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

Лабораторна робота № 3

з дисципліни «Програмування інтелектуальних інформаційних систем »

Тема: «Дерева прийняття рішень 102»

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

Перевірив:
вик. кафедри ІПІ
Баришич Лука Маріянович

Київ 2022

Лабораторна №3

Тема:

Дерева прийняття рішень 102

Мета:

Розробити програмне рішення дерев прийняття рішень для гри в Шахи

Завдання:

1. Реалізувати алгоритм NegaMax
2. Реалізувати алгоритм NegaScout
3. Реалізувати алгоритм PVS

Опис програмного коду

Програмний код на GitHub: https://github.com/KonovalchukA-IT04/PiisLabs/tree/master/lab3

Програмний код реалізований на базі бібліотеки python-chess, де наперед підготована гра, візуалізація у вигляді вимальовування дошки у форматі SVG, а також API-інтерфейс, який дозволяє рухати фігурами та слідкувати за станом дошки. Також була використана бібліотека-надбудова python-chess-board, створена на pygame, яка дозволяє візуалізовувати дошку та слідкувати за грою. Тож завдання лабораторної роботи зводиться до написання алгоритмів NegaMax, NegaScout та PVS, функції для визначення евристики. Алгоритми протестовані і візуалізовані на грі двох ботів (чорні та білі) один проти одного.

1. Евристика

Еврестична функція (рис. 1.1) взята з відкритих джерел (статті на програмування ботів для шах) і модифікована під нашу лабораторну роботу.

Принцип простий: умови виходу/кінця гри; набір оцінок для кожної фігури; обрахунок кількості фігур; обрахунок евристики за формулою за кількістю фігур; обрахунок евристики за формолою для кожної фігури окремо; сума всіх евристик; зміна знаку в залежності від граючої сторони.

```
def evaluation(board):

    if board.is_checkmate():
        if board.turn:
            return -9999
        else:
            return 9999
    if board.is_stalemate():
        return 0
    if board.is_insufficient_material():
        return 0

    wp = len(board.pieces(chess.PAWN, chess.WHITE))
    bp = len(board.pieces(chess.PAWN, chess.BLACK))
    wn = len(board.pieces(chess.KNIGHT, chess.WHITE))
    bn = len(board.pieces(chess.KNIGHT, chess.BLACK))
    wb = len(board.pieces(chess.BISHOP, chess.WHITE))
    bb = len(board.pieces(chess.BISHOP, chess.BLACK))
    wr = len(board.pieces(chess.ROOK, chess.WHITE))
    br = len(board.pieces(chess.ROOK, chess.BLACK))
    wq = len(board.pieces(chess.QUEEN, chess.WHITE))
    bq = len(board.pieces(chess.QUEEN, chess.BLACK))

    material = 100 * (wp - bp) + 320 * (wn - bn) + 330 * (wb - bb) + 500 * (wr - br) + 900 * (wq - bq)
    pawnsq = sum([pieces.pawntable[i] for i in board.pieces(chess.PAWN, chess.WHITE)])
    pawnsq = pawnsq + sum([-pieces.pawntable[chess.square_mirror(i)]
                           for i in board.pieces(chess.PAWN, chess.BLACK)])
    knightsq = sum([pieces.knightstable[i] for i in board.pieces(chess.KNIGHT, chess.WHITE)])
    knightsq = knightsq + sum([-pieces.knightstable[chess.square_mirror(i)]
                               for i in board.pieces(chess.KNIGHT, chess.BLACK)])
    bishopsq = sum([pieces.bishopstable[i] for i in board.pieces(chess.BISHOP, chess.WHITE)])
    bishopsq = bishopsq + sum([-pieces.bishopstable[chess.square_mirror(i)]
                               for i in board.pieces(chess.BISHOP, chess.BLACK)])
    rooksq = sum([pieces.rookstable[i] for i in board.pieces(chess.ROOK, chess.WHITE)])
    rooksq = rooksq + sum([-pieces.rookstable[chess.square_mirror(i)]
                           for i in board.pieces(chess.ROOK, chess.BLACK)])
    queensq = sum([pieces.queenstable[i] for i in board.pieces(chess.QUEEN, chess.WHITE)])
    queensq = queensq + sum([-pieces.queenstable[chess.square_mirror(i)]
                             for i in board.pieces(chess.QUEEN, chess.BLACK)])
    kingsq = sum([pieces.kingstable[i] for i in board.pieces(chess.KING, chess.WHITE)])
    kingsq = kingsq + sum([-pieces.kingstable[chess.square_mirror(i)]
                           for i in board.pieces(chess.KING, chess.BLACK)])

    eval = material + pawnsq + knightsq + bishopsq + rooksq + queensq + kingsq
    if board.turn:
        return eval
    else:
        return -eval
```

