COMP6231: Search Heuristics for Isolation

Julio Cesar Aguilar Jimenez[1,2,3]

[1] *School of Electronics and Computer Science, University of Southampton*
[2] *MSc. Artificial Intelligence, ID. 29312175*
[3] *Author email: jcaj1n17@soton.ac.uk*

January 9, 2018

The results of applying 7 different heuristics using alpha-beta pruning with iterative deepening against different opponents that use different algorithms in the "Isolation" game is presented. The evaluation functions proposed include simple approaches, use of weights, dynamic weights, aggressive behaviour and strategy changes through the game, showing these last a better performance.

Galileo: 'You cannot teach a man anything; you can only help him discover it in himself'

1. INTRODUCTION

A **heuristic** is a technique designed for *solving* a problem more quickly when classic methods are too *slow*, or for finding an approximate solution when classic methods fail to find any *exact* solution.

Isolation is a deterministic, *two-player* game of perfect information in which the players alternate turns moving a *single piece* from one cell to another on a *board*.

In this paper we will present different **heuristics** that will work as *evaluation functions* in order to *calculate* a *win* condition for our *player* or *agent*.

The **Agent** must *compete* against other agent called **Archenemy** who is *highly competent*, this two players will play several matches against *opponents* that use the algorithms of *minimax*, *alpha beta pruning*, and **simple** heuristics. This will allow us to know how *well* the new heuristics are *performing*.

2. APPROACH

2.1. Game

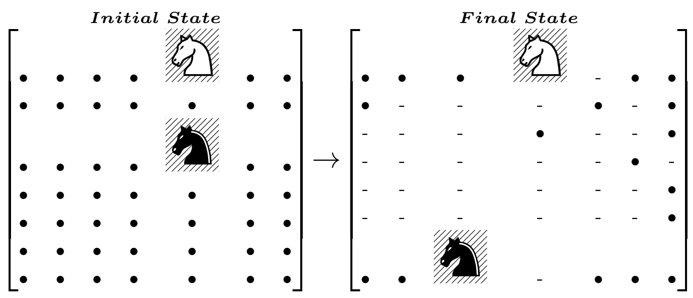
In this **Isolation** version, each agent is restricted to *L-shaped* movements (like a *knight* in chess) on a rectangular (chess) grid of size 7×7.

The *players* can move to any *open* cell on the board that is *2-rows and 1-column or 2-columns and 1-row* away from their current position on the board. Whenever either player *occupies* a *cell*, that cell becomes **blocked** for the remainder of the game. The first player with no remaining *legal moves* **loses**, and the opponent is declared the **winner**.

Movements are blocked at the edges of the board, however, the player can **jump** blocked or occupied spaces like a *knight* in chess.

To avoid *misinterpretation* of the results, the **initial** state is *random*, *i.e.*, the *positions* of both players are random and the *number* of games will be *even* in this way both players will have the *opportunity* to be the **first** or **second** to move the same amount of times.

Below the *matrices* indicate a possible **initial** and **final** state for the **Isolation** game, where the *white knight* is the **winner** due to the next to move is the *black knight* and it has not available moves anymore.



2.2. Opponents

The **efficiency** of the *heuristics* is tested as a *competition* against 7 *opponents*. 3 *opponents* use the **minimax** algorithm with a depth of 3, 3 $\boldsymbol{\alpha - \beta}$ **pruning** with depth of 5, and a *random* opponent that only chooses *random* positions. *Udacity* offer sample players to test your **Agent**. [4]

As shown in the Figure 2.1, each opponent uses *null*, *open* and *improved* **simple heuristics** presented in Sections 2.6.1, 2.6.2, 2.6.3.

