Shogi AI

Hamza Benkhalifa Shreyas Chaturvedi Daniel J Barton

December 16, 2021

Abstract

The problem we create a solution for is an AI for Shogi. Shogi is a substantially more complex variant of Chess. We created a rudimentary AI to play Shogi. Alongside this AI, we developed programs to handle Tsume, a Shogi miniature problem to checkmate the king of the opposite player. In general, there are two approaches to develop an AI for games like Chess. These two approaches are machine learning and traditional artificial intelligence. We engage in a detailed discussion about the differences between the machine learning and traditional AI approaches to Chess-like games, including discussion on concrete examples of these two approaches being used within the field. Particularly we discuss the machine learning AI, AlphaZero, alongside with a couple of concrete traditional engines, YSS, Elmo, and Bonanza. We reflect some of the technical details of these traditional engines, and how we apply their concepts within our AI to produce an AI of similar nature.

Our approach was to use traditional artificial intelligence with α-β pruning alongside with selective deepening to create a powerful AI. To increase the accuracy of our AI, we created evaluator functions to interpret a board position in a quantifiable manner that reflects the semantical understanding of a board position which a human player has.

In order to analyze the performance of our AI, we created a record of our AI playing Shogi against different variants of agents. Particularly, we had our AI play 100 games against strictly random moves to ensure that our AI is decisively better than just selecting random moves. We then played the AI against an intermediately skilled human player in order to assess whether or not our AI was viable within the realm of competitive play.

What we found from playing the AI v.s. an intermediate-level Shogi player was that our AI was lacking in the capability to outperform humans at the intermediate skill level, as 20/20 games for each version of the AI were decisively lost by the AI. However, we discovered features which reinforce and provide proof of concept for archetypes within traditional AI for game-playing. Primarily being the power that selective deepening offers to games of high game-tree complexity. Moreover, with the selective deepening method of generating moves geared specifically toward solving Tsume puzzles, we were able to apply the concept of selective deepening, which directly extends to the game of Shogi in its entirety.

1 Problem Description

We are attempting to solve the problem of creating an AI for Shogi, also known as Japanese Chess. Shogi is very similar to traditional Chess, however the complexity of the game is notably higher. One intuitive way of determining the difference in complexity between Chess and Shogi is through the average number of moves per game. Chess has an average game length of 80 moves, whereas Shogi has an average game length of 110 moves [8]. Moreover, what distinguishes the two games the most is that captured pieces can be conditionally placed back onto the board. This "revivability" of pieces present in Shogi inherently increases the complexity of the game relative to Chess, since as soon as a piece is in hand, it can be placed almost anywhere on the 9x9 board. Further rules regarding the play of Shogi can be found in [5]. This document on how to play Shogi was also used to advise decision-weighting of the Shogi AI during game situations.

Creating a comprehensive AI that is competent relative to intermediate Shogi players requires significant implementation of Shogi-specific knowledge, which presents the challenge of spending a considerable amount of deciphering different conditional situations in Shogi when attempting to implement an algorithm targeted towards a certain aspect of the game (e.g: finding potent positionings to attack the enemy king, fortifying a defensive structure around the player's king, etc.). In addition, the extreme branch complexity of Shogi relative to that of Chess, paired with real-world time limits and the modest performance of currently known Shogi algorithms render a Shogi engine unable to realistically perceive all moves present to the player or, more importantly, to be able to perceive the relative potency of any given move beyond a certain depth.

Alongside creating an AI for the game of Shogi as a whole, we specifically sought out creating an AI that has good performance on solving Tsumes, which are effectively checkmating puzzles. By demonstrating the efficacy of an AI specifically targeted at checkmating patterns, we provide a strong foundation for an AI to convert winning positions into checkmates in the endgame. A Tsume problem provides an environment where we can implement and verify the quality of specialized features localized to the environment of a Tsume problem. Therefrom, we can extend said features into the game of Shogi in general.

2 Background

2.1 Description and Overview of Methodologies and Approaches

The game of Chess and its variants are some of the most widely-studied domains in Artificial Intelligence. Many attempts have been made to solve Chess and its variants, Shogi included. There have been in general, two types of approaches to creating AI for Chess its variants. These two methods being: (1) Machine Learning, and (2) Traditional artificial intelligence. Within Shogi, it has been apparent that machine learning has resulted in the strongest of opponents, with the primary competitor being alpha-zero. Alpha-zero convincingly outperformed the top traditional AI for Shogi – Elmo – winning 90 games, 2 ties, and losing 8 out of 100 games [13]. Traditional engines like Elmo use state of the art, domain specific, editions of α-β pruning.

2.1.1 Machine Learning and Alpha-Zero

Alpha Zero uses *tabula rasa* reinforcement learning to learn from games of self-play[13]. The significance of this is that Alpha Zero is able to learn without specific evaluation functions and move order heuristics [13]. This means that with this form of general reinforcement learning, Alpha Zero has been shown to be utilized for three different games that are of similar nature to Shogi. These three games being Chess, Shogi, and Go. Alpha Zero utilizes a general purpose Monte-Carlo tree search algorithm for how it produces moves [13]. This is significantly different compared to traditional AI which uses hand crafted and tuned α-β pruning. For significant portions of history within the AI community, there was a widely held belief that α-β pruning dominated any other search within these domains[1]. With Alpha Zero's indisputable outperformance of Elmo, this widely held claim is thrown into question.

2.1.2 Traditional AI: Elmo

Elmo is one of the traditional Shogi engines which utilizes α-β pruning. However, with Elmo's engine, there are handcrafted features for it to have a strong performance within Shogi. Because of this utilization of alpha-beta pruning, inherently Elmo needs to evaluate a considerable number of positions. Elmo evaluates 35 million positions per second[13], which is a significant quantity of positions to compute.

