Utrecht University
Bachelor Artificial Intelligence

Bachelor Thesis AI (KI3V12011) - 7.5 ECTS

Solving Solo Chess

Supervisor:
Tomas Klos

Second Reader:
Nima Motamed

Candidate:
Stan Verlaan
6813526

July 1st - Academic Year 2021/2022

This thesis is made as a completion of my Bachelor education in Artificial Intelligence at Utrecht University. Several persons have contributed academically, practically and with support to this Bachelor thesis. Firstly, I would like to thank my supervisor Tomas Klos for his excellent guidance during this process, sometimes spending more time than originally assigned in order to help me. Furthermore, I wish to thank Lucas Meijer for brainstorming ideas with me. Of course I would also like to thank my fellow students and good friends who have shown great interest in what I do: Merlijn, Lianne, Xavi, Floris, and Giel. I have benefitted immensely from debating these topics with you. Lastly, I would like to express my gratitude to my family and especially to my brother. If it wasn't for his love of chess, I would not have stumbled upon this great topic for my research.

Abstract

In this paper, we study a simplified puzzle variant of the game of chess, termed Solo Chess. While the game is proven to be NP-complete, and an efficient algorithm for solving it does not seem to exist, we propose an algorithmic approach for solving the game by formulating the problem as a Bounded Classical Planning Problem and designing and applying a general backtracking algorithm. In order to investigate which tactics and approaches work best when playing Solo Chess, we incorporate knowledge and intuition about the game into the algorithm in the form of heuristics that guide the search to a goal by picking which moves to consider first. Results from testing on $11,000$ puzzles generated ourselves show that a heuristic based on a ranking of the piece types is consistently effective on all kinds of puzzles, and that a heuristic based on moving the pieces to the center can achieve outstanding performance under circumstances in which there are few pieces on the board or when there are relatively more high ranked pieces present in the start configuration.

Contents

1 Introduction

1.1 General Introduction

With origins that can supposedly be traced back to seventh century Indian civilization [1], the game of chess has attracted more popularity than ever during the recent pandemic years following the COVID-19 outbreak.

Conventionally played between two opponents on opposite sides of the 64-squared board, the objective of this turn based, perfect information game is to checkmate the opponent's King by threatening it with an inescapable capture using some of the sixteen wooden chess pieces consisting of one King, one Queen, two Rooks, two Bishops, two Knights, and eight Pawns for each player, with each piece type having its own characteristic way of moving across the board.

While the board configuration at the start of a game appears well organised, a total derangement of this orderly setup already starts to take place within the very first few moves. After both players have posed their opening move, 400 possible board configurations exist. This number grows to $197,742$ after the second pair of turns and there are more than 121 million possible game setups after the third [2]. With every move, a progressively more distinctive path is formed, to the extend that seemingly ordinary games can unfold into ones that have never been played before.

In his groundbreaking 1950 paper *Programming a Computer for Playing Chess* [3], Claude Shannon, often referred to as the father of information theory, extrapolated that the number of possible variations to be calculated from the initial position is of the magnitude 10^{120}, based on a typical game lasting about 40 pairs of moves with an average of about 10^3 possibilities for each such pair. A classical comparison is often made with the estimated total number of atoms in the observable universe, which lies between 10^{78} and 10^{82}. To this day, the *Shannon Number* is still regarded as an accurate conservative lower bound on the game-tree complexity of chess and is often cited as one of the barriers of solving chess using an exhaustive analysis.

The infeasibility of brute force methods caused by this incredibly large state space, affirmed by proof of generalized chess being complete for the class EXPTIME [4], has forced researchers in the field of artificial intelligence to turn to more intelligent approaches to try and solve the game. Examples of current state of the art chess engines are the open-source engine Stockfish [5] and DeepMind's AlphaZero [6]. While the former incorporates hand-crafted evaluation functions, minimax alpha-beta pruning and so called Efficiently Updateable Neural Networks (NNUE), the latter achieved superhuman performance by self-play training with a general reinforcement learning algorithm. These sophisticated engines have become increasingly more advanced in the last decade, to the extent that even renowned chess grandmasters cannot compete. However, by incorporating machine learning algorithms, these approaches have become difficult to understand and outcomes become hard to interpret. Furthermore, because they are based on data that is limited compared to the total number of possible variations, they will never be truly optimal. Although chess is solvable in theory, in practice it seems that perfect play is unreachable.

1.2 Solo Chess

In this paper, we study a simplified and arguably natural variant of the game of chess, termed Solo Chess. This game was introduced at the start of 2018 on `chess.com`, a website that with approximately 77 million users is the most widely used resource for online chess competition and practice. Solo Chess is a single player game that uses chess pieces and can be seen as a kind of logic puzzle. In this section, we explain the rules of the game, go into the main differences compared to chess, and go over an example instance.

1.2.1 How It Works

The game of Solo Chess can be played by oneself using only the knowledge of how the pieces move. Here is how it works [7]:

> **Solo Chess.** Given a board configuration, the object of Solo Chess is to move the pieces one by one in such a way that they all capture each other until only one piece remains. The rules are as follows:
>
> 1. Every move must capture a piece
> 2. No piece may capture more than 2 times per puzzle
> 3. If there is a King on the board, it must be the final piece
>
> The difficulty increases as more and more pieces are added to the puzzles, and one must navigate the maze of captures and find the path that leads to a lone remaining piece. Note that a specific instance may have more than one solution. Try it out here: `https://www.chess.com/solo-chess`

1.2.2 Differences to Chess

The most notable difference to chess is that Solo Chess is a single player game. Hence, there is no ambiguity or anticipation concerning other player's moves involved. Since there is no competition directly on the board, start configurations only consist of white pieces. To indicate that a piece has reached its maximum number of captures (two, in the standard case of Solo Chess) and thus that its position has been fixed, a piece often turns black in visualisations of the game.

Furthermore, there is no restriction on the number of pieces of a certain type that are present in start configurations (with the exception of the King), meaning there might be more than eight Pawns on the board (or multiple Queens, Rooks, etcetera, although this could theoretically be possible in standard chess due to promotion). In contrast to the free movement pieces are allowed in chess, movement in Solo Chess only consists of captures, although the characteristic way of how each piece type captures is sustained.

As a result of these restrictions, Solo Chess has a more confined form and a much smaller state space compared to chess, which makes the game more easily solvable using exact algorithms.

1.2.3 Example Instance

The following is a level 5 (meaning it consists of six pieces) puzzle instance from `chess.com` to demonstrate how Solo Chess works. The board configuration given at the start of this specific level is illustrated in Figure 1.

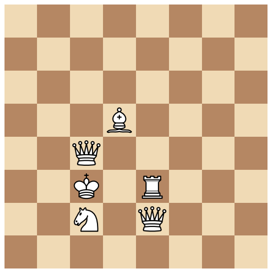
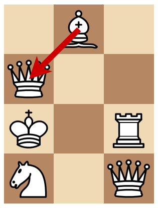
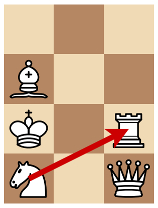
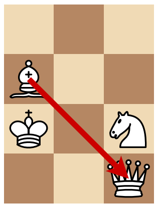


Figure 1: A level 5 puzzle's start configuration

To solve the instance, we need to perform a sequence of five captures which results in the King being the only piece left on the board. However, by making the wrong captures a dead-end can be found, as illustrated in Figure 2.


(a) Bishop takes Queen


(b) Knight takes Rook


(c) Bishop takes Queen

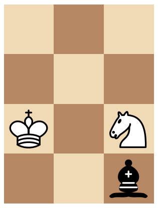
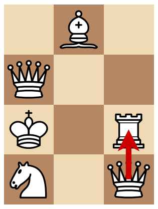

(d) No possible moves left

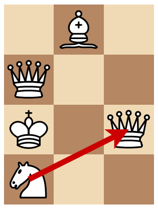
Figure 2: A sequence of steps that leads to a dead-end

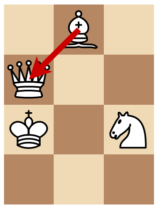
The configuration in (d) is a dead end, since more than one piece remains on the board without any valid move possible. Note that the Bishop has turned black because its position has been fixed, due to it reaching its maximum of two captures. In this case, it would not have made a difference, since the Bishop is not in a position to attack anyway.

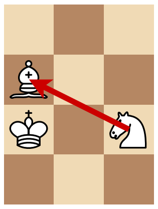
The wrong move that resulted in the configuration becoming unsolvable was made in (b). If instead of the Knight capturing the Rook, the move was made in which the Bishop takes the second Queen, the configuration could subsequently be solved by using the Rook twice: once to capture the Bishop and once to capture the Knight. Then, the King could finish with capturing the Rook that would be placed in the square south of it.

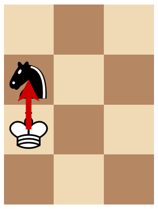
For clarification and to show that multiple solutions can exist, another sequence of steps that solves the puzzle are pictured in Figure 3.


(a) Queen takes Rook


(b) Knight takes Queen


(c) Bishop takes Queen


(d) Knight takes Bishop


(e) King takes Knight

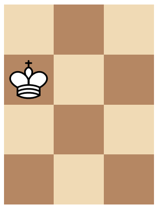

(f) Valid solution found

Figure 3: A possible solution

Again, observe how the Knight turns black in (e) as a result of it having made two moves. A solution has been found in (f), since all steps beforehand were valid and the King is the only remaining piece on the board. Keep in mind that this is not mandatory: a King might be absent in the start configuration, which means that any piece is allowed to be the last one standing.

1.3 Related Literature

At the time of writing, there has only been one research paper done on the topic of Solo Chess. Aravind et al. [8] show that a generalized version of Solo Chess on a board of size $n \times n$ is NP-complete. However, their contribution focuses on the decision problem of determining whether a given Solo Chess instance is solvable, while restricting themselves to board configurations that only consists of chess pieces of the same kind (e.g. only Rooks).

To be more precise, Aravind et al. introduce a natural generalization of Solo Chess that they call `Generalized Solo Chess`(P, d), where $P \subseteq \{$♛, ♖, ♝, ♘, ♙$\}$ is a collection of piece types and $d \in \mathbb{N}$.

This generalized version is played on a infinite integer lattice, where initially the positions of n pieces (each one of the types given in P) are given, as well as for each piece the number of captures it can make, bounded by d. The goal and rules of the game are the same as described in section 1.2.1, only with the absence of rule 3 (they do not consider the King) and with the modification of rule 2: No piece may capture more than the given integer number of times for that piece, specified by the input of a certain instance.

