Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**КУРСОВОЙ ПРОЕКТ**

**Дисциплина:** Алгоритмы и структуры данных

**Тема:** Разработка GUI приложения для взвешенного лабиринта

Выполнил

студент гр. 3530903/90003 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ковалевский В.М.

(подпись)

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ахин М.Х.

(подпись)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

Санкт-Петербург   
2020

**СОДЕРЖАНИЕ**

**[ТЕХНИЧЕСКОЕ ЗАДАНИЕ 3](#_Toc1690342041)**

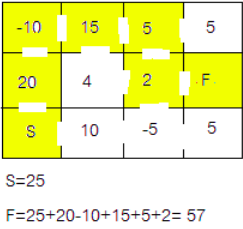
**[МЕТОД РЕШЕНИЯ 4](#_Toc1925383787)**

**[ОПИСАНИЕ КЛАССОВ И МЕТОДОВ. 5](#_Toc243828091)**

**[РАБОТА ПРОГРАММЫ 7](#_Toc471003971)**

# **ТЕХНИЧЕСКОЕ ЗАДАНИЕ**

Лабиринт представляет собой матрицу NxM комнат. Между некоторыми из них есть двери. У человека есть исходная сумма денег. Когда человек входит в комнату, сумма его денег изменяется (может учеличиться или уменьшиться) на величину заданную для этойкомнаты. Найти путь от Start (S) до Finish (F), чтобы собрать максимальное количетво денег, если он существует.



Пример задачи и ее решение

Взаимодействие с пользователем осуществляется через GUI.Есть возможность сохранения/загрузки через специальный формат файла.

Всего в программе будет 2 основных окна. В первом, которое открывается при запуске, есть рабочее пространство в центре для модификации лабиринта и меню сверху, где можно сохранить/загрузить лабиринт, создать новый, изменить размер и собственно, решить задачу, перейдя во второе основное окно. Здесь уже можно посмотреть на результаты работы алгоритма и сохранить его при необходимости.

Фреймворк - TornadoFX (Kotlin)

**Ссылка на репозиторий:**<https://github.com/Slava0135/cashfinder>

# **МЕТОД РЕШЕНИЯ**

Лабиринт можно представить как взвешенный граф: комнаты это взвешенные узлы, они соединены друг с другом, если между ними нет стены.

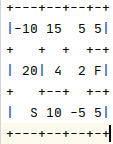
Задача NP-полная: по сути нам нужно найти самый длинный путь, если представлять деньги как некий “вес”. Кроме того существуют узлы с отрицательным весом, что делает невозможным использование некоторых алгоритмов. И более того, в таком графе могут существовать циклы с отрицательной длиной (в нашем случае “можно” ходить по кругу и бесконечно собирать монеты).

Поэтому единственным точным алгоритмом решения будет поиск в глубину (хотя также можно использовать и поиск в ширину, но он требует больших затрат памяти). По сути мы просто перебираем все возможные решения.

У нас есть очередь из вершин. Первый элемент - старт. Рекурсивно для последней вершины в очереди: для каждого ребенка - добавляем его в очередь, если его там нет, вызываем рекурсивную функцию и удаляем из очереди. База - последняя вершина является финишем, обновляем результат, если он получился лучше. Используемая структура данных -LinkedHashSet, т.к. нам, собственно, нужна сама очередь. Кроме того мы проверяем, были ли мы уже в какой-либо комнате, т.е. нам нужно хэш-множество для выполнения операции за константу.

Сложность алгоритма: допустим всего у нас N комнат, в каждой максимум по 4 двери, максимальная длина пути - N. Тогда худшее возможное время работы: 4 \* (первая комната) \* 4 (вторая комната)... Максимум - N комнат, как уже было сказано, тогда получим время 4 ^ N. Как видно, время выполнения сильно зависит от среднего кол-во дверей на каждом узле.

Сам лабиринт можно создать напрямую через графический интерфейс или загрузить из файла вида:



Формат файла лабиринта

**Символы:**

1. Плюс - перекрестие стен. Плюсы задают сетку-разметку.
2. Минус - горизонтальная стена (если на одном уровне с перекрестием).
3. Вертикальная черта - вертикальная стена.
4. Собственно числа или символы старта (S) и финиша (F)

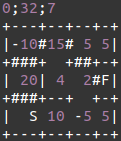
**Есть некоторые требования к виду записи:**

1. Все строки должны быть одинаковой длины
2. Должны быть все внешние стены.
3. Ширина столбца постоянная, т.е. перекрестия образуют сетку.
4. Проходы должны быть закрыты или открыты полностью (т.е. между двумя перекрестиями не могут чередоваться пробелы и черточки)
5. Ровно один старт и финиш.
6. В ячейках не используются иные символы.

Если файл представлен в неправильном формате, загрузочный метод бросит исключение.

Расширение файла лабиринта: **.csh**

Решение также можно сохранить. Формат файла немного другой:



Формат файла решения

В первой строке - начальное кол-во денег, затем идет результат и длина пути.

Пройденые двери обозначаются знаком решетки. Все остальное - аналогично.

Расширение файла решения: **.sol**

# **ОПИСАНИЕ КЛАССОВ И МЕТОДОВ.**

В общем, программа состоит из двух модулей - модели и представления.

В модели основными классами являются граф-лабиринт и его решенный “аналог”.Сначала поговорим оклассе **Graph**.

У него есть статический класс - фабрика **GraphFactory**, где можно создать новый граф с заданным содержимым (т.е. можно указать параметры генерации) или загрузить граф из файла.

У самого графа есть методы генерации стен (по краям, все стены внутри, случайно), метод сохранения в файл, метод изменения размеров (возвращает новый граф), метод связывания узлов по стенам, метод валидации и, конечно, метод преобразования в строку.

