Министерство образования и науки Российской Федерации Федеральное государственное автономное образовательное учреждение высшего профессионального образования «Севастопольский государственный университет»

ОТЧЕТ

о выполнении лабораторной работы № 2 по дисциплине

«Методы и системы искусственного интеллекта»

Выполнил:

ст. гр. ИС/б-22-1-о

Гюнтер М. Ю .

Проверил:

доцент кафедры

“Информационные системы”

Бондарев В. Н.

Севастополь,2025

# 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.

# Ход работы

## Изучение среды AI Pacman

Продемонстрируем работу методов problem.getStartState(), problem.isGoalState(), problem.getSuccessors() среды AI Pacman (рисунок ‎2.1), необходимых для получения начального состояния проблемы, проверки соответствия текущего состояния целевому и получения дочерних вершин для текущей вершины.

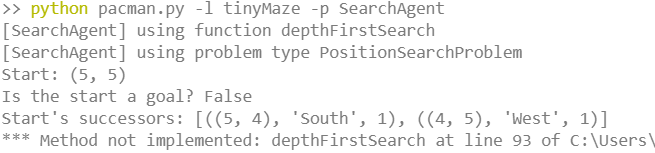


Рисунок ‎2.1 – Результат работы методов problem.getStartState(), problem.isGoalState(), problem.getSuccessors()

Также проверим работоспособность поискового агента SearchAgent (рисунок ‎2.2) посредством выполнения программы pacman.py с рядом параметров (листинг ‎2.1), позволяющих провести поиск по лабиринту небольшого размера tinyMaze с использованием метода tinyMazeSearch().

Листинг ‎2.1 – Проверка работоспособности поискового агента

python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch

Изображение выглядит как текст, снимок экрана, Шрифт, алгебра

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок ‎2.2 – Результаты работы поискового агента

## Поиск в глубину

Для метода depthFirstSearch() среды AI Pacman был реализован алгоритм поиска в глубину (листинг ‎2.2).

Листинг ‎2.2 – Алгоритм поиска в глубину

def depthFirstSearch(problem):

"""

Поиск в глубину.

Ваш алгоритм поиска должен возвращать список действий, которые

ведут к цели. Убедитесь, что реализуете алгоритм поиска на графе

Прежде чем кодировать,полезно выполнить функцию с этими простыми

командами,чтобы понять смысл задачи (problem), передаваемой на вход:

print("Start:", problem.getStartState())

print("Is the start a goal?", problem.isGoalState(problem.getStartState()))

print("Start's successors:", problem.getSuccessors(problem.getStartState()))

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

OPEN = util.Stack()

CLOSED = set()

PATHS = {}

def pushSuccessors(parentCrds, OPEN, CLOSED, PATHS):

parentPath = PATHS[parentCrds]

children = problem.getSuccessors(parentCrds)

for childCrds, childAct, \_ in children:

if childCrds not in CLOSED:

childPath = parentPath + [childAct]

OPEN.push(childCrds)

PATHS[childCrds] = childPath

startStateCrds = problem.getStartState()

PATHS[startStateCrds] = []

OPEN.push(startStateCrds)

while not OPEN.isEmpty():

stateCrds = OPEN.pop()

if problem.isGoalState(stateCrds):

return PATHS[stateCrds]

CLOSED.add(stateCrds)

pushSuccessors(stateCrds, OPEN, CLOSED, PATHS)

Результаты работы алгоритма DFS для трех лабиринтов разного размера приведены на рисунке ‎2.3.

