**ЛАБОРАТОРНАЯ РАБОТА № 4**

**«ИССЛЕДОВАНИЕ МЕТОДОВ МУЛЬТИАГЕНТНОГО ПОИСКА»**

**Цель работы**

Исследование состязательных методов поиска в мультиагентных средах, приобретение навыков программирования интеллектуальных состязательных агентов, возвращающих стратегию поиска на основе оценочных функций.

**Постановка задачи**

**Задание 1 (16 баллов). Рефлекторный агент ReflexAgent**

Усовершенствуйте поведение рефлекторного агента ReflexAgent в multiAgents.py, чтобы он мог играть достойно. Предоставленный код рефлекторного агента содержит несколько полезных методов, которые запрашивают информацию у класса GameState. Эффективный рефлекторный агент должен учитывать, как расположение пищевых гранул, так и местонахождение призраков.

Усовершенствованный агент должен легко съесть все гранулы на поле игры testClassic:

python pacman.py -p ReflexAgent -l testClassic

Проверьте работу рефлекторного агента на поле mediumClassic по умолчанию с одним или двумя призраками (и отключением анимации для ускорения отображения):

python pacman.py --frameTime 0 -p ReflexAgent -k 1

python pacman.py --frameTime 0 -p ReflexAgent -k 2

Как ведет себя ваш агент? Скорее всего, он будет часто погибать при игре с двумя призраками (на доске по умолчанию), если ваша оценочная функция недостаточно хороша.

Подсказка. Помните, что логический массив newFood можно преобразовать в список с координатами пищевых гранул методом asList().

Подсказка. В качестве принципа построения функции оценки попробуйте использовать обратные значения расстояний от Пакмана до пищевых гранул и призраков, а не только сами значения этих расстояний.

Примечание. Функция оценки для рефлекторного агента, которую вы напишете для этого задания, оценивает пары состояние-действие; для других заданий функция оценки будет оценивать только состояния.

Примечание. Будет полезно просмотреть внутреннее содержимое различных объектов для отладки программы. Это можно сделать, распечатав строковые представления объектов. Например, можно распечатать newGhostStates с помощью print(newGhostStates).

Опции. Поведение призраков в данном случае является случайным; можно поиграть с более умными направленными призраками, используя опцию -g DirectionalGhost. Если случайность не позволяет оценить, улучшается ли ваш агент, то можно использовать опцию -f для запуска с фиксированным начальным случайным значением (одинаковый начальный случайный выбор в каждой игре).

Можно также запустить несколько игр подряд с опцией -n. Отключите при этом графику с помощью опции -q, чтобы играть быстрее.

Автооценивание. В ходе оценивания ваш агент запускается для игры на поле openClassic 10 раз. Вы получите 0 баллов, если ваш агент просрочит время ожидания или никогда не выиграет. Вы получите 1 очко, если ваш агент выиграет не менее 5 раз, или 2 очка, если ваш агент выиграет все 10 игр. Вы получите дополнительный 1 балл, если средний балл вашего агента больше 500, или 2 балла, если он больше 1000. Вы можете оценить своего агента для этих условий командой

python autograder.py -q q1

Для работы без графики используйте команду

python autograder.py -q q1 --no-graphics

Для приведения оценки, выставленной автооценивателем, к 100-балльной итоговой шкале её необходимо умножить на 4. Не тратьте слишком много времени на усовершенствование решения этого задания, поскольку основные задания лабораторной работы впереди.

**Задание 2 (5 баллов). Минимаксный поиск**

Необходимо реализовать минимаксного агента, для которого имеется «заглушка» в виде класса MinimaxAgent (в файле multiAgents.py). Минимаксный агент должен работать с любым количеством призраков, поэтому вам придется написать алгоритм, который будет немного более общим, чем тот, который представлен в разделе 4.2.2. В этом случае минимаксное дерево поиска будет иметь несколько минимальных слоев (по одному для каждого призрака) для каждого максимального слоя. Реализуемый код агента также должен будет выполнять поиск до заданной глубины поддерева игры. Оценки в концевых вершинах минимаксного дерева вычисляются с помощью функции self.evaluationFunction, которая по умолчанию соответствует реализованной функции scoreEvaluationFunction. Класс MinimaxAgent наследует свойства суперкласса MultiAgentSearchAgent, который предоставляет доступ к функциям self.depth и self.evaluationFunction. Убедитесь, что ваш код использует эти функции, где это уместно, поскольку именно эти функции вызываются путем обработки соответствующих параметров командной строки.

Обратите внимание на то, что один слой дерева поиска соответствует одному действию Pacman и последовательным действиям всех агентов-призраков.

