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

Group Members:

- Luka Rancic, 453423
- Stefan Krakan 450838,
- David Maximilian Fränkle, 457054
- Joris Christopher Gabrisch, 457230

Repository: https://github.com/PraxTube/chess-ai/

Project Report

Python Chess AI

Group C - The Plebs

This report presents a detailed account of a project focused on the development of a King-of-the-Hill chess AI. The team faced several challenges during the project, including dealing with large functions with complex logical statements, managing object-oriented programming (OOP), and optimizing code for speed. These challenges, however, provided valuable learning opportunities. The team learned the importance of making atomic changes and gradually improving the code base, especially during the early stages of a project. The AI saw noticeable improvements over time, and the team was able to implement vast amounts of features by the final milestone. Overall, the project was a positive learning experience for the team.

Contents

1 Introduction

Starting the development of a chess AI from the ground up presents a considerable challenge. Not only will you need to write a whole chess backend, you will also need to implement the AI features. A significant hurdle in this process is the development of a bug-free chess backend, coupled with the need for rigorous testing of the AI to ensure that the integration of new features enhances its performance.

Our group chose to use Python for the whole project driven by our collective familiarity and proficiency with this language. his choice, however, was not without its trade-offs. While Python facilitates rapid prototyping, it is also inherently slow and can be susceptible to errors. Some team members expressed an interest in exploring the use of Rust, a language known for its performance and memory safety. However, in retrospect, the adoption of Rust [2] might have led to the termination and frustration in the early stages of the project, given the extensive workload associated with the development of the chess engine alone. That being said, some of us have since gained experience with Rust and, in light of this, would now prefer to use it over Python in a project like this.

Throughout the course of this project, we utilized Git and GitHub for version control, irrespective of our individual programming language preferences. This approach facilitated a seamless and efficient workflow with minor merge conflicts while working simultaneously on this project.

This project represented our first substantial engagement with AI development. Our initial aim was to create a basic AI, a goal we not only achieved but surpassed. The final product, while not the most powerful due to our use of Python, exceeded our expectations and provided us with valuable learning experiences.

In the following sections, we will dive into the specifics of our AI's creation, discussing the methods we employed, the reasoning behind our decisions and the lessons we learned. We will also provide a reflective analysis of the project as a whole.

2 Development

Before diving into the details of the development process, it is beneficial to first provide an overview of the final product we created. Please note that comprehensive documentation can be accessed by clicking on the respective milestone name.

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

The primary focus of this milestone was the development of the chess backend. Initially, we utilized the Python package python-chess[1], which served as a valuable reference for structuring our own backend. We first implemented the AI using python-chess, and when we transitioned to our own backend, we were able to retain the structure provided by python-chess. Our backend was largely influenced by another repository[2] and a YouTube video[3], which both provided crucial insights. Additionally, the Chess Programming website[4] was an invaluable resource, offering a deeper understanding of the structure and functionality of a chess backend.

We conducted benchmarks on both our backend and the python-chess backend, with the intention of comparing their performance. Contrary to our expectations, our backend demonstrated superior speed. The reason for this remains unclear, but it was a reassuring outcome that bolstered our confidence to proceed with the project.

As this milestone was primarily focused on the backend, the main challenges and lessons learned were inherently linked to it. One significant issue was that the backend development was essentially a single-person task. While it might have been possible to divide the work between the AI and the backend, having multiple group members working on a single backend could lead to complications. Consequently, this task was undertaken by a single team member, resulting in over 900 lines of code in this milestone.

In this initial iteration of the backend, we focused on implementing the most essential features, leaving out the following:

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

The initial code base for the backend was admittedly disorganized, characterized by inconsistent naming conventions, redundant code, complex nested conditional and loop statements, and methods that were challenging to debug. However, this was anticipated and subsequently addressed in later milestones. It's worth noting that this approach aligns with a common software development strategy that has proven effective in my experience:

The strategy involves initially developing the software to follow the "happy path"[5] until it functions as intended. Any potential restructuring of the code is undertaken as and when it is identified. Once

[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

this is accomplished, additional checks and validations can be incorporated. This approach was highly effective across all our milestones, essentially embodying a "fail-fast"[6] methodology.