2.1.3 Traditional AI: YSS

Another traditional engine for Shogi is YSS. Similar to Elmo, it utilizes α-β pruning. However, we have some more insight onto how YSS expands nodes to explore with α-β pruning. Particularly, at depth of at least 2, YSS generates the following types of moves:

> "(1) capture the opponent piece that just moved, (2) capture an undefended piece, (3) promote a piece, (4) give a check that does not sacrifice material, and (5) move an attacked piece with highest value. At depth2 or higher we find: (6) defend against a strong threat, (7) attack material, (8) discovered check, (9) attack King from the front, (10) discovered attack, (11)

attack pinned pieces,and (12) drops of Bishop and Rook in the camp of the opponent."[7]

What is surmisable from this then, is that YSS utilizes these generator functions to allow for selective deepening for its α-β pruning. This selective deepening allows for YSS to consider more plausible moves that are beneficial. Furthermore, this shows how YSS also is handcrafted for Shogi based on human knowledge and strategy.

2.2 Differences between Alpha-Beta Pruning and Monte Carlo Tree Search

2.2.1 Alpha-Beta Pruning

Alpha-beta pruning is an adaptation of the minimax algorithm that makes search more efficient by eliminating certain nodes from exploration. The algorithm was an amplification of a previous modification made to the minimax algorithm called the "branch-and-bound" algorithm. The "branch-and-bound" algorithm improves on the rudimentary minimax algorithm by ignoring moves that are incapable of leading to a better outcome than currently known moves. In game terminology, a move to p2 can be "refuted" relative to the alternative move p1 if the opposing player can make a reply to p2 that is at least as good as their best reply to p1. Once a move has been refuted, we need not search for the best possible refutation. The alpha-beta pruning algorithm further improves on the "branch-and-bound" technique by introducing proper upper and lower bounds. The algorithm contains three parameters: p, alpha, and beta. Alpha is used to represent the lowest value that can be achieved by the player trying to maximize the value and beta is used to represent the highest value that can be achieved by the player trying to minimize the value.

In functional terms, alpha-beta pruning can be represented by F2 given p, α, β, where alpha $<$ beta such that:

$$
\begin{array}{ll}
F2(p,\alpha,\beta) \leq \alpha & \text{if } F(p) \leq \alpha, \\
F2(p,\alpha,\beta) - F(p) & \text{if } \alpha < F(p) < \beta, \\
F2(p,\alpha,\beta) \geq \beta & \text{if } F(p) \geq \beta.
\end{array}
$$

Thus, whenever the maximum score that the minimizing player (i.e. the "beta" player) is guaranteed becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is guaranteed (i.e. $\beta < \alpha$), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.

One drawback of alpha-beta pruning is that a computer playing a complex game will rarely be able to reach all possibilities until truly terminal positions are reached. However, the algorithm can still be used if the generator function is modified to terminate positions beyond a certain depth [10].

2.2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is based on two ideas: 1. The random simulation of a game will be able to approximate the value of a move 2. The value acquired can be used to orient towards a best-first search

The MCTS algorithm works by recursively exploring a partially formed game tree and building nodes deeper into the tree depending on the results of its exploration. Every single node in the tree is a representation of a single state within the search domain. As the tree is built, the approximation of the value of the moves that was were expanded upon by the algorithm are known clearer and the algorithm is able to choose the optimal move. The goal of MCTS is to get an approximate evaluation of the real game value of all actions that could be taken from the player's current situation/position in the game. Specifically, the algorithm requires the iterative construction of a search tree until a certain point at which a specified time or memory restriction is met. There are four phases in the algorithm: selection, expansion, simulation, and backpropagation. In selection, the algorithm recursively visits every node of the initial partial tree starting at the head node until it reaches the expandable node of the highest priority, which is when it reaches the highest valued part of the tree that has no children. Then, the algorithm expands this prioritized expandable node and runs a simulation from the created children nodes to determine the outcome and value of the nodes. The final step for the recursive subroutine is to backpropagate the node, which is to store the results of the simulation step into the node.

Since MCTS is domain-independent (i.e: doesn't require any domain-specific knowledge), it is very easy to adapt to any game that can be modelled using a game tree. Additionally, MCTS offers significant performance benefits over minimax-based algorithms such as Alpha-Beta pruning for games that do not have a very reliable heuristic function [2]. Another significant difference between a game evaluation done through MCTS and Alpha-Beta pruning lies in the fact that Alpha-Beta pruning must explore most of the search space while MCTS doesn't need to. Thus, in order to have an efficient Alpha-Beta pruning algorithm relative to the performance of a MCTS algorithm, it is imperative to integrate plausible move generators to further reduce the amount of game states visited by the $\alpha - \beta$ pruning algorithm.

2.3 Technical Challenges

2.3.1 Complexity Relations Between Shogi and European Chess

Shogi, as mentioned earlier, is analogous to European Chess. However, complexity-wise, they are very different from one another. For a computer to be able to solve a game, the computer would need to know every single possible board state that can be constructed from the initial position. Within Chess, the number of possible combinations is 10^{123}, whereas within Shogi, the number of possible combinations is 10^{226} based on assumptions made[12]. There are a few key reasons why Shogi has a significantly higher amount of possible combinations than Chess. This clearly tells us that the policies we create our AIs to abide by only reaches a fraction of the possible moves there are within these types of games, making the empirical value of these policies of the utmost importance. The most apparent difference between Chess and Shogi is in the piece reusability offered by Shogi. Unlike Chess, where captured pieces are eliminated from the game, Shogi pieces are able to be reclaimed during a player's move at the exchange of them moving their piece for the turn. Needless to say, the ability to reuse pieces adds a significant amount of options in Shogi that don't exist in Chess. In terms of game programming, the ability to reuse pieces makes Shogi a diverging game, where

the number of options for a player increases later in the game, instead of a converging game like Chess where player options are narrowed down as the game progresses.

