Mancala Game

Project Report

BY:

Name	Code
Nariman Karam Sabek	1601812
Nada Emam Kamal	1601558
Nermeen Osama Ramadan	1601575
Hend Muhammed Mahmoud	1601659

Faculty of Engineering

Ain shams university

Cairo 2021

1. Brief description the game and implementation

- Game Setup:
  Mancala is a board game where board is made up of two rows of six pockets each. Four pieces are placed in each of the 12 pockets. Each player has a "store" (also called a "Mancala") to his/her right side of the Mancala board.

- Object:
  The object of the game is to collect the most pieces by the end of the game.

- Game Play:
  1. The game begins with one player picking up all of the pieces in any one of the pockets on his/her side.
  2. Moving counter-clockwise, the player deposits one of the stones in each pocket until the stones run out.
  3. If you run into your own Mancala (store), deposit one piece in it. If you run into your opponent's Mancala, skip it and continue moving to the next pocket.
  4. If the last piece you drop is in your own Mancala, you take another turn.
  5. If you are playing at stealing mode and the last piece you drop is in an empty pocket on your side, you capture that piece and any pieces in the pocket directly opposite.
  6. The game ends when all six pockets on one side of the Mancala board are empty.
  7. The player who still has pieces on his/her side of the board when the game ends captures all of those pieces.
  8. Count all the pieces in each Mancala. The winner is the player with the most pieces.

- Game implementation:
  The game is console based implemented using functional procedure using several functions each one is responsible for a certain functionality of the game.
  - **Main function:**
    This function is responsible for game setup by taking user input which defines whether to play in:
    1. Easy mode.
    2. Medium mode.
    3. Hard mode.

    And whether to play at:
    1. Stealing mode.
    2. Non stealing mode.

    And finally if he wants to:
    1. Play first.
    2. Let computer play first.

Finally it calls the start game function to begin playing.
It is enclosed in an infinite loop until the user chooses to exit the game.

- **Start game function**
  Handles the game sequence by checking the current player in each iteration.
  If it's the player turn it takes the pocket number to move its stones and validates if it's a valid input then update the board state according to this action.
  If it's is the computer turn it first builds the game tree with depth matching the difficulty level calling the build tree function then it applies the minimax with alpha beta pruning function on the resulting tree and finally calls the get best state function to select the board state best chosen by the computer.
  At the end of each iteration it checks the board if any of the players has all his pockets empty then declare the winner.
  After that the user is asked whether to play again or exit the game.

- **Update state function**
  Updates the board state by taking the stones in the given pocket and play the game according to the rules whether it's player or the computer turn.

- **Print board function**
  Prints a visualization of the board state.

- **Find next possible states**
  Given a node with a board state this function add al the next possible states from this state as children to the parent node.it also calculate the value of the utility function for the node states if they are leaf nodes or an end game state.

- **Build game tree**
  This function builds the tree with depth 3 for easy mode depth 6 for medium mode and depth 9 for hard mode by calling find next possible state function recursively starting with the root node until reaching the required depth.

- **Apply min max with alpha beta bruning**
  It takes the tree from the built tree function and explore it using depth first search while applying minimax with alpha beta bruning algorithm to indicate the best next move.

- **Get best state**
  Search the first level of the tree for the node of maximum value and its state will be the next board state played by the computer.

- Along with a Class that represents the nodes of the game tree built by computer
  With attributes:

- State: the board state at this node.
- Value: utility function value for the node state.
- Beta value.
- Alpha value.
- Type: weather it's a maximizer or minimizer node.
- Children: list of nodes with all possible states from the parent's node state.
- Parent: parent node.

And add node method to append children nodes to a parent node.

2. Description of the utility function used

The utility function used to calculate the value of the board state is the number of stones in the computer mancala and the goal is to maximize this number.

3. Bonus :

Difficulty levels proportional to tree depth is supported.
3 levels easy mode.
6 levels medium mode.
9 levels hard mode.

