2024

Rayyan Masood.
Enrollment no: 01-134212-151
Class: BSCS 6-C
Date: April 28th ,2024

[“ARITIFICIAL INTELLIGENCE LAB

[“ASSIGNMENT NO:7”]

Lab 7

Adversarial Search:

- Some computational problems can have many solutions. We say that these solutions are lying in the search space.
- The idea is that out of the many possible solutions, we use a searching algorithm that finds the optimal solution present in the search space.
- The solution found can be optimal locally or globally, but that depends on the type of search that has been implemented.

Local Search:

- Local search can be used on problems where a solution must be found that maximizes a criterion among a number of candidate solutions.
- A local search algorithm starts from a candidate solution and then iteratively moves to a neighbour solution.

Games vs. search problems:

- "Unpredictable" opponent → Solution strategy should be to specify a move for every possible opponent reply.
- Time limits → if time limits are in place it becomes unlikely to find goal as performing search for every possible move the opponent makes is expensive. In such a situation, the algorithm must approximate.

Optimal Decision in Games:

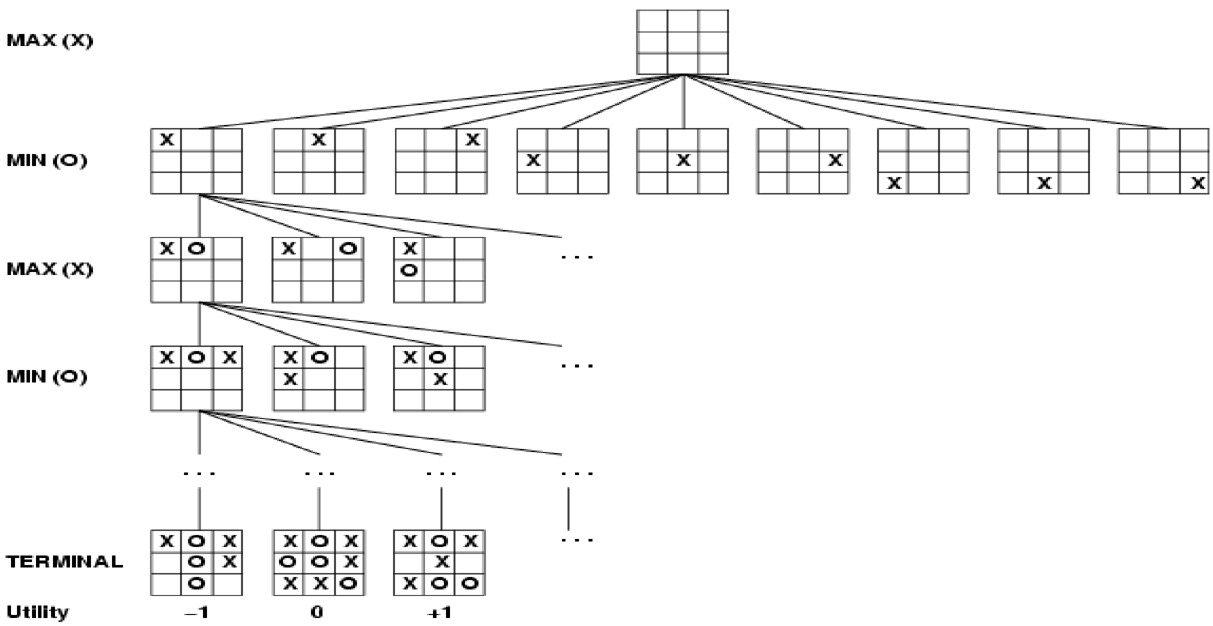
- The initial state: It identifies the player to move and the board position
- A successor function: It returns a list of (move, state) pairs, each indicating a legal move and the resulting state.
- A terminal test: determines when the game is over. States where the game ends are called terminal states.

- A utility function: Gives a numerical value to the terminal states. For example the outcome is a Win (+1), a loss (-1) or a draw (0).

Game Tree:

- It is a tree where you define moves for every possible move/reply of the opponent
- Your aim is to find the route that gives the terminal state as a **<u>win</u>**

Game tree for tic-tac-toe(2-player, deterministic, turns):



Minimax Algorithm:

Minimax and Alpha-Beta Types and Functions

Any implementation of minimax and alpha-beta must be supplied with the following types and routines.