Поля графа: ширина, высота, матрица стен, матрица ячеек, старт и финиш, а также объект - пустая ячейка, используемая при создании пустого графа.

Стены представлены своим классом данных **Wall**, который имеет 4 поля для каждой из сторон представленных Boolean: true - стена есть, и false - стены нет.

Ячейки представлены классом данных **Node**, у которого есть поля: координаты, isStart, isFinish, список соседей и значение (кол-во денег).

Теперь о классе “решенного” графа **SolvedGraph**. В нем есть только методы загрузки/сохранения и преобразования в строку. Он также хранит ширину, высоту, стены и ячейки. Кроме того он хранит информацию о решении (длина пути, начальное кол-во денег, счет), которое передается в конструктор при создании.

Ячейки представлены классом **SolvedNode**с тремя полями isOnPath, isEnd, isStart. Стены - **SolvedWall**, где теперь 3 состояния (нет стены, есть стена, на пути), реализованных через enum класс.

Мостиком между этими двумя основными классами является enum класс **Solver**, хранящий функции-решатели, которые принимают граф и начальное кол-во денег, а возвращают “решенный” граф. Я сделал лишь одну такую функцию (см. Метод решения), т.к. пока не смог придумать хороший эвристический алгоритм решения задачи.

Переходим к модулю представления. Для разработки интерфейса использовалась библиотека TornadoFX для Kotlin.

Точка входа в приложение - класс **MainFrame**, собственно первое окно о котором я говорил в начале. Корневой элемент - borderpane, сверху расположено меню (**Menu**), по центру - рабочее пространство (**Workspace**). Кроме того существует глобальная переменная **graph**, которая хранит текущий граф. При присвоении ей нового значения (ссылки на новый граф), **MainFrame** создает новое рабочее пространство (**Workspace**).

Сначала немного о меню. Всего есть 4 вкладки:

1. File - можно создать новый граф (через класс **GraphCreationMenu**), загрузить/сохранить через файл.
2. Edit - изменение размера графа через класс **EditingMenu**.
3. Solve - загрузка решения, решение текущего графа через класс **GraphSolutionMenu**.
4. About - о программе (класс **About**).

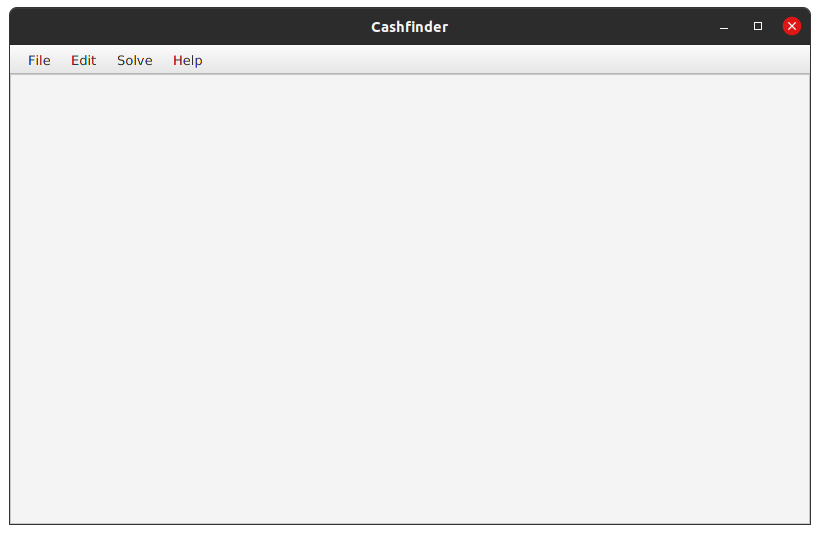
О “подменю” особо сказать нечего - это просто формы для заполнения, которые передают параметры методам из моделии как-то взаимодействуют за графом.

Теперь о рабочем пространстве. Оно представляет собой сетку. Она заполнена вертикальными/горизонтальными стенами, “столбами-пересечениями” и ячейками. На все стены, кроме крайних, можно нажать, убрав их или поставив. В ячейки можно ввести числа от -99 до 99 или букву S (старт) и F (финиш). Старт/финиш может быть только один, если что-то из них отсутствует нельзя сохранить лабиринт или решить его. Перемещать лабиринт можно зажатой кнопкой мыши.