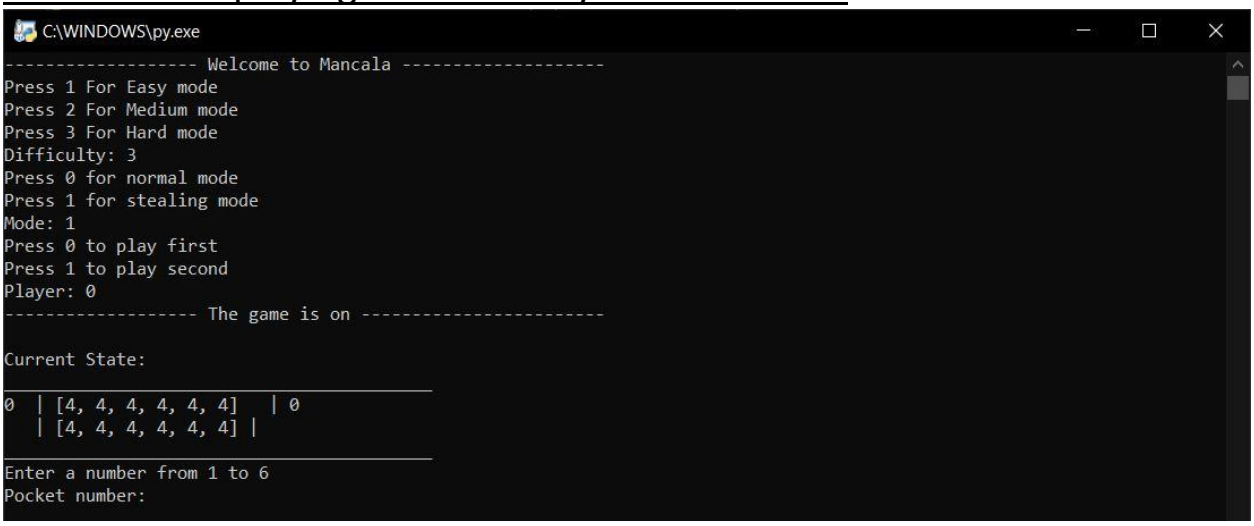
4. User guide(How to play):

