

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ НИЖЕГОРОДСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМ Н. И. ЛОБАЧЕВСКОГО
Радиофизический факультет

Отчет к лабораторной работе
по Методам программирования
Поиск выхода из лабиринта
(объектно-ориентированная реализация с паттернами)

Студент группы 427

Пронин Владислав Владимирович

Преподаватель

Морозов Н. С.

Н. Новгород, 2026

Содержание

1	Цель работы	3
2	Описание задачи и выбранных паттернов	3
3	Диаграмма классов	4
4	Листинги Классов	4
4.1	Maze Solver	4
4.2	Maze Builder	6
4.3	OBserver	7
5	Результаты	9
6	Анализ эффективности	11

1 Цель работы

Разработать гибкую, расширяемую программу для загрузки лабиринта из файла, поиска пути от старта до выхода с возможностью выбора алгоритма, визуализации процесса и экспериментального сравнения алгоритмов. В ходе работы необходимо применить минимум 3 паттерна проектирования из списка GoF, обосновать их выбор и продемонстрировать преимущества такой архитектуры.

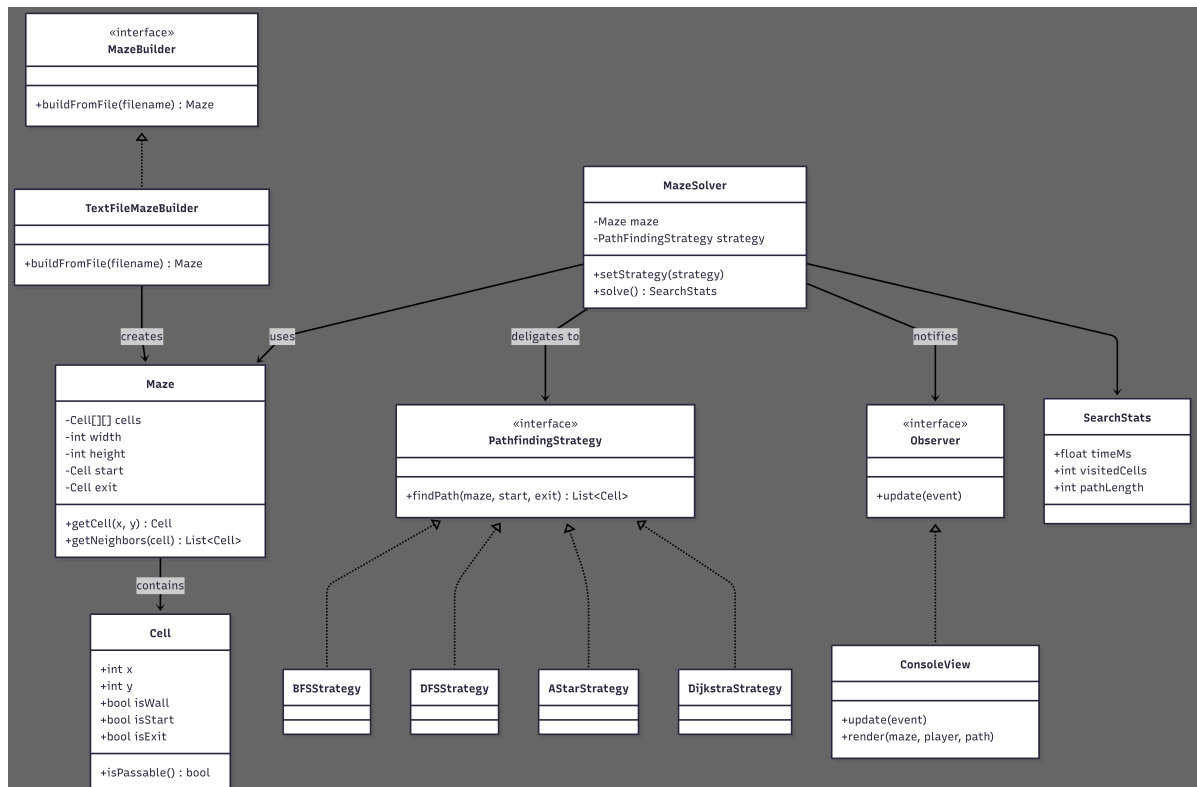
2 Описание задачи и выбранных паттернов

Используемые Паттерны:

- Strategy (Стратегия) — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы. Выбран, так как в данной лабораторной работе используются несколько алгоритмов, выполняющих одно и то же действие — обход графа.
- Builder (строитель) — абстрактный класс/интерфейс, который определяет все этапы, необходимые для производства сложного объекта-продукта. Позволяет отделить построение сложного объекта от его представления, создает сложные объекты, используя простые объекты и поэтапный подход. Выбран для изоляции сложного процесса парсинга текстового файла.
- Observer (Наблюдатель) — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах. Выбран

для отделения логики приложения от вывода на экран (принцип MVC). Класс `ConsoleView` подписывается на события `GameController` и перерисовывает карту только тогда, когда игрок перемещается или путь найден.

3 Диаграмма классов



4 Листиги Классов

4.1 Maze Solver

```

1      import time
2
3      from Maze import Maze
4
5      from strategy import PathFindingStrategy
6
7      class MazeSolver:
  
```

```
6         def __init__(self, maze: Maze, strategy:
PathFindingStrategy):
7             self._maze = maze
8             self._strategy = strategy
9             self._observers = []
10
11         def addObserver(self, observer):
12             """                (, ConsoleView)"""
13             self._observers.append(observer)
14
15         def notify(self, event):
16             """                """
17             for observer in self._observers:
18                 observer.update(event)
19
20         def setStrategy(self, strategy):
21             self._strategy = strategy
22
23         def solve(self):
24
25             if not self._maze or not self._strategy:
26                 raise ValueError("                !")
27
28             start_time = time.perf_counter()
29
30             path, visited_count = self._strategy.findPath(
31                 self._maze, self._maze.start, self._maze.exit)
32
33             end_time = time.perf_counter()
34
35             execution_time_ms = (end_time - start_time) * 1000
36
37             path_length = len(path)
38
```

```
39         from ConsoleView import Event
40         self.notify(Event("path_found", {"maze": self._maze, "
path": path}))
41
42         return SearchStats(execution_time_ms, visited_count,
path_length, path)
43
```

4.2 Maze Builder

```
1
2     from abc import ABC, abstractmethod
3     from Maze import Maze, Cell
4
5     class MazeBuilder(ABC):
6         @abstractmethod
7         def buildFromFile(self, filename):
8             pass
9
10
11     class TextFileMazeBuilder(MazeBuilder):
12         def __init__(self):
13             self._maze = None
14
15         @property
16         def maze(self):
17             return self._maze
18
19         def buildFromFile(self, filename: str):
20
21             with open(filename, mode='r', encoding='utf-8') as
file:
22                 lines = file.read().splitlines()
```

```
23
24         height = len(lines)
25         width = len(lines[0])
26         self._maze = Maze(height, width)
27
28         for y, line in enumerate(lines):
29             for x, char in enumerate(line):
30                 cell = self._maze.getCell(x, y)
31
32                 if char == '#':
33                     cell.isWall = True
34                 elif char == 'S':
35                     cell.isStart = True
36                     self._maze.start = cell
37                 elif char == 'E':
38                     cell.isExit = True
39                     self._maze.exit = cell
40                     self._validate()
41                 return self._maze
42
43     def _validate(self):
44         if self._maze.start is None:
45             raise " "
46         if self._maze.exit is None:
47             raise " "
48
49
50
```

4.3 OBserver

```
1
2
```

```

3         from Observer import Observer, Event
4
5
6         class ConsoleView(Observer):
7             def update(self, event: Event) -> None:
8                 if event.type == "maze_loaded":
9                     print("\n      []      !")
10                    self.render(event.data.get("maze"))
11
12                    elif event.type == "path_found":
13                        print("\n      []      !")
14                        self.render(event.data.get("maze"), path=event.
data.get("path"))
15
16                    elif event.type == "move":
17                        print(
18                            f"\n      []      : ({event.data.get('
player_pos').x}, {event.data.get('player_pos').y})")
19                        self.render(event.data.get("maze"),
20                                    player_position=event.data.get("player_pos"))
21
22                    def render(self, maze, player_position=None, path=None)
-> None:
23                        path_set = set(path) if path else set()
24
25                        for y in range(maze.height):
26                            row_chars = []
27                            for x in range(maze.width):
28                                cell = maze.getCell(x, y)
29
30                                if player_position and cell ==
player_position:
31                                    row_chars.append("P")
32                                elif cell.isStart:

```



```
33         row_chars.append("S")
34     elif cell.isExit:
35         row_chars.append("E")
36     elif cell in path_set:
37         row_chars.append(".")
38     elif cell.isWall:
39         row_chars.append("#")
40     else:
41         row_chars.append(" ")
42     print("".join(row_chars))
43
44
```

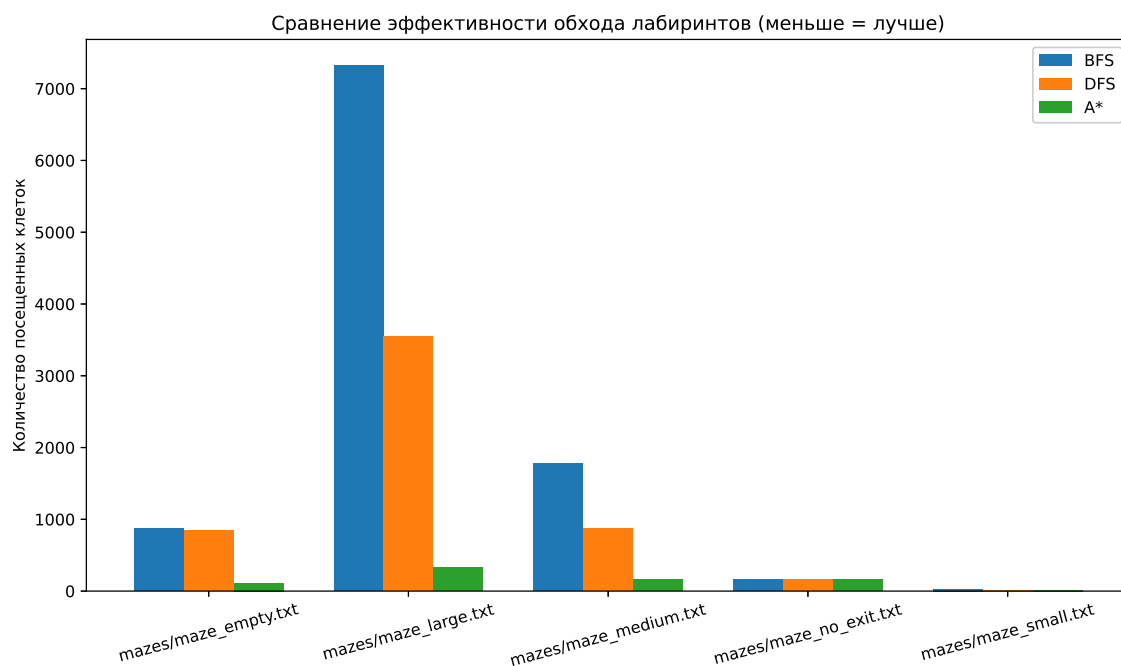
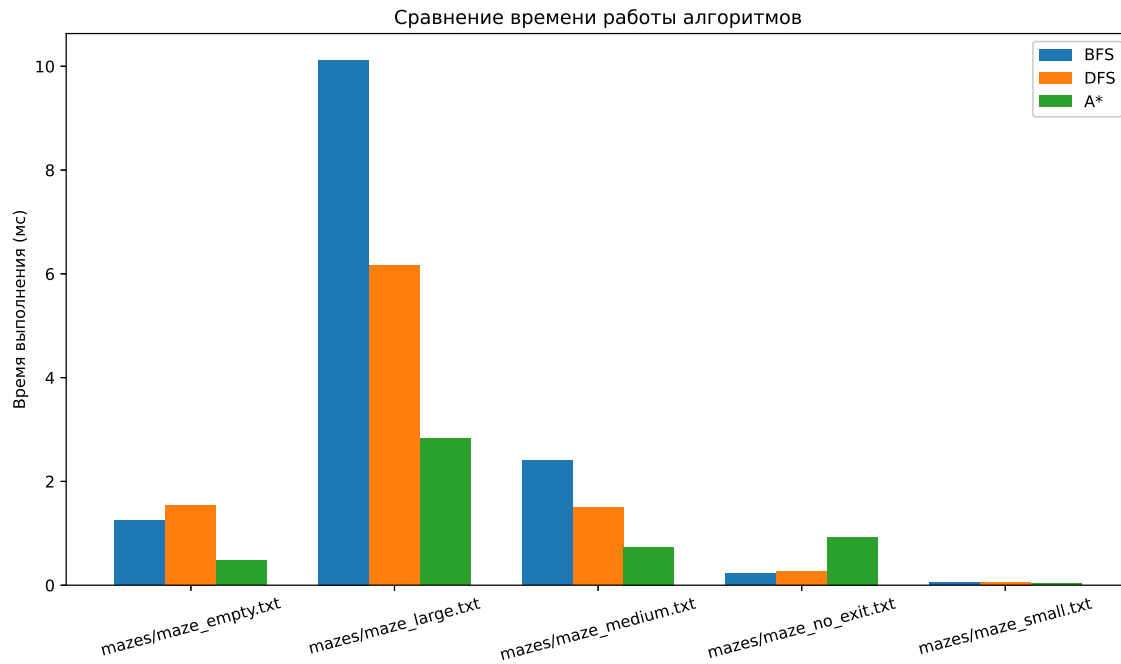
5 Результаты

