**ЛАБОРАТОРНАЯ РАБОТА № 3**

**«ИССЛЕДОВАНИЕ ИНФОРМИРОВАННЫХ МЕТОДОВ ПОИСКА**

**РЕШЕНИЙ ЗАДАЧ В ПРОСТРАНСТВЕ СОСТОЯНИЙ»**

**Цель работы**

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

**Постановка задачи**

**Задание 1** (20 баллов). A\*- поиск

Реализуйте A\*-алгоритм на графе состояний в шаблоне функции aStarSearch в файле search.py, которая принимает в качестве аргумента эвристическую функцию. Эвристическая функция имеет два аргумента: состояние агента (основной аргумент) и задача (problem) (для справочной информации). Эвристическая функция nullHeuristic в search.py является тривиальным примером.

Протестируйте свою реализацию A\*-поиска на задаче поиска пути в лабиринте, используя эвристику манхэттенского расстояния (уже реализованную как manhattanHeuristic в searchAgents.py):

python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

Вы должны увидеть, что A\*- алгоритм находит оптимальное решение немного быстрее, чем поиск в соответствии с алгоритмом равных цен (он раскрывает около 549 узлов по сравнению с 620 узлами, из-за учета приоритета узлов числа могут немного отличаться).

Проверьте результаты поиска на лабиринте openMaze. Что можно сказать о различных стратегиях поиска?

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация А\*-поиска все тестовые примеры автооценивателя:

python autograder.py -q q4

**Задание 2** (20 баллов). Поиск всех углов

Настоящая сила A\*-поиска станет очевидной только при решении более сложной задачи. Сформулируем новую проблему и разработаем эвристику для ее решения.

В углах лабиринта есть четыре точки, по одной в каждом углу. Необходимо найти кратчайший путь, который связан с посещением всех четырех углов (независимо от того, есть ли в лабиринте еда или нет). Обратите внимание, что для некоторых лабиринтов, таких как tinyCorners, кратчайший путь не всегда ведет к ближайшей точке в первую очередь!

Подсказка: кратчайший путь через tinyCorners состоит из 28 шагов.

Примечание. Обязательно выполните задание 2 предыдущей лабораторной работы, прежде чем решать это задание

Реализуйте задачу поиска углов, дописав участки кода в определении класса CornersProblem в файле searchAgents.py. Вам нужно будет выбрать такое представление состояния, которое кодирует всю информацию, необходимую для определения достижения цели: посетил ли агент все четыре угла.

Протестируйте поискового агента, выполнив команды:

python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

Чтобы получить высокую оценку за выполнение задания, вам необходимо определить абстрактное представление состояния, которое не содержит несущественную информацию (например, положение призраков, где находится дополнительная еда и т. п.). В частности, не используйте Pacman GameState в качестве состояния поиска. Ваш код будет очень медленным.

Подсказка 1. Единственные части игрового состояния, на которые вам нужно ссылаться в своей реализации, — это начальная позиция Pacman и расположение четырех углов.

Подсказка 2: при написании кода getSuccessors не забудьте добавить потомков в список преемников со стоимостью 1.

Наша реализация breadthFirstSearch расширяет почти 2000 поисковых узлов на задаче mediumCorners. Однако эвристика (используемая в A\*-поиске) может уменьшить этот объем.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация агента, выполняющего поиск углов, все тесты автооценивателя:

python autograder.py -q q5

**Задание 3** (20 баллов). Эвристика для задачи поиска углов

Реализуйте нетривиальную монотонную эвристику для задачи поиска углов в методе cornersHeuristic класса CornersProblem. Проверьте реализацию, выполнив команду:

python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

Здесь AStarCornersAgent — это сокращение (ярлык) для

-p SearchAgent -a fn = aStarSearch, prob = CornersProblem, heuristic = cornersHeuristic

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация эвристической функции все тесты автооценивателя

python autograder.py -q q6

**Задание 4** (20 баллов). Задача поедания всех гранул

Теперь мы решим сложную задачу поиска: агент должен съесть всю еду за минимальное количество шагов. Для этого нам понадобится новое определение задачи поиска, которое формализует поедание всех пищевых гранул. Эта задача реализуется классом FoodSearchProblem в searchAgents.py.