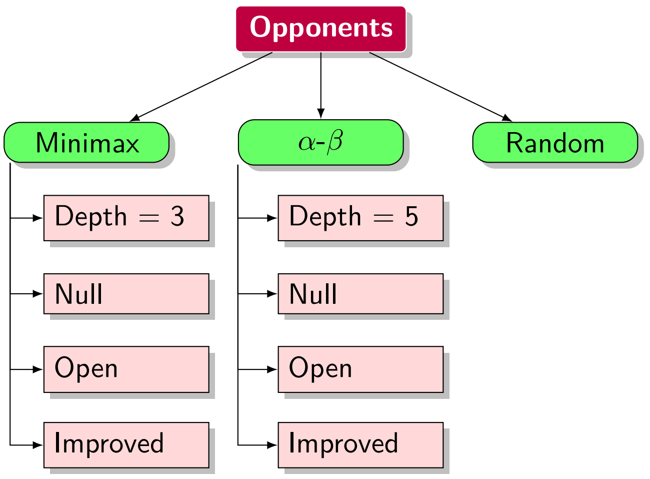


Fig. 2.1. Opponents to test the heuristics

2.3. Agent and Archenemy

Both will use the $\boldsymbol{\alpha - \beta}$ **pruning** algorithm with *iterative deepening*. Our **player** will use the different *heuristics* presented in the Section 2.7 and the **Archenemy** will always use the *improved* heuristic as shown in Figure 2.2. The objective is to surpass this player.

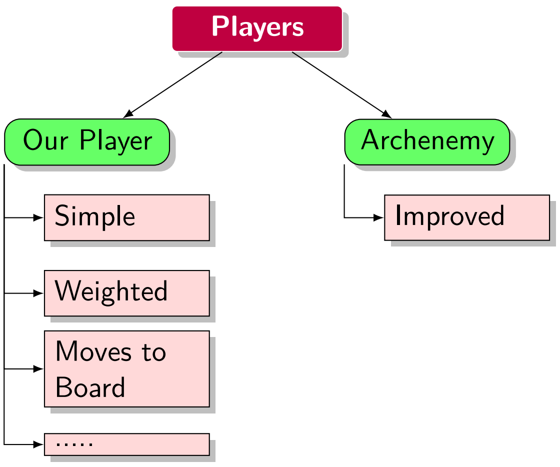


Fig. 2.2. Players to test in the competition

2.4. Language

The **Udacity** online platform [1], has a graphical interface in *HTML5* and *JavaScript* languages to test this game, it contains the *board* and a *text field* to put the game's movement history and observe the behaviour and result. [2].

Additionally **Udacity** has an *API* [3] for the control of the *board* written in **Python**, so the **Python** language was *selected* for the creation of the different *classes* and *methods*.

2.5. Algorithms

2.5.1. Minimax (MM)

Minimax is a *backtracking* algorithm and it has 2 players. The **maximiser** tries to get the *highest* score possible while the **minimiser** tries to get the *lowest*.

In Figure 2.3, assuming 4 **final** states and being **maximiser** who moves *first*, we can see that by taking the *left node*, **minimiser** will take the *minimum* value $min(3,5)$ so 3 will be the **final** value of that *node*.

On the *right* side **minimiser** will decide $min(2,9)$, so the value of the node is 2.

Since **maximiser** will take the *highest* value $max(3,2)$, the optimal move is *left node* with 3.

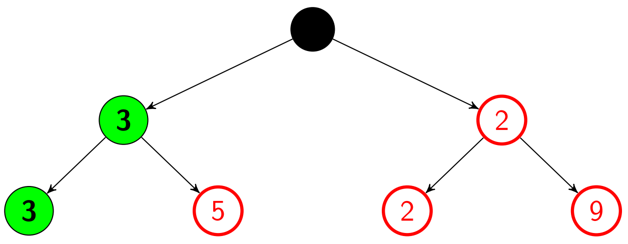


Fig. 2.3. Minimax example

2.5.2. Alpha-Beta Pruning

Alpha-Beta Pruning is an optimisation technique for *minimax* algorithm. It *cuts off* branches in the game tree which need not be *searched* because there already exists a *better* move available.

