Playing Klondike Solitaire variants with stochastic simulations

Bachelor's thesis

Dominik Guido Milas
377332

16.10.2025

Supervisor: Prof. Dr. Benjamin Blankertz
Dr. Stefan Fricke

Technische Universität Berlin
School of Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Neurotechnology

Kurzfassung

Klondike Solitär ist eines der bekanntesten Kartenspiele. Obwohl es schon seit geraumer Zeit im Fokus der Forschung steht, wurden die bekanntesten und beliebtesten Varianten noch nicht umfassend untersucht. Bei diesen Varianten werden die Karten verdeckt ausgeteilt, wobei eine oder drei Karten aus dem Stapel gezogen werden.

Diese Asbchlussarbeit vergleicht die Ergebnisse von Monte Carlo Tree Search (MCTS) und Hindsight Optimization (HOP) zum Spielen der Varianten. Im Rahmen der Bachelorarbeit wurden eine Heuristik für Simulationen implementiert. Durch zusätzliche Zufallsknoten wurde die Unvorhersehbarkeit der verdeckten Karten abgebildet..

Die Siegrate von MCTS erreichte 63,20% für die Variante mit einer gezogenen Karte vom Stapel und bis zu 44,40% für die Variante mit drei vom Stapel gezogenen Karten. Dabei übertraf MCTS sowohl die Siegrate von HOP als auch die Ergebnisse früherer Forschungen. Auf Basis der vorgestellten Ergebnisse lässt sich sagen, dass Algorithmen in der Lage sind, zwei Drittel aller gewinnbaren Klondike Solitär Spiele der Variante mit einer gezogenen Karte vom Stapel und über die Hälfte aller gewinnbaren Spiele der Variante mit drei vom Stapel gezogenen Karten zu gewinnen.

Abstract

Klondike Solitaire is one of the most famous single-player card games. While it has been in the focus of scientific research for quite a while, the most familiar and popular variants have not been extensively investigated. These variants feature face-down cards, with one or three cards being drawn from the stock.

This thesis compares the performance of Monte Carlo tree search(MCTS) and Hindsight Optimization(HOP) playing these variants. A heuristic for rollouts and chance nodes for MCTS were implemented using domain-specific knowledge. The win rate of MCTS reached 63.20% for the one-card draw variant and up to 44.40% for the three-card draw variant outperforming HOP as well as the results of previous research. Based on the results presented, it can be said that algorithms using stochastic simulations have the ability to win two-thirds of all winnable one-card draw Klondike Solitaire games and over half of all winnable three-card draw games.

Contents

List of Figures

List of Tables

1. Introduction

Klondike Solitaire is one of the most famous single-player card games[1]. Just the digital version of Microsoft Windows alone is played over 100 million times a day[1]. It is pre-installed on all major Windows desktop versions[2]. Its popularity and complexity [2] have made Klondike Solitaire the subject of extensive research, particularly within the field of computer science. Many different approaches to understand and solve the game have been investigated. Despite this effort of the scientific community, Klondike Solitaire's uncertainty is often referred as "one of the embarrassments of applied mathematics that we cannot determine the odds of winning the common game of Solitaire" by researchers and mathematicians[3, 4, 5, 6]

Klondike Solitaire has a high complexity and a high degree of stochastic variation[2]. Because of this, even the newest research is not able to fully figure it out and currently available solvers are not able to win every winnable position. Especially because most implementations focus on the thoughtful variant of Klondike Solitaire, where all cards are revealed from the start. The standard variant is less researched, and results regarding AIs performance playing it might already be dated, especially in the face of technological advances.

In this thesis, the goal is to algorithmically win standard Klondike Solitaire games using stochastic simulation algorithms such as Monte Carlo tree search and Hindsight Optimization. By implementing these algorithms and running them on current hardware, this thesis aims to improve the results of previous research. In addition, this thesis investigates the difference of algorithmic performance when playing the two most played variants, drawing one card or three cards per draw from stock, of Klondike Solitaire.

Structure of the thesis

Following the introduction is a section that familiarizes the reader with the terms and rules of Klondike Solitaire. The relevant knowledge concerning the algorithms implemented is introduced, and related works by scientists and enthusiasts are presented.
Building on top of that, the decisions made regarding implementation are described. Adaptations such as simplifications and improvements to the algorithms used are explained. Subsequently, the setup to gather statistics as well as the critical parameters are introduced.
The subsequent section is devoted to the presentation of outcomes and gathered statistics. These are then discussed by way of contextualizing them with existing literature. Furthermore, an interpretation of the impact of key parameters is presented. After this, limitations of this thesis are explained.
The thesis closes with a conclusion that highlights the impact of this thesis and presents an outlook for future exploration.

[1] https://news.xbox.com/en-us/2020/05/22/celebrating-30-years-microsoft-solitaire/

2. Background

This section gives an overview of the relevant background knowledge. It starts with an in-depth description of Klondike Solitaire, continues with explanations of relevant algorithms, and introduces related works by scientists and enthusiasts.

2.1. Klondike Solitaire

Before explaining the rules of Klondike Solitaire, it is important to clarify any confusion about the name. Blake and Gent have stated on this topic:

> Names of particular patience games are even more confused than the terminology for rules, with different names used for the same game and the same name used for different games. For example, the game we call Klondike in this paper is often just called 'Patience' (Parlett, 1980) or 'Solitaire' which are also names for the general family of single-player card games. Worse than that, Klondike can also be called 'Canfield', which is the name we use here for a completely different game. Both games have many other names, for example both being sometimes called 'Demon'[6, p. 4].

In light of the former citation, from now on this thesis will use Klondike to refer to Klondike Solitaire.

2.1.1. Terms

In academic research, different terms are used for elements of solitaire games such as Klondike[5, 6]. This thesis provides a short definition of important terms, mostly staying in line with what Blake and Gent previously defined[6]:

- **Deck**: Klondike is played with a standard 52-card pack, referred to as deck.
- **Card**: Unique part of a deck, can be one of four different suits and one of 13 ranks.
- **Position**: A position is a concrete arrangement of cards that changes with each move.
- **Move**: A move is defined as the act of one or more cards changing their position in accordance with the established rules.
- **Foundation**: Klondike has four foundations, one for each suit. Each foundation only allows cards of one suit to be collected, and it is only possible to collect them in the order of the lowest-ranking card (Ace) to the highest-ranking card (King).
- **Stock**: Stack of cards from which you can draw cards. The drawn cards are used or placed on the waste. The waste can be reused again to refill the stock.

- **Pile**: Card placed on top of each other.
- **Build policy**: Rule set for placing cards on piles. Only Kings can be placed on empty piles. Cards placed on another face-up card must be of the opposite color and one rank lower. No cards can be placed on Aces.

2.1.2. Rules

At the start of a Klondike game, all cards from a shuffled deck are dealt in the same starting position. Seven piles are created in a triangle configuration, the first pile has one card, the second pile has two cards, etc. The last pile consists of seven cards. For each pile only the top card is face-up and the other cards are face-down. The rest of the deck is placed in one stack as the stock. The foundations are left empty. The goal of the game is to move all the cards to their respective foundations. These types of moves are allowed:

- One card from stock to a foundation
- One card from stock to a pile
- One card from a pile to a foundation
- One card from a foundation to a pile
- One or multiple cards from a pile to a pile
- Three cards from stock to the waste

Moves from stock are only allowed to use the currently drawn card, which is also the topmost card of the waste. Moves from a foundation are only allowed to use the topmost card of a foundation. Moves to a foundation can only be done following the described order for foundations. Moves to a pile have to follow the described build policy. When moving multiple cards between piles, it is only allowed to take face-up cards from one pile and place them on the other pile in the same order as taken while following the build policy. Each draw from stock is made up of 3 cards or the last cards left that are placed on the waste. All drawn cards are face-up but it is only allowed to move the topmost card of the waste. When the stock is empty, it is permissible to return the cards in the same order to the stock.

The game ends when the goal of game is achieved or when no more moves are possible, but it is more likely to stop a game because no more progress can be achieved even though moves are still possible.

2.1.3. Variations

The described rules are the most well-known variant of Klondike with many more being played. In many variants, the rules regarding the stock are different. The draw might only be one card at a time, or it is possible to move cards back from the waste to stock only a fixed number of times or even not at all. Another popular variation is called thoughtful solitaire. In this variant, the positions of all cards are known at the beginning of the game, which is practically the same as playing with open cards. One variant is often referred to as the Las Vegas variant and has historically been used to gamble. The

player pays 52$ for a game and receives back 5$ per card played to a foundation. In this variant, a win would mean that at least 11 cards have been played to the foundations[7].

2.2. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) was introduced by Coulum[8] in 2006 as a combination of tree search and Monte Carlo evaluation. In the same year, MCTS achieved its breakthrough with the development of Upper Confidence Bounds for Trees (UCT) by Kocsis and Szepesvári[8]. Using MCTS as the foundation, multiple achievements were accomplished, such as beating top human professionals at 9×9 Go in 2008[9].

Browne et al.[10, p. 6] describe MCTS as an algorithm that "can be used with little or no domain knowledge for which more computing power generally leads to better performance", and highlight its successful use "with little or no domain knowledge" for "difficult problems where other techniques have failed".

2.2.1. General MCTS Algorithm

> Monte Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results. [10, p. 1]

The General MCTS Algorithm (see Algorithm 1) is an iterative algorithm that can be divided into the four distinct phases of *Selection*, *Expansion*, *Simulation* and *Backpropagation*, which will be explained in the sections immediately following. The phases are repeated by the algorithm until the given computational budget is depleted. The computational budget can be defined as the amount of repetitions, time elapsed, CPU time, etc. During the repetitions, the performance of each available action in the current state of the game is saved and continuously updated. When the computational budget is exhausted, the repetitions are stopped and the best performing action is chosen.

