

## 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 `Расман` и последовательным действиям всех агентов-призраков.

Автооценщик определит, исследует ли ваш агент правильное количество игровых состояний. Это единственный надежный способ обнаружить некоторые очень тонкие ошибки в реализациях минимакса. В результате автооценщик будет очень требователен к числу вызовов метода `GameState.generateSuccessor`. Если метод будет вызываться больше или меньше необходимого количества раз, автооценщик 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 `Расман`. 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 Расман 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
```

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

- Все состояния в случае минимаксного поиска должны относиться к типу GameState 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 Расман 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 расман. Чтобы запустить код без графики, используйте команду:

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

Правильная реализация альфа-бета отсечения приводит к тому, что Расман 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 `Расман`.

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 `Расман`, 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 Exprectimax приведет к тому, что Расман 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 Расман 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, Расман 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 автооценителя, 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 автооценителем 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 рефлексного 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 призраков
    # 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.



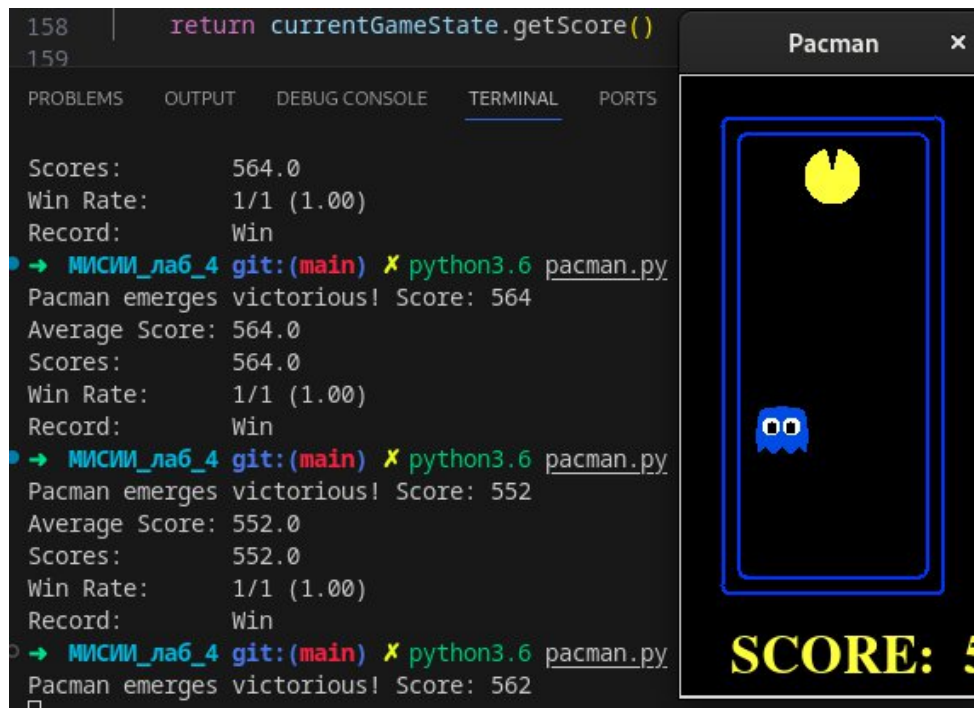


Рисунок 1 – Результат работы рефлексного агента на поле testClassic

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.

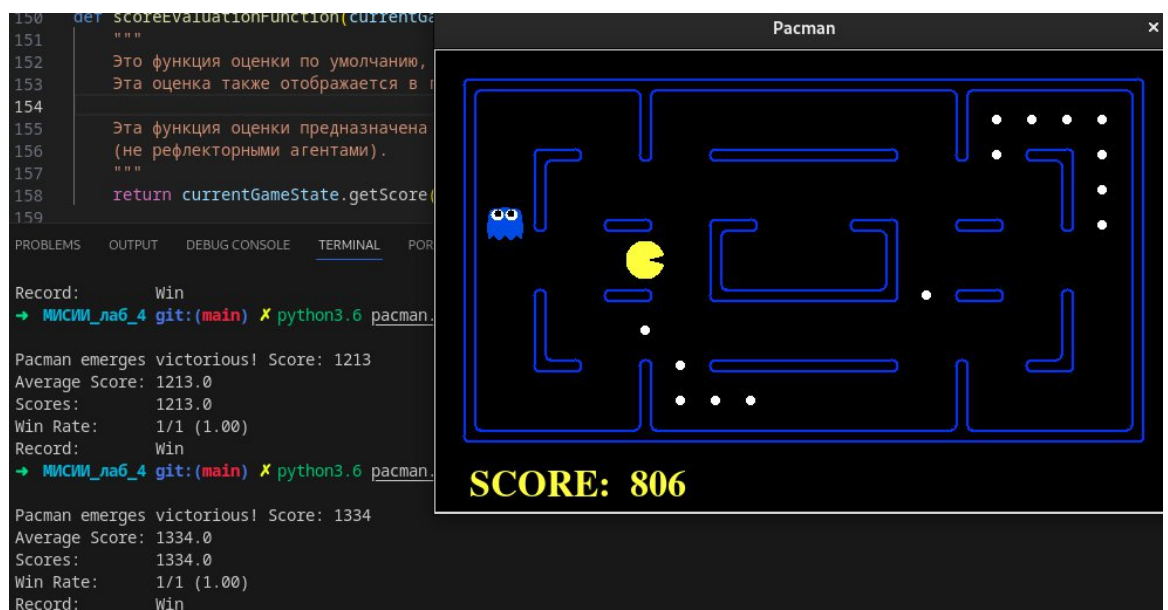


Рисунок 2 – Результат работы рефлексного агента на поле mediumClassic с 1 призраком

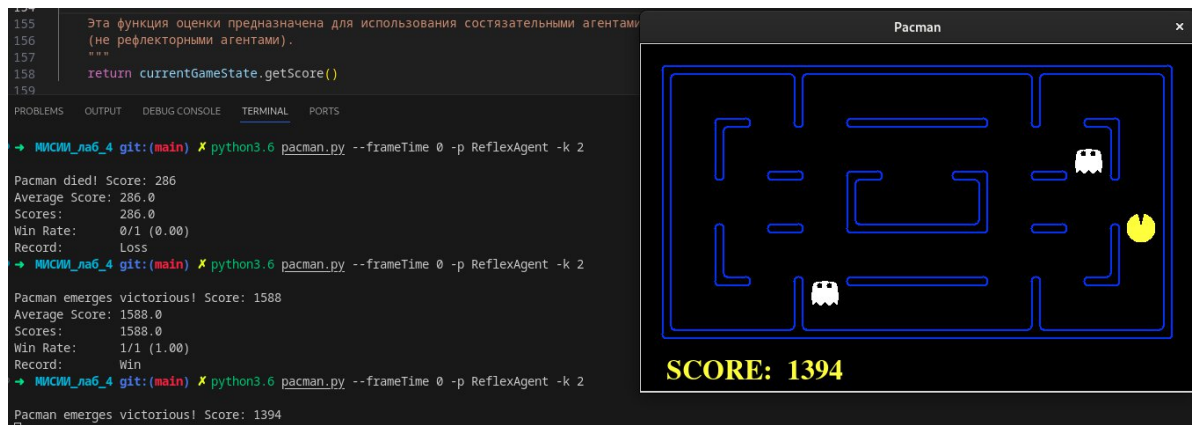


Рисунок 3 – Результат работы рефлексного агента на поле mediumClassic с 2 призраками

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