Рисунок 1.1 – Код еврестичної функції

2. NegaMax

Алгоритм Negamax – це варіант мінімаксного пошуку, який спирається на властивість нульової суми гри для двох гравців.Цей алгоритм грунтується на тому, що max(a,b)=-min(-a,-b) для спрощення реалізації мінімаксного алгоритму.

```python
from evaluation import evaluation

def negamax(depth, board):
    bestScore = float("-inf")

    if depth == 0:
        return -evaluation(board)

    for move in board.legal_moves:
        board.push(move)
        score = -negamax(depth - 1, board)

        board.pop()
        if score > bestScore:
            bestScore = score

    return bestScore
```

Рисунок 2.1 – Код алгоритму

Усі подальші функції будуть повертати саме евристичну оцінку. А вибір кращого ходу відбувається далі, після виклику функції з алгоритмом. Кращі ходи записують в список, і список очищується, якщо знаходять ходи з кращою оцінкою. Якщо ходи мають однакову оцінку, то вони вибираються зі списку випадково.

```python
def bestMove(depth, board, searchfunc):
    legalMoves = board.legal_moves
    bestMove = None
    maxScore = -INF
    randomList = []

    for move in legalMoves:
        board.push(move)

        match searchfunc:
            case 'negamax':
                score = negamax(depth - 1, board)
            case 'negascout':
                score = negascout(depth - 1, board, -INF, INF)
            case 'pvs':
                score = pvs(depth - 1, board, -INF, INF)
            case _:
                print("Repeat!")
                exit()
        board.pop()
        if score >= maxScore:
            if score == maxScore:
                randomList.append(move)
            else:
                randomList.clear()
                randomList.append(move)
            maxScore = score
            bestMove = move
    if len(randomList) > 0:
        bestMove = random.choice(randomList)
    return bestMove
```

Рисунок 2.2 – Вибір накращого кроку

3. NegaScout

NegaScout є альфа-бета вдосконаленням алгоритму Negamax. Удосконалення прибирає і деякі помилки, що стосуються двох останніх шарів, які не потребують повторного пошуку. Дуже схожий з PVS, і тому про нього більше в наступному заголовку.

```python
from evaluation import evaluation

def negascout(depth, board, alpha, beta):
    bestScore = float("-inf") # a = alpha = -inf
    b = beta
    if depth == 0:
        return -evaluation(board)

    for move in board.legal_moves:
        board.push(move)
        score = -negascout(depth - 1, board, -b, -alpha)

        if score > bestScore:
            if alpha < score < beta:
                bestScore = max(score, bestScore)
            else:
                bestScore = -negascout(depth - 1, board, -beta, -score)

        board.pop()
        alpha = max(score, alpha)
        if alpha > beta:
            return alpha
        b = alpha + 1

    return bestScore
```

Рисунок 3.1 – Код алгоритму

4. PVS

PVS іноді ототожнюють з NegaScout, і як видно з коду, вони дійсно відрізняються всього лише одним рядком. Він є швидшим за alpha-beta pruning і домінує над ним тим, що він ніколи не досліджуватиме вузол, який можна відрізати альфа-бета-версією; однак він покладається на точне впорядкування вузлів, щоб скористатися цією перевагою.

```python
from evaluation import evaluation

def pvs(depth, board, alpha, beta):
    bestScore = float("-inf") # a = alpha = -inf
    b = beta
    if depth == 0:
        return -evaluation(board)

    for move in board.legal_moves:
        board.push(move)
        score = -pvs(depth - 1, board, -b, -alpha)

        if score > bestScore:
            if alpha < score < beta:
                bestScore = max(score, bestScore)
            else:
                bestScore = -pvs(depth - 1, board, -beta, -score)

        board.pop()

        if alpha > beta:
            return alpha
        b = alpha + 1

    return bestScore
```

Рисунок 4.1 – Код алгоритму

Висновок

В ході даної лабораторної роботи було створено програмне рішення дерев прийняття рішення для найоптимальніших рухів фігур на шаховій дошці.

Було реалізовано наступні алгоритми: NegaMax, NegaScout, PVS; також реалізовано евристичну функцію для цих алгоритмів.

Лабораторна робота виконана на основі бібліотеки python-chess.

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