Автооцениватель определит, исследует ли ваш агент правильное количество игровых состояний. Это единственный надежный способ обнаружить некоторые очень тонкие ошибки в реализациях минимакса. В результате автооцениватель будет очень требователен к числу вызовов метода GameState.generateSuccessor. Если метод будет вызываться больше или меньше необходимого количества раз, автооцениватель отметит это. Чтобы протестировать и отладить код задания, выполните команду:

python autograder.py -q q2

Результаты тестирования покажут, как ведет себя ваш алгоритм на нескольких небольших деревьях, а также в целом в игре Pacman. Чтобы запустить код без графики, используйте команду:

python autograder.py -q q2 --no-graphics

- Реализуйте алгоритм рекурсивно, используя вспомогательные функции;

- Правильная реализация минимакса приведет к тому, что Pacman будет проигрывать игру в некоторых тестах. Это не станет проблемой при тестировании: это правильное поведение агента, он пройдет тесты;

- Функция оценки для этого задания уже написана (self.evaluationFunction).

Вы не должны изменять эту функцию, но обратите внимание, что теперь мы оцениваем состояния, а не действия, как это было с рефлекторным агентом. Планирующие агенты оценивают будущие состояния, тогда как рефлекторные агенты оценивают действия, исходя из текущего состояния;

- Минимаксные значения начального состояния для игры на поле MinimaxClassic равны 9, 8, 7, -492 для глубин 1, 2, 3 и 4, соответственно. Обратите внимание, что ваш минимаксный агент часто будет выигрывать (665 игр из 1000), несмотря на пессимистичный прогноз для минимакса глубины 4

python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4

- Pacman всегда является агентом 0, и агенты совершают действия в порядке увеличения индекса агента;

- Все состояния в случае минимаксного поиска должны относиться к типу GameStates и передаваться в getAction или генерироваться с помощью GameState.generateSuccessor;

- На больших игровых полях, таких как openClassic и mediumClassic (по умолчанию), вы обнаружите, что минимаксный агент устойчив к умиранию, но плохо ведет себя в отношении выигрыша. Он часто суетится, не добиваясь прогресса. Он может даже метаться рядом с гранулой, не съев ее, потому что не знает, куда бы он пошел после того, как съест гранулу. Не волнуйтесь, если вы заметите такое поведение, в задании 5 эти проблемы будут устранены;

- Когда Пакман считает, что его смерть неизбежна, он постарается завершить игру как можно скорее из-за наличия штрафа за жизнь. Иногда такое поведение ошибочно при случайных перемещениях призраков, но минимаксные агенты всегда исходят из худшего:

python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3

Убедитесь, что вы понимаете, почему Pacman в этом случае нападает на ближайший призрак.

**Задание 3 (5 баллов). Альфа-бета отсечение**

Необходимо реализовать программу агента в классе AlphaBetaAgent, который использует альфа-бета отсечение для более эффективного обследования минимаксного дерева. Ваш алгоритм должен быть более общим, чем псевдокод, рассмотренный в разделе 4.2.4. Суть задания состоит в том, чтобы расширить логику альфа-бета отсечения на несколько минимизирующих агентов.

Вы должны увидеть ускорение работы (возможно альфа-бета отсечение с глубиной 3 будет работать так же быстро, как минимакс с глубиной 2). В идеале, при глубине 3 на игровом поле smallClassic игра должна выполняться со скоростью несколько секунд на один ход или быстрее. python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic

Минимаксные значения начального состояния при игре на поле minimaxClassic равны 9, 8, 7 и -492 для глубин 1, 2, 3 и 4, соответственно.

Оценивание: т.к. проверяется, исследует ли ваш код требуемое количество состояний, то важно, чтобы вы выполняли альфа-бета отсечение без изменения порядка дочерних элементов. Иными словами, состояния-преемники всегда должны обрабатываться в порядке, возвращаемом GameState.getLegalActions.

Также не вызывайте GameState.generateSuccessor чаще, чем необходимо.

Вы не должны выполнять отсечение при равенстве оценок, чтобы соответствовать набору состояний, который исследуется автооценивателем. Для реализации этого задания используйте код из раздела 4.2.4.

python autograder.py -q q3

Результаты покажут, как ведет себя ваш алгоритм на нескольких небольших деревьях, а также в целом на игре pacman. Чтобы запустить код без графики, используйте команду:

python autograder.py -q q3 --no-graphics

Правильная реализация альфа-бета отсечения приводит к тому, что Pacman будет проигрывать на некоторых тестах. Это не создаст проблем при автооценивании: так как это правильное поведение. Ваш агент пройдет тесты.

**Задание 4 (5 баллов). Expectimax**

Минимаксный и альфа-бета поиски предполагают, что игра осуществляется с противником, который принимает оптимальные решения. Это не всегда так. В этом задании необходимо реализовать класс ExpectimaxAgent, который предназначен для моделирования вероятностного поведения агентов, которые могут совершать неоптимальный выбор.

Чтобы отладить свою реализацию на небольших игровых деревьях, используя команду: python autograder.py -q q4

Если ваш алгоритм будет работать на небольших деревьях поиска, то он будет успешен и при игре в Pacman.