Решение определяется как путь, вдоль которого собирается вся еда в мире Pacman. Для данного задания не учитываются призраки или энергетические гранулы; решения зависят только от расположения стен, пищевых гранул и агента. (Конечно, призраки могут ухудшить решения! Мы вернемся к этому в следующих лабораторных работах.) Если будут правильно написаны общие методы поиска, то A\*-поиск с нулевой эвристикой (эквивалент поиска с равномерной стоимостью) должен быстро найти оптимальное решение для testSearch без изменения кода с вашей стороны (общая стоимость пути 7). Проверьте:

python pacman.py -l testSearch -p AStarFoodSearchAgent

Здесь AStarFoodSearchAgent - это сокращение (ярлык) для

-p SearchAgent -a fn = astar, prob = FoodSearchProblem, эвристика = foodHeuristic

Вы должны обнаружить, что алгоритм UCS начинает замедляться даже при простом лабиринте tinySearch.

Примечание. Обязательно выполните задание 1, прежде чем работать над заданием 4.

Допишите код в функции foodHeuristic в файле searchAgents.py, определив монотонную (согласованную) эвристику для класса FoodSearchProblem. Проверьте работу агента на сложной задаче поиска (требует времени):

python pacman.py -l trickySearch -p AStarFoodSearchAgent

Любая нетривиальная неотрицательная согласованная эвристика, разработанная Вами, получит 5 баллов. Убедитесь, что ваша эвристика возвращает 0 при каждом целевом состоянии и никогда не возвращает отрицательное значение. В зависимости от того, сколько узлов будет раскрывать ваша эвристика, вы получите дополнительные баллы.

Помните: если ваша эвристика немонотонна, вы не получите баллов. Сможете ли вы решить mediumSearch за короткое время? Если да, то это либо впечатляющий результат, либо ваша эвристика немонотонна.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тесты автоценивателя:

python autograder.py -q q7

**Задание 5** (20 баллов). Субоптимальный поиск

Иногда даже с помощью A\*-поиска при хорошей эвристике найти оптимальный путь через все точки накладно. В таких случаях можно просто выполнить быстрый поиск “достаточно” хорошего пути. В этом задании необходимо реализовать агента, который всегда жадно ест ближайшую гранулу. Такой агент ClosestDotSearchAgent реализован в файле searchAgents.py, но в нем отсутствует ключевая функция, которая находит путь к ближайшей точке.

Реализуйте функцию findPathToClosestDot в searchAgents.py. Проверьте решение:

python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

Агент выполнит поиск пути в этом лабиринте субоптимально, менее чем засекунду со стоимостью пути 350.

Подсказка: самый быстрый способ завершить findPathToClosestDot - это заполнить в классе AnyFoodSearchProblem функцию проверки достижения цели isGoalState. А затем завершить определение findPathToClosestDot с помощью соответствующей функции поиска, написанной ранее. Решение должно получиться очень коротким!

Ваш агент ClosestDotSearchAgent не всегда будет находить кратчайший путь через лабиринт. Убедитесь, что вы понимаете почему, и попробуйте придумать небольшой пример, где многократный переход к ближайшей точке не приводит к нахождению кратчайшего пути для съедания всех точек.

Выполните приведенную ниже команду, чтобы проверить, проходит ли ваша реализация все тесты автооценивателя:

python autograder.py -q q8

**Ход работы**

1. Был реализован A\*-алгоритм на графе состояния, код которого приведён в листинге 1.

