Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота № 2

з дисципліни «Програмування інтелектуальних інформаційних систем »

Тема: «Дерева прийняття рішень 101»

Виконав:
студент групи ІТ-04
Коновальчук Андрій
Дата здачі _____
Захищено з балом _____

Перевірив:
вик. кафедри ІПІ
Баришич Лука Маріянович

Київ 2022

Лабораторна №2

Тема:

Дерева прийняття рішень 101

Мета:

Розробити програмне рішення дерев прийняття рішень для гри Pacman

Завдання:

1. Додати генерацію ворогів з поведінкою. При генерації ви вказуєте кількість згенерованих ворогів одного з двох типів. Перший тип шукає дорогу до гравця (оптимальність цього пошуку на вашій совісті). Другий тип рухається випадково

2. Ви маєте реалізувати алгоритми перемоги за вашою грою - minimax з alpha-beta pruning та expectimax. Функція оцінки має оцінювати “силу” поточної позиції - чим більше число, тим краща позиція.

Опис програмного коду

Програмний код на GitHub: https://github.com/KonovalchukA-IT04/PiisLabs/tree/master/lab2

Програмний код реалізований на базі проекту Berkeley C188 Pacman, де наперед підготована гра, візуалізація, а також архітектура, яка дозволяє розширити і доповнити код. Тож завдання лабораторної роботи зводилось до написання алгоритмів minimax, alpha-beta pruning та expectimax, функції для визначення евристики. Весь код в рамках даної лабораторної написаний у файлі multiAgents.py.

1. RandomGhost, DirectionalGhost

Дані методи були передбачені курсом Berkeley, і не потребують нашого редагування. Методи передбачають поведінку привидів з випадкового руку та переслідування ПакМена. Для виклику відповідної поведінки потрібно вказати відповідний параметр (назва методу) з прапорцем -g при запуску гри.

2. Дерева прийняття рішень

2.1. MiniMax

Мінімакс – правило прийняття рішень, що використовується в теорії ігор, теорії прийняття рішень, дослідженні операцій, статистиці і філософії для мінімізації можливих втрат з тих, які особа, яка приймає рішення не може уникнути при розвитку подій за найгіршим для неї сценарієм. У даній лабораторній роботі ми провели максимізацію дій Пакмена та мінімізацію дій Привидів.

