Analysis of Search Agents in Knight's Isolation Game

Daniel Szemerey

Udacity

Author Note

This project is an obligatory delivery for Project III in

Udacity's Artificial Intelligence Nanodegree

Abstract

This report summarizes the analysis done for Project III, Adversarial Search of Udacity's Artificial Intelligence Nanodegree. The core analysis was done by running both player's and opponent's agent using **Alpha-Beta Pruning** with Iterative Deepening. Player's heuristics has been varied to test **advanced heuristics**, while opponent's strategy remained constantly *Liberty Difference*.

Additionally, a **Monte Carlo Tree Search** strategy has been implemented, using *UCT selection function*. Test results showed highly varied (therefore poor) performance even against *Random Player*, which results from the game cutting of the match in an unexplainable position. The tree is being built properly and running diagnostics shows the algorithm is correctly implemented.

Keywords: Artificial Intelligence, Alpha-Beta Pruning, Heuristics, Monte Carlo Tree Search, UCT

Table of Contents

Abstract **2**

Charting the Data **4**

Advanced Heuristics **7**

Base strategy: Liberty Difference 7

Weighted Self - Defensive 7

Weighted Opponent – Aggressive 8

Weighted Linear function 8

Weighted Relu – Rectified combination of Weighted Self & Weighted Opponent 9

Weighted Binary 10

Monte Carlo Tree Search **11**

References **14**

Dataset **14**

Charting the Data

Both players used the same Alpha Beta Pruning algorithm, but different heuristic function to determine the value of intermediate nodes. Each game was run using fair_matches flag enabled, with a search depth of 5 using iterative deepening. Games were played for 120 rounds, using 10 parallel computing processes.

Which heuristic is a better proxy for the actual utility of the state?

This analysis shows that some advanced heuristics are a better proxy of the actual utility of the state (like *Weighted Binary*), than *Liberty Difference*, while others are worse. In order to show this, the depth limit was capped for both player at 5 to make a meaningful comparison.

Below shows the result of each heuristics:

