МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ
ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ им. А. И. ГЕРЦЕНА»

Направление подготовки
09.03.01 Информатика и вычислительная техника

Направленность (профиль)
«Технологии разработки программного обеспечения»

Выпускная квалификационная работа

Разработка веб-приложения по визуализации алгоритмов поиска на графе

Обучающегося 4 курса
очной формы обучения
Кононова Сергея Владимировича

Руководитель выпускной квалификационной
работы: кандидат педагогических наук, доцент,
Илья Борисович Государев

Санкт-Петербург
2024

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность данной работы обусловлена тем, что нахождение кратчайшего пути из точки А в точку Б является одной из самых распространенных задач в повседневной жизни каждого человека. В математике же разработка алгоритмов для решения лабиринтов на сегодняшний день является такой же востребованной. В русскоязычном интернете на данный момент нет такого источника информации об алгоритмах поиска решений лабиринтов, где была бы возможность не только получить развернутое объяснение самих алгоритмов в удобном формате, но и «песочница», где каждый из этих алгоритмов можно было бы изучить визуально, увидеть, как они работают в тех или иных условиях.

Цель – разработка интерактивного веб-приложения по изучению и визуализации алгоритмов поиска на графе. На основе поставленной цели были составлены следующие задачи:

1. Изучить существующие алгоритмы поиска на графе и выбрать наиболее подходящие для визуализации.
2. Разработать дизайн и интерфейс веб-приложения, обеспечивающий удобное и интуитивное понятное взаимодействие с веб-приложения.
3. Реализовать выбранные алгоритмы поиска на графе и визуализировать их работу с помощью графических элементов.
4. Провести тестирование и отладку веб-приложения.
5. Обеспечить доступность и надежность работы веб-приложения.

Предполагаемый результат работы - интерактивное веб-приложение, которое будет предоставлять пользователям возможность изучать и визуализировать алгоритмы поиска на графе. Таким образом, разработав интерактивный интернет-ресурс с исчерпывающей информацией о проблеме решения лабиринтов, и возможностью визуально исследовать алгоритмы поиска решений лабиринтов самостоятельно, появится централизованный достоверный ресурс для самостоятельного изучения данной темы, что будет способствовать повышению

уровня подготовки студентов и специалистов в области информатики, а также будет полезно для всех, кто интересуется алгоритмами и математикой.

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

2.1. ПОНЯТИЕ ГРАФА И ЕГО СВОЙСТВА

2.1.1. ОПРЕДЕЛЕНИЕ ГРАФА. ТИПЫ ГРАФОВ

Граф – математический объект, состоящий из двух множеств. Одно из них – любое конечное множество, его элементы называются вершинами графа. Другое множество состоит из пар вершин, эти пары называются ребрами графа. Если множество вершин графа обозначено буквой *V*, множество ребер – буквой *E*, а сам граф – буквой *G*, то пишут *G= (V, E)*.

В данном определении необходимо уточнить, являются ли рёбра упорядоченными или неупорядоченными парами вершин. В частности, следует определить, считаются ли рёбра (a, b) и (b, a) различными или одним и тем же ребром. Если рёбра представляют собой упорядоченные пары, то граф называется ориентированным (орграфом). В противном случае, если рёбра неупорядоченные, то граф является неориентированным. Существует множество различных типов графов, которые используются в теории графов для моделирования различных систем и процессов. Например:

1. Простой граф - граф, который не содержит петель или кратных ребер.
2. Ориентированный граф (дираф) - граф, в котором ребра имеют направление.
3. Мультиграф - граф, который может содержать кратные ребра, но не петли.
4. Псевдограф - граф, который может содержать петли, но не кратные ребра.
5. Взвешенный граф - граф, в котором каждому ребру присваивается вес, представляющий собой числовое значение.
6. Планарный граф - граф, который может быть нарисован на плоскости без пересечений ребер.
7. Регулярный граф - граф, в котором все вершины имеют одинаковое число инцидентных ребер.
8. Дерево - связный граф без циклов.

9. Остова́ - подграф графа, который содержит все его вершины и является деревом.
10. Полный граф - граф, в котором каждая пара вершин соединена ребром.

Говорят, что ребро (a, b) соединяет вершины a и b. Заметим, что в графе может быть не более одного ребра, соединяющего две данные вершины. Ребро типа (a, a), т. е. соединяющее вершину с ней же самой, называют петлей. Иногда петли разрешаются, иногда запрещаются. В последнем случае говорят, что рассматриваются графы без петель. В дальнейшем, если не оговаривается иное, под графом понимается неориентированный граф без петель, такие графы называют обыкновенными.[1]

Дерево является одним из самых распространенных типов графа в теории графов и прикладных областях из-за своей связности без циклов, эффективности вычислительных ресурсов и гибкости. Связность без циклов обеспечивает единственный путь между любыми двумя вершинами, что делает дерево идеальным для моделирования иерархических систем. Эффективность вычислительных ресурсов обеспечивается минимальным количеством ребер для соединения всех вершин и отсутствием необходимости хранения информации о циклах, что делает дерево идеальным для приложений, где скорость и эффективность вычислений критичны. Гибкость обеспечивается легким изменением путем добавления или удаления вершин и ребер, что делает дерево идеальным для приложений, где необходимо динамическое моделирование или адаптация к меняющимся условиям. (рисунок 1). Примером дерева может служить структура каталога файлов в компьютерной системе. Существуют также более сложные структуры, такие как сильно связные компоненты, мосты и

[1] Алексеев, В. Е. Теория графов : учебное пособие / В. Е. Алексеев, Д. В. Захарова. — Нижний Новгород : ННГУ им. Н. И. Лобачевского, 2017. — 119 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/153421 — Режим доступа: для авториз. пользователей. — С. 6.

шарниры[2], которые используются для анализа связности и цикличности графов.

Рис 1. Пример древовидного графа

Одним из типов частного графа для моделирования двумерного лабиринта, где вершины могут быть либо пустыми, либо препятствиями, является граф клеточного разбиения или grid graph. В этом графе каждая вершина соответствует клетке лабиринта, а ребра соединяют соседние клетки. При этом, если клетка является препятствием, то соответствующая вершина графа не соединена ребрами с другими вершинами.

Grid graph является простым и наглядным способом моделирования лабиринта, и обладает рядом преимуществ. Во-первых, он позволяет легко определить, какие клетки являются доступными, а какие нет, и находить пути между доступными клетками. Во-вторых, он позволяет использовать стандартные алгоритмы обхода

[2] «Шарнир (точка сочленения) в графе – вершина, при удалении которой увеличивается число компонент связности.» (Алексеев, В. Е. Теория графов : учебное пособие / В. Е. Алексеев, Д. В. Захарова. — Нижний Новгород : ННГУ им. Н. И. Лобачевского, 2017. — 119 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/153421 — Режим доступа: для авториз. пользователей. — С. 13.).

графа для поиска пути, например алгоритм BFS или алгоритм Дейкстры. В-третьих, он позволяет легко модифицировать лабиринт, добавляя или удаляя препятствия, и соответствующим образом изменяя граф.

Однако, grid graph также обладает рядом ограничений и недостатков. Во-первых, он не позволяет моделировать лабиринты с диагональными переходами между клетками. Во-вторых, он может быть неэффективен для моделирования лабиринтов с большим количеством препятствий, поскольку требует большого количества вершин и ребер для их представления. В-третьих, он может быть неприменим для моделирования лабиринтов с неправильной геометрией или сложными пространственными отношениями между клетками.

В данной курсовой работе именно grid graph является основным полем для экспериментов и исследований, поскольку он позволяет моделировать и анализировать различные алгоритмы обхода графа, поиска кратчайшего пути и нахождения связных компонент в простых и наглядных двумерных лабиринтах. Кроме того, grid graph может быть использован для моделирования и анализа более сложных графов, таких как планарные графы, графы с весами на ребрах, и графы с несколькими связанными компонентами, путем преобразования или расширения базовой структуры клеточного разбиения.

2.1.3. ФУНДАМЕНТАЛЬНЫЕ КОНЦЕПЦИИ В ТЕОРИИ ГРАФОВ

В теории графов связность и циклы являются фундаментальными концепциями, которые используются для изучения и моделирования сложных систем и процессов. Граф называется связным, если между любыми двумя его вершинами существует путь, т. е. последовательность ребер, соединяющих эти вершины. В противном случае граф называется несвязным. Связность графа является важной характеристикой, которая используется во многих приложениях теории графов, таких как анализ сетей, планирование маршрутов и др.

Маршрут в графе — это последовательность (a_1,a_2,...,a_k) (a_1,a_2,...,a_k) такая, что все пары (a_i, a_i+1), i=1,2,...,k–1, (a_i, a_i+1), i=1,2,...,k–1, являются ребрами графа. Эти ребра называются ребрами маршрута. Число k–1 называется длиной маршрута. Маршрут называется замкнутым, если a_1=a_k a_1=a_k.

Обратите внимание, что ребра маршрута не обязательно являются различными. Например, если граф имеет ребро (a,b) (a,b), то последовательность (a,b,a,b,a,b) (a,b,a,b,a,b) является маршрутом длиной 5, а (a,b,a,b,a) (a,b,a,b,a) - замкнутым маршрутом длины 4. Путь — это маршрут, в котором все ребра попарно различны.

Цикл — это замкнутый путь. Рассмотрим следующий граф G (рисунок 2) и несколько последовательностей вершин. Последовательность (4,3,2,1) (4,3,2,1) не является маршрутом в G, поскольку (2,1) (2,1) не является ребром в G.

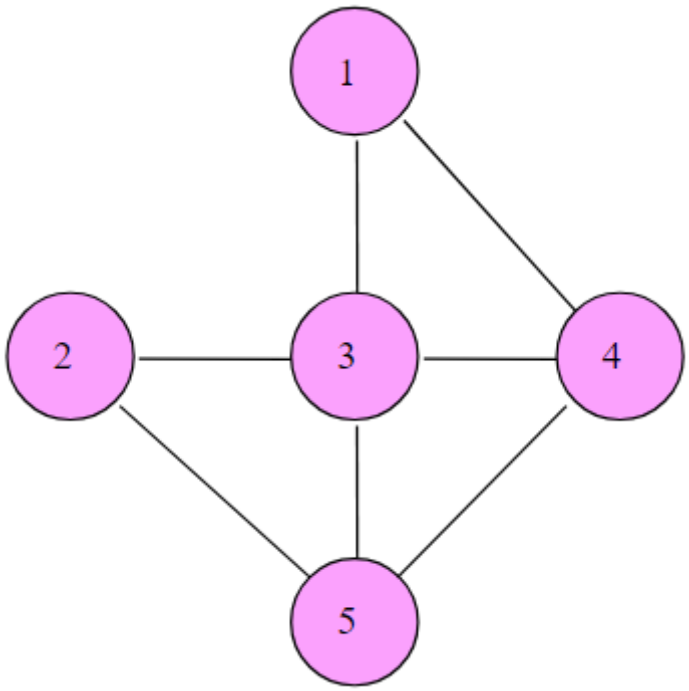

Рис 2. Граф G

Последовательность (4,1,3,5,2,3,1) (4,1,3,5,2,3,1) является маршрутом в G. Однако, это не путь, поскольку ребро (1,3) (1,3) встречается дважды. Этот маршрут не замкнутый. Последовательность (1,3,4,5,2) (1,3,4,5,2) является путем в G, но это не замкнутый путь. Последовательность (1,3,4,5,3,4,1) (1,3,4,5,3,4,1) является замкнутым маршрутом в G. Однако, это не цикл, поскольку ребро (3,4) (3,4) встречается дважды. Наконец, последовательность (1,4,5,3,1) (1,4,5,3,1) является циклом в G.

Существует несколько утверждений о путях и циклах в графе:

ТЕОРЕМА 1. Если любая вершина графа имеет степень не менее 2, то граф содержит цикл.

ТЕОРЕМА 2. Пусть $G=(V, E)$ $G=(V, E)$ - граф с $|V|=n$, $|E|=m$ $|V|=n$, $|E|=m$ и неравенство $m \geq n$; $m \geq n$ выполняется. Тогда G содержит цикл.

Если между любыми двумя вершинами графа существует путь, то такой граф называется связным. В случае, когда граф не является связным, он разделяется на

несколько связных подграфов, между которыми отсутствуют ребра. Такие подграфы называются компонентами связности.

Граф H расположенный на рисунке 3 имеет 7 связанных компонентов (включая 3 изолированные вершины: каждая из них является уникальным компонентом)

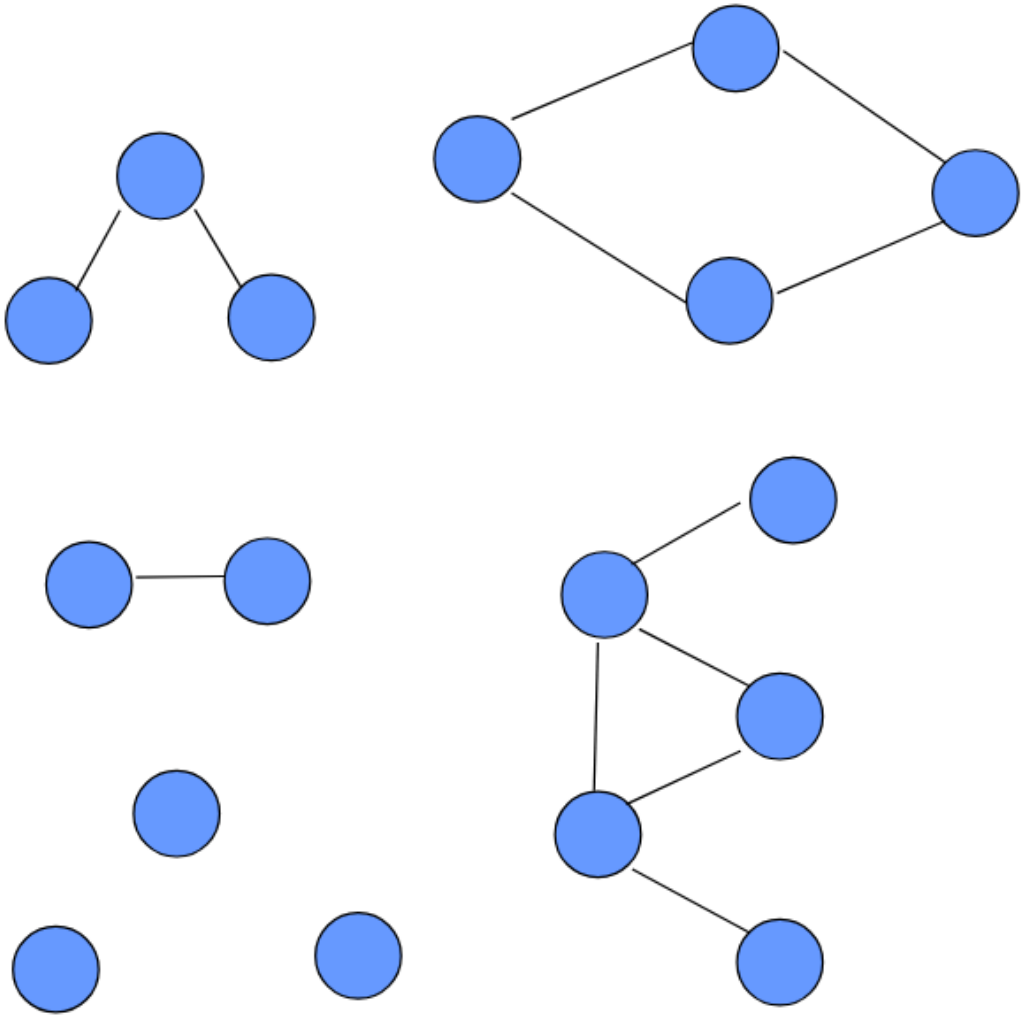

Рис 3. Граф H

ТЕОРЕМА 3. Для любого связного графа G = (V, E) с |V| = n, |E| = m выполняется неравенство $m \geq n - 1$.

