CSC 412: Intro to AI

Project 1: Minimax search and alpha-beta pruning

The purpose of this project is to implement some of the adversarial search techniques for two-player alternate move games, that you have learned. In particular, you will implement minimax search with alpha-beta pruning, and (optionally, for extra credit) a game board evaluator (i.e., heuristic evaluation) for Connect-4. Your goals are to implement alpha-beta pruning correctly, and to (optionally, for extra credit) create the best AI player you can for the Connect-4 game.

The basecode in java is available at http://www.cs.usm.edu/~banerjee/CSC412/Proj1_code.zip

This codebase has been developed for General Game Playing (see http://games.stanford.edu/), where an arbitrary game (described in the GDL language) has to be played without exploiting the benefit of game-specific knowledge. However, for the purpose of this project, we will focus on just a few, small, 2-player alternate move games, such as Tic-tac-toe and Connect-4. The directories and files that will be of specific interest to you are

- games/*.kif: These are the GDL descriptions of games (kif=knowledge interchange format) that can be played. If you want the player-programs to play a new game, you will need to generate a GDL description of that game. Again, for this project, focus on games/ttt.kif (Tic-tac-toe) and games/connect4.kif, although you will not need to understand these formats.
- ggp/players/*.java: These are the various player programs, including HumanPlayer.java and RandomPlayer.java, that give you an idea of how you can go about developing an AlphaBetaPlayer.java. The AlphaBetaPlayer class that you develop will need the following parameters for the constructor:
  ○ GameDescription gd,
  ○ Simulator sim,
  ○ int role
  ○ (optionally) Heuristic h

The GameDescription and Simulator classes are pre-defined. These parameters allow access to various utilities for the AlphaBetaPlayer to figure out the condition of the game. Every player needs to implement 3 specific methods:
  ○ public boolean processReward( List<Integer> rewards ): For the AlphaBetaPlayer, this can simply return true, and do nothing else
  ○ public int act( Set<Integer> state, List<SortedSet<Integer>> legalMoves ): This is the most important player function, that returns the chosen action/move, given a state (e.g., board configuration). This move will have to be generated by minimax search with alpha-beta pruning. Unless you are trying the extra credit, you will not need to understand the state description. You will only need to utilize the following functions of state:
    ■ List<SortedSet<Integer>> legalMoves = sim.computeLegalMoves( state ); This will return the legal moves of all players in the current state/board. For turn-taking games, this means the player whose turn it is, can retrieve the set of his own moves by calling SortedSet<Integer> moves = legalMoves.get( role ); //where 'role' is his role in the game. For the other player, “moves” will only contain “Noop”, since it is not his turn.


Set<Integer> nextState = sim.computeNextState(state, moves); This will return the next state resulting from applying the moves of all the players to the current state/board. The parameter “moves” should be a List<Integer> (size 2 if only two players) which contains the specific moves made by the players in the given state (including the Noop by the player that is out-of-turn). You can use the combination of the computeLegalMoves function and the computeNextState function to emulate the SUCCESSORS function in the book.

Boolean sim.isTerminal(state); Returns true if the current state is a terminal state. You can use this as the TERMINAL-TEST function in the book.

List<Integer> goals = sim.computeGoals(state); This function returns the rewards/scores due to all the players (the list is indexed by their roles), once a terminal state is reached. You can use this as the UTILITY function in the book. But you will have to call goals.get(role) to retrieve the reward of the individual player in “role”.

public void actTimed(Set<Integer> state, List<Integer> lastMoves, GamePlayer gp, PoliteThread thread); Leave this function empty for your AlphaBetaPlayer class, since it will not be tested in a timed setting.

The parameter “GameDescription gd” provides access to various utilities such as

- gd.getLowestReward(role) and gd.getHighestReward(role) give you the minimum (losing) and the maximum (winning) scores/rewards to a player in “role” in the game. Use these in place of -infinity and +infinity in the alpha-beta search.
- gd.getMoveTerm(move): Gives you the English interpretation of the (integer) move made by a player, e.g., “MARK (1,3)” in Tic-tac-toe. You may not need to access this in this project, though.
- gd.getNumRoles() gives you the number of distinct roles (i.e. number of players) in the game.

- ggp/heuristics/*.java: Various general-purpose heuristics have been defined in this directory. You may ignore this directory completely if you are not going to attempt the extra-credit. If you do attempt the extra-credit, you will need to write a new heuristic class (for Connect-4 only) in this directory. Note the following:
  - Any heuristic class must define the function public double getValue(Set<Integer> state): You can treat this function as the equivalent of the EVAL function in the book.
  - You will need to construct your Connect-4 heuristic with GameDescription gd as a parameter. Then you may want to call List<KIFTerm> terms = gd.getStateTermsFor(state) to understand how the pieces are placed in the Connect-4 board. **You will need to research the codebase further to figure out how these work, and the other bits and pieces that you may want to use.**

- ggp/OfflineSim.java: This is where you will find the “main” function. This simulates the game manager off-line (i.e., not in the competition setting). You will see that two player objects are defined:
  - players[0] is defined to be a RandomPlayer(0), where 0 is the role. This player makes a random move in every state, so you should be able to beat it easily. When you define your AlphaBetaPlayer class, use (i.e., uncomment) the alternative definition
    players[0] = new AlphaBetaPlayer(gd,sim,0) instead. Again, 0 is the role.
  - players[1] is defined as the HumanPlayer(gd,1), where 1 is its role. This allows you to play against the RandomPlayer initially, to check out how everything works.
How to run?

After you have run “make”, make sure that ggp.jar is in the top directory, and then run “./demo ttt 1 1” at this directory. This will let you play the game of Tic-tac-toe against a RandomPlayer (ignore the two 1's at the end of the command, for this project). There is also a visualizer built-in (outputs in html) for various games in the directory games/.

Important notes

- You must use a transposition table (page 170-171 in textbook) to save every state-value that the search comes across (either through UTILITY calculations on terminal states, or alpha/beta values when pruning), so that it does not have to repeat these expensive recursive calculations for the same states in future calls.
- Assume that your program will only be run on 2-player, alternate move games only; so if it crashes on other kinds of games in the games/ directory, do not worry.
- Point distribution
  - 50 pts: Correct implementation of minimax search
  - 50 pts: Correct implementation of alpha-beta pruning
  - (Extra credit) 30-50 pts: Connect-4 evaluation function that significantly improves the speed of the computer player's moves, and beats me repeatedly.