4. ЛАБОРАТОРНАЯ РАБОТА № 4
«ИССЛЕДОВАНИЕ МЕТОДОВ МУЛЬТИАГЕНТНОГО ПОИСКА»

4.1. Цель работы

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

4.2. Краткие теоретические сведения

4.2.1 Поиск решений в игровых программах

Существует много разных типов игр. В играх могут выполняться детерминированные или стохастические (вероятностные) ходы, в них могут принимать участие один или несколько игроков.

Первый класс игр, который мы рассмотрим, - это **детерминированные игры с нулевой суммой** (шашки, шахматы, го и др.). Такие игры характеризуются полной информацией о текущей игровой ситуации (имеются игры с неполной информацией), где два игрока-противника по очереди делают ходы. Успех одного игрока – такая же по величине потеря для другого игрока. Самый простой способ представить себе такие игры - это их определение с помощью единственной переменной, значение которой агент-игрок пытается максимизировать, а его противник пытается минимизировать. Например, в игре Pacman такая переменная соответствует набранным баллам, которые Pacman пытается максимизировать, поедая гранулы, в то время как призраки пытаются свести к минимуму эти баллы, съедая агента. Фактически игроки в этом случае соревнуются (состязаются), поэтому поиск в игровых программах называют **состязательным поиском**.

Сложность поиска в играх весьма высока, так как вершины дерева игры имеют высокую степень ветвления. Поэтому при поиске по дереву игры необходимо: прогнозировать граничную глубину поиска; оценивать перспективность позиций игры с помощью оценочных функций. Для этого в каждой позиции игры формируется дерево возможных продолжений игры, имеющее определенную глубину, и с помощью некоторой оценочной функции вычисляются оценки концевых вершин такого дерева. Затем полученные оценки распространяются вверх по дереву, и корневая вершина, соответствующая текущей позиции, получает оценку, позволяющую оценить перспективность того или иного хода из этой вершины.

В отличие от рассмотренных ранее методов поиска, которые возвращали исчерпывающий план, состязательный поиск возвращает **стратегию или политику**, которая просто рекомендует наилучший возможный ход с учетом некоторой конфигурации игры.

4.2.2. Минимаксный поиск.

В соответствии с минимаксным методом поиска вместо полного просмотра дерева игры обследуется лишь его небольшая часть. В этом случае говорят, что дерево игры подвергается **подрезке.** Простейший способ подрезки — это просмотр дерева игры на определенную глубину. Поэтому **дерево поиска** — это только верхняя часть дерева игры.

Различают два вида оценок вершин дерева поиска: статические и динамические [1]. **Статические оценки** приписываются терминальным вершинам (состояниям) дерева игры. Вычисление этих оценок реализуются с помощью **функции полезности** (utility function). **Динамические оценки** получаются при распространении статических оценок вверх по дереву. Метод, которым это достигается, называется **минимаксным**. Для вершины, в которой ход выполняет игрок (МАКС), выбирается наибольшая из оценок вершин нижнего уровня (т.е. уровня МИН'а). Для вершины, в которой ход выполняет противник, выбирается наименьшая из оценок дочерних вершин. На рисунке 4.1 изображен пример распространения оценок вверх по дереву в соответствии с указанным минимаксным принципом. Из рисунка 4.1 следует, что лучшим ходом МАКС'а в вершине *S* будет ход *S-D*, а лучшим ходом МИН'а в вершине *D* будет ход *D-L*.