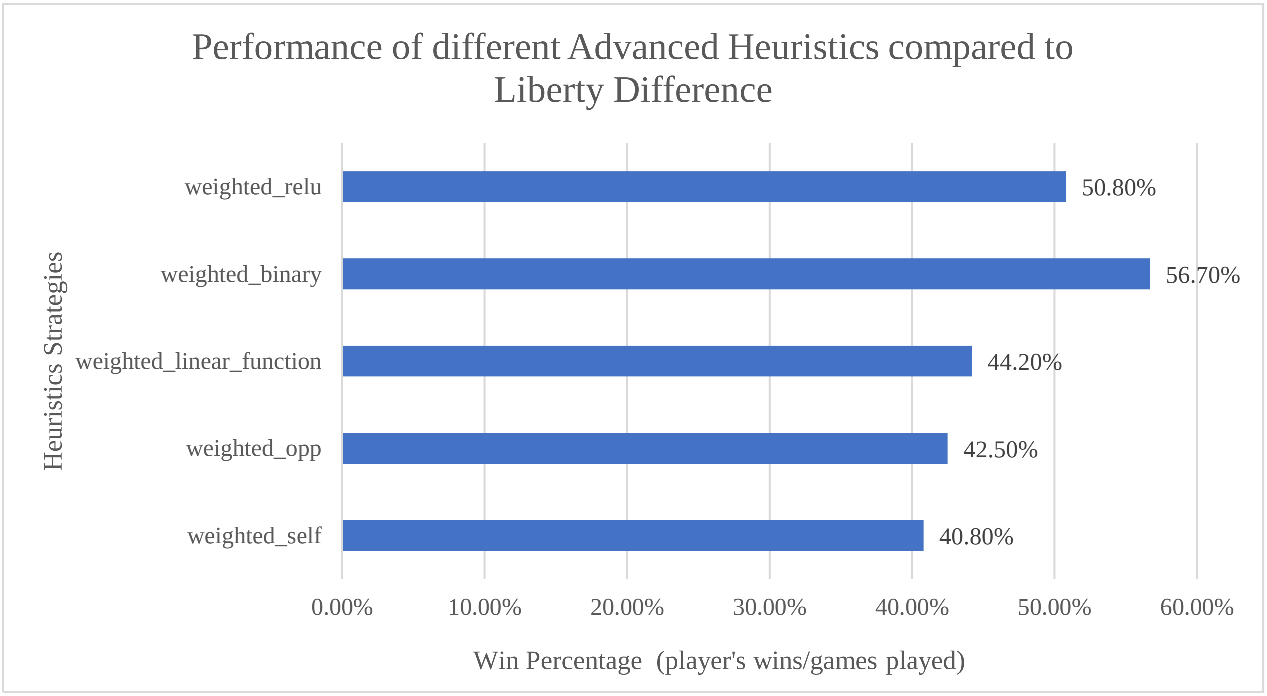
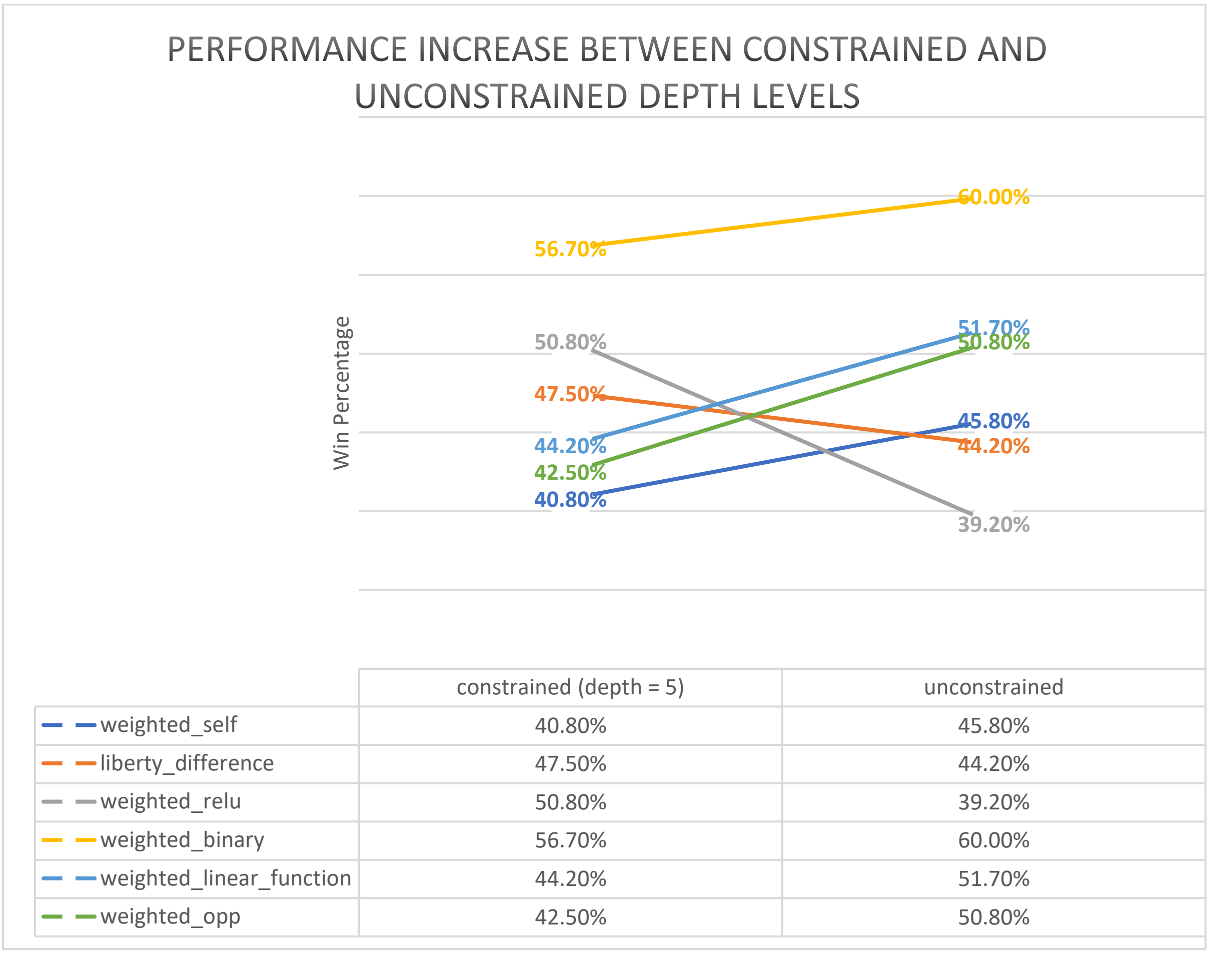


Figure 1 Win Percentage for each heuristic in the Alpha Beta Pruning Algorithm. Opponent always used the difference between each player in their available moves.

