую расстояние от старта до этой клетки матрицы (обе имеют ту же размерность, что и чар матрица поля класса `maze`). Также имеется очередь, в которую в конец записываются новые найденные допустимые точки. Сначала проверяются точки из начала очереди, т.е. старые. Таким образом алгоритм сначала просмотрит точки, которые были обнаружены раньше и пойдёт вширь, а не в глубину новых точек.

Алгоритм DFS использует такие же матрицы, но хранит точки не в очереди, а в стеке, следовательно сначала проверяются точки из конца, т.е. новые. Происходит проход в глубину до тех пор, пока новых точек не появится (тупик).

Алгоритм A star хранит точки в приоритетной очереди, значение приоритета определяется по сумме пути и эвристической функции Манхэттенского расстояния.

По окончании (когда точки в очереди закончились или был найден финиш) каждый алгоритм начинает собирать кратчайший путь до финиша, пользуясь матрицей **distance**, сравнивая пути на перекрёстках. Параллельно этому в чар матрицу класса `maze` записываются путь и последняя цифра расстояния от старта до этой точки для зрительной оценки пути.

3.3 Сравнительное тестирование

Сравнительное тестирование проводилось во втором режиме работы программы «Накопление статистических данных». В нём создавались лабиринты с размером стороны от 500 до 4500 с шагом 500 (9 лабиринтов). Затем лабиринты решаются выбранным алгоритмом с подсчетом времени решения каждого, результаты записываются в файл `results.txt` в формате размер-посетенные-время (рисунок 20).

```
course > results.txt
1 501 124998 63
2 1001 499998 252
3 1501 1124999 561
4 2001 1999999 1005
5 2501 3124998 1553
6 3001 4499998 2231
7 3501 6124998 3059
8 4001 7999999 4039
9 4501 10124998 5283
```

Рисунок 20 – Пример содержимого файла `results.txt`

По полученным данным тестирования алгоритмов в лабиринте комбинированного северо-восточного смещения был построен график зависимости времени решения от размера (рисунок 21) и график зависимости посещенных точек от размера лабиринта (рисунок 22).

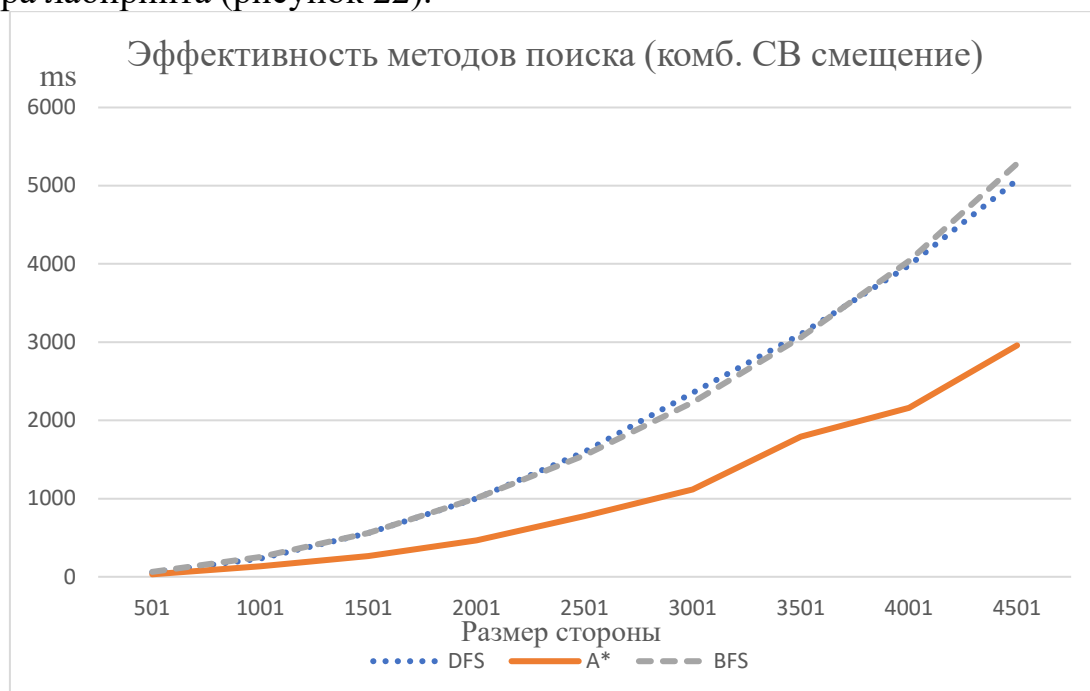


Рисунок 21 – Время поиска в лабиринте с комбинированным северо-восточным

По результатам тестирования видно, что в лабиринте, сформированном алгоритмом Комбинированного северо-восточного смещения, A star показывает наилучший результат, поскольку такой лабиринт не имеет как тенденции идти в ширь, так и вглубь.

