ETH
Swiss Federal Institute of Technology Zurich

Seminar for Statistics

Department of Mathematics

Bachelor Thesis Summer 2024

Paul Kröger

Schokoban: Using Monte Carlo Tree Search for Solving Sokoban

Submission Date: July 18, 2024

Advisor: Prof. Dr. Peters

Abstract

This thesis investigates the adaptation of the Monte Carlo Tree Search (MCTS) algorithm, developed initially for two-player games, to the single-player puzzle game Sokoban. Prior studies have demonstrated the effectiveness of MCTS in handling some single-player games, including Sokoban. However, a notable challenge remains that different move sequences can lead to the same board configuration, causing MCTS to include redundant nodes in its search tree. After explaining the MCTS algorithm and modeling Sokoban as a Markov Decision Process, this thesis proposes a novel method to eliminate these redundant nodes, thus avoiding repetitive calculations. Finally, the empirical section of this thesis demonstrates the effectiveness of the proposed approach. Despite these advancements, the most effective documented Sokoban solvers rely predominantly on variants of the Iterative Deepening A* search algorithm rather than MCTS. The source code for this thesis is available at https://github.com/paulkroe/Schokoban.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Doctor Peters, for his invaluable guidance throughout this research. His expertise and insightful feedback have been instrumental in the completion of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sokoban is a well-known computer puzzle game that challenges players to push boxes to designated storage locations. This thesis explores the application of the Monte Carlo Tree Search (MCTS) algorithm to Sokoban. Developed originally for adversarial two-player games, MCTS must be thoughtfully adapted to efficiently handle deterministic single-player games like Sokoban (Schadd, Winands, Tak, and Uiterwijk, 2012; Crippa, Lanzi, and Marocchi, 2021). In Sokoban, different move sequences can result in the same board state. Consequently, MCTS regularly finds various paths to the same state. In its original form, MCTS includes such states multiple times in its search tree. This thesis avoids these redundant representations by presenting a variant of MCTS eliminating them. Empirical evidence shows that this adaptation enhances performance compared to a baseline approach. Nonetheless, the work does not introduce a state-of-the-art Sokoban solver.

When solving puzzles like Sokoban, a human player typically seeks to explore many different states quickly until discovering a particularly promising one, which is then intensively explored. Crucially, a human player would rarely intentionally revisit the same board state multiple times using different paths, considering it an inefficient use of resources. This observation forms the basis of the hypothesis that reducing redundant state visits could improve solver efficiency. The research question addressed in this thesis focuses on how MCTS can be adjusted to effectively manage redundant nodes in its search tree when applied to Sokoban. The proposed algorithm leverages complete knowledge of the game's dynamics, enabling it to simulate outcomes from any state and action combination. This ability markedly simplifies problem-solving compared to approaches lacking insights into game mechanics. Each puzzle is treated independently, with no learning transferred across different puzzles.

Numerous algorithms have been developed to address Sokoban. Their straightforward implementation makes basic search algorithms like breadth-first and depth-first search as described by Kozen (1992) a convenient option, yet they frequently fail to yield satisfactory results. Zhou and Dovier (2011) utilize a dynamic programming approach, showing promising results, though the state space for complex levels presents a considerable challenge to their method. Junghanns and Schaeffer (2001a) adapt the Iterative Deepening A* (IDA*) algorithm, introduced by Korf (1985), for Sokoban, achieving even more remarkable outcomes. Other approaches frame Sokoban as a reinforcement learning problem. Shoham and Elidan (2021) explore deep TD(0) learning and train neural networks for Sokoban. Alternatively, Schadd et al. (2012) propose modifications to the basic MCTS algorithm,

tailoring it to single-player games. Subsequently, Crippa et al. (2021) develop an MCTS-based approach for solving Sokoban. Unfortunately, precise details of many state-of-the-art Sokoban solvers remain proprietary, restricting the scope for detailed comparative analysis in academic research.

Building on these previous works, this thesis introduces Vanillaban, an MCTS variant developed for Sokoban puzzles. The efficacy of Vanillaban is improved by eliminating redundancies in the search tree, leading to the development of an advanced version named Schokoban. Schokoban's advantage over Vanillaban is confirmed experimentally. All the code and experimental results are available at `https://github.com/paulkroe/Schokoban`. The algorithms in this thesis are designed to be straightforward, and although they are not state-of-the-art Sokoban solvers, they are robust enough to solve engaging puzzles. Hopefully, they can serve as a starting point for further development of MCTS-based solvers for Sokoban, reducing the entry barriers associated with the absence of a standardized, well-tested library implementation of MCTS.

This thesis begins by outlining the rules of Sokoban in Chapter 2. It then proceeds to a comprehensive analysis of the MCTS method in Chapter 3. The discussion advances to modeling Sokoban as a Markov Decision Process, culminating in the development of Vanillaban, as detailed in Chapter 4. The subsequent chapter, Chapter 5, explores the handling of redundant states within Sokoban puzzles. An empirical evaluation follows, contrasting the performance of Vanillaban with that of Schokoban in Chapter 6. Lastly, the appendices offer insights into the search space complexity for the Sokoban puzzles analyzed (Appendix A) and provide a detailed case study of applying the general MCTS algorithm to the two-player adversarial game of Tic-Tac-Toe (Appendix B).

Chapter 2

Sokoban

Sokoban, which translates to ”warehouse keeper” in Japanese, is a puzzle video game developed by Hiroyuki Imabayashi and published in December 1982 by the Japanese company Thinking Rabbit (Murase, Matsubara, and Hiraga, 1996; Balyoáš and Froleyks, 2022). In Sokoban, the player's objective is to push boxes around a warehouse maze to designated storage locations.

2.1 Sokoban Rules

Sokoban is a game with simple rules discussed in this section. The player can move in four directions: up, down, left, and right, as long as no wall blocks the path. If a box is in the way, the player can push it, provided no box or wall is directly behind it. To win the game, one must place all boxes on the designated goal locations. Some moves in

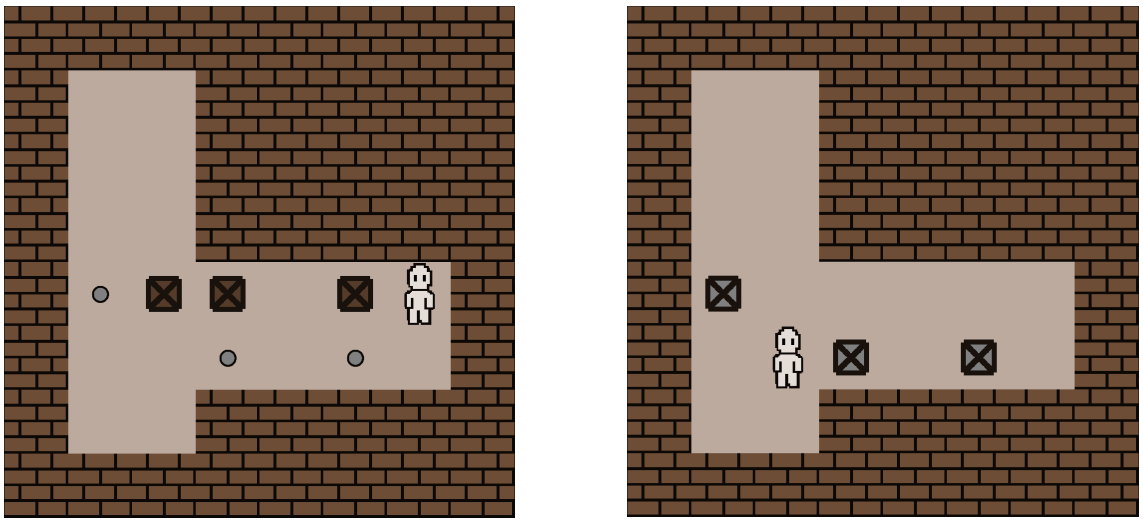


Figure 2.1: This example from the Microban III collection by Skinner (2009) starts in the configuration on the left. It concludes when the player successfully pushes all boxes onto the goals, as shown on the right.

Sokoban are irreversible, meaning no sequence of valid moves can return the board to its previous state. Irreversible moves can render the board unsolvable. Such board positions are referred to as deadlocks.

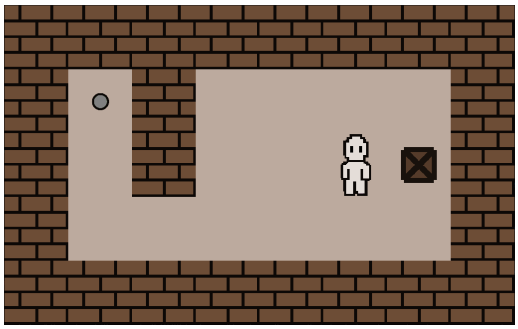


Figure 2.2: From this deadlock state, the game is not winnable anymore.

2.2 The Challenge of Solving Sokoban

Sokoban is a challenging problem in the sense that it is NP-complete (Fryers and Greene, 1995; Dor and Zwick, 1999) and PSPACE-complete (Culberson, 1997). Besides these theoretical results, Sokoban puzzles present several practical challenges for computer algorithms. To begin with, the search space for Sokoban problems can be immense due to the numerous possible configurations of the player and boxes on the board. However, the exact size of the search space varies significantly depending on the board under consideration. For instance, Junghanns and Schaeffer (2001b) estimate the search space complexity of one particular 20×20 board at around 10^{98}. While the board's size is typical for complex Sokoban puzzles, the level analyzed is unusual. Therefore, the above figure represents a conservative upper bound instead of a practical estimate. Junghanns (1999, p.61) reports that the average search space complexity for more realistic boards is significantly lower at approximately 10^{13}. The search space complexities for the levels analyzed in this thesis, specified in Appendix A, are even smaller. For context, the well-known Rubik's Cube puzzle features roughly 10^{19} possible configurations (Agostinelli, McAleer, Shmakov, and Baldi, 2019) while Shannon (1950) poses 10^{120} as a conservative upper limit for the state space size of Chess. Additionally, Sokoban puzzles frequently demand long move sequences to solve. For instance, many levels in the Sokoban collection by Myers (2001), commonly used to assess the performance of state-of-the-art solvers, feature optimal solutions that typically involve several hundred moves.

Chapter 3

Monte Carlo Tree Search

MCTS is an adversarial search algorithm that operates under conditions similar to the Minimax Algorithm (Russell and Norvig, 2016, Section 5). It assumes the existence of an adversary whose actions aim to minimize the player's rewards. Initially, Kocsis and Szepesvári (2006) and Coulom (2007) developed MCTS for addressing the game of Go (Świechowski, Godlewski, Sawicki, and Mańdziuk, 2023). However, in their survey Kemmerling, Lütticke, and Schmitt (2024) find that MCTS has applications across numerous fields. After introducing Markov Decision Processes, the formal framework necessary for discussing MCTS, the subsequent chapter gives a general outline of the MCTS algorithm as described by Sutton and Barto (2018, Section 8.11) and Chaslot, Bakkes, Szita, and Spronck (2008).

3.1 Discrete Markov Decision Processes

MCTS is employed with discrete Markov Decision Processes (MDPs). MDPs offer a rigorous framework for analyzing a wide range of planning problems. While it is possible to gain an intuitive understanding of MCTS without prior knowledge of MDPs, a thorough understanding of MDPs is essential for optimizing MCTS methodologies in practical applications. The following section describes MDPs as presented in Feinberg and Shwartz (2002).

3.1.1 Definition and Basic Properties

Definition 3.1.1.1 (Markov Decision Process)**.** *A Markov Decision Process is defined as a tuple* $(\mathcal{S}, \mathcal{A}, p, r)$*, where:*

- $\mathcal{S}$ *is a set called the state space, consisting of all possible states.*
- $\mathcal{A}$ *is a set called the action space, which comprises all possible actions.*
- $\mathcal{A}(s) \subseteq \mathcal{A}$ *for each* $s \in \mathcal{S}$*, denotes the set of permissible actions available from state* s*.*
- $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ *is a function called the transition probability function. For each state* $s \in \mathcal{S}$*, action* $a \in \mathcal{A}(s)$*, and state* $y \in \mathcal{S}$*,* $p(y|s, a)$ *specifies the probability of transitioning to state* y *when action* a *is taken in state* s*.*

- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ *is a function known as the reward function. For each state* $s \in \mathcal{S}$ *and action* $a \in \mathcal{A}(s)$*,* $r(s, a)$ *denotes the immediate reward received after taking action* a *in state* s*.*

An MDP models a stochastic system within a state space $\mathcal{S}$. In this model, a decision-making agent, upon encountering a state $s \in \mathcal{S}$, selects an action $a \in \mathcal{A}(s)$. The system, often referred to as the environment, then transitions to a new state according to the probability distribution $p(\cdot|s, a)$, and the agent receives the corresponding reward $r(s, a)$. The action selection by the agent can be deterministic or randomized, with the latter involving the agent choosing actions based on a probability distribution over the available actions. Time is indexed by $t \in \mathbb{N}$, differentiating states and actions across different time steps. By definition, for any state $s \in \mathcal{S}$ and action $a \in \mathcal{A}(s)$, it is guaranteed that $\sum_{s' \in \mathcal{S}} p(s'|s, a) = 1$. A state space or action space is said to be discrete if it is finite or countably infinite. Accordingly, an MDP is called discrete if its state and action spaces are discrete. This thesis assumes all MDPs are discrete. For reference, Feinberg and Shwartz (2002) discuss nondiscrete MDPs. A trajectory is defined as a sequence $s_0, a_0, s_1, a_1, \ldots$ in $H_{\mathbb{N}} = (\mathcal{S} \times \mathcal{A})^{\mathbb{N}}$. The sequence of events up to time step n specifies a history, denoted $h_n = s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n$, and the space of all such histories is labeled $H_n = \mathcal{S} \times (\mathcal{A} \times \mathcal{S})^n$.

