CSE 332: Data Structures and Parallelism

P3: Chess

Checkpoint 1: Tue, Feb 20
Checkpoint 2: Tue, Feb 27
P3 Due Date: Wed, Mar 07

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 compete against other chess bots on the CSE 332 chess server. 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. Unlike in previous projects, **you should feel free to use any and all Java data structures** (since you've now implemented them all yourself).

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. For each new algorithm, you will implement the sequential version first followed by a parallel version.

Project Restrictions

- You *must* work in a group of two unless you successfully petition to work by yourself.
- 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 GUI, a connection to the chess server, communicates with the chess server, 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 that are 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 you might need for these classes 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`:
  - `Engine.java`: This class sets up the bot that will be used with the `EasyChess` client. You can change any and all of the configuration parameters. In general, you will need to change the class of the bot, the depth, and the sequential cutoff.
- `chess.play`: This package contains classes that allow you to connect to the CSE 332 Chess Server.
  - `EasyChess.java`: This is the main client you will use to play chess using your bot on the chess server. All of the setup uses the GUI. So, just run it as a "Java Application" in Eclipse, and you're good to go.

- `CloudClient.java`: This client is for running your bot in the cloud. It automatically starts a game with the provided bot when it runs. To watch the game, log on using `EasyChess` on your computer and use the "`watch`" command.

- `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. In substance, this is a lot like the `DataCount` object from p2.
  - `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.

The CSE332 Chess Server

For this project, you will eventually need to compete with at least one of the bots. To do this, you will connect to the CSE 332 Chess Server using the `EasyChess` client. We recommend playing games with your bots on the server relatively frequently, because it is a fun and interesting way to debug your code. Additionally, you can use the server to compete with other students' bots.

Commands

Command	Description
`help`	Displays a help message with all of these commands listed.
`match <name>`	Challenges `name` to a match. If `name` is one of our bots, the match will start immediately. Otherwise, the server will wait for an `accept` command.
`accept #<game>`	Accepts a challenge from another player. Once you accept, the game will start immediately.
`watch #<game>`	Allows you to watch a game currently being played. To get a list of valid games, use the `games` command.
`who`	Lists all the players currently on the chess server.
`games`	Lists all of the games currently being played
`scores <bot>`	Lists the results of the last ten games with `bot`.

The Bots

Bot Name	Description
`calculon`	You should be able to beat this bot with your `AlphaBetaSearcher`.
`clamps`	You should be able to beat this bot with a well-tuned `JamboreeSearcher`.
`flexo`	This bot is more difficult to beat than `clamps`, and you will need to go above and beyond to beat it.
`bender`	This bot is substantially more difficult to beat than `flexo`, and you will need to go significantly above and beyond to beat it.

Playing On The Server

To have your bot play on the server, edit `Engine.java` until you're satisfied and then run `EasyChess`. You will need to log in with your teamname and password you received in your original project confirmation email from `blank@cs` referencing your gitlab repo. Your account may log in more than once, but the server will start adding numbers (e.g., `husky`, `husky1`, etc.). Additionally, your account may only play one game at a time.

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 `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:

```
1 if (moves.isEmpty()) {
2    if (board.inCheck()) {
3       return -evaluator.mate() - depth;
4    } else {
5       return -evaluator.stalemate();
6    }
7 }
```

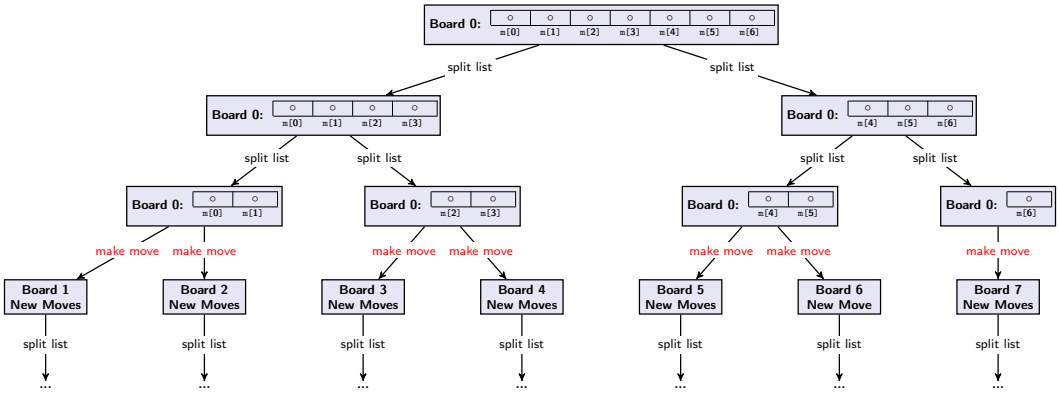
In other words, if there are no moves, it's either a stalemate or a mate. Mate is either very bad or very good, but it depends slightly on how many moves away it is.