```
*** 10 games not timed out (0 of 0 points)
*** Grading scheme:
*** < 10: fail
*** >= 10: 0 points
*** 10 wins (2 of 2 points)
*** Grading scheme:
*** < 1: fail
*** >= 1: 0 points
*** >= 5: 1 points
*** >= 10: 2 points

### Question q1: 4/4 ###

Finished at 17:36:21

Provisional grades
=====
Question q1: 4/4
-----
Total: 4/4
```

Рисунок 4 – Прохождение тестов при помощи автооценителя

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 соответствует 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 автооценителя. Результаты изображены на рисунке 5.

```

Pacman died! Score: 84
Average Score: 84.0
Scores:      84.0
Win Rate:    0/1 (0.00)
Record:      Loss
*** Finished running MinimaxAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases/q2/8-pacman-game.test

### Question q2: 5/5 ###

Finished at 17:51:40

Provisional grades
=====
Question q2: 5/5
-----
Total: 5/5

```

Рисунок 5 – Прохождение тестов при помощи автооценителя

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.

```
Pacman emerges victorious! Score: 516
Average Score: 516.0
Scores: 516.0
Win Rate: 1/1 (1.00)
Record: Win
→ МИСИИ_лаб_4 git:(main) X python3.6 pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4

Pacman emerges victorious! Score: 516
Average Score: 516.0
Scores: 516.0
Win Rate: 1/1 (1.00)
Record: Win
→ МИСИИ_лаб_4 git:(main) X python3.6 pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4

Pacman emerges victorious! Score: 516
Average Score: 516.0
Scores: 516.0
Win Rate: 1/1 (1.00)
Record: Win
→ МИСИИ_лаб_4 git:(main) X python3.6 pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4

Pacman died! Score: -492
Average Score: -492.0
Scores: -492.0
Win Rate: 0/1 (0.00)
Record: Loss
→ МИСИИ_лаб_4 git:(main) X python3.6 pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```


A screenshot of a Pacman game window titled "Pacman". The game board shows Pacman (yellow circle) and three ghosts (blue, orange, and cyan). The score "SCORE: 514" is displayed in large yellow text at the bottom right of the game area.

