CSE 332: Data Structures and Parallelism

P3: Chess

Checkpoint 1: Fri, August 3rd, in class
Checkpoint 2: Wed, August 8th, in class
P3 Due Date: Mon, August 13th, 11:30 PM

The purpose of this project is to compare sequential and parallel algorithms on some intractable problems. You will also learn some new graph algorithms and a bit of combinatorial game theory.

Overview

In this project, you will write several chess bots and analyze their performance. You will implement several (graph/tree) algorithms (both sequential and parallel) and be able to see a significant difference in the quality of the bots.

Before attempting this project, you should read the handout on the algorithms! (games.pdf)

The project is designed so that you need minimal chess knowledge, but we recommend you familiarize yourself with the basic rules just in case. We have written all of the chess-specific code (evaluator, move generation, board, GUI, etc.); all you will be responsible for is implementing the game tree searching algorithms. You may, of course, improve the board/evaluator/etc. to your liking.

The parts of this project alternate between sequential code and parallel code. You will begin by writing a sequential mini-max implementation; then, you will write an implementation in parallel, using what you've learned about parallelism in this course. Finally, you will write a sequential implementation of the Alpha-Beta Pruning algorithm discussed in class. **This quarter, you will not write a parallel implementation of Alpha-Beta**. As cool as the algorithm is, we simply do not have enough time to ask you to do it. You may ignore any part of the handouts or code you read that explicitly refers to JamboreeSearcher (i.e. Parallel Alpha-Beta). In particular, this means you may fail tests on gitlab which reference JamboreeSearcher (without it affecting your grade). You may also ignore any references to a Chess Server, as we will not be asking you to attempt to run your bots on it.

Now that you have implemented (essentially) every core data structure, you are free to import and use data structures from java.util. You can also continue to use your own implementations. :)

Project Restrictions

- You *must* work in a group of two unless Robbie has already talked to you.
- You may not edit any file in the `cse332.*` packages.
- The *design and architecture* of your code are a *substantial* part of your grade.
- The Write-Up is a *substantial* part of your grade; do **not** leave it to the last minute.
- **DO NOT MIX** any of your experiment or above and beyond files with the normal code. Before changing your code for experiments or above and beyond, copy the relevant code into the corresponding package (e.g., `aboveandbeyond`, `experiments`). If your code does not compile because you did not follow these instructions, you will receive a 0 for all automated tests.

Provided Code

- `cse332.*`: You shouldn't need to look at any of the files in this package. The code in these packages sets up a connection to the chess server and communicates with the chess server (which we won't be using this quarter) and sets up several interfaces. You shouldn't need to understand any of this code to complete the project.

- `chess.board`: You also shouldn't need to look at any of these files. In chess, the *game position* consists of the board and some auxiliary information that these classes keep track of. We list below the only relevant methods you need to be aware of from the `ArrayBoard` class:

`List<Move>` **`generateMoves`**`()`
Generates a list of valid moves that the current player could make.

`void` **`applyMove`**`(Move move)`
Applies the provided move to the board changing the state of the game.

`void` **`undoMove`**`()`
Undoes the last move applied to the board.

`ArrayBoard` **`copy`**`()`
Copies the board Object in its entirety. This operation is *expensive*; you should avoid using it whenever possible.

- `chess.game`: This package contains classes related to playing a game of chess. It includes our provided evaluator (which you may edit) and a timer class which might be useful if you want to stop your bot after a certain amount of time. We list the methods from these classes that you might need below.

  - `SimpleEvaluator.java`:

`int` **`infty`**`()`
Returns a number larger than any actual board evaluation to represent infinity.

`int` **`mate`**`()`
Returns the value of a board in checkmate. (Depending on the current player, this could either be very high or very low.)

`int` **`stalemate`**`()`
Returns the value of a board in a stalemate (a draw).

`int` **`eval`**`(Board board)`
Returns a number representing "how good" the provided board is. Note that the `Board` class maintains information about the current player (white or black), and `eval` will return a value *from the perspective of the current player*.

  - `SimpleTimer.java`: This class gives you a way to allow `generateMove` (the method that finds the next best move) to be time limited. You do not have to use it, but if you do, feel free to add/change methods to your liking.

