

Отчёт по лабораторной работе:

Поиск выхода из лабиринта (объектно-ориентированная реализация с паттернами)

Выполнил: Волков В. А.

Группа: 4286

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.

Данная работа посвящена изучению архитектурных паттернов проектирования ПО. В рамках задачи реализована система автоматизированного поиска кратчайшего пути в лабиринте, а также инструменты для анализа производительности различных алгоритмов (BFS, DFS, A).

Для реализации цели лабораторной работы я использовал следующие паттерны:

Builder:

Использовался для процесса изолирования создания лабиринта(чтение файла, создание стен, проверка границ) от других структур лабиринта. Таким образом этот паттерн сможет «прочитать» и создать различные лабиринты

Strategy:

Изолирует алгоритмы поиска (BFS, DFS, A*) в отдельные модули. Благодаря чему система повышает скорость и эффективность поиска выхода из лабиринта

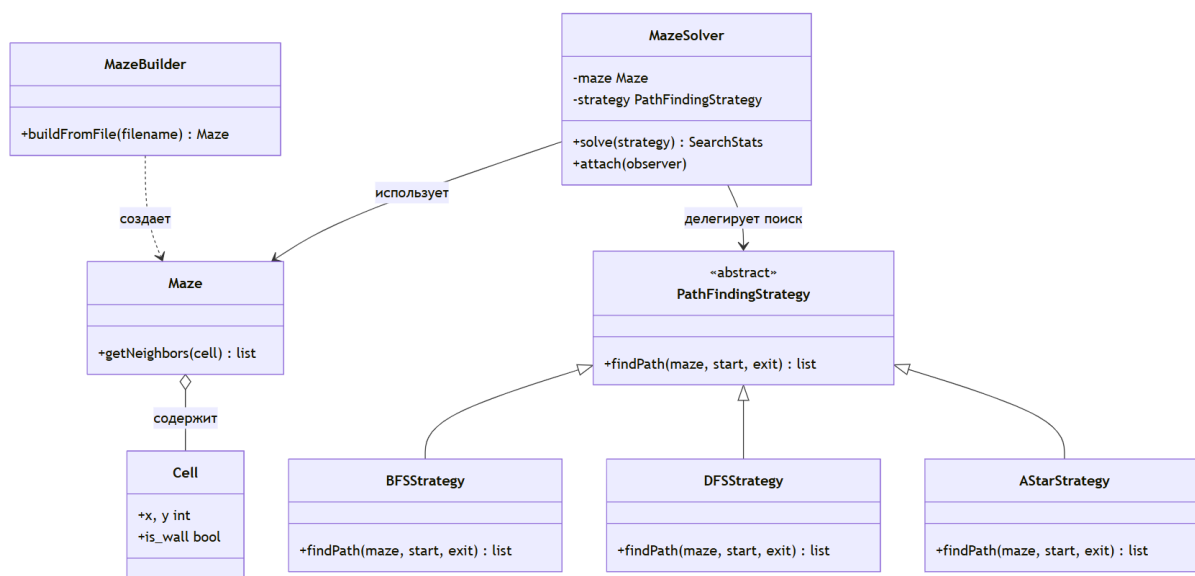
Observer:

Позволяет системе оповещать внешние компоненты (например, консоль или графический интерфейс) о событиях, происходящих внутри логики игры (например, "игрок сдвинулся" или "найдена стена"), без жесткой связи между логикой и представлением.

Command:

Превращает каждое действие игрока (перемещение) в самостоятельный объект. Это позволяет легко реализовать дополнительные функции, такие как запись логов действий или отмена последнего хода (undo), так как у каждой команды есть метод выполнения (execute) и метод отмены (undo).

