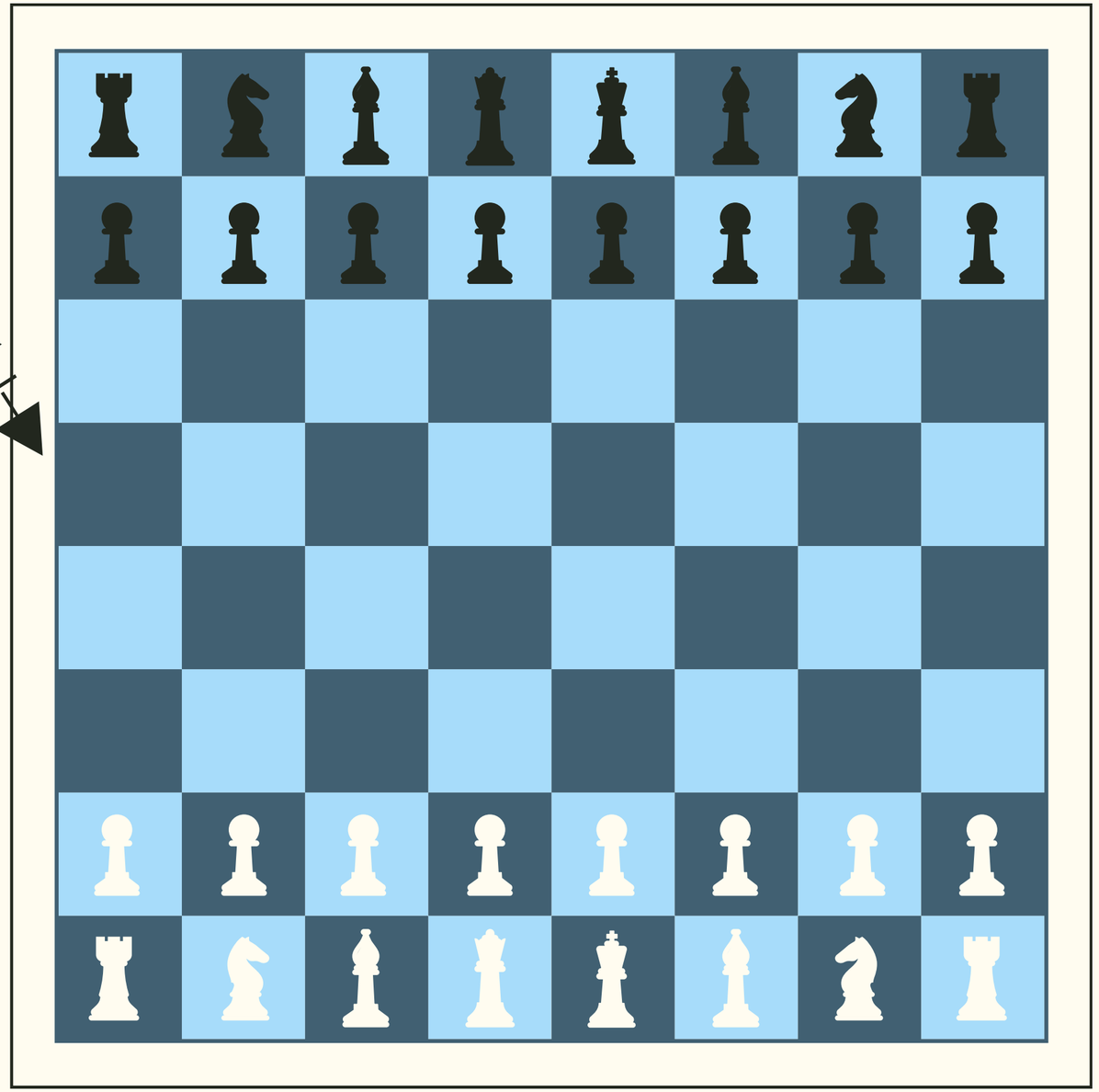
SURUTHI LAVANYA A
CB.EN.U4CSE21262

ABINEHA P
CB.EN.U4CSE21201

19CSE212-DSA

OPTIMAL MOVES IN CHESS

HYBRID DATA STRUCTURES



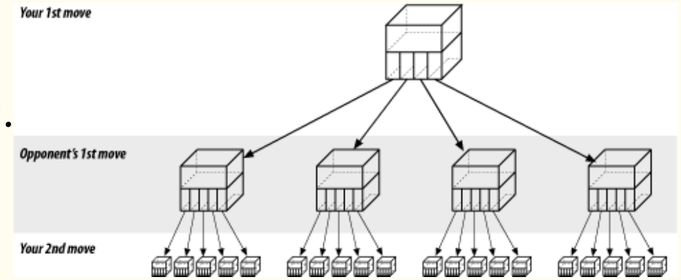
INTRODUCTION

HYBRID DATA STRUCTURES

Hybrid data structures refer to data structures that combine the characteristics and functionalities of two or more different data structures. These combinations are often designed to leverage the strengths of each individual data structure to optimize performance and efficiency for specific use cases.
Examples: Hashed array tree, B-Trees with hashing

Benefits of using hybrid data structures:

- They can be more efficient than a single data structure for solving complex problems.
- They can be more flexible and adaptable to different types of problems.
- They can be easier to implement and maintain than custom data structures.



OBJECTIVE:

The objective of integrating hybrid data structures and optimal moves in a chess application is to enhance gameplay, increase the accuracy of move suggestions, and provide efficient analysis and storage of game states. By combining the strengths of hybrid data structures and optimal move algorithms, the application aims to offer players a more enjoyable and competitive chess experience.

Benefits of Hybrid Data Structures and Optimal Moves in Chess Application:

i. Efficient Game State Storage
ii. Enhanced Move Generation
iii. Performance Optimization
iv. Learning and Analysis Tools
v. Strategic Guidance
vi. Personalized Recommendations

SEGMENTATION OF PROBLEM

1. **Simulating Chess openings:**

The Italian Game, Sicilian Defense, The French Defense, The Ruy-Lopez, The Slav Defense. Simulation involving turn wise moves to give player a demo to learn the best and efficient openings that could optimize their moves and increase win rate during gameplay.
On user's input of desired Chess opening, the first few moves of the opening are simulated sequentially involving turn based gameplay between white and black pieces.

2. **Chess game involving AI:**