2.3.2 Mimicking Human Behavior and Strategy Within Traditional AI

Furthermore, human strategy within these board games is hard to mimic. In order for an AI to have any human-like strategy while employing alpha-beta pruning, there are two primary ways this can be attempted. Firstly, through the evaluator, one could create weights for statically evaluable features of the board. Adjusting these weights would change the behavior of the AI immensely, with the claim that the the feature that weight is attached to has a higher priority than others (if scaled properly). However, another approach would be to create generators for moves of certain types, and instead of having the alpha-beta pruning algorithm operate on all possible moves, instead look at a selective subset of moves. This is coined as selective deepening, and allows for deeper analysis at the cost of not exploring every possible move. This is something that is employed by YSS as mentioned in 2.1.3 Traditional AI: YSS. Almost certainly, every Shogi AI that uses α-β pruning utilizes some form of selective deepening, as there are a lot of obviously non-productive moves that can be played. To exemplify how this second approach of selective deepening using generator functions could change the behavior entirely for the AI, consider the example of adding a generator function for 3 types of moves to be explored. 1) moves that check the opponent king. 2) moves that capture opponent pieces. 3) moves that recapture a piece that was just taken. 4) waiting moves (a move just for the sake of playing a move that doesn't lose). Obviously from only looking at these three generator functions, we have quite the aggressive behavior for our AI, regardless of strength.

2.3.3 Is it possible to solve Chess and Shogi using AI

Can AI ever be used to solve strategy board games as complex as Chess? Mathematically, yes, it is possible. Chess is a finite game. It has a limited number of moves. Therefore it has a limited number of move sequences. An AI That examines every single move sequence will surely be able to solve Chess. Realistically, however, it is unlikely that we'll reach the level of computation prowess to solve such a game. The main reason why games like Chess remain unsolved is simple. There are just too many moves. Chess's complexity is 10^{123}. For reference, there are only about 10^{78} to 10^{82} atoms in the absorbable universe [9]. There are more possible move sequences in Chess than there are atoms in the observable universe. The next question is would an AI need to investigate all possible move sequences to solve Chess. The answer is no. Many algorithms, some of which discussed in this paper, have proven to recognize that a move sequence is not efficient early on and ignore it. Doing so reduces the complexity of solving Chess, depending on the algorithm used. With this in mind, Chess remains a game that is far away from being solved. However, Chess engines continuously keep improving at a very rapid paste. In the future, AI technologies might be able to solve Chess[11]. If Chess is solved, it might be possible to solve Shogi too.

2.3.4 Bonanza and its application to Shogi

As mentioned previously in the document, solving Shogi has been attempted on numerous occasions. However, none of the search engines that developed were able to solve Shogi. Early on, all Shogi search engines were incompetent and could only compete at an amateur level. All of this changed when a Shogi-engine, Bonanza, was introduced in 2005. Kunihito Hoki, the engineer behind Bonanza, developed a flexible evaluation function that can handle Shogi. What makes this evaluation function so efficient is that it factorizes the position of the Shogi pieces into the position of any three pieces including and two kings. The three-piece combination can capture important features of the Shogi board. Additionally, Hoki added restrictions to Bonanza to reduce the effective parameters to optimize the search process. This gave Bonanza an advantage over other Shogi engines. In its essence, Bonanza is an approximate terminal-value function that relies on backward induction on a truncated game tree. Bonanza is an empirical model of professional Shogi players that can predict human experts' action, using logic-style regression. In 2006 and 2013 Bonanza won the world championship in computer Shogi. In 2017, Bonanza even managed to beat a Meijin, the most prestigious Shogi title[6]. This is to say that even though Shogi remains unsolved, AI has become extremely efficient at handling Shogi's complexity and even managed to beat the best Shogi players in the world.

2.4 Conclusions and Implementation Specifics

In order to support higher performance for an Alpha-Beta pruning algorithm, it is necessary to implement plausible move generators. Due to the $\alpha-\beta$ pruning algorithm's vast exploration of the game tree, it is vital to minimize the areas it explores since it is evident from the research on MCTS and plausible move generator functions that evaluating plausible moves is an effective strategy for obtaining faster moves that are still optimal in a game scenario. Particularly, we will explore plausible move generators for capturing the opponent piece that just moved, responding to an opponent's attacking moves, as well as generating our own offensive and defensive moves [7]. Over time, it seems that the direction of AI within abstract strategy games such as Chess, Go, and Shogi is tending towards Machine Learning. Machine learning algorithms such as AlphaZero and Elmo are drastically overperforming traditional AI-based algorithms.

We are also planning on creating a naive evaluator function to assess the value of a given ply. The largest challenges in implementing an such a function for Shogi is weighting how a potential ply will impact various aspects of the game. The plan for our team's project is to have, at minimum, two of the many possible characteristics for a Shogi evaluation function: material evaluation (i.e: pieces), and mobility evaluation. Material evaluation is important for Shogi because of the aforementioned variety in pieces, promotability of multiple pieces, and Shogi's unique reusability of pieces. Mobility evaluation is worth considering because the endgame for Shogi depends largely on the strength of the defensive structure around the king and the presence of escape routes for the king. Since Shogi pieces can be reused, the mobility of pieces holds more importance as the endgame isn't as dependent on the number of pieces on the board as it is on how fortified the king is at all times [3].

3 Approach

3.1 Representation

We are attempting to create the Shogi AI in three different steps. The Shogi board will be treated as our environment. A bit-board, such as the one in Figure 1 can be used to represent a Shogi board. The idea for implementing our Shogi board as a bit-board originated from a Python-Shogi repository on GitHub which sought to provide a fully-fledged game-board representation for Shogi [4].