Реализованная архитектура опирается на четкое разделение ответственности между компонентами системы. Для воплощения выбранных паттернов были спроектированы следующие ключевые классы:



1. Паттерн Builder: Класс MazeBuilder

```
class MazeBuilder:
    def buildFromFile(self, filename):
        path = filename
        if "docs/data/" not in path:
            path = os.path.join("docs", "data", filename)

        with open(path, 'r') as f:
            lines = []
            for line in f:
                stripped = line.strip()
                if stripped:
                    lines.append(stripped)

        h = len(lines)
        w = len(lines[0])
        grid = []
        for y in range(h):
            row = []
            for x in range(w):
                is_wall = False
                if x < len(lines[y]):
                    if lines[y][x] == '#':
                        is_wall = True
                row.append(Cell(x, y, is_wall))
            grid.append(row)

        maze = Maze(w, h, grid)
        for y in range(h):
            for x in range(len(lines[y])):
                if lines[y][x] == 'S':
                    maze.start_cell = maze.grid[y][x]
                if lines[y][x] == 'E':
                    maze.exit_cell = maze.grid[y][x]
        return maze
```

Паттерн Strategy:

```

class SearchStats:
    def __init__(self, time_ms, visited, length):
        self.time_ms = time_ms
        self.visited = visited
        self.length = length

```

```

class PathFindingStrategy(ABC):
    @abstractmethod
    def findPath(self, maze, start, exit):
        pass

```

1. Поиск в ширину (BFS) - оригинальный алгоритм

```

class BFSStrategy(PathFindingStrategy):
    def findPath(self, maze, start, exit):
        queue = deque([start])
        visited = {start: None}
        while len(queue) > 0:
            curr = queue.popleft()
            if curr == exit:
                break
            for n in maze.getNeighbors(curr):
                if n not in visited:
                    visited[n] = curr
                    queue.append(n)

        path = []
        curr = exit
        while curr is not None:
            path.append(curr)
            curr = visited.get(curr)
        return path[::-1], len(visited)

```

2. Поиск в глубину (DFS) - добавлен

```

class DFSStrategy(PathFindingStrategy):
    def findPath(self, maze, start, exit):
        stack = [start]
        visited = {start: None}
        while len(stack) > 0:
            curr = stack.pop()
            if curr == exit:
                break
            for n in maze.getNeighbors(curr):
                if n not in visited:
                    visited[n] = curr
                    stack.append(n)

        path = []
        curr = exit
        while curr is not None:
            path.append(curr)
            curr = visited.get(curr)
        return path[::-1], len(visited)

```

```

class AStarStrategy(PathFindingStrategy):
    def findPath(self, maze, start, exit):
        counter = 0
        queue = [(0, counter, start)]
        came_from = {start: None}
        g_score = {start: 0}

        while len(queue) > 0:
            _, _, curr = heapq.heappop(queue)

            if curr == exit:
                break

            for n in maze.getNeighbors(curr):
                tentative_g_score = g_score[curr] + 1
                if n not in g_score or tentative_g_score < g_score[n]:
                    came_from[n] = curr
                    g_score[n] = tentative_g_score

                f_score = tentative_g_score + abs(n.x - exit.x) + abs(n.y - exit.y)
                counter += 1
                heapq.heappush(queue, (f_score, counter, n))

        path = []
        curr = exit
        while curr is not None:
            path.append(curr)
            curr = came_from.get(curr)
        return path[::-1], len(came_from)

```

Паттерн Observer:

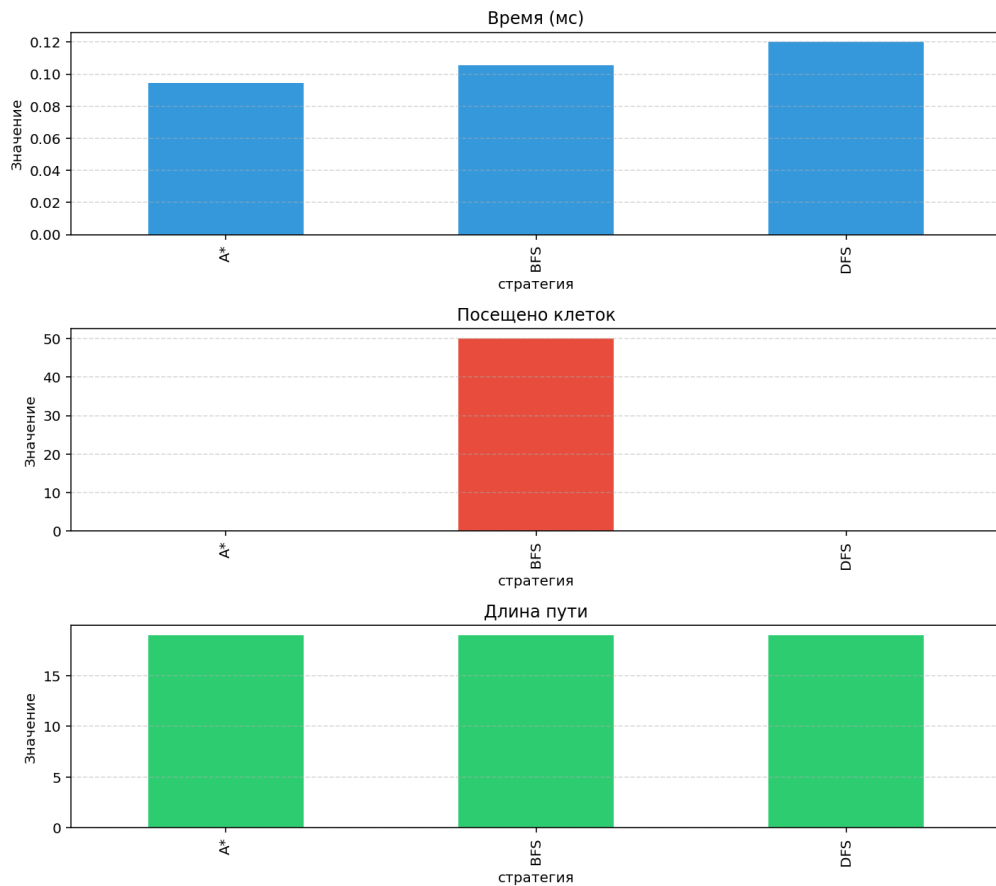
```
class MazeSolver:
    def __init__(self, maze, player=None):
        self.maze = maze
        self.player = player
        self.observers = []
    def attach(self, obs):
        self.observers.append(obs)
    def notify(self, event, data):
        for o in self.observers:
            o.update(event, data)
    def solve(self, strat):
        t0 = time.perf_counter()
        path, visited = strat.findPath(self.maze, self.maze.start_cell, self.maze.exit_cell)
        t1 = time.perf_counter()
        return SearchStats((t1 - t0) * 1000, visited, len(path))
```

```
class ConsoleView:
    def update(self, event, data):
        print(f"[INFO] {event.upper()}: {data}")
```

