ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА»

(СПбГУТ)

Факультет инфокоммуникационных Сетей и систем (иксс)

кафедра программной инженерии и вычислительной техники (пи и вт)

Дисциплина: «Алгоритмы и структуры данных»

Лабораторная работа №6.

Тема: «Эвристические алгоритмы (лабиринты)»

Выполнили:

Студенты группы ИКПИ-05

Принял:

Доцент кафедры ПИиВТ

Молошников Ф.А., Мартынюк А.А.

Подпись\_\_\_\_\_\_\_\_\_\_\_\_

Дагаев А.В.

Подпись\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_\_»\_\_\_\_\_\_\_\_2022

СОДЕРЖАНИЕ

[1. Цель работы 3](#_Toc102513461)

[2. Описание Программы 3](#_Toc102513462)

[2.1. Описание программы 3](#_Toc102513463)

[2.2. Разработка классов и функций 3](#_Toc102513464)

[2.2.1. maze 3](#_Toc102513465)

[2.2.2. mazeWeighted 6](#_Toc102513466)

[2.2.3. statisticsMaze 6](#_Toc102513467)

[2.2.1. statisticsMazeSearch 7](#_Toc102513468)

[3. Описание алгоритмов 8](#_Toc102513469)

[3.1. Алгоритмы генерации лабиринта 8](#_Toc102513470)

[3.1.1. Олдоса-Бродера 8](#_Toc102513471)

[3.1.2. Уилсона 11](#_Toc102513472)

[3.1.3. Уилсона (модификация) 19](#_Toc102513473)

[3.1.4. Двоичным деревом 20](#_Toc102513474)

[3.2. Алгоритмы поиска пути в лабиринте 23](#_Toc102513475)

[3.2.1. Подготовка лабиринта 23](#_Toc102513476)

[3.2.2. Ли 23](#_Toc102513477)

[3.2.3. Ли (модификация с двумя волнами) 25](#_Toc102513478)

[3.2.4. Дейкстры 28](#_Toc102513479)

[3.2.5. А\* (AStar; А со звездой) 41](#_Toc102513480)

[3.2.6. Восстановление свойств кучи. 41](#_Toc102513481)

[3.2.7. Построение кучи. 42](#_Toc102513482)

[3.2.8. Сортировка кучей 42](#_Toc102513483)

[3.3. Быстрая сортировка Хоара 42](#_Toc102513484)

[3.4. Сортировка Шелла 42](#_Toc102513485)

[4. Сравнительный анализ 43](#_Toc102513486)

[5. Результаты работы 49](#_Toc102513487)

[6. Выводы 51](#_Toc102513488)

[7. Исходный код 51](#_Toc102513489)

[7.1. CMakeLists.txt 51](#_Toc102513490)

[7.2. main.cpp 51](#_Toc102513491)

[7.3. statistics.inl 52](#_Toc102513492)

[7.4. HeapSort.inl 54](#_Toc102513493)

[7.5. QuickSortHoare.inl 56](#_Toc102513494)

[7.6. ShellSort.inl 58](#_Toc102513495)

[7.7. PlotScript.gpi 59](#_Toc102513496)

1. Цель работы

Ознакомление с эвристическими алгоритмами поиска пути в лабиринте и методикой оценки их эффективности. Для сравнения были выбраны:

1. Алгоритмы генерации лабиринта:
   1. Уилсона
   2. Уилсона (модифицированный)
   3. Олдоса-Бродера
   4. Бинарного дерева
2. Алгоритмы поиска пути в лабиринте:
   1. Ли
   2. Ли (модификация с 2 волнами)
   3. Дейкстры
   4. А\* (AStar)

Первые два алгоритма поиска пути работают на ДРП (дискретном рабочем поле), где, с точки зрения прокладки путей, все ячейки одинаковы. 3 и 4 алгоритмы работают на графах. Мы используем их модификацию для работы на ДРП (частный случай графа), где ячейки могут иметь разные веса. Вес означает путь в эту ячейку из соседней ячейки. В общем случае, вес ячейки соответствует ребру соответствующего ориентированного графа.

1. Описание Программы
   1. Описание программы

Для исследования алгоритмов генерации лабиринтов и поиска в них пути были разработаны классы maze и mazeWeighted, которые реализуют хранение лабиринтов и доступ к ним. Для сбора информации о времени генерации лабиринтов и о времени поиска в них путей были разработаны классы StatisticsMaze и StatisticsMazeSearch. В работе использовались классы presenthandler и plothandler, скрипт PlotScript.bash, разработанные в рамках предыдущих лабораторных работ и предназначенные для упрощения вывода промежуточных и конечных результатов работы алгоритмов. Работа также содержит вспомогательные функции для ввода и вывода. Помимо двух основных целей, которые собираются из файлов StatisticsMaze\_main.cpp и StatisticsMazeSearch\_main.cpp и служат для исследования времени генерации лабиринтов и поиска путей в них соответственно, работа содержит отдельные цели для демонстрации алгоритмов, а так же для демонстрации и отладки разработанных классов.

1. Язык: C++
2. Среда: VS Code
3. ОС: Debian 11
4. Оболочка: xfce 4
5. Библиотеки: -
6. Фреймворк: -

Остановимся подробнее на некоторых классах и функциях, существенных в данной работе:

* 1. Разработка классов и функций
     1. maze

Класс, реализующий лабиринт. Позволяет хранить лабиринт и обращаться к нему.

Лабиринт хранится внутри вектора BaseVector в виде символов. Нечетные строки хранят информацию о верхних и нижних стенах ячеек, четные – о боковых стенах ячеек и значения, хранящиеся в ячейках. Используются следующие символы:

1. # - стена между ячейками
2. ? – отсутствие стены между ячейками
3. + - угол ячейки (Служит для удобства представления. Никак не используется)
4. . – Значение ячейки по умолчанию.

Такой подход к хранению лабиринта нельзя назвать оптимальным по используемой памяти, т.к. треть памяти занята символами «+», которые нигде не используются (например, более оптимальным было бы хранить в массиве только нижнюю и левую стены), однако он обеспечивает довольно быстрое обращение к своим элементам, поскольку использует контейнер vector для хранения элементов.

Также он позволяет в относительно наглядном виде просматривать лабиринт и относительно просто его редактировать (для замены стены на проем или проем на стену достаточно поменять символ # на ? или ? на #).

Например, этот лабиринт

┌───────┬───┬───────┬───┐

│ m a │ z │ e . │ . │

├──── │ │ ┌───┘ │

│ b y │ . │ m │ e . │

│ ────┤ ┴ ├──── │

│ . . │ . . │ . . │

├───┐ │ ┬ │ ────┤

│ . │ . │ . │ . │ . . │

│ ┴ └───┘ │ ────┤

│ . . . . │ . . │

│ ┬ ────┐ ┴ ────┤

│ . │ . . │ . . . │

└───┴───────┴───────────┘

Будет иметь следующее внутреннее представление:

+#+#+#+#+#+#+

#m?a#z#e?.#.#

+#+?+?+?+#+?+

#b?y#.#m#e?.#

+?+#+?+?+#+?+

#.?.#.?.#.?.#

+#+?+?+?+?+#+

#.#.#.#.#.?.#

+?+?+#+?+?+#+

#.?.?.?.#.?.#

+?+?+#+?+?+#+

#.#.?.#.?.?.#

+#+#+#+#+#+#+

Доступ к ячейкам реализован по индексам i j. При этом нумерация ячеек аналогична нумерации элементов в матрицах: Увеличение индекса строк i идет сверху вниз, увеличение индекса столбцов j слева направо.

Вывод лабиринтов в декоративном виде осуществляется с помощью символов Unicode и доступен в двух вариантах: с мелкими и крупными ячейками. Вариант с крупными ячейками уже был показан выше, вариант с мелкими ячейками показан ниже:

┌─┬─────┬───┬───┬─┬─┐

│ │ ──┐ │ ──┤ ──┘ ┴ │

│ └───┘ └── ┴ ┬ ────┤

├── ┌── ┌───┬─┼── ┬ │

│ ┬ └─┬─┴─┐ ┴ └───┤ │

├─┼── ┴ ──┘ ┬ ──┬─┤ │

│ └── ┬ ──┬─┤ ┬ │ │ │

│ ──┬─┘ ┌─┘ │ │ │ ┴ │

├── │ ┬ ┴ ──┴─┤ │ ┬ │

│ ┌─┴─┴──── ──┼─┴─┘ │

└─┴───────────┴─────┘

* + - 1. Свойства:

|  |  |
| --- | --- |
| int n=0 | высота лабиринта |
| int m=0; | ширина лабиринта |
| std::vector<std::vector<char>> BaseVector; | Вектор, в котором хранится лабиринт |

* + - 1. Методы:

|  |  |
| --- | --- |
| maze(int Nn, int Nm) | Конструктор с параметрами. Создает лабиринт высотой Nn и шириной Nm, ячейки которого изолированы друг от друга стенами. |
| void SetCellValue(int i, int j, char c) | Устанавливает значение в ячейке |
| void ResetValues() | Сбрасывает значения ячеек (по умолчанию «.») |
| char GetCellValue(int i, int j) | Возвращает значение в ячейке |
| void SetCellWalls(int i, int j, int alpha, bool HasWall, bool Protected=false) | Позволяет установить или убрать стену между ячейками. alpha – полярный угол, под которым радиус вектор из центра ячейки встречает соответствующую стену (измеряется в pi/4). Т.е. если alpha=0, то будет изменено состояние правой стены ячейки, если alpha=1, то верхней стены ячейки, если alpha=2, то левой стены, если alpha=3, то нижней стены и т.д. Флаг Protected позволяет замораживать внешние стены лабиринта, не позволяя изменять их состояние. |
| bool HasWall(int i, int j, int alpha) | Позволяет определять наличие стены вокруг ячейки. alpha имеет тот же смысл, что и в методе SetCellWalls |
| int GetNeighborI(int i, int alpha)  int GetNeighborJ(int j, int alpha) | Позволяют получить координаты (в I, j) соседней ячейки, радиус вектор до которой находится под углом alpha. |
| std::string Show(char\* filename=(char\*)"cin") | Не используется |
| void CharFromWallFlags(std::stringstream& ss, std::string& WallFlags) | Вспомогательная функция для получения символа по флагу, т.е. по его описанию. |
| GetCellValueFromStruct 3 перегрузки | Вспомогательные функции для организации вывода в декоративном виде данных внутри ячеек. |
| std::string StrFromInt(T value) |
| template <typename T=const std::string>  void ShowDecorate(char\* filename=(char\*)"cout", int Mode=0, int Scale=1, bool IsWithValues=false, T& Values=std::string("GetCellValue"), int member=0) | Шаблонный метод для вывода лабиринта в терминал или файл. Первый аргумент – имя файла для вывода (по умолчанию в терминал), второй аргумент – режим вывода 0 – внутреннее представление и декоративный вид. 1 – только декоративный вид. Scale – 1 мелкие ячейки. (высота каждой ячейки 0.5+0.5=1 строка; ширина – 0.5+1+0.5=2 символа), 2 крупные ячейки. В случае крупных ячеек возможен вывод внутри ячеек значений из внешних контейнеров различных типов. member служит для выбора поля внутри элемента контейнера для представления внутри ячейки. Для того чтобы выводить какой-либо свой тип контейнера внутри ячейки, нужно написать перегрузку функции GetCellValueFromStruct. В ячейку можно выводить значения длиной до 3х символов. Если выводимое значение имеет большую длину, то можно, например, выводить остаток этого значения от деления на 1000 либо отсечь часть символов. |

* + 1. mazeWeighted

Наследует класс maze. Позволяет создавать лабиринты с весами ячеек.

* + - 1. Свойства:

|  |  |
| --- | --- |
| std::vector<std::vector<int>> Weights; | Контейнер с весами ячеек |

* + - 1. Методы:

|  |  |
| --- | --- |
| mazeWeighted(int Nn, int Nm, int FillWeight=0) | Конструктор с параметрами. Создает лабиринт из изолированных друг от друга ячеек с весом по умолчанию равным нулю. |
| void WeightsToValues() | Не используется. Лучше использовать ShowDecorate с параметром Weights. Копирует веса в значения ячеек для удобного представления. |
|  |  |

* + 1. statisticsMaze

Класс для исследования функций, выполняющих генерацию лабиринтов.

* + - 1. Свойства:

|  |  |
| --- | --- |
| int CurrentN=0; | Текущая высота лабиринта |
| int CurrentM=0; | Текущая ширина лабиринта |
| int NStart=0, NEnd=0, NStep=0; | Начальное, конечное значения высоты лабиринта и шаг её изменения |
| double MRatio=1; | M/N отношение ширины к длине |
| int NumberOfRuns=1; | число прогонов для текущих значений N M |

* + - 1. Методы:

|  |  |
| --- | --- |
| StatisticsMaze(int NNStart, int NNEnd, int NNStep, double NMRatio, int NNumberOfRuns, std::default\_random\_engine& generator, presenthandler& PresentHandler, std::string filename, void (\*CallBackGenerate)(maze&, std::default\_random\_engine&, presenthandler&, CallBackParamsTail&... ), CallBackParamsTail& ...callbackparamstail) | Конструктор. Запускает функцию генерации лабиринта со значениями высоты от NNStart до NNEnd с шагом NNStep и отношением ширины к высоте NMRatio. Записывает время генерации в файл. |
| void printLabel(std::string filename) | Запись информации об условиях выполнения сортировки в файл |
| void printLabel(std::string filename) | Запись искомых значений времени в файл |

* + 1. statisticsMazeSearch

Аналогичен классу statisticsMaze. Следует отметить, что передача в конструктор генератора позволяет для различных алгоритмов поиска генерировать одинаковую последовательность лабиринов, причем последовательности стартовых и финишных ячеек так же будут совпадать.

