**ASSIGNMENT 2**

**SUBJECT CODE: CS 6300**

**SUBJECT: ARTIFICIAL INTELLIGENCE**

**LEENA KORA**
**EMAIL**:leenak@cs.utah.edu

**Unid:** u0527667

**TEEKO GAME IMPLEMENTATION**

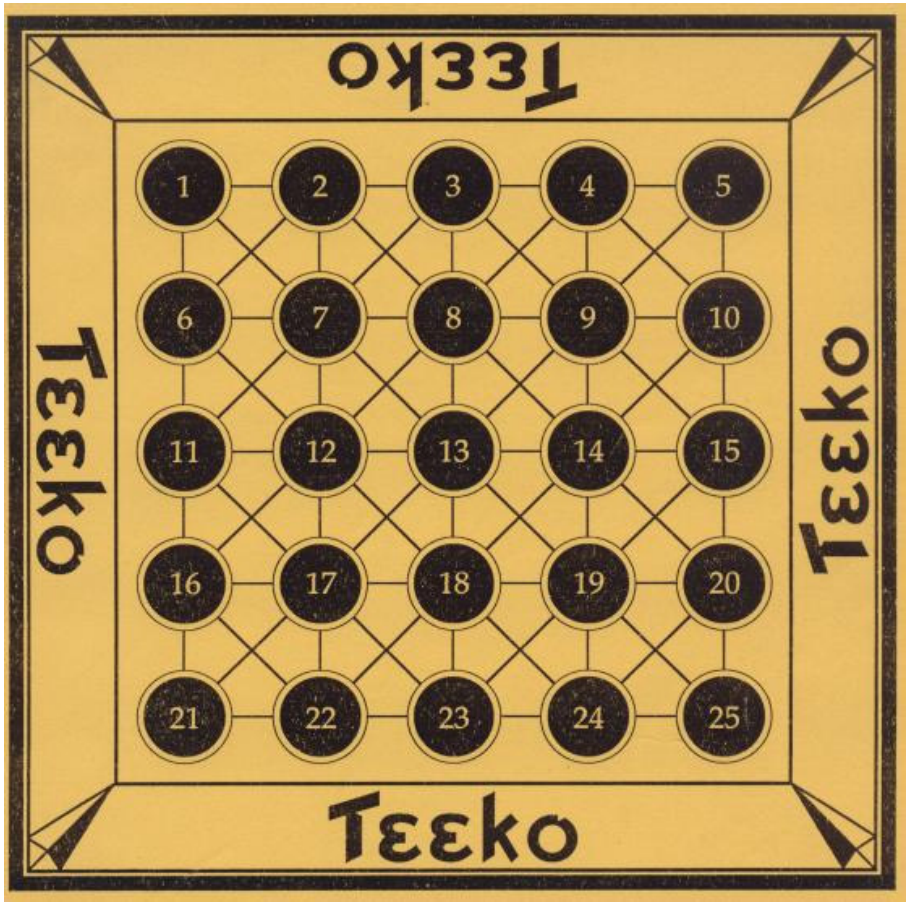
# Documentation and Discussion

1 . A short problem description

Explanation: The problem in hand is regarding implemenation of the teeko game. Teeko is an interesting game invented by John Scarne in 1945 and rereleased in refined form in 1952 and again in the 1960s. The Teeko game includes five by five square board.. It has eight discs in total to play with the Teeko board. Four are black and four are red. One of the two players choose to go for "Black" plays with the black discs, and the other for "Red" and plays with the red discs.

The game starts with an empty board. Black player moves first, and places his disc on any square/space on the board. Red player then places his/her disc on any unoccupied space., black does the same; and so on until all of the eight discs are being placed on the board. Black plays first and then the play alternates between the two players. (Adjacency is horizontal, vertical, or diagonal, but

does not wrap around the edges of the board.) . A disc can only be moved to an adjacent place. The goal of the game is for either player to win by having his/her discs situated in a straight line (vertical, horizontal, or diagonal) or square of four adjacent spaces. In each move player tries to maximize his scores. This game is win-lose game where the player can lose or win.




This Teeko game problem is very interesting because it involves simulation of how the computer might do the legal moves with given constraints. Figuring out how the task can done was very tricky and interesting. This problem gave an experience of how to write

Aritificial Intelligent programs and also how to make AI systems act in a rational way.

2. A description of the solution.

Explanation: I started contructing the data structures for the game state representation. The game can start with the empty board or it can read the given game state configuration from the file. I started out reading the input file and storing the start configuration in a Node class. The Node class maintained all the required information such as the character array to store the teeko board status, who is going to play next by the variable 'black_to_play', old disc position , new disc position. Then I defined Game class which maintains the node information and gets the user input. It also has function to check for the goal state, to evaluate the next possible move for the computer. It keeps track of the count of black discs and red discs and which player will be the computer.

User specifies his moves and the computer uses the searches to get the best possible move to its next move. Here I used minimax and alpha-beta search algorithm to figure out the best move. Here I used informed searches so that the serach would be intelligent and efficient. The minmax algorithm is guaranteed to find the solution which works fine for the given problem. It considers the computer as the maximizing player and the user as the minimizing player. It helps to estimate the best possible move for the computer using the heuristic function. The alpha beta cutoff helps to prun the search by reducing the number of moves to be considered for the search. I defined MinMax_game.h , ab_MInMax_Game.h files to implement them. Then I used plausible move ordering and forward pruning to reduce the amount of the search.

3. A discussion of the problems I encountered in solving the problem and how I overcame them, the weaknesses and strengths of your solution, and any other interesting observations.

Explanation: First problem I encountered was to figure out how to design the goal check function. I need to consider all the conditions and code for them. Once getting goal check function working then it was easy to code for evaluation function.