2.2 Mst2 - Basic AI

Despite the limited timeframe of two weeks for this milestone, we made notable modifications to both the backend and the AI. The backend was updated to represent the board using arrays of integers instead of strings, enhancing speed and facilitating quick evaluations with numpy arrays, a crucial aspect for the subsequent milestone. We also restructured the backend to favor functional programming[7] over object-oriented programming[8], enhancing its efficiency. Furthermore, we divided some methods into smaller ones to improve logical flow and readability. The time on this milestone was only 2 weeks, so there wasn't too much progress compared to the first one.

These changes had a minor increase in performance but a massive increase in readability and scalability, as will be evident in subsequent stages. Adopting a strategy of daily refactoring proved beneficial, allowing for incremental progress rather than confronting the daunting task of overhauling a large code base all at once. However, it's important to note that this approach is most effective during the early or prototype stages of a project. It is less advisable to apply this strategy to a substantial code base that is concurrently being worked on by other team members or across different branches.

The AI also underwent significant enhancements during this phase. We transitioned from the minimax algorithm to alpha-beta pruning. Initially, this appeared to slow down our AI, but we later discovered a bug in our move ordering. Once rectified, our alpha-beta pruning implementation proved to be considerably faster than the minimax approach.

In addition to performance improvements, the AI's strength was augmented by incorporating the PeSTO[9] evaluation function. Although this modification significantly increased computational demands with our initial version, we will discuss how we optimized this in the subsequent milestone.

We also refined the AI's time management strategy. Instead of allocating a constant amount of time for each move, we adjusted the time allocation based on the stage of the game and the remaining time (less time for early moves, more time for mid-to-late game moves).

2.3 Mst3 - Advanced AI

This milestone was the most productive, aided by the fact that we had the most time allocated for it. The backend underwent a substantial restructuring, as detailed in the class diagram in Figure 4. We standardized all names to use snake_casing consistently and replaced complex classes with simpler data structures, such as lists[10]. Interestingly, we found that using numpy arrays was often slower than

[6] https://en.wikipedia.org/wiki/Fail-fast
[7] https://en.wikipedia.org/wiki/Functional_programming
[8] https://en.wikipedia.org/wiki/Object-oriented_programming
[9] https://www.chessprogramming.org/PeSTO%27s_Evaluation_Function
[10] 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.

pure Python lists due to the time-consuming indexing process.

We also incorporated guard clauses[11] wherever possible to enhance readability and reduce indentation. Finally, we eliminated the string board representation entirely, opting to use only integers instead. For a more detailed description of these changes, please refer to the commits in the provided GitHub link[12].

During this milestone, the AI demonstrated improvements in both performance and strength. We incorporated a proper king of the hill win condition into the evaluation and backend, refactored the move ordering to enhance speed (albeit at the expense of some accuracy, a trade-off we deemed worthwhile), and made minor adjustments to the time management. Most significantly, we attempted to implement a transposition table, but this effort was fraught with challenges.

The primary issue with the transposition table was an excessive number of collisions relative to the number of hashed boards. For approximately 10,000 hashed boards, there were around 20 collisions. If the table was not cleared after each ply, the number of collisions would surge dramatically. For instance, searching 50,000 boards resulted in 25,000 collisions, a collision rate of 50%. Any attempt at collision handling under these circumstances would be futile.

It is evident that there are issues to be addressed in this scenario. However, when considering the birthday paradox, the situation may not be as unusual as it initially appears. Applying the rule of thumb, we find that

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

with the probability of collisions p, the number of boards hashed n and the amount of entries in the board m. In our specific scenario, the following parameters apply:

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

Given these parameters, if we hash 4096 boards, we should anticipate at least one collision with a probability of 50%. Therefore, collisions are not only extremely likely with the number of boards we hashed, but their occurrence is expected to increase exponentially. This explains the substantial number of collisions we encountered. However, the excessive frequency suggests there may have been a bug in our implementation.

[11] https://en.wikipedia.org/wiki/Guard_(computer_science)
[12] For a more detailed description see the commits in https://github.com/PraxTube/chess-ai/pull/37/commits

How did we address this issue? Ultimately, we chose avoidance. After exploring numerous potential solutions[13], we decided to accept the limitations of our approach and forego the implementation of transposition tables. Interestingly, had we conducted a preliminary analysis of the potential effectiveness of transposition tables, we would have discovered that even a successful implementation would have resulted in a modest performance boost of 10

Despite this setback, this milestone remained the most productive of all.

