Course: Project AI - Symbolic artificial intelligence
Professor: Dr.-Ing. Stefan Fricke
Semester: SOSE23
Date: 2023/07/26

Group Members: Who the fuck cares?
Repository: https://github.com/PraxTube/chess-ai/

Project Report

Python Chess AI

Group C - The Plebs

This is a dummy abstract.
Hello there.

Contents

1 Introduction

Creating a chess AI from scratch is quite a challenging undertaking. Not only will you need to write a whole chess backend, you will also need to implement the AI features. One of the major issues here is to write a chess backend without any bugs and to test your AI properly to make sure the features you add actually make it play better.

Our group chose to use Python for the whole project given that that is what we were most familiar with. The obvious trade-off here is of course that it's easy to prototype but painfully slow and very error prone. I personally would have liked to try to use Rust, though in hindsight we would have probably abandoned the project if we had used Rust simply because the chess engine alone was so much work. On the other side I have acquired some Rust experience now and if I were to write the chess engine (or something of a similar level) I would probably go with Rust.

Regardless of our programming language, for version control we obviously used git and to share our code base we used github. The overall workflow here was pretty smooth.

This was pretty much the first *real* AI project we took on. Our goal was to at least get a basic AI done, that was what we wanted to to reach at least. Our final AI is actually fairly advanced for what we seeked to accomplish. It's obviously not the strongest (given that we are using python that isn't too surprising). However we are very pleased with the end result and with what we created and learned during this project.

In the following chapters we will go into more detail into what, why and how we created our AI. We will also reflect on all these things.

2 Development

Before we are going into the details of the development process, let's first look at an overview of what we created (note that you can click on the Mst name to see whole documentation of it).

Mst1 - Dummy AI:

- Chess Backend
- Dummy AI (minimax)
- Basic Evaluation (Material only)

Mst2 - Basic AI:

- Improve Backend
- Alpha-Beta Tree Search
- Improved Evaluation (PeSTO)
- Better time management

Mst3 - Advanced AI:

- Restructured Chess Backend
- Speed up Evaluation (through numpy)
- Improve move ordering
- Include King of the Hill in evaluation
- Restructure internal debug info

Mst4 Optimized AI:

- Improve evaluation
- Use Monte Carlo Tree Search
- Implement PVS/negamax
- Add Nullsearch

2.1 Mst1 Dummy AI

This milestone was primarily about the chess backend. At first we used the python package python-chess[1] which was very useful to see what kind of structure our backend should have. We implemented the AI using python-chess and later when we switched the backend to our own, we were able to use the structure of python-chess. Our backend is mainly based on one other repo[2] and a youtube video[3]. Of course we also consulted the chess programming[4] website to gain a deeper understanding of how a chess backend should be structured.

The benchmarks were run on both our backend and the python-chess backend, with the goal in mind to see just how much slower our implementation is. To our surprise however, our backend was actually *faster*. It's still not clear why that is, but it was very reassuring and gave us the confidence to continue with the project.

Given that his milestone was mainly about the backend, naturally the main issues and lessons we learned stem from it. Firstly, the main problem with the backend was that really only one person could work on it. Sure you might have been able to split the work of AI and backend, but having multiple people work on one backend just isn't a good idea. That is why this task was done by only one person. The backend ended up with 900+ lines of code in this milestone.

For this first iteration of the backend we only went with the most necessary features, we didn't implement the following:

- King of the hill condition
- En passant
- Only queen promotion
- Fen loading only semi-working

The code base of the backend was also really messy with inconsistent naming, redundant code, many nested if's and for's and hard to debug methods. However this was to be expected and was cleaned up in later milestones. It's also worth noting that this is actually the better way to developing software in general (at least for in my experience):

Write whatever software you need, following only the happy path[5] until it works. If you notice that you can restructure the code, do it. Once that is done you can go back and add all the checks and validations. This worked really well in pretty much all of our milestones. It's essentially like a fail fast approach.[6]