Algorithm 1 Monte Carlo Tree Search Algorithm

```
1:  function MCTSSEARCH(p_0)
2:      create root node n_0 with position p_0
3:      while used computational ressources ≤ computational budget do
4:          n_x ← apply tree policy by selecting next rollout node & expanding nodes when needed
5:          r_x ←rollout from selected node n_x
6:          backpropagate rollout result r_x
7:      end while
8:      n_b ← select best action / child node of n_0 based on policy
9:      return n_b
10: end function
```

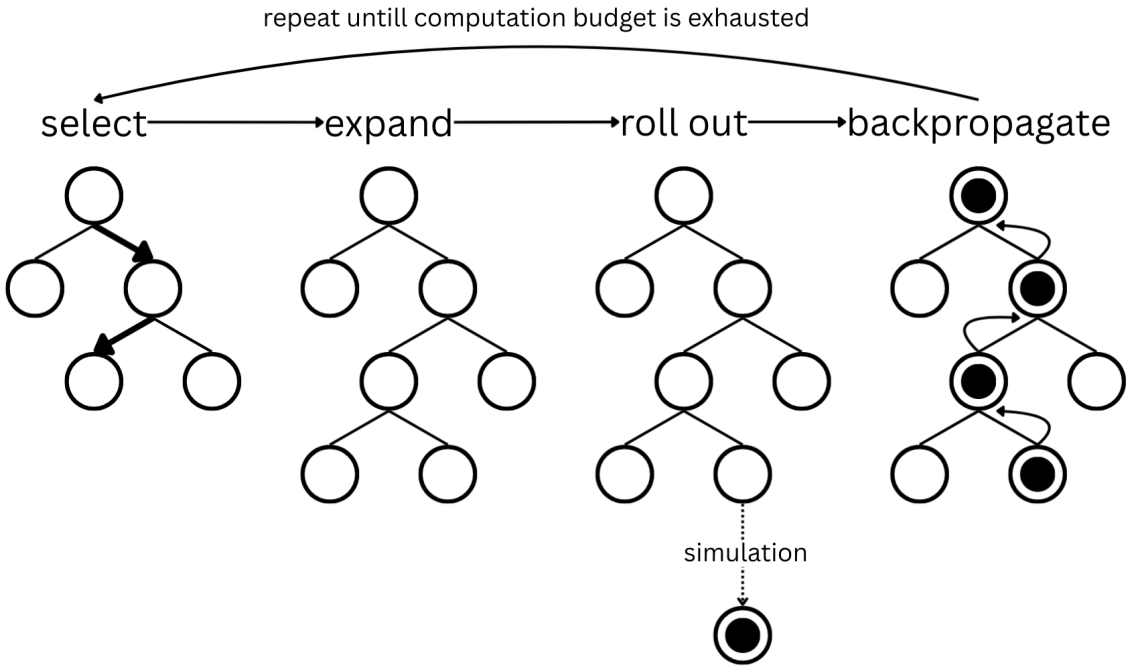

Figure 2.1.: Example of one iteration of the Monte Carlo Tree Search algorithm. Circles represent a node, thick arrows represent the selection process, thin arrows represent going up the tree during backpropagation and the filled circle represents a rollout result

As illustrated in Figure 2.1, MCTS uses a search tree composed of nodes that each represent a state of the game, for example, a position. Starting from a root node, each possible action connects the nodes with their child nodes. With each repetition, a node gets selected, is potentially expanded and the results of a simulated rollout are backpropagated up the tree. The algorithm 1 showcases the basic approach of the MCTS steps in action.

Selection

The phase of *Selection* starts in the root node. As show in algorithm 2, the function loops until it finds a node that has not been rolled out, which means it will as a starting point for a simulation. In each iteration of the loop, the algorithm traverses one more node down, choosing the child node based on a constant tree policy and expanding it when necessary. The expansion is deemed necessary when a node was selected as the best child node, was already rolled out once but has not been expanded beforehand. After leaving the loop, the last visited node is returned.

Algorithm 2 Selection

```
1: function SELECT(n)
2:     n_x ← n
3:     while node n_x already rolled out once do
4:         if n_x not expanded then
5:             expand node n_x
6:         end if
7:         n_x ← apply tree policy to n_x to select best child node
8:     end whilereturn n_x
9: end function
```

Expansion

The *Expansion* phase of MCTS adds all possible child nodes to the node when deemed necessary in the selection phase. To do this, the expand function shown in algorithm 3 first computes all available actions for the position represented by the chosen node. For each available action, the function creates a child node and then links these to the chosen node.

Algorithm 3 Expansion

```
1: function EXPAND(n_x)
2:     l_actions ← compute available actions in position p_x of node n_x
3:     l_childnodes ← generate child nodes of node n_x based on available actions l_actions
4:     connect l_childnodes to n_x
5: end function
```

Simulation

In the *Simulation* phase, starting from the position of the chosen node, a Monte Carlo simulation is started. For this, the rollout function simulates a potential course of the game by continuously playing random moves until the game terminates or another condition such as a maximum number of moves is done. As shown in algorithm 4, the function then returns the result of the rollout.

Algorithm 4 Simulation

```
1: function ROLLOUT(n_x)
2:     p_x ← position of node n_x
3:     while p_x is not terminated do
4:         p_x ← apply random action to position p_x
5:     end while
6:     r_x ← boolean representation of terminal state of p_x
7:     return r_x
8: end function
```

Backpropagation

Backpropagation is the last repeating phase of the MCTS algorithm. Starting from the node in which the rollout was done in in the *Simulation* phase, the algorithm moves to the corresponding parent node of the current node with each iteration of the loop shown in algorithm 5. For each node visited in this procedure, the visit count is incremented by 1 and in the case of a successful rollout in the *Simulation* phase the score is also incremented by one. When the root node is reached, the algorithm increments the visit count and score once more before ending the loop.

Algorithm 5 Backpropagation

```
1:  function BACKPROPAGATE(n_x, r_x)
2:      reached_root_node ← False
3:      while reached_root_node is False do
4:          add visit to visit count of node n_x
5:          if r_x = True then
6:              add win to score of n_x
7:          end if
8:          if n_x is root node then
9:              reached_root_node ← True
10:         else
11:             n_x ← parent node of node n_x
12:         end if
13:     end while
14: end function
```

Choosing the best action

When the computational budget is exhausted and the MCTS algorithm finished the repetition of the four phases *Selection*, *Expansion*, *Simulation* and *Backpropagation* the best available action is chosen by choosing the corresponding child node based on some criteria. Schadd[11] describes multiple criteria, two being the child node with the most visits (robust child) and the child node with the highest score (max child).

2.2.2. Upper Confidence Bounds for Trees

The upper confidence bounds for trees(UCT) is the ”most popular algorithm in the MCTS family” [10, p. 6]. UCT enhances the selection process by introducing a tree policy based on a formula that combines exploitation and exploration. UCT evaluates the expected payoff of the available actions by Monte Carlo simulations and treats the choice as a multiarmed bandit problem. For each selection of nodes in the tree, the child node is selected based on

$$UCT = \frac{Q(n')}{N(v')} + 2C_p\sqrt{\frac{2\ln N(n)}{N(n')}}$$

where:

- n is the current node

- n' is a child node
- $Q(n')$ is the total score of child node n'
- $N(n')$ is the count of visits to child node n'
- C_p is the contant exploration term
- $N(n)$ is is the count of visits to parent node n

$\frac{Q(n')}{N(v')}$ represents the exploitation part of the UCT equation. It is understood to be within $[0, 1]$ and, in general, a value of $N(n') = 0$ produces a UCT value of ∞[10]. This ensures that all child nodes are evaluated once before any child mode is expanded further by making the UCT value the highest possible value. A policy for ties in UCT value can be implemented, usually the policy is random selection[10].

The exploration part is represented by $2C_p\sqrt{\frac{2\ln N(n)}{N(n')}}$ and ensures that every node has a non-zero chance of being selected. The constant exploration term C_p can be chosen based on the context to adjust the amount of exploration performed. $C_p = \frac{1}{\sqrt{2}}$ was introduced by Kocsis et al. and because of its quality "to satisfy the Hoeffding inequality with rewards in the range $[0, 1]$", it is a good starting point for any MCTS implementation.

When running, there is a certain balance between the exploitation part and the exploration part of the UCT equation. By visiting a node, the exploration part decreases because the denominator increases. Visits to other child nodes of the parent node result in an increase of the exploitation part because the numerator increases. Browne et al.[10, p. 7] also attest "restart property" to UCT through which "even low-reward children are guaranteed to be chosen eventually (given sufficient time), and hence different lines of play explored" and describe UCT as "very simple and efficient and guaranteed to be within a constant factor of the best possible bound on the growth of regret".

2.3. HOP

Hindsight optimization (HOP) was first introduced by Chong et al.[12] and Chang et al.[13] in 2001. In an earlier work by Tesauro and Galperin[14] from 1996 the approach was also used but not named as such.

2.3.1. Base Hindsight Optimization algorithm

> HOP is a heuristic approach for online action selection based on the idea of computing optimistic bounds on action values via problem determinization, allowing for the use of deterministic solvers for probabilistic planning[15, p. 120].