3.1.2 Policies

Let $s \in \mathcal{S}$. A nonrandomized policy ϕ consists of a series of mappings $\{\phi_n\}_{n \in \mathbb{N}}$, where each ϕ_n maps a history H_n to the action space $\mathcal{A}$. For a history sequence $(s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n)$, the action $\phi_n(s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n)$ is selected from $\mathcal{A}(s_n)$. A policy is said to be Markov if ϕ_n relies solely on s_n hence a Markov policy can be expressed as a sequence of mappings $\phi_n : \mathcal{S} \to \mathcal{A}$, with $\phi_n(s) \in \mathcal{A}(s)$ for each $n \in \mathbb{N}$. A Markov policy ϕ is stationary if $\pi_n(\cdot|s) = \pi_m(\cdot|s)$ for any $n, m \in \mathbb{N}$. This framework also applies to random policies. A randomized policy π is characterized by a sequence of transition probabilities $\pi_n(a_n|h_n)$ mapping histories in H_n to actions in $\mathcal{A}$. For any history $h_n \in H_n$, the sum of probabilities for all possible actions equals one, more precisely $\sum_{a \in \mathcal{A}(s)} \pi_n(a|h_n) = 1$. A randomized policy is Markov if π_n depends only on the current state s_n, thus $\pi_n(a_n|h_n) = \pi_n(a_n|s_n)$. As for nonrandomized policies, a random policy is stationary if it is time-invariant. Given the flexibility of random policies, all policies discussed henceforth may be random.

3.1.3 State Value Function

Consider a Markov policy π and an environment in state $s \in \mathcal{S}$ at time $n \in \mathbb{N}$. When policy π selects an action $a \in \mathcal{A}(s)$ this action causes the environment to transition to a new state $s' \in \mathcal{S}$ at time $n + 1$ with a probability of $p(s'|s, a)$. Thus, a random policy π in conjunction with a starting state $s \in \mathcal{S}$ defines a stochastic sequence. Denote the expectation associated with policy π and starting state s, by $\mathbf{E}_s^\pi$. Further introduce a constant discount factor $\beta \in [0, 1]$, together with a terminal reward function f. The expected total reward over the first $N \in \mathbb{N}$ steps, denoted by v_N, is defined as

$$v_N(s, \pi) = \mathbf{E}_s^\pi \left[\sum_{n=0}^{N-1} \beta^n r(s_n, a_n) + \beta^N f(s_N) \right],$$

assuming the expectation exists. When $\beta \in [0, 1)$, v_N is referred to as the total discounted reward, if $\beta = 1$, it is known as the total undiscounted reward. In an infinite-horizon

scenario, omitting the terminal reward function f, the value function v is given by

$$v(s, \pi) = \mathbf{E}_s^{\pi} \left[\sum_{n=0}^{\infty} \beta^n r(s_n, a_n) \right],$$

provided this quantity exists. If the reward function r is bounded and $\beta \in [0, 1)$, the existence of v is guaranteed for any $s \in \mathcal{S}$ and any policy π by the convergence of the geometric series.

The state value function v, as its name implies, is employed to assess the value of a state under a given policy. Value functions are vital for developing algorithms designed to deal with MPDs (Sutton and Barto, 2018).

3.1.4 Modeling Sokoban as a Markov Decision Process

This section contextualizes the theoretical concepts of MDPs by modeling a fixed Sokoban puzzle as an MDP. The state space encompasses all possible configurations of the Sokoban board. Given that the arrangement of walls and goals remains fixed, the state can be defined by the player's coordinates and the positions of the boxes. The action space consists of the four possible directions in which the player can move: up, down, left, and right. The available actions remain constant across all states. Additionally, the agent receives a reward of -1 for each move that does not resolve the level, while successfully solving the level yields a reward of $+10$.

While the described elements provide a functional framework for modeling Sokoban as an MDP, these choices are not exclusive. The specific requirements of the algorithms applied may necessitate alternative representations of state and action spaces or different reward functions. Chapter 4 delves into more advanced state and action representations and reward functions utilized by the algorithms discussed in this thesis.

3.2 Monte Carlo Tree Search in Detail

Consider an environment modeled as an MDP, where for each state $s \in \mathcal{S}$, $A(s)$ consists of a finite set of actions, and the transition probability function is deterministic. In particular, for every state $s \in \mathcal{S}$, there is a state $s' \in \mathcal{S}$ and an action $a \in \mathcal{A}(s)$ such that $p(s'|a, s) = 1$. Assume further that the agent fully understands the dynamics of the environment, allowing it to predict outcomes from actions taken. In this context, MCTS can be used to plan an optimal action based on the current state of the environment. The algorithm consists of four phases: selection, expansion, simulation, and backpropagation. MCTS loops through these four phases until it has used up its computational budget. Notably, MCTS can be halted at any time to estimate an optimal action, though the accuracy of this estimate is contingent upon the computational effort invested in generating it. MCTS builds up a search tree where every node stores a representation of some environment state, a visit count n, and a scalar value v used to estimate the value of the node's state. Nodes might store additional information, including prior probabilities, as in Silver, Schrittwieser, Simonyan, Antonoglou, Huang, Guez, Hubert, Baker, Lai, Bolton, Chen, Lillicrap, Hui, Sifre, van den Driessche, Graepel, and Hassabis (2018). MCTS starts by initializing a root node representing the current state of the environment. Then, the algorithm enters its first selection phase.

3.2.1 Selection Phase

The selection phase identifies the most promising leaf node using a selection or tree policy. The policy aims to balance exploiting nodes known to hold promise and exploring less frequently visited nodes. The process begins at the root node and continues recursively. At each step, the selection policy assigns a score to each child of the current node, and the child with the highest score is selected. Upon reaching a leaf node, the algorithm proceeds to the expansion phase. Otherwise, the selection process repeats until a leaf node is discovered.

3.2.2 Expansion Phase

If the node selected during the selection phase has a visit count of zero, the algorithm directly proceeds to the simulation phase. Otherwise, a new child node is created for every action possible from the selected leaf node. Specifically, if the chosen leaf node represents a state $s \in \mathcal{S}$, then for each action $a \in \mathcal{A}(s)$, a child node is added to the tree. This child node represents a new state s', which results from applying action a to state s. One of those newly added children is chosen randomly and passed on to the simulation phase.

3.2.3 Simulation Phase

The simulation phase, also known as the rollout in literature, starts with the state of the node selected for simulation. The algorithm simulates the game or problem scenario following a predetermined policy until it reaches a terminal state. During the backpropagation phase, the final reward from the simulation is used to update the tree. Typically, the simulation policy chooses actions randomly with uniform probability.

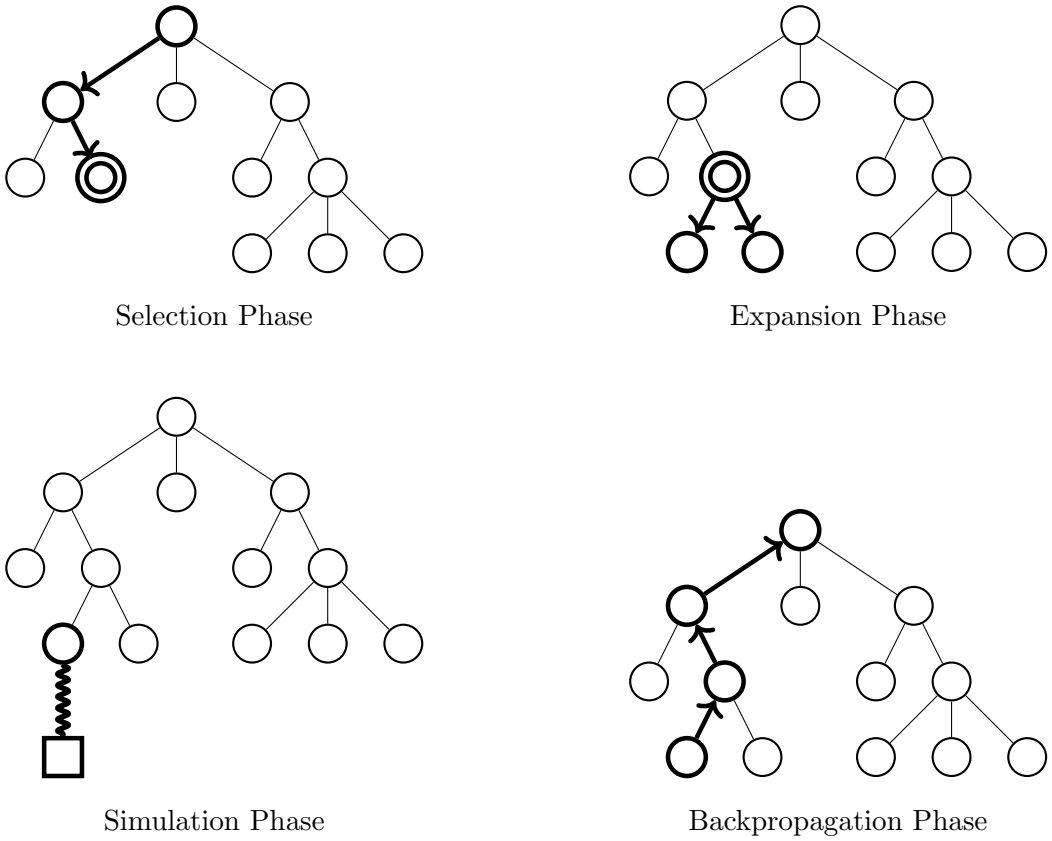
3.2.4 Backpropagation Phase

During the backpropagation phase of MCTS, the value derived from the simulation phase updates the estimated values of all nodes traversed in the current iteration. One iteration of MCTS inherently comprises a cycle through the selection, expansion, simulation, and backpropagation phases. Thus, updates are applied beginning at the node where the last rollout concluded and progressing up to the root. Notably, the MCTS updates the root node in every backpropagation phase. The initial update during the backpropagation phase incorporates the value received from the rollout. Each node's value is adjusted using the formula:

$$v \leftarrow \frac{n \cdot v + u}{n + 1}.$$

In this formula, u represents the score used to modify the node's value. Therefore, the estimated value of node i is simply the average of all values from rollouts that either passed through node i during the selection phase or started from node i. After updating a node's value v, its visit count n is incremented by one. Crucially, the value u is negated before being passed to the parent node. This practice, stemming from MCTS's application in adversarial games like Go or Chess, ensures that a beneficial outcome for one player is interpreted as unfavorable for the opponent. The principle of alternating values during backpropagation, along with a practical example, is further explored in Appendix B. Once backpropagation is complete, the algorithm returns to the selection phase for the next cycle.

Once MCTS concludes, various methods are employed to determine the next move. Typically, this involves examining the child nodes of the root node to identify which one has the highest visit count. The move that connects the root to this particular child is the estimated optimal move.



Selection Phase | Expansion Phase

Simulation Phase | Backpropagation Phase

Figure 3.1: The MCTS algorithm cycles through these four phases until it is stopped and an estimate can be made.

3.2.5 UCT Selection Policy

Auer, Cesa-Bianchi, and Fischer (2002) introduced the Upper Confidence Bound (UCB1) approach to tackle the multi-armed bandit problem (Sutton and Barto, 2018). Building on these insights, Kocsis and Szepesvári (2006) crafted a variant specifically for trees, termed UCB1 for Trees or UCT. This adaptation has become the predominant selection policy in MCTS (Ameneyro and Galvan, 2022). To illustrate how UCT is implemented within MCTS, consider a parent node p and its child node i. Using n_i to denote node i's visit count and v_i is estimated value, the UCT value for node i is calculated using the formula:

$$UCT(node_i) = v_i + c \cdot \sqrt{\frac{2\ln(n_p)}{n_i}}. \tag{3.2.5.1}$$

The exploration constant c is typically a tunable hyperparameter. Algorithm 1 summarizes the selection phase using the UCT policy. It is important to note one technical detail. If UCT is used as a selection policy, one has to be careful to not try to calculate the UCT score for a node that has a visit count of zero since that would result in a division by zero.

If during the selection phase the algorithm runs into child nodes that have a visit count of zero, one of them is selected at random.

Algorithm 1 Selection Phase of MCTS using UCT

Initialize current node as the root node
while current node is not a leaf node **do**
    Identify all child nodes of the current node
    if any child node has not been visited **then**
        Randomly select an unvisited child node
    else
        Calculate the UCT score for each child node:

$$UCT(node) = v_{node} + c \cdot \sqrt{\frac{2\ln(n_{parent})}{n_{node}}}$$

        Select the child node with the highest UCT score
    end if
    Update current node to the selected child node
end while
return current node

3.2.6 State Diagram of the MCTS Algorithm

The state diagram in Figure 3.2 summarizes the MCTS algorithm described above. The implementation of MCTS in this thesis closely follows the logical flow depicted.