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.

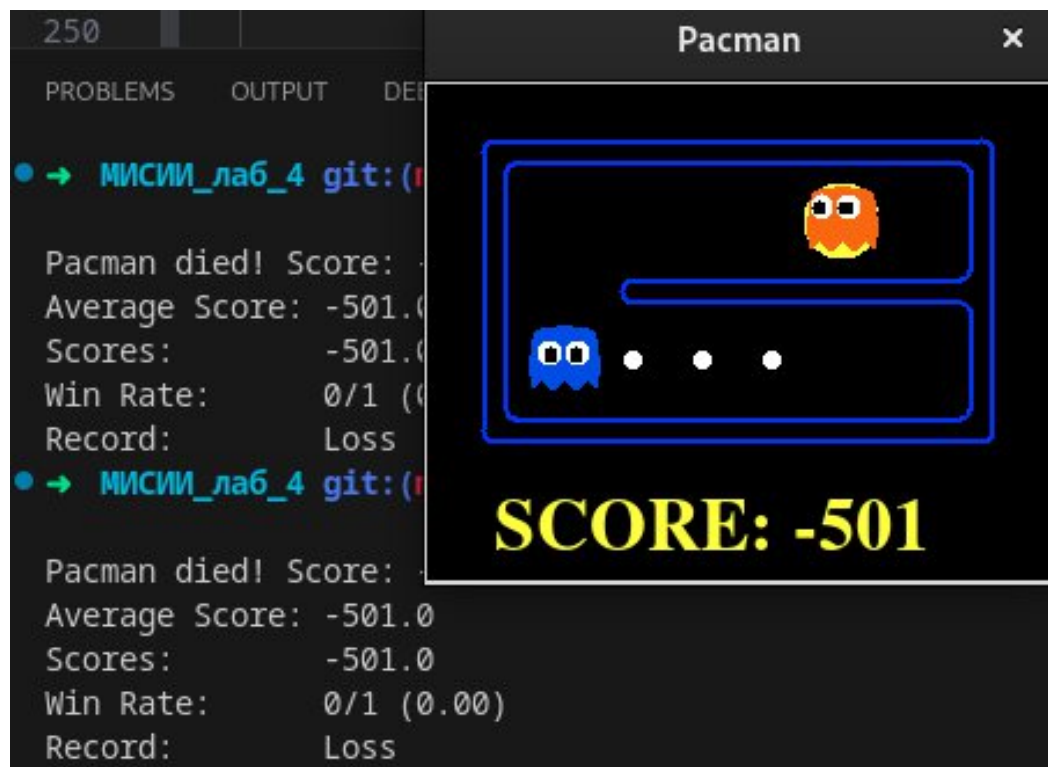
A screenshot of an IDE window. The left pane shows a terminal with the command `python3.6 pacman.py -p MinimaxAgent -l trappedClassic -a depth=3` and its output, which includes "Pacman died! Score: -501.0" and "Average Score: -501.0". The right pane shows a Pacman game window titled "Пакман" (Pacman) with a score of "SCORE: -501" in large yellow text. The game board shows Pacman (yellow circle) and three ghosts (blue, orange, and cyan).

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.

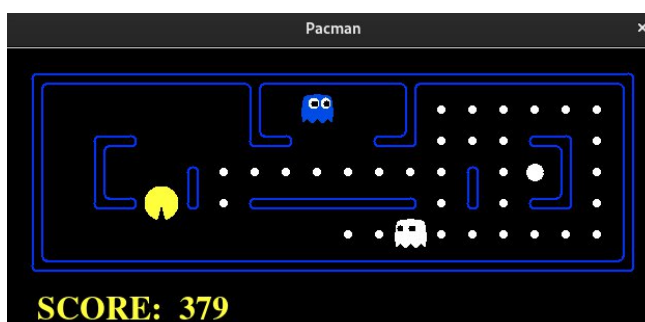


Рисунок 8 – Результат выполнения команды

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

```

Score: 84.0
Win Rate: 0/1 (0.00)
Record: Loss
*** Finished running AlphaBetaAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.00000 ***
*** PASS: test_cases/q3/8-pacman-game.test

### Question q3: 5/5 ###

Finished at 18:04:39

Provisional grades
=====
Question q3: 5/5
-----
Total: 5/5