In any position p the goal of HOP is to identify the best move m. Issakkimuthu et al.[15, p. 121] explain that this is done by "calculating an optimistic estimate" for each available move m and "and choosing the action with highest estimate". As can be seen in Figure 2.2 and algorithm 6, HOP first finds all available actions or rather moves m' in a position p and then for each move m' HOP

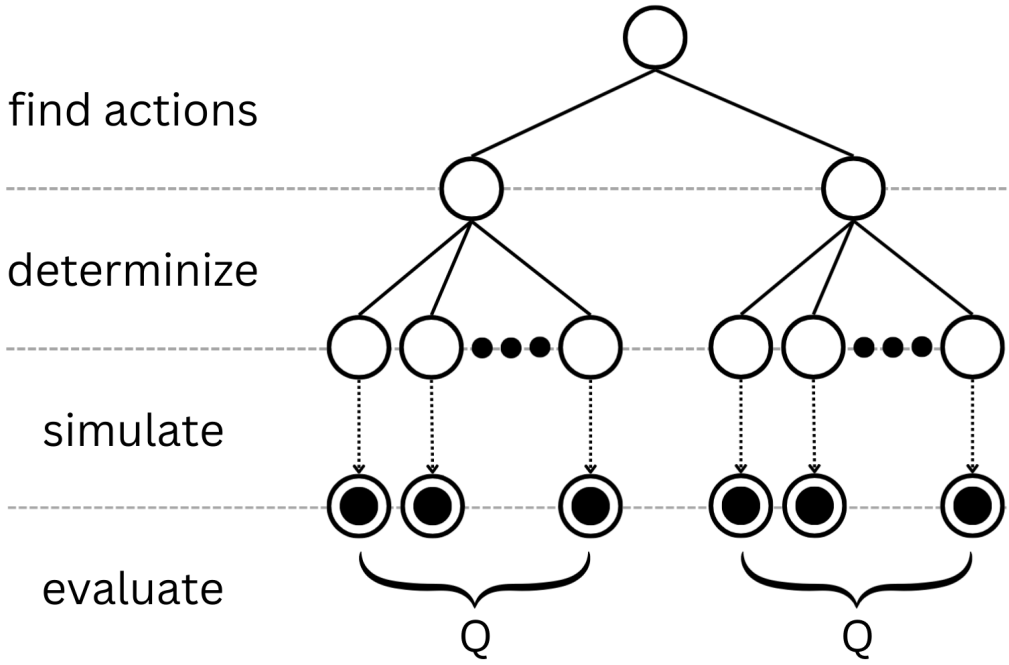

Figure 2.2.: To choose the best action, HOP determinizes the state for each available action a certain amount of times and simulate a play-through of all the determinized states. Afterwards the value of each action is evaluated by averaging the value of the determinized results.

establishes so called ”futures”[15, p. 121]. Each ”future” is a position p' that was determinized by randomly selecting one of the potential positions possible. To obtain the optimistic estimate, HOP then uses a deterministic planner to obtain the value of position p'. These evaluations get averaged into an average value Q_i for each move m, with the best move being the one with the best averaged value.

Algorithm 6 Hindsight Optimization Algorithm

```
1:  function HOP(p)
2:      m ← find available moves
3:      m_i ← get first available move in m
4:      while m_i is not null do
5:          p'_i ← transition position p to futures by applying move m_i and determinizing position
6:          Q_i ← rollout positions p'_i and obtain average evaluation
7:          m_i ← get next available move in m
8:      end while
9:      m_best ← choose best move from m_i based on evaluations v_i
10:     return m_best
11: end function
```

2.3.2. Potential advantages and disadvantages in comparison to MCTS

HOP allows for the usage of deterministic planners and solvers for probabilistic planners. These deterministic solvers are then capable of performing deep searches in the determinized futures[15] . Similarly to MCTS there have been successful applications of HOP, but MCTS is widely more popular and used in a wider range of domains. The downsides of HOP, as described by Issakkimuthu et al.[15, p. 121], are its ”optimistic nature” which ”can lead to poor performance in certain types of domains” and ”leads to suboptimal plans in general”. In addition, Issakkimuthu et al.[15, p. 121] state, ”HOP is not guaranteed to select an optimal action even when given an exhaustive set of futures”. In contrast, ”UCT allows MCTS to converge to the minimax tree and is thus optimal”[10, p. 8]. Issakkimuthu et al.[15, p. 121] observe that ”the complexity of HOP in previous work scales linearly with the number of actions, which is not feasible for large factored action spaces”.

2.4. Related Works

2.4.1. Game collections

The different names and rule sets of Solitaire games can be confusing. Early attempts to systemize the games as well as their name and rules can be seen in the 1913 book “Official Rules of Card Games” by Barney and Curtis[16] and the 1914 book “Lady Cadogan’s Illustrated Games of Solitaire Or Patience [...]” by Cadogan[7]. The 1949 published book “The Complete Book of Solitaire and Patience Games” by Morehead seemingly concludes the systematization and collection of Solitaire games.

2.4.2. Scientific Research

Solitaire games have been a subject of scientific research for multiple decades. For Klondike, multiple relevant discoveries and works have been published.

In 2004 Yan et al. compared the success rate of algorithms when playing thoughtful Klondike and uses the performance of a human expert to contextualize them. Their heuristic approach wins 13.05% of thoughtful Solitaire games by assigning points to actions and maximizing those points while also introducing tie-break rules. Using the rollout method Yan et al. reach a win rate of approximately 70% throughout 200 games. In comparison, a human expert won 36,6% of their 2,000 games of thoughtful Klondike[4]. In a later published article the human expert is revealed as Irving Kaplansky, who was a former president of the American Mathematical Society [3].

Bjarnason et al. published their first estimate of the chance of winning for Klondike games in 2007. For thoughtful Klondike games with draw size of three they demonstrate a chance of winning of atleast 82% and no more than 91,44%. They implemented a multistage nested rollout algorithm which applies different heuristics based on the stage of the search process. In addition, they proved the ability to determine the winnability of thoughtful Klondike games in less than 4 seconds for 80% of the games and introduced ”K+ Solitaire”, a new state representation for Klondike. ”K+ Solitaire” makes all reachable cards from stock and drawn cards extremely playable and eliminates drawing cards as moves, decreasing the depth of the search tree. [5].

In 2009 Bjarnason et al. published a lower bound for an optimal Klondike policy. They managed to win over 35% of the 3 card draw Klondike games. Instead of most other research using the thoughtful variant of Klondike Soltiaire, they managed to do this for the standard variant with unknown possibilities for unrevealed cards. Using UCT and Hindsight optimizations as well as novel combinations and variants of those, such as a combination of HOP and UCT, Sparse UCT and more, they compare their results with a random and a heuristic strategy. As they introduce the use of smaple-based approaches and UCT in the context of Klondike, Bjarnason et al. are an important motivation for this thesis[2].

In 2009 Longpré and McKenzie provided the proof for the NP-completeness of Klondike Solitaire for n-cards and three as well as four suits[17].

In a Bachelor's thesis from 2021 Voima collected information regarding the history of Klondike, especially with an emphasis on the efforts to solve the game completely. In addition, the work provides a way to identify unwinnable instances of Klondike without making any moves[18].

Dang et al. developed constraint models for Klondike in 2024. By proving the impossibility of certain moves, Dang et al. were able to identify 70% of all unwinnable Klondike. Instead of doing complete searches of the game, Dang et al. introduced the concept of blocking sets. Blocking sets are specifically ordered cards in a pile that result in blocking the progress of a game completely[1].

The idea of quantum safety for Klondike was introduced by Waller in 2024. Quantum safety as implemented by Waller ensures in case of uncertainty the player will receive the best result. In Klondike, this means that when a card is revealed, it is ensured that the card revealed allows the player to still win. This approach removes all the components based on luck from Klondike, resulting in a 99.97% chance of winning a random Quantum Safe Klondike[19].

In 2024 Blake and Gent updated their previously published exploration of the winning chances for various Solitaire games including Klondike. To this end, they provide an extensive overview of previous scientific achievements in the realm of Solitaire games and thoroughly establish definitions of game-related terms. They present 'Solvitaire', an AI that can solve single-player card games based on a set of provided rules. Using 'Solvitaire', Blake and Gent established more accurate estimates regarding the winnability of 35 patience games. For the thoughtful variant of Klondike, they established a winnability of 81.95% for games with a draw size of three and a winnability of 90.48% for games with a draw size of one. They also introduced proofs for previously used dominances, which can be used to eliminate or commit to moves in searches[6].

2.4.3. Enthusiast Exploration and Published Solvers

Non-academic research has played an influential role in the exploration and understanding of Solitaire games, so much that it can be said that "the world has owed far more to non-academic than academic research to know the winnability of patience games"[6, p. 5]. For Klondike enthusiasts have delevoped several solvers with different approaches and provided interesting perspectives that extend the understanding of the game.

Enthusiasts

The website jupiterscientific.org published in 2013 an estimate of the upper bounds of a human's win rate for standard Klondike[1]. Their staff member was able to win almost 43% of the games using their strategies[2] focused on increasing the chances of winning.
In a suit of solvers for various Solitaire games from 2014, Wolter[3] introduced a possible dominance that was later also used by Birrell[4]in his 2017 published solver for Klondike. Birrells solver "Klondike-Solver" can be used to find the shortest winning solution for a thoughtful Klondike instance. The dominance used by Wolter and Birrell was later examined and proven in a slightly updated version by Blake and Gent[6].

Solvers

Enthuasiasts have shared their development of all kinds of Klondike solver online. `SolitaireAI` was designed by github.com user `aigamer1` to win Klondike in the pre-installed "Microsoft Solitaire Collection" on Windows 10[5]. Nazar implemented a depth-first search to check the winnabilty of given Klondike instances[6]. An heuristic approach similar to A* was shared by the github.com user `Uspectacle`[7] The solver mentioned above by Birrell[8] used a breadth-first approach and was used by Blake and Gent[6] as a comparison to their own solver. Birrell later published an updated version of the mentioned solver[9]. The solver `lonelybot`[10] by github.com user `vuonghy2442` was published 2024 and seemingly outperforms the results of Blake and Gent[6] when it comes to estimating the winnability of thoughtfu Klondike and Bjarnason et al.[2] when it comes to winning instances of standard Klondike. `vuonghy2442` claims a win rate greater than 47% in standard Klondike.