First: choose difficulty

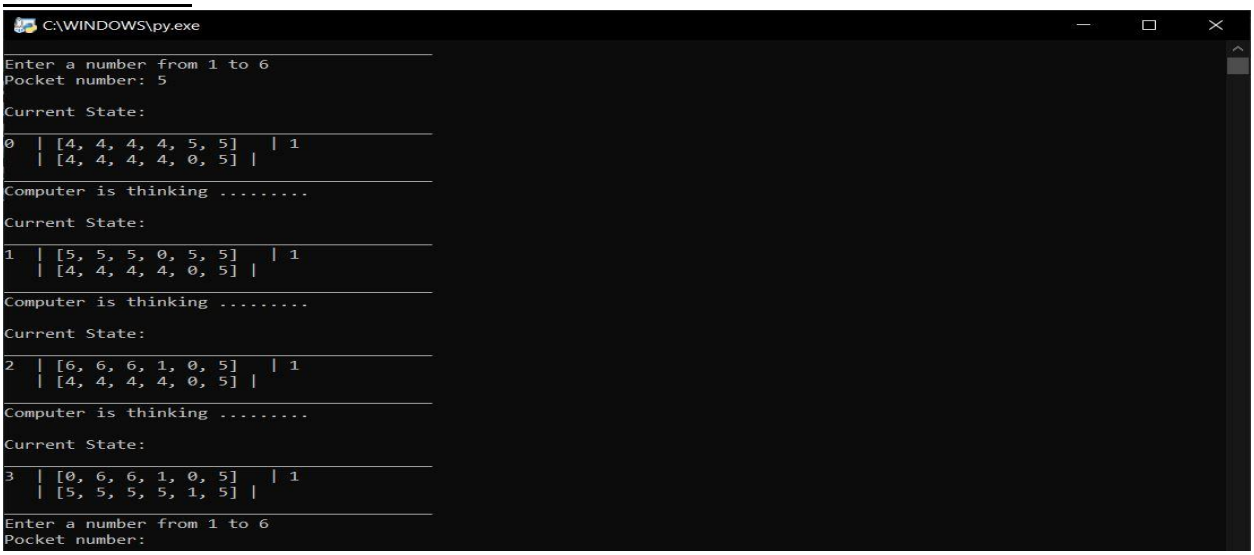
Second: choose mode

Third: choose who plays first

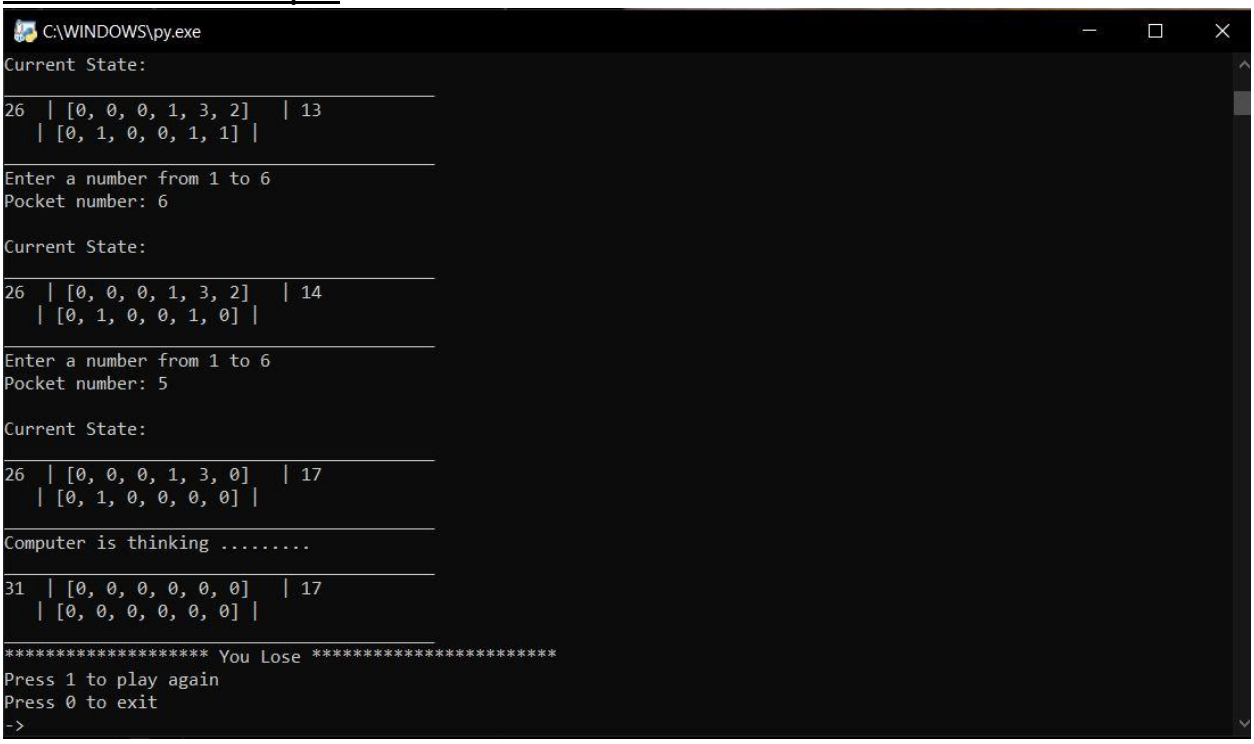
Fourth: Start playing and choose your next move



Fifth: a text (“computer is thinking”) appears every time the computer is making his next move and then his choice is printed afterwards



After several Steps:



5. Member's rules

- Nada: Build game tree, Find next possible states functions.
- Nariman: Update state, print board functions.
- Nermeen: start game, main functions.
- Hend: Apply min max with alpha beta bruning, get best state functions.

6. Code documentation

1. class Node:

Represents the game tree node

def __init__(self,state=None,value=None,beta=None,alpha=None,type=None):

Parameters: state: List

Board state associated with the node.

value: int

The value associated with the nod's state according to the utility function.

alpha: int

alpha value of node according to the minimax algorithm

beta: int

beta value according the minimax algorithm.

type: boolean
node type in the tree True for a maximizer, False for minimizer.

def addNode(self, node):

Append the given node to the calling node children.

Parameters: node: object
The child node

2. def state_update(state,cell_number,mode)

Gives the resulting state after moving stones in a given pocket.

Parameters: state: List
The current board state.
cell_number: int
The number of pocket must be from 1 to 12 only.
mode: int
Determine mode 1 for stealing and 0 for non-stealing.

Return new_state,double_turn: tuple of list and int
The list contains the next state or empty if the pocket is empty.
double_turn is 1 if the user is to be play again
-1 if the computer is to play again
0 otherwise

3. def print_state(state)

Prints the board's state.

Parameters: state: List

4. def find_next_possible_states(node,count_value,mode)

Add children nodes to the given node with the next possible states from the input node state.

Parameters: node: node object
The input node containing the state to add children to it.
count_value: boolean
True count the value of the added node states and false otherwise.
mode: int
Determine mode 1 for stealing and 0 for non-stealing.

5. def build_tree(root,num_levels,mode)

Build the game tree starting from the node root input by generating all possible state nodes played at each turn.

Parameters: root: object
The input node containing the current state of the game.
num_levels: int
Tree required depth.

mode: int
Determine mode 1 for stealing and 0 for non-stealing.

6. def apply_minimax_alpha_beta(root):

Traverse the game tree while applying minimax algorithm with alpha beta pruning to calculate node values.

Parameters: root: node object
The root of the tree to apply the algorithm on.

7. def get_best_node(root)

Get the node with maximum value from the root of the input tree direct children

Parameters: root: node object
The root of the tree.

Return: max_node: node object
The node that has the max value.

8. def start_game(state,mode,player,level)

Starts the game loop until one of the players win.

Parameters: state: List
The initial board state.
player: int
player code 0 for player.
1 for computer
mode: int
Determine mode 1 for stealing and 0 for non-stealing.
level: int
determines the difficulty 1 for easy
2 for medium
3 for hard

return: play_again: int

1 if the user wants to play again

0 to exit