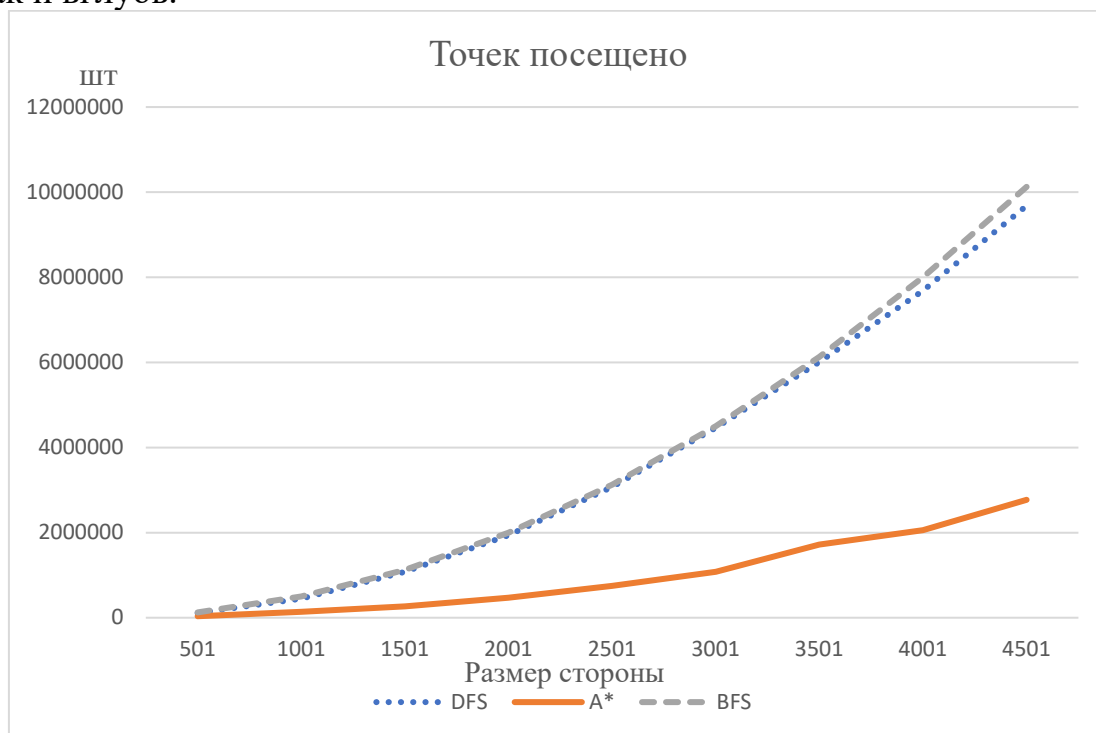


Рисунок 22 – Количество посещенных точек

По графику посещенных точек видно, что время поиска напрямую зависит от количества посещенных точек тестируемым алгоритмом. Можно сделать вывод, что вычисление эвристики алгоритма A star занимает меньше времени, нежели проход по лишним точкам.

Также тестирование проводилось на лабиринте, сформированном алгоритмом северо-восточного смещения с верной диагональной верной траекторией (рисунок 23). Особенностью такого лабиринта является то, что единственно верный путь в нём является самым отдалённым по диагонали. Такой путь наименее соответствует эвристике метода A star.

По графику посещенных точек (рисунок 24) видно, что эвристика поиска метода A star снижает свою эффективность, время, потраченное на вычисления никак не ускоряет алгоритм, а даже наоборот – замедляет его. Особенно это заметно, когда размерность лабиринта превысила 3500 – BFS стал искать путь быстрее при том же количестве посещенных точек.

Эффективность поиска в глубину DFS в таком лабиринте наоборот увеличивается, поскольку путь находится на глубине, однако в нём отсутствует стабильность – время на вычисление зависит от того повезёт ли найти финиш в глубине конкретного пути.

