A)

An evaluation function is an estimate of the expected utility of the game from a given position. One of the ways is to give a score for the number of possible 4 in a line a player can still make with his coins. A higher weight is given to lines with a 3 in a row than lines with a 2 in a row.

Evaluation Function 1

```python
def evaluate_line(line):

    player = 2
    opponent = 1
    emptyspace = 0
    score = 0
    #based on the number of pieces on the line
    if line.count(player) == 4:
        score+= 1000
    elif line.count(player) == 3 and line.count(emptyspace) == 1:
        score+= 50
    elif line.count(player) == 2 and line.count(emptyspace) == 2:
        score+= 5
    elif line.count(emptyspace) == 3 and line.count(player) == 1:
        score+= 2
    elif line.count(emptyspace) == 4:
        score+= 1
    return score
```

Here, a score of 1000 is given if there is a 4 in a line and 50 for 3 in a line, and it reduces so on.

```
Game Over
Number of Games      : 100
Game Tree won        : 85
Myopic won           : 14
Avg moves to win     : 19.31764705882353
```

This results from such an evaluation function in 100 games with a cutoff depth of 3.

A variation to this evaluation function is to add a negative score if the opponent could have a connect 4 in a line.

Evaluation Function 2

```python
def evaluate_line(line):

    player = 2
    opponent = 1
    emptyspace = 0
    score = 0
    #based on the number of pieces on the line
    if line.count(player) == 4:
        score+= 1000
    elif line.count(player) == 3 and line.count(emptyspace) == 1:
        score+= 50
    elif line.count(player) == 2 and line.count(emptyspace) == 2:
        score+= 5
    elif line.count(emptyspace) == 3 and line.count(player) == 1:
        score+= 2
    elif line.count(emptyspace) == 4:
        score+= 1


    if line.count(opponent) == 4:
        score -= 2000
    elif line.count(opponent) == 3 and line.count(emptyspace) == 1:
        score -= 100
    elif line.count(opponent) == 2 and line.count(emptyspace) == 2:
        score -= 5
    elif line.count(emptyspace) == 3 and line.count(opponent) == 1:
        score -= 2
```

Since this evaluation function considers opponents possibly connect 4 lines as well, it has a lower number of losses as it is more prophylactic and cautious. Hence, there are more draws, fewer losses, and a better function overall.

```
Game Over
Number of Games      : 100
Game Tree won        : 92
Myopic won           : 3
Avg moves to win     : 22.58695652173913
```

Another evaluation function is creating a score matrix for each square on the Connect 4 matrix. As the squares in the middle can connect with more connect4 lines, we give it a higher score. As we go away from the center, we reduce the scores.

Evaluation Function 3

```
def evaluate_board2(currentState):
    matrix = [[1,2,3,4,3,2,1],
              [2,3,4,5,4,3,2],
              [3,4,5,6,5,4,3],
              [3,4,5,6,5,4,3],
              [2,3,4,5,4,3,2],
              [1,2,3,4,3,2,1]]
    sum = 0
    for i in range(ROWS):
        for j in range(COLOUMNS):
            if currentState[i][j]==1:
                sum -=matrix[i][j]
            elif currentState[i][j]==2:
                sum +=matrix[i][j]
    return sum
```

This evaluation function does slightly worse compared to the other boards because it might prefer keeping coins in the center of the board rather than connecting 4 coins in a straight line.

However, just trying to keep coins in the center is a decent enough strategy.

```
Game Over
Number of Games      : 100
Game Tree won        : 88
Myopic won           : 9
Avg moves to win     : 24.65909090909091
```

The number of losses and the average number of moves to win have worsened.

B

Alpha beta pruning dosent affect the results of the game , it only reduces and speeds up the time to calculate the best move.

```
beta = min(beta,curr_score)
if alpha>=beta:
    break
```

```
alpha = max(alpha,curr_score)
if alpha>=beta:
    break
```

These two lines help cutoff unnecessary branches of the game tree.

Results of the three evaluation functions with cutoff depth 3

```
Game Over
Number of Games      : 100
Game Tree won        : 85
Myopic won           : 14
Avg moves to win     : 19.31764705882353
```

Function 1

```
Game Over
Number of Games      : 100
Game Tree won        : 92
Myopic won           : 3
Avg moves to win     : 22.58695652173913
```

Function 2

```
Game Over
Number of Games      : 100
Game Tree won        : 88
Myopic won           : 9
Avg moves to win     : 24.659090909090091
```

Function 3

Results of the three evaluation functions with cutoff depth 5

```
Game Over
Number of Games      : 100
Game Tree won        : 96
Myopic won           : 4
Avg moves to win     : 21.6875
```

Function 1

```
Game Over
Number of Games      : 100
Game Tree won        : 97
Myopic won           : 1
Avg moves to win     : 24.742268041237114
```

Function 2

```
Number of Games      : 100
Game Tree won        : 96
Myopic won           : 1
Avg moves to win     : 20.333333333333332
```

Function 3

C)

I used a matrix with values for each square on the board to implement the move order heuristic. As the squares in the center are more likely to make a connect 4, I gave it a higher value. As you go further away from the center, the value decreases.

```
matrix = [[1,2,3,4,3,2,1],
          [2,3,4,5,4,3,2],
          [3,4,5,6,5,4,3],
          [3,4,5,6,5,4,3],
          [2,3,4,5,4,3,2],
          [1,2,3,4,3,2,1]]
```

I then sorted the possible coloumns with respect to these values.

```
move_order_heauristic_pos_coloumns = []
for i in range(len(pos_coloumns)):
    r = GameTreePlayer.row_of_col(currentState,pos_coloumns[i])
    move_order_heauristic_pos_coloumns.append((pos_coloumns[i],matrix[r][pos_coloumns[i]]))

move_order_heauristic_pos_coloumns = sorted(move_order_heauristic_pos_coloumns, key=lambda x: x[1], reverse=True)
pos_coloumns = [item[0] for item in move_order_heauristic_pos_coloumns]
```

This prioritizes the more promising moves (based on the matrix), thereby increasing the likelihood of early pruning and avoiding unnecessary exploration of unpromising branches.

The results do show this effect. The results do show this effect.

```
Number of Games        : 10
Game Tree won          : 10
Myopic won             : 0
Avg moves to win       : 20.8
Avg number of times Minimax function was called: 23837.8
PS C:\Users\Admin\Downloads\ai assignment2>
```

Avg number of times recursive function was called is 24000 without move-order heuristic

```
Number of Games        : 25
Game Tree won          : 23
Myopic won             : 1
Avg moves to win       : 18.52173913043478
Avg number of times Minimax function was called: 11038.54
```

It reduces down to 11000 with move-order heuristic.

Thus , this is a more efficient algorithm and reduced the number of time the recursive function is called and the average duration of the game , without affecting the correctness.

D)

Increasing cut-off depth to 5 does increase the frequency of winning .

```
Number of Games       : 200
Game Tree won         : 183
Myopic won            : 11
Avg moves to win      : 21.149484536082475
Avg number of times Minimax function was called: 1135.96
```

Results with cutoff depth 3

```
Number of Games       : 200
Game Tree won         : 198
Myopic won            : 1
Avg moves to win      : 21.13065326633166
Avg number of times Minimax function was called: 11750.41
```

Results with cutoff depth 5

The number of wins significantly increases; however, the average number of moves does stay the same. This may be because connect 4 in a relatively more simpler game, while average moves to win a game converge to a certain number, especially with a large number of games played.