COMP424 - Final Project Report

Pentago-swap strategy approach

Name: Yuzhou Guo
ID: 260715042

Introduction of the Project

Pentago-swap is a variant of the popular game called Pentago. Pentago is a classic *Moku* game where the board is six by six unit huge and separated to four quadrants, each with a size of three by three. The first player plays as white and the second player plays as black. The board begins empty and for each turn, players need to pick a move and twist one quadrant.

In our case with Pentago-swap, players need to process a move and instead of twisting, they actually need to choose two quadrants to swap, which rapidly increases the complexity of the game and the branching factor of certain algorithm chosen. Another challenge, which is also the main focus of the code implementation is the balance between aggressively winning from the agent's side and preventing its opponent from winning.

This report will be briefly talking about the strategies of winning the game and analyzing the pros and cons of each, the reason why we implement the algorithm we've chosen and possible improvements we can do in the future.

Technical Approach

As the main software skeleton and user interface are provided, the goal of this project will be finishing the chooseMove method and return a Move object which should be the best move possible according to the board situation.

Alpha-beta pruning is the algorithm has been implemented on the agent as it provides the capability to reduce the branching factor by the "pruning" process, which dramatically helps improve the runtime performance of the program *(Efficiency of alpha-beta pruning, L7-Games page 38)*.

Implementation wise, three additional classes are added which are the AlphaBeta class, Heuristic class, and the Blocking class. The Heuristic class and the Blocking class serve as the evaluation function which is part of the alpha-beta pruning algorithm.

A starting strategy is designed which is simply choose the four center positions to place if possible, meaning no blocking action needed to be performed. Other than this, the working flow is that the chooseMove method then directly calls the pruning method in the AlphaBeta class.

Inside the pruning method, the algorithm will go over all the legal moves by doing recursion and select the best move from children of the root. During this process, the alpha-beta pruning algorithm will be comparing the value returned by the “shouldIBlock” method from the Blocking class and the one returned by the “eva” method from the Heuristic class and choose the appropriate one according to the value. According to the priority order of doing blocking opponent and winning, the best move will be returned from the “chooseMove” method and processed on the board.

Motivation and Explanation in detail

Based on the naive approach of the minimax algorithm, the main progress of the alpha-beta pruning is that we “prune” some of the game options from the search tree by following the logic that both players in the game aim to win in a greedy way, ie. maximizing the score for the maximizer and minimizing the score for the minimizer.

Whenever the maximum score that the minimizing player (i.e. the "beta" player) is assured of becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta ≤ alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play *(Core Idea of alpha-beta pruning, https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning)*.

The heuristic function of the alpha-beta algorithm is separated into two functions which are the “shouldIBlock” function and the “eva” function. Both of two functions will be returning a certain value to the “pruning” function in the AlphaBeta class.

The “shouldIBlock” function evaluates the value according to the number of consecutive piece of the opponent, and the value return will be huger if the situation is more urgent. For instance, the value returned will be relatively huge compared to the value returned from the “eva” function if the opponent has been placed four pieces within a row/column/diagonal. Specifically, we only proceed to the blocking action if the opponent has more than or equal to three pieces in a row/column/diagonal so that the blocking function will be relatively light and quick to run.

On the other hand, the “eva” function will be focusing on picking the move eventually leads to the agent winning considering the opponent’s situation at the same time. There are four cases being discussed: the row, the column, the diagonal case from top left to bottom right and the diagonal case from top right to bottom left. We set a variable for each case calculating the evaluation value and another variable seeing how many consecutive pieces does the player have. A certain number to the power of the value of consecutive will be added to the evaluation value as following:

- If three consecutive pieces are detected
    - Add 20 to the power of three to the evaluation value
- If four consecutive pieces are detected
    - Add 100 to the power of four to the evaluation value
- If five consecutive pieces are detected
    - Immediately return the maximum integer since the agent, or its opponent will be winning in this board situation

Other than consecutive pieces, a small amount of value will also be added if any number of pieces from the agent/opponents is detected. After calculating and adding up all the values, we make a decision depending on who is the player in that particular case:

- Player using white piece
    - Return sum_white subtracting sum_black
- Player using black piece
    - Return sum_black subtracting sum_white

Back to the “pruning” method in the AlphaBeta class, the method will choose the larger one from the values from “shouldIBlock” function and “eva” function in the case of alpha (maximizer); and it will choose the smaller value in the case of beta (minimizer). Finally, the appropriate move will be generated and returned as an object.

Pros/Cons of Chosen Approach

In our case with the alpha-beta pruning algorithm:

Pros	**Cons**
Alpha-beta pruning itself as a subtype of a greedy algorithm. ● Assuming that both the maximizer and the minimizer are greedy (always want to maximize/minimize the value of each turn)	It is not predicted within the whole search tree if the opponent gives a ”trick move”. In that case, the agent implemented using alpha-beta pruning will be ended up pruning the cases may actually lead to win at the end

Reduces the branching factor and improves runtime performance overall. ● Reduction of search space/depth ● No time out Exception ● Each move decision can be made within one second	The branching factor is being reduced based on the foundation of assuming both players being greedy. ● If this is not the case, then the pruning process can be interpreted as “giving up situations the agent may win”
Controllable depth searching within the alpha-beta searching tree. ● Compare and contrast the relationship between move goodness and time spending being possible	Larger depth searching usually leads to longer time spending. If the depth value has been set to a relatively small value, then we may have a chance missing seeing the full situation of the board which leads to wrong decision fo move.

(Drawbacks of alpha-beta, L7-Games page 40)

Future Improvement

Other than alpha-beta pruning, Monte-Carlo tree search algorithm can also be implemented as the searching algorithm, which helps with solving some of the issues listed in the pros/cons table. Since there are random moves involved in the Monte-Carlo algorithm, it helps with not losing the possibilities to win in case the opponent of the agent does a not-so-greedy move. Also, the feature of the Monte-Carlo algorithm focusing on one path lets it have the property of good program performance from the perspective of time-consuming as well.

Language-wise, it can be a good improvement or controllable experiment to implement both the alpha-beta pruning and the Monte-Carlo in C++ rather than Java since the time performance can be even better if C pointers are being used in a smart way.