Звіт до комп'ютерного практикуму №4.

Методи пошуку в умовах протидії

ПІБ: Дмитрієва Ірина Ігорівна
Група: ІТ-01

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: 37
Завдання для варіанту:

37	Альфа-бета відсікання	Порівняння з MiniMax для випадкових привидів

Середовище і задача:
У даній лабораторній роботі ми працюємо з грою "Пакмен". Гравець виступає в ролі агента, і його завданням є рухатись по рівню у одному з чотирьох напрямків, збирати всі капсули з їжею і уникати зіткнень з привидами. Стан середовища визначається позиціями гравця, привидів та капсул. З кожного стану агент може перейти в один з доступних станів (максимум - 4), переміщаючись вгору, вниз, вліво або вправо. Кожен крок агента супроводжується штрафом в 1 бал, що мотивує гравця збирати капсули якнайшвидше. У разі зіткнення з привидом, гравець отримує штраф в 500 балів, а за кожну зібрану капсулу агент отримує бонус в 10 балів.

Метод вирішення задачі:
Альфа-бета відсікання є методом оптимізації для алгоритму мінімаксу. Його суть полягає в тому, щоб не витрачати час на обчислення станів, які точно не будуть вибрані алгоритмом. У випадку ходу гравця, це означає стани, які мають оцінку меншу за поточне максимальне значення. У випадку ходу противника, навпаки, це стани, які мають оцінку меншу за поточне мінімальне значення. Для отримання початкового максимального і мінімального значення алгоритм повинен прорахувати хоча б одну послідовність ходів до кінця.

Реалізація методу:

```python
class AlphaBetaAgent(MultiAgentSearchAgent):
    def alpha_beta(self, gameState, agentIndex, iterationsLeft, maxValue, minValue):
        agentsCount = gameState.getNumAgents()
        legalMoves = gameState.getLegalActions(agentIndex)

        if len(legalMoves) == 0:
            return self.evaluationFunction(gameState), 0

        successors = [gameState.generateSuccessor(agentIndex, action) for action in legalMoves]
        scores = [self.evaluationFunction(state) for state in successors]

        if agentIndex >= agentsCount - 1 and iterationsLeft <= 1:
            bestScore = min(scores)
            bestScoreIndexes = [index for index in range(len(scores)) if scores[index] == bestScore]
            chosenIndex = random.choice(bestScoreIndexes)
            return bestScore, chosenIndex
        else:
            currentValue = self.evaluationFunction(gameState)
            isGhost = agentIndex != 0

            if agentIndex == 0:
                if currentValue < maxValue:
                    return currentValue, None
                elif isGhost and currentValue > minValue:
                    return currentValue, None
            elif isGhost and currentValue > minValue:
                return currentValue, None

            if agentIndex != agentsCount - 1:
                pass
            else:
                if isGhost:
                    minValue = currentValue
                else:
                    maxValue = currentValue

            if agentIndex >= agentsCount - 1:
                newIterationsLeft = iterationsLeft - 1
                newAgentIndex = 0
            else:
                newAgentIndex = agentIndex + 1
                newIterationsLeft = iterationsLeft

            scores = [
                self.alpha_beta(state, newAgentIndex, newIterationsLeft, maxValue, minValue)[0]
                for state in successors
            ]

            bestScore = max(scores) if agentIndex == 0 else min(scores)
            bestScoreIndexes = [index for index in range(len(scores)) if scores[index] == bestScore]
            chosenIndex = random.choice(bestScoreIndexes)
            return scores[chosenIndex], chosenIndex

    def getAction(self, gameState):
        start = time.time()
        _, chosenIndex = self.alpha_beta(gameState, 0, self.depth, -99999, 99999)
        end = time.time()
        times.append(end - start)
        print("Average time:", sum(times) / len(times))
        return gameState.getLegalActions(0)[chosenIndex]
```

Результати застосування розробленого методу:

Конфігурація запуску середовища:

- Глибина прорахунку: 3
- Кількість привидів: 2
- Конфігурація привидів: випадкова
- Рівень: класичний
- Параметри: `-p AlphaBetaAgent -a depth=3 -n 20 -k 2 -q`

Результат роботи програми:

pacman % python pacman.py -p AlphaBetaAgent -a depth=3 -n 20 -k 2 -q
Pacman emerges victorious! Score: 1482
Pacman emerges victorious! Score: 1596
Pacman emerges victorious! Score: 1227
Pacman emerges victorious! Score: 1833
Pacman emerges victorious! Score: 1465
Pacman emerges victorious! Score: 1551
Pacman emerges victorious! Score: 1667
Pacman emerges victorious! Score: 1149
Pacman emerges victorious! Score: 1357
Pacman died! Score: 13
Pacman emerges victorious! Score: 1516
Pacman died! Score: 436
Pacman emerges victorious! Score: 1705
Pacman emerges victorious! Score: 1732
Pacman emerges victorious! Score: 1477
Pacman died! Score: 872
Pacman emerges victorious! Score: 1472
Pacman died! Score: 593
Pacman emerges victorious! Score: 1893
Pacman emerges victorious! Score: 1133
('Average Score:', 1308.45)
('Scores: ', '1482, 1596, 1227, 1833, 1465, 1551, 1667, 1149, 1357, 13, 1516, 436, 1705, 1732, 1477, 872, 1472, 593, 1893, 1133')

