Performance Engineering of Software Systems November 5, 2016
Massachusetts Institute of Technology 6.172/6.871
Profs. Saman P. Amarasinghe and Charles E. Leiserson Handout 27

Project 4: Leiserchess

Beta I: 11:59 PM on Monday, Nov. 14
Beta I Writeup: 11:59 PM on Tuesday, Nov. 15
MITPOSSE Meetings: Monday, Nov. 14 to Wednesday, Nov. 23
Beta II: 11:59PM on Friday, Dec. 2
Beta II Writeup: 11:59 PM on Saturday, Dec. 3
Presentation: Thursday, Dec. 8 and Tuesday, Dec. 13
Final: 5:00 PM on Wednesday, Dec. 14
Final writeup: 5:00 PM on Wednesday, Dec. 14

1 Introduction

In this final assignment, you will put the skills you have learned so far to the test. Each group will start with a high quality game-playing AI for Leiserchess—a chess-like game with lasers! The game-playing AI that we provide implements an algorithm called Principal Variation Search (or PVS). The PVS algorithm is a refinement of the alpha-beta search algorithm and it is commonly used in high-performance chess engines. We recommend that you briefly review the materials on the Chess Programming Wiki to learn a bit about this algorithm:

`https://chessprogramming.wikispaces.com/Principal+Variation+Search`

Unlike previous projects, the code that you start with is not using a naive algorithm. This project simulates situations that you will encounter in real life. As a performance engineer you will often be tasked with finding opportunities to improve the performance of sizable and complex programs that have been implemented by domain experts. It is your job to improve the performance of such programs without compromising their correctness.

This project simulates the situations you will encounter in real life. After designing and implementing a fairly sizable program, you find that it doesn't run fast enough. At this point, you have to figure out what is slow, and decide how to fix the performance issues. We've left in some low-hanging fruit, but you'll find the need to make design changes to the game engine for substantial performance improvement.

2 Leiserchess

The starter program implements a Leiserchess game engine that plays games through the UCI interface (`https://en.wikipedia.org/wiki/Universal_Chess_Interface`). Leiserchess (pronounced "LYE-sir-chess") is a two-player laser-chess game similar to Laser Chess and Khet. The goal is to performance-engineer the game engine so that you improve the player's ability to play the game. There are two common avenues to improve the player's ability.

The first is to improve the search algorithm, such as by coming up with more sophisticated heuristics for the evaluation function or by improving the PVS algorithm provided. The second is to performance-engineer the existing algorithm in order to allow it to search more positions in the game. There are many changes that can be made to the game-engine that do not change its search behavior, but improve performance. If two game-playing programs implement the same search algorithm, then the faster program will be the stronger player because it is able to consider more positions per second.

In this project, you will find that it is more productive to focus on the second avenue. The heuristics used by the evaluation function we have given you are already reasonably good, and the PVS search algorithm is used by domain-experts in chess programming. The implementations of the evaluation function and the PVS search algorithm, however, have not been heavily optimized.

Nevertheless, you are completely free to employ any strategy you wish to improve your program's game-playing performance. The final measure of performance, will be your program's ability to beat a selection of game-playing programs in round-robin tournaments using Fischer Time Control. We'll discuss these details in greater depth in the Getting Started section.

3 Background: Alpha-Beta Search Overview

The given program is an AI for playing Leiserchess that uses Principal Variation Search (PVS) to find the best move from a given position. Principal Variation Search (PVS) is a refinement of alpha-beta search that is itself an optimization of the Minimax algorithm. Let us begin with a brief overview of the alpha-beta search algorithm upon which Principal Variation Search is based.

3.1 Game Trees

In this algorithm, each possible game state, where a state is usually a board layout, is modeled as a node in a tree. The children of a node are the states that are possible by making any of the player's moves. The children of these nodes are the states possible from the opponent's moves, and so forth. For simple games like Tic-Tac-Toe, this tree is small enough that it can be completely generated, allowing for perfect knowledge. For complex games like chess and Leiserchess, this tree is far too large to generate completely. The branching factor, defined as how many states are possible from a given state, is several dozens for the opening move of Leiserchess! The height of the tree is the number of moves from start to finish, which is probably far more than 50. The size of the tree is $branching^{height}$, which is impossible to generate in any reasonable amount of time.

