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

ОТЧЕТ

о выполнении лабораторной работы № 3 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 информированных методов поиска решений задач в пространстве состояний, приобретение навыков программирования интеллектуальных агентов, планирующих действия на основе методов эвристического поиска решений задач.

# Ход работы

## Алгоритм A\*

Для метода aStarSearch() среды AI Pacman с помощью эвристической функции манхэттенского расстояния, которая на координатной плоскости с возможностью перемещения в 4 направлениях отвечает требованиям полноты и монотонности, был реализован алгоритм A\* (листинг ‎2.1).

Листинг ‎2.1 – Алгоритм A\*

def aStarSearch(problem, heuristic=nullHeuristic):

"""

Находит узел с наименьшей комбинированной стоимостью, включающей эвристику

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

OPEN = util.PriorityQueue()

CLOSED = set({})

TOTAL\_COSTS\_AND\_ESTS = {}

def pushSuccessors(parentState, OPEN, CLOSED, TOTAL\_COSTS\_AND\_ESTS):

parentTotalCost = TOTAL\_COSTS\_AND\_ESTS[parentState][0]

children = problem.getSuccessors(parentState)

for childState, childAct, parentChildCost in children:

childCost = parentTotalCost + parentChildCost

childEstCost = childCost + heuristic(childState, problem)

if childEstCost < TOTAL\_COSTS\_AND\_ESTS.get(childState, (float('inf'), float('inf'), None, None))[1]:

CLOSED.discard(childState)

TOTAL\_COSTS\_AND\_ESTS[childState] = (childCost, childEstCost, childAct, parentState)

OPEN.update(childState, childEstCost)

def findPath(goalState, TOTAL\_COSTS\_AND\_ESTS):

goalAct = TOTAL\_COSTS\_AND\_ESTS[goalState][2]

parentState = TOTAL\_COSTS\_AND\_ESTS[goalState][3]

parent = TOTAL\_COSTS\_AND\_ESTS[parentState]

path = [goalAct]

while parent[3]:

parentAct = parent[2]

parentState = parent[3]

parent = TOTAL\_COSTS\_AND\_ESTS[parentState]

path.append(parentAct)

path.reverse()

return path

startState = problem.getStartState()

TOTAL\_COSTS\_AND\_ESTS[startState] = (0, 0, None, None)

OPEN.push(startState, 0)

while not OPEN.isEmpty():

state = OPEN.pop()

if problem.isGoalState(state):

return findPath(state, TOTAL\_COSTS\_AND\_ESTS)

CLOSED.add(state)

pushSuccessors(state, OPEN, CLOSED, TOTAL\_COSTS\_AND\_ESTS)

Результаты работы алгоритма A\* с эвристической функцией манхэттенского расстояния, для которой на координатном пространстве с возможностью перемещения в 4 направлениях выполняются условия как допустимости, так и монотонности, для лабиринта bigMaze приведены на рисунке ‎2.1.

Изображение выглядит как текст, Шрифт, алгебра, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок ‎2.1 – Результаты работы алгоритма A\* с эвристической функцией манхэттенского расстояния

В результате обследования Pacman-ом лабиринта им было раскрыто 549 вершин, что соответствует указанному в методических указаниях числу вершин. При этом полученный результат правда несколько меньше по сравнению с полученным ранее для алгоритма равных цен, в ходе работы которого было раскрыто 620 вершин.

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 ‎2.2).

Изображение выглядит как шаблон, искусство, Красочность, снимок экрана

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок ‎2.2 – Схема обследования Pacman-ом вершин лабиринта при поиске алгоритмом A\* с эвристической функцией манхэттенского расстояния

На схеме хорошо видно, как из-за движения в неоптимальном с точки зрения манхэттенского расстояния направлении Pacman сначала обследует контуры равных цен в правой части лабиринта. Тем не менее, когда Pacman выходит на участок лабиринта, при движении по которому эвристическая функция убывает, он не обследует правый участок лабиринта, и вместо этого движется напрямую к целевой вершине.

Результаты работы автооценивателя autograder.py для алгоритма A\* с эвристической функцией манхэттенского расстояния приведены на рисунке ‎2.3.