* + - 1. Свойства:

|  |  |
| --- | --- |
| int CurrentN=0; | Текущая высота лабиринта |
| int CurrentM=0; | Текущая ширина лабиринта |
| int NStart=0, NEnd=0, NStep=0; | Начальное, конечное значения высоты лабиринта и шаг её изменения |
| double MRatio=1; | M/N отношение ширины к длине |
| int NumberOfRuns=1; | число прогонов для текущих значений N M |

* + - 1. Методы:

|  |  |
| --- | --- |
| void StatisticsMazeSearchFunc(int NNStart, int NNEnd, int NNStep, double NMRatio, int NNumberOfRuns, std::default\_random\_engine& generator, presenthandler& PresentHandler, std::string filename, void (\*CallBackSearch)(T&, int starti, int startj, int finishi, int finishj, std::vector<std::pair<int,int>>& Path, presenthandler& PrHandler)) | Генерирует лабиринт со значениями высоты от NNStart до NNEnd с шагом NNStep и отношением ширины к высоте NMRatio. Запускает для него функцию поиска между случайно выбранными ячейками и записывает время поиска в файл |
| void printLabel(std::string filename) | Запись информации об условиях выполнения сортировки в файл |
| void printLabel(std::string filename) | Запись искомых значений времени в файл |

1. Описание алгоритмов
   1. Алгоритмы генерации лабиринта

В данной работе реализованы алгоритмы генерации идеальных лабиринтов, т.е. лабиринтов без недостижимых областей и без циклов. Генерация происходит методом вырезания проходов.

* + 1. Олдоса-Бродера
       1. Теоретические сведения

Тип(Дерево/Множество): Дерево

Фокус(Возможность вырезать/добавлять стены): Оба

Смещенность(одинаково ли легко перемещаться по лабиринту в различных направлениях): отсутствует

Однородность(Генерирует ли алгоритм все возможные лабиринты с равной вероятностью): Да

Память: 0

* + - 1. Алгоритм

1. Лабиринт состоит из изолированных стенами ячеек
2. Выбирается случайная ячейка и отмечается как посещенная
3. Счетчик посещенных вершин k устанавливается равным 1
4. Пока k не равен числу ячеек в лабиринте
   1. Выбирается случайное направление движения
   2. Если это направление не направлено в стену, то
      1. проверяем, была ли уже посещена эта ячейка. Если нет, то прорезаем стену к ней, переходим к ней и помечаем её как посещенную
   3. В противном случае просто переходим к ней
      * 1. Пример работы

Символом «1» функция помечает посещенные ячейки, «.» - непосещенные. Также будем обозначать символом «\*» текущую ячейку

Система координат: (i, j) i увеличивается сверху вниз. j-слева направо. Счет с нуля.

1. Начальная ячейка была выбрана с координатами (1, 1)

┌───┬───┬───┐

│ . │ . │ . │

├───┼───┼───┤

│ . │ 1\*│ . │

├───┼───┼───┤

│ . │ . │ . │

└───┴───┴───┘

1. Случайным образом было выбрано направление в правую сторону. Поскольку ячейка справа ещё не была посещена, то стена к ней прорезается. Происходит переход к этой ячейке ячейке (1, 2)

┌───┬───┬───┐

│ . │ . │ . │

├───┼───┴───┤

│ . │ 1 1\*│

├───┼───┬───┤

│ . │ . │ . │

└───┴───┴───┘

1. Аналогично проходится ячейка (0,2)

┌───┬───┬───┐

│ . │ . │ 1\*│

├───┼───┘ │

│ . │ 1 1 │

├───┼───┬───┤

│ . │ . │ . │

└───┴───┴───┘

1. Выбирается направление вниз. Так как нижняя ячейка уже была посещена, то стена к ней не прорезается(в случае, если бы она там была, это было бы важно) и происходит переход к этой ячейке

┌───┬───┬───┐

│ . │ . │ 1 │

├───┼───┘ │

│ . │ 1 1\*│

├───┼───┬───┤

│ . │ . │ . │

└───┴───┴───┘

1. Аналогично 4) возвращаемся к ячейке (0, 2)

┌───┬───┬───┐

│ . │ . │ 1\*│

├───┼───┘ │

│ . │ 1 1 │

├───┼───┬───┤

│ . │ . │ . │

└───┴───┴───┘

1. Аналогично шагу 2) шаги 6)-11)

┌───┬───────┐

│ . │ 1\* 1 │

├───┼──── │

│ . │ 1 1 │

├───┼───┬───┤

│ . │ . │ . │

└───┴───┴───┘

┌───────────┐

│ 1\* 1 1 │

├───┬──── │

│ . │ 1 1 │

├───┼───┬───┤

│ . │ . │ . │

└───┴───┴───┘

┌───────────┐

│ 1 1 1 │

│ ┌──── │

│ 1\*│ 1 1 │

├───┼───┬───┤

│ . │ . │ . │

└───┴───┴───┘

┌───────────┐

│ 1 1 1 │

│ ┌──── │

│ 1 │ 1 1 │

│ ├───┬───┤

│ 1\*│ . │ . │

└───┴───┴───┘

┌───────────┐

│ 1 1 1 │

│ ┌──── │

│ 1 │ 1 1 │

│ └───┬───┤

│ 1 1\*│ . │

└───────┴───┘

1. После 11 шага работа алгоритма завершается, т.к. количество посещенных ячеек стало равно количеству ячеек в лабиринте.

┌───────────┐

│ 1 1 1 │

│ ┌──── │

│ 1 │ 1 1 │

│ └───────┤

│ 1 1 1\*│

└───────────┘

* + - 1. Примеры лабиринтов



фрагмент лабиринта, сгенерированного алгоритмом Олдоса-Бродера (2500х2500 ячеек)



лабиринт, сгенерированный алгоритмом Олдоса-Бродера (100х100 ячеек)

* + 1. Уилсона
       1. Теоретические сведения

Тип(Дерево/Множество): Дерево

Фокус(Возможность вырезать/добавлять стены): Оба

Смещенность(одинаково ли легко перемещаться по лабиринту в различных направлениях): отсутствует

Однородность(Генерирует ли алгоритм все возможные лабиринты с равной вероятностью): Да

Память: N^2

* + - 1. Алгоритм

1. Лабиринт состоит из изолированных стенами ячеек
2. Выбрать случайную ячейку и добавить её в множество UST (uniform spanning trees — однородные остовные деревья).
3. Пока есть ячейки, не входящие в UST, выбрать случайную ячейку и совершить от неё случайную прогулку, пока не будет встречена какая-нибудь ячейка из UST. Во время прогулки в ячейки записываются путевые координаты. Причем если во время такой прогулки мы наткнемся на ячейку, в которой уже были, т.е. сделаем петлю, то её путевые координаты перезапишутся, т.е. эта петля не попадет в итоговый путь (по-видимому, это одна из причин, по которой этот алгоритм такой медленный)
4. Проходим по путевым координатам от ячейки, с которой мы начали прогулку, до ячейки из UST по путевым координатам, прорезая стены (сразу это нельзя сделать из-за петель, которые срезает алгоритм во время первого прохода).
   * + 1. Пример работы

Символом «U» будем помечать ячейки в UST, «>»^»«<»«v» - путевые координаты. «.» - непосещенные ячейки. Также будем обозначать символом «\*» текущую ячейку

Система координат: (i, j) i увеличивается сверху вниз. j-слева направо. Счет с нуля.

1. Начальная ячейка была выбрана с координатами (1, 2). Добавляем её в UST.

┌───┬───┬───┐

│ . │ . │ . │

├───┼───┼───┤

│ . │ . │ U\*│

├───┼───┼───┤

│ . │ . │ . │

└───┴───┴───┘

1. Выберем случайную ячейку, которой нет в UST. Например, (2, 0)

┌───┬───┬───┐

│ . │ . │ . │

├───┼───┼───┤

│ . │ . │ U\*│

├───┼───┼───┤

│ \* │ . │ . │

└───┴───┴───┘

1. Начнем блуждание, пока не дойдем до ячеек, которые есть в UST

┌───┬───┬───┐

│ . │ . │ . │

├───┼───┼───┤

│ . │ . │ U │

├───┼───┼───┤

│ ^\*│ . │ . │

└───┴───┴───┘

┌───┬───┬───┐

│ . │ . │ . │

├───┼───┼───┤

│ ^\*│ . │ U │

├───┼───┼───┤

│ ^ │ . │ . │

└───┴───┴───┘

┌───┬───┬───┐

│ >\*│ . │ . │

├───┼───┼───┤

│ ^ │ . │ U │

├───┼───┼───┤

│ ^ │ . │ . │

└───┴───┴───┘

┌───┬───┬───┐

│ > │ v\*│ . │

├───┼───┼───┤

│ ^ │ . │ U │

├───┼───┼───┤

│ ^ │ . │ . │

└───┴───┴───┘

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┼───┤

│ ^ │ <\*│ U │

├───┼───┼───┤

│ ^ │ . │ . │

└───┴───┴───┘

1. Путевую координату ячейки (1, 0) пезезаписали, т.к. мы случайным образом выбрали направление вниз, а не вверх.

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┼───┤

│ v\*│ < │ U │

├───┼───┼───┤

│ ^ │ . │ . │

└───┴───┴───┘

1. Путевую координату ячейки (2, 0) пезезаписали, т.к. мы случайным образом выбрали направление вправо, а не вверх.

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┼───┤

│ v │ < │ U │

├───┼───┼───┤

│ >\*│ . │ . │

└───┴───┴───┘

1. Продолжаем блуждание

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┼───┤

│ v │ < │ U │

├───┼───┼───┤

│ > │ ^\*│ . │

└───┴───┴───┘

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┼───┤

│ v │ >\*│ U │

├───┼───┼───┤

│ > │ ^ │ . │

└───┴───┴───┘

1. Мы дошли до ячейки в UST.

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┼───┤

│ v │ > │ U\*│

├───┼───┼───┤

│ > │ ^ │ . │

└───┴───┴───┘

1. Прорезаем путь от исходной ячейки до ячейки в UST. Добавляя ячейки в UST.

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┼───┤

│ v │ > │ U │

├───┴───┼───┤

│ U\* ^ │ . │

└───────┴───┘

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┼───┤

│ v │ > │ U │

├───┘ ├───┤

│ U U\*│ . │

└───────┴───┘

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┴───┤

│ v │ U\* U │

├───┘ ┌───┤

│ U U │ . │

└───────┴───┘

1. Путь прорезан. Путевые координаты ячеек, которые были в петлях, оставляем. На работу алгоритма никакого влияния они не окажут.

┌───┬───┬───┐

│ > │ v │ . │

├───┼───┴───┤

│ v │ U U\*│

├───┘ ┌───┤

│ U U │ . │

└───────┴───┘

1. Аналогично включаем в UST оставшиеся ячейки.