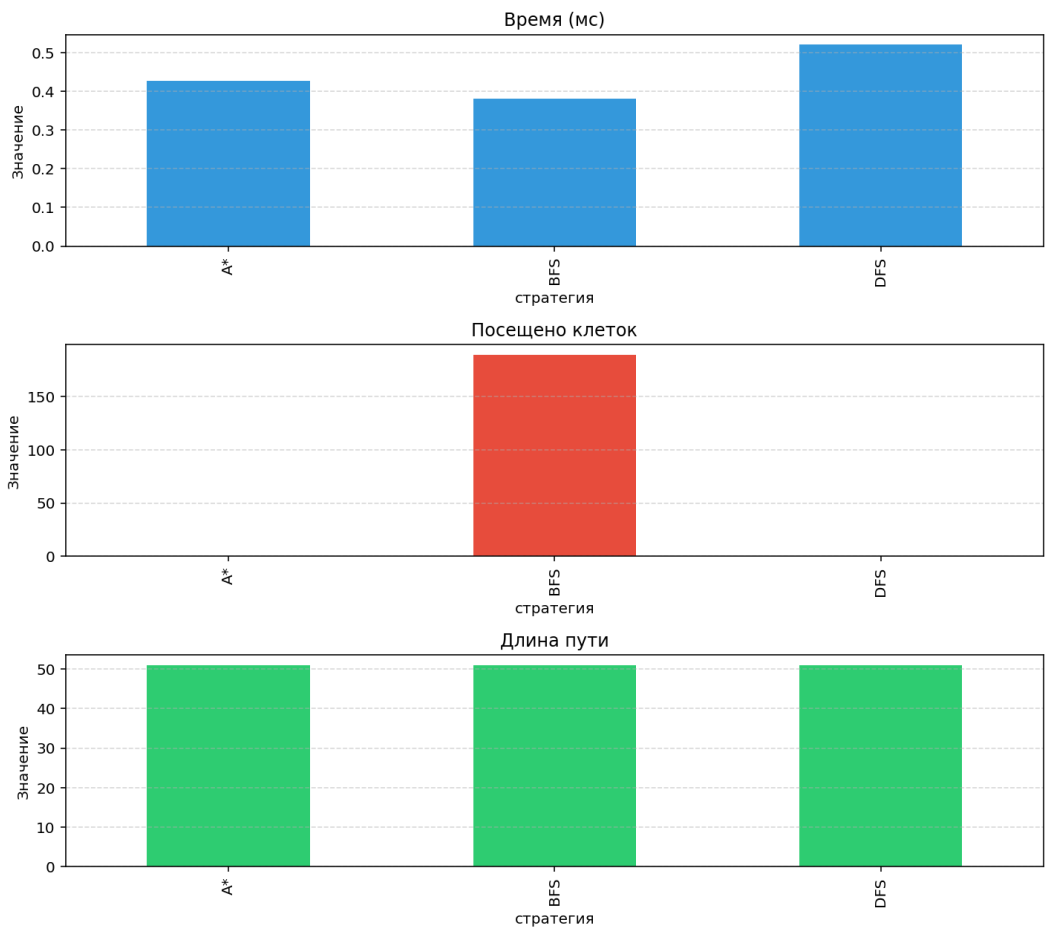
Результаты эксперимента.

Файл	Метод	Время (мс)	Посещено	Путь
maze10-10.txt	BFS	0.08	50.0	19.0
maze10-10.txt	DFS	0.07	50.0	19.0
maze10-10.txt	A*	0.11	48.0	19.0
maze50-50.txt	BFS	0.58	189.0	51.0
maze50-50.txt	DFS	0.57	198.0	51.0
maze50-50.txt	A*	0.24	99.0	51.0
maze100-100.txt	BFS	8.94	4307.0	202.0
maze100-100.txt	DFS	0.61	452.0	250.0
maze100-100.txt	A*	10.66	4032.0	202.0
maze0.txt	BFS	0.43	225.0	29.0
maze0.txt	DFS	0.21	211.0	113.0
maze0.txt	A*	0.57	225.0	29.0
maze777.txt	BFS	0.01	9.0	1.0
maze777.txt	DFS	0.02	9.0	1.0
maze777.txt	A*	0.02	9.0	1.0

