Application of Alpha-Beta Pruning to Checkers

Moncef Bouhabel
moncef.bouhabel@u-paris.fr

Paul Chambaz
paul.chambaz@tutanota.com

Abstract

This project was conducted as part of the *Artificial Intelligence* course, taught by *Elise Bonzon*, within the third-year undergraduate curriculum at *Paris Cité University*. The primary objective of this project is to develop an artificial intelligence capable of solving two-player games by implementing the alpha-beta algorithm. The project aims to explore this algorithm in order to create an AI capable of competing against players of varying skill levels. Checkers was chosen as the game of focus due to its strategic complexity despite simple rules and the variety of evaluation methods available for a given position.

---

1. Description of the game and program

1.1. Motivation

In this project, we have chosen checkers as our game for investigation. This decision was influenced by multiple factors. First and foremost, checkers is relatively simple to implement, enabling us to concentrate on the essential aspects of our algorithm rather than the game's complexity. Nevertheless, despite its apparent simplicity, checkers possesses considerable strategic depth. Although it may not be on par with the intricacies of classic games such as chess or go, checkers boasts a potential move count of approximately 5×10^{20}. It was not until 1994 that a checkers algorithm secured the world champion title for this game, and a comprehensive solution was not achieved until 2007. To this day, checkers is the most complex game that has been fully solved.

Secondly, in contrast to other projects considered, checkers emerged as a fertile domain for the diverse methods employed in evaluation functions. When observing an ongoing game, it appears relatively straightforward to assess the strength of a player's position. While this type of judgment is inadequate for analyzing high-level play, it is probable that the evaluation function required for the alpha-beta pruning algorithm permits a more inventive and effective solution.

Lastly, during our research on this topic, we uncovered the importance of checkers in the history of artificial intelligence. As early as 1952, a preliminary algorithm was developed. In 1959, the first paper to use the term *Machine Learning* was titled *"Some Studies in Machine Learning Using the Game of Checkers"* [1]. Furthermore, with the *Chinook* [2] program, checkers has been entirely solved by artificial intelligence since 2007. This more modest project will focus exclusively on the application of alpha-beta pruning.

1.2. Rules of checkers

Checkers is played on an eight-by-eight square board, comprising two types of squares: white and black. Throughout the game, pieces can only move on the black squares. The game involves two players, one controlling the white pieces and the other the black pieces.

At the beginning of the game, each player recieves twelve pawns that are placed on the black squares within the first three rows. Only the two middle rows are empty at the game's commencement, and the white player initiates play.

In checkers, a piece may only move diagonally and proceed to an unoccupied black square. If

a piece occupies a square, it is not possible to move to that square. Once a piece has moved, it becomes the other player's turn. A pawn can only advance forward.

A pawn can capture an ennemy piece by *jumping* over it and landing on the diagonal square beyond the pawn. If this square is occupied, the pawn cannot move there and is unable to capture the enemy piece. If it can, the enemy piece is removed from the board, rendering it unusable by the opponent. Moreover, a pawn must capture if it can; if multiple options are available, the player can choose which pawn to capture. Additionally, a pawn may execute a chain capture: if, after capturing, a pawn can capture again immediately, it must do so. The opponent's turn is delayed until the pawn can no longer capture.

A pawn can become a king when it reaches the opponent's back row. Upon promotion, the pawn transforms into a king, which is a highly powerful piece in the game of checkers. First, it can move in all directions. Second, it can advance as many unoccupied squares as desired. Lastly, after capturing, it can land on any available square beyond the captured piece. This makes a king an extremely agile piece capable of easily capturing multiple pawns when left unprotected. After promotion, the turn passes to the opponent, even if the kind might be able to capture additional pieces.

The game can end for two reasons : if a player has no pieces remaining or no valid moves, they lose. A draw can also occur if the same sequence happens three times in a row, in an ABABAB pattern, or if no piece has captured in the last 32 turns.