It passes 2 extra *parameters* in the *minimax* function, namely α and β. The initial values of *alpha* and *beta* are $[\alpha = -\infty, \beta = \infty]$ and the condition to prune is $\beta <= \alpha$. α and β are the best values that the **maximiser** and **minimiser** currently can guarantee at that level or above respectively.

In Figure 2.4, assuming that **maximiser** moves first, then **minimiser**, then **maximiser** will decide between 3 and 5, α will evaluate $max(3, -\infty)$ and then since the condition $\beta(\infty) <= \alpha$ is false it will continue with $max(3,5)$, so 5 will be the value of D and pass it to B. At B, $\beta = min(-\infty, 5)$ so $\beta = 5$.

At $E\,(\alpha = -\infty,\ \beta = 5)$, going to the left $\alpha = 6$ and as $\beta(5) <= \alpha(6)$ is true, it breaks returning 6 to B, so $\beta = min(5, 6)$ and 5 is returned to A .

At $C\,[\alpha = 5, \beta = \infty]$ and after repeating the process in F we have $\beta = 2$ and since $\beta(2) <= \alpha(5)$, it prunes the entire G, returning 2 to A. Since $max(5, 2) = 5$, then the optimal value is 5 .

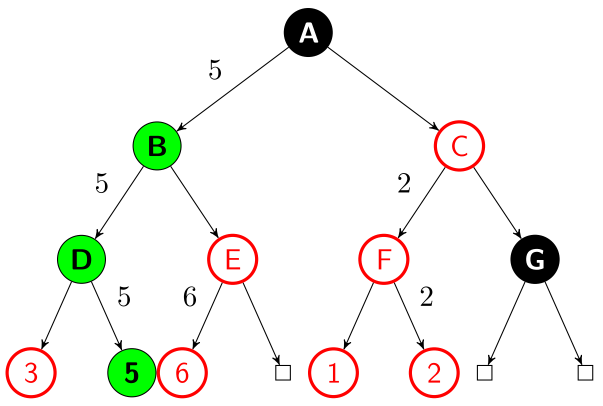


Fig. 2.4. Alpha-Beta pruning example

2.6. Heuristics of Opponents

Minimax picks the *least* or *greatest* score each round, the next step is to compute **scoring heuristic** that *maximise* our player's moves and *minimise* the opponent's moves leading to a *win condition* for our player.

The *time* spent in the **evaluation function** reduces time that can be spent exploring the *game tree*. For this reason, it's generally considered to keep the evaluation function *simple* and within *linear* or *constant* time. Opponents use the following 3 *evaluation functions.*

2.6.1. Null Score

This *heuristic* presumes no knowledge for **non-terminal** states, and returns 0 for all other *states.*

$$H(t) = \begin{cases} -\infty, & \text{if } lose \\ \infty, & \text{if } win \\ 0, & any\ other\ state \end{cases} \tag{2.1}$$

2.6.2. Open Move Score

This *heuristic* evaluates the number of *moves open* ***n*** for the *agent*, the values for **win** and **lose** *state* are the same as in Eq. (2.1).

$$H(t) = \#\ legalMoves \tag{2.2}$$

2.6.3. Improved Score

This *heuristic* returns the **difference** between the movements of the *player* and *opponent*, the **win** and **lose** state values are the same as in Eq. (2.1). In Eq. (2.3) $p = \#playerMoves$ and $o = \#opponentMoves$.

$$H(t) = p - o \tag{2.3}$$

2.6.4. Random Player

The **random player** do not have a defined **heuristic**, it simply choose a *random movement* from the list of available movements, in case it has not, it returns an *invalid* movement indicating the **lose** *state* and the end of the game.

2.7. Agent Heuristics

The **win** and **lose** *states* return the same values for every *heuristic* as in the Eq. (2.1), *i.e.*, ∞, $-\infty$ respectively.