- `chess.setup`, `chess.play`: These packages contain classes related to a chess server. We will not ask that you attempt to connect to the server for this assignment, so you may ignore the classes in here.
- `chess.bots`
  - `BestMove.java`: This class will be useful when writing your bots, because you'll need to return both a move and its value.
  - `LazySearcher.java`: This is a very dumb searching implementation that returns the first move it finds. It's intended to show you what a working bot looks like.

A Warning

All of the parts of this project involve understanding exactly how the previous parts worked. It would be a *giant* mistake to split up the work by having one groupmate do half of the parts and the other one do the rest.

Part 1: Minimax and Parallel Minimax

In this phase, you will write two `Searcher`s: `SimpleSearcher` and `ParallelSearcher`.

We *strongly recommend* that you look at `LazySearcher` before you begin to see what the structure of the bot should look like. In particular, you should `extend AbstractSearcher` and use the instance variable `ply`. If you want to use a timer, you should use the instance variable `timer`.

[**NOTE**: If you have not read the games handout yet, do so now! The algorithms described in the games handout are only partial pseudocode, they are not complete algorithms. They are meant to get you started, but you will need to think more deeply about the algorithms described there before implementing them yourself.]

(1) `SimpleSearcher`: Implementing Minimax

`SimpleSearcher` should implement the Minimax algorithm as described in the games handout. This first version should have no parallelism. While you may use a `bestMove` global variable that keeps track of the "best move so far", we recommend that you return a `BestMove` object from your `minimax` method instead.

The pseudocode in the games handout does not handle the case where there are no moves quite correctly. You should handle it as follows:

```
if (moves.isEmpty()) {
   if (board.inCheck()) {
      return −evaluator.mate() − depth;
   } else {
      return −evaluator.stalemate();
   }
}
```

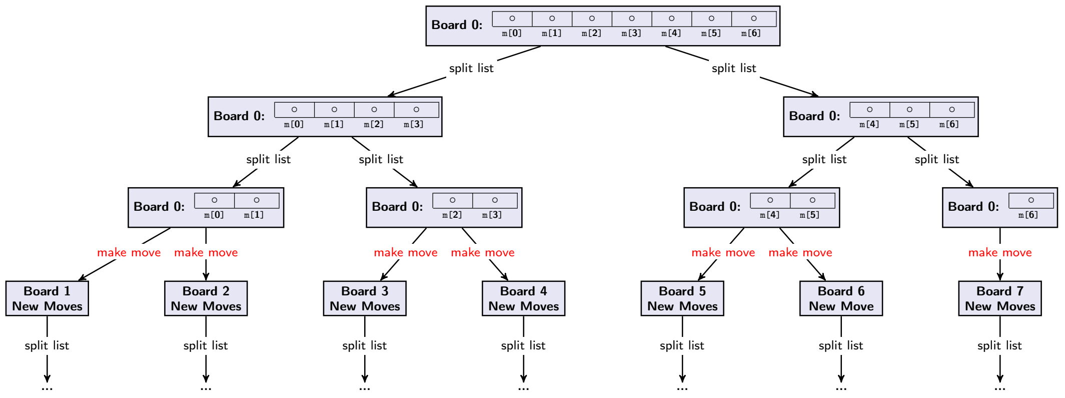
In other words, if there are no moves, it's either a stalemate or a mate. Mate is either very bad or very good, but it depends slightly on how many moves away it is.

Additionally, you may notice a call to `reportNewBestMove` in `LazySearcher`; this method is responsible for updating the "current best move" on screen, and, so, you may use it as (in)frequently as you like.

(2) `ParallelSearcher`: Implementing Parallel Minimax

`ParallelSearcher` should implement the parallel minimax algorithm as described in the games handout. This version should be parallel. This version should be able to get further down the game tree than just regular minimax. Make sure you do all the standard parallelism things: divide-and-conquer, sequential cutoff, etc. Make sure you use the `cutoff` instance variable defined in `AbstractSearcher` rather than creating your own (for the sequential cutoff); the reason is later, when you want to configure all the variables, there is a `setCutoff` method you can use to quickly edit the cutoff.