Случайные призраки, конечно, не являются оптимальными минимаксными агентами, и поэтому применение в этой ситуации минимаксного поиска не является подходящим. Вместо того, чтобы выбирать минимальную оценку для состояний с действиями призраков, ExpectimaxAgent выбирает ожидаемую оценку.

Чтобы упростить код, предполагается, что в этом случае призрак выбирает одно из своих действий, возвращаемых getLegalActions, равновероятно.

Чтобы увидеть, как ExpectimaxAgent ведет себя при игре в Pacman, выполните команду:

python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3

Теперь вы должны наблюдать иное поведение агента в непосредственной близости к призракам. В частности, если Пакман понимает, что может оказаться в ловушке, но может убежать, чтобы схватить еще несколько гранул еды, он, по крайней мере, попытается это сделать. Изучите результаты этих двух сценариев:

python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10

python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10

Вы должны обнаружить, что теперь ExpectimaxAgent выигрывает примерно в половине случаев, в то время как ваш AlphaBetaAgent всегда проигрывает. Убедитесь, что вы понимаете, почему поведение этого агента отличается от минимаксного случая.

Правильная реализация Expectimax приведет к тому, что Pacman будет проигрывать некоторые тесты. Это не создаст проблем при автооценивании. Ваш агент пройдет тесты.

**Задание 5 (6 баллов). Функция оценки**

Реализуйте лучшую оценочную функцию для игры Pacman в предоставленном шаблоне функции betterEvaluationFunction. Функция оценки должна оценивать состояния, а не действия, как это делала функция оценки рефлекторного агента. При поиске до глубины 2 ваша функция оценки должна обеспечивать выигрыш на поле smallClassic с одним случайным призраком более чем в половине случаев и по-прежнему работать с разумной скоростью (чтобы получить хорошую оценку за это задание, Pacman должен набирать в среднем около 1000 очков, когда он выигрывает).

Оценивание: в ходе автооценивания агент запускается 10 раз на поле smallClassic. При этом вы получаете следующие баллы:

Если вы выиграете хотя бы один раз без тайм-аута автооценивателя, вы получите 1 балл. Любой агент, не удовлетворяющий этим критериям, получит 0 баллов;

+1 за победу не менее 5 раз, +2 за победу в 10 попытках;

+1 для среднего количества очков не менее 500, +2 за среднее количество очков не менее 1000 (включая очки в проигранных играх);

+1, если ваши игры с автооценивателем в среднем требуют менее 30 секунд при запуске с параметром --no-graphics;

Дополнительные баллы за среднее количество очков и время вычислений будут начислены только в том случае, если вы выиграете не менее 5 раз. Пожалуйста, не копируйте файлы из предыдущих лабораторных работ, так как они не пройдут автооценивание на поле Gradescope.

Вы можете оценить своего агента, выполнив команду

python autograder.py -q q5

Для выполнения с отключенной графикой используйте команду:

python autograder.py -q q5 --no-graphics

**Ход работы**

1. Было усовершенствовано поведение рефлекторного агента ReflexAgent в multiAgents.py. Код представлен в листинге 1.

Листинг 1 – Класс ReflexAgent

def evaluationFunction(self, currentGameState, action):

newFood = successorGameState.getFood()

#print("newFood:", newFood)

# определяем новое состояние для призраков newGhostStates

newGhostStates = successorGameState.getGhostStates()

# определяем время испуга прираков

# пример значения для newScaredTimes при 2-х призраках: [40, 40]

newScaredTimes = [ghostState.scaredTimer for ghostState in newGhostStates]

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

currentFood = currentGameState.getFood()

score = successorGameState.getScore()

new\_ghost\_positions = successorGameState.getGhostPositions()

current\_food\_list = currentFood.asList()

new\_food\_list = newFood.asList()

closest\_food = float('+Inf')

closest\_ghost = float('+Inf')

add\_score = 0

if newPos in current\_food\_list:

add\_score += 10.0

distance\_from\_food = [manhattanDistance(newPos, food\_position) for food\_position in new\_food\_list]

total\_available\_food = len(new\_food\_list)

if len(distance\_from\_food):

closest\_food = min(distance\_from\_food)

score += 10.0 / closest\_food - 4.0 \* total\_available\_food + add\_score

for ghost\_position in new\_ghost\_positions:

distance\_from\_ghost = manhattanDistance(newPos, ghost\_position)

closest\_ghost = min([closest\_ghost, distance\_from\_ghost])

if closest\_ghost < 2:

score -= 50.0

return score

Работа агента затем была протестирована на поле игры testClassic, для этого была выполнена команда: python3.6 pacman.py -p ReflexAgent -l testClassic. Результат работы агента представлен на рисунке 1.