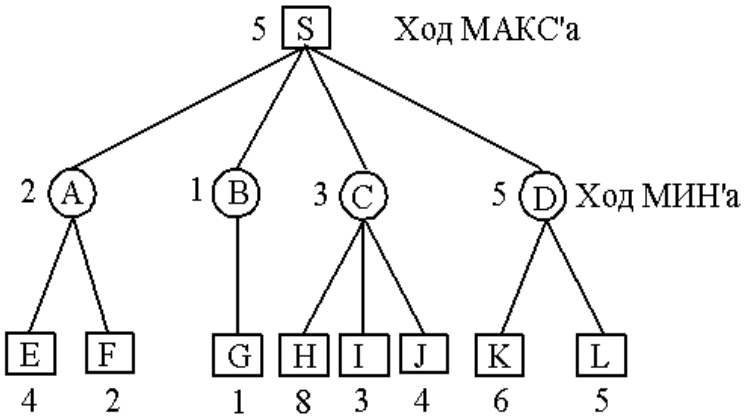

Рисунок 4.1 – Распространение оценок при минимаксной игре

Если обозначить статическую оценку в вершине (состоянии) *s* через *utility(s)*, а динамическую – через *V(s)*, то формально оценки, приписываемые вершинам (состояниям) в соответствии с минимаксным принципом, можно записать так [1]:

$$V(s)=utility(s),$$

если *s* – терминальная вершина (состояние) дерева поиска;

$$V(s)=\max_i V(s_i),$$

если *s* – вершина (состояние) с ходом МАКС'а;

$$V(s)=\min_i V(s_i),$$

если s – вершина (состояние) с ходом МИН'а. Здесь s_i – дочерние вершины для вершины s.

Результирующий псевдокод (в виде функции **value(state)** с косвенной рекурсией, возвращающей оценку (ценность, полезность) состояния **state**) для минимаксного поиска представлен ниже:

```
def value(state):
    if state является терминальным: return utility(state)
    if  агент MAX: return max_value(state)
    if  агент MIN: return min_value(state)

def max_value(state):
    v = -∞
    for succ in successor(state): # для всех дочерних состояний
            v=max(v, value(succ))  # рекурсивный вызов value
    return v

def min_value(state):
    v = +∞
    for succ in successor(state):
            v=min(v, value(succ))  # рекурсивный вызов value
    return v
```

Минимаксный поиск за счет рекурсивного погружения ведет себя подобно поиску в глубину, вычисляя оценки состояний в том же порядке, что и DFS. Поэтому временная сложность алгоритма $O(b^m)$, где m — максимальная глубина дерева поиска.

4.2.3. Игры с несколькими игроками

Многие игры допускают наличие более 2-х игроков. Рассмотрим, как можно распространить идею минимаксного поиска на игры с несколькими игроками (агентами).

Для этого можно заменить единственное значение оценки для каждого узла вектором значений оценок [3]. Например, в игре с тремя игроками A, B и C (рисунок 4.2) с каждым узлом ассоциируется вектор (V_A , V_B , V_C) с тремя оценками. Этот вектор задает полезность состояния с точки зрения каждого игрока.

Для пояснения вычисления динамических оценок в нетерминальных узлах рассмотрим узел, обозначенный на рисунке 4.2, как X. В этом состоянии ход выбирает игрок C, анализируя векторы оценок дочерних состояний: (1, 2, 6) и (4, 2, 3). Поскольку для игрока C состояние с $V_C = 6$ предпочтительнее, то игрок C в состоянии X выбирает первый ход и состоянию X приписывается вектор оценок (1, 2, 6).