2.4 Mst4 - Optimized AI

During this milestone, our efforts were primarily directed towards optimizing the AI, rather than refactoring the backend. We encountered no significant obstacles during this phase of the project, likely due to our increasingly efficient collaboration and the fact that the features we implemented had been partially addressed in other courses.

For example, we implemented the Monte Carlo Tree Search (MCTS)[14], a topic some team members had already explored in a separate courses[15]. This prior knowledge facilitated a swift and smooth implementation without any significant issues. The primary challenge was effectively balancing exploration and exploitation to enhance the AI's decision-making capabilities. Furthermore, fine-tuning the exploration parameter and the number of simulations was crucial for optimizing performance. However, given our existing familiarity with MCTS, we were able to successfully navigate these challenges and integrate it effectively into our AI.

The implementation of Principal Variation Search (PVS)[16] in conjunction with Negamax[17] presented the most significant challenge. The primary difficulty lay in comprehending the complexities of the algorithm and ensuring the accuracy and efficiency of our PVS/Negamax implementation. However, through diligent research and a process of trial and error, we were able to successfully navigate these challenges.

We also encountered difficulties when integrating this feature into our existing code base, primarily due to the limitations of our AI framework. While we had refactored the chess backend in the previous milestone, we had not done the same for the AI. However, it's worth noting that the AI is inherently less complex. Consequently, we partially restructured the AI framework in conjunction with the features implemented in this milestone.

Another feature we introduced during this milestone was the nullsearch. The primary challenge associated with this feature was identifying the appropriate circumstances and locations for applying null moves while maintaining the accuracy of the search results. The process of tuning the nullmove heuristic and refining the implementation required some time, but we were ultimately successful in incorporating nullsearch into our AI.

[13] https://github.com/PraxTube/chess-ai/blob/master/docs/milestones/3-advanced-AI/transposition-tables.md#trying-to-solve-the-problem
[14] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
[15] The courses we are referring to is *Einfuehrung in die KI* and *Cognitive Algorithms*
[16] https://www.chessprogramming.org/Principal_Variation_Search
[17] https://www.chessprogramming.org/Negamax

Lastly, we made several enhancements 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 aforementioned additions to the evaluation process resulted in slower overall performance, we believe this trade-off was justified by the enhanced strength of the AI. The fine-tuning of several hyperparameters, achieved through research and trial and error, also contributed to the AI's increased strength without any negative trade-offs. We attempted to implement mobility evaluation, but it proved to be excessively slow in our case, leading us to discard it. Similarly, the process of checking for the presence of two bishops was too slow relative to the benefit it provided.

The effectiveness of certain features in the evaluation largely hinged on our ability to properly vectorize them with numpy. Features that required calculations that could be efficiently executed in a single numpy call were the most successful. Conversely, features that required loop-based checks over the board were simply too slow compared to the rest.

3 Results

In this section, we will concentrate on the benchmarks and the more quantifiable outcomes of the project. The tests were conducted on a PC equipped with the following specifications:

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

To begin, let's examine Figure 1.