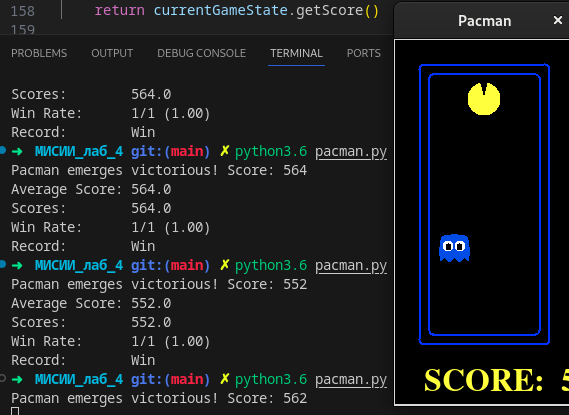


Рисунок 1 – Результат работы рефлекторного агента на поле testClassic

Затем работа агента была протестирована на поле mediumClassic с одним и двумя призраками. Для этого были выполнены команды:

python3.6 pacman.py --frameTime 0 -p ReflexAgent -k 1

python3.6 pacman.py --frameTime 0 -p ReflexAgent -k 2

Результаты изображены на рисунках 2 и 3.



Рисунок 2 – Результат работы рефлекторного агента на поле mediumClassic с 1 призраком

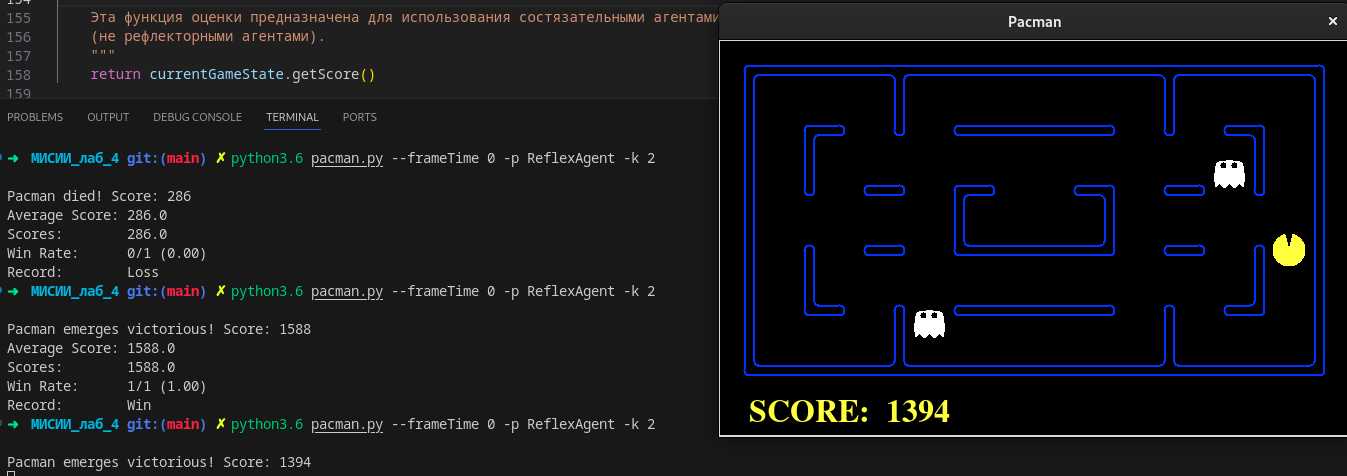


Рисунок 3 – Результат работы рефлекторного агента на поле mediumClassic с 2 призраками

После чего разработанное решение было проверено при помощи автооценивателя. Результаты прохождения тестов продемонстрированы на рисунке 4.

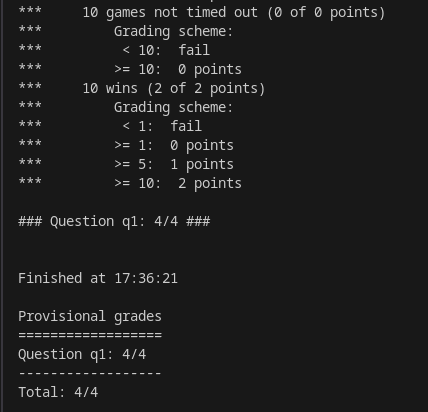


Рисунок 4 – Прохождение тестов при помощи автооценивателя

2. Был реализован минимаксный агент MinimaxAgent в файле multiAgents.py. Код агента представлен в листинге 2.

Листинг 2 – Класс MinimaxAgent

class MinimaxAgent(MultiAgentSearchAgent):

"""

Ваш минимаксный агент (задание 2)

"""

def getAction(self, gameState):

"""

Возвращает минимаксное действие для текущего состояния gameState,

используя self.depth и self.evaluationFunction.