Листинг 1 – A\*-алгоритм

def aStarSearch(problem, heuristic=nullHeuristic):

"""

Находит узел с наименьшей комбинированной стоимостью, включающей эвристику

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

frontier = util.PriorityQueue()

visited = []

start = problem.getStartState()

startNode = (start, [], 0)

frontier.push(startNode, 0)

while not frontier.isEmpty():

currentState, actions, currentCost = frontier.pop()

currentNode = (currentState, currentCost)

visited.append((currentState, currentCost))

if problem.isGoalState(currentState):

return actions

else:

successors = problem.getSuccessors(currentState)

for sState, sAction, sCost in successors:

newAction = actions + [sAction]

newCost = problem.getCostOfActions(newAction)

newNode = (sState, newAction, newCost)

alreadyVisited = False

for visit in visited:

vState, vCost = visit

if (sState == vState) and (newCost >= vCost):

alreadyVisited = True

if not alreadyVisited:

frontier.push(newNode, newCost + heuristic(sState, problem))

visited.append((sState, newCost))

return actions

Затем, реализация данного алгоритма была протестирована на задаче поиска пути в лабиринте, используя эвристику манхеттенского расстояния. Для этого была выполнена команда: python3.6 pacman.py -l bigMaze -z .6 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic. Результат изображен на рисунке 1.

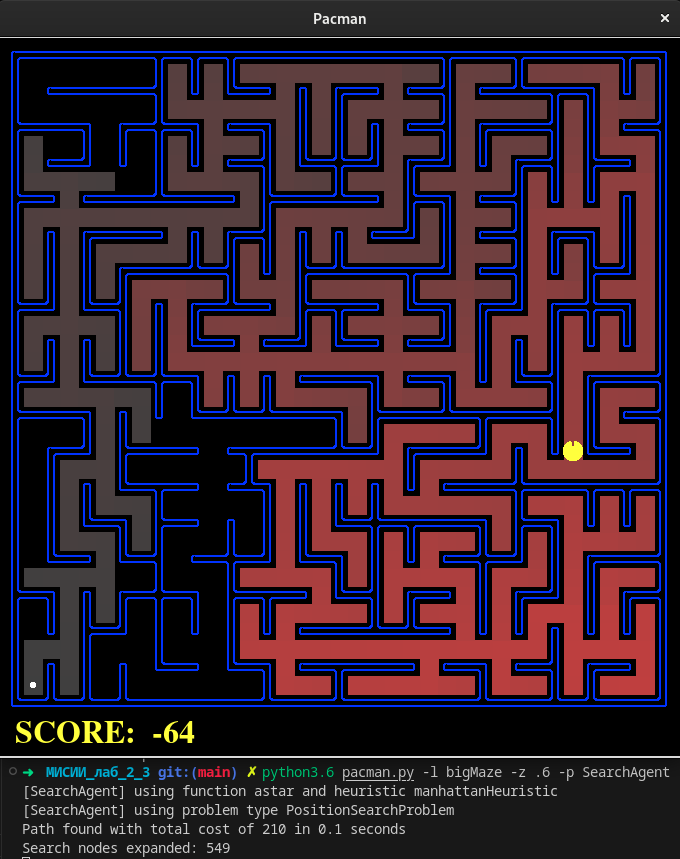


Рисунок 1 – Результат поиска с использованием алгоритма A\*

Как и ожидалось, количество раскрытых узлов оказалось равным 549, что может говорить о правильности написанного алгоритма.

Также была проверена корректность разработанного алгоритма при помощи автооценивателя. На рисунке 2 продемонстрирована корректность прохождения тестов.