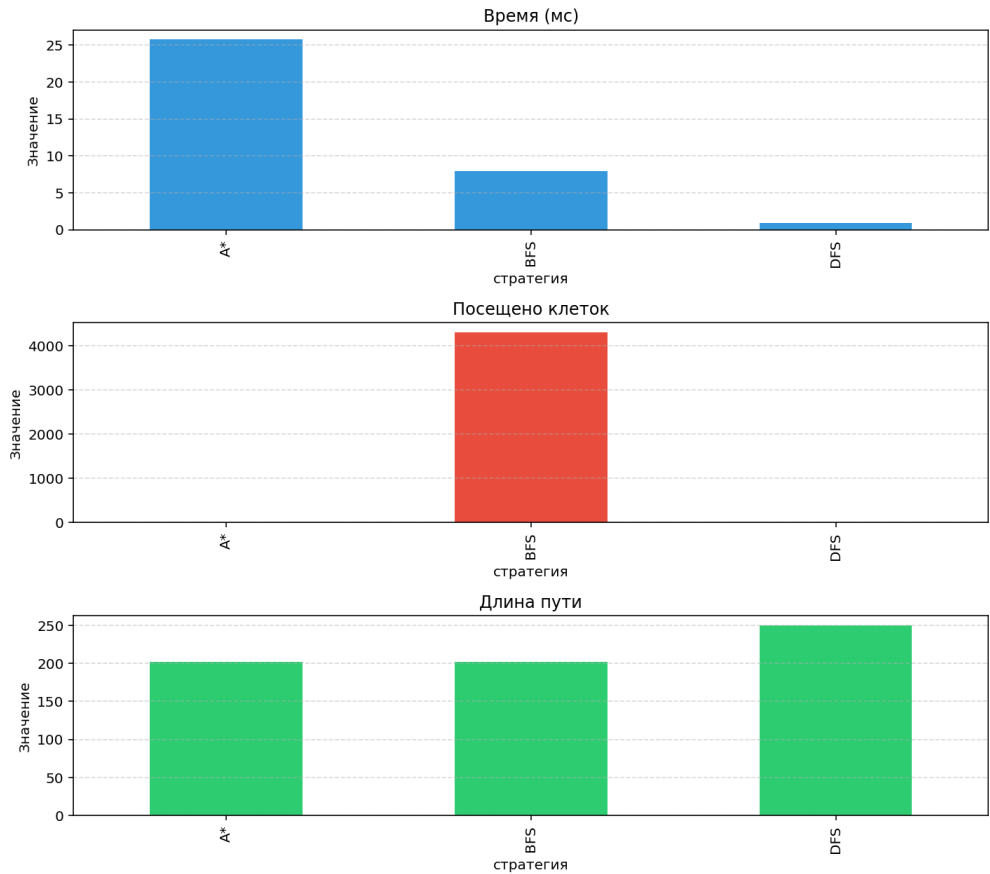
Результаты для: docs/data/maze10-10.txt



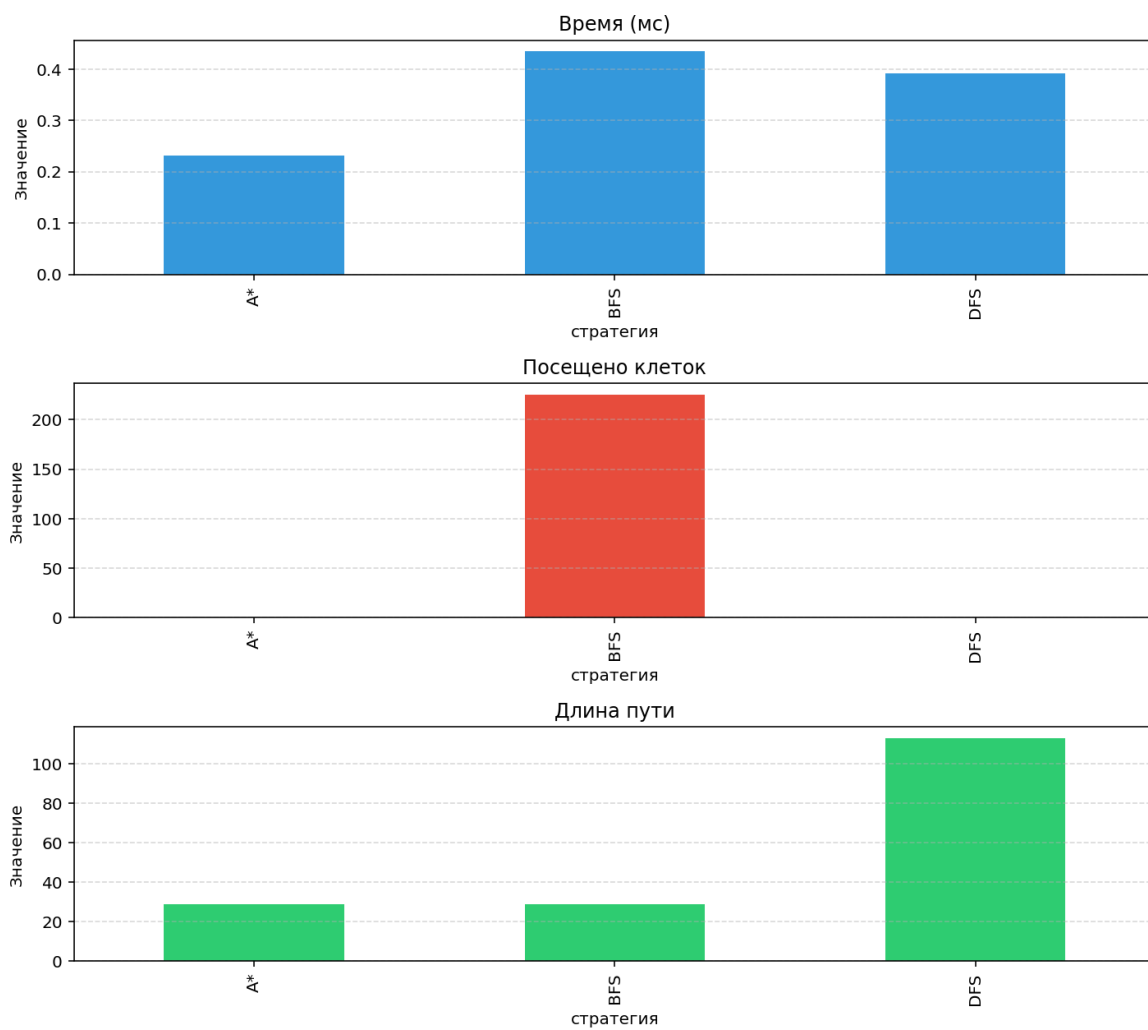