Analyze the search depth your agent achieves using your custom heuristic. Does search speed matter more or less than accuracy to the performance of your heuristic?

The below table shows how with some advanced heuristics agents perform better, even if the search depth is slightly shallower



	constrained (depth = 5)	unconstrained
weighted_self	40.80%	45.80%
liberty_difference	47.50%	44.20%
weighted_relu	50.80%	39.20%
weighted_binary	56.70%	60.00%
weighted_linear_function	44.20%	51.70%
weighted_opp	42.50%	50.80%

Figure 2 For some heuristics the time required to do the more complex calculation outweigh the gain in performance. For some like weighted_binary, the additional calculation comes with a significant performance gain.

Player's Strategy	**Win Percentage**	**Player's Avg. Search Depth**	**Opponent's Avg. Search Depth**	**Opponent's Strategy**	**Depth Difference**
weighted_relu	39.20%	26.77	27.00	liberty_difference	0.23

Player's Strategy	**Win Percentage**	**Player's Avg. Search Depth**	**Opponent's Avg. Search Depth**	**Opponent's Strategy**	**Depth Difference**
weighted_binary	60.00%	26.37	26.59	liberty_difference	0.22
weighted_linear_function	51.70%	27.54	27.89	liberty_difference	0.35
weighted_opp	50.80%	25.23	26.13	liberty_difference	**0.90**
weighted_self	45.80%	25.57	25.96	liberty_difference	0.39

Figure 3 Table showing the difference in the avarage max depth that each heuristic is able to achieve, compared to the baseline liberty_difference heuristic

Advanced Heuristics

"What features of the game does your heuristic incorporate, and why do you think those features matter in evaluating states during search?"

Base strategy: Liberty Difference

This strategy was used by the opponent for base comparison. As the data shows only the *Weighted Binary (0.567)* and the *Weighted Relu (0.508)* heuristic outperformed this strategy. Given the error margin, only the *Weighted Binary* shows solid performance advantage to *Liberty Difference*.

The formula is:

$$score_p\ (x) = \sum m_p - \sum m_o$$

Where,

- $score_p$ is an assigned intermediate utility value
- m_p is a liberty of the player
- m_o is a liberty of the opponent

Weighted Self - Defensive

This heuristic gives a higher score, if the player has more moves. Compared to the *Base Strategy* this algorithm gives a higher value if the player has a high number of liberties.

This heuristic tries to make sure the player has as many moves as possible, resulting in a defensive game.

The formula is:

$$score_p\ (x) = \left(\sum m_p\right)^2 - \sum m_o$$

Where,

- $score_p$ is an assigned intermediate utility value
- m_p is a liberty of the player
- m_o is a liberty of the opponent

Weighted Opponent – Aggressive

This heuristic gives a higher score, if the opponent has more moves. Compared to the *Base Strategy* this algorithm gives a higher value if the opponent has a high number of liberties. This heuristic tries to make sure the player has as many moves as possible, resulting in a defensive game.

The formula is:

$$score_p\ (x) = \sum m_p - \left(\sum m_o\right)^2$$

Where,

- $score_p$ is an assigned intermediate utility value
  - m_p is a liberty of the player
  - m_o is a liberty of the opponent

Weighted Linear function

This heuristic combines *Weighted Self* and *Weighted Opponent*. As the game progresses, *Weighted Self (Defensive)* strategy gradually increases in relevance, while the *Weighted Opponent (Aggressive)* strategy gradually decreases in relevance.

The formula is:

$$score_p\ (x) = w_1 * \left(\left(\sum m_p\right)^2 - \sum m_o\right) - w_2 * \left(\sum m_p - \left(\sum m_o\right)^2\right)$$

Where,

- $score_p$ is an assigned intermediate utility value

- m_p is a liberty of the player
- m_o is a liberty of the opponent
- w_1 is the weight of weighted self, with value:

$$w_1(ply_{count}) = \frac{ply_{count}}{\boldsymbol{board\_size}}$$

- w_2 is the weight of weighted opponent, with value:

$$w_2(ply_{count}) = 1 - \frac{ply_{count}}{\boldsymbol{board\_size}}$$

- ply_{count} is number of moves played until that state in the game

Weighted Relu – Rectified combination of Weighted Self & Weighted Opponent

Rather than a gradual fade in - fade out like with the *Weighted Linear function* this heuristic has a gradual change with a drop of point during the progression of the game, from which on the value of the weight is 0.