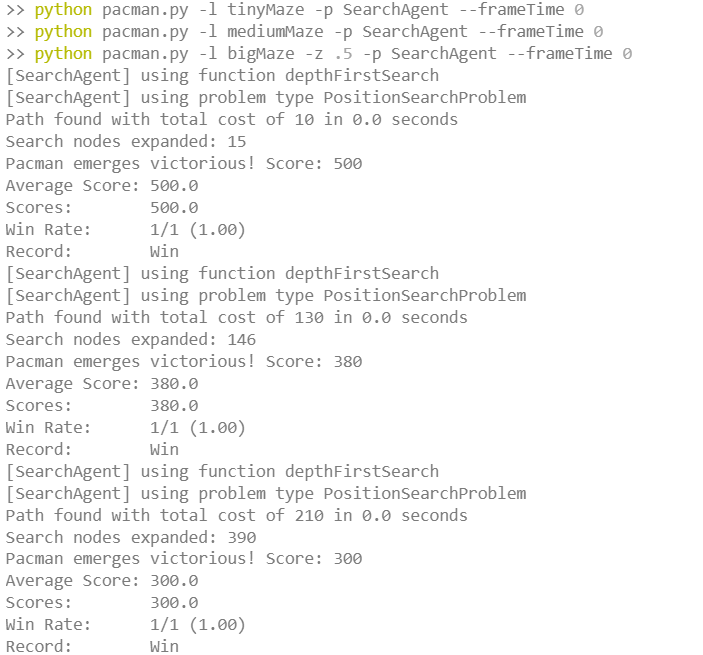


Рисунок ‎2.3 – Результаты работы алгоритма DFS

Приведем также схему, отображающую порядок обследования Pacman-ом вершин лабиринта, в которой яркость окраски вершины уменьшается в зависимости от промежутка времени, на котором Pacman обследовал вершину (рисунок ‎2.4).

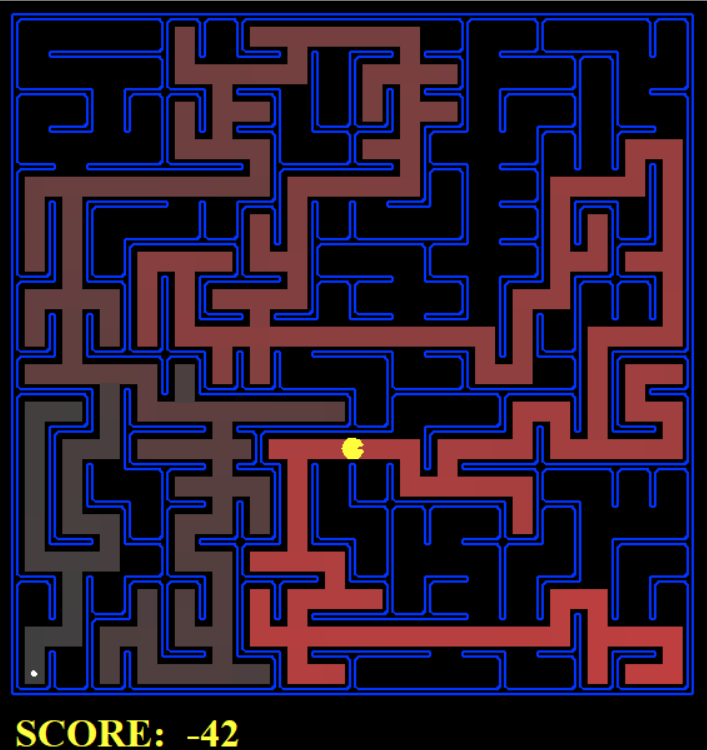


Рисунок ‎2.4 – Схема обследования Pacman-ом вершин лабиринта при поиске в глубину

На схеме хорошо видно, что в конце лабиринта Pacman изначально выбрал неверный путь, находящийся справа. Обследовав этот путь до самого конца, Pacman вернулся в исходную точку и начал обследовать левый путь, который привел его к выходу из лабиринта. Это поведение совпадает с ожидаемым для алгоритма поиска в глубину – агент сначала обследует “наиболее глубокие” состояния и, не найдя среди них целевого состояния, возвращается на шаг назад и осуществляет поиск среди альтернатив.

Стоит отметить, что сам агент не будет перемещаться по всем обследованным вершинам – в результате обследования он получит путь без “тупиков”, в котором каждое новое состояние приближает его ближе к целевому. Тем не менее, полученный путь не обязательно будет оптимальным.

Результаты работы автооценивателя autograder.py для алгоритма DFS приведены на рисунке ‎2.5.

Изображение выглядит как текст, Шрифт, снимок экрана, рукописный текст

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок ‎2.5 – Результаты работы автооценивателя для алгоритма DFS

Для классического алгоритма поиска в глубину при поиске по дереву временная сложность является экспоненциальной, и пропорциональна , где B – коэффициент ветвления, а m – максимальная глубина залегания решения. Пространственная же сложность является линейной, и пропорциональная .

## Поиск в ширину

Для метода breadthFirstSearch() среды AI Pacman был реализован алгоритм поиска в ширину (листинг ‎2.3).

