Министерство науки и образования РФ

Федеральное государственное бюджетное учреждение

высшего образования

**«Тверской государственный технический университет»**

(ТвГТУ)

Кафедра программного обеспечения

**Отчет по лабораторной работе №6**

По дисциплине: «Компьютерные системы моделирования»

Тема: «Исследование неинформированных методов поиска решений задач в пространстве состояний»

|  |
| --- |
| Выполнил:  студент группы  Б.ПИН.РИС - 22.06  Иванов А.М. |
| Проверила:  старший преподаватель  кафедры ПО  Корнеева Е.И. |

Тверь 2025

**Оглавление**

[**Цель и постановка задачи для варианта.** 3](#_Toc194268707)

[**Теоретические основы** 5](#_Toc194268708)

[**Краткий обзор методов слепого поиска решений задач в пространстве состояний** 5](#_Toc194268709)

[**Описание представления задачи в AI Pacman (описание состояний, операторов, начального и конечного состояний, критериев достижения цели)** 8](#_Toc194268710)

[**Представление пространства состояний в виде графа** 10](#_Toc194268711)

[**Практическая часть** 12](#_Toc194268712)

[**Тексты реализованных функций с комментариями** 12](#_Toc194268713)

[**Файл search.py** 12](#_Toc194268714)

[**Файл searchAgents.py** 16](#_Toc194268715)

[**Полученные результаты для разных лабиринтов и их анализ** 22](#_Toc194268716)

[**Задание 1** 22](#_Toc194268717)

[**Задание 2** 24](#_Toc194268718)

[**Задание 3** 26](#_Toc194268719)

[**Результаты автооценивания** 28](#_Toc194268720)

[**Заключение** 30](#_Toc194268721)

# **Цель и постановка задачи для варианта.**

**Цель работы**:

Исследование неинформированных методов поиска решений задач в пространстве состояний, приобретение навыков программирования интеллектуальных агентов, планирующих действия на основе методов слепого поиска решений задач.

**Постановка задачи:**

**Задание 1.**

DFS: Реализовать поиск в глубину в depthFirstSearch (файл search.py). Проверить на лабиринтах:

python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

python pacman.py -l bigMaze -z .5 -p SearchAgent

Вопросы:

Соответствует ли порядок исследования ожиданиям?

Проходит ли Pacman все посещенные клетки?

Является ли найденный путь оптимальным (для mediumMaze длина должна быть 130)?

**Задание 2.**

BFS: Реализовать поиск в ширину в breadthFirstSearch.

Проверить

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

Вопрос: Находит ли BFS кратчайший путь?

**Задание 3**

UCS: Реализовать алгоритм равных цен в uniformCostSearch. Проверить на лабиринтах с разными стоимостями:

python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

**Тестирование**

После каждой задачи запускать автооцениватель:

python autograder.py -q q1

python autograder.py -q q2

python autograder.py -q q3

# **Теоретические основы**

## **Краткий обзор методов слепого поиска решений задач в пространстве состояний**

Слепой (неинформированный) поиск применяется, когда у агента нет дополнительной информации о расстоянии до цели или стоимости переходов между состояниями. В этом разделе рассмотрены основные алгоритмы, их свойства и особенности реализации в среде *AI Pacman*.

**Поиск в ширину (BFS — Breadth-First Search)**

**Принцип работы:**

* Исследует все состояния уровня глубины d, прежде чем перейти к уровню d+1.
* Использует **очередь (FIFO)** для хранения вершин.

**Свойства:**

* **Полнота**: Гарантированно находит решение, если оно существует.
* **Оптимальность**: Находит кратчайший путь (по числу действий).
* **Сложность**:
  + Время: O(b^d) (экспоненциальная).
  + Память: O(b^d) (хранит все раскрытые вершины).

**Поиск в глубину (DFS — Depth-First Search)**

**Принцип работы:**

* Идет «вглубь» по одному пути, пока не достигнет тупика, затем возвращается (backtracking).
* Использует **стек (LIFO)**.

**Свойства:**

* **Полнота**: Не гарантирует нахождение решения на бесконечных пространствах или с циклами.
* **Оптимальность**: Не оптимален (может найти длинный путь).
* **Сложность**:
  + Время: O(b^m), где m — максимальная глубина.
  + Память: O(b\*m) (линейная, хранит только текущий путь).

