Vedant Sinha
Sinehan Ezhilmuthu
Nihar Palkonda

Lab Report L Game

In the first page, the members of the group and the contribution of each member to the assignment; and the list of sources used (as noted in the course web page in general for all homeworks and assignments).

Contributions ->

Sinehan ->

-> Set up basic ascii grid
-> Set up framework for move making/legal move checking
-> Coded legal move generation for Ls
-> Coded legal move generation for neutral pieces
-> Coded minimax & alpha beta pruning algorithm
-> Coded cache, memoization, to improve time

Vedant ->

-> Set up how moves are dealt with
-> Set up the orientation of the L pieces
-> Parsed the string and moved the pieces
-> Made the default game state changeable

Nihar Palkonda->

-> Helped set up orientation to work with legal move(wouldn't align with our original nesting)
-> Similar issue with neutral pieces with the ai (wouldn't always move to the desired location)
-> Helped in choosing data structures to make sure there is optimization.
-> helped implement the cache(where is it useful)

Sources Used ->

-> https://hwwmath.looiwenli.com/l-game -> We used this interactive game to learn how to play.
-> AIMA 4th edition ch 3 and 4 -> specifically from these chapters we made sure to implement an admissible heuristic and decided on a difference based heuristic.
-> A difference based heuristic was the simplest because the goal of the program is to make it so your opponent cannot make a move. Any move you make that makes it so the opponent can make a smaller amount of moves is a good move.
-> OS library to manage game
->

A description of the design decisions you took, in particular the choice for data structures (e.g. for the representation of the state and legal actions, the search tree, etc.) and functions, with an explanation of what they do.

-> We decided to start by designing the game representation as a 2D matrix and print it through text in the terminal, at least for now. The 2D matrix allowed for simplicity. By using a matrix we were able to
easily index and access each cell.

-> Our representation made it so that a neutral piece is represented by an 'N'. Player 1 L piece represented by 'L1" and player 2 L piece represented by 'L2'. The way that we represented an empty space was marking it as a zero.

Vedant Sinha
Sinehan Ezhilmuthu
Nihar Palkonda

```
"""
  self.grid = [['0' for _ in range(4)] for _ in range(4)]

  #this is the loop that lets us create a 4 x 4 grid
"""
```

-> To check valid moves we indexed the matrix and did a simple nested for loop.

```
"""
for x, y in positions:
            if not (0 <= x < 4 and 0 <= y < 4):
                return False
            if self.grid[x][y] != '0':
                return False
        return True

    #code used to validate move
"""
```

-> When generating a data structure that holds all the possible moves a player can make, we were looking for something that would allow us to loop through/nest through so that every unique position is taken into account. The optimal data structure to do so is an array.
-> We later use this genLegal move for most of our other functions. Getting the legal moves serves to be very important for moving the pieces around.

```
"""
ef genLegalMoves(self, player):
        currPos = self.p1Pos if player == 'L1' else self.p2Pos
        for x, y in currPos:
            self.grid[x][y] = '0'
        legalMoves = []
        for i in range(4):
            for j in range(4):
                for pos in self.lPositions.values():
                    newPos = [(i + dx, j + dy) for dx, dy in pos]
                    if self.isValidMove(newPos):
                        legalMoves.append(newPos)
        currPosPermutations = list(permutations(currPos))
        legalMoves = [
            move for move in legalMoves
            if tuple(sorted(move)) not in map(tuple, map(sorted, currPosPermutations))
        ]
        for x, y in currPos:
```

Vedant Sinha
Sinehan Ezhilmuthu
Nihar Palkonda

```
            self.grid[x][y] = player
        return legalMoves

    We store an array of tuples
"""
```

-> To represent legal actions or moves a list of possible moves is maintained for each player during their turn.
-> After the array is created, it is then turned into a set so that in future use everything is unique as well as the search cost is cheaper.
-> The alpha beta pruning is implemented within our method by reinitializing alpha with each move. The highest alpha value is used to store the best move. Beta allows for pruning because only when alpha is greater than beta the move is stored.

```
"""
legal_moves = self.genLegalMoves(self.currentPlayer)
        valid_moves_set = set(valid_moves)
        valid_moves_mirrored_set = set(valid_moves_mirrored)
        legal_moves_sets = [set(move) for move in legal_moves]
"""
```

