

Федеральное агентство связи

Федеральное государственное образовательное бюджетное
учреждение высшего профессионального образования
«Санкт-Петербургский государственный университет
телекоммуникаций
им. проф. М. А. Бонч-Бруевича»

**ГЕНЕРАЦИЯ И ПОИСК КРАТЧАЙШЕГО ПУТИ В 2D
ЛАБИРИНТАХ**

Проектная работа
по дисциплине «Алгоритмы и структуры данных»

Студенты: Коваленко Л. А.
Курс: 2 Группа: ИКПИ-81
Преподаватель: Дагаев А. В.

Санкт-Петербург
2020

ОГЛАВЛЕНИЕ

1. ГЕНЕРАЦИЯ ЛАБИРИНТОВ	3
1.1 Алгоритмы генерации идеальных лабиринтов	3
1.1.1 Алгоритм Олдоса – Бродера	6
1.1.2 Алгоритм Уилсона	7
1.1.3 Алгоритм двоичного дерева.....	9
1.1.4 Алгоритм Recursive Backtracking	11
1.1.5 Алгоритм рекурсивного деления.....	13
1.1.6 Алгоритм Эллера.....	15
1.1.7 Алгоритм растущего дерева.....	17
1.1.8 Алгоритм Краскала	19
1.1.9 Алгоритм Прима (упрощенный).....	21
1.1.10 Алгоритм Прима (модифицированный)	21
1.1.11 Алгоритм Sidewinder.....	23
1.1.12 Примеры лабиринтов.....	27
1.2 Алгоритмы генерации неидеальных лабиринтов	29
1.2.1 Алгоритм змеевидного лабиринта	29
1.2.2 Алгоритм маленьких комнат	29
1.2.3 Алгоритм спирального лабиринта	30
1.2.4 Примеры лабиринтов.....	31
2. ПОИСК КРАТЧАЙШЕГО ПУТИ В ЛАБИРИНТАХ.....	32
2.1 Алгоритм итеративного поиска в ширину.....	32
2.2 Алгоритм Дейкстры	32
2.3 Алгоритм A*	33
2.4 Примеры поиска	36
3. РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ	42
4. ВЫВОДЫ	45
5. КОД ПРОГРАММЫ.....	46
СПИСОК ЛИТЕРАТУРЫ.....	62

1. ГЕНЕРАЦИЯ ЛАБИРИНТОВ

1.1 Алгоритмы генерации идеальных лабиринтов

Идеальный лабиринт – такой лабиринт, в котором одна клетка связана с другой одним единственным путём. Иначе говоря, остовное дерево.

Для создания стандартного идеального лабиринта обычно необходимо «выращивать» его, обеспечив отсутствие петель и изолированных областей.

Следующие алгоритмы генерируют идеальные лабиринты либо методом добавления стен, либо методом вырезания проходов.

Таблица 1. Алгоритмы генерации идеальных лабиринтов

Алгоритм	% тупиков	Тип	Фокус	Отсутствует смещенность	Однородн ость	Память
Алгоритм Олдоса– Бродера <i>Решение: 13%</i>	14	Дерево	Оба	Да	Да	0
Алгоритм Уилсона <i>Решение: 5%</i>	24	Дерево	Оба	Да	Да	N ²
Двоичное дерево <i>Решение: 3%</i>	24	Множество	Оба	Нет	Никогда	0
Алгоритм Recursive Backtracking <i>Решение: 22%</i>	13	Дерево	Проходы	Да	Никогда	N ²
Рекурсивное деление <i>Решение: 7%</i>	24	Дерево	Стены	Да	Никогда	N
Алгоритм Эллера <i>Решение: 6%</i>	24	Множество	Оба	Нет	Нет	N
Алгоритм растущего дерева <i>Решение: 3/20/4%</i>	2/13/23	Дерево	Оба	Да	Нет	N ²
Алгоритм Краскала <i>Решение: 5%</i>	25	Множество	Оба	Да	Нет	N ²
Алгоритм Прима (упрощ.) <i>Решение: 4%</i>	26	Дерево	Оба	Да	Нет	N ²
Алгоритм Прима (мод.) <i>Решение: 4%</i>	28	Дерево	Оба	Да	Нет	N ²
Алгоритм Sidewinder <i>Решение: 4%</i>	24	Множество	Оба	Нет	Никогда	0*

Решение. Это процент ячеек лабиринта, по которым проходит его решение для типичного лабиринта, создаваемого алгоритмом. Здесь предполагается, что лабиринт состоит из 100×100 проходов, а начало и конец находятся в противоположных углах. Этот параметр является показателем «извилистости» пути решения. Максимальную извилистость имеют одномаршрутные лабиринты, потому что решение проходит по всему лабиринту. Минимально возможную извилистость имеет двоичное дерево, у которого решение просто пересекает лабиринт и никогда не отклоняется и не приостанавливает движение по направлению к концу. Обычно генерация добавлением стен имеет те же свойства, что и вырезание проходов, но если значения сильно отличаются, то в скобках указывается процент в случае добавления стен.

Тупики. Это приблизительный процент ячеек, являющихся тупиками в лабиринте. Обычно при добавлении стен процент такой же, как и при вырезании проходов, но если они значительно отличаются, то в скобках указан процент при добавлении стен. Значение для алгоритма выращивания дерева на самом деле варьируется от 10% (если всегда выбирается самая новая ячейка) до 49% (если всегда выбирается самая старая ячейка). При достаточно высоком показателе проходов количество тупиков Recursive Backtracking может становиться ниже 1%. Наибольший вероятный процент тупиков в двухмерном ортогональном идеальном лабиринте равен 66% — это одномаршрутный проход с кучей тупиков единичной длины по обеим сторонам от него.

Тип. Существует два типа алгоритмов создания идеальных лабиринтов:

- Алгоритм на основе дерева выращивает лабиринт подобно дереву, всегда добавляя к тому, что уже есть, и на каждом этапе имея правильный идеальный лабиринт.
- Алгоритм на основе множеств выполняет построения там, где ему хочется, отслеживая части лабиринта, соединённые друг с другом, чтобы соединить всё и создать правильный лабиринт на момент завершения.

Фокус. Большинство алгоритмов можно реализовать или как вырезание проходов, или как добавление стен. Очень немногие можно реализовать только как один или другой подход. В одномаршрутных лабиринтах всегда используется добавление стен, потому что в них задействуется разбиение проходов стенами на две части, однако базовый лабиринт можно создать любым способом. Recursive Backtracking нельзя реализовать с добавлением стен, потому что в этом случае он склонен создавать путь решения, который следует вдоль внешнего края, где вся внутренняя часть лабиринта соединена с границей единственным проходом. Рекурсивное деление нельзя использовать для вырезания проходов, потому что это приводит к созданию очевидного решения, которое или следует вдоль внешнего края, или иначе напрямую пересекает внутреннюю часть.

Отсутствие смещенности. Одинаково ли воспринимает алгоритм все направления и стороны лабиринта так, что последующий анализ лабиринта не может обнаружить никакой смещенности проходов. Алгоритм двоичного дерева чрезвычайно смещён, в нём легко перемещаться в один угол и сложно в противоположный. Sidewinder тоже смещён, в нём легко перемещаться к одному краю и сложно к противоположному. Алгоритм Эллера склонен к созданию проходов, приблизительно синхронизируя начальные или конечные края.

Однородность. Генерирует ли алгоритм все возможные лабиринты с равной вероятностью. «Да» означает, что алгоритм полностью однороден. «Нет» означает, что

алгоритм потенциально может генерировать все возможные лабиринты в пределах любого пространства, но не с равной вероятностью. «Никогда» означает, что существуют возможные лабиринты, которые алгоритм никогда не сможет сгенерировать. Учтите, что только алгоритмы с полным отсутствием смещенности могут быть полностью однородными.

Память. Объём дополнительной памяти или стека, необходимый для реализации алгоритма. Эффективные алгоритмы требуют только битовой карты самого лабиринта, в то время как другие требуют объёма памяти, пропорционального одной строке (N), или пропорционального количеству ячеек (N^2). Некоторым алгоритмам даже не нужно иметь в памяти весь лабиринт, и они могут добавлять части лабиринта бесконечно (такие алгоритмы помечены звёздочкой). Алгоритму Эллера нужен объём памяти для хранения строки, но большего ему не требуется, потому что достаточно хранить только одну строку лабиринта. Алгоритму Sidewinder тоже нужно хранить только одну строку лабиринта, в то время как двоичному дереву нужно отслеживать только текущую ячейку. Для рекурсивного деления требуется стек объёмом вплоть до размера строки, но ему не нужно смотреть на битовую карту лабиринта.

1.1.1 Алгоритм Олдоса – Бродера

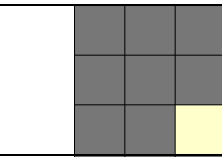
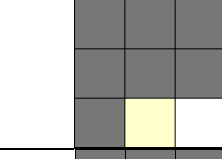
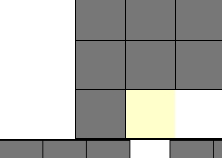
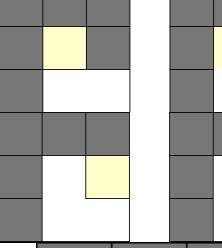
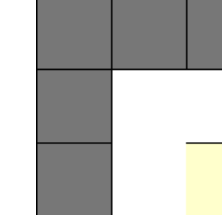
Алгоритм Олдоса–Бродера — однородный алгоритм, то есть он с равной вероятностью создаёт все возможные лабиринты заданного размера. Кроме того, ему не требуется дополнительной памяти или стека.

Алгоритм:

1. Изначально все поле содержит стены.
2. Выбираем любую ячейку.
3. Перемещаемся в любую соседнюю ячейку в пределах поля.
4. Если мы попали в не вырезанную ячейку, то вырезаем в неё проход из предыдущей.
5. Пока не вырежем проходы во все ячейки (пока все ячейки не будут посещены), повторяем №3-4: продолжаем двигаться в соседние ячейки, пока не вырежем проходы во все ячейки.

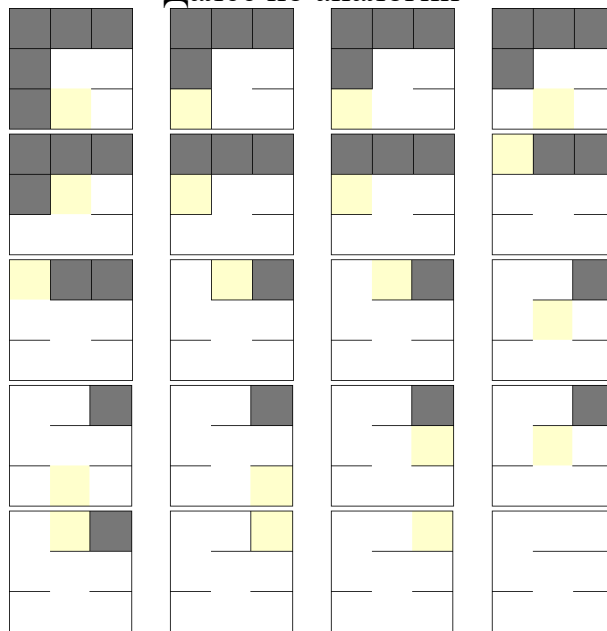
Плохо в этом алгоритме то, что он очень медленный, так как не выполняет интеллектуального поиска последних ячеек, то есть, по сути, не имеет гарантий завершения. Однако из-за своей простоты он может быстро проходить по множеству ячеек, поэтому завершается быстрее, чем можно было бы подумать. В среднем его выполнение занимает в семь раз больше времени, чем у стандартных алгоритмов, хотя в плохих случаях оно может быть намного больше, если генератор случайных чисел постоянно избегает последних нескольких ячеек.

Таблица 2

Пример работы	
Выбираем ячейку наугад	
Идем к соседу	
Поскольку соседа раньше не посещали, вырезаем проход между ними	
Выбираем случайного соседа и делаем проход по нему.	
Поскольку сосед был посещен ранее, мы не вырезаем проход к нему из предыдущей ячейки.	

Пример работы

Далее по аналогии



1.1.2 Алгоритм Уилсона

Алгоритм Уилсона — усовершенствованная версия алгоритма Олдоса-Бродера, создаёт лабиринты точно с такой же текстурой (алгоритмы однородны, то есть все возможные лабиринты генерируются с равной вероятностью), но выполняется гораздо быстрее.

Алгоритм:

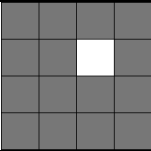
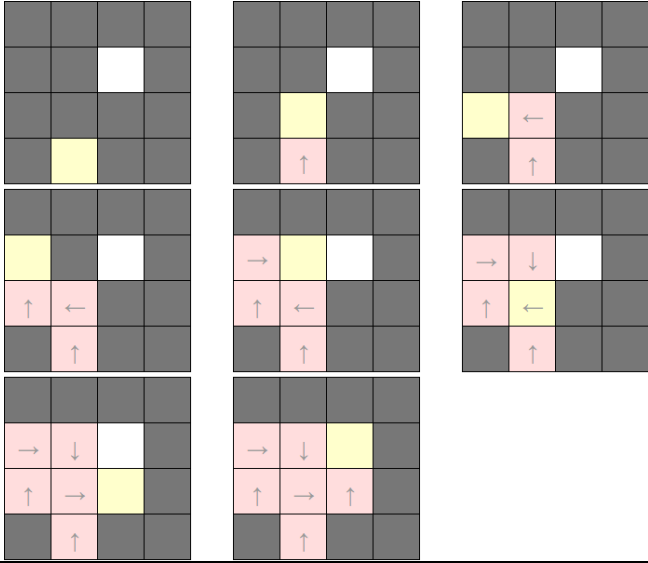
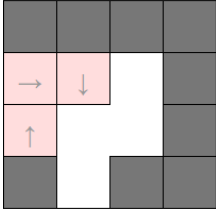
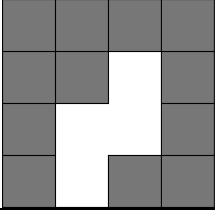
1. Изначально все поле содержит стены.
2. Выбираем любую ячейку и добавляем ее в UST.
3. Выбираем любую ячейку, которой еще нет в UST, и выполняем случайную прогулку, пока не встретим ячейку, которая находится в UST.
4. Добавляем ячейки и ребра, затронутые в случайном блуждании, к UST.
5. Повторяем №3-4, пока все ячейки не будут добавлены к UST.

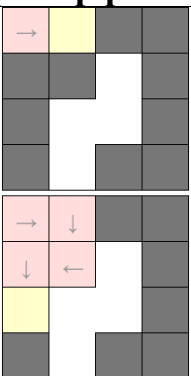
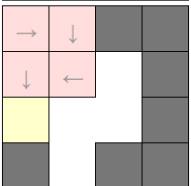
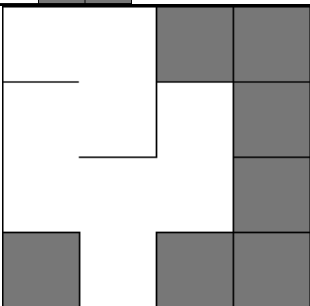
UST (uniform spanning trees) — однородные остовные деревья.

Остовное дерево — это дерево, которое соединяет все вершины графа.

Однородное остовное дерево — это любое из возможных остовных деревьев графа, выбранное случайным образом и с равной вероятностью.

Алгоритм имеет те же проблемы со скоростью, что и алгоритм Олдоса-Бродера, потому что может уйти много времени на нахождение первого случайного пути к начальной ячейке, однако после размещения нескольких путей остальная часть лабиринта вырезается довольно быстро. В среднем он выполняется в пять раз быстрее Олдоса-Бродера, и менее чем в два раза медленнее лучших по скорости алгоритмов. Стоит учесть, что в случае добавления стен он работает в два раза быстрее, потому что вся стена границы изначально является частью лабиринта, поэтому первые стены присоединяются гораздо быстрее.

Пример работы	
Начинаем со случайного добавления ячейки в лабиринт	
<p>Затем мы выбираем случайно другую не посещённую ячейку и совершаем случайный обход, пока не встретим эту первую ячейку. Обратите внимание, что у этого случайного обхода есть несколько ограничений: хотя он может пересекать уже пройденные ячейки (если они еще не в лабиринте), мы не хотим, чтобы в конечном пути были какие-либо петли. Таким образом, мы также записываем направление, последнее использовавшееся для выхода из каждой ячейки, и будем использовать эти направления для формирования окончательного пути, как только прогулка встретится со стартовой ячейкой.</p> 	
<p>Как только мы достигаем ячейки, которая уже является частью лабиринта, прогулка заканчивается. Следующая фаза просто возвращается к ячейке в начале прогулки и следует за стрелками, добавляя вершины и ребра в лабиринт, пока мы не достигнем последней ячейки прогулки.</p> 	
Все остальные ячейки, которые были посещены во время прогулки, но которые не сделали «окончательный срез», просто сбрасываются.	
<p>Теперь мы делаем это снова. Обратите внимание, что на этот раз в лабиринте четыре ячейки, а не одна, что дает нам гораздо больше целей для попадания. Это то, что позволяет алгоритму сходиться быстрее: каждый проход по алгоритму увеличивает вероятность того, что следующий проход закончится раньше.</p>	

Пример работы			
			
			
Затем мы идем по пути и снова добавляем ячейки и ребра в лабиринт			
К настоящему времени паттерн становится виден: каждый проход добавит еще одну случайную «ветвь» к дереву, пока все ячейки не будут добавлены в лабиринт.			

1.1.3 Алгоритм двоичного дерева

Алгоритм двоичного дерева — это самый простой и быстрый из возможных алгоритмов. Однако создаваемые лабиринты имеют текстуру с очень высокой смещённостью.

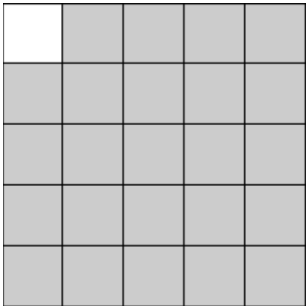
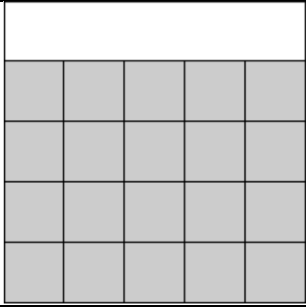
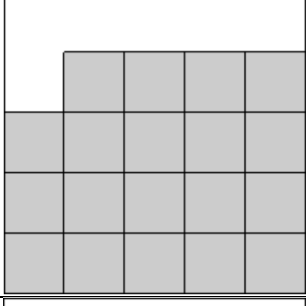

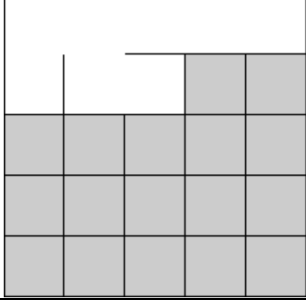
Алгоритм:

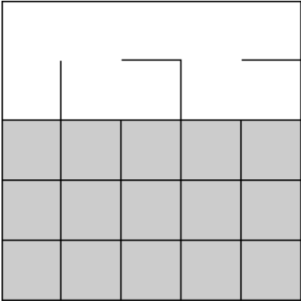
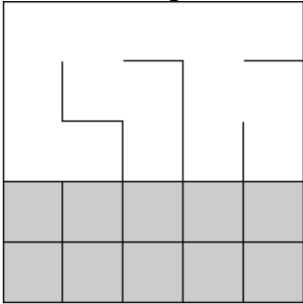
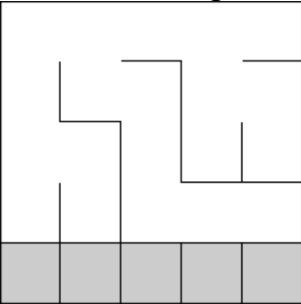
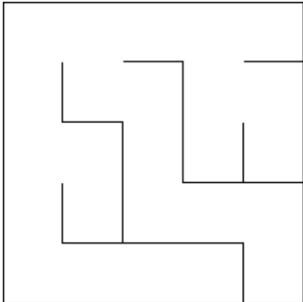
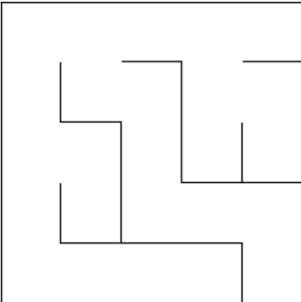
1. Изначально все поле содержит стены.
2. Для каждой ячейки в сетке случайным образом разделяем проход на север или запад (север или восток / юг или запад / юг или восток).