[1] https://pypi.org/project/python-chess/
[2] https://github.com/Jabezng2/Star-Wars-Chess-AI-Game
[3] https://www.youtube.com/watch?v=EnYui0e73Rs
[4] https://www.chessprogramming.org/Chess
[5] https://en.wikipedia.org/wiki/Happy_path
[6] https://en.wikipedia.org/wiki/Fail-fast

2.2 Mst2 - Basic AI

The time on this milestone was only 2 weeks, so there wasn't too much progress compared to the first one. However there were nonetheless changes to both the backend and the AI. Firstly, the backend uses arrays of integers to represent the board instead of strings now. This is not only faster, but it allows to quickly calculate evaluations using numpy arrays which is very important for the next milestone. We restructured the backend to use less OOP[7] and more FP[8]. In addition we split some methods into smaller ones to increase logical flow and readability.

These changes had a minor increase in performance but a massive increase in readability and scalability, as we will see later. Refactoring at least a bit every day was a good way to make progress here. This way you are not meet with the daunting task of restructuring a huge code base, but rather improvement it step by step. This rule only works in the early stages (prototype stages) of a project though, as you don't want to mess with a big code base that is potentially getting worked on by other people or even yourself on other side branches.

The AI got some noticeable improvements as well. We went from the minimax algorithm to alpha-beta, which seemed to be actually slower in our AI, however we later on realized that there was a bug in our move ordering, and so our alpha-beta was actually significantly faster then the minimax implementation. Apart from the performance the strength of the AI also increased by adding PeSTO[9] to the evaluation. This change was computationally very expensive with our naive version, we will look at how we improved that in the next milestone. The time management was also improved from using a constant time every move to allocating time depending on the stage of the game and the time available (less time for early moves, lots of time for mid-late game moves).

2.3 Mst3 - Advanced AI

This milestone was the most productive out of all of them (we also had the most time for this one). The backend received a huge restructure[10], we cleaned up all the names to use snake_casing consistently, we replaced boilerplate heavy classes with smaller datastructures such as lists[11], we added guard clauses[12] whenever possible to increase readability and reduce indentation and finally we got rid of the string board representation entirely and used only integers instead.[13]

The AI increased both in performance and in strength during this milestone. We implemented a proper king of the hill win condition to the evaluation (and to the backend), we refactored the move ordering to be much faster (at the cost of being less accurate, a worthwhile trade), made some slight tweaks to

[7] https://en.wikipedia.org/wiki/Object-oriented_programming
[8] https://en.wikipedia.org/wiki/Functional_programming
[9] https://www.chessprogramming.org/PeSTO%27s_Evaluation_Function
[10] See the class diagram in Figure 4.
[11] Using numpy arrays in most instances was actually slower then pure python lists. The reason for this is that indexing into numpy arrays is pretty slow and most of the time we had to loop through the lists and index the elements.
[12] https://en.wikipedia.org/wiki/Guard_(computer_science)
[13] For a more detailed description see the commits in https://github.com/PraxTube/chess-ai/pull/37/commits

the time management and we most importantly tried to implement transposition table. However that last point went horribly wrong.

The issue with the transposition table was that we had way too many collision for the number of boards that were being hashed. For roughly 10k boards that were hashed there were around 20 collisions. If you didn't wipe the table after each ply, then the number of collisions would skyrocket after that, to the point were searching 50k boards resulted in 25k collisions, so 50%. Any type of collision handling would be more then futile here.

It goes without saying that something must be going wrong here, though if we consider the birthday paradox[14], then it doesn't seem so bizarre actually. Using the rule of thumb of

$$p \approx \frac{n^2}{2m}$$

with the probability of collisions p, the number of boards hashed n and the amount of entries in the board m. In our case we have:

$$\begin{aligned} m &= 2^{24} \\ p &= \frac{1}{2} \\ n &\implies \sqrt{2 \cdot p \cdot m} = \sqrt{2^{24}} = 2^{12} = 4096 \end{aligned}$$

so if we hash 4096 boards we should expect that there will be at least one collision with a chance of 50%. So collisions with the number of boards we hashed are actually extremely likely, and given that they grow exponentially, it's not too surprising to find these huge numbers of collisions. Though there was still likely a bug in our implementation because we had significantly more.