┌───┬───┬───┐

│ v\*│ v │ . │

├───┼───┴───┤

│ v │ U U │

├───┘ ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───┬───┐

│ v │ v │ . │

├───┼───┴───┤

│ >\*│ U U │

├───┘ ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───┬───┐

│ v │ v │ . │

├───┼───┴───┤

│ > │ U\* U │

├───┘ ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───┬───┐

│ U\*│ v │ . │

│ ├───┴───┤

│ > │ U U │

├───┘ ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───┬───┐

│ U │ v │ . │

│ └───┴───┤

│ U\* U U │

├──── ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───┬───┐

│ U │ v │ . │

│ └───┴───┤

│ U U\* U │

├──── ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───┬───┐

│ U │ v │ <\*│

│ └───┴───┤

│ U U U │

├──── ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───┬───┐

│ U │ v\*│ < │

│ └───┴───┤

│ U U U │

├──── ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───┬───┐

│ U │ v │ < │

│ └───┴───┤

│ U U\* U │

├──── ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───────┐

│ U │ v U\*│

│ └───────┤

│ U U U │

├──── ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───────┐

│ U │ U\* U │

│ ┴ ────┤

│ U U U │

├──── ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───────┐

│ U │ U U │

│ ┴ ────┤

│ U U\* U │

├──── ┌───┤

│ U U │ . │

└───────┴───┘

┌───┬───────┐

│ U │ U U │

│ ┴ ────┤

│ U U U │

├──── ┌───┤

│ U U │ <\*│

└───────┴───┘

┌───┬───────┐

│ U │ U U │

│ ┴ ────┤

│ U U U │

├──── ┌───┤

│ U U\*│ < │

└───────┴───┘

┌───┬───────┐

│ U │ U U │

│ ┴ ────┤

│ U U U │

├──── ────┤

│ U U U\*│

└───────────┘

┌───┬───────┐

│ U │ U U │

│ ┴ ────┤

│ U U U │

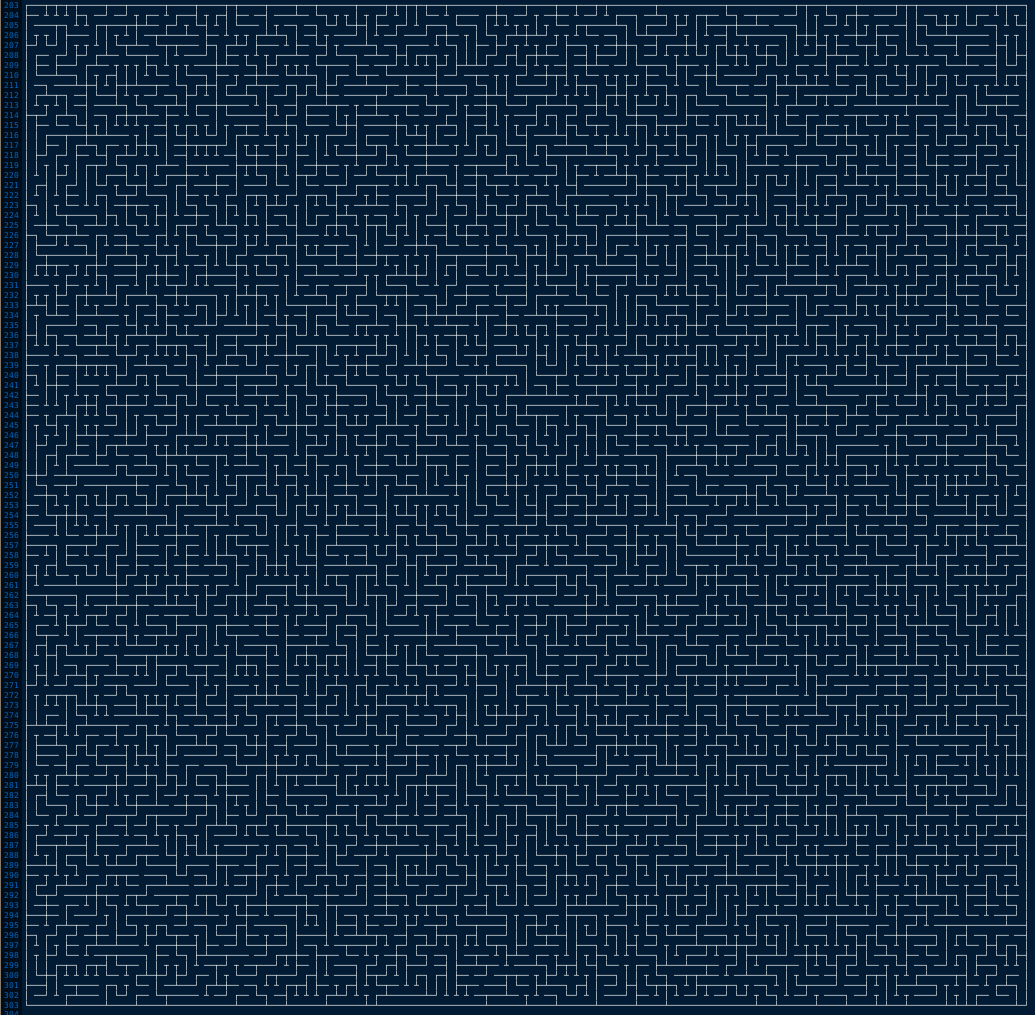
├──── ────┤

│ U U\* U │

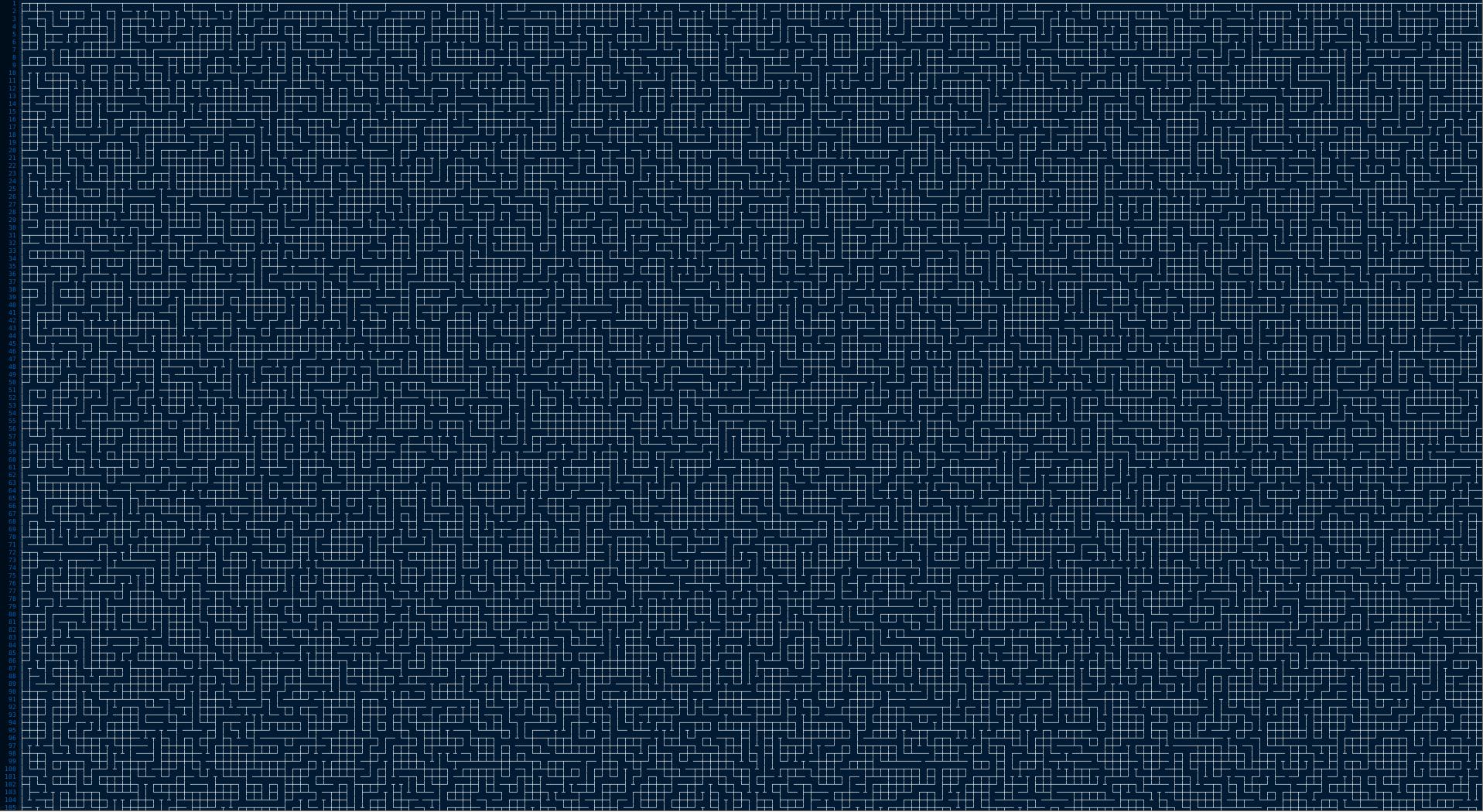
└───────────┘

* + - 1. Примеры лабиринтов

Лабиринты, генерируемые алгоритмом Уилсона, по характеру не отличимы от лабиринтов, генерируемых алгоритмом Олдоса-Бродера.



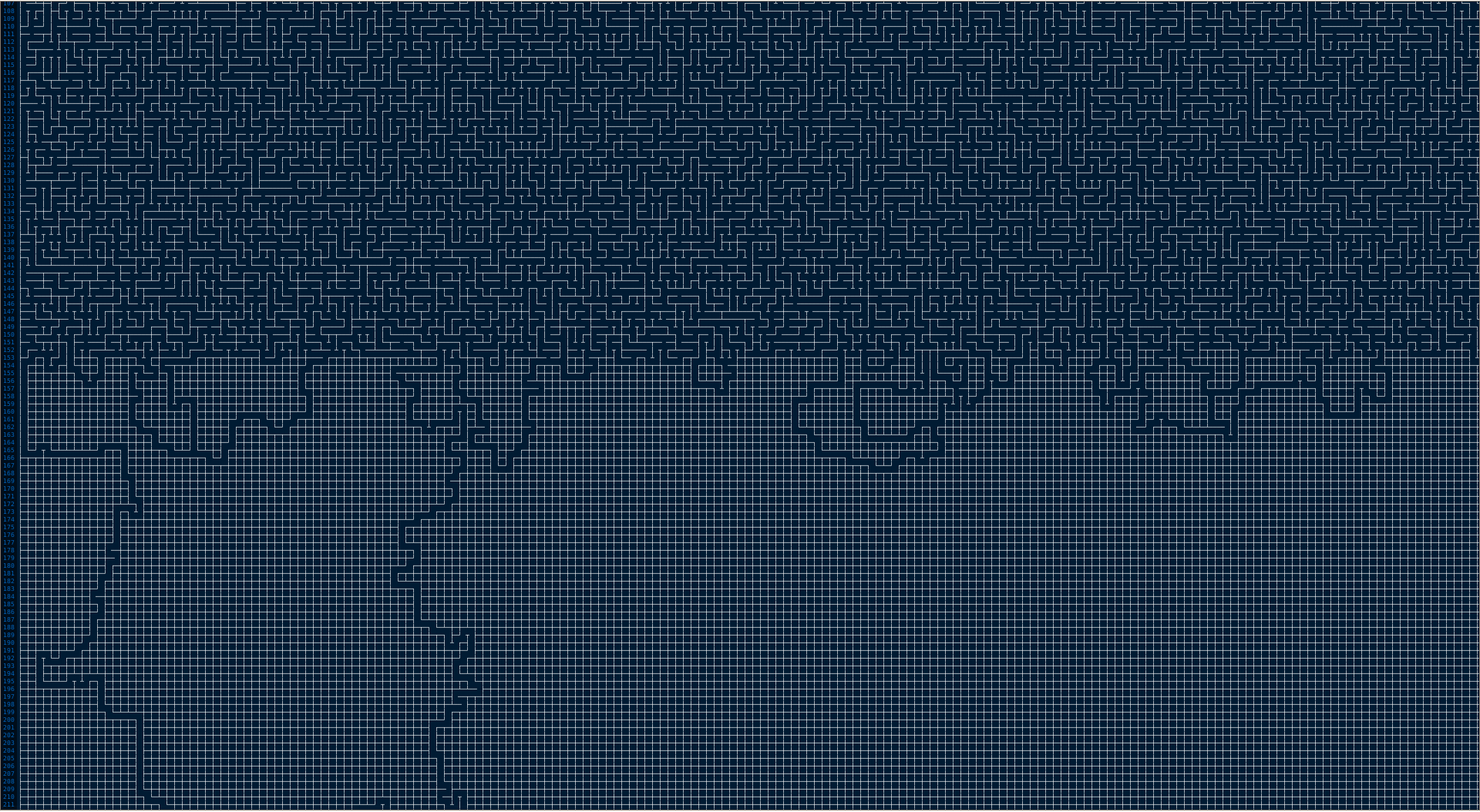
Лабиринт, сгенерированный алгоритмом Уилсона. 100x100 ячеек



Фрагмент лабиринта во время генерации алгоритмом Уилсона. 1000x1000 ячеек

* + 1. Уилсона (модификация)

Данный алгоритм отличается от предыдущего методом выбора ячейки, которой ещё нет в множестве UST. В отличие от предыдущего, здесь ячейки выбираются последовательно от начала лабиринта. Отличие наглядно показано на рисунке:



Фрагмент лабиринта во время генерации алгоритмом Уилсона (модификация). 500x500 ячеек

* + 1. Двоичным деревом
       1. Теоретические сведения

Тип(Дерево/Множество): Множество

Фокус(Возможность вырезать/добавлять стены): Оба

Смещенность(одинаково ли легко перемещаться по лабиринту в различных направлениях): Присутствует.

Однородность(Генерирует ли алгоритм все возможные лабиринты с равной вероятностью):Никогда

Память: 0\*

* + - 1. Алгоритм

1. Лабиринт состоит из изолированных стенами ячеек
2. Для каждой ячейки в сетке случайным образом разделяем проход на север или запад (север или восток / юг или запад / юг или восток) двигаясь построчно. В данной реализации сразу вырезается один из столбцов (такой же результат можно было бы получить, двигаясь построчно, пробегая все столбцы).
   * + 1. Пример работы

Продемонстрируем генерацию для выбранного направления север-восток

1. Изначальное состояние поля

┌─┬─┬─┐

├─┼─┼─┤

├─┼─┼─┤

└─┴─┴─┘

1. Вырезается первая/последняя строка (при выбранном направлении север/юг)

┌─┬───┐

├─┼─┬─┤

├─┼─┼─┤

└─┴─┴─┘



┌─────┐

├─┬─┬─┤

├─┼─┼─┤

└─┴─┴─┘

1. Вырезается левый/правый столбец, согласно выбранному направлению (запад/восток)

┌─────┐

├─┬─┐ │

├─┼─┼─┤

└─┴─┴─┘



┌─────┐

├─┬─┐ │

├─┼─┤ │

└─┴─┴─┘

1. Далее построчно (для данных направлений построчно сверху вниз)

┌─────┐

├─┐ ┬ │

├─┼─┤ │

└─┴─┴─┘



┌─────┐

├── ┬ │

├─┬─┤ │

└─┴─┴─┘



┌─────┐

├── ┬ │

├─┬─┘ │

└─┴───┘



┌─────┐

├── ┬ │

│ ┌─┘ │

└─┴───┘

* + - 1. Примеры лабиринтов

Лабиринты, сгенерированные этим алгоритмом, имеют заметную на глаз смещенность.