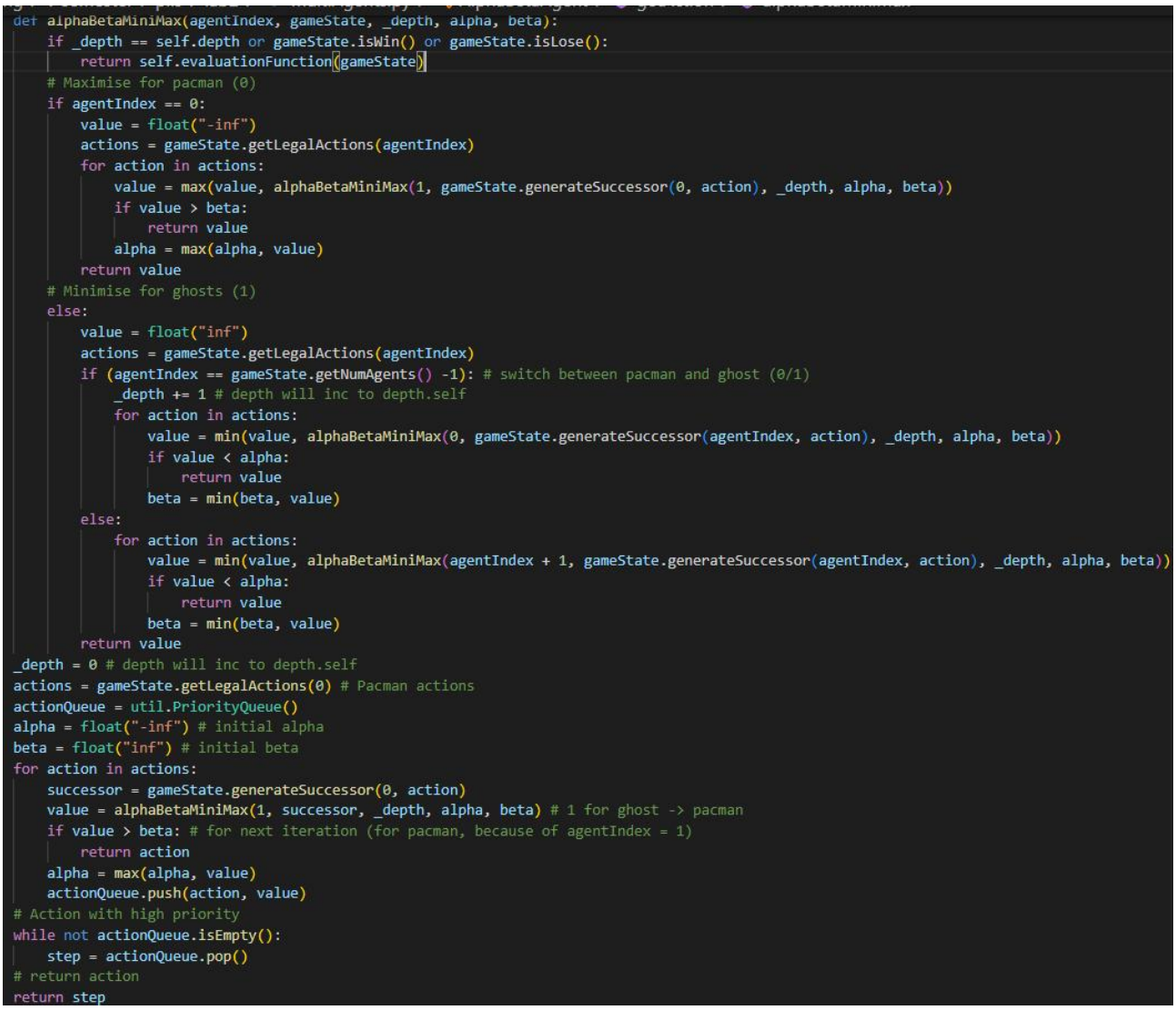
```
def miniMax(agentIndex, gameState, _depth):
    if _depth == self.depth or gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    # Maximise for pacman (0)
    if agentIndex == 0:
        value = float("-inf")
        actions = gameState.getLegalActions(agentIndex)
        for action in actions:
            value = max(value, miniMax(1, gameState.generateSuccessor(0, action), _depth))
        return value
    # Minimise for ghosts (1)
    else:
        value = float("inf")
        actions = gameState.getLegalActions(agentIndex)
        if (agentIndex == gameState.getNumAgents() -1): # switch between pacman and ghost (0/1)
            _depth += 1 # depth will inc to depth.self
            for action in actions:
                value = min(value, miniMax(0, gameState.generateSuccessor(agentIndex, action), _depth))
        else:
            for action in actions:
                value = min(value, miniMax(agentIndex + 1, gameState.generateSuccessor(agentIndex, action), _depth))
        return value
_depth = 0 # depth will inc to depth.self
actions = gameState.getLegalActions(0) # Pacman actions
actionQueue = util.PriorityQueue()
for action in actions:
    successor = gameState.generateSuccessor(0, action)
    value = miniMax(1, successor, _depth) # 1 for ghost -> pacman
    actionQueue.push(action, value)
# Action with high priority
while not actionQueue.isEmpty():
    step = actionQueue.pop()
# return action
return step
```

Рисунок 2.1.1 – Код алгоритму

Функцію ми зробили монолітною, і не розбивали на 2-3 (максимізація, мінімізація, логіка). “Перемикання” між мінімізацією і максимізацію відбувається через індексацію агентів agentIndex (0 – Пакмен, >1 – Привиди), яка необхідна також для отримання відповідних дій (actions) із стану гри (gameState). Повертається значення еврестичної функції (evaluationFunction, котра буде розглянута пізніше), котра є пріоритетом для розглянутих перед викликом функції minimax дій (actions), і визначає в кінці виклику методу getActions агенту найкращі дії для Пакмена. Розгляд глубини працює від 0 до вказаного у виклику програми максимуму (self.depth).