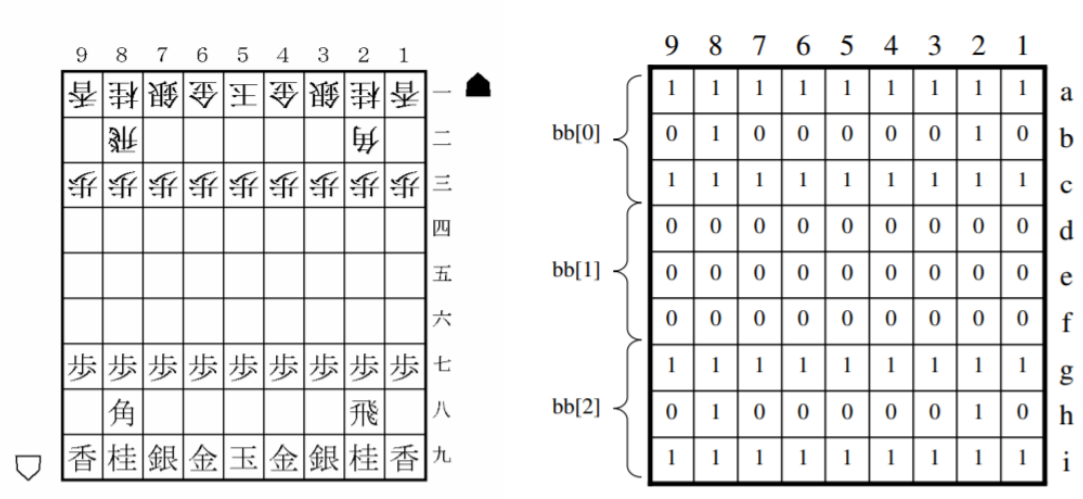

Figure 1: A Shogi board, with its bit-board representation side-by-side

The advantage of using a bit-board to represent a Shogi board is that pieces in a game can be moved using bitwise operations, which offer a fast and computationally-cheap way to move pieces in a Shogi game board. Figures 2 and 3 outline our bit-board initialization in Python:

```
SQUARES = [
    A9, A8, A7, A6, A5, A4, A3, A2, A1,
    B9, B8, B7, B6, B5, B4, B3, B2, B1,
    C9, C8, C7, C6, C5, C4, C3, C2, C1,
    D9, D8, D7, D6, D5, D4, D3, D2, D1,
    E9, E8, E7, E6, E5, E4, E3, E2, E1,
    F9, F8, F7, F6, F5, F4, F3, F2, F1,
    G9, G8, G7, G6, G5, G4, G3, G2, G1,
    H9, H8, H7, H6, H5, H4, H3, H2, H1,
    I9, I8, I7, I6, I5, I4, I3, I2, I1,
] = range(81)
```

Figure 2.1: Bit-board squares representation

```
BB_SQUARES = [
    BB_A9, BB_A8, BB_A7, BB_A6, BB_A5, BB_A4, BB_A3, BB_A2, BB_A1,
    BB_B9, BB_B8, BB_B7, BB_B6, BB_B5, BB_B4, BB_B3, BB_B2, BB_B1,
    BB_C9, BB_C8, BB_C7, BB_C6, BB_C5, BB_C4, BB_C3, BB_C2, BB_C1,
    BB_D9, BB_D8, BB_D7, BB_D6, BB_D5, BB_D4, BB_D3, BB_D2, BB_D1,
    BB_E9, BB_E8, BB_E7, BB_E6, BB_E5, BB_E4, BB_E3, BB_E2, BB_E1,
    BB_F9, BB_F8, BB_F7, BB_F6, BB_F5, BB_F4, BB_F3, BB_F2, BB_F1,
    BB_G9, BB_G8, BB_G7, BB_G6, BB_G5, BB_G4, BB_G3, BB_G2, BB_G1,
    BB_H9, BB_H8, BB_H7, BB_H6, BB_H5, BB_H4, BB_H3, BB_H2, BB_H1,
    BB_I9, BB_I8, BB_I7, BB_I6, BB_I5, BB_I4, BB_I3, BB_I2, BB_I1,
] = [1 << i for i in SQUARES]
```

Figure 2.2: Board bit-shifting

The speed of using bit-shifting to manipulate Shogi pieces during a simulation allows us to keep track of the board after every move by printing out the bit-board after each player's move. For each piece, we can make a set of movements, which are our nodes, that lead to a certain outcome. The collection of all possible outcomes is our state space, and all possible sequences of outcomes from this space that result in a checkmate will comprise our search space proper.

3.2 Algorithms

3.2.1 Minimax and Alpha-Beta Pruning

Since Shogi is a 2-player, zero-sum game, we decided to use a minimax algorithm as the foundation of our AI search. In particular, we have decided to use alpha-beta pruning to allow us to ignore redundant or nonfruitful subtrees within the game tree. This decision was largely predicated on the fact that the state space for Shogi is much larger than the state space for Chess, and it is extremely vital for us to be able to eliminate chunks of such a large search space where we can. The alpha-beta pruning algorithm aids us in reducing unnecessary computations by eliminating non-beneficial branches. This makes for a more efficient solution than just a naive minimax implementation. The use of alpha-beta pruning is more applicable to Shogi in comparison to Chess because of the aforementioned state space size disparity between Shogi and Chess. Since the branching factor for Shogi is significantly greater than the branching factor for Chess, implementing an alpha-beta pruning algorithm specific to Shogi would prune far more nodes and, thus, save a relatively larger amount of total time when applied to Shogi.