The following are some results of the paper:

1. (Intractability for Rooks). `Generalized Solo Chess`(♖, 2) is NP-complete.
2. (Intractability for Queens). `Generalized Solo Chess`(♛, 2) is NP-complete, even when all Queens are allowed to capture at most twice.
3. (A characterization for Pawns). `Generalized Solo Chess`(♙, 2) with n white Pawns, each of which can capture at most twice, can be decided in $O(n)$ time.

We see that Aravind et al. focus on settings where $|P| = 1$. Consequently, as Aravind et al. remark themselves at the end of their paper, they did not explicitly study `Generalized Solo Chess` with multiple piece types on the board. However, most such variants would be hard given the complexity results established in the paper for pieces of one kind.

Even though they do not try to solve the game, the paper provides a lot of insights for certain approaches which can help in finding a solution and suggestions for further research into different variants of the game, to which we return later.

1.4 Our Contributions

In this bachelor thesis, we propose a generic algorithm and framework for solving the generalized variant of the game of Solo Chess. Here, generalized means that the instances are bound by some parameters that can change between different instances (such as the board size kxl and the maximum number of allowed captures per piece d), as in the case of the paper by Aravind et al. We argue that due to its NP-completeness, the best approach to solving the game would be to apply an algorithmic approach called backtracking.

However, we are not only interested in solving the game, but another purpose of this research is to investigate which tactics help in doing so. When playing the game, humans can apply different strategies in order to find a solution. For instance, one could pay attention to the piece types on the board and their relative strength (a Queen seems to be more important than a Pawn), take into account the number of pieces each piece is attacking, or try and move every piece to the center of the board in order to have all pieces grouped together.

We propose three different heuristic functions that incorporate such tactics and investigate the impact they have on the ability of the solver algorithm to solve certain difficulty levels, compared to a randomized version of backtracking without any sort of guidance. We expect that the use of these heuristics speeds up the process of finding a solution. Our focus here lies on instances which are known to be solvable, such as the puzzles provided by chess.com and puzzles generated by ourselves.

The value of solving Solo Chess lies in the value the game of chess has in the research field of artificial intelligence. It can provide new insights and extend the literature in this area, especially since Solo Chess is such an understudied research topic. While chess itself is too complex to solve using exact algorithms, restricting our research to solving Solo Chess can create more possibilities for different intelligent approaches, which can provide new information regarding strategies to play chess, since there are common goals between the two games and insights gathered from Solo Chess puzzles can help in determining the right moves for certain chess positions. Solo Chess provides an excellent domain for all kinds of problem-solving techniques, with applications outside of the domain of chess as well. Information learned from research in this area can help in solving other logic puzzles, some of which may be generalized to real world problems in planning and logistics.

In Chapter 2 and 3 we discuss the theory related to the problem of Solo Chess. In the former, we formalize the problem representation as a planning problem, while giving formal definitions regarding a problem instance, state and solution. In the latter, we consider different approaches to solving the game algorithmically, define our heuristics, discuss techniques to reduce the search space, and go into the generation of puzzle instances. In Chapter 4, we go over our methodology, hypothesize about our approaches and describe the experimental design required for testing. The results from the testing phase are discussed in Chapter 5, and we draw our conclusions and hold a discussion in Chapter 6 and 7 respectively.

2 Problem Representation

In this section, we propose a formulation of the Solo Chess solving problem as a Bounded Classical Planning Problem. In order to achieve this formulation, we give formal definitions of the problem instance, the state representation, the possible action space, a planning model, and a plan, which is a solution to the problem instance. By formulating our problem this way, we lay the groundwork for modelling possible solution approaches, which we discuss in the next chapter.

2.1 Problem Instances

The problem instances of Solo Chess are the puzzles given that need to be solved. Practically, these are start configurations that consist of a chessboard and some chess pieces that are positioned on the board, as well as the number of captures each piece can make. We define this formally as follows:

Definition 2.1 (Solo Chess Instance)**.** An instance I of the Solo Chess problem, corresponding to a start configuration of n pieces on a $k \times l$ board, consists of n tuples $t_1, ..., t_n$, where $t_i = (p_i, x_i, y_i, \mathit{capturesLeft}_i)$ with $p_i \in P$ a piece, $x_i \leq k \in \mathbb{N}$ the row position of the piece, $y_i \leq l \in \mathbb{N}$ the column position of the piece, and $\mathit{capturesLeft}_i$ the number of captures the piece can make.

Note that an instance is defined in a generalized form, in which the number of pieces n, the board size $k \times l$, and the maximum number of allowed captures $\mathit{capturesLeft}_i$ per piece p_i are all parameters which may vary between instances.

2.2 State Representation

During the process of solving, the problem can be in different states. In the case of Solo Chess, we represent these states as follows:

Definition 2.2 (State Representation)**.** A state $s = \langle P, Q, square, capturesLeft \rangle$ consists of

- Multiset of pieces currently on the board $P = \{p_1, ..., p_n\}$,
- Set of squares on the board currently occupied by some piece $Q = \{q_1, ..., q_n\}$ where $q_j = (x_j, y_j)$,
- Occupation function $\mathit{square} : P \to Q$ which assigns a square $q_j \in Q$ to each piece $p_i \in P$ signifying that p_i occupies q_j,
- function $\mathit{capturesLeft} : P \to \mathbb{N}$ which assigns the number of captures left $\mathit{capturesLeft}_i \in \mathbb{N}$ to each piece $p_i \in P$.

Note that P is a multiset of all pieces enumerated. The reason for this is that we need more than just a piece type in order to distinguish between states that have the same piece types on the same squares but for some piece(s) a different number of captures left. Properties of pieces such as its type can be encapsulated by some function $\mathit{type} : P \to \{$♕, ♖, ♗, ♘, ♙$\}$. However, since its type is fixed during the whole puzzle, it is not relevant to include it in the representation of specific states.

The reason Q only consists of squares that are occupied by some piece is because of the following:

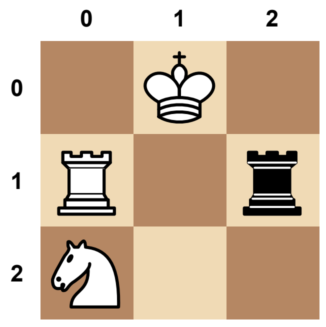
Observation 2.1. *Squares corresponding to state s_i that are unoccupied on the board, remain unoccupied in succeeding states s_{i+j} for all $j > 0$.*

Proof. Suppose that a certain square q is unoccupied during states s_i up to and including s_{i+j-1}, and becomes occupied in state s_{i+j}. This means that the move made in s_{i+j-1} must have led to a piece p moving to q. By the first rule of Solo Chess in Definition 1.2.1 (every move must capture a piece), this move is only valid if there was some piece p' standing on q during the state s_{i+j-1}, which we assumed not to be the case. This holds for all $j > 0$, and so by contradiction it follows that once a square becomes unoccupied, it remains unoccupied. ■

Hence, empty squares are irrelevant in the current state and stay irrelevant in succeeding states.

The functions *square* and *capturesLeft* are essential to the state representation, since these capture the differences between certain states the best. There are many states with n pieces and n occupied squares, but it is the coupling of pieces to squares and the number of captures left per piece that differentiates these states. *square* is a bijective function, meaning that there is a one-to-one correspondence between P and Q.

An example of a mid-puzzle board configuration and its state representation is illustrated in Figure 4.



(a) A mid-puzzle board configuration

$s = \langle P, Q, \mathit{square}, \mathit{capturesLeft} \rangle$ with

- $P = \{p_K, p_{R1}, p_{R2}, p_N\}$,
- $Q = \{(1,0), (0,1), (2,1), (0,2)\}$,
- $\mathit{square} = \begin{cases} \mathit{square}(p_K) = (1,0) \\ \mathit{square}(p_{R1}) = (0,1) \\ \mathit{square}(p_{R2}) = (2,1) \\ \mathit{square}(p_N) = (0,2) \end{cases}$
- $\mathit{capturesLeft}(p) = \begin{cases} 2 & \text{for } p \in \{p_K, p_{R1}, p_N\} \\ 0 & \text{for } p = p_{R2} \end{cases}$

(b) The corresponding state representation

Figure 4: Example state representation

2.3 Possible Action Space

States can be turned into new states by performing actions. However, the actions applicable depends on the current state:

Definition 2.3 (Possible Action Space, Move)**.** The possible action space, denoted $A(s)$, is the set of all actions applicable in state s:

$$A(s) = \{(p, p') \mid p, p' \in P,\ \mathit{capturesLeft}(p) > 0,\ p \textit{ can reach } p'\}.$$

In words, the set of possible actions in a certain state s consists of tuples (p, p'), which signifies the capture of piece p' by piece p. Such a tuple is an *action*, often referred to as a *move* or *capture*. A move is valid if and only if p can reach p' according to standard chess movement rules and p has not reached its maximum number of captures yet.

2.4 Planning Model

Now that we have clear definitions for instances and states, we can formalize the problem as a Classical Planning Problem. Classical planning is the problem of finding a sequence of actions for achieving a goal from an initial state, assuming that actions have deterministic effects [9]. Applied to the problem of Solo Chess, we get the following planning model:

Definition 2.4 (Solo Chess Planning Model)**.** Given an instance I of Solo Chess as defined in Definition 2.1, the Solo Chess Planning Model $\Pi = \langle S, s_o, s_G, A, f \rangle$ consists of:

- a finite and discrete set of all possible states S that can be generated by the problem instance, i.e. the state space.
- the initial state $s_0 = \langle P_0, Q_0, \mathit{square}_0, \mathit{capturesLeft}_0 \rangle \in S$ generated by I.
- a set of goal states $S_G \subseteq S$, where

  $$S_G = \{\langle P_{n-1}, Q_{n-1}, \mathit{square}_{n-1}, \mathit{capturesLeft}_{n-1} \rangle \mid P_{n-1} = \{p\},\ \mathit{type}(p) = ♔\},$$

  if there was a King present in the start configuration of I, otherwise the *type* condition for the last piece can be left out.
- a set of actions $A = \{(p, p') \mid p, p' \in P_0\}$. We often use $A(s) \subseteq A$ to refer to the actions applicable in a certain state s, see Definition 2.3.
- a deterministic state transition function $f : S \times A \to S$, where $s_{t+1} = f(s_t, (p, p'))$ for some action $(p, p') \in A(s_t)$. Then f transforms P, Q, *square*, and *capturesLeft* as follows:

  $$\begin{aligned} P_{t+1} &= P_t \setminus \{p'\}, \\ Q_{t+1} &= Q_t \setminus \{\mathit{square}(p)\}, \\ \mathit{square}_{t+1}(p) &= \mathit{square}_t(p'), \\ \mathit{capturesLeft}_{t+1}(p) &= \mathit{capturesLeft}_t(p) - 1, \end{aligned}$$

  where *square* and *capturesLeft* remain the same for all pieces other than p and p'. In words, the execution of action (p, p') causes the piece p' and the square on which p was standing to be removed in the succeeding state. In addition, the square assigned to p changes to the square previously assigned to p' and the number of captures left for p is decreased by one. These changes are often called the *effects* of the action, while the requirement $(p, p') \in A(s_t)$ encapsulates the so called *preconditions* [10].