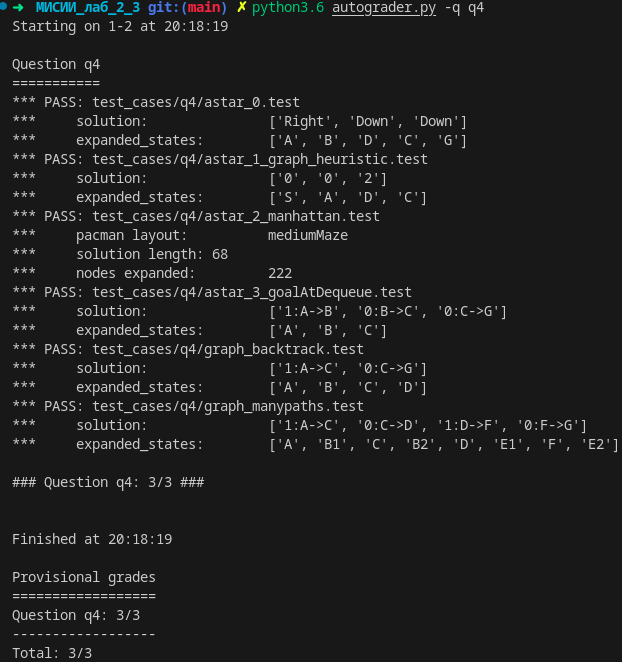


Рисунок 2 – Прохождение тестов при помощи автооценивателя

2. Была реализована задача поиска углов, дописав участки кода в определении класса CornersProblem в файле searchAgents.py. Код класса CornersProblem приведен в листинге 2.

Листинг 2 – Класс CornersProblem

class CornersProblem(search.SearchProblem):

"""

Эта задача поиска находит пути через все четыре угла схемы игры.

Вы должны выбрать подходящее пространство состояний и функцию-преемник.

"""

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

"""

Хранит стены, исходную позицию Пакмана и углы.

"""

self.walls = startingGameState.getWalls()

self.startingPosition = startingGameState.getPacmanPosition()

top, right = self.walls.height-2, self.walls.width-2

self.corners = ((1,1), (1,top), (right, 1), (right, top))

for corner in self.corners:

if not startingGameState.hasFood(\*corner):

print('Warning: no food in corner ' + str(corner))

self.\_expanded = 0 # НЕ МЕНЯЙТЕ; Количество раскрытых поисковых узлов

# Пожалуйста, добавьте сюда любой код, который вы хотели бы использовать

# при инициализации задачи

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

self.startingGameState = startingGameState

visitedCorners = [False, False, False, False]

if self.startingPosition == self.corners[0]:

visitedCorners[0] = True

if self.startingPosition == self.corners[1]:

visitedCorners[1] = True

if self.startingPosition == self.corners[2]:

visitedCorners[2] = True

if self.startingPosition == self.corners[3]:

visitedCorners[3] = True

self.startingState = (self.startingPosition, tuple(visitedCorners))

def getStartState(self):

"""

Возвращает начальное состояние (в вашем пространстве состояний, а

неполное состояние пространства игры Pacman)

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

return self.startingState

def isGoalState(self, state):

"""

Проверяет, является ли это состояние поиска целевым состоянием задачи.

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

visitedCorners = state[1]

if visitedCorners[0] and visitedCorners[1] and visitedCorners[2] and visitedCorners[3]:

return True

else:

return False

def getSuccessors(self, state):

"""

Возвращает состояния-преемники, действия, и стоимость 1.

Как отмечено в search.py:

Для данного состояния возвращает список из триплетов (successor,

action, stepCost), где 'successor' - это преемник текущего состояния,

'action' - это действие, необходимое для его достижения,

'stepCost' - затраты для шага перхода к этому преемнику.

"""

successors = []

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

# Добавьте состояние-приемник в список приемников, если действие является

# допустимым

# Ниже фрагмент кода, который проверяет, не попадает ли новая позиция на

# стену лабиринта:

# x,y = currentPosition

# dx, dy = Actions.directionToVector(action)

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

# hitsWall = self.walls[nextx][nexty]

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

x, y = state[0]

dx, dy = Actions.directionToVector(action)

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

hitsWall = self.walls[nextx][nexty]

if not hitsWall:

nextState = (nextx, nexty)

visitedCorners = list(state[1])

if nextState == self.corners[0]:

visitedCorners[0] = True

if nextState == self.corners[1]:

visitedCorners[1] = True

if nextState == self.corners[2]:

visitedCorners[2] = True

if nextState == self.corners[3]:

visitedCorners[3] = True

cost = 1

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

self.\_expanded += 1 # НЕ МЕНЯЙТЕ!

return successors

def getCostOfActions(self, actions):

"""

Возвращает стоимость определенной последовательности действий. Если эти

действия включают недопустимый ход, вщзвращает 999999.