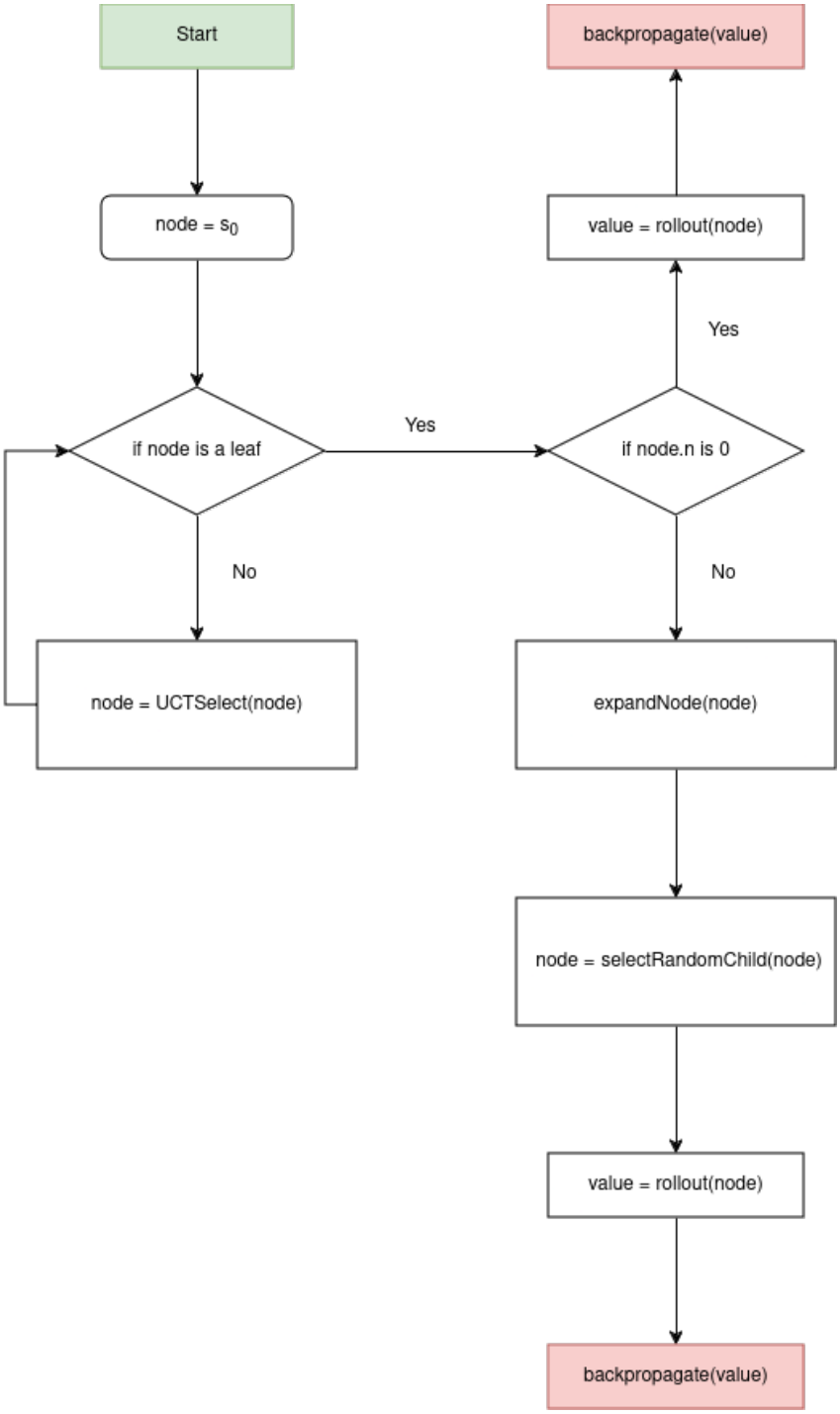


Figure 3.2: Each iteration of the MCTS begins at the green start node and progresses through the nodes of the state diagram, eventually concluding at one of the red nodes. The next iteration restarts from the green start node.

Chapter 4

Monte Carlo Tree Search for Sokoban

MCTS has undergone extensive study (Świechowski et al., 2023) and is frequently applied to two-player board games (Arneson, Hayward, and Henderson, 2011; Robles, Rohlfshagen, and Lucas, 2011; Teytaud and Teytaud, 2010). A notable application of MCTS was in the development of AlphaGo by Silver, Huang, Maddison, Guez, Sifre, van den Driessche, Schrittwieser, Antonoglou, Panneershelvam, Lanctot, Dieleman, Grewe, Nham, Kalchbrenner, Sutskever, Lillicrap, Leach, Kavukcuoglu, Graepel, and Hassabis (2016), which marked a milestone by exhibiting superhuman performance in Go. Building on this success, Silver et al. (2018) introduced AlphaZero, mastering Go, Chess, and Shogi.

Following up on these achievements by applying MCTS to Sokoban is not straightforward due to fundamental differences in game dynamics. Two-player games introduce randomness and unpredictability through the opponent's moves, and algorithms such as AlphaGo and AlphaZero rely heavily on learning from self-play. In contrast, Sokoban is a deterministic single-player game that lacks a natural way for the agent to engage in self-play. To overcome these challenges, Schadd et al. (2012) propose adopting MCTS for single-player games. Advancing this research, Crippa et al. (2021) address Sokoban using MCTS. Their work demonstrated that with proper modeling decisions and algorithmic adjustments, MCTS can efficiently solve Sokoban puzzles. Nevertheless, the authors find that IDA* remains superior to MCTS in solving Sokoban.

Leveraging insights from Crippa et al. (2021), this thesis introduces Vanillaban, an MCTS-based Sokoban solver. Specifically, Vanillaban uses the reward function, Recursive Node Elimination, and the state and action representations from Crippa et al. (2021) while deliberately excluding the more advanced enhancements outlined in Crippa et al. (2021, Section 7). The focus of this thesis is not to replicate the algorithms developed in Crippa et al. (2021) but to explore the impact of redundant nodes in the MCTS search tree. By adopting only the elements above, Vanillaban balances simplicity and performance, making the algorithm understandable and sufficiently powerful to yield meaningful results. Particularly, Vanillaban is developed as a baseline for comparison with Schokoban, demonstrating the advantages of avoiding redundant state representations in the MCTS search tree.

4.1 Modelling Sokoban

Based on Crippa et al. (2021), the following section models Sokoban as an MDP by selecting appropriate state and action spaces and defining the reward signal.

4.1.1 Equivalence States for Sokoban

Initially, one might define the state space in Sokoban by considering all possible board configurations, with the actions being the four possible movements: up, down, left, and right. However, this approach experimentally yielded unsatisfactory results, necessitating a redefinition of the state and action spaces. Arguably, only moves that change the position of a box are crucial. Therefore, Junghanns (1999) suggests an action should consist of a sequence of moves, where only the last move pushes a box. Figure 4.1 depicts such an action. This approach increases the branching factor whenever the player can access more

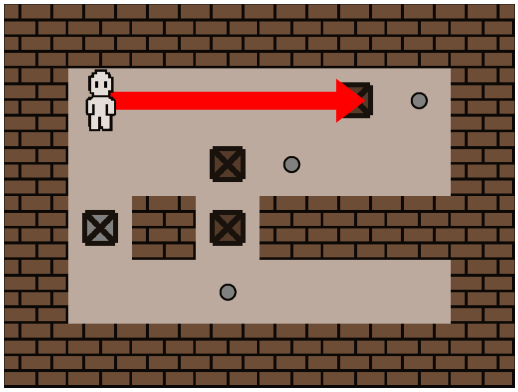


Figure 4.1: This move sequence culminates in a box push, constituting one action.

than four boxes. Nevertheless, based on empirical evidence, the algorithms in this thesis use the latter type of action. Accordingly, a state in Sokoban is represented by the positions of walls, goals, and boxes. However, rather than tracking the player's exact position, the state representation includes the connected component that the player currently occupies. This connected component is defined by the set of goals and floor tiles reachable by the player without pushing a box. Figure 4.2 visualizes a connected component. For future reference, a state represented in this way is termed an equivalence state. Figure 4.3 shows an equivalence state. Consequently, the available actions in a given equivalence state are the box pushes that can be executed from any position within the player's current connected component.

4.1.2 A Reward Function for Sokoban

The chosen reward function heavily influences the effectiveness of MCTS. A common approach for designing reward functions involves incorporating domain knowledge to mimic human decision-making processes. In this context, reward shaping is a commonly applied technique where the agent receives small auxiliary rewards (Gupta, Pacchiano, Zhai, Kakade, and Levine, 2022). These rewards are designed to guide the agent towards achieving specific subgoals. For instance, in Sokoban, one could consider assigning a reward for each box placed on a goal. However, rewarding subgoals may lead to a phenomenon known as reward hacking or specification gaming, where the agent manipulates the reward system in unintended ways to maximize rewards without progressing towards the

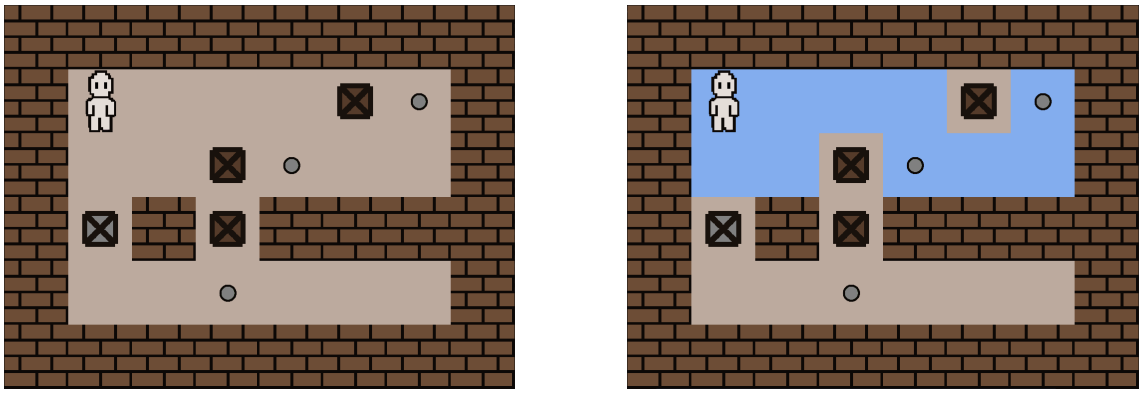


Figure 4.2: The connected component currently occupied by the player is marked in light blue. Any of the tiles in the connected component can be reached by the player without pushing a box.

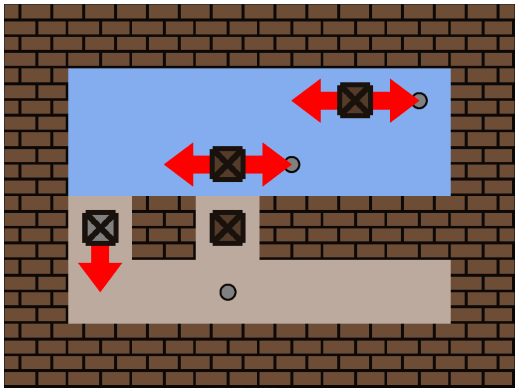


Figure 4.3: In the visualization of the state representations above, arrows indicate available actions.

true objective (Krakovna, Uesato, Mikulik, Rahtz, Everitt, Kumar, Kenton, Leike, and Legg, 2020; Skalse, Howe, Krasheninnikov, and Krueger, 2022). Considering the reward function just mentioned, the agent might repeatedly push the same box onto and off a goal to accumulate rewards quickly. While one might counter this by penalizing the removal of boxes from goals, this example highlights the complexity of designing a reward signal. Reward hacking is not a challenge exclusive to games. It is a prevalent issue in reinforcement learning. Amodei, Olah, Steinhardt, Christiano, Schulman, and Mané (2016) stress that addressing reward hacking is crucial for developing safe AI systems. A minimalist approach to designing a reward function might reward the agent solely upon achieving the final goal. This strategy is exemplified by the work of Silver et al. (2018) on games like Chess, Go, and Shogi. However, this approach can result in too sparse a reward signal, making it challenging for the agent to learn effectively through trial and error (Dawood, Dengler, de Heuvel, and Bennewitz, 2023; Riedmiller, Hafner, Lampe, Neunert, Degrave, van de Wiele, Mnih, Heess, and Springenberg, 2018). During the development of the algorithms in this thesis rewarding only wins proved to be ineffective.

In their survey, Crippa et al. (2021) address these challenges and explore four different reward functions for Sokoban. They identify the negative minimum cost perfect matching to be the most promising. To compute the minimum cost perfect matching for a given

Sokoban board a graph is constructed by creating a node for every goal and every box. Edges are added between goal and box nodes, with each edge weighted by the Manhattan distance between goal and box on the board. Using the negated value of the minimum cost perfect matching of this induced graph as a reward is well-defined since the graph is inherently bipartite, meaning it consists of two sets of vertices, goals, and boxes, with edges only between vertices of different sets. A perfect matching pairs each box with a goal such that every box is connected to exactly one goal, and each goal is matched to one box. A minimum cost perfect matching seeks to minimize the total distance between paired boxes and goals across all possible matchings. Intuitively, minimum cost perfect matching provides a rough estimate of the fewest moves needed to solve a Sokoban board by directly linking each box to a goal. However, this method does not consider any obstacles on the board, the position of the player, or the constraint that boxes can be pushed and not pulled. Notably, the reward is not computed based on the state $s \in \mathcal{S}$ and the action $a \in \mathcal{A}(s)$, but rather it is derived from the state resulting from the application of action a in state s.