**Проблемы в Pacman:**

* Может «застрять» в бесконечном цикле.
* Решение: ограничение глубины (depth-limited DFS).

**Поиск с итеративным углублением (IDS — Iterative Deepening Search)**

**Принцип работы:**

* Комбинация BFS и DFS. Запускает DFS с постепенным увеличением глубины (d = 0, 1, 2, ...).

**Свойства:**

* **Полнота**: Да.
* **Оптимальность**: Да (находит кратчайший путь).
* **Сложность**:
  + Время: O(b^d).
  + Память: O(b\*d) (как DFS).

**Преимущества:**

* Экономия памяти по сравнению с BFS.
* Избегает недостатков DFS (бесконечные ветви).

**Поиск с равными ценами (UCS — Uniform-Cost Search)**

**Принцип работы:**

* Обобщение BFS для задач с различной стоимостью переходов.
* Выбирает вершину с **наименьшей суммарной стоимостью пути** (приоритетная очередь).

**Свойства:**

* **Полнота**: Да.
* **Оптимальность**: Да (находит путь с минимальной стоимостью).
* **Сложность**: O(b^(1 + C/ε)), где C — стоимость оптимального решения, ε — минимальная стоимость шага.

**Сравнение методов**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Алгоритм** | **Полнота** | **Оптимальность** | **Время** | **Память** | **Применимость в Pacman** |
| BFS | Да | Да (по шагам) | O(b^d) | O(b^d) | Поиск кратчайшего пути. |
| DFS | Нет\* | Нет | O(b^m) | O(b\*m) | Не рекомендуется. |
| IDS | Да | Да | O(b^d) | O(b\*d) | Компромисс между BFS и DFS. |
| UCS | Да | Да (по cost) | O(b^(1+C/ε)) | O(b^d) | Поиск пути с минимальной стоимостью. |

\*DFS не полон на бесконечных пространствах или при наличии циклов.

## **Описание представления задачи в AI Pacman (описание состояний, операторов, начального и конечного состояний, критериев достижения цели)**

В среде *AI Pacman* задача поиска формализуется как **проблема нахождения пути в пространстве состояний**, где агент (Pacman) должен достичь цели, минимизируя затраты. Рассмотрим ключевые компоненты этой формулировки.

**Состояния**

Состояние описывает текущую конфигурацию мира:

* **Для нахождения пути**:
  + (x, y) — координаты Pacman.
  + Пример: (3, 5) — Pacman находится в клетке с координатами 3 по горизонтали и 5 по вертикали.
* **Для поедания всех гранул**:
  + ( (x, y), food\_grid )
    - (x, y) — позиция Pacman.
    - food\_grid — матрица N x M, где True/False обозначает наличие гранулы.

**Операторы (действия)**

Действия переводят Pacman из одного состояния в другое:

* **Допустимые действия**:
  + North (Вверх), South (Вниз), East (Вправо), West (Влево).
* **Функция-преемник**:
  + Принимает состояние и действие, возвращает новое состояние и стоимость перехода.

**Начальное и целевое состояния**

* **Начальное состояние**:
  + Задается методом problem.getStartState().
* **Целевое состояние**:
  + Проверяется методом problem.isGoalState(state).
  + **Критерии достижения цели**:
    - *Для нахождения пути*: state == (x\_goal, y\_goal).
    - *Для поедания гранул*: all(not cell for row in food\_grid for cell in row).

**Критерии оптимальности**

Решение считается оптимальным, если:

1. **Полнота**: Алгоритм находит решение, если оно существует.
   * *Пример*: BFS всегда находит путь, если он есть.
2. **Оптимальность по стоимости**:
   * Минимальное число шагов (для BFS).
   * Минимальная суммарная стоимость (для UCS).
3. **Эффективность**:
   * Минимизация времени (раскрытых состояний) и памяти.

**Особенности реализации в AI Pacman**

1. **Границы мира**:
   * Лабиринт ограничен стенами (walls[x][y] = True).
   * Действия, ведущие в стену, исключаются из getSuccessors().
2. **Стоимость действий**:
   * По умолчанию: 1 за шаг.
   * В UCS может учитывать:
     + Штраф за приближение к привидениям.
     + Бонус за поедание гранул.
3. **Визуализация**:
   * Состояния отображаются на карте через graphicsDisplay.py.