[1] https://www.jupiterscientific.org/sciinfo/KlondikeSolitaireReport.html
[2] https://www.jupiterscientific.org/sciinfo/AStrategyForWinningKlondikeSolitaire.html
[3] https://politaire.com/article/intro.html
[4] https://github.com/ShootMe/Klondike-Solver
[5] https://github.com/aigamer1/SolitaireAI
[6] https://imrannazar.com/articles/solitaire-ai-js
[7] https://github.com/Uspectacle/Solver-Solitaire
[8] https://github.com/ShootMe/Klondike-Solver
[9] https://github.com/ShootMe/MinimalKlondike
[10] https://github.com/vuonghy2442/lonelybot

3. Methods

This section explains the decisions made in this thesis, such as the variations chosen to compare. Regarding the implementation, the overall structure of the code as well as important details of algorithms, optimizations, and simplifications are introduced. To make reproduction of the achieved results possible, the setup is described for aggregating the statistics.

3.1. Approach

Unlike a human approach to Klondike which focuses on strategies and logic, stochastic simulations use the computational power of modern computers to make choices informed by statistics. For Klondike, this algorithmic approach was already employed by Bjarnason et al. when they used MCTS and HOP as well as combinations of those two algorithms to play Klondike[2].

Bjarnason et al. only used these algorithms for the standard variation of Klondike. Since Klondike is played in many variations, it is still left open how stochastic simulations perform for other variations of Klondike. For this, the 1 draw and 3 draw variants of Klondike will be compared when played using MCTS and HOP. The goal is to determine which parameters are relevant to performance and which combination of parameters performs best for both algorithms. This approach will facilitate confirming the results obtained by Bjarnason et al.

3.1.1. Klondike Variations

From the many known implementations of Klondike, the main variations are:

- Classic Klondike: Drawing three cards at once from stock with infinite redeals for the stock. Probably the most played variant. Also the focus of most scientific research[2, 4, 5] and because of that a good way to compare results with other research. We will call this 3-draw-Klondike from now on.
- Standard Klondike with relaxed draw rules: Drawing only one card at a time from stock with infinite redeals. This will make every card from stock available to the algorithm and can be considered an easy variant. It is shown to have a potential winning chance of over 90%[6]. It is a popular variant to play and in some implementations the default variant to play, for example on solitaired[1]. We will call this 1-draw-Klondike from now on.

[1] solitaired is a website to play Klondike and other Patience games: https://solitaired.com

3.1.2. MCTS tree policies

In the MCTS selection phase, a tree policy is employed to choose the child nodes. For this tree policy, the de facto standard is UCT with a C_p of $\frac{1}{\sqrt{2}}$ for:

$$UCT = \underset{n' \in \text{ children of } n}{\arg\max} \frac{Q(n')}{N(n')} + 2C_p\sqrt{\frac{2\ln N(n)}{N(n')}}$$

For the domain of Klondike, Bjarnason et al.[2] introduced the domain-specific tree policy with $c = 1$, which we will call `Lowerbound`:

$$Lowerbound = \underset{n' \in \text{ children of } n}{\arg\max} \frac{Q(n')}{N(n')} + c\sqrt{\frac{\log N(n)}{N(n')}}$$

Because `Lowerbound` is domain specific, it will be the tree policy used by the MCTS implementation if it's not declared otherwise. To confirm this choice, a comparison of the influence on performance will be presented further down.

3.2. Implementation

The repository containing the implementation can be found at `https://git.tu-berlin.de/midas/klondike-solitaire-ai-player`.

3.2.1. Python

Python was used to implement this project. It is known for its easy-to-read code and a rich collection of standard libraries[20]. In this project, it enabled fast development cycles, an object-oriented approach, and compatibility in running locally as well as in the cluster used for longer computation tasks. More details on the cluster can be found in Section 3.3.

3.2.2. Structure of the repository

```
/
├── runs_and_results_and_graphs
│   └── ...
├── hop.py
├── hop_manager.py
├── manager.py
├── probabilistic_mcts.py
├── probabilistic_mcts_manager.py
├── README.md
└── utils.py
```

`utils.py` contains static methods used by multiple classes, such as identifying available moves and applying move to positions. It also provides methods to generate a random starting position and to create the string representation of a position.

Heuristic rollouts are managed via `manager.py`, which also provides a basic framework for the other manager classes like `hop_manager.py` and `probabilistic_mcts_manager.py`. Both inherit functionality form `manager.py` such as the gathering of statistics.

`hop_manager.py` uses Hindsight Optimization provided by `hop.py` while `probabilistic_mcts_manager.py` uses an implementation of MCTS from `probabilistic_mcts.py`.

Jobs to be run on the cluster, as well as their results, are placed in the `runs_and_results_and_graphs` directory. When graphics were created based on these results, the scripts can also be found in that directory. More details on how to reproduce the results can be found in the Results of the thesis and how to reproduce them section of the `README.md`.

3.2.3. Heuristic for rollouts

Rollouts for HOP, MCTS and `manager.py` can be fully random or employ a heuristic based on dominances. The heuristic can be configured to use one or multiple of the dominances. Dominances are used to identify and force moves that are safe to play and lead to a possible win. This is implemented by only allowing dominant moves when they are available. The two implemented dominances are:

- **Safe moves to foundation:** Moving card from stock or a pile to a foundation is safe to do when the card is at most 3 ranks above the top card on the other foundation with the same color and at most 2 ranks above the top cards on the foundations of the other color.
- **Critical pile moves:** Moving parts of a pile is safe and critical to do when the card left on top of the pile is possible to be moved safely to foundation.

The dominances were theorized and used before, and finally proven by Blake and Gent[6].

3.2.4. K^+ Solitaire: Simplified representation of the stock

Usually, only the card at the top of the waste can be played and other cards are reachable by drawing cards from stock. To remove the drawing from stock as a separate move, the K^+ Solitaire state representation was introduced by Bjarnsen et al.[5]. It makes all reachable cards immediately playable, which according to Bjarnsen et al. amount to ”as few as one-third, or as many as two-thirds of the cards”[5, p. 113] in stock. This does not alter gameplay and does not make the game easier to win, but it compresses the search tree and reduces the amount of moves needed to compute.

The implementation of K^+ Solitaire is shown in algorithm 7. When asked for the available cards of a position p, each card in stock is checked for its reachability. Only reachable cards are returned. The reachable cards are chosen by checking if they fulfill one of these two conditions:

- Reachable when drawing from full stock according to drawing rules. For the 3 draw variant of Klondike, this would make cards with index 1, 4, 7, ... and so on available.
- Reachable after a card from waste is moved and the indexes shift when stock gets refilled. For the 3 draw variant of Klondike, after for example moving the card with stock index 4, the cards with previous indexes 1, 5 (now 4), 8 (now 7),... and so on playable.

Algorithm 7 Available stock cards

```
1: function GET AVAILABLE STOCK CARDS(position p)
2:     d_policy ← stock draw policy of position p
3:     i_last ← index of last drawn card from stock
4:     i ← 0
5:     while i ≤ amount of cards in stock do
6:         if index mod d_policy = 1 OR i ≥ d_last AND i mod d_policy = 2 then
7:             c_available ← append card with index i
8:         end if
9:         increment index
10:    end while
11:    return c_available
12: end function
```

3.2.5. MCTS - Node types

The standard MCTS algorithm is less effective in probabilistic settings[21], such as Klondike. The probabilistic element of Klondike is the uncertainty regarding the unrevealed cards. When these cards are turned face up, they have an equal chance of being one of the unknown cards.

To represent the probabilistic element of Klondike, chance nodes[21] in the form of move nodes are used. As can be seen in figure 3.1, the tree is made up of position nodes that represent a position and move nodes that represent an available move. The root of the tree is always the current position represented as a position node. For each position node, all possible moves are linked as child nodes. For each move node, all possible variations resulting from the move are linked as child nodes of the move node type. Different positions resulting from the same move are possible, because of the uncertainty regarding the reveal of cards. Some moves do not reveal a new card and only have one position node as child node.

This was implemented using two classes, `PositionNode` and `MoveNode` in `probabilistic_mcts.py`. While the backpropagation phase of the MCTS algorithm ignores the node type, the selection phase was adjusted to select the child nodes of the move nodes randomly. This adjustment can be seen in algorithm 8, where the selection phase proceeds differently on the basis of the type of node it encounters in the loop.

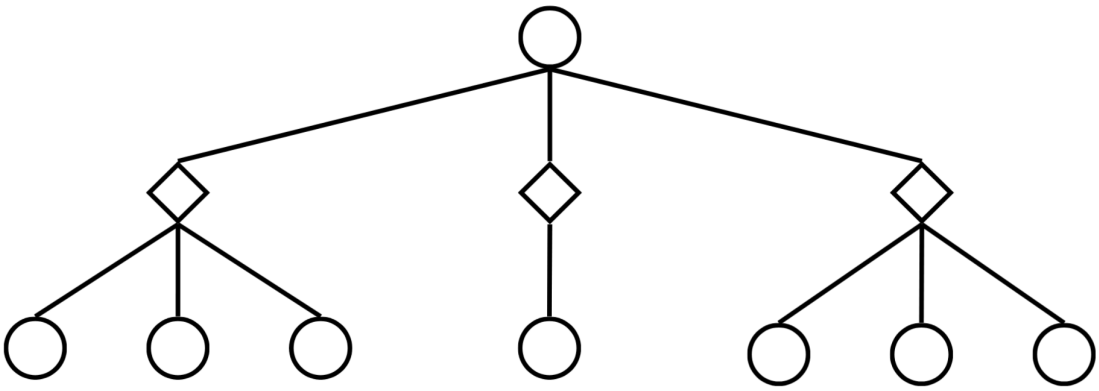

Figure 3.1.: Tree representation of MCTS using move nodes as chance nodes. Round nodes represent position nodes, with the top one being the root node and the current position. Rhombus shaped nodes represent move nodes. When selecting a child node of a position node, the tree policy is used. When selecting a child node of a move node, the selection happens randomly.