4.2 Deadlocks

To manage the potentially vast state spaces in Sokoban deadlock detection is essential. The algorithms in this thesis treat deadlock states as terminal, preventing their exploration. This approach enhances the efficiency of computational resources. Section 4.2.3 details utilizing information about deadlocks.

4.2.1 Detecting Deadlocks

Trivially, any state without an available action is a deadlock state. However, detecting deadlocks earlier is a complex task and is not the primary focus of this thesis. The following section discusses deadlock detection, focusing only on identifying basic deadlocks and not encompassing all possible scenarios. Enhancing deadlock detection capabilities would improve the performance of the algorithms discussed in this thesis.

Locking Tiles

Some tiles of a Sokoban board immediately cause deadlocks. A corner without a goal is a typical example since any box pushed into such a corner immediately results in a deadlock. In this thesis, such tiles are referred to as Locking Tiles because any box moved onto them becomes locked in place, regardless of the positions of the other pieces on the board.

The following algorithm identifies Locking Tiles by allowing the agent to pull boxes instead of pushing them. The process begins by clearing the board of all boxes and placing a box on a selected goal square. For each of the four directions (up, down, left, and right), the player is positioned adjacent to the box (above, below, left, or right). The player then executes every possible pull, exploring all reachable tiles. Tiles occupied by the box during these pulls are recorded. Once this procedure is completed for all directions and goal squares, the algorithm identifies tiles that never hosted a box. Those tiles are locking tiles, as no sequence of pushes can move a box from any of those tiles to a goal. Algorithm 2 summarized the procedure just described.

Algorithm 2 Find Locking Tiles

```
1: Clear the initial board of all boxes and the player.
2: n ← height of the level.
3: m ← width of the level.
4: marked ← initialize a n × m matrix with all entries set to zero.
5: visitedStates ← initialize an empty list.
6: for each goal in the set of goals on the initial board do
7:     for dir in {up, down, left, right} do
8:         Place a box on goal.
9:         if There is no wall in the direction dir from goal then
10:            Place the player adjacent to the box in the direction dir.
11:            Initialize an empty queue Q.
12:            hash ← computeHash(initialBoard).
13:            visitedStates.append(hash)
14:            Enqueue initialBoard into Q.
15:            marked[box_x, box_y] ← 1          ▷ Mark the box's initial position as visited
16:            while Q is not empty do
17:                currentBoard ← dequeue from Q.
18:                for each move available from currentBoard do
19:                    newBoard ← currentBoard.applyMove(move)      ▷ Get new board
20:                    newHash ← newBoard.getHash()
21:                    if newHash not in visitedStates then
22:                        visitedStates.append(newHash)
23:                        marked[newBoard.box_x, newBoard.box_y] ← 1
24:                        Enqueue newBoard into Q.
25:                    end if
26:                end for
27:            end while
28:        end if
29:    end for
30: end for
31: All tiles in marked not set to 1 are identified as freezing tiles.
```

Wall Deadlocks

In Sokoban, boxes positioned against an outer wall cannot be pushed away from the wall due to the absence of floor tiles in the required direction. For instance, consider two boxes aligned against the leftmost wall. It is not possible to move these boxes to the right. Thus, the board is deadlocked if there are fewer than two goals along this wall. This specific type of deadlock is known as a wall deadlock. The same principle applies to all four boundaries of the game board.

Algorithm 3 detects wall deadlocks along the leftmost boundary. It can be easily adapted to handle any of the other three directions. As indicated in Figure 4.6, this procedure does not detect all wall deadlocks.

Algorithm 3 Wall Deadlock Detection for the Leftmost Boundary

```
1:  Initialize a sweepline at the leftmost boundary
2:  while sweepline has not reached the right boundary do
3:      if sweepline encounters non-wall elements then
4:          boxes ← count boxes on sweepline
5:          goals ← count goals on sweepline
6:          if boxes ≤ goals then
7:              return False          ▷ no wall deadlock detected on the leftmost wall
8:          else
9:              return True           ▷ wall deadlock detected on the leftmost wall
10:         end if
11:     end if
12:     Move the sweep line one step to the right
13: end while
```

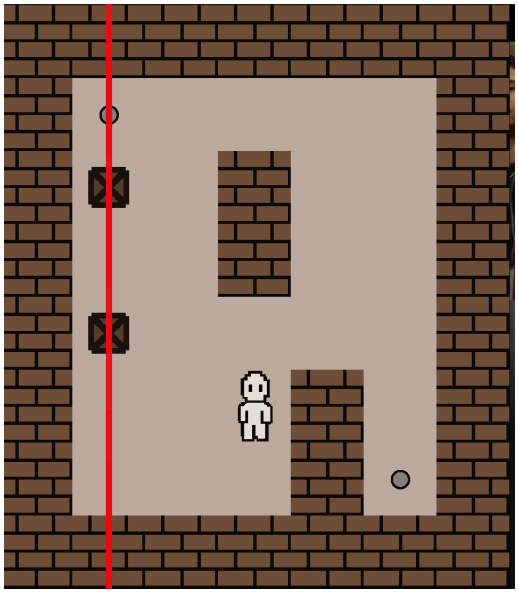


Figure 4.4: The red sweep line detects a wall deadlock. Since neither of the two boxes can reach the goal on the right of the screen, the board is deadlocked.

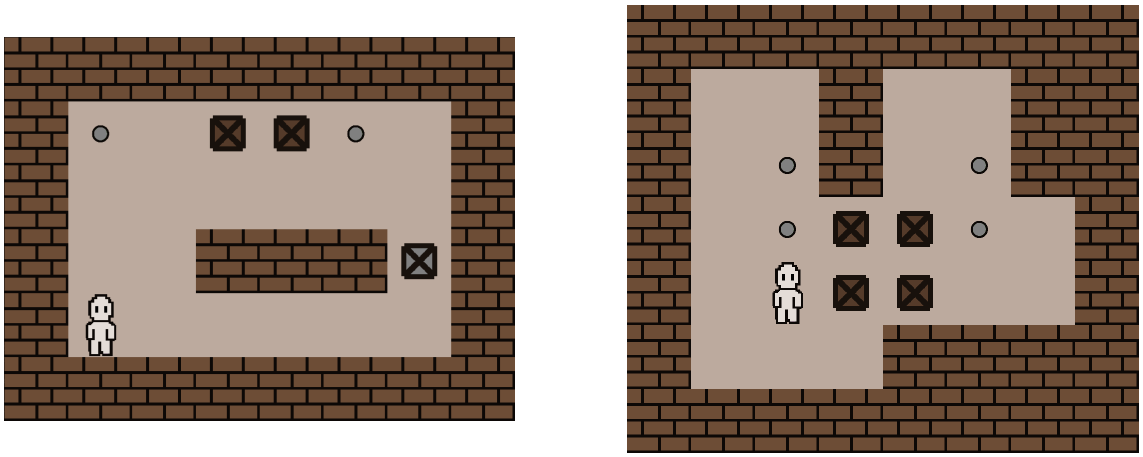


Figure 4.5: Both sides show examples of boxes locking each other such that they cannot move. Hence, the level is deadlocked.

Locked Boxes

In Sokoban, boxes can block each other, rendering them immovable. A board is deadlocked if a locked box is not on a goal tile. The Sokoban Wiki (2024) provides a method for identifying locked boxes.

To check if a box is locked, one verifies its horizontal and vertical mobility. A box is locked horizontally if any of the following conditions is true: the box has a wall on either its left or right side; the box has locking tiles on both its left and right sides; or the box has a locked box on its left or right side. Similar checks apply for determining if a box is locked vertically. However, caution is necessary for the last check, as it can easily lead to an infinite loop. For example, consider two boxes placed side by side, with the left box being checked horizontally. If the first two checks are negative, the third condition examines if the box on the right is locked. It is horizontally locked because it cannot move left or right due to the presence of the left box. Thus, the right box is locked if it is locked vertically, which in turn means the left box is also locked. In essence, to recursively check if a box is locked horizontally by another box, it is necessary to check if the other box is locked vertically, proceeding in an alternating fashion. However, this approach does not prevent potential infinite recursive checks. For example, consider the right-hand side of Figure 4.5. Starting by horizontally examining the top left box leads to a sequence of checks: top left box horizontally, top right box vertically, bottom right box horizontally, bottom left box vertically, and finally looping back to checking the top left box horizontally. However, once the same box is checked twice in the same direction, the box is part of a square of four boxes and is thus locked.

4.2.2 Undetected Deadlocks

This paragraph highlights some deadlocks undetected by the algorithms in this thesis. Firstly, algorithm 3 does not detect wall deadlocks that are not directly adjacent to a boundary wall, as shown in Figure 4.6. Additionally, there are situations where the player moves into a position that permanently locks off a part of the level, often rendering a board unsolvable. Figure 4.7 depicts an example.

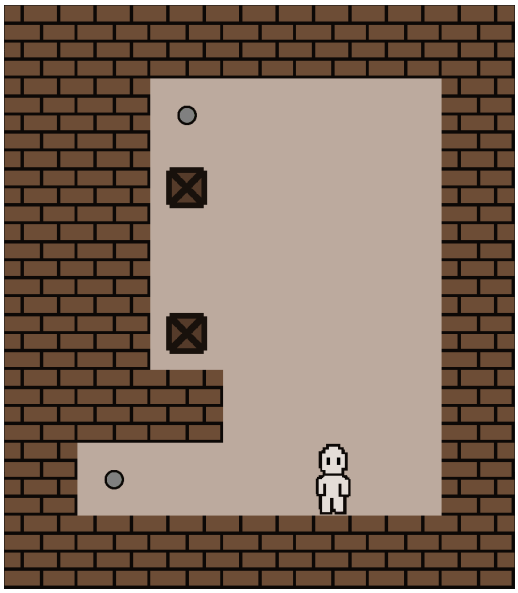


Figure 4.6: Wall deadlocks like this are currently not detected.

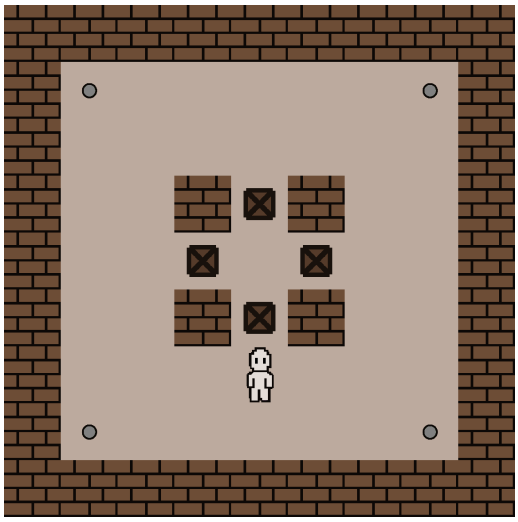


Figure 4.7: The player sealed off the area in the center, and the level is lost but not yet detected as a deadlock.

4.2.3 Recursive Node Elimination

To address information about deadlock states, Crippa et al. (2021) propose performing Recursive Node Elimination after every backpropagation phase. In Recursive Node Elimination, the last node used as the starting point for the simulation phase is removed from the search tree if it represents a deadlock state. This removal triggers a recursive process starting at the just deleted node's parent. If a node has no children, it is deleted, and the same check is applied to its parent. This method effectively eliminates subtrees that have been fully explored but contain no solutions. Eliminating unnecessary subtrees is crucial for optimizing MCTS in Sokoban, as it prevents the selection phase from exploring parts of the tree that lack solutions.

To illustrate the importance of Recursive Node Elimination, consider the alternative ap-

proach of imposing substantial negative rewards for deadlock states. This method has two major drawbacks. First, imagine the root node has two children, each leading to subtrees containing at least one deadlock. During backpropagation, both children have received large negative values, making their selection equally unappealing according to the UCT formula. However, if the first child is the root of a fully explored subtree, it should never be selected since it is known to not lead to a solution. Recursive Node Elimination addresses this by deleting fully explored subtrees. Secondly, in complex Sokoban puzzles, states close to solutions may also closely approach deadlock conditions. From a specific state, two short sequences of moves could lead to either a deadlock or a solution. Severe penalties for deadlocks might prevent MCTS from recognizing potential solutions near deadlock states, as overly penalizing these states could deter exploration of promising areas. Therefore, it is essential not to penalize ancestors of deadlock states too harshly. While these arguments merely illustrate the potential benefits of Recursive Node Elimination, experimental evidence substantiates its effectiveness.

4.3 Modifications to the MCTS

Facilitating MCTS for Sokoban necessitates a modification of the simulation and backpropagation phases.

4.3.1 Modifications to the Simulation Phase

This thesis omits the traditional simulation phase typical to MCTS. Rather than performing a rollout to determine the value of a state, the algorithm directly uses the reward computed from the current board as an estimate. This approach is justified by experimental comparisons of several rollout strategies, none of which significantly improved upon the use of immediate rewards. Detailed results of these experiments are not provided.

The conventional method of using random rollouts in MCTS did not provide significant benefits. In Sokoban, reaching promising states through random actions seems unlikely, which probably contributes to their observed inefficiency in accurately estimating state values. Efforts to enhance the signal quality involved averaging the outcomes of multiple random rollouts. However, reaching a termination state in these simulations often requires considerable time due to the sparse distribution of terminal states, and the computational demand escalates with multiple rollouts. Additional experiments entailed initiating a breadth-first search from the node selected for simulation and continuing up to a predetermined depth. The value of the rollout was determined by first computing the reward associated with each board configuration discovered during the breadth-first search. The maximum of these computed rewards was used for backpropagation. None of these strategies produced results that justified their computational costs, leading to their exclusion from this thesis.