What is the solution to this problem? Avoidance, after trying out many things[15] things to solve the issue, we eventually settled to simply take our losses and not implement transposition-tables. Funnily enough, had we taken some time to research how effective the transposition-tables would have been, we would have realized that even a working implementation would have at best given a 10% to 20% boost in performance.

This milestone, even with that setback, was still the most productive of them all.

[14] https://en.wikipedia.org/wiki/Birthday_problem

[15] https://github.com/PraxTube/chess-ai/blob/master/docs/milestones/3-advanced-AI/transposition-tables.md#trying-to-solve-the-problem

2.4 Mst4 - Optimized AI

In this milestone we focused on optimizing the AI rather then refactoring anything about the backend. We didn't face any major issues during this process of the project, probably because we worked more smoothly together and because the features we implemented were already somewhat covered in other courses.

For instance we implemented MCTS[16] which some members of the team already covered in a separate course[17]. This actually allowed us to implement it fairly fast without any major hiccups. The main challenge here was to balance exploration and exploitation effectively to improve the AI's decision-making. Additionally, tuning the exploration parameter and the number of simulations proved to be crucial for achieving good performance. Though given that we were already quite familiar with MCTS we managed to overcome this challenge and integrate it successfully into our AI.

PVS[18] combined with negamax[19] was the hardest feature to implement. The primary challenge we encountered was understanding the intricacies of the algorithm. Ensuring the correctness and efficiency of our PVS/negamax implementation was the main obstacle. However, with thorough research and trail and error, we were able to overcome these challenges successfully We also had some problems when implementing this feature in our code base simply because the AI framework was pretty bad. While we did refactor the chess backend in the previous milestone, we didn't do the same for the AI (though to be fair, the AI is much less complex). So we also restructured the AI framework (partially) with the features we implemented in this milestone.

The nullsearch was another feature we implemented during this milestone. The main challenge here was determining when and where to apply null moves and ensuring that the search results remained accurate. Tuning the nullmove heuristic and fine-tuning the implementation took some time, but we managed to integrate nullsearch into our AI effectively.

Finally we made some improvements to the evaluation:

- Check if it's late game, if so, use different PeSTO table
- Evaluate King danger
- Punish bad pawn structure (isolated, backward, not right aligned)
- Pigs on the 7th rank (rooks on 7th rank)

While the above additions make the evaluation slower overall, we think that it was a good trade-off for a stronger AI. Fine tuning a few hyperparameters (through research and trial and error) also helped to increase the strength without any trade-offs. We also tried to implement mobility evaluation, however in our case it was just way too slow so we removed it. Also, checking if two bishops are present was too slow (compared to the benefit it provided).

[16] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
[17] The courses we are referring to is *Einfuehrung in die KI* and *Cognitive Algorithms*, though primarily the former one.
[18] https://www.chessprogramming.org/Principal_Variation_Search
[19] https://www.chessprogramming.org/Negamax

The reason some features in the evaluation worked well and others didn't was mainly a question about can we vectorize it properly with numpy. Features that required calculations that could be cleverly put into one single numpy call worked the best. On the other, features that only required checks with loops over the board were simply to slow compared to the rest.

3 Results

In this section we will focus on the benchmarks and the more tangible results of the project. The tests were run on a PC with the following specs

- CPU: Intel i5-4590, Threads: 4, Cores: 4, 3.7GHz
- RAM: 24GB DDR3
- OS: Zorin 16.2 (Ubuntu based)

To start, lets take a look at Figure 1.