Приведем сгенерированные лабиринты (20x20) для различных выбранных направлений:

Север-восток:

┌───────────────────────────────────────┐

│ ┬ ┌──── ┬ ┬ ┌── ┌────────── ┬ ┌── ┬ ┬ │

│ ├─┘ ┬ ┌─┘ │ │ ┌─┴── ┌── ┬ ┬ │ │ ┬ │ │ │

│ ├── │ ├── │ ├─┴──── │ ┌─┘ │ │ ├─┴─┘ │ │

│ ├── ├─┘ ┌─┘ ├── ┌── │ │ ┬ │ │ │ ┌── │ │

│ ├── ├── ├───┘ ┌─┴── │ │ │ ├─┴─┴─┘ ┌─┘ │

├─┴── ├── │ ┌───┘ ┌───┴─┘ │ ├── ┌───┴── │

│ ┬ ┬ ├── ├─┴──── ├───────┘ │ ┬ │ ┌── ┬ │

├─┴─┴─┘ ┬ │ ┌── ┬ │ ┬ ┌──── ├─┴─┘ │ ┌─┘ │

├── ┬ ┌─┴─┴─┘ ┌─┘ ├─┘ │ ┬ ┬ │ ┬ ┌─┴─┘ ┬ │

│ ┬ │ │ ┬ ┌── ├───┘ ┬ ├─┴─┴─┘ │ ├──── │ │

│ │ │ │ ├─┴── ├── ┬ ├─┴────── ├─┴──── │ │

│ │ │ ├─┘ ┬ ┬ ├───┴─┴─────────┴───────┘ │

│ ├─┘ │ ┌─┘ ├─┴── ┌── ┬ ┬ ┬ ┬ ┬ ┬ ┬ ┌── │

│ ├───┴─┘ ┬ │ ┬ ┌─┴───┴─┴─┘ │ ├─┘ ├─┴── │

│ │ ┬ ┬ ┌─┘ ├─┴─┘ ┌────── ┌─┘ │ ┬ │ ┌── │

│ │ ├─┴─┘ ┬ ├─────┴───────┘ ┬ │ ├─┴─┴── │

├─┘ │ ┬ ┬ │ ├── ┬ ┌──────── ├─┘ ├──── ┬ │

├───┴─┴─┴─┴─┘ ┬ │ │ ┬ ┌─────┘ ┬ ├── ┌─┘ │

├── ┌─────────┴─┴─┴─┴─┘ ┌─────┴─┴───┘ ┬ │

└───┴───────────────────┴─────────────┴─┘

Север-запад:

┌───────────────────────────────────────┐

│ ┬ ──┐ ┬ ┬ ──────┐ ┬ ┬ ┬ ────┐ ──┐ ┬ ┬ │

│ │ ┬ └─┴─┴─┐ ────┤ └─┤ │ ┬ ──┤ ──┤ └─┤ │

│ │ │ ──────┤ ┬ ──┤ ┬ │ └─┴─┐ └─┐ │ ┬ │ │

│ └─┴─────┐ │ └───┤ └─┴─┐ ──┴───┴─┴─┤ │ │

│ ────┐ ──┴─┤ ──┐ │ ┬ ──┴─────┐ ┬ ──┤ │ │

│ ┬ ┬ └─────┤ ┬ └─┤ │ ──┐ ┬ ┬ └─┴─┐ └─┴─┤

│ └─┤ ┬ ────┤ └───┴─┤ ──┤ └─┤ ┬ ┬ └─┐ ──┤

│ ┬ └─┤ ────┴─────┐ └───┤ ┬ │ │ └───┤ ──┤

│ │ ┬ └─┐ ┬ ────┐ └─┐ ┬ │ └─┴─┴─┐ ──┤ ┬ │

│ │ │ ──┴─┴─────┴─┐ │ └─┴───────┴─┐ └─┤ │

│ │ └───┐ ──────┐ │ │ ──┐ ┬ ──────┤ ──┴─┤

│ └─┐ ┬ │ ┬ ────┴─┤ └───┤ └───────┴───┐ │

│ ──┴─┤ │ └─┐ ────┤ ┬ ┬ └─────┐ ┬ ──┐ │ │

│ ────┤ └─┐ │ ┬ ┬ │ └─┴─┐ ┬ ┬ └─┤ ──┤ └─┤

│ ────┴───┴─┤ │ │ └─────┤ │ └───┤ ──┤ ──┤

│ ┬ ──┐ ──┐ └─┤ │ ──┐ ┬ └─┴─────┴─┐ └───┤

│ └───┤ ┬ │ ──┴─┴─┐ └─┤ ────┐ ────┴─┐ ──┤

│ ──┐ └─┤ │ ────┐ │ ┬ └─┐ ──┤ ┬ ┬ ┬ └───┤

│ ┬ │ ──┤ └─────┴─┤ │ ──┤ ──┴─┴─┴─┴─────┤

└─┴─┴───┴─────────┴─┴───┴───────────────┘

Юг-запад:

┌─────┬───┬───┬─┬─┬─┬───┬─┬─────┬─┬─┬───┐

│ ────┤ ┌─┤ ──┘ ┴ ┴ ┴ ──┤ │ ┌─┬─┘ ┴ │ ┌─┤

│ ┌───┘ │ │ ┌─────┬───┬─┘ │ ┴ │ ┌─┬─┤ ┴ │

│ │ ──┬─┘ ┴ │ ┌─┬─┘ ┌─┤ ┌─┘ ┌─┤ │ ┴ ┴ ──┤

│ ┴ ──┤ ────┤ │ ┴ ┌─┘ ┴ │ ──┤ ┴ ┴ ┌─┬─┬─┤

│ ────┤ ┌───┤ ┴ ┌─┤ ┌─┬─┤ ──┘ ┌─┬─┘ │ │ │

│ ┌───┤ │ ──┘ ┌─┘ │ ┴ │ ┴ ──┬─┘ │ ──┤ ┴ │

│ ┴ ──┘ ┴ ┌─┬─┘ ──┤ ──┘ ┌─┬─┘ ┌─┘ ──┘ ┌─┤

│ ┌─────┬─┤ │ ┌───┤ ┌─┬─┘ ┴ ──┘ ┌─────┘ │

│ │ ┌───┤ ┴ │ ┴ ──┘ │ ┴ ────────┘ ┌─────┤

│ ┴ │ ──┘ ┌─┘ ┌───┬─┘ ──────────┬─┘ ┌───┤

│ ┌─┘ ┌───┘ ┌─┤ ┌─┘ ──┬───┬───┬─┘ ┌─┘ ──┤

│ ┴ ──┤ ────┘ ┴ │ ┌─┬─┤ ┌─┤ ──┘ ──┤ ──┬─┤

│ ┌───┘ ────────┤ ┴ ┴ │ │ │ ──┬─┬─┘ ┌─┘ │

│ │ ────┬─────┬─┤ ┌─┬─┘ │ │ ┌─┘ │ ──┤ ──┤

│ │ ┌───┘ ┌───┘ │ │ │ ──┤ │ │ ──┤ ┌─┤ ──┤

│ │ ┴ ────┤ ──┬─┤ ┴ ┴ ──┤ │ ┴ ──┤ ┴ │ ──┤

│ ┴ ┌─┬───┤ ──┤ │ ────┬─┤ ┴ ────┤ ──┤ ┌─┤

│ ──┤ │ ──┘ ┌─┘ ┴ ──┬─┘ ┴ ──┬─┬─┘ ┌─┘ ┴ │

│ ──┘ ┴ ────┘ ──────┘ ──────┘ ┴ ──┘ ────┤

└───────────────────────────────────────┘

Юг-восток:

┌─┬─┬─┬─────────┬───┬─┬─┬─┬─┬───┬─┬─┬───┐

│ │ │ ├──────── ├── ┴ ┴ ┴ │ └── ┴ │ ├── │

│ │ │ └─┬─┬─┬─┐ └─┬─┬──── └───┬── │ └── │

│ │ └── │ ┴ │ └── ┴ ├─┬─┬──── ├─┐ ├───┐ │

│ ├─┬─┐ ├─┐ └─┬─┬── ┴ ┴ ├──── │ │ ├─┐ │ │

│ ┴ ┴ │ ┴ └── │ ├───┬─┐ ├─┬── │ ┴ │ ┴ │ │

├───┐ └───┬─┐ │ └── ┴ ┴ ┴ ├── └─┐ └─┐ │ │

├── ├───┐ │ ┴ └─────┬───┐ ├─┬── └─┐ ┴ ┴ │

├── └─┐ ┴ └─┬───┬─┐ ├── ┴ ┴ └───┐ └──── │

├─┬── ├──── ├── │ │ └─┬─┬───┬─┐ └───┬── │

│ ├─┐ ├──── └─┐ ┴ └─┐ │ ├── ┴ ├──── ├─┐ │

│ │ ┴ └─┬─┬── └──── ┴ ┴ └─┬─┐ ├──── ┴ ┴ │

│ ├──── ┴ └─┬───┬───┬─┬─┐ ┴ ┴ └───┬─┬── │

│ └─┬─┬───┐ └── └── ┴ ┴ ├───┬───┐ │ ├── │

├── │ ├─┐ ├───────┬─┬─┐ ├── └─┐ │ │ └─┐ │

├── ┴ │ │ └────── ┴ │ │ └──── │ │ └── ┴ │

├──── ┴ └─────┬─┬── ┴ ├─┬───┐ │ └───┬─┐ │

├─┬─┬─────┬─┐ │ ├─┬─┐ ┴ ├─┐ │ └───┐ │ ┴ │

│ │ └───┐ │ ┴ ┴ │ ┴ ├── │ │ └─┬── │ ├─┐ │

│ └──── ┴ └──── └── └── ┴ └── └── ┴ ┴ ┴ │

└───────────────────────────────────────┘

* 1. Алгоритмы поиска пути в лабиринте
     1. Подготовка лабиринта

Алгоритмы генерации, реализованные ранее, создают идеальные лабиринты. Это значит, что между любыми двумя ячейками существует путь, причем единственный. При поиске путей более интересной, на мой взгляд, является ситуация, когда между ячейками могут существовать альтернативные пути, причем стоимость (вес) пути между различными ячейками может быть различным.

Поэтому был создан класс mazeWeighted и написаны следующие функции:

WallsReduce – для разряжения лабиринта, т.е. случайного удаления его стен с заданной вероятнсотью.

void RandomCircules(mazeWeighted& MazeWeighted, std::default\_random\_engine& generator, int minWeight, int maxWeight, double probability, int meanRadius, double stddevRadiusRatio)

– для генерации весов лабиринта в виде кругов со случайным радиусом (имеющим нормальное распределение вероятности) и находящихся в случайных ячейках. мат.ожидание радиуса задается с помощью meanRadius, stddevRadiusRatio – среднеквадратическое отклонение радиуса круга, деленное на матожидание круга. Примеры получившихся лабиринтов можно видеть в примерах алгоритмов поиска.

* + 1. Ли
       1. Теоретические сведения

Алгори́тм волново́й трассиро́вки (волновой алгоритм, алгоритм Ли) — алгоритм поиска пути, алгоритм поиска кратчайшего пути на планарном графе. Принадлежит к алгоритмам, основанным на методах поиска в ширину.

В основном используется при компьютерной трассировке (разводке) печатных плат, соединительных проводников на поверхности микросхем. Другое применение волнового алгоритма — поиск кратчайшего расстояния на карте в компьютерных стратегических играх.

Волновой алгоритм в контексте поиска пути в лабиринте был предложен Э. Ф. Муром. Ли независимо открыл этот же алгоритм при формализации алгоритмов трассировки печатных плат в 1961 году.

Алгоритм работает на дискретном рабочем поле (ДРП), представляющем собой ограниченную замкнутой линией фигуру, не обязательно прямоугольную, разбитую на прямоугольные ячейки, в частном случае — квадратные. Множество всех ячеек ДРП разбивается на подмножества: «проходимые» (свободные), т. е при поиске пути их можно проходить, «непроходимые» (препятствия), путь через эту ячейку запрещён, стартовая ячейка (источник) и финишная (приемник). Назначение стартовой и финишной ячеек условно, достаточно — указание пары ячеек, между которыми нужно найти кратчайший путь.

Алгоритм предназначен для поиска кратчайшего пути от стартовой ячейки к конечной ячейке, если это возможно, либо, при отсутствии пути, выдать сообщение о непроходимости.

Работа алгоритма включает в себя три этапа: инициализацию, распространение волны и восстановление пути.

