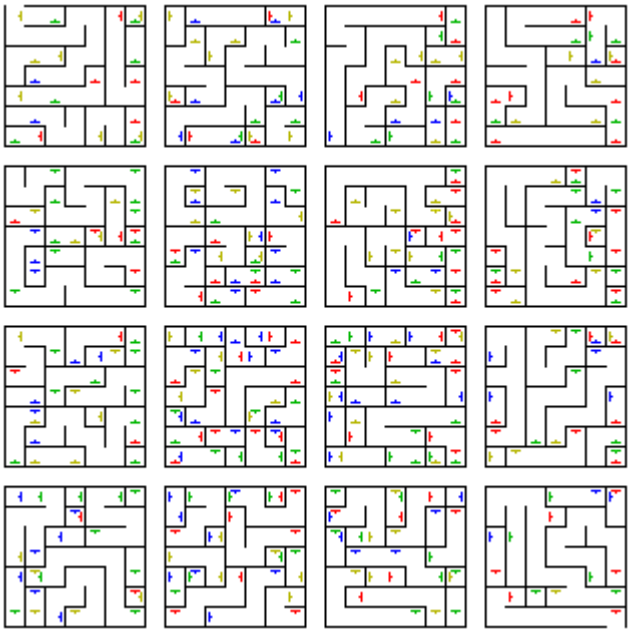
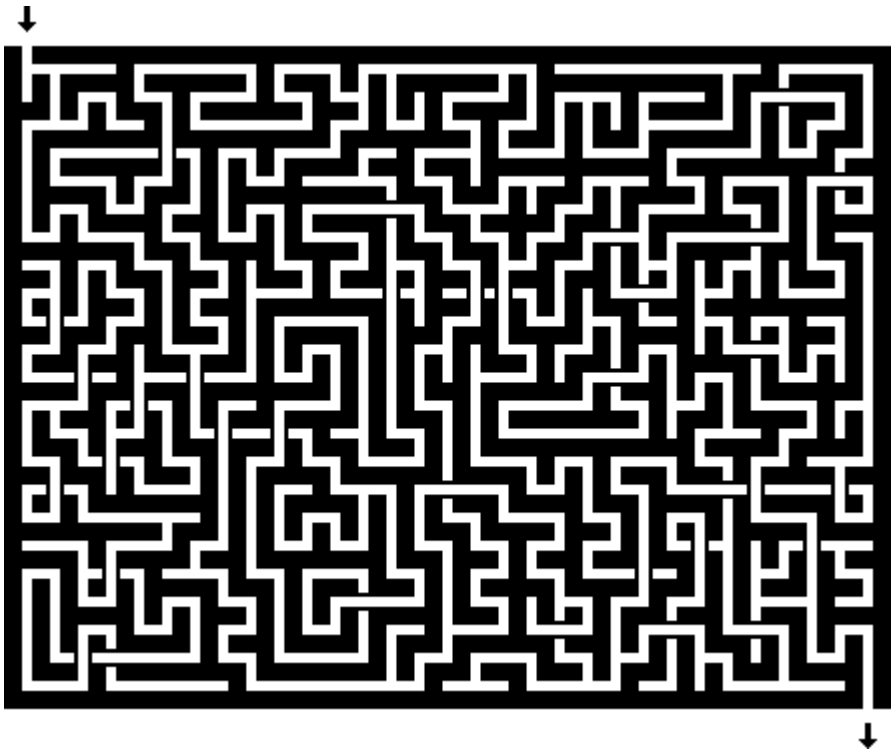
Классификация лабиринтов

Лабиринты в целом (а значит, и алгоритмы для их создания) можно разбить по семи различным классификациям: размерности, гиперразмерности, топологии, тесселяции, маршрутизации, текстуре и приоритету. Лабиринт может использовать по одному элементу из каждого класса в любом сочетании.
Размерность: класс размерности по сути определяет, сколько измерений в пространстве заполняет лабиринт. Существуют следующие типы:

- **Двухмерные:** большинство лабиринтов, как бумажных, так и реальных, имеют эту размерность, то есть мы всегда можем отобразить план лабиринта на листе бумаги и двигаться по нему, не пересекая никаких других коридоров лабиринта.
- **Трёхмерные:** трёхмерный лабиринт имеет несколько уровней; в нём (по крайней мере, в ортогональной версии) проходы могут кроме четырёх сторон света опускаться вниз и подниматься вверх. 3D-лабиринт часто визуализируют как массив из 2D-уровней с пометками лестниц «вверх» и «вниз».

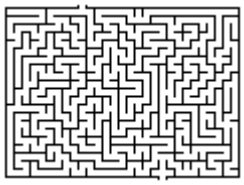


Более высокие размерности: можно создавать четырёхмерные лабиринты и ещё более многомерные. Иногда их визуализируют как 3D-лабиринты с «порталами», ведущими сквозь четвёртое измерение, например, порталы в «прошлое» и «будущее».

- 

  Переплетения: лабиринты с переплетениями — это, по сути двухмерные (или, точнее 2,5-мерные) лабиринты, в которых, однако, проходы могут накладываться друг на друга. При отображении обычно вполне очевидно, где находятся тупики и как один проход находится над другим. Лабиринты из реального мира, в которых есть мосты, соединяющие одну часть лабиринта с другой, частично являются переплетениями.

Гиперразмерность: классификация по гиперразмерности соответствует размерности объекта, двигающегося через лабиринт, а не самого лабиринта. Существуют следующие типы:

- 

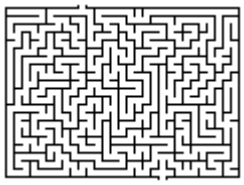
  Не-гиперлабиринты: практически все лабиринты, даже созданные в высокой размерности или имеющие особые правила, обычно являются не-гиперлабиринтами. В них мы работаем с точкой или небольшим объектом, например, шариком или самим игроком, которые движутся от точки к точке, а проложенный маршрут образует линию. В каждой точке существует легко просчитываемое количество вариантов выбора.

- **Гиперлабиринты:** гиперлабиринты — это лабиринты, в которых решаемый объект является не просто точкой. Стандартный гиперлабиринт (или гиперлабиринт первого порядка) состоит из линии, которая при перемещении по пути образует поверхность. Гиперлабиринт может существовать только в 3D или среде с большей размерностью, и входом в гиперлабиринт

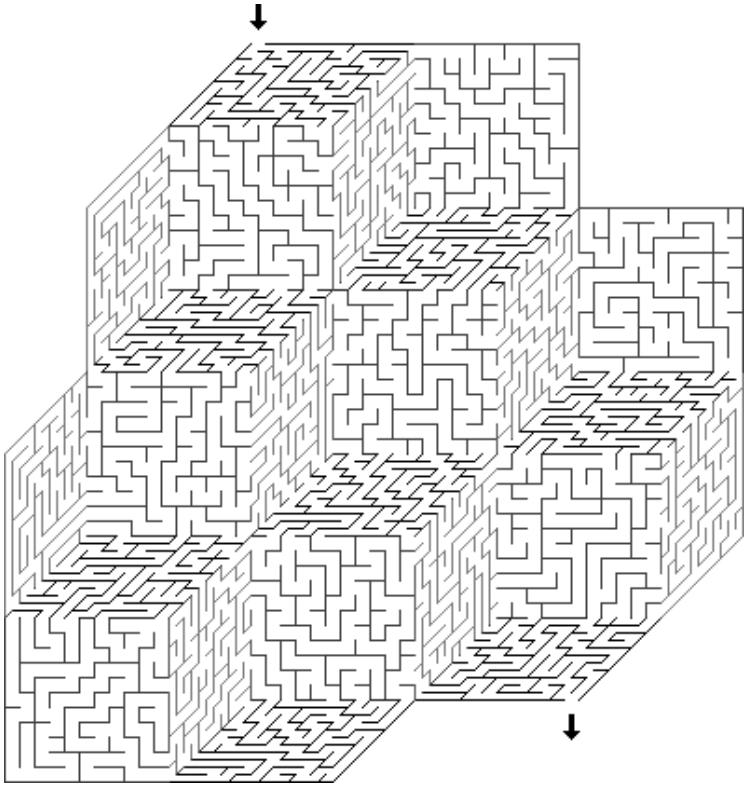
часто является не точка, а линия. Гиперлабиринт фундаментально отличается, потому что учитывать и одновременно работать с несколькими частями вдоль линии, а в любой момент времени существует практически бесконечное количество состояний и вариантов того, что можно сделать с линией. Линия решения бесконечна, или её конечные точки находятся за пределами гиперлабиринта, чтобы избежать сжатия линии в точку, ведь в этом случае можно считать это не-гиперлабиринтом.

- **Гипер-гиперлабиринт:** гиперлабиринты могут быть произвольно высокой размерности. Гипер-гиперлабиринт (или гиперлабиринт второго порядка) снова повышает размерность решаемого объекта. Здесь решаемый объект — это плоскость, которая при перемещении по пути образует объёмную фигуру. Гипер-гиперлабиринт может существовать только в 4D или среде большей размерности.

Топология: класс топологии описывает геометрию пространства лабиринта, в котором тот существует как целое. Есть следующие типы:

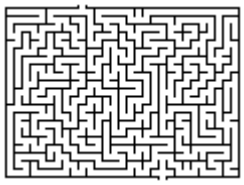
- 

  Обычный: это стандартный лабиринт в эвклидовом пространстве.

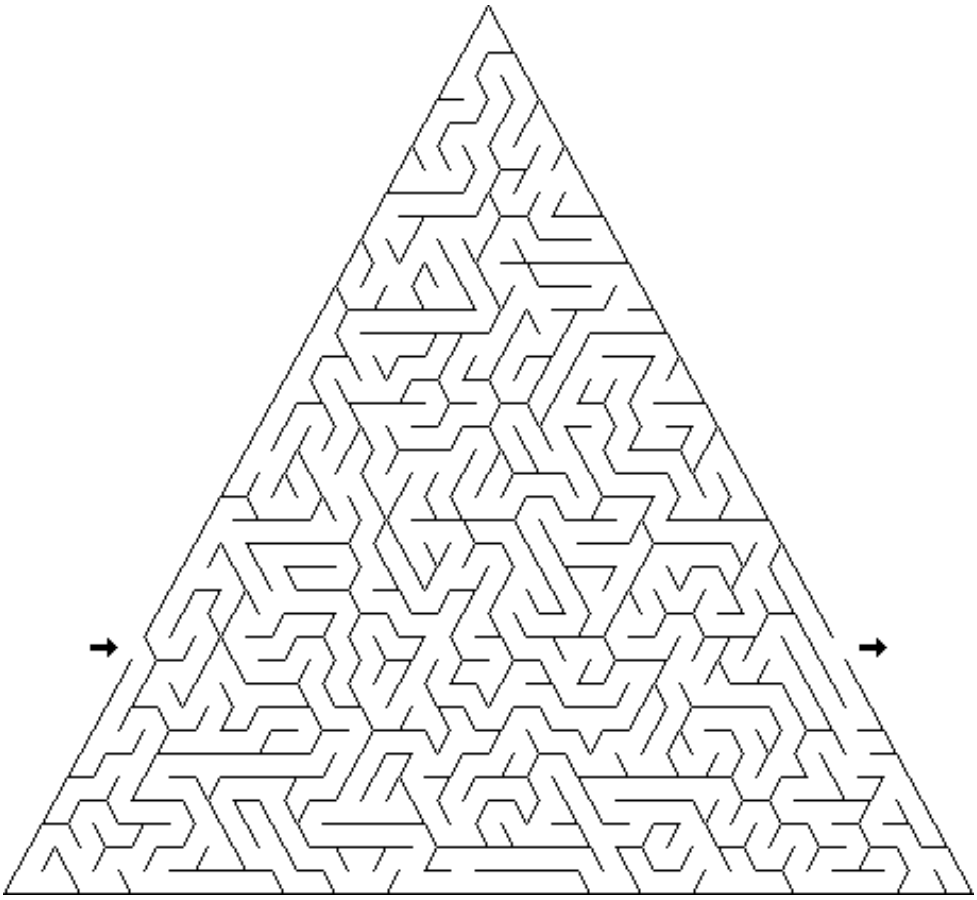
- 

Planair: термин «planair» описывает любой лабиринт с необычной топологией. Обычно это означает, что края лабиринта соединены интересным образом. Примеры: лабиринты на поверхности куба, лабиринты на поверхности ленты Мёбиуса и лабиринты, эквивалентные находящимся на торе, где попарно соединены левая и правая, верхняя и нижняя стороны.

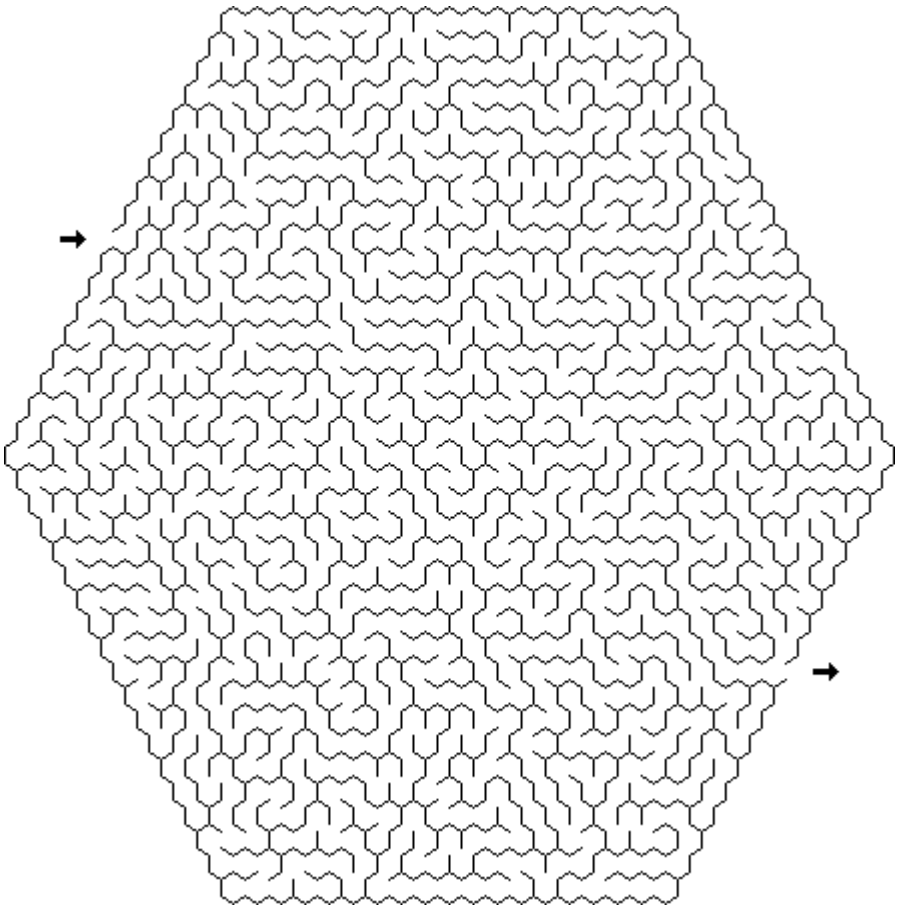
Тесселяция: классификация геометрии отдельных ячеек, из которых состоит лабиринт. Существующие типы:

- 

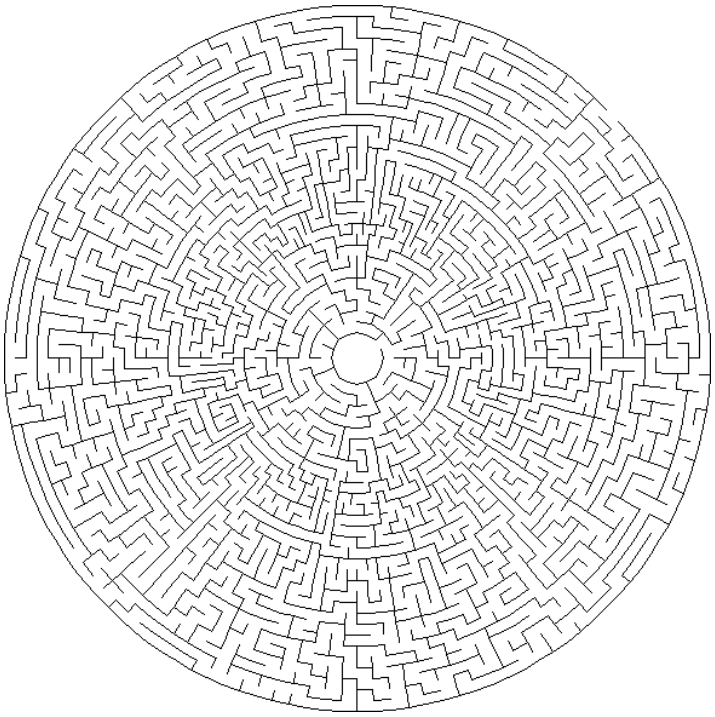
  Ортогональный: это стандартная прямоугольная сетка, в которой ячейки имеют проходы, пересекающиеся под прямыми углами. В контексте тесселяции их также можно назвать гамма-лабиринтами.

- 

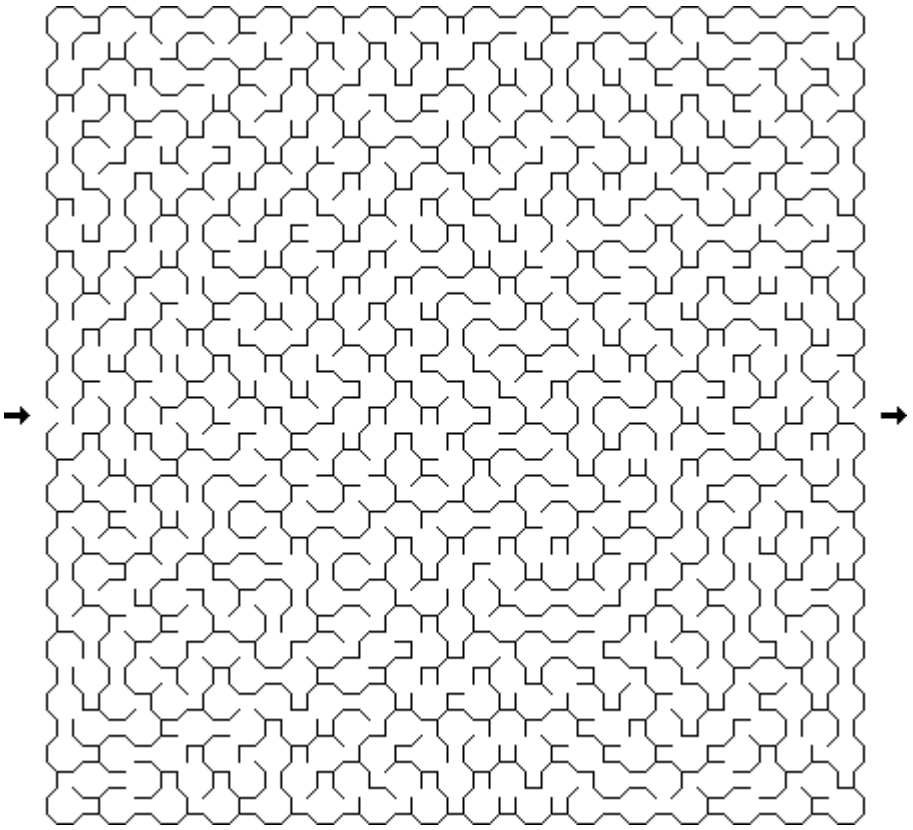
  Дельта: дельта-лабиринты состоят из соединённых треугольников, и у каждой ячейки может быть до трёх соединённых с ней проходов.