Implementing the chess game involving player vs AI. AI moves are decided by implementing the NegaMax algorithm and adding alpha beta pruning. The random moves of each piece satisfying valid moves are made more accurate by utilizing weighted graphs.
Implementing Real-Time Chess between user vs AI. Board state stored as graph and optimal moves decided using specific algorithms for graphs.

3. Displaying graph for each piece in chess:

The valid moves for each piece are calculated and positions are described as vertices of a graph where the edge of the graph displays the move transition from one tile to another tile. The Knight's tour problem coded and transitions to be displayed as a graph.
Graphical representation of moves of each unique piece which shows behavior of each piece.

OVERVIEW OF HYBRID DATA STRUCTURE

The chosen data structure - Combination of Linked Hash Map, graphs, search trees and arrays.

1. **Simulating Chess openings:**

   Linked Hash Map - HashMap + Singly Linked List

   Design Choice:

   LinkedHashMap uses a hybrid data structure to maintain the order of entries in which they were inserted according to specific indexes. Easy way to maintain sequential order of moves to be displayed on the board one by one.
   HashMap indexes are the name of the Chess openings which the user enters. Using the index, the value is retrieved which is a singly linked list with ordered moves.

2. **Chess game involving AI**

   Graph + Weighted Graph + Search Trees

   Design Choice:

   Since, the 8x8 board is of the same representation as a 2-D matrix, placing each piece on the board would be easy as a graph. The position of each piece or tile could be calculated as considering a graph spread out across the coordinate axes. Weighted graph provides the AI with precise vertices on which the piece would have high score making it the best possible move. Game search trees are important in artificial intelligence because one way to pick the best move in a game is to search the game tree which has a record of all future gameboard states and utilizes numerous tree search algorithms, combined with minimax-like rules to prune the tree.