Причем: любой выбранный вами диагональный набор должен использоваться последовательно во всем лабиринте.

Каждая ячейка независима от всех других ячеек, потому что не нужно при её создании проверять состояние каких-либо других ячеек. Следовательно, это настоящий алгоритм генерации лабиринтов без памяти, не ограниченный по размерам создаваемых лабиринтов (Такое свойство отлично подходит для визуального отображения бесконечно большого лабиринта при перемещении по нему.).

Лабиринты на основе двоичных деревьев отличаются от стандартных идеальных лабиринтов, потому что в них не может существовать больше половины типов ячеек. Например, в них никогда не будет перекрёстков, а все тупики имеют проходы, ведущие вверх или влево, и никогда не ведущие вниз или вправо. Лабиринты склонны иметь проходы, ведущие по диагонали из верхнего левого в нижний правый угол, и по ним гораздо проще двигаться из нижнего правого в верхний левый угол. Всегда можно перемещаться вверх или влево, но никогда одновременно в оба направления, поэтому всегда можно детерминированно перемещаться по диагонали вверх и влево, не сталкиваясь с барьерами. Иметь возможность выбора и попадать в тупики вы начнёте, перемещаясь вниз и вправо.

Пример работы		
<p>Поскольку этот алгоритм не должен учитывать состояние каких-либо соседних ячеек, мы можем начать с любого угла.</p> <p>Начнем с верхнего левого.</p>		
<p>Для первой строки мы вырезаем весь проход.</p>		
<p>Вырезать весь проход по первому столбцу можно сразу. Мы будем делать это последовательно.</p>		
<p>Идем сверху-вниз, слева-направо. Подкидываем монетку — прорезаем сверху-вниз.</p> <p>(Мы могли бы прорезать слева-направо, как представлено на втором рисунке.)</p>		<p>Могло быть так:</p> 
<p>Далее снова подкидываем монетку. Прорезаем слева-направо.</p>		

Пример работы	
Таким же случайным образом генерируется вторая строка	
Затем третья	
Затем четвертая	
Затем пятая	
Получается идеальный лабиринт	
	

1.1.4 Алгоритм Recursive Backtracking

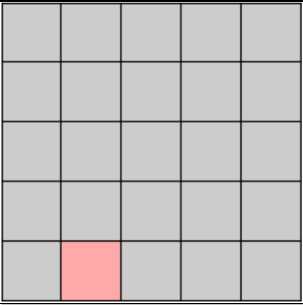
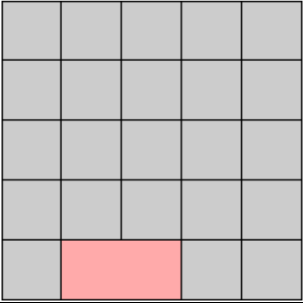
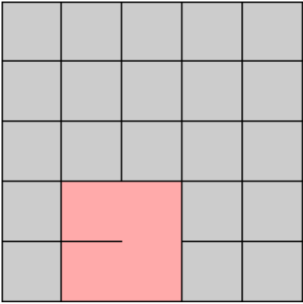
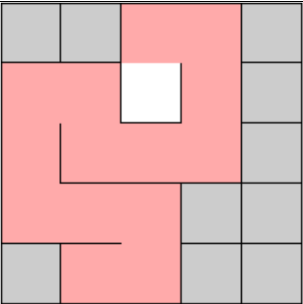
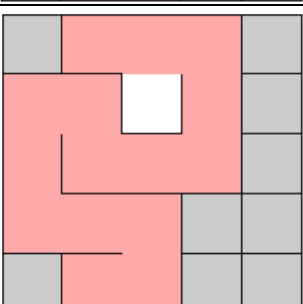
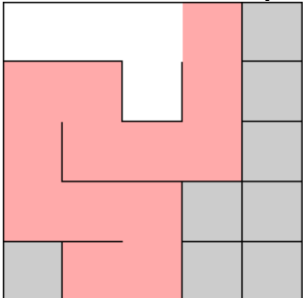
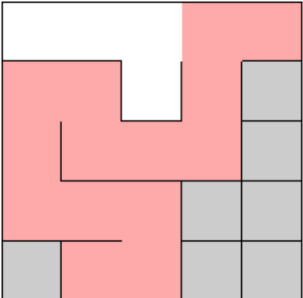
Алгоритм Recursive Backtracking — алгоритм генерации лабиринта поиском в глубину. Требует стека, объём которого может достигать до размеров лабиринта.

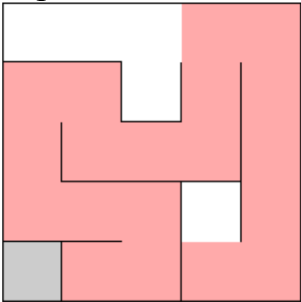
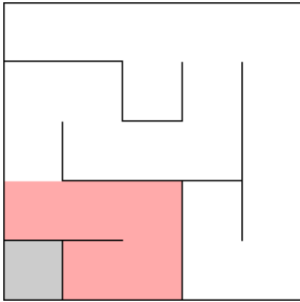
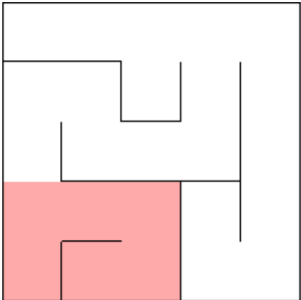
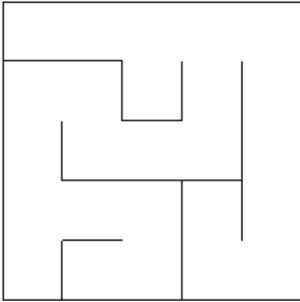
Алгоритм:

1. Изначально все поле содержит стены.
2. Выбираем любую ячейку.
3. Выбираем любую стену и делаем проход в соседнюю ячейку, но только если соседняя ячейка еще не посещена. Новая ячейка становится текущей ячейкой.
4. Если все соседние ячейки были посещены, возвращаемся к последней ячейке, которая имеет не вырезанные стенки, и повторяем с №3.
5. Алгоритм заканчивается, когда процесс полностью возвращается к начальной ячейке.

Такой метод приводит к созданию лабиринтов с максимальным показателем текучести, тупиков меньше, но они длиннее, а решение обычно оказывается очень долгим и извилистым. При правильной реализации он выполняется быстро.

Recursive Backtracking нельзя реализовать с добавлением стен, потому что в этом случае он склонен создавать путь решения, который следует вдоль внешнего края, где вся внутренняя часть лабиринта соединена с границей единственным проходом.

Пример работы	
Случайно выбираем ячейку	
В этой ячейке случайным образом выбираем стену и делаем проход в соседнюю ячейку, но только если соседняя ячейка еще не посещена. Новая ячейка становится текущей ячейкой.	
Продолжаем...	
Мы оказались в той ситуации, когда соседние ячейки уже были посещены.	
Если все соседние ячейки были посещены, то нам следует вернуться к последней ячейке, которая имеет не вырезанные стенки, и повторить алгоритм для неё.	
Текущая ячейка имеет справа стенку	... в которую мы и вырезаем проход
	

Пример работы	
<p>Мы оказались далеко от одной необработанной ячейки</p> 	<p>Доходим до неё</p> 
<p>Вырезаем проход</p> 	

1.1.5 Алгоритм рекурсивного деления

Алгоритм рекурсивного деления — алгоритм, работающий со стенами.

Алгоритм:

1. Изначально все поле не содержит стены.
2. Разделяем поле стеной по горизонтали или вертикали.
3. Добавляем один проход через стену.
4. Повторяем шаги №2-3 с областями по обе стороны от стены. Продолжаем рекурсивно, пока лабиринт не достигнет желаемого разрешения.

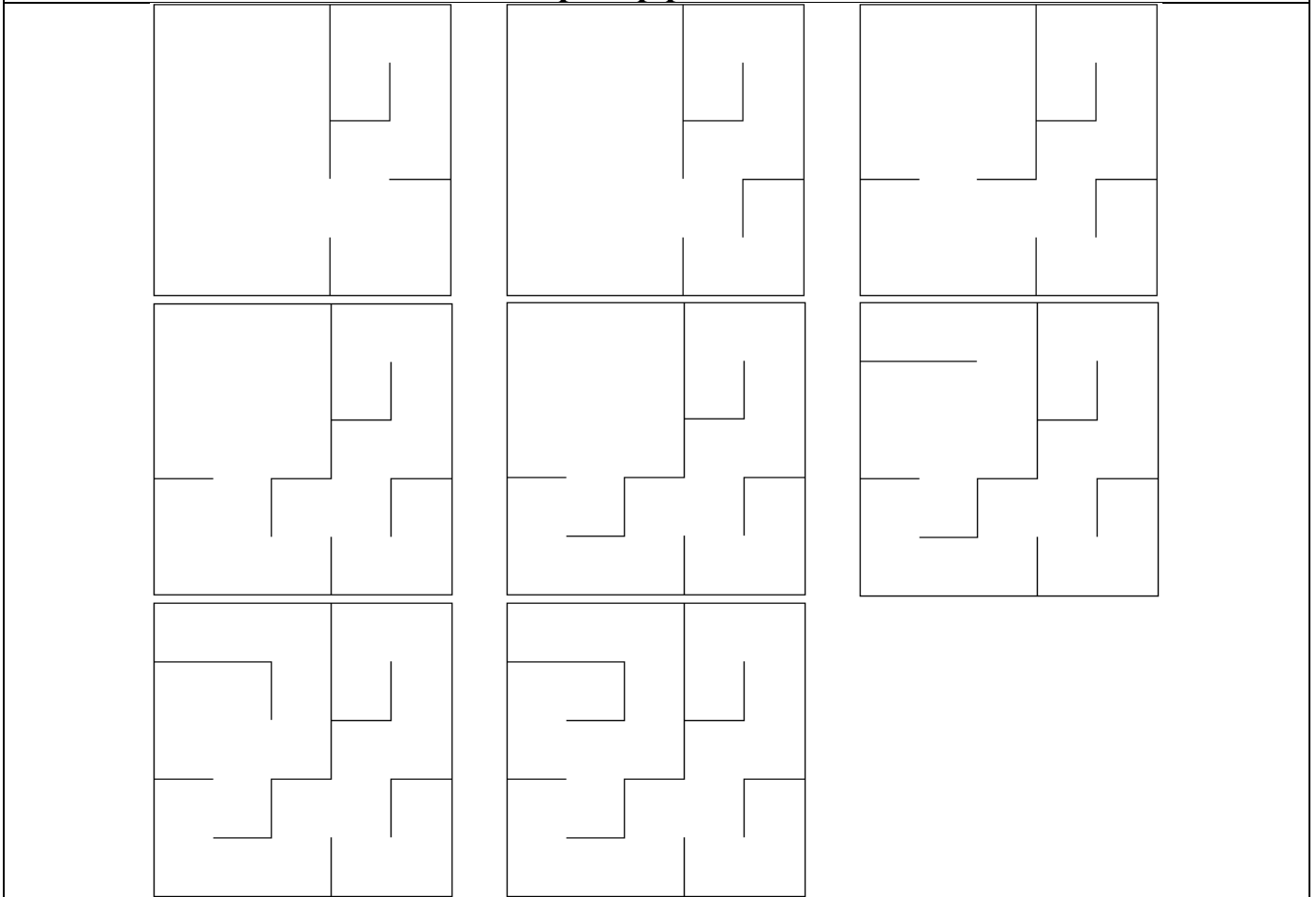
Причем: для наилучших результатов следует добавить отклонение в выборе горизонтали или вертикали на основе пропорций области, например, область, ширина которой вдвое больше высоты, должна более часто (или всегда) делиться вертикальными стенами.

Это самый быстрый алгоритм без отклонений в направлениях, и часто он может даже соперничать с лабиринтами на основе двоичных деревьев, потому что он создаёт одновременно несколько ячеек, хоть и имеет очевидный недостаток в виде длинных стен, пересекающих внутренности лабиринта.

Рекурсивное деление нельзя использовать для вырезания проходов, потому что это приводит к созданию очевидного решения, которое или следует вдоль внешнего края, или иначе напрямую пересекает внутреннюю часть.

Пример работы			
Начинаем с пустого поля. В нашем случае 5×5 .			
Разделяем поле стеной по горизонтали или вертикали.			
Затем мы случайным образом добавляем пробел в стену, чтобы сохранить связность.			
<p>И это завершает первую итерацию.</p> <p>Теперь, теоретически, мы можем делить по вертикали или по горизонтали две области слева и справа. Однако для достижения наилучших результатов, если область больше по ширине, то делим вертикально, иначе горизонтально (как сейчас).</p>			
Сначала обрабатываем поле справа. Разрежем это поле пополам горизонтально, а затем добавим проход.			
Поле в верхнем правом углу является квадратом, поэтому мы можем произвольно выбирать между горизонтальным или вертикальным делением пополам. Пойдем по вертикали.			
<p>Мы не можем делить их дальше, так как наша сетка составляет всего 5×5. Итак, рекурсия заканчивается, и мы перематываем обратно в стек.</p> <p>Далее мы обрабатываем нижний правый край. Потом нижний левый. И наконец верхний левый.</p>			

Пример работы



1.1.6 Алгоритм Эллера

Алгоритм Эллера — особый алгоритм, потому что он не только довольно быстр, но и не имеет очевидной смещенности или недостатков; кроме того, при его создании память используется наиболее эффективно. (Для него не требуется, чтобы в памяти находился весь лабиринт; он использует объём, пропорциональный размеру строки.)

Алгоритм:

1. Инициализируем ячейки первой строки, чтобы каждая существовала в своем собственном уникальном наборе (множестве). По итогу этого пункта должно получиться N наборов, где N — количество ячеек первой строки. Т. е. для 8-ми ячеек: [1 2 3 4 5 6 7 8].
2. Теперь случайным образом объединяем соседние ячейки в один набор (но только если они не находятся в одном наборе) или разделяем их. При объединении соседних ячеек объединяем ячейки обоих наборов в один набор, указывая, что все ячейки в обоих наборах теперь связаны. По итогу может получиться следующее: [1 1 1|4 5|6 6 6].
3. Добавляем нижние границы. Убедимся, что каждый набор имеет хотя бы одну ячейку без нижней границы. Если это условие не будет выполнено, то мы создадим изолированные области. По итогу: [1 _1_1_|4 _5_|6 6 _6_].
4. Перемещаем на следующий ряд те ячейки наборов, которые не имели нижних границ. Следующая строка получается так: [1 4 6 6].
5. Присоединяем ячейки, не принадлежащие множествам к своим уникальным множествам. Т. е. так: [1 2 3 4 5 6 6 7].
6. Повторяем №2-5, пока не будет достигнут последний ряд.

7. В последнем ряду: (1) объединяем смежные ячейки различных множеств, (2) не удаляем границу между двумя смежными ячейками одного множества.

Проблема этого алгоритма заключается в несбалансированности обработки разных краёв лабиринта; чтобы избежать пятен в текстурах нужно выполнять соединение и пропуск соединения ячеек в правильных пропорциях.

Таблица 7

Пример работы	
Создаём первую строку	— — — — — — — —
Инициализируем ячейки первой строки, чтобы каждая существовала в своем собственном наборе	1 2 3 4 5 6 7 8
Далее объединяем или создаем границы — выбор производится случайно. НО объединять ячейки одного набора нельзя! (Первой строки не касается.)	(1 2) 3 4 5 6 7 8 1 (1 3) 4 5 6 7 8 1 1 (1 4) 5 6 7 8
Результат	1 1 1 4 4 6 6 6
Добавляем нижние границы. Убедимся, что каждый набор имеет хотя бы одну ячейку без нижней границы. Если это условие не будет выполнено, то мы создадим изолированные области.	1 _1_ _1_ 4 _4_ 6 6 _6_
Перемещаем на следующий ряд те ячейки наборов, которые не имели нижних границ.	1 _1_ _1_ 4 _4_ 6 6 _6_ 1 4 6 6
Присоединим ячейки, не принадлежащие множествам к своим уникальным множествам	1 2 3 4 5 6 6 7
Далее объединяем или создаем границы — выбор производится случайно. НО объединять ячейки одного набора нельзя!	1 2 3 4 5 6 6 7
	...
	1 2 2 2 (5 6) 6 7
Следующие две ячейки — члены одного набора, поэтому мы должны добавить границу. Если не добавим, то это приведет к циклам	1 2 2 2 5 (6 6) 7
6 и 7 объединяем	1 2 2 2 5 6 (6 7)
Добавляем нижние границы. Убедимся, что каждый набор имеет хотя бы одну ячейку без нижней границы.	1 2 _2_ _2_ 5 _6_ 6 _6_

Пример работы	
<p>Таким образом мы можем добавить столько строк, сколько захотим</p> <p>Номера множеств сами по себе роли не играют, поэтому их можно менять на те, которых нет в текущей строке.</p>	
...	...
<p>Завершение лабиринта.</p> <p>Последняя строка отличается от обычных тем, что:</p> <p>1) Каждая ячейка имеет границу снизу.</p> <p>2) Каждая ячейка должна принадлежать одному множеству.</p> <p>В последнем ряду: (1) объединяем смежные ячейки различных множеств, (2) не удаляем границу между двумя смежными ячейками одного множества.</p>	

1.1.7 Алгоритм растущего дерева

Алгоритм растущего дерева — обобщённый алгоритм, способный создавать лабиринты с разной текстурой. Требуемая память может достигать размера лабиринта.

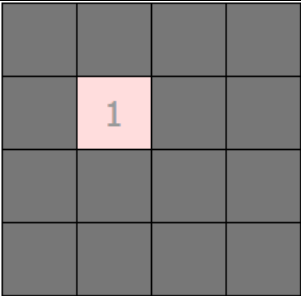
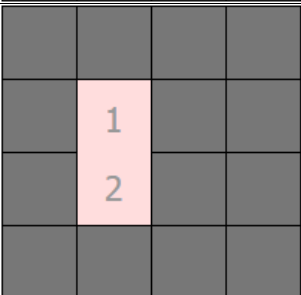
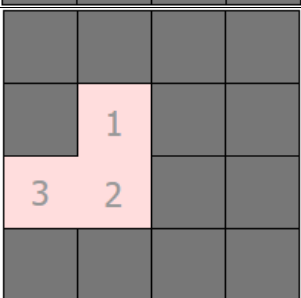
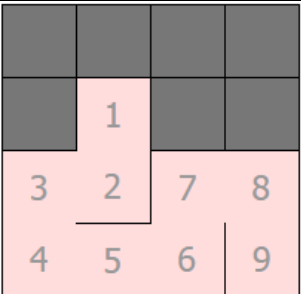
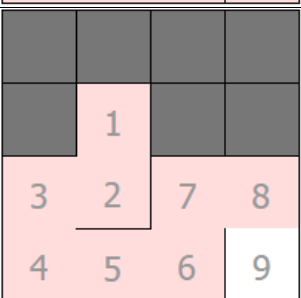
Алгоритм:

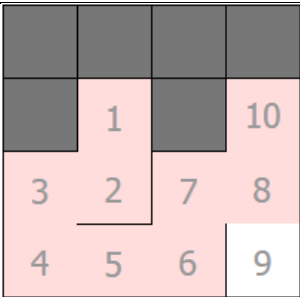
1. Изначально все поле содержит стены.
2. Пусть C будет списком ячеек, изначально пустым.
3. Добавляем любую ячейку к C .
4. Берем ячейку из C и прорезаем проход к любому не посещённому соседу этой ячейки, добавив этого соседа также в C . Если соседей нет, удаляем текущую ячейку из C .
5. Повторяем №4, пока C не станет пустым.