Algorithm 8 Selection

```
1:  function SELECT(n)
2:      n_x ← n
3:      while node n_x already rolled out once do
4:          if node n_x is of type move node then
5:              n_x ← choose random child node of n_x
6:          else
7:              if n_x not expanded then
8:                  expand node n_x
9:              end if
10:             n_x ← apply tree policy to n_x to select best child node
11:         end if
12:     end whilereturn n_x
13: end function
```

3.2.6. Optimizations

To gather meaningful statistics, many attempts were needed to be recorded. To increase efficiency, attempts with clear outcomes needed to be shortened. This was done in three different ways:

- **Cut off unsuccessful runs:** For each move up to hundreds of heuristic rollouts are simulated. Not winning a single rollout in a move makes the game practically not winnable for the algorithm, especially in a reasonable amount of moves. When a game has multiple moves in a row in which the algorithm does not win a single rollout, the game is considered lost.
- **Switch algorithm when all cards are revealed:** A game with all cards revealed could already be considered won and some implementations of Solitare such as solitaired[2] do so. This is because the path to winning is as easy as placing cards on the foundation in each move. The algorithm switches to the heuristic used in the heuristic rollout for move selection, because it employs dominances that guarantee one of shortest paths to win.
- **Switch algorithm when all MCTS / HOP rollouts are won:** This is in a sense the opposite of "Cut off unsuccessful runs". Even when not all cards are revealed, if every rollout is won, this can be seen as a position where the positions of the unrevealed cards are not relevant to the outcome of the game and a win is extremely likely. The algorithm switches to the heuristic used in the heuristic rollout for move selection, because it is much faster in selecting moves. This also circumvents a problem of MCTS, which does struggle to choose winning moves when essentially every move is winning.

Profiling

Even a singular attempt via MCTS relies on thousands of rollouts with some parts of the code getting called hundreds of thousands of times. Profiling was used to identify bottlenecks in performance. The python package `cProfile`[3] was utilized to collect statistical data, which was subsequently visualized with `snakeviz`[4]. This led to many efficiency improvements.

3.2.7. Parameters and metrics

Manager classes (`manager.py`, `hop_manager.py` and `probabilistic_mcts_manager.py`) are used to simulate a given number of games with specified parameters and collect the results. The collected results contain these metrics for each game:

- **Moves:** Count of moves played in the attempt before it was either won or lost.
- **Runtime:** Time used for the attempted game.
- **Result:** Win or Loss.

All managers can be configured with these parameters:

[2] solitaired is a website to play Klondike and other Patience games: https://solitaired.com
[3] https://docs.python.org/3/library/profile.html
[4] https://pypi.org/project/snakeviz/

- **Number of games:** The number of games attempted in a row.
- **Maximum of moves:** The number of moves a game can go on before it gets interrupted and called a loss.
- **Stock draw:** The number of cards drawn from stock.
- **Dominances:** The list of dominances used in the heuristic, with multiple dominances being possible to be combined.
- **Debug:** Display debug messages and the position of the game after each move.

For attempts using stochastic simulations, additional parameters can be chosen:

- **Moves per rollout:** The algorithms use rollouts to evaluate the score of moves. This metric establishes the maximum number of moves permitted for these rollouts.
- **Switch strategy when all cards are revealed:** When all cards are revealed, the game becomes trivial and the strategy to find moves gets switched to using the heuristic, which finds the shortest way to a win. This parameter can be turned on and off.
- **Switch strategy when all rollouts are won:** In the event that all rollouts are successful, the position of the unknown cards stops impacting the success of rollouts. The algorithm switches to using the heuristic. This parameter can be turned on and off.
- **Interrupt attempt after x moves with zero successful rollouts:** When hundreds of rollouts in multiple moves in a row are all unsuccessful, it is highly probable that the position cannot be won by the algorithm. The attempt is then interrupted and called a loss. The number of consecutive moves before an interruption can be configured.

The MCTS algorithm can be configured to use a custom amount of seconds as the computational budget to compute a move. The HOP algorithm has a parameter to set the number of trajectories that are evaluated for each available move.

3.3. Test environment

To gather a meaningful number of results, all computation was done on Hydra[5], a cluster used by the ML, MLSEC, UNIML, and Cognition research groups at Technische Universität Berlin. Hydra provides dedicated nodes for CPU and GPU tasks managed by SLURM[6], an open source workload manager. Detailed hardware specifications can be found in the documentation[7]. The python version used by Hydra is `Python 3.10.6`.

To gather statistics, scripts defining the computational resources needed, the locations to log and the command to run were handed over to SLURM via the `sbatch` command. The used scripts are

[5] https://git.tu-berlin.de/ml-group/hydra/documentation
[6] https://slurm.schedmd.com/overview.html
[7] https://git.tu-berlin.de/ml-group/hydra/documentation#specs

collected and explained in the repository's `README` in the section "Results of the thesis and how to reproduce them section".

The resulting data was saved in the logs which contain information regarding the runtime, amount of moves and result of each game. When the `-debug` option was active, each new position in a game was recorded. At the end of each logfile, the total amount of won games, information regarding their move counts as well as the total runtime was summarized.

4. Results

The implementations of both MCTS and HOP permit the configuration of numerous parameters. This facilitates the exploration of a variety of different scenarios, but also complicates the identification of optimized settings to achieve objectives such as efficiency or a high win rate.

This section explores many of the configurable parameters and their potential impact by providing statistics. To begin with, the game length of heuristic rollouts is explored. Heuristic rollouts are the basis of both MCTS's and HOP's decision-making. Building on top of that, for MCTS the different tree policies, changes to the computational budget, and the move limit are explored. For HOP, this section provides results of different amounts of computed trajectories and variations of the move limit. Subsequently, HOP and MCTS are compared with settings that use a comparable amount of time to compute an average game.

These statistics apply to both the 1-draw-Klondike and the 3-draw-Klondike. The 1-draw-Klondike is colored blue in the graphs, and the 3-draw-Klondike is colored orange. The results were achieved using the Hydra cluster and setup described in section 3.3. To make reproduction of the results possible, the commands and statistical methods used are available in the repository[1] and are listed in the Results of the thesis and how to reproduce them section of the `README`.

4.1. Heuristic rollouts

Using the heuristic explained in section 3.2.3, 20.00% out of 100,000 games of the 1 draw Klondike variant and 9.89% out of 100,000 games of the 3 draw Klondike variant were won. Each game was randomly generated and allowed for a maximum of 100,000 moves. As shown in Figure 4.1, most wins were achieved in less than 500 moves for both variants. Still, respectively 15.28% for 1-draw-Klondike and 14.58% for 3-draw-Klondike of wins were achieved with 500 moves or more.

Figure 4.2 shows how efficient different lengths of games are in winning random games. For this, the metric used is won games per million moves, because the amount of computational resources used scales linearly with the amount of moves simulated. It can be seen that the efficiency gets lower the more moves are simulated in a game. This pattern exists for both the 1-draw-Klondike and the 3-draw-Klondike variants.

[1] https://git.tu-berlin.de/midas/klondike-solitaire-ai-player/-/tree/main/runs_and_results_and_graphs?ref_type=heads

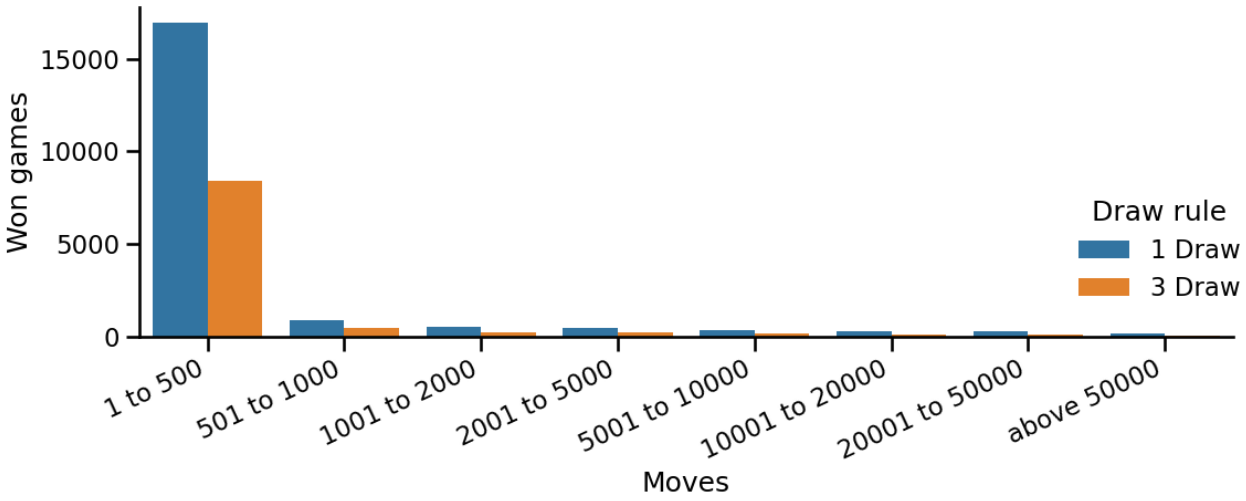

Figure 4.1.: Graphic showing the amount of games won per game length out of 100,000 games using the heuristic rollouts.