3. Displaying Graph for each piece in chess

Graph + Array

Design Choice:

Since, the 8x8 board is of the same representation as the 2D matrix, placing each piece on the board would be easy as a graph. The position of each piece and a tile could be calculated as considering a graph spread out across the coordinate axes.

```
self.map={"The Italian Game":0,"The Sicilian Defense":1,"The French Defense":2,
          "The Ruy-Lopez":3,"The Slav Defense":4}
```

```
    self.map["The Italian Game"]=n1
    self.map["The Sicilian Defense"] = n2
    self.map["The French Defense"] = n3
    self.map["The Ruy-Lopez"] = n4
    self.map["The Slav Defense"] = n5
```

```
def __init__(self):
    self.board= [["bR","bN","bB","bQ","bK","bB","bN","bR"],
                 ["bp","bp","bp","bp","bp","bp","bp","bp"],
                 ["--", "--", "--", "--", "--", "--", "--", "--"],
                 ["--", "--", "--", "--", "--", "--", "--", "--"],
                 ["--", "--", "--", "--", "--", "--", "--", "--"],
                 ["--", "--", "--", "--", "--", "--", "--", "--"],
                 ["wp", "wp", "wp", "wp", "wp", "wp", "wp", "wp"],
                 ["wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"]]
```

```
piece_score = {"K": 0, "Q": 9, "R": 5, "B": 3, "N": 3, "p": 1}
CHECKMATE=1000
STALEMATE=0
DEPTH=3
knight_scores = [[0.0, 0.1, 0.2, 0.2, 0.2, 0.2, 0.1, 0.0],
                 [0.1, 0.3, 0.5, 0.5, 0.5, 0.5, 0.3, 0.1],
                 [0.2, 0.5, 0.6, 0.65, 0.65, 0.6, 0.5, 0.2],
                 [0.2, 0.55, 0.65, 0.7, 0.7, 0.65, 0.55, 0.2],
                 [0.2, 0.5, 0.65, 0.7, 0.7, 0.65, 0.5, 0.2],
                 [0.2, 0.55, 0.6, 0.65, 0.65, 0.6, 0.55, 0.2],
                 [0.1, 0.3, 0.5, 0.55, 0.55, 0.5, 0.3, 0.1],
                 [0.0, 0.1, 0.2, 0.2, 0.2, 0.2, 0.1, 0.0]]
```

```
n1=SLinkedList()
n2 = SLinkedList()
n3 = SLinkedList()
n4 = SLinkedList()
n5 = SLinkedList()
n1.headval=Node("6444")
n1.AtEnd("1434")
n1.AtEnd("7655")
n1.AtEnd("0122")
n1.AtEnd("7542")
n2.headval = Node("6444")
n2.AtEnd("1232")
n3.headval=Node("6444")
n3.AtEnd("1424")
n3.AtEnd("6343")
n3.AtEnd("1333")
```

PRACTICAL APPLICATIONS

1. Linked Hash Map:

The main advantage of using LinkedHashMap is that it maintains and tracks the order of insertion where elements can be inserted and accessed in their order. It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. If one inserts the key again into the LinkedHashMap, the original order is retained.

- Shopping cart (Cart No: Vs Items chosen)
- Cache implementation (Preserves insertion order and allows efficient removal)
- LRU (Least Recently Used)
- Browser History (URL as entry in map)
- Job Scheduling (Maintains order and allows quick access)

2. Graphs:

- Social Networking (model and analyze connections between users)
- Recommendation systems (items as nodes and captures relationships as edges)
- Supply chain systems (locations as nodes, transportation as routes)
- Computer networks (devices as nodes and connections as edges)
- Biological systems (protein-protein interaction networks, gene regulatory networks, or metabolic pathways)