Листинг ‎2.3 – Алгоритм поиска в ширину

def breadthFirstSearch(problem):

"""Находит самые поверхностные узлы в дереве поиска """

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

OPEN = util.Queue()

CLOSED = set()

PATHS = {}

def pushSuccessors(parentCrds, OPEN, CLOSED, PATHS):

parentPath = PATHS[parentCrds]

children = problem.getSuccessors(parentCrds)

for childCrds, childAct, \_ in children:

# The first time a child node is found, the shortest path to it is already known

if childCrds not in CLOSED and not PATHS.get(childCrds):

childPath = parentPath + [childAct]

OPEN.push(childCrds)

PATHS[childCrds] = childPath

startStateCrds = problem.getStartState()

PATHS[startStateCrds] = []

OPEN.push(startStateCrds)

while not OPEN.isEmpty():

stateCrds = OPEN.pop()

if problem.isGoalState(stateCrds):

return PATHS[stateCrds]

CLOSED.add(stateCrds)

pushSuccessors(stateCrds, OPEN, CLOSED, PATHS)

Результаты работы алгоритма BFS для двух лабиринтов разного размера приведены на рисунке ‎2.6.

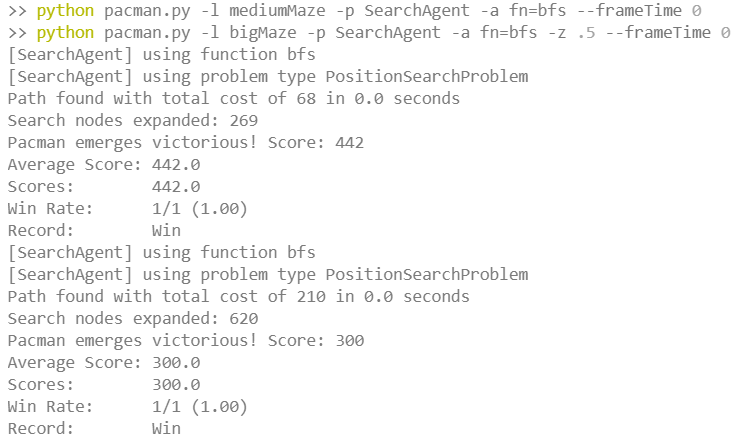


Рисунок ‎2.6 – Результаты работы алгоритма BFS

Приведем также схему, отображающую порядок обследования Pacman-ом вершин лабиринта (рисунок ‎2.7).

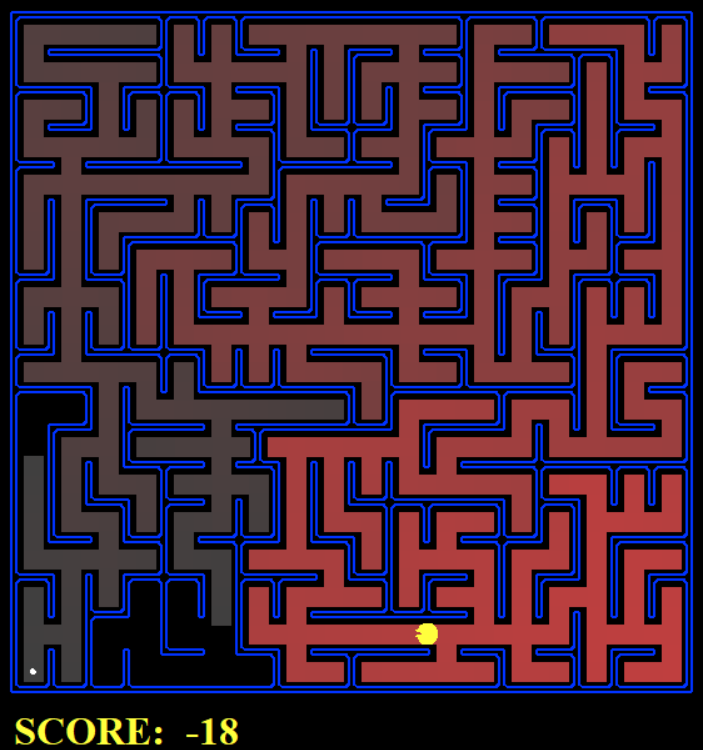


Рисунок ‎2.7 – Схема обследования Pacman-ом вершин лабиринта при поиске в ширину

На схеме хорошо видно, что на протяжении всего лабиринта Pacman обследует все возможные состояния, пока не достигает целевого. Учитывая, что при поиске в ширину все вершины имеют равную стоимость, такой подход обеспечивает нахождение минимального решения, то есть оптимального пути из начальной вершины к целевой вершине.

Результаты работы автооценивателя autograder.py для алгоритма BFS приведены на рисунке ‎2.8.

Изображение выглядит как текст, снимок экрана, Шрифт, число

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок ‎2.8 – Результаты работы автооценивателя для алгоритма BFS

Для алгоритма поиска в ширину при поиске по дереву временная и пространственная сложности являются экспоненциальными, и пропорциональны , где B – коэффициент ветвления, а s – глубина залегания наиболее поверхностного решения.