Вот несколько вызовов методов, которые могут быть полезны при реализации

минимаксного агента.

gameState.getLegalActions (agentIndex):

Возвращает список допустимых (легальных) действий для агента

agentIndex=0 соответсвует Пакману, а для призраков agentIndex > = 1

gameState.generateSuccessor(agentIndex, action):

Возвращает состояние-преемник после того, как агент совершит действие action.

gameState.getNumAgents():

Возвращает общее количество агентов в игре.

gameState.isWin():

Возвращает True если состояние игры является выигрышным.

gameState.isLose ():

Возвращает True если состояние игры является проигрышным.

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

best\_action = self.max\_value(gameState=gameState, depth=0, agent\_idx=0)[1]

return best\_action

def is\_terminal\_state(self, gameState, depth, agent\_idx):

if gameState.isWin():

return gameState.isWin()

elif gameState.isLose():

return gameState.isLose()

elif gameState.getLegalActions(agent\_idx) is 0:

return gameState.getLegalActions(agent\_idx)

elif depth >= self.depth \* gameState.getNumAgents():

return self.depth

def max\_value(self, gameState, depth, agent\_idx):

value = (float('-Inf'), None)

legal\_actions = gameState.getLegalActions(agent\_idx)

for action in legal\_actions:

successor\_state = gameState.generateSuccessor(agent\_idx, action)

number\_of\_agents = gameState.getNumAgents()

expand = depth + 1

current\_player = (depth + 1) % number\_of\_agents

value = max([value, (self.value(gameState=successor\_state, depth=expand, agent\_idx=current\_player), action)], key=lambda idx: idx[0])

return value

def min\_value(self, gameState, depth, agent\_idx):

value = (float('+Inf'), None)

legal\_actions = gameState.getLegalActions(agent\_idx)

for action in legal\_actions:

successor\_state = gameState.generateSuccessor(agent\_idx, action)

number\_of\_agents = gameState.getNumAgents()

expand = depth + 1

current\_player = (depth + 1) % number\_of\_agents

value = min([value, (self.value(gameState=successor\_state, depth=expand, agent\_idx=current\_player), action)], key=lambda idx: idx[0])

return value

def value(self, gameState, depth, agent\_idx):

if self.is\_terminal\_state(gameState=gameState, depth=depth, agent\_idx=agent\_idx):

return self.evaluationFunction(gameState)

elif agent\_idx is 0:

return self.max\_value(gameState=gameState, depth=depth, agent\_idx=agent\_idx)[0]

else:

return self.min\_value(gameState=gameState, depth=depth, agent\_idx=agent\_idx)[0]

Написанный код был протестирован при помощи автооценивателя. Результаты изображены на рисунке 5.

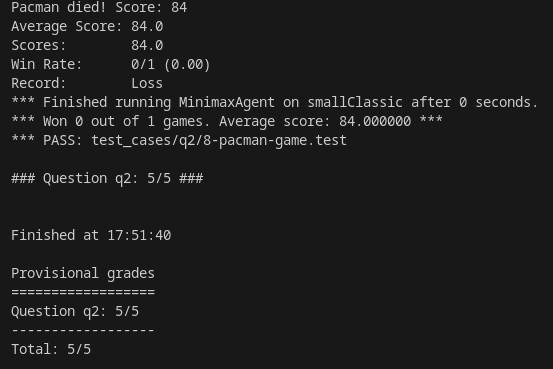


Рисунок 5 – Прохождение тестов при помощи автооценивателя

Минимаксные значения начального состояния для игры на поле MinimaxClassic равны 9, 8, 7, -492 для глубин 1, 2, 3 и 4, соответственно. Обратите внимание, что ваш минимаксный агент часто будет выигрывать (665 игр из 1000), несмотря на пессимистичный прогноз для минимакса глубины 4

python3.6 pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4

Результат выполнения команды представлен на рисунке 6.