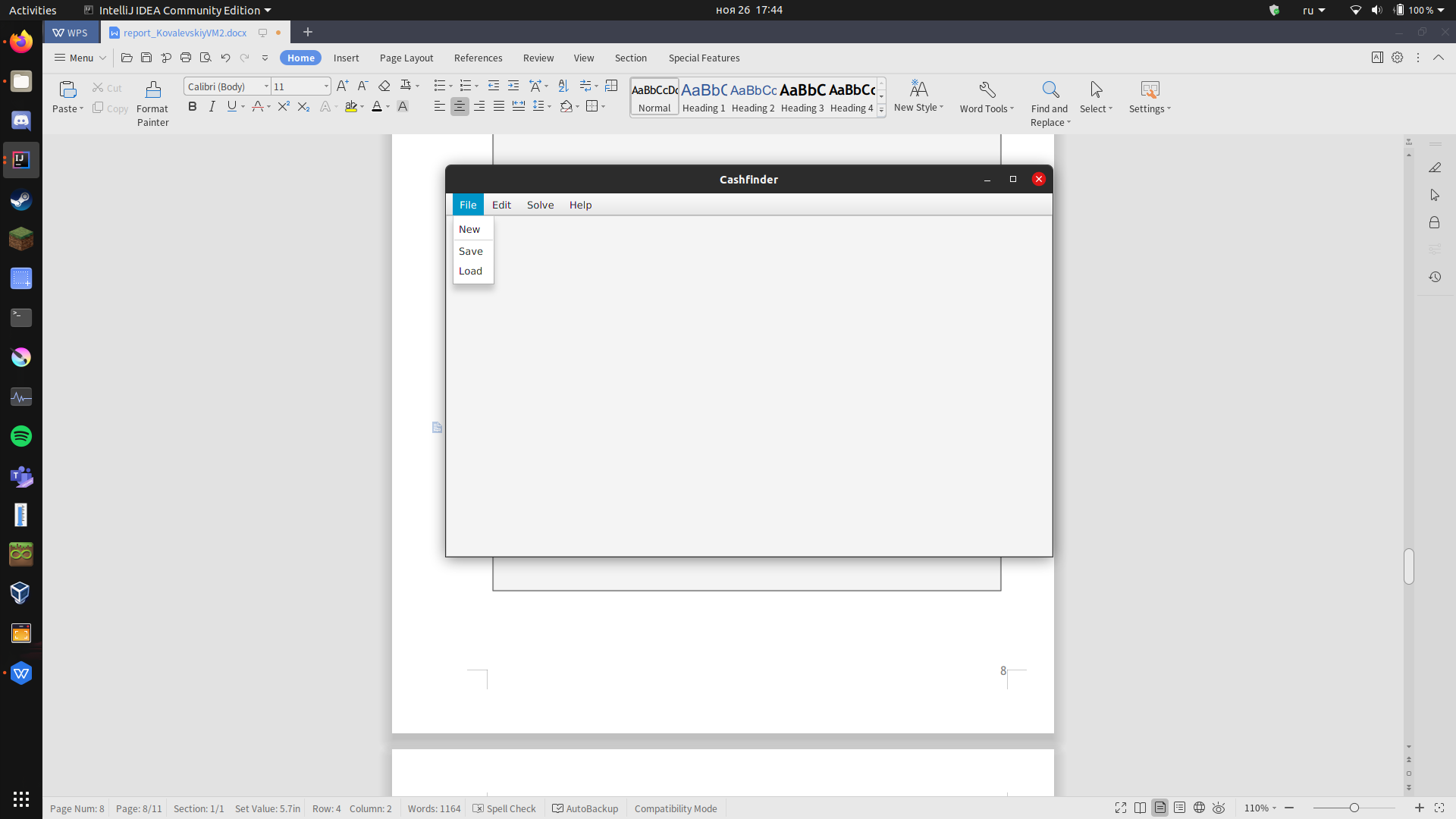
“Решив” граф мы переходим во второе основное окно просмотра решения **SolutionFrame**, которое состоит из панели информации **SolutionTotal** и собственно лабиринта на основе **SolvedGraph**, представленный **SolutionView**. Первое находится слева, второе - посередине. На панели написаны результаты решения и есть кнопка сохранения решения. **SolutionView**аналогичен рабочему пространству, но с ним нельзя взаимодействовать. Путь подсвечен зеленым цветом.

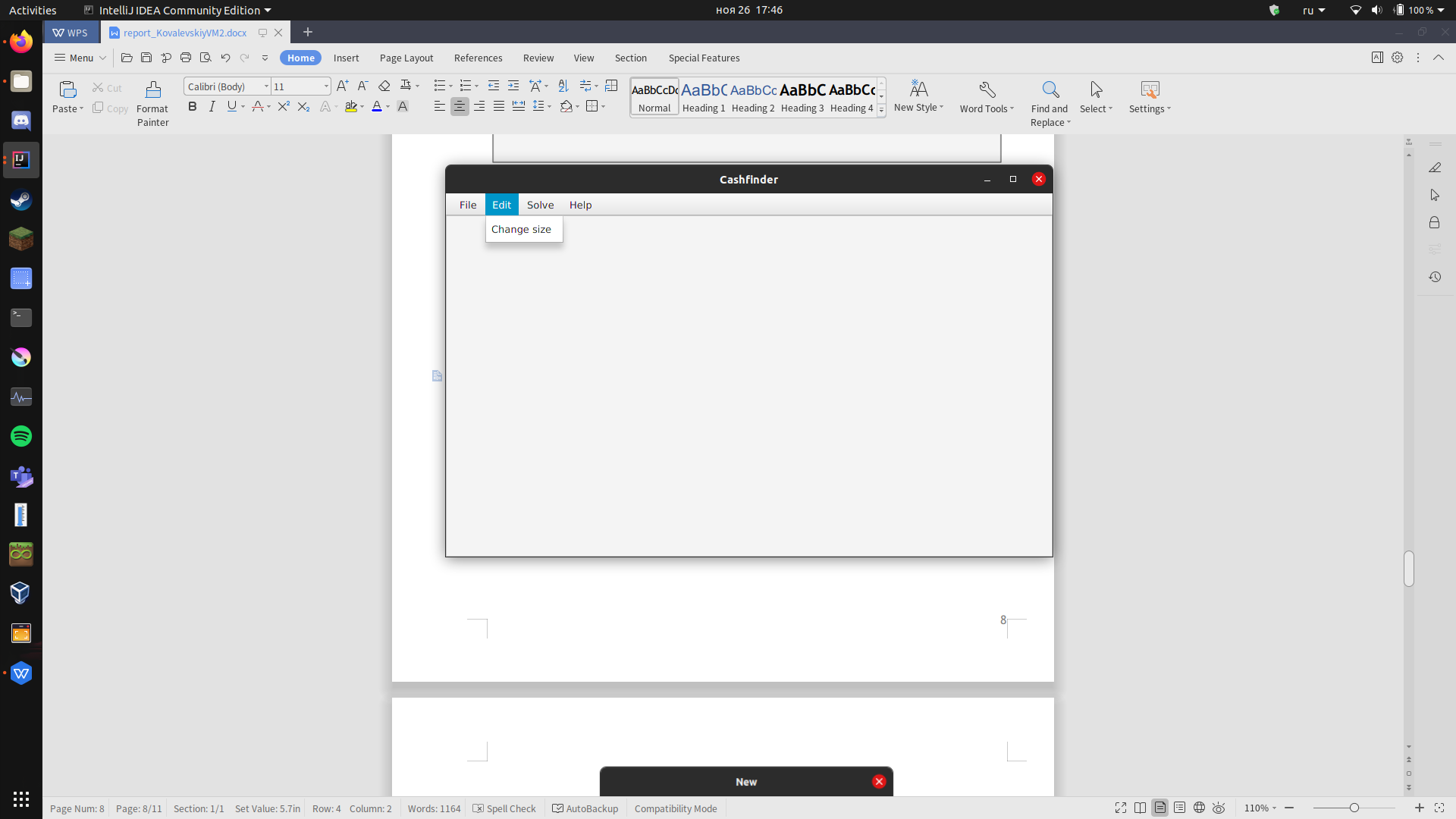
Еще есть объект-конфигуратор**AppConfig**, хранящий разнообразные константы для разных классов.