Now that the planning problem is modelled, we can define a solution to an instance of Solo Chess:

Definition 2.5 (Solo Chess Solution)**.** A solution π, also called a plan, for a problem instance $I = t_1, ..., t_n$, is a sequence of actions $\pi = a_0, ..., a_{n-2}$ that generates a sequence of states $s_0, ..., s_{n-1}$, such that

- s_0 corresponds to the initial state generated by I,
- $a_t \in A(s_t)$ is applicable in s_t and results in state $s_{t+1} = f(s_t, a_t)$,
- $s_{n-1} \in S_G$ is a goal state. That is, $s_0[\pi] \in S_G$.

Note how a solution always consists of $n - 1$ actions with n being the number of pieces in the start configuration corresponding to I. This is a consequence of the fact that each action decreases the number of pieces by one, until exactly one piece remains in the goal state. Hence, the solution space is bounded and the problem of Solo Chess can be termed a Bounded Classical Planning Problem.

2.5 Alternative Graph Representation

To make the model and state representation defined in the previous subsections more intuitive, we illustrate a different way to represent a state by extending the state definition of Definition 2.2 with the possible actions space of Definition 2.3 using graph theoretic notions.

Definition 2.6 (Graph State Representation)**.** The directed graph $G_t = \langle V_t, E_t \rangle$ models state $s_t = \langle P_t, Q_t, square_t, capturesLeft_t \rangle \in S$ and the possible actions space $A(s_t) \subseteq A$, when

$$\begin{aligned} V_t &= \{\langle p, q, capturesLeft_t(p) \rangle \mid p \in P_t,\ q = square_t(p)\} \\ E_t &= \{\langle v, w \rangle \mid v, w \in V_t,\ (v.p, w.p) \in A(s_t)\}. \end{aligned}$$

A vertex $v \in V_t$ combines a piece, the square it is occupying and its number of captures left. An edge $(v, w) \in E_t$ is a directed edge from vertex v to vertex w, that signifies that the piece of v can capture the piece of w. This way, the vertices taken together model the current state, while the edges model possible actions. We define $out_s(p)$ as the number of pieces piece p is currently attacking in state s, which is equal to the number of outgoing edges of the vertex v that corresponds to p. Similarly, we define $in_s(p)$ as the number of pieces that are currently attacking p in state s, which is equal to the number of incoming edges. These metrics prove to be useful later on.

For clarification, the alternative graph representation of the board configuration and state in Figure 4 is illustrated in Figure 5

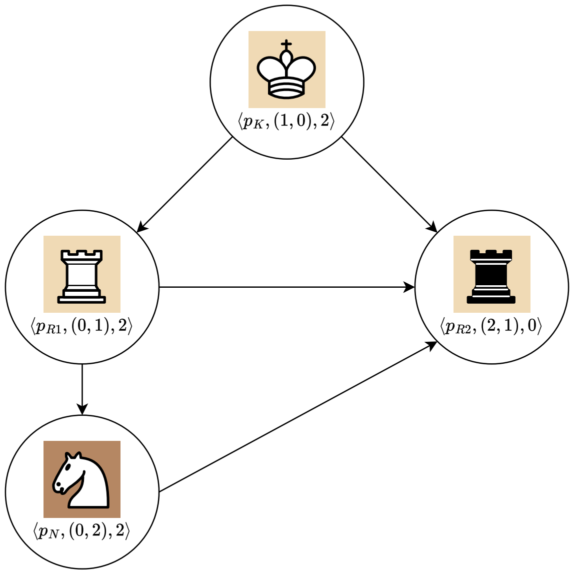

$\langle p_K, (1,0), 2 \rangle$
$\langle p_{R1}, (0,1), 2 \rangle$
$\langle p_{R2}, (2,1), 0 \rangle$
$\langle p_N, (0,2), 2 \rangle$

Figure 5: Graph Representation Example

3 Approach

Using our representation of a Bounded Classical Planning Problem formulated in the previous chapter, we can now focus on approaches for solving the problem of Solo Chess.

3.1 Enumeration

Since generalized Solo Chess is NP-hard [8], as discussed in Section 1.3, there probably does not exist a polynomial time algorithm to solve it. While it is the case that generalized Solo Chess is polynomial time verifiable, by simply applying a proposed plan $\pi = a_0, ..., a_{n-2}$ and checking whether a goal state is reached, efficient algorithms that can construct such a plan from scratch do not exist.

This is in accordance with the complexity of Bounded SATplan [10], which asks for a classical planning problem whether there is a solution of length at most k. Bounded SATplan is in NP for many specific domains, such as blocks world on n blocks, a problem that shows similarities to Solo Chess.

Since no efficient algorithm exists but solutions can be verified quickly, a simple and naive solver may resort to enumerating all possible plans $\pi = a_0, ..., a_{n-2}$ and checking for each of them whether it yields a valid solution. However, one thing to keep in mind is that in the domain of Solo Chess the possible action space depends on the current state, meaning that some action valid in one state may not be valid in another. Therefore, enumerating all possible sequences of actions in such a brute-force way is unnecessarily expensive due to the fact that when an action a_i in the sequence is invalid, all plans that can be constructed by extending this sub-plan will not yield a solution. For example, the action that prescribes the capture of a Pawn by a Knight is invalid in all states in which (1) the Pawn is not within reach of the Knight, (2) the Knight's position has been fixed, or (3) the Knight or Pawn is absent due to a previous capture. Each plan that consists of such an action is bound to fail, no matter all succeeding actions, and thus is not worth extending further. This inefficiency can be solved by constructing plans in order. This way, the problem becomes a search problem: we can search from the initial state through the space of states (only consisting of states that can be reached by performing valid actions), looking for a goal, while keeping track of the plan so far [10]. This can be done in a breadth-first manner, see Algorithm 1.

Algorithm 1 *Breadth-First Search*

```
1: create queue
2: enqueue (initial state $s_0$, empty plan $\pi_0$)
3: while queue is not empty do
4:     (state $s$, plan $\pi$) ← dequeue
5:     if $s$ is a goal state then
6:         output $\pi$
7:     end if
8:     for all actions $a$ in $A(s)$ do
9:         state $s \leftarrow f(s, a)$
10:        plan $\pi \leftarrow Append(\pi, a)$
11:        enqueue $(s, \pi)$
12:    end for
13: end while
```

By only choosing actions that are valid in line 8, there will not be any invalid plans constructed. Non-solutions will simply never reach a goal state because they are cut off when there is no action possible to extend it. To optimize memory usage, plans can be traced back afterwards when a goal state has been found by only storing the action that caused the state transition in line 9.

The breadth-first search algorithm outputs all possible solutions. In order to do this, it goes through the whole state space and explores every state that can be reached from the initial state. While it is nice to get every possible solution, such a complete exhaustive search is very expensive. When solving Solo Chess puzzles, finding a single solution suffices. This philosophy is reinforced by the fact that there is no such thing as an optimal plan: every solution consists of $n-1$ captures and all costs that could be assigned to moves are uniform, hence there is no solution that can be regarded superior to any other.

3.2 Backtracking

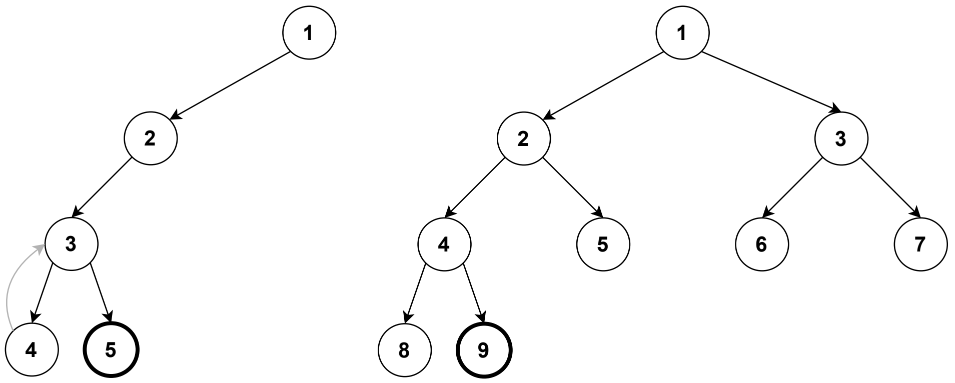
Another approach would be to search through the state space in depth-first order using a general purpose algorithm called backtracking. Extensively studied, backtracking has been around for a long time and has been used as a strategy for solving many combinatorial problems. For the reasons that it may be very inefficient in worst case situations and a systematic analysis of efficiency is very difficult to begin with, it is often regarded as a last resort. Nevertheless, backtracking algorithms are widely used, especially on problems that are NP-complete [11].

With this technique, a single subplan $\pi = a_0, ..., a_i$ is constructed until there is no valid action a_{i+1} that can extend the plan. In that case the algorithm backtracks to a previous state and tries another valid action that has not been explored yet, possibly backtracking again if there is no such action. See Algorithm 2.

Algorithm 2 *Backtracking*(state s, plan π)

```
1: if $s$ is a goal state then
2:     output $\pi$
3: end if
4: for all actions $a$ in $A(s)$ do
5:     state $s \leftarrow f(s, a)$
6:     plan $\pi \leftarrow Append(\pi, a)$
7:     $Backtracking(s, \pi)$
8: end for
```

Again, we initialize the algorithm with the initial state s_0 and an empty plan π. The advantage of this approach over Breadth-First Search (BFS) is that unnecessary exploration of states can be avoided by terminating as soon as a solution has been found, see Figure 6. In the worst case backtracking takes the same amount of time as BFS, but in practice this rarely occurs, especially since Solo Chess puzzles often have multiple solutions.