Интересно в алгоритме то, что в зависимости от способа выбора ячейки из списка можно создавать множество разных текстур. Например, если всегда выбирать последнюю добавленную ячейку, то этот алгоритм превращается в Recursive Backtracking. Если всегда выбирать ячейки случайно, то он ведёт себя похоже (но не одинаково) на алгоритм Прима. Если всегда выбирать самые старые ячейки, добавленные в список, то мы создадим лабиринт с наименьшим возможным показателем текучести, даже ниже, чем у алгоритма Прима. Если обычно выбирать самую последнюю ячейку, но время от времени выбирать случайную ячейку, то лабиринт будет иметь высокий показатель текучести (короткое и прямое решение).

Если случайно выбирать одну из самых последних ячеек, то лабиринт будет иметь низкий показатель текучести (долгое и извилистое решение).

Таблица 8

Пример работы				
Будем использовать метод выбора ячеек «выбрать самую новую».				
Алгоритм начинается с добавления произвольной ячейки в список. Кроме того, ячейки красного цвета находятся в списке «живых» ячеек; они станут белыми, как только они будут удалены из списка.				
Далее мы выбираем новую ячейку из списка, случайным образом выбираем одного из ее не посещённых соседей, вырезаем путь к ней и добавляем соседа в список.				
Давайте сделаем это снова, еще раз выбрав самую новую ячейку из списка:				
Видите образец? Всегда выбирая ячейку, последнюю из которых добавили в список, каждый последующий шаг просто продлевает переход еще на один шаг, эффективно выполняя случайный обход. Но как ведет себя алгоритм, когда переход не может быть расширен дальше?				
Давайте перенесемся немного вперед и посмотрим на поведение. Еще шесть итераций, и мы зашли в тупик в ячейке № 9.				
В этот момент алгоритм выберет самую новую ячейку №9, а затем попытается найти соседнюю ячейку, которая не посещалась. Нет ни одной! Итак, клетка №9 удалена из списка С.				

Пример работы				
Затем алгоритм снова обходит, отбирая самую новую ячейку из списка. На этот раз это №8, и, конечно же, есть соседняя не посещённая ячейка, в которую мы можем перейти:				
				
Шаг 8→10 не был преднамеренным; просто случилось так, что алгоритм выбора ячейки был тем же, что и алгоритм возврата. Вместо этого мы могли бы выбрать не 8, а любую другую, если бы мы использовали метод выбора ячеек «выбрать случайную». Таким образом алгоритм будет продолжать генерацию лабиринта до тех пор, пока каждая ячейка не будет посещена.				

1.1.8 Алгоритм Краскала

Алгоритм Краскала — алгоритм, создающий минимальное связующее дерево. Он не «выращивает» лабиринт подобно дереву, а скорее вырезает проходы по всему лабиринту случайным образом, и тем не менее в результате создаёт идеальный лабиринт. Для его работы требуется объём памяти, пропорциональный размеру лабиринта, а также возможность перечисления каждого ребра или стены между ячейками лабиринта в случайном порядке (обычно для этого создаётся список всех рёбер и перемешивается случайным образом).

Алгоритм:

1. Изначально все поле содержит стены.
2. Помечаем каждую ячейку уникальным идентификатором.
3. Добавляем все рёбра в множество *Edges*.
4. Берем случайное ребро из множества *Edges*.
 - Если ячейки с обеих сторон от взятого ребра имеют разные идентификаторы, то удаляем ребро (из *Edges* и в лабиринте) и задаем всем ячейкам с одной стороны тот же идентификатор, что и ячейкам с другой.
 - Если ячейки с обеих сторон от взятого ребра имеют одинаковые идентификаторы, то между ними уже существует какой-то путь, поэтому текущее ребро пропускаем (удаляем только из *Edges*).
5. Повторяем №4, пока множество *Edges* не станет пустым.

Примечание. Объединение двух множеств по обеим сторонам стены будет медленной операцией, если у каждой ячейки есть только номер и они объединяются в цикле. Объединение, а также поиск можно выполнять почти за постоянное время благодаря использованию алгоритма объединения-поиска (union-find algorithm): помещаем каждую ячейку в древовидную структуру, корневым элементом является идентификатор. Объединение выполняется быстро благодаря сращиванию двух деревьев. При правильной реализации этот алгоритм работает достаточно быстро, но медленнее большинства из-за создания списка рёбер и управления множествами.

Пример работы

Выбираем ребро случайным образом и соединяем ячейки, которые он соединяет, если они еще не соединены путем. Мы можем знать, подключены ли ячейки, если они находятся в одном наборе.

Итак, давайте выберем грань между (2, 2) и (2, 3). Ячейки находятся в разных наборах, поэтому мы объединяем их в один набор и соединяем ячейки.

A	B	C
D	E	F
G	H	I
A	B	C
D	E	F
G	E	I

Давайте сделаем еще несколько проходов алгоритма, чтобы перейти к интересной части:

A	A	C
D	E	F
G	E	I

A	A	C
D	E	C
G	E	I

A	A	C
D	E	C
G	E	E

A	A	C
D	E	C
E	E	E

Обратите внимание, что происходит с ребром между (2, 1) и (2, 2). Два дерева A и E, были объединены в один набор A.

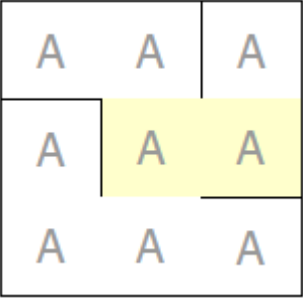
A	A	C
D	A	C
A	A	A

Объединяем (1, 2) и (1, 3).

A	A	C
A	A	C
A	A	A

Теперь рассмотрим ребра (1, 1) – (1, 2) и (1, 2) – (2, 2). В обоих случаях ячейки по обе стороны от ребер принадлежат одному и тому же набору. Соединение ячеек в любом случае приведет к циклу, поэтому мы

После еще одного прохода у нас будет:

Пример работы		
пропускаем эти ребра (удаляем из рассмотрения).		
Алгоритм завершается, когда больше нет ребер для рассмотрения.		

1.1.9 Алгоритм Прима (упрощенный)

Алгоритм Прима (упрощенный) — алгоритм, создающий минимальное связующее дерево. Все веса рёбер одинаковы. Для него требуется объём памяти, пропорциональный размеру лабиринта.

Алгоритм:

1. Изначально все поле содержит стены.
2. Выбираем произвольную ячейку и помечаем её как обработанную.
3. Прорезаем любое ребро, соединяющее обработанную часть лабиринта с ячейкой, которая ещё не обработана (новая ячейка становится обработанной).
4. Повторяем №3, пока все ячейки не будут обработаны.

Более формально:

1. Изначально все поле содержит стены.
2. Выбираем любую ячейку и добавляем её к некоторому (изначально пустому) множеству V .
3. Выбираем любое ребро, которое проходит между ячейкой в V и другой ячейкой не в V .
4. Прорезаем проход через это ребро, а новую ячейку добавляем к V .
5. Повторяем №3-4, пока V не включит каждую ячейку сетки.

Примечание. Так как рёбра не имеют веса и не упорядочены, их можно хранить как простой список.

Создаваемая текстура лабиринта будет иметь меньший показатель текучести и более простое решение, чем у истинного алгоритма Прима (который не рассматривается), потому что распространяется из начальной точки равномерно, как пролитый сироп, а не обходит фрагменты рёбер с более высоким весом, которые учитываются позже.

Пример работы схож с примером работы следующего алгоритма за незначительным отличием: в данном алгоритме ячейка достигается через одно из ребер (которое ведет в неё), а в другом ячейка достигается от случайно выбранной пограничной ячейки.

1.1.10 Алгоритм Прима (модифицированный)

Алгоритм Прима (модифицированный) — алгоритм, создающий минимальное связующее дерево, в котором все веса рёбер одинаковы. Он реализован таким образом, что вместо рёбер смотрит на ячейки. Объём памяти пропорционален размеру лабиринта.

В процессе создания каждая ячейка может иметь один из 3-х типов:

- «Внутренняя»: ячейка является частью лабиринта и уже вырезана в нём;

- «Граничная»: ячейка не является частью лабиринта и ещё не вырезана в нём, но находится рядом с ячейкой, которая уже является «внутренней»;
- «Внешняя»: ячейка ещё не является частью лабиринта, и ни один из её соседей тоже не является «внутренней» ячейкой.

Алгоритм:

1. Изначально все поле содержит стены.
2. Выбираем ячейку, делаем её «внутренней», а для всех её соседей задаем тип «граничная».
3. Выбираем случайным образом «граничную» ячейку и вырезаем в неё проход из соседней «внутренней» ячейки. Поменяем состояние этой «граничной» ячейки на «внутреннюю» и изменим тип всех её «внешних» соседей на «граничный».
4. Повторяем №3, пока больше не останется «граничных» ячеек.

Этот алгоритм создаёт лабиринты с очень низким показателем текучести, имеет множество коротких тупиков и довольно прямолинейное решение. Полученный лабиринт очень похож на результат упрощённого алгоритма Прима. Кроме того, алгоритм очень быстр, быстрее упрощённого алгоритма Прима, потому что ему не нужно составлять и обрабатывать список рёбер.

Таблица 10

Пример работы				
Выбираем случайную ячейку				
Теперь мы выбираем одну из этих пограничных ячеек наугад и делим проход из этой пограничной ячейки в ту, которая соседняя ячейка уже является частью лабиринта. Затем мы отметим соседей бывшей пограничной ячейки самими «пограничными ячейками».				
Теперь интересный момент. Посмотрите, что произойдет, если мы случайно выберем ячейку в (1,0) (верхняя середина). Он примыкает к двум клеткам, которые уже в лабиринте. Алгоритм решает эту проблему, просто выбирая одного из соседей наугад. Ему не важно, какой сосед выбран, только то, что каждая пограничная клетка в конечном итоге будет соединена с клеткой, уже находящейся в лабиринте.				

1.1.11 Алгоритм Sidewinder

Алгоритм Sidewinder — простой алгоритм, очень похож на алгоритм двоичного дерева, но сложнее.

Алгоритм:

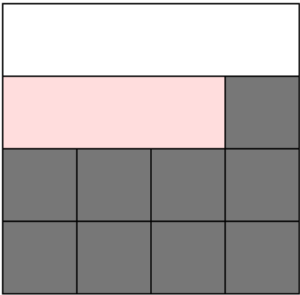
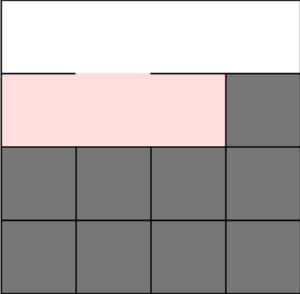
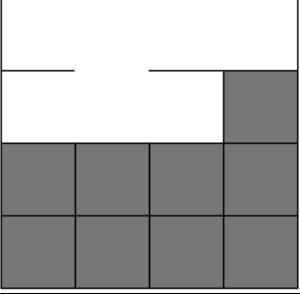
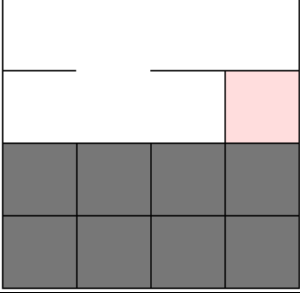
1. Изначально все поле содержит стены.
2. Работаем по сетке построчно, начиная с ячейки (0, 0). Инициализируем набор *Run* как пустой.
3. Добавляем текущую ячейку в набор *Run*.
4. Для текущей ячейки, случайным образом решаем, следует ли вырезать восток или нет.
5. Если фрагмент был вырезан, сделаем новую ячейку текущей и повторяем №3-5. Если проход не был вырезан, выбираем любую из ячеек в наборе *Run* и вырезаем проход на север, затем очищаем набор *Run*, устанавливаем следующую ячейку в строке как текущую и повторяем №3-5.
6. Продолжаем, пока все строки не будут обработаны.

В то время как лабиринт двоичного дерева всегда поднимается вверх от самой левой ячейки горизонтального прохода, лабиринт Sidewinder поднимается вверх от случайной ячейки. В двоичном дереве у лабиринта в верхнем и левом крае есть один длинный проход, а в лабиринте Sidewinder есть только один длинный проход по верхнему краю. Как и лабиринт двоичного дерева, лабиринт Sidewinder можно без ошибок и детерминировано решить снизу вверх, потому что в каждой строке всегда будет ровно один проход, ведущий вверх. Решение лабиринта Sidewinder никогда не делает петель и не посещает одну строку больше одного раза, однако оно «извивается из стороны в сторону». Единственный тип ячеек, который не может существовать в лабиринте Sidewinder — это тупик с ведущим вниз проходом, потому что это будет противоречить правилу, что каждый проход, ведущий вверх, возвращает нас к началу.

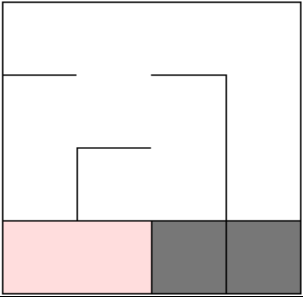
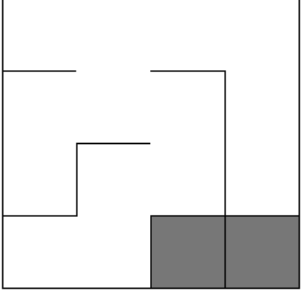
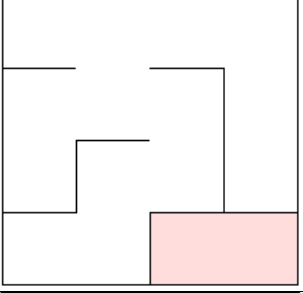
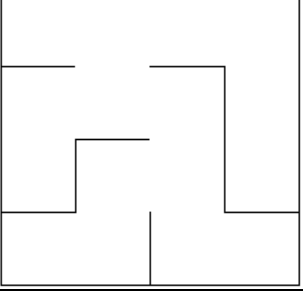
Лабиринты Sidewinder склонны к появлению элитного решения, при котором правильный путь оказывается прямым, но рядом с ним есть множество длинных «ошибочных» путей, идущих сверху вниз.

Таблица 11

Пример работы				
Сетка 4×4 .				
Начнем с первой (верхней) строки, хотя алгоритм также работает хорошо, если вы начнете с последней. Начиная сверху, исключаем один особый случай: весь первый ряд должен быть одним проходом (потому что вы не можете вырезать север от самого северного ряда).				

Пример работы			
Допустим, нам удастся прорезать 3/4 пути через ряд, прежде чем мы решим (случайным образом и произвольно) не продвигаться на восток.			
Ячейки желтого цвета являются текущим «набором пробежек» <i>Run</i> . Так как мы закончили текущий пробег горизонтальных ячеек, мы теперь выбираем одну из этих ячеек наугад и делим проход на север.			
Затем мы очищаем набор <i>Run</i>			
Выбираем следующую ячейку в качестве текущей:			
<p>И попробуем решить, стоит ли резать на восток или нет. Тем не менее, на данный момент мы не можем разделить восток. Это вывело бы нас за пределы лабиринта, что запрещено. Таким образом, мы должны закончить пробежку и выбрать одну из ячеек в наборе, чтобы выехать на север. Решение не в наших руках: нам нужно выбрать только одну ячейку в наборе. Итак, мы высекаем на север.</p> <p>Затем мы начинаем следующий ряд. Обратите внимание, что каждая строка не зависит от предшествующей ей строки, очень похоже на алгоритм Эллера.</p>			

Пример работы			
Для следующего ряда мы решаем, что мы хотим прекратить вырезать восток в самой первой ячейке.			
Итак, остановимся, выберем ячейку из набора и уйдем на север.			
Продолжая, мы соединяем следующие две ячейки.			
Случайно выбираем одну из них, чтобы высечь север.			
И так как мы находимся в конце ряда, мы снова вынуждены остановиться и уйти на север.			

Пример работы			
Для последнего ряда, мы соединяем первые две ячейки.			
Вырезаем север и очистим набор.			
Затем соединяем последние две ячейки:			
И завершим это, добавив окончательное северное соединение:			

1.1.12 Примеры лабиринтов

Следующие лабиринты сгенерированы алгоритмами генерации идеальных лабиринтов файла-программы `perfect_maze.py` (см. раздел «Код программы»).