Таблицы замеров времени и посещенных клеток:

Таблица 1: Результаты экспериментального сравнения алгоритмов поиска пути

Лабиринт	Стратегия	Время (мс)	Посещено клеток	Длина пути
Маленький (10×10)	BFS	0.0516	28	15
	DFS	0.0275	15	15
	A*	0.0360	16	15
	Дейкстра	0.0722	28	15
Пустой (30×30)	BFS	1.1863	870	58
	DFS	1.5568	842	842
	A*	0.4405	113	58
	Дейкстра	2.8607	870	58
Без выхода (15×15)	BFS	0.2230	160	0
	DFS	0.2959	160	0
	A*	0.9378	160	0
	Дейкстра	0.4148	160	0
Средний (50×50)	BFS	3.2247	1779	95
	DFS	1.6985	873	873
	A*	0.7348	158	95
	Дейкстра	6.1264	1779	95
Большой (100×100)	BFS	10.1308	7320	195
	DFS	6.1878	3549	3549
	A*	2.8441	328	195
	Дейкстра	35.2250	7320	195

Графики:



6 Анализ эффективности

Так как в нашем лабиринте вес всех ребер равны 1, то Дейкстра вырождается в Поиск в ширину. Также Дейкстра несколько медленнее из-за дополнительных расчетов на сортировку стоимостей.

Самым лучшим по скорости стал алгоритм A*. Он в среднем 3-4 раза быстрее поиска в ширину, так как на каждом шаге он выбирает самого оптимального соседа для каждого узла, а поиск в ширину проверяет всех соседей.

В разработанной рекурсивной стратегии DFS метрика посещенных клеток совпадает с длиной пути, так как алгоритм фиксирует состояние успешно развернутого стека вызовов в момент достижения целевой точки. Все тупиковые ветви, из которых рекурсия вышла до момента нахождения exit, отсекаются архитектурой возврата флага True, что демонстрирует специфику работы рекурсивного бэктрекинга в Python

7 Выводы по ООП

В ходе выполнения лабораторной работы была спроектирована и реализована объектно-ориентированная система поиска пути в лабиринтах. Применение принципов ООП и паттернов проектирования GoF позволило полностью разделить зоны ответственности классов (принцип Single Responsibility) и обеспечить высокий уровень гибкости и расширяемости приложения.

1. Как паттерны помогли сделать код гибким и расширяемым

- Разделение логики построения и представления (Паттерн Builder): Процесс создания лабиринта инкапсулирован внутри класса TextFileMazeBuilder. Сам лабиринт (Maze) и алгоритмы поиска никак не завязаны на формат хранения данных. Если в будущем потребуется сменить текстовый формат .txt на структуру .json достаточно будет создать нового строителя, реализующего интерфейс MazeBuilder.
- Изоляция и динамическая смена алгоритмов (Паттерн Strategy): Каждый алгоритм обхода графа вынесен в отдельный класс-стратегию с единым

интерфейсом PathfindingStrategy. Класс-оркестратор MazeSolver работает исключительно с абстракцией.

- Использование Observer позволило отделить вычислительную составляющую от графической. Maze Solver никак не учитывает где и как будут отображаться данные, он только отдает сигнал о событиях. Это позволяет если нужно изменить графический интерфейс.