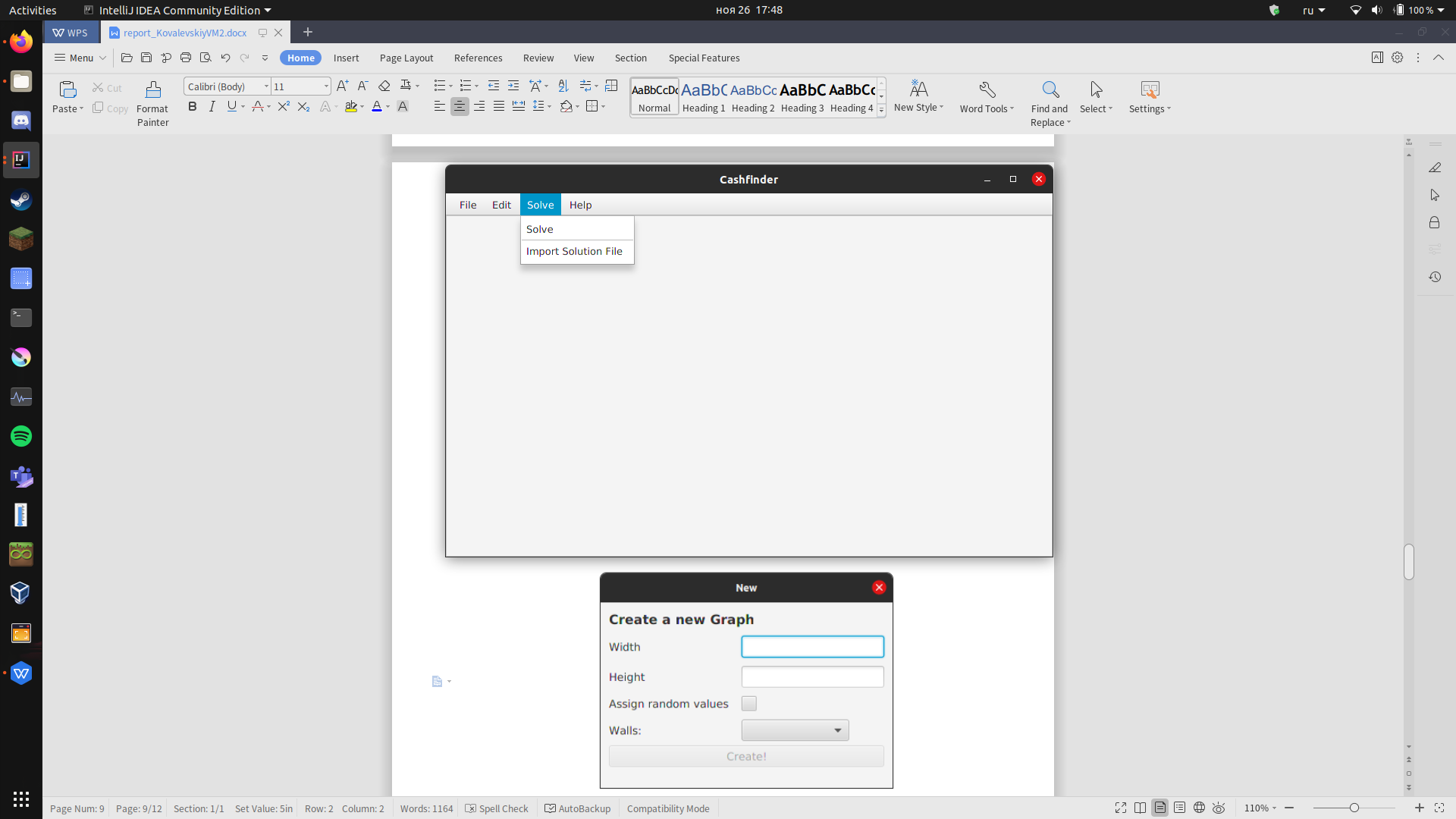
# **РАБОТА ПРОГРАММЫ**

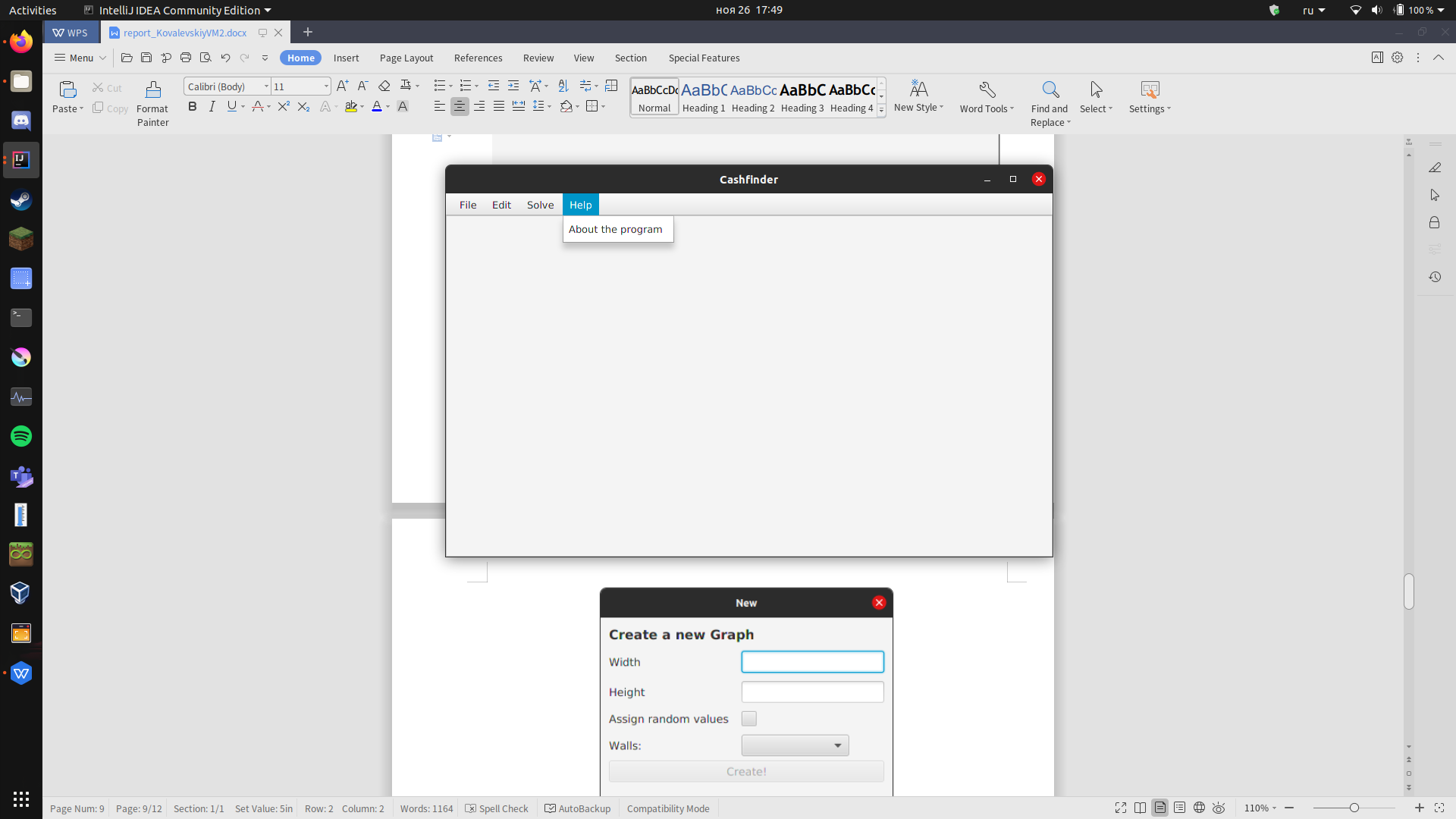