3.2.2 Selective Deepening

Due to the greater amounts of movement and piece options (including reusing captured pieces) which lead to a large branching factor of each level of the Shogi game tree, it is important to select and evaluate as few moves as possible while not removing potentially promising moves if the Shogi AI is to function in real-time Shogi game-play, where a player is generally required to select a play/move within 60 seconds of their turn beginning. One approach we used to evaluate less branches in the game tree while maintaining the in-game performance for our AI was Selective Deepening. In order to categorize different moves, we used three particular plausible move generators: the first to generate any moves that captured enemy pieces, another to generate moves that would result in a check of the enemy king, and a third to generate all legal moves in Shogi. The generator for checking moves, called *generate_checking_moves()*, works by evaluating all legal moves and checking these moves before finalizing the generation process. A snippet of the implementation can be seen in Figure 2 on the next page.

The plausible move generator for generating all legal moves was used only when there were no possible checking or attacking moves present for the player based on the current outlook of the game board. Since the plausible move generators focused mostly on offensive moves and there wasn't a plausible move generator to focus on defensive structure and escapability, the AI was inclined towards making aggressive decisions during game simulations.

```
board_copy = ps.Board(board.sfen()) # Needed to view checks
# Piece on Board move
for piece_type in ps.PIECE_TYPES:
    if move_flags[piece_type]:
        movers = board.piece_bb[piece_type] & board.occupied[board.turn]
        from_square = ps.bit_scan(movers)

        while from_square != -1 and from_square is not None:
            # & board.occupied[~board.turn] is to check that
            # all the attack moves are attacking enemy pieces.
            moves = ps.Board.attacks_from(piece_type, from_square,
                                          board.occupied, board.turn) \
                    & ~board.occupied[board.turn]
            to_square = ps.bit_scan(moves)
            while to_square != -1 and to_square is not None:
                # reset board
                board = ps.Board(board_copy.sfen()) # This is expensive
                piece = board.piece_at(from_square)

                board.remove_piece_at(from_square)
                board.set_piece_at(to_square, piece)

                board.turn = not board.turn
                if board.is_check():
                    # must remove so can_move_without_promotion works as
                    # intended
                    board.remove_piece_at(to_square, False)
                    # reset turn to appropriate turn
                    board.turn = not board.turn
                    if ps.can_move_without_promotion(to_square, piece_type, \
                                                     board.turn):
                        yield ps.Move(from_square, to_square)
                    if ps.can_promote(from_square, piece_type, board.turn):
                        yield ps.Move(from_square, to_square, True)
```

Figure 2: A snippet of *generate_checking_moves()*, demonstrating the process the function goes through in order to generate a move that can check the opponent's king.

The function *generate_checking_moves()* works by looking at all legal moves, and if the result of the legal move creates a check, we yield that result. However, this is clearly expensive as we must reconstruct the board for each move to see if the move creates a check.

3.2.3 Static Board Evaluation Algorithms

In order for the AI to better distinguish between the quality of one move relative to another, we decided to add four primary board evaluation algorithms. The first algorithm, detailed in Figure 3, was used to evaluate the materials present on the board. The second algorithm, detailed in Figure 4, was used to evaluate the mobility from a player's position (i.e: to provide a numerical value to the mobility of any playable piece for the current player). A third algorithm was used to evaluate the safety of the

opponent's king by establishing a radius around the king. The last algorithm integrated into the alpha-beta pruning AI was to assess the checkmate value of a move, where the value would be set to ∞ or $-\infty$ if a move could yield a checkmate, and the value would be set to 0 if a move could not yield a checkmate.

These 4 evaluators were passed into the overall evaluator function through a 4-entry integer array *eval_flags[]* as displayed in this snapshot of hte evaluator function:

```
def evaluator(board : ps.Board, eval_flags):
# eval_dictionary = { "count_material" : 1, "count_mobility" : 1,
#              "checkmate_value" : 1, "king_safety" : 1}

evaluation = 0
if (eval_flags[0] == 1):
    evaluation += count_material(board)
if (eval_flags[1] == 1):
    evaluation += count_mobility(board)
if (eval_flags[2] == 1):
    evaluation += checkmate_value(board)
if (eval_flags[3] == 1):
    radius = 1 # arbitrary radius
    evaluation += ks.king_safety(board, radius)

return round(evaluation, sig_figs)
```

The purpose of combining these evaluator functions was to obtain a more accurate e4stimation for the value of any given ply made by the AI. This estimation would thus serve as a means for guiding the pruning process for the algorithm and help increase the information the agent has prior to selecting a move.

3.3 Implementation of Algorithms

The generators and evaluation methods mentioned above are integrated into the base alpha-beta pruning of the AI as such:

3.3.1 Parameters

As can be seen from the function *my_alpha_beta()* in section 4.3.2, the current game board, generator function, and evaluator function flags are all parameters of the function alongside the requested depth of the game tree, the starting alpha and beta values, and a boolean variable "maximizingplayer" which represents which player (black or white), is the maximizing player at the start of the game (i.e: which color's actions will be simulated by the AI).

3.3.2 *my_alpha_beta()*

```
def my_alpha_beta(board : ps.Board,depth, maximizingplayer,
```

```
                    alpha, beta, generator, evaluator, eval_flags):
if (depth == 0 or board.is_game_over()):
    return [None,evaluator(board, eval_flags)]
elif(maximizingplayer):
    max_value = float('-inf')
    best_move = ""
    moves = generator(board)

    try:
        next(generator(board))
        moves = generator(board)
    except StopIteration:
        moves = board.generate_legal_moves(board)

    for move in moves:
        current_move = move.usi()
        board.push_usi(current_move)
        evl = my_alpha_beta(board, depth-1, False,alpha,beta, generator, evaluator
        if (evl[1] > max_value):
            best_move = current_move
            max_value = evl[1]
        board.pop()
        alpha = max(alpha,evl[1])
        if beta <= alpha:
            break

    return [best_move, max_value]
else:
    min_value = float('inf')
    best_move = ""
    moves = generator(board)

    try:
        next(generator(board))
        moves = generator(board)
    except StopIteration:
        moves = board.generate_legal_moves(board)

    for move in moves:
        current_move = move.usi()
        board.push_usi(current_move)
        evl = my_alpha_beta(board, depth-1,True,alpha,beta, generator, evaluator,
        if (evl[1] < min_value):
            min_value = evl[1]
            best_move = current_move
        board.pop()
        beta = min(beta,evl[1])
```

```
            if beta <= alpha:
                break
        return [best_move,min_value]
```