* + - 1. Алгоритм

В данной работе дискретное поле не содержит непроходимых ячеек, однако между ячейками на этом поле могут быть стены. Стоит отметить, что в данной реализации не используется очередь, характерная для алгоритмов поиска в ширину. Что, предположительно, может являться причиной того, что этот алгоритм более быстрый, чем алгоритм Дейкстры (который, будучи примененным для невзвешенных полей, должен давать аналогичный результат за, возможно, чуть большее время) на полях небольших размеров и значительно более медленный на полях больших размеров. (вместо извлечения элементов из очереди алгоритм пробегает ячейки в прямоугольной области, ограниченной ячейками фронта волны)

1. Вводится матрица весов путей (стоимости путей от стартовой ячейки до текущей)
2. Матрица так же используется для отличия посещенных ячеек от непосещенных (для непосещенных ячеек устанавливается вес, максимальный для int на данной платформе)
3. Начинается распространение волны от стартовой ячейки: ей присваивается нулевой вес.
4. Далее идет распространение волны: для каждой ячейки фронта волны просматриваются соседи. Если соседи ещё не были посещены, то им присваивается вес. Также отслеживается, достигнута ли финишная ячейка и удалось ли продвинутся фронту волны.
5. Если финишная ячейка достигнута, либо же фронт волны больше не может распространятся, то происходит завершение распространения волны. Если завершение было связано с достижением финишной ячейки, то начинается восстановление пути, в противном случае выдается сообщение о несуществовании пути, и происходит выход из функции.
6. Восстановление пути идет из финишной в начальную ячейку. Для того, чтобы восстановить путь нужно двигаться из финишной в начальную ячейку так, чтобы при движении вес пути уменьшался на единицу.
   * + 1. Пример работы

Пример, когда путь существует. (цифры – веса пути; ~ - непосещенные ячейки)

Поиск пути из (0, 1) в (1, 1)

1. Распространение волны от (0, 1)

┌───────┬───┐

│ ~ 0 │ ~ │

│ ────┘ │

│ ~ ~ ~ │

│ . . │

│ ~ ~ ~ │

└───────────┘

┌───────┬───┐

│ 1 0 │ ~ │

│ ────┘ │

│ ~ ~ ~ │

│ . . │

│ ~ ~ ~ │

└───────────┘

┌───────┬───┐

│ 1 0 │ ~ │

│ ────┘ │

│ 2 ~ ~ │

│ . . │

│ ~ ~ ~ │

└───────────┘

┌───────┬───┐

│ 1 0 │ ~ │

│ ────┘ │

│ 2 3 ~ │

│ . . │

│ ~ ~ ~ │

└───────────┘

1. Восстановление пути с конца.

┌───────┬───┐

│ 1 0 │ ~ │

│ ────┘ │

│ 2 3\* ~ │

│ . . │

│ ~ ~ ~ │

└───────────┘

Way= (1, 1)

┌───────┬───┐

│ 1 0 │ ~ │

│ ────┘ │

│ 2\* 3 ~ │

│ . . │

│ ~ ~ ~ │

└───────────┘

Way= (1, 0) (1, 1)

┌───────┬───┐

│ 1\* 0 │ ~ │

│ ────┘ │

│ 2 3 ~ │

│ . . │

│ ~ ~ ~ │

└───────────┘

Way= (0, 0) (1, 0) (1, 1)

┌───────┬───┐

│ 1 0\*│ ~ │

│ ────┘ │

│ 2 3 ~ │

│ . . │

│ ~ ~ ~ │

└───────────┘

Way= (0, 1) (0, 0) (1, 0) (1, 1) – Искомый путь.

* + 1. Ли (модификация с двумя волнами)

Более сложная реализация алгоритма Ли. Отличие заключается в том, что помимо волны, распространяющейся от стартовой ячейки, создается волна, распространяющаяся от финишной ячейки. Фронты волн продвигаются по дискретному полю поочередно, пока волны не встретятся, или пока одна из волн не сможет продвигаться дальше. В этой реализации помимо матрицы весов путей вводится матрица идентификаторов волн.

Если волны встретились, то путь восстанавливается в два этапа: сначала от стартовой ячейки до ячейки пересечения волн, затем от ячейки пересечения волн до финишной ячейки.

Данная реализация позволяет посещать примерно в два раза меньшее количество ячеек, чем вариант с одной волной.

* + - 1. Пример работы

Поиск пути от (0, 1) до (2, 4). Зеленый – волна от стартовой ячейки, голубой – от финишной.

Wave1 CurrDist=1

Wave2 CurrDist=1

┌───────────┬───┬───┐

│ 1 0 1 │ ~ │ ~ │

│ ┌───┐ │ ┴ │

│ ~ │ ~ │ ~ │ ~ ~ │

│ ┴ ┴ ┴ ────┤

│ ~ ~ ~ 1 0 │

│ . ────────────┤

│ ~ ~ ~ ~ ~ │

└───────────────────┘

Wave1 CurrDist=2

Wave2 CurrDist=2

┌───────────┬───┬───┐

│ 1 0 1 │ ~ │ ~ │

│ ┌───┐ │ ┴ │

│ 2 │ ~ │ 2 │ 2 ~ │

│ ┴ ┴ ┴ ────┤

│ ~ ~ 2 1 0 │

│ . ────────────┤

│ ~ ~ ~ ~ ~ │

└───────────────────┘

Волны пересеклись в ячейке (2, 2). Обозначим её красным цветом.

Wave1 CurrDist=3

┌───────────┬───┬───┐

│ 1 0 1 │ ~ │ ~ │

│ ┌───┐ │ ┴ │

│ 2 │ ~ │ 2 │ 2 ~ │

│ ┴ ┴ ┴ ────┤

│ 3 ~ 3 1 0 │

│ . ────────────┤

│ ~ ~ ~ ~ ~ │

└───────────────────┘

Рассмотрим теперь восстановление пути:

От стартовой до ячейки пересечения:

┌───────────┬───┬───┐

│ 1 0 1 │ ~ │ ~ │

│ ┌───┐ │ ┴ │

│ 2 │ ~ │ 2 │ 2 ~ │

│ ┴ ┴ ┴ ────┤

│ 3 ~ 3\* 1 0 │

│ . ────────────┤

│ ~ ~ ~ ~ ~ │

└───────────────────┘

Way= (2, 2)

┌───────────┬───┬───┐

│ 1 0 1 │ ~ │ ~ │

│ ┌───┐ │ ┴ │

│ 2 │ ~ │ 2\*│ 2 ~ │

│ ┴ ┴ ┴ ────┤

│ 3 ~ 3 1 0 │

│ . ────────────┤

│ ~ ~ ~ ~ ~ │

└───────────────────┘

Way= (1, 2) (2, 2)

┌───────────┬───┬───┐

│ 1 0 1\*│ ~ │ ~ │

│ ┌───┐ │ ┴ │

│ 2 │ ~ │ 2 │ 2 ~ │

│ ┴ ┴ ┴ ────┤

│ 3 ~ 3 1 0 │

│ . ────────────┤

│ ~ ~ ~ ~ ~ │

└───────────────────┘

Way= (0, 2) (1, 2) (2, 2)

┌───────────┬───┬───┐

│ 1 0\* 1 │ ~ │ ~ │

│ ┌───┐ │ ┴ │

│ 2 │ ~ │ 2 │ 2 ~ │

│ ┴ ┴ ┴ ────┤

│ 3 ~ 3 1 0 │

│ . ────────────┤

│ ~ ~ ~ ~ ~ │

└───────────────────┘

Way= (0, 1) (0, 2) (1, 2) (2, 2)

От ячейки пересечения до финишной:

┌───────────┬───┬───┐

│ 1 0 1 │ ~ │ ~ │

│ ┌───┐ │ ┴ │

│ 2 │ ~ │ 2 │ 2 ~ │

│ ┴ ┴ ┴ ────┤

│ 3 ~ 3 1\* 0 │

│ . ────────────┤

│ ~ ~ ~ ~ ~ │

└───────────────────┘

Way= (0, 1) (0, 2) (1, 2) (2, 2) (2, 3)

┌───────────┬───┬───┐

│ 1 0\* 1 │ ~ │ ~ │

│ ┌───┐ │ ┴ │

│ 2 │ ~ │ 2 │ 2 ~ │

│ ┴ ┴ ┴ ────┤

│ 3 ~ 3 1 0\*│

│ . ────────────┤

│ ~ ~ ~ ~ ~ │

└───────────────────┘

Way= (0, 1) (0, 2) (1, 2) (2, 2) (2, 3) (2, 4) – Искомый путь

* + 1. Дейкстры
       1. Алгоритм

Алгоритм Дейкстры – это алгоритм на графе. На ДРП этот алгоритм применяется, когда веса (стоимости) перехода (не путать с путевыми весами) из одной ячейки в другую могут быть различны. Если все веса одинаковы, то алгоритм Дейкстры будет давать тот же результат, как и поиск в ширину.

Основное отличие алгоритма Дейкстры от алгоритма поиска в ширину – использование очереди с приоритетами.

Итак, при работе алгоритма будут введены: PathCoordWeights – матрица, каждый элемент которой – это путевые координаты ячейки и её путевой вес.

PriorQueue – очередь с приоритетами, меньший вес – больший приоритет. Будет хранить в себе координаты ячеек.

1. Устанавливаем путевой вес стартовой ячейки равным 0.
2. Помещаем стартовую ячейку в очередь
3. Извлекаем ячейку из очереди
4. Проверяем всех её соседей (всего 4 соседа) :Если сосед ни разу не был посещен, то устанавливаем для него путевые координаты(с какого направления к нему пришли) и путевые веса. Если сосед уже был посещен, то находим для него новый путевой вес и сравниваем с текущим. Если новый вес оказался не меньше текущего, то не делаем ничего, если же новый вес оказался меньше текущего, то посещенному соседу присваиваем новый вес, новые путевые координаты и помещаем в очередь (он оказывается в ней уже повторно)
5. Продолжаем выполнять пункты 3-4, пока извлеченный элемент не окажется финишной ячейкой, либо пока в очереди не останется элементов.
6. Узнав причину остановки, делаем вывод о том, существует ли путь или нет.
7. Если путь существует, то восстанавливаем его, двигаясь по путевым координатам от финиша к старту.
   * + 1. Пример работы

Пусть исходный лабиринт выглядит так:

┌───────┬───────────┐

│ 1 1 │ 1 1 1 │

│ ────┘ ┬ ┌───┤

│ 1 1 8 │ 1 │ 6 │

│ . ┬ ┴ ┴ │

│ 1 8 │ 8 8 6 │

│ ────┴───┬───┐ │

│ 2 1 8 │ 1 │ 6 │

│ ┌──── ┴ └───┤

│ 6 │ 3 3 3 1 │

└───┴───────────────┘

Цифры означают веса ячеек, т.е. стоимость при перемещении в данную ячейку из соседней.

Найдем путь из ячейки (0, 0, ) в ячейку (2, 2). Отмечены желтым.

1. Выставим атрибуты первой ячейки и поместим её в очередь.

Путевые веса:

┌───────┬───────────┐

│ 0\* 0 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 0 0 0 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 0 0 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 0 0 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

Путевые координаты:

┌───────┬───────────┐

│ s\* ~ │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ ~ ~ ~ │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ ~ ~ │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ ~ ~ ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

1. Извлечем эту ячейку из очереди и пройдемся по её соседям. Вначале посмотрим левого соседа:

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 0 0 0 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 0 0 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 0 0 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ ~ ~ ~ │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ ~ ~ │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ ~ ~ ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

1. Потом нижнего соседа

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 0 0 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 0 0 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 0 0 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v ~ ~ │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ ~ ~ │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ ~ ~ ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

1. Теперь в очереди находятся 2 ячейки с одинаковым приоритетом 1. Это ячейки (0, 1) и   
   (1, 0). Извлечем из очереди ячейку (0, 1). Она имеет единственного соседа, неотделенного стеной – это ячейка (0, 0). Найдем новый вес для неё: новый путевой вес(0, 0)=путевой вес(0, 1)+вес перемещения(0, 0)=1+1=2 Это больше, чем 0. Значит, с ячейкой (0, 0) мы ничего не делаем.

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1\*│ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 0 0 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 0 0 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 0 0 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s >\*│ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v ~ ~ │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ ~ ~ │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ ~ ~ ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

1. Теперь в очереди находится единственная ячейка (1, 0). Она имеет трех соседей. Обойдем их последовательно.

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1\* 2 0 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 0 0 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 0 0 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v\* > ~ │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ ~ ~ │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ ~ ~ ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1\* 2 0 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 0 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 0 0 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v\* > ~ │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v ~ │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ ~ ~ ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

1. В очереди теперь две ячейки (1, 1) и (2, 0). Обе эти ячейки имеют приоритет 2. Извлечем из очереди ячейку (1, 1) и пройдемся по её соседям.

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 2\* 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 0 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 0 0 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v >\* > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v ~ │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ ~ ~ ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 2\* 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 0 0 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v >\* > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ ~ ~ ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

1. Теперь в очереди лежит по прежнему ячейка (2, 0) с приоритетом 2 и лежат ячейки (1, 2) и (2, 1) с приоритетами 10. Выберем наиболее приоритетную ячейку, т.е. ячейку (2, 0). Пройдемся по её соседям.

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2\* 10 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 4 0 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v > > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v\* v │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ v ~ ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

1. Продолжаем в том же духе

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 4 5 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│ 0 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v > > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ v > ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ ~ │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 4 5 0 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v > > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ v > ~ │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │ 0 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ~ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v > > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │ 0 0 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v > > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ ~ ~ ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ ~ ~ ~ │

└───┴───────────────┘

1. На этом шаге мы уже добрались до финишной ячейки. Но она не самая приоритетная в очереди, поэтому поиск продолжается (существуют случаи, когда удается найти путь короче, чем тот, который бы мы получили, сразу остановив поиск. Более того, когда мы дождемся, когда «фронт волны» полностью прокатится через полученный вес, то найденный путь окажется наикратчайшим из возможных. Т.е. алгоритм Дейкстры (в данной интерпретации, т.е. когда мы можем многократно ставить в очередь уже просмотренные элементы) гарантированно находит наикратчайший путь.).