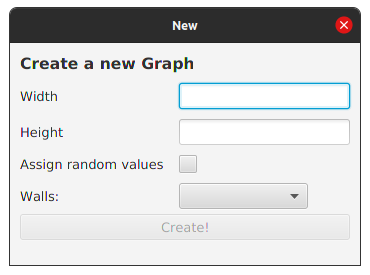
Основное окно работы с лабиринтом



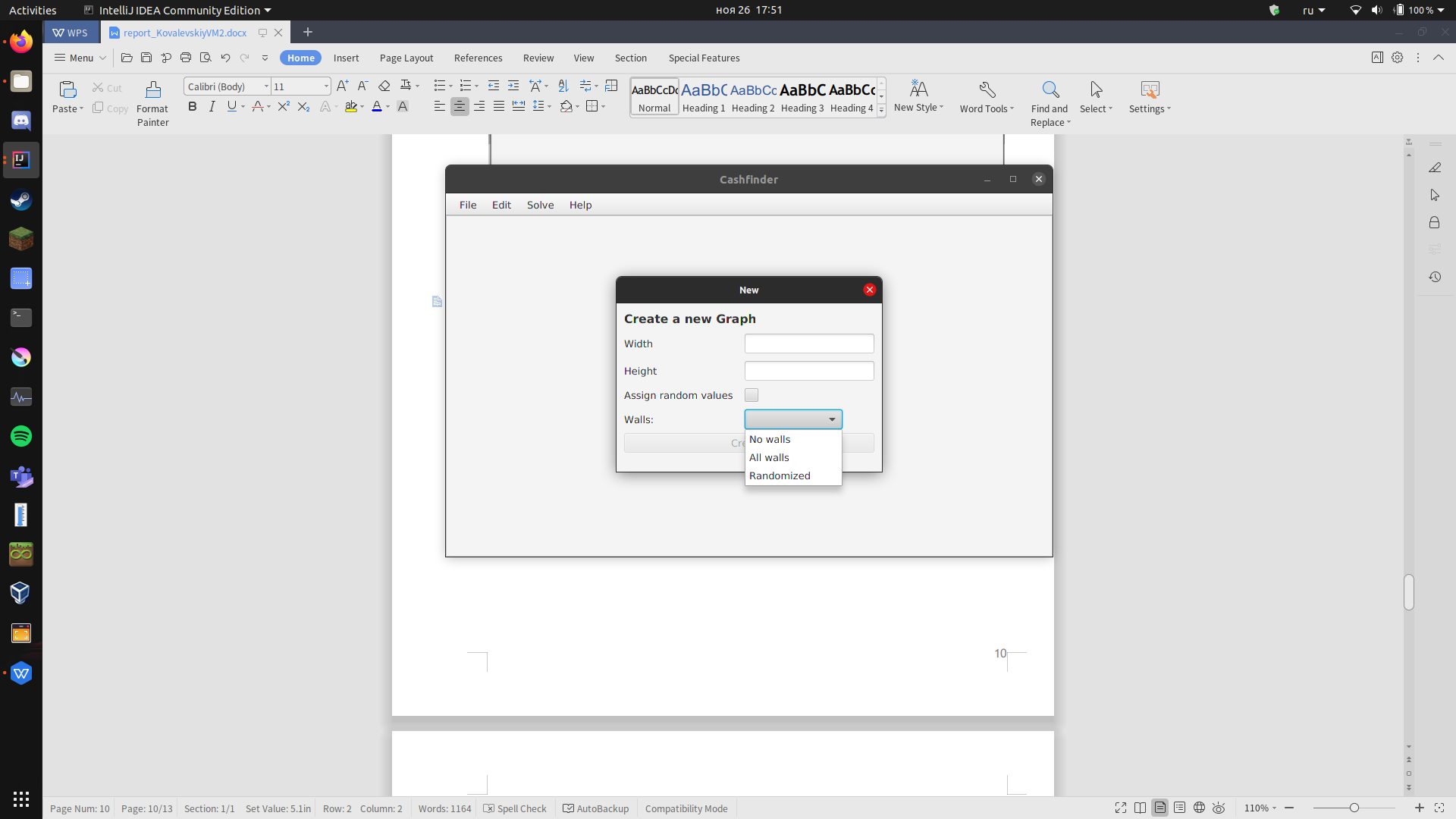