(a) Searching in depth-first order, backtracking once before finding the goal state

(b) Breadth-first search visits more states before finding the goal state. This effect increases as the depth and branching factor of the tree increases

Figure 6: Example of a depth-first and breadth-first search tree

3.3 Heuristic Search

Both the enumeration approach and the backtracking approach are domain independent. By using domain specific knowledge, we can potentially speed up the process of finding a solution. Best first search, which is an informed search strategy, uses an evaluation function to decide which path looks promising to explore first, which has been shown to speed up the search process [12, 13]. In our case, this applies to the choice of which action to perform in a given state. In Solo Chess, certain captures may look more promising that others: capturing a Pawn with a Queen seems more beneficial than the other way around. However, this does not give any guarantee that such a move leads to a solution, nor does it include any distinctive information about how close the new state is to a goal state. Therefore, we use the term heuristic when describing such an evaluation function. In heuristic search, the search process is guided by heuristic functions, unlike the uninformed search of the BFS and Depth-First Search (DFS) algorithms in which the visiting order of immediate succeeding states is not dependent on how favourable they seem.

The heuristic search algorithm is almost identical to the previous backtracking algorithm, with the addition of the possible action space $A(s)$ being sorted in decreasing order of heuristic value. This means that each move (p, p') valid in some state is evaluated and scored by some heuristic function h. For simplicity, we define our heuristics here in such a way that the higher the h-value, the "better" the move. As a consequence, the most promising actions are attempted first. If some actions have the same heuristic value, they are ordered randomly. By incorporating heuristics, the general backtracking algorithm is tailored to the particular application of Solo Chess. To come up with qualitative heuristics, certain knowledge and intuition regarding the solving of Solo Chess puzzles is needed. We propose the following three heuristics: The `Rank`-heuristic, the `Attack`-heuristic and the `Center`-heuristic.

3.3.1 Rank Heuristic

By using predetermined ranking values for each piece type, each move can be evaluated by comparing the attacking piece and the attacked piece. It should come as no surprise that rank of a piece type is directly related to the strength of the piece. Standard chess already uses a weighting scheme for this [14], which we modified to get the values in Table 1.

Piece Type	Chess Rank	Solo Chess Rank
Queen	9	9
Rook	5	5
Bishop	3	*5*
Knight	3	3
Pawn	1	1
King	-	*99*

Table 1: Ranking of piece types

We adjusted the ranking scheme of standard chess to increase the rank of the Bishop from 3 to 5 in order to achieve the equality between Rooks and Bishops in Solo Chess. In standard chess, Bishops are ranked lower since they are stuck to one color complex (a white square Bishop can never end up on a black square), while the Rook can essentially end up everywhere. However, in Solo Chess all of the pieces are restricted to the possible directions in which they are able to capture a piece. Since both Rooks and Bishops can move along two axes, we treat them as equal regarding strength. We also added a ranking for the King piece, which we set to be 99 to reflect the importance of the King. The exact ranking is not important here, we simply want the King under all circumstances to be treated as being much more valuable than any other piece, since each puzzle only has at most one King piece with which the last capture must be made. The reason standard chess does not have a ranking for the King is because it cannot be traded with, for which the ranking was originally intended.

In the ranking evaluation function of a capture, we want to incorporate that (1) capturing a low ranked piece is favoured over capturing a high ranked piece, and (2) capturing with a low ranked piece is favoured over capturing with a high ranked piece. The reason for the latter is that the number of captures for each piece is limited, and we want to prevent highly ranked pieces from using up all their captures and becoming fixed prematurely. However, (1) is more important than (2), since we would rather waste one capture of a Queen than to have the Queen taken off the board entirely.

The total ranking score of a state s can be calculated as follows:

$$rank_{total}(s) = \sum_{p \in P} \Big(rank(type(p)) \cdot nCapturesLeft(p) \Big)$$

To evaluate a move (p, p'), in which piece p captures piece p', we can take the difference between the total rank of the state s_t before the move and the total rank of the state s_{t+1} after the move. We have that

$$\begin{aligned} rank_{total}(s_t) - rank_{total}(s_{t+1}) &= \sum_{piece \in P_t} \Big(rank(type(piece)) \cdot nCapturesLeft(piece) \Big) \\ &\quad - \sum_{piece \in P_{t+1}} \Big(rank(type(piece)) \cdot nCapturesLeft(piece) \Big) \\ &= rank(type(p)) + nCapturesLeft(p') \cdot rank(type(p')), \end{aligned}$$

since the total ranking decreases because of p using one of its captures and p' disappearing from the board. We want to minimize this difference as it seems beneficial to have the ranking stay as high as possible. This comes down to the following heuristic function:

$$h_{rank}(p, p') = \frac{1}{rank(type(p)) + nCapturesLeft(p') \cdot rank(type(p'))},$$

which perfectly incorporates (1) and (2). Note how when a piece p' is fixed (meaning its number of captures left equals zero), it does not have any additional costs associated with capturing it, no matter the type of the piece. In that case, only the rank of the capturing piece p matters. An example of the Rank-heuristic can be found in Figure 7(a).

3.3.2 Attack Heuristic

Instead of looking at the rank of the type of a piece to determine its strength, we can also define it based on its relative position. It seems beneficial when a piece is in the position of having a lot of options for possible attacks, meaning that the piece is flexible, compared to when a piece only has limited options to capture and is essentially secluded. We want to incorporate that (1) capturing a secluded piece is preferred over capturing a flexible piece, and (2) the piece that captures should gain flexibility rather than lose it. Using the definitions and alternative graph representation formulated in Section 2.5, we can calculate the total number of possible moves in a state s as

$$|A(s)| = \sum_{piece \in P} out_s(piece)$$

By calculating the difference between the total number of possible moves in the state s_t before the move (p, p') and the total number of possible moves in the state s_{t+1} after the move, we have that

$$\begin{aligned} |A(s_t)| - |A(s_{t+1})| &= \sum_{piece \in P_t} out(piece) - \sum_{piece \in P_{t+1}} out(piece) \\ &= out_{s_t}(p) + in_{s_t}(p) + out_{s_t}(p') - out_{s_{t+1}}(p), \end{aligned}$$

which perfectly encapsulates (1) and (2) when we minimize this difference. Hence, by multiplying with -1, we get the following heuristic function:

$$h_{attack}(p, p') = out'(p) - out(p) - out(p') - in(p),$$

with $out'(p)$ the number of possible attacks for p after the potential move. The value of $h_{attack}(p, p')$ is prone to be negative because of the fact that after a capture there is one piece less on the board and the possible action space is likely to shrink. However, this is no problem, since higher values (less negative) are still regarded as more favourable. An example of the Attack-heuristic can be found in Figure 7(b).

3.3.3 Center Heuristic

Another possible strategy to solve Solo Chess puzzles would be to try and move every piece towards the center as much as possible. When all pieces are within the same local board region, all captures happen within that region and configurations in which certain pieces are secluded are likely to occur less. Instead of focusing on piece types or number of possible attacks, for this heuristic only the square positions of the pieces are needed:

$$h_{center}(p, p') = dist(p, p') + dist(p, center),$$

where $dist(a, b)$ is simply the euclidean distance between $square(a)$ and $square(b)$, which can be calculated using the Pythagorean theorem. We take into account (1) the distance between piece p and p', so that moves over a greater distance are preferred, and (2) the distance between p and the *center*-square, so that moves from further away are prioritized. Note how the euclidean distance, due to the squaring in the formula, does not explicitly use the sign of the direction of the moves. This is convenient in our case, since we do not only want to stress on moving towards the center: this would lead to pieces moving to the center immediately, leaving secluded pieces that cannot move to the center on their own (such as a Pawn) behind. In our heuristic, moves away from the center can also be evaluated highly (see (1)), but only if the pieces were not near the center to begin with (incorporated in (2)). This way, a piece that is far away can also be promoted to capture another piece that is even further away.

One thing left to determine is the *center* itself. When there is a King present in the start configuration of a puzzle instance, it seems logical to let the center correspond to its square, since the last capture has to be made by the king and this is likely to happen close to this area. Another option would be to pick the square that is under attack the most in the start configuration. The fact that most pieces can reach that square seems to indicate that this square is important and serves as a good center point. This way, the *center*-square can be seen as some kind of gravitational center of the board. This last option is a good alternative for when a King is absent. Efficiency-wise, the center only has to be determined once for each puzzle instance at the start of the algorithm. An example of the Center-heuristic can be found in Figure 7(c).

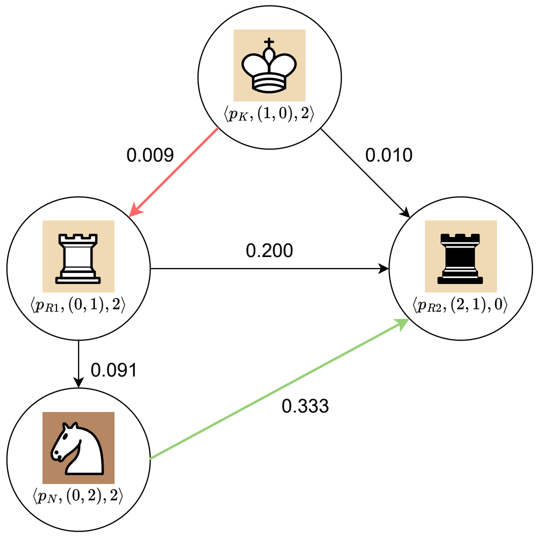
There are many more possible heuristics to think of, under which possible combinations of the proposed heuristics above. With these three heuristics, we tried to incorporate three key characteristics that determine whether a Solo Chess state is favourable or not, namely (1) what piece types are on the board and what are their strength, (2) how many possible captures are there and how are they distributed over the pieces, and (3) how centralized are the positions of the pieces.

3.4 State Space Reduction

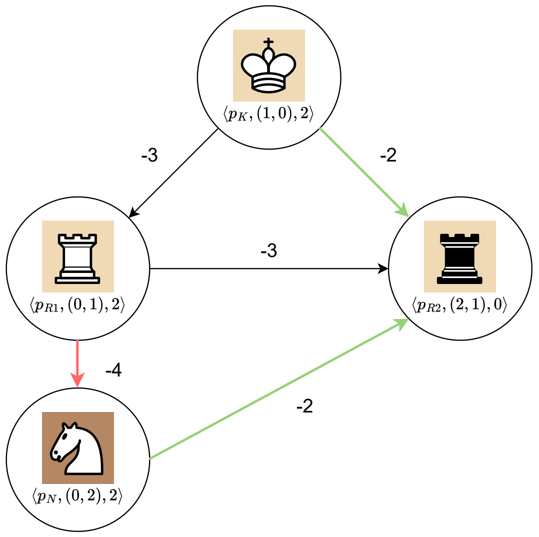
To reduce the large state space and speed up solution finding, we consider the following additional steps which we apply to our backtracking algorithm, no matter the heuristic:

- The King cannot be captured. Since the King should be the last piece standing, there is no solution in which the King is captured.
- The King cannot use all of its number of captures left and become fixed unless it is in the last move, for the same reason as described above.
- We can stop once we know for certain that a specific non King-type piece cannot be captured in succeeding states. This happens when there is a secluded piece that has no occupied square within its region from which any of all piece types currently on the board would be in the position to capture it. This can be seen as some kind of *forward-checking* [10]. For its implementation, we refer to our code, which is referenced in Section 4.2.1.
- We keep track of already visited states to prevent unnecessary double work using a HashSet.

