Heuristic Analysis

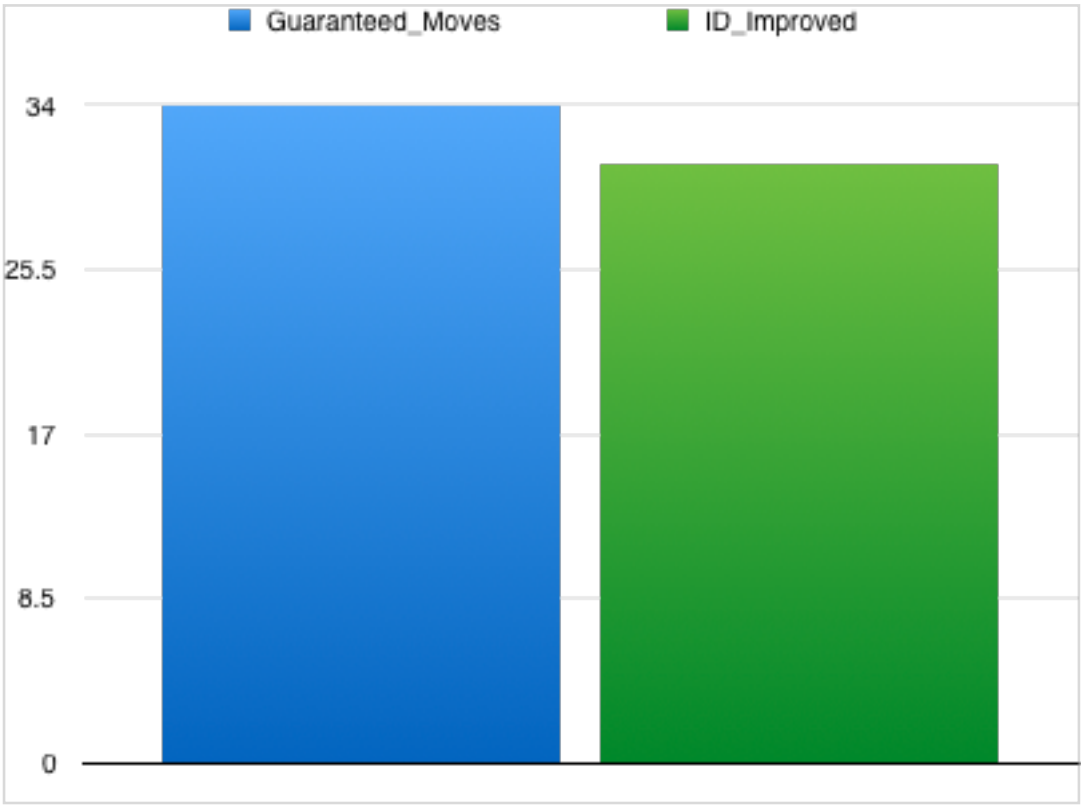
Throughout my testing of various heuristics, I was unable to find a particularly successful scoring algorithm; however, through this report, I hope to detail the rationale underpinning three of my attempts to create a better heuristic. I am not sure whether my inability to create a successful algorithm is a result of a poorly implemented `Custom_Player` class, underwhelming hardware, or simply a lack of inventiveness in creating an algorithm.

Heuristic 1: Guaranteed Moves

```
def custom_score(game, player):
    if game.is_winner(player):
        return float("inf")
    if game.is_loser(player):
        return float("-inf")
    opponent = game.get_opponent(player)
    player_moves = game.get_legal_moves(player)
    opponent_moves = game.get_legal_moves(opponent)

    player_moves_sum = len(player_moves)
    opponent_moves_sum = len(opponent_moves)

    if player == game.active_player:
        for (x, y) in opponent_moves:
            if game.move_is_legal((x, y)):
                opponent_moves_sum -= 1
    else:
        for (x, y) in player_moves:
            if game.move_is_legal((x, y)):
                player_moves_sum -= 1

    return float(player_moves_sum - opponent_moves_sum)
```

The first heuristic evaluated is a relatively rudimentary extension of the `improved_score` heuristic provided in the course materials, with one key difference. This new algorithm discounts moves to spots on the board that could be filled by the opponent's next move. In

theory, this should help mitigate the risk of the player being trapped by his or her opponent by proactively accounting for "escape routes" getting cut off. However, this only resulted in a modest improvement over the `ID_Improved` player using the `improved_score` heuristic.