- 

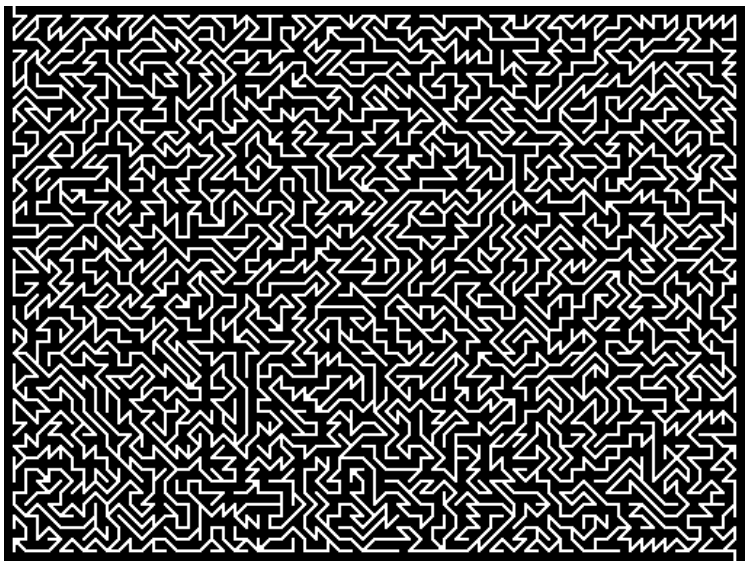
Сигма: сигма-лабиринты составлены из соединённых шестиугольников; у каждой ячейки может быть до шести проходов.

- 

Тета: тета-лабиринты состоят из концентрических окружностей проходов, в которых начало или конец находится в центре, а другой — на внешнем крае. Ячейки обычно имеют четыре возможных соединения проходов, но их может быть и больше благодаря большему количеству ячеек во внешних кольцах проходов.

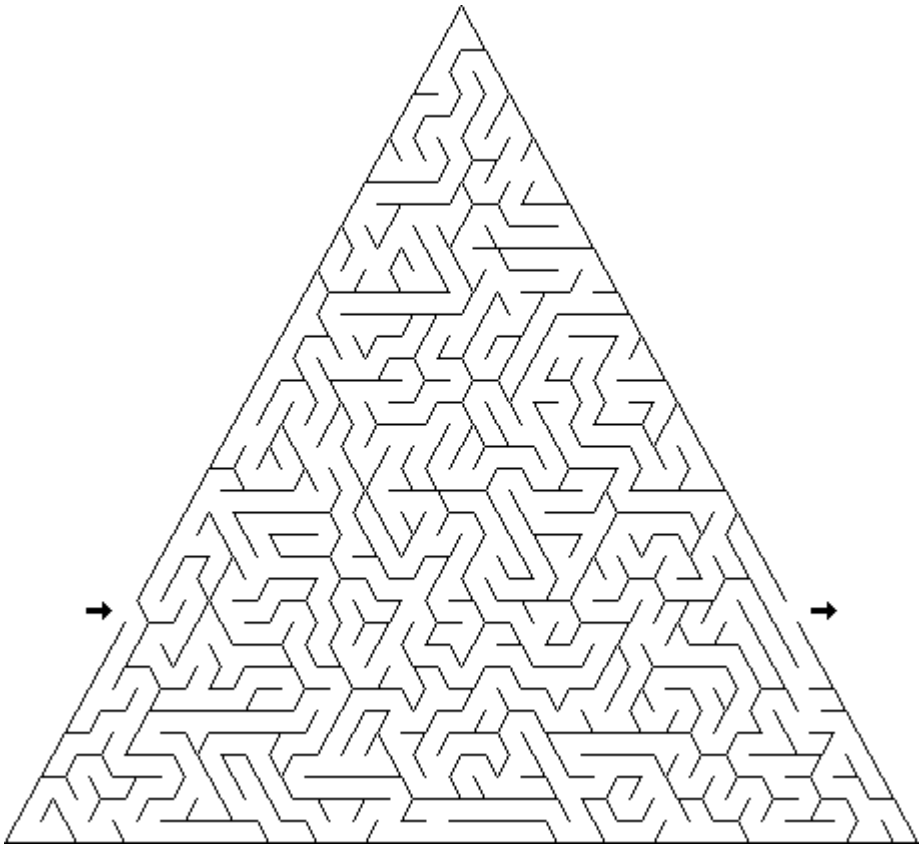
- 

Ипсилон: ипсилон-лабиринты состоят из соединённых восьмиугольников или квадратов, в них каждая ячейка может иметь до восьми или четырёх проходов.

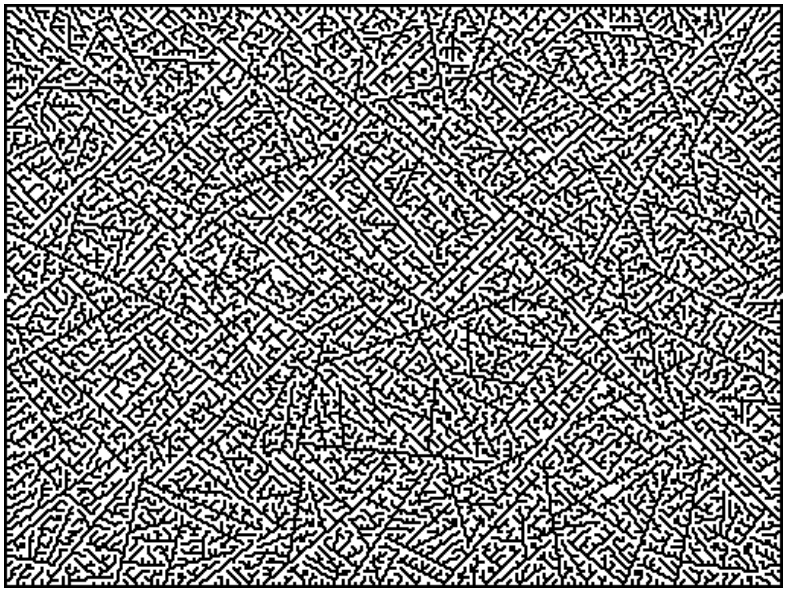
- 

Дзета: дзета-лабиринт расположен на прямоугольной сетке, только в дополнение к

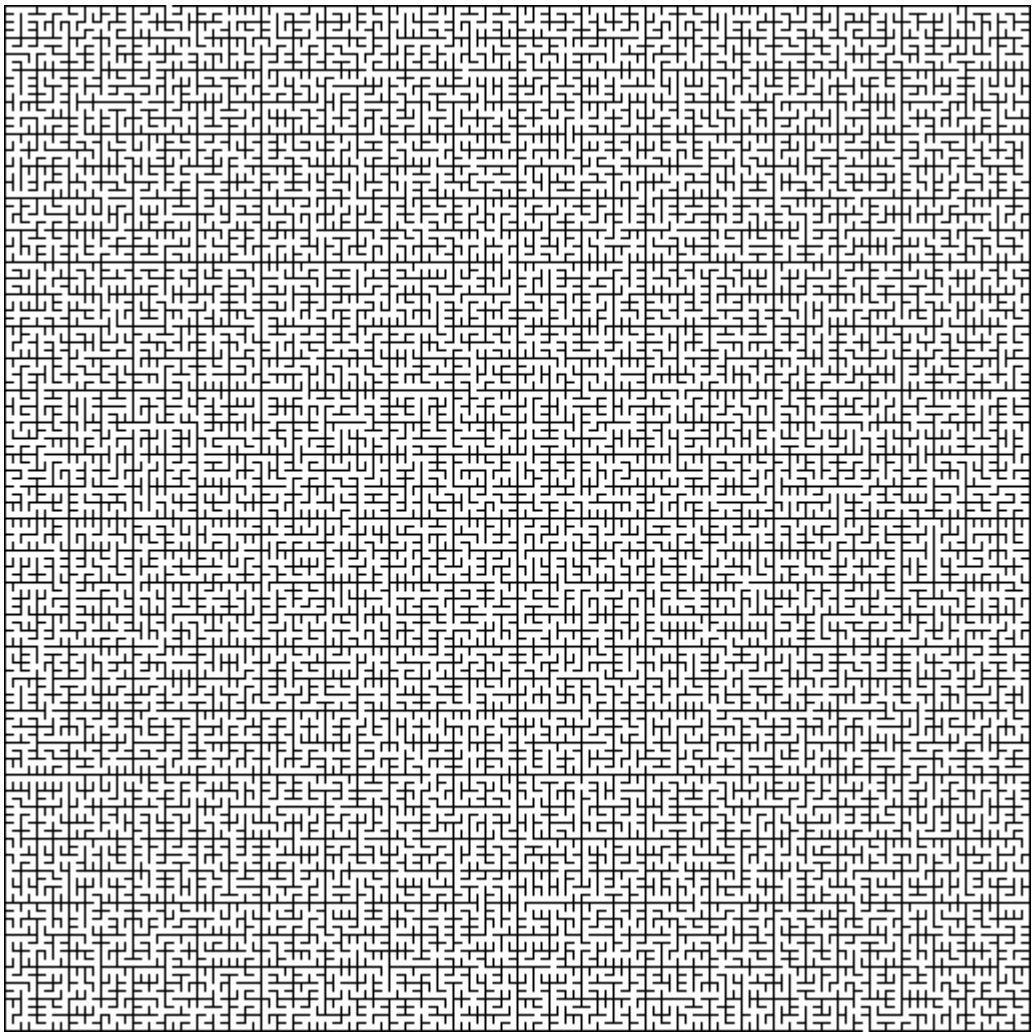
горизонтальным и вертикальным проходам допускаются диагональные проходы под углом 45 градусов.

- 

Омега: термин «омега» относится практически к любому лабиринту с постоянной неортогональной тесселяцией. Дельта-, сигма-, тета- и ипсилон-лабиринты относятся к этому типу, как и многие другие схемы, которые можно придумать, например, лабиринт, состоящий из пар прямоугольных треугольников.

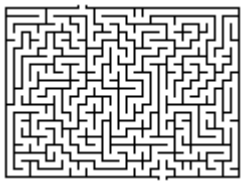
- 

Crack: crack-лабиринт — это аморфный лабиринт без постоянной тесселяции, в котором стены и проходы расположены под случайными углами.

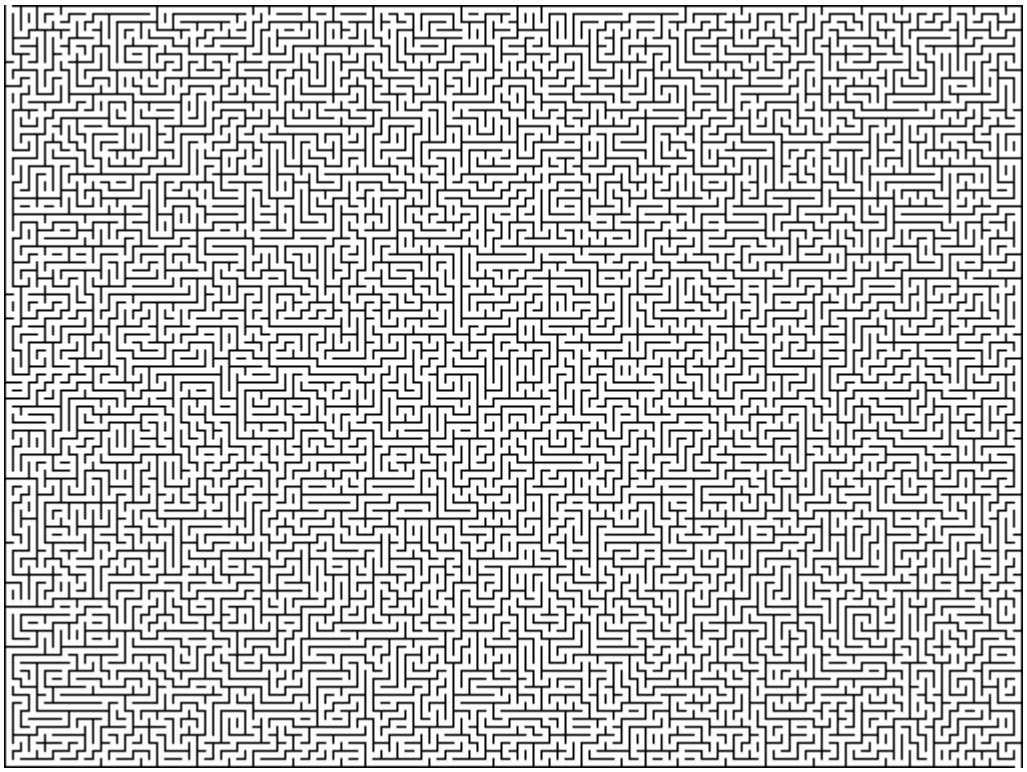
- 

Фрактальный: фрактальный лабиринт — это лабиринт, составленный из меньших лабиринтов. Фрактальный лабиринт из вложенных ячеек — это лабиринт, в каждой ячейке которого размещены другие лабиринты, и этот процесс может повторяться несколько раз. Бесконечно рекурсивный фрактальный лабиринт — это истинный фрактал, в котором содержимое лабиринта копирует себя, создавая по сути бесконечно большой лабиринт.

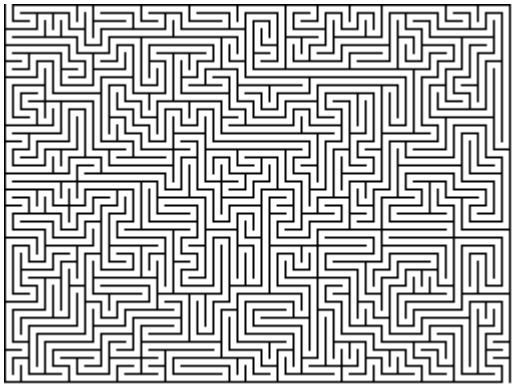
Маршрутизация: классификация по маршрутизации — это, вероятно, наиболее интересный аспект в генерации лабиринтов. От связан с типами проходов в пределах геометрии, определённой в описанных выше категориях.

- 

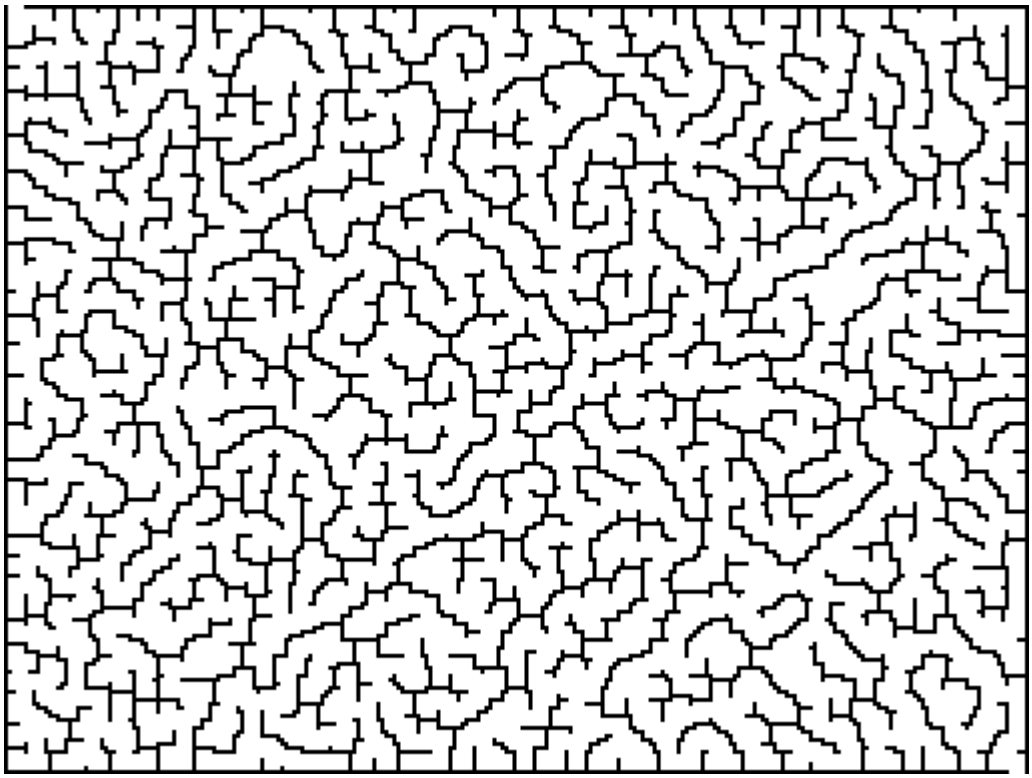
Идеальный: «идеальным» называют лабиринт без петель или замкнутых цепей и без недостижимых областей. Также он называется лабиринтом с одиночным соединением (simply-connected Maze). Из каждой точки существует ровно один путь к любой другой точке. Лабиринт имеет только одно решение. С точки зрения программирования такой лабиринт можно описать как дерево, связующее множество ячеек или вершин.

- 

Плетёный (Braid): «плетёный» означает, что в лабиринте нет тупиков. Также его называют лабиринтом с многократными соединениями (purely multiply connected Maze). В таком лабиринте используются проходы, замыкающиеся и возвращающиеся друг к другу (отсюда название «плетёный»), они заставляют тратить больше времени на ходьбу кругами вместо попадания в тупики. Качественный плетёный лабиринт может быть гораздо сложнее идеального лабиринта того же размера.

- 

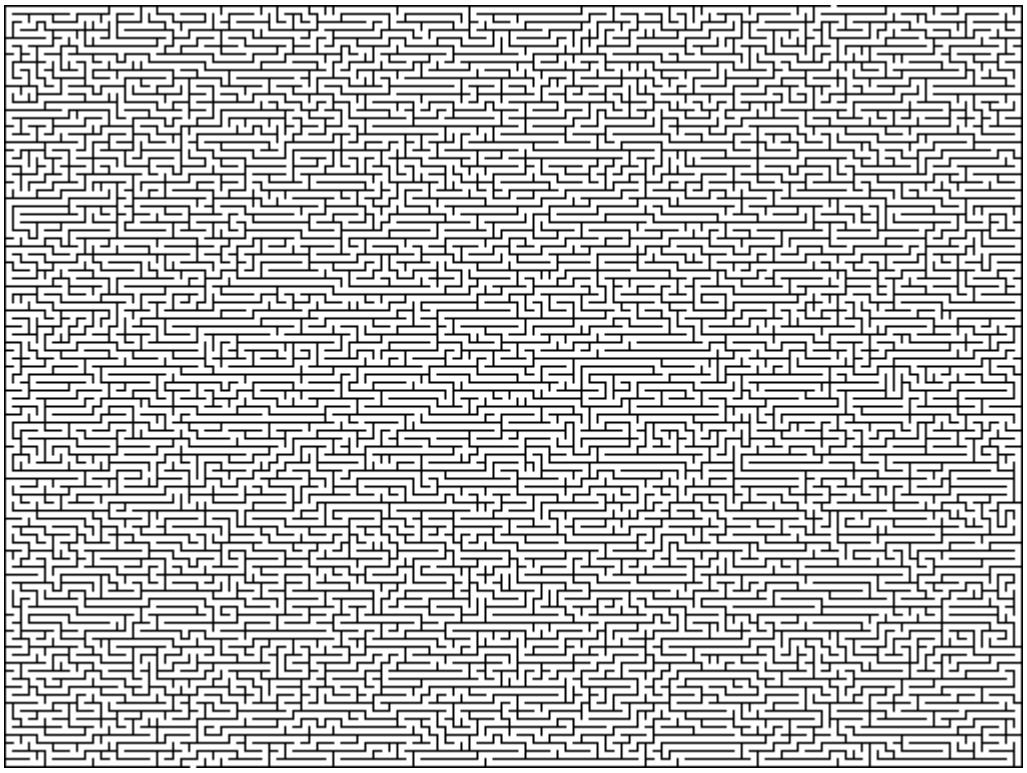
  Одномаршрутный (Unicursal): под одномаршрутным подразумевается лабиринт без развилок. Одномаршрутный лабиринт содержит один длинный извивающийся проход, который меняет направление на всём протяжении лабиринта. Он не очень сложен, только если вы случайно не повернёте назад на полпути и не вернётесь в начало.

