```
In [1]: from IPython.display import Image
```

Three scoring functions were developed for the isolation tournaments to compete against the ID_improved scoring function. The ID_improved scoring function scores each game state using the difference of the primary agent moves and the opponent moves. That is, agent_moves - opponent_moves. This favors states in which the primary agent has more options to move than its opponent.

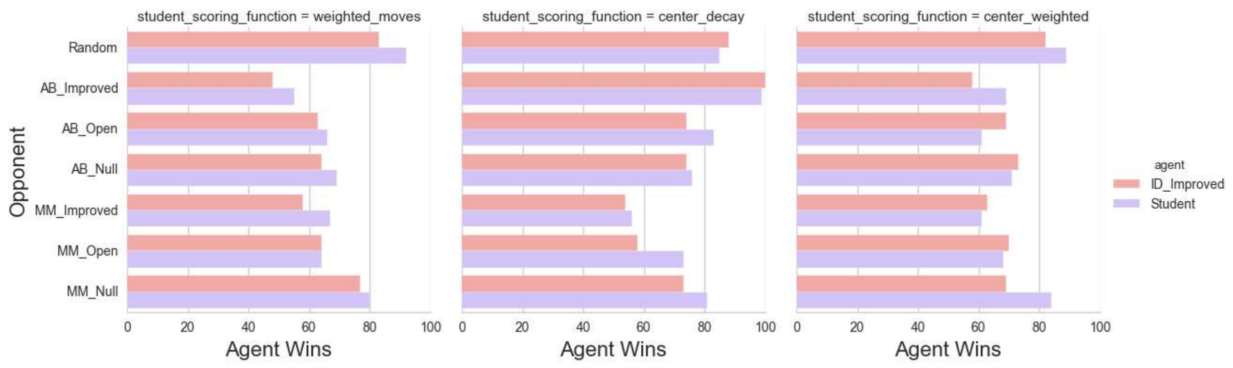
Student Agent Scoring Functions

1. **Weighted Move Differences**: This is similiar to the ID improved scoring function but weights the opponent moves by an optional argument supplied by the user. The default argument is 2 meaning that 2 times the opponents move number will be subtracted from the primary agent move number. This heavily favors states in which the opponent has less moves.
2. **Center Weighted Moves**: This adds a term to weighted moves differences which favors moves to the center with a user supplied center_weight argument. The default center_weight is 2, favoring moves to the center by the primary agent by 2-fold and penalizing opponent moves to the center by 2-fold.
3. **Center Weighted Moves with Decay**: This adds a decay term to the center weighted moves scoring function which diminishes the center_weight term as spaces are blocked. The rationale for this was arrived at by playing and reading about isolation. A very succesful strategy is to create islands of squares where both players moves on a seperate island. The biggest island wins.