Мост в графе - это ребро, удаление которого увеличивает количество связных компонент. Граф на рисунке 4 имеет 6 мостов (жирные красные линии на рисунке).

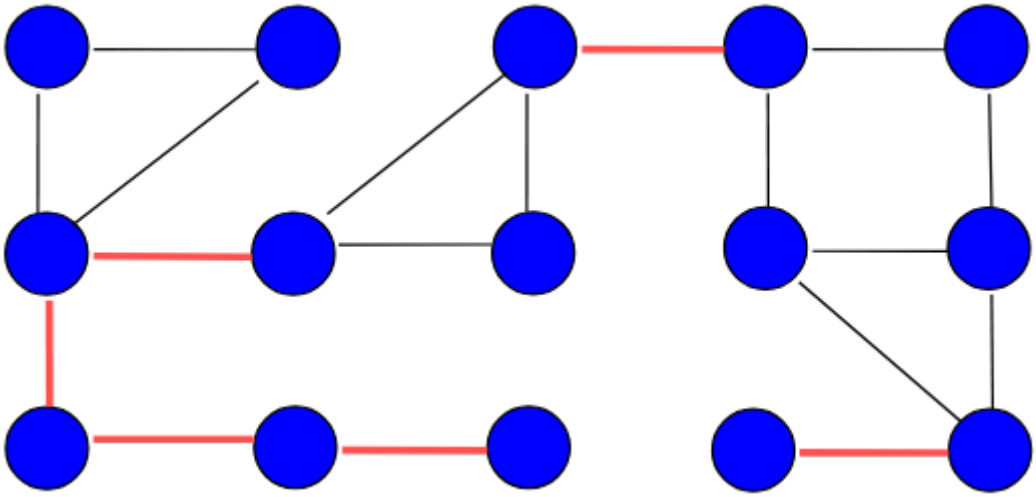

Рис. 4. Граф с отмеченными мостами

ТЕОРЕМА 4. Ребро является истмусом в графе тогда и только тогда, когда оно не принадлежит любому циклу.

2.2. АЛГОРИТМЫ ПОИСКА НА ГРАФЕ

Исследование графов является важной частью изучения алгоритмов и структур данных в информатике. Одной из ключевых задач при работе с графами является поиск эффективных путей между вершинами. В этой главе будут рассмотрены самые распространенные алгоритмы поиска на графе, их принципы и особенности. Два самых распространённых алгоритма являются поиск в глубину (DFS) и поиск в ширину (BFS). Оба этих алгоритма имеют свои преимущества и недостатки, которые будут рассмотрены далее. Кроме того, в этой главе будет представлен алгоритм A*, который является более сложным, но и более эффективным алгоритмом поиска на графе.

Принципы, лежащие в основе этих алгоритмов, различаются существенно. Алгоритм DFS основан на рекурсивном обходе графа, в то время как алгоритм BFS использует очередь для обхода вершин. Алгоритм A* же использует более сложную очередь, и применяет эвристические функции для нахождения оптимального пути. В следующих параграфах будет более подробно рассмотрено каждый из этих алгоритмов, их реализация и применение в практике.

2.2.1. АЛГОРИТМ ПОИСКА В ГЛУБИНУ (DFS)

Алгоритм поиска в глубину (DFS) представляет собой один из фундаментальных алгоритмов обхода графа. Суть его работы заключается в том, что он проходит в глубь через по всем вершинам ветки графа до самого последнего уровня, затем возвращается назад и продолжает обход с той вершины, где он остановился. Просмотр начинается с начальной вершины *v*, которая будет считаться «открытой». Затем выбирается вершина u, которая является смежной с *v*, и она также становится «открытой» (помечается значением 1). Затем процесс повторяется с одной из вершин, смежных с *u*.

На каждом шаге, если мы работаем с вершиной q и нет смежных с q вершин, которые еще не были открыты, то мы возвращаемся на предыдущий шаг. Если мы возвращаемся в начальную вершину, то это означает, что обход графа завершен. Во время обхода графа строится дерево поиска в глубину. Вершина *v*, через которую была открыта вершина *u*, является ее предком. Эта информация сохраняется в массиве Parent.

Для отслеживания состояния вершины (неоткрытая, открытая или обработанная) используется массив пометок Mark. Если вершина *i* открыта, то Mark[i] = 1, если обработана - 2, в исходном состоянии Mark[i] = 0. Для хранения последовательности открытых, но еще не обработанных вершин используется стек. Эта структура данных обеспечивает возврат к предыдущей открытой вершине.[3]

Алгоритм DFS можно описать следующим образом (пример на рисунке 5):

1. Выбираем любую вершину графа в качестве начальной.
2. Отмечаем эту вершину как посещенную.
3. Переходим к одной из соседних не посещённых вершин.
4. Повторяем шаги 2–3, пока все соседние вершины не будут посещены.

[3] Дольников, В. Л. Основные алгоритмы на графах : текст лекций / В. Л. Дольников, О. П. Якимова; Яросл. гос. ун-т им. П. Г. Демидова. – Ярославль : ЯрГУ, 2011. – С.6.

5. Возвращаемся к предыдущей вершине и повторяем шаги 4–5, пока не вернемся в начальную вершину.

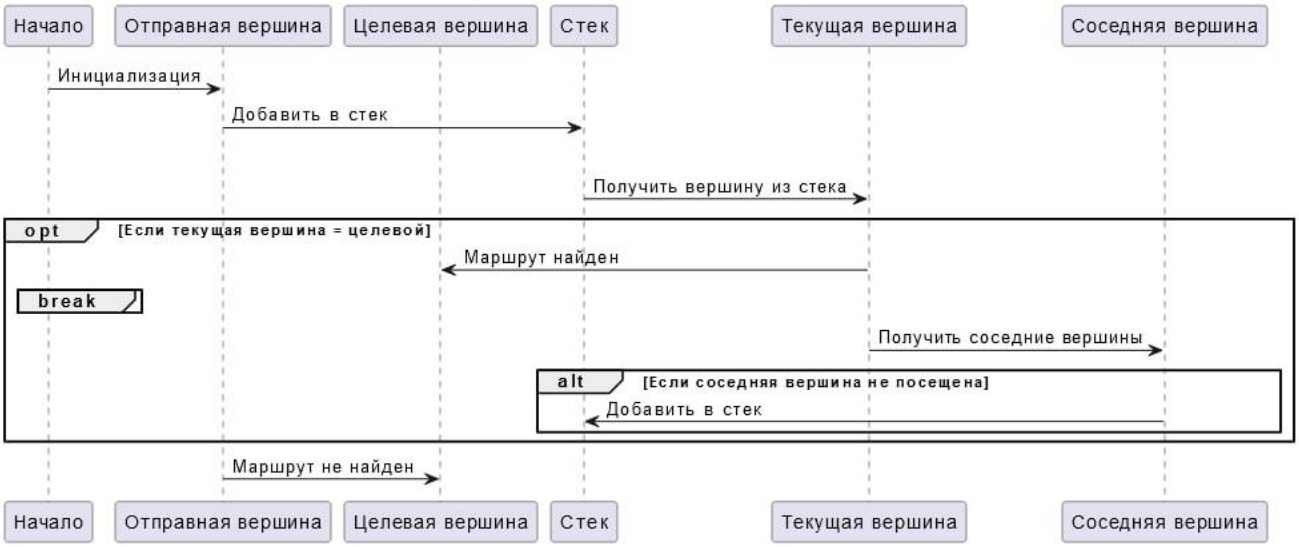

Рис. 5. UML диаграмма последовательности алгоритма DFS

Плюсы алгоритма DFS:

- Простота реализации
- Низкие требования к памяти при обходе дерева.
- Возможность обнаружения циклов в графе.

Минусы алгоритма DFS:

- Высокие требования к памяти при обходе графа с большим количеством вершин.
- Неэффективность при поиске кратчайшего пути между вершинами.
- Алгоритм может зациклиться при обходе графа с циклами, если не применять дополнительные проверки.

2.2.2. АЛГОРИТМ ПОИСКА В ШИРИНУ (BFS)

Алгоритм поиска в ширину (BFS) также является одним из самых популярных методов обхода графа, который используется для поиска кратчайшего пути от одной вершины графа до всех остальных.

Пусть задан граф $G = (V, E)$ и выделена начальная вершина v. Алгоритм поиска в ширину осуществляет систематический обход всех ребер графа G с целью «открытия» всех вершин, которые можно достичь из вершины v. В ходе обхода формируется дерево поиска в ширину, корнем которого является начальная вершина, и которое включает все достижимые вершины. Следует отметить, что расстояние (количество ребер) от корневой вершины до любой вершины этого дерева является кратчайшим. Название «поиск в ширину» объясняется тем, что при обходе графа помечаются все вершины, находящиеся на расстоянии k, прежде чем начнется обработка любой вершины, находящейся на расстоянии k+1.[4]

Пошаговый принцип работы алгоритма (рисунок 6):

1. Выбираем стартовую вершину графа и помечаем ее как посещенную.
2. Добавляем всех соседей стартовой вершины в очередь.
3. Извлекаем вершину из очереди и помечаем ее как посещенную.
4. Добавляем всех не посещённых соседей текущей вершины в очередь.
5. Повторяем шаги 3–4, пока очередь не станет пустой.

[4] Дольников, В. Л. Основные алгоритмы на графах : текст лекций / В. Л. Дольников, О. П. Якимова; Яросл. гос. ун-т им. П. Г. Демидова. – Ярославль : ЯрГУ, 2011. – С.5

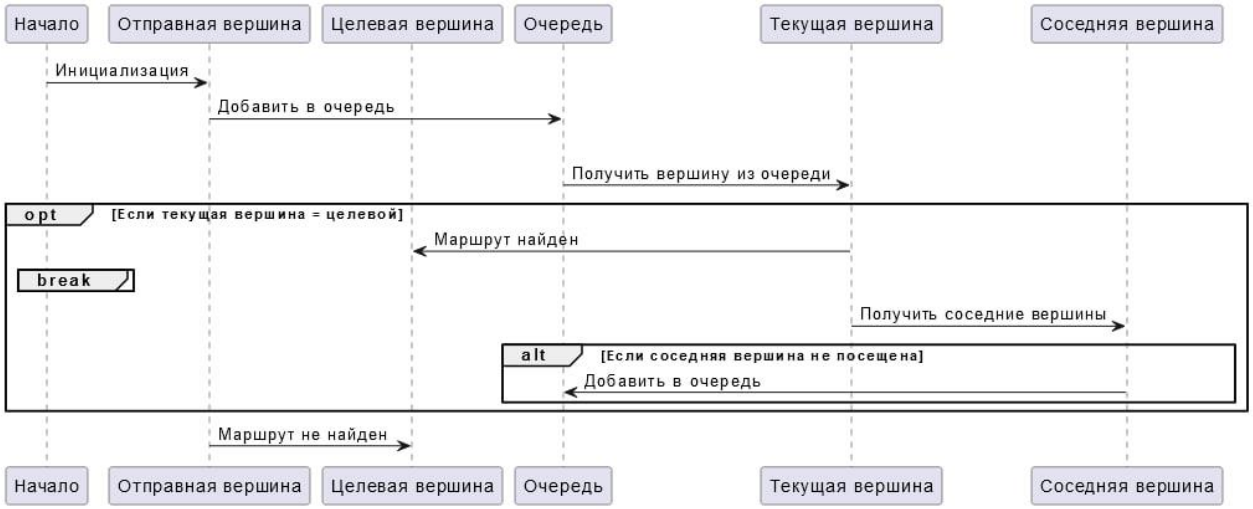

Рис. 6. UML диаграмма последовательности алгоритма BFS

Плюсы алгоритма BFS:

1. Гарантирует нахождение кратчайшего пути от стартовой вершины до всех остальных вершин графа.
2. Может быть использован для поиска любой вершины в графе, если она достижима из стартовой вершины.
3. Может быть модифицирован для решения других задач, таких как поиск компонент связности в графе.

Минусы алгоритма BFS:

1. Использует большое количество памяти для хранения очереди, особенно при работе с большими графами.
2. Может быть медленным при работе с графами, содержащими вершины с большим количеством соседей.
3. Не может быть использован для поиска кратчайшего пути в графе с отрицательными весами ребер.

Дополнительные сведения:

1. Алгоритм BFS может быть реализован с помощью рекурсии, но обычно используется итеративная реализация с помощью очереди.
2. Алгоритм BFS может быть использован для обхода как неориентированных, так и ориентированных графов.

3. Время работы алгоритма BFS составляет O (|V| + |E|), где |V| - количество вершин в графе, а |E| - количество ребер.

2.2.3. АЛГОРИТМ A*

Алгоритм A* является одним из наиболее популярных и эффективных алгоритмов поиска кратчайшего пути в графах, особенно в случаях, когда графы являются взвешенными. Он был разработан в 1968 году Питером Хартом, Нильсом Нильссоном и Бертрамом Рафаэлем. Алгоритм A* комбинирует в себе преимущества алгоритма Дейкстры и алгоритма поиска с итеративным углублением.

Алгоритм A* использует эвристическую функцию для оценки расстояния от текущего узла до целевого узла, что позволяет алгоритму направлять поиск в наиболее перспективные области графа. Эвристическая функция должна быть допустимой, то есть она не должна переоценивать истинное расстояние до цели. При этом, чем точнее эвристика, тем лучше работает алгоритм.