- 

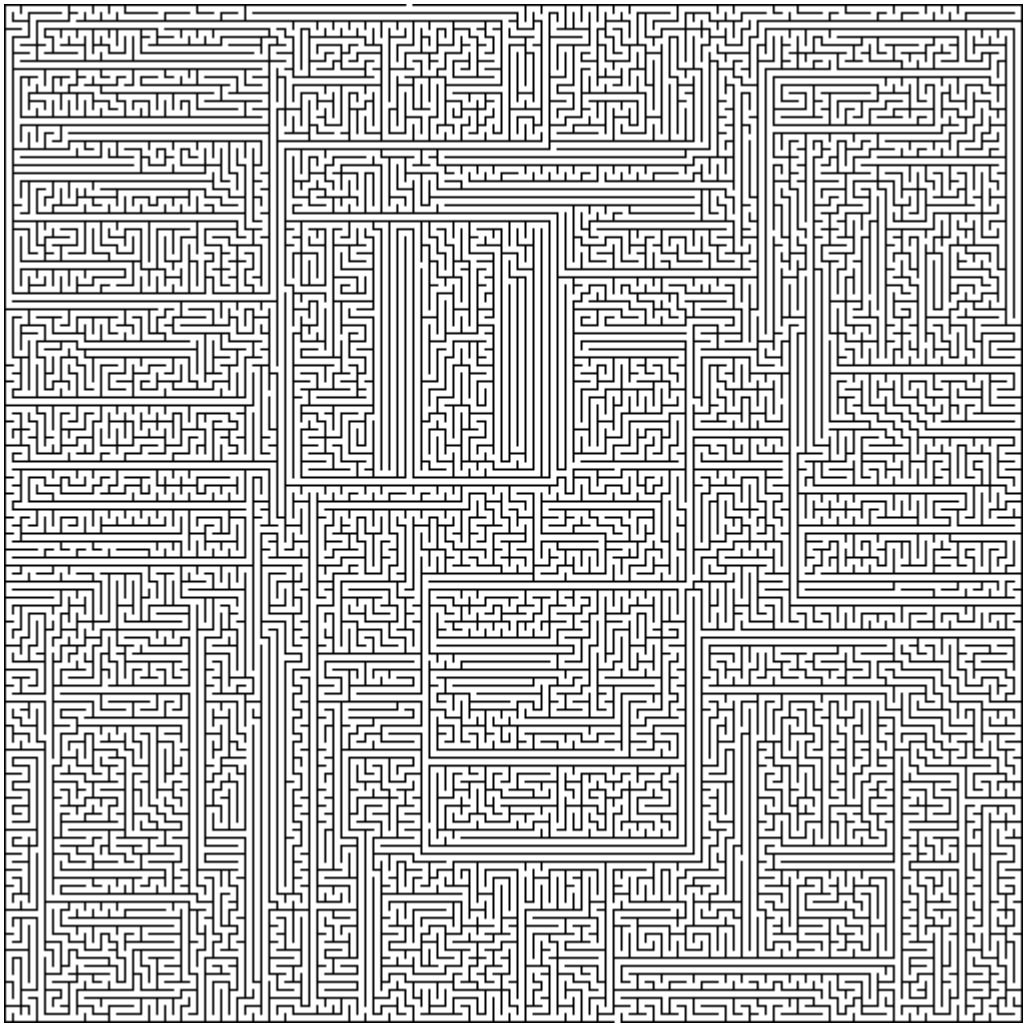
  Разреженный: разреженный лабиринт не прокладывает проходы через каждую ячейку, то есть некоторые из них не создаются. Это подразумевает наличие недостижимых областей, то есть он в некотором смысле противоположен плетёному лабиринту. Похожую концепцию можно применить и при добавлении стен, благодаря чему можно получить неравномерный лабиринт с широкими проходами и комнатами.

- **Частично плетёный:** частично плетёный лабиринт — это смешанный лабиринт, в котором есть и петли, и тупики. Слово «плетёный» можно использовать для количественной оценки, то есть «лабиринт с сильным плетением» — это лабиринт со множеством петель или убранных стен, а в «лабиринте со слабым плетением» их всего несколько.

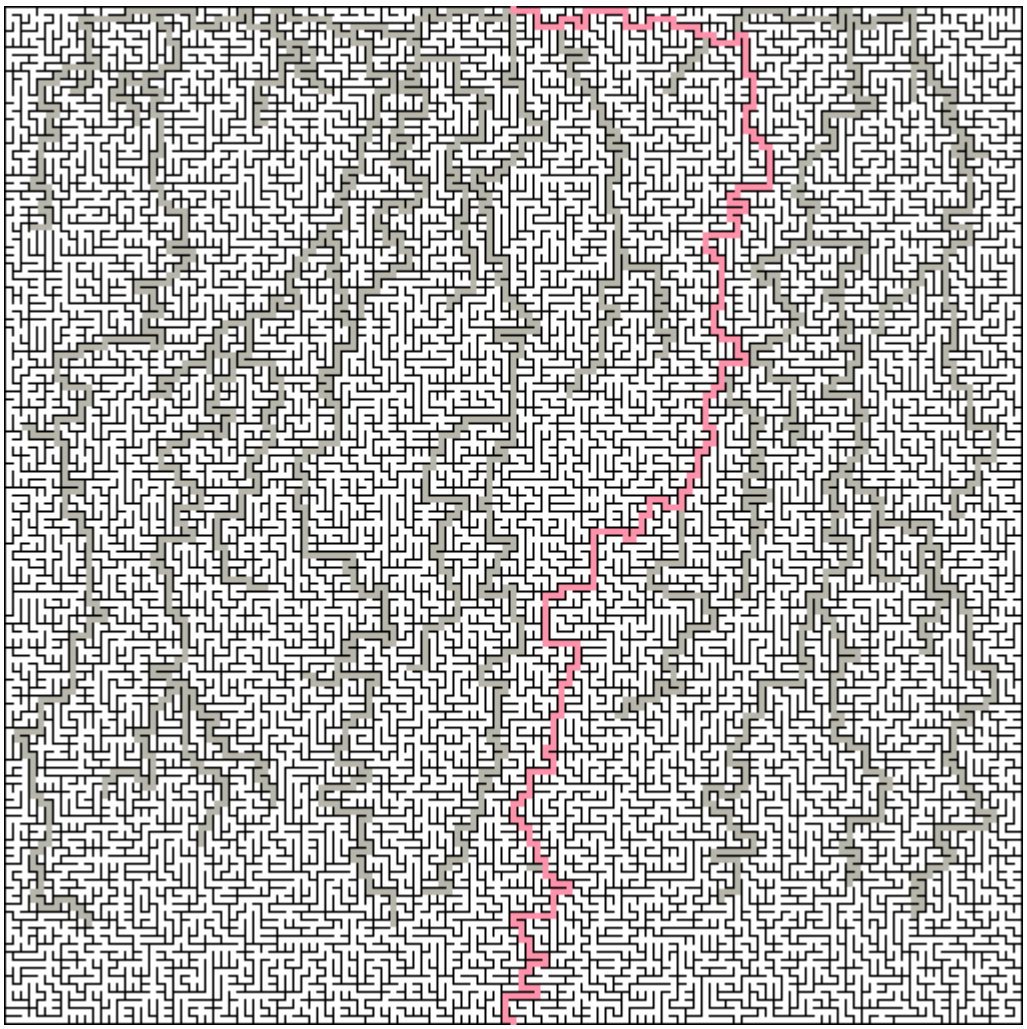
Текстура: классификация по текстуре описывает стиль проходов при различной маршрутизации и геометрии. Текстура — это не просто параметры, которые можно включать или выключать. Вот несколько примеров переменных:

- 

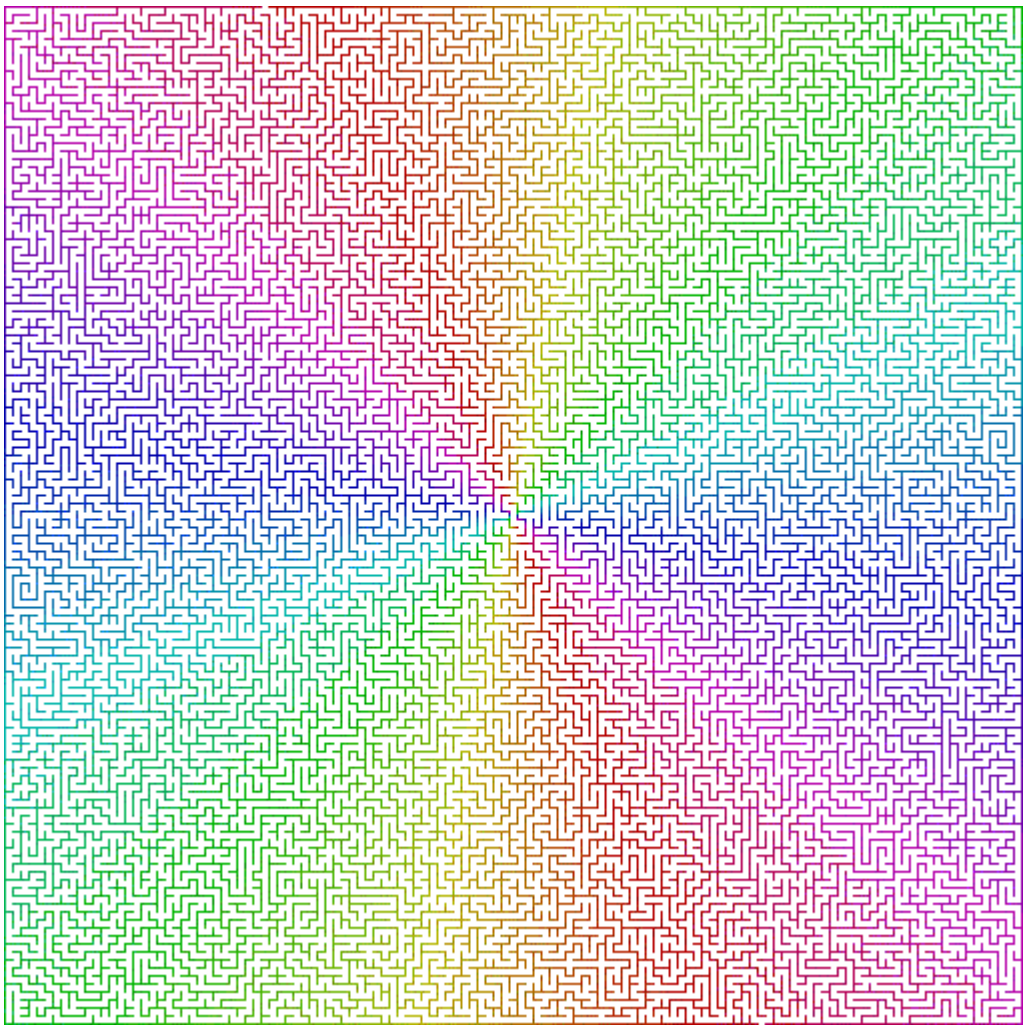
  Смещённость (Bias): в лабиринте со смещёнными проходами прямые проходы склонны больше идти в одном направлении, чем в других. Например, в лабиринте с высокой горизонтальной смещённостью у нас будут длинные проходы слева направо, и только короткие проходы сверху вниз, соединяющие их. Такой лабиринт обычно сложнее проходить «поперёк волокон».

- 

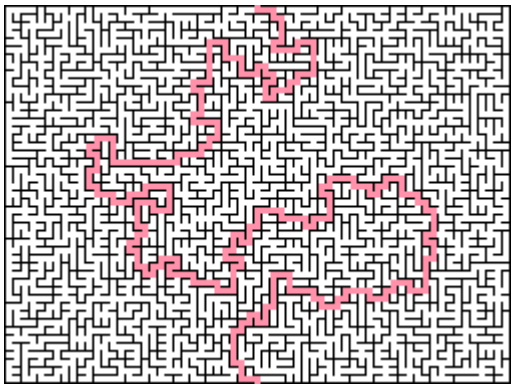
  Пролёты: показатель «пролётов» определяет, как долго будут идти длинные проходы, прежде чем появятся вынужденные повороты. В лабиринте с низким показателем пролётов не будет прямых проходов длиннее трёх-четырёх ячеек, и он будет выглядеть очень случайным. В лабиринте с высоким показателем пролётов на протяжении лабиринта будет большой процент длинных проходов, из-за чего он будет походить на микрочип.

- 

Элитность: показатель «элитности» лабиринта определяет длину решения относительно размера лабиринта. У элитных лабиринтов обычно есть короткое прямое решение, а в неэлитных лабиринтах решение проходит по большой части площади лабиринта. Качественный элитный лабиринт может быть намного сложнее, чем неэлитный.

- 

  Симметрия: симметричный лабиринт имеет симметричные проходы, например, в симметрией вращения относительно центра, или отражённый по горизонтальной или вертикальной осям. Лабиринт может быть частично или полностью симметричным, и может повторять паттерн любое количество раз.

- 

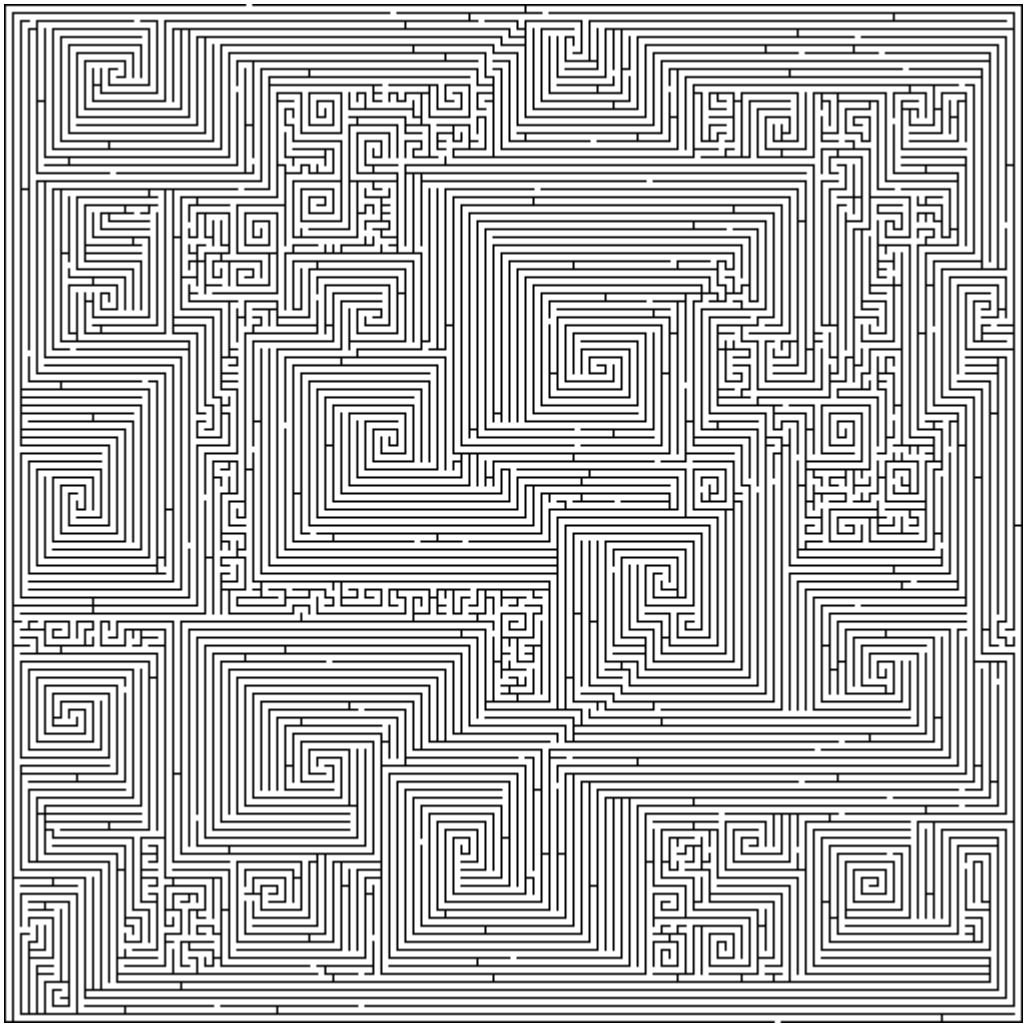
  Однородность: однородный алгоритм генерирует все возможные лабиринты с равной вероятностью. Лабиринт можно назвать имеющим однородную текстуру, если он выглядит как типичный лабиринт, сгенерированный однородным алгоритмом. Неоднородный

алгоритм теоретически тоже может сгенерировать все возможные лабиринты в любом пространстве, но не с равной вероятностью. Неоднородность может зайти и ещё дальше — возможно существование лабиринтов, которые алгоритм никогда не сгенерирует.

- **Текучесть (River):** характеристика «текучесть» означает, что при создании лабиринта алгоритм будет искать и очищать соседние ячейки (или стены) до текущей, то есть будет течь (отсюда и термин «текучесть») в ещё несозданные части лабиринта, как вода. В идеальном лабиринте с меньшим показателем текучести обычно будет множество коротких тупиков, а в более «текучем» лабиринте будет меньше тупиков, но они окажутся длиннее.

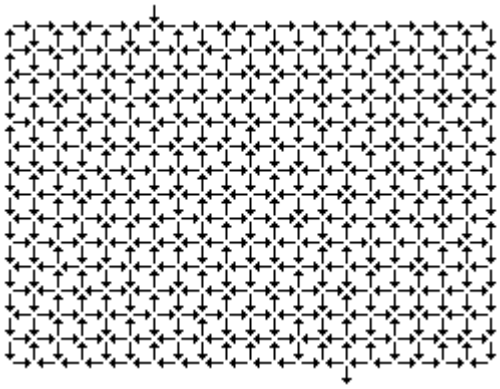
Приоритет: эта классификация показывает, что процессы создания лабиринтов можно разделить на два основных типа: добавляющие стены и вырезающие проходы. Обычно при генерации это сводится только к разнице в алгоритмах, а не к заметным отличиям лабиринтов, но всё равно это полезно учитывать. Один и тот же лабиринт часто генерируется обоими способами:

- **Добавление стен:** алгоритмы, для которых приоритетом являются стены, начинают с пустой области (или внешней границы), в процессе работы добавляя стены. В реальном мире настоящие лабиринты, состоящие из изгородей, перекрытий или деревянных стен, определённо являются добавляющими стены.
- **Вырезание проходов:** алгоритмы, приоритетом которых являются проходы, начинают со сплошного блока и в процессе работы вырезают в нём проходы. В реальном мире такими лабиринтами являются туннели шахт или лабиринты внутри труб.