2.2. Alpha-betapruning

Алгоритм пошуку, що зменшує кількість вузлів, які необхідно оцінити в дереві пошуку мінімаксного алгоритму і при цьому дозволяє отримати ідентичний результат.



```
def alphaBetaMiniMax(agentIndex, gameState, _depth, alpha, beta):
    if _depth == self.depth or gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    # Maximise for pacman (0)
    if agentIndex == 0:
        value = float("-inf")
        actions = gameState.getLegalActions(agentIndex)
        for action in actions:
            value = max(value, alphaBetaMiniMax(1, gameState.generateSuccessor(0, action), _depth, alpha, beta))
            if value > beta:
                return value
            alpha = max(alpha, value)
        return value
    # Minimise for ghosts (1)
    else:
        value = float("inf")
        actions = gameState.getLegalActions(agentIndex)
        if (agentIndex == gameState.getNumAgents() -1): # switch between pacman and ghost (0/1)
            _depth += 1 # depth will inc to depth.self
            for action in actions:
                value = min(value, alphaBetaMiniMax(0, gameState.generateSuccessor(agentIndex, action), _depth, alpha, beta))
                if value < alpha:
                    return value
                beta = min(beta, value)
        else:
            for action in actions:
                value = min(value, alphaBetaMiniMax(agentIndex + 1, gameState.generateSuccessor(agentIndex, action), _depth, alpha, beta))
                if value < alpha:
                    return value
                beta = min(beta, value)
        return value
_depth = 0 # depth will inc to depth.self
actions = gameState.getLegalActions(0) # Pacman actions
actionQueue = util.PriorityQueue()
alpha = float("-inf") # initial alpha
beta = float("inf") # initial beta
for action in actions:
    successor = gameState.generateSuccessor(0, action)
    value = alphaBetaMiniMax(1, successor, _depth, alpha, beta) # 1 for ghost -> pacman
    if value > beta: # for next iteration (for pacman, because of agentIndex = 1)
        return action
    alpha = max(alpha, value)
    actionQueue.push(action, value)
# Action with high priority
while not actionQueue.isEmpty():
    step = actionQueue.pop()
# return action
return step
```

Рисунок 2.2.1 – Код алгоритму

Код алгоритму майже аналогічний попередньому. Відрізняється лише додатковою логікою “відсікань”, а також додатковими змінним (alpha, beta), необхідними для роботи цього алгоритму.

2.3 Expectimax

Алгоритм expectiminimax є різновидом алгоритму minimax для використання в системах штучного інтелекту, які грають у ігри з нульовою сумою для двох гравців, такі як нарди, в яких результат залежить від комбінації навичок гравця та елементів випадковості, таких як кидання кубиків. На додаток до вузлів “min” і “max” традиційного мінімаксного дерева, цей варіант має вузли “chance” (“move by nature”), які приймають очікуване значення випадкової події, що відбувається.

```
def expectiMax(agentIndex, gameState, _depth):
    if _depth == self.depth or gameState.isWin() or gameState.isLose():
        return self.evaluationFunction(gameState)
    # Maximise for pacman (0)
    if agentIndex == 0:
        value = float("-inf")
        actions = gameState.getLegalActions(agentIndex)
        for action in actions:
            value = max(value, expectiMax(1, gameState.generateSuccessor(0, action), _depth))
        return value
    # Minimise for ghosts (1)
    else:
        value = 0
        actions = gameState.getLegalActions(agentIndex)
        if (agentIndex == gameState.getNumAgents() -1): # switch between pacman and ghost (0/1)
            _depth += 1 # depth will inc to depth.self
            for action in actions: # values sum
                value += expectiMax(0, gameState.generateSuccessor(agentIndex, action), _depth)
        else:
            for action in actions: # values sum
                value += expectiMax(agentIndex + 1, gameState.generateSuccessor(agentIndex, action), _depth)
        return value/len(actions) # avarage value
_depth = 0 # depth will inc to depth.self
actions = gameState.getLegalActions(0) # Pacman actions
betterAction = Directions.STOP # initial best action
bettervalue = float("-inf") # initial better value
for action in actions:
    successor = gameState.generateSuccessor(0, action)
    value = expectiMax(1, successor, _depth) # 1 for ghost -> pacman
    if value > bettervalue:
        bettervalue = value
        betterAction = action
# return action
return betterAction
```