3.2 Alpha-Beta Pruning

Alpha-beta search uses a concept called ***pruning*** to avoid searching the entire tree. Imagine a game tree where you are exploring the two subtrees of the root, as in Figure 1, from left to right. We begin to explore the left subtree and analyze the moves our opponent could make. One game state is an immediate loss for us, labeled B in the figure. Assuming our opponent is as intelligent

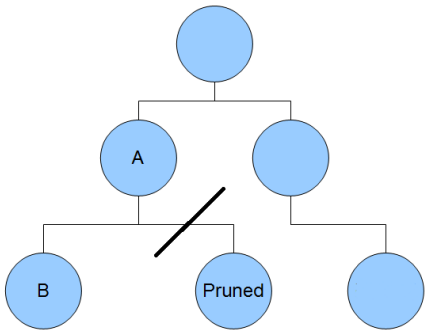

Figure 1: Alpha-Beta Pruning

as we are, if we make the move corresponding to game state A, our opponent will make us lose by making the move which leads to B. There is no point in exploring any of the other children of the A, as we already know our opponent already has a very good move B. We stop exploring this subtree, or prune away the subtree, and instead focus on the other subtree in the hopes of finding a better move. Another way of thinking of this is that once you have found out that a move is bad, there is no need to find out exactly how bad it is.

3.3 Static Evaluation

Pruning helps cut away part of the game tree, but it doesn't address the issue that the tree is still too big to explore to the point of a loss. Each game state is assigned a score by a static evaluator function, where this score is a measure of how good the position is for the player. The static evaluator can be expensive, and it is often difficult to write an accurate one. Instead we explore the game tree to a given depth, and we use our static evaluator on the leaves. Using these scores, we can establish lower and upper bounds, `alpha` and `beta`, respectively, on the scores of the moves available to us at any game state. The `alpha` value is the highest score we are guaranteed to get, and the `beta` is the lowest score our opponent can force us to get. In Figure 2, the `alpha` and `beta` values are initialized to negative infinity and positive infinity, since we have no knowledge of possible scores. We evaluate the left child to have the score of 5. Now we know we are guaranteed at least a 5 and update our `alpha` value accordingly. We then begin to explore the right child. The right child's values are initialized to that of the parent, 5 for `alpha` and infinity for `beta`. Node A evaluates to -3, meaning our opponent can force us into a score at least as bad as -3 if we go the right child's state. We update the `beta` value accordingly. Pruning occurs whenever `alpha` exceeds `beta`. In this example we saw that we had a move with a score of 5 available in the left subtree. By entering the right tree, our opponent can force us to get at most -3. There is no point in exploring the rest of the moves to find out as no better outcome can occur.

While having an accurate static evaluator looks to be the most important thing, there is a big trade-off between having an accurate static evaluator and a fast one. A fast static evaluator allows for a deeper search of the tree in a given time limit, which usually translates into a stronger

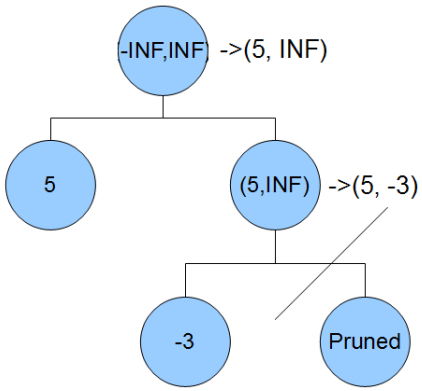

Figure 2: Pruning With Alpha-Beta Values

player, as it can look more moves ahead. The provided evaluation function is the best the course staff could come up with. There is probably room for improvement, but you are advised to focus on producing high-performance code first.

3.4 Parallelization

We do not expect any students to parallelize their program for the first beta deadline. It may, in fact, be counter-productive to attempt this too early since there are significant performance optimizations that are possible in the serial code.

Parallelization, however, will be a necessary change in order to achieve good performance on the second beta and final versions of your program. The staff has designed the code to be fairly easy to parallelize, but we leave the details of the parallelization to you. These changes require some insight into the structure of the program, however, that you will build as you improve the performance of the serial program.

We will release more instructions on how to parallelize your program after the first beta submission. Let us briefly mention a few concepts, however, so that you might keep them in mind as you are optimizing the serial code.