2.7.1. Simple

This is the **simplest** *heuristic* and its behaviour is the same as in the Section 2.6.3, a *subtraction* between de *agent* moves and the *opponent*. It was created for *testing* purposes, to observe how the algorithms of *minimax*, *alpha beta pruning* and *iterative deepening performed.*

2.7.2. Weighted

This is a modified *version* of *Simple Score* 2.7.1 based on **weights**, where our movements will have a *greater weight* than the opponent's, *i.e.*, for our *agent* having **more** moves is more important than having **less** moves for the opponent.

$$H(t) = (p \times 2) - o \tag{2.4}$$

2.7.3. Moves to Board

Now to our **Weighted** model we will *add* a sense of *time*, increasing the *importance* of the board places through the game, in this way the importance of the *movements* changes. In Eq. (2.5), $w = p \times 2$ and $m = currentMoves/boardSize$.

$$H(t) = (w \times m) - o \tag{2.5}$$

2.7.4. Weighted with Board

For this approach we will take the **remaining available** *places* (b) and add *weights* to the movements of the *agent* and the *opponent*, prioritising our movements.

$$H(t) = (p \times 3) - (o \times 2) + (b \times 1) \tag{2.6}$$

2.7.5. Defensive to Offensive

The next step is to play **defensively** prioritising our *available movements* during the *first half* of the game, then play **offensively** trying to *exhaust* the possible moves for the *opponent*, for this we use the value of $\boldsymbol{m}$ previously seen in the Section 2.7.3.

$$H(t) = \begin{cases} (p \times 2) - o, & \text{if } m \leq 0.5 \\ p - (o \times 2), & \text{if } m > 0.5 \end{cases} \tag{2.7}$$

2.7.6. Offensive to Defensive

This approach is similar to the one in the Section 2.7.5, but **inversely**, in the *first half* we will play *offensively* and in the second *defensively*.

$$H(t) = \begin{cases} p - (o \times 2), & \text{if } m \leq 0.5 \\ (p \times 2) - o, & \text{if } m > 0.5 \end{cases} \tag{2.8}$$

2.7.7. Blocking the Opponent

Finally, this *heuristic* is completely **aggressive** and seeks to *hunt* the *opponent*. To carry it out we need to *create* a **list** of possible moves for our *agent* $\boldsymbol{(Np)}$ and the *opponent* $\boldsymbol{(No)}$, then create an *array* of those movements that are **equal** and use them in our *calculation*.

$$\begin{gathered} \boldsymbol{A}_{i,j} = \sum_{i=1}^{Na}\sum_{j=1}^{No} if\ (p_i == o_j),\ then\ p_i \\ H(t) = p - (o \times 2) + size(\boldsymbol{A}) \end{gathered} \tag{2.9}$$

3. RESULTS

The application was tested on a *MacBook Pro* (13-inch, 2017), with a *2.3* GHz Intel Core *i5* processor, 8 GB 2133 MHz LPDDR3 memory and *solid state hard drive*.

In the Figures 3.1, 3.2 we can see that the **highest** index of victories is against the opponent *Random*, which indicates good behaviour.

As the game progresses, the curve of victories *decreases*, which is to be expected because *opponents* propose a better *performance* with the use of the algorithms **minimax** and $\boldsymbol{\alpha - \beta}$, **pruning** being those that use the *heuristic* of **improved** the most *complicated*.

The Figure 3.3 presents the *overall percentage* of victories, where it can be appreciated that those who have different strategies at different moments of the game have a *better performance*.