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 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.

(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).

(4) `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.

Part 3: Using Your Algorithms

In this part, you will attempt to beat at least one bot on the CSE 332 Chess Server and apply your code to a completely different problem. **Please refer to "Playing On The Server" above.** In particular, remember that you need to edit `Engine.java`.

(5) Beating Clamps

We have designed `clamps` so that you should be able to beat him with the code you've written for this project. **Ten percent** of your grade on this project will be based on beating (or drawing)against `clamps` a sufficient number of times. We will take the *very last ten games you play*, count a loss as 0, a draw as 0.5, and a win as 1; your score on this part of the project will be the sum of these scores (capped at a max of 8.0) divided by eight. **Notice that if you play 30 games against `clamps`, only the last ten you run will count.** (Note: even though we will sum your wins and draws from the last ten games you played, it will not be possible to score above 8.0/8.0 .)

If you are having problems beating `clamps`, we recommend first checking your code using the `gitlab-ci` tests. If it's working, then try to make the `ply` parameter higher. If you time out before you start winning, try running your bot in the cloud using a 32 core machine (see below for instructions). You may wish to take a look at the Write-Up at this point, in particular, the section on "Optimizing Experiments". As tweaking various parameters may help you beat clamps.

Depending on your implementation of `JamboreeSearcher`, it might not be good enough to consistently beat `clamps`; in the event that this happens, you should *make a new class* and implement any of the following:

- Use "Iterative Deepening". Alpha-beta is effectively a suped-up DFS over the game graph. Because move ordering is important (as discussed below), it is often worth it to first try 1-ply, then start over and try 2-ply, etc. Keep in mind that the bulk of the graph is always at the later plys; so, this strategy doesn't redo a ton of work. Furthermore, you can use your "current guess" from the lower plys to maybe avoid large chunks of the later ones.
- Use "Move Ordering". Alpha-beta and Jamboree are very sensitive to the order that you actually look at the moves in. There are several heuristics (heuristics, because they don't always work) that often result in better ordering. Key words to look up include: "killer move heuristic", "MVV-LVA", "History Heuristic" and others: https://chessprogramming.wikispaces.com/Move+Ordering
- Use a "Transposition Table". As you explore the chess graph, you will run into the same positions many times. We solved this problem in DFS by keeping track of the nodes we'd already seen. A "transposition table" is a fancy data structure for games that does exactly this. The idea is that the number of nodes you are visiting is very large; so, instead of keeping track of everything, we keep track of the most recent nodes we've seen in a hash table. Note that we have implemented

something called "Zobrist Hashing" for you to make this easier. Using this idea results in a significant speed-up. For more detail on transposition tables, check the internet.

The bot you use to beat Clamps MUST be a derivative of Jamboree–you must use parallelism here!

(6) 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/Jamboree 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. You may use either your `AlphaBetaSearcher` or `JamboreeSearcher` as a base.

Congratulations! By editing two lines of code, you solved a completely different problem!

Part 4: Write-Up

(7) 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!

For instructions on how to use Google Compute Engine, read this handout

Project Checkpoints

This project will have **two** checkpoints (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 each checkpoint, you (and your partner) will sign up for a 10-minute time slot during which you will meet with a staff member and discuss where you are on the project.

As long as you show up to a time-slot and you do not miss multiple checkpoints in a row, the checkpoint will not affect your grade in any way.

Checkpoint 1: (1), (2)	**Tue, Feb 20**
Checkpoint 2: (3), (4)	**Tue, Feb 27**
P3 Due Date: (5), (6), (7)	**Wed, Mar 07**

Above and Beyond

For this project, beating `flexo` and `bender` will involve a substantial amount of work improving your bot. If you manage to beat `flexo` 5 times out of your last 10 games, that would be substantial. If you manage to beat `bender`, 2 times out of your last 10 games, that would be significantly substantial. Anything that goes toward these goals counts as Above and Beyond! Have fun!