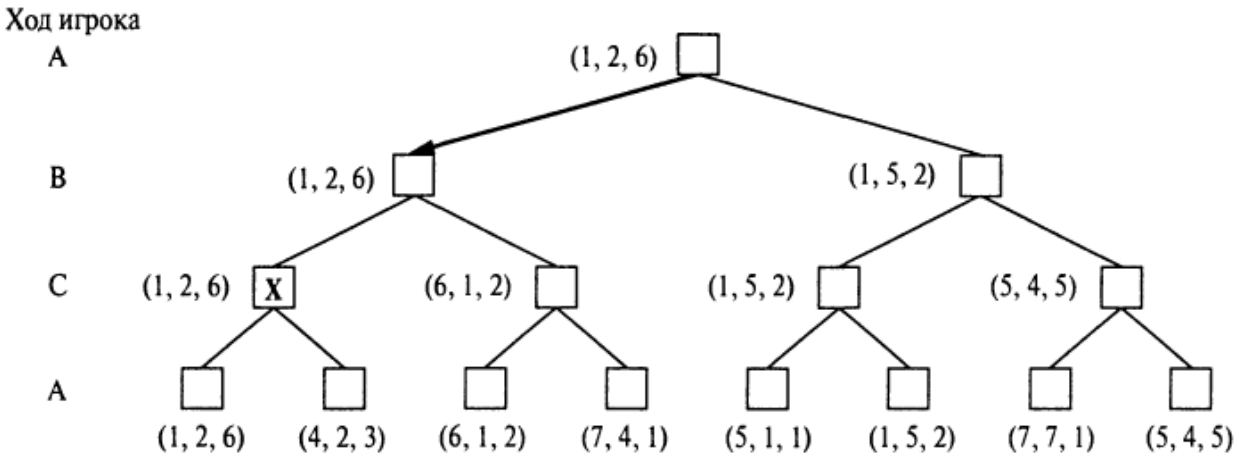

Рисунок 4.2 – Дерево игры с несколькими игроками

Действуя аналогично при всех других состояниях игры, получаем для игрока A в корневом состоянии вектор оценок, равный (1, 2, 6), т.е. игрок A в этом состоянии выберет ход, обозначенный на рисунке 4.2 стрелкой.

4.2.4. Альфа-бета поиск

Минимаксный метод поиска предполагает систематический обход дерева поиска. Альфа-бета поиск позволяет избежать последовательного обхода дерева (рисунок 4.3) за счет отсечения некоторых поддеревьев поиска. Предполагается, что поиск осуществляется сверху вниз и слева направо.

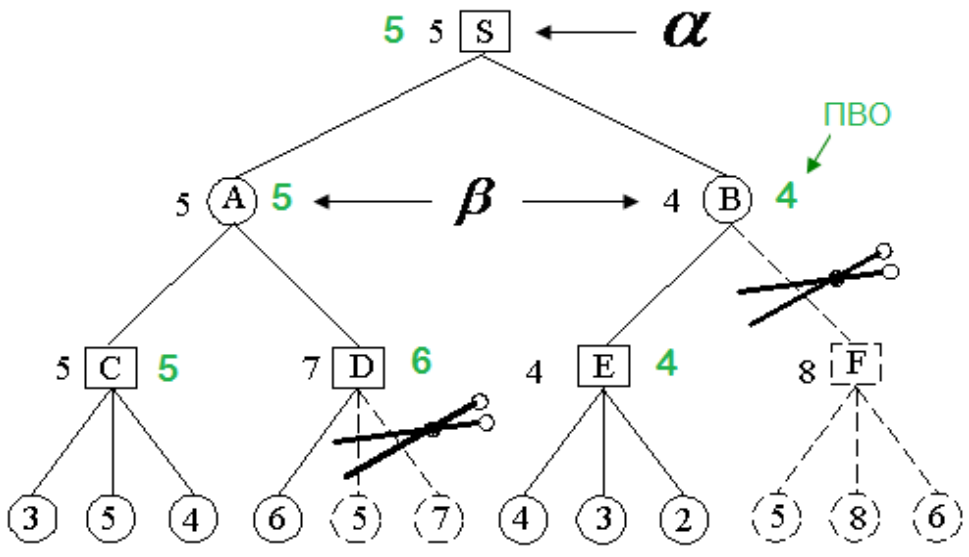

Рисунок 4.3 – Альфа-бета поиск

В альфа-бета методе статическая оценка каждой терминальной вершины вычисляется сразу, как только такая вершина будет построена. Затем полученная оценка распространяется вверх по дереву и с каждой из родительских вершин связывается **предположительно возвращаемая оценка** (ПВО). При этом гарантируется, что для родительской вершины, в которой ход выполняет игрок (ее также