Heuristic 2: Reachable Spaces

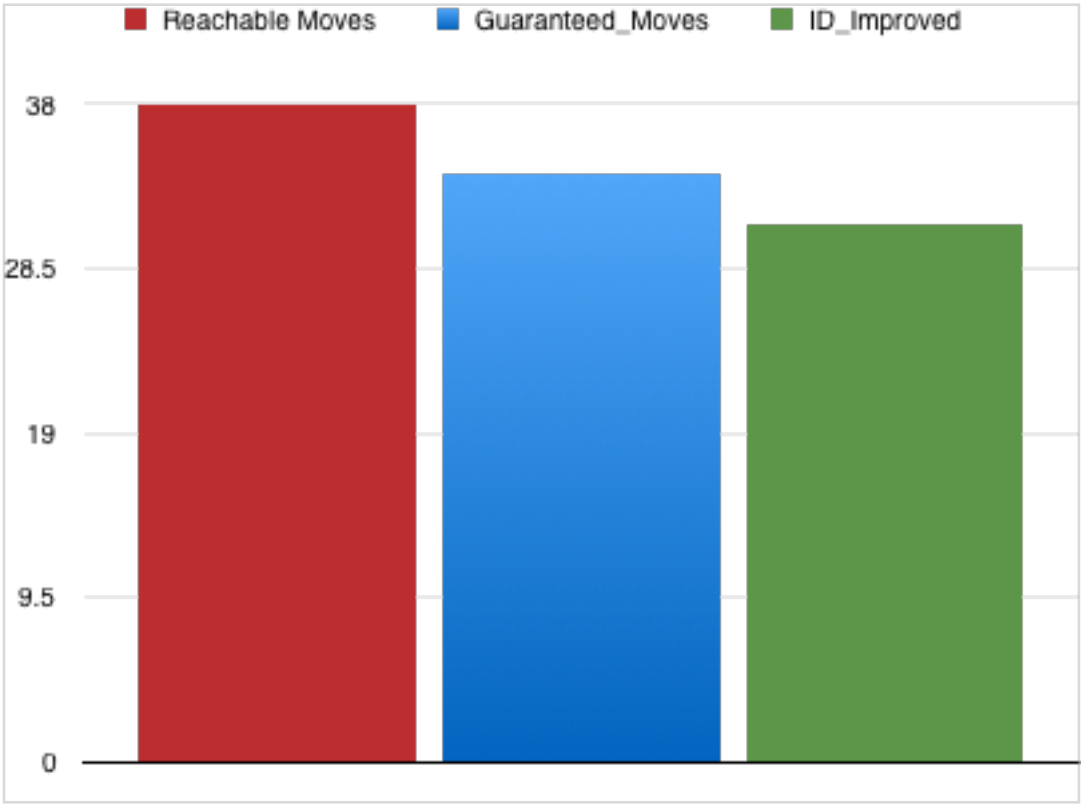
```
def custom_score(game, player):
    if game.is_winner(player):
        return float("inf")
    if game.is_loser(player):
        return float("-inf")

    player_reachable = get_reachable_spaces(game, player)
    opponent_reachable = get_reachable_spaces(game, game.get_opponent(player))
    return float(player_reachable - opponent_reachable)

def get_reachable_spaces(game, player):
    movements = [(1, 2), (1, -2), (-1, 2), (-1, -2)]
    avail_spaces = game.get_blank_spaces()
    legal_moves = game.get_legal_moves(player)
```

```
    reachable_count = len(legal_moves)
    moves_to_search = legal_moves[:]
    searched_moves = legal_moves[:]
    searched_moves.append(game.get_player_location(player))

    def checkmove(new_move, reachable_count):
        if avail_spaces.count(new_move) and not searched_moves.count(new_move):
            reachable_count += 1
            searched_moves.append(new_move)
            moves_to_search.append(new_move)
        return reachable_count

    while len(moves_to_search):
        move = moves_to_search.pop(0)
        (x, y) = move
        for (i, j) in movements:
            reachable_count = checkmove((x + i, y + j), reachable_count)
            reachable_count = checkmove((x + j, y + i), reachable_count)

    return reachable_count
```

The second heuristic attempts to look beyond the immediate set of moves and instead evaluate the ultimate potential value of a state in terms of reachable spaces. This is accomplished by first forming a queue of reachable spaces, starting with `game.get_legal_moves(player)`, and creating a list of items that have entered the queue. The queue is processed node-by-node; for each node:

1. all of eight possible moves are calculated
2. for each possible move, if that spot is a member of `game.get_blank_spaces()` and **not** a member of the list of visited nodes
3. if those conditions are met, the node is added to the queue and to the list of searched moves.

This strategy offers a modest improvement of the first heuristic, but is still far from perfect. Especially towards the beginning of the game this value function is extremely computationally expensive.



Heuristic 3: Best of Both Worlds

```
def custom_score(game, player):
    if game.is_winner(player):
        return float("inf")
    if game.is_loser(player):
        return float("-inf")

    if game.move_count > 12:
        player_reachable = get_reachable_spaces(game, player)
        opponent_reachable = get_reachable_spaces(game,
game.get_opponent(player))
        return float(player_reachable - opponent_reachable)

    else:
        opponent = game.get_opponent(player)
        player_moves = game.get_legal_moves(player)
        opponent_moves = game.get_legal_moves(opponent)

        player_moves_sum = len(player_moves)
        opponent_moves_sum = len(opponent_moves)
```