4.3.2 Modifications to the Backpropagation Phase

As detailed in Section 3.2.4, classical MCTS typically involves repeatedly negating the rollout value during backpropagation. However, this negation is unnecessary in deterministic, single-player environments like Sokoban, where no adversary is counteracting the player's moves.

Chapter 5

Schokoban, Eliminating Redundancies in the Search Tree

Chapter 4 introduced Vanillaban for solving Sokoban puzzles. Analyzing Vanillaban reveals that it frequently constructs search trees larger than the total number of possible states. This over-representation occurs because different sequences of moves in Sokoban can result in identical board configurations, leading the MCTS to unintentionally revisit states. In their related discussion on Afterstates, Sutton and Barto (2018, Section 6.8) emphasize that managing redundant states can significantly enhance performance. To do so, Schokoban resolves redundant state representations in the search tree, with its implementation details presented in the subsequent chapter.

5.1 Resolving Redundant State Representations

Consider a scenario during the expansion phase where node i is chosen for expansion. Suppose one of the child nodes about to be added to the tree corresponds to a state $s \in \mathcal{S}$ already represented in the tree, say by node j. If the depth of node i plus one is greater than or equal to the depth of the existing node j, incorporating a new node is redundant. In this case, visiting s via node i represents a path that is longer or equally long compared to the previously discovered route that ended in node j. Conversely, if the depth of node i plus one is less than the depth of node j, this indicates a more efficient path has been found. Following this discovery, the subtree rooted at node j is relocated, making it a direct child of node i. This reconfiguration shortens the paths to all descendant nodes of j. Since the former parent of node j loses a child due to this transfer, reassessing the node's role in the tree is necessary. As described in Section 4.2.3, this involves checking whether Recursive Node Elimination should be initiated starting from the old parent. Algorithm 4 summarizes the procedure described above. Additionally, such reorganization requires updating the tree's statistics to ensure the UCT policy works. Section 5.2 elaborates on these updates.

5.2 Updating the Monte Carlo Search Tree

Assume the subtree rooted at node j needs to be relocated to become a child of node i. Before relocating the subtree, the predecessors of node j are updated. Their visit counts

Algorithm 4 Tree Restructuring in Schokoban

```
1: Create a node l corresponding to state discovered during expansion
2: if State of l already exists in the tree at node j then
3:     if Depth of i + 1 ≥ Depth of j then
4:         Discard node l                                   ▷ Path is not shorter
5:     else
6:         formerParent ← j.parent
7:         j.parent ← i                  ▷ Move subtree rooted at j to be a child of i
8:         formerParent.removeChild(j)   ▷ Remove j as child node of formerParent
9:         Update depths of nodes in subtree rooted at j
10:        Recursively update statistics starting at i and formerParent
11:        Call RecursiveNodeElimination(formerParent)
12:    end if
13: else
14:    Add l as a child of i                                ▷ New state found
15: end if
```

are decreased by the number of descendants removed. Simultaneously, their averages are adjusted to reflect the removal of the descendants' values. Specifically, let n_j and v_j represent the visit count and value of node j respectively. For any predecessor k with visit count n_k and value v_k, the updates are applied using the following formulas:

$$v_k \leftarrow \frac{v_k \cdot n_k - v_j \cdot n_j}{n_k - n_j}$$

$$n_k \leftarrow n_k - n_j$$

After moving the subtree to its new position, the next step involves updating the statistics of node j's new predecessors, beginning with node i. If l represents one of these new predecessors of j, then the visit count n_l and value v_l of node l are adjusted according to the following formulas:

$$v_l \leftarrow \frac{v_l \cdot n_l + v_j \cdot n_j}{n_l + n_j}$$

$$n_l \leftarrow n_l + n_j$$

After relocating the subtree, node j's old parent is left with one fewer child. Thus, if node j's old parent now has no child, the process of Recursive Node Elimination, as detailed in Section 4.2.3, is initiated, pruning redundant nodes from the tree. The final step involves recursively updating the depths of node j and its descendants. Starting with node j, the depth of each node is set to be one more than its parent's depth. This adjustment reflects the discovery of a quicker route to the state represented by node j and, by extension, to all its descendant nodes.

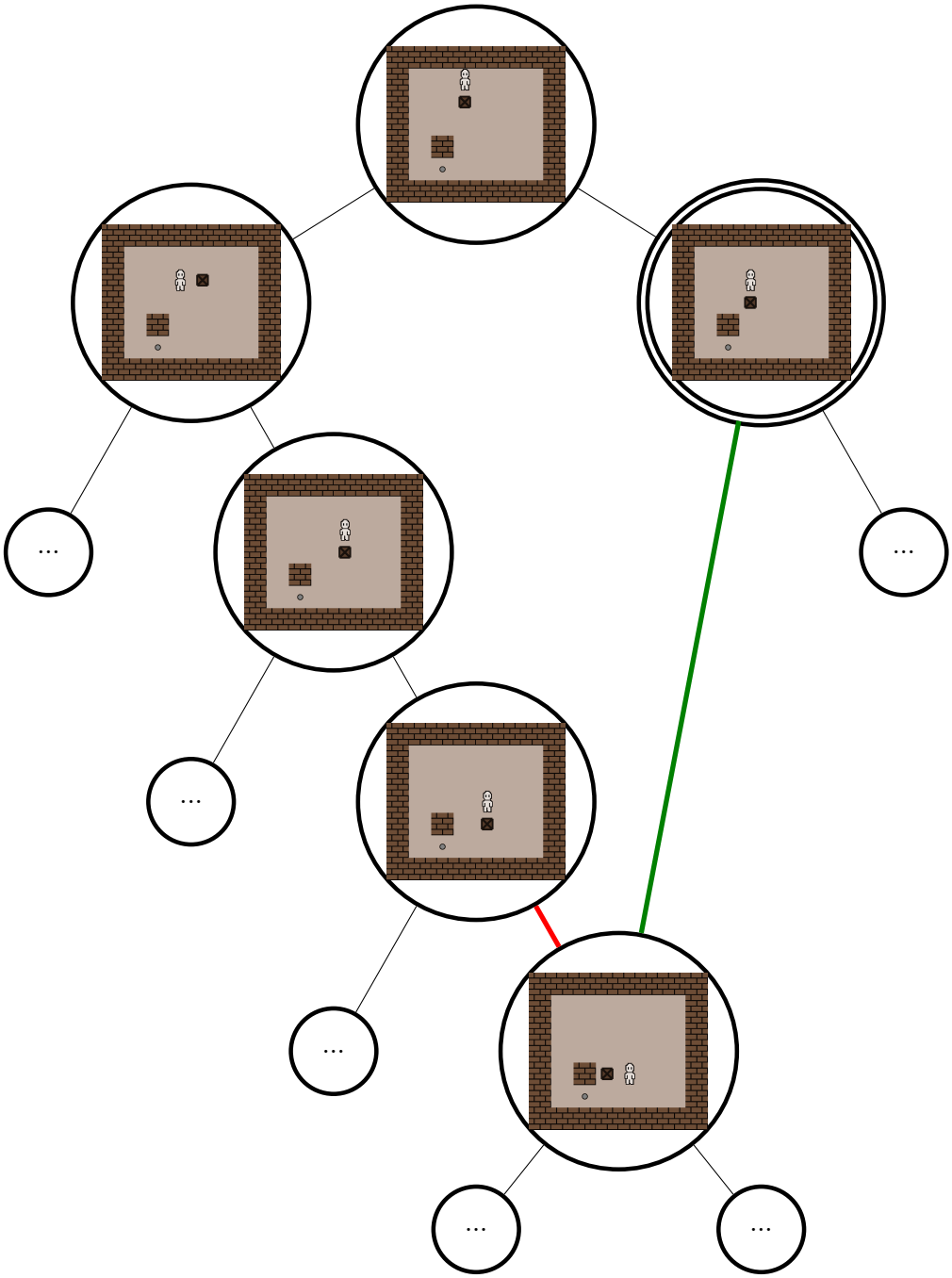


Figure 5.1: The double-encircled node is expanded, discovering that one of its successor states is already represented deeper in the tree. To avoid duplication, the corresponding subtree is moved up by adding the green and deleting the red edge. Since Schokoban utilizes equivalence states, boards where the player can move freely within the same connected component and where all box positions are identical are considered the same.

Chapter 6

Experimental Results

This chapter provides a comparative analysis of Vanillaban, as discussed in Chapter 4, and Schokoban, as detailed in Chapter 5. The objective is to assess the effectiveness and efficiency of each approach by testing them on two sets of Sokoban levels, which differ markedly in complexity and design. To facilitate this comparison, each algorithm is given a fixed number of iterations to complete a level. One iteration refers to a complete cycle through the four stages of the MCTS, including selection, expansion, simulation, and backpropagation phases. The data supporting the figures and tables in this section are available for review and replication here.

Computational Resources

All experiments discussed in this section were conducted on a *Lenovo ThinkPad X1 Carbon Gen 9*, featuring an *Intel Core i7-1165G7 11th Gen* and 16 GB RAM. In this study, the number of iterations each algorithm can perform to solve a level is variable, but most experiments are halted at 100,000 iterations. This limit is set based on performance constraints observed on a *Lenovo ThinkPad X1 Carbon Gen 9*. For levels with numerous boxes, surpassing the suggested iteration threshold can lead to runtimes exceeding ten minutes.

CBC Level Collection

The first test set includes 60 levels sourced from the CBC Sokoban Level Collection (2024). Visualizations of these levels are accessible online, with some selected examples presented in Section 6.2. They start out trivial but become increasingly more challenging. Vanillaban solves most levels, whereas Schokoban solves all, as illustrated in Table 6.1. Additionally, while Schokoban always solves any level that Vanillaban manages within a fixed iteration limit, it often requires fewer iterations, as depicted in Figure 6.1.

Number of Iterations	25	50	100	500	1000	2000	5000	10000	100000
Vanillaban	4	12	16	38	41	44	49	49	50
Schokoban	9	23	41	60	60	60	60	60	60

Table 6.1: The table above shows the number of levels, out of the 60 in the CBC Sokoban Level Collection (2024), the respective algorithm solves within the given number of iterations. Within a fixed iteration limit, Schokoban solves all the levels that Vanillaban does.

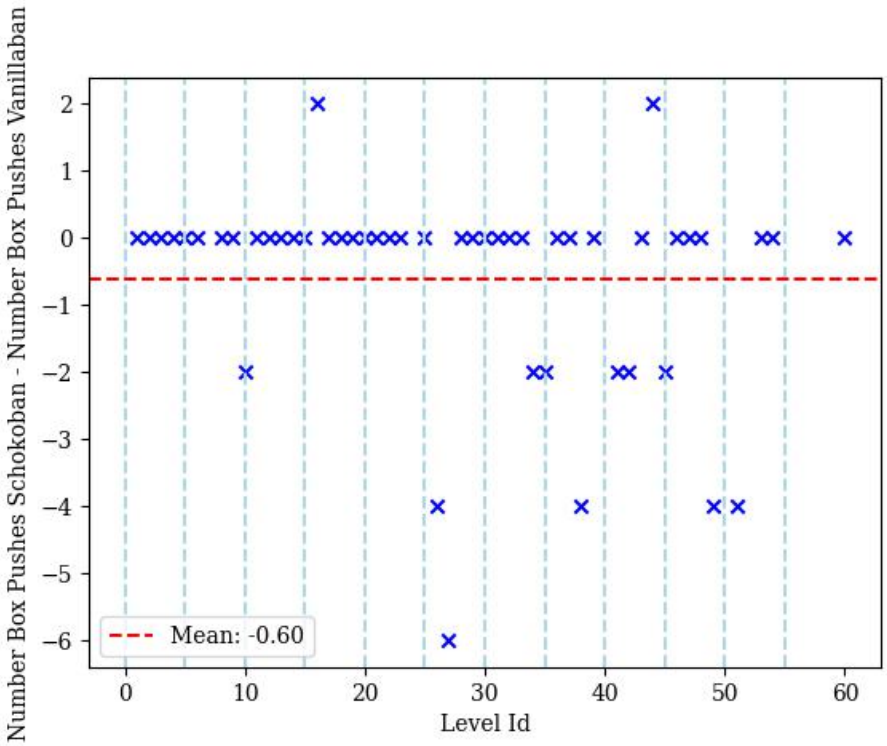


Figure 6.1: For each level in the CBC Sokoban Level Collection (2024) that Vanillaban solves within 100,000 iterations, the plot depicts the difference in solution lengths between Schokoban and Vanillaban. By the design of state and action spaces for these algorithms, the solution length equals the number of box pushes each algorithm performs. The data demonstrate that, on average, Schokoban achieves shorter solution sequences than Vanillaban. For reference, the average number of box pushes by Vanillaban across these levels is approximately 12.

Microban Level Collection

The second level collection, composed of 100 levels, developed by Skinner (2009), comprises more intricate puzzles, with a corresponding increase in search space complexity as detailed in Appendix A. Again, visualizations are provided online, with a few selected examples shown in Section 6.2. Schokoban solves a significant fraction of those levels, while Vanillaban handles a considerably smaller share, as shown in Table 6.2. Furthermore, Schokoban always solves any level that Vanillaban can manage within a fixed iteration limit, typically needing fewer iterations, as illustrated in Figure 6.2.