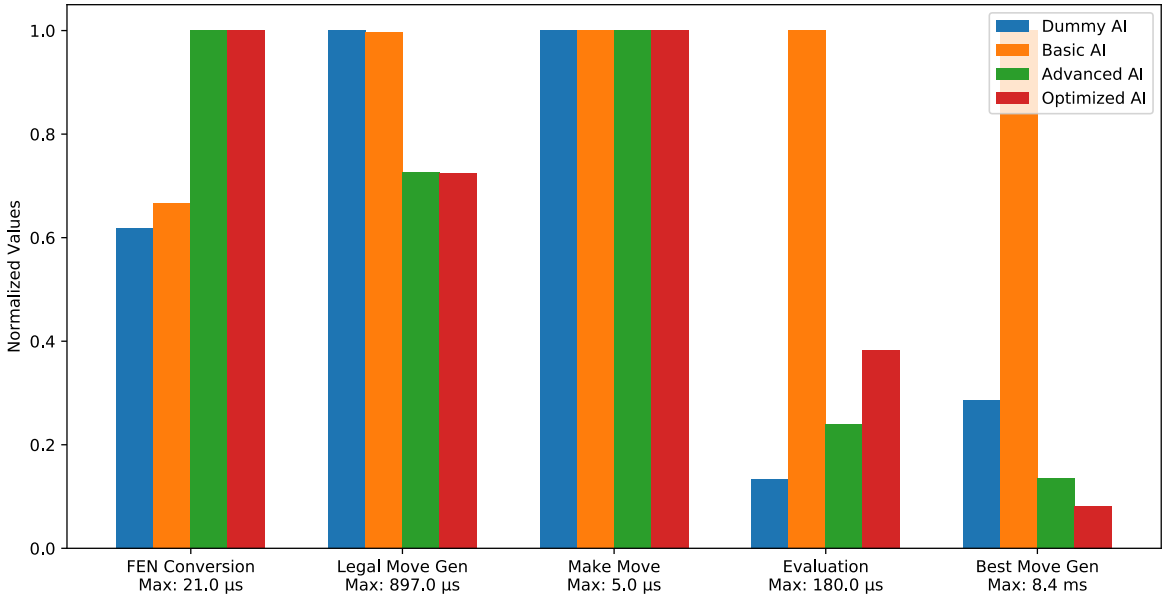

Figure 1: Benchmarks of the different categories across the AI versions.

We can see five categories, of which the first three are determined by the chess backend and the later two by the AI. Important to note in the backend benchmarks is the legal move generation. This is the main bottleneck of the backend and a increasing the performance there has quite the effect on the overall performance of the AI. The decrease in time stems from the refactor of using lists of integers and reducing boilerplate[20]. The most important category however is the last one, the *Best Move Generation*. As the name suggests, this indicates how fast the AI calculated the best move, here at a depth of one. The optimized AI has the best overall time, even though it's slower in the evaluation compared to the previous version. This illustrates very nicely that the end result of an AI, it's strength and speed, is determined by multiple factors. In this case the amount of nodes searched was drastically decreased by the AI features implemented which led to this overall improvement.

[20]The Fen conversion is only used for debugging purposes, the fact that it increased is not noticeable in an optimized AI that doesn't debug anything.

This leads us to Figure 2, which shows the time and number of nodes searched in respect to the depths. Here we can see numbers of nodes searched (bottom plot) is significantly lower for the optimized AI compared to the advanced AI[21]. We can also see that the overall time is smaller for the optimized AI, regardless of depth.

This is actually not necessarily the case, for instance if we would have made poor decisions for the MCTS with regard to exploration/exploitation and this could have actually been slower for higher depths. This is where regular benchmarks, or atomic benchmarks are very useful, you change one little nod and see how the system responds. Research also helped to fine tune.