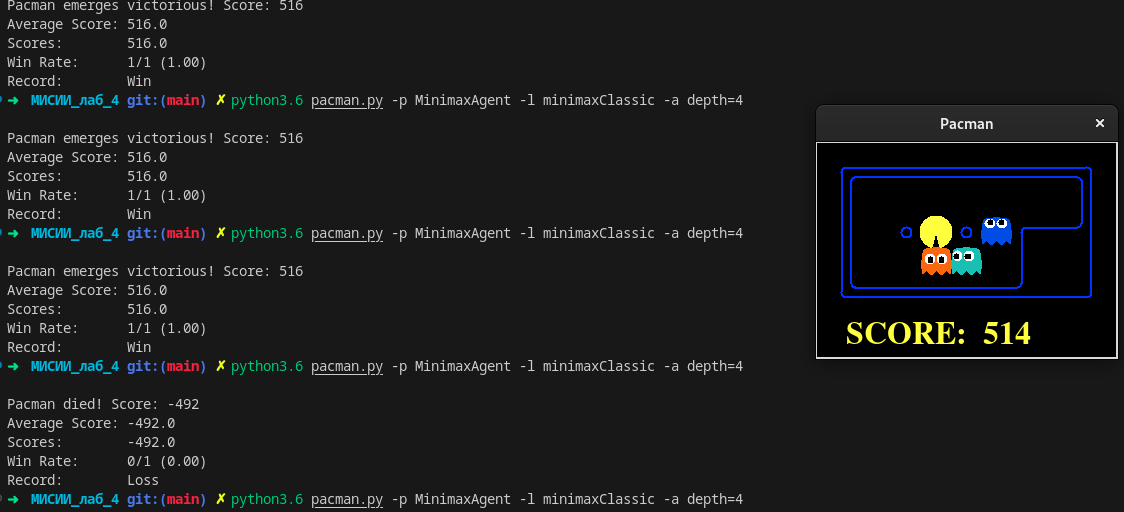


Рисунок 6 – Результат работы агента минимаксноог поиска

Когда Пакман считает, что его смерть неизбежна, он постарается завершить игру как можно скорее из-за наличия штрафа за жизнь. Иногда такое поведение ошибочно при случайных перемещениях призраков, но минимаксные агенты всегда исходят из худшего:

python3.6 pacman.py -p MinimaxAgent -l trappedClassic -a depth=3

Результат выполнения команды продемонстрирован на рисунке 7.

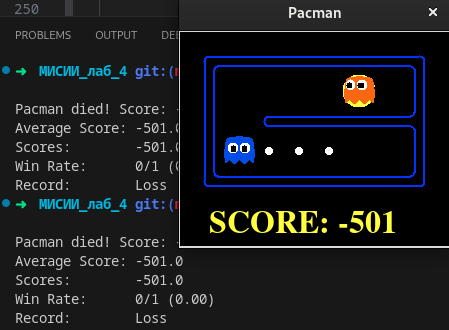


Рисунок 7 – Результат выполнения команды

3. Была реализована программа агента в классе AlphaBetaAgent, который использует альфа-бета отсечение для более эффективного обследования минимаксного дерева. В листинге 3 представлен разработанный код.

Листинг 3 – Класс AlphaBetaAgent

class AlphaBetaAgent(MultiAgentSearchAgent):

"""

Ваш минимаксный агент, реализующий альфа-бета отсечение (задание 3)

"""

def getAction(self, gameState):

"""

Возвращает минимаксное действие, используя

self.depth and self.evaluationFunction

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

alpha = float('-Inf')

beta = float('+Inf')

depth = 0

best\_action = self.max\_value(gameState=gameState, depth=depth, agent\_idx=0, alpha=alpha, beta=beta)

return best\_action[1]

def is\_terminal\_state(self, gameState, depth, agent\_idx):

if gameState.isWin():

return gameState.isWin()

elif gameState.isLose():

return gameState.isLose()

elif gameState.getLegalActions(agent\_idx) is 0:

return gameState.getLegalActions(agent\_idx)

elif depth >= self.depth \* gameState.getNumAgents():

return self.depth

def max\_value(self, gameState, depth, agent\_idx, alpha, beta):

value = (float('-Inf'), None)

legal\_actions = gameState.getLegalActions(agent\_idx)

for action in legal\_actions:

successor\_state = gameState.generateSuccessor(agent\_idx, action)

number\_of\_agents = gameState.getNumAgents()

expand = depth + 1

current\_player = expand % number\_of\_agents

value = max([value, (self.value(gameState=successor\_state, depth=expand, agent\_idx=current\_player, alpha=alpha, beta=beta), action)], key=lambda idx: idx[0])

if value[0] > beta:

return value

alpha = max(alpha, value[0])

return value

def min\_value(self, gameState, depth, agent\_idx, alpha, beta):

value = (float('+Inf'), None)

legal\_actions = gameState.getLegalActions(agent\_idx)

for action in legal\_actions:

successor\_state = gameState.generateSuccessor(agent\_idx, action)

number\_of\_agents = gameState.getNumAgents()

expand = depth + 1

current\_player = expand % number\_of\_agents

value = min([value, (self.value(gameState=successor\_state, depth=expand, agent\_idx=current\_player, alpha=alpha, beta=beta), action)], key=lambda idx: idx[0])

if value[0] < alpha:

return value

beta = min(beta, value[0])

return value

def value(self, gameState, depth, agent\_idx, alpha, beta):

if self.is\_terminal\_state(gameState=gameState, depth=depth, agent\_idx=agent\_idx):

return self.evaluationFunction(gameState)

elif agent\_idx is 0:

return self.max\_value(gameState=gameState, depth=depth, agent\_idx=agent\_idx,alpha=alpha, beta=beta)[0]

else:

return self.min\_value(gameState=gameState, depth=depth, agent\_idx=agent\_idx, alpha=alpha, beta=beta)[0]

Затем код был проверен, для чего была выполнена команда python3.6 pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic. Результат выполнения программы изображен на рисунке 8.