## **Представление пространства состояний в виде графа**

**Формальное определение графа состояний**

Пространство состояний в Pacman может быть представлено в виде **ориентированного графа**:

Copy

G = (V, E), где:

- V — множество вершин (состояния),

- E — множество рёбер (переходы между состояниями).

**Пример для задачи нахождения пути:**

* **Вершины**: Все возможные координаты (x, y).
* **Рёбра**: Допустимые перемещения между соседними клетками (например, (1,1) → (1,2) при действии East).

**Свойства графа в Pacman**

1. **Конечность**:  
   Граф конечен, если лабиринт ограничен (нет бесконечных коридоров).

*Пример*: Лабиринт tinyMaze имеет 9 клеток → 9 вершин.

1. **Цикличность**:  
   Возможны циклы, если Pacman может перемещаться между одними и теми же состояниями.

*Пример*: Движение по квадрату (1,1) → (1,2) → (2,2) → (2,1) → (1,1).

1. **Взвешенность**:  
   Рёбра могут иметь стоимость (например, 1 для обычного шага, 2 для движения через зону привидений).

**Явное vs неявное представление графа**

В AI Pacman используется **неявное представление**:

* **Явное**: Хранение всех вершин и рёбер в памяти (неэффективно для больших лабиринтов).
* **Неявное**:
  + Вершины генерируются "на лету" через getSuccessors().
  + Рёбра определяются допустимыми действиями (North, South и т.д.).

**Особенности графа для разных задач**

|  |  |  |  |
| --- | --- | --- | --- |
| **Задача** | **Вершины** | **Рёбра** | **Целевые вершины** |
| Поиск пути | (x, y) | Перемещения между клетками | Конечная точка (x\_goal, y\_goal) |
| Поедание всех гранул | ( (x, y), food\_grid ) | Действия, изменяющие food\_grid | Все food\_grid[i][j] = False |
| Избегание привидений | ( (x, y), ghost\_positions) | Действия, минимизирующие риск | Безопасные состояния |

# **Практическая часть**

## **Тексты реализованных функций с комментариями**

### **Файл search.py**

**SearchProblem**

from util import Stack, Queue, PriorityQueue

class SearchProblem:

"""

Абстрактный класс для поисковых задач.

Наследуется и реализуется для конкретных задач (нахождение пути, поедание гранул и т.д.).

"""

def getStartState(self):

"""Возвращает начальное состояние."""

raise NotImplementedError

def isGoalState(self, state):

"""Проверяет, является ли состояние целевым."""

raise NotImplementedError

def getSuccessors(self, state):

"""Возвращает список кортежей (next\_state, action, step\_cost)."""

raise NotImplementedError

def getCostOfActions(self, actions):

"""Вычисляет суммарную стоимость последовательности действий."""

raise NotImplementedError

**Алгоритм поиска в глубину (DFS)**

def depthFirstSearch(problem):

"""

Реализация поиска в глубину (Depth-First Search).

Использует стек для хранения вершин. Находит любой путь к цели (не обязательно оптимальный).

Аргументы:

problem -- экземпляр класса SearchProblem (например, PositionSearchProblem)

Возвращает:

Список действий для достижения цели или пустой список, если решение не найдено.

"""

frontier = Stack() # Стек для хранения (состояние, действия)

frontier.push((problem.getStartState(), []))

visited = set() # Множество посещённых состояний

while not frontier.isEmpty():

state, actions = frontier.pop()

# Проверка на достижение цели

if problem.isGoalState(state):

return actions

# Раскрытие состояния, если оно не посещалось

if state not in visited:

visited.add(state)

# Генерация преемников

for next\_state, action, \_ in problem.getSuccessors(state):

frontier.push((next\_state, actions + [action]))

return [] # Решение не найдено

**Алгоритм поиска в ширину (BFS)**

def breadthFirstSearch(problem):

"""

Реализация поиска в ширину (Breadth-First Search).

Использует очередь для хранения вершин. Находит кратчайший путь по числу шагов.

Аргументы:

problem -- экземпляр класса SearchProblem

Возвращает:

Список действий для достижения цели или пустой список.

"""

frontier = Queue() # Очередь для хранения (состояние, действия)

frontier.push((problem.getStartState(), []))

visited = set() # Множество посещённых состояний

while not frontier.isEmpty():

state, actions = frontier.pop()

if problem.isGoalState(state):

return actions

if state not in visited:

visited.add(state)

for next\_state, action, \_ in problem.getSuccessors(state):

frontier.push((next\_state, actions + [action]))

return []

**Алгоритм поиска с равными ценами (UCS)**

def uniformCostSearch(problem):

"""

Реализация поиска с равными ценами (Uniform-Cost Search).