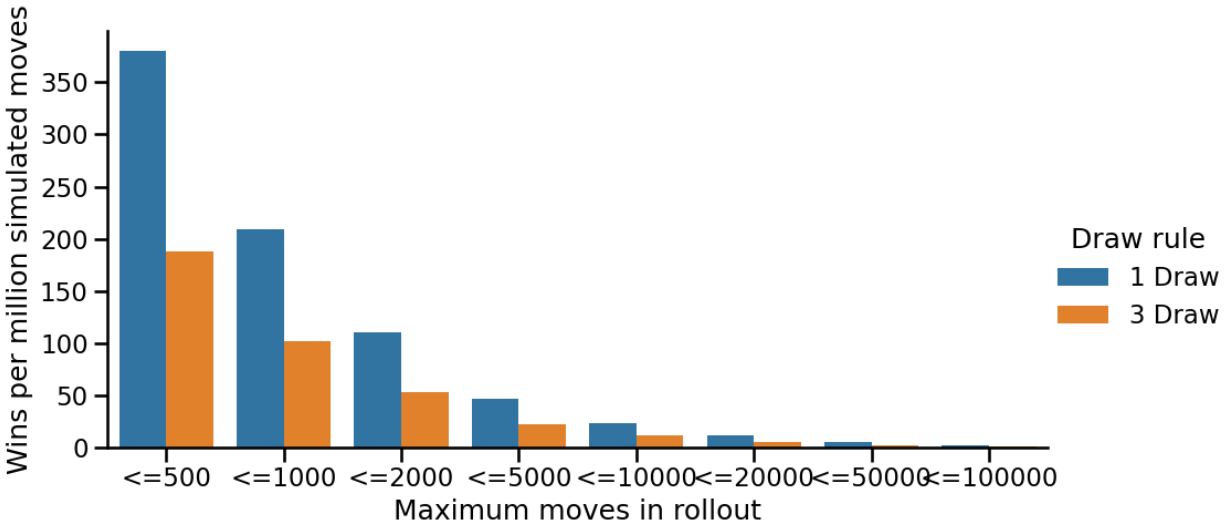

Figure 4.2.: Graphic showing the efficiency of different move limits by comparing the metric of won games per million moves for heuristic rollouts.

Tree policy	1 draw	3 draw
Lowerbound	50.60%	33.00%
UCT	52.40%	32.80%

Table 4.1.: Comparing the winrate for the available tree policies UCT and ”Lowerbound” introduced by Bjarnson et al.[2] for the MCTS algorithm attempting 1-draw-Klondike and 3-draw-Klondike. Each result based on 1,000 attempted random games.

Max. moves in rollout	1 draw	3 draw
250	53.40%	32.20%
500	48.40%	30.80%
750	49.40%	30.20%

Table 4.2.: Comparing the win rate for different move limits for the MCTS rollout for 1-draw-Klondike and 3-draw-Klondike. Each result based on 1,000 attempted random games.

4.2. MCTS

To obtain some data points for optimizing the MCTS run parameters, a variation of statistics was gathered. All the runs in this chapter were completed using a few common settings. The games were interrupted after five consecutive unsuccessful moves as described in section 3.2.6 and the algorithm switched from MCTS to heuristic rollout when either all cards were revealed or all MCTS simulations were won. More details on the algorithm switch can be found in section 3.2.6.

4.2.1. Tree policy

As described in more detail in section 3.1.2, UCT and `Lowerbound` were explored as a tree policy. To measure the impact the tree policy has on the win rate, for each tree policy 1,000 games in the 1-draw-Klondike and the 3-draw-Klondike were attempted.
As can be seen in Table 4.1, the tree policy used in the other statistics of this section achieved a win rate of 50.60% in the 1 draw-Klondike and 33.00% in the 3 draw-Klondike. For UCT, the win rate in the 3-draw-Klondike was only slightly impacted, while the win rate in the 1-draw-Klondike was higher with 52.40%.

4.2.2. Move limit per rollout

The MCTS algorithm performs hundreds or thousands of rollouts for each move. Each rollout could potentially continue for an incredible high number of moves, as shown in Section 4.1. Because of this, the moves in a rollout are limited. The results of comparing the different move limits for rollouts can be seen in Table 4.2. Standing out is the result for the 250 move limit, which produces the highest win rate for both the 1-draw-Klondike(53.40%) and 3-draw-Klondike(32.20%).

	1 draw	3 draw
Won games	5151	3312
Won games with more than 1,000 moves	24	24
Share of won games	0.47%	0.69%

Table 4.3.: Comparing the abolute and relative number of won games with more than 1,000 moves for 1-draw-Klondike and 3-draw-klondike.

4.2.3. Move limit per game

Throughout all MCTS recorded attempts with a variety of settings, only 32 runs out of 16,000 games were stopped because they reached the move limit of 1,000. This is 0.20%. The highest recorded percentage for a single setting was 0.40%(4 out of 1,000 games), with the run parameters being `Lowerbound` as tree policy, 500 maximum rollout moves and computational budget of 5 seconds for 3-draw-Klondike. Of the 351 games that exceeded 500 moves, 82 (23,36%) were won before reaching 1,000 moves and getting stopped.

When attempting 10,000 games with 10,000 maximum moves, both the 1-draw-Klondike and the 3-draw-Klondike variant had 24 games, 0.24% of all attempts, which were won with more than 1,000 moves. As can be seen in table 4.3, this makes up 0.47% of all wins in the 10,000 games of 1-draw-Klondike and 0.69% of all wins in the 10,000 games of 3-draw-Klondike.

4.2.4. Computational budget

The computational budget is the amount of time that the algorithm has available for each move before having to choose a move. In Figure 4.3, the win rate of 1-draw-Klondike and 3-draw-Klondike, each with 5 seconds and 50 seconds of computational budget, are compared. The win rate increases considerably for both Klondike variants, for 1-draw-Klondike it goes from 50% to 59.30% and for 3-draw-Klondike the win rate increases from 32.20% to 44.40%.

4.3. HOP

Similarly to MCTS, statistics were gathered to allow an optimization of the parameters based on data. The common parameters of all statistics are the same as for the MCTS results above in section 4.2.

4.3.1. Move limit per rollout

For the HOP algorithm, in a given position, each possible move is evaluated on the number of successful rollouts. Each rollout is done for trajectory and the amount of maximum moves before unsuccessful termiantion can be configured. Figure 4.4 compares the results of various limits on the amount of moves in rollouts for both 1-draw-Klondike and 3-draw-Klondike. For 1-draw-Klondike a move limit of 500 results in the win rate of 44.60% and for 3-draw-Klondike a move limit of 250 results in the win rate of 31.30%, both being the highest in win rates of the compared configurations. A move limit of 250 is the second best configuration for 1-draw-Klondike.

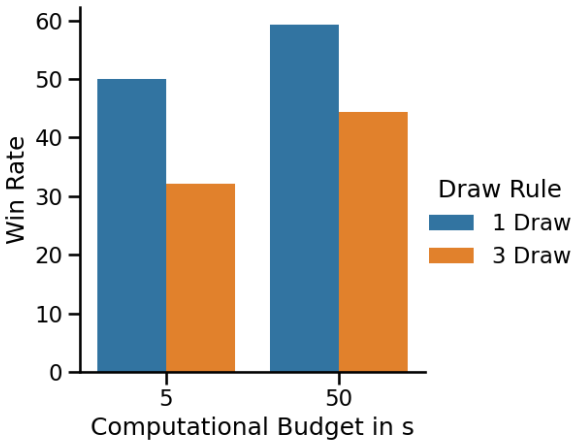

Figure 4.3.: Graphic showing the win rate of games attempted per available computational budget for both 1-draw-Klondike and 3-draw-Klondike. Results for 5 sec are based on 2,000 attempts while Results for 50 sec are based on 1,000 attempts.

Max. moves in rollout	1 draw	3 draw
250	43.70%	31.30%
500	44.60%	28.10%
750	43.19%	30.60%

Table 4.4.: Comparing the win rate for different move limits for the HOP rollout for 1-draw-Klondike and 3-draw-Klondike. Each result based on 1,000 attempted random games (947 games for 1-draw Klondike with 750 max. moves in rollouts).

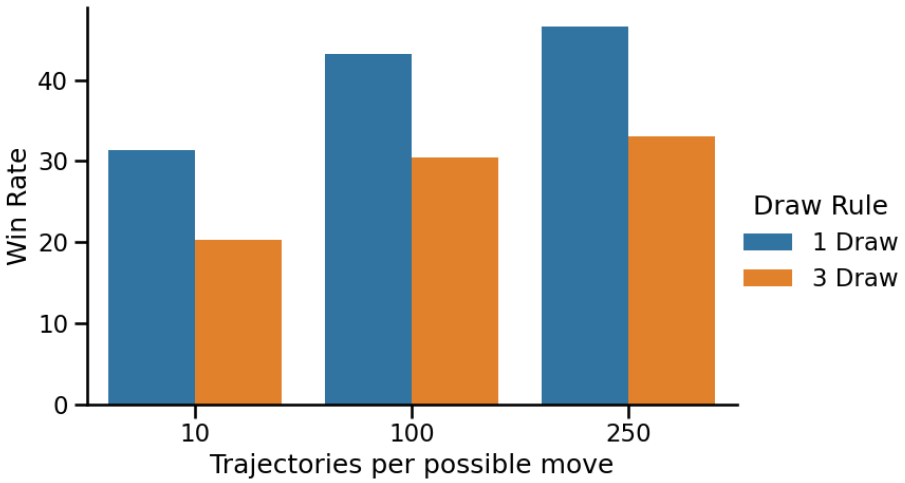

Figure 4.4.: Comparison of HOP win rates with different amount of evaluated trajectories. Results displayed are for 1-draw-Klondike and for 3-draw-Klondike. The results are all based on 1,000 attempts except the results of HOP attempting 999 games of 1-draw-Klondike with 100 trajectories and 994 games of 1-draw-Klondike with 250 trajectories.