## Поиск алгоритмом равных цен

Для метода uniformCostSearch() среды AI Pacman был реализован алгоритм поиска в ширину (листинг ‎2.4).

Листинг ‎2.4 – Алгоритм равных цен

def uniformCostSearch(problem):

"""Находит узел минимальной стоимости """

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

def pushSuccessors(parentCrds, OPEN, CLOSED, PRIORITIES):

parentCost = PRIORITIES[parentCrds][0]

children = problem.getSuccessors(parentCrds)

for childCrds, childAct, parentChildCost in children:

if childCrds not in CLOSED:

childCost = parentCost + parentChildCost

OPEN.update(childCrds, childCost)

if childCost < PRIORITIES.get(childCrds, (float('inf'), None, None))[0]:

PRIORITIES[childCrds] = (childCost, childAct, parentCrds)

def findPath(goalCrds, PRIORITIES):

goalAct = PRIORITIES[goalCrds][1]

parentCrds = PRIORITIES[goalCrds][2]

parent = PRIORITIES[parentCrds]

path = [goalAct]

while parent[2] != None:

parentAct = parent[1]

path.append(parentAct)

parent = PRIORITIES[parent[2]]

path.reverse()

return path

OPEN = util.PriorityQueue()

CLOSED = set({})

PRIORITIES = {}

startStateCrds = problem.getStartState()

PRIORITIES[startStateCrds] = (0, None, None)

OPEN.push(startStateCrds, 0)

while not OPEN.isEmpty():

stateCrds = OPEN.pop()

if problem.isGoalState(stateCrds):

return findPath(stateCrds, PRIORITIES)

CLOSED.add(stateCrds)

pushSuccessors(stateCrds, OPEN, CLOSED, PRIORITIES)

Результаты работы алгоритма UCS для трех лабиринтов с существенными различиями в стоимостях путей приведены на рисунке ‎2.9.

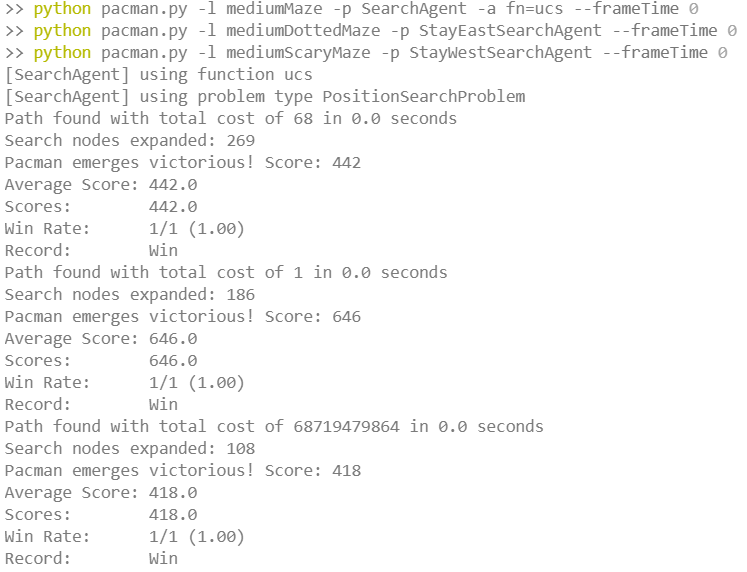


Рисунок ‎2.9 – Результаты работы алгоритма UCS для трех лабиринтов с существенными различиями в стоимостях путей

Результаты работы автооценивателя autograder.py для алгоритма UCS приведены на рисунке ‎2.10.

Изображение выглядит как текст, снимок экрана, письмо, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок ‎2.10 – Результаты работы автооценивателя для алгоритма UCS

Для алгоритма равных цен при поиске по дереву временная и пространственная сложности являются экспоненциальными, и пропорциональны , где B – коэффициент ветвления, С – стоимость оптимального решения, c – средняя минимальная стоимость действия.

# Вывод:

В результате выполнения лабораторной работы были изучены алгоритмы “слепого”, или неинформированного поиска – поиск в ширину (Breadth First Search, BDF), поиск в глубину (Depth First Search, DFS), алгоритм равных цен (Uniform Cost Search, UCS). Были детально рассмотрены отличия между алгоритмами, как по критериям оценки эффективности алгоритмов, таких, как полнота, оптимальность, минимальность, временная сложность, пространственная сложность, так и по общим принципам работы алгоритмов. Каждый из исследованных алгоритмов слепого поиска был реализован в среде AI Pacman с использованием специальных структур данных – очереди, стека и очереди с приоритетами, необходимых для реализации обозначенных алгоритмов.