Win Rate: 16/20 (0.80)
('Record: ', 'Win, Win, Win, Win, Win, Win, Win, Win, Loss, Win, Loss, Win, Win, Win, Loss, Win, Loss, Win, Win')

Оцінка результатів:

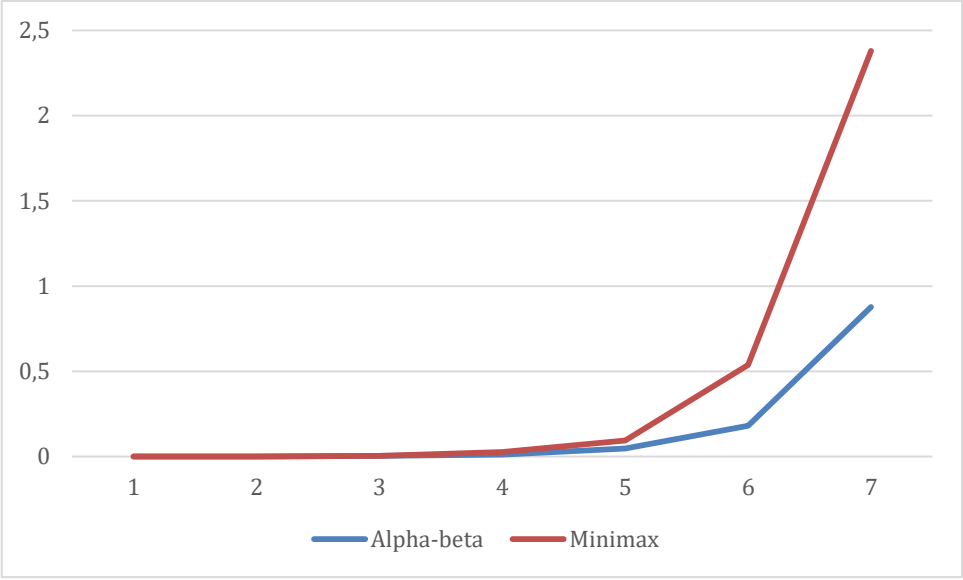
Основна перевага альфа-бета відсікання полягає в тому, що алгоритм працює швидше за звичайний мінімакс, не втрачаючи при цьому ефективності. Ця ефективність стає особливо помітною на більшій глибині прорахунку, оскільки алгоритм відсікає непотрібні гілки. Завдяки оптимізації можна прораховувати більше ходів і отримувати кращі результати.

Однак основною проблемою алгоритму є його обмежений "поле зору". Якщо агент не може досягти капсули чи привида протягом всієї глибини ходів, він може "зациклитись" і почати робити випадкові кроки, не знаходячи вихід з цього стану. Цю проблему можна вирішити, покращивши функцію оцінки стану.

Також може виникнути ситуація, коли агент вважає противника розумнішим, ніж він є. Наприклад, якщо ситуація схожа на оточення, але пакмен має можливість вибратися за рахунок випадкових рухів привида, агент може не розглянути цю можливість, вважаючи, що противник обере оптимальну стратегію з точки зору агента. В такому випадку агент може намагатись швидше завершити гру, кидаючись на привидів, щоб отримувати менше штрафів за хід, навіть якщо в у нього є шанси вибратися.

Найбільш очевидну недолік цього алгоритму можна побачити на рівні "trappedClassic", де агент завжди буде рухатись вправо, незважаючи на те, що синій привид може випадково рухатись в зворотному напрямку.

З урахуванням цих недоліків, використання альфа-бета відсікання не є дуже доцільним в даній задачі. Для цього краще підійде алгоритм Expectimax або подібні до нього за принципом алгоритми.



Як видно з результатів, альфа-бета відсікання на глибоких рівнях прорахунку забезпечує значну перевагу у швидкості. Ця перевага має важливе значення, особливо якщо ми обмежені у часі. Швидший алгоритм здатен прорахувати більшу глибину і вибрати більш точну та ефективну стратегію.

Висновки:

Під час виконання лабораторної роботи ми ознайомились з методами пошуку в умовах протидії та дослідили їх застосування для створення інтелектуального агента в типовому ігровому середовищі. Ми реалізували інтелектуального агента для гри "Пакмен" за допомогою альфа-бета відсікання, поліпшили його за допомогою більш вдосконаленої функції оцінки та дослідили переваги альфа-бета відсікання над мінімаксом.