4.3.2. Move limit per game

Similarly to the MCTS algorithm, the length of attempted games is limited by a maximum of computed moves. For HOP, all attempts throughout this theses were made with a move maximum of 1,000. In these 11,940 attempts, 7,10% of games reached this maximum and were counted as a loss. The configuration with the highest amount of games interrupted due to reaching the move limit was set to 10 trajectories for the 1-draw-Klondike. In 3.03% of those 11,940 games, a win was achieved with more than 500 moves needed.

4.3.3. Trajectories

The HOP algorithm determinizes and then evaluates a configurable number of trajectories for each available move. In figure 4.4, the win rate achieved by the HOP algorithm for 1-draw-Klondike and 3-draw-Klondike and varying numbers of configured trajectories are compared. For both Klondike variants, the highest win rate is achieved when 250 trajectories are used: 46.58% for 1-draw-Klondike and 33.00% for 3-draw-Klondike. All results are based on 1,000 games for the 10 and 100 trajectories configurations, 994 games for 1-draw-Klondike with 250 trajectories per move and 1,000 games for 3-draw-Klondike with 250 trajectories.

4.4. HOP vs MCTS with similar average computational time

While all of the previous results are somewhat comparable, there are significant differences in the average computational time per attempted game. For comparison of the performance of both algorithms, parameters were used that result in similar average computational times. As can be seen in table 4.5, the average per game is between 2,700 and 4,200 seconds for all configurations.

Algorithm and variant	Average run time in s
HOP - 1-draw-Klondike	4,198.11
HOP - 3-draw-Klondike	3,602.78
MCTS - 1-draw-Klondike	3,480.43
MCTS - 3-draw-Klondike	2,793.70

Table 4.5.: Showcasing the average length of games in seconds for configurations of HOP and MCTS. The configurations were choosen to achieve similar run times while also being based on the parameters of the best previous results. For both algorithms the 1-draw-Klondike and the 3-draw-Klondike variants were attempted. The results are based on 1,000 attempts per configuration.

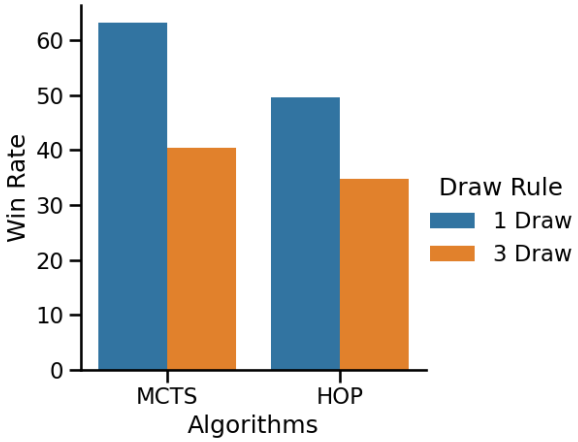

Figure 4.5.: Comparing win rates of HOP and MCTS algorithm attempting 1-draw-Klondike and 3-draw-Klondike while taking a similar amount of time per attempted game on average. See table 4.5 for details on average run times. The results are based on 1,000 attempts per configuration.

Using similar amounts of times to compute, the difference in performance between MCTS and HOP becomes clear in Figure 4.5. The win rates of the MCTS algorithm was higher in both the Klondike variants with 63.20% for 1-draw-Klondike and 40.40% for 3-draw-Klondike out of a thousand games. The configuration of the HOP algorithm wins 49.60% out of 1,000 1-draw-Klondike games and 34.80% out of 1,000 3-draw-Klondike games.

All configurations attempt to use the most successful parameters based on the previously collected results while maintaining a similar average run time. Because of this they are very similar for all attempts in these sections, with these parameters being the same:

- **Maximum of moves:** 1,000
- **Dominances:** all active

- **Debug:** turned off
- **Moves per rollout:** 250
- **Switch strategy & interruption:** all turned on

Additionally, the HOP algorithm was configured to use 250 trajectories per available move and the MCTS algorithm was given a 50 second computational budget per move.

5. Discussion

This thesis investigates the usage of MCTS and HOP to play Klondike in the 1-draw and 3-draw variants. This was done by collecting results of different parameter configurations and combining those results in a potentially improved configuration. The results suggest, that MCTS is a more efficient and appropriate algorithm for winning Klondike games than HOP.

5.1. Lengths of rollouts and games

Most results were generated using a rollout length of 500 moves because it was shown to be the most efficient. The choice to focus on high efficiency was made because rollouts are the part of both MCTS and HOP that take up the most time when finding a move. Still, approximately 15% of successful rollouts where ignored this way and potentially winnable positions that took longer sequences of moves to get won never got successfull rollouts.

To check the choice of rollout length for both algorithms, a comparison of win rates with rollout lengths of 250, 500 and 750 moves were made. Only for MCTS a clear pattern was identifiable, which surprisingly was the highest win rates being achieved with a rollout length of 250 moves. On one hand, shorter rollouts result in higher total amount of rollouts per computed move, on the other hand, shorter rollouts miss even more potential won rollouts.

The game length is a parameter that might profit from further optimization, especially for HOP. All runs of MCTS and HOP were done with a move limit of a maximum of 1,000 moves. For MCTS, at most 0.40% of attempted games reached this amount of moves which means the potential win rates were lowered by at most this percentage. This is different for HOP, where 7.10% of games were stopped at 1,000 moves. These were winnable positions in which the algorithm did not have enough moves to definitely come to a result. It might be worthwhile to explore the potential of these interrupted games in further research.

5.2. MCTS tree policy and computational budget

The MCTS algorithm was tested by recording the impact of different parameters on the win rate. The tree policy seemed to have only a slight impact at most with UCT being more suitable than the costum tree policy suggested by Bjarnson et al.[2]. This might be because Bjarnson et al. employed implemented optimizations such as transposition tables and did not apply a heuristic in their MCTS rollouts.

The computational budget seems to have the highest impact on the MCTS performance. Increasing the budget resulted in a considerable win rate increase. This matches the expectations regarding MCTS with UCT as tree policy, as its evaluations get more precise the more computational budget

is allocated. When comparing the impact of the computational budget on the two researched variants, the results suggest that 3-draw-Klondike profited more from the increase of budget. Its win rate improved by 12.20 percent points, with its total wins inreasing by 27.48%. Especially the relative increase is noteworthy, as it is close to double the 15.68% the increase of 1-draw-Klondike. This difference may the result of the chosen parameters favoring 3-draw-Klondike or might be just a statistical outlier. To confirm this, more research is needed.

5.3. HOP trajectories

The results suggest, that the amount of evaluated trajectories per available move has a positive correlation with the win rate of HOP. Increasing the evaluated trajectories resulted in increases of the win rate achieved for both 1-draw-Klondike and 3-draw-Klondike.
On average, each trajectory is expected to have the same time taken to compute. So increasing the amount of trajectories evaluated can be seen as increasing the allocated computational resources, similar to the computational budget used for MCTS. When thinking about it this way,

5.4. MCTS vs HOP

By configuring MCTS and HOP to have similar run times, a direct comparison is possible. In the presented results, MCTS outperforms HOP in both tested Klondike variants, especially in 1-draw-Klondike. The difference in 3-draw-Solitaire is smaller than expected as previous results show MCTS winning up to 44.40% of attempted games in 3-draw-Klondike. This might be the result of configurations that favor the 1-draw variant.

5.5. Klondike variants

The differences in performance of the algorithms in the two Klondike variants in bigger than expected. While the theoretically achievable win rates of 90.48% for 1-draw-Klondike and 81.95% for 3-draw-Klondike are less than 9% apart[6], many results in this thesis have a difference of more than 15%, some more than 20% difference such as the MCTS results in section 4.4 and none of the configurations achieved a performance difference of less than 10% between 1-draw-Klondike and 3-draw-Klondike. At the same time, the results suggest that algorithms can win atleast two thirds (63,20% out of 90.48%) of winnable 1-draw-Klondike and more than half (44.40% out of 81.95%) of winnable 3-draw-Klondike games.

5.6. Comparison to previous research

When compared previous research done by Bjarnson et al. [2], this thesis shows higher win rates for both MCTS and HOP. The previously highest win rate in 3-draw-Klondike for MCTS was 34.41%, which was improved in this thesis to 44.40%. Similarly, the best win rate of the HOP algorithm in 3-draw-Klondike was 27.20% previously and was improved to 34.80% in this thesis.
For MCTS, the key differences of this thesis implementation seem to be the lack of transposition tables, the usage of heuristic rollouts and the implementation of move nodes as chance nodes to

reduce the search tree. For HOP, heuristic rollouts were used to evaluate the determinized positions instead of an more sophisticated algorithm with a higher win rate.
This is more than 15 years after the previous results were published. These years have seen an enormous increase in computational power, which enabled run configurations for both MCTS and HOP that exceeded the settings chosen by Bjarnson et al. Especially in HOP it was possible to use up to 250 trajectories while still being able to attempt 1,000 games for each configuration.

5.7. Limitations

Several limitations have to be acknowledged. While the gathered results are all based on nearly 1,000 attempts or more, they have not been investigated in regards to their statistical significance and no confidence intervals were calculated. So the results should be used cautiously and with that in mind.

Some of the statistics, especially when comparing different configurations for the same parameter, such as comparing and modifying tree policies, and rollout lengths for MCTS, lack a clear result. More tests and especially a more extensive collection of data might be needed to clarify the impact of some parameters on the algorithms win rate. For this either a more computational resources can be used or the implementation could be improved regarding efficiency and collection of data. The impact of each parameter was tested individually, but the absence mutual influence was not determined.