3. Search Trees

- Databases (B-trees, AVL trees, are used in database management systems)
- File systems (efficient searching and navigation through the file system)
- Autocomplete and spell checks (store a dictionary of words, allowing for efficient prefix-based searches to suggest completions or identify potential spelling errors)
- NLP (store n-grams, linguistic patterns, or language models)
- Game AI and Decision Trees (decision trees for making strategic choices)

4. Arrays

- Data storage and retrieval
- Mathematical and statistical computations
- Image and signal processing
- Matrices and linear algebra
- Performance optimization

5. Optimization of chess moves

- Tactical Awareness
- Calculation and Visualization
- Endgame Knowledge
- Continuous learning
- Time Management

PERFORMANCE ANALYSIS

1. **Simulating chess opening**

HashMap = O(1) : In selecting the index for the corresponding chess opening.

SLL = O(n) : n is the number of elements in SLL. Implemented case n<=6, since n is small could be considered constant O(1). To simulate entire chess game starting with a specific opening would be O(n)

2. **Chess game involving AI**

Board = O(1) : In selecting moving a chess piece form one tile to another tile.

Scores = O(n^2) : n is the number of elements in SLL. Implemented case n<=6, since n is small could be considered constant O(1). To simulate entire chess game starting with a specific opening would be O(n). The time complexity of the Minimax algorithm is O(bd), where b is the branching factor (number of branches for each node) and d is the depth of the tree.

3. **Displaying graph for each piece in chess**

O(n*n): to traverse throughout 8x8 board and generate all possible moves to plot graph using matplotlib.

O(1): to access index of HashMap

SIMULATING CHESS OPENINGS

The Italian Game Opening

HashMap = O (1) (time complexity): In selecting the index for the corresponding chess opening.
O(n) (space complexity): for n positions in table
SLL = O (n) (time complexity): n is the number of elements in SLL. Implemented case n<=6, since n is small could be considered constant O (1). To simulate entire chess game starting with a specific opening would be O(n).
O(n) (space complexity): for n entries in list

CHESS GAME INVOLVING AI

PLAYER VS AI (WHITE VS BLACK)

O(n*n) - (time complexity): for each possible position
O(n*n) - (time complexity): for knight's tour
O(n^2) (space complexity): 2d array each array of O(n)

Implementing the chess game involving player vs AI. AI moves are decided by implementing the NegaMax algorithm and adding alpha beta pruning. The random moves of each piece satisfying valid moves are made more accurate by utilizing weighted graphs.

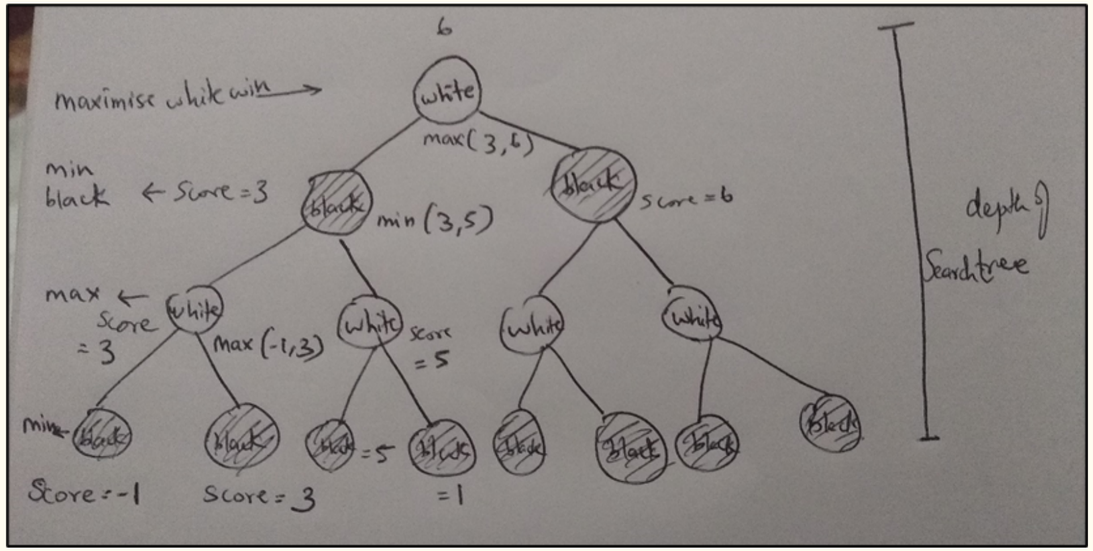
NEGAMAX WITH ALPHA-BETA PRUNING

Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally. Mini-Max algorithm uses recursion to search through the game-tree. Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree. The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