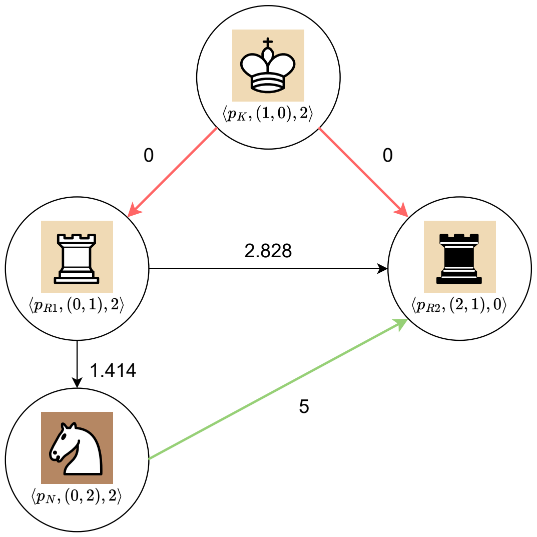
By applying these rules, we can prune branches early and save some time and number of backtracks without losing the possibility of finding a solution, since exploring these branches would indisputably lead to a dead-end. In all other cases, we only backtrack once there is no valid move possible.


$\langle p_K,(1,0),2\rangle$
0.009
0.010
$\langle p_{R1},(0,1),2\rangle$
0.200
$\langle p_{R2},(2,1),0\rangle$
0.091
0.333
$\langle p_N,(0,2),2\rangle$

(a) The `Rank`-heuristic prefers capturing the fixed Rook with a low ranked Knight, and rejects capturing with the highly ranked King.


$\langle p_K,(1,0),2\rangle$
-3
-2
$\langle p_{R1},(0,1),2\rangle$
-3
$\langle p_{R2},(2,1),0\rangle$
-4
-2
$\langle p_N,(0,2),2\rangle$

(b) The `Attack`-heuristic prefers capturing the secluded Rook with the King or the Knight, which maximizes the number of edges in the succeeding state, and rejects moving the non-fixed Rook, since that would cause the greatest loss in flexibility.


$\langle p_K,(1,0),2\rangle$
0
0
$\langle p_{R1},(0,1),2\rangle$
2.828
$\langle p_{R2},(2,1),0\rangle$
1.414
5
$\langle p_N,(0,2),2\rangle$

(c) The `Center`-heuristic prefers moving the far away Knight closer to the King, and rejects moving the King away from the center.

Figure 7: Our heuristic functions applied to the mid-puzzle state example of Figure 4. Note how the ordering of moves can be quite different between the approaches. However, they all recommend (green) a move that leads to a solution, namely the Knight taking the fixed Rook, and all rejected moves (red) would lead to a dead-end.

3.5 Generating Puzzle Instances

In this chapter, we have mostly talked about approaches for solving Solo Chess puzzles. However, we first need to generate instances to solve. For our reseach, the following two things are essential:

1. The puzzles are guaranteed to be solvable, meaning there must be at least one solution.
2. The start configuration of the instances are randomly constructed in an uniform way, such that there is no bias that inherently affect the workings of our heuristics.

We achieve (1) by working backwards from a solution, meaning we start with only a King piece on the board (or any other piece if a King is not desired in the puzzle) and expand this by adding a piece each step until we have the desired number of pieces n on the board. At every step, the expansion is chosen uniformly from all possible expansions in order to accomplish (2). The expansion step is as shown in Algorithm 3.

Algorithm 3 *Expand*(p)

```
1: $s_{old} \leftarrow square(p)$
2: Place $p$ on new random square from which $p$ can reach $s_{old}$
3: Generate new piece $p'$ with random $type(p') \in \{$♕, ♖, ♗, ♘, ♙$\}$
4: Place $p'$ on $s_{old}$
```

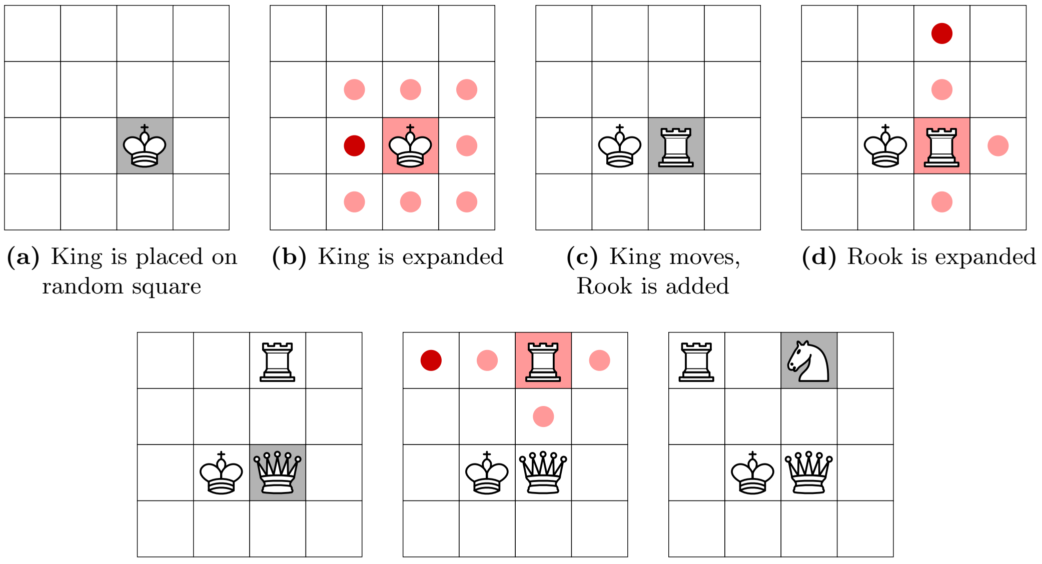
In line 2, the new square on which p is placed is chosen uniformly from all possible unoccupied squares from which a piece of type *type*(p) can reach the square on which p was standing before. In line 3, the type of p' is chosen uniformly from all piece types, with the exception of the King-type. The full generation algorithm comes down to $n - 1$ expansions, see Algorithm 4

Algorithm 4 *GenerateInstance*(n)

```
1: Place the King piece on a random square
2: repeat
3:     Pick a random piece $p$ currently on the board that is expandable
4:     $Expand(p)$
5: until there are $n$ pieces on the board
```

In line 3 of the full generation algorithm, a piece p is uniformly chosen from all pieces on the board in the current iteration that are *expandable*, meaning they are not stuck. A piece can become stuck if all possible squares to move to (in line 2 from Algorithm 3) are already occupied by some other piece or when it has reach its maximum number of expansions (which coincides with the desired number of captures left). In the first iteration, the King will be expanded, since this is the only piece on the board at that time.

The solution of the generated instance comes down to the series of expansion in reverse order, where each expansion is undone. An example with the first few iterations of the generation algorithm is illustrated in Figure 8.



(a) King is placed on random square **(b)** King is expanded **(c)** King moves, Rook is added **(d)** Rook is expanded

(e) Rook moves, Queen is added **(f)** Rook is expanded **(g)** Rook moves, Knight is added

Figure 8: The first few iterations of the instance generation algorithm

If $n = 4$ was desired, the algorithm would terminate at (g). The corresponding solution would then be: Rook takes Knight, Rook takes Queen, King takes Rook. If we desire that the maximum number of captures for the Rook is two, we make sure that it cannot be expanded more than two times, thus fixing the Rook to its place in (g). For the full implementation of the instance generation algorithm, we refer to our code.

4 Methodology

After discussing all the theory regarding our problem representation and possible approaches for solving Solo Chess, we now want to specify our methodology for answering our research questions. As stated in Section 1.4, the aim of this paper is to investigate solving Solo Chess using an algorithmic approach and to determine which tactics help in doing so.

4.1 Hypothesis

As discussed in the previous chapter, a backtracking algorithm seems to be the best choice for a Solo Chess solver. The different tactics to solve the game can then be incorporated by the algorithm in the form of the proposed heuristics from Section 3.3.

4.1.1 Comparing on the number of pieces

We expect that the use of any of these three heuristics speeds up the process of finding a solution, since they take certain intuition regarding the game into account. As a consequence, the search turns into informed search, which has been proven to be more effective in related research than its uninformed counterpart, as discussed in Section 3.3.

We anticipate that the effect of the heuristics increases as we increase the number of pieces on the board. Generally, with more pieces on the board, the search space expands and guidance is more useful. We especially think that the effect of the `Attack`-heuristic improves as n increases, since this heuristic makes direct use of the number of possible moves for each piece. The `Rank`-heuristic will probably be more consistent in its performance over different values of n, since the ranking of the pieces is a static metric that does not change. However, we have no clear indication of which approach works best in general.

4.1.2 Comparing on the piece type distribution

`chess.com` uses the number of pieces in the start configuration to classify puzzles into different difficulty levels. Even though this categorisation seems fitting in some way because of the mentioned state space expansion (more options to consider, more steps to think ahead), we argue that this should not be the only factor taken into account when determining a puzzle's difficulty level. For example, one could find oneself struggling more with a puzzle of 6 pieces than with another consisting of 8 pieces, for a number of different reasons. We also want to take into account the piece types on the board: a puzzle consisting of mostly queens seems more easily solvable than a puzzle with only pawns and knights. To quantify this, we could take into account certain metrics such as the total, average, median or standard deviation of the ranking of the pieces on the board. The reason the latter can be useful is to distinguish between puzzles with the same total ranking that are realised in a different way. For example, a puzzle with 2 Pawns and 2 Queens is totally different than a puzzle with 4 Rooks, even though they have the same total and mean ranking of the pieces.