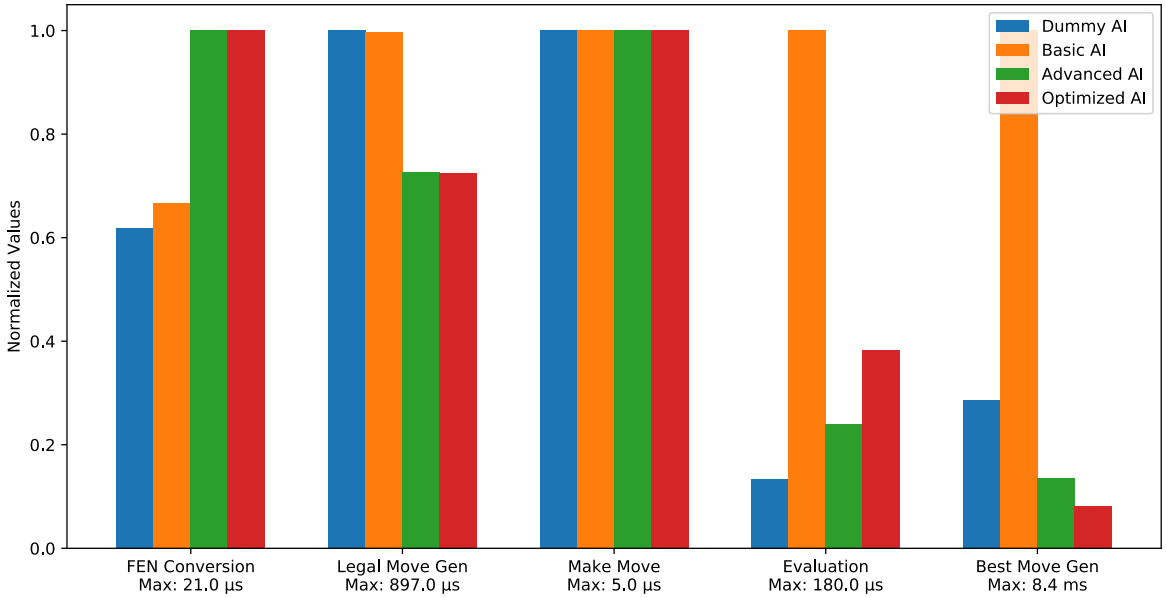

Figure 1: Benchmarks of the different categories across the AI versions.

The results are divided into five categories, with the first three determined by the chess backend and the latter two by the AI. It's important to note the significance of the legal move generation in the backend benchmarks. As the main bottleneck of the backend, any performance improvement in this area has a substantial impact on the overall performance of the AI. The reduction in time can be attributed to the refactoring process, which involved using lists of integers and minimizing boilerplate[18] code.

However, the most critical category is the final one, *Best Move Generation*. As the name implies, this measures the speed at which the AI calculates the best move, in this case at a depth of one. Despite a slower evaluation compared to the previous version, the optimized AI demonstrates the best overall time. This underscores the fact that the final performance and strength of an AI are determined

[18]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.

by multiple factors. In this instance, the number of nodes searched was significantly reduced by the implemented AI features, leading to an overall performance improvement.

This brings us to Figure 2, which illustrates the relationship between time and the number of nodes searched in relation to the depths. As shown in the lower plot, the number of nodes searched is significantly lower for the optimized AI compared to the advanced AI[19]. It's important to note that the apparent lower number of nodes searched by the dummy and basic AI is due to a bug in the early implementations of the AI related to move ordering, which was fixed in the third milestone. Therefore, the benchmarks for the dummy and basic AI in relation to depth should be interpreted with caution.

The plot also shows that the overall time is shorter for the optimized AI, regardless of depth. However, this is not always the case. For instance, if we had made poor decisions regarding the balance of exploration and exploitation in the MCTS, the performance could have actually been slower at higher depths. This highlights the importance of regular or atomic benchmarks, where a single element is changed to observe the system's response. Research also played a crucial role in fine-tuning the system.