Board type

This type contains all information specific to the current state of the game, including layout of the board and current player.

Score type
This data type indicates piece advantage, strategic advantage, and possible wins. In most games, strategic advantage includes the number of moves available to each player with the goal of minimizing the opponent's mobility.

neg_infinity and **pos_infinity**
The most extreme scores possible in the game, each most disadvantageous for one player in the game.

generate_moves
This function takes the current board and generates a list of possible moves for the current player.
apply_move
This function takes a board and a move, returning the board with all the updates required by the given move.
null_move
If the chosen game allows or requires a player to forfeit moves in the case where no moves are available, this function takes the current board and returns it, after switching the current player.
static_evaluation
This function takes the board as input and returns a score for the game.
compare_scores
This function takes 2 scores to compare and a player, returning the score that is more advantageous for the given player. If scores are stored as simple integers, this function can be the standard $<$ and $>$ operators.

Minimax Pseudocode

```
function ALPHA-BETA-DECISION(state) returns an action
   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

```
function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
   same as MAX-VALUE but with roles of α, β reversed
```

Pseudocode:

```
alpaBetaMinimax(node, alpha, beta)

   """
   Returns best score for the player associated with the given node.
   Also sets the variable bestMove to the move associated with the
   best score at the root node.
   """

   # check if at search bound
   if node is at depthLimit
      return staticEval(node)
```

```
# check if leaf
children = successors(node)
if len(children) == 0
   if node is root
      bestMove = []
   return staticEval(node)

# initialize bestMove
if node is root
   bestMove = operator of first child
   # check if there is only one option
   if len(children) == 1
      return None

if it is MAX's turn to move
   for child in children
      result = alphaBetaMinimax(child, alpha, beta)
      if result > alpha
         alpha = result
         if node is root
            bestMove = operator of child
      if alpha >= beta
         return alpha
   return alpha

if it is MIN's turn to move
   for child in children
      result = alphaBetaMinimax(child, alpha, beta)
      if result < beta
         beta = result
         if node is root
            bestMove = operator of child
      if beta <= alpha
         return beta
   return beta
```

Standard Implementation of Minimax:

The standard implementation of the Minimax algorithm frequently includes three functions: `minimax(game_state)`, `min_play(game_state)` and `max_play(game_state)`.
Note that Python is used here as working pseudo-code.

```
def minimax(game_state):
  moves = game_state.get_available_moves()
  best_move = moves[0]
  best_score = float('-inf')
  for move in moves:
    clone = game_state.next_state(move)
    score = min_play(clone)
    if score > best_score:
      best_move = move
      best_score = score
  return best_move
```

To summarize, Minimax is given a game state, obtains a set of valid moves from the game state, simulates all valid moves on clones of the game state, evaluates each game state which follows a valid move and finally returns the best move.

The following two helper functions simulate play between both the opposing player and the current player through the `min_play` and `max_play` procedures respectively. With the aid of these two helper functions, the entire game tree is traversed recursively given the current state of the game.

```
def min_play(game_state):
  if game_state.is_gameover():
    return evaluate(game_state)
  moves = game_state.get_available_moves()
  best_score = float('inf')
  for move in moves:
    clone = game_state.next_state(move)
    score = max_play(clone)
    if score < best_score:
      best_move = move
      best_score = score
  return best_score

def max_play(game_state):
  if game_state.is_gameover():
    return evaluate(game_state)
  moves = game_state.get_available_moves()
  best_score = float('-inf')
  for move in moves:
    clone = game_state.next_state(move)
    score = min_play(clone)
    if score > best_score:
      best_move = move
      best_score = score
  return best_score
```

In particular, the opponent intends to minimize the current player's score and the current player intends to maximize their own score. Note that the helper functions short-circuit and return early if the game is over.

Lab Journal 8:

a. Implement minmax with alpha beta pruning algorithm on following given leaf nodes of thetree.
values=[3,5,2,9,12,5,23,24]

Code:

```
MAX_VALUE, MIN_VALUE = 1000, -1000

def minimax(depth, node_index, is_maximizing, values, alpha, beta):
    if depth == 3:
        return values[node_index]

    if is_maximizing:
        best_value = MIN_VALUE

        for i in range(0, 2):
            value = minimax(depth + 1, node_index * 2 + i, False, values, alpha, beta)
            best_value = max(best_value, value)
            alpha = max(alpha, best_value)

            if beta <= alpha:
                break

        return best_value

    else:
        best_value = MAX_VALUE

        for i in range(0, 2):
            value = minimax(depth + 1, node_index * 2 + i, True, values, alpha, beta)
            best_value = min(best_value, value)
            beta = min(beta, best_value)

            if beta <= alpha:
                break

        return best_value


if __name__ == "__main__":
    values=[3,5,2,9,12,5,23,24]
    print("The optimal value is:", minimax(0, 0, True, values, MIN_VALUE, MAX_VALUE))
```

Output:

```
PS R:\Learning Curve\Python> & C:
pruningtask.py"
The optimal value is: 12
PS R:\Learning Curve\Python>
```

b. Your task is to implement a Tic-Tac-Toe game in Python where a human player can play against one AI opponent. The AI opponent will use the Minimax algorithm with Alpha-Beta Pruning for more efficient decision-making.

Code:

```python
import random

def print_board(board):
    for row in board:
        print(" ".join(row))
    print()

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or all(board[j][i] == player for j in
range(3)):
            return True

    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in
range(3)):
        return True

    return False

def is_board_full(board):
    return all(board[i][j] != ' ' for i in range(3) for j in range(3))

def evaluate(board):
    for player in ['X', 'O']:
        if check_winner(board, player):
            return 1 if player == 'X' else -1

    return 0

def minimax(board, depth, alpha, beta, maximizing_player):
    if check_winner(board, 'X'):
```

```python
        return -1
    if check_winner(board, 'O'):
        return 1
    if is_board_full(board):
        return 0

    if maximizing_player:
        max_eval = float('-inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'
                    eval = minimax(board, depth + 1, alpha, beta, False)
                    board[i][j] = ' '
                    max_eval = max(max_eval, eval)
                    alpha = max(alpha, eval)
                    if beta <= alpha:
                        break
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'
                    eval = minimax(board, depth + 1, alpha, beta, True)
                    board[i][j] = ' '
                    min_eval = min(min_eval, eval)
                    beta = min(beta, eval)
                    if beta <= alpha:
                        break
        return min_eval

def ai_move(board):
    best_val = float('-inf')
    best_move = (-1, -1)

    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'O'
                move_val = minimax(board, 0, float('-inf'), float('inf'), False)
                board[i][j] = ' '
                if move_val > best_val:
                    best_move = (i, j)
                    best_val = move_val

    board[best_move[0]][best_move[1]] = 'O'

def play_game():
```

```python
    board = [[' ' for _ in range(3)] for _ in range(3)]
    print("~~~~~~~~~Tic-Tac-Toe!~~~~~~~~~~")
    print_board(board)

    while True:
        while True:
            row = int(input("Enter row (0, 1, or 2): "))
            col = int(input("Enter column (0, 1, or 2): "))
            if 0 <= row <= 2 and 0 <= col <= 2 and board[row][col] == ' ':
                board[row][col] = 'X'
                break
            else:
                print("Invalid option selected, Try again")
        print_board(board)

        if check_winner(board, 'X'):
            print("You win ... Somehow(Gotta check my code again)")
            break

        if is_board_full(board):
            print("Draw!")
            break

        print("AI is preparing a move")
        ai_move(board)
        print_board(board)

        if check_winner(board, 'O'):
            print("Defeat ... AI wins!.")
            break

        if is_board_full(board):
            print("Draw!")
            break

if __name__ == "__main__":
    play_game()
```

Output:

```
  O O
    X

Enter row (0, 1, or 2): 0
Enter column (0, 1, or 2): 1
Invalid option selected, Try again
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 0
O X X
X O O
    X

AI is preparing a move
O X X
X O O
O   X

Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 1
O X X
X O O
O X X

Draw!
```