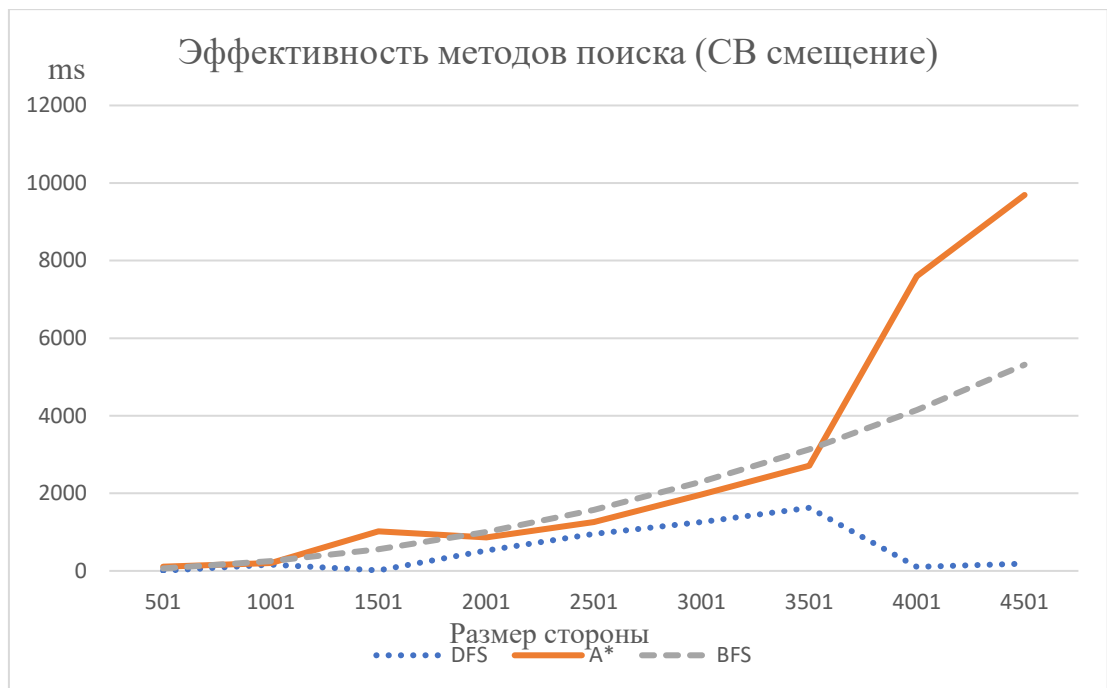


Рисунок 23 – Время поиска в лабиринте с северо-восточным смещением

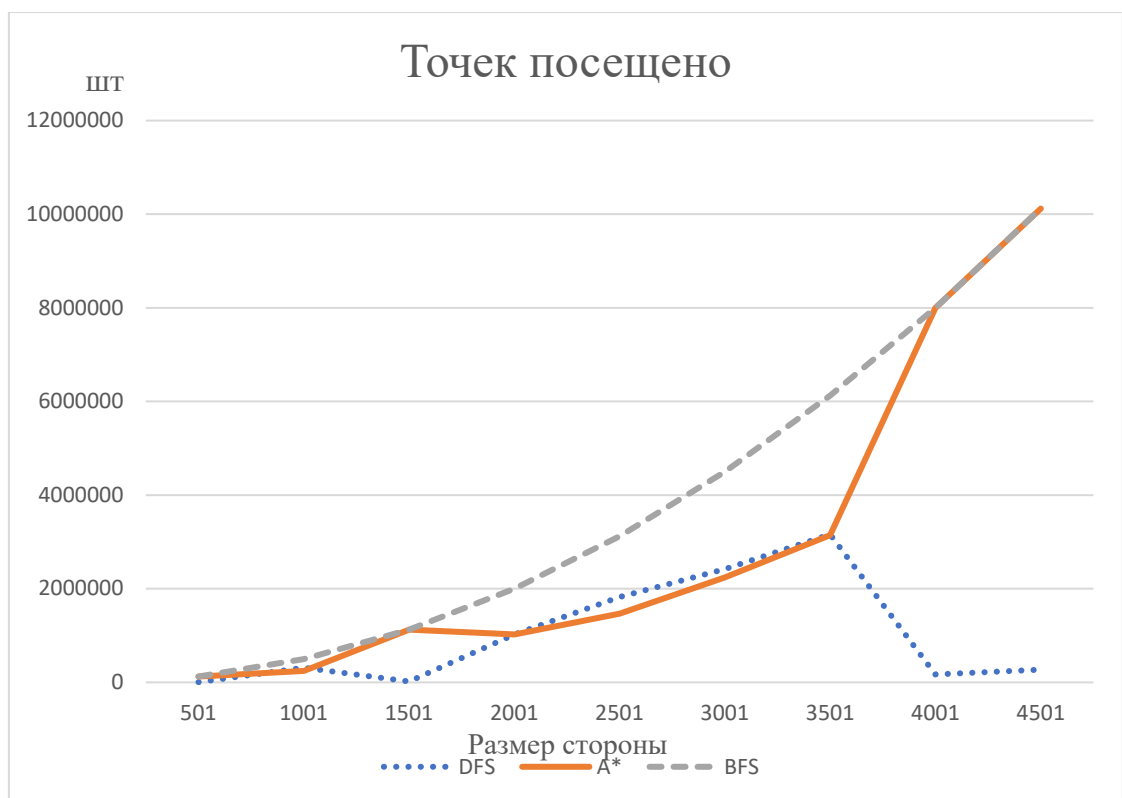


Рисунок 24 – Количество посещенных точек в лабиринте с северо-восточным смещением

Третье тестирование проводилось на северо-восточном смещении, но с максимально простым путем до финиша без изломов (рисунок 10). Такой лабиринт максимально соответствует эвристике A star и максимально не соответствует DFS, т.к. до финиша DFS зайдёт в глубину множества неверных путей.

