**Комп’ютерний практикум №4**

**Методи пошуку в умовах протидії**

**ПІБ: Мєшков Андрій Ігорович, Ткач Владислав Анатолійович**

**Група: ІП-15**

**Мета роботи:** ознайомитись з методами пошуку в умовах протидії та дослідити їх використання для інтелектуального агента в типовому ігровому середовищі.

***З*авдання:** обрати середовище, що моделює гру з нульовою сумою, та задачу, що містить декілька агентів, які протидіють один одному. В обраному середовищі вирішити поставлену задачу, реалізувавши один з методів пошуку в умовах протидії. Реалізувати власну функцію оцінки станів. Виконати дослідження впливу деякого фактору середовища.

**Номер варіанту: 23**

**Завдання для варіанту:** обрати середовище, що моделює гру з нульовою сумою, та задачу, що містить декілька агентів, які протидіють один одному. В обраному середовищі вирішити поставлену задачу, реалізувавши Альфа-бета відсікання. Реалізувати власну функцію оцінки станів. Виконати дослідження впливу Вплив глибини пошуку.

**Середовище і задача:**

Обране середовище для вирішення завдання - гра "Pacman" (імітаційна версія аркадної гри). У цій грі гравець керує Пакменом, який має збирати їжу в лабіринті, уникаючи зіткнень з привидами, що рухаються. Привиди намагаються зійти на Пакмена і втратити його. Гра має нульову суму, оскільки виграш одного гравця (наприклад, Пакмена) відповідає програш іншого (привидів) і навпаки.

Формалізація середовища:

* Агенти: Пакмен та привиди.
* Стани: конфігурація лабіринту та розташування Пакмена та привидів.
* Можливі дії агентів: для Пакмена - рух у чотирьох напрямках (вгору, вниз, вліво, вправо), для привидів - рух у напрямку Пакмена або рух у випадковому напрямку.
* Переходи: кожен агент робить крок у межах лабіринту, і гра переходить до наступного стану.
* Винагороди: Пакмен отримує винагороду за кожну зібрану одиницю їжі, а привиди отримують винагороду за з'їдання Пакмена.

**Метод вирішення задачі:** Метод альфа-бета відсікання є ефективним алгоритмом для пошуку оптимального рішення в деревоподібних структурах, таких як гри. Він призначений для випадків, коли можливі рішення представляються у вигляді дерева розгалужень (наприклад, граф пошуку), і ці рішення оцінюються за допомогою певної функції оцінки.

Основна ідея методу альфа-бета відсікання полягає в тому, щоб зменшити кількість вузлів, які потрібно оцінювати у дереві, шляхом відсікання піддерев, які точно не впливають на кінцевий результат. Це досягається завдяки двом параметрам, що використовуються для керування процесом обходу дерева:

1. Альфа (α) - найкращий (найбільший) варіант, який вже було знайдено для певного гравця або рівня глибини.

2. Бета (β) - найгірший (найменший) варіант, який дозволяє опоненту гравця або рівня глибини.

Процес відсікання відбувається наступним чином:

- Коли варіант перевищує або дорівнює значенню β, його можна відкинути, оскільки опонент не буде вибирати такий варіант.

- Коли варіант менше або дорівнює значенню α, він не буде вибрано гравцем, оскільки існує кращий варіант.

Цей метод дозволяє значно зменшити кількість варіантів, які потрібно розглядати, що робить його ефективним для розв'язання складних задач, таких як ігри з великою кількістю можливих ходів.