We propose to look at the distribution of the piece types, using their ranking, to categorize puzzle instances. Unlike in the formulation of the `Rank`-heuristic in Section 3.3.1, we discard the ranking of the King and the number of captures left when talking about the ranking of puzzles in this section. While the King and the number of captures left was necessary for evaluating specific moves, it has no distinctive quality when comparing the distribution of piece types between the start configurations of puzzle instances.

A useful metric we can use here, which takes all factors such as the mean, median and standard deviation into account, is the *skewness*. Our puzzles are generated in a uniform way (see Section 3.5), meaning that the expected ranking of the pieces for a specific puzzle instance is $(1 + 3 + 5 + 5 + 9)/5 = 4.6$. The ranking of multiple puzzles therefore follows a normal distribution. However, the distribution of a puzzle itself is likely to be skewed, meaning that the ranking of the pieces are not evenly distributed and cluster more towards one side of the scale than the other.

The *sample skewness* is computed as the Fisher-Pearson coefficient of skewness [15], i.e.

$$g_1 = \frac{m_3}{(m_2)^{3/2}},$$

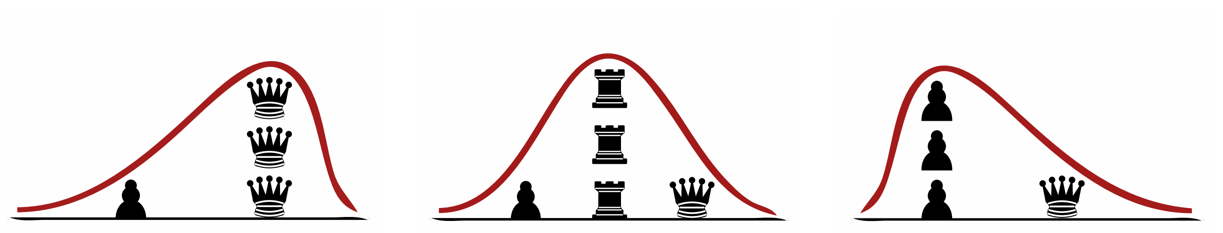
where

$$m_i = \frac{1}{N}\sum_{n=1}^{N}(x[n] - \bar{x})^i$$

is the biased sample ith central moment, and $\bar{x}$ is the sample mean. Moments are popularly used to describe the characteristic of a distribution, where the moments of a random variable of interest X are defined as the expected values of X: $E(X)$, $E(X^2)$, $E(X^3)$, and so on [16].

In our case, the x inserted into this formula corresponds to a list of the rankings of the types of the pieces on the board in the start configuration of a puzzle instance. For example: for a puzzle with one Pawn, one Knight and four Queens, thus $x = [1, 3, 9, 9, 9, 9]$, we have $g_1 \approx -0.8$. We distinguish three cases for interpreting this value:

- **High negative skew** ($g_1 < -0.5$): A left-skewed distribution in which most values are clustered around the right tail of the distribution while the left tail of the distribution is longer. For a Solo Chess puzzle, this means that there is a high frequency of some high ranked piece types and a low frequency of some low ranked piece types. See Figure 9(a).
- **Low skew** ($-0.5 \leq g_1 \leq 0.5$): A low-skewed, or zero skew, distribution is more symmetrical and is more similar to a normal distribution. For a Solo Chess puzzle, this means that either the piece types are (almost) evenly distributed, or all piece types on the board are of the same kind. See Figure 9(b).
- **High positive skew** ($g_1 > 0.5$): A right-skewed distribution in which most values are clustered around the left tail of the distribution while the right tail of the distribution is longer. For a Solo Chess puzzle, this means that there is a high frequency of some low ranked piece types and a low frequency of some high ranked piece types. See Figure 9(c).



(a) High negative skew for $x = [1, 9, 9, 9]$ **(b)** Zero skew for $x = [1, 5, 5, 5, 9]$ **(c)** High positive skew for $x = [1, 1, 1, 9]$

Figure 9: Examples of skewness in Solo Chess puzzles

Regarding our heuristics, we expect that the `Center`-heuristic is most effective in puzzles with a high negative skewness coefficient. In these puzzles, most pieces are highly ranked and thus the pieces on the board are mostly Queens, Rooks and Bishops, which are exactly the "sliding pieces" that can cover the most distance and can be moved to the center the best. The `Attack`-heuristic is also anticipated to work better in cases of negative skew, since the high ranked pieces often have more potential for possible capture options. The `Rank`-heuristic is expected to work well for puzzles with much variety in piece types, since this increases the range for the heuristic values. This corresponds to the zero skew distribution. While the cases of zero skew in which puzzles only consist of pieces of the same type may contradict this, we want to stress that these are exceptional cases and most distributions of zero skew consists of multiple piece types.

4.2 Experimental Design

4.2.1 Code

In order to test our hypothesis, we have implemented the game of Solo Chess in the programming language `C#`. The project consists of five main modules:

- `Generator(`n, m`)`: Generates m instances of Solo Chess puzzles with n pieces in the start configuration, as described in Section 3.5. The generator outputs all instances in a separate textfile, using the format for instances proposed in Definition 2.1.
- `Puzzle`: a class which keeps track of all information regarding a puzzle's state, such as its pieces (including piece type, position, number of moves left), possible moves, etcetera. A `Puzzle` can be parsed from a textfile of a specific instance.
- `Solver(`*Puzzle*`)`: Constructs a solution for a given `Puzzle`. This module contains the actual backtracking algorithm, including options regarding the usage of heuristics. It outputs a valid plan π, a sequence of moves that reach a goal state (see Definition 2.5), to the console and/or to a textfile.
- `Tester`: Handles testing the solver algorithm on a set of generated instances. It keeps track of important information and outputs the results for our experiment, see Section 4.2.3.
- `Visualizer`: Visualizes game play and solving. It is the front-end of the program, providing a graphical user interface for playing the game and simulating solution-finding. While it is not concerned with the internal algorithms that make it possible to play and solve the game, it can be very useful for debugging, solution verification, and investigating specific instances that stand out during testing.

Naturally, there are many more modules contained within the full project, such as a class `Piece`, whose subclasses handle the movement of each piece type. However, they are confined within one of the main modules specified above and their implementation is not relevant to the experimental setup. For more information, we refer to our project repository on `GitHub`[1].

4.2.2 Comparison Metric

In order to compare the effect the different heuristics have, we need a comparison metric. Often when testing efficiency, time is an important factor for indicating how fast some algorithm runs.

[1]Our full project, including code implementation and analysis of results, can be found at our GitHub repository on `https://github.com/Sverlaan/SoloChess`

However, total processing time (for instance, measured in milliseconds) is machine-dependent, since CPU's differ in clock speed, and even external conditions such as temperature can influence it immensely. Consequently, our results would be more difficult to replicate.

In addition, the processing time depends on the implementation, which is not a desirable factor. To illustrate, it may be the case that non-heuristic randomized backtracking is faster time-wise for certain small input sizes than some heuristically guided version, because it can be implemented more efficiently and therefore has some overhead to compensate for searching through more states. This is likely to occur, considering the fact that calculating heuristic values and ordering moves based on these values naturally take more time than not doing it at all. Different heuristics also differ in their implementation: for some evaluation functions, more information is needed and/or acquiring the necessary information is more expensive, compared to others. However, this does not mean that efficient implementation is not important. When testing on thousands of Solo Chess puzzles, time is a valuable resource and saving time in any way can allow us to test to a greater extent and achieve more reliable results. This especially makes a difference for puzzles with a high number of pieces on the board, since the search space can become extremely large.

A metric that we use that does not have any of these drawbacks is the number of backtracks. By counting the number of backtracks during the construction of a solution, which is proportional to the number of explored states, we can acquire some insight into how well a backtracking approach works. When a solver algorithm only needs a few backtracks to find a solution to a puzzle, it indicates that the choices made were fruitful. Conversely, when many backtracks are needed, it indicates that the used strategy was not as effective.

4.2.3 Testing

Our approach to testing is as follows: first, we randomly generate 1000 solvable instances for each number of pieces in the start configuration between 4 and 14. In order to keep testing conditions equal as much as possible, we restrict ourselves to puzzles with:

- 8×8 board size,
- the maximum number of captures for each piece is 2,
- exactly one King present in the start configuration.

We specifically choose the first two specifications, since this is the official way the game is played on `chess.com`, and the third specification for simplification. Nevertheless, we argue that experimental findings can be generalized to different board sizes and configurations with a different number of maximum captures and King pieces present and we want to stress that our approach and heuristics are not entirely dependent on these exact specifications. A few nuances are considered when discussing the testing results.

Results from testing on puzzles with $n < 4$ does not provide much insight, since the number of possible moves is very restricted in those cases. Take for instance a configuration with only two pieces, then there is only one move possible, which is always immediately the solution. On the other hand, testing on puzzles with a much higher number of pieces becomes infeasible to do within limited amount of time. For each of these $11,000$ instances, we keep a record of the number of pieces n and the skewness of the distribution of the piece types in the start configuration.

Secondly, we solve each puzzle using all three heuristics separately while keeping track of the number of backtracks each time. To set a benchmark that serves as a point of reference, we also solve each puzzle using the backtracking version without any sort of guidance. In order to remove any bias, we randomize this backtracking approach by choosing each move uniformly from the set of all possible moves in each state. We refer to this approach as *randomized backtracking.* Due to its stochastic behaviour, the results (the number of backtracks) gathered from randomized backtracking can fluctuate a lot. To compensate, we let the randomized backtracking algorithm solve each puzzle a hundred times additionally and average the results.

Lastly, we combine and analyse the results from all four approaches by using `Python` and its packages for statistical analysis. Full coverage can be found in the Jupyter Notebook uploaded on our GitHub.

5 Results

In this chapter, we discuss the results gathered from the testing phase.

5.1 Comparing on the number of pieces

When we group the puzzle instances in our test set by the number of pieces n in the start configuration and average the number of backtracks it takes to solve the puzzles, we get the values in Table 2 and the graphs in Figure 10.

n	Random	Rank	Attack	Center
4	1.0	0.7	1.2	0.2
5	4.2	2.7	4.2	0.9
6	14.1	9.2	13.8	4.0
7	43.7	28.0	41.9	15.5
8	137.5	85.0	129.6	65.9
9	435.0	291.5	388.9	217.1
10	1,375.8	828.9	1,215.0	724.0
11	3,988.4	2,615.4	3,492.7	2,315.8
12	11,373.4	6,438.8	10,164.8	6,995.0
13	34,498.8	19,442.7	28,919.3	29,873.2
14	96,643.0	66,013.4	83,419.6	82,283.9

Table 2: Total number of backtracks for each heuristic, grouped by n

The absolute counts in the table and the graphs give some sense of scope regarding the number of backtracks it takes to solve a puzzle and how much it grows with each piece. While a puzzle instance with $n = 4$ generally takes only one backtrack, a puzzle with $n = 14$ takes almost $100,000$ backtracks in the case of randomized backtracking. With every piece, the number of backtracks seems to increase by a factor between three and four, which coincides with the exponential growth of the number of states and possible moves discussed in Chapter 3. The fact that the number of backtracks grows exponentially with n can be seen even better when we scale the y-axis logarithmically in the lower plot of Figure 10, where the lines are almost straight.