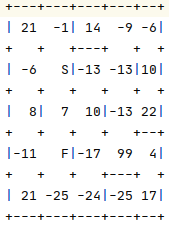




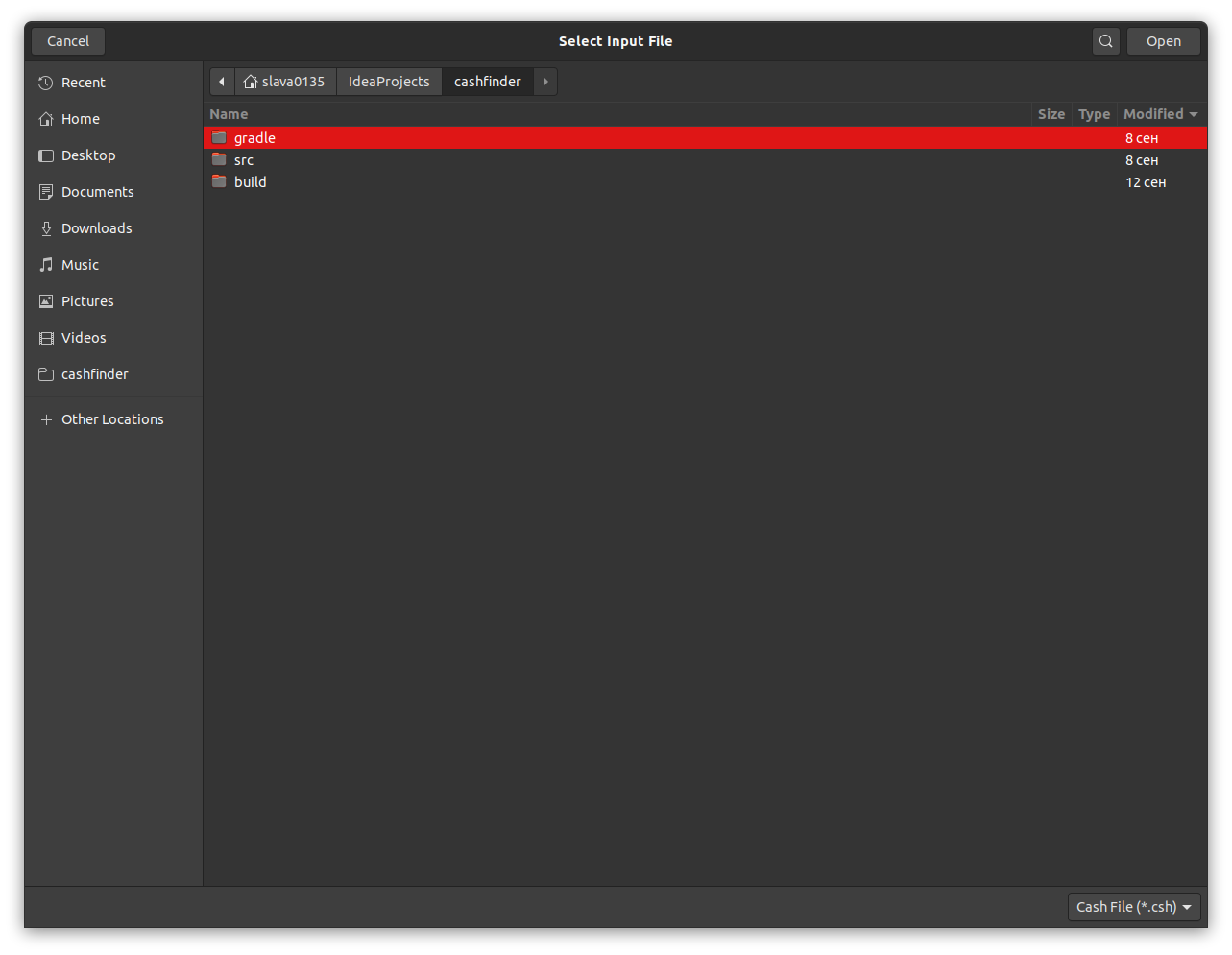


Создание нового графа (File => New)