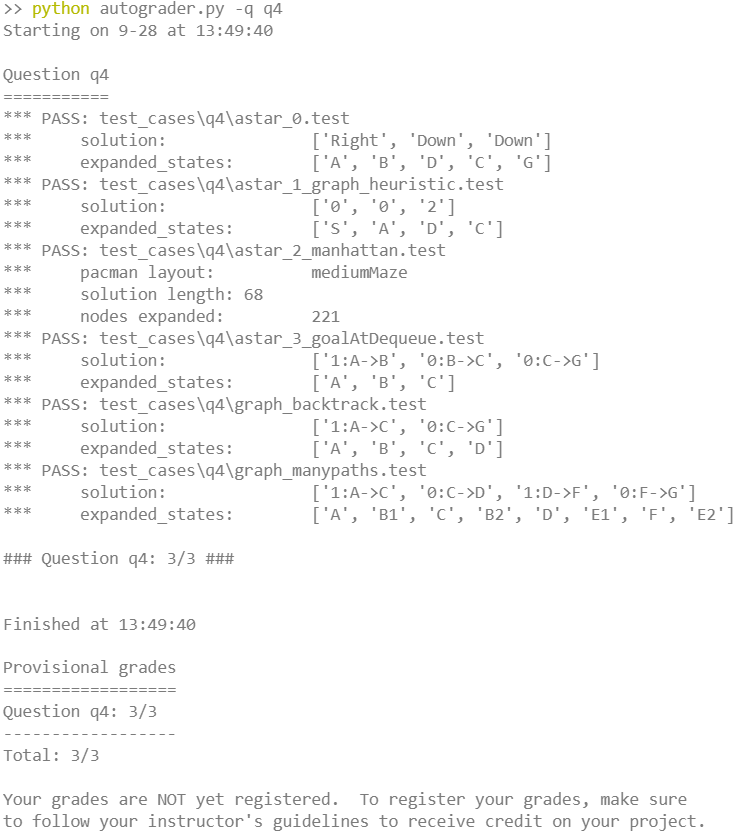


Рисунок ‎2.3 – Результаты работы автооценивателя для алгоритма A\* с эвристической функцией манхэттенского расстояния

Для алгоритма A\* временная и пространственная сложности являются экспоненциальными.

## Поиск всех углов

Для класса CornersProblem среды AI Pacman были определены конструктор \_\_init\_\_(), а также методы getStartState(), isGoalState() и getSuccessors() (листинг ‎2.2).

Листинг ‎2.2 – 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

def getStartState(self):

"""

Возвращает начальное состояние (в вашем пространстве состояний, а

не в полном пространстве состояний игры Pacman)

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

return (self.startingPosition, ())

def isGoalState(self, state):

"""

Проверяет, является ли это состояние поиска целевым состоянием задачи.

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

stateCorners = state[1]

return len(stateCorners) == 4

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.

"""

stateCrds, stateCorners = state

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 выясняет, 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 = stateCrds

dx, dy = Actions.directionToVector(action)

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

hitsWalls = self.walls[nextX][nextY]

if not hitsWalls:

nextStateCrds = (nextX, nextY)

nextStateCorners = stateCorners

if (nextX, nextY) in self.corners and (nextX, nextY) not in stateCorners:

# Create a new independent tuple based on the stateCorners tuple

nextStateCorners = stateCorners + (nextStateCrds, )

successors.append(((nextStateCrds, nextStateCorners), action, 1))

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)

В качестве состояния, используемого для задания пространства состояний, было решено использовать кортеж ((x, y), (c1, c2, … cn)), состоящий из кортежа координат местоположения агента и кортежа, содержащего кортежи координат всех посещенных им углов.