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 0 0 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │18 0 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ ~ ~ │

│ ────┘ ┬ ┌───┤

│ v > > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v ~ ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 12 0 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │18 0 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ > ~ │

│ ────┘ ┬ ┌───┤

│ v > > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v ~ ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 12 13 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │ 0 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │18 0 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > > │ ~ │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v ~ ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 12 13 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │13 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │18 0 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v ~ ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 12 13 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │13 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │18 21 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 0 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ ~ ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 12 13 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │13 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │18 21 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 16 0 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ v ~ ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 12 13 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │13 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │18 21 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 16 19 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ v > ~ │

└───┴───────────────┘

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 12 13 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │13 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │18 21 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │19 16 19 0 │

└───┴───────────────┘

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ < v > ~ │

└───┴───────────────┘

1. Наконец, очередной элемент, извлеченный из очереди, оказывается финишной ячейкой. На этом поиск прекращается, и по путевым координатам восстанавливается путь.( с конца в начало)

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v\* v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ < v > ~ │

└───┴───────────────┘

Path= (2, 2)

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > >\*│ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ < v > ~ │

└───┴───────────────┘

Path= (1, 2) (2, 2)

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v >\* > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ < v > ~ │

└───┴───────────────┘

Path=(1, 1) (1, 2) (2, 2)

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v\* > > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ < v > ~ │

└───┴───────────────┘

Path=(1, 0) (1, 1) (1, 2) (2, 2)

┌───────┬───────────┐

│ s\* > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ < v > ~ │

└───┴───────────────┘

Path=(0, 0) (1, 0) (1, 1) (1, 2) (2, 2) – искомый путь

Вершина очереди при «распространении волны» на каждой итерации выглядит так (приоритет\_он\_же\_путевой\_вес, (I, j)):

PriorQueue all items:

PriorQueue.top()=(0, (0, 0))

PriorQueue.top()=(1, (0, 1))

PriorQueue.top()=(1, (1, 0))

PriorQueue.top()=(2, (1, 1))

PriorQueue.top()=(2, (2, 0))

PriorQueue.top()=(4, (3, 0))

PriorQueue.top()=(5, (3, 1))

PriorQueue.top()=(10, (1, 2))

PriorQueue.top()=(10, (2, 1))

PriorQueue.top()=(10, (4, 0))

PriorQueue.top()=(11, (0, 2))

PriorQueue.top()=(12, (0, 3))

PriorQueue.top()=(13, (0, 4))

PriorQueue.top()=(13, (1, 3))

PriorQueue.top()=(13, (3, 2))

PriorQueue.top()=(16, (4, 2))

PriorQueue.top()=(18, (2, 2))

* + 1. А\* (AStar; А со звездой)
       1. Теоретические сведения

Поиск A\* (произносится «А звезда» или «А стар», от англ. A star) — в информатике и математике, алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как f(x)). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как g(x) и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как h(x)).

Функция h(x) должна быть допустимой эвристической оценкой, то есть не должна переоценивать расстояния к целевой вершине. Например, для задачи маршрутизации h(x) может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.

Этот алгоритм был впервые описан в 1968 году Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем. Это по сути было расширение алгоритма Дейкстры, созданного в 1959 году. Новый алгоритм достигал более высокой производительности (по времени) с помощью эвристики. В их работе он упоминается как «алгоритм A». Но так как он вычисляет лучший маршрут для заданной эвристики, он был назван A\*.

Обобщением для него является двунаправленный эвристический алгоритм поиска.

* + - 1. Алгоритм

Реализация данного алгоритма мало отличается от реализации алгоритма Дейкстры.

Для оценки расстояния от текущей ячейки до финишной вводится эвристическая функция, состоящая из одного оператора:

return abs(finishi-Currenti)+abs(finishj-Currentj);

Отличие от алгоритма Дейкстры в том, что в качестве приоритета в очереди берутся не путевые веса, а сумма путевого веса и эвристической оценки расстояния до финишной ячейки.

* + - 1. Пример работы

Исходный лабиринт с весами ячеек:

Weights:

┌───────┬───────────┐

│ 1 1 │ 1 1 1 │

│ ────┘ ┬ ┌───┤

│ 1 1 8 │ 1 │ 6 │

│ . ┬ ┴ ┴ │

│ 1 8 │ 8 8 6 │

│ ────┴───┬───┐ │

│ 2 1 8 │ 1 │ 6 │

│ ┌──── ┴ └───┤

│ 6 │ 3 3 3 1 │

└───┴───────────────┘

Вершина очереди на каждой итерации:

PriorQueue all items:

PriorQueue.top()=(4, (0, 0))

PriorQueue.top()=(4, (0, 1))

PriorQueue.top()=(4, (1, 0))

PriorQueue.top()=(4, (1, 1))

PriorQueue.top()=(4, (2, 0))

PriorQueue.top()=(7, (3, 0))

PriorQueue.top()=(7, (3, 1))

PriorQueue.top()=(11, (1, 2))

PriorQueue.top()=(11, (2, 1))

PriorQueue.top()=(13, (0, 2))

PriorQueue.top()=(14, (3, 2))

PriorQueue.top()=(14, (4, 0))

PriorQueue.top()=(15, (0, 3))

PriorQueue.top()=(15, (1, 3))

PriorQueue.top()=(17, (0, 4))

PriorQueue.top()=(18, (2, 2))

PathWeights % 1000:

┌───────┬───────────┐

│ 0 1 │11 12 13 │

│ ────┘ ┬ ┌───┤

│ 1 2 10 │13 │ 0 │

│ . ┬ ┴ ┴ │

│ 2 10 │18 21 0 │

│ ────┴───┬───┐ │

│ 4 5 13 │ 0 │ 0 │

│ ┌──── ┴ └───┤

│10 │ 0 16 0 0 │

└───┴───────────────┘

Как видим, путевые веса аналогичны путевым весам в алгоритме Дейкстры, но приоритеты внутри очереди стали другими. Например, ячейка (4, 2) имела в очереди приоритет 18, т.е. такой же, как и у финишной ячейки (в вершину очереди эта ячейка не попала).

PathCoordinates:

┌───────┬───────────┐

│ s > │ ^ > > │

│ ────┘ ┬ ┌───┤

│ v > > │ v │ ~ │

│ . ┬ ┴ ┴ │

│ v v │ v v ~ │

│ ────┴───┬───┐ │

│ v > > │ ~ │ ~ │

│ ┌──── ┴ └───┤

│ v │ ~ v ~ ~ │

└───┴───────────────┘

Построение пути для A\* аналогично построению пути в алгоритме Дейкстры.

The finish cell is reached!

Path= (0, 0) (1, 0) (1, 1) (1, 2) (2, 2)

Тип: Выбором

Сложность: **O(n\*log(n))**

Соритровка двоичной кучей состоит из двух этапов: построение двоичной кучи и удаление отсоритрованных элементов по одному с последующей перестройкой дерева(восстановление свойств кучи). Далее будем рассматривать сортировку по возрастанию.

Известно, что дерево, удовлетворяющее 2 и 3 условиям, можно представить в виде массива длиной n. Причем элемент **i** будет иметь потомков с индексами **2i+1** и **2i+2**

1. Значение в любой вершине не меньше, чем значения её потомков[К 1].
2. Глубина всех листьев (расстояние до корня) отличается не более чем на 1 слой.
3. Последний слой заполняется слева направо без «дырок».
   * 1. Восстановление свойств кучи.

Если в куче изменить корневой элемент, то она может перестать удовлетворять условиям кучи. Для восстановления свойств кучи была написана функция Heapify.

1. Сравнивается значение корневого элемента со значениями его непосредственных потомков, и если значение корня оказалось не наибольшим, то производится обмен значения корня со значением потомка, у которого значение наибольшее из этих 3 элементов.
2. Рекурсивно вызывается Heapify, для поддерева, которое было затронуто в результате такого обмена. Тривиальным случаем здесь является ситуация, когда у узла нет потомков.
   * 1. Построение кучи.
3. Для всех элементов, имеющих потомки, вызывается Heapify начиная от наиболее младших элементов.
   * 1. Сортировка кучей
4. После построения кучи корневой элемент (т.е. элемент с индексом **0**) является наибольшим. Происходит обмен корневого элемента(0го в массиве) и наиболее правого листа в дереве (последнего элемента в массиве), таким образом, элемент с наибольшим значением занимает свою позицию в массиве и исключается из дерева.
5. Для корня дерева, получившегося после исключения элемента с наибольшим значением, вызывается функция Heapify, после чего в корень дерева попадает наибольший элемент из оставшихся.
6. Этот элемент меняется с наиболее правым листом дерева(предпоследним элементом в массиве), в результате чего этот элемент занимает свою позицию в отсортированном массиве.
7. И так далее, пока массив не окажется отсортированным.

.

* 1. Быстрая сортировка Хоара

Тип: Обменная

Сложность: O(n^2)

Один из самых быстрых известных универсальных алгоритмов сортировки массивов.

1. Среди элементов массива выбирается базовый ключ. В нашем алгоритме это элемент, расположенный посередине массива.
2. Индексы **i j** предназначены, чтобы пробегать по массиву с начала в сторону конца и с конца в сторону начала. Итак, после выбора базового ключа **i** начинает пробегать от начала в конец, пока не встретит элемент, который окажется не меньше, чем базовый ключ.
3. Индекс **j** начинает пробегать с конца в начало, пока не встретит элемент, который окажется не больше, чем базовый ключ.
4. i и j элементы меняются местами. (таким образом, все элементы, стоящие в начале и в конце массива оказываются соответственно меньше и больше базового ключа)
5. 2 и 3 шаг повторяются, пока I и j не встретятся. В результате чего массив оказывается разделен на 2 части: в левой части элементы <= базовому ключу, а в правой части >= базовому ключу.
6. Далее для левой и правой частей массива функция вызывается рекурсивно. Тривиальный случай – когда массив состоит из одного эелмента.
   1. Сортировка Шелла

Тип: Вставками

Сложность: O(n^2)

Алгоритм представляет собой 3 вложенных в друг друга циклы

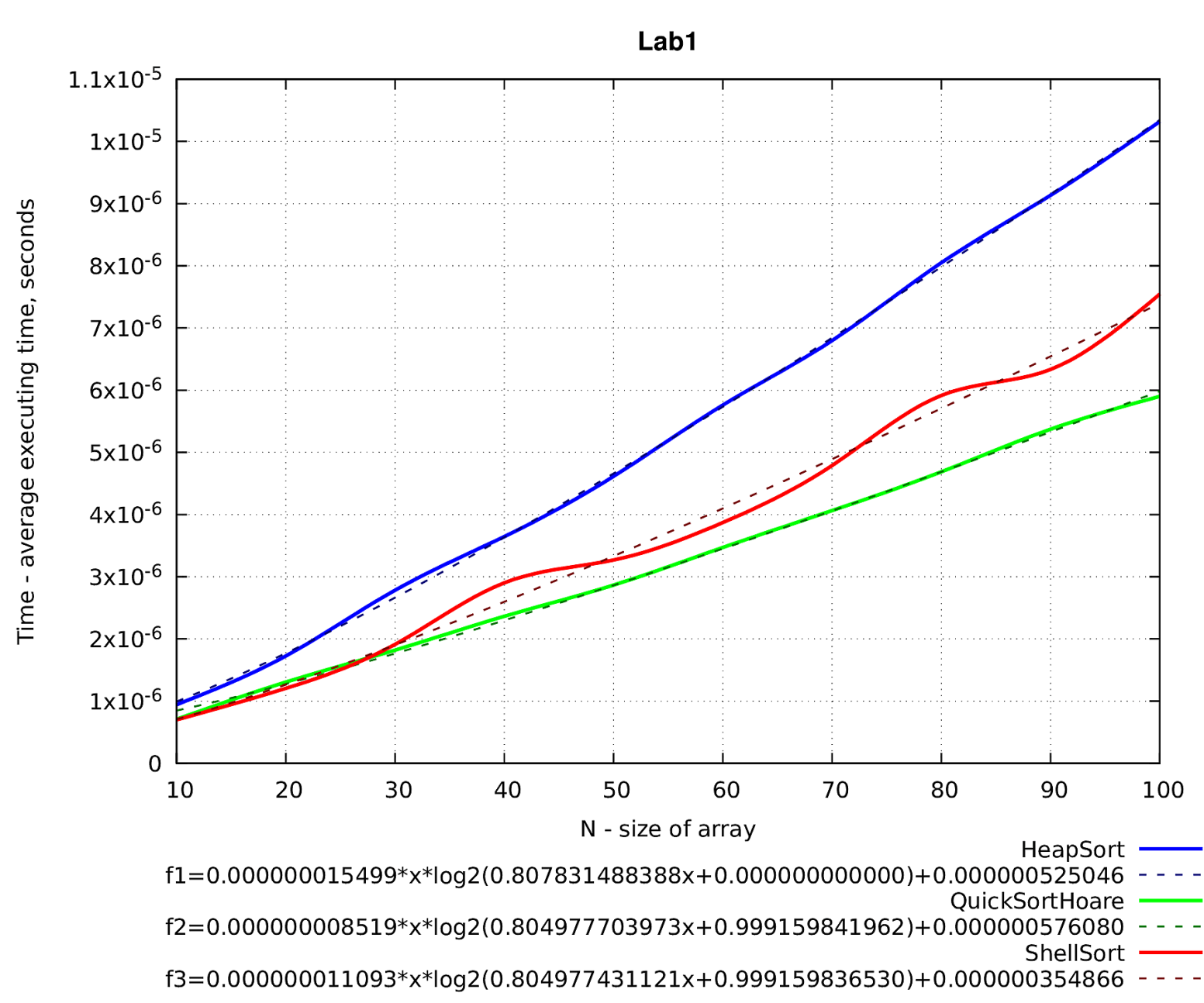
1. В цикле с уровнем вложенности **0** выбирается начальный шаг (размер массива пополам), через который отстоят элементы в каждом подсписке, на которые (умозрительно) разбивается массив, и которые не пересекаются друг с другом. В циклическом выражении цикла for шаг уменьшается в 2 раза.
2. Цикл с уровнем вложенности **1** выполняет операции над подсписками. Т.е. каждая его итерация связана с сортировкой некоторого подсписка. Но сортировка каждого подсписка может и не выполнятся за одну итерацию. Т.е. одной итерации соответствует один подсписок, но одному подсписку может соответствовать несколько итераций этого цикла.
3. Цикл с уровнем вложенности **2**. В этом цикле происходит сортировка части подсписка. После того, как этот цикл будет выполнен целиком, окажется упорядочена некоторая часть подсписка (его начало). Сколько конкретно будет упорядоченно элементов в начальной части массива зависит от текущего положения счетчика цикла с уровнем вложенности **1**.
4. Сравнительный анализ