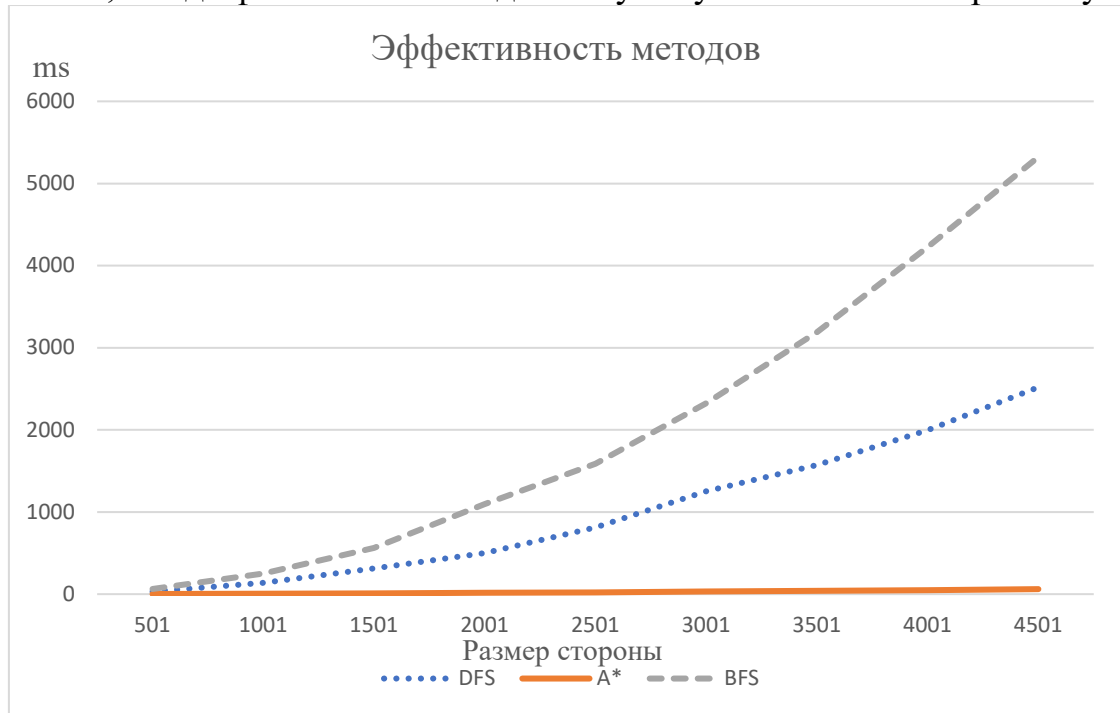


Рисунок 25 – Время поиска в лабиринте с прямой траекторией пути

3.4 Вывод

В этом разделе курсовой работы с помощью программы были протестированы в различных условиях описанные ранее методы поиска путей в лабиринте.

Было выявлено, что поиск в глубину DFS показывает себя эффективнее всего, когда искомый путь находится в глубине лабиринта за множеством перекрёстков.

Метод A star с эвристикой Манхэттенского расстояния имеет наибольшую эффективность при тестировании на лабиринтах, имеющих в решении наиболее прямолинейную траекторию с наименьшим числом поворотов. Стоит отметить, что использования другой эвристики с этим методом привело бы к смене его поведения.

ЗАКЛЮЧЕНИЕ

При выполнении курсовой работы были рассмотрены некоторые методы составления лабиринтов и поиска путей в них. На языке C++ была успешно написана и протестирована программная реализация лабиринта, позволяющая определить эффективность различных методов поиска в разных условиях.

Были выполнены следующие задачи:

- изучены теоретические основы алгоритмов создания лабиринтов;
- изучены теоретические основы алгоритмов поиска путей;
- разработан программный код алгоритмов на языке C++ с визуализацией поиска пути;
- проведено тестирование программы на различных видах лабиринтов.

Список использованных источников

1. HABR.COM : сайт. - Классические алгоритмы генерации лабиринтов. 2017 – URL: <https://habr.com/ru/articles/320140/> (Дата обращения: 24.12.2023) – Текст: электронный
2. HABR.COM : сайт. - Реализация алгоритма A* – 2017 – URL: <https://habr.com/ru/articles/331220/> (Дата обращения: 24.12.2023) – Текст: электронный
3. RU.WIKIPEDIA.ORG : сайт. – A* – 2018 – URL: https://ru.wikipedia.org/wiki/A* (Дата обращения: 24.12.2023) – Текст: электронный
4. RU.WIKIPEDIA.ORG : сайт. – BFS – 2018 – URL: <https://ru.wikipedia.org/wiki/BFS> (Дата обращения: 24.12.2023) – Текст: электронный
5. RU.WIKIPEDIA.ORG : сайт. – DFS – 2018 – URL: https://en.wikipedia.org/wiki/Depth-first_search (Дата обращения: 24.12.2023) – Текст: электронный