Note that you are doing parallelism *on a graph* here. In particular, you absolutely should not treat the children as a linked list by forking each thread in a loop, because that wouldn't be divide-and-conquer. Imagine that you have a `SearchTask` class that extends `RecursiveTask<BestMove<M>>`, your recursive calls should look like the following:



Once your divide-and-conquer tree gets to a certain depth with respect to `cutoff`, you should switch to a sequential algorithm (namely, `minimax`). Furthermore, as above, you will be doing a divide-and-conquer algorithm; so, there will be a second cutoff, `divideCutoff`, which will tell the algorithm when to stop dividing nodes.
This bears repeating: **your code will have TWO cutoffs in it**:

- `cutoff` tells the algorithm when the number of plies remaining is small enough that the rest should be executed sequentially (use the existing super class field for this).
- `divideCutoff` tells the algorithm when to stop forking children via divide-and-conquer and instead fork them sequentially (note: this does NOT mean to execute them sequentially). (Make your own constant for this).

Note that creating an instance of a class (e.g., `SimpleSearcher`) to run your sequential algorithm would work, but it would be prohibitively slow. Instead, note that your `minimax` method in `SimpleSearcher` is `static`; so, you can call it without instantiating a new instance every recursive call.

Part 2: Alpha-Beta

In this phase, you will write a substantially more interesting `Searcher`: `AlphaBetaSearcher`. This `Searcher` should extend `AbstractSearcher` and use the `ply` and `cutoff` variables like in the previous part. Debugging this implementation will be substantially more time consuming than the previous ones.

(3) `AlphaBetaSearcher`: Implementing Alpha-Beta Pruning

When starting to implement this `Searcher`, it will help to copy over your `SimpleSearcher` code and edit it directly. The hardest part of this particular implementation is understanding exactly how the algorithm works. We recommend you look very carefully at the diagrams in the games handout. Feel free to look up other explanations of the algorithm on the internet or in Weiss 10.5.2 (p. 495).

Part 3A: Using Your Algorithms

In this part, you will apply your code to a completely different problem.

(4) Analyzing Traffic

We've discussed multiple variations of the shortest paths problem in lecture. Now, we'll introduce one more and use the algorithms you've already written to solve it! Consider the *best* path from A to B that factors in *speed limits* and *traffic*. There are multiple possible things you might optimize for in this problem. For example, you might just want the absolute shortest path. Or maybe, you really don't like traffic and you don't mind the ride taking a few extra minutes if you can avoid more traffic. It turns out that this problem can be phrased as a two-player game and we can use Minimax/Alphabeta to solve it. The sublety here is that the other player is *nature*; there is a "worst" move they can throw at you, but, in general, the "nature" player just does some move. So, the *moves* in this game look as follows:

- On your turn, you choose a direction to turn (or continue straight).
- On nature's turn, they reveal how much traffic you're running into.

The game ends when you get to your target location. Clearly, some amount of time will have elapsed during your trip. The normal shortest path problem would attempt to minimize that time. In our version, our evaluation function works as follows:

- You choose a threshold (number of seconds) that is "acceptable" for the trip.
- Any dead-end has a very negative value.
- Reaching your destination after that threshold has a very negative value.
- In any other situation, the evaluation function *attempts to minimize the number of seconds in traffic* during your trip.

To facilitate running your code on this problem, we have created a map of downtown Seattle, complete with real speed limits and traffic data. Look in the traffic folder in your repository for some new classes that solve most of this problem. The only missing piece is the searcher!
You will need to *very slightly* modify one of your searchers to make it work for the traffic problem. The only change you will need to make is to the base case when moves is empty; since stalemate and checkmate don't make sense in this context, replace that part of the code with a call to `eval` on the board. You should modify this file in the `traffic` package.
Congratulations! By editing two lines of code, you solved a completely different problem!

Part 3B: Write-Up

(5) Write Up

In the repository, you will find a file `WriteUp.md`. A large portion of your grade is filling out answers to the questions in this write-up. Make sure to fill it out!

Project Checkpoints

This time, this project will have **two** checkpoints (and the final due date). As long as you have made a good-faith effort to complete the checkpoint, the checkpoint will not directly affect your grade. However, if you find yourself falling behind the checkpoints, it is likely you will be unable to finish the project on time. You have been warned.

Checkpoint 1: (1), (2)	**Fri, August 3rd, in class**
Checkpoint 2: (3)	**Wed, August 8th, in class**
P3 Due Date: (4), (5)	**Mon, August 13th, 11:30 PM**