**Реалізація методу:**

class AlphaBetaAgent(MultiAgentSearchAgent):

def getAction(self, gameState: GameState) -> str:

def alphaBeta(state, depth, alpha, beta, agentIndex):

*if* state.isWin() or state.isLose() or depth == *self*.depth:

*return* *self*.evaluationFunction(state)

*if* agentIndex == 0:

*return* maxValue(state, depth, alpha, beta)

*else*:

*return* minValue(state, depth, alpha, beta, agentIndex)

def maxValue(state, depth, alpha, beta):

v = float('-inf')

actions = state.getLegalActions(0)

*if* not actions:

*return* *self*.evaluationFunction(state)

*for* action *in* actions:

successor = state.generateSuccessor(0, action)

v = max(v, alphaBeta(successor, depth, alpha, beta, 1))

*if* v >= beta:

*return* v

alpha = max(alpha, v)

*return* v

def minValue(state, depth, alpha, beta, agentIndex):

v = float('inf')

nextAgent = (agentIndex + 1) % state.getNumAgents()

nextDepth = depth + 1 *if* nextAgent == 0 *else* depth

actions = state.getLegalActions(agentIndex)

*if* not actions:

*return* *self*.evaluationFunction(state)

*for* action *in* actions:

successor = state.generateSuccessor(agentIndex, action)

v = min(v, alphaBeta(successor, nextDepth, alpha, beta, nextAgent))

*if* v <= alpha:

*return* v

beta = min(beta, v)

*return* v

bestAction = None

alpha = float('-inf')

beta = float('inf')

bestScore = float('-inf')

*for* action *in* gameState.getLegalActions(0):

successor = gameState.generateSuccessor(0, action)

score = alphaBeta(successor, 0, alpha, beta, 1)

*if* score > bestScore:

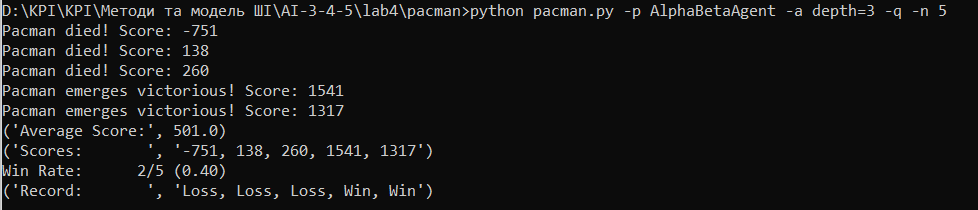
bestScore = score

bestAction = action

alpha = max(alpha, bestScore)

*return* bestAction

**Результати застосування розробленого методу:**

****

**Оцінка результатів:**

Із результатів можемо спостерігати, що алгоритм AlphaBetagent показав відносно середні результати, враховуючи низьку частоту перемог та великі відхилення в рахунках.

Можна зробити висновок що алгоритм на є доволі стабільним в прийнятті рішень та на завжди здатний приймати оптимальні рішення, що може бути пов’язано із глибиною для тестування (3), далі буде зроблено дослідження впливу глибини на якість.

**Власна функція оцінки станів:**

Реалізація функції betterEvaluationFunction, для оцінки поточного стану гри pacman, використовуючи різноманітні параметри та стани.