"""

if actions == None: return 999999

x,y= self.startingPosition

for action in actions:

dx, dy = Actions.directionToVector(action)

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

if self.walls[x][y]: return 999999

return len(actions)

def cornersHeuristic(state, problem):

"""

Эвристика для задачи поиска углов, которую необходимо определить.

state: текущее состояние поиска

(структура данных, которую вы выбрали в своей поисковой задаче)

problem: экземпляр CornersProblem для схемы лабиринта.

Эта функция всегда должна возвращать число, которое является нижней границей

кратчайшего пути от состояния к цели задачи; т.е. она должна быть

допустимой (а также монотонной).

"""

corners = problem.corners # Координаты углов

walls = problem.walls # Стены лабиринта в виде объекта Grid (game.py)

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

return 0 # Default to trivial solution

Затем, поисковой агент был протестирован, выполнив команды:

python3.6 pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

python3.6 pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

Результат выполнения команд представлен на рисунках 3 и 4.



Рисунок 3 – Задача поиска углов в маленьком лабиринте

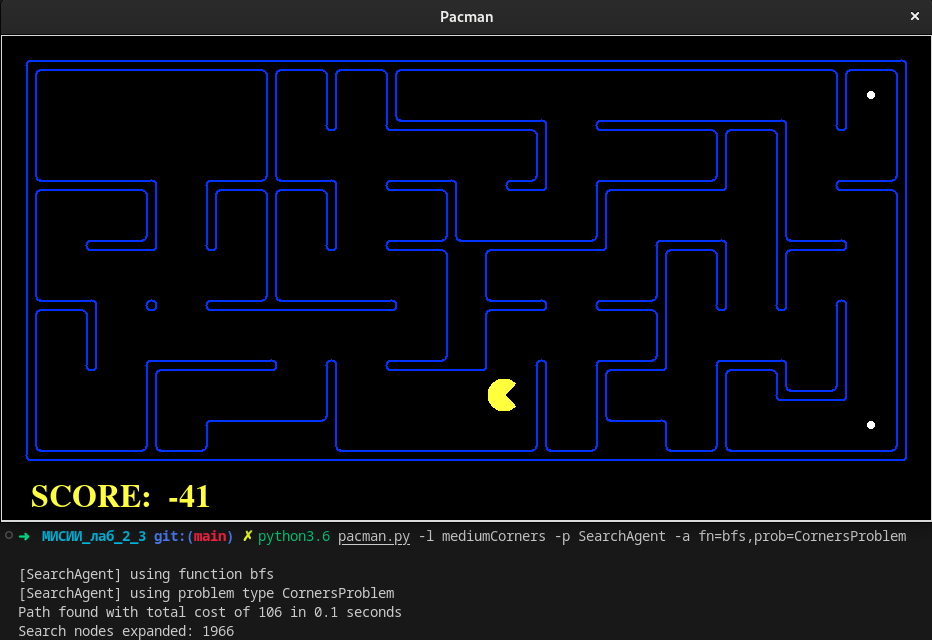


Рисунок 4 – Задача поиска углов в среднем лабиринте

Далее была проверена корректность реализованного агента при помощи автооценивателя. На рисунке 5 изображено корректное прохождение тестов.

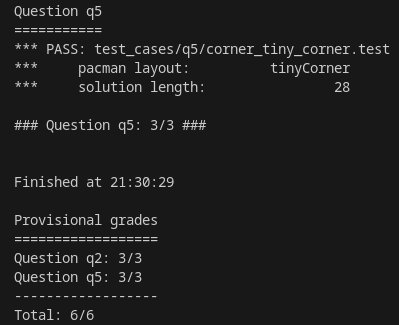


Рисунок 5 – Прохождение тестов при помощи автооценивателя

3. Была реализована нетривиальная монотонная эвристика для задачи поиска углов в методе cornersHeuristic класса CornersProblem. Код представлен в листинге 3.

Листинг 3 – Эвристика для задачи поиска углов

def cornersHeuristic(state, problem):

"""

Эвристика для задачи поиска углов, которую необходимо определить.

state: текущее состояние поиска

(структура данных, которую вы выбрали в своей поисковой задаче)

problem: экземпляр CornersProblem для схемы лабиринта.

