Алгоритмы создания лабиринта.

**Recursive backtracker:** требует стека, объём которого может доходить до размеров лабиринта. При вырезании проходов 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.

**Алгоритм Краскала:** создаёт минимальное связующее дерево. Он вырезает сегменты проходов по всему лабиринту случайным образом. Для его работы требуется объём памяти, пропорциональный размеру лабиринта, а также возможность перечисления каждого ребра или стены между ячейками лабиринта в случайном порядке. 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. Объединение двух множество по обеим сторонам стены будет медленной операцией. 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. Этот алгоритм работает достаточно быстро, но медленнее большинства из-за создания списка рёбер и управления множествами.

**Алгоритм Прима (истинный):** создаёт минимальное связующее дерево, обрабатывая уникально случайные веса рёбер. Объём требуемой памяти пропорционален размеру лабиринта. Начинаем с любой вершины. 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 (можно с помощью кучи), хранящая все рёбра границы. Тем не менее, этот алгоритм достаточно медленный, потому что для выбора элементов из обработки кучи требует времени log(n).

**Алгоритм Прима (упрощённый):** 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.

**Алгоритм Прима (модифицированный):** 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: "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», и "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. Алгоритм очень быстр, быстрее упрощённого алгоритма Прима, потому что ему не нужно составлять и обрабатывать список рёбер.

**Алгоритм Олдоса-Бродера:** выбираем точку и случайным образом перемещаемся в соседнюю ячейку. Если мы попали в не вырезанную ячейку, то вырезаем в неё проход из предыдущей ячейки. Продолжаем двигаться в соседние ячейки, пока не вырежем проходы во все ячейки. Алгоритм 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.

**Алгоритм Уилсона:** 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 проблемы со скоростью, потому что может уйти много времени на нахождение первого случайного пути к начальной ячейке.

**Алгоритм Hunt and kill:** не требует дополнительной памяти или стека, а потому подходит для создания огромных лабиринтов. 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. Он выполняется медленнее из-за времени, потраченного на охоту за последними ячейками.

**Алгоритм выращивания дерева (Growing tree algorithm):** 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.

**Алгоритм выращивания леса (Growing forest algorithm):** основанный на деревьях и множествах. Начинаем со всех ячеек, случайным образом отсортированных в список «новых»; кроме того, у каждой ячейки есть собственное множество. Сначала 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. 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.

**Алгоритм Эллера:** 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. 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.

**Алгоритм Sidewinder:** 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). Поднимаемся вверх от случайной ячейки. То есть в каждой строке всегда будет ровно один проход, ведущий вверх. Алгоритм никогда не делает петель и не посещает одну строку больше одного раза, однако оно «извивается из стороны в сторону».

**Рекурсивное деление (Recursive division):** использует стек, 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. Достаточно быстрый алгоритм без отклонений в направлениях, имеет очевидный недостаток в виде длинных стен, пересекающих внутренности лабиринта. То есть 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 решения, которое или следует вдоль внешнего края, или иначе напрямую пересекает внутреннюю часть.

**Лабиринты на основе двоичных деревьев:** простой и быстрый алгоритм. Для каждой ячейки мы вырезаем проход или вверх, или влево, но никогда 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 (в них никогда не будет перекрёстков, а все тупики имеют проходы, ведущие вверх или влево, и никогда не ведущие вниз или вправо).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Алгоритм | Тип | Приоритет | Память | Время |
| Recursive backtracker | Дерево | |  |  | | --- | --- | | Проходы |  | |  | ? |
| Алгоритм Краскала | Множество | Проходы и стены |  | ? |
| Алгоритм Прима (истинный) | Дерево | Проходы и стены |  | ? |
| Алгоритм Прима (упрощённый) | Дерево | Проходы и стены |  | ? |
| Алгоритм Прима (модифицированный) | Дерево | Проходы и стены |  | ? |
| Алгоритм Олдоса-Бродера | Дерево | Проходы и стены | 0 | ? |
| Алгоритм Уилсона | Дерево | Проходы и стены |  | ? |
| Hunt and kill | Дерево | |  |  | | --- | --- | | Проходы |  | | 0 | ? |
| Алгоритм выращивания дерева | Дерево | Проходы и стены |  | ? |
| Алгоритм выращивания леса | Дерево и Множество | Проходы и стены |  | ? |
| Алгоритм Эллера | Множество | Проходы и стены | N | ? |
| Алгоритм Sidewinder | Множество | Проходы и стены | 0 | ? |
| Рекурсивное деление | Дерево | Стены | N | ? |
| Лабиринты на основе двоичных деревьев | Множество | Проходы и стены | 0 | ? |