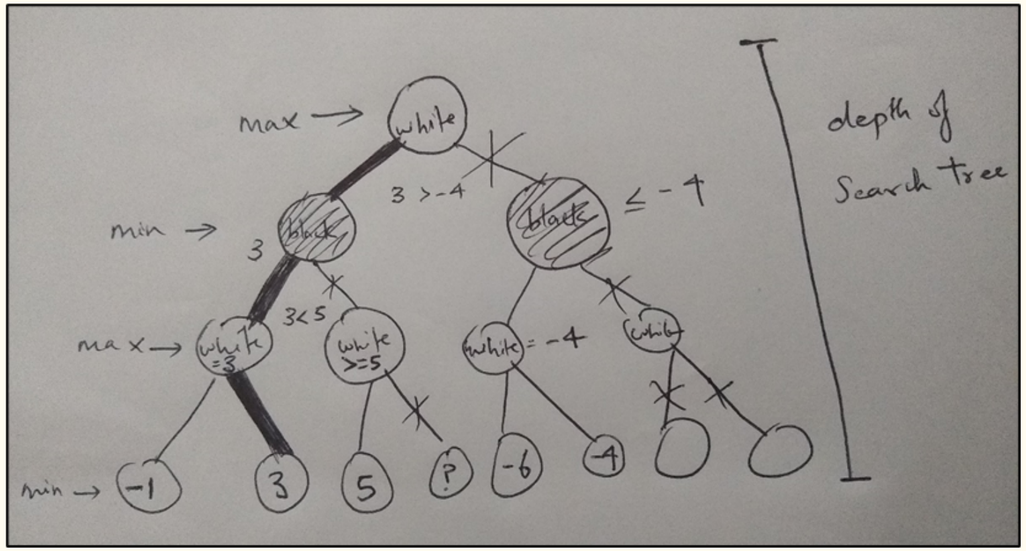
Negamax is a more efficient version of Minimax for games where the scoring is zero-sum. Zero-sum is a situation, in game theory, in which one person's gain is equivalent to another's loss, so the net change in benefit is zero. Alpha-beta pruning can decrease the number of nodes the negamax algorithm evaluates in a search tree.

The need for pruning came from the fact that in some cases decision trees become very complex. In that tree, some useless branches increase the complexity of the model. So, to avoid this, Alpha-Beta pruning comes to play so that the computer does not have to look at the entire tree.

These search algorithms are essential in turn-based games. It allows the program to look ahead at possible future positions before deciding what move it wants to make in the current position.



MINI-MAX SEARCH TREE



MINI-MAX SEARCH TREE WITH ALPHA-BETA PRUNING

PSEUDOCODE

```
Function minimax(position,depth,alpha,beta,maximizingplayer)
 If depth == 0 or game over in position
  return static evaluation of position
 If maximisingplayer
  maxEval = -infinity
  for each child of position
   eval=minimax (child, depth-1, alpha, beta, false)
   maxEval=max(maxEval,eval)
   alpha=max(alpha,eval)
   if beta<=alpha
    break
  return maxEval
else:
 miniEval = +infinity
 for each child of position
  eval=minimax (child, depth-1, apha, beta. true)
  minEval=min(minEval,eval)
  beta=min(beta,eval)
  if beta<=alpha
   break
 return miniEval
//initial call
Minimax (currentPosition,3, -infinity, +infinity, true)
```