Таблица 12

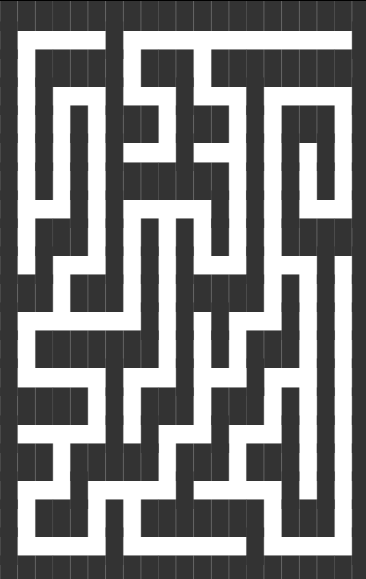
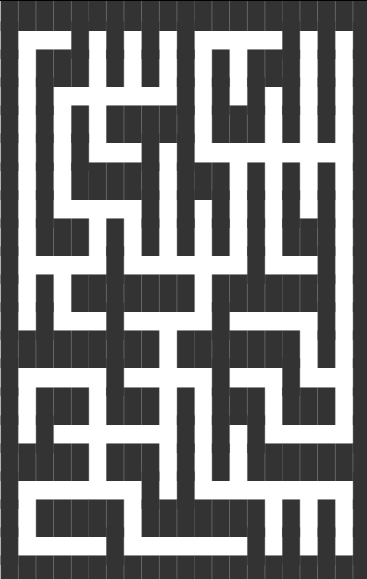
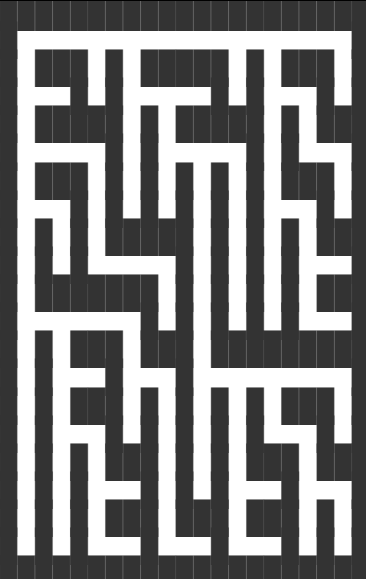
Алгоритм Олдоса–Бродера	Алгоритм Уилсона	Алгоритм двоичного дерева
		

Таблица 13

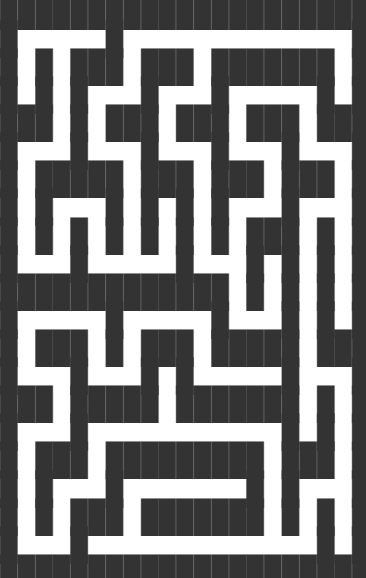
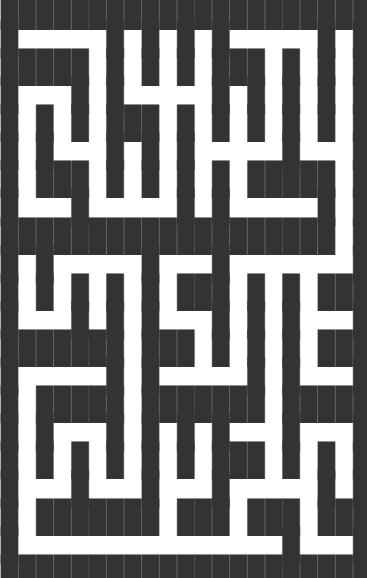
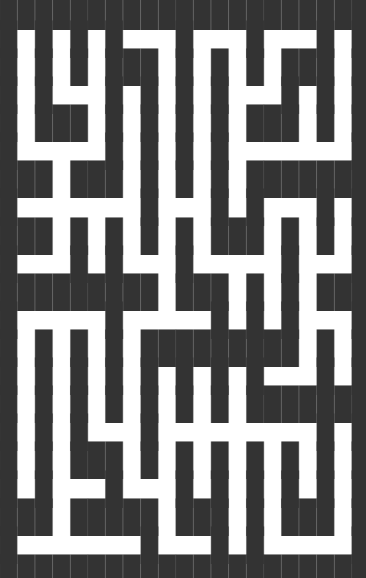
Алгоритм Recursive Backtracking	Алгоритм рекурсивного деления	Алгоритм Эллера
		

Таблица 14

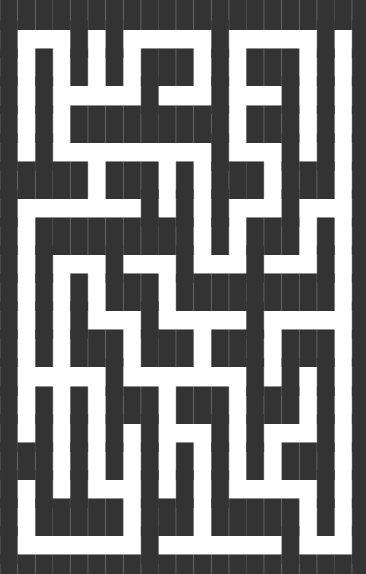
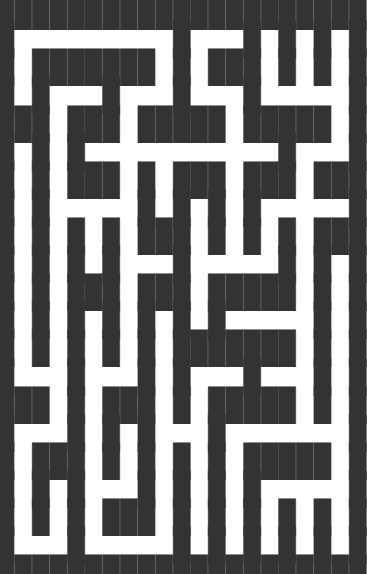
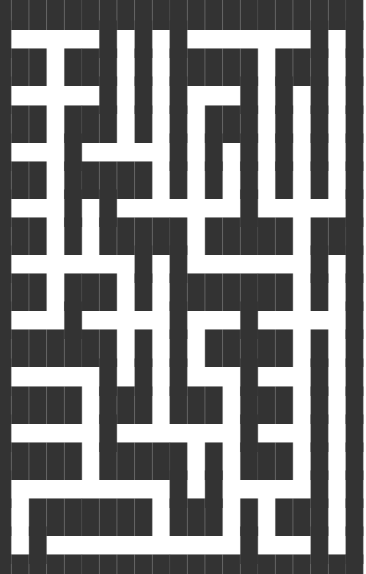
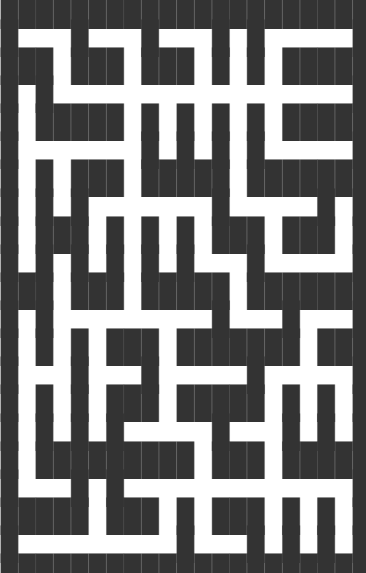
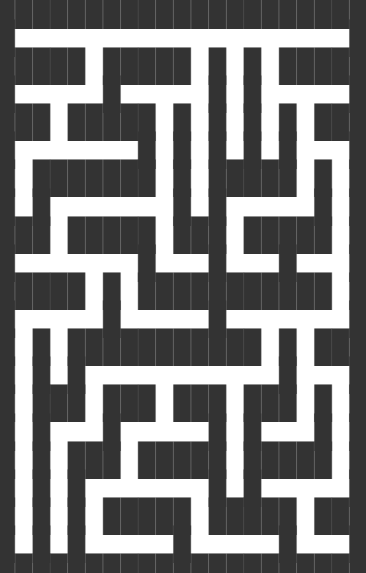
Алгоритм растущего дерева	Алгоритм Краскала	Алгоритм Прима (упрощённый)
		

Таблица 15

Алгоритм Прима (модифицированный)	Алгоритм Sidewinder
	

1.2 Алгоритмы генерации неидеальных лабиринтов

Неидеальный лабиринт — лабиринт с петлями и (возможно) с недостижимыми областями. Число путей между двумя произвольными ячейками лабиринта может быть любым. Это также значит, что путь между ними может отсутствовать.

Таблица 16. Алгоритмы генерации неидеальных лабиринтов

Алгоритм	% тупиков	Фокус	Отсутствует смещенность	Однородность	Память
Алгоритм змеевидного лабиринта <i>Решение: 26%</i>	≈ 0	Проходы	Нет	Никогда	0
Алгоритм маленьких комнат <i>Решение: 2%</i>	39	Проходы	Да	Никогда	0
Алгоритм спирального лабиринта <i>Решение: 2%</i>	≈ 0	Проходы	Нет	Никогда	C

1.2.1 Алгоритм змеевидного лабиринта

Алгоритм змеевидного лабиринта — алгоритм генерации неидеального лабиринта с циклами без недостижимых областей.

Алгоритм:

1. Изначально все поле содержит стены.
2. Выбираем направление движения (вверх – вправо – вниз – вправо / влево – вниз – вправо – вниз).
3. В соответствии с направлением выбираем исходную ячейку (нижняя левая / верхняя правая).
4. Передвигаемся по лабиринту в первую сторону, прорезая стены.
5. Если достигли одного из краёв лабиринта:
 - Если ячейка находится в нижнем правом углу, то генерация завершена, переход к №6.
 - Делаем два шага по второй стороне.
 - Передвигаемся по лабиринту в третью сторону, прорезая стены.
 - Если достигли одного из краёв лабиринта:
 - Если ячейка находится в правом нижнем углу, то генерация завершена.
 - Сделаем два шага по четвертой стороне.
 - Переходим к №4.

6. Прорезаем случайное количество стен в лабиринте.

Алгоритм тривиален, поэтому пример работы не приводится. Увидеть пример такого лабиринта можно в «1.2.4. Примеры лабиринтов».

1.2.2 Алгоритм маленьких комнат

Алгоритм связанных маленьких комнат — алгоритм генерации неидеального лабиринта с циклами без недостижимых областей.

Алгоритм:

1. Изначально все поле содержит стены, стартовая ячейка (0, 0).
2. Прорезаем две стены между текущей и следующей, и следующей и последующей ячейками.
3. Прорезаем проход вниз у следующей ячейки.
4. Смещаемся на 4 ячейки вперёд от текущей.
5. Повторяем №2-4 пока не будет достигнут край.
6. Переходим на 2 строки вниз, к левой стороне лабиринта.
7. Повторяем №5-6 пока не будет достигнут нижний край.
8. Прорезаем случайное количество стен в лабиринте.

Алгоритм тривиален, поэтому пример работы не приводится. Увидеть пример такого лабиринта можно в «1.2.4. Примеры лабиринтов».

1.2.3 Алгоритм спирального лабиринта

Алгоритм спирального лабиринта — алгоритм генерации неидеального лабиринта с циклами без недостижимых областей.

Алгоритм:

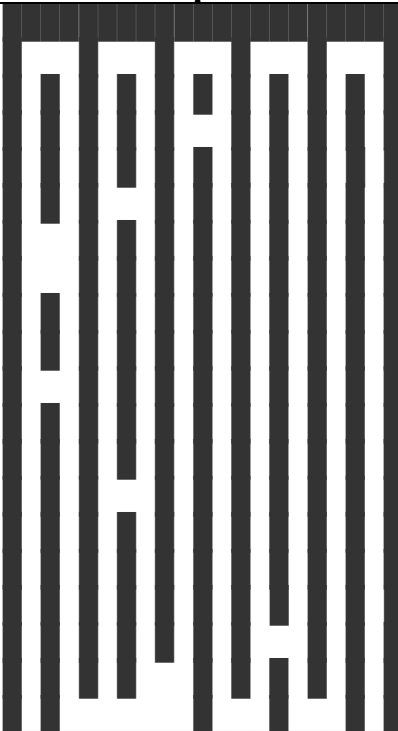
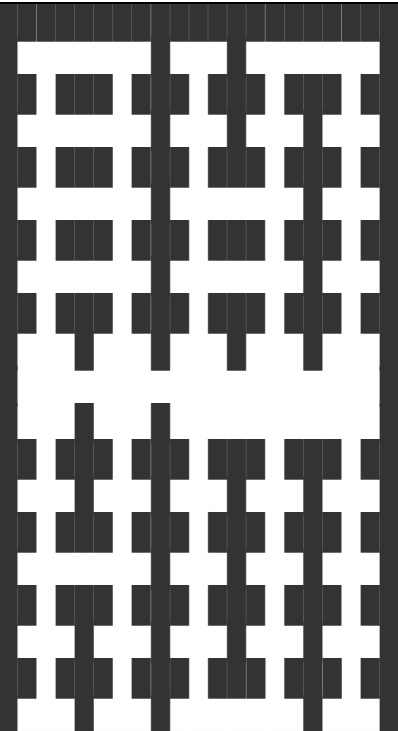
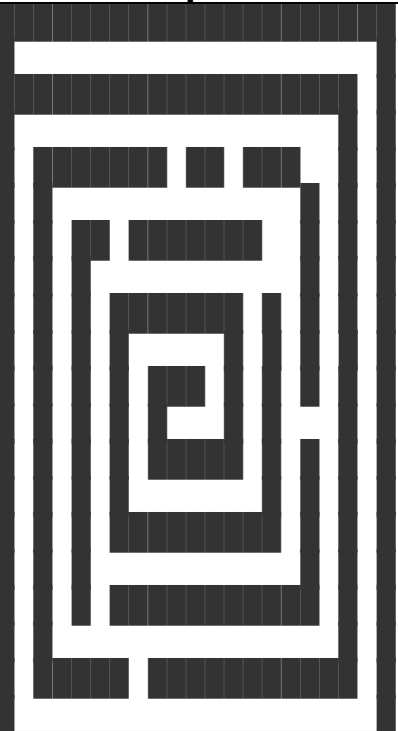
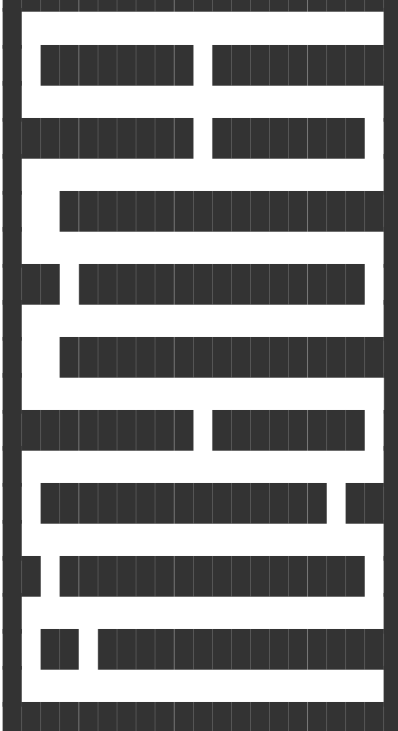
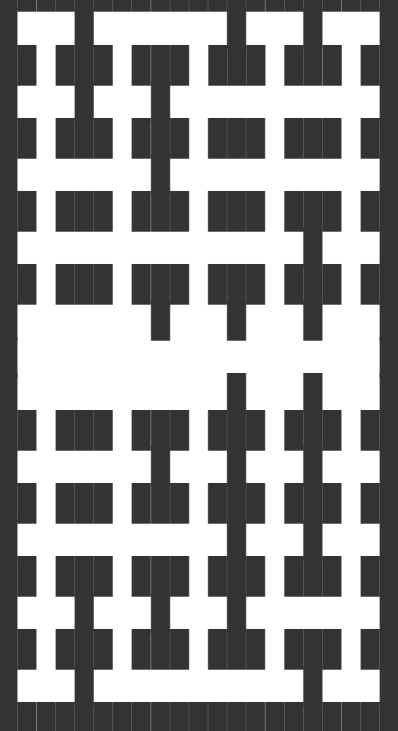
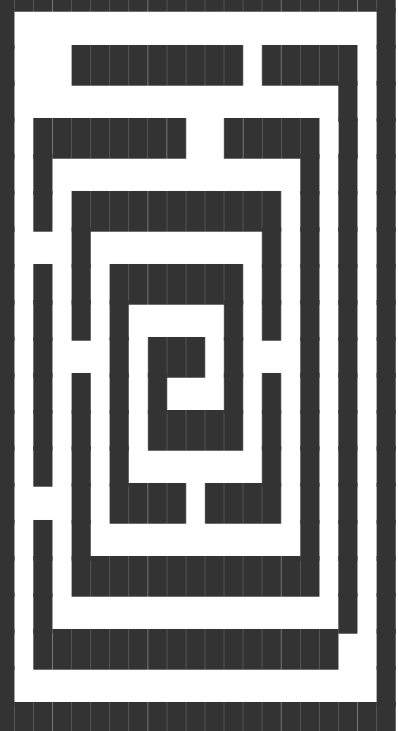
1. Изначально все поле содержит стены, стартовая ячейка (0, 0).
2. Передвигаемся по лабиринту «вправо», прорезая стены.
 - Если все соседи текущей ячейки были обработаны, то генерация завершена, переходим к №6.
 - Если достигли края лабиринта или стены со следующей уже обработанной ячейкой, сменяем направление на «вниз».
3. Передвигаемся по лабиринту «вниз», прорезая стены.
 - Если все соседи текущей ячейки были обработаны, то генерация завершена, переходим к №6.
 - Если достигли края лабиринта или стены со следующей уже обработанной ячейкой, сменяем направление на «влево».
4. Передвигаемся по лабиринту «влево», прорезая стены.
 - Если все соседи текущей ячейки были обработаны, то генерация завершена, переходим к №6.
 - Если достигли края лабиринта или стены со следующей уже обработанной ячейкой, сменяем направление на «вверх».
5. Передвигаемся по лабиринту «вверх», прорезая стены.
 - Если все соседи текущей ячейки были обработаны, то генерация завершена, переходим к №6.
 - Если достигли края лабиринта или стены со следующей уже обработанной ячейкой, переходим к №2.
6. Прорезаем случайное количество стен в лабиринте.

Алгоритм тривиален, поэтому пример работы не приводится. Увидеть пример такого лабиринта можно в «1.2.4. Примеры лабиринтов».

1.2.4 Примеры лабиринтов

Следующие лабиринты сгенерированы алгоритмами генерации неидеальных лабиринтов файла-программы `imperfect_maze.py` (см. раздел «Код программы»).

Таблица 17

Алгоритм змеевидного лабиринта	Алгоритм маленьких комнат	Алгоритм спирального лабиринта
		
		

2. ПОИСК КРАТЧАЙШЕГО ПУТИ В ЛАБИРИНТАХ

2.1 Алгоритм итеративного поиска в ширину

Алгоритм итеративного поиска в ширину для матрицы — алгоритм поиска кратчайшего пути в матрице (лабиринте).

Алгоритм:

1. Пусть S — стартовая ячейка, E — конечная ячейка, $parentNode$ — пустой ассоциативный массив родителей, Q — очередь.
2. Начинаем со стартовой ячейки S .
3. Добавляем в $parentNode$ стартовую ячейку с ассоциативным значением самой стартовой ячейки. $parentNode[S] = S$.
4. Добавляем в очередь Q стартовую ячейку S .
5. Пока очередь Q не пуста:
 - Берём элемент из очереди Q (с удалением) — текущий элемент.
 - Если он является конечной ячейкой E , то выходим из цикла.
 - Для каждого соседа $neighbor$ вокруг текущей ячейки:
 - Если сосед не ограждён стеной и не был посещён ранее (его нет в $parentNode$), то добавляем его в очередь Q , а также добавляем его в $parentNode$ в качестве ключа со значением текущей ячейки $parentNode[neighbor] = current$.
6. Пусть $path$ — путь от начальной к конечной ячейке (т. е. список ячеек).
7. Если текущая ячейка $current$ является конечной ячейкой E , то
 - Добавляем в $path$ текущую ячейку $current$.
 - Пока текущая ячейка $current$ не является начальной ячейкой:
 - Текущей ячейкой $current$ становится $parentNode[current]$.
 - Добавляем в $path$ текущую ячейку $current$.
8. Возвращаем $path$ в реверсивном (обратном) порядке.

Причём: алгоритм можно упростить так, что реверс $path$ в конце не потребуется: следует начать не со стартовой ячейки, а с конечной, и идти к стартовой.

Свойства алгоритма:

- Эвристики отсутствуют: может искать там, где не нужно.
- Поиск происходит методом полного обхода в ширину.

Примеры работы алгоритма можно посмотреть в «2.4 Примеры поиска».

2.2 Алгоритм Дейкстры

Алгоритм Дейкстры для матрицы — алгоритм поиска кратчайшего пути в битовой матрице (лабиринте).

Алгоритм:

1. Пусть S — стартовая ячейка, E — конечная ячейка, $parentNode$ — пустой ассоциативный массив родителей, $costMap$ — пустой ассоциативный массив стоимостей, PQ — приоритетная очередь (по возрастанию стоимостей).
2. Начинаем со стартовой ячейки S .
3. Добавляем в $parentNode$ стартовую ячейку с ассоциативным значением самой стартовой ячейки. $parentNode[S] = S$.
4. Добавляем в $costMap$ стартовую ячейку с ассоциативным значением 0.
5. Добавляем в очередь PQ стартовую ячейку S с приоритетным значением 0.
6. Пока очередь PQ не пуста:

- Берём элемент из очереди PQ (с удалением) — текущий элемент.
- Если он является конечной ячейкой E , то выходим из цикла.
- Новая стоимость $cost = costMap[current] + 1$.
- Для каждого соседа $neighbor$ вокруг текущей ячейки:
 - Если сосед не ограждён стеной и стоимость $cost$ меньше стоимости $costMap[neighbor]$ (или $costMap$ не содержит $neighbor$), то устанавливаем $costMap[neighbor]$ равным $cost$, обновляем соседа в $parentNode$ в качестве ключа со значением текущей ячейки $parentNode[neighbor] = current$, добавляем соседа в очередь PQ с приоритетным значением $cost$.