1.3. Program description

For this project, we opted to utilize the *Lisp* programming language, as it was the original language developed for artificial intelligence. Lisp's grammar is often characterized as *meta-programmable*, and a *functional language*, we believed it would be conducive to implementing mathematical concepts. Moreover, the *Lisp* dialect we employed, *Common Lisp*, along with the graphical library *SDL2* for displaying the game, are highly performant and can facilitate extensive search depth. The program is organized into seperate sub-files for ease of maintenance:

- `main.lisp`: Contains the main loop of the program, setting up SDL2 and enabling the human versus machine gameplay.
- `logic.lisp`: Contains the complete set of rules for the game of checkers. This file implements state, action, result, actions and terminal-test.
- `intelligence.lisp`: Contains all functions related to artificial intelligence, including minimax and utility, as well as the functions used to create the endgame database.
- `training.lisp`: Contains all functions utilized for training the program.
- `interface.lisp`: Contains the code responsible for displaying the program to the user.
- `constants.lisp`: Contains various values used across the program.
- `package.lisp`: Contains information about the package.

2. Artificial Intelligence Description

2.1. Chinook

Before delving into the description of our program, it is essential to establish the state of the art. The primary program that achieved what we are attempting is called *Chinook* [2]. This program is renowned for defeating the world champion in 1994. However, it is worth noting that it accomplished this without beating the legendary champion *Marion Tinsley*, who remained undefeated for over 40 years. Unfortunately, Tinsely passed away in 1994, never have been defeated. In 2007, researchers from the *University of Alberta* unveiled the results of their work: checkers was now a conquered game, and the outcome of perfect play was a draw. Our program is considerably simpler, but there are still insights we can glean from Chinook.

Accustomed to the state of modern artificial intelligence with deep neural networks, such as *Deep Blue*, which defeated *Garry Kasparov* in 1997, we were intrigued to discover that *Chinook* employs techniques strikingly similar to our program. Chinook uses an opening database, crafter from the analysis of expert players. It then applies minimax for the middle game, with the evaluation function designed by an expert player to achieve optimal performance. Lastly, the researchers develoed an engame database, enabling *Chinook* to determine, for every position, whether it will result in a win, loss or draw, and the appropriate move to execute.

2.2. Program PEAS

Before initiating our program, it is crucial to define its PEAS (Performance measure, Environment, Actuators and Sensors). Our agent type is a checkers program, with the performance measure being the AI's strength against human players. The environment encompasses the checkers board and the game's rules. The actuators is straighforward : the AI communicates the move it intends to make, and a human operator can then execute the corresponding move in a real game. If the game is entirly computer-based, the AI has the capacity to play the move it has chosen. As the AI lacks real-world sensors, the game's state must be described within the computer, but once this is accomplished, the AI can utilize its internal representation of the state.

This environment is **fully observable**, as the AI can, in theory, explore all possible states. It is a **single-agent** environment since the AI competes against a human opponent. The environment is **deterministic**, as an action performed on a given state will consistently result in the same state. It is **episodic**, as there is a clear progression to the game, with actions taken in one turn affect subsequent turns. The environment is static, as it does not change unless an action is decided upon and executed by the AI or the player during their respective turns. Lastly, since the game is turn-based, it is **discrete**.

2.3. Formal definition of a state

We will begin by providing a formal definition of a state, which much encapsulate all the necessary information required to make informed decisions about future gameplay. As this state will be passed to the search tree, it is preferable to have a simpler state rather than a more complex one. In this project, we have chosen to represent the state in the following manner:

- The list of the values of the squares on the board, where `0` means empty, `1` means a white pawn, `2` means a black pawn, `3` means a white king and `4` means a black king. It should be noted that both black and white pieces are stored in this for convenience, though this incurs a memory cost.
- The number of the player who must act, with `0` for white and `1` for black.
- The number of the square where a player is required to capture an opponent's piece. As stated in the rules, sometimes a player must continue capturing pieces. In such cases, the number of its position will be stored; if no piece is required to capture another, then `-1` will be stored.
- The list of the last six actions performed. This list is used to determine games that end in draws.
- The countdown until the game is declared a draw, which resets to 32 every time a piece captures another.