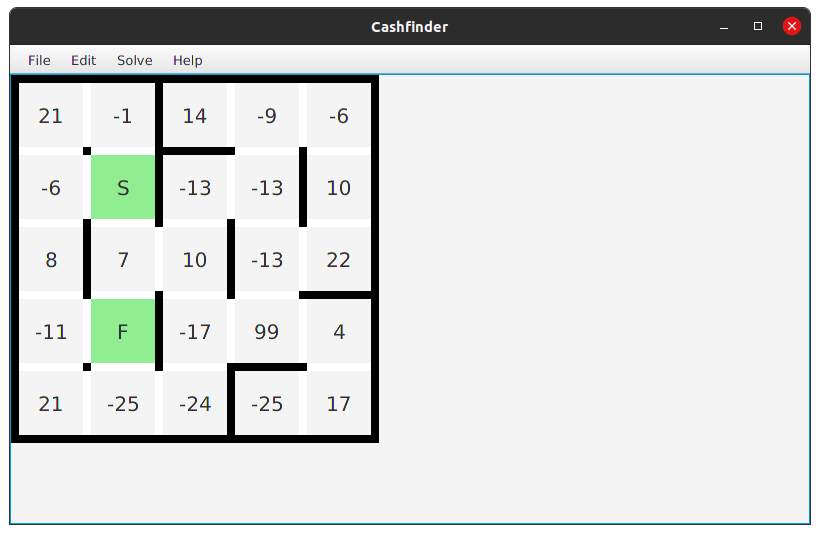




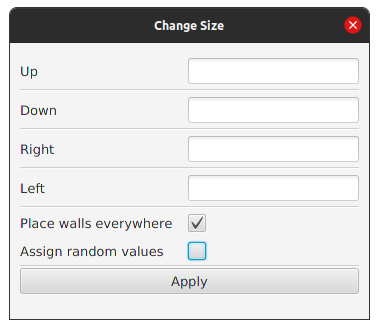
Файл с лабиринтом



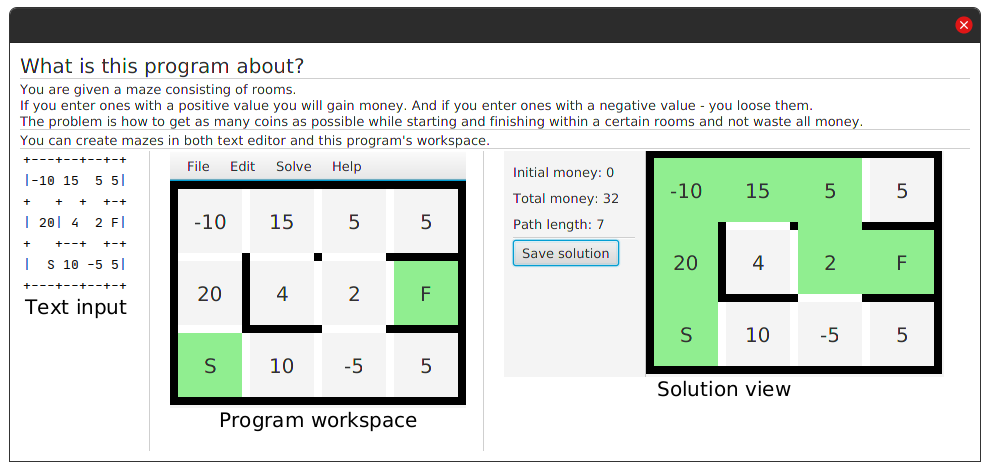
Окно выбора файла



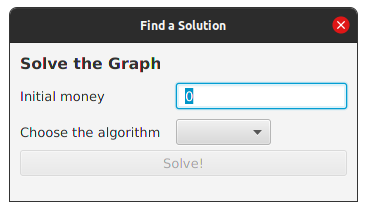
Работа с лабиринтом, загруженным из файла



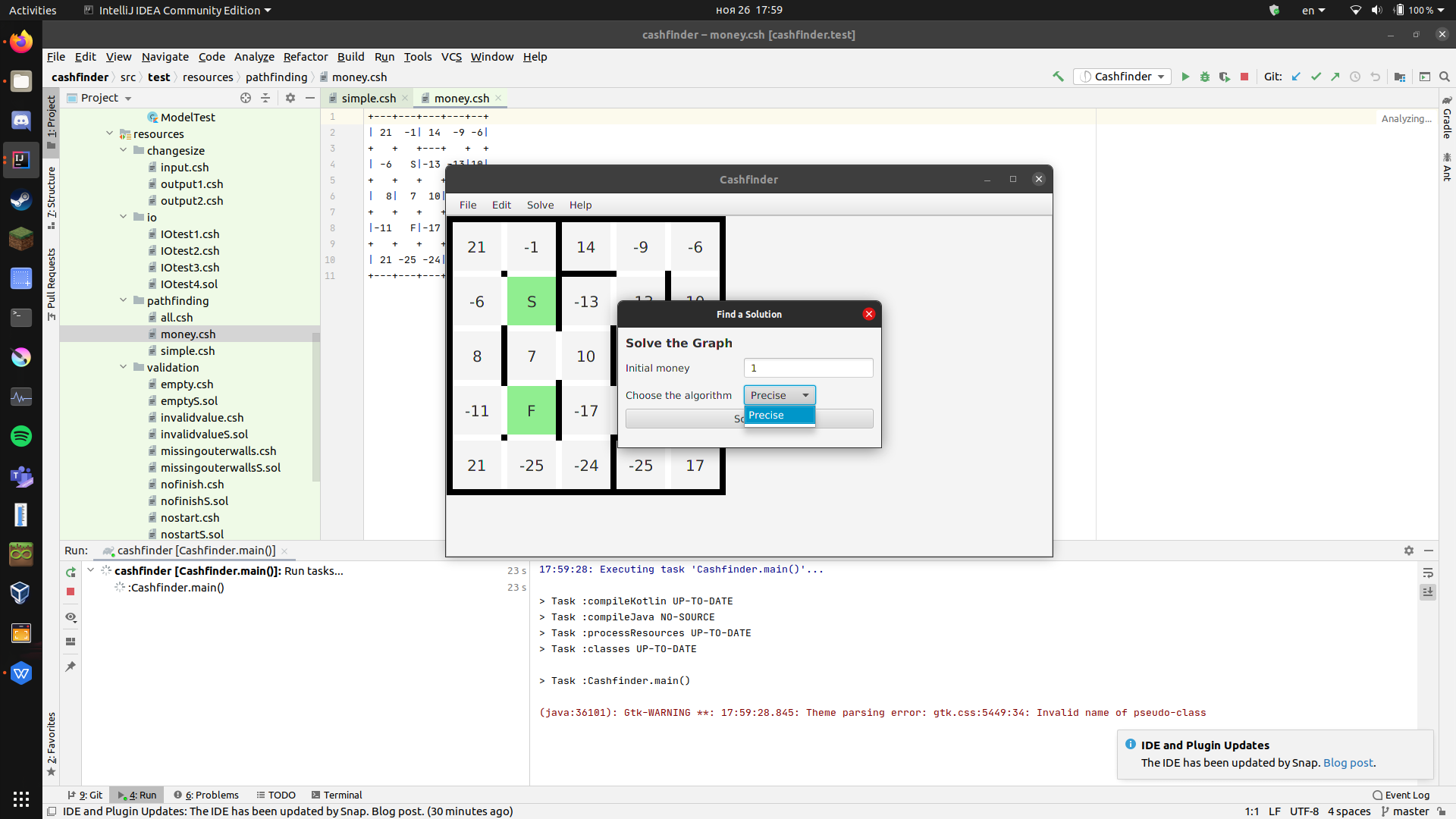
Изменение размера (Edit => Change Size)