Ниже на рисунке 7 приведена UML диаграмма, описывающая алгоритм A*:

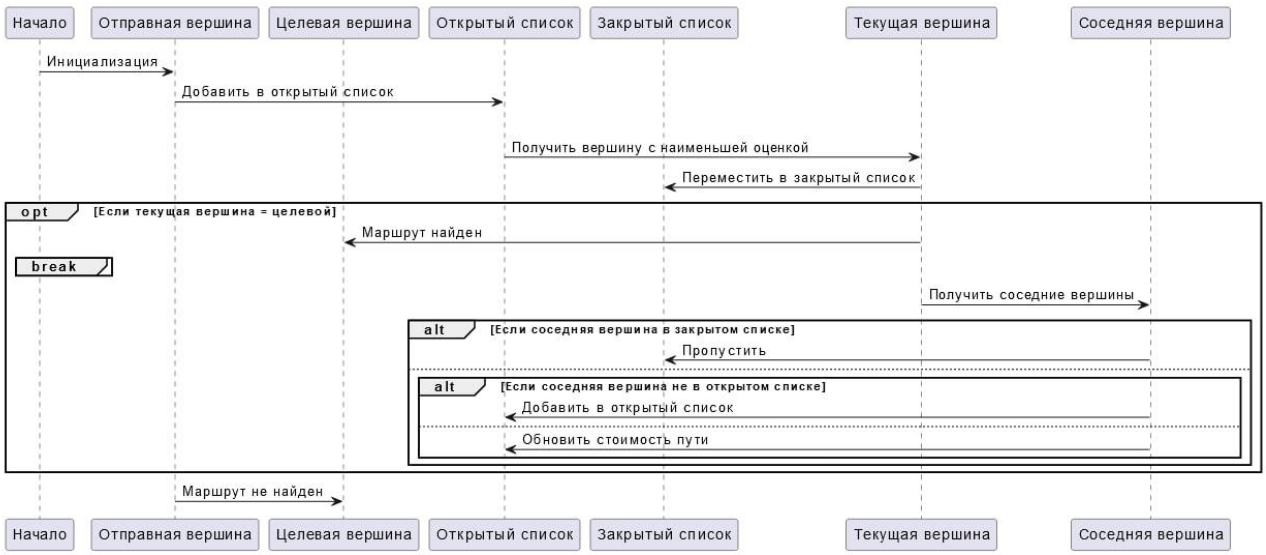

Рис. 7. UML диаграмма последовательности алгоритма A*

Плюсы алгоритма A*:

1. Эффективность: Алгоритм A* обычно работает быстрее, чем другие алгоритмы поиска кратчайшего пути, благодаря использованию эвристической функции.

2. Оптимальность: Если эвристическая функция является допустимой, алгоритм A* всегда находит кратчайший путь между начальным и целевым узлами.
3. Гибкость: Алгоритм A* может быть адаптирован для решения различных задач, связанных с поиском пути, например, для поиска пути в играх, робототехнике и навигационных системах.

Минусы алгоритма A*:

1. Сложность реализации: Алгоритм A* требует больше времени и усилий для реализации, по сравнению с более простыми алгоритмами, такими как алгоритм Дейкстры.
2. Зависимость от эвристики: Эффективность алгоритма A* сильно зависит от качества эвристической функции. Недостаточно точная эвристика может привести к тому, что алгоритм будет работать медленнее, чем другие алгоритмы.
3. Память: Алгоритм A* может потреблять большое количество памяти, особенно при поиске пути в больших графах, что может стать проблемой для некоторых приложений.

2.3. АЛГОРИТМЫ ГЕНЕРАЦИИ ЛАБИРИНТОВ

Для решения лабиринтов сначала нужно получить или создать этот лабиринт. Этим вопросом как раз занимаются алгоритмы генерации лабиринтов. Они решают проблему автоматического создания сложных и уникальных лабиринтов, которые могут быть использованы в различных сферах, например в генерации уровней в компьютерной игре.

Генерация лабиринтов является нетривиальной задачей, которая требует решения несколько взаимосвязанных проблем. Во-первых, лабиринт должен быть связным, то есть существовать путь между любыми двумя точками внутри него. Во-вторых, лабиринт должен быть достаточно сложным, чтобы представлять интерес для исследования, но при этом не быть слишком запутанным, чтобы избежать фрустрации у пользователей. В-третьих, лабиринт должен быть уникальным, чтобы обеспечить разнообразие при повторном использовании.

Существует множество алгоритмов генерации лабиринтов, каждый из которых имеет свои преимущества и недостатки.

В целом, алгоритмы генерации лабиринтов представляют собой интересную и сложную область исследований, которая объединяет в себе и теорию и практику.

2.3.1. АЛГОРИТМ РОБЕРТА К. ПРИМА

Алгоритм Прима был впервые разработан в 1930 году чешским математиком Войтехом Ярником, но получил свое название в честь Роберта К. Прима, ученого, изучающего компьютерные науки, который переоткрыл его в 1957 году. Он работает аналогично алгоритму Дейкстры, начиная с одной точки сетки и затем двигаясь наружу, как текущая вода, но в случае алгоритма Прима он делает больше, чем просто измеряет расстояния и затраты в ходе поиска. Конечный результат алгоритма Прима — это минимальное остовное дерево — или, говоря иначе, лабиринт. В данной курсовой работе будет рассмотрена упрощенная версия алгоритма Прима.

Алгоритм Прима для создания минимального остовного дерева в упрощенном варианте работает следующим образом (диаграмма представлена на рисунке 8):

1. Начинается со случайной вершины, которая становится частью лабиринта.
2. Из списка всех доступных ребер, соединяющих вершины лабиринта с вершинами, которые еще не являются частью лабиринта, выбирается ребро случайным образом.
3. Выбранное ребро и соответствующая вершина добавляются к лабиринту.
4. Шаги 3–4 повторяются, пока не останется больше ребер для рассмотрения.
5. Лабиринт считается завершенным, когда все вершины будут связаны в единое дерево (все ячейки посещены).

Особенностью этого алгоритма является отсутствие весов у ребер и их неупорядоченность, что позволяет сохранять их в виде простого списка и выбирать элементы из списка очень быстро, за постоянное время. Кроме того, текстура полученного лабиринта не будет иметь много ответвлений и извилистых путей с более простым решением, чем в алгоритме не упрощенного типа.

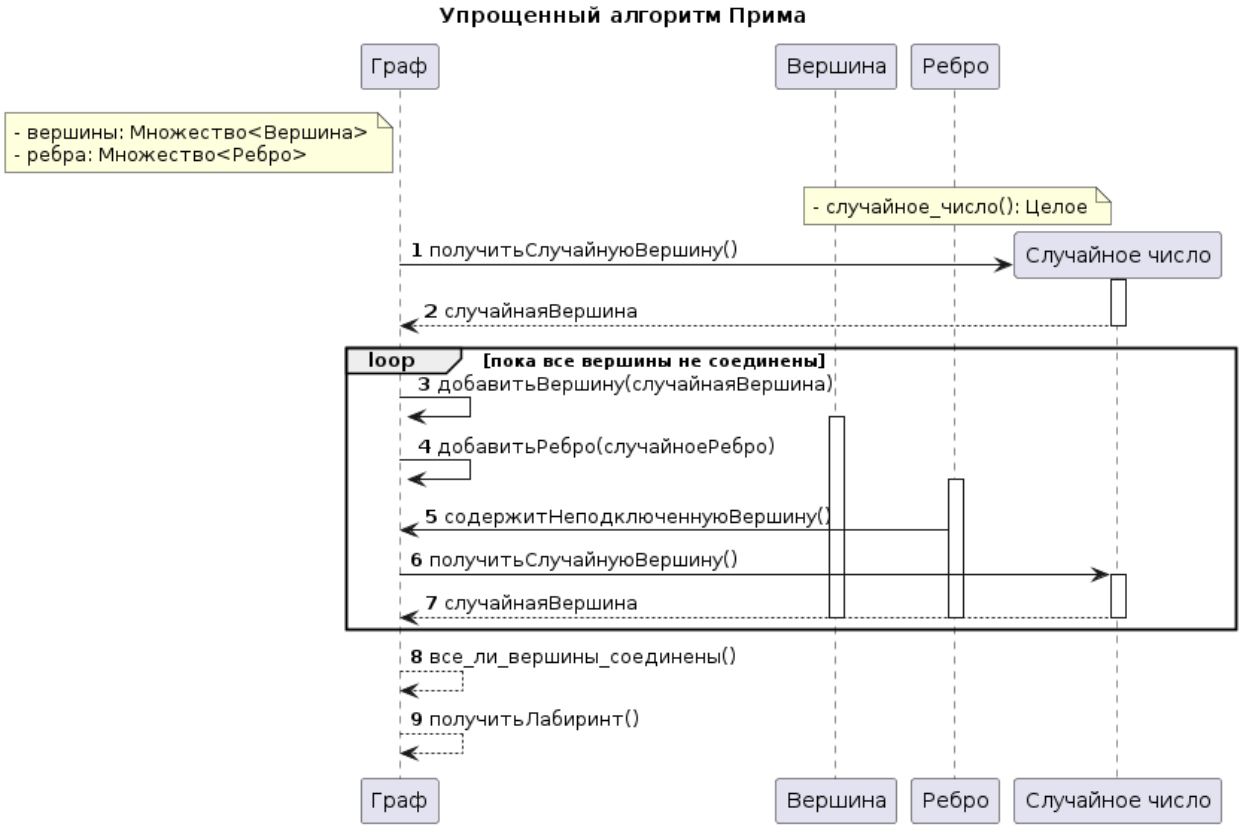

Рис. 8. UML диаграмма последовательности упрощенного алгоритма Прима

Плюсы упрощенного алгоритма Прима:

1. Простота реализации: алгоритм не требует сложных структур данных и может быть реализован с использованием простых списков.
2. Быстрая работа: выбор элементов из списка выполняется очень быстро, за постоянное время, что делает алгоритм намного быстрее, чем «настоящий» алгоритм Прима со случайными весами ребер.
3. Малый объем памяти: алгоритм требует объем памяти, пропорциональный размеру лабиринта, что делает его подходящим для решения задач на больших графах.

Минусы упрощенного алгоритма Прима:

1. Отсутствие гарантии оптимального решения: алгоритм не гарантирует, что найденное им дерево будет иметь минимальный вес, так как ребра выбираются случайным образом.

2. Неприменимость к взвешенным графам: алгоритм не может быть использован для решения задач на взвешенных графах, так как он не учитывает веса ребер при выборе.
3. Невозможность контроля над формой лабиринта: форма лабиринта полностью зависит от случайного выбора ребер, что делает невозможным контроль над его формой.

2.3.2. АЛГОРИТМ РЕКУРСИВНОГО ДЕЛЕНИЯ

Алгоритм рекурсивного деления работает путем последовательного разделения пространства на равные части, начиная с корневого узла, который представляет собой все пространство. Каждый узел в графе представляет собой ячейку пространства, и имеет два дочерних поля, которые получились после разделения первоначального поля по горизонтали или вертикали. Алгоритм также сам принимает решение делить ему по вертикали или горизонтали в зависимости от фигуры, полученной в результате деления: горизонтальные фигуры он старается поделить вертикально, а вертикальные горизонтально. Так как алгоритм является рекурсивным при каждой итерации алгоритм проверяет не является ли новое полученное поле «тривиальным» (которое уже не нужно делить на части). Если поле не является «тривиальным», то алгоритм случайным образом выбирает стену и проход в этой стене в начале или конце. Стены также не могут находиться прижаты друг к другу. Этими двумя условиями мы всегда гарантируем возможность прохождения этого лабиринта.

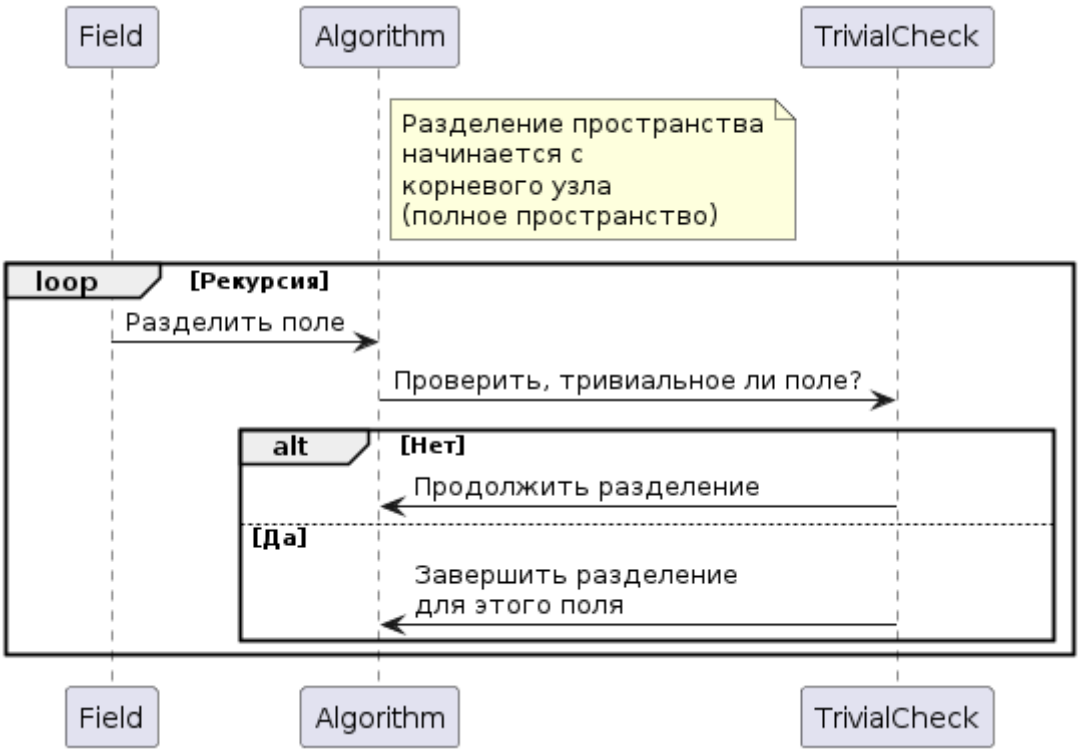

Рис. 9. UML диаграмма последовательности алгоритма рекурсивного деления

Плюсы алгоритма рекурсивного деления:

1. Простота реализации: алгоритм не сложен для понимания и реализации, даже для новичков в программировании.
2. Гарантированное решение: алгоритм всегда генерирует лабиринт с правильным решением, так как он гарантирует, что нет изолированных областей и что всегда есть путь от входа к выходу.
3. Масштабируемость: алгоритм легко масштабируется для генерации лабиринтов любого размера, просто изменив размер корневого узла.
4. Разнообразие: алгоритм может генерировать множество различных лабиринтов, даже при использовании одних и тех же параметров, благодаря использованию случайных выборов при разделении ячеек.