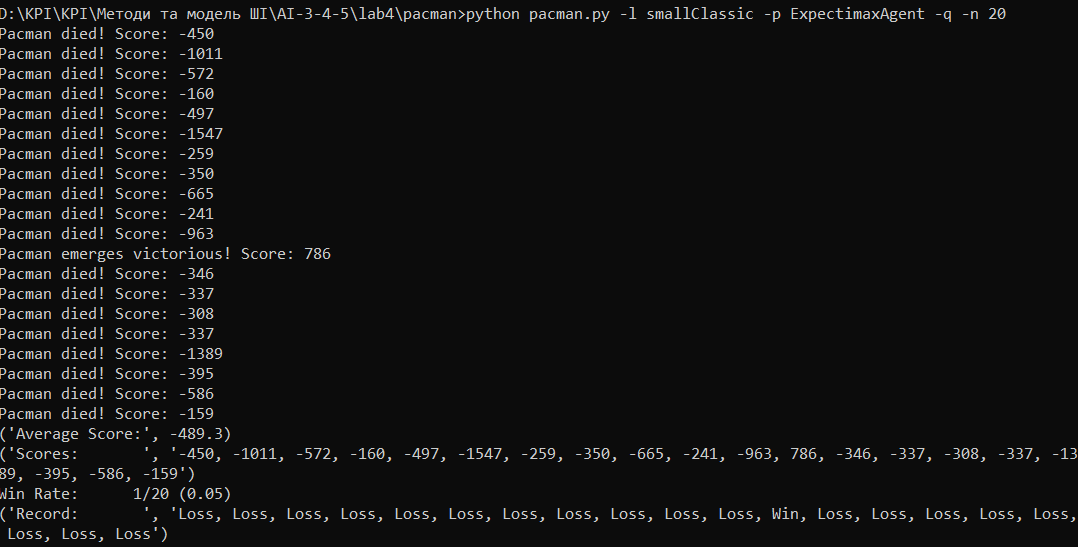
Функція перевіряє, чи є поточний стан переможним або програшним, і відповідно повертає нескінченний позитивний або негативний бал. Вона також враховує позицію pacman, рахунок гри, відстань до найближчої їжі, кількість залишеної їжі та капсул, а також відстань до привидів (як активних, так і наляканих), розраховуючи зважену суму цих факторів, де позитивні значення заохочують збір їжі та капсул, а також уникнення активних привидів і переслідування наляканих привидів. Таким чином, функція надає комплексну оцінку стану гри, що сприяє кращим стратегіям для pacman.

**Реалізація власної функції оцінки станів:**

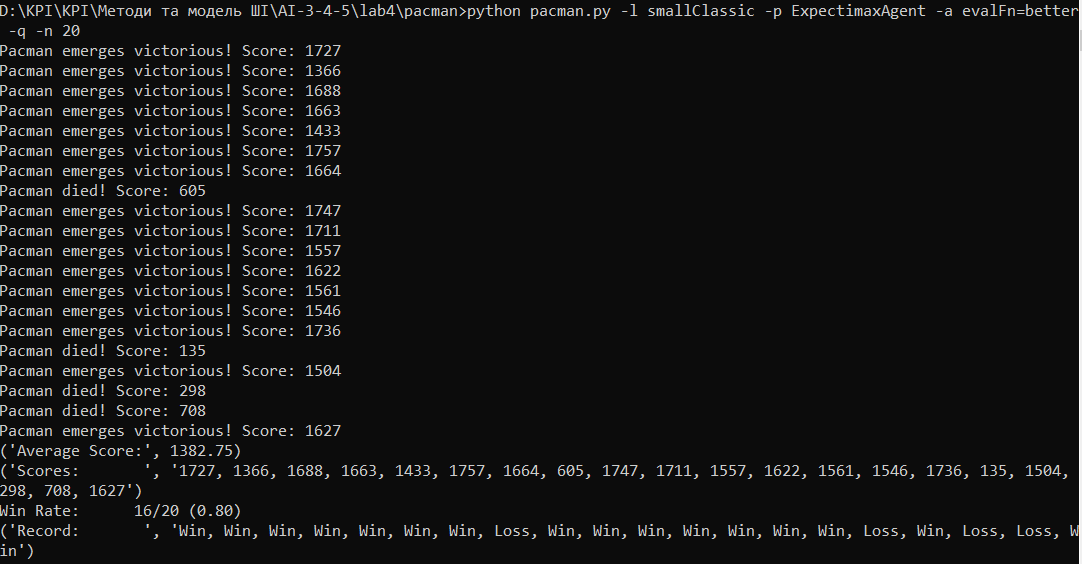
def betterEvaluationFunction(currentGameState: GameState) -> float:  
 *"""  
 Your extreme, unstoppable evaluation function (problem 4). Note that you can't fix a seed in this function.  
 """* # BEGIN\_YOUR\_CODE (our solution is 13 lines of code, but don't worry if you deviate from this)  
 if currentGameState.isLose():  
 return -float("inf")  
 elif currentGameState.isWin():  
 return float("inf")  
 pos = currentGameState.getPacmanPosition()  
 currentScore = scoreEvaluationFunction(currentGameState)  
 foodPos = currentGameState.getFood().asList()  
 manhattanDistanceToClosestFood = min(map(lambda x: manhattanDistance(pos, x), foodPos))  
 distanceToClosestFood = manhattanDistanceToClosestFood  
 numberOfFoodsLeft = len(foodPos)  
 numberOfCapsulesLeft = len(currentGameState.getCapsules())  
  