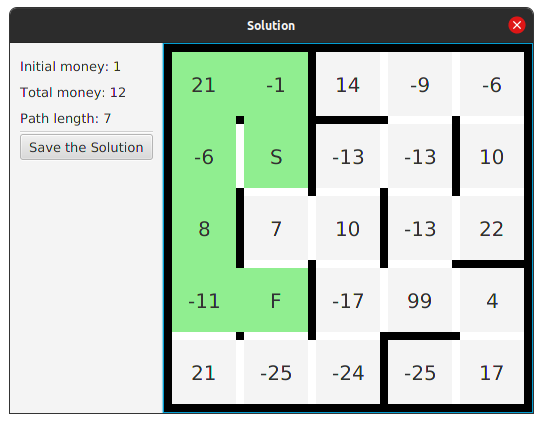


Меню информации (Help => About the Program), слева показан формат файла

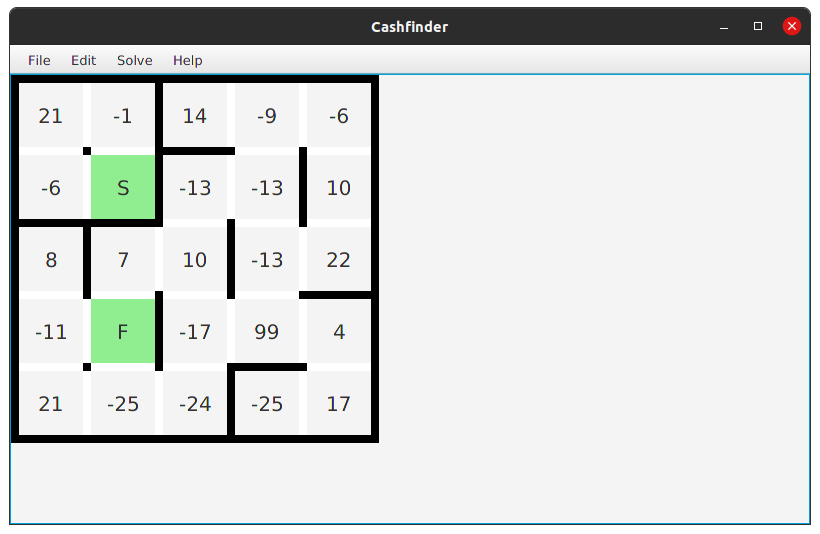


Меню решения (Solve => Solve)

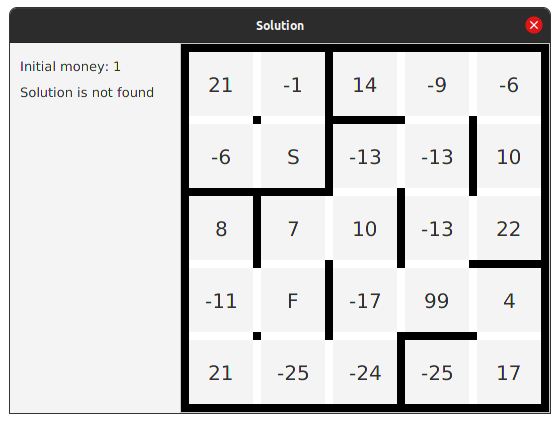




Просмотр решения (после нажатия на кнопку **solve**). Можно сохранить решение нажав на кнопку слева.



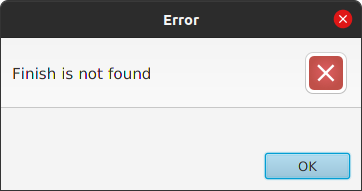
Закроем проход



Решение не найдено



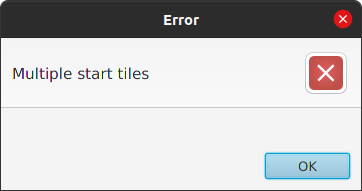
Пример: нет финиша



Получим такую ошибку при попытке загрузить файл. Симметрично для файла без старта.



Пример: два старта



Получим такую ошибку при попытке загрузить файл. Симметрично для файла с двумя финишами.