Использует приоритетную очередь для выбора состояния с минимальной суммарной стоимостью.

Находит путь с наименьшей общей стоимостью.

Аргументы:

problem -- экземпляр класса SearchProblem

Возвращает:

Список действий для достижения цели или пустой список.

"""

frontier = PriorityQueue() # Приоритетная очередь: (приоритет, (состояние, действия, стоимость))

frontier.push((problem.getStartState(), [], 0), 0)

visited = set()

while not frontier.isEmpty():

state, actions, cost = frontier.pop()

if problem.isGoalState(state):

return actions

if state not in visited:

visited.add(state)

for next\_state, action, step\_cost in problem.getSuccessors(state):

new\_actions = actions + [action]

new\_cost = cost + step\_cost

frontier.push((next\_state, new\_actions, new\_cost), new\_cost)

return []

Все алгоритмы следуют шаблону:

1. Инициализация структуры данных (стек/очередь/приоритетная очередь).
2. Проверка цели при извлечении состояния.
3. Добавление преемников в frontier с обновлением пути и стоимости.

**Различия**:

|  |  |  |  |
| --- | --- | --- | --- |
| **Алгоритм** | **Структура данных** | **Порядок обработки** | **Оптимальность** |
| DFS | Стек | LIFO | Нет |
| BFS | Очередь | FIFO | Да (по шагам) |
| UCS | Приоритетная очередь | По стоимости | Да (по стоимости) |

**Важные детали**:

* + Для избежания повторного посещения используется множество visited.
  + В UCS приоритет = суммарная стоимость пути (new\_cost).
  + Возвращаемый формат: список действий (['North', 'East', ...]).

### **Файл searchAgents.py**

**Базовый класс для поиска пути к определенной позиции**

from game import Agent, Directions, Actions

from util import manhattanDistance, PriorityQueue

import util

from search import SearchProblem, breadthFirstSearch, depthFirstSearch, uniformCostSearch

class PositionSearchProblem(SearchProblem):

"""

Реализует поиск пути от начальной позиции до целевой.

Используется как основа для других поисковых задач.

"""

def \_\_init\_\_(self, gameState, costFn=lambda x: 1, goal=None):

self.walls = gameState.getWalls()

self.startState = gameState.getPacmanPosition()

if goal is None:

food = gameState.getFood()

self.goal = food.asList()[0] if food.count() > 0 else self.startState

else:

self.goal = goal

self.\_visited = set()

self.\_expanded = 0

self.costFn = costFn

def getStartState(self):

return self.startState

def isGoalState(self, state):

return state == self.goal

def getSuccessors(self, state):

successors = []

for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

x,y = state

dx, dy = Actions.directionToVector(action)

nextx, nexty = int(x + dx), int(y + dy)

if not self.walls[nextx][nexty]:

nextState = (nextx, nexty)

cost = self.costFn(nextState)

successors.append((nextState, action, cost))

self.\_expanded += 1

return successors

def getCostOfActions(self, actions):

return len(actions)

**Класс для поиска всех точек с едой**

class FoodSearchProblem(SearchProblem):

"""

Реализует поиск пути для сбора всех точек с едой на карте.

Состояние включает текущую позицию и оставшуюся еду.

"""

def \_\_init\_\_(self, gameState):

self.walls = gameState.getWalls()

self.startState = gameState.getPacmanPosition()

self.food = gameState.getFood()

self.\_visited = set()

self.\_expanded = 0

def getStartState(self):

return (self.startState, self.food)

def isGoalState(self, state):

return state[1].count() == 0

def getSuccessors(self, state):

pos, food = state

successors = []

for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

x,y = pos

dx, dy = Actions.directionToVector(action)

nextx, nexty = int(x + dx), int(y + dy)

if not self.walls[nextx][nexty]:

nextFood = food.copy()

nextFood[nextx][nexty] = False

successors.append((((nextx, nexty), nextFood), action, 1))

self.\_expanded += 1

return successors

def getCostOfActions(self, actions):

return len(actions)

**Класс для поиска углов лабиринта**

class CornersProblem(SearchProblem):

"""

Реализует поиск пути для посещения всех углов лабиринта.