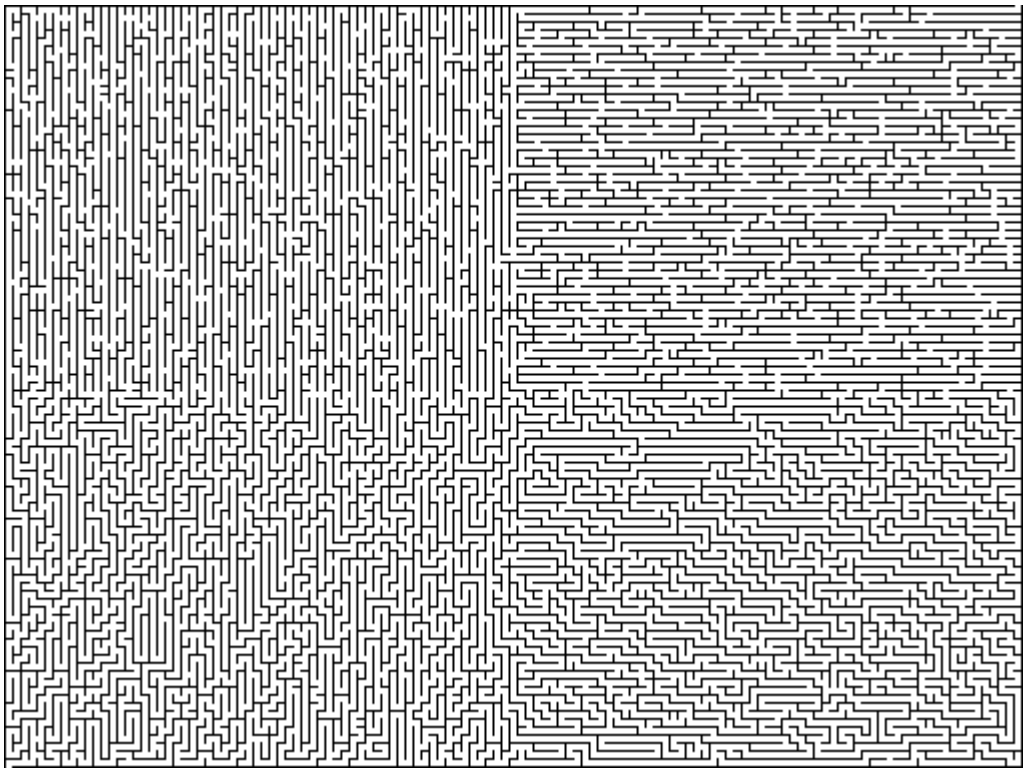
- 

  Шаблон: разумеется, лабиринты могут одновременно и вырезать проходы, и добавлять стены; некоторые компьютерные алгоритмы так и делают. Шаблоном лабиринта называют графическое изображение, не являющееся лабиринтом, которое за наименьшее количество шагов превращается в настоящий лабиринт, но всё равно сохраняет текстуру исходного графического шаблона. Сложные стили лабиринтов, например, пересекающиеся спирали, проще реализовать в компьютере как шаблоны, а не пытаться создать правильный лабиринт, в то же время сохраняя его стиль.

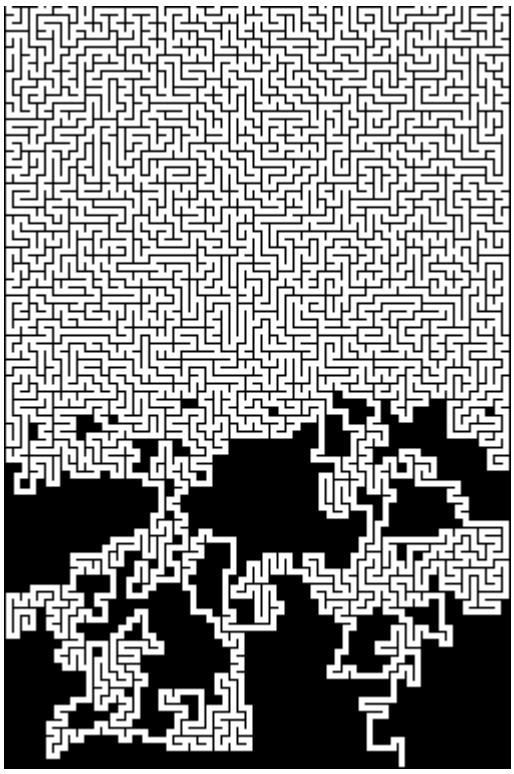
Прочие: описанное выше ни в коем случае не является исчерпывающим списком всех возможных классов или элементов внутри каждого класса. Это только те типы лабиринтов, которые я создавал сам. Заметьте, что почти каждый тип лабиринта, в том числе и лабиринты с особыми правилами, можно выразить в виде ориентированного графа, в котором будет конечное количество состояний и конечное количество вариантов выбора в каждом состоянии, и это называется **эквивалентностью лабиринтов**. Вот несколько других классификаций и типов лабиринтов:

- 

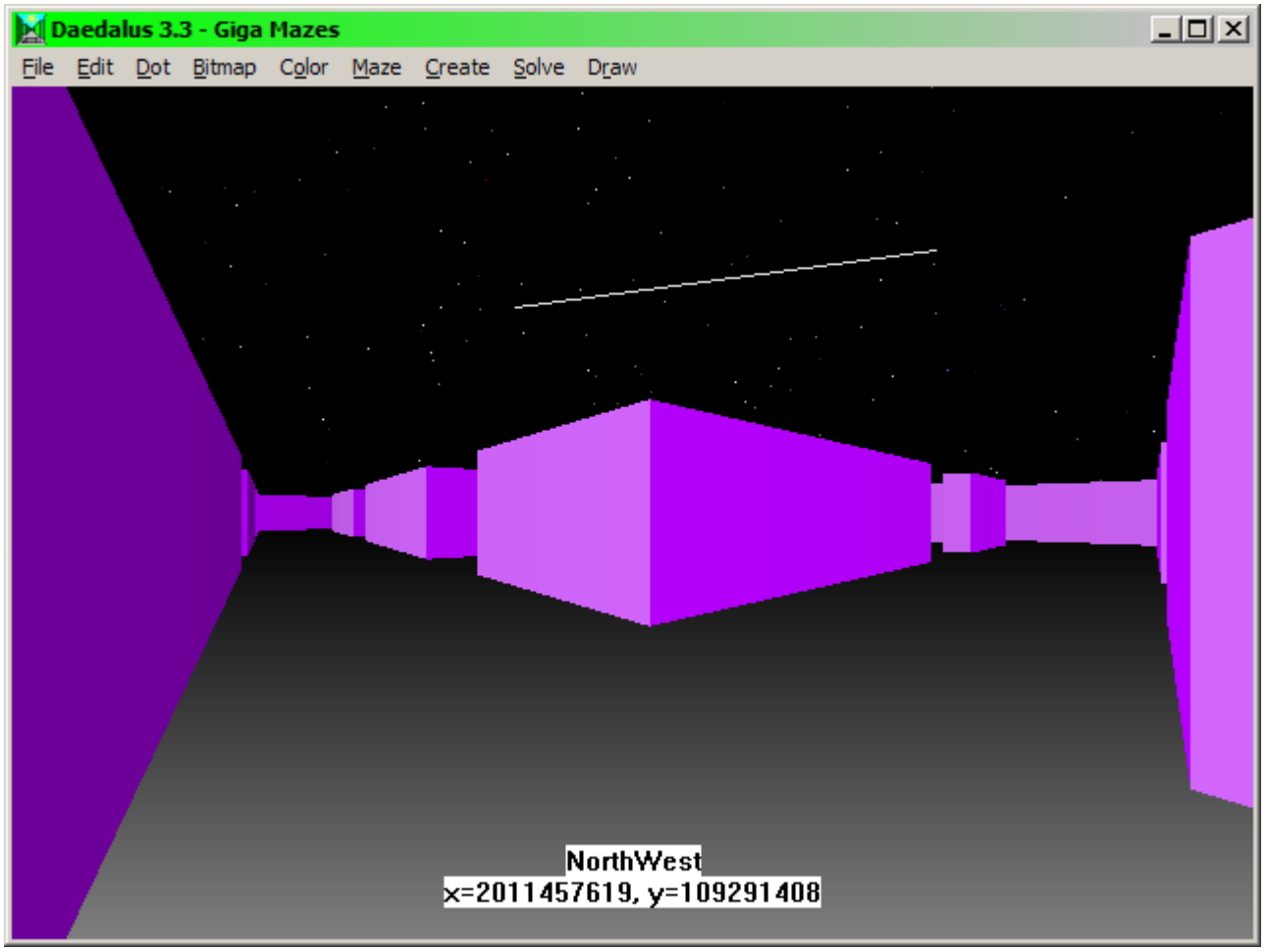
Направленность: по определённым проходам можно перемещаться только в одном направлении. С точки зрения программирования такой лабиринт будет описан ориентированным графом, в отличие от неориентированного графа, описывающего все остальные типы.

- 

Сегментированность: лабиринт может иметь различные части, соответствующие разным классам.

- 

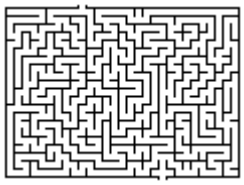
Лабиринты бесконечной длины: мы можем создать бесконечно длинный лабиринт (конечное количество столбцов и любое количество строк), но одновременно хранить в памяти только одну часть лабиринта, «прокручивая» от одного конца до другого и уничтожая предыдущие строки при создании следующих. Один из примеров — это модифицированная версия алгоритма Hunt and Kill. Можно представить потенциально бесконечный лабиринт в виде длинной киноплёнки, составленной из отдельных кадров. Одновременно в памяти хранятся только два последовательных кадра. Запустим алгоритм Hunt and Kill, хотя он и создаёт смещённость, склонную к верхнему кадру, поэтому он завершается первым. После завершения кадр больше не нужен, его можно распечатать или сделать с ним что-то ещё. Как бы то ни было, отбрасываем его, делаем частично созданный нижний кадр новым верхним кадром и очищаем новый нижний кадр. Повторяем процесс, пока не решим остановиться, после чего ждём, пока Hunt And Kill завершит оба кадра. Единственное ограничение заключается в том, что лабиринт никогда не будет иметь пути, разветвляющегося в сторону входа на длину больше двух кадров. Простейший способ создания бесконечного лабиринта — это алгоритм Эллера или алгоритм Sidewinder, потому что они и так создают лабиринты по одной строке за раз, поэтому можно просто позволить им бесконечно добавлять строки к лабиринту.

- 

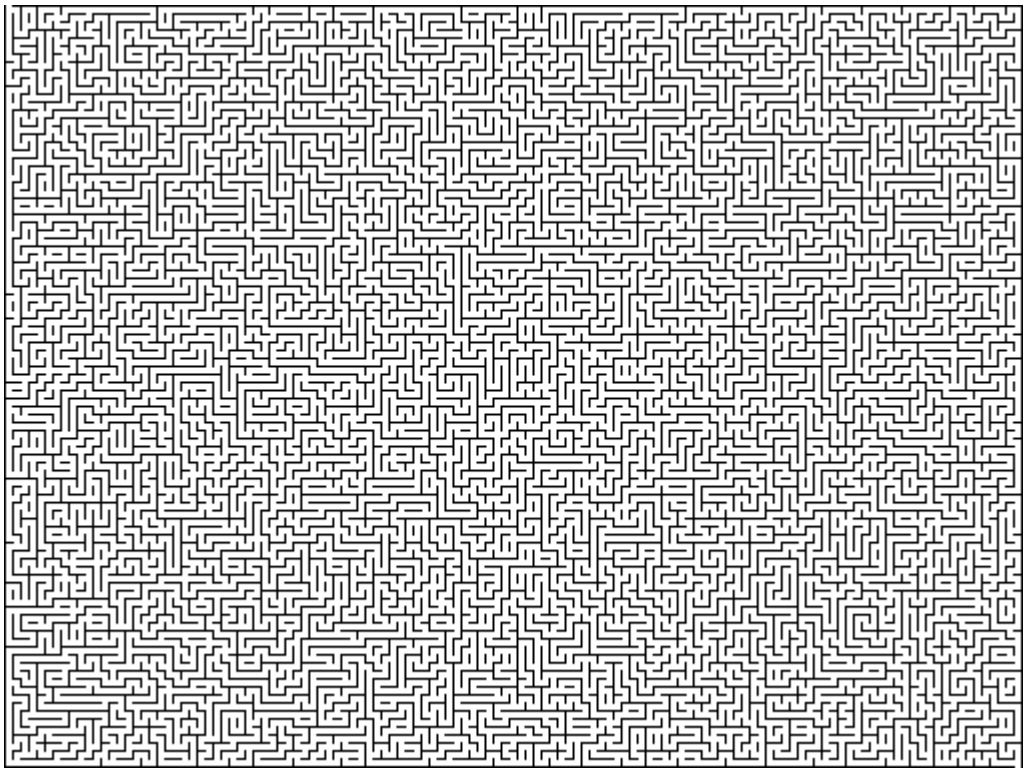
Виртуальные фрактальные лабиринты: виртуальный — это такой лабиринт, в котором весь лабиринт не хранится в памяти одновременно. Например, он может хранить только часть проходов размером 100x100 рядом с вашим местоположением в симуляции, где вы ходите по большому лабиринту. Расширение вложенных фрактальных лабиринтов можно использовать для создания виртуальных лабиринтов огромных размеров, например, миллиард на миллиард проходов. Если бы мы построили реальную копию лабиринта миллиард на миллиард проходов (с расстоянием между проходами в шесть футов), то он бы заполнил поверхность Земли более 6000 раз! Рассмотрим лабиринт из 10^9 на 10^9 проходов, или вложенный лабиринт всего с 9 уровнями. Если мы хотим, чтобы вокруг нас было хотя бы часть размером 100x100, то нам достаточно создать на самом нижнем уровне подлабиринт из 100x100 проходов и семь лабиринтов 10x10, в которые он вложен, чтобы точно знать, где находятся стены в пределах части размером 100x100. (На самом деле лучше иметь четыре соседние части размером 100x100, образующие квадрат на случай, если вы находитесь рядом с краем или углом части, но тут применима та же концепция.) Чтобы обеспечить постоянство и неизменность лабиринта при перемещении по нему у нас есть формула, определяющая случайное порождающее число (seed) для каждой координаты на каждом уровне вложенности. Виртуальные фрактальные лабиринты схожи с фракталом Мандельброта, в изображениях которого он существует виртуально, и нам нужно посетить определённую координату при достаточно высоком увеличении,. чтобы он проявился.

Алгоритмы создания лабиринтов

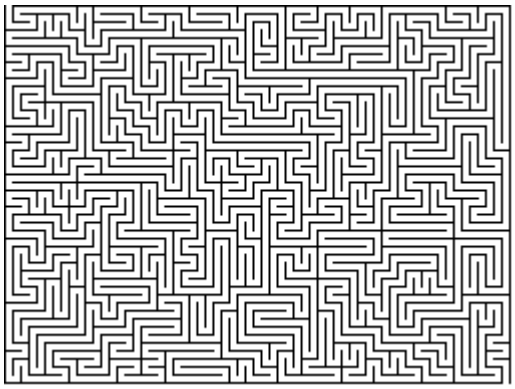
Вот список обобщённых алгоритмов для создания различных классов лабиринтов, описанных выше:

- 

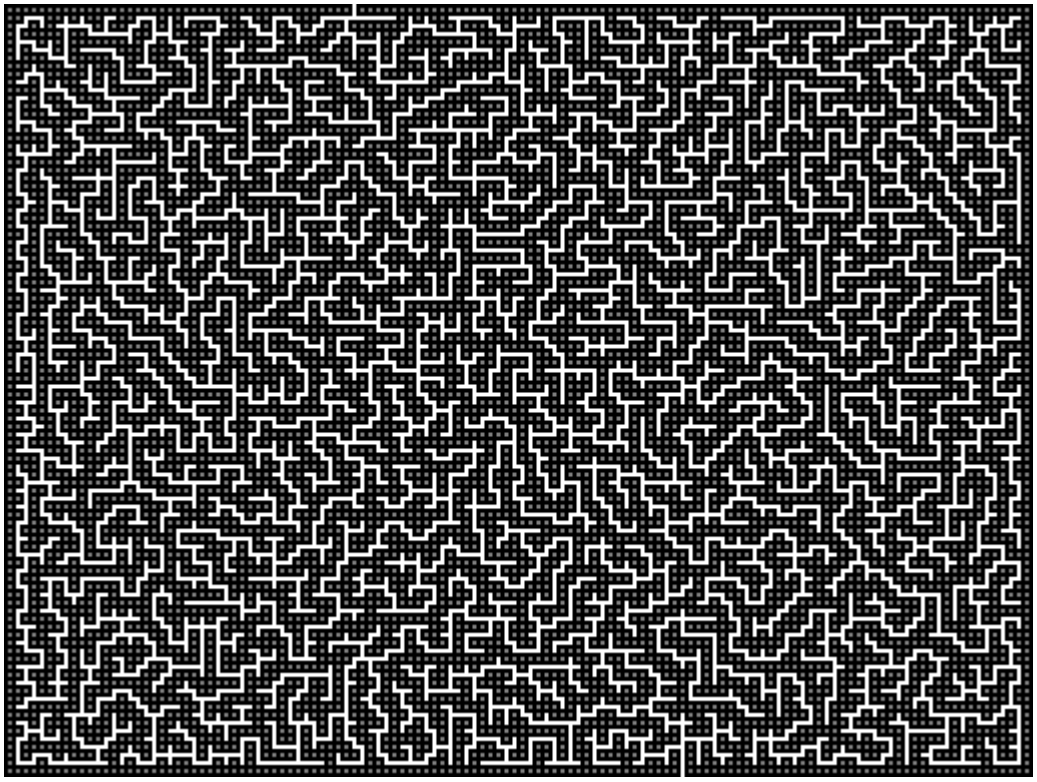
  Идеальный: для создания стандартного идеального лабиринта обычно необходимо «выращивать» его, обеспечив отсутствие петель и изолированных областей. Начинаем с внешней стены и случайным образом добавляем касающийся её фрагмент стены. Продолжаем случайным образом добавлять в лабиринт сегменты стен, но проверяем, что каждый новый сегмент касается с одного конца существующей стены, а его другой конец находится в ещё несозданной части лабиринта. Если вы добавляете сегмент стены, оба конца которого отделены от остального лабиринта, то это создаст несоединённую стену с петлёй вокруг, а если добавить сегмент, оба конца которого касаются лабиринта, то это создать недостижимую область. Это метод добавления стен; почти аналогично ему вырезание проходов, при котором части проходов вырезаются таким образом, чтобы существующего прохода касался только один конец.

- 

Плетёный: для создания лабиринта без тупиков по сути нужно добавлять в лабиринт сегменты стен случайным образом, но делать так, чтобы каждый новый добавляемый сегмент не приводил к созданию тупика. Я создаю их за четыре этапа: (1) начинаю с внешней стены, (2) обхожу лабиринт и добавляю отдельные сегменты стены, касающиеся каждой вершины стены, чтобы в лабиринте не было открытых комнат или небольших стен-«столбов», (3) обхожу все возможные сегменты стен в случайном порядке, добавляя стену там, где она не создаст тупика, (4) или запускаю процедуру удаления изолированных областей в конце, чтобы лабиринт был правильным и имел решение, или поступаю умнее на этапе (3) и делаю так, чтобы стена добавлялась только тогда, когда она не может привести к изолированной области.

