Project 3: Chess

Checkpoint 1: Mon, August 12 at 11:59 PM (online survey)
Final Turn-in: Mon, August 19 at 11:59 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.

1 Overview

In this project, you will write two chess bots and analyze their performance. You will implement two versions of a (graph/tree) algorithm (both sequential and parallel) and be able to see a significant difference in the quality of the bots using one method over the other.

Before attempting this project, you should read the handout on the minimax algorithm! (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. **This quarter, you will not be required to write a serial or 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 Alpha-Beta or JamboreeSearcher (i.e. Parallel Alpha-Beta). In particular, this means you may fail tests on gitlab which reference Alpha-Beta or 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 you have already talked to Rob or worked on P2 alone.
- 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 files 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.

2 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.

3 Project Checkpoints

This project will have **one** checkpoint (and a final due date). A *checkpoint* is a check-in on a certain date to make sure you are making reasonable progress on the project. For the checkpoint, you (and your partner) will each fill out a survey asking about your progress and experience so far with the project. At the same time, you will also have the opportunity to sign up for an (optional) meeting with a class staff member to seek additional help or guidance.

As long as you fill out the checkpoint survey, and have made a good-faith effort to complete the relevant part of the project, you will get full credit for the checkpoint.

Checkpoint: (1), (2)	Mon, August 12 at 11:59 PM (online survey)
P3 Due Date: (3), (4)	Mon, August 19 at 11:59 PM

Project Restrictions

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

4 Minimax and Parallel Minimax

In this phase, you will write two Searchers: `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

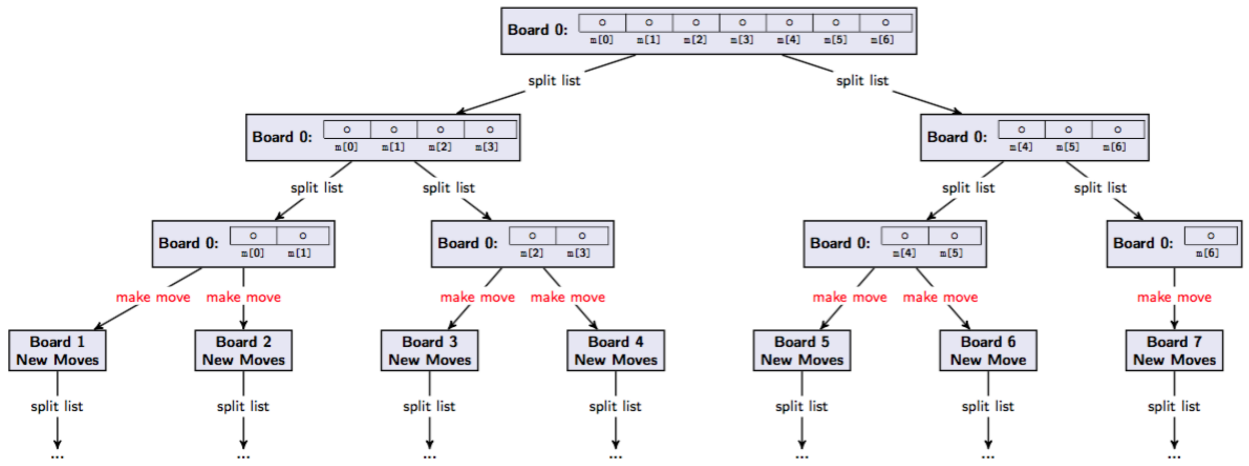
`SimpleSearcher` should implement the Minimax algorithm as described in the games handout. This first version should have no parallelism. While you may use a `bestMove` global variable that keeps track of the best move so far, we recommend that you return a `BestMove` object from your `minimax` method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

5 Using Your Algorithms

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

(3) 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 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 (you can copy code from `ParallelSearcher`). 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!

(4) Write-Up

Approximately half of your grade will be based on your write-up. The analysis part of this project is very important, and we expect you to spend an appropriate amount of time on it.

Like past projects, write-up files will be submitted via GradeScope.

6 Above and Beyond

DO NOT MIX any of your above and beyond files with the normal code. Before changing your code for above and beyond, copy the relevant files into the `aboveandbeyond` package. If your code does not compile because you did not follow these instructions, you will receive a 0 for all automated tests.

- Playing a Game: Consider playing an actual game of chess with your bot. First, create a new evaluation function that you think will make your bot smarter. Justify what changes you made and why you think they will improve the bot's performance over the given eval function. These should be non-trivial changes from the provided eval function. Then download the LocalBvB.java file on the assignments page and use it to pit your bot against another group's. Report some stages of the game with commentary as well as the final outcome in your write-up.
- Serial and Parallel Alpha-Beta: We don't have time in the summer to require you to implement the alpha-beta optimization of minimax, but if you want to go that extra mile consider adding it anyway. We recommend implementing the serial version of alpha-beta first to get used to it, but you will need to make a working version of parallel alpha-beta (called Jamboree in this project) to get any extra credit. The rest of this document gives as much info about this part of the project as is normally offered.

Alpha-Beta and Jamboree

In this phase you will write two substantially more interesting Searchers: AlphaBetaSearcher and JamboreeSearcher. These Searchers should extend AbstractSearcher and use the ply and cutoff variables like in the previous part. Debugging these implementations will be substantially more time consuming than the previous ones.

`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).

`JamboreeSearcher`: Implementing Parallel Alpha-Beta Pruning

This searcher combines ideas from all the previous ones. It is parallel in a similar way to ParallelSearcher, but it uses Alpha-Beta Pruning as a base, sequential algorithm. You should probably start by copying in the alphabeta code from the previous implementation.
For better or worse, combining the complicated algorithm with the complicated parallelism leads to a host of new concerns/issues. We list things you will almost certainly run into here for your convenience:

- As with ParallelSearcher, you will need to use divide-and-conquer which means looping through the nodes when you are supposed to be parallelizing is not acceptable.
- In the sequential algorithms, you never had to copy the board, because only one thread needed it. In ParallelSearcher, you always needed to copy the board, because every thread needed one. Here, you get a weird in-between. You will often need to copy the board, but you should always do it in the child, rather than the parent. The reasoning is that if you copy in a parent thread, all of its children are waiting on the copy, but if you do it in the child, all the children can get started earlier. Note that you should avoid copying as much as possible, because it is the most expensive piece of the algorithm.
- All your recursive calls should be to jamboree - not minimax, not parallelMinimax - in particular, it's tempting to make the parallel part of the recursion be a call to your

ParallelSearcher and the ”sequential part” a call to your minimax. This will lead to significantly worse performance and is not the jamboree algorithm.

- A good implementation of this algorithm will get to depth 6.