DISPLAYING THE GRAPH FOR EACH PIECE IN CHESS

O(n*n) - (time complexity): for each possible position
O(n*n) - (time complexity): for knight's tour
O(n^2) (space complexity): 2d array each array of O(n)

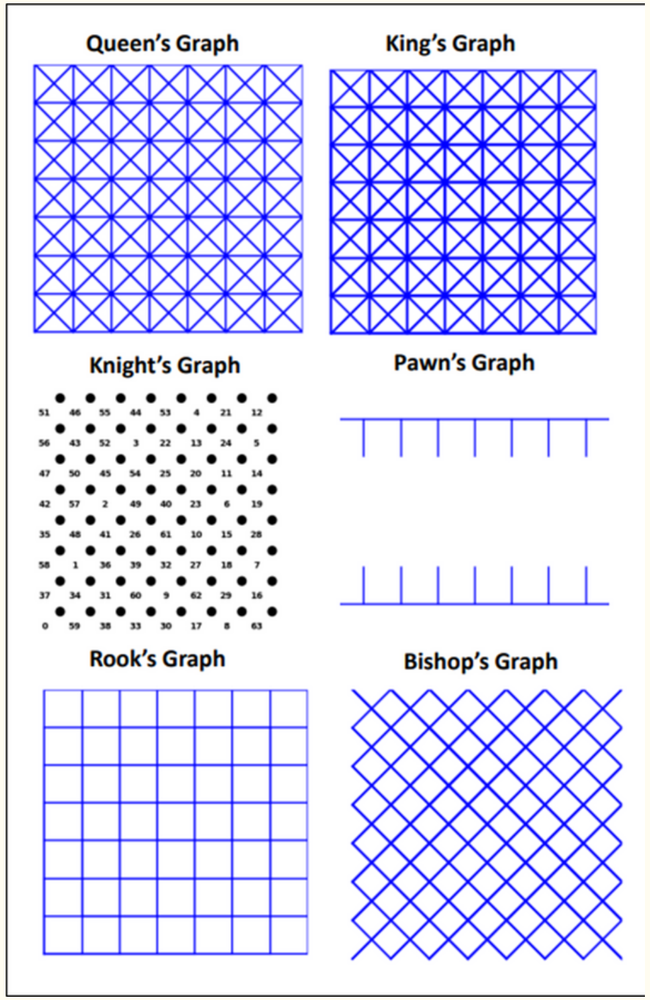
```python
def plot_solution(self):
    fig, ax = plt.subplots()
    ax.set_aspect('equal')
    ax.axis('off')

    for row in range(self.board_size):
        for col in range(self.board_size):
            ax.text(col, row, str(self.board[row][col]), va='center', ha='center', fontsize=10, fontweight='bold')
            ax.plot(col + 0.5, row + 0.5, 'ko', markersize=10)

    for row in range(self.board_size):
        for col in range(self.board_size):
            for move in self.moves:
                next_x = row + move[0]
                next_y = col + move[1]
                if self.is_valid_move(next_x, next_y):
                    ax.plot([col + 0.5, next_y + 0.5], [row + 0.5, next_x + 0.5], 'b-')

    plt.show()
```

Matplotlib generated graphs



CONCLUSION

We have implemented chess, optimized possible chess moves and evaluated the score for each status of the gameboard. The utilized hybrid data structures helped in having a clear representation and depiction of complex game states. The hybrid data structures efficiently stored data making it easy to track past moves and store future valid moves.
Trade-offs had to be done in the field of utilizing space. The space complexity should be further reduced as our code have multiple graphs for each chess piece making it not the best in terms of space and memory utilization.
The logic used in our code could be further applied in real-time situation requiring the need of complex operations and optimization requirements.

REFERENCES

1.https://www.researchgate.net/figure/Enhanced-NegaMax-with-Alpha-Beta-Property-Pseudo-Code_fig4_262672371
2.https://observablehq.com/@peatroot/chess-graphs
3.https://runestone.academy/ns/books/published/pythonds/Graphs/BuildingtheKnightsTourGraph.html
.