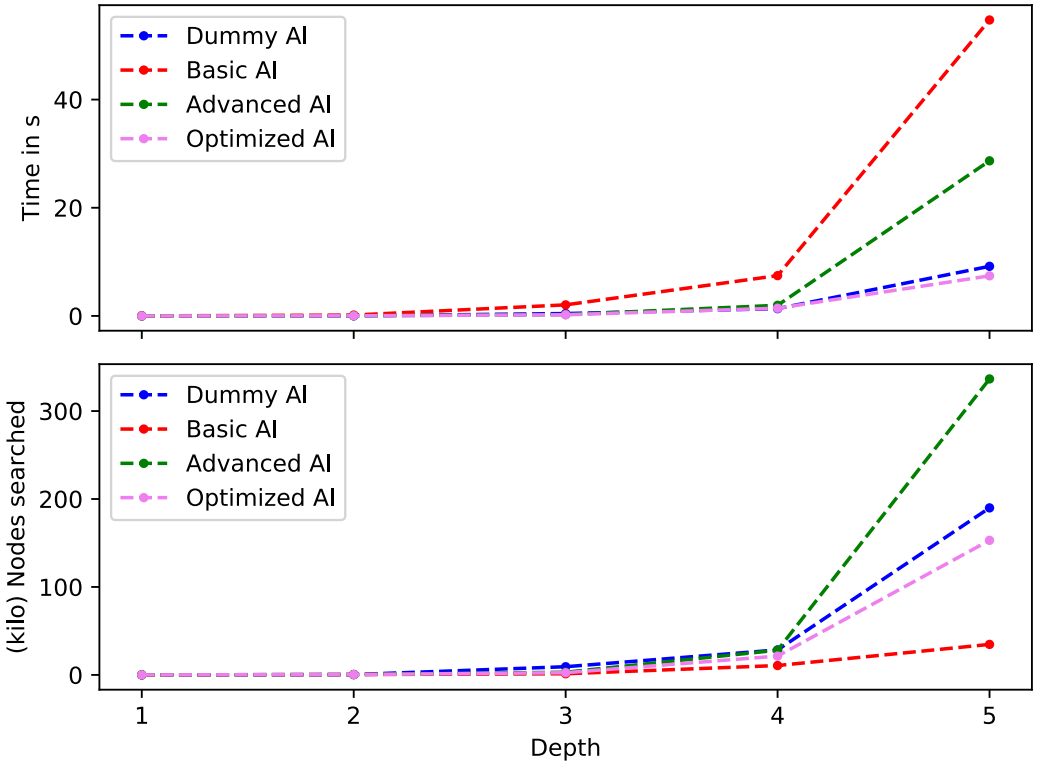

Figure 2: Benchmarks of different AI versions in respect to search depth. Note that the number of nodes searched is in thousands (kilo).

[19]The apparent lower number of nodes searched by the dummy and basic AI can be attributed to a bug in the early implementations of the AI related to move ordering. This issue was resolved in the third milestone. Therefore, the benchmarks for the dummy and basic AI in relation to depth should be interpreted with caution.

4 Issues faced

This section presents a compilation of challenges we encountered throughout the project. While they are not arranged in a specific order, they implicitly follow the sequence of milestones from one to four.

- Balancing the trade-off between the ease of prototyping in Python and its slower execution speed.
- Working with Object-Oriented Programming (OOP) can be challenging, particularly when the software design lacks clarity.
- The process of writing and debugging the chess engine was challenging. A more structured approach would have been preferable to trial and error.
- Early unit testing proved to be less beneficial than anticipated.
- The initial phase was the most difficult, as we were uncertain about where to begin.
- The indiscriminate use of numpy does not automatically enhance code performance and can, in fact, slow it down.
- Dealing with extensive functions with large logical statements.
- Implementing advanced AI algorithms, such as Principal Variation Search (PVS) and Monte Carlo Tree Search (MCTS), in a way that effectively improved the AI's performance.
- Dealing with the complexities of implementing a transposition table, including handling hash collisions.
- Ensuring the accuracy and efficiency of the AI's evaluation process, particularly when introducing new features.
- Managing the project's scope and complexity, especially when adding new features or making significant changes to the AI or backend.
- Overcoming the limitations of the initial AI framework and restructuring it to accommodate new features.
- Making multiple changes simultaneously, as opposed to implementing atomic changes.
- Committing to tasks that ultimately did not yield significant benefits.
- Coordinating team efforts, primarily due to the fact that the engine could effectively be worked on by only one person at a time.
- Navigating and resolving GitHub merge conflicts among team members, which required a clear understanding of the code changes made by each individual and effective communication to ensure that important updates were not inadvertently discarded.

Despite these challenges, most provided valuable learning opportunities. The lessons we gleaned from this project will be discussed in the subsequent chapter.