Приложение А Листинг программы

maze.cpp:

```
#ifndef MAZE_H
#define MAZE_H
//Класс точка
struct Point {
public:
    int x, y;
    Point(): x(0),y(0){}
    Point(int _x, int _y) : x(_x), y(_y) {}
    bool operator<(const Point& other) const {
        // В данном случае, сравниваем по x, а затем по y
        if (x == other.x) {
            return y < other.y;
        }
        return x < other.x;
    }
    bool operator==(const Point& other) const {
        return x == other.x && y == other.y;
    }
    void set(int _x,int _y){
        x = _x;
        y = _y;
    }
    Point& operator=(const Point& B) {
        x = B.x;
        y = B.y;
        return *this;
    }
};
//Класс лабиринт
class maze{
    int sX,sY;
    char **matrix; //H - Внешняя стена, X - Внутренняя стена
    Point start, end;
public:
    maze() : sX(0), sY(0), matrix(nullptr){}
    maze(int X,int Y){
        if (X%2!=1){
            sX = X+1;
            sY = X+1;
        }
        else{
            sX = X;
            sY = Y;
        }
        matrix = new char * [sX];
        start.set(1,1);
        end.set(sX-2,sY-2);
        for (int i = 0; i<sX; i++){
            matrix[i] = new char [sY];
        }
        //Поле внутренних стен
        for(int i = 0; i<sX; i++){
            for (int j = 0; j<sY; j++){
                matrix[i][j] = 'H';
            }
        }
        //Создание внутренних стен
        for(int i = 1; i<sX-1; i++){
            for (int j = 1; j<sY-1; j++){
                if (i%2 == 0 || j%2 == 0){
                    matrix[i][j] = 'X';
                }
                else{

```

```

        matrix[i][j] = ' ';
    }
}
}
//Копирование
maze(const maze& B){
    copy(B);
};
void copy(const maze& B){
    sX = B.sX;
    sY = B.sY;
    start = B.start;
    end = B.end;
    matrix = new char * [sX];
    for (int i = 0; i<sX; i++){
        matrix[i] = new char [sY];
    }
    for(int i = 0; i<sX; i++){
        for (int j = 0; j<sY; j++){
            matrix[i][j] = B.matrix[i][j];
        }
    }
}
maze& operator=(const maze& B) {
    copy(B);
    return *this;
}
//Деструктор
~maze(){Clear();}
void Clear(){
    for (int i = 0; i<sX; i++){
        delete[] matrix[i];
    }
    delete[] matrix;
    matrix=nullptr;
    sX = sY = 0;
}
//Вывод в поток с форматированием
void get(ostream& out){
    out<<"Лабиринт размером "<<sX<<" x "<<sY<<":\n";
    for(int i = 0; i<sX; i++){
        for (int j = 0; j<sY; j++){
            if (Point(i,j) == start)
                cout<<"\033[1;32m"<<'S'<<"\033[0m ";
            else if (Point(i,j) == end)
                cout<<"\033[1;32m"<<'E'<<"\033[0m ";
            else if (isdigit(matrix[i][j]))
                cout<<"\033[;90m"<<matrix[i][j]<<"\033[0m"<<" ";
            else if ((matrix[i][j]) == '$')
                cout<<"\033[33m"<<matrix[i][j]<<"\033[0m"<<" ";

            else
                cout<<matrix[i][j]<<" ";
        }
        cout<<"\n";
    }
};
//Начало и конец лабиринта
void dest(int x1,int y1,int x2, int y2){
    start.set(y1,x1);
    end.set(y2,x2);
}
//Задать размер
void setS(int _x,int _y){
    if (_x%2!=1){
        sX = _x+1;
        sY = _y+1;
    }
}

```

```

    }
    else{
        sX = _x;
        sY = _y;
    }
    start.set(1,1);
    end.set(sX-2,sY-2);
    matrix = new char * [sX];
    for (int i = 0; i<sX; i++){
        matrix[i] = new char [sY];
    }
    //Поле внутренних стен
    for(int i = 0; i<sX; i++){
        for (int j = 0; j<sY; j++){
            matrix[i][j] = 'H';
        }
    }
    //Создание внутренних стен
    for(int i = 1; i<sX-1; i++){
        for (int j = 1; j<sY-1; j++){
            if (i%2 == 0 || j%2 == 0){
                matrix[i][j] = 'X';
            }
            else{
                matrix[i][j] = ' ';
            }
        }
    }
}
int getS(){
    return sX;
}
//Задать значение координаты
void setCords(int a, int b, char val){
    matrix[a][b] = val;
    cout<<"| "<<matrix[a][b]<<"|\n";
}
//Создание лабиринта
//Метод северо-восточного смещения
void set_Northeast_alg(){
    srand(time(0));
    for(int i = 1; i<sX; i+=2){
        for (int j = 1; j<sY; j+=2){
            int a = rand()%2;
            if ((a || matrix[i-1][j] == 'H') && matrix[i][j+1] != 'H'){
                matrix[i][j+1] = ' ';
            }
            else{
                matrix[i-1][j] = ' ';
            }
        }
    }
}
//Восстановление рамки
for(int i = 0; i<sX; i++){
    for (int j = 0; j<sY; j++){
        if (i == 0 || j == 0 || i == sX-1 || j == sY-1)
            matrix[i][j] = 'H';
    }
}
};
//Метод комбинированного северо-восточного смещения
void set_Westeast_alg(){
    srand(time(0));
    for(int i = 1; i<sX; i+=2){
        for (int j = 1; j<sY; j+=2){
            if (i%16 > 8){
                int a = rand()%2;
                if ((a || matrix[i-1][j] == 'H') && matrix[i][j+1] != 'H'){