|  |
| --- |
| HeapSort:  ===Arr (before sort):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  BuildHeap Started!  BuildHeap: element will be Heapified:2  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  BuildHeap: element will be Heapified:1  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  -Heapify started!  -===Arr (Before Heapify):===  -Arr[0]=87  -Arr[1]=78  -Arr[2]=28  -Arr[3]=26  -Arr[4]=6  -Heapify finished!  -===Arr (After Heapify):===  -Arr[0]=87  -Arr[1]=78  -Arr[2]=28  -Arr[3]=26  -Arr[4]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=87  Arr[1]=78  Arr[2]=28  Arr[3]=26  Arr[4]=6  BuildHeap: element will be Heapified:0  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=87  Arr[1]=78  Arr[2]=28  Arr[3]=26  Arr[4]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=87  Arr[1]=78  Arr[2]=28  Arr[3]=26  Arr[4]=6  BuildHeap Finished!  ===Arr (After BuildHeap):===  Arr[0]=87  Arr[1]=78  Arr[2]=28  Arr[3]=26  Arr[4]=6  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=6  Arr[1]=78  Arr[2]=28  Arr[3]=26  -Heapify started!  -===Arr (Before Heapify):===  -Arr[0]=78  -Arr[1]=6  -Arr[2]=28  -Arr[3]=26  - -Heapify started!  - -===Arr (Before Heapify):===  - -Arr[0]=78  - -Arr[1]=26  - -Arr[2]=28  - -Arr[3]=6  - -Heapify finished!  - -===Arr (After Heapify):===  - -Arr[0]=78  - -Arr[1]=26  - -Arr[2]=28  - -Arr[3]=6  -Heapify finished!  -===Arr (After Heapify):===  -Arr[0]=78  -Arr[1]=26  -Arr[2]=28  -Arr[3]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=78  Arr[1]=26  Arr[2]=28  Arr[3]=6  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  -Heapify started!  -===Arr (Before Heapify):===  -Arr[0]=28  -Arr[1]=26  -Arr[2]=6  -Heapify finished!  -===Arr (After Heapify):===  -Arr[0]=28  -Arr[1]=26  -Arr[2]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=28  Arr[1]=26  Arr[2]=6  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=6  Arr[1]=26  -Heapify started!  -===Arr (Before Heapify):===  -Arr[0]=26  -Arr[1]=6  -Heapify finished!  -===Arr (After Heapify):===  -Arr[0]=26  -Arr[1]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=26  Arr[1]=6  Heapify started!  ===Arr (Before Heapify):===  Arr[0]=6  Heapify finished!  ===Arr (After Heapify):===  Arr[0]=6  ===Arr (after sort):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  Arr[3]=78  Arr[4]=87  CurrentSize=5; Iteration 0 is complete  QuickSortHoare:  ===Arr (before sort):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  pivot is Arr[2]=p=28  Current i:(Arr[i] !< p) =0; Current j:(Arr[j] !> p) =3  ===Arr (After swap):===  Arr[0]=26  Arr[1]=6  Arr[2]=28  Arr[3]=87  Arr[4]=78  Current i:(Arr[i] !< p) =2; Current j:(Arr[j] !> p) =2  ===Arr (After swap):===  Arr[0]=26  Arr[1]=6  Arr[2]=28  Arr[3]=87  Arr[4]=78  pivot is Arr[1]=p=6  Current i:(Arr[i] !< p) =0; Current j:(Arr[j] !> p) =1  -===Arr (After swap):===  -Arr[0]=6  -Arr[1]=26  pivot is Arr[1]=p=78  Current i:(Arr[i] !< p) =0; Current j:(Arr[j] !> p) =1  -===Arr (After swap):===  -Arr[0]=78  -Arr[1]=87  ===Arr (after sort):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  Arr[3]=78  Arr[4]=87  CurrentSize=5; Iteration 0 is complete  ShellSort:  ===Arr (before sort):===  Arr[0]=87  Arr[1]=6  Arr[2]=28  Arr[3]=26  Arr[4]=78  ===Arr (After Step s=2):===  Arr[0]=28  Arr[1]=6  Arr[2]=78  Arr[3]=26  Arr[4]=87  ===Arr (After Step s=1):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  Arr[3]=78  Arr[4]=87  ===Arr (after sort):===  Arr[0]=6  Arr[1]=26  Arr[2]=28  Arr[3]=78  Arr[4]=87  CurrentSize=5; Iteration 0 is complete |

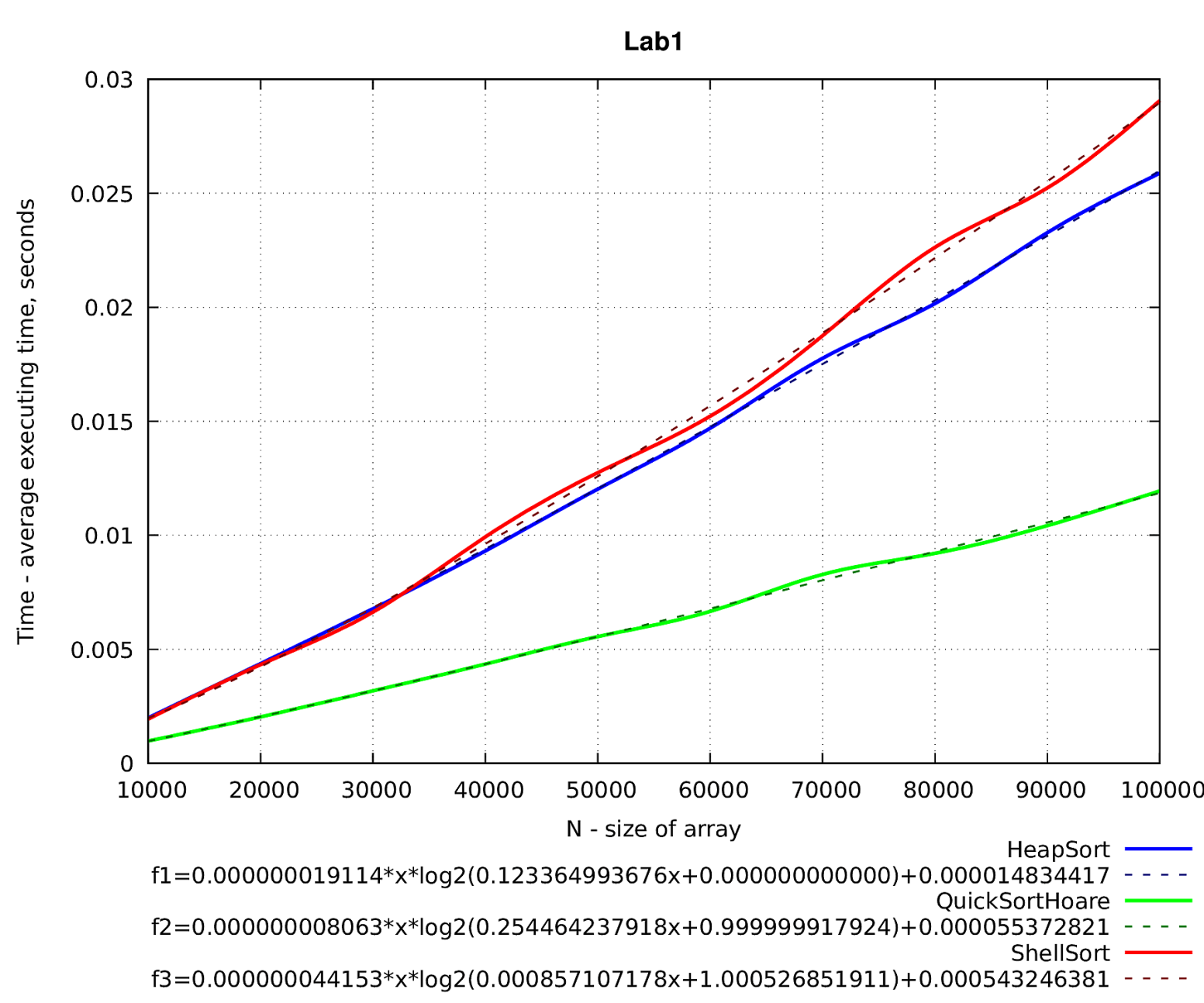
Пошаговое выполнение для массива размера 5

В таблице 1 показано пошаговое выполнение алгоритмов сортировки. Текущий уровень рекурсии обозначен «-».

1. Результаты работы



Время работы от размера массива



Время работы от размера массива.

Как видно из графиков, наиболее быстрой сортировкой оказалась быстрая сортировка Хоара. Сортировка двоичной кучей по времени работы сопоставима с сортировкой Шелла. Сортировка двоичной кучей оказалась медленнее соритровки Шелла на массивах меньшего размера(до 40000 элементов) и быстрее на массивах большего размера (40000 элементов и более)

При n=10^4

* t\_HeapSort/t\_QuickSortHoare= 2,03731037310373
* t\_ShellSort/t\_QuickSortHoare= 1,98236982369824
* t\_ShellSort/t\_HeapSort=0,973032803380962

При n=10^5

* t\_HeapSort/t\_QuickSortHoare=2,16707832712354
* t\_ShellSort/t\_QuickSortHoare=2,43336542889438
* t\_ShellSort/t\_HeapSort=1,1228783927364

Линии тренда, представляющие собой функцию вида

хорошо ложатся на полученные данные.

При рассмотрении графиков можно заметить, что сортировка Шелла в большей степени чувствительна к характеристикам входного массива, чем сортировка кучей и быстрая сортировка Хоара.

1. Выводы
2. В ходе проведенной работы были изучены алгоритмы сортировки двоичной кучей, быстрой сортировки Хоара, сортировки Шелла.
3. Была освоена программа gnuplot, были расширены знания по использованию утилиты CMake.
4. Был разработан шаблонный класс для сбора статистики с функций сортировки, а также функции для сортировок двоичной кучей, быстрой сортировкой Хоара, сортировкой Шелла.
5. Анализ показал, что алгоритм быстрой сортировки Хоара значительно быстрее алгоритмов сортировки двоичной кучей и Шелла.
6. Исходный код
   1. CMakeLists.txt

cmake\_minimum\_required(VERSION 3.0.0)

project(statistics VERSION 0.1.0)

#include(CTest)

#enable\_testing()

add\_executable(statistics main.cpp)

#install(FILES PlotScript.gpi DESTINATION ${CMAKE\_CURRENT\_BINARY\_DIR}) Не подходит, т.к. не сохраняет права доступа

install(PROGRAMS PlotScript.gpi DESTINATION ${CMAKE\_CURRENT\_BINARY\_DIR})

#add\_custom\_target(new\_target

# ${CMAKE\_COMMAND} -E copy PlotScript.gpi ${CMAKE\_BINARY\_DIR}/PlotScript.gpi

# )

#add\_executable(QuickSortHoare QuickSortHoare.cpp)

set(CPACK\_PROJECT\_NAME ${PROJECT\_NAME})

set(CPACK\_PROJECT\_VERSION ${PROJECT\_VERSION})

include(CPack)

* 1. main.cpp

#include <iostream>

#include "statistics.inl"

#include "HeapSort.inl"

#include "QuickSortHoare.inl"

#include "ShellSort.inl"

int main(int, char\*\*) {

srand(time(0));

int seed=rand();

//Аргументы подставлены для пошагового выполнения

/\*

std::cout<<"HeapSort:"<<std::endl;

statistics<int> s1(HeapSort<int>, 5, 5, 10000, false, true, 100, 1,(char\*)"HeapSortOut.txt", seed);

std::cout<<"QuickSortHoare:"<<std::endl;

statistics<int> s2(QuickSortHoare, 5, 5, 10000, false, true, 100, 1,(char\*)"QuickSortHoareOut.txt", seed);

std::cout<<"ShellSort:"<<std::endl;

statistics<int> s3(ShellSort, 5, 5, 10000, false, true, 100, 1,(char\*)"ShellSortOut.txt", seed);

\*/

//Агрументы для табуляции среднего времени выполнения

std::cout<<"HeapSort:"<<std::endl;

statistics<int> s1(HeapSort<int>, 1000, 10000, 1000, false, false, 100, 1,(char\*)"HeapSortOut.txt", seed);

std::cout<<"QuickSortHoare:"<<std::endl;

statistics<int> s2(QuickSortHoare, 1000, 10000, 1000, false, false, 100, 1,(char\*)"QuickSortHoareOut.txt", seed);

std::cout<<"ShellSort:"<<std::endl;

statistics<int> s3(ShellSort, 1000, 10000, 1000, false, false, 100, 1,(char\*)"ShellSortOut.txt", seed);

system("./PlotScript.gpi");