Для метода breadthFirstSearch() среды AI Pacman был доработан алгоритм поиска в ширину для обеспечения возможности работы с любым пространством состояний, а не только с кортежами координат (x, y) (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(parentState, OPEN, CLOSED, PATHS):

parentPath = PATHS[parentState]

children = problem.getSuccessors(parentState)

for childState, childAct, \_ in children:

if childState not in CLOSED and childState not in PATHS:

childPath = parentPath + [childAct]

PATHS[childState] = childPath

OPEN.push(childState)

startState = problem.getStartState()

PATHS[startState] = []

OPEN.push(startState)

while not OPEN.isEmpty():

state = OPEN.pop()

if problem.isGoalState(state):

return PATHS[state]

CLOSED.add(state)

pushSuccessors(state, OPEN, CLOSED, PATHS)

Отметим, что по сравнению с предыдущим подходом, основанном на состоянии, представленном в виде кортежа координат (x, y), теперь хранение данных как в очереди OPEN, так и в множестве CLOSED, осуществляется не на основании кортежа координат, а на основании полного состояния, включающего как кортеж координат (x, y), так и кортеж посещенных углов (c1, c2, … cn). Таким образом одна и та же вершина может неоднократно включаться в оба списка в зависимости от того, какие углы были пройдены агентом.

Фактически в ходе поиска агент сначала обнаружит оптимальный путь из своей начальной позиции к одному из углов, после чего он найдет оптимальный путь из этого угла к другому углу, и так далее, пока не будет получено целевое состояние, включающее в себя все 4 угла. Такой подход позволяет найти наикратчайший путь из стартовой вершины, проходящий через все 4 угла лабиринта.

Результаты работы алгоритма BFS для двух лабиринтов разного размера для проблемы поиска всех углов приведены на рисунке ‎2.4.

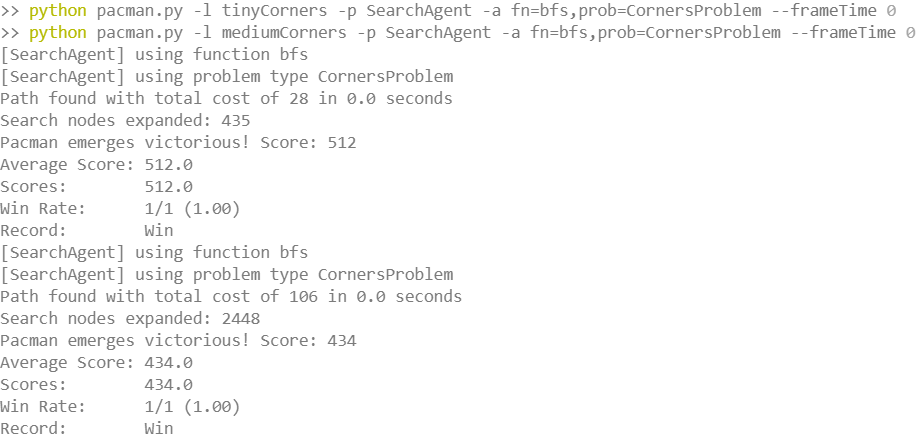


Рисунок ‎2.4 – Результаты работы алгоритма BFS для двух лабиринтов разного размера для проблемы поиска всех углов

По результатам работы алгоритма BFS, решающего задачу поиска всех углов в лабиринте tinyMaze, можно говорить о корректной реализации проблемы CornersProblem и алгоритма breadthFirstSearch() – количество раскрытых вершин, равное 28, совпадает с указанным в методических указаниях. Для лабиринта mediumMaze также получен верный результат – число раскрытых в процессе поиска вершин превышает 2000. Очевидно, что для задачи небольшой размерности такое число раскрытий достаточно велико, и необходимо прибегать к использованию эвристической функции для ускорения процесса поиска.

Результаты работы автооценивателя autograder.py для задачи поиска всех углов приведены на рисунке ‎2.5.