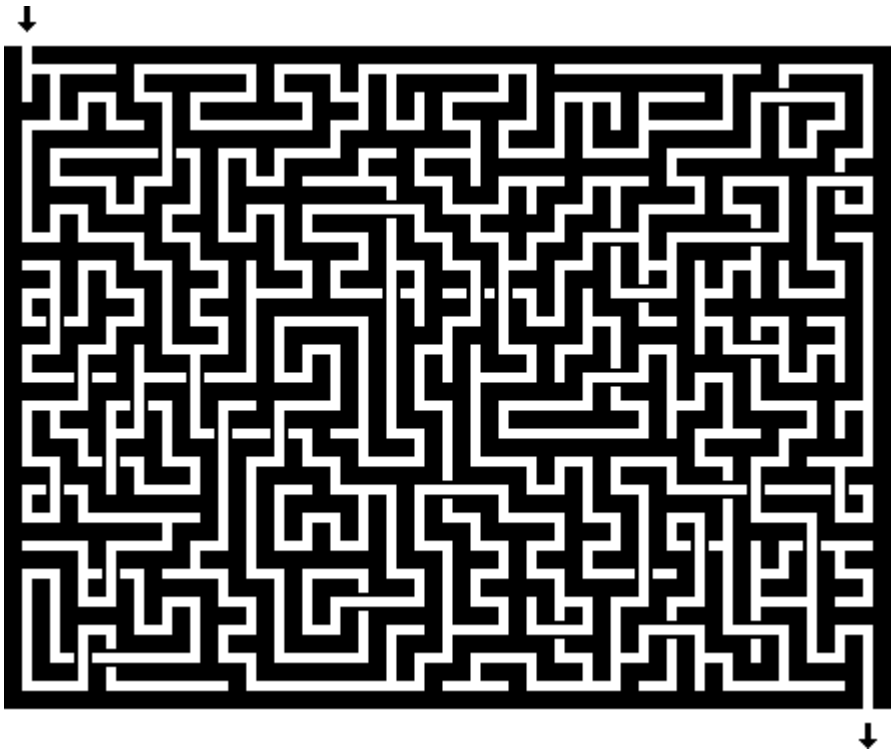
- 

Одномаршрутный: один из способов создания случайного одномаршрутного лабиринта — создать идеальный лабиринт, закрыть выход, чтобы остался только один вход, а затем добавлять стены, разделяющие каждый проход на две части. Это превращает каждый тупик в U-образный поворот, и у нас получится одномаршрутный проход, начинающийся и заканчивающийся в начале исходного лабиринта, который будет следовать по тому же пути, как и перемещение вдоль стены исходного лабиринта. Новый одномаршрутный лабиринт будет иметь удвоенный относительно исходного идеального лабиринта размер. Можно добавить небольшие хитрости, чтобы начало и конец не всегда находились друг рядом с другом: при создании идеального лабиринта мы никогда не будем добавлять сегменты, соединяющиеся с правой или нижней стенами, поэтому получившийся лабиринт будет иметь простое решение, следующее вдоль стены. Поставим вход в правом верхнем углу, и после разбиения пополам для создания одного маршрута удалим правую и нижнюю стены. В результате получится одномаршрутный лабиринт, начинающийся в правом верхнем углу и заканчивающийся в левом нижнем.

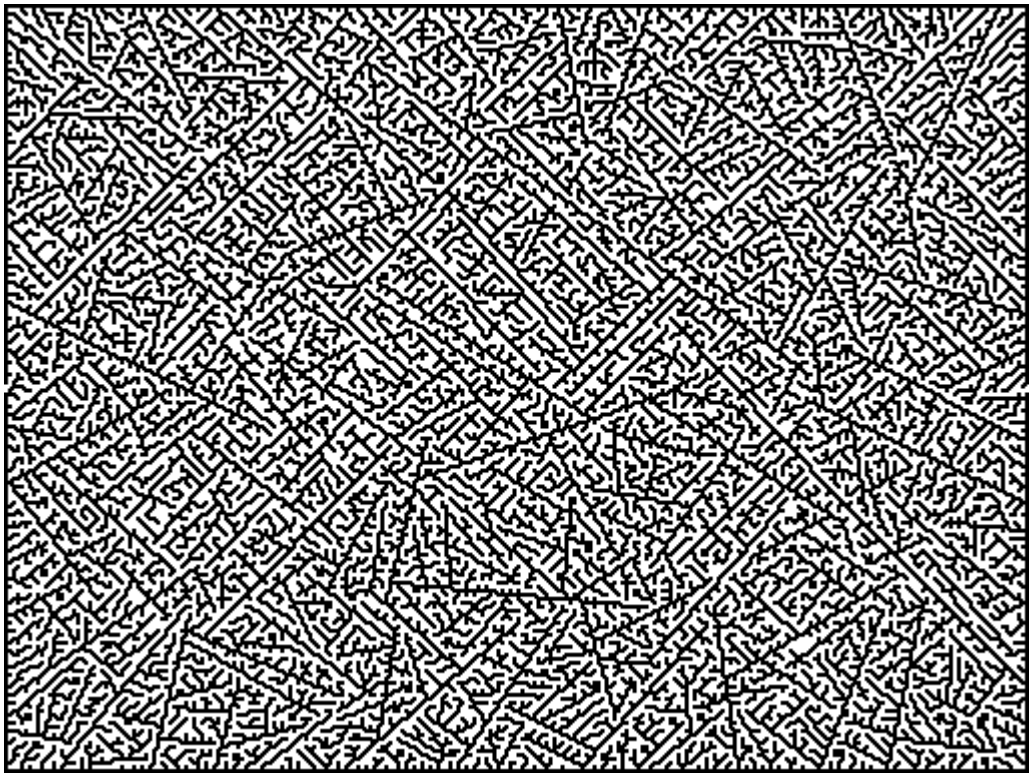
- 

  Разреженный: разреженные лабиринты создаются решением не выращивать лабиринт в областях, которые будут нарушать правило разреженности. Для целостной реализации этого нужно при выборе новой вырезаемой ячейки сначала проверять все ячейки, находящиеся в полукруге выбранного радиуса ячеек, расположенных впереди от текущего направления. Если какая-то из этих ячеек уже является частью лабиринта, то мы не позволяем рассматривать эти ячейки, потому что в противном случае они будут слишком близко к существующей ячейке, а значит, превратят лабиринт в неразреженный.

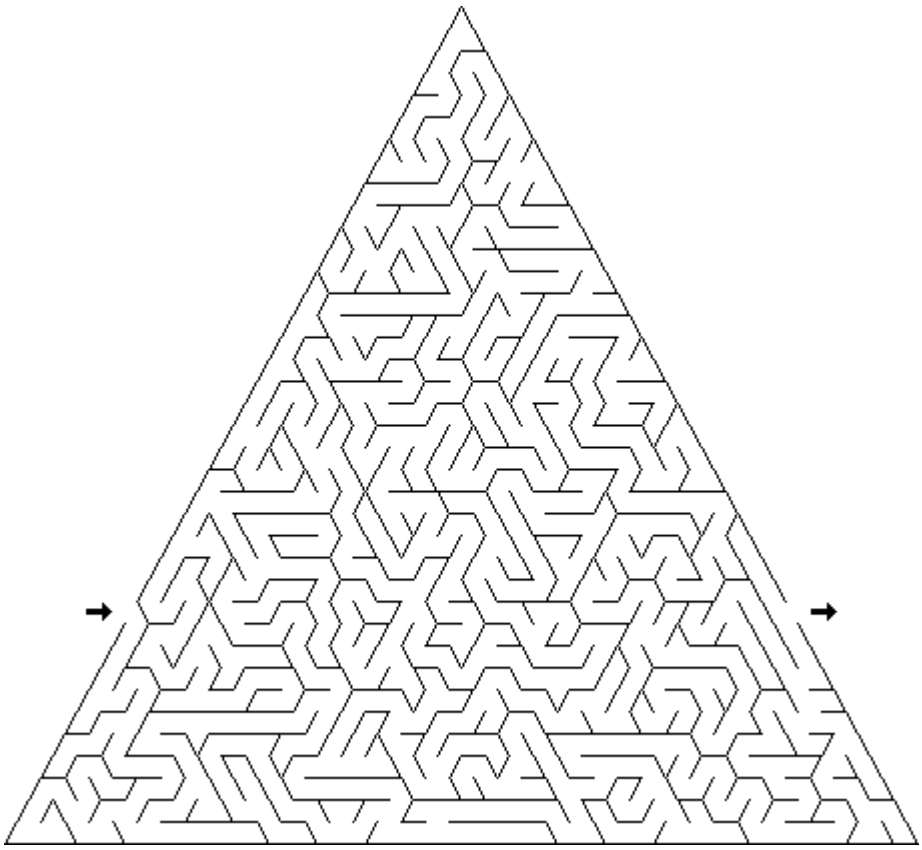
- **3D:** трёхмерные лабиринты и лабиринты большей размерности можно создавать точно так же, как стандартный двухмерный идеальный лабиринт, только из каждой ячейки можно случайным образом двигаться в шесть, а не в четыре ортогональные ячейки. Из-за дополнительных размерностей в этих лабиринтах обычно используется вырезание проходов.



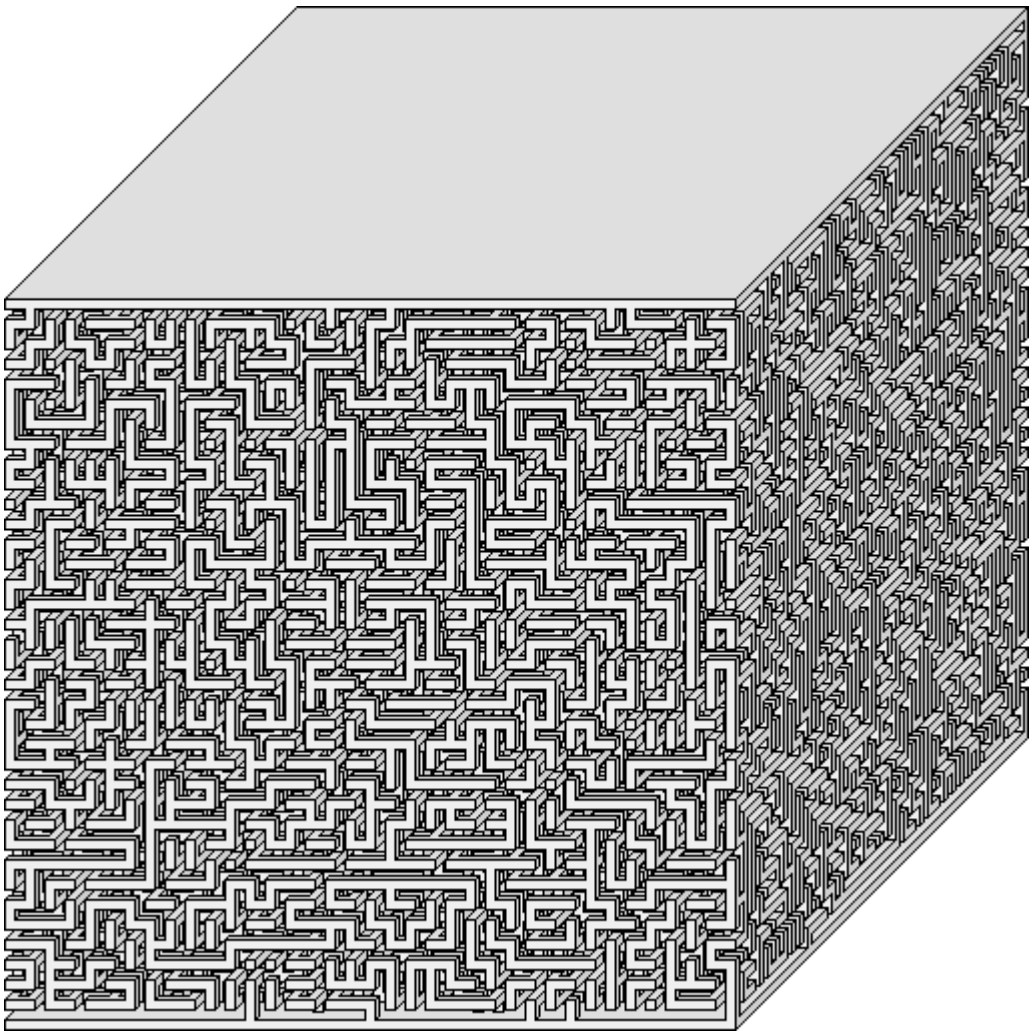
Переплетённый: переплетённые лабиринты по сути создаются как идеальные лабиринты с вырезанием проходов, только при вырезании проходов мы не всегда ограничены существующим проходом, потому что у нас есть возможность пройти под ним и всё равно сохранить идеальность лабиринта. В монохромном растровом изображении переплетённый лабиринт можно представить четырьмя строками на проход (двух строк на проход достаточно для стандартного идеального лабиринта): одна строка для самого прохода, а остальные три строки недвусмысленно показывают, когда другой соседний проход идёт под ним, а не просто образует тупик рядом с первым проходом. Ради эстетичности перед вырезанием под уже готовым проходом можно заглядывать вперёд, чтобы убедиться. что можно продолжить вырезание, находясь под ним; так можно избежать тупиков, расположенных прямо под другими проходами. Также после вырезания под проходом можно инвертировать соседние с пересечением пиксели, чтобы новые проходы проходили над старыми, а не под ними.

- 

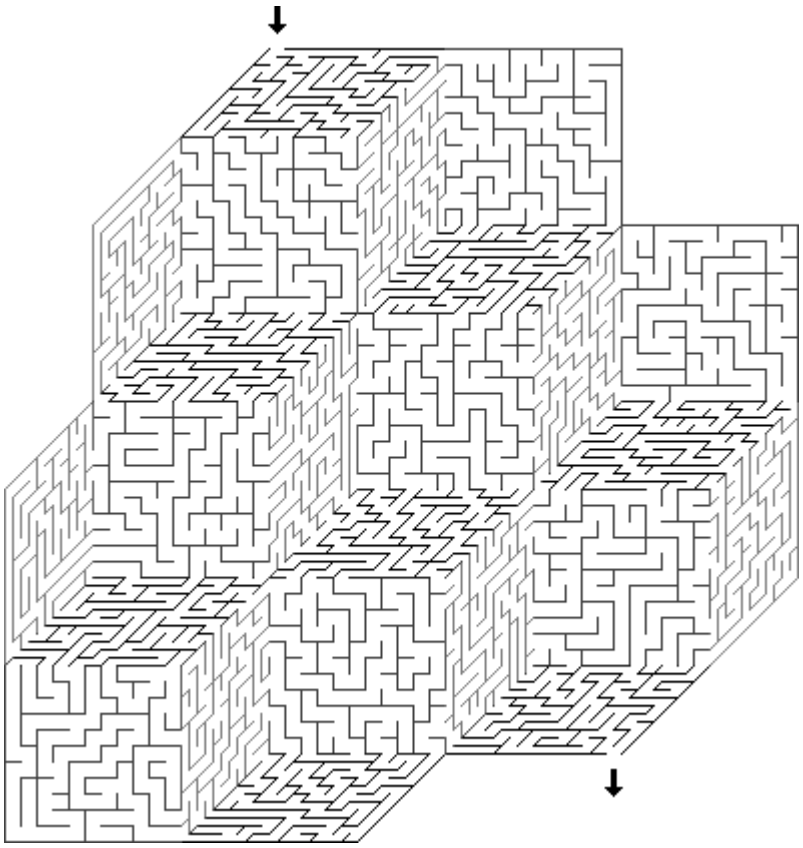
Crack: Crack-лабиринты по сути создаются как идеальные лабиринты с добавлением стен, только в них нет отчётливой тесселяции, за исключением случайного расположения пикселей. Выбираем пиксель, который уже поставлен как стена, выбираем ещё одну случайную локацию, и «стреляем» или начинаем рисовать стену в сторону второй локации. Однако нужно останавливаться, не доходя до уже существующих стен, чтобы не создать изолированности. Завершаем работу, если какое-то время не можем добавить новых значимых стен. Учтите, что случайные локации для рисования могут находится в любом месте лабиринта, поэтому по лабиринту будет проходить несколько прямых линий и множество пропорционально меньших стен; количество стен ограничено только пиксельным разрешением. Это делает лабиринт очень похожим на поверхность листа, то есть по сути это фрактальный лабиринт.

- 

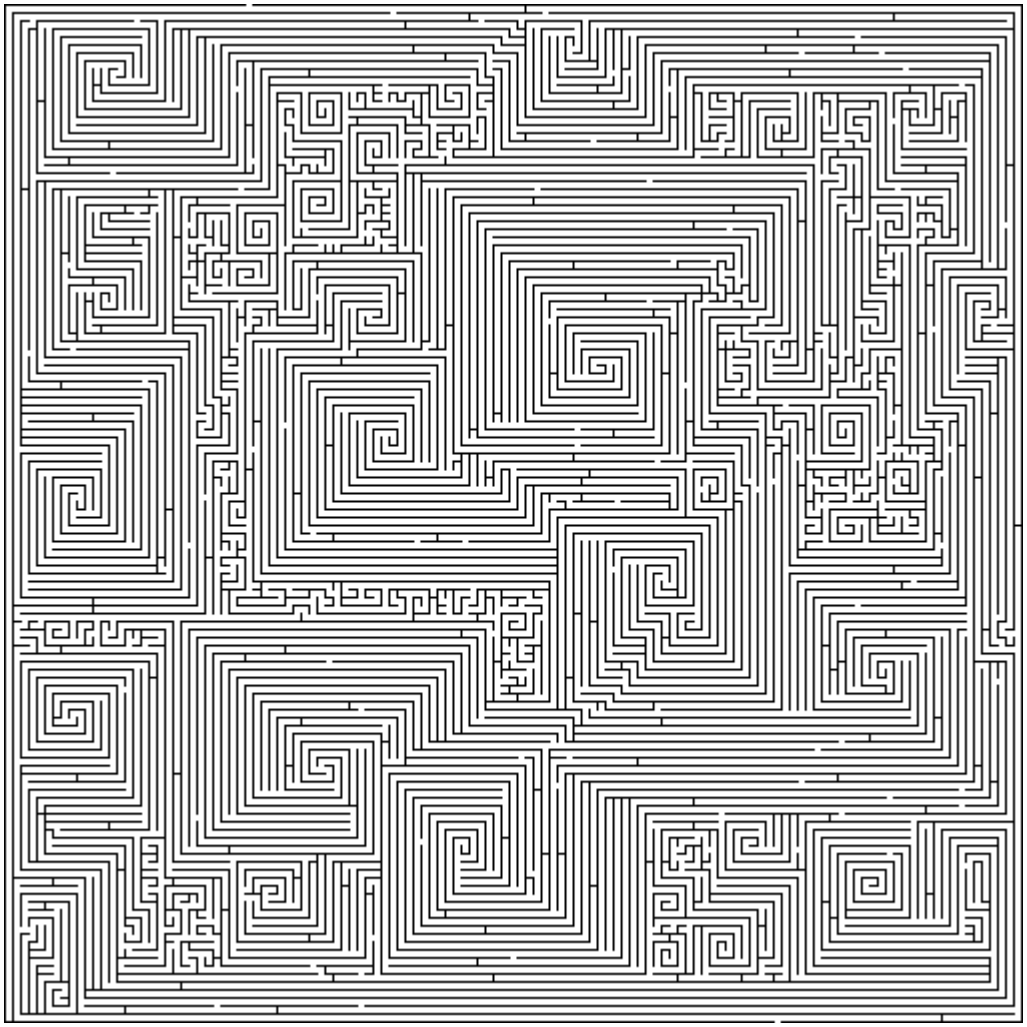
Омега: для лабиринтов в стиле «омега» необходимо задать некую сетку, способ соединения ячеек друг с другом и привязку к экрану вершин, окружающих каждую ячейку. Например, для треугольного дельта-лабиринта с соединяющимися треугольными ячейками: (1) есть сетка, в которой количество ячеек в каждой следующей строке увеличивается на две. (2) Каждая ячейка соединяется с ячейками, соседними с ней в этом ряду, за исключением третьего прохода, который соединён с соответствующей ячейкой строкой выше или ниже, в зависимости от того, нечётный или чётный этот столбец (т.е. смотрил ли треугольник вверх или вниз). (3) Каждая ячейка использует математику треугольников, чтобы определить, где отрисовывать его на экране. Можно заранее отрисовать все стены на экране и вырезать в лабиринте проходы, или хранить в памяти некий изменяемый массив и рендерить всё после завершения.