Состояние включает текущую позицию и непосещенные углы.

"""

def \_\_init\_\_(self, gameState):

self.walls = gameState.getWalls()

self.startState = gameState.getPacmanPosition()

self.corners = ((1,1), (1,gameState.data.layout.height-2),

(gameState.data.layout.width-2,1),

(gameState.data.layout.width-2,gameState.data.layout.height-2))

self.\_visited = set()

self.\_expanded = 0

def getStartState(self):

return (self.startState, tuple(corner for corner in self.corners))

def isGoalState(self, state):

return len(state[1]) == 0

def getSuccessors(self, state):

pos, corners = state

successors = []

for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

x,y = pos

dx, dy = Actions.directionToVector(action)

nextx, nexty = int(x + dx), int(y + dy)

if not self.walls[nextx][nexty]:

nextPos = (nextx, nexty)

visitedCorners = list(corners)

if nextPos in visitedCorners:

visitedCorners.remove(nextPos)

successors.append(((nextPos, tuple(visitedCorners)), action, 1))

self.\_expanded += 1

return successors

def getCostOfActions(self, actions):

return len(actions)

**Базовый агент для выполнения поиска**

class SearchAgent(Agent):

"""

Общий агент, который использует различные алгоритмы поиска

для нахождения пути к цели.

"""

def \_\_init\_\_(self, fn='depthFirstSearch', prob='PositionSearchProblem', heuristic='nullHeuristic'):

super(SearchAgent, self).\_\_init\_\_()

if fn == 'depthFirstSearch' or fn == 'dfs':

self.searchFunction = depthFirstSearch

elif fn == 'breadthFirstSearch' or fn == 'bfs':

self.searchFunction = breadthFirstSearch

elif fn == 'uniformCostSearch' or fn == 'ucs':

self.searchFunction = uniformCostSearch

else:

raise ValueError(f"Unknown search function: {fn}")

if prob == 'PositionSearchProblem':

self.searchType = PositionSearchProblem

elif prob == 'StayEastSearchProblem':

self.searchType = StayEastSearchProblem

elif prob == 'StayWestSearchProblem':

self.searchType = StayWestSearchProblem

elif prob == 'FoodSearchProblem':

self.searchType = FoodSearchProblem

elif prob == 'CornersProblem':

self.searchType = CornersProblem

else:

raise ValueError(f"Unknown problem type: {prob}")

self.heuristic = util.lookup(heuristic, globals())

def registerInitialState(self, gameState):

problem = self.searchType(gameState)

self.actions = self.searchFunction(problem)

if not self.actions:

self.actions = [Directions.STOP]

def getAction(self, gameState):

if not hasattr(self, 'actions') or len(self.actions) == 0:

return Directions.STOP

return self.actions.pop(0)

**Простейшая эвристика, всегда возвращает 0**

def nullHeuristic(state, problem=None):

"""

Нулевая эвристика - всегда возвращает 0.

Используется как базовый случай для других эвристик.

"""

return 0

**Эвристика для поиска еды**

def foodHeuristic(state, problem=None):

"""

Эвристика для задачи поиска еды.

Возвращает максимальное манхэттенское расстояние до любой точки с едой.

"""

position, foodGrid = state

foodList = foodGrid.asList()

if not foodList:

return 0

return max(manhattanDistance(position, food) for food in foodList)

**Эвристика для поиска углов**

def cornersHeuristic(state, problem=None):

"""

Эвристика для задачи поиска углов.

Возвращает максимальное манхэттенское расстояние до любого непосещенного угла.

"""

position, corners = state

if not corners:

return 0

return max(manhattanDistance(position, corner) for corner in corners)

**Класс для поиска с предпочтением движения на восток**

class StayEastSearchProblem(PositionSearchProblem):

"""

Модифицированная задача поиска, где стоимость увеличивается

при движении на запад.

"""

def \_\_init\_\_(self, gameState, costFn=lambda x: 1, goal=None):

super(StayEastSearchProblem, self).\_\_init\_\_(gameState, costFn, goal)

self.costFn = lambda pos: 2 \*\* (gameState.data.layout.width - pos[0])

**Класс для поиска с предпочтением движения на запад**

class StayWestSearchProblem(PositionSearchProblem):

"""

Модифицированная задача поиска, где стоимость увеличивается

при движении на восток.