While the win rate of both algorithms regarding win rate can be classified as satisfactory, improvements are possible and should be implemented and examined. This could entail the utilization of transposition tables and the circumvention of circular moves. A re-implementation in a different programming language or the usage of a performance optimized framework should also be considered.

5.8. Recommendations

Avenues for future research should focus on using domain-specific knowledge to improve the MCTS algorithm. Improving the effiency of the algorithm or increasing its computational budget could yield an improvement in win rate. Identifying potential patterns of the algorithm might lead to new strategic insights which could further improve algorithmic approaches to Klondike as well as improve human understanding of the game.

6. Conclusion

Klondike Solitaire is a popular game that introduced playing games on the computer to many. Despite its well-known digital adoption, computers have not been able to win all or even close to all games presented to them. The uncertain position of hidden cards and the complexity the game stand in the way of an Artificial Intelligence achieving anything close to perfection when playing Klondike with unrevealed cards. Previous research attempted to provide a baseline of performance for algorithms playing and achieved rates at most close to 37%. The objective of this thesis was to enhance the performance of algorithms by attempting different optimizations and leveraging technological advancements. Since previous research did not compare the win rates of algorithms playing different Klondike variants, this thesis aimed to address that gap.

By first testing the impact of individual parameters, the potentially most successful configurations were identified. For both 1-draw-Klondike and 3-draw-Klondike, MCTS was the better performing algorithm. In its best configuration, MCTS won approximately two thirds of all 1-draw-Klondike games and over half of all 3-draw-Klondike games. The respective win rates were 63.20% and 44.40%. HOP did perform worse, achieving win rates of up to 46.58% in 1-draw-Klondike and 34.80% in 3-draw-Klondike. Besides the parameters directly configuring computational resources utilized, the impact of the various parameters was either small or not identifiable by the run tests. Further limitations included only negligible improvements in efficiency and limited statistical analysis of the results.

Future research on the application of stochastic simulations at Klondike should focus on improving the efficiency of the algorithms and optimizing their configurations. It might be possible to establish an upper limit for the win rates of MCTS and HOP by doing this. Furthermore, while this thesis focused only on MCTS and HOP, many more approaches could be employed and compared with the currently available results.

In conclusion, the best win rate achieved by previous algorithms for playing Klondike Solitaire was 37%. This study's MCTS algorithm surpassed that by more than 7%, but further research is needed to optimize the configurations of algorithms playing Klondike Solitaire and potentially establish an upper limit for win rates.

Bibliography

[1] Nguyen Dang, Ian P. Gent, Peter William Nightingale, Felix Ulrich-Oltean, and Jack Waller. Constraint models for relaxed klondike variants. In -, 2024.

[2] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower bounding klondike solitaire with monte-carlo planning. In *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling*, ICAPS'09, page 26–33. AAAI Press, 2009.

[3] David Aldous and Persi Diaconis. Longest increasing subsequences: from patience sorting to the baik-deift-johansson theorem. *Bulletin of the American Mathematical Society*, 36(4):413–432, 1999.

[4] Xiang Yan, Persi Diaconis, Paat Rusmevichientong, and Benjamin Van Roy. Solitaire: man versus machine. In *Proceedings of the 18th International Conference on Neural Information Processing Systems*, NIPS'04, page 1553–1560, Cambridge, MA, USA, 2004. MIT Press.

[5] Ronald Bjarnason, Prasad Tadepalli, and Alan Fern. Searching solitaire in real time. *ICGA Journal*, 2007.

[6] Charlie Blake and Ian P. Gent. The winnability of klondike solitaire and many other patience games, 2024.

[7] A. Cadogan. *Lady Cadogan's Illustrated Games of Solitaire Or Patience: New Rev. Ed., Including American Games ...* D. McKay, 1914.

[8] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers, editors, *Computers and Games*, pages 72–83, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[9] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. FUEGO: an Open-Source Framework for Board Games and Go Engine Based on Monte Carlo Tree Search . *IEEE Transactions on Computational Intelligence and AI in Games*, 2(04):259–270, October 2010.

[10] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[11] Frederik Christiaan Schadd. Monte-carlo search techniques in the modern board game thurn and taxis. Master's thesis, Maastricht University. Maastricht, Netherlands, 2009. Available: https://project.dke.maastrichtuniversity.nl/games/files/msc/Fschadd_thesis.pdf.

[12] E.K.P. Chong, R.L. Givan, and Hyeong Soo Chang. A framework for simulation-based network control via hindsight optimization. In *Proceedings of the 39th IEEE Conference on Decision and Control (Cat. No.00CH37187)*, volume 2, pages 1433–1438 vol.2, 2000.

[13] Hyeong Soo Chang, Robert Givan, and Edwin K. P. Chong. On-line scheduling via sampling. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, AIPS'00, page 62–71. AAAI Press, 2000.

[14] Gerald Tesauro and Gregory Galperin. On-line policy improvement using monte-carlo search. In M.C. Mozer, M. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9. MIT Press, 1996.

[15] Murugeswari Issakkimuthu, Alan Fern, Roni Khardon, Prasad Tadepalli, and Shan Xue. Hindsight optimization for probabilistic planning with factored actions. *Proceedings of the International Conference on Automated Planning and Scheduling*, 25(1):120–128, Apr. 2015.

[16] Walter H. Barney and David A. Curtis. *Official Rules of Card Games*. The United States Playing Card Company, Cincinatti, USA, 1913.

[17] Luc Longpré and Pierre McKenzie. The complexity of solitaire. *Theoretical Computer Science*, 410(50):5252–5260, 2009. Mathematical Foundations of Computer Science (MFCS 2007).

[18] Mikko Voima. Klondike solitaire solvability. Bachelor's thesis, Tampere University of Applied Sciences. Tampere, Finland, 2021. Available: https://www.theseus.fi/bitstream/handle/10024/501330/Voima_Mikko.pdf.

[19] Jack Waller. Klondike solitaire with quantum safety. Master's thesis, University of St Andrews. St Andrews, Scotland, 2023. Available: https://jack.waller.systems/Dissertation.pdf.

[20] KR Srinath. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357, 2017.

[21] Nicolas Jouandeau and Tristan Cazenave. Monte-carlo tree reductions for stochastic games. In Shin-Ming Cheng and Min-Yuh Day, editors, *Technologies and Applications of Artificial Intelligence*, pages 228–238, Cham, 2014. Springer International Publishing.

A. Archived URLs

- https://news.xbox.com/en-us/2020/05/22/celebrating-30-years-microsoft-solitaire/ Accessed: September 2025.
  - Archived: https://web.archive.org/web/20200522210053/https://news.xbox.com/en-us/2020/05/22/celebrating-30-years-microsoft-solitaire/
- https://www.jupiterscientific.org/sciinfo/londikeSolitaireReport.html. Accessed: September 2025.
  - Archived: https://web.archive.org/web/20250220162009/https://www.jupiterscientific.org/sciinfo/KlondikeSolitaireReport.html.
- https://www.jupiterscientific.org/sciinfo/AStrategryForWinningKlondikeSolitaire.html. Accessed: September 2025.
  - Archived: https://web.archive.org/web/20250801140512/https://www.jupiterscientific.org/sciinfo/AStrategryForWinningKlondikeSolitaire.html.
- https://politaire.com/article/intro.html. Accessed: September 2025.
  - Archived: https://web.archive.org/web/20241006080834/https://politaire.com/article/intro.html.
- https://github.com/ShootMe/Klondike-Solver. Accessed: September 2025.
  - Archived: https://web.archive.org/web/20250902155738/https://github.com/ShootMe/Klondike-Solver
- https://github.com/aigamer1/SolitaireAI. Accessed: September 2025.
  - Archived: https://web.archive.org/web/20250906145536/https://github.com/aigamer1/SolitaireAI.
- https://imrannazar.com/articles/solitaire-ai-js. Accessed: September 2025.
  - Archived: https://web.archive.org/web/20240907172909/https://imrannazar.com/articles/solitaire-ai-js.
- https://github.com/Uspectacle/Solver-Solitaire. Accessed: September 2025.
  - Archived: https://web.archive.org/web/20240408081708/https://github.com/Uspectacle/Solver-Solitaire.
- https://github.com/ShootMe/MinimalKlondike. Accessed: September 2025.

  - Archived: https://web.archive.org/web/20250902155738/https://github.com/ShootMe/Klondike-Solver.

- https://github.com/vuonghy2442/lonelybot Accessed: September 2025.
  - Archived: https://web.archive.org/web/20250914193412/https://github.com/vuonghy2442/lonelybot

- https://project.dke.maastrichtuniversity.nl/games/files/msc/Fschadd_thesis.pdf Accessed: October 2025.
  - Archived: https://web.archive.org/web/20250817044305/https://project.dke.maastrichtuniversity.nl/games/files/msc/Fschadd_thesis.pdf

- https://www.theseus.fi/bitstream/handle/10024/501330/Voima_Mikko.pdf Accessed: October 2025.
  - Archived: https://web.archive.org/web/20241117204912/https://www.theseus.fi/bitstream/handle/10024/501330/Voima_Mikko.pdf

- https://jack.waller.systems/Dissertation.pdf Accessed: October 2025.
  - Archived: https://web.archive.org/web/20251002131327/https://jack.waller.systems/Dissertation.pdf

- https://git.tu-berlin.de/ml-group/hydra/documentation Accessed: October 2025.
  - Archived: https://web.archive.org/web/20250319085229/https://git.tu-berlin.de/ml-group/hydra/documentation

Statement on the use of generative AI tools

Generative AI was used to support the writing process of this thesis:

- **Writefull integration in Overleaf**: Writefull free tier is enabled for all Overleaf accounts and allows a fixed set of uses. It was used for language suggestions for singular words and parts of sentences.
- **DeepL**: DeepL free tier was used for translations and for language suggestions for singular words and parts of sentences.