называют ***альфа-вершиной***), уточненная оценка, вычисляемая на последующих этапах, будет не ниже ПВО. Если же в родительской вершине ход выполняет противник (***бета-вершина***), то гарантируется, что последующие оценки будут не выше ПВО. Это позволяет отказаться от построения некоторых вершин и сократить объем поиска. При этом используются следующие правила:

- если ПВО для бета-вершины становится меньше или равной ПВО родительской вершины, вычисленной на предыдущем шаге, то нет необходимости строить дальше поддерево, начинающееся ниже этой бета-вершины;

- если ПВО для альфа-вершины становится больше или равной ПВО родительской вершины, вычисленной на предыдущем шаге, то нет необходимости строить дальше поддерево, начинающееся ниже этой альфа-вершины;

Первое правило соответствует ***альфа-отсечению***, а второе – ***бета-отсечению***. Для дерева поиска, изображенного на рисунке 4.2, ситуация альфа-отсечения имеет место при вычислении ПВО вершины *D* ([*V(D)*=6] > *V(A)*=5]), а ситуация бета-отсечения – для вершины *B* ([*V(B)*=4] < [*V(S)*=5]).

Псевдокод альфа-бета поиска аналогичен минимаксному поиску, но требует переопределения функций, вычисляющих минимальные и максимальные оценки:

```
def max_value(state, α, β):
    v = -∞
    for succ in successor(state):
            v=max(v, value(succ, α, β))
            if v >= β: return v                  # альфа отсечение
            α=max(α, v)
    return v

def min_value(state, α, β):
    v = +∞
    for succ in successor(state):
            v=min(v, value(succ, α, β))
            if v <= α: return v                  # бета отсечение
            β =min(β, v)
    return v
```

Обратите внимание, что в функциях предусмотрен ранний выход из циклов для случаев бета и альфа отсечений, т.е. полный анализ всех состояний приемников может не выполняться.

Результаты применения альфа-бета поиска зависят от порядка, в котором строятся дочерние вершины. Альфа-бета поиск позволяет увеличить глубину дерева поиска примерно в два раза по сравнению минимаксным алгоритмом, что приводит к более сильной игре [1, 2]. Оценка временной сложности алгоритма альфа-бета поиска соответствует $O(b^{m/2})$.

4.2.5. Функции оценки

В ходе минимаксного поиска процесс генерации ходов останавливают в узлах (состояниях), расположенных на некоторой выбранной глубине. Полезность

этих состояний определяют с помощью тщательно выбранной **функции оценки** (evaluation function), которая дает приближенное значение полезности этих состояний. Чаще всего функция оценки *eval(s)* представляет собой линейную комбинацию признаковых функций $f_i(s)$:

$$eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s),$$

где каждая функция $f_i(s)$ вычисляет некоторую характеристику (признак) состояния s, и каждой характеристике назначается соответствующий вес w_i. Например, в игре в шашки мы могли бы построить оценочную функцию с 4-мя характеристиками: количество пешек у агента, количество королев у агента, количество пешек у противника и количество королев у противника. Структура оценочной функции может быть совершенно произвольной, и она не обязательно должна быть линейной.

4.2.6. Expectimax

Минимаксный поиск бывает чрезмерно пессимистичным в ситуациях, когда оптимальные ответы противника не гарантируются. Для организации поиска в указанных условиях разработан метод, известный как Expectimax.

Expectimax вводит в дерево игры **узлы жеребьевки** (chance node), вместо узлов в которых выбирается минимальная оценка. При этом в узлах жеребьевки вычисляется ожидаемая оценка в виде взвешенного среднего значения. Правило определения ожидаемых значений оценок для узлов жеребьевки выглядит следующим образом:

$$V(s) = \sum_i p(s_i \mid s)\, V(s_i), \qquad (4.1)$$

где $p(s_i|s)$ – условная вероятность того, что данное недетерминированное действие с индексом i приведет к переходу из состояния s в s_i. Суммирование в (4.1) выполняется по всем индексам i.