"""

def \_\_init\_\_(self, gameState, costFn=lambda x: 1, goal=None):

super(StayWestSearchProblem, self).\_\_init\_\_(gameState, costFn, goal)

self.costFn = lambda pos: 2 \*\* pos[0]

**Агент для движения на восток**

class StayEastSearchAgent(SearchAgent):

"""

Агент, использующий UCS для поиска пути с предпочтением

движения на восток.

"""

def \_\_init\_\_(self):

super(StayEastSearchAgent, self).\_\_init\_\_(fn='uniformCostSearch', prob='StayEastSearchProblem')

**Агент для движения на запад**

class StayWestSearchAgent(SearchAgent):

"""

Агент, использующий UCS для поиска пути с предпочтением

движения на запад.

"""

def \_\_init\_\_(self):

super(StayWestSearchAgent, self).\_\_init\_\_(fn='uniformCostSearch', prob='StayWestSearchProblem')

**Агент для поиска ближайшей точки**

class ClosestDotSearchAgent(SearchAgent):

"""

Агент, который ищет путь к ближайшей точке с едой.

"""

def findPathToClosestDot(self, gameState):

problem = PositionSearchProblem(gameState)

return self.searchFunction(problem)

## **Полученные результаты для разных лабиринтов и их анализ**

### **Задание 1**

В рамках задания 1 была реализована стратегия поиска в глубину (Depth-First Search, DFS) в функции depthFirstSearch (файл search.py). Для проверки корректности работы алгоритма были проведены тесты на трех различных лабиринтах:

1. **tinyMaze** – простой лабиринт небольшого размера.
2. **mediumMaze** – лабиринт средней сложности.
3. **bigMaze** – крупный лабиринт с увеличенной свободной областью.

**Результаты запусков**

**tinyMaze**

python pacman.py -l tinyMaze -p SearchAgent

**Вывод:**

Pacman emerges victorious! Score: 500

Average Score: 500.0

Scores: 500.0

Win Rate: 1/1 (1.00)

Record: Win

**Анализ:**

1. Алгоритм успешно нашел путь к цели.
2. Ввиду простоты лабиринта порядок исследования узлов соответствует ожиданиям: Pacman двигался вглубь, пока не достиг выхода.
3. Путь не является оптимальным, но это ожидаемо для DFS.

**mediumMaze**

python pacman.py -l mediumMaze -p SearchAgent

**Вывод:**

Pacman emerges victorious! Score: 380

Average Score: 380.0

Scores: 380.0

Win Rate: 1/1 (1.00)

Record: Win

**Анализ:**

* + Pacman успешно прошел лабиринт, но найденный путь **не оптимален**.
  + Для mediumMaze оптимальная длина пути должна составлять 68, однако DFS нашел более длинное решение.
  + Это подтверждает свойство DFS: он находит **какой-либо** путь, но не обязательно кратчайший.

**bigMaze**

python pacman.py -l bigMaze -z .5 -p SearchAgent

**Вывод:**

Pacman emerges victorious! Score: 300

Average Score: 300.0

Scores: 300.0

Win Rate: 1/1 (1.00)

Record: Win

**Анализ:**

* + Алгоритм справился с крупным лабиринтом.
  + Как и в предыдущих случаях, путь **не оптимален**, что характерно для DFS.
  + Pacman исследовал множество тупиковых ветвей, прежде чем нашел выход.

**Общие выводы по DFS**

* **Порядок исследования** соответствует ожиданиям: алгоритм идет вглубь, пока не упрется в тупик, затем откатывается.
* **Pacman не проходит все посещенные клетки** – он движется по одной ветви, игнорируя альтернативы до момента возврата.
* **Найденный путь не оптимален**, особенно заметно на mediumMaze (длина >68).
* **DFS подходит для нахождения любого пути**, но неэффективен для поиска кратчайшего.

### **Задание 2**

В рамках задания 2 была реализована стратегия поиска в ширину (Breadth-First Search, BFS) в функции breadthFirstSearch (файл search.py). Для проверки корректности работы алгоритма были проведены тесты на лабиринтах mediumMaze и bigMaze.

**Результаты запусков**

**mediumMaze**

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

**Вывод:**

Copy

Pacman emerges victorious! Score: 442

Average Score: 442.0

Scores: 442.0

Win Rate: 1/1 (1.00)

Record: Win

**Анализ:**

* + **BFS нашел кратчайший путь** (длина 68), что подтверждается значением **Score.**
  + В отличие от DFS, алгоритм исследует все соседние узлы на текущей глубине перед переходом на следующий уровень, что гарантирует оптимальность пути.