Number of Iterations	25	50	100	500	1000	2000	5000	10000	100000
Vanillaban	0	1	7	18	22	29	34	36	44
Schokoban	0	7	18	50	60	70	77	80	90

Table 6.2: The table above shows the number of levels, out of the 100 levels in the level collection by Skinner (2009), the respective algorithm solves within the given number of iterations. Within a fixed iteration limit, Schokoban solves all the levels that Vanillaban does.

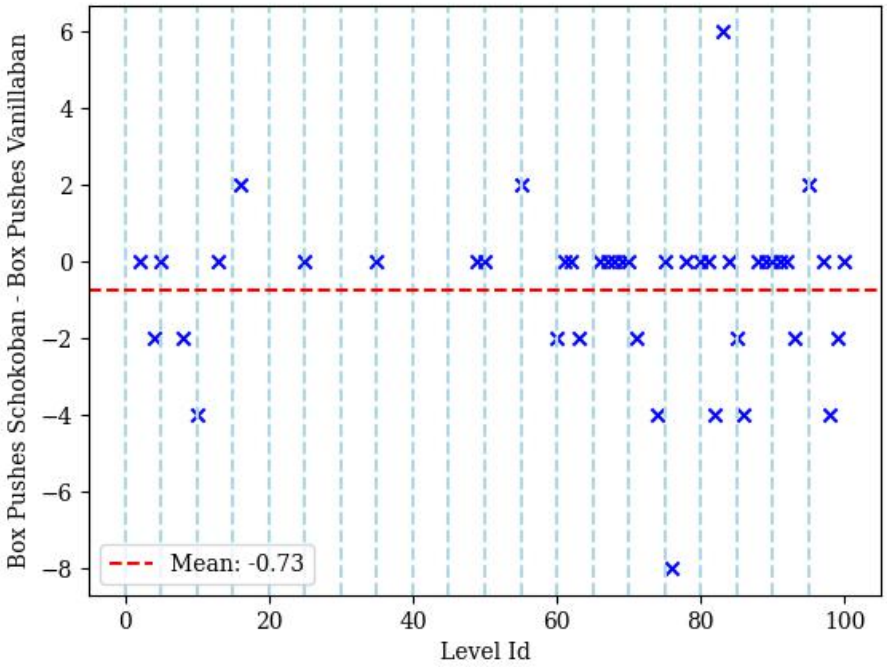


Figure 6.2: For each level in the Microban level collection by Skinner (2009) that Vanillaban solves within 100,000 iterations, the difference in solution lengths between Schokoban and Vanillaban is calculated. Again, the solution length equals the number of box pushes each algorithm performs. The plot reveals that Schokoban, on average, finds shorter solutions than Vanillaban. The average solution length achieved by Vanillaban for these levels is roughly 17.

Performance on Other Popular Sokoban Puzzles

The algorithms in this thesis are robust enough to address levels outside the two collections mentioned above. For instance, the first level from the original Sokoban game, as depicted in Figure 6.3, can be tackled by Schokoban within 250,000 iterations. Vanillaban does not successfully tackle the original level one within 250,000 iterations. Level 78 in the XSokoban level collection by Myers (2001), frequently used as a performance benchmark for state-of-the-art Sokoban solvers, is solved by Schokoban within 250,000 iterations. Vanillaban is not able to match that performance. Level 78 is shown in Figure 6.4.

6.1 Discussion

These experiments demonstrate that Schokoban outperforms Vanillaban in two key aspects. Firstly, Schokoban matches and frequently exceeds Vanillaban's capacity to solve levels within a predetermined number of iterations. Whenever Vanillaban completes a level within a given limit, Schokoban can also solve it, typically requiring fewer iterations but never needing more. Furthermore, Schokoban can solve numerous levels beyond Vanillaban's capabilities. These conclusions are supported by the results shown in Tables 6.1 and 6.2. Secondly, Schokoban tends to find solutions that require fewer box pushes, as highlighted in Figures 6.1 and 6.2. Although Schokoban typically discovers more straight-

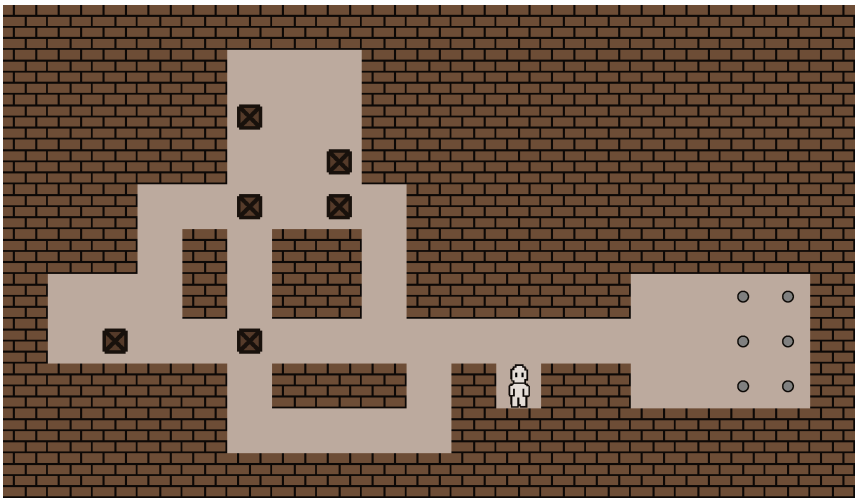


Figure 6.3: The first Level in the original Sokoban game has an estimated search space complexity of approximately 10^8 and is solved by Schokoban within 250,000 iterations.

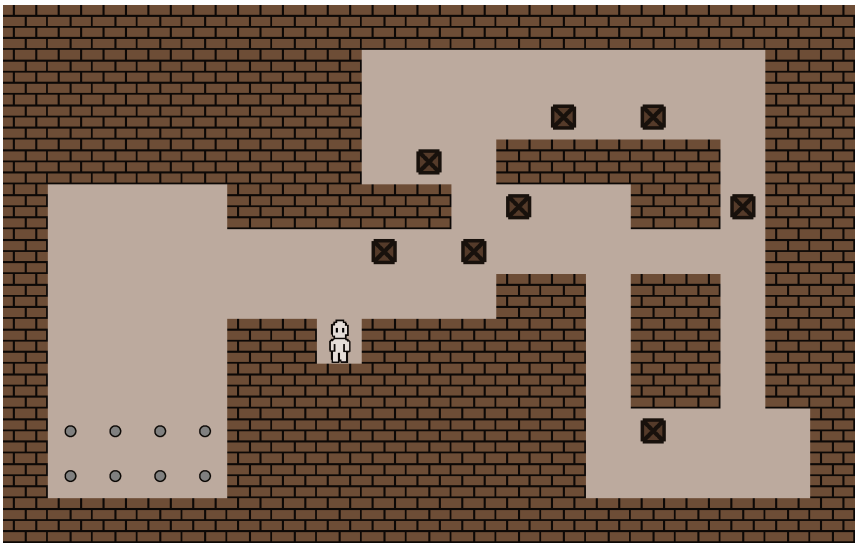


Figure 6.4: Level 78 in the XSokoban Collection by Myers (2001) has an estimated search space complexity of approximately 10^{14}. Sokoban solves it within 100,000 iterations.

forward solutions than Vanillaban, it is not guaranteed to find push optimal solutions for Sokoban puzzles.

6.1.1 Evaluating State Space Exploration and Its Impact on Solving Capabilities

Given that Sokoban and Vanillaban only differ in their handling of redundant nodes, the observed disparities in performance might be explainable by Vanillaban's tendency to revisit some states repeatedly, resulting in an inflated search tree containing redundant nodes. This inefficiency becomes evident when Vanillaban constructs a search tree that surpasses the search space complexity for a level without finding a solution, indicating that many states are included multiple times within the tree. Level 7 in the CBC Sokoban Level Collection (2024) has an estimated search space complexity of 1,512. Despite this, Vanillaban fails to find a solution even after 100,000 iterations, having added approxi-

mately 100,000 nodes to the search tree. Every time Vanillaban revisits the same state, it effectively misses the opportunity to discover a new, more promising state. In contrast, Schokoban's method ensures the continual discovery of new states, raising the question of whether Schokoban's effectiveness is primarily due to its comprehensive exploration of the state space. Under this hypothesis, a simple algorithm like breadth-first search, which systematically explores the entire state space, should match Schokoban's performance. However, initial evidence indicates that Schokoban often resolves levels before its search tree encompasses a significant portion of the estimated state space complexity. For instance, in Level 55 of the Microban collection, illustrated in Figure 6.5, Schokoban identifies approximately 250,000 equivalence states, as defined in Section 4.1.1, before finding a solution, despite the level's estimated search space complexity in the order of 10^{10}. This example suggests that Schokoban efficiently prioritizes more promising states early in the search, an advantage over a simple breadth-first search. However, the reliability of this

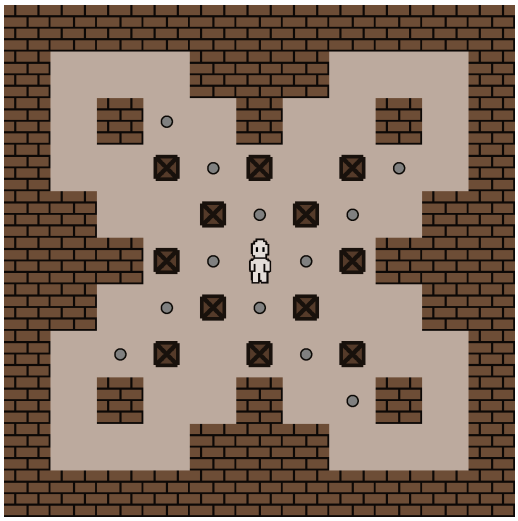


Figure 6.5: After less than 100,000 iterations, Schokoban solves level 55 above.

indication should be approached cautiously for two reasons. First, the estimated upper bound likely greatly overestimates the actual search space complexity for this particular level of Sokoban. Second, this estimate counts simple states defined by the precise positions of the player and boxes, which differs from the equivalence states Schokoban employs. Therefore, to see if a conventional breadth-first search could match Schokoban's performance, it is applied to Level 55 for 100,000 iterations, during which it discovered nearly 10^6 equivalence states but failed to solve the level. The inability of breadth-first search to solve Level 55 under these conditions suggests that Schokoban's effectiveness extends beyond merely iterating over the entire state space. Schokoban shows that it can strategically prioritize promising states in its exploration of the state space.

6.2 Examples

This section includes several illustrative examples, providing an intuitive insight into Schokoban's capabilities. Figure 6.6 features two levels from the CBC Sokoban Level Collection (2024), completed by both Vanillaban and Schokoban. Conversely, Figure 6.7 illustrates levels that only Schokoban can complete. Figure 6.8 presents two more complex levels designed by Skinner (2009), which Schokoban addresses effectively, unlike Vanilla-

ban. Lastly, Figure 6.9 displays two additional levels from Skinner (2009) that prove too challenging for both algorithms.

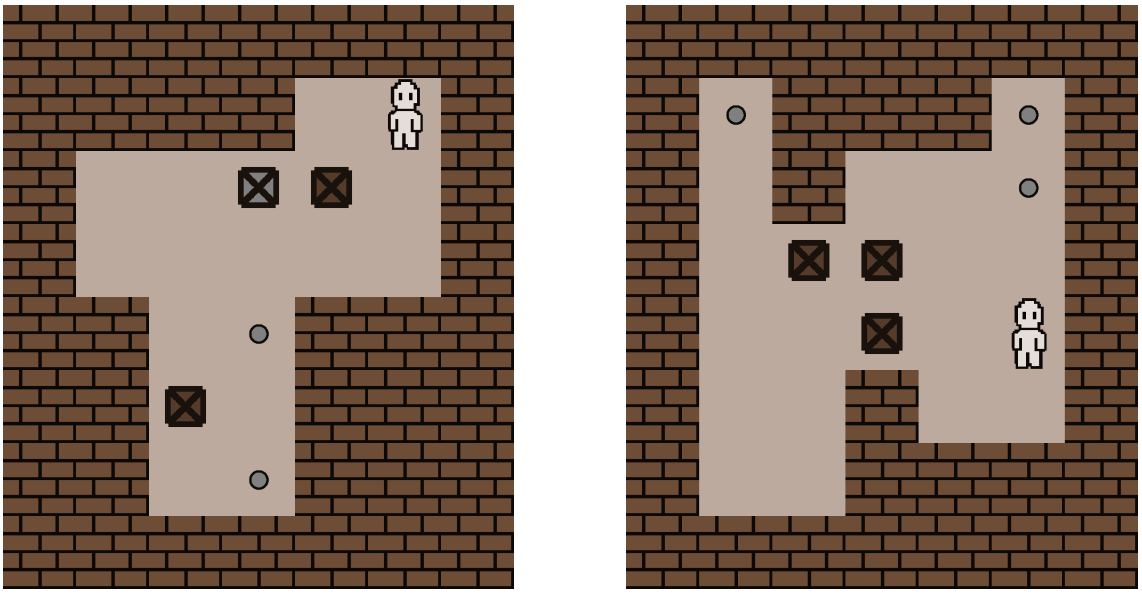


Figure 6.6: Both Vanillaban and Schokoban can solve the levels above within 100,00 iterations.