```

## Рисунок 9 – Прохождение тестов при помощи автооценителя

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
```



```

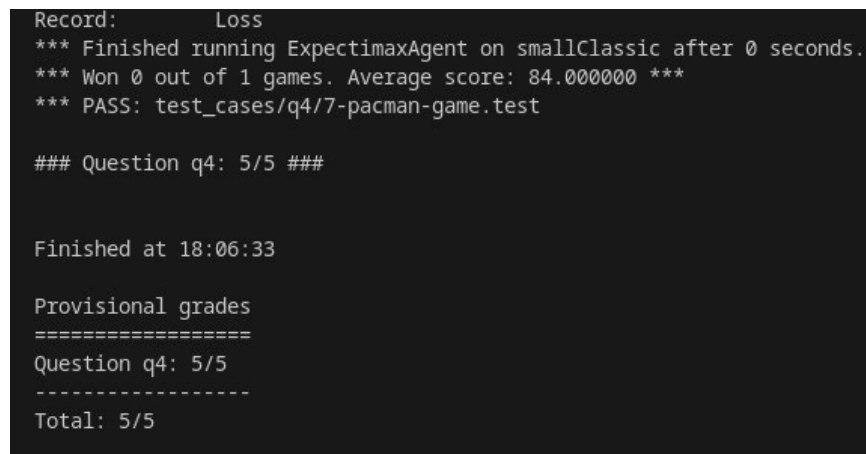
        current_player = (depth + 1) % number_of_agents
        value.append(self.value(gameState=successor_state,
                                agent_idx=current_player))
        expected_value = sum(value) / len(value)
        return expected_value

    def value(self, gameState, depth, agent_idx):

        if self.is_terminal_state(gameState=gameState,
                                agent_idx=agent_idx):
            return self.evaluationFunction(gameState)
        elif agent_idx is 0:
            return self.max_value(gameState=gameState,
                                agent_idx=agent_idx)[0]
        else:
            return self.expected_value(gameState=gameState,
                                agent_idx=agent_idx)

```

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



```

Record:      Loss
*** Finished running ExpectimaxAgent on smallClassic after 0 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases/q4/7-pacman-game.test

### Question q4: 5/5 ###

Finished at 18:06:33

Provisional grades
=====
Question q4: 5/5
-----
Total: 5/5

```

Рисунок 10 – Прохождение тестов при помощи автооценителя

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

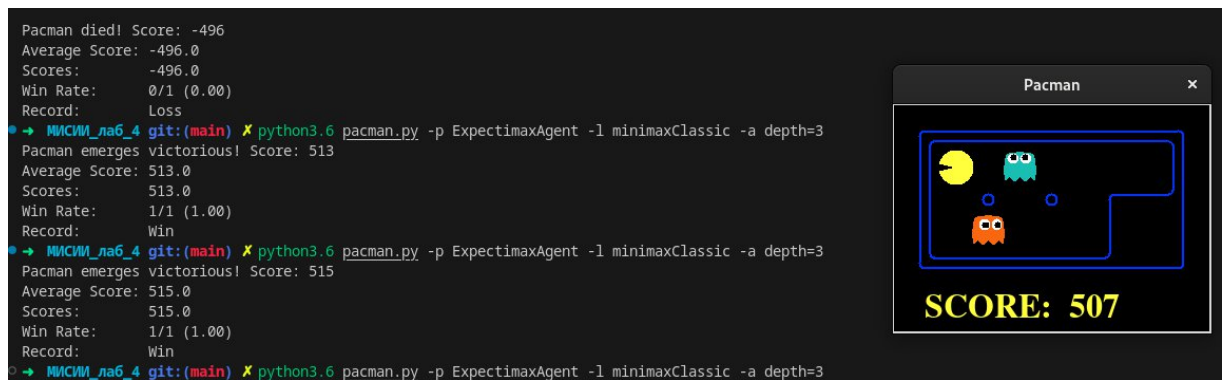


Рисунок 11 – Агент ExpectimaxAgent при игре в Расман

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

### Листинг 5 – Функция betterEvaluationFunction

```
def betterEvaluationFunction(currentGameState):
    """
    Ваша усовершенствованная функция оценки (вопрос 5)

    ОПИСАНИЕ: <вставьте 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.

```
***          >= 10:  1 points
***      10 wins (3 of 3 points)
***      Grading scheme:
***          < 1:  fail
***          >= 1:  1 points
***          >= 5:  2 points
***          >= 10: 3 points

### Question q5: 6/6 ###

Finished at 18:09:03

Provisional grades
=====
Question q5: 6/6
-----
Total: 6/6
```

Рисунок 12 – Прохождение тестов при помощи автооценителя

### 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.