ЛАБОРАТОРНАЯ РАБОТА №3

«ИССЛЕДОВАНИЕ ИНФОРМИРОВАННЫХ МЕТОДОВ ПОИСКА РЕШЕНИЙ ЗАДАЧ В ПРОСТРАНСТВЕ СОСТОЯНИЙ»

1. 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.

2. 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**

3. Ход работы

3.1. Был написан код алгоритма А\*-поиска в файле search.py, который представлен в листинге 1.

Листинг 1 – Код функции

def aStarSearch(problem, heuristic=nullHeuristic):

root = problem.getStartState()

visited = set()

fringe = util.PriorityQueue()

fringe.push((root, [], 0), 0)

while not fringe.isEmpty():

location, path, cost = fringe.pop()

if problem.isGoalState(location):

return path

if location not in visited:

visited.add(location)

for x,y,z in problem.getSuccessors(location):

if x not in visited:

backwardCost = z + cost

forwardCost = heuristic(x, problem)

fx = backwardCost+forwardCost

fringe.push((x, path + [y], backwardCost), fx)

return []

Правильность написанного кода была проверена с помощью автооценивания. Результаты представлены на рисунке 1.

Также работа функции была протестирована на задаче поиска пути в лабиринте с использованием эвристики манхэттенского расстояния (уже реализованную как manhattanHeuristic в searchAgents.py):

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

Результат изображен на рисунке 2.

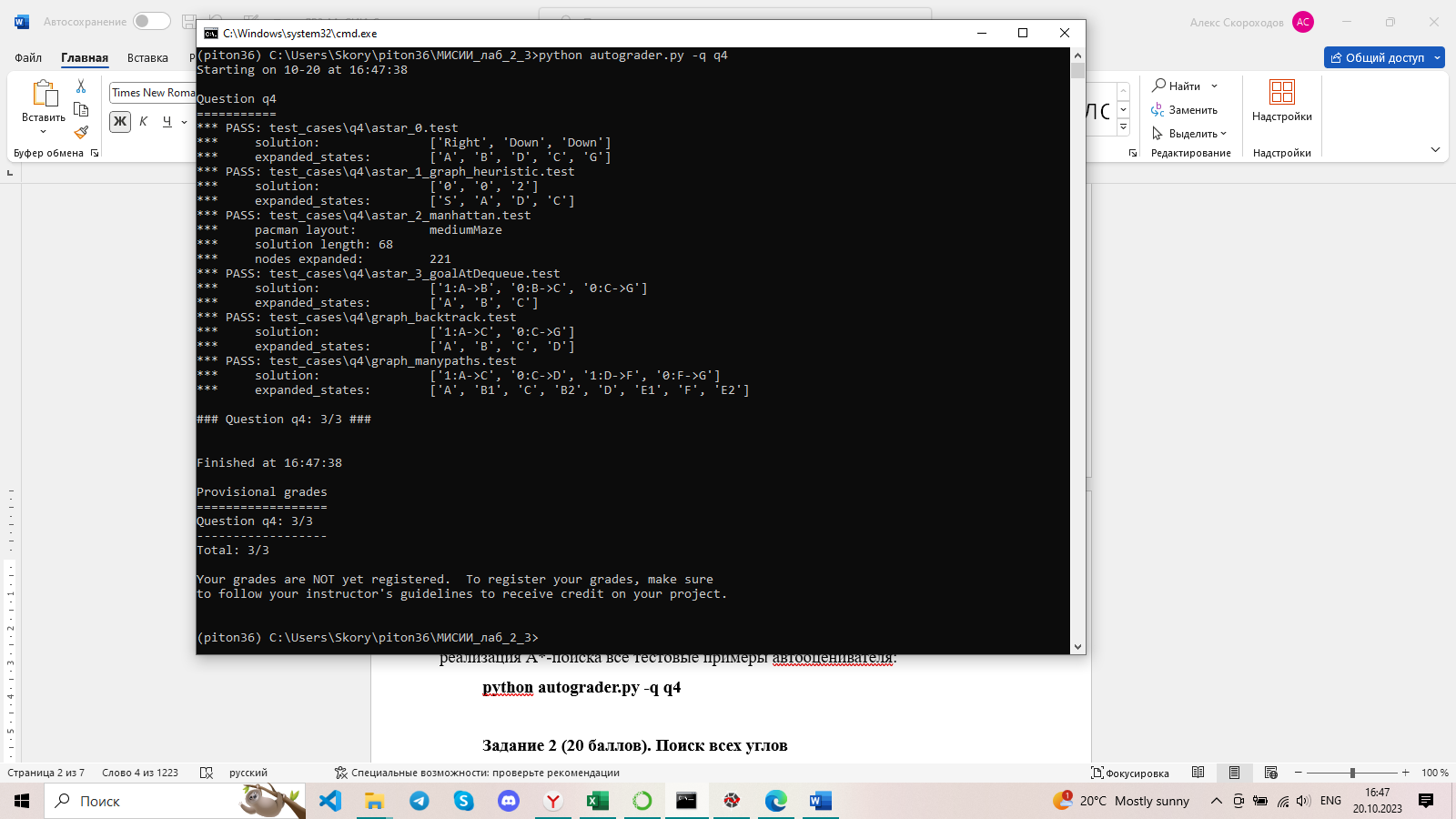


Рисунок 1 – Результаты автооценивания функции А\*-поиска (задание 1)

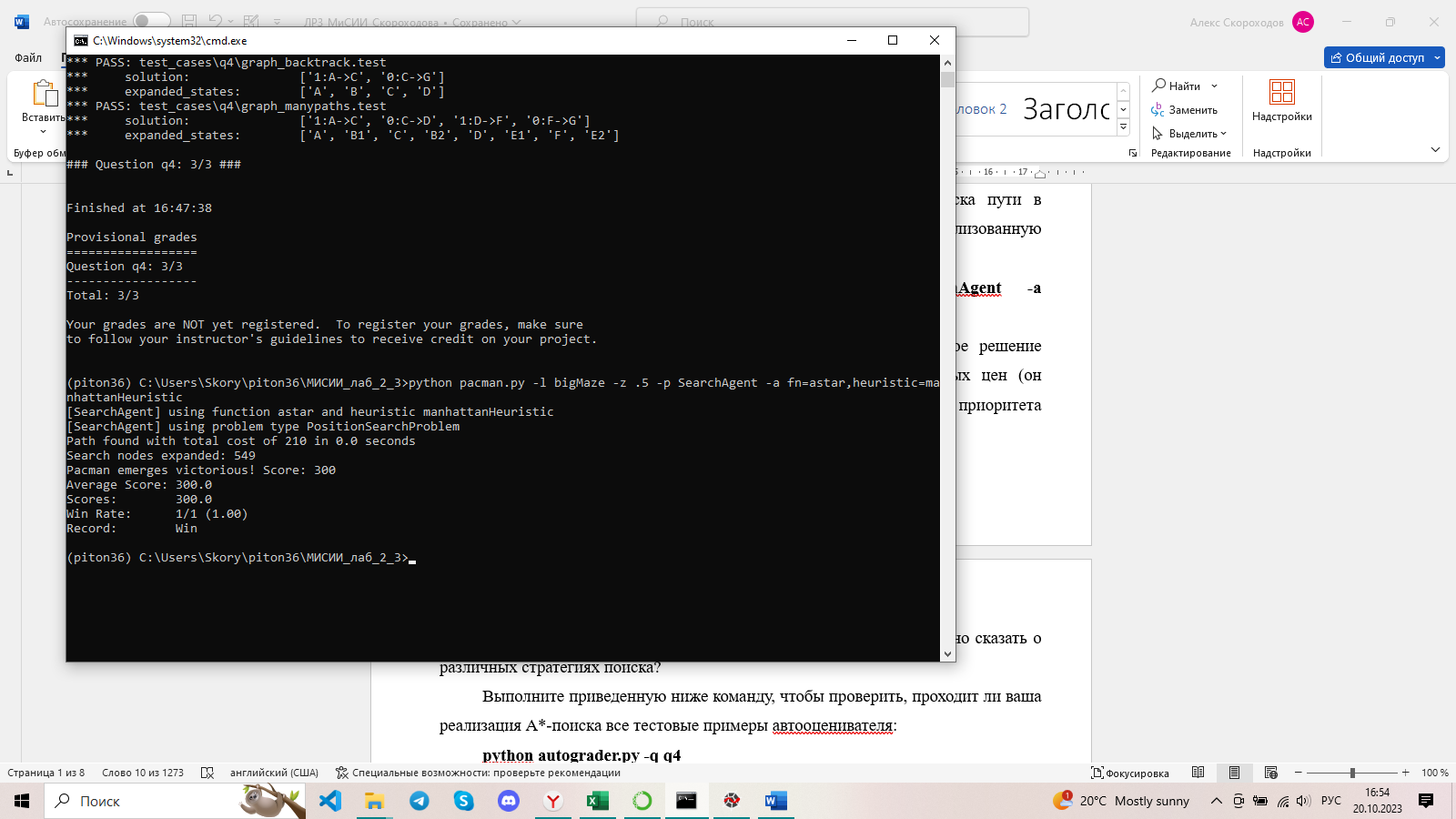


Рисунок 2 – Результаты поиска пути в лабиринте

Из рисунка 2 можно увидеть, что A\*- алгоритм раскрывает 549 узлов, как и ожидалось. Следовательно, функция поиска была написана верно.

3.2. Требовалось реализовать задачу поиска углов, дописав участки кода в определении класса CornersProblem в файле searchAgents.py. Полный код класса представлен в листинге 2.

Листинг 2 – Код класса 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 \*\*\*"

def getStartState(self):

"""

Возвращает начальное состояние (в вашем пространстве состояний, а

неполное состояние пространства игры Pacman)

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

return self.startingPosition, self.corners

def isGoalState(self, state):

"""

Проверяет, является ли это состояние поиска целевым состоянием задачи.

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

# print state[1]

curr, corners = state

# print curr

# print state

if len(corners) == 0:

return True

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]:

currPos, corners = state

dx, dy = Actions.directionToVector(action)

x, y = currPos

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

hitsWall = self.walls[nextx][nexty]

if not hitsWall:

nextPos = nextx, nexty

nextState = (nextPos, corners) if nextPos not in corners else (nextPos, tuple([i for i in corners if i != nextPos]))

cost = 1 #Given above

successors.append((nextState, 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)

С помощью команды автооценивания была протестирована работа написанного кода. Результат изображен на рисунке 3.