/\*

int Arr[10];

Arr[0]=132;

Arr[1]=97;

Arr[2]=93;

Arr[3]=170;

Arr[4]=176;

Arr[5]=131;

Arr[6]=48;

Arr[7]=32;

Arr[8]=92;

Arr[9]=114;

QuickSortHoare(Arr, 10, false, true, 0);

printArr(Arr, 10, (char \*)"AfterSort", 0); \*/

}

* 1. statistics.inl

#ifndef STATISTICS\_INL\_INCLUDED

#define STATISTICS\_INL\_INCLUDED

#include <limits> //для определения максимального числа в типе

#include <time.h> //для измерения времени

#include <fstream> //для файлов

#include <iomanip> // std::setprecision

template<typename T>

class statistics

{

private:

int CurrentSize=0; //текущее число элементов в массиве

T\* Arr=nullptr;

int NStart=0, NEnd=0, NStep=0;

int RandMax=-1; //-1 случ значения в массиве будут генерироваться на всем допустимом диапазоне для данного типа.

int NumberOfRuns=1; //число прогонов для каждого размера массива

public:

statistics(void (\*callback)(T\*, int, bool, bool, int), int NNStart, int NNEnd, int NNStep, bool SortOrder, bool IsStepbystep, int NRandMax, int NNumberOfRuns, char\* filename, int seed): NStart(NNStart), NEnd(NNEnd), NStep(NNStep), RandMax(NRandMax), NumberOfRuns(NNumberOfRuns) //Начальное, конечное значение числа элементов; IsStepbystep выводить пошагово; RandMax макс случайное число (-1 будут значения от 0 до максимального в данном типе). filename - файл для вывода seed для генерации одинаковых массивов

{

if(seed!=-1) srand(seed);

printLabel(filename);

for(CurrentSize=NStart; CurrentSize<=NEnd; CurrentSize+=NStep)

{

Arr=new T [CurrentSize];

double timeSeconds=0;

for(int k=0; k<NumberOfRuns; k++) //

{

for(int j=0; j<CurrentSize; j++) //заполняем массив

{

if(RandMax==-1) Arr[j]=rand() % ((unsigned long long)std::numeric\_limits<T>::max()+1); //по максимальное значение в этом типе

else Arr[j]=rand() % RandMax;

}

if(IsStepbystep) printArr((char\*)"before sort"); //выводим массив со случайными значениями

clock\_t timeStart=clock(); //число тиков с начала выполнения программы

callback(Arr, CurrentSize, SortOrder, IsStepbystep, 0);

clock\_t timeEnd=clock();

timeSeconds+=double(timeEnd-timeStart)/CLOCKS\_PER\_SEC; //время работы алгоритма в секундах

if(IsStepbystep) printArr((char\*)"after sort"); //выводим массив после сортировки

std::cout<<"CurrentSize="<< CurrentSize <<"; Iteration "<<k<<" is complete"<<std::endl;

}

timeSeconds/=NumberOfRuns;

delete[] Arr;

printValue(CurrentSize, timeSeconds, filename);

}

}

void printArr(char\* label)

{

std::cout<<"===Arr ("<< label<<"):==="<<std::endl;

for(int i=0; i<CurrentSize; i++)

{

std::cout<<"Arr["<<i<<"]="<<Arr[i]<<std::endl;

}

}

void printLabel(char\* filename) //пишем подпись оси размера массива в файл

{

std::ofstream fd(filename, std::ios\_base::out|std::ios\_base::trunc); //для записи очистив

fd<< "#filename="<< filename<<"; RandMax="<<RandMax<<"; NumberOfRuns="<<NumberOfRuns<<std::endl;

/\* for(int i=NStart; i<=NEnd;i+=NStep)

{

fd<<i<<" ";

}

fd<<std::endl; \*/

fd.close();

}

void printValue(int CurrentSize, double value, char\* filename) //пишем время выполнения в файл

{

std::ofstream fd(filename, std::ios\_base::app); //для записи в конец

fd<< CurrentSize<< "\t"<<std::fixed <<std::setprecision(9)<<value<<std::endl;

// fd<<std::endl;

fd.close();

}

};

#endif //STATISTICS\_INL\_INCLUDED

* 1. HeapSort.inl

#ifndef HEAPSORT\_INL\_INCLUDED

#define HEAPSORT\_INL\_INCLUDED

template<typename T>

void printArr(T \*Arr, int Length, const char\* label, int reclvl) //

{

for(int j=0; j<reclvl;j++) std::cout<<"\t-";

std::cout<<"===Arr ("<< label<<"):==="<<std::endl;

for(int i=0; i<Length;i++)

{

for(int j=0; j<reclvl;j++) std::cout<<"\t-";

std::cout<<"Arr["<<i<<"]="<<Arr[i]<<std::endl;

}

}

template<typename T>

void Heapify(T \*Arr, int Length, int i, bool SortOrder, bool IsStepbystep, int reclvl) //Восстанавливает свойства кучи, начиная с корня i. Предполагается, что левое и правое деревья удовлетворяют куче. SortOrder - направление сортировки. по умолчанию - по возрастанию(т.е. в корне кучи стоит наибольший элемент).

{

if(IsStepbystep)

{

for(int i=0; i<reclvl;i++) std::cout<<"\t-";

std::cout<<"Heapify started!"<<std::endl;

printArr(Arr, Length, (char\*)"Before Heapify", reclvl);

getchar();

}

reclvl++;

int LargestLowest=i;

if(!SortOrder) //по умолчанию сортировка по возрастанию

{

if(2\*i+1<Length && Arr[2\*i+1]>Arr[LargestLowest]) LargestLowest=2\*i+1;

if(2\*i+2<Length && Arr[2\*i+2]>Arr[LargestLowest]) LargestLowest=2\*i+2;

}

else

{

if(2\*i+1<Length && Arr[2\*i+1]<Arr[LargestLowest]) LargestLowest=2\*i+1;

if(2\*i+2<Length && Arr[2\*i+2]<Arr[LargestLowest]) LargestLowest=2\*i+2;

}

if(LargestLowest!=i)

{

T temp=Arr[LargestLowest];

Arr[LargestLowest]=Arr[i];

Arr[i]=temp;

Heapify<T>(Arr, Length, LargestLowest, SortOrder, IsStepbystep, reclvl);

}

reclvl--;

if(IsStepbystep)

{

for(int i=0; i<reclvl;i++) std::cout<<"\t-";

std::cout<<"Heapify finished!"<<std::endl;

printArr(Arr, Length, (char\*)"After Heapify", reclvl);

getchar();

}

}

template<typename T>

void BuildHeap(T \*Arr, int Length, bool SortOrder, bool IsStepbystep) //строит кучу из неупорядоченного массива SortOrder - направление сортировки. по умолчанию - по возрастанию.(т.е. в корне кучи стоит наибольший элемент).

{

if(IsStepbystep) std::cout<<"BuildHeap Started!"<<std::endl;

for(int i=Length/2; i>=0; i--)

{

if(IsStepbystep) std::cout<<"BuildHeap: element will be Heapified:"<<i<<std::endl;

Heapify<T>(Arr, Length, i, SortOrder, IsStepbystep, 0);

}

if(IsStepbystep)

{

std::cout<<"BuildHeap Finished!"<<std::endl;

printArr(Arr, Length, (char\*)"After BuildHeap", 0);

getchar();

}

}

template<typename T>

void HeapSort(T \*Arr, int Length, bool SortOrder, bool IsStepbystep, int reclvl) //сортировка кучей. По умолчанию - по возрастанию.

{

BuildHeap<T>(Arr, Length, SortOrder, IsStepbystep);

for(int i=0; i<Length-1; i++)

{

T temp=Arr[0];

Arr[0]=Arr[Length-1-i];

Arr[Length-1-i]=temp;

Heapify<T>(Arr, Length-1-i, 0, SortOrder, IsStepbystep, 0);

}

}

#endif //HEAPSORT\_INL\_INCLUDED

* 1. QuickSortHoare.inl

#ifndef QUICKSORTHOARE\_INL\_INCLUDED

#define QUICKSORTHOARE\_INL\_INCLUDED

#include <iostream>

//using namespace std;

/\*template<typename T>

void printArr(T \*Arr, int Length, char\* label, int reclvl) //

{

for(int j=0; j<reclvl;j++) std::cout<<"\t-";

std::cout<<"===Arr ("<< label<<"):==="<<std::endl;

for(int i=0; i<Length;i++)

{

for(int j=0; j<reclvl;j++) std::cout<<"\t-";

std::cout<<"Arr["<<i<<"]="<<Arr[i]<<std::endl;

}

}

\*/

void QuickSortHoare(int \*Arr, int size, bool SortOrder, bool IsStepbystep, int reclvl) {

int i = 0, j = size-1; //номера элементов, движущиеся от концов к центру

int temp, p;

p = Arr[size>>1]; //берем средний элемент

// std::cout<<"size>>1 ="<< (size>>1) <<std::endl;

if(IsStepbystep)

{

std::cout<<"pivot is Arr["<<(size>>1)<<"]=p="<< p <<std::endl;

}

do {

while (Arr[i] < p) i++;

while (Arr[j] > p) j--;

if(IsStepbystep)

{

std::cout<<"Current i:(Arr[i] !< p) ="<< i <<"; Current j:(Arr[j] !> p) ="<< j << std::endl;

getchar();

}

if (i <= j)

{

temp = Arr[i];

Arr[i] = Arr[j];

Arr[j] = temp;

i++;

j--;

if(IsStepbystep)

{

printArr<int>(Arr, size, (char\*)"After swap", reclvl);

getchar();

}

}

} while (i<=j);

reclvl++;

if (j >= 1) QuickSortHoare(Arr, j+1, SortOrder, IsStepbystep, reclvl);

if (size-2 >= i) QuickSortHoare(Arr+i, size-i, SortOrder, IsStepbystep, reclvl);

reclvl--;

}

#endif //QUICKSORTHOARE\_INL\_INCLUDED

/\*

int main()

{

setlocale(LC\_ALL,"Rus");

int size, i;

int \*Arr;

srand(time(NULL));

//cout<<"Введите размер массива"<<endl;

//cin>>size;

size=8;

Arr=new int [size];

for (i=0; i<size; i++)

{

Arr[i]=i;

}

Arr[3]=1;

for (i=0; i<size; i++)

{

//Arr[i]=rand()%100;

cout<<Arr[i]<<" ";

}

cout<<endl;

QuickSortR(Arr, size);

cout<<"Отсортированный массив"<<endl;

for (i=0; i<size; i++)

{

cout<<Arr[i]<<" ";

}

delete[] Arr;

cout<<endl;

}

\*/

* 1. ShellSort.inl

#ifndef SHELLSORT\_INL\_INCLUDED

#define SHELLSORT\_INL\_INCLUDED

void ShellSort(int \*array, int size, bool SortOrder, bool IsStepbystep, int reclvl)

{

for (int s = size / 2; s > 0; s /= 2) { //s - шаг в подсписке

for (int i = s; i < size; ++i) { //каждая итерация - каждый подсписок

for (int j = i - s; j >= 0 && array[j] > array[j + s]; j -= s) { //если перестановок не будет, то в этом подсписке для элементов с меньшими индексами в этом подсписке не нужно делать перестановку

/\*обмен между j и j+s\*/

int temp = array[j];

array[j] = array[j + s];

array[j + s] = temp;

}

}

if (IsStepbystep)

{

printArr<int>(array, size, std::string("After Step s=").append(std::to\_string(s)).c\_str(),0);

}

}

}

#endif //SHELLSORT\_INL\_INCLUDED

* 1. PlotScript.gpi

#! /usr/bin/gnuplot -persist

# Будем этим скриптом строить графики

#Тут неизменяемые настройки

set grid

set datafile separator '\t'

#set terminal png size 1024, 768

#set output "LabPlot.png"

set linestyle 1 lc rgb "#0000FF" lw 2

set linestyle 2 lc rgb "#00FF00" lw 2

set linestyle 3 lc rgb "#FF0000" lw 2

#Тут изменяемые настройки

set title "Lab1" font "Helvetica Bold, 14"

set xlabel "N - size of array"

set ylabel "Time - average executing time, seconds"

#set yrange [-1.1:1.1]

#set xrange[-pi:pi]

#plot sin(x) title "sinux" lc rgb "red", cos(x) title "cosinus" lc rgb "green"

#plot "HeapSortOut.txt" w lp lc 3 pt 7 ps 30 smooth bezier

#plot "HeapSortOut.txt" w lp lc 2 pt 1 ps 2 smooth acsplines

plot "HeapSortOut.txt" using 1:2 notitle smooth csplines with lines linestyle 1, \

1 / 0 title "HeapSort" with lines linestyle 1, \

\

"QuickSortHoareOut.txt" using 1:2 notitle smooth csplines with lines linestyle 2, \

1 / 0 title "QuickSortHoare" with lines linestyle 2, \

\

"ShellSortOut.txt" using 1:2 notitle smooth csplines with lines linestyle 3, \

1 / 0 title "ShellSort" with lines linestyle 3

#Справка

# \ позволяет записать одну строку как несколько

# -persist позволяет держать окно открытым после завершения скрипта

#

#

#