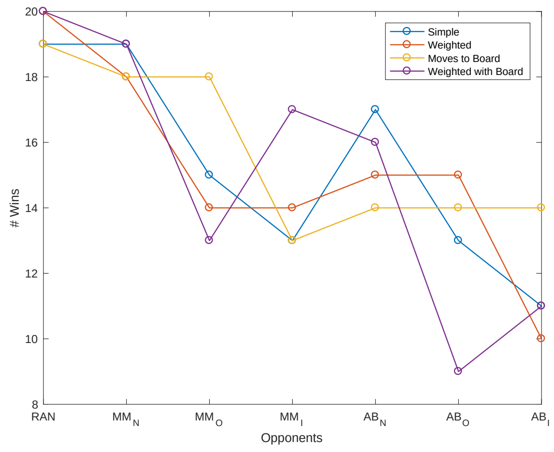


Fig. 3.1. Number of victories against opponents using the first 4 heuristics

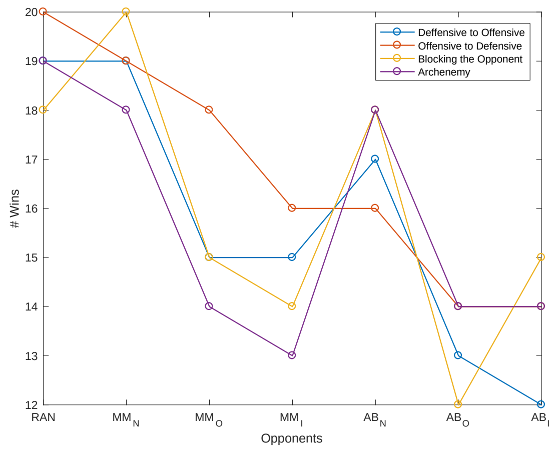


Fig. 3.2. Number of victories against opponents using the last 3 heuristics and the archenemy score

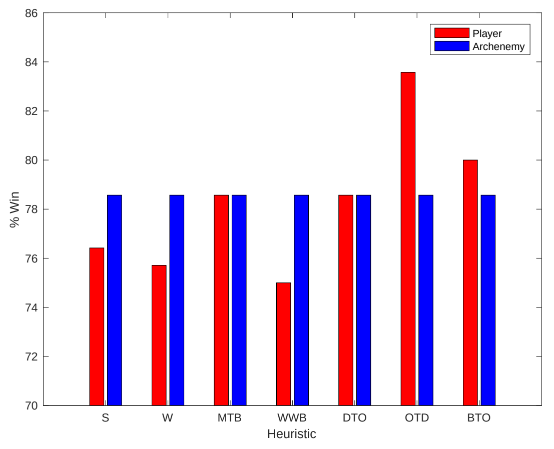


Fig. 3.3. Overall win percentage

4. CONCLUSIONS

The *heuristics* that present dynamic **weights**, this is that their value changes depending on the *state* of the game promote a more *regular* behaviour.

The use of *white spaces* in the evaluation function has the **lowest** performance, however, has the **best** score against the opponent **minimax** *improved*.

The use of **heuristics** with changes of *strategy* in different moments of the game or with *aggressive* behaviorus are more **efficient**.

$\alpha - \beta$ *pruning* with *iterative deepening* has a *better* performance than those that do not have it since its percentage of victories for all cases is *above* 50%, therefore the **Archenemy** has turned out to be an *extremely complicated* adversary.

5. LIMITATIONS AND WEAKNESSES

Although **Python** has become a *powerful* and relatively *simple* language to use, it still has the *disadvantage* of being an *interpreted* language, so implementation in a *compiled* language as **C++** would lead to faster performance.

Since we work with an external **API** for the management of the *board*, it keeps fixed *constants* for its *size*, preventing *testing* the *behaviour* of the **heuristics** in a way that *increases* the *size* of the *board*.

The focus on *adversarial games* is clearly *winning*, however the **time** needed to carry it out and the *number* of **nodes expanded** by the algorithms can be relevant, the *code* is made to extract these values however in this paper these *statistics* are not presented.

The **heuristics** that present *different behaviours* over time proved to be more *effective*, so the implementation of one that had 3 or 4 strategies at different moments of the game would be *valuable* to prove, however it was not done in this work.