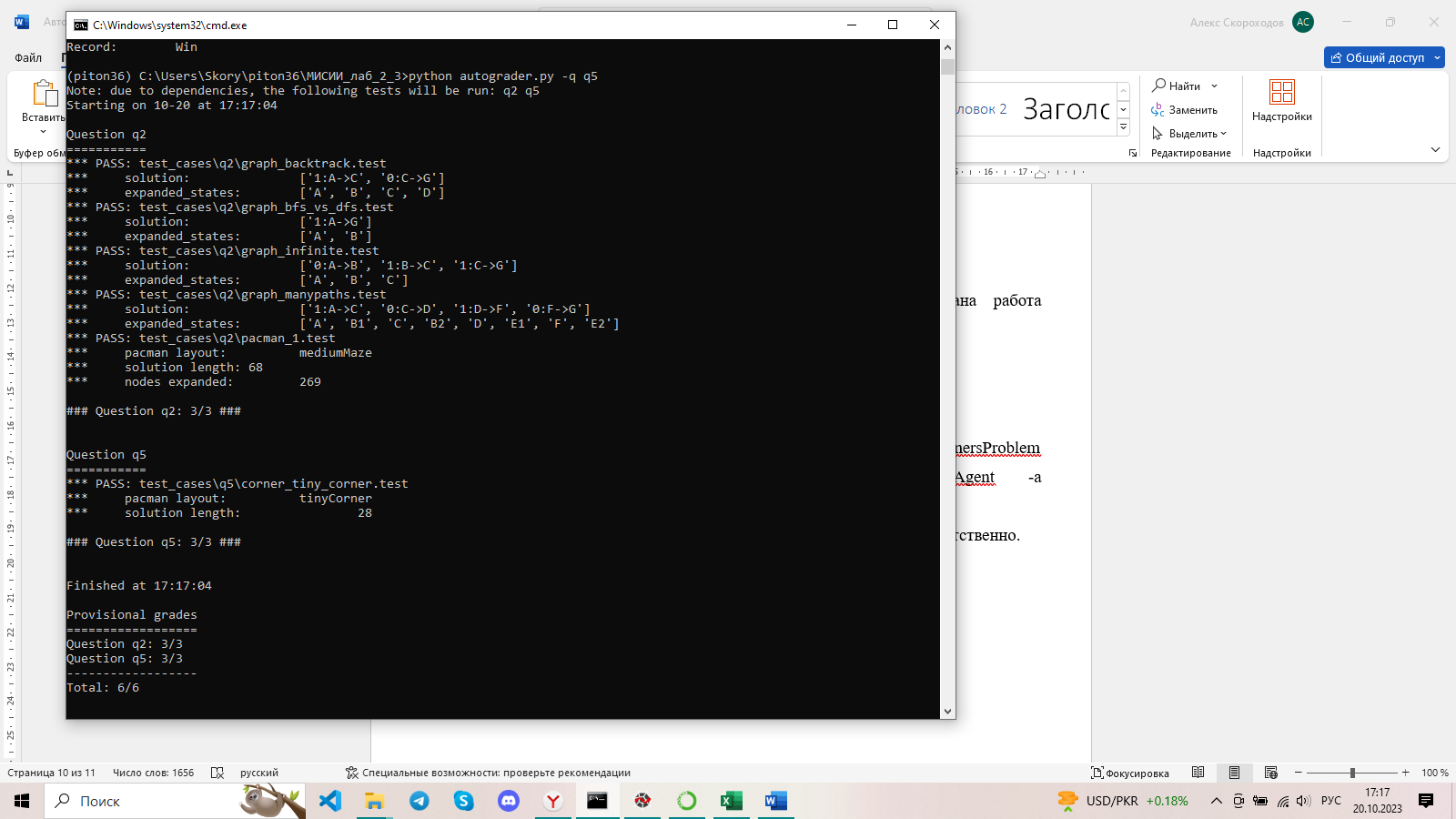


Рисунок 3 – Результаты автооценивания задачи поиска углов (задача 2)

Было произведено тестирование агента с помощью команд:

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

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

Результаты тестирования представлены на рисунках 4 и 5 соответственно.

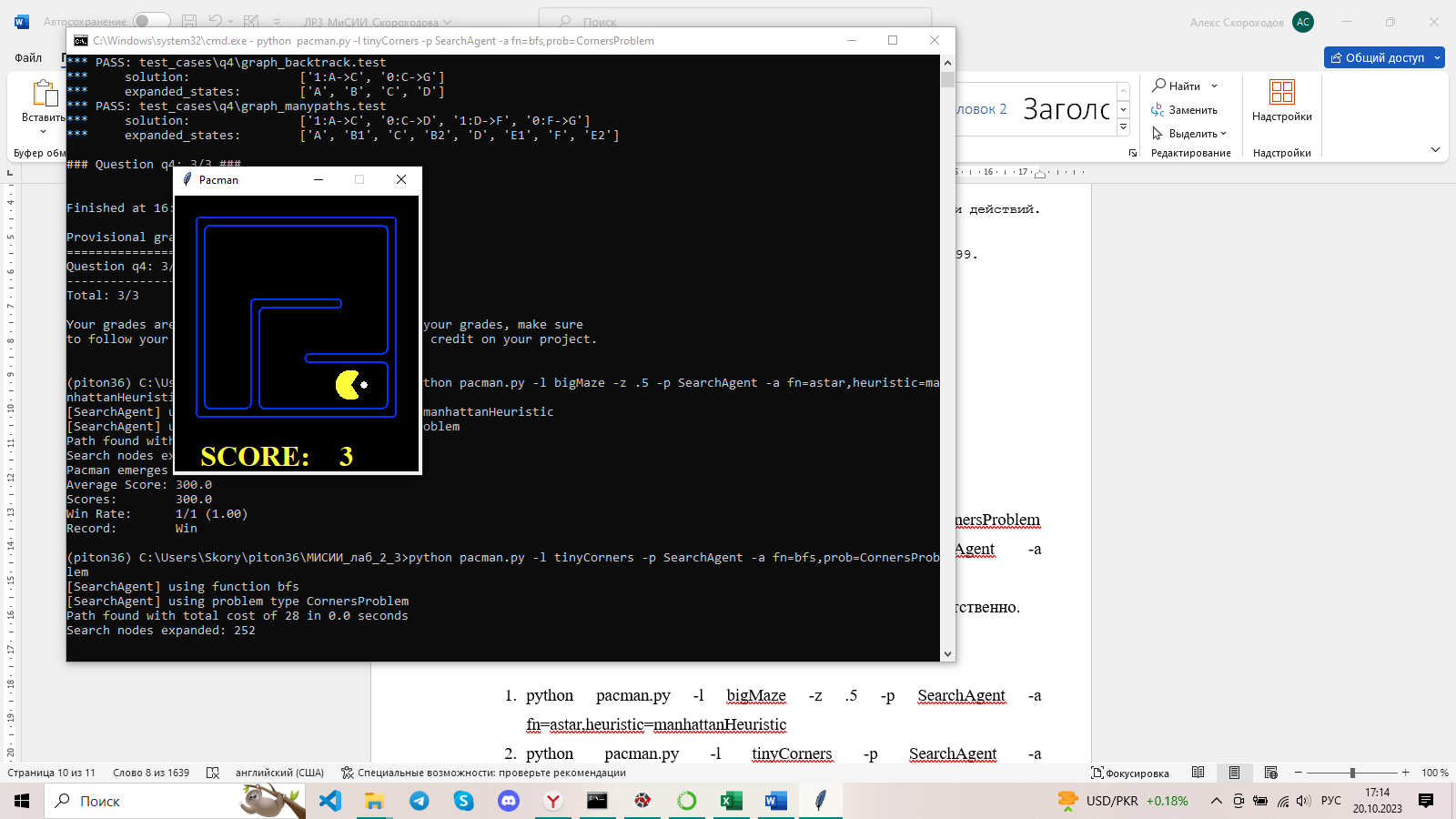


Рисунок 4 – Результат прохождения маленького лабиринта

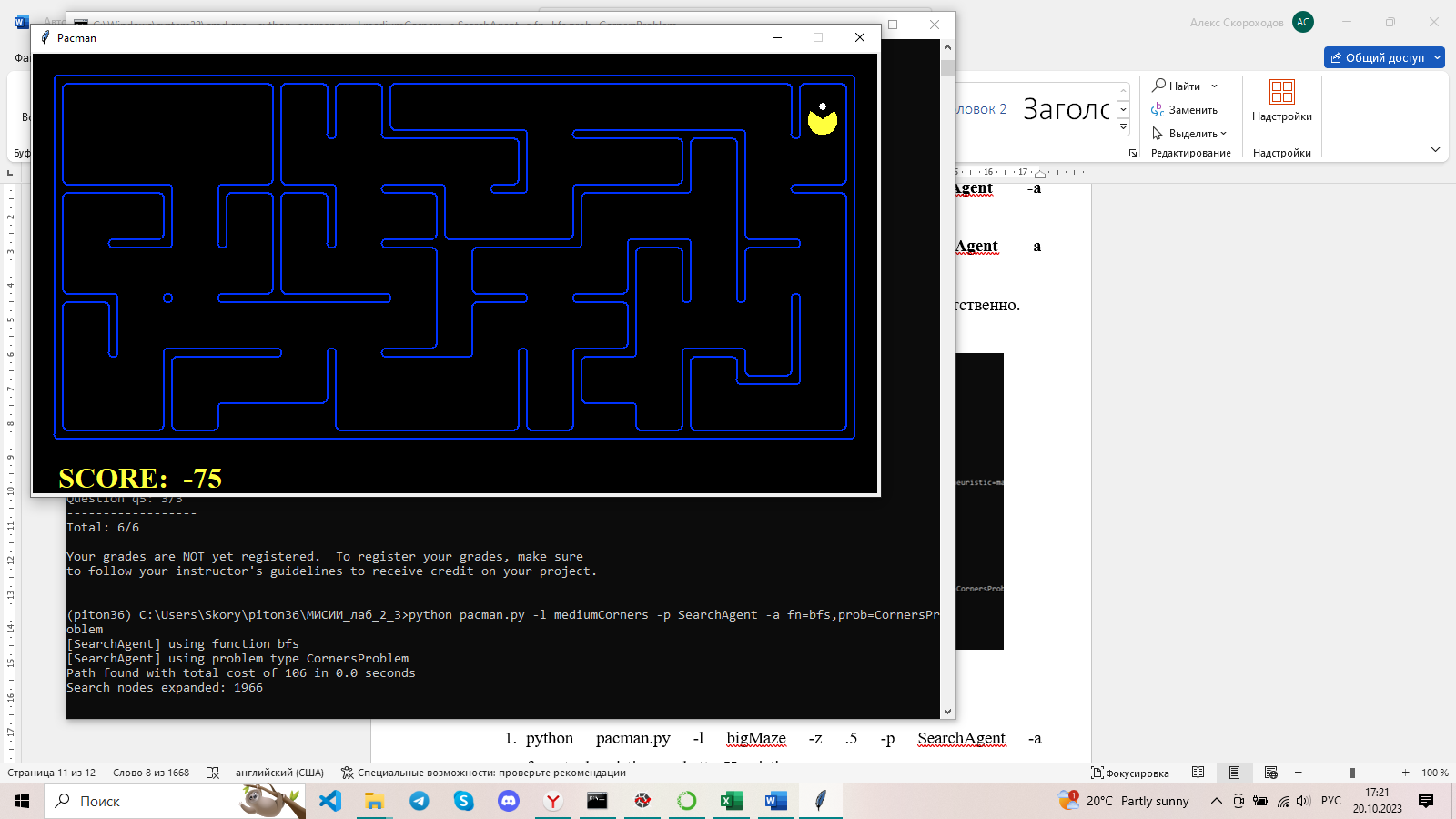


Рисунок 5 – Результат прохождения среднего лабиринта

3.3. Требовалось реализовать нетривиальную монотонную эвристику для задачи поиска углов в методе cornersHeuristic. Написанный код представлен в листинге 3.