Минимакс - это частный случай Expectimax: узлы, возвращающие минимальную оценку - это узлы, которые присваивают вероятность 1 своему дочернему узлу с наименьшим значением и вероятность 0 всем другим дочерним узлам. Поэтому псевдокод метода Expectimax очень похож на минимакс, за исключением необходимых изменений, связанных с вычислениями ожидаемой оценки полезности вместо минимальной оценки полезности:

```
def value(state):
    if state является терминальным: return utility(state)
    if  агент MAX: return max_value(state)
    if  агент EXP: return exp_value(state)

def max_value(state):                  # максимальная оценка
    v = -∞
    for succ in successor(state):
            v=max(v, value(succ))
    return v
```

```
def exp_value(state):                    # ожидаемая оценка
    v = 0
    for succ in successor(state):
            p=probability(succ)
            v+=p* value(succ))
    return v
```

Временная сложность Expectimax пропорциональна $O(b^m n^m)$, где n — количество вариантов выпадения жребия.

4.3. Задания для выполнения

Задание 1. 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
```

Не тратьте слишком много времени на усовершенствование решения этого задания, поскольку основные задания лабораторной работы впереди.

Задание 2. 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
```

Подсказки и замечания:

- 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 ближайшее привидение.

Задание 3. 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 сокростью 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.generateSuccesso**r 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.