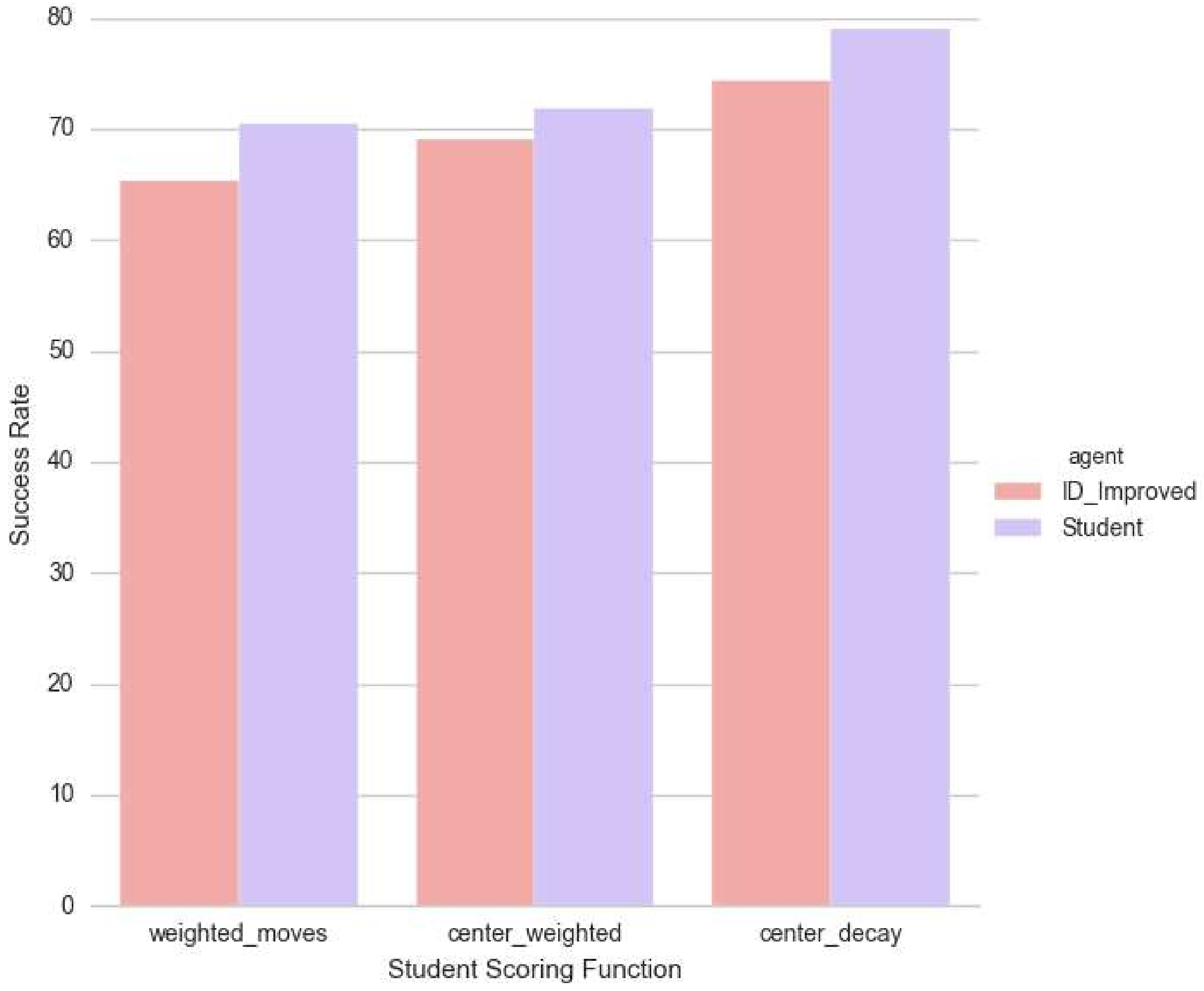
Evaluation

Functions were evaluated by running 100 matches against each of 7 opponents for each primary agent (ID_Improved, Student). ID_improved used the ID_Improved scoring function described above. Student used one of three scoring functions as described above.

Success Count ID_Improved Compared to Student, by Opponent



Aggregated Success Rate ID_Improved Compared to Student



Are these differences statistially signficant?

A two tailed Z-test was conducted for the aggregated results of ID_Improved and Student for each scoring function and was found to be signfiant at 95% confidence level. Our best performer is center weighted with decay.

Scoring Heurisitic choice

My choice of scoring heuristic is center weighted moves with decay (centerdecay_weighted_moves). The reasoning behind this choice was threefold:

- Highest performance of the 3 scoring heuristics investigated.
- Most closely captures the strategy of creating islands by dominating the center and later occupying an island with more moves than ones opponent (needs work but this is an approximation of that strategy)
- Has room for development by adjusting the decay function and the two weight terms.

Future Improvement

The scoring function could be improved in the following manner:

- optimize decay function

- optimize both weight terms by collecting data with a variety of weights and performing a regression analysis
- add a term to further enhance the creation of an island early in the game and staying there. That is, put in some terms where the primary agent distances itself from its opponent and sticks on an island with the most moves. This is pretty complex. I imagnine a scenario where we use reinforement learning where we allow our agent to evaluate the utility of sampled states and develop its own strategy.