7. Пусть $path$ — путь от начальной к конечной ячейке (т. е. список ячеек).

8. Если текущая ячейка $current$ является конечной ячейкой E , то

- Добавляем в $path$ текущую ячейку $current$.
- Пока текущая ячейка $current$ не является начальной ячейкой:
 - Текущей ячейкой $current$ становится $parentNode[current]$.
 - Добавляем в $path$ текущую ячейку $current$.

9. Возвращаем $path$ в реверсивном (обратном) порядке.

Причём: алгоритм можно упростить так, что реверс $path$ в конце не потребуется: следует начать не со стартовой ячейки, а с конечной, и идти к стартовой.

Свойства алгоритма:

- Эвристики отсутствуют: может искать там, где не нужно.
- Поиск происходит методом полного обхода в ширину (с дополнительными затратами), так как задано независимое отношение $cost = costMap[current] + 1$. Вместо этого можно сделать зависимое отношение $cost = costMap[current] + f_{cost}(current, neighbor)$, где f_{cost} — функция, возвращающая неотрицательную стоимость между первой $current$ и второй $neighbor$ ячейкой. Т. е. добавляются возможности по управлению поиском в соответствии с разными стоимостями для разных путей (может быть полезно, если работать с 2D лабиринтом как с 3D со спусками и подъёмами). Следовательно, при независимом отношении выигрыша по времени работы не будет по сравнению с временем работы алгоритма поиска в ширину.

Примеры работы алгоритма можно посмотреть в «2.4 Примеры поиска».

2.3 Алгоритм A*

Алгоритм A* для матрицы — алгоритм поиска кратчайшего пути в битовой матрице (лабиринте).

Алгоритм:

1. Пусть S — стартовая ячейка, E — конечная ячейка, $parentNode$ — пустой ассоциативный массив родителей, $costMap$ — пустой ассоциативный массив стоимостей, PQ — приоритетная очередь (по возрастанию стоимостей).
2. Начинаем со стартовой ячейки S .
3. Добавляем в $parentNode$ стартовую ячейку с ассоциативным значением самой стартовой ячейки. $parentNode[S] = S$.
4. Добавляем в $costMap$ стартовую ячейку с ассоциативным значением 0.
5. Добавляем в очередь PQ стартовую ячейку S с приоритетным значением 0.
6. Пока очередь PQ не пуста:

- Берём элемент из очереди PQ (с удалением) — текущий элемент.
- Если он является конечной ячейкой E , то выходим из цикла.
- Новая стоимость $cost = costMap[current] + 1$.
- Для каждого соседа $neighbor$ вокруг текущей ячейки:
 - Если сосед не ограждён стеной и стоимость $cost$ меньше стоимости $costMap[neighbor]$ (или $costMap$ не содержит $neighbor$), то устанавливаем $costMap[neighbor]$ равным $cost$, обновляем соседа в $parentNode$ в качестве ключа со значением текущей ячейки $parentNode[neighbor] = current$, добавляем соседа в очередь PQ с приоритетным значением $cost + heuristic(current, endNode)$, где $heuristic((y_1, x_1), (y_2, x_2))$ — функция эвристики, возвращающая абсолютное (неотрицательное) значение*.
- 7. Пусть $path$ — путь от начальной к конечной ячейке (т. е. список ячеек).
- 8. Если текущая ячейка $current$ является конечной ячейкой E , то
 - Добавляем в $path$ текущую ячейку $current$.
 - Пока текущая ячейка $current$ не является начальной ячейкой:
 - Текущей ячейкой $current$ становится $parentNode[current]$.
 - Добавляем в $path$ текущую ячейку $current$.
- 9. Возвращаем $path$ в реверсивном (обратном) порядке.

Причём: алгоритм можно упростить так, что реверс $path$ в конце не потребуется: следует начать не со стартовой ячейки, а с конечной, и идти к стартовой.

* $heuristic(current, endNode)$. Функция, которая возвращает абсолютное значение расстояния между текущей ячейкой $current$ и конечной ячейкой $endNode$.

Свойства алгоритма:

- Эвристики присутствуют: ищет там, где ближе всего может находиться конечная ячейка.
- Поиск происходит методом полного обхода в ширину с учётом стоимостей, так как задано зависимое отношение $cost = costMap[current] + 1 + heuristic(current, endNode)$. Т. е. помимо возможности по управлению поиском в соответствии с разными стоимостями для разных путей (может быть полезно, если работать с 2D лабиринтом как с 3D со спусками и подъёмами), добавляется возможность строго направленного поиска. Следовательно, выигрыш будет по сравнению с поиском в ширину только в случае неидеальных лабиринтов с большим количеством свободных ячеек.

Возможные эвристики:

- **Расстояние Евклида** (прямая линия).
Расстояние измеряется по прямой линии от начальной до конечной точки.
Аналог в шахматах: пешка (может ходить только в одну сторону).

$$\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$
- **Расстояние Манхэттена L1** (4 направления).
Расстояние измеряется 4 направлениями от начальной до конечной точки.
Аналог в шахматах: ладья (может ходить только по 4 направлениям).

$$|y_2 - y_1| + |x_2 - x_1|$$
- **Расстояние «Октиль»** (Octile) (диагонали).
Расстояние измеряется диагоналями от начальной до конечной точки.

Аналог в шахматах: слон (может ходить только по диагоналям).

$$\max(|y_2 - y_1|, |x_2 - x_1|) + (\sqrt{2} - 1) * \min(|y_2 - y_1|, |x_2 - x_1|)$$

- **Расстояние Чебышёва** (4 направления и диагонали).

Расстояние измеряется 4 направлениями и диагоналями (шахматными проходами) от начальной до конечной точки.

Аналог в шахматах: ферзь (может ходить по 4 направлениям и диагоналям).

$$\max(|y_2 - y_1|, |x_2 - x_1|)$$

Алгоритм A* имеет важное достоинство по сравнению с другими: можно задавать эвристическую функцию, которая может либо концентрироваться на прямолинейном пути (если какую-либо эвристику возвести в приемлемую степень), либо работать как поиск в ширину (возвращать -1 при любых входных данных) или алгоритм Дейкстры (возвращать 0 при любых входных данных), что даёт большие возможности для корректировки поиска.

Примеры работы алгоритма можно посмотреть в «2.4 Примеры поиска».

2.4 Примеры поиска

В соответствии с ранее описанными алгоритмами поиск происходит по четырём направлениям: влево, вверх, вправо, вниз. Стрелками указаны направления — от какой ячейки происходит поиск:

- < — влево (т. е. правая ячейка — родитель текущей ячейки);
- ^ — вверх (т. е. нижняя ячейка — родитель текущей ячейки);
- > — вправо (т. е. левая ячейка — родитель текущей ячейки);
- v — вниз (т. е. верхняя ячейка — родитель текущей ячейки).

Пусть S — стартовая ячейка, E — конечная ячейка.

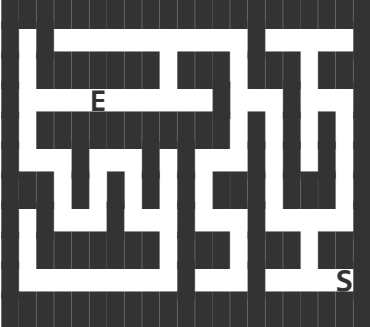
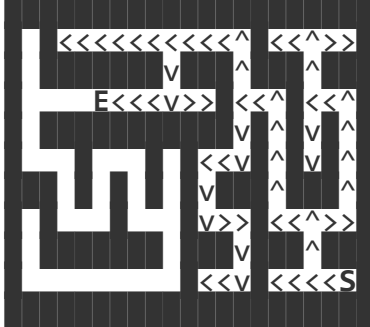
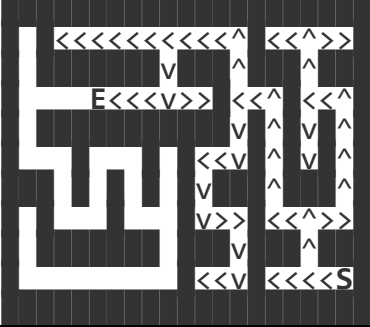
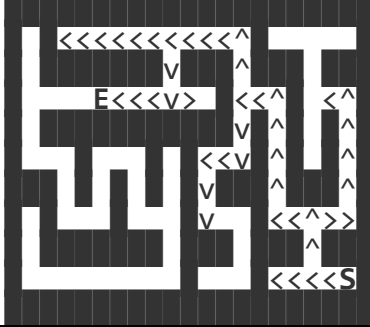
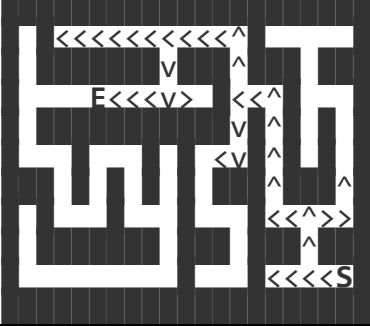
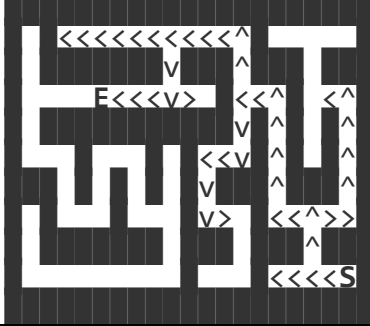
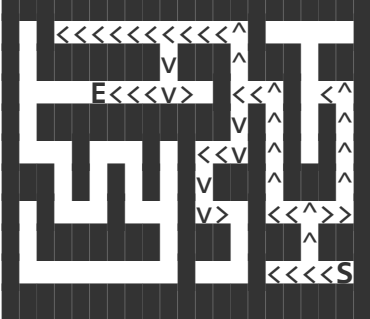
В следующих примерах показана сама суть поиска каждого алгоритма: какие ячейки затрагиваются, какие не затрагиваются. Чтобы найти кратчайший путь, достаточно двигаться от E к S по противоположным направлениям.

Таблица 18

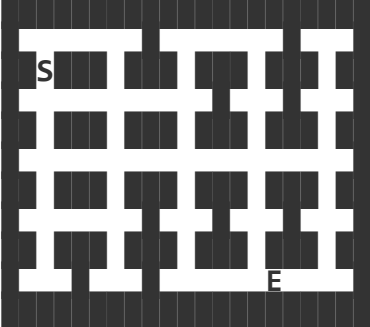
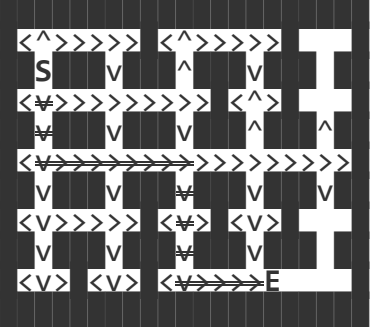
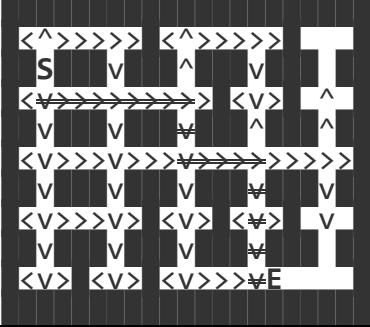
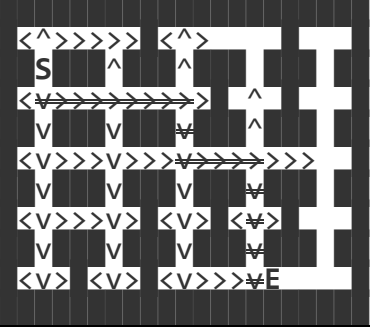
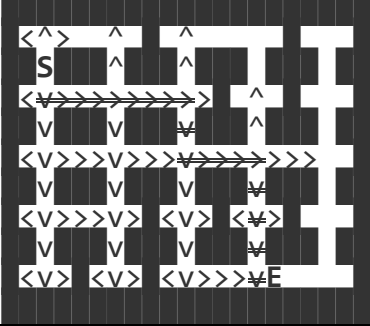
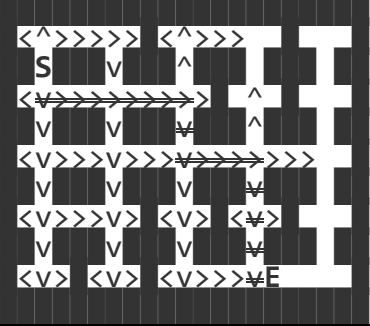
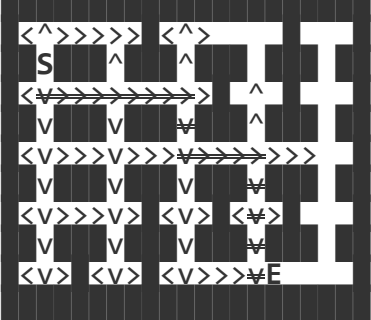
Исходный лабиринт	Итеративный поиск в ширину
Алгоритм Дейкстры	Алгоритм A* (октильное расстояние)
Алгоритм A* (расстояние L1)	Алгоритм A* (расстояние Чебышева)
Алгоритм A* (расстояние Евклида)	

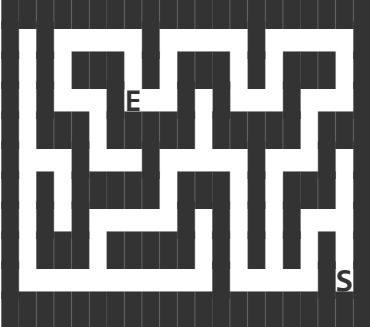
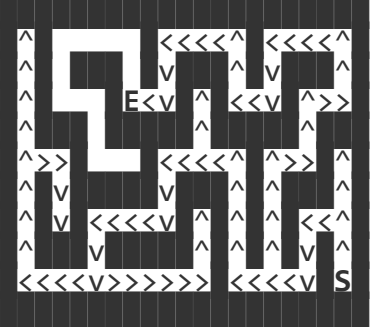
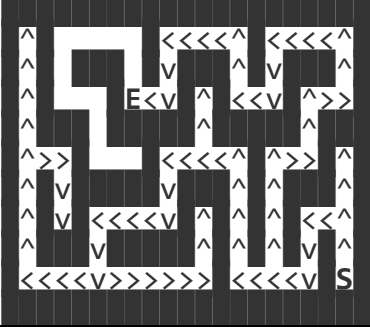
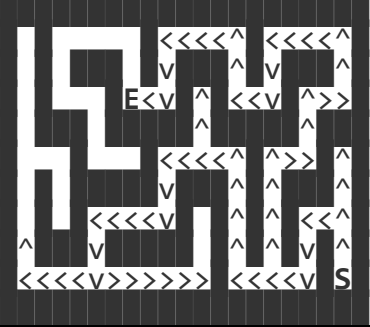
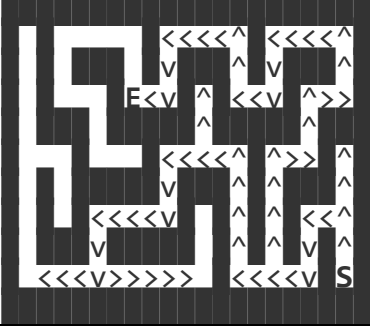
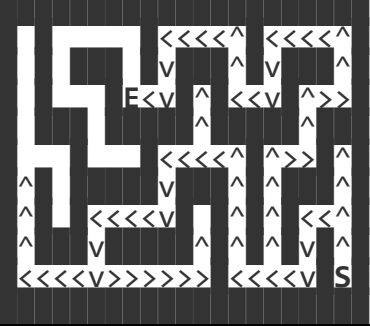
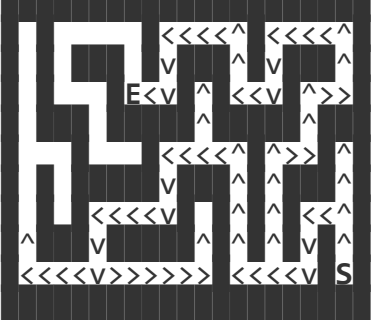
Как видим, длина пути (17) одинаковая, но сами пути могут быть разными.

Исходный лабиринт	Итеративный поиск в ширину
Алгоритм Дейкстры	Алгоритм A* (октильное расстояние)
Алгоритм A* (расстояние L1)	Алгоритм A* (расстояние Чебышева)
Алгоритм A* (расстояние Евклида)	
<p>Найденный путь (y, x): (3, 13), (3, 12), (3, 11), (2, 11), (1, 11), (1, 10), (1, 9), (1, 8), (1, 7), (2, 7), (3, 7), (4, 7), (5, 7), (6, 7), (7, 7), (7, 6), (7, 5), (8, 5), (9, 5), (9, 4), (9, 3), (9, 2), (9, 1).</p> <p>Длина пути: 23.</p>	

Исходный лабиринт	Итеративный поиск в ширину
	
Алгоритм Дейкстры	Алгоритм A* (октильное расстояние)
	
Алгоритм A* (расстояние L1)	Алгоритм A* (расстояние Чебышева)
	
Алгоритм A* (расстояние Евклида)	
	
<p>Найденный путь (y, x): (9, 19), (9, 18), (9, 17), (8, 17), (7, 17), (7, 16), (7, 15), (6, 15), (5, 15), (4, 15), (3, 15), (3, 14), (3, 13), (2, 13), (1, 13), (1, 12), (1, 11), (1, 10), (1, 9), (2, 9), (3, 9), (3, 8), (3, 7), (3, 6), (3, 5).</p> <p>Длина пути: 25.</p>	