4 Experiment Design

To be able to simulate the game of Shogi, we will be using Python. It is possible for us to either write or code our software to run such a game. An example of software that we can use is called Python-Shogi. This software simulated Shogi in the format of a bit-board. The bit board resembles an actual Shogi game, which makes playing the game and conducting our research easier. Python-Shogi provides a simple, high-level library. It is easy to use and install. We believe that using this software and adjusting a bit would serve the goals of our research. We will also be using the Minimax algorithm with alpha-beta pruning in conjunction with the software described above to create an AI for Shogi. We plan on writing a minimax algorithm to fit Shogi ourselves. Source [7] discusses alternative solutions for creating an AI for Shogi within popular state-of-the-art engines such as YSS, Elmo, and others, which goes into further detail on evaluator functions and methods of selective deepening that are used.

4.1 Game Simulation

In order to test the effectiveness of our Shogi AI, we decided simulate a set of Shogi matches between our α-β pruning based AI, a random move generator, and a human player. The human player is an intermediate-level Shogi player, so we believed that having a person play a set of games against the Shogi engine and report his feedback would help us gain insight into the competence of the AI and areas of the game we could improve the engine on.

4.1.1 Simulation-Specific Implementation

We decided to create a set of functions that would be used for testing and simulating matches between the AI agent, a random agent, and playing our AI as users.

Code for running n number of game simulations was stored in an individual Python file called SimulateGames.py. This file contained two primary methods: simulate_games(), and process_stats().

While the design of the function is suboptimal as it doesn't take in user arguments as parameters, the intention of using such a function to simulate games was to ensure proper testing of the AI rather than extending a game simulation UI as a feature of the engine. However, with parametrization of some variables (such as game_count, eval_flags) used in the function, this function can be extended for the purpose of a Shogi UI simulator.

4.2 Tsume

A Tsume, which is a Shogi checkmating puzzle, is the traditional way of training beginner Shogi students to become better at the game, serves as a good point of evaluation for our Shogi AI. Given a random n-move sequence from which it is possible to threaten

the opponent king or checkmate the opponent, a solution to a Tsume puzzle involves a player arriving at a checkmate victory. Figure 4 depicts a rudimentary Tsume:

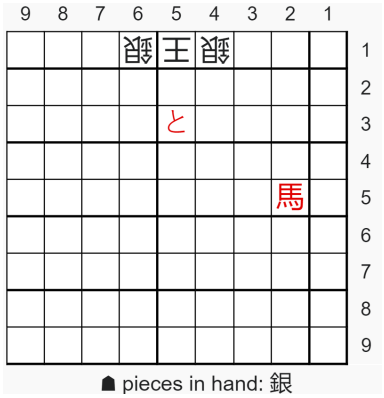

Figure 3: A basic Tsume puzzle, where the player has to decide on a series of moves that would lead them towards a checkmate.

Testing the capability of our Shogi AI to solve Tsumes in a timely manner will verify the checkmate evaluation capability of the AI.

4.2.1 Tsume Capabilities

Tsume is a type of puzzle that is very good for understanding the tactics of Shogi. The rules are simple.

1. The attacking side is Black
2. All of the attacking side's moves must be checks
3. White (the defender, or the side with the king) must move in such a way to delay checkmate as long as possible.
4. White has in hand all pieces not in the board or in the attacking side's hand, not counting the other King
5. White can drop any piece in hand to delay or prevent checkmate.

4.3 Test Design and Results

In total, we decided to create a set of 360 total shogi matches/simulations, split into the following categories: Random Agent v.s. Random Agent, Alpha-Beta Agent without King Safety v.s. Random Agent, Alpha-Beta Agent without King Safety v.s. Random Agent, Human v.s. Aggressive Alpha-Beta without King Safety, Human v.s. Aggressive Alpha-Beta, and Human v.s. Alpha-Beta with all moves.

4.3.1 Random Agent v.s. Random Agent

The 200 games played between two random Shogi move generators served to measure the variability in the success between the two starting positions of black or white.

4.3.2 Alpha-Beta Agent without King Safety v.s. Random Agent

There were 100 games played between an Alpha-Beta Agent that did not account for King Safety and a random move generator. This simulation was done to evaluate the baseline effectiveness of the Alpha-Beta Agent against a "beginner" level player without any knowledge of Shogi.

4.3.3 Human v.s. Various Alpha-Beta Agents

There were 60 total played by a Human against three different variations (20 games per variation) of our Alpha-Beta Pruning algorithm. While playing only 20 games per variation provides a small sample size, it was the maximum amount of games that could be played in one 3-4 hour session by a Human per day since the time taken by the Alpha-Beta pruning agents for generating one individual move, on average, was about 2 minutes. Realistically, allocating more than 15 of the 50 hours for testing and verification of the Shogi AI would leave too little time to focus on the implementation. Thus, with the Human v.s. Alpha-Beta simulations taking over 3 hours to complete a set of 20 tests, our team decided that it would be best to complete 20 games with two other variations of the Alpha-Beta pruning algorithm instead. The aim of these games was to assess the effectiveness of the AI against a more experienced Shogi player.

Table 1 displays the results recorded by the Shogi simulation system we built after running 360 total game simulations across 5 subsets of testing. Note that the human used in this simulations would be considered an intermediate-level player in competitive Shogi play. KS is an abbreviation for "king safety", which refers to the presence or absence of the king safety evaluator function in an agent. Every α-β agent can be assumed to have the king safety evaluator unless otherwise specified.