The idea behind parallelizing alpha-beta and Principal Variation Search is to search some selected subtrees in parallel. This is an example of ***speculative parallelism*** because you might search a subtree that would have been pruned due to a beta cutoff.

As we mentioned in the lecture on speculative parallelism, it is necessary to consider two things when introducing speculative parallelism into your program. The first is to ensure that you *abort* computations that are not needed as soon as possible. This is necessary in order to ensure that a beta cutoff correctly terminates searches of subtrees. The second is to avoid the execution of computations that are likely to be aborted. The second consideration is needed to allow your parallel program to obtain the full benefits of parallelization. Afterall, even if searches are correctly aborted your program will not actually search deeper in the game-tree if your additional processing cores are used to search subtrees that are highly likely to be aborted.

An additional consideration when parallelizing your code is contending with races. Care must be taken, for example, to ensure there are no races on the `alpha` and `beta` values. You will also find that the search algorithm maintains some tables for memoizing moves, etc. Races can occur on these tables and you will have to decide whether the races are tolerable. **Nota bene: unlike other programs you have seen so far, some races in game engines are OK.** In particular, you should consider how concurrency impacts the transposition table, killer-move table, and best-move history used by your programs. Some races may be tolerable, others may result in incorrect behavior, and others might be correct but compromise the benefits of various heuristics.

4 Administrivia and Deliverables

Compared to previous projects, this project is loosely structured in order to cut the overhead and give you the most time to work on this project. Here is an overview of the deliverables. Each group needs to submit three versions of the project codebase: Beta I, Beta II, and final. Please also submit a Beta I report describing strategies, progress, issues concerning Beta I. Both the Beta I codebase and the Beta I report will be the basis for MITPOSSE meeting, which happens after the Beta I submission. For the final code base submission, we also require each group to create slides and deliver a 10-minute presentation with the course staff. Each group also needs to submit a final report, which contains a conclusive description of the project and a review of the MITPOSSE meeting. The following provides a breakdown of the details for each item.

4.1 Groups

You are expected to work in groups of **three** or **four** students. You may work with anybody in the class, including previous project partners and classmates who have the same MITPOSSE mentor. We suggest you choose and form your group as soon as possible. Please fill out the following Google form to initiate the process of forming teams.

`https://goo.gl/forms/zu2sXhC9ygGwYBYR2`

4.2 Beta Submissions

There are two Beta submissions. During Beta I, you will optimize the game engine any way you wish *with the constraint that all parallelism is disabled.* During Beta II, you will optimize a parallel version of the code. After each Beta submission, the course staff will run the autotester on all possible pairs of submitted binaries. The staff will publish a performance report on the submitted binaries so that you can get a sense of where your player stands at the point of the Beta submission. Use this information to evaluation what strategy your group should take for the next milestone.

4.2.1 Beta I: Serial optimizations

Beta I is about serial optimizations. In expectation that students will parallelize their program, however, we will evaluate your implementation by running

```
$ make PARALLEL=0
```

So you may use that preprocessor definition in your code to turn features on and off depending on whether you want your serial or parallel code to be evaluated. Feel free to experiment with the parallel code if you have time. Just be aware that you will be judged only on the serial version for this checkpoint.

4.2.2 Beta I Writeup and MITPOSSE Meeting

One the due date of Beta I Writeup, please submit a written report on your Beta I software. This report should be around 1000 words (roughly two pages in a word processor), and will be read by your MITPOSSE mentor and TAs.

At this point, you should have a strong understanding of the game engine's structure and performance characteristics. You should also have made some progress with optimizations, but most importantly, you should have a plan for what optimizations you are going to implement. There are many potential changes you could make, but you should focus on the areas that will give you the best return on time spent. Your report should be submitted on Stellar. To summarize, your review should contain:

- Profiling data on the reference implementation.
- The changes you've implemented so far, and their impact (supported by profiling measurements).
- Optimizations you plan to make (and how you prioritize them), supported by profiling data. You should also estimate the impact of each optimization based on your profiling.
- A work breakdown of how your team plans on dividing the work. You should convince the course staff that your group members are performing equal work and that everyone is doing work worthy of a final project for 6.172.