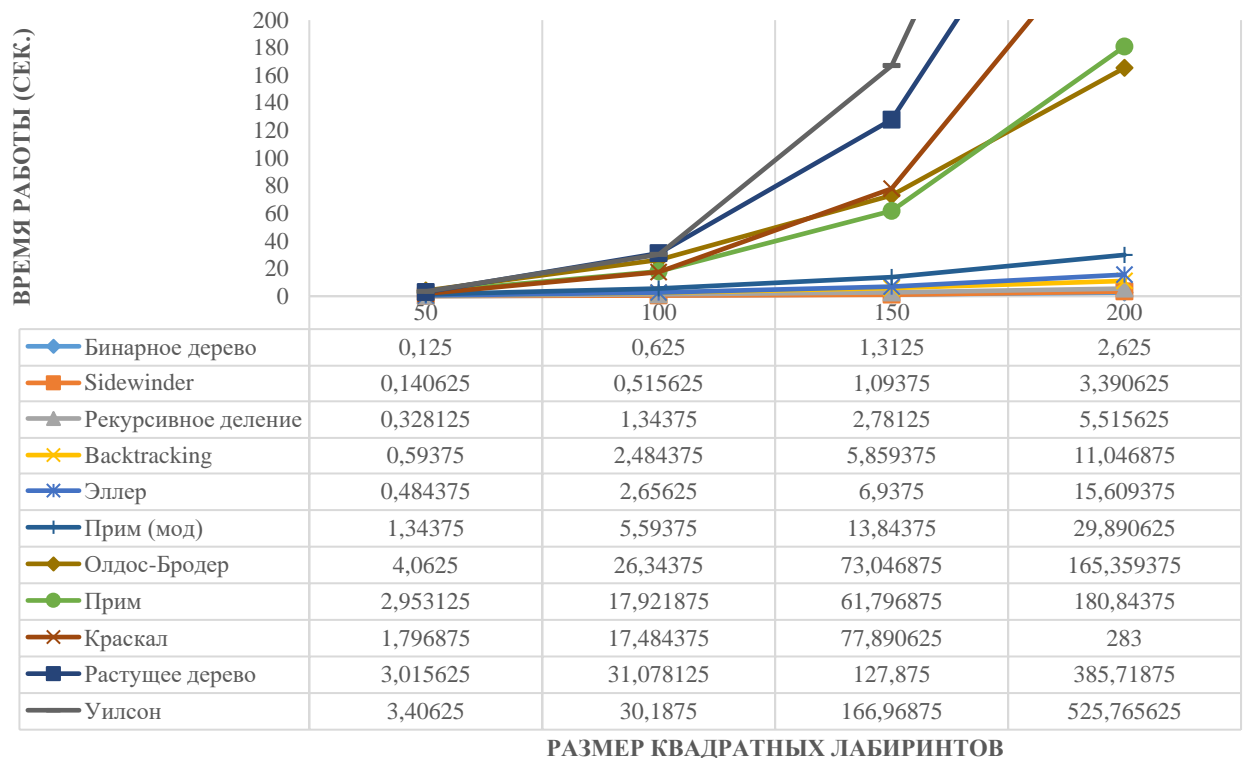
Исходный лабиринт	Итеративный поиск в ширину
Алгоритм Дейкстры	Алгоритм A* (октильное расстояние)
Алгоритм A* (расстояние L1)	Алгоритм A* (расстояние Чебышева)
Алгоритм A* (расстояние Евклида)	
<p>Найденный путь (y, x): (3, 4), (3, 5), (4, 5), (5, 5), (5, 6), (5, 7), (6, 7), (7, 7), (7, 8), (7, 9), (7, 10), (7, 11), (7, 12), (7, 13), (7, 14), (7, 15), (7, 16), (7, 17), (8, 17), (9, 17).</p> <p>Длина пути: 20.</p>	

Исходный лабиринт	Итеративный поиск в ширину
	
Алгоритм Дейкстры	Алгоритм A* (октильное расстояние)
	
Алгоритм A* (расстояние L1)	Алгоритм A* (расстояние Чебышева)
	
Алгоритм A* (расстояние Евклида)	
	
<p>Как видим, длина пути (21) одинаковая, но сами пути могут быть разными.</p> <p>Кратчайший путь поиска в ширину (y, x): (2, 2), (3, 2), (4, 2), (5, 2), (5, 3), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (6, 10), (7, 10), (8, 10), (9, 10), (9, 11), (9, 12), (9, 13), (9, 14), (9, 15).</p> <p>Кратчайший путь остальных алгоритмов (y, x): (2, 2), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (3, 10), (4, 10), (5, 10), (5, 11), (5, 12), (5, 13), (5, 14), (6, 14), (7, 14), (8, 14), (9, 14), (9, 15).</p>	

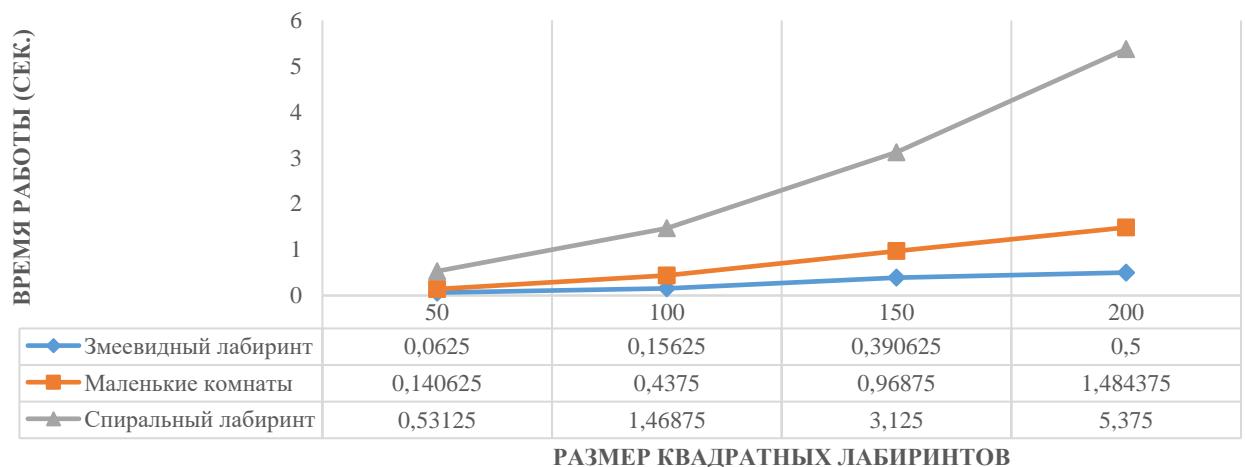
Исходный лабиринт	Итеративный поиск в ширину
	
Алгоритм Дейкстры	Алгоритм A* (октильное расстояние)
	
Алгоритм A* (расстояние L1)	Алгоритм A* (расстояние Чебышева)
	
Алгоритм A* (расстояние Евклида)	
	
<p>Найденный путь (y, x): (9, 19), (8, 19), (7, 19), (7, 18), (7, 17), (8, 17), (9, 17), (9, 16), (9, 15), (8, 15), (7, 15), (6, 15), (5, 15), (5, 16), (5, 17), (4, 17), (3, 17), (3, 18), (3, 19), (2, 19), (1, 19), (1, 18), (1, 17), (1, 16), (1, 15), (2, 15), (3, 15), (3, 14), (3, 13), (2, 13), (1, 13), (1, 12), (1, 11), (1, 10), (1, 9), (2, 9), (3, 9), (3, 8), (3, 7)</p> <p>Длина пути: 39.</p>	

3. РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

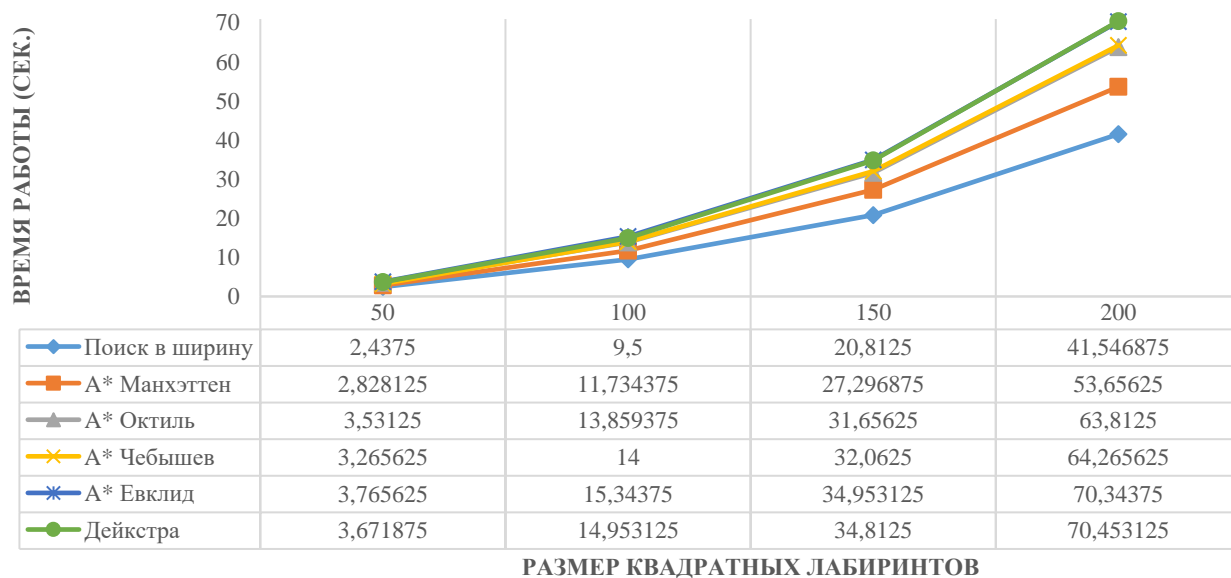
ВРЕМЯ РАБОТЫ АЛГОРИТМОВ ГЕНЕРАЦИИ ИДЕАЛЬНЫХ ЛАБИРИНТОВ КОЛИЧЕСТВО ТЕСТОВ: 100



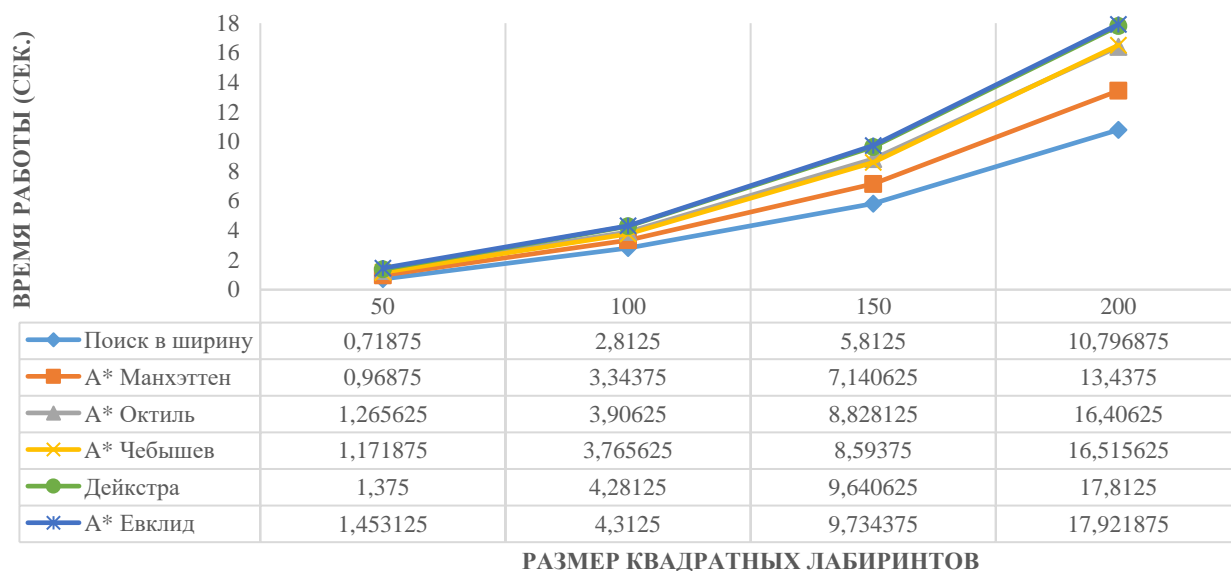
ВРЕМЯ РАБОТЫ АЛГОРИТМОВ ГЕНЕРАЦИИ НЕИДЕАЛЬНЫХ ЛАБИРИНТОВ КОЛИЧЕСТВО ТЕСТОВ: 100



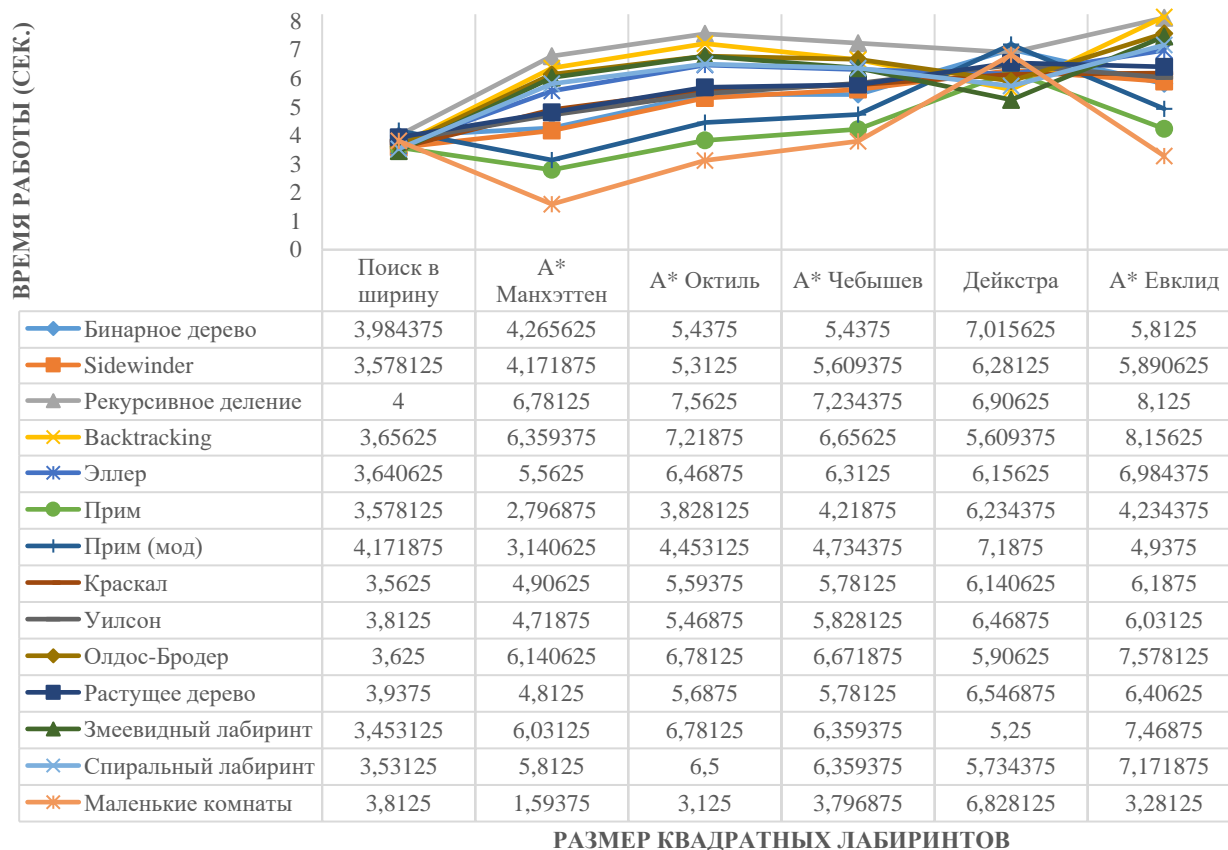
ВРЕМЯ РАБОТЫ АЛГОРИТМОВ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ В ИДЕАЛЬНЫХ ЛАБИРИНТАХ КОЛИЧЕСТВО ТЕСТОВ: 100



ВРЕМЯ РАБОТЫ АЛГОРИТМОВ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ В НЕИДЕАЛЬНЫХ ЛАБИРИНТАХ КОЛИЧЕСТВО ТЕСТОВ: 100



ВРЕМЯ РАБОТЫ АЛГОРИТМОВ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ В ЛАБИРИНТАХ РАЗМЕРА 200 КОЛИЧЕСТВО ТЕСТОВ: 100



Параметры тестирования

Язык: Python 3.7.6 (default) [MSC v.1916 64 bit (AMD64)] on win32.

Операционная система: Microsoft © Windows 8.1 для одного языка.

Комментарий

Как можно заметить, алгоритм Олдоса – Бродера значительно обошел по скорости алгоритм Уилсона на больших размерах лабиринтов (при 50 не обошел). Это связано:

1. С хорошим распределением псевдослучайных чисел генератора (в случае алгоритма Олдоса – Бродера).
2. Со значительными накладными расходами на память (в случае алгоритма Уилсона), в том числе на работу с памятью посредством самого языка.

Также можно заметить, что алгоритм поиска в ширину быстрее всех остальных в большинстве случаев, хотя A* имеет хорошие эвристики. Это связано:

1. С простотой работы самого алгоритма поиска в ширину.
2. С накладными расходами на память (в случае A* и Дейкстры) и на работу с памятью посредством самого языка.
3. С узкоспециализированными тестами. В случае с сеткой, имеющей гораздо большее количество свободных ячеек, накладные расходы A* будут гораздо меньше по сложности, чем обход в ширину алгоритмом поиска в ширину.

4. ВЫВОДЫ

В результате проделанной работы были проанализированы и реализованы:

- Алгоритмы создания идеальных лабиринтов (алгоритм бинарного дерева, алгоритм Sidewinder, алгоритм рекурсивного деления, алгоритм Эллера, алгоритм Recursive Backtracking, алгоритм Прима, модифицированный алгоритм Прима, алгоритм Краскала, алгоритм растущего дерева, алгоритм Олдоса–Бродера, алгоритм Уилсона).
- Алгоритмы создания неидеальных лабиринтов (алгоритм змеевидного лабиринта, алгоритм маленьких комнат, алгоритм спирального лабиринта).
- Алгоритмы поиска кратчайшего пути в лабиринтах (алгоритм поиска в ширину, алгоритм Дейкстры, алгоритм A*).

Все алгоритмы успешно справились со своей задачей.

Далее приведены распределения мест на основе результатов.

Таблица 24. Результаты

Алгоритмы создания идеальных лабиринтов (отсортированы по увеличению времени выполнения)	Алгоритмы создания неидеальных лабиринтов (отсортированы по увеличению времени выполнения)	Алгоритмы поиска кратчайших путей (отсортированы по кол-ву возможностей настройки поиска)
Sidewinder	Алгоритм змеевидного лабиринта	A* (расстояние Манхэттена) / A* (расстояние «Octile»)
Алгоритм бинарного дерева		
Алгоритм рекурсивного деления		
Алгоритм Recursive Backtracking	Алгоритм маленьких комнат	/ A* (расстояние Чебышёва) / A* (расстояние Евклида)
Алгоритм Эллера		
Алгоритм Прима (модифицированный)		
Алгоритм Олдоса – Бродера		
Алгоритм Прима	Алгоритм спирального лабиринта	Алгоритм Дейкстры
Алгоритм Краскала		
Алгоритм растущего дерева		Поиск в ширину
Алгоритм Уилсона.		

5. КОД ПРОГРАММЫ

Далее в таблицах содержится код файлов программы:

- **test_maze.py. Главный файл (файл запуска).**
Данный файл запускается для тестирования всех алгоритмов.
Тип окна: окно командной строки интерпретатора.
Ввод: с клавиатуры.
Вывод: в командную строку интерпретатора.
Как работает: вводится количество тестов, минимальные и максимальные размеры лабиринтов для алгоритмов генерации идеальных и неидеальных лабиринтов, выводятся результаты тестирования алгоритмов генерации и алгоритмов поиска кратчайших путей.
- **perfect_maze.py. Содержит алгоритмы генерации идеальных лабиринтов.**
Данный файл может запускаться для отображения примеров идеальных лабиринтов и тестирования на скорость алгоритмов генерации идеальных лабиринтов.
Тип окна: окно командной строки интерпретатора.
Ввод: с клавиатуры,
Вывод: в командную строку интерпретатора.
Как работает: вводятся размеры лабиринта, далее выводятся сгенерированные лабиринты всех алгоритмов файла для этих размеров, потом вводится количество тестов, минимальные и максимальные размеры лабиринтов для тестирования на скорость алгоритмов генерации лабиринтов.
- **imperfect_maze.py. Содержит алгоритмы генерации неидеальных лабиринтов.**
Данный файл может запускаться для отображения примеров неидеальных лабиринтов и тестирования на скорость алгоритмов генерации неидеальных лабиринтов.
Тип окна: окно командной строки интерпретатора.
Ввод: с клавиатуры,
Вывод: в командную строку интерпретатора.
Как работает: вводятся размеры лабиринта, далее выводятся сгенерированные лабиринты всех алгоритмов файла для этих размеров, потом вводится количество тестов, минимальные и максимальные размеры лабиринтов для тестирования на скорость алгоритмов генерации лабиринтов.
- **solve_maze.py. Содержит алгоритмы поиска кратчайших путей в лабиринтах.**
Данный файл не следует запускать напрямую (ничего не будет происходить).

Все файлы необходимо располагать в одной папке.