Листинг 3 – Код cornersHeuristic

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

currPos, corners = state

cornersLeft = corners

referencePoint = currPos

while len(cornersLeft) > 0:

closestCorner = closestPoint(referencePoint, cornersLeft)

heuristic += manhattanDistance(referencePoint, closestCorner)

referencePoint = closestCorner

cornersLeft = tuple([i for i in cornersLeft if i != closestCorner])

return heuristic

Реализация была проверена с помощью команды: python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5

Результат изображен на рисунке 6.

На рисунке 7 представлен результат проверки автооцениванием.

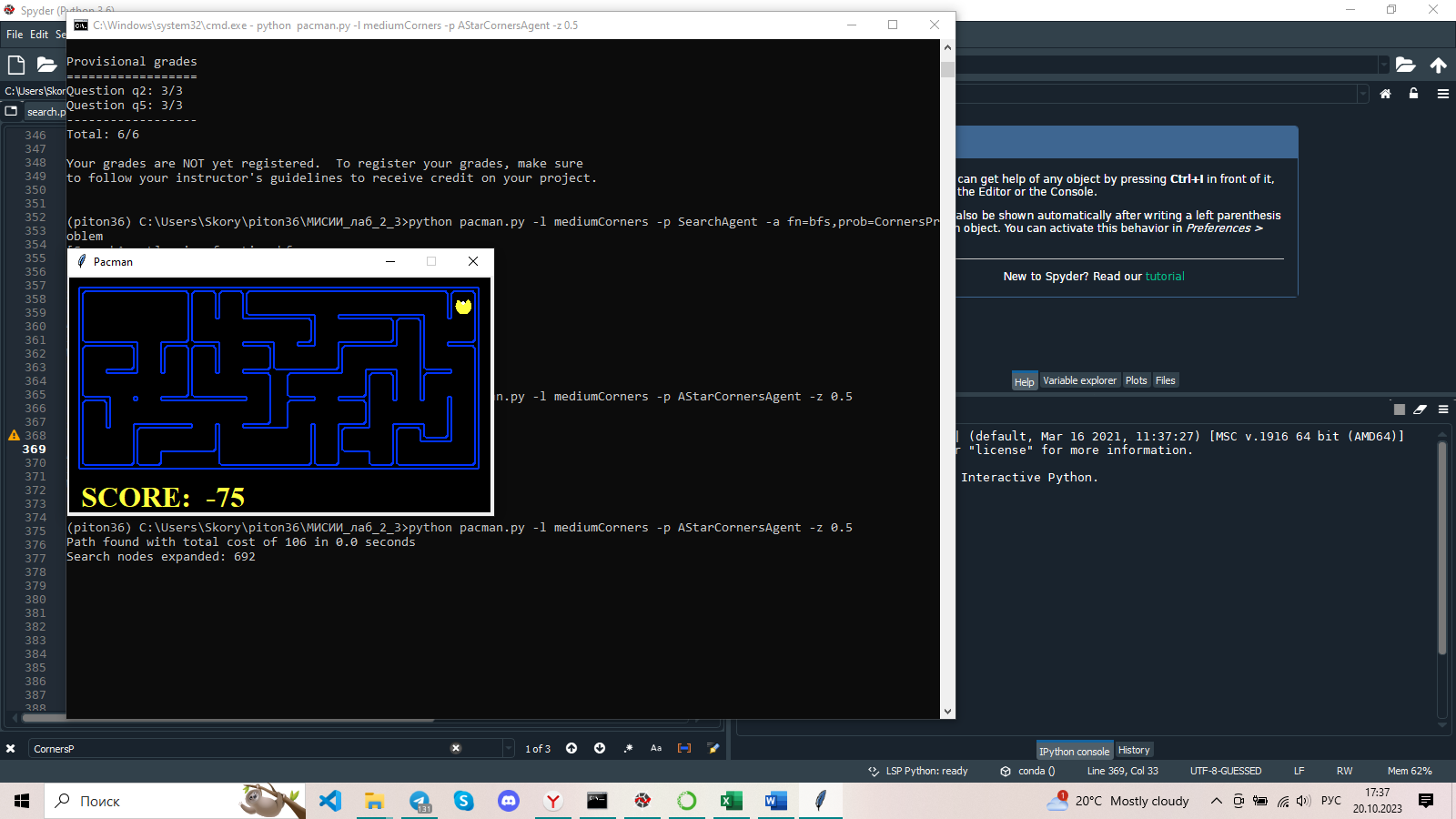


Рисунок 6 – Тестирование работы кода для cornersHeuristic (задание 3)

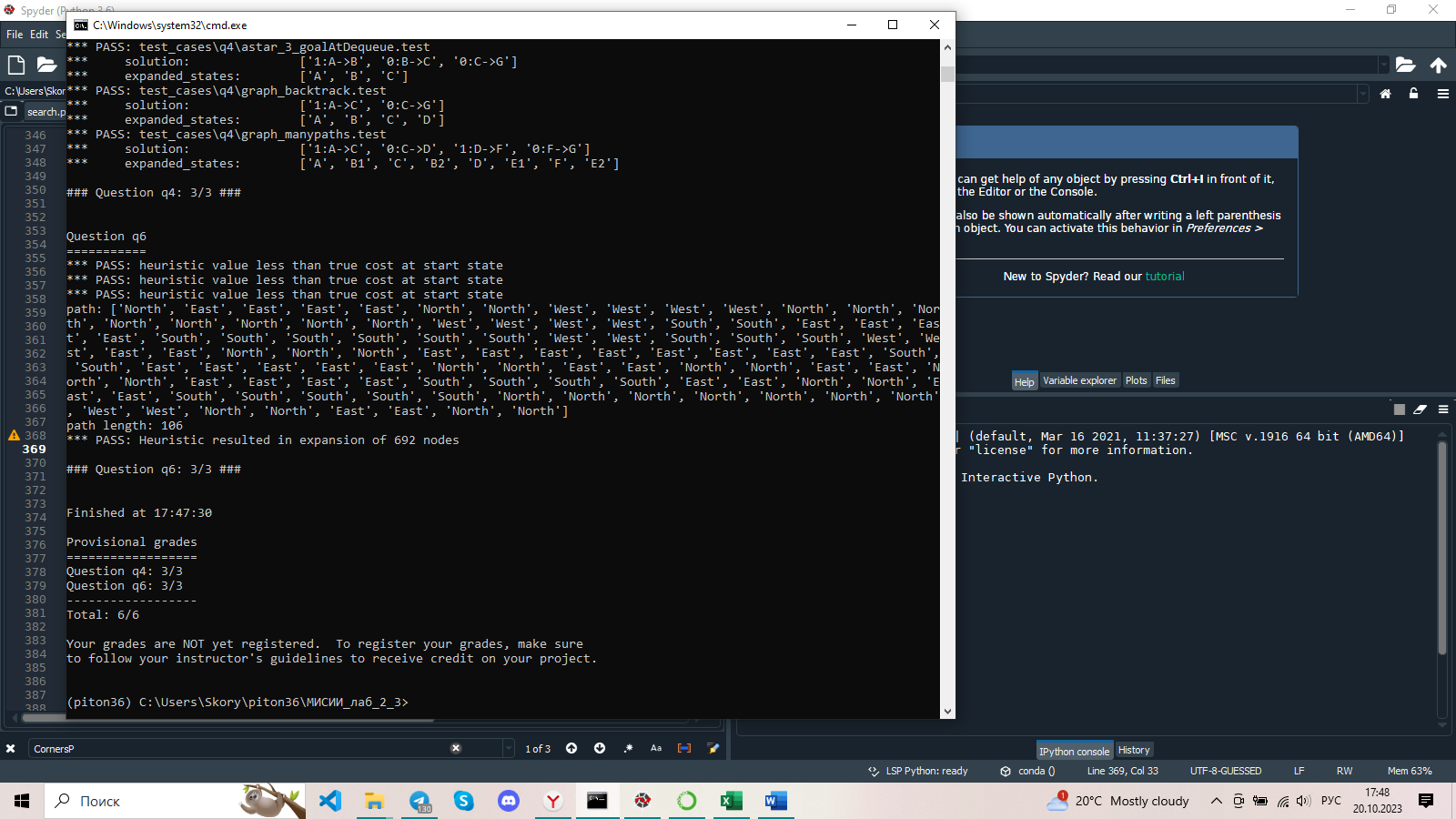


Рисунок 7 – Результат автооценивания задания 3

3.4. Требовалось дописать код в функции foodHeuristic в файле searchAgents.py, определив монотонную (согласованную) эвристику для класса FoodSearchProblem.

Разработанный код представлен в листинге 4.

Листинг 4 – Код функции foodHeuristic

def foodHeuristic(state, problem):

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

#5543 nodes

return d1 + leftPoints2

def farthestPoint (fromPoint, candidatesList):

if len(candidatesList) == 0:

return None

farthestPoint = candidatesList[0]

farthestCost = manhattanDistance(fromPoint, farthestPoint)

for candidate in candidatesList[1:]:

thisCost = manhattanDistance(fromPoint, candidate)

if thisCost > farthestCost:

farthestCost = thisCost

farthestPoint = candidate

return farthestPoint

В листинге 5 также представлены функции, необходимые для корректной работы foodHeuristic.

Листинг 5 – Прочие функции для foodHeuristic

#Used for food heuristic too

def closestPoint (fromPoint, candidatesList):

if len(candidatesList) == 0:

return None

closestPoint = candidatesList[0]

closestCost = manhattanDistance(fromPoint, closestPoint)

for candidate in candidatesList[1:]:

thisCost = manhattanDistance(fromPoint, candidate)

if thisCost < closestCost:

closestCost = thisCost

closestPoint = candidate

return closestPoint

def manhattanDistance (pointA, pointB):

return abs(pointA[0] - pointB[0]) + abs(pointA[1] - pointB[1])

def euclideanDistance (pointA, pointB):

return (abs(pointA[0] - pointB[0])\*\*2 + abs(pointA[1] - pointB[1])\*\*2)\*\*0.5

Разработанный код был протестирован с помощью команды: python pacman.py -l trickySearch -p AStarFoodSearchAgent

На рисунке 8 изображен результат тестирования функции. На рисунке 9 изображен результат автооценивания.