This means after a gradual decrease of relevance and a cut-off point, the *Weighted Opponent* is no longer consider, and antagonistically the *Weighted self* has a weight of 0 until a starting point, from which on it gradually increases. Starting point and Cut-off point are both 0.5 in the below formula and the test cases.

The formula is:

$$progress\ (ply_{count}) = ply_{count}/board\_size$$

$$w_1\ (progress) = \max(0.0, (\,progress * 2) - 1)$$

$$w_2\ (progress) = \ \max\ (0.0, (\ 1 - \ progress * 4))$$

$$score_p\ (x) = w_1 * \left(\left(\sum m_p\right)^2 - \sum m_o\right) - w_2 * \left(\sum m_p - \left(\sum m_o\right)^2\right)$$

Where,

- $score_p$ is an assigned intermediate utility value
- m_p is a liberty of the player
- m_o is a liberty of the opponent
- ply_{count} is number of moves played until that state in the game
- $progress$ is how far in percentage we are from the game end (0 – started, 1 – finished)

Weighted Binary

This herustic is similar to *Weighted Relu*, except it doesn't have a period of gradual increase or decrease of weight; rather the weight remains constant 1 or 0 until a cut-off or starting point.

$$progress\ (ply_{count}) = ply_{count}/board\_size$$

$$score_p\ (x) = \left(\sum m_p\right)^2 - \sum m_o\ , if\ progress > 0.5$$

$$score_p\ (x) = \sum m_p - \left(\sum m_o\right)^2 , if\ progress \leq\ 0.5$$

Where,

- $score_p$ is an assigned intermediate utility value
- m_p is a liberty of the player
- m_o is a liberty of the opponent
- ply_{count} is number of moves played until that state in the game
- $progress$ is how far in percentage we are from the game end (0 – started, 1 – finished)

Monte Carlo Tree Search

An implementation based on Artificial Intelligence: A Modern Approach has been created. Below is the pseudocode of the implementation. Due to unresolved issues, the performance of the program is severely hindered by an unknown reason cutting off the game midway through. In addition, an implementation to pass the tree forward has been created, but serializing an object severely impacted the number of simulations that method can run in the given time.

```
class TreeNode(state, parent, origin_action)
      init()
            Parent ← parent
            State ← state
            Children ← list: children

            Uct ← 0
            Win_count ← 0 (or 2)
            Sim_count ← 0 (or 2)

            Origin_action ← origin_action

      init_children()
            Children = list: for all available actions return a TreeNode with
            the resulting state


class MonteCarloTreeSearch(state)
      take_action()
            Tree ← new TreeNode()
            while (available resources):
                  leaf ← select(tree)
                  result ← simulate(leaf.state)
                  backpropagate(result, leaf)
                  return the child of tree with the maximum sim_count

      select(tree)
            if tree.state is terminal state
                  return
```

```
        if tree.children length is greater than 0
            all children calculate uct with uct function
            best_child ← select child with highest score
            return select(best_child)
        else
            tree.init_children()
            random_child ← randomly choose from tree.children
            calculate uct of random_child
            return random_child

    simulate(state)
        if state is a terminal state
            return the utility of the state for the player
        else
            next_action ← choose a random next step
            simulate(next_action)

    backpropagate(result, leaf)
        leaf.sim_count ← incremented by +1

        if result is a tie
            leaf.win_count ← incremented by +1
        if leaf.state is the same as the players
            leaf.win_count ← incremented by +1

        if leaf doesn't have a parent
            end
        else
            backpropagate(result, leaf.parent)

    uct(state)
        return sqrt(log10(state.parent.sim_count)/state.sim_count) * an
exploration weight + (state.win_count/state.sim_count)
```

References

Artificial Intelligence a Modern Approach (AIMA) 4th edition, Stuart Russell and Peter Norvig, 2020

Dataset

Entire Project Folder with code can be found here (folder): https://github.com/szemyd/ai-udacity/tree/main/Adversarial%20Search

Raw Dataset of tests can be found here (csv): https://github.com/szemyd/ai-udacity/blob/main/Adversarial%20Search/results.csv