Эта функция всегда должна возвращать число, которое является нижней границей

кратчайшего пути от состояния к цели задачи; т.е. она должна быть

допустимой (а также монотонной).

"""

corners = problem.corners # Координаты углов

walls = problem.walls # Стены лабиринта в виде объекта Grid (game.py)

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

heuristic = 0

currentLocation = state[0]

cornersUnvisited = state[1]

#unvisited corners

unvisitedCorners = []

for i in range(len(cornersUnvisited)):

if not cornersUnvisited[i]:

unvisitedCorners.append(corners[i])

#calculate the distance from current node to all corner nodes

if len(unvisitedCorners) > 0:

closestPoint = findClosestPoint(currentLocation, unvisitedCorners)

farthestPoint = findFarthestPoint(currentLocation, unvisitedCorners)

closestPointIndex = closestPoint[0]

farthestPointIndex = farthestPoint[0]

currentNode = problem.startingGameState

closestNode = unvisitedCorners[closestPointIndex]

farthestNode = unvisitedCorners[farthestPointIndex]

#mazeDistance returns maze distance btw 2 points: eg. mazeDistance( (2,4), (5,6), gameState)

#distance between current location and closest manhattan node

currentToClosest = mazeDistance(currentLocation, closestNode, currentNode)

#distance between closest manhattan node and farthest manhattan node

closestToFarthest = mazeDistance(closestNode, farthestNode, currentNode)

heuristic = currentToClosest + closestToFarthest

return heuristic

Затем была проверена реализация написанного метода, для чего была выполнена команда: python3.6 pacman.py -l mediumCorners -p AStarCornersAgent. На рисунке 6 представлен результат выполнения команды.

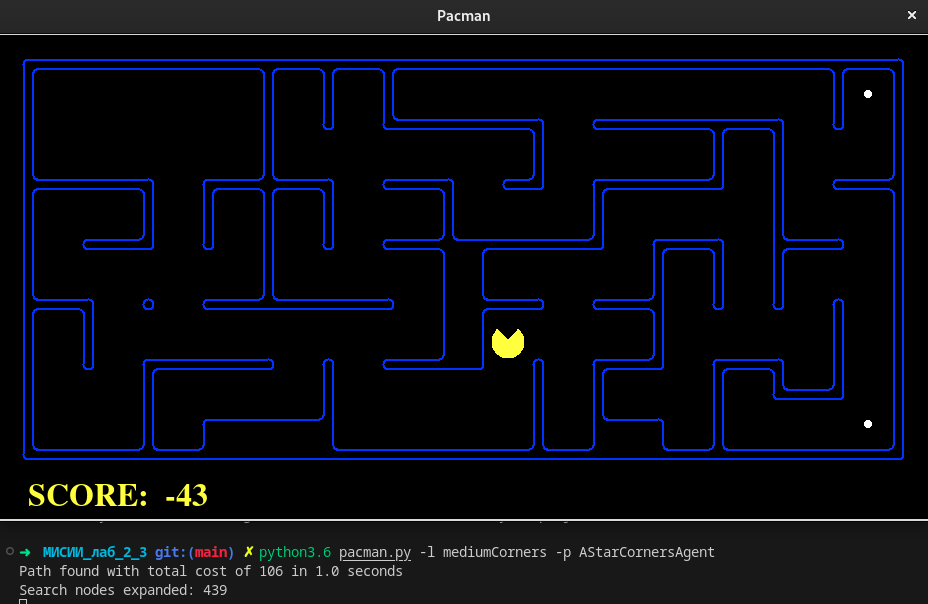


Рисунок 6 – Задача поиска углов

После чего реализация разработанной эвристика была проверена при помощи автооценивателя. На рисунке 7 представлены результаты тестов.