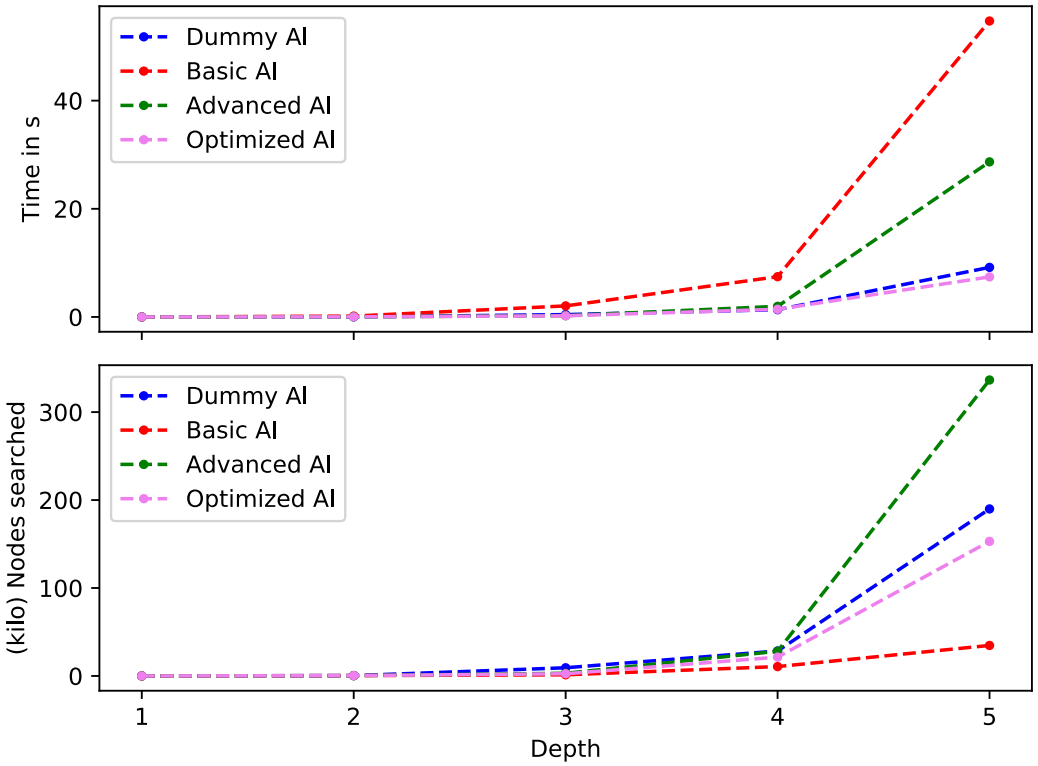

Figure 2: Benchmarks of different AI versions in respect to search depth. Note that the number of nodes searched is in thousands (kilo).

[21] The reason the dummy and basic AI seem to have less nodes searched is because there was a bug in the early implementations of the AI. This bug had to do with the move ordering and was fixed in milestone three. For that reason the dummy and basic AI's benchmarks in regard to depth should be taken with a grain of salt.

4 Issues faced

This is a collection of issues that we faced throughout the project. They are in no particular order, though they are somewhat implicitly ordered by milestone 1 to milestone 4.

- Team coordination (mainly due to the fact that only really one person can work on the engine)
- Writing the chess engine and debugging it was difficult, a more guided approach would have been better instead of trial and error
- Writing unit tests early wasn't very useful
- Getting started was the hardest part, as we didn't quite know what to do (where to begin)
- Using numpy mindlessly doesn't automatically make your code faster (it can actually make it slower!)
- OOP is really annoying to work with, especially if the design of the software is not to clear
- Having huge functions with large logical statements
- Changing too many things at once (as opposed to having atomic changes)
- Commiting to things that don't pay off in the end

Most of these issues were very useful in the sense that they made for great learning opportunities. The lessons we learned during this project is what we will be discussing in the following chapter.

5 Lessons learned

The process of writing a chess AI is not an easy one. There are many things that can and will go wrong. That's why this was a great way to learn, because the first step to mastery is failure. The following list is a collection of lessons we learned during this project, they are in no particular order.

- Benchmarks are extremely useful, not only to see improvements over different versions of your code, but also to compare incremental changes to the code. We realized this when we tried to refactor the evaluation function to use numpy. When we had many small np.ndarray it was actually slower then the pure python list implementation. This is because we always have overhead when calling numpy, so reducing the amount of times we call numpy draws out the full potential of numpy, i.e. use as big as arrays as possible.
- We also observed that background tasks can significantly influence the result of benchmarks. One should try to run them in the same-ish environment as possible (or use a server for that if possible).
- Writing debugging tools as early as possible really pays off (same with logging tools).