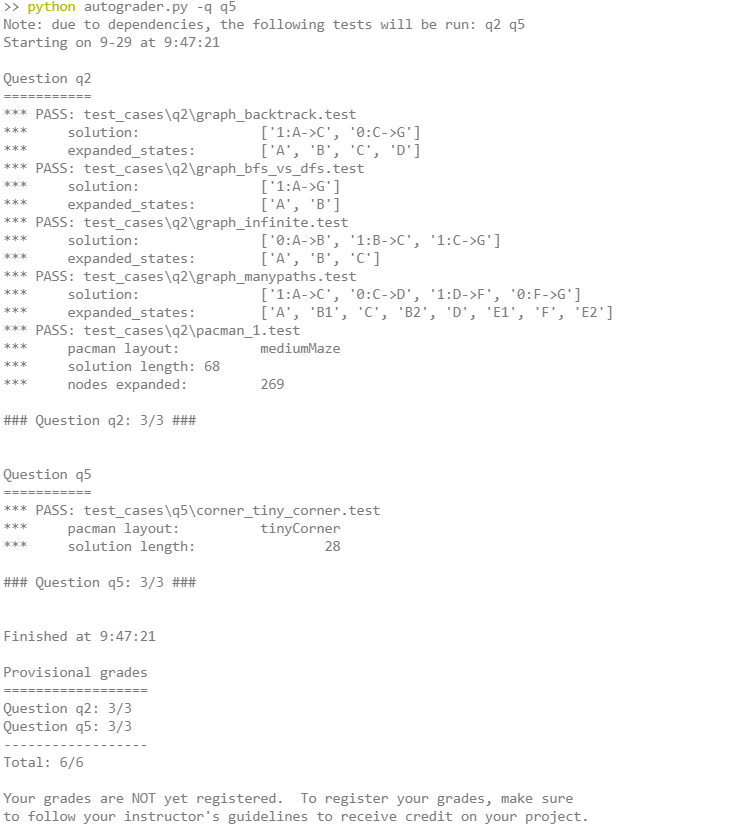


Рисунок ‎2.5 – Результаты работы автооценивателя для задачи поиска всех углов

## Определение эвристической функции

Для увеличения эффективности поиска в пространстве состояний для ранее разработанного алгоритма A\* была подобрана эвристическая функция, которая итерационно высчитывает сумму манхэттенских расстояний из текущей позиции, в которой находится Pacman, до ближайшего непройденного им угла. После расчёта манхэттенского расстояния до очередного угла этот угол отмечается как пройденный и удаляется из списка непройденных углов. Алгоритм эвристической функции приведен в листинге ‎2.4.

Листинг ‎2.4 – Алгоритм эвристической функции cornerHeuristic

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 \*\*\*"

stateCrds, stateCorners = state

x, y = stateCrds

heuristic = 0

untouchedCorners = set(corners).difference(set(stateCorners))

while untouchedCorners:

shortestDistance = float('inf')

shortestCorner = ()

for corner in untouchedCorners:

manhattanDistance = abs(corner[0] - x) + abs(corner[1] - y)

if manhattanDistance < shortestDistance:

shortestDistance = manhattanDistance

shortestCorner = corner

x, y = shortestCorner

heuristic += shortestDistance

untouchedCorners.remove(shortestCorner)

return heuristic

Результаты работы алгоритма A\*, использующего разработанную эвристическую функцию cornersHeuristic, удовлетворяющую условиям допустимости и монотонности, для лабиринта mediumCorners приведены на рисунке ‎2.6.

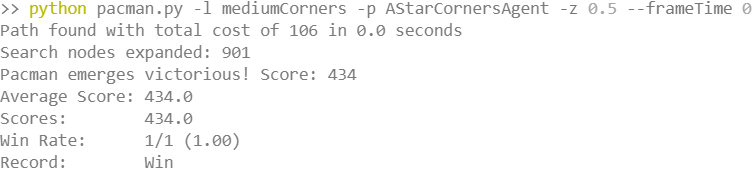


Рисунок ‎2.6 – Результаты работы алгоритма A\*, использующего разработанную эвристическую функцию, для лабиринта mediumCorners

Результаты работы автооценивателя autograder.py для алгоритма A\*, использующего разработанную эвристическую функцию cornersHeuristic, приведены на рисунке ‎2.7.