 scaredGhosts, activeGhosts = [], []  
 for ghost in currentGameState.getGhostStates():  
 if not ghost.scaredTimer:  
 activeGhosts.append(ghost)  
 else:  
 scaredGhosts.append(ghost)  
  
 def getManhattanDistances(ghosts):  
 return map(lambda g: util.manhattanDistance(pos, g.getPosition()), ghosts)  
  
 distanceToClosestActiveGhost = distanceToClosestScaredGhost = 0  
  
 if activeGhosts:  
 distanceToClosestActiveGhost = min(getManhattanDistances(activeGhosts))  
 else:  
 distanceToClosestActiveGhost = float("inf")  
 distanceToClosestActiveGhost = max(distanceToClosestActiveGhost, 5)  
  
 if scaredGhosts:  
 distanceToClosestScaredGhost = min(getManhattanDistances(scaredGhosts))  
 else:  
 distanceToClosestScaredGhost = 0  
  
 score = 1 \* currentScore + \  
 -1.5 \* distanceToClosestFood + \  
 -2 \* (1. / distanceToClosestActiveGhost) + \  
 -3 \* distanceToClosestScaredGhost + \  
 -10 \* numberOfCapsulesLeft + \  
 -5 \* numberOfFoodsLeft  
 return score  
  
 # END\_YOUR\_CODE  
  
  
# Abbreviation  
better = betterEvaluationFunction

Протестуємо алгоритм відносно звичайної функції та покращеної:

Результати звичайної:



Результати покращеної:



Із результатів бачимо що покращена функція працює набагато краще у порівнянні із звичайною.

**Задача дослідження впливу параметра алгоритму чи фактору середовища:**

Проведемо дослідження параметра глибини на якість роботи алгоритму AlphaBeta:

Провівши кілька експериментів, отримали таблицю результатів(для кожного параметру Depth проводилось 50 випробувань):

|  |  |  |
| --- | --- | --- |
| Depth | Avg score | Win rate |
| 1 | -411,98 | 0,02 |
| 2 | 11,16 | 0,08 |
| 3 | 454,12 | 0,24 |
| 4 | 648,12 | 0,32 |
| 5 | 581,28 | 0,32 |

Із результатів можемо зробити висновок, що відсоток перемог збільшується із збільшення параметру глибини алгоритму, але після глибини в 4 показники майже не змінились.

**Висновок:**

У нашій роботі ми дослідили методи пошуку в умовах протидії та їх використання для інтелектуального агента в грі "pacman".

Ми реалізували алгоритм альфа-бета відсікання для знаходження оптимальних ходів pacma. Результати показали, що наш алгоритм має середній рівень успішності, що вказує на нестабільність в прийнятті рішень, що може бути пов'язано з недостатньою глибиною пошуку.

Також покращили функцію оцінки станів гри використовуючи алгоритм ExpectimaxAgent, підхід включав класифікацію станів гри як переможних або програшних, врахування позиції pacman, відстані до їжі, капсул і привидів. Визначили зважену суму цих факторів, де позитивні значення заохочують збір їжі та капсул, а також уникнення активних привидів і переслідування наляканих привидів, що дозволило нашому агенту приймати більш комплексні рішення під час гри.

Дослідження впливу глибини пошуку показало, що зі збільшенням глибини покращується якість роботи алгоритму, однак після досягнення глибини 4 покращення стає незначним. Це свідчить про те, що збільшення глибини пошуку не завжди призводить до значного покращення результатів і може потребувати оптимізації інших параметрів алгоритму.

Отже, результати підтвердили ефективність методів в умовах протидії та важливість розробки якісної функції оцінки станів для покращення стратегій гри pacman.