COMP 424 – Artificial Intelligence

Project Final Report

Submitted to:

Prof. Jackie Cheung

Mr. Matt Grenander

Student: Yifan Bai

McGill ID: 260562421

April 11, 2019

Contents

Introduction 3

Problem Motivation 3

Technical Approach 3

Pros and Cons, Expected Failure Modes 5

Alternative Approach 5

Improvements in the Future 6

References 7

Introduction

This report documents a number of important aspects in implementing a Pentago-Swap board game AI agent. It comprises of a brief explanation to the algorithms and motivation to the problem, detailed technical implementation, an overview of pros and cons, as well as potential improvements. In addition, an earlier solution is also discussed in comparison with the final solution. Various concepts covered in class are used and adapted to this particular problem, and are reflected in both solutions.

Problem Motivation

A few game-play searching algorithms learned in class could be used. Since this is a competitive game between two players head-to-head, the basic intuition is that one tends to maximize the chance of himself winning and minimize that of the opponent. Conversely, from a defensive perspective, it is desirable to maximize obstacles to opponents, such as a move that limits the maximize blockage to the opponent. This hints at a search tree, and the basis of employing a Minimax algorithm to find the best moves.

However, several limitations present which requires reinforcing the algorithm. There is a timeout constraint to each move, as well as memory usage. This require a mechanism that limits the branches of search, whereby focusing on those that are more promising. Alpha-Beta Pruning comes into play, where moves to worse than previous is not evaluated further. When applying to a search tree, it always prunes branches that have no influence over the final result. [1] An alternative solution is using a Monte-Carlo Tree Search (MCTS). It offers great advantage in minimizing the search space. However, it tends to converge slower, which makes it less favourable in this project where computation limitations are key. [2]

In addition, naïve implementation of MCTS could lead to a search path where it could lead to a loss, and it is difficult to rectify when found at random. This is believed by some to be a potential explanation to AlphaGo's one loss against Lee.

Lastly, both Alpha-Beta Pruning and MCTS require tweaking evaluation functions and policies, which is the central part of optimizing them. MCTS's policies are much more complex to tune, and for the scope of this project, Alpha-Beta Pruning is used as its evaluation functions could be easily defined within the framework of gameplay rules. It is more reasonable to focus on improving scoring criterions of each move as well as different gameplay scenarios.

Technical Approach

The student player implementation has following source code hierarchy:

```
pentago_swap
|
|-- src
| |
| |-- student_player
|     |
|     |-- StudentPlayer.java
|     |
|     |-- MyTools.java
```

Note that no data file is used. A number of methods in each class are in charge of the evaluation functions. In `MyTools.java`, `getScore(int[][] board)` is used to evaluate the moves and assign a score to them. Since the key idea is Min/Max, as described earlier, if there are 5 inline, i.e. win, a score of ±100000000 is awarded to winner and loser respectively; if there are 4, the score awarded is 1/10 of that for the previous case. The pattern goes on until there is only 1 inline, which is most common at the early phase.

Another method implemented, `CheckPieceStatus(int p, List<String> list, int type)`, is used to check the board's state in terms of how the pieces are positioned. A few rules are defined, such as an all-white 5 inline would be scored as 11111, or 22222 for all-black. If there are 4 white pieces inline, and there are one empty spot at each end, it is represented as 011110. This is converted to string and stored. This method also feeds important information to `getScore(int[][] board)`, which evaluates and award points at each move.

The Alpha-Beta Pruning is implemented in method `alphabeta(PentagoBoardState tmp, int player, long endtime, int alpha , int beta, int depth)`. It initializes the board using a 6-6 2D array. It returns the score when it reaches Level 0, where a winner is apparent to be seen. Otherwise, it traverses through all possible moves and conducts calculation using the methods described above. This ensures that within the limit of computation, it returns a 'locally' optimum solution. The limiting factor is the timeout , which is controlled by `endtime`.

The StudentPlayer class inherits the PentagoPlayer class, which uses the methods defined in the tools to interface in gameplay. It contains a few global definitions, such as board size, numbering conventions for pieces (White = 1, Black = 2). At the beginning of the game, it initializes scores to infinity, and impose time constraints at each move.

Since the framework is defined, the work is now left mainly in tweaking the parameters and evaluation functions to find the balance between computation limits and search results, meaning having an efficient search results with reasonable search efforts are crucial.