By dividing these counts by the average number of random backtracks, we normalize the counts it takes to solve the puzzles in each group. This gives us the *Performance Ratio* (*PR*):

$$PR_{approach} = \frac{\#backtracks_{approach}}{\#backtracks_{random}}$$

The Performance Ratio signifies the effectiveness of a certain approach, which can be interpreted as follows:

- $PR < 1$: The approach is effective, since it requires less backtracks to find a solution than random. A lower value of *PR* indicates a better performance compared to random.
- $PR = 1$: The approach is just as effective as random, since it takes the same number of backtracks.
- $PR > 1$: The approach is ineffective, since it requires more backtracks compared to random.

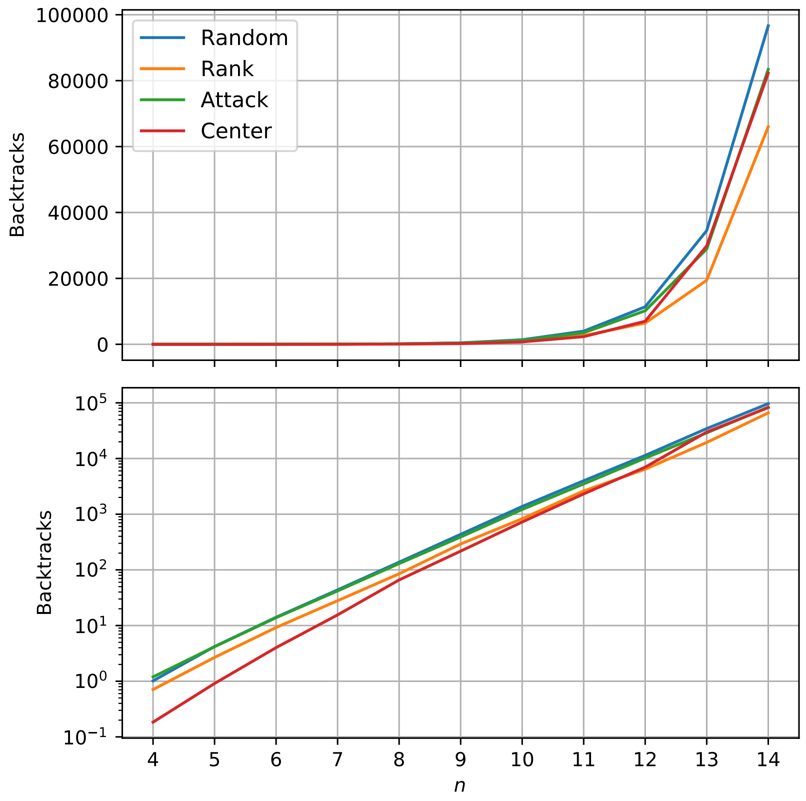


Figure 10: Total number of backtracks for each heuristic approach, grouped by n. The lower figure is log scaled.

When we plot the Performance Ratios of all approaches grouped by n, we get Figure 11. The horizontal blue line represents the performance of the randomized backtracking, whose PR value is equal to one by definition, which is used as a reference point. The other lines represent the results of the heuristic approaches, for which the PR values can be read as the percentage of backtracks it takes to solve a puzzle compared to random.

The `Rank`-heuristic seems to be consistently effective. For most tested values of n, the heuristic only takes between $55 - 70\%$ of the number of backtracks the randomized approach does, which is a great improvement. There is not much fluctuation in its performance and it does not seems to be highly dependent on n. This is because the ranking of each piece is a static metric that does not chance once another piece is added to the puzzle.

The `Center`-heuristic seems to perform even better on low valued n, with only taking around one fifth of the number of backtracks randomized backtracking does in the case of $n = 4$ and $n = 5$. However, this performance starts to decrease with each added piece. For $n \leq 11$, the Center-heuristic outperforms the Rank-heuristic, but for puzzles with more than eleven pieces it is the other way around. The Center-heuristic clearly follows a trend in which the performance decreases as n increases, with $PR = 0.85$ for $n = 14$. This can be explained by the fact that pieces on the board start to accumulate more and more once more pieces are added during the generation process (see Section 3.5). With only a few pieces on the board, there are a lot of potential squares on which a new piece can be placed. However, once the board becomes filled, the options for new pieces are more restricted, since some directions (horizontal, vertical or one of the two diagonals) may be blocked by some other piece. This causes new pieces to be placed relatively closer to the

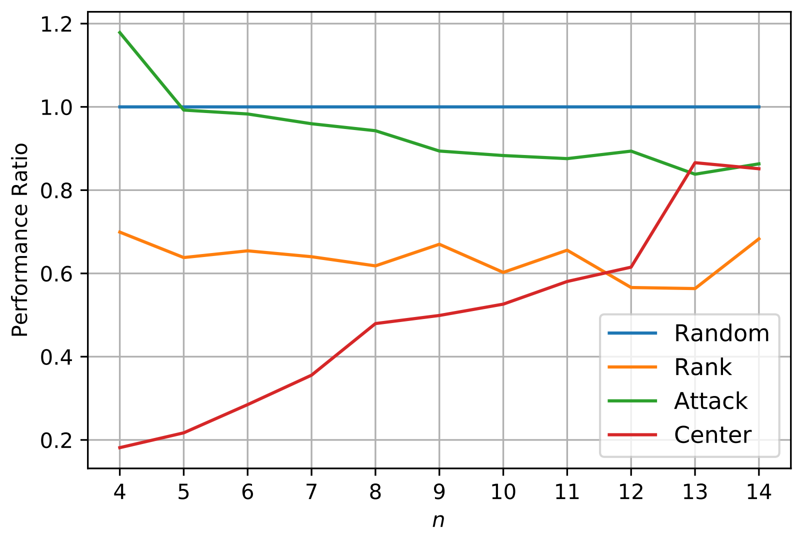


Figure 11: Performance Ratio for each heuristic approach, grouped by n.

pieces on the board than before, and so pieces start to accumulate, which in turn causes even more restriction. Because of this, pieces are less capable to move to the center and/or to move over a great distance, diminishing the factors on which the Center-heuristic is based. This effect is only enhanced by the fixed board size of 8×8, since the edges of the board also restrict placement. This factor can be removed by playing on larger (or even infinite) board sizes, which could cause the Center-heuristic to perform a bit better on puzzles with a high number of pieces.

While the `Attack`-heuristic seems to be the least effective out of all three heuristics for most n, we see that its performance gets better once more pieces are added. This is as expected, since the total number of possible moves is greater once more pieces are on the board, which in turn increases the capture options of some pieces, and this is exactly what the Attack-heuristic makes use of. A potential reason for the fact that the Attack-heuristic is the least effective out of all three heuristics could be that it keeps its options open too much. The heuristic is established in such a way that it maximizes the number of possible attacks. The idea behind this was that situations in which there are no valid moves available would occur less frequently, resulting in less backtracks. However, this could also have a downside: keeping its options open too much also leads to a higher branching factor at each level, resulting in a wider search tree and causing more backtracks in the long run for cases in which a mistake was made earlier in the solving process. Hence, the heuristic could potentially be improved by adjusting it to be more selective.

5.2 Comparing on the piece type distribution

When we look at the distribution of the ranking of the pieces for each puzzle, we can group on the *skewness coefficient* by binning the values as proposed in Section 4.1.2, see Table 3.

When we plot the Performance Ratio of each heuristic averaged per bin, we get the barchart in Figure 12.

As expected, the heuristic most sensitive to the level of skewness is the `Center`-heuristic. We hypothesized that this heuristic is at its best when there are relatively more high ranked pieces (i.e. Queens, Rooks, and Bishops), for the reason that these piece types are the most capable of moving

Degree of Skewness	Bin values	# Puzzles
Negative skew	$(-\infty, -0.5]$	1187
Low/zero skew	$(-0.5, 0.5)$	6384
Positive skew	$[0.5, \infty)$	3429

Table 3: Distribution of puzzles into skewness bins

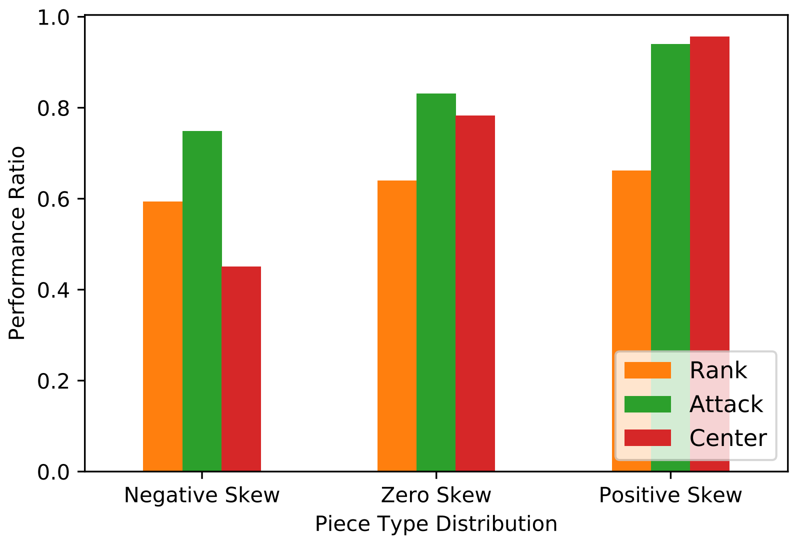


Figure 12: Peformance Ratio for each heuristic approach, grouped by degree of skewness

to the center by directly sliding into the right direction. This is the case for piece distributions that are negatively skewed, and we can see that $PR = 0.45$, which supports our hypothesis. For low and positively skewed piece distributions, the ratio of high ranked pieces in the distribution decreases and the Center-heuristic performs much worse, with $PR = 0.96$ in the case of positive skew. This way the Center-heuristic goes from being the best performing heuristic to being the worst.