- Failing fast in the early stages of the project (using python-chess to code up a simple AI) created a good base knowledge about what needs to be done.
- Writing good git commits is extremely useful for both documentation and overall work flow. Atomic commits are almost necessary when restructuring complex software like the chess backend.
- Unit tests really useful when restructuring complex software as well.
- It's important to know if something will be worthwhile before commiting to it if it will take a long time to complete. We learned this with the attempt at implementing transposition tables. If we would have known that the potential performance increase was about $\tilde{1}0\%$, we would have probably not even tried it in the first place, given how much effort went into it.
- Debugging hash tables is actually not as straightforward as it initially seemed. If you can't use a debugger properly then it's a real mess. Being able to use a debugger is very essential and an important tool in a software developers toolkit.

6 Summary

This is a dummy summary.

Hello world.

Literatur

Test

Appendix

Plagiatsklausel

Hiermit versichern wir, dass wir diese Projektarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Die Stellen unserer Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, haben wir in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Dasselbe gilt sinngemäß für Tabellen, Karten und Abbildungen. Diese Arbeit hat in dieser oder einer ähnlichen Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

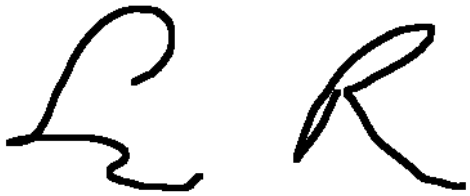

Figure 3: Signature

Class Diagram

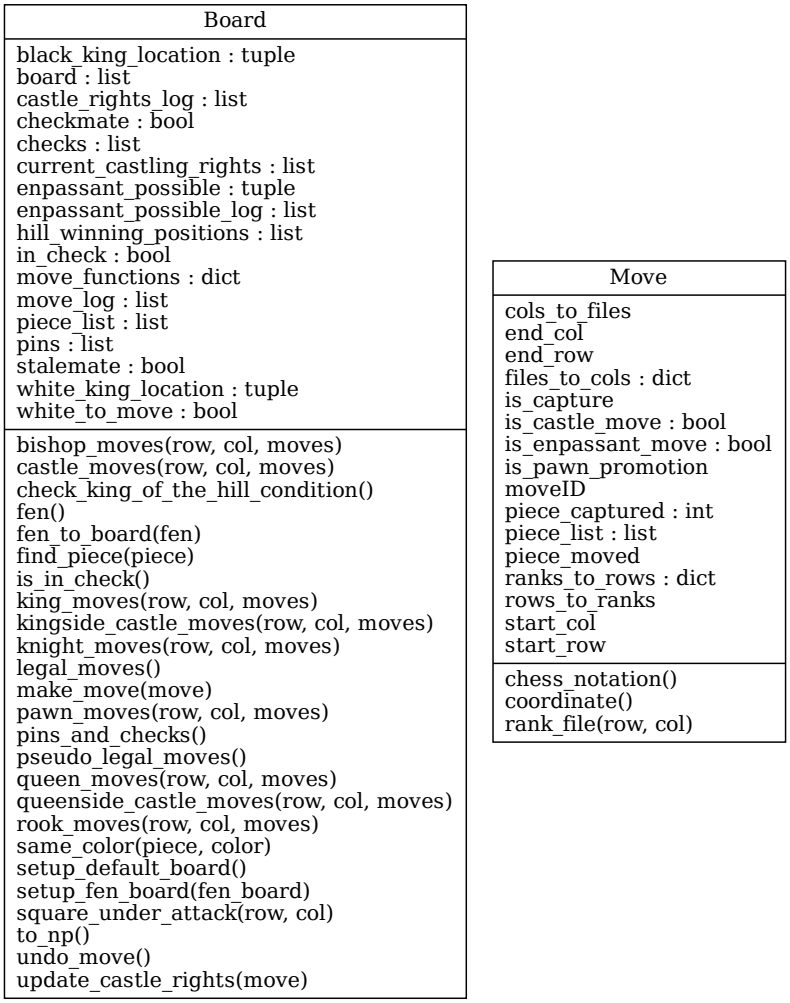

Figure 4: Class diagram of the chess backend in its latest state