Introduction to Artificial Intelligence Final Project Report

Andrés Ponce (彭思安) 0616110

June 16, 2020

1 Introduction

The final project for the introduction to Artificial Intelligence course involved making an agent to play the classic game Reversi or Othello. The rules of the game traditionally have two opposing teams: the white and black team. According to the normal rules, a team can make a valid move only when a piece can be placed such that in between the two pieces there is a continuous line of opponent pieces. Then the opponent's pieces along that line are turned over to the current team's color.

In this project, given a board with a layout of pieces, our agent merely calculates the optimal cell in which to place the next piece. To do this, we can use one of several methods which were covered in class, among them: constraint satisfaction, logical satisfiability, and even deep learning. For this implementation, the approach taken was the classical constraint satisfiability along with a rudimentary $\alpha - \beta$ pruning.

2 Implementation

Python was used as the implementation. In the Python implementation, the function `Get_Step()` returns a tuple containing the coordinates to make the move. There are some things to consider when deciding where to place the piece: contrary to the original game rules, we can place a piece on any of the inner 6x6 board. Fortunately, we do not have to handle the conditions for the game termination, since that is handled at the server.

The most basic functions in the program are the `heuristic` and the `check_opponent_line()` function. The `check_opponent_line()` function will check the tiles in a given direction. If there is an unbroken line of opponent tiles along a given direction, the function will return the amount of tiles along that direction that we can turn to another color. There are three cases when the function returns:

1. Out of bounds
2. Encounter an unoccuped piece
3. Encounter an opponent piece

The `check_opponent_line()` function gets run constantly when running the `heuristic()` function. This new function will loop through the 8 tiles adjacent to the node parameter, and run the `check_opponent_line()` function on the adjacent tiles and keep a running sum. What this running sum indicates is the amount of nodes that we can turn over to the current color by placing a node. Thus, whenever we need a "gain" function, we can terminate the recursion by getting the heuristic value for a given node.

The main decision making function in the program lies in the `minmax()` function. This function is a standard implementation of the minmax algorithm learned in class. The main idea is to have two agents: the maximizing agent and the minimizing agent. Since there are two colored tiles playing the game, we can think of the maximizing agent as controlling the black tiles and the minimizing agent as controlling the white tiles.

The minmax algorithm in a sense "takes turns" between the maximizing and minimizing agent. We recurse when taking on the adjacent tiles to the current one being tested. When we recurse on the adjacent tiles, we decrease the depth parameter, and stop the recursion once the available depth reaches 0. This

switching between the white and black agents presents a more realistic simulation of the game playing algorithm, since players in a game usually have to face different constraints depending on the moves taken by their opponents. When a piece is upated on the board, the set of possible moves able to be taken by our agents changes and we thus have to keep account for it by thinking about how our opponent might make a decision. At every step of the iteration, we have to create a copy of the board, since we will modify it when simulating an agent placing a piece on the board. When doing forward checking, we have to check what the best move for our optimizing agent is given the choice done by the minimizing agent.

The next addition is the $\alpha - \beta$ pruning. The purpose of this pruning is to significantly reduce the number of children in our recursion tree that deserve to be checked in the regular way. By adding some simple if-statements in our algorithm, we can significantly reduce the runtime of our algorithm by realizing that some options can be discarded as non-optimal even before we check them. How does keeping an α and β parameter help us cut down on the required amount of nodes we need to check? β is the best amount that the minimizing agent currently posseses, while α indicates the best that we can do using the maximizing agent. Thus, if the best amount we can get from the minimizing agent is better than the current α we are testing, then we can discard the entire subtree under this α value since there is now way that it can be better than even the worst β.