```

```

        matrix[i][j+1] = ' ';
    }
    else{
        matrix[i-1][j] = ' ';
    }
}
else{
    int a = rand()%2;
    if ((a || matrix[i-1][j] == 'H') && matrix[i][j-1] != 'H'){
        matrix[i][j-1] = ' ';
    }
    else{
        matrix[i-1][j] = ' ';
    }
}
}
}
//Восстановление рамки
for(int i = 0; i<sX; i++){
    for (int j = 0; j<sY; j++){
        if (i == 0 || j == 0 || i == sX-1 || j == sY-1)
            matrix[i][j] = 'H';
    }
}
};
//Метод Sidewinder
void set_Sidewinder_alg(){
    srand(time(0));
    for(int i = 1; i<sX-1; i+=2){
        if (i == 1){
            for (int j = 1; j<sY-1; j+=2){
                matrix[i][j+1] = ' ';
            }
        }
        else{
            int start = 1;
            while(start<sY){

                int c = 1 + rand()%4;
                int _c = 1 + rand()%c;
                if(start>1){
                    matrix[i-1][start] = ' ';
                }
                for (int j = start; j < start + c; j+=2){
                    matrix[i][j+1] = ' ';
                    if (j == start + _c){
                        matrix[i-1][j] = ' ';
                        matrix[i-2][j] = ' ';
                    }
                }
                start+=c;
            }
        }
    }
}
//Восстановление рамки
for(int i = 0; i<sX; i++){
    for (int j = 0; j<sY; j++){
        if (i == 0 || j == 0 || i == sX-1 || j == sY-1)
            matrix[i][j] = 'H';
    }
}
};
//РЕШЕНИЯ
//Эвристика A star
int heuristic(const Point& a, const Point& b) {
    //Manhattan distance
    return abs(a.x - b.x) + abs(a.y - b.y);
}

```

```

//Решение A star
int A_star() {
    int rows = sY;
    int cols = sX;
    int visitedCount = 0;
    // Массивы для хранения посещенных вершин и расстояний
    vector<vector<bool>> visited(rows, vector<bool>(cols, false));
    vector<vector<int>> distance(rows, vector<int>(cols, INT_MAX));
    // Приоритетная очередь для выполнения A* (минимальная куча)
    priority_queue<pair<int, Point>, vector<pair<int, Point>>, greater<pair<int, Point>>>
pq;

    // Начальная точка
    pq.push({0, start});
    visited[start.x][start.y] = true;
    distance[start.x][start.y] = 0;
    // Массивы для определения направлений движения: вверх, вниз, влево, вправо
    int dx[] = {-1, 1, 0, 0};
    int dy[] = {0, 0, -1, 1};
    // A* поиск
    while (!pq.empty()) {
        // Извлекаем текущую точку с минимальной стоимостью
        auto current = pq.top();
        pq.pop();
        matrix[current.second.x][current.second.y] = distance[current.second.x][current.second.y]%10 + '0';
        // Проверяем, достигли ли мы конечной точки
        if (current.second.x == end.x && current.second.y == end.y) {
            // Выводим расстояние до конечной точки
            //cout << "Минимальное расстояние до конечной точки: " << distance[end.x][end.y] << endl;
            // Восстанавливаем путь
            Point pathPoint = end;
            while (pathPoint.x != start.x || pathPoint.y != start.y) {
                matrix[pathPoint.x][pathPoint.y] = '$';
                for (int i = 0; i < 4; i++) {
                    int newX = pathPoint.x + dx[i];
                    int newY = pathPoint.y + dy[i];
                    if (isValid(newX, newY) && distance[newX][newY] == distance[pathPoint.x][pathPoint.y] - 1) {
                        pathPoint = Point(newX, newY);
                        break;
                    }
                }
            }
            break;
        }
        // Помечаем текущую точку как посещенную
        visited[current.second.x][current.second.y] = true;
        visitedCount++;
        // Проверяем соседей текущей точки
        for (int i = 0; i < 4; i++) {
            int newX = current.second.x + dx[i];
            int newY = current.second.y + dy[i];
            // Проверка на допустимость новой точки
            if (isValid(newX, newY) && !visited[newX][newY]) {
                // Вычисляем новую стоимость пути
                int newCost = distance[current.second.x][current.second.y] + 1;
                // Если новая стоимость меньше текущей, обновляем информацию и добавляем
                // точку в очередь
                if (newCost < distance[newX][newY]) {
                    distance[newX][newY] = newCost;
                    pq.push({newCost + heuristic(Point(newX, newY), end), Point(newX,
newY)}});
                }
            }
        }
    }
    cout<<"Посещено: "<<visitedCount<<"\n";
}