Пример того, что может выводить файл-программа test_maze.py, можно посмотреть в таблице вывода после кода всех четырёх файлов.

test_maze.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
    Testing shortest path search algorithms in mazes
    2D perfect maze generation algorithms:
    - Aldous-Broder algorithm (unbiased maze)
    - Wilson's algorithm (unbiased maze)
    - Backtracking (depth-first search, iterative version)
    - Binary tree algorithm
    - Division algorithm
    - Eller's algorithm
    - Growing tree algorithm
    - Kruskal's algorithm
    - Prim's algorithm
    - Prim's algorithm (modified)
    - Sidewinder algorithm
    2D imperfect maze generation algorithms:
    - Serpentine algorithm
    - Small rooms algorithm
    - Spiral algorithm
    Algorithms for finding the shortest paths in 2D mazes:
    - A* algorithm (4 heuristics)
    - BFS (breadth-first search) (iterative version)
    - Dijkstra algorithm
"""
from imperfect_maze import *
from perfect_maze import *
from solve_maze import *

import gc as garbage_collector
from random import randint, randrange
from time import process_time
from typing import Callable, List

def test_algorithms(n_tests: int, min_length: int, max_length: int, maze_functions: List[Callable],
                    maze_n_functions: List[str], solve_maze_functions: List[Callable],
                    solve_maze_n_functions: List[str], *, is_perfect_mazes: bool):
    maze_time_functions = [0 for _ in maze_functions]
    solve_maze_time_functions = [[0 for _ in maze_n_functions] for _ in solve_maze_n_functions]
    for i in range(n_tests):
        y_size, x_size = (randint(min_length, max_length) // 2) * 2 + 1, (
            randint(min_length, max_length) // 2) * 2 + 1
        for j, func in enumerate(maze_functions):
            garbage_collector.disable()
            maze_time_functions[j] -= process_time()
            maze = func(y_size, x_size)
            maze_time_functions[j] += process_time()
            garbage_collector.enable()
            (yFrom, xFrom, yTo, xTo) = 0, 0, 0, 0
            while maze[yFrom][xFrom] or maze[yTo][xTo] or (yFrom, xFrom) == (yTo, xTo):
                yFrom = randrange(y_size)
                xFrom = randrange(x_size)
                yTo = randrange(y_size)
                xTo = randrange(x_size)
            solve_results = []
            for k, solve_func in enumerate(solve_maze_functions):
                garbage_collector.disable()
                solve_maze_time_functions[k][j] -= process_time()
                result = solve_func(maze, (yFrom, xFrom), (yTo, xTo))
                solve_maze_time_functions[k][j] += process_time()
                garbage_collector.enable()
                solve_results.append(result)
            if is_perfect_mazes and not solve_results[k]:
                print('Invalid answer or imperfect maze')
                print(*[''.join('█' if col else ' ' for col in row) for row in maze], sep='\n')
                print((yFrom, xFrom), (yTo, xTo))
```

test_maze.py

```

        print(solve_maze_n_functions[k], solve_results[k])
        raise AttributeError
    if is_perfect_mazes and k and solve_results[k - 1] != solve_results[k] \
        or not is_perfect_mazes and len(solve_results[k - 1]) != len(solve_results[k]):
        print('Miscellaneous answers')
        print(*['.join('█' if col else ' ' for col in row) for row in maze], sep='\n')
        print((yFrom, xFrom), (yTo, xTo))
        print(solve_maze_n_functions[k - 1], solve_results[k - 1])
        print(solve_maze_n_functions[k], solve_results[k])
        raise AttributeError

    print('Time of execution of maze generation algorithms')
    for time, n in sorted(zip(maze_time_functions, maze_n_functions)):
        print(n + ': ' + str(time))
    print('\n' + 'Time of execution of algorithms to find the shortest paths in the mazes (briefly)')
    for time, sn in sorted(zip(solve_maze_time_functions, solve_maze_n_functions), key=lambda x:
sum(x[0])):
        print(sn + ': ' + str(sum(time)))
    print('\n' + 'Time of execution of algorithms to find the shortest paths in the mazes (in detail)')
    for time, sn in sorted(zip(solve_maze_time_functions, solve_maze_n_functions), key=lambda x:
sum(x[0])):
        print('#' * 30 + '\n' + 'Function:', sn)
        for n, t in sorted(zip(maze_n_functions, time), key=lambda x: x[1]):
            print(n + ': ' + str(t))

def main():
    def get_input(string, begin, end):
        string += f" ({begin}-{end}):"
        input_result = input(string)
        while not (input_result.isdecimal() and begin <= int(input_result) <= end):
            input_result = input(string)
        return int(input_result)

    print('=== Perfect algorithms ===')
    n_tests = get_input('Enter the number of tests', 1, 100000)
    min_length = get_input('Enter the minimum length of the maze (odd number)', 5, 100000)
    max_length = get_input('Enter the maximum length of the maze (odd number)', 5, 100000)
    maze_functions = [aldous_broder, wilson, backtracking, binary_tree, division, eller, growing_tree,
kruskal,
                    prim, modified_prim, sidewinder]
    maze_n_functions = ['Aldous-Broder algorithm', 'Wilson\'s algorithm', 'Backtracking algorithm',
                        'Binary tree algorithm', 'Division algorithm', 'Eller\'s algorithm', 'Growing tree
algorithm',
                        'Kruskal\'s algorithm', 'Prim\'s algorithm', 'Prim\'s algorithm (modified)',
                        'Sidewinder algorithm']
    solve_maze_functions = [bfs, dijkstra,
                            lambda mz, begin, end: a_star(mz, begin, end, Heuristic.octile),
                            lambda mz, begin, end: a_star(mz, begin, end, Heuristic.manhattan),
                            lambda mz, begin, end: a_star(mz, begin, end, Heuristic.chebyshev),
                            lambda mz, begin, end: a_star(mz, begin, end, Heuristic.euclidean)]
    solve_maze_n_functions = ['BFS algorithm', 'Dijkstra algorithm', 'A* algorithm (Octile heuristic)',
                              'A* algorithm (Manhattan heuristic)', 'A* algorithm (Chebyshev heuristic)',
                              'A* algorithm (Euclidean heuristic)']
    test_algorithms(n_tests, min_length, max_length, maze_functions, maze_n_functions,
solve_maze_functions,
                    solve_maze_n_functions, is_perfect_mazes=True)

    print('\n', '=== Imperfect algorithms ===', sep='\n')
    n_tests = get_input('Enter the number of tests', 1, 100000)
    min_length = get_input('Enter the minimum length of the maze (odd number)', 5, 100000)
    max_length = get_input('Enter the maximum length of the maze (odd number)', 5, 100000)
    maze_functions = [serpentine, small_rooms, spiral]
    maze_n_functions = ['Serpentine algorithm', 'Small rooms algorithm', 'Spiral algorithm']
    test_algorithms(n_tests, min_length, max_length, maze_functions, maze_n_functions,
solve_maze_functions,
                    solve_maze_n_functions, is_perfect_mazes=False)

    input()

```


test_maze.py

```
if __name__ == '__main__':  
    main()
```

Таблица 26. Файл с алгоритмами генерации идеальных лабиринтов

perfect_maze.py

```
#!/usr/bin/env python3  
# -*- coding: utf-8 -*-  
"""  
    Implementation of 2D perfect maze generation algorithms.  
    Algorithms:  
    - Aldous-Broder algorithm (unbiased maze)  
    - Wilson's algorithm (unbiased maze)  
    - Backtracking (depth-first search, iterative version)  
    - Binary tree algorithm  
    - Division algorithm  
    - Eller's algorithm  
    - Growing tree algorithm  
    - Kruskal's algorithm  
    - Prim's algorithm  
    - Prim's algorithm (modified)  
    - Sidewinder algorithm  
    """  
  
from random import choice, random, randrange, sample, shuffle  
from typing import List  
  
def aldous_broder(y_max: int, x_max: int) -> List[List[bool]]:  
    """  
    Aldous-Broder 2D perfect maze generation algorithm (unbiased maze)  
    :param y_max: height  
    :param x_max: width  
    :return: 2D perfect maze grid  
    """  
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3  
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]  
    directions = ((2, 0), (-2, 0), (0, 2), (0, -2))  
    (current_row, current_col) = (randrange(1, y_max, 2), randrange(1, x_max, 2))  
    grid[current_row][current_col] = False  
    num_visited = 1  
    max_visited = ((y_max - 1) // 2 * (x_max - 1) // 2)  
    while num_visited < max_visited:  
        neighbors = [(current_row + dy, current_col + dx) for dy, dx in directions]  
        valid_neighbors = [(y, x) for y, x in neighbors if 0 < y < y_max and 0 < x < x_max and grid[y][x]]  
        if not valid_neighbors:  
            free_neighbors = [(y, x) for y, x in neighbors if 0 < y < y_max and 0 < x < x_max and not  
grid[y][x]]  
            (current_row, current_col) = choice(free_neighbors)  
            continue  
        shuffle(valid_neighbors)  
        for new_row, new_col in valid_neighbors:  
            if grid[new_row][new_col]:  
                grid[new_row][new_col] = grid[(new_row + current_row) // 2][(new_col + current_col) // 2]  
                (current_row, current_col) = (new_row, new_col)  
                num_visited += 1  
                break  
    return grid  
  
def wilson(y_max: int, x_max: int) -> List[List[bool]]:  
    """  
    Wilson's 2D perfect maze generation algorithm (unbiased maze)  
    :param y_max: height  
    :param x_max: width  
    :return: 2D perfect maze grid  
    """  
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
```

perfect_maze.py

```

grid = [[True for _ in range(x_max)] for _ in range(y_max)]
directions = ((2, 0), (-2, 0), (0, 2), (0, -2))
free = {(y, x) for y in range(1, y_max, 2) for x in range(1, x_max, 2)}
(y, x) = (2 * randrange(y_max // 2) + 1, 2 * randrange(x_max // 2) + 1)
grid[y][x] = False
free.remove((y, x))
while free:
    y, x = key = sample(free, 1)[0]
    free.remove(key)
    path = [key]
    grid[y][x] = False
    neighbors = ((y + dy, x + dx) for dy, dx in directions)
    neighbors = [(y, x) for y, x in neighbors if 0 < y < y_max and 0 < x < x_max]
    y, x = key = choice(neighbors)
    while grid[y][x]:
        grid[y][x] = grid[(y + path[-1][0]) // 2][(x + path[-1][1]) // 2] = False
        neighbors = ((y + dy, x + dx) for dy, dx in directions)
        neighbors = [(y, x) for y, x in neighbors if 0 < y < y_max and 0 < x < x_max]
        neighbors.remove(path[-1])
        free.remove(key)
        path.append(key)
        y, x = key = choice(neighbors)
    if key in path:
        last_key = path.pop()
        free.add(last_key)
        grid[last_key[0]][last_key[1]] = True
        for key in reversed(path):
            free.add(key)
            grid[key[0]][key[1]] = grid[(last_key[0] + key[0]) // 2][(last_key[1] + key[1]) // 2] =
True
            last_key = key
    else:
        grid[(y + path[-1][0]) // 2][(x + path[-1][1]) // 2] = False
return grid

def backtracking(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Iterative version of depth-first search 2D perfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D perfect maze grid
    """
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    directions = [(0, -2), (0, 2), (-2, 0), (2, 0)]
    stack = [(2 * randrange(y_max // 2) + 1, 2 * randrange(x_max // 2) + 1)]
    while stack:
        y, x = stack.pop()
        grid[y][x] = False
        neighbors = ((y + dy, x + dx) for dy, dx in directions)
        neighbors = [(y, x) for y, x in neighbors if 0 < y < y_max and 0 < x < x_max and grid[y][x]]
        if len(neighbors) > 1:
            stack.append((y, x))
        if neighbors:
            ny, nx = choice(neighbors)
            grid[(y + ny) // 2][(x + nx) // 2] = False
            stack.append((ny, nx))
    return grid

def binary_tree(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Binary tree 2D perfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D perfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    grid[1][1] = False

```

perfect_maze.py

```

for x in range(3, x_max, 2):
    grid[1][x] = grid[1][x - 1] = False
for y in range(3, y_max, 2):
    grid[y][1] = grid[y - 1][1] = False
    for x in range(3, x_max, 2):
        if randrange(2):
            grid[y][x] = grid[y][x - 1] = False
        else:
            grid[y][x] = grid[y - 1][x] = False
return grid

def division(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Division 2D perfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D perfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    HORIZONTAL, VERTICAL = 1, 0
    grid = [[False for _ in range(x_max)] for _ in range(y_max)]
    for i in range(len(grid[0])):
        grid[0][i] = grid[-1][i] = True
    for i in range(len(grid)):
        grid[i][0] = grid[i][-1] = True
    region_stack = [(1, 1), (y_max - 2, x_max - 2)]
    while region_stack:
        current_region = region_stack[-1]
        region_stack.pop()
        ((min_y, min_x), (max_y, max_x)) = current_region
        (height, width) = (max_y - min_y + 1, max_x - min_x + 1)
        if height <= 1 or width <= 1:
            continue
        if width < height:
            cut_direction = HORIZONTAL
        elif width > height:
            cut_direction = VERTICAL
        else:
            if width == 2:
                continue
            cut_direction = randrange(2)
        cut_length = (height, width)[(cut_direction + 1) % 2]
        if cut_length < 3:
            continue
        cut_pos = randrange(1, cut_length, 2)
        door_pos = randrange(0, (height, width)[cut_direction], 2)
        if cut_direction == VERTICAL:
            for row in range(min_y, max_y + 1):
                grid[row][min_x + cut_pos] = True
            grid[min_y + door_pos][min_x + cut_pos] = False
        else:
            for col in range(min_x, max_x + 1):
                grid[min_y + cut_pos][col] = True
            grid[min_y + cut_pos][min_x + door_pos] = False
        if cut_direction == VERTICAL:
            region_stack.append(((min_y, min_x), (max_y, min_x + cut_pos - 1)))
            region_stack.append(((min_y, min_x + cut_pos + 1), (max_y, max_x)))
        else:
            region_stack.append(((min_y, min_x), (min_y + cut_pos - 1, max_x)))
            region_stack.append(((min_y + cut_pos + 1, min_x), (max_y, max_x)))
    return grid

def eller(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Eller's 2D perfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D perfect maze grid
    """

```

perfect_maze.py

```

"""
assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
grid = [[True for _ in range(x_max)] for _ in range(y_max)]
parent = {x: {x} for x in range(1, x_max, 2)}
for y in range(1, y_max - 2, 2):
    grid[y][1] = False
    for x in range(3, x_max, 2):
        if x not in parent[x - 2] and randrange(2):
            parent[x].update(parent[x - 2])
            for key in list(parent[x - 2]):
                parent[key] = parent[x]
            grid[y][x - 1] = grid[y][x] = False
        else:
            grid[y][x] = False
    for members in {frozenset(x) for x in parent.values()}:
        walls = [list(), list()]
        for x in members:
            walls[randrange(2)].append(x)
        if not walls[0]:
            walls.reverse()
        for x in walls[0]:
            grid[y + 1][x] = False
        for x in walls[1]:
            for key in parent:
                parent[key].discard(x)
            parent[x] = {x}
y = y_max - 2
grid[y][1] = False
for x in range(3, x_max, 2):
    if x not in parent[x - 2]:
        parent[x].update(parent[x - 2])
        for key in list(parent[x - 2]):
            parent[key] = parent[x]
        grid[y][x - 1] = grid[y][x] = False
    else:
        grid[y][x] = False
return grid

def growing_tree(y_max: int, x_max: int, backtrack_chance: float = 0.5) -> List[List[bool]]:
    """
    Growing tree 2D perfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :param backtrack_chance: splits the logic to either use Recursive Backtracking (RB) or Prim's (random)
    to select
        the next cell to visit (default 1.0)
    :return: 2D perfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    directions = ((2, 0), (-2, 0), (0, 2), (0, -2))
    current_row, current_col = (randrange(1, y_max, 2), randrange(1, x_max, 2))
    grid[current_row][current_col] = False
    active = [(current_row, current_col)]
    while active:
        if random() < backtrack_chance:
            current_row, current_col = active[-1]
        else:
            current_row, current_col = choice(active)
            neighbors = ((current_row + dy, current_col + dx) for dy, dx in directions)
            neighbors = [(y, x) for y, x in neighbors if 0 < y < y_max and 0 < x < x_max and grid[y][x]]
            if not neighbors:
                active = [a for a in active if a != (current_row, current_col)]
                continue
            nn_row, nn_col = choice(neighbors)
            active += [(nn_row, nn_col)]
            grid[nn_row][nn_col] = False
            grid[(current_row + nn_row) // 2][(current_col + nn_col) // 2] = False
    return grid

```

```

def kruskal(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Kruskal's 2D perfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D perfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    parent = {(y, x): (y, x) for y in range(1, y_max, 2) for x in range(1, x_max, 2)}

    def find(x):
        temp = x[:]
        while parent[temp] != temp:
            temp = parent[temp]
        return temp

    walls = [(1, x) for x in range(2, x_max - 1, 2)]
    for y in range(2, y_max - 2, 2):
        walls.extend((y, x) for x in range(1, x_max, 2))
        y += 1
        walls.extend((y, x) for x in range(2, x_max - 1, 2))
    shuffle(walls)
    for y, x in walls:
        if y % 2:
            coord1 = (y, x + 1)
            coord2 = (y, x - 1)
        else:
            coord1 = (y + 1, x)
            coord2 = (y - 1, x)
        if find(coord1) != find(coord2):
            grid[y][x] = grid[coord1[0]][coord1[1]] = grid[coord2[0]][coord2[1]] = False
            parent[find(coord1)] = find(coord2)
    return grid

def prim(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Prim's 2D perfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D perfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    (y, x) = (2 * randrange(y_max // 2) + 1, 2 * randrange(x_max // 2) + 1)
    grid[y][x] = False
    walls = {(y + dy, x + dx) for dy, dx in directions}
    while walls:
        y, x = sample(walls, 1)[0]
        walls.remove((y, x))
        if y == 0 or y == y_max - 1 or x == 0 or x == x_max - 1:
            continue
        if y % 2:
            y1 = y2 = y
            x1, x2 = x + 1, x - 1
        else:
            y1, y2 = y + 1, y - 1
            x1 = x2 = x
        if grid[y1][x1] != grid[y2][x2]:
            if grid[y1][x1]:
                grid[y1][x1] = grid[y][x] = False
                walls.update((y1 + dy, x1 + dx) for dy, dx in directions)
            else:
                grid[y2][x2] = grid[y][x] = False
                walls.update((y2 + dy, x2 + dx) for dy, dx in directions)
    return grid

```