Please arrange a meeting with your mentor before the MITPOSSE meeting due date. Remember that you are required to meet with your MITPOSSE mentor. The meeting provides a valuable opportunity to receive feedback both from an industry expert regarding your plans for the final project. Please include a summary of your meeting in your final written report.

4.2.3 Beta II: Parallel optimizations

Beta II will proceed exactly as Beta I, except, your bot will be run on a multicore machine. You're encouraged to parallelize your bot to obtain parallel speedup. Since there may be some changes to your bot that you only want active when running on multiple cores, we will compile your code using the command:

```
$ make PARALLEL=1
```

4.2.4 Beta II Writeup

One the due date of Beta II Writeup, please submit a written report on your Beta II software. This writeup has the same structure as the Beta I Writeup, and will be read by your TAs to assess your progress.

4.3 Final Deliverable

The final deliverables for this class consists of a final presentation and a final write-up, consisting of your final code submission and an overall final report on the project. Each component is detailed below.

4.3.1 Final Presentation

Before the code is due, we expect each group to deliver a 10-minute presentation. Suggested topics to cover in your presentation include:

- Your overall optimization strategies.
- How you implemented or approached your plan.
- What kind of performance bottlenecks you observed and solved.
- Brief overview of the performance results.
- Analysis, if there is any.
- Brief description of the work breakdown.
- Other thoughts, such as those strategies that you tried but didn't work.

Please submit your slides *the day before the presentation*. They will be compiled by the staff in order to reduce the startup time required for each presentation. The presentations will run for the entire day, and each team will be assigned a 10-minute slot. All group members must attend the participation. We will provide more information about the presentation format after the second beta deadline.

4.3.2 Final Code Submission, Tournament

Please turn in your code by the final turn-in deadline. We will grade your program by autotesting, but we will hold a live exhibition tournament with prizes for the winners. Come and see how you and your peers did!

4.3.3 Final Report

To help us with grading your submission, please submit a final project report that briefly outlines:

- The optimizations in your submission.
- Optimizations that you tried, but didn't work.
- The work breakdown within your group (to be submitted individually).
- Any extra information you think would be helpful for us in assigning a grade.
- Description of your meeting with your POSSE mentor, if you attended the meeting.

Unlike the Beta writeups, there is no page limit to the final writeup, but as always don't waste words. Feel free to reuse (copy/paste) material from your Beta I and II writeups where applicable.

5 Getting Started

As with the previous projects, use the following command to clone the git repository that has been set up for your group for the assignment.

```
$ git clone /afs/athena.mit.edu/course/6/6.172/student-repos/fa16/\
projects/project4/team-name.git project4
```

You will also need to install Java and Tcl on your personal Azure instances to use some of the project scripts, which you can do by running:

```
$ sudo apt-get install openjdk-7-jdk tcl
```

There is significantly more documentation in this project than in other projects; for convenience, here are additional supporting documents you will find useful:

1. `/README`: description of the code base structure, how to launch a game server and how to test your code.
2. `player/README`: summary of code files that you may find yourself modifying.
3. `doc/Leiserchess.pdf`: introduction to Leiserchess rules.
4. `doc/engine-interface.txt`: description of commands you can send Leiserchess at the commandline. See `/player/leiserchess.c` for more information.
5. Google doc:
   `https://docs.google.com/document/d/1Sg9otidhfbC44sO0UmJycarcI4CjSKB4IZpStzDZXG8/edit?usp=sharing`
   The above google doc is an *evolving* glossary of terms that the code takes for granted (i.e.,

you will probably not be familiar with them). The document is not yet complete and will fill out over the course of the project. *We have enabled student annotations on this document.* If you have a comment, would like a term clarified or would like to add an entry to the document, write an annotation and the course staff will update the document itself.

The main codebase that you will be working with is the game engine in the `player` directory. You can use `make` to compile the player code, which generates `leiserchess`. This program follows the Universal Chess Interface (UCI), which is a set of text commands sent via `stdin` that a Leiserchess engine must respond to at any point during runtime. Those commands are described in the UCI document, `doc/engine-interface.txt`. The given implementation supports an additional command `perft`. This command counts all possible moves to a given depth, which makes it a valuable debugging tool for your move-generation code.