Shogi Game Simulation Data				
Agent 1	Agent 2	Agent 1 Victories	Agent 2 Victories	Agent 1 Win Rate
Random	Random	102	98	0.51
α-β, no KS	Random	100	0	1.00
Human (intermediate)	α-β, aggressive, no KS	20	0	1.00
Human (intermediate)	α-β, aggressive	20	0	1.00
Human (intermediate)	α-β, all-moves	20	0	1.00

Table 1: Shogi Game Simulation Data

4.3.4 Tsume Simulations

There were 170 total simulations performed across 3 different types of Tsumes. The simulations were categorized by the number of moves required - and thus the depth of the game tree required - to ideally find checkmate per Tsume. A Tsume was an

excellent method of validating the system's ability to find checkmate since there is a natural conversion between the depth parameter for the *my_alpha_beta()* used by us and the minimum number of moves that could possibly lead to a checkmate in a Tsume.

Tsume Simulation Data			
Number of Moves (depth)	Number of Trials	Average Time: Tsume Moves (seconds)	Average Time: All Legal Moves (seconds)
3	100	0.67	4.10
5	50	6.23	40.91
7	20	56.72	1873.52

Table 2: Tsume Simulation Data.

A Tsume move, mentioned in the third column of this table, refers to a checking move.

5 Analysis

Clearly from the match history of our Agents, it is obvious that our AI, while decisively better than just playing at random, is still a weak AI when compared against humans. We had our AI play with both selective deepening, and all legal moves at a depth of 3 and it was still decisively swept, losing 20/20 games. This clearly indicates that there are a couple of primary candidates for problems. First, there are not enough evaluator functions to accurately depict a board position. Second, performance issues resulting in only allowing for a depth of 3 for our α-β pruning. This makes our AI effectively only able to react to one move that its opponent would play to its initial proposed moves. This is clearly very weak, and even a novice human player would be able to beat our AI.

Regarding the Tsume Simulation data, it is clear from this that selective deepening has massive performance increase. This further reinforces just how complex Shogi is, and denotes the significance of Shogi's branching factor. We can extract the size of the branching factor from this data by examining the average time for our α-β pruning algorithm to operate on Tsume moves, versus all legal moves for board positions. The first trend we notice for both of these columns, is that as the depth(number of moves for the Tsume) increases, the average time increases exponentially. This is obvious due to the nature of α-β pruning being recursive. However, because of this recursive nature, by decreasing the number of positions we need to evaluate through selective deepening at each step in the recursion, we also exponentially decrease the number of positions we need to evaluate. This is reflected through the exponential difference in time between the time it takes to find checkmate when looking at Tsume moves and all legal moves for the same depth.

However, there is something noticeably odd about the data provided here. This clearly being the jump between a depth of 5 and a depth of 7 for all legal moves to find a checkmate. Going from 40 seconds to over 30 minutes seems very odd. As such, there are some confounding variables that we need to consider as to why our time jumps so highly. One of these confounding variables is strictly from the data sample gathered for Tsumes at depth 7 compared to depth 5. With the 7-move Tsumes, they had an average number of pieces on the board of 15, meaning that the culmination of the 25 other pieces were, in some fashion, in both players' hands. Comparatively, for the 50 5-move Tsumes, there was an average of 26 pieces on the board. Clearly a lot more.

Now obviously, with more pieces in hand, the number of legal moves over a depth greater than 1 exponentially increases, as each piece in hand can be played at almost any unoccupied square on the board. This is one of the confounding variables. Another confounding variable that might contribute to the oddity mentioned in the data, is the size of the testing sets.With the selected testing sets decreasing as the depth increased, our data is more prone to outliers within the higher depth Tsume problems.

5.1 General Observations

Discernible from our AI's match history, it is very obvious how our AI is rudimentary. Primarily, our AI has several missing behaviors that a Human player intuitively knows. One of these concepts is Castling. This is a semantic concept that is difficult to program in organically, without a lookup table. As such, it was omitted for the sake of time. However, since our AI had no understanding of Castling, the AI was at an insurmountable disadvantage. This is one of the crucial behaviors that our AI was missing.

Furthermore, there were several missing concepts at the board evaluation stage. One of these being the concept of clustering pieces for an attack, and not just on the enemy king. While our AI had a concept of clustering units to attempt to checkmate its opponent through the King Safety evaluator function, our AI had no concept of clustering units around crucial board zones. Typically, these types of zones would be where the opponent is focusing their pieces, or just in general crucial squares on the board.

Positives	**Negatives**
+ Proof of Concept	- Detrimentally Constrictive Selective Deepening
+ Demonstrates the Power of Selective Deepening Through Puzzles	- Missing Crucial Behaviors/Lackluster Board Position Evaluation

Table 3: Brief Outline of General Observations

5.2 Positives

5.2.1 Proof of Concept

The concept which this implementation reinforces is multi-fold. First and foremost being the performance increase from selective deepening. Obviously, with selective deepening, α-β pruning does not need to explore as many branches since with selective deepening we are truncating the branches explored base on selection. We explored the effectiveness of this using our α-β pruning. Particularly, this was done through comparing a board position which would be simultaneously the largest possible number of legal moves, and minimizing the number of moves that are capable of being generated by our AI.

5.2.2 Demonstration of Selective Deepening

In order for us to be able to signify the power of selective deepening, we needed to analyze the performance of our AI utilizing selective deepening specifically for Tsume puzzles versus the generator function for all legal moves within the position.

What we found among our testing, is that clearly there is massive savings when using selective deepening for Tsumes. This then extends into the game of Shogi itself, since when looking at all legal moves for a board position, there are a lot of moves that are non-progressive. One such example would be moving a piece back to the square it was moved one turn ago, unprovoked. However, there are many more examples like this, which would allow for a more reasonable subset of moves to explore with α-β pruning. Clearly, this demonstrates then, a proof of concept.