Для проверки вашего кода выполните команду

```
python autograder.py -q q3
```

Результаты 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 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. 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 отладать 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. 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
```

4.4. Порядок выполнения лабораторной работы

4.4.1. Изучить по лекционному материалу и учебным пособиям [1-3] методы состязательного поиска: минимакс, альфа-бета отсечение и Expectimax.

4.4.2. Использовать для выполнения лабораторной работы файлы из архива **МиСИИ_лаб4_2024.zip.** Программный код этой лабораторной работы не сильно изменился по сравнению с работами 2 и 3, но, тем не менее, разверните его в новой папке и не смешивайте с файлами предыдущих лабораторных работ.

4.4.3. В этой лабораторной работе необходимо реализовать агентов для классической версии Pacman, включая призраков. При этом потребуется реализовать как минимаксный поиск, так и expectimax поиск, а также разработать оценочные функции.

Как и ранее набор файлов включает в себя автооцениватель, с помощью которого вы можете оценивать свои решения. Вызов автооценивателя для проверки всех заданий можно выполнить с помощью команды:

python autograder.py

Для проверки решения конкретного задания, например 2-го, автооцениватель можно вызвать с помощью команды:

python autograder.py -q q2

Для запуска конкретного теста используйте команду вида:

python autograder.py -t test_cases/q2/0-small-tree

По умолчанию автооцениватель отображает графику с параметром **–t**. Можно принудительно включить графику с помощью флага **--graphics** или отключить графику с помощью флага **--no-graphics**.

Код этой лабораторной работы содержит следующие файлы:

Файлы для редактирования:	
`multiAgents.py`	Здесь будут размещаться все ваши мультиагентные поисковые агенты, которых вы определите.
Файлы, которые необходимо просмотреть	
pacman.py	Основной файл, из которого запускают Pacman. Этот файл описывает тип Pacman GameState, который используется в лабораторных работах.
game.py	Логика, лежащая в основе мира Pacman. Этот файл описывает несколько поддерживающих типов, таких как AgentState, Agent, Direction и Grid.
util.py	Полезные структуры данных для реализации алгоритмов поиска.
Поддерживающие файлы, которые можно игнорировать:	
graphicsDisplay.py	Графика Pacman
graphicsUtils.py	Графические утилиты
textDisplay.py	ASCII графика Pacman
ghostAgents.py	Агенты, управляющие привидениями
keyboardAgents.py	Интерфейс клавиатуры для управления игрой
layout.py	Код для чтения файлов схем и хранения их содержимого
autograder.py	Автооцениватель
testParser.py	Парсер тестов автооценивателя и файлы решений
testClasses.py	Общие классы автооценивателя
test_cases/	Папка, содержащая тесты для кадого из заданий (вопросов)
multiagentTestClasses.py	Специальные тестовые классы автооценивателя для данной лабораторной работы

4.4.4. Сначала поиграйте в классический Pacman, выполнив следующую команду:

python pacman.py

Используйте клавиши со стрелками для перемещения. Запустите предоставленный рефлекторный агент **ReflexAgent** в **multiAgents.py**

python pacman.py -p ReflexAgent

Обратите внимание, что он плохо играет даже в случае простых вариантов игры:

python pacman.py -p ReflexAgent -l testClassic

Изучите код рефлекторного агента (в **multiAgents.py**) и убедитесь, что вы понимаете, как он работает. Для этого код рефлекторного агента снабжен необходимыми комментариями.

4.4.5. Выполните задание 1 - усовершенствуйте поведение рефлекторного агента **ReflexAgent.** Используйте подсказки и советы, указанные в задании 1. После вставки кода усовершенствования агента выполните автооценивание задания 1 и результаты внесите в отчет.

4.4.6. Выполните задание 2 - реализуйте минимаксный поиск с произвольным количеством агентом. В основу построения минимаксного агента положите псевдокод, указанный в разделе 4.2.2, а также общие принципы игры с несколькими игроками, изложенные в разделе 4.2.3. При этом обратите внимание на то, что одному действию (ходу) Пакмана на текущем уровне дерева игры будут соответствовать последовательные действия нескольких агентов-призраков. Пакман выбирает действие с максимальной оценкой, призраки – действие с минимальной оценкой. Глубина дерева поиска увеличивается на 1, каждый раз, когда право совершить действие вновь возвращается к Пакману.

Чтобы организовать такой сценарий мультиагентной игры необходимо расширить пседокод функции **min_value(state)** (см. раздел 4.2.2) на случай нескольких агентов-призраков. Для этого функция **min_value** должна обеспечивать реализацию поиска в глубину путем косвенных рекурсивных вызовов функции **value** с дополнительными входными параметрами, задающими индекс агента **agentIndex** и уровень глубины **depth.** При очередном косвенном рекурсивном вызове **min_value** на одном и том же уровне **depth** дерева поиска параметр **agentIndex** должен увеличиваться на 1 (для передачи хода следующему агенту-призраку). Пример псевдокода:

```
def min_value(state, depth, agentIndex):
        …
    v = +∞
    # Для всех допустимых действий агента с номером agentIndex
    for action in gameState.getLegalActions(agentIndex):
        if agentIndex == NumberOfAgents()-1:
            v = min(v, value(successor(state,agentIndex, action),depth + 1, 0))
        else:
            v = min(v, value(successor(state, agentIndex, action), depth, agentIndex+1))
      return v
```

Когда все агенты-призраки (число которых определяется значением **NumberOfAgents)** выполнят свои действия, ход передается Пакману (индекс агента 0) и глубина поиска увеличивается на 1.

После отладки программы выполните автооценивание задания 2 и результаты внесите в отчет.

4.4.7. Выполните задание 3 – реализуйте альфа-бета поиск. Для этого используйте код минимаксного агента, который был реализован в задании 2 и дополните его альфа- и бета-отсечениями, рассмотренными в разделе 4.2.4.

После отладки программы выполните автооценивание задания 3 и результаты внесите в отчет.

4.4.8. Выполните задание 4 – реализуйте класс **ExpectimaxAgent**, который моделирует вероятностное поведение агентов. Для этого используйте код минимаксного агента, в котором замените код функции **min_value() (**см. п.4.4.6) на код функции **exp_value()** . Псевдокод функции приведен в разделе 4.2.6. При реализации функции **exp_value()** используйте тот же перечень параметров, который использует функция **min_value().**

При реализации агента **ExpectimaxAgent** выбор допустимых действий в соответствии с заданием 4 должен быть равновероятным в этом случае формула (4.1) сводится к формуле обычного среднего значения, т.к. $p(s_i \mid s) = 1/N$, N – число состояний-приемников для состояния s.

Код функции **exp_value()** легко получается путем замены в функции **min_value()** строк