**bigMaze**

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

**Вывод:**

Pacman emerges victorious! Score: 300

Average Score: 300.0

Scores: 300.0

Win Rate: 1/1 (1.00)

Record: Win

**Анализ:**

* + BFS успешно прошел крупный лабиринт, найдя **кратчайший путь**.
  + **Score: 300** (исходные 500 - 200 × 1 = 300) соответствует минимально возможному пути.
  + Алгоритм демонстрирует предсказуемо высокую эффективность на больших картах без тупиковых ветвлений.

**Общие выводы по BFS**

* **Находит кратчайший путь** — в отличие от DFS, BFS гарантирует оптимальность решения за счет поуровневого исследования.
* **Требует больше памяти**, чем DFS, так как хранит все узлы текущего уровня в очереди.
* **Эффективен для задач с малым branching factor**, но может быть избыточен для простых лабиринтов.

### **Задание 3**

В рамках задания 3 был реализован алгоритм поиска с равными стоимостями (Uniform Cost Search, UCS) в функции uniformCostSearch (файл search.py). Для проверки работы алгоритма были использованы следующие лабиринты:

1. **mediumMaze** – стандартный лабиринт с единичной стоимостью перемещения.
2. **mediumDottedMaze** – лабиринт с точками, где стоимость перемещения зависит от направления (предпочтение восточному направлению).
3. **mediumScaryMaze** – лабиринт с "опасными" зонами, где западное направление предпочтительнее.

**Результаты запусков**

**mediumMaze (стандартные условия)**

python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

**Вывод:**

Copy

Pacman emerges victorious! Score: 442

Average Score: 442.0

Scores: 442.0

Win Rate: 1/1 (1.00)

Record: Win

**Анализ:**

* + UCS ведет себя аналогично BFS, так как все шаги имеют одинаковую стоимость (1).
  + Найденный путь **является кратчайшим** (длина 68)
  + Результат совпадает с BFS, что ожидаемо для графа с равными весами ребер.

**mediumDottedMaze (предпочтение восточному направлению)**

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

**Вывод:**

Copy

Pacman emerges victorious! Score: 646

Average Score: 646.0

Scores: 646.0

Win Rate: 1/1 (1.00)

Record: Win

**Анализ:**

* + **Score значительно выше (646)**, что указывает на **более выгодный маршрут** (меньшие штрафы за перемещение, больше дополнительных очков из-за большего количество «яблок»).
  + Алгоритм минимизирует общую стоимость, выбирая путь с наименьшими весами (в данном случае — движение на восток).
  + Если бы использовался стандартный UCS, он также нашел бы оптимальный путь, но с учетом стоимости шагов.

**mediumScaryMaze (предпочтение западному направлению)**

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

**Вывод:**

Copy

Pacman emerges victorious! Score: 418

Average Score: 418.0

Scores: 418.0

Win Rate: 1/1 (1.00)

Record: Win

**Анализ:**

* + **Score ниже, чем в mediumDottedMaze (418 vs 646)**, что связано с **более высокой стоимостью** некоторых направлений, которые приходится выбирать, чтобы избежать призраков.
  + Алгоритм корректно учитывает веса ребер и выбирает путь, минимизирующий общую стоимость.
  + В отличие от BFS, UCS **не всегда находит кратчайший по длине путь**, но находит **самый дешевый по стоимости**.

**Общие выводы по UCS**

* **Оптимален для графов с различными стоимостями переходов.**
* **В случае равных весов (mediumMaze) ведет себя как BFS.**
* **Эффективно находит путь с минимальной общей стоимостью**, даже если он не самый короткий в шагах.
* **Требует больше вычислений, чем BFS**, так как использует приоритетную очередь вместо FIFO.

## **Результаты автооценивания**

В ходе выполнения лабораторной работы были последовательно протестированы три алгоритма поиска с использованием встроенной системы автоматической проверки **autograder.py**. Приведем детализированные результаты по каждому заданию.