5.3 Shortcomings and Their Origin

5.3.1 Generator Functions Affecting Behavior

Three generator functions were used to adjust the behavior of our AI. These three being

1. Checking Moves
2. Attacking Enemy Material Moves
3. Default Generator for All Legal Moves

In the event neither of the first two generator functions could get any qualifying moves, a fallback of all legal moves was used. This behavior of utilizing *ONLY* the first two generator functions if either of them produced a move resulted in recklessly aggressive moves. Even if the moves are clearly detrimental for the AI's board position. To elaborate, the figure below represents an example of such a situation:

$$\left|\begin{array}{ccccccccc} l & n & s & . & . & g & s & n & . \\ r & . & g & . & k & . & . & b & l \\ p & p & p & p & p & . & p & p & p \\ . & . & . & . & . & B & . & . & . \\ P & P & . & . & . & . & . & . & . \\ . & . & P & . & . & . & . & . & . \\ . & . & . & P & P & P & P & P & P \\ . & . & . & . & . & . & . & R & . \\ L & N & S & G & K & G & S & N & L \end{array}\right|$$

SFEN: lns2gsn1/r1g1k2bl/ppppp1ppp/5B3/PP7/2P6/3PPPPPP/7R1/LNSGKGSNL b P 13

Within this example, it is Player1's turn to move(our AI). Since it is restricted to either checking moves or attacking moves, our AI must capture either of the pawns diagonal to it. Both resulting in a clearly losing position, as our AI will be trading its Bishop for a pawn.

5.3.2 Missing Crucial Behaviors

This implementation of our AI does not have any evaluation which quantifies the benefits of defensive moves. As such, our AI inadvertently considers these moves to be detrimental, as they are not beneficial, and attacking moves almost always exist. This behavior obviously led to the reckless aggression of our AI, which in turn made it weak.

Another missing behavior that our AI, is that our AI has no concept of Castling. Castling is obviously a specific sequence of moves to create a structure surrounding the King to protect it. Not having this caused our AI to be at an unnecessary disadvantage, which was easily capitalized on. However, this behavior is not easy to implement, and as such was not.

Lastly, one of the other key behaviors that is intuitive to humans, is the concept of Zone Control. What is meant by this is the amount of pieces and "strength" a player has over a certain zone of squares on the board, making this zone of significance. While there is an evaluator method for King Safety which replicates this idea of Zone Control, our AI does not do this in general for squares on the board. This further makes our AI more aggressive, and dismissive of other areas of the board which may be of strategic advantage.

6 Conclusion

We attempted to create a solution for the problem of playing the game Shogi, also known as Japanese Chess. What distinguished Shogi from European Chess in the realm of AI is the game-tree complexity. Shogi has a game-tree complexity of about 10^{226}, whereas in chess it is considered to be 10^{123}. Clearly, this makes constructing an AI for Shogi more difficult.

We developed an AI to play Shogi using techniques currently utilized by the top engines used to play games such as Chess and Shogi. Primarily, α-β pruning, alongside with selective deepening to allow for a stronger AI. However, as clearly seen by the results of our AI, it was ineffective at even beating an intermediate player, let alone top engines. The key factors that led to the underperformance of this AI were the lack of integration of additional evaluator functions that accounted for more behaviors. Additionally, there were certainly more generator functions to help the AI consider moves — such as defensive or positional moves — that weren't solely attacking moves. Furthermore, our AI had no understanding of what castling is (because it's stupid), and thus had a massive disadvantage comparatively.

However, this AI was able to successfully and performatively solve Tsume puzzles. The AI did this through demonstrating the power of selective deepening, and provided itself as a proof-of-concept. Particularly, as clearly seen by the data, we were able to cut down on time used to solve 7-move Tsume's by 3303%, or more specifically, from 1873.52 seconds to 56.72 seconds on average.

To improve upon the AI's ability to compete with others, the first priority would be to increase the number of generator functions, thereby adding functions that consider more types of moves analogous to what top engines like YSS have as mentioned in 2.1.3 Traditional AI: YSS. Next, prioritizing additional static board evaluation functions for additional features would greatly improve the accuracy of our AI, and our AI would have a more holistic understanding of any board position. With these changes, this AI

would be able to contend with intermediate players, and may even be able to compete with some of the top engines with some degree of success.

References

[1] T. Anthony, Z. Tian, and D. Barber. Thinking fast and slow with deep learning and tree search, 2017.

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[3] R. Grimbergen. An evaluation function for shogi. In *Proceedings of Game Programming Workshop in Japan*, volume 97, pages 159–168. Citeseer, 1997.

[4] gunyarakun. Python-shogi. https://github.com/gunyarakun/python-shogi.git%7D%7D,commit={74c668874dc9b17c5f211308a9cec8df59e82a49}, 2021.

[5] R. Hare. A brief introduction to shogi. 211 Pages, Free online resource, January 2021.

[6] M. Igami. Artificial intelligence as structural estimation: Deep Blue, Bonanza, and AlphaGo. *The Econometrics Journal*, 23(3):S1–S24, 03 2020.

[7] H. Iida, M. Sakuta, and J. Rollason. Computer shogi. *Artificial Intelligence*, 134(1):121–144, 2002.

[8] M. Kamiya. Shogi and artificial intelligence. *Discuss Japan – Japan Foreign Policy Forum No. 32*, 2016.

[9] P. Kiernan. Which is greater? the number of atoms in the universe or the number of chess moves?

[10] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[11] N. Kumar et al. Can chess ever be solved. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, 12(2):2814–2819, 2021.

[12] K. Matake. Shogi and artificial intelligence. In *Discuss Japan. Japan Foreign Policy Forum*, 2016.

[13] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.