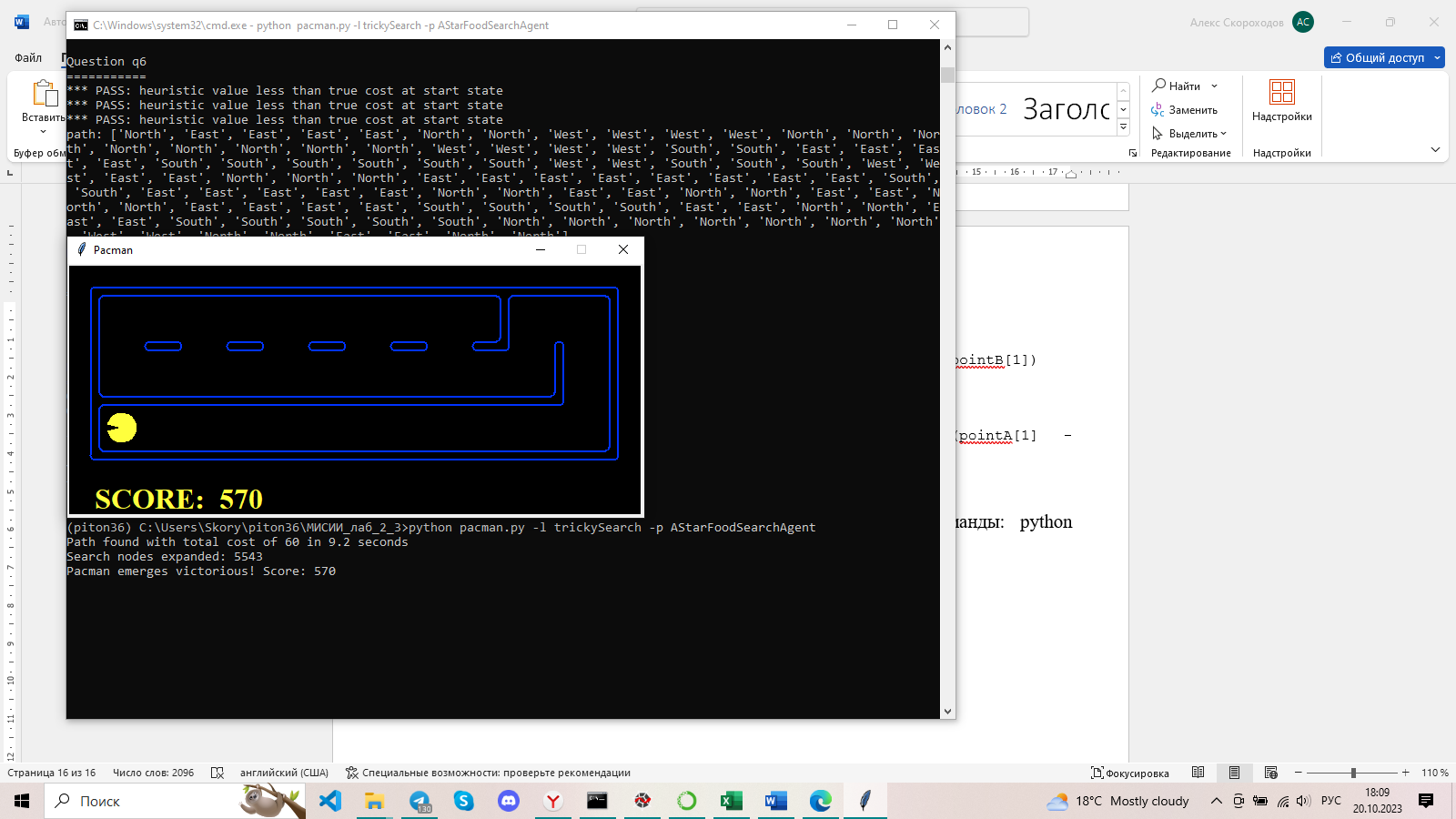


Рисунок 8 – Результат тестирования задачи поедания всех гранул (задание 4)

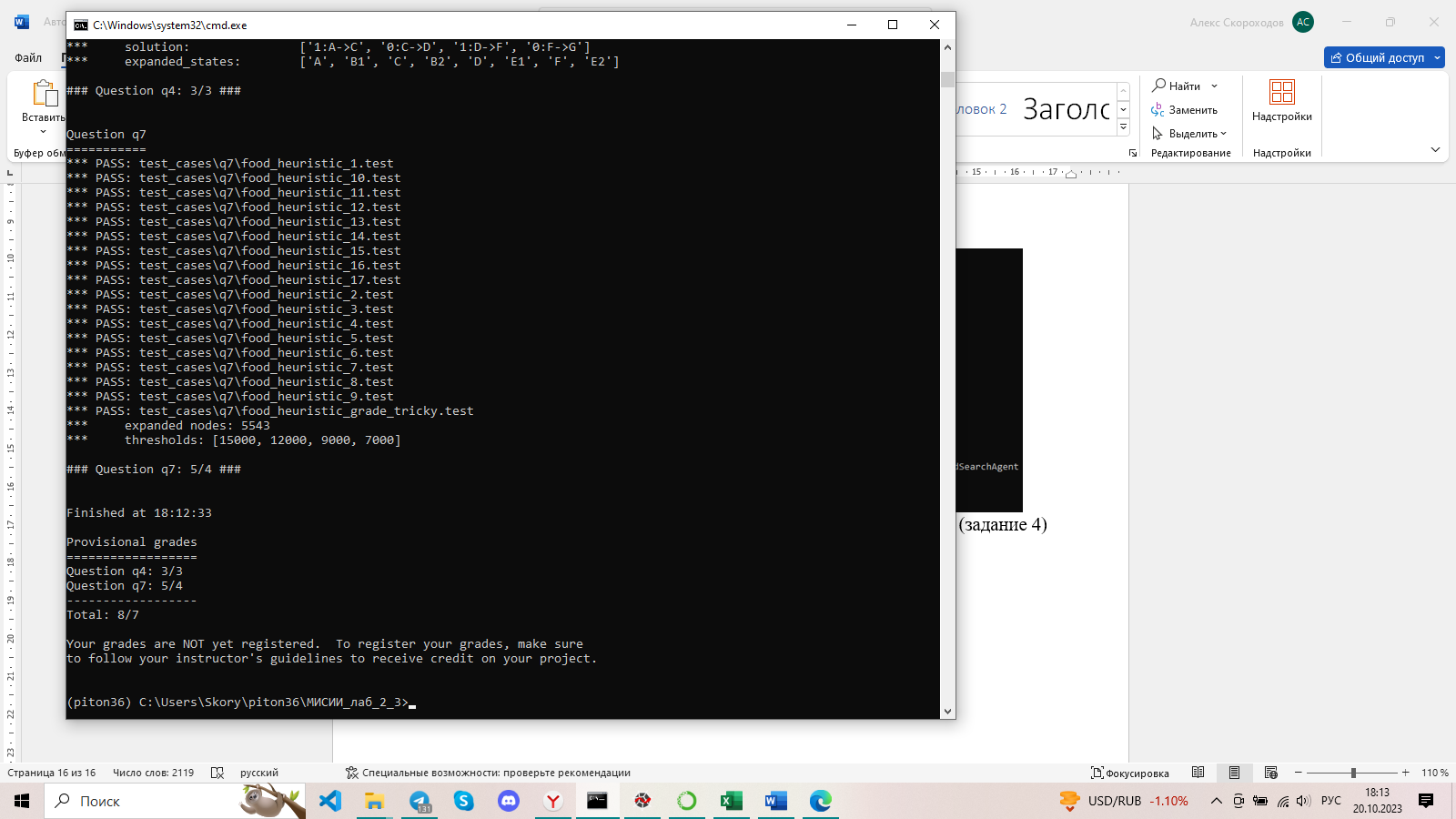


Рисунок 9 – Результат автооценивания задания 4

3.5. Требовалось реализовать функцию findPathToClosestDot в searchAgents.py. В листинге 6 представлен написанный код.

Листинг 6 – Код для реализации задания 5

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.aStarSearch(problem)

"\*\*\* YOUR CODE HERE \*\*\*"

"""foodList = food.asList()

closestFood = closestPoint(startPosition, foodList)

if closestFood != None:

prob = PositionSearchProblem(gameState, start=startPosition, goal=closestFood, warn=False)

return search.uniformCostSearch(prob)

"""

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

return self.food[x][y]

Было проведено тестирование разработанного кода с помощью команды: python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5

Результат представлен на рисунке 10. На рисунке 11 представлен результат работы автооценивания написанного кода.