Результаты для: docs/data/maze50-50.txt



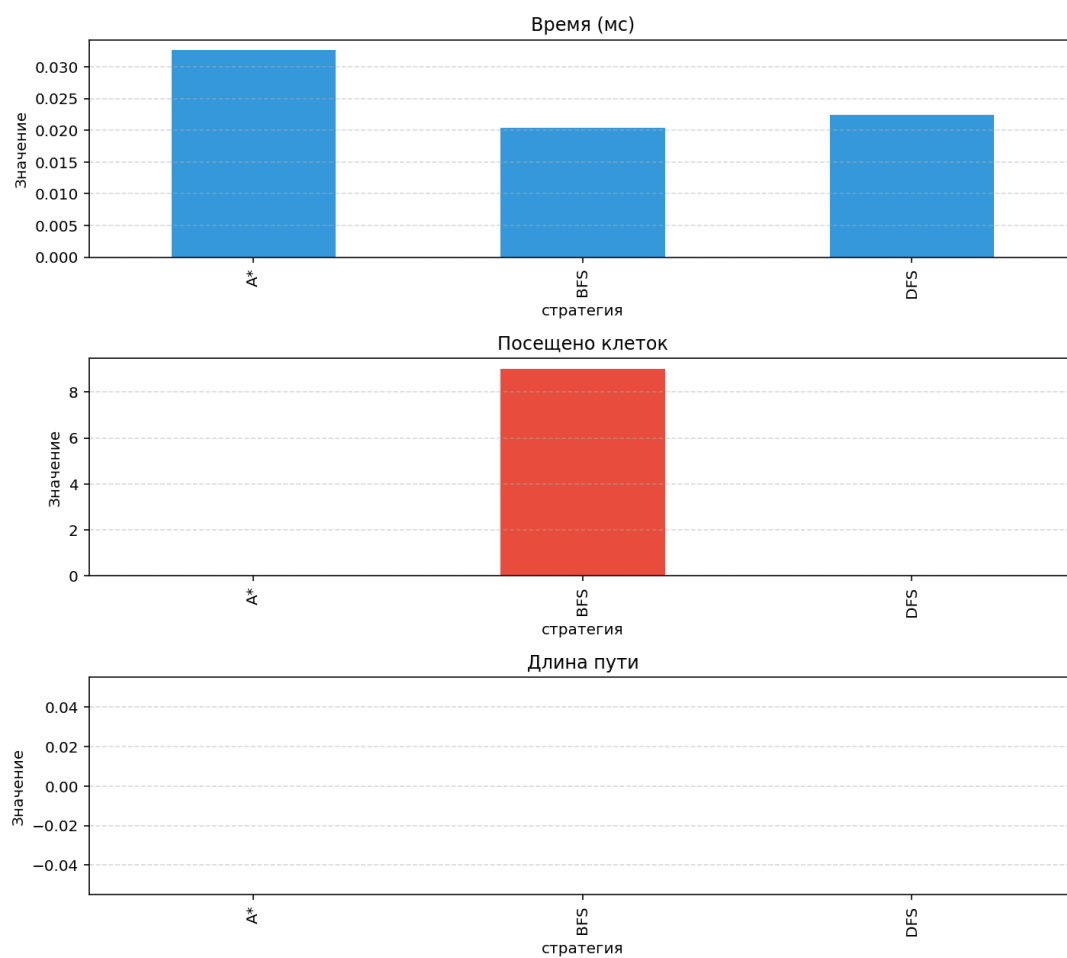
Результаты для: docs/data/maze100-100.txt