Минусы алгоритма рекурсивного деления:

1. Отсутствие контроля над сложностью: алгоритм не позволяет контролировать сложность лабиринта, так как она зависит от многих факторов, таких как размер лабиринта, количество разделений и случайные выборы.
2. Неэффективность для больших лабиринтов: алгоритм может быть неэффективным для генерации очень больших лабиринтов, так как он требует большого количества памяти и вычислительных ресурсов для хранения и обработки всех узлов.
3. Ограниченная гибкость: алгоритм имеет ограниченную гибкость в плане настройки параметров и модификации для получения лабиринтов с конкретными характеристиками.
4. Не всегда естественный вид: лабиринты, сгенерированные алгоритмом рекурсивного деления, могут выглядеть неестественно и симметрично, что может не подходить для некоторых применений, требующих более реалистичных лабиринтов.

2.3.3. АЛГОРИТМ ГЕНЕРАЦИИ ЛАБИРИНТА НА ОСНОВЕ ГПСЧ С ИСПОЛЬЗОВАНИЕМ РЕГИСТРА СДВИГА С ОБРАТНОЙ ЛИНЕЙНОЙ СВЯЗЬЮ

В настоящее время существует огромное количество генераторов псевдослучайных чисел, каждый из которых обладает свои алгоритмом и способом реализации. Из наиболее известных стоит упомянуть:

1. Линейные и нелинейные (квадратические, кубические) конгруэнтные генераторы;
2. Генераторы, основанные на регистрах сдвига с линейной или обобщенной обратной связью (LFSR)
3. Аддитивные генераторы с запаздываниями на основе последовательностей чисел Фибоначчи
4. Высокопроизводительный генератор «вихрь Мерсенна», использующий свойства простых чисел

Разделяя генераторы псевдослучайных последовательностей на классы, можно составить модель, приведённую на рис. 1.

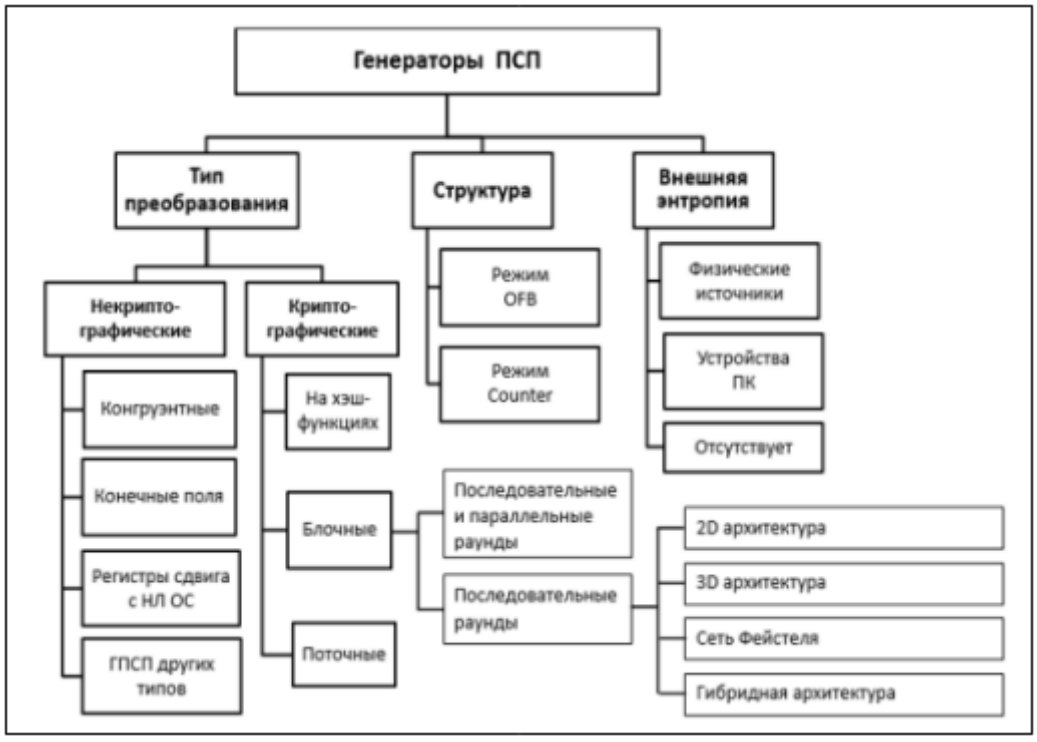

Рис. 8. Классы ГПСЧ

Генератор на основе регистра сдвига с обратной линейной связью (LFSR). – микросхема с ячейками памяти, в которые записан один бит информации. Ячейки имеют по одному входу и выходу. Количество ячеек равно длине регистра (рис. 14). Также выбираются «ссылки» (их ещё называют отводами) на ячейки памяти, которые будут подвергаться функции обратной связи, обычно это XOR всех отводов. Изначально мы имеем некоторые значения на выходе регистра. Булева функция, исходя из этих значений, подает на вход регистра некоторое число. Затем, как только подаётся тактовый сигнал, все значения сдвигаются на 1 вправо, в нулевой триггер попадает тот самый результат булевой функции. Теперь значения на выходе регистра совсем другие. Булева функция заново считает результат и подает его на вход. Далее следующий такт, и все повторяется.

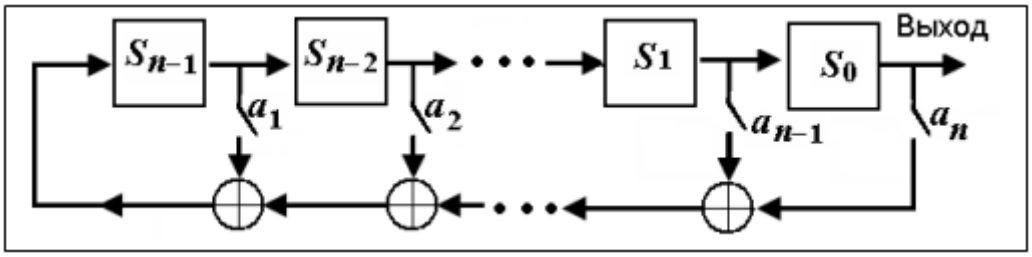

Рис. 9. Схема работы LFSR

После полученные значения используются как решающий фактор при выборе «поставить в данной клетке стену или нет». Таким образом, с помощью генератора псевдослучайных чисел на основе регистра сдвига с обратной линейной связью (LFSR) мы можем создавать лабиринты с заданной плотностью стен. При этом, как и в случае с другими генераторами, важно следить за тем, чтобы последовательность чисел была достаточно длинной и не повторялась. Кроме того, существуют и другие способы использования генераторов псевдослучайных чисел в создании лабиринтов. Например, можно использовать их для определения направления движения при создании лабиринта с помощью

алгоритма hunt and kill, или для определения расположения комнат и переходов между ними в лабиринте, построенном по принципу алгоритма Роберта К. Прима.

ПРАКТИЧЕСКАЧ ЧАСТЬ

3.1. ВЫБОР ИНСТРУМЕНТОВ И ТЕХНОЛОГИЙ ДЛЯ РАЗРАБОТКИ

При разработке новых продуктов, особенно в сфере веб-приложений, выбор инструментов и технологий играет решающую роль. Этот этап не просто формальность, а ключевой момент, определяющий успешность всего проекта. Ошибка в выборе инструментов может привести к серьезным проблемам на более поздних этапах разработки, включая увеличение времени и затрат на проект, технические сложности, а в итоге - недовольство пользователей.

Одной из наиболее острых проблем при выборе инструментов является несоответствие потребностям проекта. Нередко разработчики выбирают технологии, с которыми они уже знакомы или которые кажутся им наиболее "популярными", не учитывая специфику задачи. Это может привести к использованию неэффективных средств или даже к невозможности реализации необходимой функциональности.

Важно также учитывать факторы, связанные с масштабируемостью, безопасностью, поддержкой и сообществом разработчиков. Технологии, выбранные для проекта, должны быть гибкими и способными адаптироваться к изменяющимся требованиям рынка и потребностям пользователей.

Таким образом, правильный выбор инструментов и технологий — это не просто шаг, а стратегическое решение, определяющее будущее. Внимательный анализ требований проекта и обдуманный подход к выбору технологического стека позволяют избежать множества проблем в будущем и обеспечить устойчивость и эффективность разработки. Не зря вопросом выбора «стека» архитектуры занимаются высококвалифицированные специалисты с большим опытом работы со всевозможными инструментами, потому что именно от опыта архитектора

зависит насколько хорошо проект будет чувствовать себя на поздних этапах разработки.

Для разработки веб-приложения в ходе данной курсовой работе были выбраны следующие технолгии:

- React
- TypeScript
- Next.js
- Zustand + immer
- Jotai
- Material-UI (Icons и Material) + Emotion (React и Styled)
- clsx
- Tailwind CSS
- PostCSS
- Autoprefixer
- ESLint (с различными плагинами и конфигурациями)
- Prettier

3.1.1. ОБОСНОВАНИЕ ВЫБОРА «Next.js» КАК ОСНОВНОГО ФРЕЙМВОРКА

В контексте разработки веб-приложения по визуализации алгоритмов на графе выбор правильного фреймворка играет ключевую роль.

Вот как пишут разработчики Next.js о своем продукте: «Next.js - это фреймворк React для создания full-stack приложений. Используете компоненты React для построения пользовательских интерфейсов, а Next.js добавляет дополнительные функции и оптимизации.

Под капотом Next.js также абстрагирует и автоматически настраивает инструменты, необходимые для React, такие как сборка, компиляция и многое другое. Это позволяет вам сосредоточиться на создании вашего приложения, а не тратить время на настройку.»[5]

Next.js представляет собой мощный инструмент для разработки веб-приложений, который сочетает в себе преимущества как серверного, так и клиентского рендеринга. Его возможности по предварительному рендерингу страниц обеспечивают быструю загрузку и отзывчивость интерфейса, что критически важно для визуализации алгоритмов на графе, где каждое изменение должно мгновенно отражаться на экране.

Сравнивая существующие альтернативы, можно отметить, что React, который лежит в основе Next.js, обладает большим сообществом разработчиков и множеством доступных компонентов и библиотек, что облегчает процесс разработки и поддержки приложения. Исходя из особенностей проекта, в том числе его требований к производительности и в сравнении с другими фреймворками, такими как Angular или Vue.js, скорости разработки и расширяемости, Next.js является наиболее подходящим выбором. Его возможности по созданию быстрых и динамичных интерфейсов, в сочетании с широкой экосистемой React и простотой в настройке, делают его оптимальным решением для успешной реализации данного веб-приложения.

[5] Next.js documentation // Nextjs URL: https://nextjs.org/docs (дата обращения: 01.05.2024).

3.1.2. ОБОСНОВАНИЕ ВЫБОРА «TypeScript», «ESLint» и «Prettier»

Использование TypeScript, ESLint и Prettier при разработке веб-приложения для визуализации алгоритмов на графиках важно для обеспечения высокого качества кода, удобочитаемости и масштабируемости проекта.

TypeScript — это строгий «superset» над JavaScript, который обнаруживает и исправляет ошибки во время разработки, сокращает количество ошибок во время выполнения и улучшает поддержку кода. Его строгая типизация улучшает поддержку IDE, предотвращая «глупые» ошибки разработчика. В сегодняшних реалиях рынка ни одна команда не будет писать высоконагруженные приложения без «TypeScript» в своем списке технологий, поэтому данный выбор был просто вынужден.

ESLint — это инструмент анализа кода для JavaScript, позволяющий выявлять и исправлять потенциальные проблемы и стилистические ошибки. Он позволяет настраивать правила компоновки, обеспечивая согласованность и удобочитаемость стиля кодирования, а также, не давая разработчику случайно написать плохой код, исправление которого в будущем может занять большое количество времени.

Prettier — это инструмент автоматического форматирования кода, поддерживающий единый стиль кодирования в проекте. Это автоматически стандартизирует код, устраняя необходимость в ручном управлении форматированием, упрощая командную работу и сокращая время, затрачиваемое на обсуждение форматирования кода.

3.1.3. ОБОСНОВАНИЕ ВЫБОРА «Zustand» И «Jotai» ДЛЯ УПРАВЛЕНИЯ СОСТОЯНИЕМ

Выбор Zustand и Jotai для управления состоянием веб-приложения по визуализации алгоритмов поиска на графе обоснован стремлением к эффективному и удобному управлению состоянием приложения.

Zustand и Jotai представляют собой современные библиотеки для управления состоянием в React-приложениях, которые предлагают инновационные подходы к управлению состоянием компонентов. Использование этих библиотек обеспечивает простоту в использовании и понимании, что упрощает разработку и поддержку кода.

Zustand предлагает простую API для создания глобального состояния, используя принцип хранилища, что позволяет эффективно управлять состоянием приложения без лишней сложности. Также в данной курсовой работе была использована расширенная версия Zustand вкупе с immer, что позволило использовать концепцию «неизменяемости данных» с более простым и удобным синтаксисом. Данная возможность особенно хорошо себя проявила в работе с многоуровневой матрицей.

Jotai, в свою очередь, предлагает альтернативный подход к управлению состоянием, основанный на концепции атомов. Этот подход пропагандирует простоту и прозрачность в управлении состоянием, предоставляя чистый и декларативный API для работы с атомами состояния.

Подключения сразу двух библиотек управления состоянием было обусловлено разными концептуально разными подходами в вышеперечисленных библиотеках. Данный выбор основывается, что каждая библиотека выполняет свою задачу, определение выбора, какую же концепцию нужно выбрать «атомную» или «reducer», представлена на рисунке 10.