5 Lessons learned

The development of a chess AI presents a multitude of challenges, making it a fertile ground for learning. Failure is often the first step towards mastery, and this project was no exception. The following list encapsulates the key lessons we gleaned from this project, presented in no particular order:

- Benchmarks are invaluable, not only for tracking improvements across different code versions but also for comparing the impact of incremental changes. This became evident during our attempt to refactor the evaluation function using numpy.
- Effective team coordination is crucial, especially when working on complex projects like a chess AI. Clear communication and division of tasks can help prevent misunderstandings and ensure that everyone is on the same page.
- Resolving merge conflicts on GitHub can be challenging, but it's an essential skill for collaborative software development. It's important to understand the changes made by each team member and ensure that important updates are not accidentally discarded.
- It's important to be flexible and adaptable in your approach. If a particular method or tool isn't working as expected, don't be afraid to try something different.
- Research is a critical part of the development process. Understanding the underlying principles and algorithms can help you make more informed decisions and avoid potential pitfalls.
- Regularly reviewing and refactoring your code can lead to improvements in both performance and readability. It can also make it easier to add new features or make changes in the future.
- It's important to balance the pursuit of optimal performance with the practical limitations of your chosen programming language and tools. Sometimes, the most efficient solution may not be the most practical one.
- Implementing advanced features and algorithms can significantly improve the performance of your AI, but it's important to understand these features thoroughly to avoid introducing new issues or inefficiencies.
- The influence of background tasks on benchmark results should not be underestimated. Consistency in the testing environment is crucial for obtaining reliable results.
- Developing debugging and logging tools early in the project can significantly streamline the development process.
- Adopting a 'fail fast' approach in the project's early stages, such as using python-chess to create a simple AI, can provide a solid foundation for future development.
- Writing clear and concise git commits is crucial for both documentation and workflow. Atomic commits are particularly beneficial when restructuring complex software like the chess backend.
- Unit tests are invaluable when restructuring complex software.

- Before committing to a task that requires significant time and effort, it's important to assess whether the potential benefits justify the investment. Our attempt to implement transposition tables served as a lesson in this regard.
- Debugging hash tables can be more complex than it initially appears. Proficiency in using a debugger is a vital skill for software developers.
- Creating a chess AI is a more demanding task than it may initially seem. It involves not only writing the chess backend and ensuring the AI functions as intended, but also debugging both the AI and the backend, and regularly benchmarking to ensure progress is on track. This observation likely extends to the creation of AI in general, even without venturing into the realm of machine learning.

These points reflect the diverse range of technical and project management lessons we learned during the development of the chess AI.

6 Future Plans

As we reflect on the journey of developing our chess AI, we are excited about the potential directions for future work. The completion of this project marks not an end, but the beginning of a new phase of exploration and innovation.

We are interested in exploring the use of neural networks [1] for position evaluation. Traditional chess AIs, including ours, use handcrafted evaluation functions to assess the desirability of a given board position. However, these functions are inherently limited by our understanding of the game. Neural networks, on the other hand, have the potential to uncover subtle patterns and strategies that are beyond human comprehension. By training a neural network on a large dataset of high-level chess games, we could potentially develop a more nuanced and powerful evaluation function.

Another area for future work is the optimization of our AI's search algorithm. While our current implementation of the PVS/negamax algorithm has proven effective, there is always room for improvement. For instance, we could explore more sophisticated move ordering strategies to increase the efficiency of the search. We could also investigate the use of more advanced pruning techniques to cut off unproductive branches of the search tree.

Finally, we plan to continue refining the backend of our chess system. While the backend has served us well so far, we believe that there are opportunities for further optimization and enhancement. For example, we could implement support for additional chess variants, or develop a more user-friendly interface for interacting with the AI.

In conclusion, while we are proud of what we have achieved with our chess AI, we are even more excited about what the future holds. We look forward to continuing our journey.

7 Conclusion

The development of our King-of-the-Hill Python-based Chess AI has been an interesting journey through the landscape of software development. This project has allowed us to explore various aspects of software engineering, from the design and implementation of a complex system to the nuances of team collaboration in a development project.

Throughout this process, we have learned the importance of effective team coordination, the power of iterative development, and the value of regular benchmarking. We have also gained a deeper understanding of the complexities of software design and the role of efficient coding practices in enhancing performance.

Our Chess AI, while not the strongest due to the limitations of Python, has exceeded our initial expectations. We are proud of the software we have created and the knowledge we have gained in the process. The lessons learned from this project will undoubtedly serve us well in our future endeavors in software development.

Despite the challenges we faced, including debugging complex code, resolving merge conflicts on GitHub, and dealing with performance trade-offs, we persevered and emerged with a functional, efficient Chess AI. These experiences have underscored the importance of resilience, adaptability, and continuous learning in the field of software development.