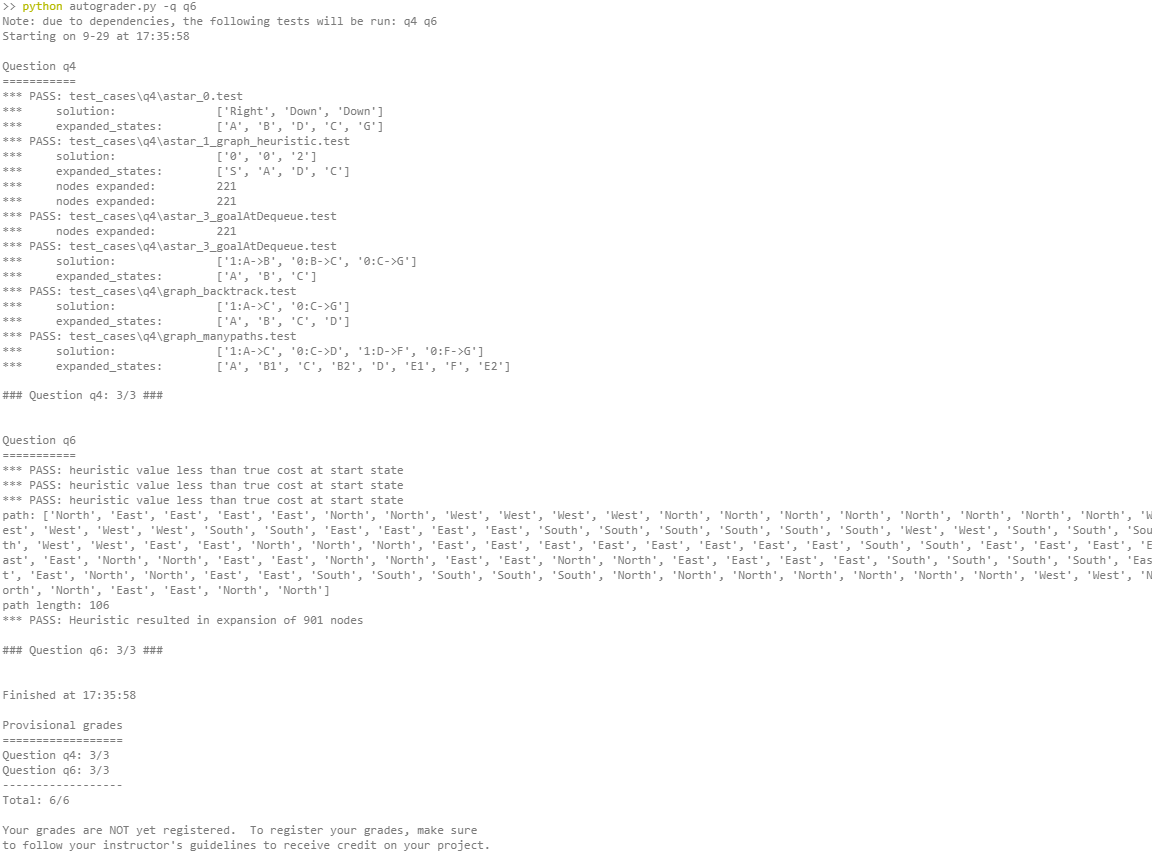


Рисунок ‎2.7 – Результаты работы автооценивателя для алгоритма A\*, использующего разработанную эвристическую функцию

Из результатов работы автооценивателя можно сделать вывод о достаточной эффективности разработанной эвристической функции, поскольку количество раскрытых вершин, равное 901, меньше максимального числа раскрытых вершин, необходимого для получения равного отличной оценки, 1200.

## Поедание всех гранул

Для задачи поедания всех гранул FoodProblem была разработана эвристическая функция, которая возвращает реальное расстояние от текущей позиции агента до наиболее удаленной позиции с едой. Расчёт расстояния производится на основании функции mazeDistance(), которая находит кратчайшее расстояние между двумя вершинами на основании алгоритма поиска в ширину BFS. Такой подход, в отличие от вычисления манхэттенского расстояния, учитывает стены лабиринта, таким образом давая более точные оценки расстояния между двумя вершинами и сокращая время поиска. Алгоритм эвристической функции приведен в листинге ‎2.5.

Листинг ‎2.5 – Алгоритм эвристической функции 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 вам нужен доступ к такой информации, как стены, гранулы и т. п.,

Вы можете обратиться problem. Например, problem.walls вернет вам Crid c

расположением стен.

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 \*\*\*"

heuristic = 0

untouchedFood = foodGrid.asList()

farthestDist = 0

for food in untouchedFood:

dist = mazeDistance(position, food, problem.startingGameState)

if farthestDist < dist:

farthestDist = dist

heuristic = farthestDist

return heuristic

Результаты работы алгоритма A\*, использующего разработанную эвристическую функцию foodHeuristic, удовлетворяющую условиям допустимости и монотонности, для лабиринта trickySearch приведены на рисунке ‎2.8.