In a more formal manner, we define :

$$n \in \mathbb{N}, S_n \in \mathbb{S}$$

Where n is the turn of the game, S_n is the state, and $\mathbb{E}$ is the space of all possible states.

2.4. Formal definition of an action

An action represents a single movement by a player, meaning that chained actions are depicted as multiple actions. The approach allows the search tree to choose the best combination of actions. An action is composed of:

- The square from which the action starts.
- The square to which the action moves.

- The index of the player who will play after the current turn is completed, with `0` for white and `1` and black.
- The index of the piece that has been eaten, `-1` when none.

In a more formal manner, we define :

$$a \in \mathbb{A}$$

$$u : \mathbb{A} \times \mathbb{S} \to \mathbb{S}, (a, S_n) \mapsto S_{n+1}$$

$$v : \mathbb{S} \to \mathbb{A}^m, S_n \mapsto \{a_1,..,a_m\}$$

Where $\mathbb{A}$ represents the space of actions. u is the result function, a function that takes an action and a state and returns the new state. Finally, v is the actions function, which takes a state and returns the list of legal actions. We will use the functions for searching. The new list of states to search can be expressed concisely. Moreover, this relation is recursive.

$$\forall E_n, \forall a \in v(E_n), u(a, E_n)$$

2.5. Evaluation Function

The evaluation function is crucial for creating a robust program. If we do not have a clear understanding of a state's quality, searching through thousands of them will not be very helpful. In this program, we define the evaluation function as follows:

$$f : \mathbb{E} \to \mathbb{R}, E_n \mapsto f(E_n)$$

$$f(E_n) = w_1 \times f_1(E_n) + ... + w_p \times f_p(E_n)$$

$$W = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_p \end{pmatrix}$$

Our evaluation function is essentially a composition of p sub-functions. All these sub-functions are sub-evaluation functions that each measure an atomic observation of the state. Once all these observations are made, we combine them with weights stored in the vector W. The various sub-functions are unbiased, meaning they can return observations that might be considerd good, such as the number of pieces the player has, or they can return observations that might be considered bad, such as the number of pieces the opponent has. The weights role is to determine which information is good and which is bad, and to assign weights to those judgments, hence the name. Later on, we will address the questions of determining the best or at least very strong set of weights, as well as identifying which observations are useful, good, or bad, and which are not really significant factors.

This is the list of all evaluation function, seperated in categories:

- **Simple counts:**
  - The number of pawns for the player.
  - The number of pawns for the opponent.
  - The number of checkers for the player.
  - The number of checkers for the opponent.
- **Mobility counts:**
  - The mobility for the player.
  - The mobility for the opponent.
  - The eating mobility for the player.
  - The eating mobility for the opponent.
- **Positional counts:**
  - The piece count in the center for the player.
  - The piece count in the center for the opponent.
  - The piece count in the front for the player.
  - The piece count in the front for the opponent.
  - The piece count in the back for the player.
  - The piece count in the back for the opponent.
  - The piece count in the left for the player.
  - The piece count in the left for the opponent.
  - The piece count in the right for the player.
  - The piece count in the right for the opponent.
  - The piece count in the sides for the player.
  - The piece count in the sides for the opponent.
  - The piece count in the diagonal for the player.

- The piece count in the diagonal for the opponent.

2.6. Optimization of Minimax Algorithm in Checkers

The minimax algorithm is a decision-making strategy widely employed in two-player games, including checkers, to determine the optimal move. While the alpha-beta pruning technique significantly enhances the effiency of the minimax algorithm, additional optimization methods can be employed to improve its speed even further. In this paper, we discuss the implementation of an iterative deepening algorithm and move ordering techniques to optimize the performance of the minimax algorithm in checkers.

The iterative deepening algorithm is a depth-first strategy that iteratively explores states at increasing depths. This method offers three key advantages. By setting a time limit, the search process can be terminated if the time budget is exceeded, ensuring consistent playing speed and an improved user experience. If a guaranteed win or loss is found, there is no need to search deeper or for other move combinations. The iterative deepening algorithm allows for the collection of data on useful moves, resulting in a substantial speedup when combined with move ordering techniques.