```
def modified_prim(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Modified Prim's 2D perfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D perfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    directions = ((2, 0), (-2, 0), (0, 2), (0, -2))
    (y, x) = (2 * randrange(y_max // 2) + 1, 2 * randrange(x_max // 2) + 1)
    grid[y][x] = False
    cells = ((y + dy, x + dx) for dy, dx in directions)
    cells = {(y, x) for y, x in cells if 0 < y < y_max and 0 < x < x_max}
    while cells:
        y, x = sample(cells, 1)[0]
        cells.remove((y, x))
        neighbors = ((y + dy, x + dx) for dy, dx in directions)
        neighbors = [(y, x) for y, x in neighbors if 0 < y < y_max and 0 < x < x_max]
        ny, nx = choice([(y, x) for y, x in neighbors if not grid[y][x]])
        grid[y][x] = grid[(ny + y) // 2][(nx + x) // 2] = False
        cells.update(((y, x) for y, x in neighbors if grid[y][x]))
    return grid

def sidewinder(y_max: int, x_max: int, skew: float = 0.5) -> List[List[bool]]:
    """
    Sidewinder 2D perfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :param skew: if the skew is set less than 0.5 the maze will be skewed East-West, if it set greater
    than 0.5 it will
    be skewed North-South. (default 0.5)
    :return: 2D perfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    for x in range(1, x_max - 1):
        grid[1][x] = False
    for y in range(3, y_max, 2):
        run = []
        for x in range(1, x_max, 2):
            grid[y][x] = False
            run.append((y, x))
            carve_east = (random() >= skew)
            if carve_east and x < (x_max - 2):
                grid[y][x + 1] = False
            else:
                north = choice(run)
                grid[north[0] - 1][north[1]] = False
                run = []
    return grid

def main():
    from timeit import timeit

    def print_maze(maze):
        print(*[''.join('█' if col else ' ' for col in row) for row in maze], sep='\n')

    def get_input(string, begin, end):
        string += f" ({begin}-{end}):"
        input_result = input(string)
        while not (input_result.isdecimal() and begin <= int(input_result) <= end):
            input_result = input(string)
        return int(input_result)

    functions = [aldous_broder, wilson, binary_tree, backtracking, division, eller, growing_tree, kruskal,
```

perfect_maze.py

```

prim, modified_prim, sidewinder]
n_functions = ['Aldous-Broder algorithm', 'Wilson\'s algorithm', 'Binary tree algorithm',
               'Backtracking algorithm', 'Division algorithm', 'Eller\'s algorithm', 'Growing tree
algorithm',
               'Kruskal\'s algorithm', 'Prim\'s algorithm', 'Prim\'s algorithm (modified)',
               'Sidewinder algorithm']
time_functions = [0 for _ in n_functions]
print('== Perfect maze ==')
width = (get_input('Enter the width of the maze (odd number)', 5, 100000) // 2) * 2 + 1
height = (get_input('Enter the height of the maze (odd number)', 5, 100000) // 2) * 2 + 1
for n, func in zip(n_functions, functions):
    print(n)
    print_maze(func(height, width))
    input('next >>')
print('\n', '=== Time test ===', 'Tests: 15', 'Sizes: 11-41', sep='\n')
for i in range(15):
    y, x = (randrange(10, 40) // 2) * 2 + 1, (randrange(10, 40) // 2) * 2 + 1
    for j, func in enumerate(functions):
        name = func.__name__
        time_functions[j] += timeit(f"{name}({y}, {x})", f"from __main__ import {name}", number=15)
for time, n in sorted(zip(time_functions, n_functions)):
    print(n + ': ' + str(time))
input()

if __name__ == '__main__':
    main()

```

Таблица 27. Файл с алгоритмами генерации неидеальных лабиринтов

imperfect_maze.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
    Implementation of 2D imperfect maze generation algorithms.
    Algorithms:
    - Serpentine algorithm
    - Small rooms algorithm
    - Spiral algorithm
"""

from random import randrange
from typing import List

def serpentine(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Serpentine 2D imperfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D imperfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    if randrange(2):
        for row in range(1, y_max - 1):
            grid[row][randrange(1, x_max - 1)] = False
            for col in range(1, x_max - 1, 2):
                grid[row][col] = False
        for col in range(2, x_max - 1, 4):
            grid[1][col] = False
        for col in range(4, x_max - 1, 4):
            grid[y_max - 2][col] = False
    else:
        for row in range(1, y_max - 1, 2):
            for col in range(1, x_max - 1):
                grid[row][col] = False
        for row in range(2, y_max - 1, 4):
            grid[row][1] = False

```

imperfect_maze.py

```

        grid[row][randrange(2, x_max - 1)] = False
    for row in range(4, y_max - 1, 4):
        grid[row][x_max - 2] = False
        grid[row][randrange(1, x_max - 2)] = False
    return grid

def small_rooms(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Small rooms 2D imperfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D imperfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    for y in range(1, y_max - 1):
        if y % 2:
            for x in range(1, x_max - 3, 4):
                grid[y][x] = False
                grid[y][x + 1] = False
                grid[y][x + 2] = False
                if x < x_max - 4 and not randrange(3):
                    grid[y][x + 3] = False
            else:
                for x in range(2, x_max - 1, 4):
                    grid[y][x] = False
    y_mid = y_max // 2
    for x in range(1, x_max - 1):
        grid[y_mid][x] = False
    return grid

def spiral(y_max: int, x_max: int) -> List[List[bool]]:
    """
    Spiral 2D imperfect maze generation algorithm
    :param y_max: height
    :param x_max: width
    :return: 2D imperfect maze grid
    """
    assert y_max % 2 and x_max % 2 and y_max >= 3 and x_max >= 3
    grid = [[True for _ in range(x_max)] for _ in range(y_max)]
    directions = [(-2, 0), (0, 2), (2, 0), (0, -2)]
    if randrange(2):
        directions.reverse()
    current = (1, 1)
    grid[1][1] = False
    next_dir = 0
    while True:
        y, x = current
        new_y, new_x = (y + directions[next_dir][0], x + directions[next_dir][1])
        neighbors = ((y + dy, x + dx) for dy, dx in directions)
        neighbors = [(y, x) for y, x in neighbors if 0 < y < y_max and 0 < x < x_max if grid[y][x]]
        if (new_y, new_x) in neighbors:
            grid[(y + new_y) // 2][(x + new_x) // 2] = False
            grid[new_y][new_x] = False
            current = (new_y, new_x)
        elif not neighbors:
            break
        else:
            next_dir = (next_dir + 1) % 4
    for i in range(max(y_max, x_max)):
        grid[randrange(1, y_max - 1)][randrange(1, x_max - 1)] = False
    return grid

def main():
    from timeit import timeit

    def print_maze(maze):

```


imperfect_maze.py

```
print(*[''.join('█' if col else ' ' for col in row) for row in maze], sep='\n')

def get_input(string, begin, end):
    string += f" ({begin}-{end}):"
    input_result = input(string)
    while not (input_result.isdecimal() and begin <= int(input_result) <= end):
        input_result = input(string)
    return int(input_result)

functions = [serpentine, small_rooms, spiral]
n_functions = ['Serpentine algorithm', 'Small rooms algorithm', 'Spiral algorithm']
time_functions = [0 for _ in n_functions]
print('== Imperfect maze ==')
width = (get_input('Enter the width of the maze (odd number)', 5, 100000) // 2) * 2 + 1
height = (get_input('Enter the height of the maze (odd number)', 5, 100000) // 2) * 2 + 1
for n, func in zip(n_functions, functions):
    print(n)
    print_maze(func(height, width))
    input('next >>')
print('\n', '=== Time test ===', 'Tests: 15', 'Sizes: 11-41', sep='\n')
for i in range(15):
    y, x = (randrange(10, 40) // 2) * 2 + 1, (randrange(10, 40) // 2) * 2 + 1
    for j, func in enumerate(functions):
        name = func.__name__
        time_functions[j] += timeit(f"{name}({y}, {x})", f"from __main__ import {name}", number=15)
    for time, n in sorted(zip(time_functions, n_functions)):
        print(n + ': ' + str(time))
    input()

if __name__ == '__main__':
    main()
```

Таблица 28. Файл с алгоритмами поиска кратчайших путей в лабиринтах

solve_maze.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
    Implementation of algorithms for finding the shortest paths in 2D mazes.
    Algorithms:
    - A* algorithm
    - BFS (breadth-first search) (iterative version)
    - Dijkstra algorithm
"""

import heapq
from collections import deque
from math import inf
from typing import Callable, List, Tuple

class Heuristic:
    """
    A class with basic heuristics for the A * algorithm
    """

    @staticmethod
    def octile(y0, x0, y1, x1):
        (ty, tx) = (abs(y0 - y1), abs(x0 - x1))
        return max(ty, tx) + (2 ** 0.5 - 1) * min(ty, tx)

    @staticmethod
    def manhattan(y0, x0, y1, x1):
        return abs(y0 - y1) + abs(x0 - x1)

    @staticmethod
    def chebyshev(y0, x0, y1, x1):
        return max(abs(y0 - y1), abs(x0 - x1))
```

```

@staticmethod
def euclidean(y0, x0, y1, x1):
    return ((y0 - y1) ** 2 + (x0 - x1) ** 2) ** 0.5

def a_star(maze: List[List[bool]], beginNode: Tuple[int, int], endNode: Tuple[int, int],
           heuristic: Callable = Heuristic.manhattan) -> List[Tuple[int, int]]:
    """
    A* algorithm
    This implementation uses heapq - a priority queue.
    :param maze: 2D maze grid
    :param beginNode: initial position
    :param endNode: target position
    :param heuristic: heuristic function
    :return: path from initial to target position
    """
    current = (-1, -1)
    (y_max, x_max) = (len(maze), len(maze[0]))
    q = []
    cost = 0
    heapq.heappush(q, (cost, endNode))
    directions = ((-1, 0), (0, -1), (1, 0), (0, 1))
    parentNode = {endNode: endNode}
    costMap = {endNode: cost}
    while q:
        current = heapq.heappop(q)[1]
        if current == beginNode:
            break
        cost = costMap[current] + 1
        for y_dir, x_dir in directions:
            (dy, dx) = (current[0] + y_dir, current[1] + x_dir)
            if dy >= y_max or dy < 0 or dx >= x_max or dx < 0 or maze[dy][dx]:
                continue
            neighbor = (dy, dx)
            if cost < costMap.get(neighbor, inf):
                costMap[neighbor] = cost
                parentNode[neighbor] = current
                heapq.heappush(q, (cost + heuristic(current[0], current[1], beginNode[0], beginNode[1]),
neighbor))
            path = list()
            if current == beginNode:
                path.append(current)
                while current != endNode:
                    current = parentNode[current]
                    path.append(current)
            return path

def bfs(maze: List[List[bool]], beginNode: Tuple[int, int], endNode: Tuple[int, int]) -> List[Tuple[int,
int]]:
    """
    Breadth-first search algorithm (iterative version)
    (!) This implementation uses a deque instead of a queue. This choice is only related to the speed.
    :param maze: 2D maze grid
    :param beginNode: initial position
    :param endNode: target position
    :return: path from initial to target position
    """
    current = (-1, -1)
    (y_max, x_max) = (len(maze), len(maze[0]))
    q = deque()
    q.append(endNode)
    directions = ((-1, 0), (0, -1), (1, 0), (0, 1))
    parentNode = {endNode: endNode}
    while q:
        current = q.popleft()
        if current == beginNode:
            break
        for y_dir, x_dir in directions:

```

solve_maze.py

```

(dy, dx) = (current[0] + y_dir, current[1] + x_dir)
if dy >= y_max or dy < 0 or dx >= x_max or dx < 0 or maze[dy][dx]:
    continue
neighbor = (dy, dx)
if neighbor not in parentNode:
    q.append(neighbor)
    parentNode[neighbor] = current
path = list()
if current == beginNode:
    path.append(current)
    while current != endNode:
        current = parentNode[current]
        path.append(current)
return path

def dijkstra(maze: List[List[bool]], beginNode: Tuple[int, int], endNode: Tuple[int, int]) ->
List[Tuple[int, int]]:
    """
    Dijkstra algorithm
    This implementation uses heapq - a priority queue.
    :param maze: 2D maze grid
    :param beginNode: initial position
    :param endNode: target position
    :return: path from initial to target position
    """
    current = (-1, -1)
    (y_max, x_max) = (len(maze), len(maze[0]))
    q = []
    heapq.heappush(q, (0, endNode))
    directions = ((-1, 0), (0, -1), (1, 0), (0, 1))
    parentNode = {endNode: endNode}
    costMap = {endNode: 0}
    while q:
        current = heapq.heappop(q)[1]
        if current == beginNode:
            break
        cost = costMap[current] + 1
        for y_dir, x_dir in directions:
            (dy, dx) = (current[0] + y_dir, current[1] + x_dir)
            if dy >= y_max or dy < 0 or dx >= x_max or dx < 0 or maze[dy][dx]:
                continue
            neighbor = (dy, dx)
            if cost < costMap.get(neighbor, inf):
                costMap[neighbor] = cost
                parentNode[neighbor] = current
                heapq.heappush(q, (cost, neighbor))
    path = list()
    if current == beginNode:
        path.append(current)
        while current != endNode:
            current = parentNode[current]
            path.append(current)
    return path

```

Таблица 29. Файл с выводом результата работы файла-программы test_maze.py

Вывод программы

```

=== Perfect algorithms ===
Enter the number of tests (1-100000):100
Enter the minimum length of the maze (odd number) (5-100000):200
Enter the maximum length of the maze (odd number) (5-100000):200
Time of execution of maze generation algorithms
Sidewinder algorithm: 2.046875
Binary tree algorithm: 2.625
Division algorithm: 5.515625
Backtracking algorithm: 11.046875
Eller's algorithm: 15.609375

```

```

Prim's algorithm (modified): 29.890625
Aldous-Broder algorithm: 165.359375
Prim's algorithm: 180.84375
Kruskal's algorithm: 283.0
Growing tree algorithm: 385.71875
Wilson's algorithm: 525.765625
Time of execution of algorithms to find the shortest paths in the mazes (briefly)
BFS algorithm: 41.546875
A* algorithm (Manhattan heuristic): 53.65625
A* algorithm (Octile heuristic): 63.8125
A* algorithm (Chebyshev heuristic): 64.265625
A* algorithm (Euclidean heuristic): 70.34375
Dijkstra algorithm: 70.453125
Time of execution of algorithms to find the shortest paths in the mazes (in detail)
#####
Function: BFS algorithm
Kruskal's algorithm: 3.5625
Prim's algorithm: 3.578125
Sidewinder algorithm: 3.578125
Aldous-Broder algorithm: 3.625
Eller's algorithm: 3.640625
Backtracking algorithm: 3.65625
Wilson's algorithm: 3.8125
Growing tree algorithm: 3.9375
Binary tree algorithm: 3.984375
Division algorithm: 4.0
Prim's algorithm (modified): 4.171875
#####
Function: A* algorithm (Manhattan heuristic)
Prim's algorithm: 2.796875
Prim's algorithm (modified): 3.140625
Sidewinder algorithm: 4.171875
Binary tree algorithm: 4.265625
Wilson's algorithm: 4.71875
Growing tree algorithm: 4.8125
Kruskal's algorithm: 4.90625
Eller's algorithm: 5.5625
Aldous-Broder algorithm: 6.140625
Backtracking algorithm: 6.359375
Division algorithm: 6.78125
#####
Function: A* algorithm (Octile heuristic)
Prim's algorithm: 3.828125
Prim's algorithm (modified): 4.453125
Sidewinder algorithm: 5.3125
Binary tree algorithm: 5.4375
Wilson's algorithm: 5.46875
Kruskal's algorithm: 5.59375
Growing tree algorithm: 5.6875
Eller's algorithm: 6.46875
Aldous-Broder algorithm: 6.78125
Backtracking algorithm: 7.21875
Division algorithm: 7.5625
#####
Function: A* algorithm (Chebyshev heuristic)
Prim's algorithm: 4.21875
Prim's algorithm (modified): 4.734375
Binary tree algorithm: 5.4375
Sidewinder algorithm: 5.609375
Growing tree algorithm: 5.78125
Kruskal's algorithm: 5.78125
Wilson's algorithm: 5.828125
Eller's algorithm: 6.3125
Backtracking algorithm: 6.65625
Aldous-Broder algorithm: 6.671875
Division algorithm: 7.234375
#####
Function: A* algorithm (Euclidean heuristic)
Prim's algorithm: 4.234375
Prim's algorithm (modified): 4.9375
Binary tree algorithm: 5.8125

```

```

Sidewinder algorithm: 5.890625
Wilson's algorithm: 6.03125
Kruskal's algorithm: 6.1875
Growing tree algorithm: 6.40625
Eller's algorithm: 6.984375
Aldous-Broder algorithm: 7.578125
Division algorithm: 8.125
Backtracking algorithm: 8.15625
#####
Function: Dijkstra algorithm
Backtracking algorithm: 5.609375
Aldous-Broder algorithm: 5.90625
Kruskal's algorithm: 6.140625
Eller's algorithm: 6.15625
Prim's algorithm: 6.234375
Sidewinder algorithm: 6.28125
Wilson's algorithm: 6.46875
Growing tree algorithm: 6.546875
Division algorithm: 6.90625
Binary tree algorithm: 7.015625
Prim's algorithm (modified): 7.1875
=== Imperfect algorithms ===
Enter the number of tests (1-100000):100
Enter the minimum length of the maze (odd number) (5-100000):200
Enter the maximum length of the maze (odd number) (5-100000):200
Time of execution of maze generation algorithms
Serpentine algorithm: 0.5
Small rooms algorithm: 1.484375
Spiral algorithm: 5.375
Time of execution of algorithms to find the shortest paths in the mazes (briefly)
BFS algorithm: 10.796875
A* algorithm (Manhattan heuristic): 13.4375
A* algorithm (Octile heuristic): 16.40625
A* algorithm (Chebyshev heuristic): 16.515625
Dijkstra algorithm: 17.8125
A* algorithm (Euclidean heuristic): 17.921875
Time of execution of algorithms to find the shortest paths in the mazes (in detail)
#####
Function: BFS algorithm
Serpentine algorithm: 3.453125
Spiral algorithm: 3.53125
Small rooms algorithm: 3.8125
#####
Function: A* algorithm (Manhattan heuristic)
Small rooms algorithm: 1.59375
Spiral algorithm: 5.8125
Serpentine algorithm: 6.03125
#####
Function: A* algorithm (Octile heuristic)
Small rooms algorithm: 3.125
Spiral algorithm: 6.5
Serpentine algorithm: 6.78125
#####
Function: A* algorithm (Chebyshev heuristic)
Small rooms algorithm: 3.796875
Serpentine algorithm: 6.359375
Spiral algorithm: 6.359375
#####
Function: Dijkstra algorithm
Serpentine algorithm: 5.25
Spiral algorithm: 5.734375
Small rooms algorithm: 6.828125
#####
Function: A* algorithm (Euclidean heuristic)
Small rooms algorithm: 3.28125
Spiral algorithm: 7.171875
Serpentine algorithm: 7.46875

```

СПИСОК ЛИТЕРАТУРЫ

1. **Think Labyrinth: Maze algorithms** [Электронный ресурс]. – Режим доступа: <https://www.astrolog.org/labyrnth/algrithm.htm>, свободный – (28.05.2020).
2. **Implementation of A*** [Электронный ресурс]. – Режим доступа: <https://www.redblobgames.com/pathfinding/a-star/implementation.html>, свободный – (28.05.2020).
3. **Aldous–Broder algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2011/1/17/maze-generation-aldous-broder-algorithm.html>, свободный – (28.05.2020).
4. **Wilson's algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2011/1/20/maze-generation-wilson-s-algorithm>, свободный – (28.05.2020).
5. **Backtracking algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking>, свободный – (28.05.2020).
6. **Binary Tree algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2011/2/1/maze-generation-binary-tree-algorithm.html>, свободный – (28.05.2020).
7. **Division algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2011/1/12/maze-generation-recursive-division-algorithm>, свободный – (28.05.2020).
8. **Eller's algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2010/12/29/maze-generation-eller-s-algorithm>, свободный – (28.05.2020).
9. **Growing Tree algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2011/1/27/maze-generation-growing-tree-algorithm>, свободный – (28.05.2020).
10. **Kruskal's algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2011/1/3/maze-generation-kruskal-s-algorithm>, свободный – (28.05.2020).
11. **Prim's algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm.html>, свободный – (28.05.2020).
12. **Sidewinder algorithm** [Электронный ресурс]. – Режим доступа: <https://weblog.jamisbuck.org/2011/2/3/maze-generation-sidewinder-algorithm.html>, свободный – (28.05.2020).