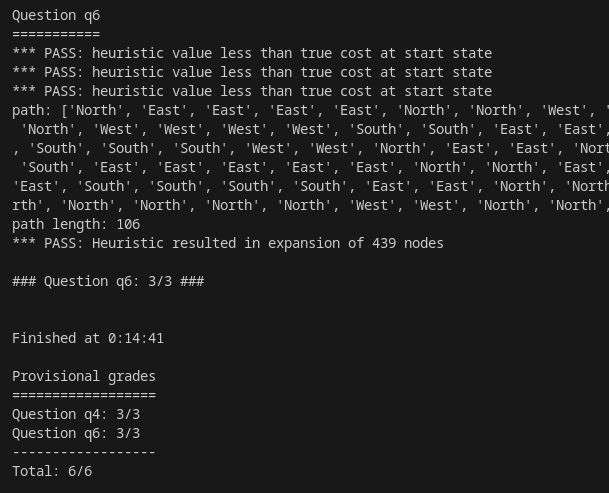


Рисунок 7 – Прохождение тестов при помощи автооценивателя

4. Был дописан код в функции foodHeuristic в файле searchAgents.py, определив монотонную (согласованную) эвристику для класса FoodSearchProblem. Код представлен в листинге 4.

Листинг 4 – Функция foodHeuristic

def foodHeuristic(state, problem):

"""

вристика для FoodSearchProblem,которую вы должны определитьь.

Эта эвристика должна быть монотонной, чтобы гарантировать правильность.

Сначала попробуйте придумать допустимую эвристику; почти все допустимые

эвристики также будут согласованными (монотонными).

Если при использовании A\*-поиска будет найдено решение, которое хуже, чем поиск

с равной стоимостью, ваша эвристика немонотонная и, вероятно, недопустима!

С другой стороны, недопустимые или немонотонные эвристики могут найти

оптимальные решения, поэтому внимательны.

Состояние - это кортеж (pacmanPosition, foodGrid), где foodGrid -

это Grid (см. game.py) со значениями True или False. Вместо этого

вы можете вызвать foodGrid.asList (), чтобы получить список координат еды.

Если вы хотите сохранить информацию для повторного использования в других

вызовах heuristic, вы можете использовать словарь problem.heuristicInfo.

Например, если вы хотите сосчитать стены только один раз и сохранить

значение, используйте: problem.heuristicInfo ['wallCount']=problem.walls.count()

Последующие вызовы этой эвристики могут получить доступ к этой информации

issue.heuristicInfo ['wallCount']

"""

position, foodGrid = state

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

foodList = foodGrid.asList()

heuristic = 0

if len(foodList) == 0:

return 0

closestFood = closestPoint(position, foodList)

farthestFood = farthestPoint(position, foodList)

heuristic = manhattanDistance(closestFood, position)

heuristic = heuristic + manhattanDistance(farthestFood, closestFood)

gameState = problem.getGameState()

d1 = mazeDistance(closestFood, position, gameState)

d2 = mazeDistance(farthestFood, closestFood, gameState)

d3 = mazeDistance(farthestFood, position, gameState)

leftPoints = 0

for (x,y) in foodList:

flag = 0

if x!=farthestFood[0] and x!=closestFood[0]:

leftPoints = leftPoints + 1

flag = 1

if flag == 0:

if y!=farthestFood[1] and y!=closestFood[1]:

leftPoints = leftPoints + 1

leftPoints2 = 0

for (x,y) in foodList:

flag = 0

if x!=position[0] and x!=closestFood[0]:

leftPoints2 = leftPoints2 + 1

flag = 1

if flag == 0:

if y!=position[1] and y!=closestFood[1]:

leftPoints2 = leftPoints2 + 1

return d1 + leftPoints2

Затем была проверена работа агента на сложной задаче поиска. Для этого была выполнена команда python3.6 pacman.py -l trickySearch -p AStarFoodSearchAgent. Результат продемонстрирован на рисунке 8.