In conclusion, this project has taught us a lot about collaboration, perseverance, and continuous learning. We look forward to applying the insights gained from this project to future projects and career paths.

References

[1] David, O. E., Netanyahu, N. S., & Wolf, L. *Deepchess: End-to-end deep neural network for automatic learning in chess.* In *Artificial Neural Networks and Machine Learning–ICANN 2016: 25th International Conference on Artificial Neural Networks, Barcelona, Spain, September 6-9, 2016, Proceedings, Part II 25*, pages 88–96, 2016. Springer.

[2] JE, O. *Why scientists are turning to Rust. Nature*, 588, 185, 2020.

[3] *Star Wars Chess AI Game.* `https://github.com/Jabezng2/Star-Wars-Chess-AI-Game`

Appendix

Plagiarism Clause

We hereby certify that we have written this project independently and have not used any sources and have not used any sources or aids other than those indicated. The passages of our work which are taken from other works in terms of wording or meaning we have in any case indicated the source as borrowing. as borrowed. The same applies mutatis mutandis to tables, maps and figures. This work has not been presented in this or a similar form in the context of another audit. in this or a similar form.

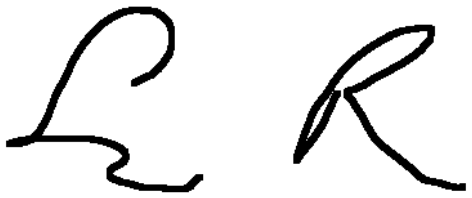

Figure 3: Signature

Class Diagram

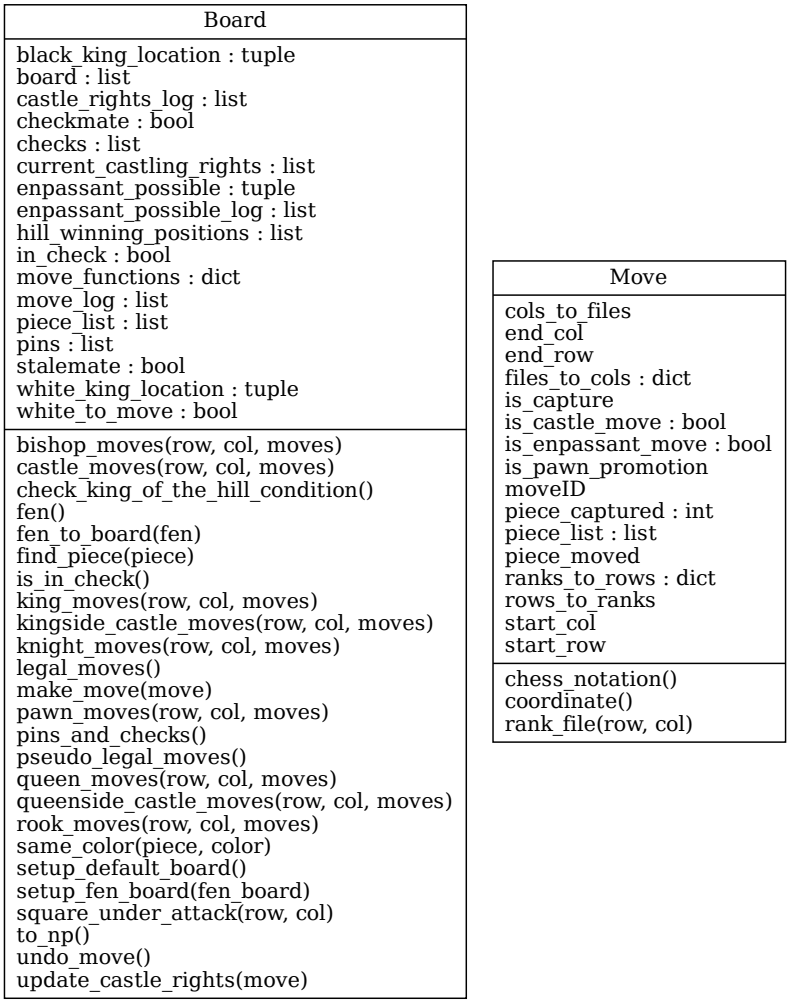

Figure 4: Class diagram of the chess backend in its latest state