-> To implement a minimax we had to parse through our possible moves and use our heuristic to determine the best move.
-> Implementing a minimax with alpha beta pruning is essential because without some form of alpha beta pruning there wouldn't be an effective way to choose the optimal move.

```
"""
for move in legalMoves:
                self.simulateMove(player, move)
                score = self.minimax(opponent, depth - 1, alpha, beta, maximizing=(opponent=='L2'))
                self.restoreState(originalGrid, originalP1Pos, originalP2Pos)
                self.neutralPieces = originalNeutrals[:]
                value = max(value, score)
                alpha = max(alpha, value)
                if beta <= alpha:
                    break
            self.cache[key] = value
            return value
        else:
            value = math.inf
            for move in legalMoves:
                self.simulateMove(player, move)
                score = self.minimax(opponent, depth - 1, alpha, beta, maximizing=(opponent=='L2'))
                self.restoreState(originalGrid, originalP1Pos, originalP2Pos)
```

Vedant Sinha
Sinehan Ezhilmuthu
Nihar Palkonda

```
            self.neutralPieces = originalNeutrals[:]
            value = min(value, score)
            beta = min(beta, value)
            if beta <= alpha:
                break
        self.cache[key] = value
        return value
```

"""

-> Caching is important because it allows us to store things that are going to be used again, and if used again, optimal search cost.
-> The optimal way of caching would be to use a dictionary because there is the ability to make the search cost low.

"""
self.cache = {} initialize as dictionary

```
 if not legalMoves:
        v = self.heuristicEvaluation()
        self.cache[key] = v
        return v
```

storing important information such as heuristic scores let you use and reassess key information without having to repeatedly parse through shit.
"""

->

An explanation of your heuristic evaluation function, backed by your understanding of the game.
->
-> A difference based heuristic was the simplest because the goal of the program is to make it so your opponent cannot make a move. Any move you make that makes it so the opponent can make a smaller amount of moves is a good move.
-> The goal of the game is to make it so your opponent cannot make a move. To truly make a great AI, you want to do it in the lowest amount of moves.
-> This can be done by making your ai able to see more depth.
-> By making our heuristic this way we could incorporate depth. Relative advantage helps with caching. Like said earlier, it is simple, only keep track of Alpha/beta values as well as the actual move/possible moves.
-> With the caching prune unnecessary branches further optimizing the ai.
-> You can account for your opponent's goal, which in this case is to be able to make a move.
->

Vedant Sinha
Sinehan Ezhilmuthu
Nihar Palkonda

A discussion of issues that are relevant for the game: what is the (typical) branching factor? How deep can the game run? Can cycles occur and if so how to deal with them? How many states are terminal? etc.

-> Branching factor depends on the total available spaces on the board. In a 4x4 there are 16 spaces.

-> Of the 16 6 are open. You also have to take into account that you don't have to move a neutral piece. You must move an L piece.

-> Depending on the positioning of what's on the board, you have anywhere from 0- the maximum amount of moves. In terms of possible states, there are infinite. But for them to be unique it would be 2^possible moves....2^1.

-> It can have an infinite depth because like Chess, if played perfectly, it continues on forever.

-> Do our heuristic cycles not occur? The branches are pruned once the heuristic score is deemed as the same, if a cycle occurs it is the best possible move for the computer, any other move would be a losing move.

-> There are many many unique states. Still there is a goal state and a draw state. In a draw state, moves are still possible for the opponent.

-> In goal state, the opponent cannot make a move. To get total possible states you find as many unique positions in which there are both L pieces on the board as well as the neutral pieces.

-> With more branching factors you get a higher cost of search.

-> To optimize the program we tried to keep the branching factor at its lowest.