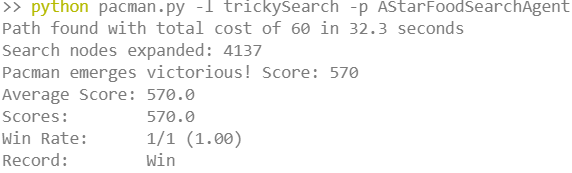


Рисунок ‎2.8 – Результаты работы алгоритма A\*, использующего разработанную эвристическую функцию, для лабиринта trickySearch

Результаты работы автооценивателя autograder.py для алгоритма A\*, использующего разработанную эвристическую функцию foodHeuristic, приведены на рисунке ‎2.9.

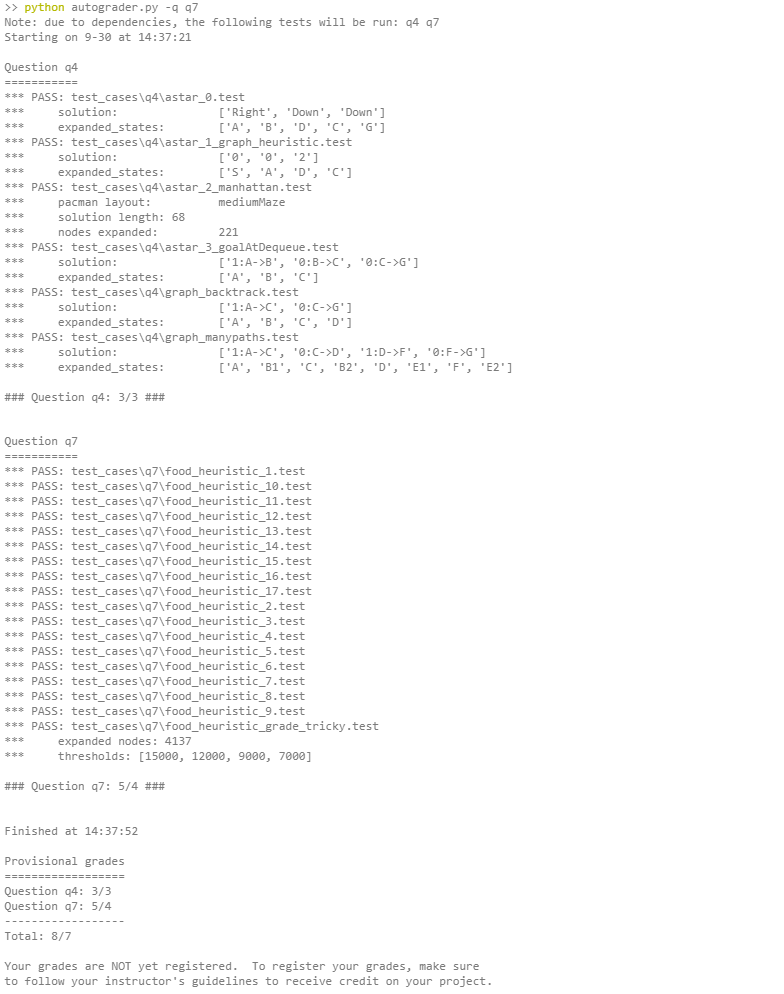


Рисунок ‎2.9 – Результаты работы автооценивателя для алгоритма A\*, использующего разработанную эвристическую функцию

Из результатов работы автооценивателя можно сделать вывод о достаточной эффективности разработанной эвристической функции, поскольку количество раскрытых вершин, равное 4137, меньше максимального числа раскрытых вершин, необходимого для получения отличной оценки, равного 7000.

## Субоптимальный поиск

Решим задачу поиска субоптимального решения, то есть задачу поиска “достаточно хорошего” пути. Для этого разработаем агента, который будет стремиться достичь целевого состояния поедания всех гранул в лабиринте посредством поедания самой близкой к нему гранулы (листинг ‎2.6).