Scoring Function Code

Weighted Move Differences

```
def weighted_openmovediff_score(game, player, weight=2):
    """This evaluation function outputs a score equal to the differenc
e in the number
    of moves available to the two players. Contains an optional weight
 parameter
    as a multiplier to the  opponents score.

    Parameters
    ----------
    game : `isolation.Board`
        An instance of `isolation.Board` encoding the current state of
 the
        game (e.g., player locations and blocked cells).

    player : hashable
        One of the objects registered by the game object as a valid pl
ayer.
        (i.e., `player` should be either game.__player_1__ or
        game.__player_2__).

    weight: int
        Optional int argument which weights the oppoents moves by opp_
moves * weight.
        This functions to penalize choices where the opponent has more
 moves.

    Returns
    ----------
    float
        The heuristic value of the current game state
    """
    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 * weight)
```

Center Weighted Move Differences

```
def center_weighted_moves(game, player, weight= 2, center_weight= 2):
    """This evaluation function outputs a score based on weighted diff
erence of
    the difference in own moves and opponent moves, further weighted t
o favor center_weight
    row and column squares.

    Parameters
    ----------
    game : `isolation.Board`
        An instance of `isolation.Board` encoding the current state of
 the
        game (e.g., player locations and blocked cells).

    player : hashable
        One of the objects registered by the game object as a valid pl
ayer.
        (i.e., `player` should be either game.__player_1__ or
        game.__player_2__).

    weight: int
        Optional int argument which weights the oppoents moves by opp_
moves * weight.
        This functions to penalize choices where the opponent has more
 moves.

    center_weight: int
        Optional int argument which further weights center moves.


    Returns
    ----------
    float
        The heuristic value of the current game state
    """
    if game.is_loser(player):
        return float("-inf")

    if game.is_winner(player):
        return float("inf")

    center_col= math.ceil(game.width/2.)
    center_row= math.ceil(game.height/2.)

    own_moves = game.get_legal_moves(player)
    opp_moves = game.get_legal_moves(game.get_opponent(player))
    num_own_moves= len(own_moves)
    num_opp_moves= len(opp_moves)
```

```
        opp_weight, own_weight= weight,1

        for move in own_moves:
            if move[0]== center_row or move[1]== center_col:
                own_weight *= center_weight

        for move in opp_moves:
            if move[0]== center_row or move[1]== center_col:
                opp_weight *= center_weight

        return float((num_own_moves * own_weight) - (num_opp_moves * opp_w
    eight))
```

Center Weighted Move Differences with Decay

```
def centerdecay_weighted_moves(game, player, weight= 2, center_weight=
 2):
        """This evaluation function outputs a score based on weighted
 difference of
        the difference in own moves and opponent moves, further weight
ed to favor center_weight
        row and column squares. An additional decay factor has been ad
ded which
        decreases center weighting as a function of the number of unbl
ocked squares.

        Parameters
        ----------
        game : `isolation.Board`
            An instance of `isolation.Board` encoding the current stat
e of the
            game (e.g., player locations and blocked cells).

        player : hashable
            One of the objects registered by the game object as a vali
d player.
            (i.e., `player` should be either game.__player_1__ or
            game.__player_2__).

        weight: int
            Optional int argument which weights the oppoents moves by
 opp_moves * weight.
            This functions to penalize choices where the opponent has
 more moves.

        center_weight: int
            Optional int argument which further weights center moves.


        Returns
        ----------
        float
            The heuristic value of the current game state
        """
        if game.is_loser(player):
            return float("-inf")

        if game.is_winner(player):
            return float("inf")

        center_col= math.ceil(game.width/2.)
        center_row= math.ceil(game.height/2.)
        own_moves = game.get_legal_moves(player)
        opp_moves = game.get_legal_moves(game.get_opponent(player))
        num_own_moves= len(own_moves)
        num_opp_moves= len(opp_moves)

        initial_moves_available= float(game.width * game.height)

        num_blank_spaces= len(game.get_blank_spaces())

        decay_factor= num_blank_spaces/initial_moves_available

        opp_weight, own_weight= weight,1

        for move in own_moves:
            if move[0]== center_row or move[1]== center_col:
                own_weight *= (center_weight * decay_factor)

        for move in opp_moves:
            if move[0]== center_row or move[1]== center_col:
                opp_weight *= (center_weight * decay_factor)

        return float((num_own_moves * own_weight) - (num_opp_moves * o
pp_weight))
```