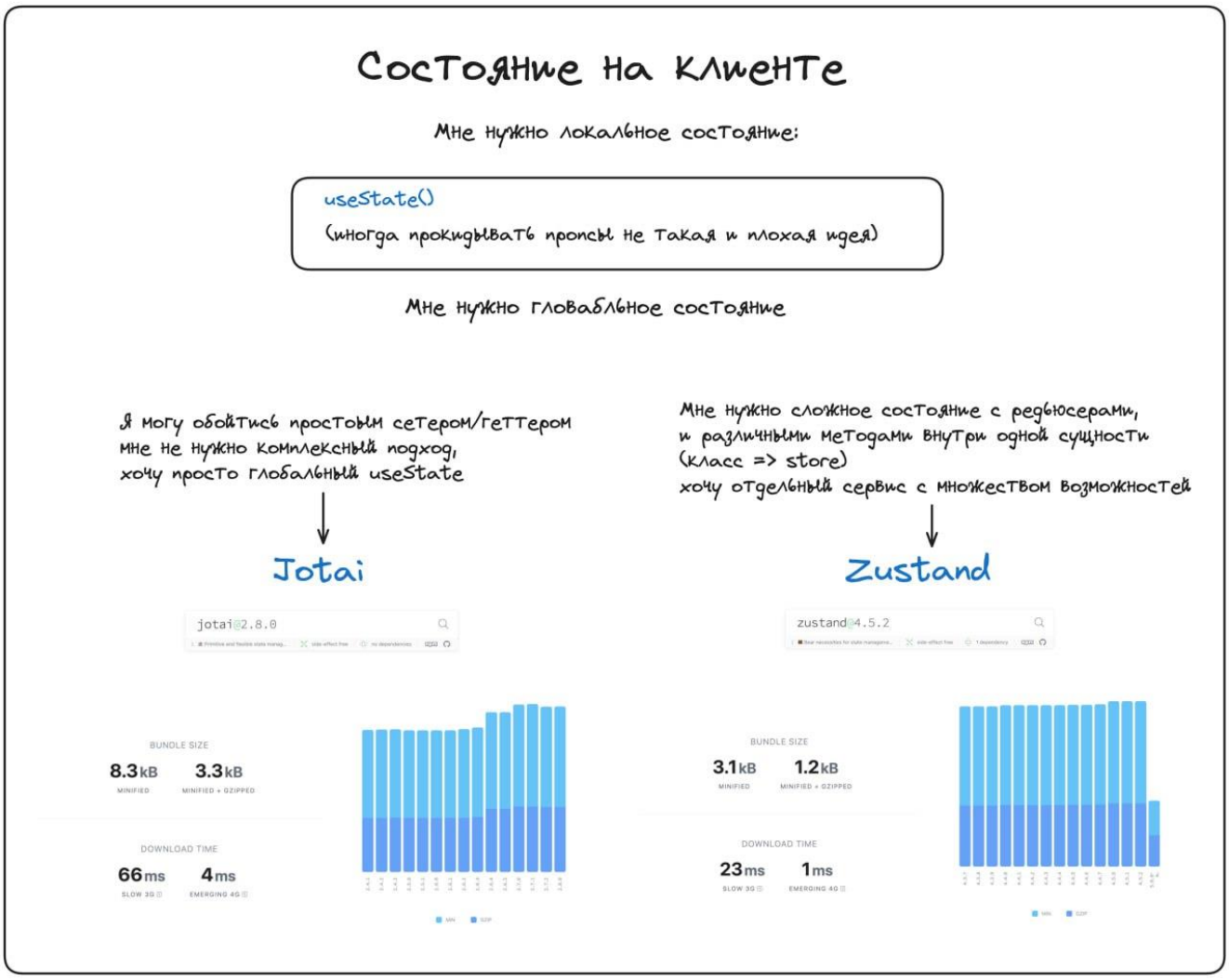

Рис. 10. Диаграмма выбора состояний в React

Выбор Zustand и Jotai обусловлен необходимостью обеспечения эффективного, простого и понятного управления состоянием приложения, что в свою очередь способствует ускорению разработки, повышению ее качества и улучшению поддерживаемой код-базы.

3.1.4. ОБОСНОВАНИЕ ВЫБОРА «Tailwind» и «Material-UI» ДЛЯ СТИЛИЗАЦИИ

При разработке веб-приложений, особенно тех, которые предполагают сложную визуализацию данных, выбор правильных инструментов для стилизации играет критическую роль. Одной из основных проблем здесь является необходимость соблюдения единого стиля и обеспечение высокой производительности при работе с компонентами интерфейса.

Tailwind CSS — это подход, основанный на использовании набора предварительно созданных классов CSS. Он позволяет быстро создавать красивые пользовательские интерфейсы, минимизируя необходимость вручную писать CSS. «Служебные классы помогают вам работать в рамках ограничений системы, и не загромождать таблицы стилей произвольными значениями. Они упрощают выбор цвета, интервалов, типографики, теней и всего остального, что составляет основу хорошо продуманной дизайн системы»[6]

Одним из главных преимуществ Tailwind CSS является возможность быстрого прототипирования и реализации дизайнерских решений без необходимости писать кастомный CSS.

Material-UI — это популярная библиотека React компонентов, основанная на дизайн-системе Material Design от Google. Она предоставляет готовые компоненты, реализующие принципы Material Design, что обеспечивает единый и современный внешний вид приложения. На официальном сайт данную библиотеку описывают следующим образом: «MUI предлагает полный набор бесплатных инструментов пользовательского интерфейса, которые помогут вам быстрее внедрять новые возможности в ваш продукт»[7]

Выбор Tailwind CSS и Material-UI обоснован их современными подходами к стилизации веб-приложений, обеспечивающими гибкость, производительность и единообразие в интерфейсе.

---

[6] Get started with Tailwind CSS // Tailwind CSS URL: https://tailwindcss.com/ (дата обращения: 01.05.2024).

[7] MUI URL: https://mui.com/ (дата обращения: 01.05.2024).

3.2. РЕАЛИЗАЦИЯ ВЕБ-ПРИЛОЖЕНИЯ

Перед началом разработки веб-приложения было необходимо подготовить окружение для работы. Проект был настроен с использованием последней версии Next.js 13, в котором уже при создании через команду «npx create-next-app@latest» можно подключить TypeScript, ESLint и Tailwind CSS.

После первоначальной инициализации проекта также нужно было кастомизировать его под данный проект, для ESLint были добавлены плагины: «eslint-config-airbnb» и «eslint-config-prettier». Также вручную были настроены правила для ESLint внутри файла конфигурации.

Следующим шагом было кастомизация Tailwind CSS и настройка поддержки синтаксиса Tailwind CSS для IDE VS Code. Были добавлены два основных шрифта в конфигурационный файл Tailwind: «Space Mono» и «Noto Sans», первый для текста названия, второй для основного текста.

После был проведен тестовый запуск приложения для тестирования всех первоначальных настроек проекта.

Выбранные библиотеки Zustand и Jotai для управления состоянием приложения устанавливаются и работают без каких-либо дополнительных настроек со стороны разработчика, что является ещё одним фактором в пользу выбора данного решения. Это позволило сразу приступить к их использованию, что значительно ускорило процесс разработки.

Впоследствии структура зависимостей изменилась: был добавлен Immer для оптимизации работы с неизменяемыми состояниями в Zustand, так как появилась необходимость часто изменять сложные структуры данных, а также библиотека clsx для удобного управления классами CSS для отрисовки разных состояний клеток в лабиринте.

Для создания пользовательских интерфейсов использовались Material-UI и Emotion. Material-UI обеспечил быстрый доступ к готовым компонентам, таким как кнопки и иконки, что сократило время на их разработку.

3.2.1. СОЗДАНИЕ БАЗОВОЙ СТРУКТУРЫ ПРИЛОЖЕНИЯ

Структура проекта следовала стандартному подходу для «фронтенд» приложений. Главная директория содержала поддиректории для компонентов, страниц, стилей, хранилища данных, типов, алгоритмов и утилит (рисунок 11). Такой подход обеспечил модульность и масштабируемость проекта, что важно для долгосрочного сопровождения и развития веб-приложения.

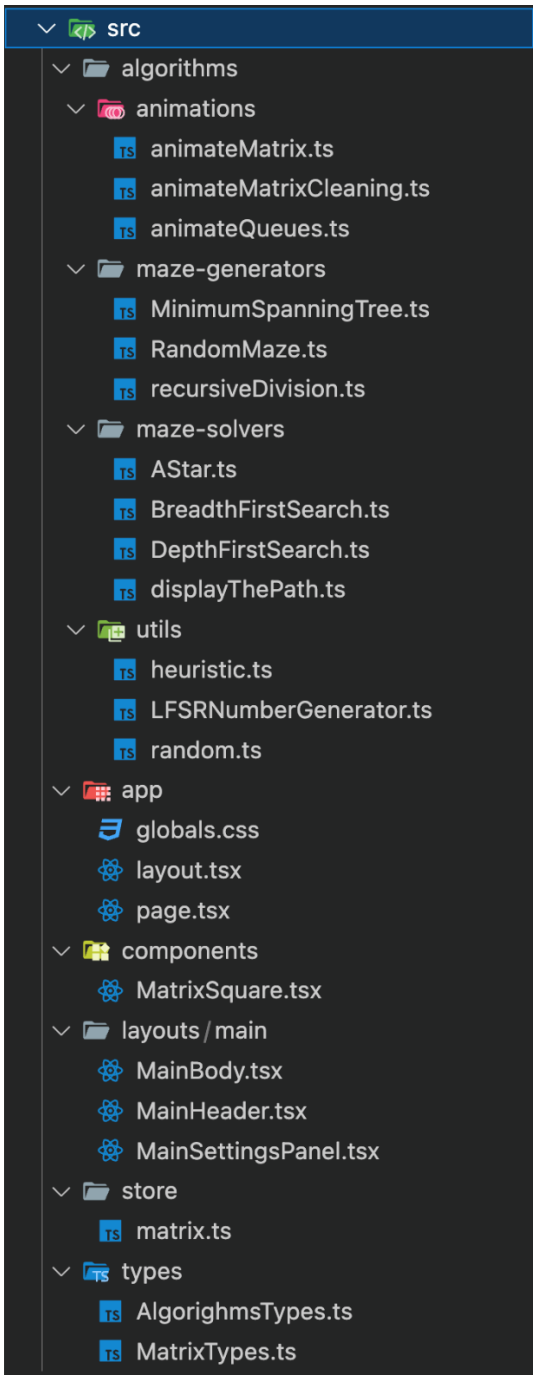

Рис. 11. Структура проекта

Папка «app» (рисунок 11) является системной и является отправной точкой для маршрутизации страниц в Next.js 13 версии.

Next.js использует маршрутизатор на основе файловой системы, где:

Папки используются для определения маршрутов. Маршрут — это единый путь к вложенным папкам, следующий по иерархии файловой системы от корневой папки до конечной папки, которая включает в себя файл page.js. Файлы используются для создания пользовательского интерфейса, который отображается для сегмента маршрута.[8]

Пример самой простой маршрутизации можно увидеть на рисунке 11 или пример с официальной документации Next.js 13 версии (рисунок 12).

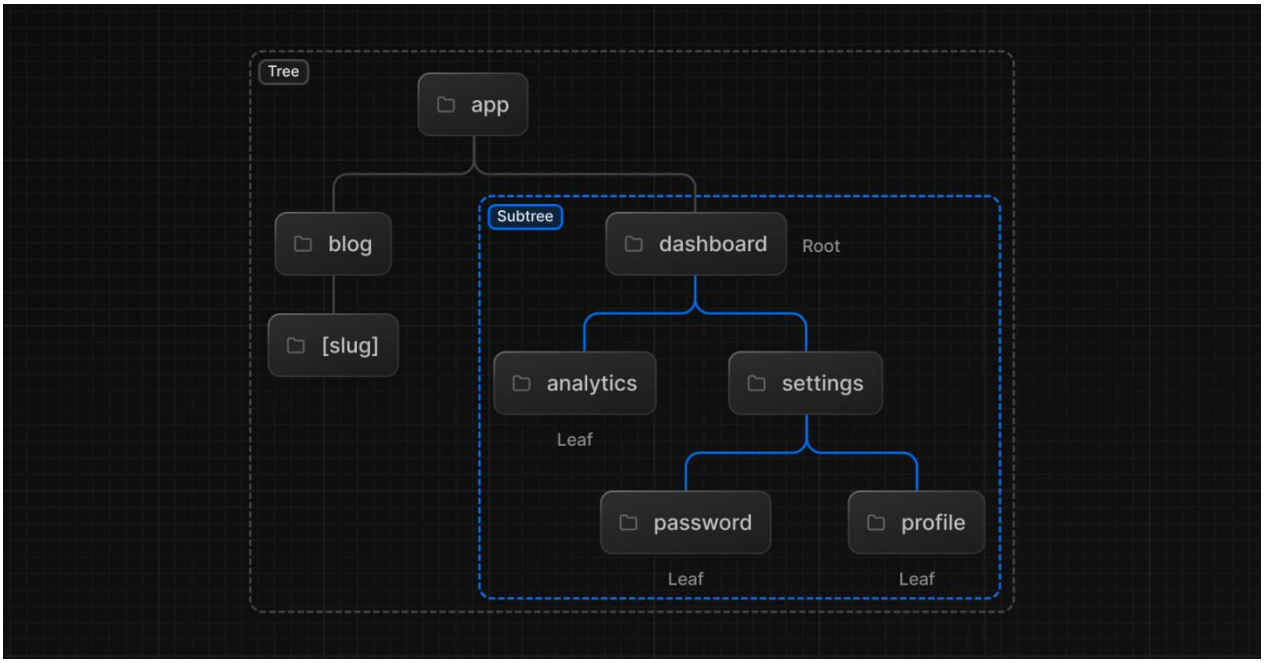

Рис. 12. Краткий справочник по маршрутизации в Next.js 13

На этом же начальном этапе была реализована структура хранилища данных, в которой, в последствии, хранилось централизованное состояние матрицы/лабиринта. Структура хранилища представляет себе классическую «машину управления состоянием» (рисунок 13) с полями и методами изменения

[8] Routing Fundamentals // Next.js URL: https://nextjs.org/docs/app/building-your-application/routing (дата обращения: 20.05.2025).

этих полей. Данный подход соответствует концепции сокрытия данных или инкапсуляции, что позволяет контролировать изменение данных на уровне абстракций, а не реализации. Полную же реализацию можно найти в приложении номер 1.

```
Codeium: Refactor | Explain
export interface MatrixState {
  matrix: Matrix;
  startCell: MatrixCell | null;
}

Codeium: Refactor | Explain
export interface MatrixAction {
  initializeMatrixShape: () => void;
  addStartAndFinishPoint: () => void;
  toggleWall: (xCord: number, yCord: number) => void;
  setCellInfo: (newCellInfo: QueueItem) => void;
  setMatrix: (newMatrix: Matrix) => void;
  clearMatrix: () => void;
}
```

Рис. 13. Интерфейс хранилища данных

3.2.2. РЕАЛИЗАЦИЯ АЛГОРИТМОВ ПОИСКА НА ГРАФЕ

В ходе работы были реализованы три алгоритма поиска на графе. Но сначала стоит объяснить специфику задачи и то, каким вообще образом был реализован «лабиринт». В данном приложении «лабиринт» представляет из себя матрицу объектов (интерфейс объекта вершины графа представлен на рисунке 14)

```
export type CellState = "default" | "wall" | "start" | "destination" | "visited" | "path";

Codeium: Refactor | Explain
export interface MatrixCell {
  id: string;
  row: number;
  column: number;
  state: CellState;
  pathLink: { row: number; column: number } | null;
}

export type Matrix = MatrixCell[][];
```

Рис. 14. Интерфейс главной матрицы

Каждая такая вершина хранит в себе всю необходимую информацию под каким номер и где она находится, для работы с алгоритмами поиска на графе. Также каждая ячейка имеет поле «pathLink», которое служит для хранения ссылки на ячейку, из которой алгоритм пришел в данную клетку, такая система односвязного списка позволяет в конце работы алгоритма с легкостью восстановить самый короткий путь из финиша к началу «лабиринта». Каждая ячейка лабиринта также имеет свое состояние, которые отображается в последствии в интерфейсе различным цветом для наглядности работы алгоритмов, и также является ключевым для работы алгоритмов.