Листинг ‎2.6 – Агент ClosestDotSearchAgent

class ClosestDotSearchAgent(SearchAgent):

" Поиск еды с помощью последовательных поисков"

def registerInitialState(self, state):

self.actions = []

currentState = state

while(currentState.getFood().count() > 0):

nextPathSegment = self.findPathToClosestDot(currentState) # The missing piece

self.actions += nextPathSegment

for action in nextPathSegment:

legal = currentState.getLegalActions()

if action not in legal:

t = (str(action), str(currentState))

raise Exception('findPathToClosestDot returned an illegal move: %s!\n%s' % t)

currentState = currentState.generateSuccessor(0, action)

self.actionIndex = 0

print('Path found with cost %d.' % len(self.actions))

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.ucs(problem)

В качестве целевого состояния выберем достижение Pacman-ом любой позиции в лабиринте, в которой содержится еда (листинг ‎2.7).

Листинг ‎2.7 – Код проблемы AnyFoodSearchProblem

class AnyFoodSearchProblem(PositionSearchProblem):

"""

Задача поиска пути к любой еде.

Эта задача поиска аналогична задаче PositionSearchProblem, но имеет другой

тест цели, который вам необходимо заполнить ниже. Пространство состояний

и функцию-преемник изменять не нужно.

Определение класса (см.выше) AnyFoodSearchProblem (PositionSearchProblem),

наследует методы PositionSearchProblem.

Вы можете использовать эту задачу поиска, для заполнения кода

метода findPathToClosestDot.

"""

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

"Хранит информацию из gameState. Вам не нужно менять этот код"

# Store the food for later reference

self.food = gameState.getFood()

# Store info for the PositionSearchProblem (no need to change this)

self.walls = gameState.getWalls()

self.startState = gameState.getPacmanPosition()

self.costFn = lambda x: 1

self.\_visited, self.\_visitedlist, self.\_expanded = {}, [], 0 # DO NOT CHANGE

def isGoalState(self, state):

"""

Состояние (state) - это позиция Pacman. Заполните кодом проверки цели

"""

x,y = state

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

return self.food[x][y]

Метод findPathToClosestDot() агента ClosestDotSearchAgent с помощью ранее разработанного алгоритма равных цен осуществляет поиск целевого состояния, соответствующего любой позиции лабиринта, где есть еда. Таким образом, агент на каждом шаге находит самую близкую позицию с едой, и завершает поиск, когда в лабиринте больше не осталось еды, то есть, когда возвращаемое выражением currentState.getFood().count() значение равно 0.

Такой подход не гарантирует нахождения оптимального решения, то есть наиболее короткого пути, проходящего через все позиции, содержащие еду. Это можно проиллюстрировать с помощью рисунка ‎2.10.