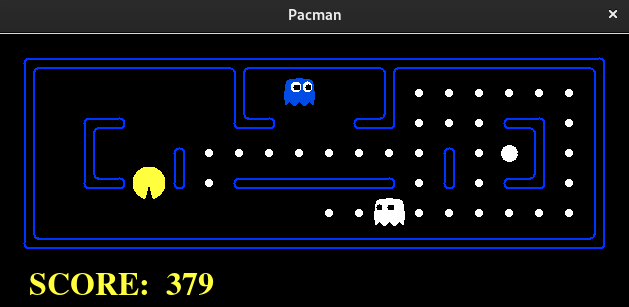


Рисунок 8 – Результат выполнения команды

После чего разработанное решение было проверено при помощи автооценивателя. Результат прохождения тестов представлен на рисунке 9.

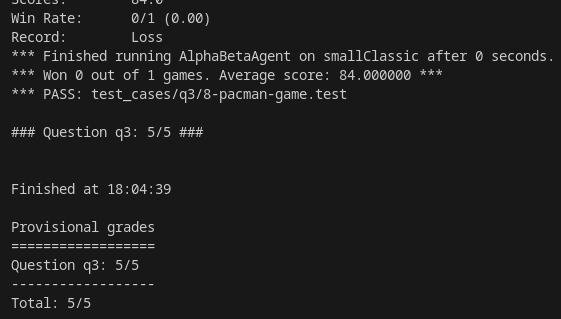


Рисунок 9 – Прохождение тестов при помощи автооценивателя

4. Был реализован класс ExpectimaxAgent, который предназначен для моделирования вероятностного поведения агентов, которые могут совершать неоптимальный выбор. Код представлен в листинге 4.

Листинг 4 – Класс ExpectimaxAgent

class ExpectimaxAgent(MultiAgentSearchAgent):

"""

Ваш expectimax агент (задание 4)

"""

def getAction(self, gameState):

"""

Возвращает действие Пакмана, используя expectimax поиск и

self.depth и self.evaluationFunction

Все призраки должны выбирать свои случайные

допустимые действия с равной вероятностью

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

best\_action = self.max\_value(gameState=gameState, depth=0, agent\_idx=0)[1]

return best\_action

def is\_terminal\_state(self, gameState, depth, agent\_idx):

if gameState.isWin():

return gameState.isWin()

elif gameState.isLose():

return gameState.isLose()

elif gameState.getLegalActions(agent\_idx) is 0:

return gameState.getLegalActions(agent\_idx)

elif depth >= self.depth \* gameState.getNumAgents():

return self.depth

def max\_value(self, gameState, depth, agent\_idx):

value = (float('-Inf'), None)

legal\_actions = gameState.getLegalActions(agent\_idx)

for action in legal\_actions:

successor\_state = gameState.generateSuccessor(agent\_idx, action)

number\_of\_agents = gameState.getNumAgents()

expand = depth + 1

current\_player = (depth + 1) % number\_of\_agents

value = max([value, (self.value(gameState=successor\_state, depth=expand, agent\_idx=current\_player), action)], key=lambda idx: idx[0])

return value

def expected\_value(self, gameState, depth, agent\_idx):

value = list()

legal\_actions = gameState.getLegalActions(agent\_idx)

for action in legal\_actions:

successor\_state = gameState.generateSuccessor(agent\_idx, action)

number\_of\_agents = gameState.getNumAgents()

expand = depth + 1

current\_player = (depth + 1) % number\_of\_agents

value.append(self.value(gameState=successor\_state, depth=expand, agent\_idx=current\_player))

expected\_value = sum(value) / len(value)

return expected\_value

def value(self, gameState, depth, agent\_idx):

if self.is\_terminal\_state(gameState=gameState, depth=depth, agent\_idx=agent\_idx):

return self.evaluationFunction(gameState)

elif agent\_idx is 0:

return self.max\_value(gameState=gameState, depth=depth, agent\_idx=agent\_idx)[0]

else:

return self.expected\_value(gameState=gameState, depth=depth, agent\_idx=agent\_idx)

Данный код был проверен при помощи автооценивателя. Результаты тестов продемонстрированы на рисунке 10.

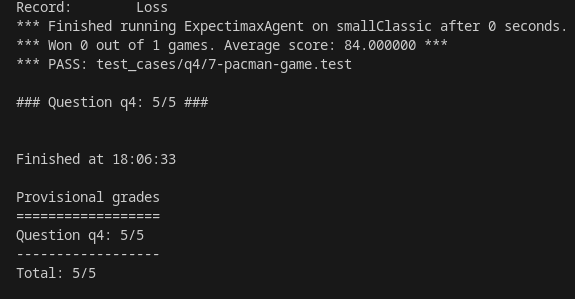


Рисунок 10 – Прохождение тестов при помощи автооценивателя

Далее была выполнена команда python3.6 pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3. Результат выполнения команды изображен на рисунке 11.