Первым был реализован алгоритм BFS (Breadth-First Search), или Поиск в ширину. Его идея заключается в последовательном исследовании всех вершин графа на заданном уровне до перехода к вершинам следующего уровня. BFS полезен для нахождения кратчайшего пути в графах без взвешенных рёбер. Также алгоритм BFS гарантирует самый оптимальный маршрут, но при этом

проигрывает по времени и памяти всем остальным алгоритмам. Для хранения вершин, которые нужно исследовать, используется очередь (FIFO). Все вершины, достижимые из стартовой, будут посещены ровно один раз. Временная сложность: O (V + E); Пространственная сложность: O(V) где V — количество вершин, E — количество рёбер
Полная реализация находится в приложении №2.
Данный алгоритм достаточно простой и в конкретной реализации в приложении не было ничего добавлено или изменено, о чем стоило бы упомянуть отдельно.

Следующим алгоритмом был выбран DFS (Depth-First Search) Поиск в глубину. В отличие от BFS, который исследует все вершины на текущем уровне перед переходом к следующему, DFS углубляется в граф, исследуя как можно дальше по одному пути, прежде чем откатываться назад. В данном случае был реализован не рекурсивный DFS, использовался стек (LIFO). В отличие от BFS, DFS не гарантирует нахождение кратчайшего пути в графе, если граф невзвешенный. Временная сложность: O (V + E); Пространственная сложность: O(V) где V — количество вершин, E — количество рёбер.
Полная реализация находится в приложении №3.

Самым сложным реализованным алгоритмом поиска пути в данном приложении является A* (A star или A звезда). Алгоритм A* (A-star) является эвристическим алгоритмом поиска пути, который используется для нахождения кратчайшего пути в графах. A* комбинирует достоинства алгоритма поиска в ширину (BFS) и жадного поиска по наилучшему совпадению, используя эвристику для выбора наиболее перспективных путей. A* использует как информацию о стоимости пути от стартовой вершины (g), так и эвристику (h), чтобы направлять поиск к цели. Алгоритм использует приоритетную очередь для выбора следующей вершины с наименьшей прогнозируемой стоимостью пути. Если эвристическая функция h является допустимой (не переоценивает стоимость), алгоритм гарантирует нахождение кратчайшего пути. Временная сложность в худшем случае O(E), где E

- количество рёбер, однако на практике эффективность зависит от точности эвристики. Пространственная сложность: O(V), где V — количество вершин. Полная реализация находится в приложении №4.

3.2.3. РЕАЛИЗАЦИЯ АЛГОРИТМОВ ГЕНЕРАЦИИ ЛАБИРИНТОВ

Упрощённый алгоритм Прима используется для создания лабиринтов и основан на модификации алгоритма Прима для минимального остовного дерева. Этот алгоритм генерирует лабиринт, стартуя с одной клетки и постепенно добавляя новые клетки к лабиринту, пока все клетки не будут включены. В данной реализации был использован «обратный» подход. Сначала вся матрица заполнялась стенами, и уже алгоритм «бурил» себе путь, который в последствии и становился проходами в лабиринте. Ход действий алгоритма представлял из себя два основных пункта:

Инициализация:

- Все клетки кроме начала и конца заполняются стенами
- Выбирается стартовая клетка и создаётся список стен, смежных с этой клеткой.

Основной цикл:

- Пока список клеток на границе не пуст:
    - Случайным образом выбирается стена из списка.
    - Если эта стена соединяет клетку в лабиринте с клеткой, которая ещё не в лабиринте:
        - Клетка добавляется к лабиринту становится проходом
        - Также добавляется клетка между начальной (из которой она добавилась в список открытых стен) и также становится проходом
        - Добавляются в список все стены, смежные с новым проходом, которые ещё не в лабиринте.
    - Удаляем текущую стену из списка.

Конец алгоритма наступает, когда у нас заканчивает список стен. Стены выбираются случайным образом, что приводит к разнообразию и случайности в структуре лабиринта. Временная сложность: O(V), где V — количество клеток. Пространственная сложность: O(E), где E — количество стен.

Преимущества:

- Простота реализации.
- Гарантированное создание связного лабиринта без изолированных частей. Равномерное распределение пути по всему лабиринту.

Недостатки:

- Лабиринт может быть слишком случайным и не всегда иметь интересную структуру.
- Менее контролируемая структура по сравнению с другими алгоритмами создания лабиринтов, такими как алгоритм рекурсивного деления.

Полную реализация алгоритма находится в приложении №5

Алгоритм рекурсивного деления работает путём рекурсивного деления области лабиринта на меньшие части, создавая стены и проходы, чтобы гарантировать, что лабиринт будет иметь связную и сложную структуру. В каждой рекурсивной итерации выбирается горизонтальное или вертикальное деление, и создаются стены и проходы, чтобы гарантировать доступность всех частей лабиринта.
Инициализация:

- Начинается с полного пространства лабиринта (прямоугольная область).
- Определяются границы текущей области, которую необходимо разделить.

Основной процесс:

- Если текущая область слишком мала, рекурсия возвращается (базовый случай рекурсии).
- Решается, делить ли область горизонтально или вертикально.
    - Выбирается случайная координата для деления (не на самой границе).
    - Строится стена по выбранной координате.
    - В стене создаётся один проход (на границе).
- Рекурсивно применяется процесс к двум новым областям, созданным делением.

Преимущества:

- Простота реализации.

- Гарантирует связность всех частей лабиринта.

Недостатки:

- Рекурсивный характер может потребовать большого количества памяти для больших лабиринтов.
- Структура лабиринта может быть предсказуемой из-за строгого деления на части.

Полная реализация алгоритма находится в приложении №6

Последний алгоритм основан на простой генерации с помощью ГПСЧ с использованием регистра сдвига с обратной линейной связью. В рамках создания веб-приложения был написан собственный генератор с уникальной функцией обратной связи на основе индексов при создании последовательности псевдослучайных чисел и числа PI. (рисунок 15)

```
const LFSRNumberGenerator = (
  buffer: number[],
  links: number[],
  numberToGenerate: number
) => {
  const innerBuffer = buffer.slice();
  const innerLinks = links.slice();
  const result = [];
  for (let i = 1; i < numberToGenerate; i++) {
    let index;
    let newNum = 0;
    for (let j = 0; j < innerLinks.length; j++) {
      index = innerLinks[j];
      newNum += innerBuffer[index] + (i + j) * +Math.PI.toFixed(0);
    }

    innerBuffer.pop();
    innerBuffer.unshift(newNum % 2);
    const bufferStr = innerBuffer.reduce((a, b) => {
      return a + b;
    }, "");

    result.push(parseInt(bufferStr, 2));
  }
  return result;
};
export default LFSRNumberGenerator;
```

Рис. 15. Реализация ГПСЧ с использованием регистра сдвига с обратной линейной связью

Сам же алгоритм проходит через каждую клетку и с вероятностью в 25% ставит состояние этой клетки «стена», тем самым лабиринт заполняется совершенно случайно.
Преимущества:

- Простота реализации.
- Нестандартный вид лабиринта для проверки алгоритмов поиска

Недостатки:

- Нет гарантии, что путь, при решении путь такого лабиринта, будет найден

Полная реализация алгоритма находится в приложении №7

3.2.4. ВИЗУАЛИЗАЦИЯ РАБОТЫ АЛГОРИТМОВ

Главным же отличием от обычной реализации алгоритмов, связанных с графами и поиском путей на графе в данном продукте является визуализация этих алгоритмов с приятной анимацией. Задача анимации таких сложных структур потребовала создания дополнительных структур и функций для полной реализации желаемого результата. В ходе разработки были созданы такие функции как «animateMatrix» «animateMatrixCleaning» «animateQueues». Главной же функцией для отрисовки является animateQueues реализация которой представлена на рисунке 16. Очередью для анимации в данном случае является встроенный в движок выполнения кода JavaScript — event loop. В данную очередь подаются макротаски которые создаются на уровне браузера, и реализация этих функций хранится относится к интерфейсу Window.

```
const animateQueues = (
  queues: QueueItem[][] | undefined,
  setStateCallback: (newCellInfo: QueueItem) => void,
  speed: number = 50
) => {
  let index = 0;

  if (!queues) {
    return;
  }

  for (let queue of queues) {
    while (queue.length > 0) {
      const newCellInfo = queue.shift();
      index++;
      if (newCellInfo) {
        setTimeout(() => {
          setStateCallback(newCellInfo);
        }, index * speed);
      }
    }
  }
};

export default animateQueues;
```

Рис. 16. Реализация функции animateQueues

На вход данной функции подаются очереди из действий, которые нужно выполнить поочерёдно с задержкой, скорость которой тоже можно контролировать с помощью параметра «speed». Данная реализация позволяет строить сложные цепочки последовательных действий, нужно всего лишь для каждого алгоритма возвращать массив со всей историей действий. Почти во всех реализованных алгоритмах этот массив называется historyQueue. (простой пример находится в приложении №2). Таким образом сами алгоритмы изменяют только внутренний «временный» лабиринт, созданный с помощью замыкания, а изменения в основном лабиринте, состояние которого связано с отображаемым для пользователя интерфейсом, происходят только в ходе выполнения функций изменения состояния из очереди макротасок. Для примера можно рассмотреть реализацию отрисовки алгоритма BFS, который запускается через функцию «paintBFS» (рисунок 17).

```
const paintBFS = () => {
  const { tempMatrix: clearedMatrix } = animateMatrixCleaning(matrix, setCellInfo);
  const { historyQueue: bfsQueue, matrix: bfsMatrix } = bfs(
    clearedMatrix,
    clearedMatrix[3][3]
  );
  const { historyQueue: pathQueue } = displayThePath(bfsMatrix);

  animateQueues([bfsQueue, pathQueue], setCellInfo, 10);
};
```

Рис. 17. Реализация отрисовки анимации алгоритма BFS

В ходе данной функции сначала происходит анимация очистки матрицы, после создается очередь работы алгоритма BFS и решенная матрица на выходе, которая потом подается в функцию displayThePath, которая в свою очередь создаёт ещё одну очередь для изменения состояния клеток матрицы на основе матрицы полученной из функции «bfs», и в конечном счете обе этих очереди попадают в функцию отрисовки изменений и пользователь начинает видеть анимацию.

Для примера, на рисунках 18, 19 и 20 представлена конечная отрисовка различных алгоритмах поиска пути для лабиринта, построенного с помощью алгоритма Прима.

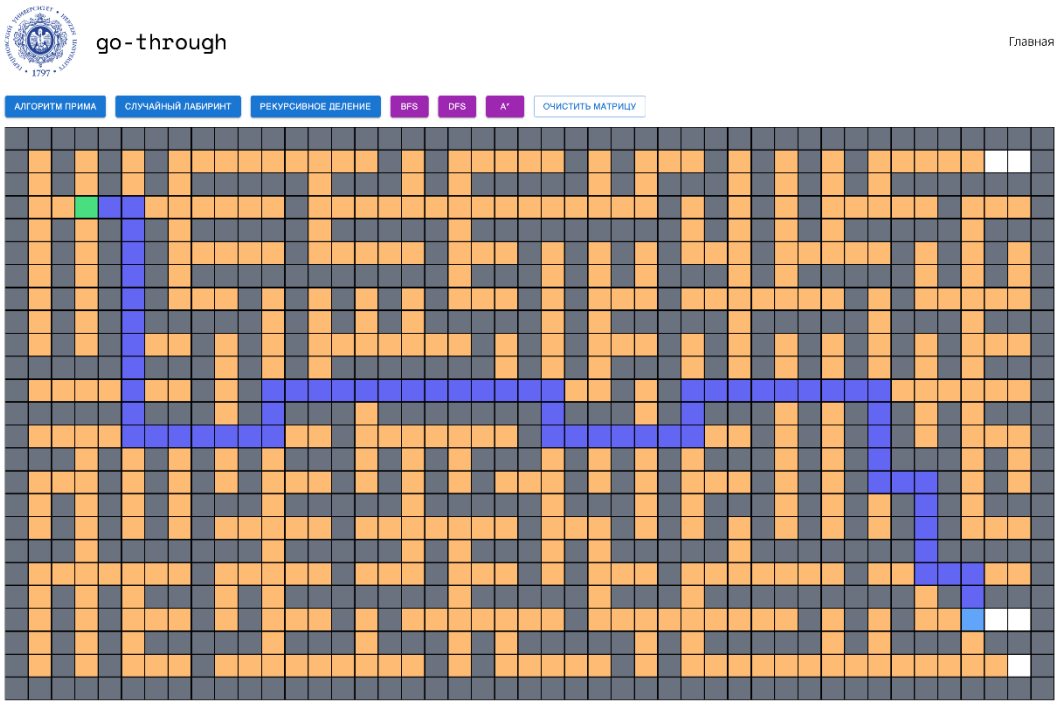

Рис. 18. Визуализация «Алгоритм BFS»