```
v = min(v, value(successor(state,agentIndex, action),…)
```

на

```
v += value(successor(agentIndex, action), …)
```

В этом случае в переменной **v** будет накапливаться сумма в соответствии с формулой (4.1). Для вычисления средней оценки необходимо определить количество допустимых действий и поделить указанную сумму **v** на количество действий:

```
v/len(gameState.getLegalActions(agentIndex))
```

После отладки программы агента **ExpectimaxAgent** выполните автооценивание задания 4 и результаты внесите в отчет.

4.4.9. Задание 5 сводится к определению улучшенной версии функции оценки **betterEvaluationFunction** состояния игры. Вы должны придумать эффективные правила вычисления оценок состояний, которые обеспечат более разумное поведение Пакмана. Руководствуйтесь рекомендациями, изложенными в разделе 4.2.5. Начните с анализа недостатков функции **evaluationFunction(self, currentGameState, action)** , которую Вы определяли в задании 1. Теперь функция должна оценивать перспективность состояний. В этом смысле определяемая функция оценки соответствует эвристической функции, применяемой в A*-алгоритме. Перспективность состояний с точки зрения победы Пакмана может быть выражена,

например, в виде дополнительных баллов, начисляемых за ход в сторону расположения ближайшей пищевой гранулы или за ход, в сторону испуганного призрака.

После отладки функции выполните автооценивание задания 5 и результаты внесите в отчет.

4.5. Содержание отчета

Цель работы, краткий обзор методов поиска решений задач в игровых программах, описание свойств методов, код реализованных агентов и функций с комментариями в соответствии с заданиями 1-5, результаты игры на разных полях игры, их анализ, результаты автооценивания заданий, выводы по проведенным экспериментам с разными состязательными агентами.

4.6. Контрольные вопросы

4.6.1 Объясните понятия: детерминированные игры с нулевой суммой, состязательный поиск.

4.6.2. Объясните следующие понятия минимаксного поиска: дерево поиска, статические оценки, динамические оценки, функция полезности.

4.6.3. Объясните на примере принцип минимаксного поиска и запишите формальные выражения, используемые для распространения оценок по дереву поиска.

4.6.4. Разработайте пример дерева игры в крестики-нолики и предложите способ вычисления статических оценок.

4.6.5. Запишите псевдокод, определяющий основные функции минимаксного рекурсивного поиска и объясните его на примере.

4.6.6. Объясните особенности игр с несколькими игроками, приведите пример соответствующего дерева игры и объясните механизм распространения оценок состояний.

4.6.7 Объясните на примере принцип альфа-бета поиска, сформулируйте правила альфа- и бета-отсечений.

4.6.8. Запишите псевдокод, определяющий основные функции альфа-бета поиска и объясните их работу на примере.

4.6.9. Что такое функция оценки? В какой математической форме её обычно определяют?

4.6.10. Почему минимаксный принцип построения агентов, функционирующих в средах с элементами случайности, недостаточно эффективен?

4.6.11. Сформулируйте метод поиска Expectimax, что такое узлы жеребьевки, как в этих узлах вычисляется ожидаемая оценка?

4.6.12. Запишите псевдокод, определяющий основные функции Expectimax поиска и объясните их работу на примере.