Move ordering techniques help to explore promising moves first, thereby improving the efficiency of the minimax algorithm with alpha-beta pruning. We discuss three move ordering techniques.

With piece move ordering, kings are prioritized over pawns, and forward pieces are prioritized over backward ones. This simple optimization frequently helps identify strong moves on the first attempt.

A frequency table involves creating a hashmap that records the number of times each move hsa cut off minimax in alpha-beta pruning. Moves are then sorted by the frequency in the hashmap, enabling the exploration of more promising moves first. By ranking moves according to their pruning frequency, the search process is guided towards branches that are more likely to yield optimal results. The frequency table allows the algorithm to learn from previous search iterations and capitalize on this information to make more informed decisions, ultimately enhancing the efficiency of the minmax search.

Killer moves heuristics is another strong method. Building upon the frequency table, killer moves heuristics store moves that cut off minimax for each depth and player in the search tree. These moves are considered highly promising and are prioritized in the search process. The underlying result in a cutoff at a specific depth in the search tree, it is likely to cause a cutoff at the same dpeth in other parts of the tree. By identifying and prioritzing such killer moves are store for each depth of the search tree, ensuring that the most promising moves are consistently prioritized throughout the search.

The combination of these optimization techniques has proven to yield significant improvements in the performance of the minimax algorithm for checkers.

2.7. Construction of an opening database

Although alpha-beta pruning is highly effective at identifying strong moves, it does have certain limitations. One such limitation is its inability to dicern the superiority of a move if the advantage is not immediately apparent. This can lead to suboptimal performance in games with less obvious evaluation criteria. Another consequence of this limitation is that the impact of small action taken at the beginning of the game may not be recognized until much later in the game, sometimes as many as 50 turns later.

To address this issue, we have implemented an opening database for our checkers program. The opening database is a simple hashtable that stores game states that may arise early in the game as keys, with the corresponding moves to play as values. Before the program begins the iterative minimax search, it checks the opening

database and plays a move from it if available. This approach helps guide the program to make stronger opening moves that may have long-term consequences not easily detected by the minimax algorithm with alpha-beta pruning.

In the absense of access to expert checkers players or a large dataset of expert games, we built our opening database using Chinook, which is widely considered the strongest checkers player. By comparing the initial moves made by our program to those played by Chinook, we observed differences that suggest the opening database has a positive impact on the overall performance of our program. Utilizing the opening database should lead to more accurate play and improved results on average.

2.8. Construction of an endgame database

Another limiation of the minimax algorithm is that the search must eventaully come to an end, either due to reaching a specified depth or exceeding the allocated time. When this occurs, the algorithm returns an evaluation of the game state. However, when there are only a few pieces remaining on the board, it is possible to precompute the game's outcome and provide the corresponding outcome to the algorithm. This enables the program to return the most suitable action without conducting the search if the game state has already been determined.

This approach is known as an endgame database and was used effectively by Chinook to solve checkers. Chinook's success hinged on the creation of a large endgame database, which encompasses all possible states with ten pieces remaining on the board. Since then, the checkers endgame database has expanded to included up to twelve pieces. To create our own endgame database, we had to enumerate all possible states, eliminate duplicates, and compute the final result for each state. This process can be conducted iteratively, allowing the addition of results fo reach piece layer to the algorithm for faster performance. In this project, we successfully built a two-piece database, which significanly accelerates and improves the precision of endgame scenarios. If the minimax algorithm reaches a state where a guaranteed win has been precomputed, the AI is certain never to lose, which is a powerful outcome. To make this process go faster, we implemented parallelism.

Out initial goal was to build a four-piece endgame database, but memory constaints prevented us from achieving this within the given time. Additionally, the complexity of certain positions meant that we could not reach the end state for all of the states in the two-piece database. Out of approximately 35000 positions considered, only 32000 were completely solved in our computations. Desplite these limitations, loading the resulting database at runtime provides an additional twelve levels of depth for endgame positions where a computed state is reached. Considering that we often achieve a depth of ten during endgame scenarios, this translates to a function depth of over 20, greatly enhancing the performance of our checkers program.

