HEURISTIC ANALYSIS

HEURISTIC #1			
EVALUATING ID_IMPROVED			
Agent #1	**Agent #2**	**Agent #1 score**	**Agent #1 score**
ID_Improved	Random	17	3
ID_Improved	MM_Null	12	8
ID_Improved	MM_open	14	6
ID_Improved	MM_Improved	13	7
ID_Improved	AB_null	13	7
ID_Improved	AB_open	12	8
ID_Improved	AB_Improved		
	ID_Improved	67.86%	

HEURISTIC #2 EVALUATING ID_IMPROVED			
Agent #1	**Agent #2**	**Agent #1 score**	**Agent #1 score**
ID_Improved	Random	17	3
ID_Improved	MM_Null	14	6
ID_Improved	MM_open	12	8
ID_Improved	MM_Improved	15	5
ID_Improved	AB_null	16	4
ID_Improved	AB_open	9	11
ID_Improved	AB_Improved	10	10
	ID_Improved	71.43%	

HEURISTIC #3			
EVALUATING ID_IMPROVED			
Agent #1	**Agent #2**	**Agent #1 score**	**Agent #1 score**
ID_Improved	Random	18	2
ID_Improved	MM_Null	14	6
ID_Improved	MM_open	13	7
ID_Improved	MM_Improved	13	7
ID_Improved	AB_null	15	5
ID_Improved	AB_open	15	5
ID_Improved	AB_Improved	6	14
	ID_Improved	67.14%	

HEURISTIC #1			
EVALUATING STUDENT			
Agent #1	**Agent #2**	**Agent #1 score**	**Agent #1 score**
Student	Random	15	5
Student	MM_Null	13	7
Student	MM_open	13	7
Student	MM_Improved	14	6
Student	AB_null	10	10
Student	AB_open	11	9
Student	AB_Improved	11	9
Student 62.14%			

HEURISTIC #2			
EVALUATING STUDENT			
Agent #1	**Agent #2**	**Agent #1 score**	**Agent #1 score**
Student	Random	18	2
Student	MM_Null	16	4
Student	MM_open	12	8
Student	MM_Improved	13	7
Student	AB_null	15	5
Student	AB_open	13	7
Student	AB_Improved	13	7
Student 71.43%			

HEURISTIC #3			
EVALUATING STUDENT			
Agent #1	**Agent #2**	**Agent #1 score**	**Agent #1 score**
Student	Random	18	2
Student	MM_Null	14	6
Student	MM_open	13	7
Student	MM_Improved	9	11
Student	AB_null	15	5
Student	AB_open	9	11
Student	AB_Improved	14	6
Student 65.71%			

Heuristic 1

```
def heuristic_1():
     if game.is_loser(player):
          return float("-inf")
     if game.is_winner(player):
          return float("inf")
     own_moves = len(game.get_legal_moves(player))
     opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
     return float(own_moves - opp_moves)
```

No. of legal moves are a good indication of the game state as higher number of legal moves provide greater mobility for the player and the difference indicates which player has higher mobility and

hence the higher chances of winning.

Heuristic 2

```
def heuristic_2():
     if own_moves == 0 and opp_moves != 0:
          return float("-inf")
     if own_moves != 0 and opp_moves == 0:
          return float("inf")
     if own_moves == 0 and opp_moves == 0:
          return -10
     own_moves = len(game.get_legal_moves(player))
     opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
     if own_moves >= opp_moves:
          return ((own_moves / opp_moves) ** 2 + (opp_moves - own_moves))
     if own_moves < opp_moves:
          return -((own_moves / opp_moves) ** 2 + (opp_moves - own_moves))
```

The evaluation function I chose first checks to see if the move will result in a winning state for either player or a draw. If it results in the opponent winning, the evaluation function assigns it a utility of -infinity. If it results in the computer winning, the evaluation function assigns it a utility of infinity. If the move results in a draw, the evaluation function assigns it a utility of -3, so that best_move will only choose a move that results in a draw if all other options are terrible.

Heuristic 3

```
def heuristic_3():
     blank_spaces = game.get_blank_spaces()
     blank_spaces = len(blank_spaces)
     own_moves = len(game.get_legal_moves(player))
     opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
     if own_moves == 0 and opp_moves != 0:
          return float("-inf")
     if own_moves != 0 and opp_moves == 0:
          return float("inf")
     if own_moves == 0 and opp_moves == 0:
          return -10
     own_moves = len(game.get_legal_moves(player))
     opp_moves = len(game.get_legal_moves(game.get_opponent(player)))
     if own_moves>=opp_moves:
          return ((opp_moves-own_moves)/(blank_spaces+1))
     if own_moves<opp_moves:
          return -((opp_moves-own_moves)/(blank_spaces+1))
```

In order to evaluate the utility of a non-leaf node in the tree, we used a combination of the number of free spaces on the board and the number of legal moves available to each player. The number of free spaces on that board gives an indication of how much of the board is filled up. It is not always necessary that the legal moves would cover every square on the board and this metric is used to find how “isolated” a player is and whether there is opportunity to move to a more open area. The number of legal moves available to each player are used to not only maximize the possibilities of oneself, but also minimize the possibilities for others.

Heuristic Used : Heuristic 2 is being used.

Reasons :

- This evaluation function allows for very quick execution since it just takes into account the move lists for the computer and the opponent at each state that it evaluates as well as the original state.

- Division function used takes into account the fact that as (#moves the opponent has) grows very small in proportion to (#moves the computer has), the move is actually better for the computer.
- Subtraction function often cause the computer to pass up opportunities to restrict the opponent's movement simply to open up more move opportunities for itself.The move greatly restricts the move options of the opponent from what they were before, it is a better move.