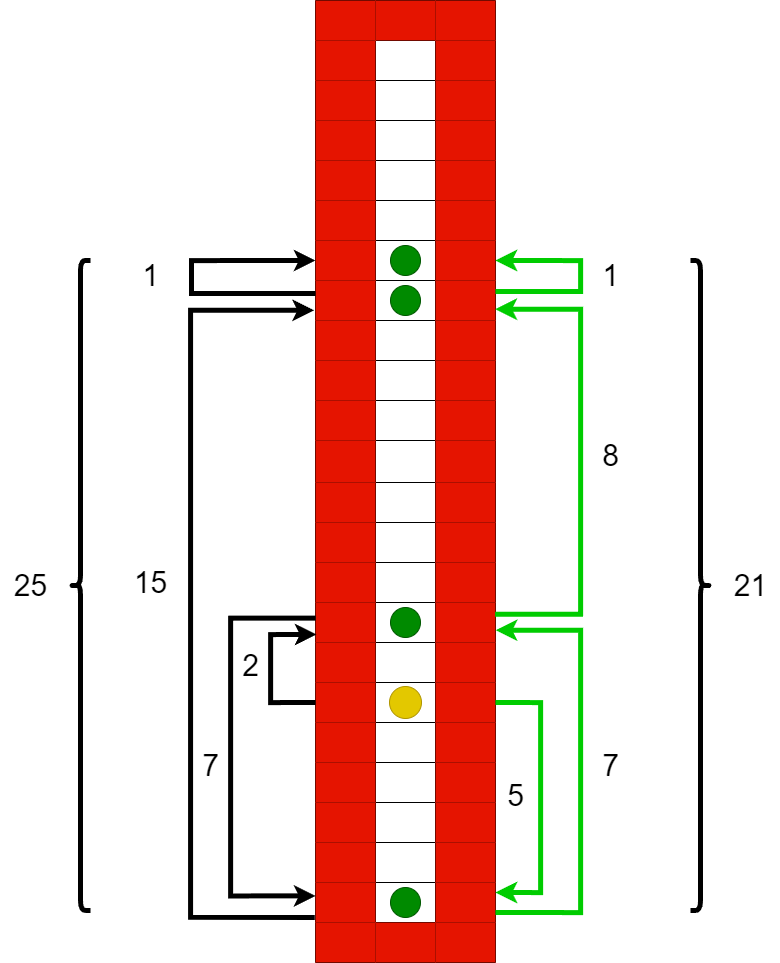


Рисунок ‎2.10 – Иллюстрация неоптимальности жадного поиска

Результаты работы алгоритма жадного поиска приведены на рисунке ‎2.8.

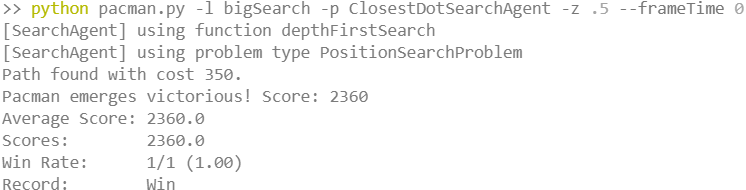
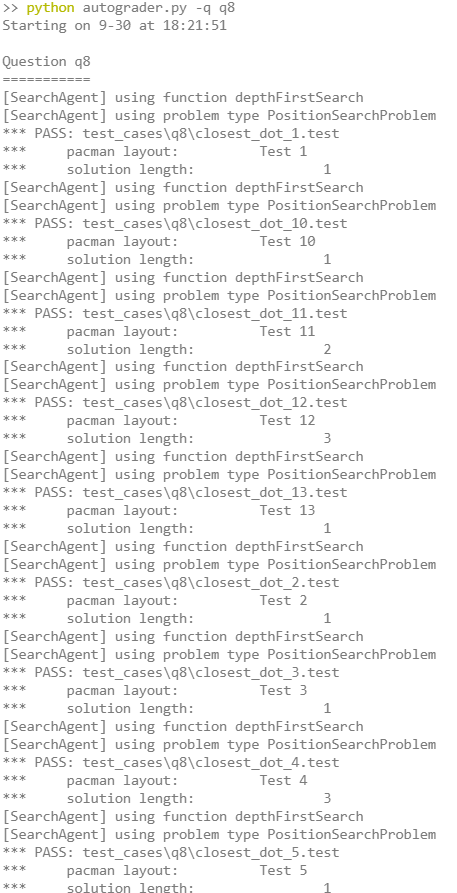


Рисунок ‎2.11 – Результаты работы алгоритма A\*, использующего разработанную эвристическую функцию, для лабиринта trickySearch

Результаты работы автооценивателя autograder.py для алгоритма жадного поиска приведены на рисунке ‎2.9.

 Изображение выглядит как текст, бумага, письмо, Шрифт

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок ‎2.12 – Результаты работы автооценивателя для алгоритма жадного поиска

На иллюстрации хорошо видно, что при жадном поиске итоговая стоимость пути равна 25, в то время как при поиске оптимальным алгоритмом итоговая стоимость пути равна 21.

# Вывод:

В результате выполнения лабораторной работы были изучены алгоритмы информированного поиска, в частности алгоритм A и его оптимальный вариант A\*. Было получено понимание об эвристических функциях и требованиях, которые к ним предъявляются для получения оптимального алгоритма: допустимости, или гарантированности, и монотонности. На основании полученных знаний была написана реализация алгоритма A\*, которая использует в качестве эвристической функции манхэттенское расстояние, которое отвечает требованиям допустимости и полноты на координатной плоскости с возможностью перемещения в 4 направлениях. Кроме того, были рассмотрены задачи поиска углов и поедания всех гранул, для которых были подобраны допустимые и монотонные эвристики, обеспечивающие нахождение оптимального решения с минимальным числом раскрытий вершин лабиринта. Также был рассмотрен субоптимальный поиск и написан алгоритм жадного поиска, который достигает целевого состояния за счет поедания ближайшей гранулы на каждой итерации.