- 

Гиперлабиринт: гиперлабиринт — это 3D-среда, похожая на обратную версию стандартного трёхмерного не-гиперлабиринта, в котором блоки становятся открытыми пространствами, и наоборот. Хотя стандартный 3D-лабиринт состоит из дерева проходов через сплошную площадь, гиперлабиринт состоит из дерева полос или лиан, проходящих по открытой площади. Для создания гиперлабиринта мы начинаем со сплошных верхней и нижней граней, а затем выращиваем извилистые лианы из этих граней для заполнения пространства между ними, чтобы усложнить прохождение отрезка прямой между этими двумя гранями. Если каждая лиана соединяется с верхней или нижней частью, то гиперлабиринт будет иметь хотя бы одно простое решение. Если ни одна лиана не соединяется и с верхней, и с нижней частями (что создаст непроходимый столбец), то пока в верхней и нижней частях нет петель лиан, которые заставляют их неразрывно соединяться друг с другом подобно цепи, гиперлабиринт будет оставаться решаемым.

- 

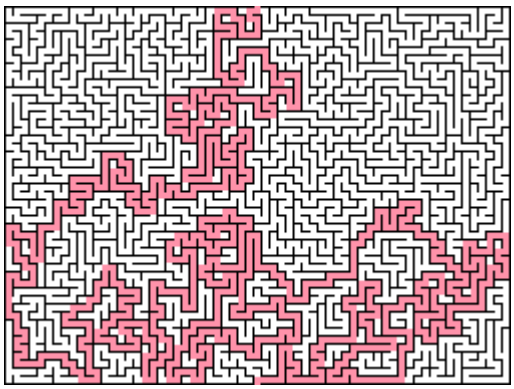
Planair: Planair-лабиринты с необычной топологией обычно создаются как массив из одного или нескольких лабиринтов или частей лабиринтов, в которых определён способ соединения краёв друг с другом. Лабиринт на поверхности куба — это всего лишь шесть квадратных частей лабиринта. Когда создаваемая часть доходит до края, то она перетекает в следующую часть и в правый край.

- 

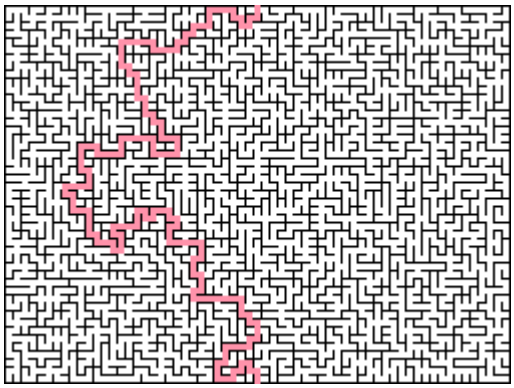
  Шаблон: лабиринты, основанные на шаблонах, создаются, начиная с базового изображения-шаблона, а затем запускается устранитель изолированных областей, обеспечивающий наличие решения лабиринта, за которым следует устранитель петель, повышающий сложность лабиринта. В результате создаётся идеальный лабиринт, очень похожий на исходное изображение. Например, для создания лабиринта, состоящего из пересекающихся спиралей нужно просто создать случайные спирали, не волнуясь о том, являются ли они лабиринтом, а затем пропустить их через устранители изолированных областей и петель.

Алгоритмы создания идеальных лабиринтов

Существует множество способов создания идеальных лабиринтов, и каждый из них имеет собственные характеристики. Ниже представлен список конкретных алгоритмов. Во всех них описано создание лабиринта вырезанием проходов, однако если не указано иное, каждый также можно реализовать добавлением стен:

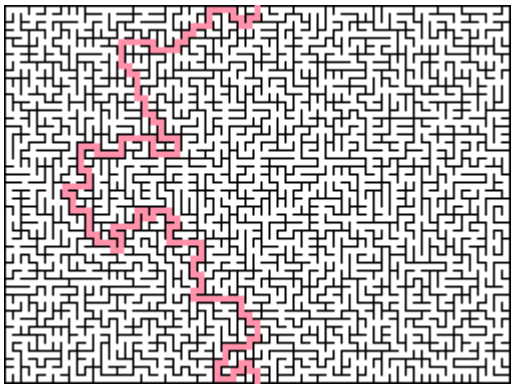
- 

Recursive backtracker: он в чём-то похож на метод решения лабиринтов recursive backtracker, и требует стека, объём которого может доходить до размеров лабиринта. При вырезании он ведёт себя максимально жадно, и всегда вырезает проход в несозданной части, если она существует рядом с текущей ячейкой. Каждый раз, когда мы перемещаемся к новой ячейке, записываем предыдущую ячейку в стек. Если рядом с текущей позицией нет несозданных ячеек, то извлекаем из стека предыдущую позицию. Лабиринт завершён, когда в стеке больше ничего не остаётся. Это 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, и быстрее работают только очень специализированные алгоритмы. Recursive backtracking не может работать с добавлением стен, потому что обычно приводит к пути решения, следующему по внешнему краю, когда вся внутренняя часть лабиринта соединена с границей одним проходом.

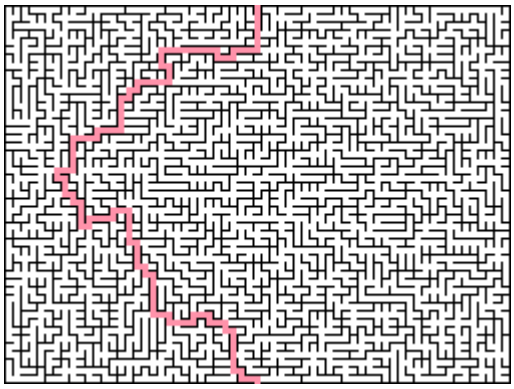
- 

Алгоритм Краскала: это алгоритм, создающий минимальное связующее дерево. Это интересно, потому что он не «выращивает» лабиринт подобно дереву, а скорее вырезает сегменты проходов по всему лабиринту случайным образом, и тем не менее в результате создаёт в конце 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). Помечаем каждую ячейку уникальным идентификатором, а затем обходим все рёбра в случайном порядке. Если ячейки с обеих сторон от каждого ребра имеют разные идентификаторы, то удаляем стену и задаём всем ячейкам с одной стороны тот же идентификатор, что и ячейкам с другой. Если ячейки на обеих сторонах стены уже имеют одинаковый идентификатор, то между ними уже существует

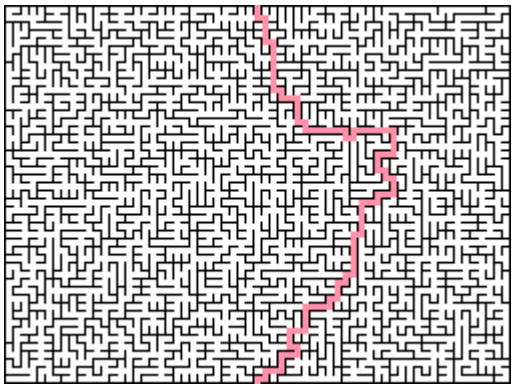
какой-то путь, поэтому стену можно оставить, чтобы не создавать петель. Этот алгоритм создаёт лабиринты с низким показателем текучести, но не таким низким, как у алгоритма Прима. Объединение двух множество по обеим сторонам стены будет медленной операцией, если у каждой ячейки есть только номер и они 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.

- 

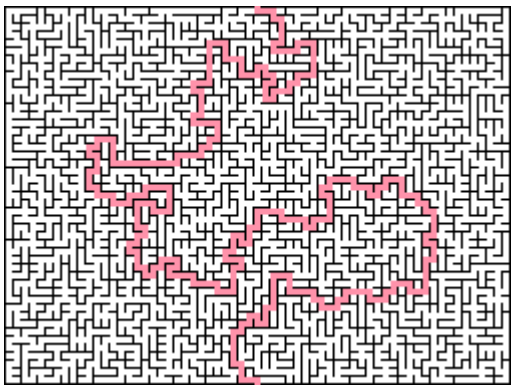
Алгоритм Прима (истинный): этот алгоритм создаёт минимальное связующее дерево, обрабатывая уникально случайные веса рёбер. Объём требуемой памяти пропорционален размеру лабиринта. Начинаем с любой вершины (готовый лабиринт будет одинаковым, с какой бы вершины мы ни начали). Выполняем выбор ребра прохода с наименьшим весом, соединяющим лабиринт к точке, которая ещё в нём не находится, а затем присоединяем её к лабиринту. Создание лабиринта завершается, когда больше не осталось рассматриваемых рёбер. Для эффективного выбора следующего ребра необходима очередь с приоритетом (обычно реализуемая с помощью кучи), хранящая все рёбра границы. Тем не менее, этот алгоритм достаточно медленный, потому что для выбора элементов из обработка кучи требует времени log(n). Поэтому лучше предпочесть алгоритм Краскала, который тоже создаёт минимальное связующее дерево, ведь он быстрее и создаёт лабиринты с идентичной структурой. На самом деле при одинаковом случайном seed алгоритмами Прима и Краскала можно создавать одинаковые лабиринты.

- 

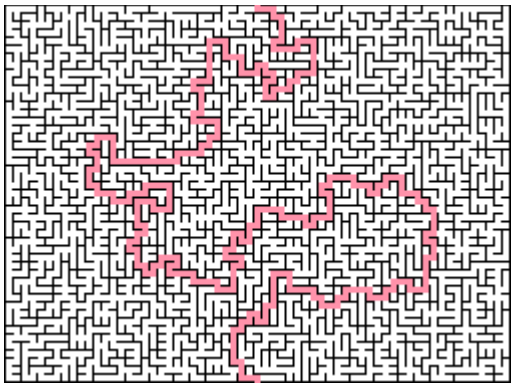
Алгоритм Прима (упрощённый): этот алгоритм Прима создаёт минимальное связующее дерево. Он упрощён таким образом, что все веса рёбер одинаковы. Для него требуется объём памяти, пропорциональный размеру лабиринта. Начинаем со случайной вершины. Выбираем случайным образом ребро прохода, соединяющее лабиринт с точкой, которая ещё не в нём, а затем присоединяем её к лабиринту. Лабиринт оказывается завершённым, когда больше не остаётся рассматриваемых рёбер. Так как рёбра не имеют веса и не упорядочены, их можно хранить как простой список, то есть выбор элементов из списка будет очень быстрым и занимает постоянное время. Поэтому он намного быстрее истинного алгоритма Прима со случайными весами рёбер. Создаваемая текстура лабиринта будет иметь меньший показатель текучести и более простое решение, чем у истинного алгоритма Прима, потому что распространяется из начальной точки равномерно, как пролитый сироп, а не обходит фрагменты рёбер с более высоким весом, которые учитываются позже.

- 

Алгоритм Прима (модифицированный): этот алгоритм Прима создаёт минимальное связующее дерево, изменённое так, что все веса рёбер одинаковы. Однако он реализован таким образом, что вместо рёбер смотрит на ячейки. Объём памяти пропорционален размеру лабиринта. В процессе создания каждая ячейка может иметь один из трёх типов: (1) "внутренняя": ячейка является частью лабиринта и уже вырезана в нём, (2) "граничная": ячейка не является частью лабиринта и ещё не вырезана в нём, но находится рядом с ячейкой, которая уже является «внутренней», и (3) "внешняя": ячейка ещё не является часть лабиринта, и ни один из её соседей тоже не является «внутренней» ячейкой. Начинаем с выбора ячейки, делаем её «внутренней», а для всех её соседей задаём тип «граничная». Выбираем случайным образом «граничную» ячейку и вырезаем в неё проход из одной из соседних «внутренних» ячеек. Меняем состояние этой «граничной» ячейки на «внутреннюю» и изменяем тип всех её соседей с «внешней» на «граничную». Лабиринт завершён, когда больше не остаётся «граничных» ячеек (то есть не осталось и «внешних» ячеек, а значит, все стали «внутренними»). 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, за незначительным отличием: пустоты в связующем дереве заполняются, только если случайным образом выбирается граничная ячейка, в отличие от утроенной вероятности заполнения этой ячейки через одну из граничных ячеек, ведущих в неё. Кроме того, алгоритм очень быстр, быстрее упрощённого алгоритма Прима, потому что ему не нужно составлять и обрабатывать список рёбер.

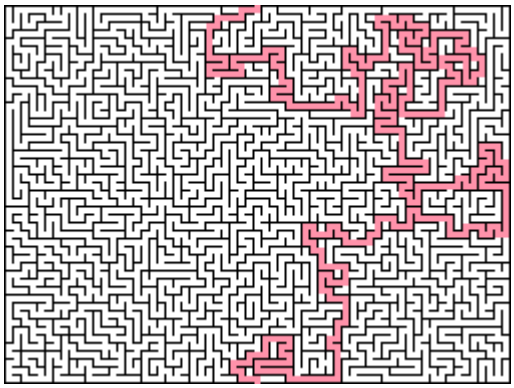
- 

Алгоритм Олдоса-Бродера: интересно в этом алгоритме то, что он однородный, то есть он с равной вероятностью создаёт все возможные лабиринты заданного размера. Кроме того, ему не требуется дополнительной памяти или стека. Выбираем точку и случайным образом перемещаемся в соседнюю ячейку. Если мы попали в невырезанную ячейку, то вырезаем в неё проход из предыдущей ячейки. Продолжаем двигаться в соседние ячейки, пока не вырежем проходы во все ячейки. Этот алгоритм создаёт лабиринты с низким показателем текучести, всего немного выше, чем у алгоритма Краскала. (Это значит, что для заданного размена существует больше лабиринтов с низким показателем текучести, чем с высоким, потому что лабиринт со средней равной вероятностью имеет низкий показатель.) 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. Он может быть реализован как добавляющий стены, если стену границы считать единой вершиной, т.е., например, если ход перемещает нас к стене границы, мы телепортируемся к случайной точке вдоль границы, а уже потом продолжаем двигаться. В случае добавления стен он работает почти в два раза быстрее, потому что телепортация вдоль стены границы позволяет быстрее получать доступ к дальним частям лабиринта.

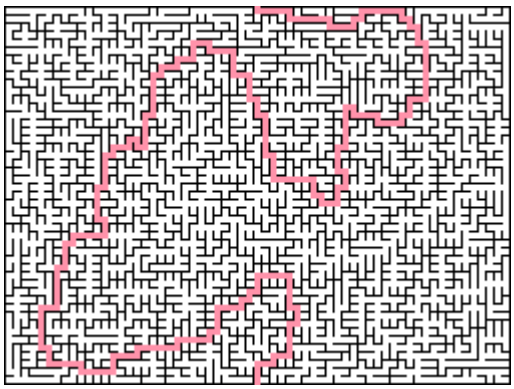
- 

Алгоритм Уилсона: это усовершенствованная версия алгоритма Олдоса-Бродера, он создаёт лабиринты точно с такой же текстурой (алгоритмы однородны, то есть все возможные лабиринты генерируются с равной вероятностью), однако алгоритм Уилсона выполняется гораздо быстрее. Он занимает память вплоть до размеров лабиринта. Начинаем со случайно

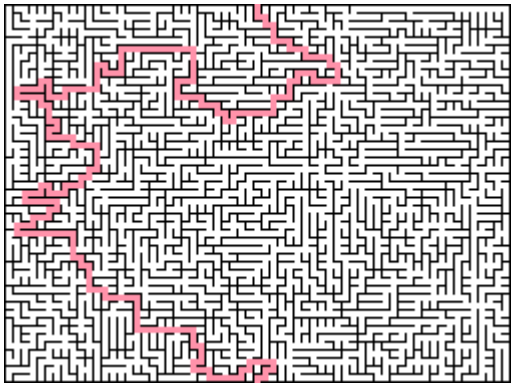
выбранной начальной ячейки лабиринта. Выбираем случайную ячейку, которая ещё не является частью лабиринта и выполняем случайный обход, пока не найдём ячейку, уже принадлежащую лабиринту. Как только мы наткнёмся на уже созданную часть лабиринта,, возвращаемся к выбранной случайной ячейке и вырезаем весь проделанный путь, добавляя эти ячейки к лабиринту. Конкретнее, при возврате по пути мы в каждой ячейке выполняем вырезание в том направлении, в котором проходил случайный обход при последнем выходе из ячейки. Это позволяет избежать появления петель вдоль пути возврата, благодаря чему к лабиринту присоединяется один длинный проход. Лабиринт завершён, когда к нему присоединены все ячейки. Алгоритм имеет те же проблемы со скоростью, что и Олдос-Бродер, 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 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.

- 

Алгоритм Hunt and kill: этот алгоритм удобен, потому что не требует дополнительной памяти или стека, а потому подходит для создания огромных лабиринтов или лабиринтах на слабых компьютерах благодаря невозможности исчерпания памяти. Так как в нём нет правил, которым нужно следовать постоянно, его также проще всего модифицировать и создавать с его помощью лабиринты с разной текстурой. Он почти схож с recursive backtracker, только рядом с текущей позицией нет несозданной ячейки. Мы входим в режим «охоты» и систематично сканируем лабиринт, пока не найдём несозданную ячейку рядом с уже вырезанной ячейкой. На этом этапе мы снова начинаем вырезание в этой новой локации. Лабиринт завершён, когда в режиме «охоты» просканированы все ячейки. Этот алгоритм склонен к созданию лабиринтов с высоким показателем текучести, но не таким высоким, как у recursive backtracker. Можно заставить его генерировать лабиринты с более низким показателем текучести, чаще входя в режим «охоты». Он выполняется медленнее из-за времени, потраченного на охоту за последними ячейками, но не намного медленее, чем алгоритм Краскала. Его можно реализовать по принципу добавления стен, если время от времени случайным образом телепортироваться, чтобы избежать проблем, свойственных recursive backtracker.

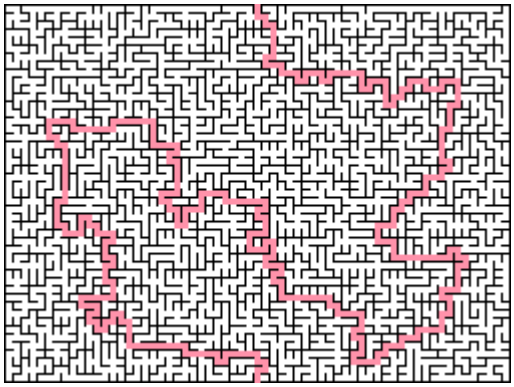
- 

Алгоритм выращивания дерева (Growing tree algorithm): это обобщённый алгоритм, способный создавать лабиринты с разной текстурой. Требуемая память может достигать размера лабиринта. При каждом вырезании ячейки мы добавляем её в список. Выбираем ячейку из списка и вырезаем проход в несозданную ячейку рядом с ней. Если рядом с текущей нет несозданных ячеек, удаляем текущую ячейку из списка. Лабиринт завершён, когда в списке больше ничего нет. 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 backtracker. Если всегда выбирать ячейки случайно, то он ведёт себя похоже, но не идентично алгоритму 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, но короткое и прямое решение. Если случайно выбирать одну из самых последних ячеек, то лабиринт будет иметь низкий показатель текучести, но долгое и извилистое решение.