The `Rank`-heuristic seems not that effected by the distribution of piece types, and has a Performance Ratio of around 0.6 for all three degrees of skewness. This indicates that the Rank-heuristic can cope with puzzles of all kinds of piece distributions, and this robustness is a nice feature. We already saw the same kind of robustness when comparing on the number of pieces in the start configuration, and the same reasoning can be applied here.

As for the `Attack`-heuristic, we see a decrease in performance as the piece distribution becomes more positively skewed, similar to the Center-heuristic only to a smaller extent. Again, this can be explained in the same way as for the comparison regarding the number of pieces n: the Attack-heuristic uses the number of possible captures per piece directly, which is more obvious in cases where there are relatively more high ranked pieces on the board. The ranking of the pieces are based on their strength, which in turn is based on the flexibility with which a piece can move. Since every move must be a capture in Solo Chess, the ranking is thus based on the number of possibilities for potential attacks. Naturally, the variety in number of possible captures increases as the ranking of the pieces on the board increases, which makes it easier for the Attack-heuristic to make distinctions when evaluating moves.

6 Conclusion

From the results in the previous chapter, we can conclude that we indeed managed to create a solution algorithm which can solve puzzle instances of the game of Solo Chess and that we gained some knowledge regarding tactics that can be applied when playing the game.

While the game is NP-complete, and an efficient algorithm for solving it does not seem to exist, we managed to design and apply a general backtracking algorithm. In order to speed up solution finding and to investigate which tactics and approaches work best, we incorporated knowledge and intuition about the game into the algorithm in the form of heuristics, which guided the search to a goal by picking which moves to consider first. The three heuristics we came up with were the `Rank`-heuristic, `Attack`-heuristic, and `Center`-heuristic. To gain any insight into the performance of these heuristics, we counted the number of backtracks it takes to solve a puzzle and compared this to a randomized approach by testing over $11,000$ puzzle instances randomly generated ourselves.

When looking at the number of pieces n in the start configuration of a puzzle, we conclude that the `Rank`-heuristic is the most consistent approach and generally performs well on puzzles with different number of pieces. It generally takes around 55-70% of the number of backtracks the randomized approach takes, which is a great improvement. However, in cases with only a small number of pieces on the board, a greater improvement can be achieved by the `Center`-heuristic. We saw that the Center-Heuristic outperforms the Rank-heuristic for puzzles with $n \leq 11$, only taking 20% the number of backtracks random does in the cases of $n = 4$ and $n = 5$. We propose that the best approach to use when solving a Solo Chess puzzle is a hybrid version in which we combine the Rank- and Center-heuristic. As long as the number of pieces on the board is greater than eleven, we recommend using the Rank-heuristic to make captures and clear the board. When the board is less crowded and the number of pieces declines to under twelve, we endorse using the approach of the Center-heuristic. When reflecting on what the underlying tactics of these heuristics actually represent, this seems like a solid approach to playing the game: while there are a lot of pieces on the board, it helps to make captures that preserve the high ranked pieces (by refusing to capture these pieces and by saving their captures left as much as possible), so that once the board is cleared more, it is exactly these sliding pieces that remain that can now focus on cleaning up any loose pieces left and moving to the center, of which they are the most capable. Eventually, only a small number of pieces are left on the board, grouped together close to the King, with the search for a solution possibly almost coming to an end.

When looking at the distribution of the piece types in the start configuration of a puzzle, the results are quite similar. The `Center`-heuristic works best on puzzles with relatively more high ranked pieces, but this performance deteriorates when the distribution shifts to more low ranked pieces. The `Rank`-heuristic is much less dependent on the piece type distribution, and has a consistent performance of around 60% compared to random for all three cases of skewness we examined.

The approach of the `Attack`-heuristic, although not bad and still a fine improvement over no tactic at all, is not found to be the best to use under any circumstance we considered, which could potentially be attributed to it being too little selective and keeping its options open too much.

Regarding the relevance to AI, we see that our conclusions demonstrate that simple, brute force approaches should not be quickly discarded. Noteworthy speed up in solution finding can be accomplished with the use of domain specific heuristics, in combination with steps taken in reducing the state space. We have shown that this can be fruitful in the case of Solo Chess, but this of course holds for all kinds of search and planning problems. Not only that, but from the investigation into the effect these heuristics have under certain circumstances, we have gained knowledge about tactics that can benefit actual humans playing the game.

7 Discussion

The conclusions we draw in the previous chapter are based on the results from Chapter 5 and should be taken with caution, especially regarding the comparisons on the skewness of the piece type distribution. For one, the set of puzzles on which we tested our approach is not generated specifically for this type of comparison. We can see that the bins in Table 3 do not contain an equal number of puzzles, and the results are sensitive to different number of bins and their sizes. Another thing to consider is that we only measure the skewness for puzzles regarding their start configuration. During solving, depending on the moves chosen, the board configuration may change a lot which also affects the skewness for succeeding sub-puzzles. However, it is important to keep in mind that the first few moves chosen have the greatest significance in determining the course of the remainder of the solution. An unfortunate capture made early in the game can lead to many more backtracks than one made later on. Hence, we should not dismiss the results gathered from only looking at start configurations.

However, while it was beyond the scope of this thesis, it may be interesting to look in more detail at the significance of the first few captures during the solving of the game. Now that we have an algorithm which can solve the game, we could do further investigation into how many solutions are possible and acquire some insights into how fast a puzzle can become unsolvable when an wrong move was made in the early stages of solution finding. In this paper, we looked at puzzles which were guaranteed to be solvable. However, further research could also examine puzzle which do not have a solution. As Aravind et al. [8] also mention, a natural optimization objective for unsolvable puzzles would be to play as many moves as possible, or equivalently, leave as few pieces as possible on the board. It would be interesting to test our heuristics under these circumstances, as another measure of performance.

The biggest bottleneck when tackling the problem of Solo Chess is the large state space associated with it. The number of options grows exponentially with the number of pieces on the board and so there seems to be a restriction to how deep we can search within limited amount of time. In our research, we did not test on puzzle instances with $n > 14$. The reason for this was that the randomized backtracking approach simply takes too much time to solve puzzles consisting of fifteen pieces for us to test this on 1000 puzzles a hundred times. Even though we could have tested on less puzzles or skip the additional hundred tries for the random approach, this would have led to more unreliable results. While we want to stress that solving one puzzle, especially using some heuristic, can be done fairly quickly, there is unfortunately no way of knowing how long it is going to take upfront. We managed to test on puzzles with $n = 14$ because of the steps we took in reducing the state space. While we implemented some obvious rules, such as the King not being able to be captured or become fixed, and keeping track of visited states, we are under the impression that some further improvements can be made here. Future research may investigate this and come up with other, although probably less intuitive and more complex, ways to restrict the number of states to be searched before finding a solution, making it possible for us to test our heuristics on puzzles with many more pieces. Especially since Solo Chess is such an understudied research topic, we think there is a lot more to explore here.

When reflecting back on the process of writing this bachelor thesis, I can only do so with much joy. During my three years of studying at Utrecht University, I have followed many courses for which I had to make fun assignments. However, none of those projects would I refer to as "my own" as much as this last one. Of course, this has to do with the level of freedom I was given regarding the topic and organisation of my thesis. When playing the game of Solo Chess, I saw a lot of opportunities for me to apply some of the knowledge learned during my bachelor and I was even more stimulated to do so once I saw that the literature regarding the puzzle game consisted of merely one paper. Looking back, I am especially content with the fact that I managed to incorporate my own approaches and intuition regarding the game into the heuristics and the fact that these seem to work. When starting this project, my main focus was simply to create an algorithm that could solve puzzle instances. However, this lacked some room for experimentation, which I did manage to find in using and comparing the heuristics. As a consequence of incorporating these heuristics, the solver algorithm can be attributed some level of intelligence, which, of course, is a must when talking AI. In addition to applying what I have learned from the past few years, I learned a lot of new things from working on the project itself too. Not only did I learn a great deal regarding the subject matter, I have also gotten better at academic writing and planning, qualities on which this process relied more than any course I have taken before and which I am glad to improve and take with me in future work.

References

[1] H.J.R. Murray. *A History of Chess*. Clarendon Press, 1913.

[2] Natalie Wolchover. *FYI: How Many Different Ways Can a Chess Game Unfold?* Posted on popsci. Accessed: 05-02-2022. Dec. 2010. URL: `https://www.popsci.com/science/article/2010-12/fyi-how-many-different-ways-can-chess-game-unfold/`.

[3] Claude E. Shannon. "Programming a Computer for Playing Chess". In: *Philosophical Magazine* 41.314 (Mar. 1950), pp. 256–275.

[4] Aviezri S. Fraenkel and David Lichtenstein. "Computing a perfect strategy for $n \times n$ chess requires time exponential in n". In: *Journal of Combinatorial Theory* Series A.31 (1981), pp. 199–214.

[5] *Stockfish*. Accessed: 05-10-2022. URL: `https://stockfishchess.org/`.

[6] David Silver, Thomas Hubert, and Julian Schrittwieser. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* (Dec. 2017).

[7] *What is Solo Chess? How do I Play?* Accessed: 05-02-2022. URL: `https://support.chess.com/article/289-what-is-solo-chess-how-do-i-play`.

[8] N.R. Aravind, Neeldhara Misra, and Harshil Mittal. "Chess is hard even for a single player". In: *CoRR* (Mar. 2022).

[9] Nir Lipovetzky. *Structure and inference in classical planning*. AI Access, 2014.

[10] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2010.

[11] Hilary Priestley and Martin Ward. "A Multipurpose Backtracking Algorithm". In: *Journal of Symbolic Computation* 18 (July 1994), pp. 1–40.

[12] Rina Dechter and Judea Pearl. "Pearl, J.: Generalized best-first search strategies and the optimality of A*". In: *J. ACM* 32 (July 1985), pp. 505–536.

[13] Richard E. Korf. "Linear-space best-first search". In: *Artificial Intelligence* 62.1 (1993), pp. 41–78.

[14] MasterClass staff. *Chess 101: All the Chess Piece Names and Moves to Know*. Posted on MasterClass. Accessed: 06-08-2022. Mar. 2022. URL: `https://www.masterclass.com/articles/chess-piece-guide#6-chess-pieces-in-standard-chess-sets`.

[15] Stephen Kokoska and Daniel Zwillinger. *CRC Standard Probability and Statistics Tables and Formulae*. 1999.

[16] Chirag Goyal. *Moments – A Must Known Statistical Concept for Data Science*. Posted on analyticsvidhya. Accessed: 06-23-2022. Jan. 2022. URL: `https://www.analyticsvidhya.com/blog/2022/01/moments-a-must-known-statistical-concept-for-data-science/`.