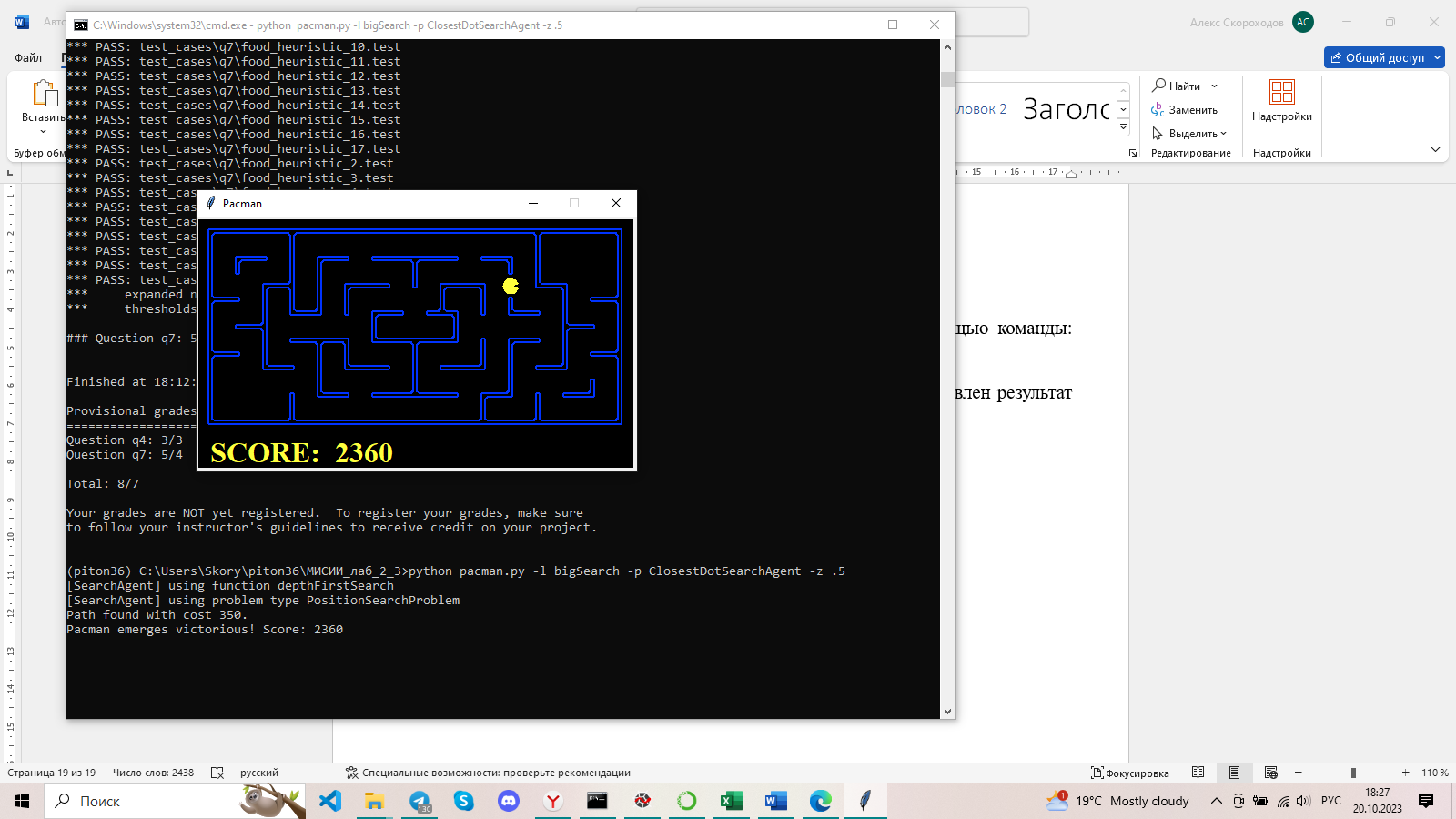


Рисунок 10 – Результат работы функции субоптимального поиска (задание 5)

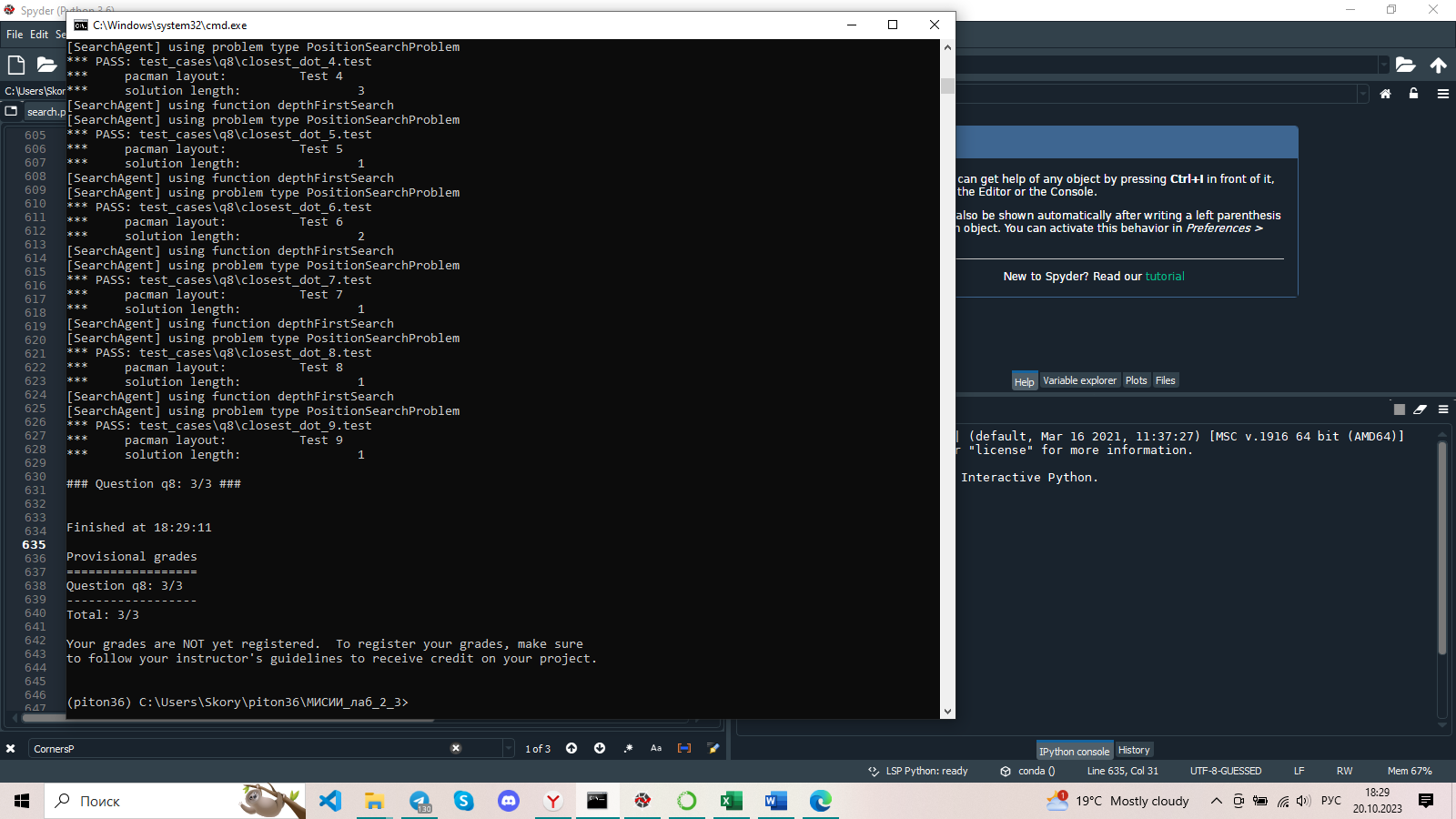


Рисунок 11 – Результат автооценивания задания 5

4. Выводы

В ходе выполнения лабораторной работы было проведено исследование информированных методов поиска решений задач в пространстве состояний посредством написания функций, необходимых для корректной работы агента в среде Pacman AI. Вследствие были приобретены навыки программирования интеллектуальных агентов, планирующих действия на основе методов эвристического поиска решений задач.

5. Контрольные вопросы

**5.1. Что называют эвристикой?**

Основная идея таких методов состоит в использовании дополнительной информации для ускорения процесса поиска. Эта дополнительная информация формируется на основе эмпирического опыта, догадок и интуиции исследователя, т.е. **эвристик**. Использование эвристик позволяет сократить количество просматриваемых вариантов при поиске решения задачи, что ведет к более быстрому достижению цели.

**5.2. Объясните основной принцип построения процедур эвристического поиска. Запишите вид оценочной функции и объясните её составляющие.**

Основной принцип построения процедур эвристического поиска заключается в использовании эвристической функции для оценки стоимости или потенциальной пользы каждого возможного шага или состояния во время поиска. Эвристическая функция предоставляет информацию о том, насколько близкое или оптимальное состояние относительно целевого состояния.

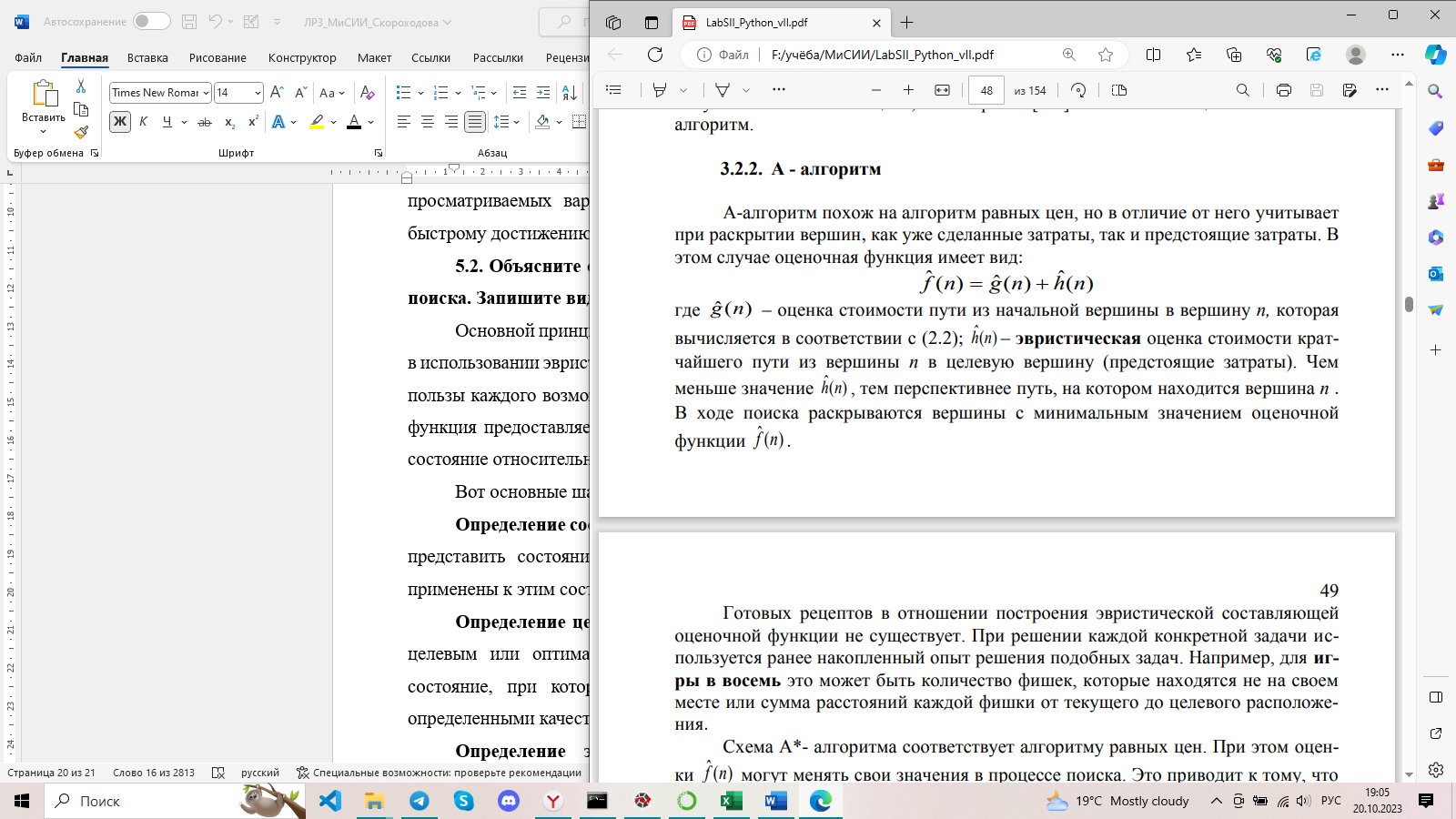
Вот основные шаги построения процедур эвристического поиска:

**Определение состояний и операторов**: Сначала необходимо определить, как представить состояния задачи и какие операторы или действия могут быть применены к этим состояниям для перехода к новым состояниям.