- 

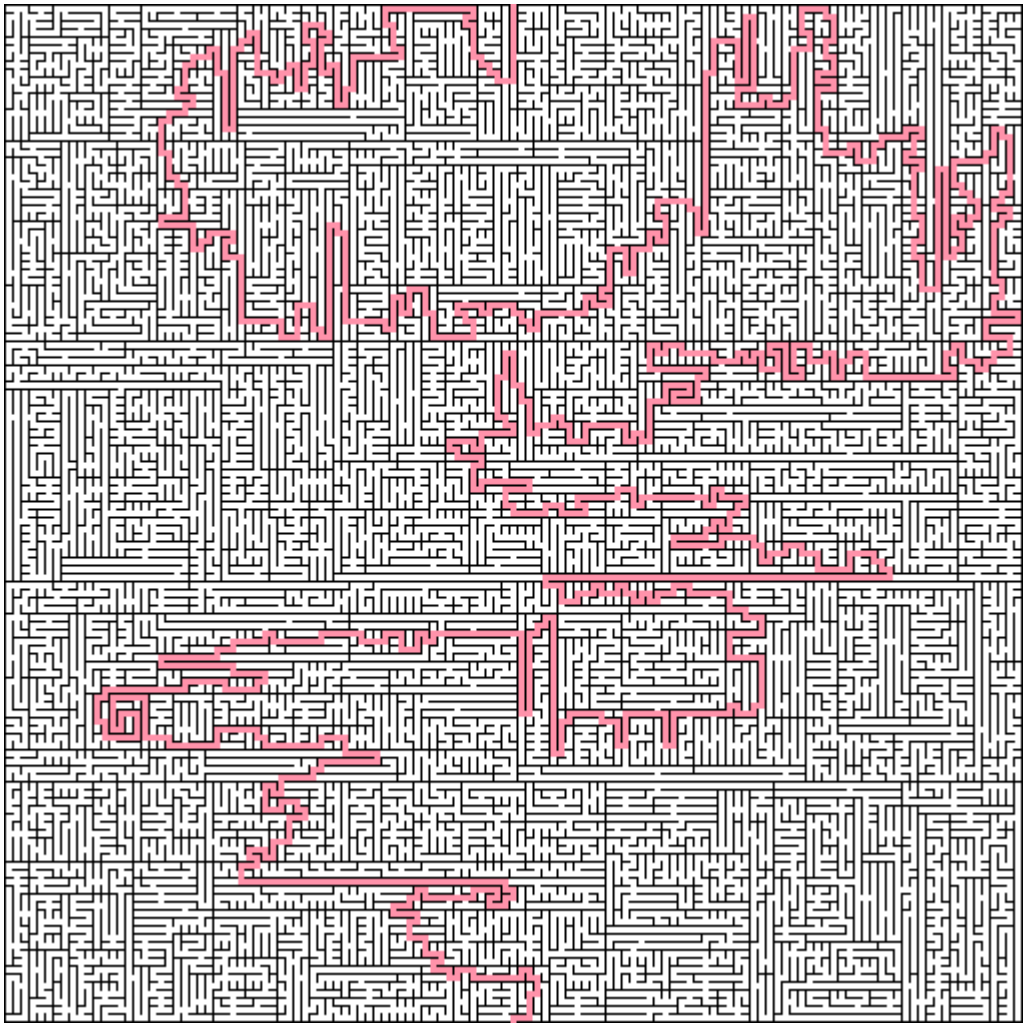
Алгоритм выращивания леса (Growing forest algorithm): это более обобщённый алгоритм, сочетающий в себе типы, основанные на деревьях и множествах. Он является расширением алгоритма выращивания дерева, по сути включающего в себя несколько экземпляров, расширяющихся одновременно. Начинаем со всех ячеек, случайным образом отсортированных в список «новых»; кроме того, у каждой ячейки есть собственное множество, как в начале алгоритма Краскала. Сначлаа выбираем одну или несколько ячеек, перемещая их из списка «новых» в список «активных». Выбираем ячейку из «активного» списка и вырезаем проход в соседнюю несозданную ячейку из «нового» списка, добавляя

новую ячейку в список «активных» и объединяя множества двух ячеек. Если предпринята попытка выполнить вырезание в существующую часть лабиринта, то разрешить её, если ячейки находятся в разных множествах, и объединить ячейки, как это делается в алгоритме Краскала. Если рядом с текущей ячейкой нет несозданных «новых» ячеек, то перемещаем текущую ячейку в список «готовых». Лабиринт завершён, когда становится пустым список «активных». В конце выполняем объединение всех оставшихся множеств, как в алгоритме Краскала. Периодически можно создавать новые деревья, перемещая одну или несколько ячеек из списка «новых» в список «активных», как это делалось в начале. Управляя количество изначальных деревьев и долей новых создаваемых деревьев, можно сгенерировать множество уникальных текстур, сочетающихся с и так уже гибкими параметрами алгоритма выращивания дерева.

- 

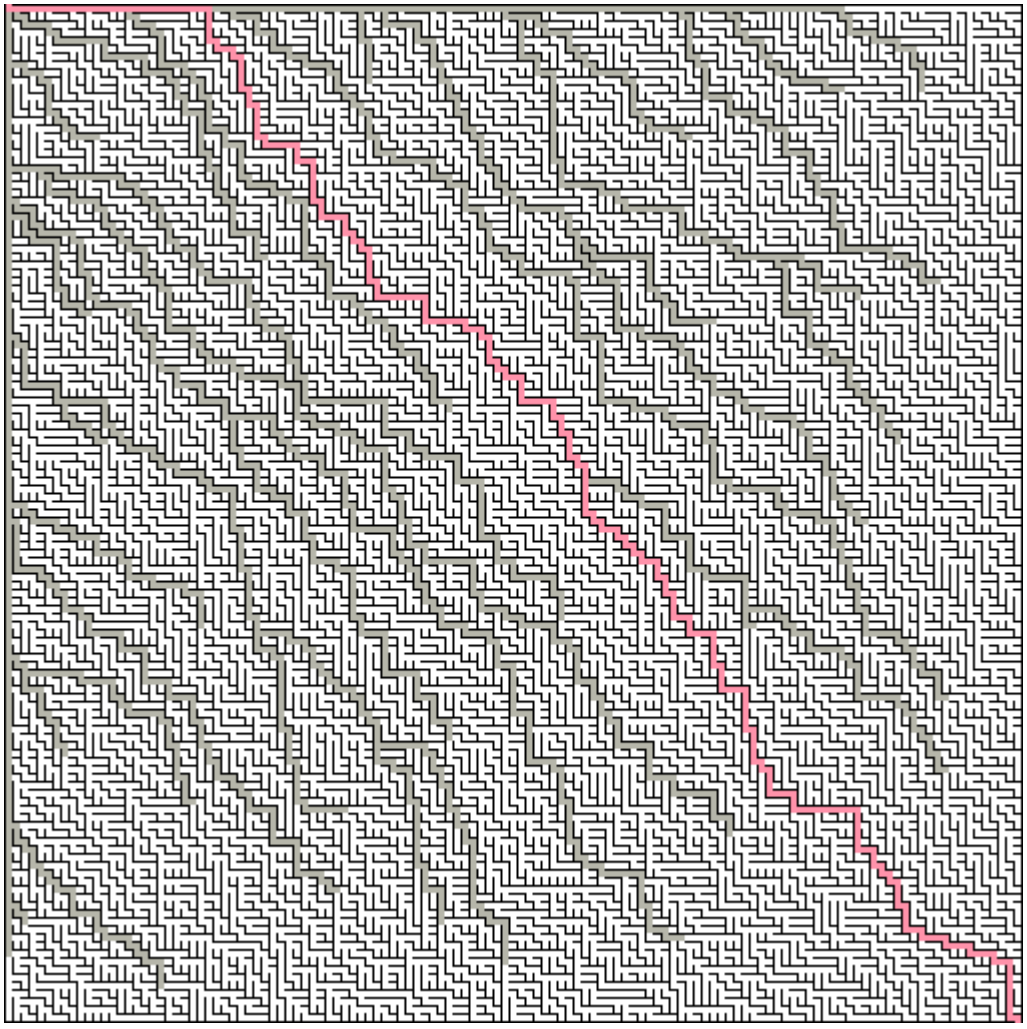
Алгоритм Эллера: это особый алгоритм, потому что он не только быстрее всех остальных, но и не имеет очевидной смещённости или недостатков; кроме того, при его создании память используется наиболее эффективно. Для него даже не требуется, чтобы в памяти находился весь лабиринт, он использует объём, пропорциональный размеру строки. Он создаёт лабиринт построчно, после завершения генерации строки алгоритм больше её не учитывает. Каждая ячейка в строке содержится во множестве; две ячейки принадлежат одному множеству, если между ними есть путь по уже созданному лабиринту. Эта информация позволяет вырезать проходы в текущей строке без создания петель или изолированных областей. На самом деле это довольно похоже на алгоритм Краскала, только здесь формируется по одной строке за раз, в то время как Краскал просматривает весь лабиринт. Создание строки состоит из двух частей: случайным образом соединяем соседние в пределах строки ячейки, т.е. вырезаем горизонтальные проходы, затем случайным образом соединяем ячейки между текущей и следующей строками, т.е. вырезаем вертикальные проходы. При вырезании горизонтальных проходов мы не соединяем ячейки, уже находящиеся в одном множестве (потому что иначе создастся петля), а при вырезании вертикальных проходов мы должны соединить ячейку, если она имеет единичный размер (потому что если её оставить, она создаст изолированную область). При вырезании горизонтальных проходов мы соединяем ячейки, если они находятся в одинаковом множестве (потому что теперь между ними есть путь), а при вырезании вертикальных проходов когда не соединяемся с ячейкой, помещаем её в отдельное множество (потому что теперь она отделена от остальной части лабиринта). Создание начинается с того, что перед соединением ячеек в первой строке каждая ячейка имеет собственное множество. Создание завершается после соединения ячеек в последней строке. Существует особое правило завершения: к моменту завершения каждая ячейка должна находиться в одинаковом множестве, чтобы избежать изолированных

областей. (Последняя строка создаётся объединением каждой из пар соседних ячеек, ещё не находящихся в одном множестве.) Лучше всего реализовывать множество с помощью циклического двусвязного списка ячеек (который может быть просто массивом, привязывающим ячейки к парам ячеек с обеих сторон того же множества), позволяющего выполнять за постоянное время вставку, удаление и проверку нахождения соседних ячеек в одном множестве. 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.

- 

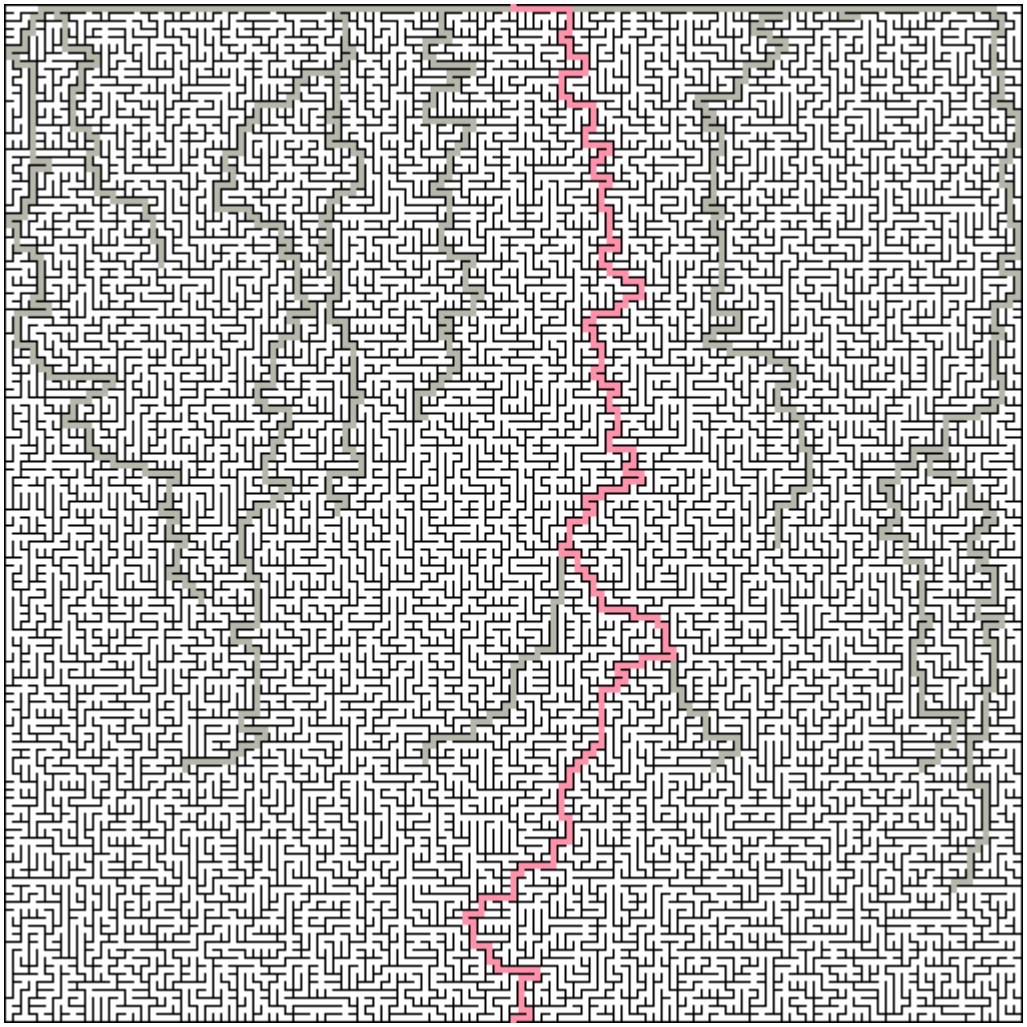
Рекурсивное деление (Recursive division): этот алгоритм чем-то похож на recursive backtracking, потому что в них обоих применяются стеки, только он работает не с проходами, а со стенами. Начинаем с создания случайной горизонтальной или вертикальной стены, пересекающей доступную область в случайной строке или столбце, и размещаем вдоль неё случайным образом пустые места. Затем рекурсивно повторяем процесс для двух подобластей, сгенерированных разделяющей стеной. Для наилучших результатов нужно добавить отклонение в выборе горизонтали или вертикали на основе пропорций области, например, область, ширина которой вдвое больше высоты, должна более часто 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. Этот алгоритм является разновидностью вложенных фрактальных лабиринтов, только вместо постоянного создания лабиринтов ячеек фиксированного размера с лабиринтами одного размера внутри каждой ячейки, он случайным образом делит заданную область на лабиринт случайного размера: 1x2 или 2x1. 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.

- 

Лабиринты на основе двоичных деревьев: по сути, это самые простые и быстрые из возможных алгоритмов, однако создаваемые лабиринты имеют текстуру с очень высокой смещённостью. Для каждой ячейки мы вырезаем проход или вверх, или влево, но никогда не в обоих направлениях. В версии с добавлением стен для каждой вершины добавляется сегмент стены, ведущий вниз или вправо, но не в обоих направлениях. Каждая ячейка независима от всех других ячеек, потому что нам не нужно при её создании проверять состояние каких-то других ячеек. Следовательно, это настоящий алгоритм генерации лабиринтов без памяти, не ограниченный по размерам создаваемых лабиринтов. По сути, это двоичное дерево, если рассматривать верхний левый угол как корень, а каждый узел или ячейка имеет один уникальный родительский узел, являющийся ячейкой сверху или слева от

неё. 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. Учтите, что если перевернуть лабиринт двоичного дерева вниз головой и считать проходы стенами, и наоборот, то результатом по сути будет другое двоичное дерево.

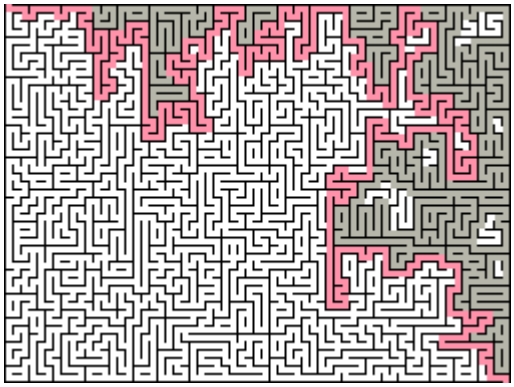
- 

Лабиринты Sidewinder: этот простой алгоритм очень похож на алгоритм двоичного дерева, но немного более сложен. Лабиринт генерируется по одной строке за раз: для каждой ячейки случайным образом выбирается, нужно ли вырезать проход, ведущий вправо. Если проход не вырезан, то мы считаем, что только что завершили горизонтальный проход, образованный текущей ячейкой и всеми ячейками слева, вырезавшими проходы, ведущими в неё. Случайным образом выбираем одну ячейку вдоль этого прохода и вырезаем проход, ведущий из неё вверх (это должна быть текущая ячейка, если соседняя ячейка не вырезала проход). 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 детерминированно 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 очень прямым, но рядом с ним есть множество длинных ошибочных путей, ведущих сверху вниз.

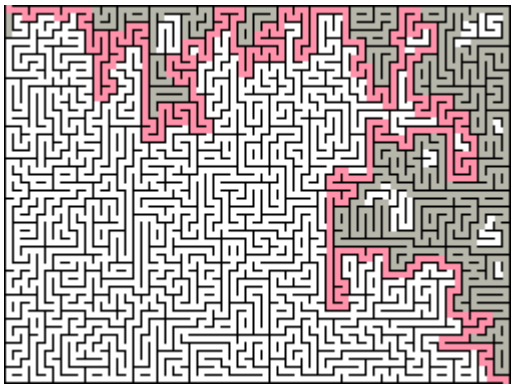
Алгоритмы решения лабиринтов

Существует множество способов решения лабиринтов, и каждый из них имеет собственные характеристики. Вот список конкретных алгоритмов:

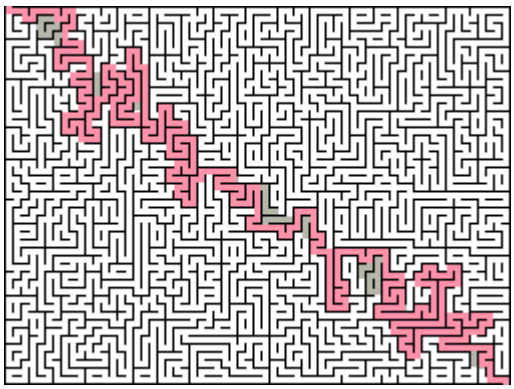
- 

  Следование вдоль стен (Wall follower): это простой алгоритм решения лабиринтов. Приоритетом для него является проходящий лабиринт объект («вы»), он всегда очень быстр и не использует дополнительной памяти. Начинаем идти по проходам и при достижении развилки всегда поворачиваем направо (или всегда налево). Чтобы применить такое решение лабиринта в реальном мире, нужно положить руку на правую (или левую) стену и постоянно держать её на стене в процессе прохождения лабиринта. При желании можно помечать уже посещённые ячейки и ячейки, посещённые дважды. В конце можно вернуться назад по решению, следуя только по ячейкам, посещённым один раз. Этот метод необязательно найдёт кратчайшее решение, и он совершенно не работает, если цель находится в центре лабиринта и его окружает замкнутая цепь, потому что вы будете ходить вокруг центра и со временем придёте к началу. Следование вдоль стены в 3D-лабиринте можно реализовать детерминированным способом, спроецировав 3D-проходы на 2D-плоскость, т.е.