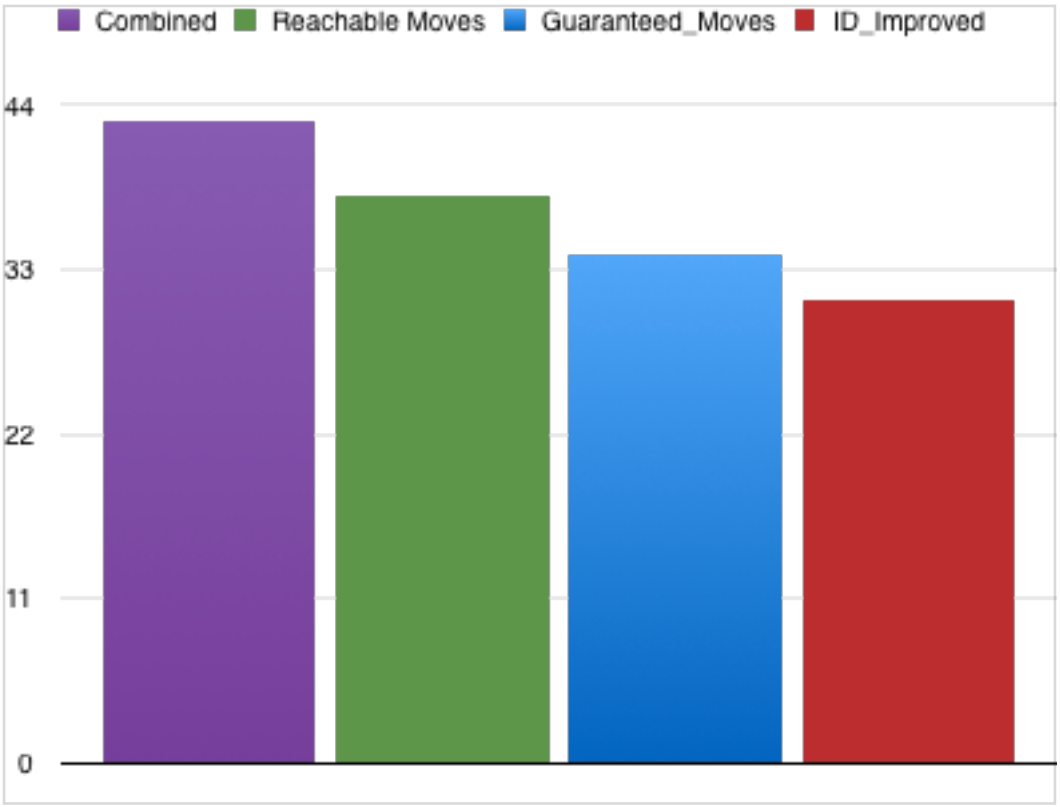
```
        if player == game.active_player:
            for (x, y) in opponent_moves:
                if game.move_is_legal((x, y)):
                    opponent_moves_sum -= 1
        else:
            for (x, y) in player_moves:
                if game.move_is_legal((x, y)):
                    player_moves_sum -= 1

        return float(player_moves_sum - opponent_moves_sum)

def get_reachable_spaces(game, player):
    movements = [(1, 2), (1, -2), (-1, 2), (-1, -2)]
    avail_spaces = game.get_blank_spaces()
    legal_moves = game.get_legal_moves(player)
    opponent_moves = []

    reachable_count = len(legal_moves)
    moves_to_search = legal_moves[:]
    searched_moves = legal_moves[:]
    searched_moves.append(game.get_player_location(player))

    if player != game.active_player:
        opponent_moves = game.get_legal_moves(game.active_player)

    def checkmove(new_move, reachable_count):
        if avail_spaces.count(new_move) and not searched_moves.count(new_move):
            reachable_count += 1
            searched_moves.append(new_move)
            moves_to_search.append(new_move)
        return reachable_count

    while len(moves_to_search):
        move = moves_to_search.pop(0)
        (x, y) = move
        for (i, j) in movements:
            if not opponent_moves.count((x, y)):
                reachable_count = checkmove((x + i, y + j), reachable_count)
                reachable_count = checkmove((x + j, y + i), reachable_count)
```

```
    return reachable_count
```

The third heuristic attempts to marry the strongest aspects of the two previously evaluated algorithms by combining the awareness of `guaranteed_moves` with the future focused approach of `reachable_spaces`. The primary issue with the `reachable_spaces` heuristic was its computational cost, which is addressed by two means in this third score function:

1. Not invoking the `reachable_spaces` heuristic until the middle of the game
2. Limiting the breadth of the `reachable_spaces` search by discarding nodes that are reachable by the opponent's next move.

By combining the two heuristics, we should be able to traverse to a greater depth using iterative deepening in the earlier rounds, while still using the more effective yet more expensive heuristic for later-game scenarios.



Despite passing the tests and consistently out performing the `ID_Improved` player, my scores in the tournament were very poor, leading me to believe there was an error in my iterative deepening or alpha-beta search algorithm. However, after writing and rewriting these functions, I was unable to find my error. I apologize for the delay in submission.