Carl Mueller
MP2 Report
CS165A - Winter 2016

Introduction

My Reversi program uses 2D array of enum type to implement the board logic. The AI utilizes Negamax with Alpha Beta Pruning, shallow depth move ordering, and iterative deepening and uses dynamic board stability (updates static weights of board position depending on state of game) and mobility as its heuristic function.

To run: Inside MP2 directory:
$ make clean; $ make; $ ./Reversi $1 $2 $3

Architecture

My program consists of two packages: Agent and game. Agent contains the Agent abstract class from which all game players are extended. The game package consists of all classes that implement the infrastructure of the board and game.

For the purposes of making a simple Makefile, the Reversi class that contains main() is in the default package.

agent

The agent package consists of the abstract class agent from which three classes are currently extended: Human, RandomAgent, and NegamaxAgent. The Human class gets the moves from stdin, the RandomAgent randomly chooses a moves from the list of valid moves. NegamaxAgent implements the negamax search algorithm and the associated utility function.

game

The game package consists of the Board classes, Evaluator class, game class, move class, node class, and state class. It also contains the enum Cell (peice color and player color) and Direction (North, South, East, West, Northeast, Southeast, Northwest, Southwest).

- The Game class serves as a mediator between the Reversi class (which contains main()) and all other classes. It is composed of a Board objects and the required subclasses of Agent and the players. It handles basic logic for move ordering and generalizes the move ordering coming from main so that the Agent objects can be plugged in and out of Reversi class with relative ease.
- The Evaluator class is a utility class that contains a number of functions that perform operations of a 2D array of Cell enumerations and also includes the evaluation functions. Most of the game package classes are composed of a single evaluator object.
- The Board class implements the 2D array and is responsible for maintaining the state of the current board and printing the board. It initializes the board state and makes moves and calculates score using pass through functions to make calls to its evaluator object.

- The Move class is responsible for maintaining information about a given move. It contains the coordinates and a HashMap<Direction, Integer> that returns the number of flips as a value for any direction key provided.
- The State class is used to maintain the current board state and previous board state array as two Cell[][] and the most recent move. Objects of this class are passed along as the state for the next recursive call to the Negamax search algorithm
- The Node class is used to instantiate objects that are returned by the Negamax search algorithm. It is used so that the best move can be returned from the search algorithm rather than just the best calculated utility function.

Search

My program, via the NegamaxAgent subclass of Agent, implements the following in terms of the search function:

- Negamax algorithm (an implementation variation of minimax) with alpha beta pruning
- Shallow move ordering of depth 2 by running a separate the evalutation function iteratively on valid move calculations algorithm.
- Iterative deepening to counteract time constraint. The time limiter is adjusted according to board size.

My program uses the following heuristics:.

- Mobility: The evaluation function calculates the mobility (number of valid moves) for one player vs the mobility for another player from the perpsective of the max player.
- Corner Count: The number of corners a player has is weighted highly as they are the most stable position
- Corner Proximity: X-squares (diagonal adjacent) and C-squares (adjacent) around each corner is negatively weighted as these positions often lead to the loss of the corner.
- Wipe Out: To avoid early termination, any moves that may ultimately lead to a loss by piece count prior to the end of the game via no more valid moves is weighted very negatively. This is most useful for large board sizes against players that weight the actual score highly (a la reversi.jar).
- Score differential: The score differential between players is slightly weighted and tends to affect the later game more as the differential sizes grow large.

My program attempted optimization by using HashTables to calculate valid moves for the 2D array. However, copying boards and all the various searching computations makes my program tremendously slow and inefficient. The design principals are there but there execution and implementation is inefficient.

Challenges

The biggest challenge I faced was implementing the Negamax algorithm and making sure to return a move associated with the best calculated score. Additionally, I have a hard time truly measuring and optimizing my heuristic functions since my program takes a long time to execute at each step. Additionally, I failed to recognize that the evaluation function must be calculated from the perspective of the MaxPlayer at all times given the structure of my NegaMax algorithm.

Weakness

The major weakness in my implementation is too much coupling and too much coding. Again I am not a CS major by training and so object oriented design patterns are still unfamiliar. My computation complexity must be quite high, regardless of the efficiency gains of alpha beta pruning. For boardsize 10 I average around a depth of 7-8 with a 6 second limiter. Sometimes if it jumps a larger depth the actual calculation time is 20 seconds. If a let it get there consistently sometimes it jumps above 30 seconds.