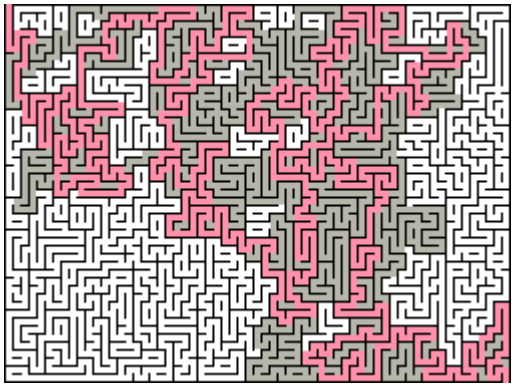
притворившись, что ведущие вверх проходы на самом деле ведут на северо-запад, а ведущие вниз ведут на юго-восток, а затем применить обычные правила следования вдоль стен.

- 

Алгоритм Пледжа: это модифицированная версия следования вдоль стены, способная перепрыгивать между «островами» для решения тех лабиринтов, которые не способно следование вдоль стен. Это гарантированный способ достижения выхода на внешнем крае любого 2D-лабиринта из любой точки внутри, однако он не способен выполнить обратную задачу, т.е. найти решение внутри лабиринта. Он отлично подходит для реализации с помощью сбегающего из лабиринта робота, потому что он сможет выбраться из любого лабиринта, не помечая и не запоминая ни каким образом путь. Начинаем с выбора направления и при возможности всегда движемся в этом направлении. Упёршись в стену, начинаем следовать вдоль неё, пока не сможем снова пойти в выбранном направлении. Стоит заметить, что следование вдоль стены нужно начинать с дальней стены, в которую мы упёрлись. Если в этом месте проход делает поворот, то это может привести к развороту посередине прохода и возврату тем же путём, которым мы пришли. При следовании вдоль стены считаем количество сделанных поворотов, например, поворот налево — это -1, а поворот направо — это 1. Прекращаем следование вдоль стен и начинаем двигаться в выбранном направлении только тогда, когда общая сумма сделанных поворотов равна, т.е. если вы повернулись на 360 градусов и более, то продолжаем следовать вдоль стены пока не «распутаемся». Подсчёт гарантирует, что рано или поздно мы достигнем дальней части «острова», в котором находимся в данный момент, и перепрыгнем на следующий остров в выбранном направлении, после чего продолжим прыгать между островами, пока не упрёмся в стену границы, после чего следование вдоль стен приведёт нас к выходу. Стоит учесть, что алгоритм Пледжа может заставить нас посетить проход или начальную точку несколько раз, однако в последующие разы вы всегда будете иметь другую сумму поворотов. Без разметки пути единственный способ узнать, что лабиринт нерешаем — постоянное увеличение суммы поворотов, хотя в решаемых лабиринтах со спиральным прохождением сумма поворотов тоже может достигать больших значений.

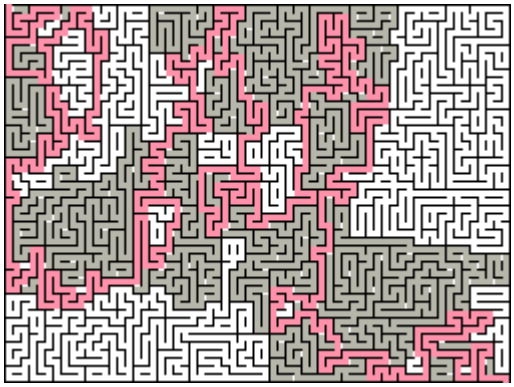
- 

  Алгоритм цепей: алгоритм цепей (Chain algorithm) решает лабиринт, воспринимая его как множество лабиринтов меньшего размера (подобно звеньям цепи) и решая их по порядку. Мы должны указать нужные места начала и конца, и алгоритм всегда найдёт путь от начала до конца, если он существует. При этом решение склонно быть разумно коротким, если даже не кратчайшим. Это означает, что таким способом нельзя решать лабиринты, в которых неизвестно точное расположение конца. Он наиболее похож на алгоритм Пледжа, потому что по сути это алгоритм следования вдоль стен со способом перепрыгивания между островами. Начинаем с рисования прямой линии (или хотя бы линии без самопересечений) от начала до конца, позволяя ей при необходимости пересекать стены. Затем просто следуем по линии от начала до конца. Если мы натыкаемся на стену, то не можем пройти через неё, а значит должны обходить. Отправляем двух следующих вдоль стен «роботов» по каждому из направлений вдоль стены, на которую наткнулись. Если робот снова пересечётся с указующей линией в той точке, где она ближе к выходу, тогда останавливаемся и следуем по этой стене, пока не доберёмся до неё сами. Продолжаем следовать по линии и повторять процесс, пока не достигнем конца. Если оба робота вернутся к их исходным локациям и направлениям, то дальнейшие точки вдоль прямой недостижимы и лабиринт нерешаем.

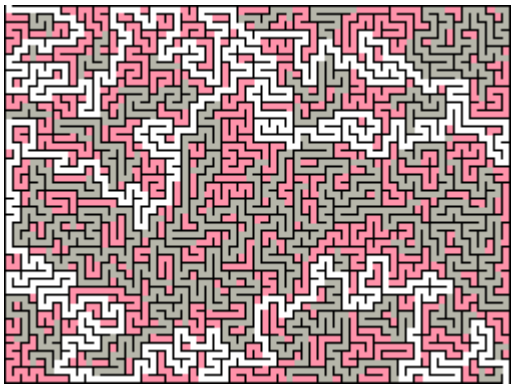
- 

  Recursive backtracker: этот алгоритм находит решение, но оно необязательно будет кратчайшим. Приритетом для него является проходящий лабиринт объект, он быстр для всех типов лабиринтов и использует стек размером вплоть до размера лабиринта. Он очень прост: если мы находимся возле стены (или в помеченной линией области), то возвращаем «неудача», иначе, если мы находимся в конце, возвращаем «успех», иначе, рекурсивно пробуем двигаться во всех четырёх направлениях. Рисуем линию, когда пытаемся пройти в новом направлении и удаляем её, если возвращено значение «неудача»; после достижения состояния «успех» у нас будет нанесено линиями единственное решение. При возврате назад

(backtracking) лучше помечать пространство особым значением посещённых мест, чтобы мы не посещали их снова, приходя с другого направления. По сути, в программировании это называется поиском в глубину. Этот метод всегда находит решение, если оно существует, но оно необязательно будет самым коротким.

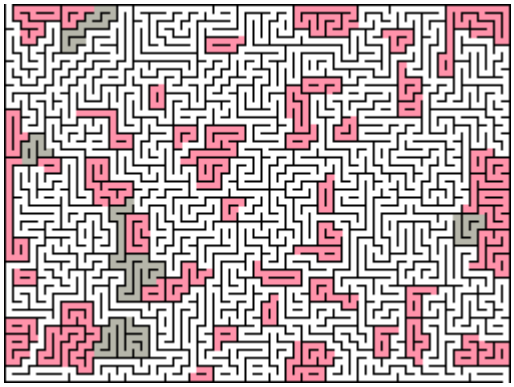
- 

Алгоритм Тремо (Trémaux's algorithm): этот метод решения лабиринтов разработан для использования человеком внутри лабиринта. Он похож на recursive backtracker и находит решение для всех лабиринтов: при движении по проходу мы рисуем линию за собой, помечающую наш путь. При попадании в тупик поворачиваемся назад и возвращаемся тем же путём, которым пришли. Дойдя до развилки, на которой ещё не были, случайным образом выбираем новый проход. Если мы проходим по новому проходу и доходим до развилки, которую посещали ранее, то считаем её тупиком и возвращаемся тем же путём, которым пришли. (Этот последний этап является самым важным, он не позволяет двигаться кругами и пропускать проходы в плетёном лабиринте.) Если двигаясь по проходу, который мы посещали раньше (т.е. раньше пометили), мы наткнулись на развилку, то выбираем любой новый проход, если это возможно, в противном случае выбираем старый проход (т.е. тот, который мы раньше пометили). Все проходы будут или пустыми, то есть мы их ещё не посещали, помеченными один раз, то есть мы там были ровно один раз, или помеченными дважды, то есть мы двигались по ним и вынуждены были возвращаться в обратном направлении. Когда мы наконец достигнем решения, то помеченные один раз пути будут составлять прямой путь до самого начала. Если у лабиринта нет решения, то мы окажемся в начале, а все проходы будут помечены дважды.

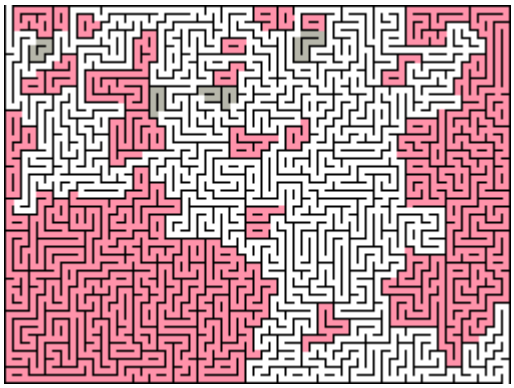
- 

Заполнитель тупиков (Dead end filler): это простой алгоритм решения лабиринтов. Приоритетом для него является лабиринт, он всегда очень быстр и не использует

дополнительной памяти. Мы просто сканируем лабиринт и заполняем каждый тупик, заливая проход в обратном порядке от тупика, пока не достигнем развилки. В том числе это относится и к заливке проходов, которые стали частями тупиков после удаления других тупиков. В конце у нас останется одно решение, или несколько решений, если у лабиринта их больше одного. Алгоритм всегда находит для идеальных лабиринтов одно уникальное решение, но не особо преуспевает в лабиринтах с сильным плетением, и на самом деле почти бесполезен во всех лабиринтах без тупиков.

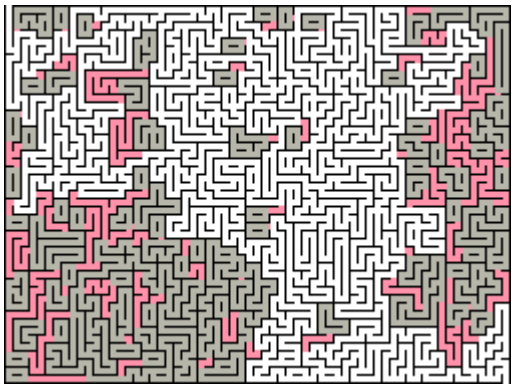
- 

Cul-de-sac filler: этот метод находит и заполняет тупиковые развязки или петли, то есть конструкции в лабиринте, состоящие из тупикового пути с единственной петлёй в конце. Как и в dead end filler, приоритетом здесь является лабиринт, алгоритм всегда быстр и не использует дополнительной памяти. Сканируем лабиринт и для каждой петлевой развилки (петлевая развилка — это такая развилка, в которой два ведущих из неё прохода соединяются друг с другом, по пути не образуя новых развилок) добавляем стену, чтобы превратить всю петлю в длинный тупик. Затем мы запускаем dead end filler. В лабиринтах могут быть петли, выдающиеся из других конструкций, которые становятся петлями после удаления первых петель, поэтому весь процесс нужно повторять, пока при сканировании ничего не станет происходить. Этот алгоритм не очень полезен в сложных, сильно плетёных лабиринтах, но будет отсекать больше путей, чем простой dead end filler.

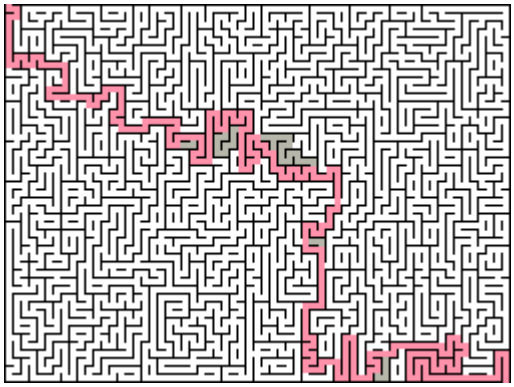
- 

Blind alley filler: этот метод находит все возможные решения, вне зависимости от того, насколько они длинные или короткие. Он делает это, заполняя все тупиковые развязки. Тупиковая развязка — это такая конструкция, в которой двигаясь в одном направлении, для достижения цели придётся возвращаться назад по тому же пути в другом направлении. Все тупики являются тупиковыми развязками, как и все петли, описанные в алгоритме cul-de-sac

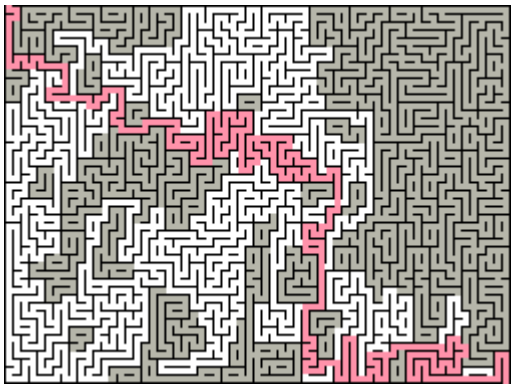
filler, а также секции проходов любого размера. соединённые с остальной частью лабиринта единственным проходом. Приоритет он отдаёт лабиринту, не использует дополнительной памяти, но, к сожалению, довольно медленный. В каждой развилке мы отправляем следующего вдоль стен робота по каждому проходу, ведущему из неё, и смотрим, вернулся ли отправленный по пути робот тем же маршрутом (то есть не пришёл с другого направления и не вышел из лабиринта). Если это происходит, то такой проход и всё после него не может быть никаким путём решения, поэтому мы перекрываем этот проход и заливаем всё за ним. Этот алгоритм заполняет всё то же, что и cul-de-sac filler и кое-что ещё, однако описанный ниже алгоритм collision solver заполнит всё то же, что и этот алгоритм и кое-что ещё.

- 

Blind alley sealer: этот алгоритм похож на blind alley filler тем, что он тоже находит все возможные решения, удаляя из лабиринта тупиковые развязки. Однако он заполняет только проходы в каждую тупиковую развязку и не касается набора проходов в их конце. В результате он создаст во всех тупиковых развязках и петлях сложнее простых тупиков недостижимые проходы. Этот алгоритм отдаёт приоритет лабиринту, работает гораздо быстрее, чем blind alley filler, хоть и требует дополнительной памяти. Мы определяем каждую соединённую секцию стен в уникальное множество. Чтобы сделать это, для каждой секции стен, ещё не находящихся в множестве, мы выполняем заливку поверх стен в этой точке, и определяем все достижимые стены в новое множество. После того, как все стены окажутся в множествах, мы проверяем каждую секцию проходов. Если стены по обеим её сторонам находятся в одном множестве, то мы перекрываем этот проход. Такой проход должен быть тупиковой развязкой, потому что стены по обеим сторонам соединены друг с другом, образуя огороженную площадку. Стоит заметить, что подобную технику можно использовать для помощи в решении гиперлабиринтов благодаря перекрытию пространства между ветвями, соединёнными друг с другом.

- 

  Поиск кратчайшего пути (Shortest path finder): как можно понять из названия, этот алгоритм находит кратчайшее решение, при наличии нескольких решений выбирая одно из них. Он множество раз отдаёт приоритет находящемуся в лабиринте, быстр для всех типов лабиринтов и требует довольно много дополнительной памяти, пропорциональной размеру лабиринта. Как и collision solver, он, по сути, заливает лабиринт «водой» так, что всё на одном расстоянии от начала заливается одновременно (в программировании это называется поиском в ширину), однако каждая «капля» или пиксель, запоминает, каким пикселем они были заполнены. Как только в решение попадает «капля», мы возвращаемся обратно от неё к началу, и это будет кратчайшим путём. Этот алгоритм хорошо работает для любых входных данных, потому что в отличие от большинства остальных не требует наличия в лабиринте каких-то проходов в пиксель шириной, по которым можно пройти. Стоит заметить, что это, по сути, алгоритм поиска пути A* без эвристики, то есть всему движению присваивается одинаковый вес.

- 

  Поиск кратчайших путей (Shortest paths finder): он очень похож на поиск кратчайшего пути, только находит все кратчайшие решения. Как и поиск кратчайшего пути, он множество раз отдаёт приоритет находящемуся в лабиринте, быстр для всех типов лабиринтов, требует дополнительной памяти, пропорциональной размеру лабиринта, и хорошо работает с любыми входными данными, потому что не требует, чтобы в лабиринте были какие-то проходы шириной в один пиксель, по которым можно пройти. Кроме того, как и поиск кратчайшего пути, он выполняет поиск в ширину, заполняя лабиринт «водой» так, что всё на одинаковом расстоянии от начала заполняется одновременно, только здесь каждый пиксель запоминает, как далеко он находится от начала. После достижения конца выполняется ещё один поиск в ширину, начинающийся с конца, однако в него включаются только те пиксели, чьё расстояние на одну единицу меньше, чем у текущего пикселя. Включённые в путь пиксели

точно помечают все кратчайшие пути, так как в тупиковых развязках и в не самых коротких путях расстояния в пикселях будут скакать или увеличиваться.

- **Collision solver:** также называется "amoeba" solver. Этот метод находит все кратчайшие решения. Он множество раз отдаёт приоритет находящемуся в лабиринте, быстр для всех типов лабиринтов и требует наличия в памяти хотя бы одной копии лабиринта в дополнение к объёму памяти вплоть до размеров лабиринта. Он заливает лабиринт «водой» так, что всё на одном расстоянии от начала заливается одновременно (поиск в ширину). Когда два «столбца воды» приближаются к проходу с обоих краёв (сигнализируя о наличии петли), мы добавляем в исходный лабиринт стену там, где они сталкиваются. Когда все части лабиринта окажутся «затопленными», мы заполняем все новые тупики, которые не могут находиться на кратчайшем пути, и повторяем процесс, пока больше не останется столкновений. (Представьте амёбу, плывущую на гребне «волны», когда она втекает в проходы. Когда волны сталкиваются, амёбы сталкиваются и выходят из строя, создавая на этом месте новую стену потерявших сознание амёб, отсюда и название алгоритма.) По сути, он аналогичен shortest paths finder, однако эффективнее использует память (так как ему нужно отслеживать координаты только переднего края каждого столбца воды) и немного медленнее (так как потенциально должен выполняться несколько раз для устранения всего).
- **Random mouse:** для контраста приведу неэффективный метод решения лабиринта, который по сути заключается в случайном перемещении, т.е. движении в одном направлении и следовании по этому проходу со всеми поворотами, пока мы не достигнем следующей развилки. Мы не делаем никаких поворотов на 180 градусов, если без них можно обойтись. Это симулирует поведение человека, случайно блуждающего по лабиринту и не помнящему, где он уже был. Алгоритм медленный и не гарантирует завершения или решения лабиринта, а после достижения конца будет столь же сложно вернуться к началу, зато он прост и не требует дополнительной памяти для реализации.