Pros and Cons, Expected Failure Modes

There are a few of benefits and drawbacks of this algorithm. The major benefit is that Alpha-Beta Pruning allows a structured analysis of possible outcomes. Compared to simple Minimax, it cuts out certain nodes, which saves search effort. The problem then falls to defining an evaluation function for the game, which, with a rather simplified game, is a process

The main drawback is that the timeout constraint limited calculation of number of steps taken by opponent, i.e. the depth of the search tree. After trial and error, it was only able to compute 3 layers. When running with 4 playing against itself, the agent was stuck in the timeout constraint and unable to make moves. This results in automatic loss at the tournament, and as compromise, could not be adopted. This is due to the fact that the time complexity is $O(b^{0.75m})$ on average [3]. Even though it is an improvement over naïve Minimax, it is still exponential.

Additionally, the agent plays in a fix rule set with all search outcomes parametrized, and that they are not entirely based on the analytical rules, and that the cases of evaluation is hard-coded. If writing data files is permitted and establish learning mechanisms, the ceiling of agent's capability could be reached within several trainings and that rivals could exploit the limits of its algorithms. However, since this is not part of this project, it is not a point of concern.

Alternative Approach

There is another complete solution with a different approach. It was primarily served as a backup solution, during the time that I was struggling to meet the timeout constraint.

The solution uses Alpha-Beta Pruning in a Minimax setting, and calculates all legal moves in each round, assessing them and find the one that maximizes the potential gain. The evaluation function works using a reward-penalty system, same as the submitted version, albeit with a different award system: If there are 5 inline, the winning player gets 100 points and losing gets -100; if there are 4 inline, i.e. one step away from end of game and neither end is blocked, the points are ±5, and decrease linearly.

Due to the timeout constraint, I ranked the possible scenarios based on estimated likelihood. Taking the scores, if in the 'max' scenario, rank the moves based on score in descending order; 'min', rank in ascending order. Another enhancement is to use a method that checks whether if there is neighours around, since the opponent is rather far at early stages and victory is less likely. This prioritize those around and close by to narrow down search scope. To further control search efforts, only BL is considered for A-Swap, which halves the search efforts and proved to be more effective in simulations.

The main advantage of this alternative solution is that it prioritizes blocking all possible winning moves of the rival at each step, therefore minimizing its own possibility of losing. The main disadvantage is that since timeout and computation resource limits, the swap mechanism is largely simplified to reduce number of nodes to be searched, thus leaving it vulnerable for unforeseen scenarios.

In a simulation with the final submission version of 100 head-to-head games, the alternative solution was beaten in 98 occasions. Even in cases where the final submission did hit the 2-second time limit and being given a random move, the alternative solution was never able to win. Therefore it was eventually dropped. However, in cases where tighter turn timeout constraints are imposed, this method could prove useful.

Improvements in the Future

One way to improve the efficiency is to introduce ordering heuristics to search the tree where it is likely to force Alpha-Beta cut-offs. One implementation in chess is called ‘killer heuristic’, where the last move causing cut-off at the same level is always examined first. [4]

Other heuristics techniques could also come into play. It could be done by considering only a narrow search window. Know as aspiration search, it has an even more extreme version where the search is performed with alpha and beta equal; a technique known as zero-window search. [5] This is especially useful for win/loss searches in near-end scenarios, where the extra depth gained from the narrow window and a simple win/loss evaluation function could lead to a straightforward yet decisive result.

In addition, since the project permits ‘read’ from data files, an alternative could also be adding a text or JSON file that lists ‘hardcoded’ sets of eventualities, where the agent is able to ‘look up’ the possible moves. Since there are cases where the agent performs same analysis, it could simply read the text to determine the next step. It could save time when running the program, giving more resources to expand the search efforts. This was attempted when implementing project, only not to be realized due to numerous errors.

References

[1] Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Upper Saddle River, New Jersey: Pearson Education, Inc. p. 167. ISBN 0-13-604259-7.

[2] Bouzy, Bruno. "Old-fashioned Computer Go vs Monte-Carlo Go" (PDF). IEEE Symposium on Computational Intelligence and Games, April 1–5, 2007, Hilton Hawaiian Village, Honolulu, Hawaii.

[3] Pearl, Judea (1982). "The Solution for the Branching Factor of the Alpha-Beta Pruning Algorithm and Its Optimality". Communications of the ACM. 25 (8): 559–64. doi:10.1145/358589.358616.

[4] Huberman (Liskov), Barbara Jane (1968). "A program to play chess end games" (PDF). Stanford University Department of Computer Science, Technical Report CS 106, Stanford Artificial Intelligence Project Memo AI-65.

[5] Plaat, Aske; Jonathan Schaeffer; Wim Pijls; Arie de Bruin (November 1996). "Best-first Fixed-depth Minimax Algorithms". Artificial Intelligence. 87 (1–2): 255–293. doi:10.1016/0004-3702(95)00126-3.