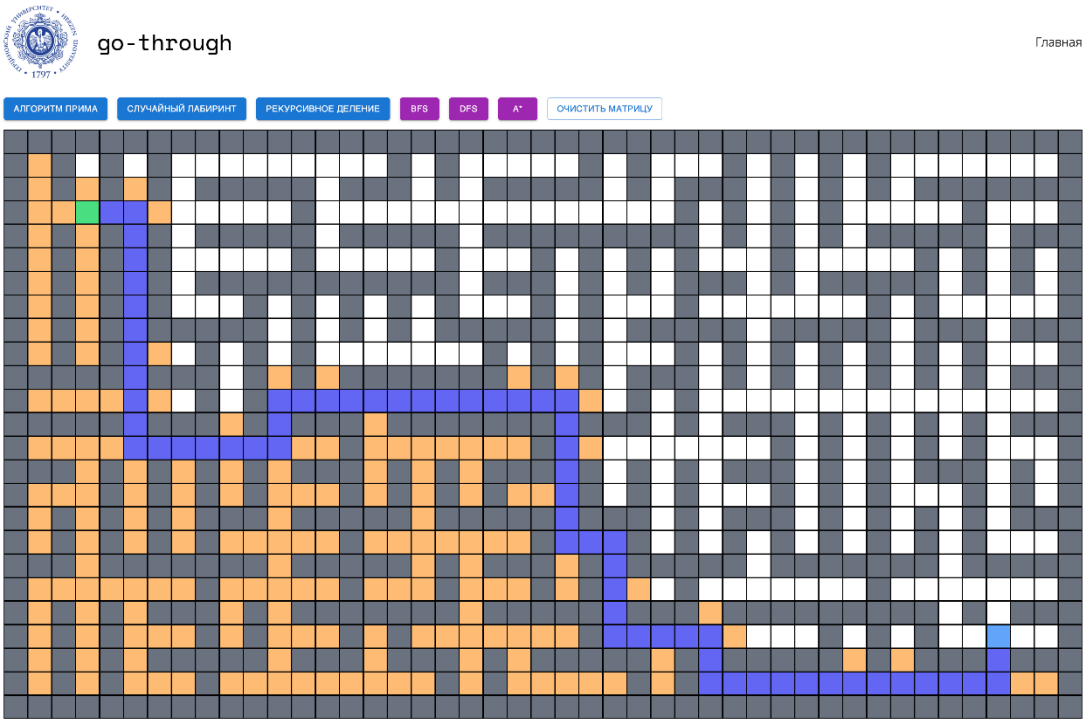

Рис. 19. Визуализация «Алгоритм DFS»

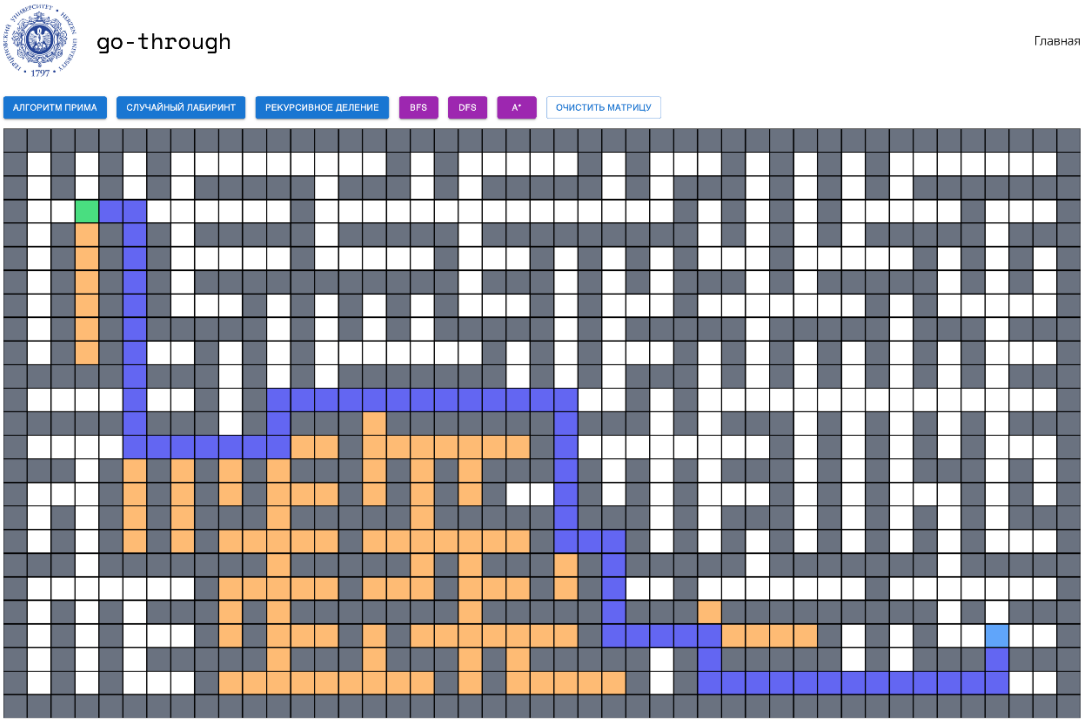

Рис. 20. Визуализация «Алгоритм A*»

3.2.5. ДОБАВЛЕНИЕ ИНТЕРАКТИВНОСТИ ДЛЯ ПОЛЬЗОВАТЕЛЕЙ

Для удобного взаимодействия был разработан простой и понятный интерфейс состоящих из панели кнопок над лабиринтом. Каждая кнопка называется именем своего алгоритма или функции, которую она вызывает.

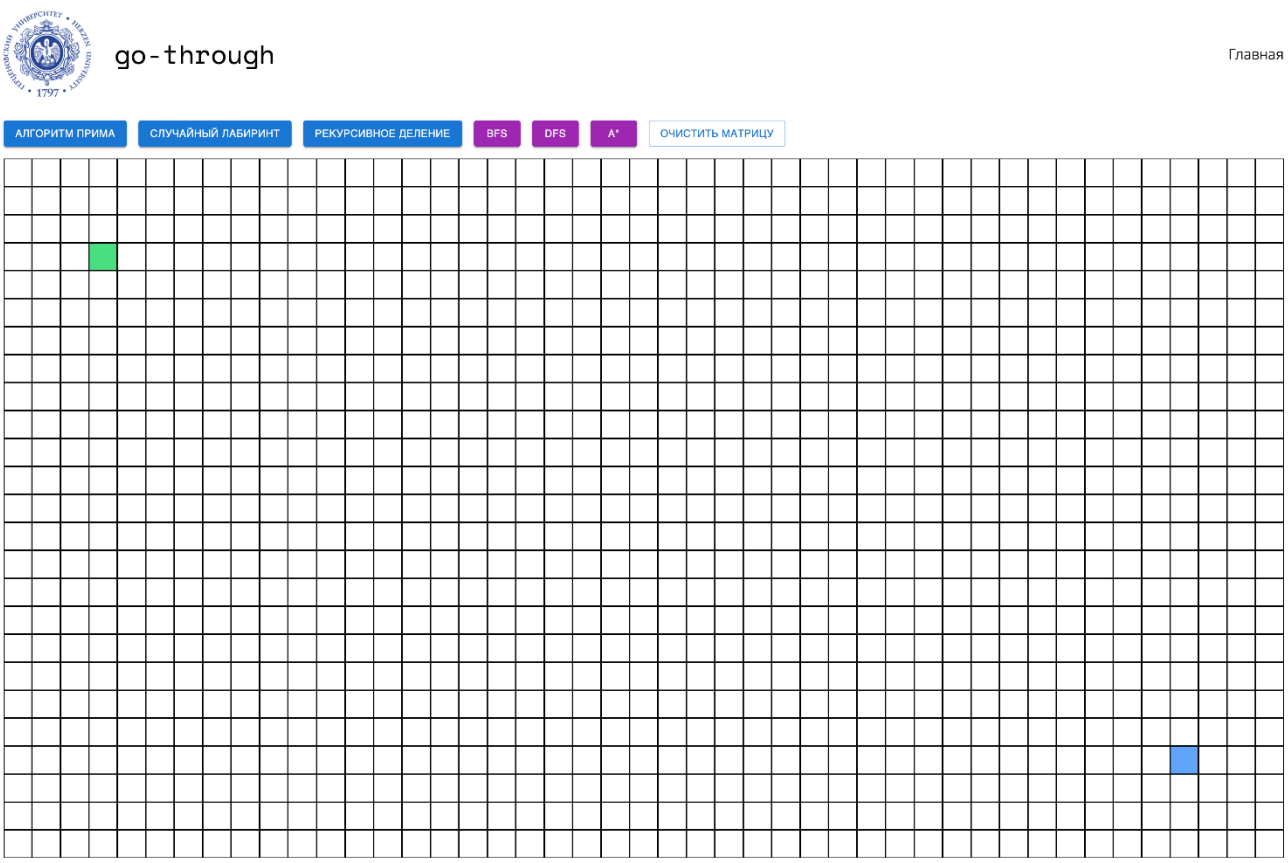

Рис. 18. Начальная страница приложения

ЗАКЛЮЧЕНИЕ

В ходе работы были изучены существующие методы, разработан дизайн интерфейса, реализованы и протестированы выбранные алгоритмы. В результате теоретической части были изучены основные алгоритмы поиска на графе, включая алгоритмы A*, BFS и DFS. Каждый алгоритм был проанализирован с точки зрения его эффективности, сложности и области применения. В результате практической части была создана интерактивное веб-приложение, позволяющие пользователям визуализировать работу различных алгоритмов, что дает не только понять теоретическую основу алгоритмов, но и увидеть их практическое применение. Все выбранные для проекта алгоритмы были успешно реализованы и визуализированы, также был разработан удобный и интуитивно понятный интерфейс, который обеспечивает простоту взаимодействия с веб-приложением. В отличие от существующих продуктов в российском интернете, данное веб-приложение предлагает не только теоретическое объяснение алгоритмов, но и их визуализацию в интерактивной форме. Это делает изучение алгоритмов более доступным и понятным для широкого круга пользователей, включая студентов. Разработанное веб-приложение имеет высокую практическую ценность. Оно может использоваться в образовательных учреждениях для повышения уровня подготовки студентов в области информатики. Кроме того, оно будет полезно всем, кто интересуется алгоритмами и математикой, предлагая уникальную возможность исследовать алгоритмы на практике. В дальнейшем возможно расширение функционала приложения, включая добавление новых алгоритмов и улучшение существующих визуализаций и исправлении несущественных багов. Итоги данной работы подтверждают достижение поставленных во введении целей и задач. Разработанное веб-приложение способствует улучшению понимания и изучения алгоритмов, что подтверждает его практическую ценность и востребованность.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Алексеев, В. Е. Теория графов : учебное пособие / В. Е. Алексеев, Д. В. Захарова. — Нижний Новгород : ННГУ им. Н. И. Лобачевского, 2017. — 119 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/153421 (дата обращения: 10.05.2024).
2. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2011. — 1296 с. : ил. — Парал. тит. англ.
3. Джеймис Бак Mazes for Programmers: Code Your Own Twisty Little Passages. - 1 изд. Pragmatic Bookshelf, 2015. - 288 с.
4. Дольников, В. Л. Основные алгоритмы на графах : текст лекций / В. Л. Дольников, О. П. Якимова; Яросл. гос. ун-т им. П. Г. Демидова. – Ярославль : ЯрГУ, 2011. – 80 с. (дата обращения: 01.05.2024).
5. Карпов, Д. В. Теория графов : учебное пособие / Д. В. Карпов. — Москва : МЦНМО, 2022. — 555 с. — ISBN 978-5-4439-3690-1. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/305501 (дата обращения: 10.05.2024). — Режим доступа: для авториз. пользователей.
6. Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. — М. : Издательский дом “Вильямс”, 2011. — 1296 с. : ил. — Парал. тит. англ. (дата обращения: 01.05.2024).
7. Построение и анализ алгоритмов обработки данных: учеб.-метод. пособие / И. А. Селиванова, В. А. Блинов. — Екатеринбург : Изд-во Урал. ун-та, 2015. — 108 с. (дата обращения: 25.03.2024).
8. Сорочан, С. В. Основы теории графов : учебно-методическое пособие / С. В. Сорочан. — Нижний Новгород : ННГУ им. Н. И. Лобачевского, 2023. — 59 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/344828 (дата обращения: 10.05.2024). — Режим доступа: для авториз. пользователей.

9. Documentation // Eslint URL: https://eslint.org/ (дата обращения: 03.05.2024).
10. Get started with Tailwind CSS // Tailwind CSS URL: https://tailwindcss.com/ (дата обращения: 01.05.2024).
11. Introduction // Zustand URL: https://docs.pmnd.rs/zustand/getting-started/introduction (дата обращения: 01.05.2024).
12. Jotai URL: https://jotai.org/ (дата обращения: 03.05.2024).
13. MUI URL: https://mui.com/ (дата обращения: 19.05.2024).
14. Next.js documentation // Nextjs URL: https://nextjs.org/ (дата обращения: 01.05.2024).
15. React URL: https://react.dev/ (дата обращения: 14.04.2024).
16. TypeScript URL: https://www.typescriptlang.org/ (дата обращения: 03.05.2024).

ПРИЛОЖЕНИЯ

Приложение №1. Структура хранилища данных.

```
export const useMatrixStore = create<MatrixState & MatrixAction>()(
  immer((set) => ({
    // states
    matrix: [],
    startCell: null,

    // actions
    Codeium: Refactor | Explain | ×
    initializeMatrixShape: () =>
      set(() => ({
        matrix: Array(25)
          .fill(0)
          .map((_row, rowIndex) =>
            Array(45)
              .fill(0)
              .map(
                (_cell, cellIndex) =>
                  ({
                    id: `${rowIndex}${cellIndex}`,
                    state: "default",
                    pathLink: null,
                    row: rowIndex,
                    column: cellIndex,
                  }) satisfies MatrixCell
              )
          ) satisfies Matrix,
      })),

    Codeium: Refactor | Explain | Generate JSDoc | ×
    addStartAndFinishPoint: () =>
      set((state) => {
        const storeMatrix = state.matrix;
        storeMatrix[3][3].state = "start";
        storeMatrix[3][3].pathLink = { column: 3, row: 3 };
        storeMatrix[storeMatrix.length - 4][
          storeMatrix[storeMatrix.length - 4].length - 4
        ].state = "destination";
        storeMatrix[storeMatrix.length - 4][
        state.startCell = storeMatrix[3][3];
      }),

    Codeium: Refactor | Explain | Generate JSDoc | ×
    toggleWall: (xCord, yCord) =>
      set((state) => {
        const cell = state.matrix[xCord][yCord];
        if (cell.state === "destination" || cell.state === "start") {
          return;
        }
        if (cell.state === "wall") {
          cell.state = "default";
          return;
        }
        cell.state = "wall";
      }),

    Codeium: Refactor | Explain | Generate JSDoc | ×
    setCellInfo: ({ xCord, yCord, newCell }) =>
      set((state) => {
        state.matrix[xCord][yCord] = newCell;
      }),

    Codeium: Refactor | Explain | Generate JSDoc | ×
    setMatrix: (matrix: Matrix) =>
      set((state) => {
        state.matrix = matrix;
      }),

    Codeium: Refactor | Explain | Generate JSDoc | ×
    clearMatrix: () =>
      set((state) => {
        state.matrix = state.matrix.map((row) =>
          row.map((cell) => {
            if (cell.state === "start" || cell.state === "destination") {
              return { ...cell, pathLink: null };
            }
            return { ...cell, pathLink: null, state: "default" };
          })
        );
      }),
  }))
```