```

```

        return visitedCount;
    }
    //Решение DFS
    int DFS(){
        int rows = sY;
        int cols = sX;
        int visitedCount = 0;
        // Массив для хранения посещенных вершин
        vector<vector<bool>> visited(rows, vector<bool>(cols, false));
        // Массив для хранения расстояний до каждой вершины
        vector<vector<int>> distance(rows, vector<int>(cols, 0));
        // Стек для выполнения DFS
        stack<Point> stk;
        // Начальная вершина
        stk.push(start);
        visited[start.x][start.y] = true;
        // Массивы для определения направлений движения: вверх, вниз, влево, вправо
        int dx[] = {-1, 1, 0, 0};
        int dy[] = {0, 0, -1, 1};
        // DFS
        while (!stk.empty()) {
            Point current = stk.top();
            stk.pop();
            matrix[current.x][current.y] = distance[current.x][current.y]%10 + '0';
            // Проверяем, достигли ли конечной вершины
            if (current.x == end.x && current.y == end.y) {
                break;
            }
            // Проверяем соседей текущей вершины
            for (int i = 0; i < 4; i++) {
                int newX = current.x + dx[i];
                int newY = current.y + dy[i];
                // Проверка на допустимость новой вершины
                if (isValid(newX, newY) && !visited[newX][newY]) {
                    // Помечаем вершину как посещенную
                    visited[newX][newY] = true;
                    // Устанавливаем расстояние до новой вершины
                    distance[newX][newY] = distance[current.x][current.y] + 1;
                    // Добавляем новую вершину в стек
                    stk.push(Point(newX, newY));
                    visitedCount++;
                }
            }
        }
        // Выводим расстояние до конечной вершины
        cout << "Минимальное расстояние до конечной точки: " << distance[end.x][end.y] << endl;

        // Восстанавливаем путь
        int x = end.x, y = end.y;
        while (x != start.x || y != start.y) {
            matrix[x][y] = '$'; // Пример обозначения пути
            bool found = false;
            for (int i = 0; i < 4; i++) {
                int newX = x + dx[i];
                int newY = y + dy[i];
                if (isValid(newX, newY) && distance[newX][newY] == distance[x][y] - 1) {
                    x = newX;
                    y = newY;
                    found = true;
                    break;
                }
            }
            // Проверка наличия подходящего направления
            if (!found) {
                cout << "Ошибка: Нет подходящего направления!" << endl;
                break;
            }
        }
    }
}

```

```

        cout<<"Посещено: "<<visitedCount<<"\n";
        return visitedCount;
    }
    //Решение BFS
    int BFS() {
        int rows = sY;
        int cols = sX;
        int visitedCount = 0;
        // Массивы для хранения посещенных вершин и расстояний
        vector<vector<bool>> visited(rows, vector<bool>(cols, false));
        vector<vector<int>> distance(rows, vector<int>(cols, 0));
        // Переменные для хранения координат текущей вершины
        int currentX, currentY;
        // Очередь для выполнения BFS
        queue<Point> q;
        // Начальная вершина
        q.push(start);
        visited[start.x][start.y] = true;
        distance[start.x][start.y] = 0;
        // Массивы для определения направлений движения: вверх, вниз, влево, вправо
        int dx[] = {-1, 1, 0, 0};
        int dy[] = {0, 0, -1, 1};
        // BFS
        while (!q.empty()) {
            // Извлекаем вершину из очереди
            Point current = q.front();
            //пока не будут просмотрены новые в старые не идём
            matrix[current.x][current.y] = distance[current.x][current.y]%10 + '0';
            q.pop();
            currentX = current.x;
            currentY = current.y;
            // Проверяем соседей текущей вершины
            for (int i = 0; i < 4; i++) {
                int newX = currentX + dx[i];
                int newY = currentY + dy[i];
                // Проверка на допустимость новой вершины
                if (isValid(newX, newY) && !visited[newX][newY]) {
                    // Помечаем вершину как посещенную
                    visited[newX][newY] = true;
                    // Устанавливаем расстояние до новой вершины
                    distance[newX][newY] = distance[currentX][currentY] + 1;
                    // Добавляем новую вершину в очередь
                    q.push(Point(newX, newY));
                    visitedCount++;
                }
            }
        }
        // Восстанавливаем путь
        int x = end.x, y = end.y;
        while (x != start.x || y != start.y) {
            matrix[x][y] = '$';
            bool found = false;
            for (int i = 0; i < 4; i++) {
                int newX = x + dx[i];
                int newY = y + dy[i];
                bool a = isValid(newX, newY);
                if (a && distance[newX][newY] == distance[x][y] - 1) {
                    x = newX;
                    y = newY;
                    found = true;
                    break;
                }
            }
        }
        // Проверка наличия подходящего направления
        if (!found) {
            cout << "Ошибка: Нет подходящего направления!" << endl;
            break;
        }
    }