2.9. Tournament between AI

In the context of our checkers program, while the opening and endgame states are addressed through specific databases, the middle game remains a crucial component in determining the outcome. The middle game often decides the majority of checkers matches, making it essential to devise the most effective evaluation function possible. Although increasing search depth can improve performance, the key factor lies in accurately evaluating a given state. As mentioned earlier, we can simplify our evaluation into a vector W of weights. To find the best evaluation function, we aim to identify W that maximiezes the number of games won, transforming the problem into an optimization task where we search a space. To tackle this issue, we employed a genetic algorithm.

The process begins by generating an initial population of 40 vectors, which we refer to as AIs

and imagine them competing in a tournament. The starting values can either be random or pretrained, and in this project, we opted for biasing them in hopes of achieving faster or better solutions. This decision was also influenced by the high dimensionality of the vector space, which chould lead to chaotic results. With the initial population established, the AIs are pitted against one another in matches.

After each match, we update the AI's ELO rating and im to pair AIs with opponent of similar ratings for improved accuracy. Once the matches are completed, we use the ELO ratings to generate a new population. AIs with higher ELO rating are more likely to be included in the next generation.

The new generation is created through a three-step process : selecting two parents based on their ELO ratings, mixing their weights around a point determined by their ELO ratio and creating a new AI from the concatenated parts of each parent. To ensure diversity in the new generation, we make slight adjustments to each of the resulting weights. This approach is heavily inspired by the biological process of DNA recombination. Though articial selection imposed by the fitness constraint of the ELO ratings, our program progressively generates stronger AIs.

We applied this method for a total of 150 generations, with each generation consisting of 40 AIs that played 5 games each. This amounted to a total of 30000 games in pursuit of the best possible checkers player.

2.10. Writing AI for humans

While developing a strong AI is a noteworthy achievement, it may not necessarily result in an engaging experience for human players. An AI that defeats most players, including experts, like Chinook, can lack a human touch and be less enjoyable to play against. To address this issue, we implemented different difficulty levels to cater to various player skill levels and ensure a more enjoyable gaming experience. The three basic difficulty levels are as follows:

- **Easy:** This AI has a depth limit of 4 and uses manually set weights. It provides a suitable challenge for beginners or casual players who want a more relaxed game.
- **Medium:** For this level, we used the weights from our hard AI to generate a new population of AIs. We mutated each weight slightly using a Gaussian distribution, then conducted a final tournament for the AIs to determine the rankings. To find the AI that best matched a medium skill level, we utilized human-assisted binary search, having family members play against the AIs. The resulting AI was chosen as the medium difficulty AI, with a search time of 3 seconds. This level is suitable for intermediate players who want a more challenging but still manageable opponent.
- **Hard:** The hard AI incorporates all the strategies and techniques we've developed throughout the project. It utilizes the best AI from our training process, the opening and endgame databases, and searches for 10 seconds before delivering a move. This level is designed for advanced players who want to face a formidable opponent and test their skills to the fullest extent.

By incorporating these difficulty levels, our checkers program provides a more balanced and enjoyable experience for human players with varying skill levels, ensuring that the game remains engaging and entertaining.

3. Conclusion

We take great pride in the results of this project and appreciate the wealth of knowledge gained troughout its development. The code has been released under a free software license, enabling others to learn from it and attempt to defeat our hard AI. Key challenges included optimizing the algorithm by thoroughly understanding its inner workings, building the endgame database by iterating through all valid game states, and optaining the best AI through the time-consum-

ing process of simulating games with the genetic algorithm.

Future improvements could involve more extensive training or the incorporation of advanced techniques like neural networks to enhance the evaluation function. Rewriting the program in a lower-level language, such as C, may also yield deeper search capabilities. Furthermore, we did not use parallelism to train the genetic algorithm, or for parallel seaching which could result in improved depth.

We would like to express our gratitude to Elise Bonzon for offering the course and providing valuable guidance throughout the project, as well as our families for their support.

Bibliography

[1] A. L. Samuel, "Some studies in machine learning using the game of checkers," 1959.

[2] J. Schaeffer, "Chinook the world man-machine checkers champion," 1996.