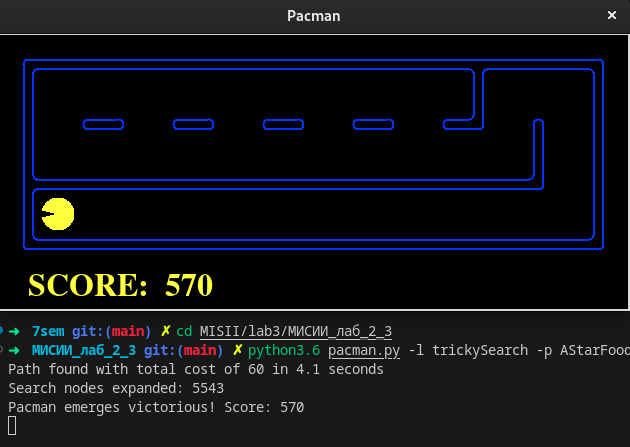


Рисунок 8 – Результат прохождения trickySearch

После чего, реализованная эвристика была проверена при помощи автооценивателя. Результаты тестов изображены на рисунке 9.

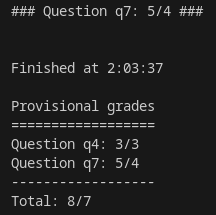


Рисунок 9 – Прохождение тестов при помощи автооценивателя

5. Была реализована функция субоптимального поиска findPathToClosestDot в searchAgents.py. Код представлен в листинге 5.

Листинг 5 – Функция findPathToClosestDot

def findPathToClosestDot(self, gameState):

"""

Возвращает путь (список действий) к ближайшей точке, начиная с

gameState.

"""

# Несколько полезных элементов startState

startPosition = gameState.getPacmanPosition()

food = gameState.getFood()

walls = gameState.getWalls()

problem = AnyFoodSearchProblem(gameState)

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

return search.breadthFirstSearch(problem)

После чего было проверено разработанное решение, для чего была выполнена команда python3.6 pacman.py -l bigSearch -p ClosestDotSearchAgent -z .8. Результат продемонстрирован на рисунке 10.

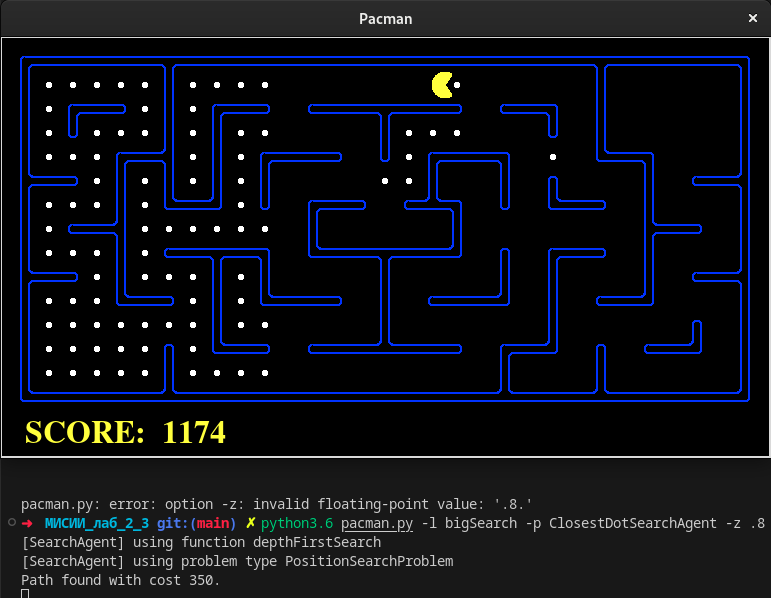


Рисунок 10 – Результат работы функции субоптимального поиска

Затем код был проверен при помощи автооценивателя. Результаты успешного прохождения тестов представлены на рисунке 11.

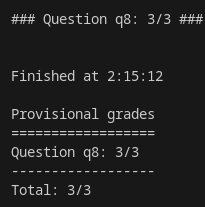


Рисунок 11 – Прохождение тестов при помощи автооценивателя

**Выводы**

В ходе выполнения лабораторной работы были исследованы информированные методов поиска решений задач в пространстве состояний, приобретение навыков программирования интеллектуальных агентов, планирующих действия на основе методов эвристического поиска решений задач, а также были написаны функции, необходимых для корректной работы агента в среде Pacman AI.