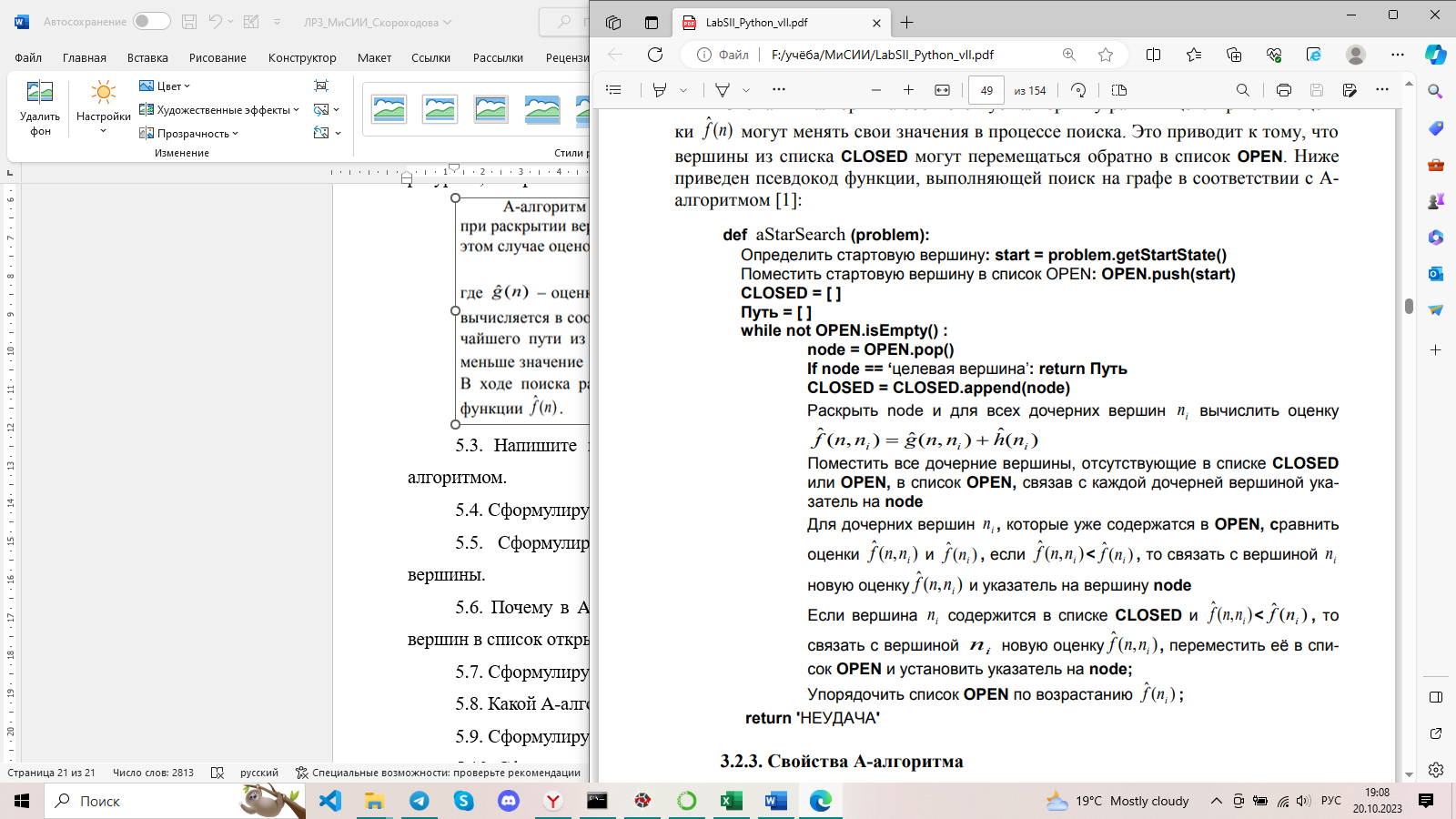
**Определение целевого состояния**: Определите, какое состояние считается целевым или оптимальным для вашей задачи. Это может быть, например, состояние, при котором задача считается решенной или которое обладает определенными качествами.

**Определение эвристической функции**: Разработайте эвристическую функцию, которая оценивает стоимость или потенциальную пользу каждого состояния относительно целевого состояния. Эта функция может использовать различные эвристические оценки, такие как эвристику расстояния или предположения о допустимости действий.

**Выбор стратегии поиска**: Определите стратегию поиска, которая будет использоваться с эвристической функцией. Некоторые из распространенных стратегий поиска включают алгоритм A\*, алгоритмы с ограничением времени или ресурсов, алгоритмы с итеративным углублением и т. д.



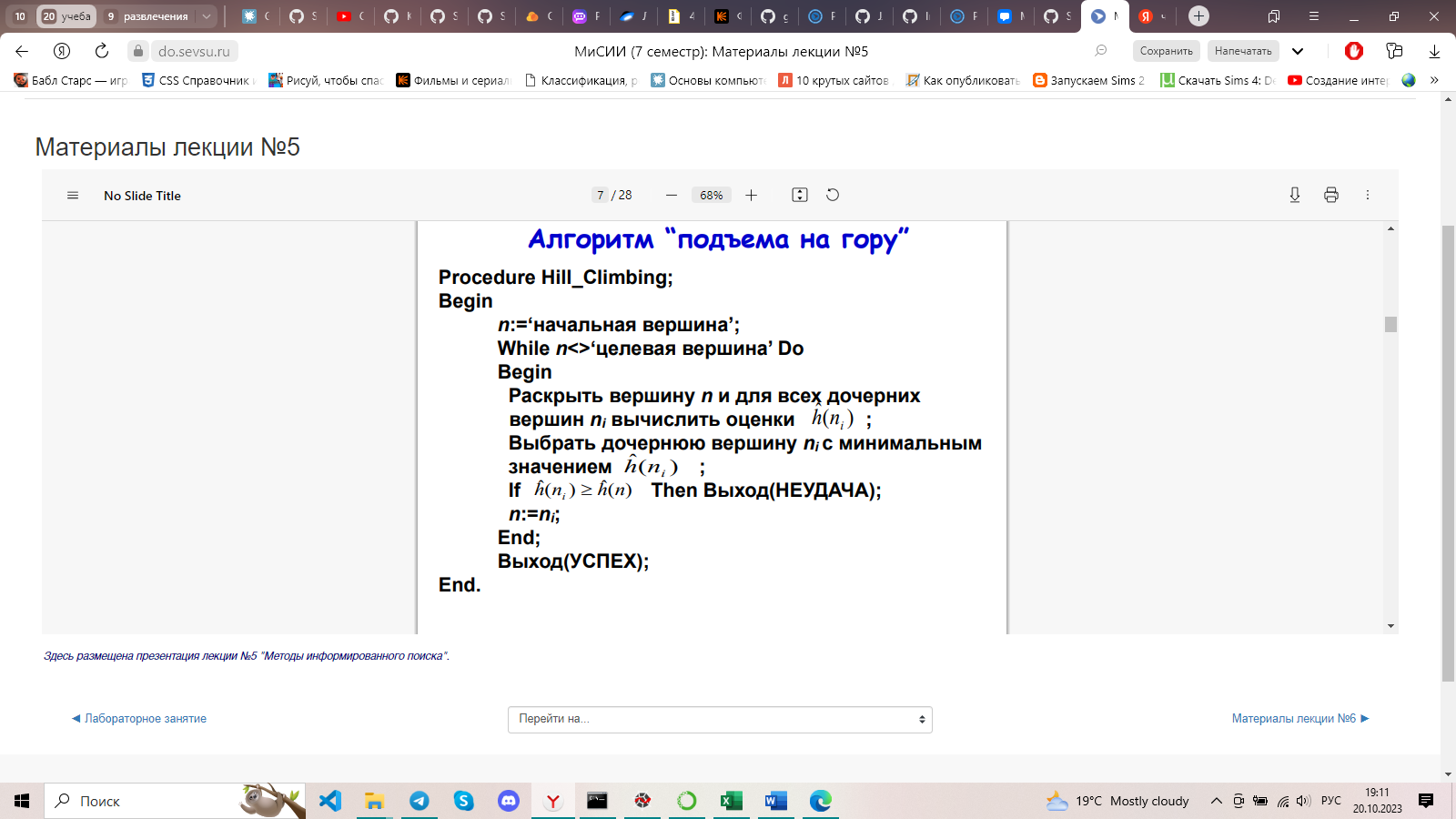
**5.3. Напишите на псевдоязыке процедуру поиска в соответствии с А\*-алгоритмом.**