5.1 Testing Tips

- For correctness checking, run your bot and the reference bot together and verify that they make the same moves at each time step for the same starting board state. Both bots should search at the same depth; otherwise, it would make sense for your bot to make a different move because it is faster and able to search deeper in the tree.
- You can use FEN strings/move lists to facilitate debugging. If your AI is exhibiting a bug of some kind, you can use a FEN string or move list for the board right before the bug appears to quickly reset the board and hopefully more quickly elicit the bug. You can use the output of the autotester to see games where the AI crashes and to get the relevant move lists for those games.
- You might be tempted to evaluate your performance by running the UCI command `go depth` and looking at the Nodes Per Second (NPS) metric. This is generally a bad idea, as NPS may vary at different points in the game depending on the complexity of the search tree traversal. It's better when possible to run many games with the autotester and see whether there's a significant difference in ELO rating.
- To spot egregious gameplay bugs, we recommend that you watch some games on the scrimmage server.
- Use git branches to isolate various optimization strategies.

6 Other considerations

The other directories contain tools that will assist you to performance engineer the player code. In particular, we have provided an autotester framework and Elo-rating software which will allow you to evaluate two different versions of your player to determine whether the modifications you have made indeed help the player to perform better. Instructions on how to run the autotester and the rating software are also included in the top-level `README` file.

The Leiserchess engine consists of a fair amount of code. You will have to decide how to use your limited time to achieve the best results. Here are some suggestions from us about how to begin:

- Make sure you are comfortable with the concepts behind how alpha-beta search works. Without this background, it will be difficult to think of substantial optimizations or troubleshoot.
- Make sure you understand how the code is organized. Although it's probably a waste of time to read through every line of code before beginning, make sure you understand the big-picture components of the code and how they interact.
- Use the profiling tools you've learned in this class to identify hotspots. Profiling should reveal substantial the low-hanging fruit.
- Use Cilksan to make sure you don't have race conditions. Program crashes count as losses when we autotest your program.
- Never lose sight of the big picture. Once you find one hotspot, it's easy to get sucked into optimizing it to death. Recognize when you've made sufficient progress and move on!
- Make incremental changes. Avoid structuring your changes such that you cannot compile or run your code for extended periods of time. Performance surprises tend to be unpleasant in nature.
- We recommend that you make any changes to the board representation as early as possible. These changes may affect the behavior of other parts of the code base.
- The static evaluator in the codebase uses the best heuristics the staff could devise. Although there could very well be better heuristics, there is plenty of low-hanging fruit from simply optimizing the program before you bother coming up with your own. Do keep in mind there is a trade-off between having an accurate static evaluator versus a fast one. A program using a less-accurate evaluator can be stronger than a program with a more accurate one if it is faster and can search deeper in the tree.
- Keep an open mind. When looking at preexisting code, it's easy to be lured into following the footsteps of the original authors. You should ask yourself whether the data structures and algorithms chosen are appropriate. Is there a faster (or more parallelizable) method?
- The chess-programming wiki (`http://chessprogramming.wikispaces.com/`) is an invaluable resource. While Leiserchess differs from chess in many important ways, many concepts in writing fast chess engines should apply to Leiserchess as well.
- Your team has a share of the computing resource on the Azure machines. Explore ways to have it do useful work, even while no team member is actively developing.

6.1 Familiarize Yourself with the Leiserchess Game

Besides getting the code to run fast, we also want you to have lots of fun—the project is about performance engineering a *game* engine after all! It is a good idea to start out by playing a few games to familiarize yourself with the rules of the game. Once you download the codebase, find the accompanying document in `doc/Leiserchess.pdf`, which describes the rules for Leiserchess. Run the game in your own browser using instructions included in the top-level `README` file in the codebase. We also have set up a server to allow you to play a game online: `http://leiserchess.csail.mit.edu/`.[1] Feel free to share this URL with your family and friends.

6.2 Rules and Fine Print

Your program must obey the following ground rules:

- You may not rely on other people's code for any significant functionality in your submission. Reference materials and borrowed code snippets should be cited in your final report.
- You need not use the Cilk extensions for parallelization. You may use TBB, Pthreads, OpenMP, or any other platform.
- Your program must run on Azure resources. You may not, for example, use a distributed-computing package to source external computational resources.

When in doubt, don't hesitate to ask the course staff for clarification.

[1] Currently, the server is only running on a one-core machine, and so it may run a bit slowly if too many people connect to it. We plan to upgrade the hardware soon.