Приложение № 2. Реализация алгоритма BFS

```
const directions = [
  { row: -1, column: 0 },
  { row: 0, column: 1 },
  { row: 1, column: 0 },
  { row: 0, column: -1 },
];

Codeium: Refactor | Explain | Generate JSDoc | ×
export const bfs = (matrix: Matrix, startCell: MatrixCell) => {
  const tempStartCell = { ...startCell };
  const tempMatrix = JSON.parse(JSON.stringify(matrix)) as Matrix;
  const queue: MatrixCell[] = [];
  const historyQueue: QueueItem[] = [];
  let destinationIsFound = false;

  queue.push(tempStartCell);
  historyQueue.push({
    xCord: tempStartCell.row,
    yCord: tempStartCell.column,
    newCell: tempStartCell,
  });

  while (queue.length > 0 && !destinationIsFound) {
    const currentCell = queue.shift();

    if (!currentCell) {
      break;
    }

    if (currentCell.state === "destination") {
      destinationIsFound = true;
      break;
    }

    for (const direction of directions) {
      const nextRow = currentCell.row + direction.row;
      const nextColumn = currentCell.column + direction.column;

      if (
        nextRow >= 0 &&
        nextRow < tempMatrix.length &&
        nextColumn >= 0 &&
        nextColumn < tempMatrix[nextRow].length &&
        tempMatrix[nextRow][nextColumn].state !== "wall" &&
        tempMatrix[nextRow][nextColumn].state !== "visited" &&
        tempMatrix[nextRow][nextColumn].state !== "start"
      ) {
        if (tempMatrix[nextRow][nextColumn].state === "destination") {
          destinationIsFound = true;
          tempMatrix[nextRow][nextColumn].pathLink = {
            row: currentCell.row,
            column: currentCell.column,
          };
          break;
        }

        tempMatrix[nextRow][nextColumn].state = "visited";
        tempMatrix[nextRow][nextColumn].pathLink = {
          row: currentCell.row,
          column: currentCell.column,
        };

        historyQueue.push({
          xCord: nextRow,
          yCord: nextColumn,
          newCell: tempMatrix[nextRow][nextColumn],
        });

        queue.push(tempMatrix[nextRow][nextColumn]);
      }
    }
  }

  return { historyQueue, matrix: tempMatrix, destinationIsFound };
};
```

Приложение № 3. Реализация алгоритма DFS

```
const directions = [
  { row: -1, column: 0 },
  { row: 0, column: 1 },
  { row: 1, column: 0 },
  { row: 0, column: -1 },
];

Codeium: Refactor | Explain | Generate JSDoc | ×
export function depthFirstSearch(matrix: Matrix, startCell: MatrixCell) {
  const stack: MatrixCell[] = [];
  const tempMatrix = JSON.parse(JSON.stringify(matrix)) as Matrix;
  const tempStartCell = { ...startCell };
  const historyQueue: QueueItem[] = [];
  let destinationIsFound = false;

  stack.push(tempStartCell);

  while (stack.length > 0 && !destinationIsFound) {
    const currentCell = stack.pop();

    if (!currentCell) {
      break;
    }

    if (currentCell.state === "destination") {
      destinationIsFound = true;
      break;
    }

    const neighbors: MatrixCell[] = [];

    for (const direction of directions) {
      const nextRow = currentCell.row + direction.row;
      const nextColumn = currentCell.column + direction.column;

      if (
        nextRow >= 0 &&
        nextRow < tempMatrix.length &&
        nextColumn >= 0 &&
        nextColumn < tempMatrix[0].length &&
        tempMatrix[nextRow][nextColumn].state !== "wall" &&
        tempMatrix[nextRow][nextColumn].state !== "visited" &&
        tempMatrix[nextRow][nextColumn].state !== "start"
      ) {
        if (tempMatrix[nextRow][nextColumn].state === "destination") {
          destinationIsFound = true;
          tempMatrix[nextRow][nextColumn].pathLink = {
            row: currentCell.row,
            column: currentCell.column,
          };
          break;
        }

        neighbors.push(tempMatrix[nextRow][nextColumn]);
      }
    }

    for (const neighbor of neighbors) {
      if (neighbor.state === "default") {
        neighbor.state = "visited";
        neighbor.pathLink = { row: currentCell.row, column: currentCell.column };
        historyQueue.push({
          xCord: neighbor.row,
          yCord: neighbor.column,
          newCell: neighbor,
        });
        stack.push(neighbor);
      }
    }
  }

  return { historyQueue, matrix: tempMatrix, destinationIsFound };
}

export default depthFirstSearch;
```

Приложение № 4. Реализация алгоритма A*

```
const AStar = (matrix: Matrix, startCell: MatrixCell, endCell: MatrixCell) => {
  const historyQueue: QueueItem[] = [];

  const tempMatrix = JSON.parse(JSON.stringify(matrix)) as Matrix;

  let openSet: MatrixCell[] = [startCell];
  let closedSet: MatrixCell[] = [];

  while (openSet.length > 0) {
    let currentCell = openSet[0];
    for (let i = 1; i < openSet.length; i++) {
      if (
        heuristic(openSet[i], endCell) < heuristic(currentCell, endCell) ||
        (heuristic(openSet[i], endCell) === heuristic(currentCell, endCell) &&
          +openSet[i].id > +currentCell.id)
      ) {
        currentCell = openSet[i];
      }
    }

    if (currentCell.id === endCell.id) {
      return { historyQueue, matrix: tempMatrix };
    }

    openSet = openSet.filter((cell) => cell.id !== currentCell.id);
    closedSet.push(currentCell);

    if (currentCell.state !== "start" && currentCell.state !== "destination") {
      currentCell.state = "visited";

      historyQueue.push({
        xCord: currentCell.row,
        yCord: currentCell.column,
        newCell: currentCell,
      });
    }

    const neighbors: MatrixCell[] = [];
    if (currentCell.row > 0)
      neighbors.push(tempMatrix[currentCell.row - 1][currentCell.column]);
    if (currentCell.row < matrix.length - 1)
      neighbors.push(tempMatrix[currentCell.row + 1][currentCell.column]);
    if (currentCell.column > 0)
      neighbors.push(tempMatrix[currentCell.row][currentCell.column - 1]);
    if (currentCell.column < tempMatrix[0].length - 1)
      neighbors.push(tempMatrix[currentCell.row][currentCell.column + 1]);

    for (let i = 0; i < neighbors.length; i++) {
      if (neighbors[i].state === "wall" || closedSet.includes(neighbors[i])) {
        continue;
      }

      const tentativeGScore =
        heuristic(currentCell, neighbors[i]) +
        (currentCell.pathLink
          ? heuristic(
              tempMatrix[currentCell.pathLink.row][currentCell.pathLink.column],
              currentCell
            )
          : 0);
      if (!openSet.includes(neighbors[i])) {
        neighbors[i].pathLink = { row: currentCell.row, column: currentCell.column };
        openSet.push(neighbors[i]);
      } else if (tentativeGScore >= heuristic(neighbors[i], endCell)) continue;
      else {
        neighbors[i].pathLink = { row: currentCell.row, column: currentCell.column };
      }
    }
  }

  return { historyQueue, matrix: tempMatrix };
};

export default AStar;
```

Приложение № 5. Реализация упрощенного алгоритма Прима

```
Codeium: Refactor | Explain | Generate JSDoc | ×
const MinimumSpanningTree = (matrix: Matrix) => {
  const historyQueue: QueueItem[] = [];

  const frontierCellsList: [number, number, number, number][] = [];

  const directions = [
    [-2, 0],
    [2, 0],
    [0, -2],
    [0, 2],
  ];

  let tempMatrix = JSON.parse(JSON.stringify(matrix)) as Matrix;

  tempMatrix = tempMatrix.map((row) =>
    row.map((cell) => {
      if (cell.state !== "destination" && cell.state !== "start") {
        historyQueue.push({
          xCord: cell.row,
          yCord: cell.column,
          newCell: { ...cell, state: "wall" },
        });
        return { ...cell, state: "wall" };
      }
      if (cell.state === "start") {
        frontierCellsList.push([cell.row, cell.column, cell.row, cell.column]);
      }
      return cell;
    })
  );

  while (frontierCellsList.length > 0) {
    const [startRow, startColumn] = frontierCellsList.splice(
      random(0, frontierCellsList.length - 1),
      1
    )[0];

    for (const [dx, dy] of directions) {
      const newRow = startRow + dx;
      const newColumn = startColumn + dy;

      if (
        newRow >= 0 &&
        newRow < tempMatrix.length &&
        newColumn >= 0 &&
        newColumn < tempMatrix[0].length &&
        (tempMatrix[newRow][newColumn].state === "wall" ||
          tempMatrix[newRow][newColumn].state === "destination")
      ) {
        const newCell = tempMatrix[newRow][newColumn];
        const midPointRow = (newRow + startRow) / 2;
        const midPointColumn = (newColumn + startColumn) / 2;
        const midPointCell = tempMatrix[midPointRow][midPointColumn];

        frontierCellsList.push([newRow, newColumn, startRow, startColumn]);
        if (newCell.state !== "destination") {
          tempMatrix[newRow][newColumn].state = "default";
          historyQueue.push({ xCord: newRow, yCord: newColumn, newCell });
        }

        if (
          tempMatrix[startRow][startColumn].state !== "start" &&
          tempMatrix[startRow][startColumn].state !== "destination"
        ) {
          tempMatrix[startRow][startColumn].state = "default";
          historyQueue.push({
            xCord: startRow,
            yCord: startColumn,
            newCell: tempMatrix[startRow][startColumn],
          });
        }

        if (midPointCell.state !== "destination") {
          midPointCell.state = "default";
          historyQueue.push({
            xCord: midPointRow,
            yCord: midPointColumn,
            newCell: midPointCell,
          });
        }
      }
    }
  }

  return { matrix: tempMatrix, historyQueue };
};

export default MinimumSpanningTree;
```

Приложение № 6. Реализация алгоритма рекурсивного разделения

```
const recursiveDivision = (
  matrix: Matrix,
  rowStart: number,
  rowEnd: number,
  colStart: number,
  colEnd: number,
  orientation: "horizontal" | "vertical",
  callBack: (newCellInfo: QueueItem) => void
) => {
  let tempMatrix = JSON.parse(JSON.stringify(matrix)) as Matrix;

  if (
    rowEnd < rowStart ||
    colEnd < colStart ||
    colEnd - colStart < 2 ||
    rowEnd - rowStart < 2
  ) {
    return tempMatrix;
  }

  if (orientation === "horizontal") {
    const randomRowWallIndex = random(rowStart + 1, rowEnd - 1);
    const randomColGapIndex = [colStart, colEnd][random(0, 1)];

    for (let column = colStart; column <= colEnd; column++) {
      if (
        column !== randomColGapIndex &&
        tempMatrix[randomRowWallIndex][column].state !== "start" &&
        tempMatrix[randomRowWallIndex][column].state !== "destination"
      ) {
        tempMatrix[randomRowWallIndex][column].state = "wall";
        callBack({
          xCord: randomRowWallIndex,
          yCord: column,
          newCell: tempMatrix[randomRowWallIndex][column],
        });
      }
    }

    const isUpperNewBlockVertical =
      colEnd - colStart + 1 > randomRowWallIndex - 1 - rowStart + 1;

    tempMatrix = recursiveDivision(
      tempMatrix,
      rowStart,
      randomRowWallIndex - 1,
      colStart,
      colEnd,
      isUpperNewBlockVertical ? "vertical" : "horizontal",
      callBack
    );

    const isLowerNewBlockVertical =
      colEnd - colStart + 1 > rowEnd - randomRowWallIndex + 1;

    tempMatrix = recursiveDivision(
      tempMatrix,
      randomRowWallIndex + 1,
      rowEnd,
      colStart,
      colEnd,
      isLowerNewBlockVertical ? "vertical" : "horizontal",
      callBack
    );
  }
```

```
  if (orientation === "vertical") {
    const randomRowGapIndex = [rowStart, rowEnd][random(0, 1)];
    const randomColWallIndex = random(colStart + 1, colEnd - 1);

    for (let row = rowStart; row <= rowEnd; row++) {
      if (
        row !== randomRowGapIndex &&
        tempMatrix[row][randomColWallIndex].state !== "start" &&
        tempMatrix[row][randomColWallIndex].state !== "destination"
      ) {
        tempMatrix[row][randomColWallIndex].state = "wall";
        callBack({
          xCord: row,
          yCord: randomColWallIndex,
          newCell: tempMatrix[row][randomColWallIndex],
        });
      }
    }

    const isLeftNewBlockHorizontal =
      rowEnd - rowStart + 1 > randomColWallIndex - 1 - colStart + 1;

    tempMatrix = recursiveDivision(
      tempMatrix,
      rowStart,
      rowEnd,
      colStart,
      randomColWallIndex - 1,
      isLeftNewBlockHorizontal ? "horizontal" : "vertical",
      callBack
    );

    const isRightNewBlockHorizontal =
      rowEnd - rowStart + 1 > colEnd - randomColWallIndex + 1;

    tempMatrix = recursiveDivision(
      tempMatrix,
      rowStart,
      rowEnd,
      randomColWallIndex + 1,
      colEnd,
      isRightNewBlockHorizontal ? "horizontal" : "vertical",
      callBack
    );
  }

  return tempMatrix;
};
```

Приложение № 7. Реализация алгоритма с использованием ГСПЧ

```
const randowMaze = (matrix: Matrix) => {
  const historyQueue: QueueItem[] = [];

  const tempMatrix = JSON.parse(JSON.stringify(matrix)) as Matrix;

  let buffer = [1, 0, 1, 1, 0, 1, 0, 1, 1, 0];
  let links = [9, 4, 0];

  const randomArray = LFSRNumberGenerator(
    buffer,
    links,
    tempMatrix.length * tempMatrix[0].length
  );

  const maxRandomValue = Math.max(...randomArray);
  const minRandomValue = Math.min(...randomArray);

  const decidingMedian = (maxRandomValue + minRandomValue) / 3;

  for (let row = 0; row < tempMatrix.length; row++) {
    for (let column = 0; column < tempMatrix[0].length; column++) {
      if (
        randomArray[row * tempMatrix[0].length + column] < decidingMedian &&
        tempMatrix[row][column].state !== "start" &&
        tempMatrix[row][column].state !== "destination" &&
        tempMatrix[row][column].state !== "wall"
      ) {
        tempMatrix[row][column].state = "wall";
        historyQueue.push({
          xCord: row,
          yCord: column,
          newCell: tempMatrix[row][column],
        });
      }
    }
  }

  return { matrix: tempMatrix, historyQueue };
};

export default randowMaze;
```