**Задание q1 (Depth-First Search)**

* **Тестовые случаи:**
  + *graph\_backtrack.test*: Алгоритм корректно нашел путь ['1:A->C', '0:C->G'] с 3 расширенными узлами (A, D, C).
  + *graph\_bfs\_vs\_dfs.test*: Решение ['2:A->D', '0:D->G'] с 2 расширенными узлами (A, D).
  + *pacman\_1.test*: Для mediumMaze длина пути составила **130** шагов при **146** расширенных узлах.
* **Оценка:** 3/3
* **Вывод:**  
  Реализация DFS прошла все тесты, включая проверку на backtracking и корректность порядка обхода. На лабиринте mediumMaze продемонстрировала характерное для DFS поведение — неоптимальный по длине путь (130 vs возможные 68 в BFS).

**Задание q2 (Breadth-First Search)**

* **Тестовые случаи:**
  + *graph\_bfs\_vs\_dfs.test*: Найден оптимальный путь ['1:A->G'] с 2 расширенными узлами (A, B).
  + *pacman\_1.test*: Для mediumMaze длина пути — **68** шагов (оптимально) при **269** расширенных узлах.
* **Оценка:** 3/3
* **Вывод:**  
  BFS успешно прошел все тесты, включая проверку на оптимальность пути. На mediumMaze подтверждено нахождение кратчайшего пути (68 шагов), однако за счет большего числа расширенных узлов (269 vs 146 в DFS).

**Задание q3 (Uniform Cost Search)**

* **Тестовые случаи:**
  + *ucs\_0\_graph.test*: Корректный путь ['Right', 'Down', 'Down'] с 5 расширенными узлами (A, B, D, C, G).
  + *ucs\_3\_problemW.test*: Для mediumMaze длина пути — **152** шага (не оптимальна по длине, но оптимальна по стоимости) при **173** расширенных узлах.
  + *ucs\_5\_goalAtDequeue.test*: Решение ['1:A->B', '0:B->C', '0:C->G'] с 3 узлами (A, B, C).
* **Оценка:** 3/3
* **Вывод:**  
  UCS продемонстрировал корректную работу в условиях переменных стоимостей переходов. В отличие от BFS, находит путь, минимизирующий общую стоимость, даже если он длиннее (например, 152 шага в *problemW*).

**Итоговые показатели эффективности**

Сравнительная характеристика алгоритмов на лабиринте mediumMaze:

|  |  |  |  |
| --- | --- | --- | --- |
| **Алгоритм** | **Длина пути** | **Расширенные узлы** | **Оптимальность** |
| DFS | 130 | 146 | Нет (подграф) |
| BFS | 68 | 269 | Да (кратчайший) |
| UCS | 68/152\* | 173–269 | Да (минимальная стоимость) |

\*Зависит от стоимости переходов в конкретном тесте.

**Заключение**

* Все реализации алгоритмов (**DFS, BFS, UCS**) успешно прошли автоматизированное тестирование, что подтверждает их корректность.
* **Наибольшая эффективность по памяти** у DFS (146 узлов), **по оптимальности пути** — у BFS и UCS.
* UCS показал свою универсальность в условиях **неоднородных стоимостей** переходов, жертвуя длиной пути для минимизации общей стоимости.

**Итоговая оценка: 9/9**

# **Заключение**

В ходе выполнения лабораторной работы были успешно реализованы и исследованы три алгоритма слепого поиска: **поиск в глубину (DFS)**, **поиск в ширину (BFS)** и **поиск с равными стоимостями (UCS)**. Каждый из алгоритмов был протестирован на различных лабиринтах, что позволило оценить их эффективность, оптимальность и применимость в зависимости от условий задачи.

**DFS**

1. Находит путь к цели, но **не гарантирует его оптимальность** (длина пути в mediumMaze — 130 шагов против возможных 68).
2. **Эффективен по памяти** (146 расширенных узлов в mediumMaze), но исследует множество тупиковых ветвей.
3. **Применимость**: Использование оправдано при ограниченных ресурсах памяти, когда допустимо неоптимальное решение.

**BFS**

1. **Гарантированно находит кратчайший путь** (68 шагов в mediumMaze).
2. **Требует больше памяти**, чем DFS (269 узлов), из-за хранения всех вершин текущего уровня.
3. **Применимость**: Оптимален для задач, где критична минимальная длина пути.

**UCS**

1. **Обобщает BFS** для случаев с переменными стоимостями переходов.
2. Находит **путь с минимальной общей стоимостью**, даже если он длиннее (например, 152 шага в problemW).
3. **Применимость**: Незаменим при неоднородных весах ребер (например, избегание привидений, предпочтение направлений).