REFERENCES

1. Udacity. (2011, Jan.) Search1-blind.pptx. [Online]. Available: https://eu.udacity.com
2. C. Oakman. (2013, Jan.) chessboard.js. [Online]. Available: http://chessboardjs.com/
3. Udacity. (2017, Apr.) Isolation. [Online]. Available: https://github.com/udacity/AIND-Isolation/tree/master/isolation
4. ——. (2017, Apr.) sample players. [Online]. Available: https://github.com/udacity/AIND-Isolation/blob/master/sample_players.py

6. APPENDIX 1 - CONSOLE OUTPUTS

```
Connected to pydev debugger (build 172.4343.14)
 |   |   |   |   |   | 2 |   |
 |   |   |   |   |   |   |   |
 |   |   |   | 1 |   |   |   |
 |   |   |   |   |   |   |   |
 |   |   |   |   |   |   |   |
 |   |   |   |   |   |   |   |
 |   |   |   |   |   |   |   |

True
[(0, 2), (0, 4), (1, 1), (1, 5), (3, 1), (3, 5), (4, 2),
(4, 4)]

Old state:
 |   |   |   |   |   | 2 |   |
 |   |   |   |   |   |   |   |
 |   |   |   | 1 |   |   |   |
 |   |   |   |   |   |   |   |
 |   |   |   |   |   |   |   |
 |   |   |   |   |   |   |   |
 |   |   |   |   |   |   |   |


New state:
 |   |   |   |   |   | 2 |   |
 |   | 1 |   |   |   |   |   |
 |   |   |   | - |   |   |   |
 |   |   |   |   |   |   |   |
 |   |   |   |   |   |   |   |
 |   |   |   |   |   |   |   |
 |   |   |   |   |   |   |   |


Winner: <__main__.RandomPlayer object at 0x10c7c3940>
Outcome: illegal move
 | - | - | - | 2 |   | - | - |
 | - | - | - | 1 | - | - |   |
 | - | - | - | - | - | - |   |
 | - | - | - | - | - | - | - |
 | - | - | - | - | - | - |   |
 | - |   | - | - | - | - |   |
 |   |   | - | - |   |   | - |

Move history:
[[4, 2], [2, 4], [3, 0], [4, 3], [1, 1], [2, 2], [3, 2],
[3, 4], [4, 0], [5, 3], [5, 2], [4, 5], [3, 3], [6, 6],
[1, 2], [5, 4], [0, 0], [3, 5], [2, 1], [1, 4], [0, 2],
[0, 6], [1, 0], [2, 5], [3, 1], [4, 4], [5, 0], [6, 3],
[6, 2], [5, 5], [4, 1], [3, 6], [2, 0], [1, 5], [0, 1],
[0, 3], [1, 3], [-1, -1]]



***************************************************
 Evaluating: Archenemy
***************************************************

Playing Matches:
---------------------------------------------------
  Match 1:   Archenemy    vs   Random     Result: 19 to
1
  Match 2:   Archenemy    vs   MM_Null    Result: 18 to
2
  Match 3:   Archenemy    vs   MM_Open    Result: 14 to
6
  Match 4:   Archenemy    vs MM_Improved  Result: 13 to
7
  Match 5:   Archenemy    vs   AB_Null    Result: 18 to
2
  Match 6:   Archenemy    vs   AB_Open    Result: 14 to
6
  Match 7:   Archenemy    vs AB_Improved  Result: 14 to
6


Results:
---------------------------------------------------
Archenemy         78.57%


***************************************************
   Evaluating: Player
***************************************************
```

```
Playing Matches:
----------------------------------------------------
  Match 1:   Player    vs   Random     Result: 19 to 1
  Match 2:   Player    vs   MM_Null    Result: 16 to 4
  Match 3:   Player    vs   MM_Open    Result: 17 to 3
  Match 4:   Player    vs MM_Improved  Result: 16 to 4
  Match 5:   Player    vs   AB_Null    Result: 18 to 2
  Match 6:   Player    vs   AB_Open    Result: 13 to 7
  Match 7:   Player    vs AB_Improved  Result: 15 to 5


Results:
----------------------------------------------------
Player              81.43%
```