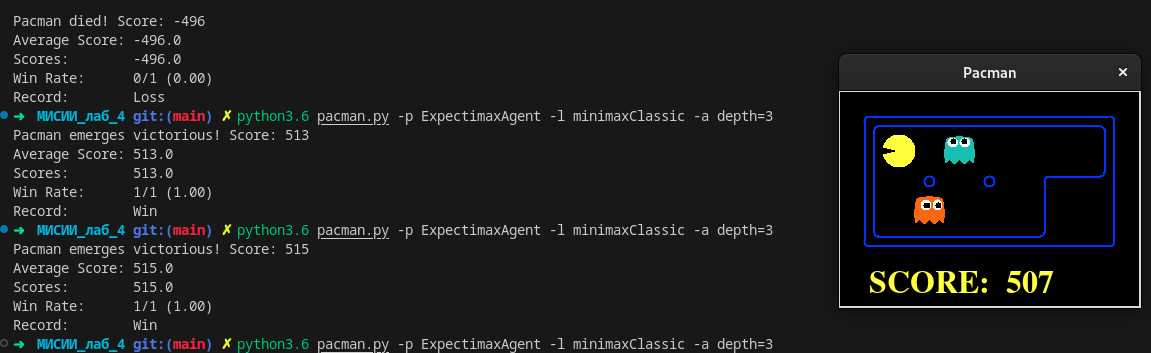


Рисунок 11 – Агент ExpectimaxAgent при игре в Pacman

5. Была реализована оценочная функция для игры Pacman в предоставленном шаблоне функции betterEvaluationFunction, код представлен в листинге 5.

Листинг 5 – Функция betterEvaluationFunction

def betterEvaluationFunction(currentGameState):

"""

Ваша усовершенствованная функция оценки (вопрос 5)

ОПИСАНИЕ: <втавьте сюда описание Вашей функции>

"""

"\*\*\* ВСТАВЬТЕ ВАШ КОД СЮДА \*\*\*"

pacman\_position = currentGameState.getPacmanPosition()

food\_positions = currentGameState.getFood().asList()

capsules\_positions = currentGameState.getCapsules()

ghost\_states = currentGameState.getGhostStates()

remaining\_food = len(food\_positions)

remaining\_capsules = len(capsules\_positions)

scared\_ghosts = list()

enemy\_ghosts = list()

enemy\_ghost\_positions = list()

scared\_ghosts\_positions = list()

score = currentGameState.getScore()

closest\_food = float('+Inf')

closest\_enemy\_ghost = float('+Inf')

closest\_scared\_ghost = float('+Inf')

distance\_from\_food = [manhattanDistance(pacman\_position, food\_position) for food\_position in food\_positions]

if len(distance\_from\_food) is not 0:

closest\_food = min(distance\_from\_food)

score -= 1.0 \* closest\_food

for ghost in ghost\_states:

if ghost.scaredTimer is not 0:

enemy\_ghosts.append(ghost)

else:

scared\_ghosts.append(ghost)

for enemy\_ghost in enemy\_ghosts:

enemy\_ghost\_positions.append(enemy\_ghost.getPosition())

if len(enemy\_ghost\_positions) is not 0:

distance\_from\_enemy\_ghost = [manhattanDistance(pacman\_position, enemy\_ghost\_position) for enemy\_ghost\_position in enemy\_ghost\_positions]

closest\_enemy\_ghost = min(distance\_from\_enemy\_ghost)

score -= 2.0 \* (1 / closest\_enemy\_ghost)

for scared\_ghost in scared\_ghosts:

scared\_ghosts\_positions.append(scared\_ghost.getPosition())

if len(scared\_ghosts\_positions) is not 0:

distance\_from\_scared\_ghost = [manhattanDistance(pacman\_position, scared\_ghost\_position) for scared\_ghost\_position in scared\_ghosts\_positions]

closest\_scared\_ghost = min(distance\_from\_scared\_ghost)

score -= 3.0 \* closest\_scared\_ghost

score -= 20.0 \* remaining\_capsules

score -= 4.0 \* remaining\_food

return score

Далее, разработанная функция была проверена при помощи автооценивателя. Результат успешного прохождения тестов продемонстрирован на рисунке 12.

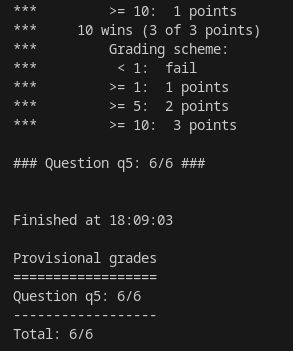


Рисунок 12 – Прохождение тестов при помощи автооценивателя

**Выводы**

В ходе выполнения лабораторной работы было проведено исследование состязательных методов поиска в мультиагентных средах и приобретены навыки программирования интеллектуальных состязательных агентов, возвращающих стратегию поиска на основе оценочных функций.