```



```

    }
    cout<<"Посещено: "<<visitedCount<<"\n";
    return visitedCount;
}
//Проверка точки на возможность в ней находиться
bool isValid(int _x, int _y) {
    return (_x >= 0 && _x < sX && _y >= 0 && _y < sY && ((matrix[_x][_y] != 'X' && matrix[_x][_y] != 'H') || (Point(_x,_y) == start || Point(_x,_y) == end))); //!visited[_x][_y]);
}
};
#endif //MAZE_H

main.cpp:
#include <iostream>
#include <fstream>
#include <string.h>
#include <fstream>
#include <algorithm>
#include <vector>
#include <queue>
#include <stack>
using namespace std;
#include "maze.h"
int main(void){
    system("chcp 1251");
    bool f = true;
    while (f){
        cout << "Главное меню\n"
        << "1) Работа с одним лабиринтом\n"
        << "2) Накопление статистических данных\n"
        << "0) <<<< Выход\n";

        int sw;
        bool ff = true;
        cin >> sw;
        switch (sw){
            while(ff){
                case 1:{
                    bool f2 = true;
                    int size;
                    cout << "Укажите размер лабиринта:\n";
                    cin >> size;
                    cout << "1) Построить лабиринт по алгоритму северо-восточного смещения\n"
                    << "2) Построить лабиринт по комб. алгоритму северо-восточного смещения\n"
                    << "3) Построить лабиринт по алгоритму Sidewinder\n";

                    int sw2;
                    cin >> sw2;
                    maze B;
                    maze A(size, size);
                    Point start(1,1);
                    Point end(size-2, size-2);
                    switch (sw2){
                        case 1:{
                            A.set_Northeast_alg();
                            break;
                        }
                        case 2:{
                            A.set_Westeast_alg();
                            break;
                        }
                        case 3:{
                            A.set_Sidewinder_alg();
                            break;
                        }
                    }

                    //Решение будет записываться в копию
                    B = A;
                    while (f2){
                        cout<< "1) Найти путь в лабиринте\n"

```



```

case 2:{
    cout << "1) Построить лабиринты по алгоритму северо-восточного смещения\n"
        << "2) Построить лабиринты по комб. алгоритму северо-восточного смещения\n"
        << "3) Построить лабиринты по алгоритму Sidewinder\n";
    int sw_1;
    int a1,b1,l;
    cin>>sw_1;
    cout<<"Введите размер лабиринта (от _ до _) и шаг через пробел:\n";
    cin>>a1>>b1>>l;
    maze *B;
    int N = (b1-a1)/l+1;
    B = new maze [N];

    switch (sw_1){
        case 1:{
            for (int i = 0; i < N-1; i++){
                B[i].setS(((i)*l)+a1,((i)*l)+a1);
                B[i].set_Northeast_alg();
                //B[i].dest(1,1,1,((i)*l)+a1-2);
            }
            break;
        }
        case 2:{
            for (int i = 0; i < N-1; i++){
                B[i].setS(((i)*l)+a1,((i)*l)+a1);
                B[i].set_Westeast_alg();
                //B[i].dest(1,1,1,((i)*l)+a1-2);
            }
            break;
        }
        case 3:{
            for (int i = 0; i < N-1; i++){
                B[i].setS(((i)*l)+a1,((i)*l)+a1);
                B[i].set_Sidewinder_alg();
                //B[i].dest(1,1,1,((i)*l)+a1-2);
            }
            break;
        }
    }
}

bool ff2 = true;
while (ff2){
    maze *C = new maze[N];
    for (int i = 0; i < N; i++){
        C[i] = B[i];
    }
    cout << "1) Поиск пути по алгоритму A*\n"
        << "2) Поиск пути по алгоритму DFS\n"
        << "3) Поиск пути по алгоритму BFS\n"
        << "0) <<<\n";
    int sw_2;
    cin>>sw_2;
    ofstream fout("results.txt");
    int c = 0;
    switch(sw_2){
        case 1:{
            for (int i = 0; i<N-1; i++){
                int t_1 = clock();
                c = B[i].A_star();
                int t_2 = clock();
                int ans = t_2 - t_1;
                cout<<"Путь найден за "<<ans<<"ms\n";
                fout<<B[i].getS()<<" "<<c<<" "<<ans<<"\n";
                //B[i].get(cout);
            }
            break;
        }
        case 2:{

```

```

        for (int i = 0; i<N-1; i++){
            int t_1 = clock();
            c = B[i].DFS();
            int t_2 = clock();
            int ans = t_2 - t_1;
            cout<<"Путь найден за "<<ans<<"ms\n";
            fout<<B[i].getS()<<" "<<c<<" "<<ans<<"\n";
            //B[i].get(cout);
        }
        break;
    }
    case 3:{
        for (int i = 0; i<N-1; i++){
            int t_1 = clock();
            c = B[i].BFS();
            int t_2 = clock();
            int ans = t_2 - t_1;
            cout<<"Путь найден за "<<ans<<"ms\n";
            fout<<B[i].getS()<<" "<<c<<" "<<ans<<"\n";
            //B[i].get(cout);
        }
        break;
    }
    case 0:{
        ff2 = false;
        break;
    }
    }
    fout.close();
}
break;
}
}
}
}
}

```

Приложение Б Проверка на оригинальность

Проверка проводилась на сайте <https://users.antiplagiat.ru>.
Результат проверки можно увидеть на рисунке Б.1.

ОТЧЕТ №1

Проверено: 26.12.2023 12:47:36

?

Начало проверки: 26.12.2023 12:47:18

Длительность проверки: 00:00:18


Модуль поиска

Совпадения

Самоцитирования

Цитирования

Оригинальность

Интернет Free 

4.77%

0%

0%

95.23%

Рисунок Б.1 – Результат проверки на оригинальность