Результаты для: docs/data/maze0.txt



Результаты для: docs/data/maze777.txt



Алгоритм A^* проявил себя как наиболее эффективное решение, обеспечивающее нахождение кратчайшего пути при минимальном количестве посещенных узлов за счет использования эвристической оценки расстояния.

Алгоритм BFS (поиск в ширину) гарантирует нахождение пути, идентичного по длине решению A^* , однако требует значительно больших затрат вычислительных ресурсов, совершая перебор практически всех узлов лабиринта, что подтверждается максимальным показателем посещенных клеток.

В свою очередь, алгоритм DFS (поиск в глубину) демонстрирует высокую скорость работы и минимальные затраты на посещение узлов, так как находит первый доступный путь, однако этот путь является значительно длиннее оптимального, что делает DFS наименее предпочтительным в задачах, требующих поиска кратчайшего маршрута.

Анализ эффективности алгоритмов и применимость паттернов проектирования

Проведенное исследование подтверждает, что выбор алгоритма поиска пути напрямую влияет на эффективность навигационной системы. Алгоритм A^* показал себя как наиболее сбалансированное решение, так как легче найти кратчайшего пути при минимальных вычислительных затратах за счет использования эвристик.

BFS (поиск в ширину) делает оптимальный результат, но требует значительных ресурсов, посещая практически все узлы лабиринта.

В то время как **DFS** (поиск в глубину) работает быстро и экономит память, но находит сложные пути, что делает его менее пригодным для задач поиска кратчайшего маршрута.

Для обеспечения гибкости и масштабируемости архитектуры были применены паттерны проектирования.

Паттерн **Strategy** инкапсулировал алгоритмы поиска в независимые классы с общим интерфейсом, что позволило переключаться между ними без изменения основного кода и легко добавлять новые стратегии в будущем.

Паттерн **Observer** обеспечил «отвязку» логики сбора статистики от самого процесса поиска: алгоритмы лишь сообщают о событиях, а «наблюдатель» фиксирует метрики, не перегружая код алгоритма лишними задачами.

Вывод: Применение этих подходов позволило избежать множества однообразных циклов и предотвратило дублирование кода для замеров производительности

Паттерны делают программу защищеннее от ошибок. Благодаря ООП и паттернам, каждый модуль системы отвечает только за свою задачу, что делает программу гибкой, удобной для тестирования и легкой в расширении.