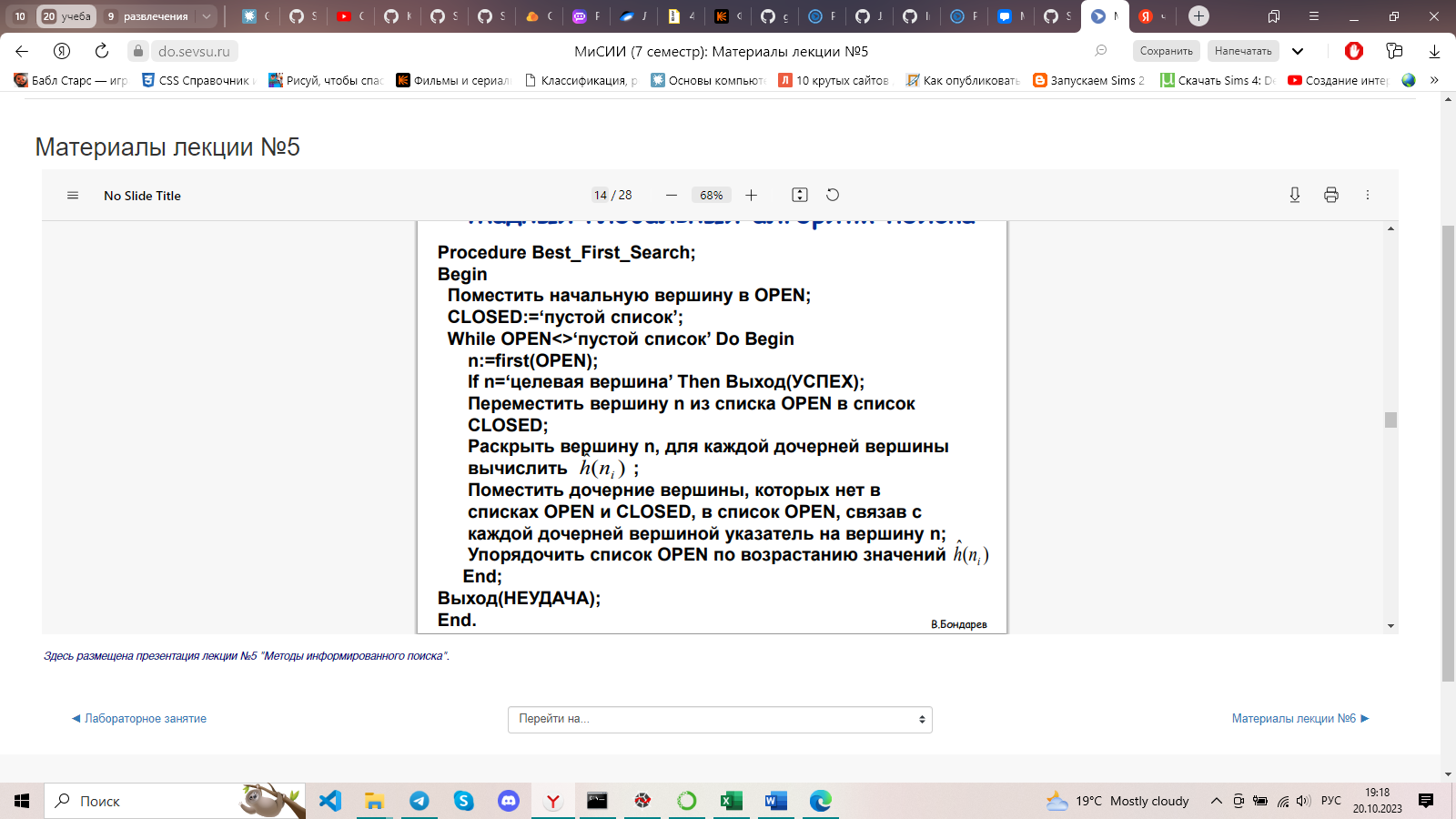
**5.4. Сформулируйте алгоритм подъема в гору.**

Алгоритм осуществляет целенаправленный поиск в направлении наибольшего убывания эвристической оценочной функции. Данная функция обеспечивает оценку (прогноз) стоимости кратчайшего пути от текущей вершины n до ближайшей целевой вершины, т.е. является мерой стоимости оставшегося пути. Чем меньше значение этой функции, тем перспективнее путь, на +котором находится вершина n.

Подобный алгоритм используется при поиске экстремумов функции. Поиск экстремума осуществляют в направлении наибольшего возрастания (убывания) градиента. Поиск максимума функции в этом случае напоминает восхождение к вершине по наиболее крутому маршруту.



**5.5. Сформулируйте алгоритм глобального выбора первой наилучшей вершины.**



Алгоритм глобального выбора первой наилучшей вершины (Global Best-First Search) - это алгоритм поиска в графе, который выбирает следующую вершину для исследования на основе эвристической функции, предоставляющей оценку "качества" вершины. Алгоритм стремится найти наилучшую вершину, которая имеет наибольшую эвристическую оценку среди всех доступных вершин.

Вот формулировка алгоритма глобального выбора первой наилучшей вершины:

1. Создайте пустой список открытых вершин.

2. Поместите начальную вершину в список открытых вершин.

3. Пока список открытых вершин не пуст:

Выберите вершину V с наилучшей эвристической оценкой из списка открытых вершин.

Если V является целевой вершиной, то задача решена. Верните найденный путь или выполните требуемые действия. Иначе, перейдите к следующему шагу.

4. Расширьте вершину V путем применения операторов или действий, доступных из V, для создания новых вершин.

5. Для каждой новой вершины N:

Вычислите эвристическую оценку для N.

Если N уже присутствует в списке открытых вершин с меньшей эвристической оценкой, пропустите N. Иначе, добавьте N в список открытых вершин и установите V как его родителя.

6. Повторите шаги 3-5.

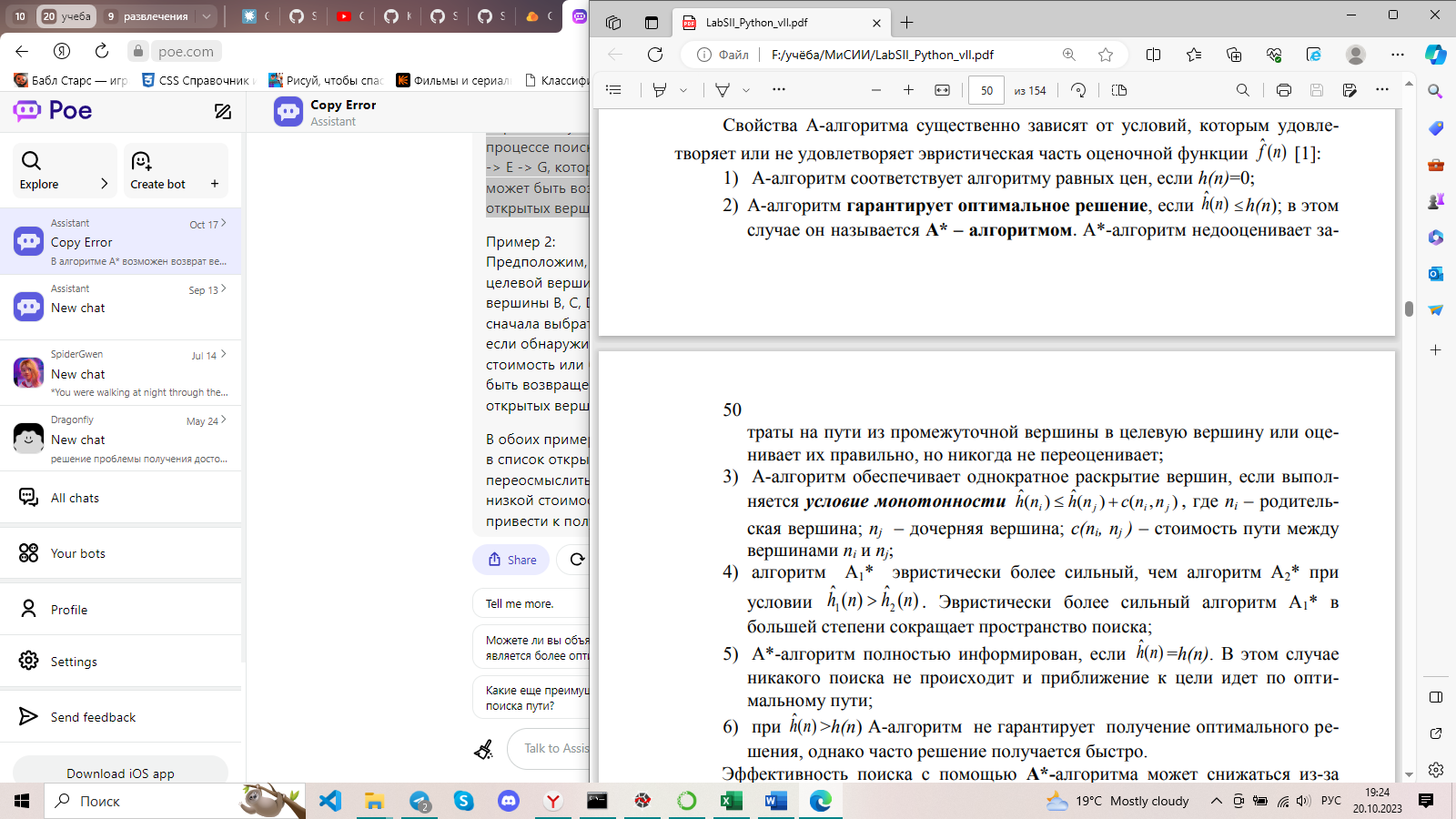
**5.6. Почему в А-алгоритме возможен возврат вершин из списка закрытых вершин в список открытых вершин? Приведите примеры.**

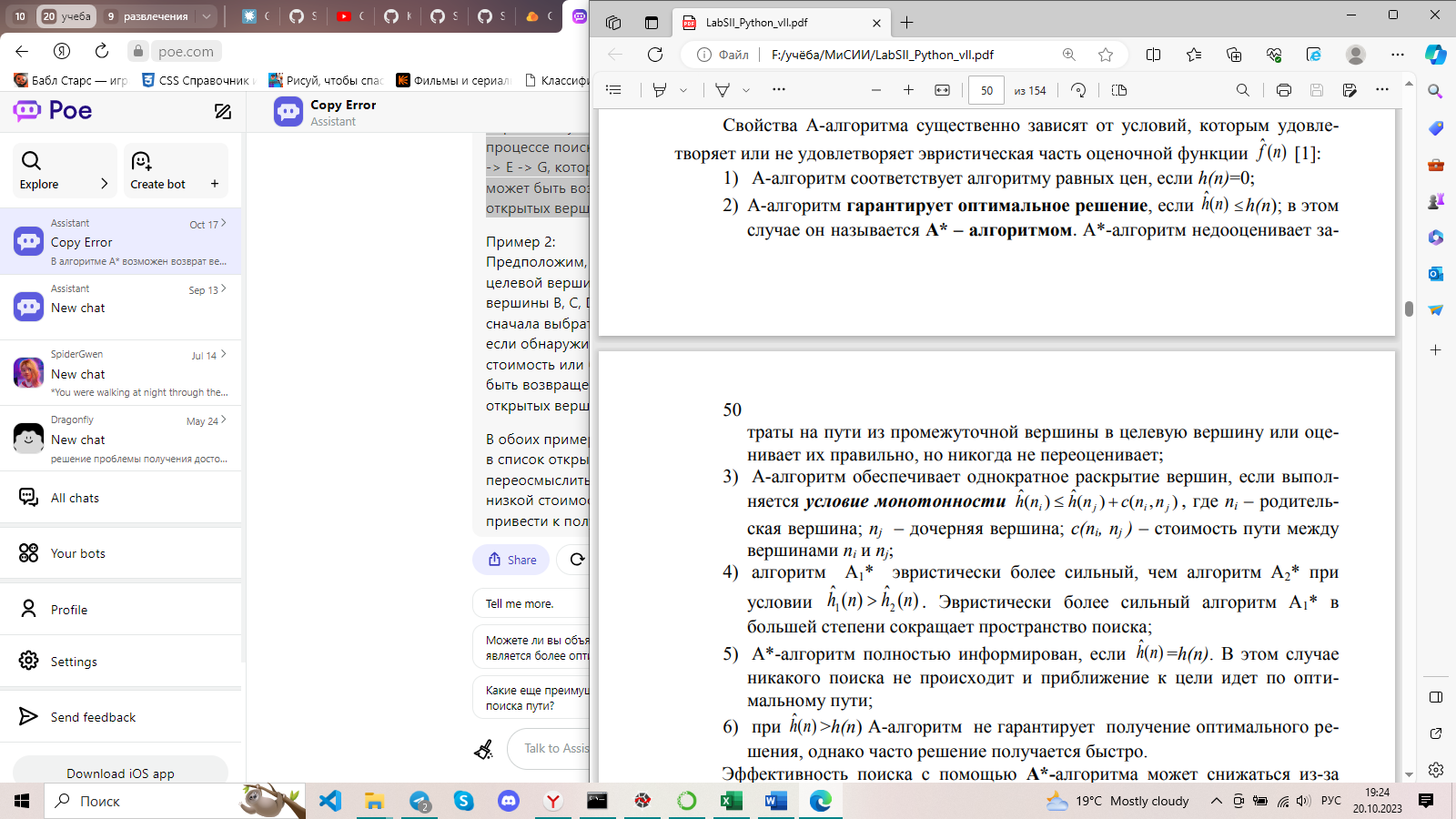
В алгоритме A\* возможен возврат вершин из списка закрытых вершин в список открытых вершин в случае, когда обнаруживается более оптимальный путь к этой вершине. Это происходит, когда новый путь к вершине предлагает меньшую стоимость или более оптимальное решение по сравнению с уже известным путем.

