Artificial Intelligence EDAP01

Report on Othello AI

Anton Palets

1 Introduction

This report consists of two parts: a user guide of sorts and an explanation on how certain parts of the program work. Of interest may be the evaluation function, the minimax, and other functions which describe the game world.

2 User Guide

There are two main ways a user can interact with the program. One is a 'user friendly' way, which asks for parameters in a nice way. The other provides more functionality, such as running multiple games at once, but the parameters are passed in a function.

To play the game simply run the program and then run `oth.play()`. This way the program will allow you to set the time limit per move (or depth) and who to play for in a nice way, i.e. by directly asking for a keyboard input.

If one were to want to test the program with, say, a random player, one can call `oth()`, which takes several optional parameters. Namely one can set the turn using `turn='b'` or `='w'`; depth using `depth=n`, where n is any reasonable integer; the total games to run through using `game_tot=n`; if one would like to have a visual representation of the board or not using the boolean `visual`; if one wants to input moves manually, one sets `human` to `True`, by default moves are made randomly; to set the maximum time allowed per move use `max_time` – it takes values in $(0, 10)$, if the value entered is more, the game defaults to depth 6.

3 Breakdown of the Program's Interesting Parts

3.1 General Aspects

A game state is represented as a tuple of lists – first of coordinates of black's tiles, second – of white's. The first coordinate is the row, the second is the column. 'Black tiles' and 'black coordinates' are used interchangeably. Within the minimax algorithm the game state is usually a tree node, also having a value attached to it.

3.2 The tree class

The `tree` class is quite important as it provides the data structure necessary to work with minimax. The tree stores a game state and a value. The game state is a tuple, and value is a float. The first element of the tuple can be accessed using `self.l`, right using `.r` and the value using `.val`. These are necessary to work with the minimax algorithm

3.3 MiniMax

The minimax algorithm is pretty much taken directly from Wikipedia's page on alpha-beta pruning (1). The pseudo code that can be found there was then adjusted to work with my tree class, which supports storage of a game state and a value. The algorithm will always return a tree node with the game state and appropriate (minimax-wise) value. The only case when this is not true is when returning on the initial call, where the function will return a tuple of black and white lists. Another main modification of the algorithm is that it knows how to treat the case where the current player doesn't have a legal move – that is the turn goes to the opponent, as by Othello rules.

3.4 The evl function

`evl` – meant to be read as evaluation, is precisely that – an evaluation function of a given game state. It assigns a value to a given game state, in a way which represents how desirable said game state is for a particular side. As someone who hasn't even heard of Othello before being presented with this assignment, I wasn't very comfortable with coming up with an evaluation function of my own. The way the function is now set up is essentially based on a StackExchange thread (2) I found on the topic. The value is based on the perceived positional and numerical advantage of a state. For the positional strength there's a matrix with values corresponding to game tiles. For the numerical advantage, in the early stages of the game it is seen as an advantage to have your opponent have more tiles. In the later stages this flips. All of these expressions have coefficients associated with them, and I basically meddled with them until the AI consistently won. To address winning and losing, the function will assign (positive or negative) infinity to a state if it is a terminal. To address the question of whether the given evaluation function is 'good enough', I had the AI play against a random player over 50 times, it lost only a few times, and the more moves ahead it looked the less it lost

3.5 The time2depth function

I didn't have any good ideas on how to implement a time limit for a move, so I opted for a way to estimate how many plies were reasonable for a given time. How long a move took on average for a given depth was computed. To be on the safe side of things, if say on depth 9, the average move took 10 seconds, and on depth 8 it was 1.8 seconds, I would assign depth 8 for any time input between 1.8 and 10 seconds.

3.6 The result function

The `result` function is essentially the transition model between moves for the game, and is thus a very basic building block. It takes as input a list of black coordinates, a list of white coordinates, the move coordinates and who is trying to make that move. The turn takes values either `1` or `2`, the black and white players accordingly. The function is essentially 8 if statements, checking for effects of a move in the 8 directions. Each, having fulfilled some reasonable requirements, checks if the opponents tiles are surrounded by our tiles, and if so moves them from the list of our opponent's coordinates into the list of our coordinates. The coordinate lists will then be compared to how they were before the move: if nothing has changed the function returns `False`, otherwise it appends the move to whoever's turn it was and returns a tuple of new coordinate lists.

3.7 The actions function

The actions function is there to provide a list of all legal moves for a given side and game state. It essentially goes through each tile on the board and runs `result` on it – if the function return a tuple, said tuple is then appended to a list of legal moves. This list is then returned by `actions`.

3.8 The term function

The purpose of the `term` function is to check a game state for being a terminal game state. The only input it takes is the game state tuple of coordinate lists. It outputs `True` if the state is a terminal state, and `False` otherwise.