Рисунок 2.3.1 – Код алгоритму

Код майже аналогічний до коду мінімаксу. Під час мінімізації ми вираховуємо середнє арифметичне значення (сумуємо value і ділимо на довжину контейнеру actions). Та порівнюємо знайдені значення для кожного actions з “кращими значенням” на попередніх ітераціях.

2.4 Евристика

Для наших попередніх функцій треба було прописати підходящу евристику для Пакмена.

```
# aka Score
evalNum = 0

# Ghost eval
listOfGhostDist = []
closestGhost = 0
for ghost in range(len(newGhostStates)): # range(len()) for indexation
    ghostPos = successorGameState.getGhostPositions()[ghost] # indexation of ghosts
    listOfGhostDist.append(manhattanDistance(newPos, ghostPos))
if listOfGhostDist != []:
    closestGhost = min(listOfGhostDist) # min distance

# Food eval
listOfFoodDist = []
closestFood = 0
for food in newFood.asList():
    listOfFoodDist.append(manhattanDistance(newPos, food))
if listOfFoodDist != []:
    closestFood = min(listOfFoodDist) # min distance

# Capsule eval
capsules = currentGameState.getCapsules()
listOfCapsDist = []
closestCapsule = 0
for capsule in capsules:
    listOfCapsDist.append(manhattanDistance(newPos, capsule))
if listOfCapsDist != []:
    closestCapsule = min(listOfCapsDist) # min distance

# Eval calc
closestScaredGhostIndex = listOfGhostDist.index(closestGhost)
foodCost = (1/(closestFood + 1)) + (1/(len(listOfFoodDist) + 1)) # min distance + count (less dots - higher score) conv to <1
capsuleCost = (1/(closestCapsule + 1)) + (1/(len(listOfCapsDist) + 1)) # same
ghostCost = (1/(closestGhost + 1)) # min distance conv to <1
if closestGhost>1:
    evalNum += foodCost + capsuleCost + successorGameState.getScore()
    if newScaredTimes[closestScaredGhostIndex]>1: # if scared --> + ghost cost
        evalNum += ghostCost
    else: # else --> - ghost cost
        evalNum += -ghostCost
else:
    evalNum = -1 # if too close -- run away
return evalNum
```

Рисунко 2.4.1 – Код еврестичної функції

В якості евристики ми вираховуємо обернені значення відстані до найближчої їжі, кількості їжі, відстані до найближчої капсули, кількості капсул, відстані до найближчого Привида. Якщо Привид наляканий, то приплюсовуємо додатнє значення, якщо не наляканий, то віднімаємо. Якщо привид занадто близько (відстань <1), то повертаємо від’ємне значення, як поганий вибір для наступного руху.

Висновок

В ході даної лабораторної роботи було створено програмне рішення дерев прийняття рішення для найоптимальнішого руху Пакмена в лабіринті.

Було реалізовано наступні алгоритми: minimax, alpha-beta pruning, expectimax; також реалізовано еврестичну функцію для цих алгоритмів.

Лабораторна робота виконана на основі проекту Berkeley C188 Pacman. Розібрано структуру та архітектуру проекту, і доповнені наступні файли: multiAgents.py.

В звіті наявні описи алгоритмів, а також скріншоти з кодом.