Пример 1:

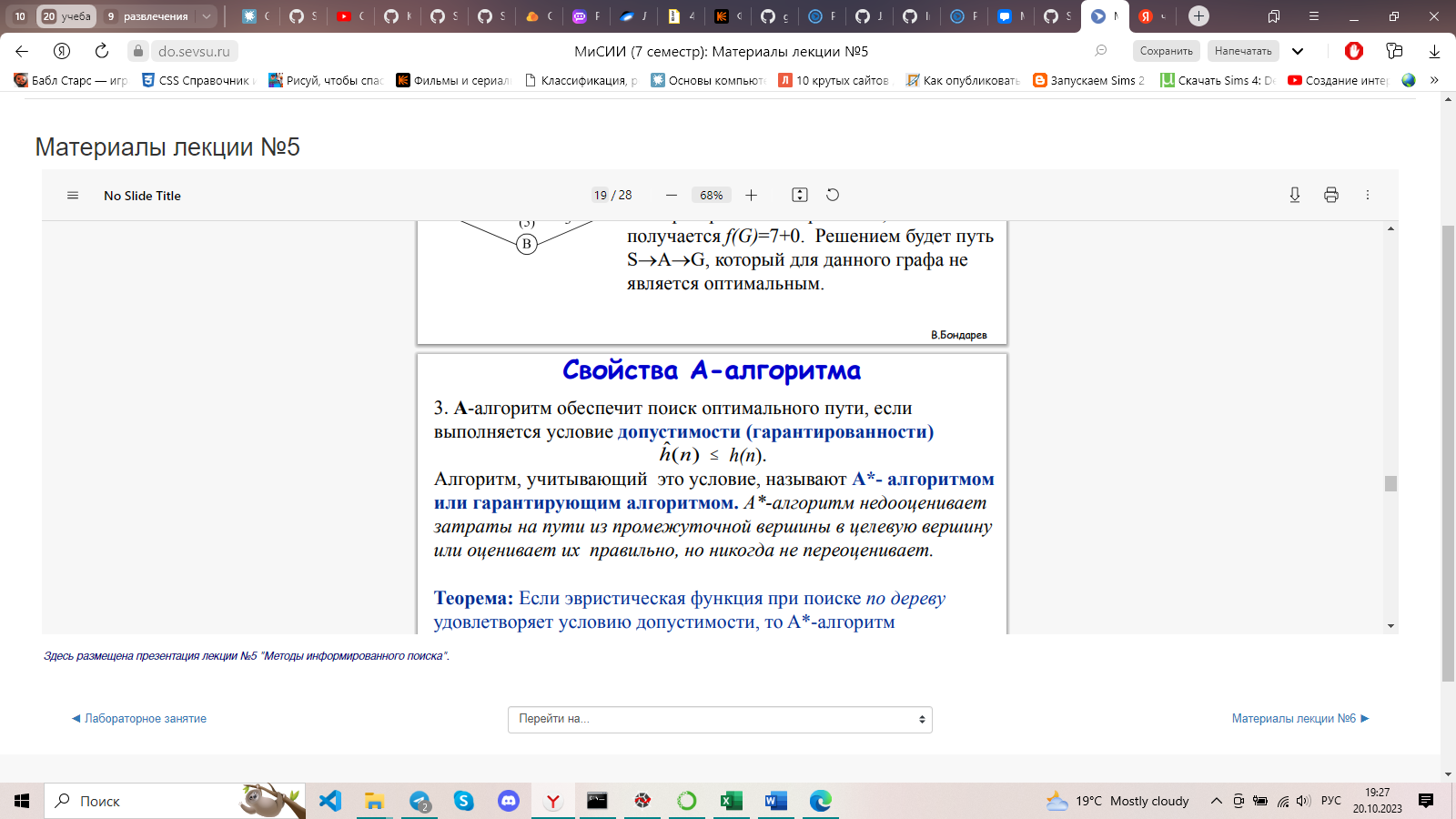
Предположим, у нас есть граф с начальной вершиной A и целевой вершиной G. Путь от A до G может проходить через вершины B, C, D и E. При первом проходе алгоритм A\* может выбрать путь A -> B -> C -> D -> E -> G, который имеет определенную стоимость. Однако, если в дальнейшем в процессе поиска будет обнаружен другой путь A -> B -> X -> Y -> E -> G, который имеет меньшую стоимость, вершина E может быть возвращена из списка закрытых вершин в список открытых вершин, чтобы рассмотреть новый путь.

**5.7. Сформулируйте и объясните свойства А-алгоритма.**

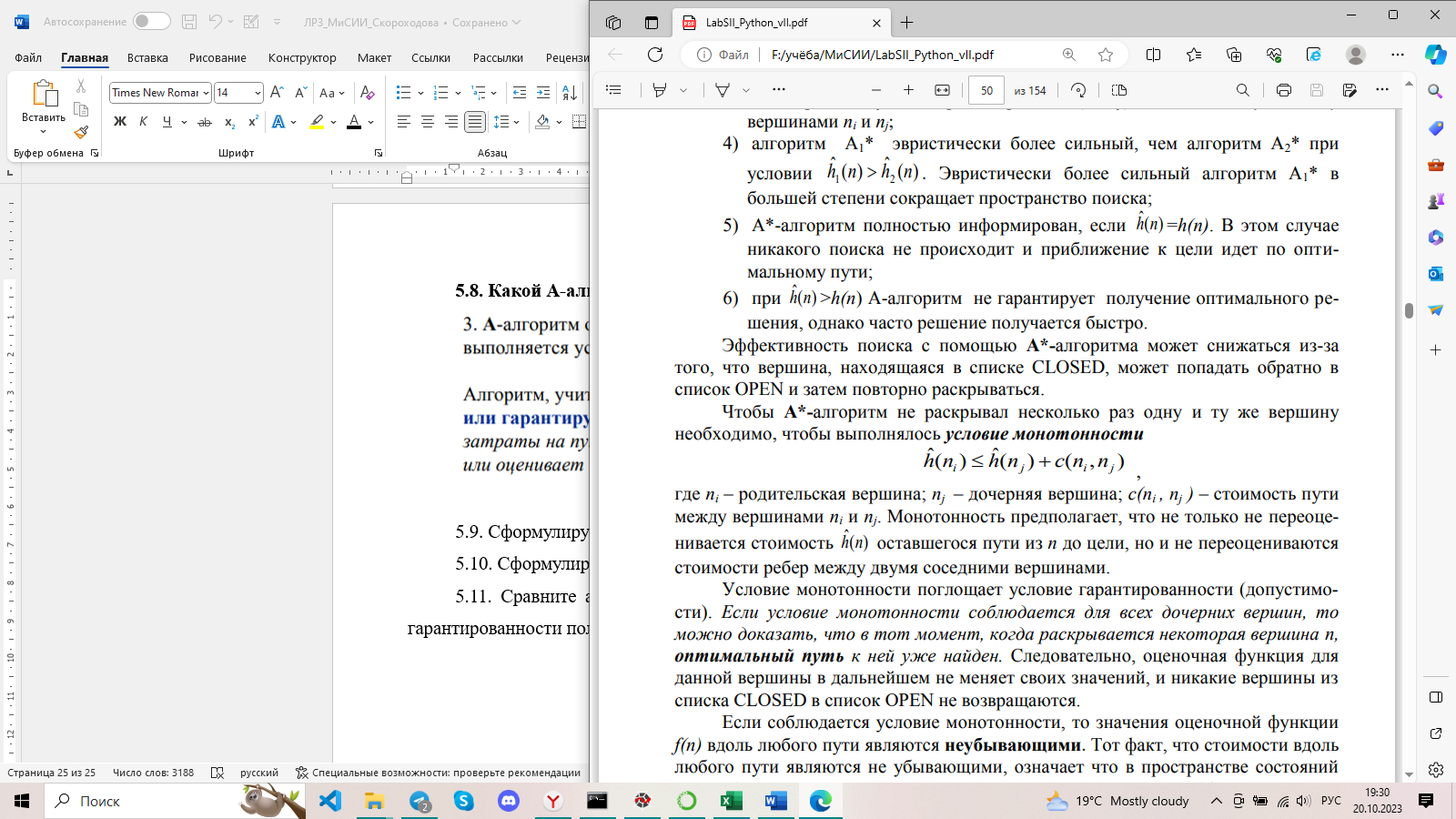




**5.8. Какой А-алгоритм называют гарантирующим (допустимым)?**



**5.9. Сформулируйте и объясните условие монотонности.**



5.10. Сформулируйте эвристику манхэттенского расстояния.

Эвристика манхэттенского расстояния, также известная как расстояние Л1 или городское расстояние, является эвристическим подходом для определения расстояния между двумя точками в сетке, основанным на сумме абсолютных разностей их координат. Вот формулировка этой эвристики:

Для двух точек A(x1, y1) и B(x2, y2) на сетке:

Эвристика манхэттенского расстояния = |x2 - x1| + |y2 - y1|

В данной формулировке используются следующие шаги:

Вычисление абсолютной разности между координатами x точек A и B: |x2 - x1|.

Вычисление абсолютной разности между координатами y точек A и B: |y2 - y1|.

Суммирование полученных значений абсолютных разностей: |x2 - x1| + |y2 - y1|.

5.11. Сравните алгоритмы слепого и эвристического поиска по критерию гарантированности получения результата и эффективности поиска.