Then the hardest part was creating successors. During the creation of the successors, I need to consider all the attributes such as where it should be a minimizer or a maximizer, where it is black or red player and so on. I used lot of pointers so i had many segmentation errors. Debugging for that was a hard one. My basic minimax algorithm is implemented very well. The test cases made me to work more on my implementation and made it to be well designed. Getting the Evaluation function right was the first important thing for the problem to be solved. So I concentrated more on that. Checking the goal state had some faws earlier but my test casses made me to design it better.

**Testing information**

Explanation: I tested my program with various test cases. following are the test cases for my program.

- First I tested basic minimax implementation by reading the file and running it". My test case is as follows

  black
  4: 6 7 8 18
  4: 11 12 13 22

1. I was the red player. I moved 22=>23
2. The computer moved 18=>14
3. Then I moved 23=> 24
4. It moved 14=>9 and won the game.
5. The purpose of the test was to see whether it detects the possible goal and tries to get it or not.

- My second test case is as follows

black
4: 6 7 12 16
4: 17 19 20 22

1. I was the red player and computer was black.
2. The computer moved 12=>18
3. By placing the black from 12 to 18 it tried to block the opponent from wining .
4. The purpose of the test was to see whether it detects the possible goal for the opponent and tries to avoid it.

- My third test case for the alpha-beta minmax is as follows

black
0:
0:

1. The above the board state indicates the game began with the empty board.
2. I placed the black disc on 25 and computer placed it on 1.
3. Then I placed it on 24 and my AI on 2.

4. I then placed my disc on 20 to make corner L-shaped configuration. It then placed on 19 to block the goal for me.
5. The purpose of the test was to see whether it detects the possible goal configuration for the opponent and tries to avoid it.

- My four test case for the minmax is as follows

  black
  0:
  0:

1. The above the board state indicates the game began with the empty board.
2. I placed the black disc on 5 and computer placed it on 1.
3. Then I placed it on 9 and my AI on 2.
4. I then placed my disc on 17 to make corner diagonal configuration. It then placed on 13 to block the goal for me.
5. The purpose of the test was to see whether it detects the possible goal configuration for the opponent and tries to avoid it.

- Then adding many cout statments to my program to check where the user provides the right , valid inputs. I asked to enter again if the move is not valid. I used while loop to get the valid input from the user.

- Then I tested by giving invalid input moves. When I enter the invalid position for the second disc it gave the output as

"Invalid entry for the second disk position"
"Enter new disc position again"

- I also added error messages for flagging when my program reads invalid input from the file.

**Answers to additional discussion questions**

1. Were the statistics for search efficiency (-stats option) as you expected when you changed the depth limit or turned on α-β pruning or plausible move ordering?

Explanation: Yes, the statistics for search efficiency were as expected when I changed the depth limit or turned on pruning. When I used to changed the depth value, I used to get more deatailed information about the search and get to know how the search actually happens. "-stats" and "trace" options were very usefull for debugging my bugs in the program. It helped to know what actually going wrong in the algorithm. When alpha-beta or plausible move ordering was enabled it clearly showed how the search was reduced in terms of node generation and cut-offs. It also showed how many times the search was done to get to the required move. It showed how many times the search backed up after reaching required depth limit or a goal state.

2. Did the game playing performance vary as you would expect when you changed the depth limit or turned on α-β pruning or plausible move ordering? Consider both the statistics printed by the -stats option and the actual execution time.

Explanation: Yes, the performance varied as I changed the depth limit or turned on alpha-beta pruning or plausible move ordering. When the depth was increased it gave better results by finding the possible goal configurations on the next level nodes but at the same time the execution time increased as the number of nodes generated and search is more. The stats option displayed the more large numbers when the depth was increased in case of nodes generated and search called.

When I turned on pruning or plausible move ordering, I observed that execution time was reduced. The search was fast because as less number of nodes were searched due pruning and plausible move ordering. When plausible move ordering is enabled it orders the nodes and helps pruning the nodes efficiently.

3. Is the complexity of the search largest early or late in the game?

Explanation: I think the complexity of the search is more complex early in the game. Because it has less information and far away from the goal state. It does not have concrete information at the start of the game. As it reaches towards the end it will be having more usefull information to get to the goal. So It has to search all possible moves and estimate the best possible move intensively at the beginning of the search.

4. How much overhead is associated with forward pruning and plausible move ordering, in terms of the increase in execution time per expanded game state? How does this compare to the execution savings?

Explanation: Forward pruning and plausible move ordering are computationally costly. They need expanding the gamestate node and generating the successors in order to sort them and prun them. But if the branching facter is too large then the forward pruning helps avoiding searches in the some of the branches by pruning it at the begin thus saving time and memory totally.
Forward pruning is best if the branching factor and detpth are large becuase it avoids searches in the

some long branches saving lot of memory and time.

5. Discuss the behavior of the program when `-auto` is enabled. Do the other flags change the outcome in ways that you would expect? (CS 6300 only)

Explanation: When -auto is enabled the user moves are chosen randomly from the set of valid moves. I generated the list of successors and chose randomly for the move . The computer just choses the moves which will be favorable for it to get to the goal state. As the moves of the opponent are random, it tries to win by choosing best possible moves for itself to win the game. In case of auto the other flags does not affect much because the opponent will not be playing optimally as assumed by the algorithm so ordering and pruning doen not affect much.

6. Suggest an alternate evaluations functions and demonstrate empirically how it compares to the default heuristic function given above. (CS 6300 only)

Explanation: My evaluation function use the duplex sum for each player. It can be calculated by adding one to the sum when there are two positions in the winning configuration which are occupied by the player's piece and the remaining position s are unoccupied.