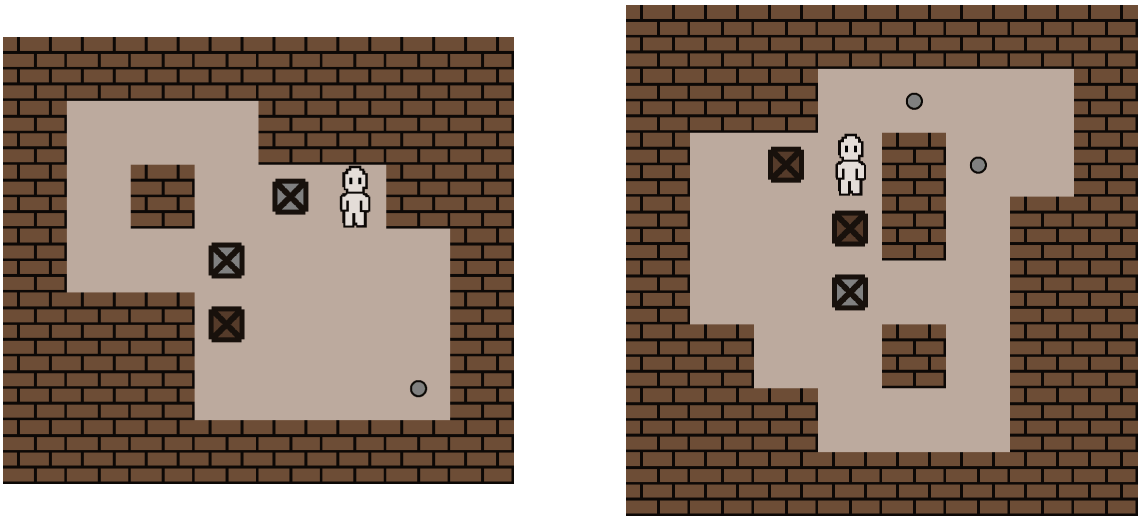


Figure 6.7: Given 100,00 iterations, Schokoban solves these levels while Vanillaban does not.

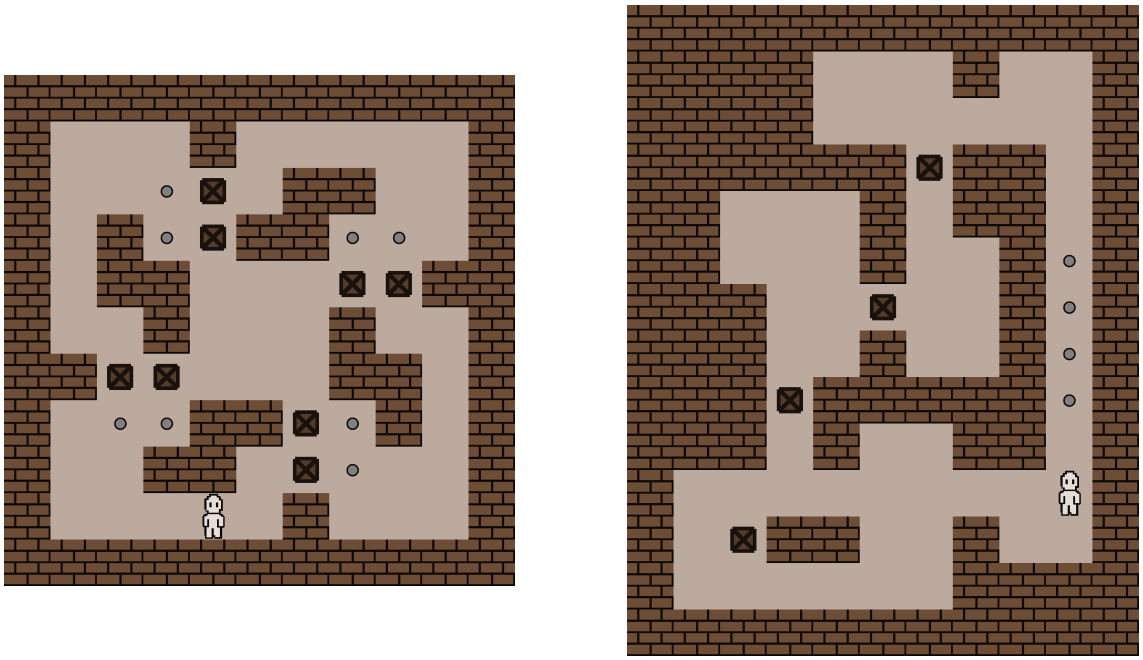


Figure 6.8: These levels from the Microban collection are solved by Schokoban within 100,000 iterations but not by Vanillaban.

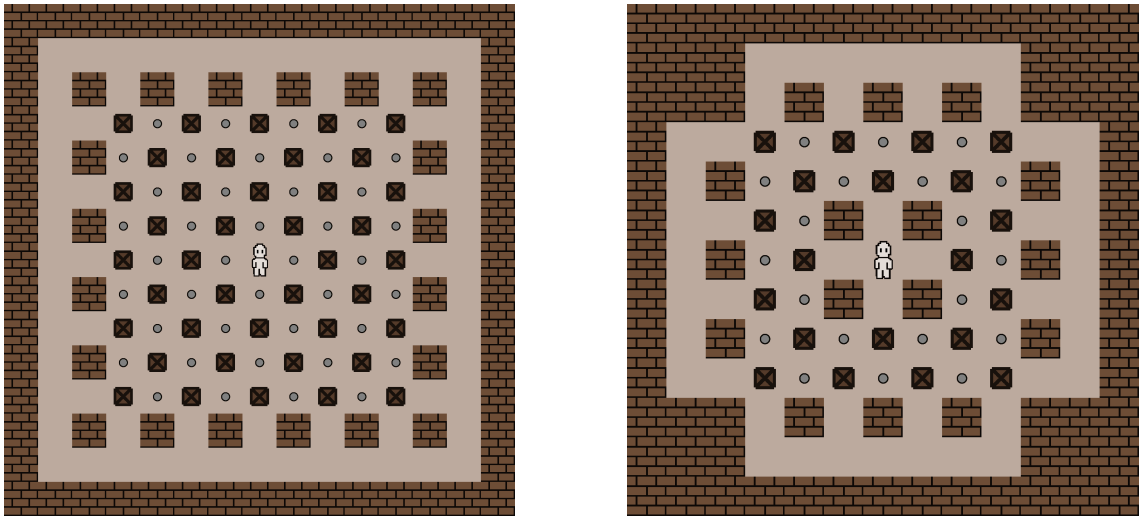


Figure 6.9: Neither Vanillaban nor Schokoban solves any of the levels above within 100,000 iterations.

Chapter 7

Conclusion

By introducing Schokoban, the research presented in this thesis shows how an MCTS-based solver for Sokoban can avoid redundant state representations in its search tree. The conducted experiments underscore the advantages of the proposed method.

While Schokoban is robust within its intended scope, its capabilities are limited, and it does not rank as a state-of-the-art Sokoban solver. The puzzles employed for testing Schokoban range from easy to intermediate. Top-performing Sokoban solvers typically do not employ MCTS strategies.

Building on the insights gained, the following recommendations are proposed to enhance the effectiveness of the Schokoban approach. A primary area for improvement is refining the algorithm itself, particularly by improving its deadlock detection capabilities. Future development should focus on identifying the undetected deadlocks described in Section 4.2.2. Another avenue for enhancement involves optimizing the existing algorithm. For instance, simplifying the implementation of the state representations could significantly reduce memory usage. Implementing the algorithm in a programming language more suited for intensive search processes, such as C or C++, especially for critical components like deadlock detection and reward calculation, could dramatically increase computational efficiency. To explore the potential of deep learning in this context, a neural network trained to estimate the value of Sokoban boards could be integrated into the Schokoban framework to determine rollout values, drawing inspiration from the methodologies used in AlphaGo, as discussed in Silver et al. (2016).

Bibliography

Agostinelli, F., S. McAleer, A. Shmakov, and P. Baldi (2019). Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence 1*(8), 356–363.

Allis, V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ponsen & Looijen.

Ameneyro, F. V. and E. Galvan (2022). Towards understanding the effects of evolving the MCTS UCT selection policy. In *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE.

Amodei, D., C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané (2016). Concrete problems in AI safety. arXiv preprint arXiv:1606.06565.

Arneson, B., R. Hayward, and P. Henderson (2011). Monte Carlo Tree Search in Hex. *Computational Intelligence and AI in Games, IEEE 2*, 251 – 258.

Auer, P., N. Cesa-Bianchi, and P. Fischer (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning 47*(2), 235–256.

Balyoáš, T. and N. Froleyks (2022). AI assisted design of Sokoban puzzles using automated planning. In M. Wölfel and J. B. S. Thiel (Eds.), *ArtsIT, Interactivity and Game Creation*, Cham, pp. 424–441. Springer International Publishing.

CBC Sokoban Level Collection (2024). https://www.cbc.ca/kids/games/play/sokoban. Accessed: May 16, 2024.

Chaslot, G., S. C. J. Bakkes, I. Szita, and P. Spronck (2008). Monte-Carlo Tree Search: A new framework for game ai. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 4*(1), 217–217.

Coulom, R. (2007). Efficient selectivity and backup operators in Monte-Carlo Tree Search. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers (Eds.), *Computers and Games*, Berlin, Heidelberg, pp. 72–83. Springer Berlin Heidelberg.

Crippa, M., P. L. Lanzi, and F. Marocchi (2021). An analysis of single-player Monte Carlo Tree Search performance in Sokoban. *Expert Systems with Applications 192*, 4–14.

Culberson, J. (1997). Sokoban is pspace-complete. Technical report, University of Alberta.

Dawood, M., N. Dengler, J. de Heuvel, and M. Bennewitz (2023). Handling sparse rewards in reinforcement learning using model predictive control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 879–885.

Dor, D. and U. Zwick (1999). Sokoban and other motion planning problems. *Computational Geometry 13*(4), 215–228.

Feinberg, E. and A. Shwartz (2002). *Handbook of Markov Decision Processes: Methods and Applications*, Volume 40. Springer Science+Business Media New York.

Fryers, M. and M. Greene (1995). Sokoban. *Eureka 54*, 25–32.

Gupta, A., A. Pacchiano, Y. Zhai, S. Kakade, and S. Levine (2022). Unpacking reward shaping: Understanding the benefits of reward engineering on sample complexity. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), *Advances in Neural Information Processing Systems*, Volume 35, pp. 15281–15295. Curran Associates, Inc.

Junghanns, A. (1999). *Pushing the Limits: New Developments in Single-Agent Search*. PhD thesis, University of Alberta.

Junghanns, A. and J. Schaeffer (2001a). Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence 129*(1), 219–251.

Junghanns, A. and J. Schaeffer (2001b). Sokoban: Improving the search with relevance cuts. *Theoretical Computer Science 1-2*(252), 151–175.

Kemmerling, M., D. Lütticke, and R. H. Schmitt (2024). Beyond games: a systematic review of neural Monte Carlo Tree Search applications. *Applied Intelligence 54*, 1020–1046.

Kocsis, L. and C. Szepesvári (2006). Bandit based Monte-Carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou (Eds.), *Machine Learning: ECML 2006*, Berlin, Heidelberg, pp. 282–293. Springer Berlin Heidelberg.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence 27*(1), 97–109.

Kozen, D. C. (1992). *Depth-First and Breadth-First Search*, pp. 19–24. New York, NY: Springer New York.

Krakovna, V., J. Uesato, V. Mikulik, M. Rahtz, T. Everitt, R. Kumar, Z. Kenton, J. Leike, and S. Legg (2020). Specification gaming: the flip side of AI ingenuity. DeepMind Blog, https://www.deepmind.com/blog/specification-gaming-the-flip-side-of-ai-ingenuity, Accessed: April 4, 2024.

Murase, Y., H. Matsubara, and Y. Hiraga (1996). Automatic making of Sokoban problems. In N. Foo and R. Goebel (Eds.), *PRICAI'96: Topics in Artificial Intelligence*, Berlin, Heidelberg, pp. 592–600. Springer Berlin Heidelberg.

Myers, A. (2001). Sokoban levels collection. http://www.cs.cornell.edu/andru/xsokoban.html. Accessed: March 6, 2024.

Riedmiller, M., R. Hafner, T. Lampe, M. Neunert, J. Degrave, T. van de Wiele, V. Mnih, N. Heess, and J. T. Springenberg (2018). Learning by playing solving sparse reward tasks from scratch. In J. Dy and A. Krause (Eds.), *Proceedings of the 35th International Conference on Machine Learning*, Volume 80 of *Proceedings of Machine Learning Research*, pp. 4344–4353. PMLR.

Robles, D., P. Rohlfshagen, and S. Lucas (2011). Learning non-random moves for playing Othello: Improving Monte Carlo Tree Search. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pp. 305 – 312.

Russell, S. J. and P. Norvig (2016). *Artificial intelligence: a modern approach*. Pearson.

Schadd, M., M. Winands, M. Tak, and J. Uiterwijk (2012). Single-player Monte-Carlo Tree Search for SameGame. *Knowledge-Based Systems 32*, 3–11.

Shannon, C. E. (1950). Programming a computer for playing Chess. *Philosophical Magazine 41*(314), 256–275.

Shoham, Y. and G. Elidan (2021). Solving Sokoban with backward reinforcement learning. In *Fourteenth International Symposium on Combinatorial Search*, pp. 191–193. AAAI Press.

Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis (2016). Mastering the game of Go with deep neural networks and tree search. *Nature 529*(7587), 484–489.

Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis (2018). A general reinforcement learning algorithm that masters Chess, Shogi, and Go through self-play. *Science 362*(6419), 1140–1144.

Skalse, J., N. Howe, D. Krasheninnikov, and D. Krueger (2022). Defining and characterizing reward gaming. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), *Advances in Neural Information Processing Systems*, Volume 35, pp. 9460–9471. Curran Associates, Inc.

Skinner, D. W. (2009). Microban level collection. http://www.abelmartin.com/rj/sokobanJS/Skinner/David%20W.%20Skinner%20-%20Sokoban.htm. Accessed: March 6, 2024.

Sokoban Wiki (2024). http://www.sokobano.de/wiki/index.php?title=How_to_detect_deadlocks. Accessed: May 16, 2024.

Sutton, R. S. and A. G. Barto (2018). *Reinforcement Learning: an Introduction*. MIT Press.

Teytaud, F. and O. Teytaud (2010). Creating an upper-confidence-tree program for havannah. In H. J. van den Herik and P. Spronck (Eds.), *Advances in Computer Games*, Berlin, Heidelberg, pp. 65–74. Springer Berlin Heidelberg.

Zhou, N.-F. and A. Dovier (2011). A tabled prolog program for solving Sokoban. In *2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pp. 896–897.

Świechowski, M., K. Godlewski, B. Sawicki, and J. Mańdziuk (2023). Monte Carlo Tree Search: a review of recent modifications and applications. *Artificial Intelligence Review 56*(3), 2497–2562.

Appendix A

Search Space Complexity of Sokoban

According to Allis (1994), the search space complexity of a game is defined as the total number of distinct game states reachable through the execution of legal moves starting from the game's initial state. Junghanns (1999, Section 3.2.2) illustrates an approach to estimate the search space complexity of a Sokoban board. Consider a blank 20×20 board. With walls along the perimeter, the playable area reduces to an 18×18 grid, which comprises 324 tiles. Given n boxes distributed across these tiles and assuming boxes are arrangable in any configuration, the total number of potential box configurations is $\binom{324}{n}$. This quantity reaches its maximum when $n = 162$. For $n = 162$, the number of possible configurations is

$$\binom{324}{162} \approx 10^{96}.$$

If one further assumes that the player can move freely among the 162 unoccupied squares, this count needs to be multiplied by 162, leading to a total estimated number of states around 10^{98}. This estimate is flawed in multiple ways. Firstly, the assumption that all $\binom{324}{n}$ configurations are achievable from a fixed starting configuration is unjustified. Secondly, even within a permissible configuration of boxes, it is unlikely that the player can access every unoccupied tile freely. Consequently, the theoretical upper bound of 10^{98} configurations should be considered a substantial overestimate.

The search space complexity can vary significantly depending on the specific Sokoban board under consideration. Therefore, to provide a more grounded perspective on the magnitude of search spaces relevant to this thesis, an upper bound for each level considered is calculated. The estimate is computed slightly differently than described by Junghanns (1999, Section 3.3.2) because it incorporates information about Locking Tiles as defined in Section 4.2.1. An illustrative example is presented for the fifth level in the Microban III level collection by (Skinner, 2009), shown in Figure A.1. Let n be the total number of floor tiles, p the number of Locking Tiles, and b the number of boxes. For level 5 specifically this yields $n = 20$, $p = 9$ and $b = 3$. Assuming that an effective search algorithm does not push boxes onto Locking Tiles and that the player can navigate floor tiles not occupied by a box,

$$\binom{n-p}{b} \times (n-b). \tag{A.0.0.1}$$

is used as an upper bound. Here, $\binom{n-p}{b}$ calculates the possible positions for the boxes, and $n-b$ represents the potential positions for the player. Applied to Level 5, Formula A.0.0.1

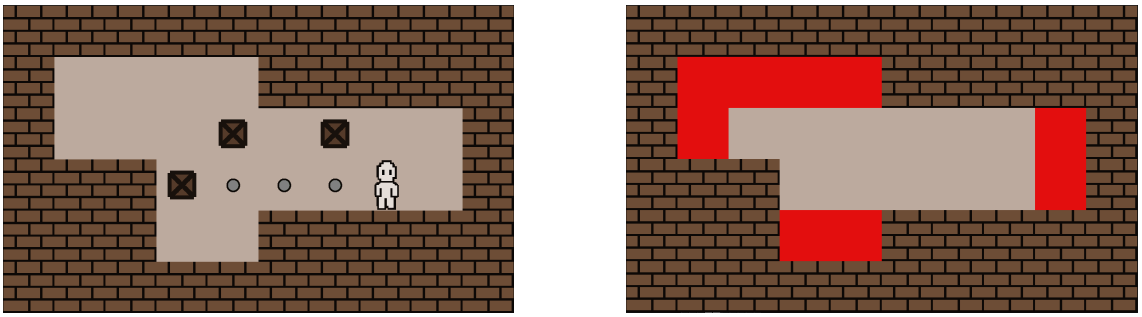


Figure A.1: The red tiles on the right highlight the Locking Tiles in level 5, shown on the left.

yields $\binom{20-9}{3} \times (20-3) = 2805$. In contrast, the approach presented by Junghanns (1999), which does not account for Locking Tiles, gives $\binom{20}{3} \times (20-3) = 22610$ possible configurations.

Strictly speaking, adjusting for Locking Tiles in Formula A.0.0.1 results in some board states being excluded. Consequently, this formula does not estimate the search space complexity but instead the maximum number of possible board configurations that a *sensible* search algorithm might need to evaluate before finding a solution. Here, a *sensible* algorithm refers to an algorithm not continuing to explore states where a box is placed on a Locking Tile, resulting in a deadlock. This adjustment more accurately reflects the number of distinct states the algorithms in this thesis have to encounter before being guaranteed to find a solution. Ideally, more deadlock states should be covered, but adjusting only for Locking Tiles offers the advantage of swift computation. For simplicity and ease of communication, this thesis still refers to the estimate provided by Formula A.0.0.1 as the state space complexity of a level. The estimated search space complexities are detailed below.

Table A.1: The table below provides estimated search space complexities for the levels in the CBC Sokoban Level Collection (2024).

Index	Tiles	non-freezing tiles	Boxes	Search Space Size
1	7	3	1	18
2	12	5	2	100
3	14	5	2	120
4	18	6	2	240
5	20	8	2	504
6	22	9	3	1596
7	21	9	3	1512
8	22	13	3	5434
9	17	12	3	3080
10	20	9	3	1428
11	18	13	3	4290
12	20	12	3	3740
13	19	14	3	5824
14	16	9	3	1092
15	20	6	3	340

Continued on next page

Index	Tiles	non-freezing tiles	Boxes	Search Space Size
16	21	10	3	2160
17	16	10	3	1560
18	20	11	3	2805
19	20	11	3	2805
20	18	12	3	3300
21	15	12	3	2640
22	20	10	3	2040
23	18	10	3	1800
24	19	9	3	1344
25	20	11	3	2805
26	24	15	3	9555
27	20	14	3	6188
28	20	9	3	1428
29	16	11	3	2145
30	19	12	3	3520
31	19	15	3	7280
32	16	10	3	1560
33	19	10	3	1920
34	18	12	3	3300
35	20	13	4	11440
36	20	12	3	3740
37	20	10	3	2040
38	25	17	3	14960
39	22	13	4	12870
40	20	16	3	9520
41	20	8	3	952
42	18	15	4	19110
43	17	12	4	6435
44	17	13	3	4004
45	19	12	3	3520
46	18	10	3	1800
47	18	14	4	14014
48	18	10	3	1800
49	20	12	3	3740
50	20	9	3	1428
51	19	9	3	1344
52	19	12	3	3520
53	18	11	3	2475
54	23	17	3	13600
55	22	15	3	8645
56	19	12	3	3520
57	23	13	3	5720
58	22	12	3	4180
59	21	13	3	5148
60	24	14	3	7644

Table A.2: The table below provides estimated search space complexities for levels in the Microban III collection by Skinner (2009).

Index	Tiles	non-freezing tiles	Boxes	Search Space Size
1	20	14	3	6188
2	16	9	3	1092
3	26	10	3	2760
4	23	13	4	13585
5	20	11	3	2805
6	23	11	4	6270
7	28	20	4	116280
8	64	36	4	3534300
9	43	25	4	493350
10	26	11	5	9702
11	40	23	3	65527
12	24	14	4	20020
13	21	11	4	5610
14	27	14	4	23023
15	25	13	4	15015
16	30	18	4	79560
17	28	15	4	32760
18	29	21	6	1248072
19	30	16	8	283140
20	36	19	6	813960
21	31	19	3	27132
22	56	29	3	193662
23	65	37	8	2200657140
24	61	33	8	735860268
25	38	18	6	594048
26	44	24	3	82984
27	47	30	4	1178415
28	32	16	6	208208
29	40	26	8	49992800
30	32	23	6	2624622
31	50	30	5	6412770
32	65	32	4	2193560
33	74	42	6	356713448
34	23	10	5	4536
35	26	14	6	60060
36	61	29	8	227483685
37	24	18	4	61200
38	28	20	5	356592
39	38	20	5	511632
40	62	35	4	3036880
41	40	22	4	263340
42	41	27	6	10360350
43	37	25	8	31365675
44	38	28	12	790965630

Continued on next page

Index	Tiles	non-freezing tiles	Boxes	Search Space Size
45	51	22	4	343805
46	84	36	12	90120794400
47	85	63	12	194794984575009
48	25	13	4	15015
49	22	12	4	8910
50	23	12	4	9405
51	29	17	4	59500
52	33	20	5	434112
53	35	21	6	1573656
54	56	37	5	22230747
55	63	33	12	18095683320
56	45	33	12	11708971560
57	89	61	20	430328622559398144
58	149	101	40	248085724116628051917213060300 8
59	95	58	13	258757688106960
60	29	11	3	4290
61	21	8	3	1008
62	18	9	3	1260
63	19	9	3	1344
64	34	16	3	17360
65	21	11	3	2970
66	27	12	3	5280
67	21	8	3	1008
68	23	10	3	2400
69	20	9	3	1428
70	25	14	3	8008
71	19	9	4	1890
72	26	12	4	10890
73	29	10	3	3120
74	25	11	3	3630
75	21	10	3	2160
76	24	12	3	4620
77	25	12	3	4840
78	20	9	3	1428
79	20	10	3	2040
80	22	10	3	2280
81	19	9	3	1344
82	18	9	3	1260
83	29	11	3	4290
84	23	9	3	1680
85	24	10	3	2520
86	23	11	3	3300
87	33	11	3	4950
88	37	22	3	52360
89	20	10	3	2040
90	19	8	3	896
91	19	8	3	896

Continued on next page

Index	Tiles	non-freezing tiles	Boxes	Search Space Size
92	27	11	3	3960
93	20	10	3	2040
94	22	11	3	3135
95	21	12	3	3960
96	25	12	3	4840
97	19	9	3	1344
98	22	10	3	2280
99	18	11	3	2475
100	17	8	3	784

Appendix B

MCTS Applied to Tic-Tac-Toe

This chapter applies MCTS to Tic-Tac-Toe, also known as Knots and Crosses. In this game, players try to align three consecutive marks horizontally, vertically, or diagonally on a 3×3 grid. Modeling Tic-Tac-Toe as an MDP, the state space is the set of possible board configurations, while actions are defined as placing a marker on any unoccupied square. The rewards system assigns 1 for a win and -1 for a loss, 0 otherwise. In this model, one player functions as the agent and the other as part of the environment. For convenience, the state diagram illustrating the control flow of the MCTS algorithm is presented again in Figure B.1.

B.1 Building the Search Tree

In the following example, MCTS computes the first move for player X, starting from an empty board. Initially, the root node of the MCTS tree is initialized with a visit count $n_0 = 0$ and a value $v_0 = 0$, depicted in Figure B.2. As the root node has no children, the first selection phase immediately returns the root node. With a visit count of $n_0 = 0$, the algorithm skips expansion and moves directly to the simulation phase. The simulation, which involves random moves by the player and the opponent, is assumed to end in a draw, resulting in a reward of $r = 0$, refer to Figure B.3. This reward is used in backpropagation, updating the root node's visit count n_0 to 1, while v_0 remains unchanged at 0, see Figure B.4. MCTS then re-enters the selection phase, beginning again at the root node. Now, with a visit count n_0 of 1, the algorithm proceeds to its first expansion phase, resulting in the tree depicted in Figure B.5. Following the expansion, one of the newly added nodes is randomly selected for the simulation phase. This time, the simulation results in a loss for player X, achieving a reward of -1, as shown in Figure B.6. During backpropagation, this reward of -1 updates node 2, setting its visit count n_2 to 1 and its value estimate v_2 to -1. Then, the negated reward updates the parent node n_0. The root node's visit count increases to 2, and the average value v_0 is adjusted from 0 to $\frac{0+1}{2} = 0.5$. Figure B.7 shows the updated tree after backpropagation. Then, the algorithm returns to the selection phase, where the root node still hosts several unvisited child nodes, prioritizing them for selection. Consequently, after choosing one of the unvisited nodes for the simulation phase, the simulation's outcome is used for backpropagation. The algorithm repeats this cycle, systematically exploring the root node's remaining unvisited child. Given the repetitive nature of these processes, they are omitted.

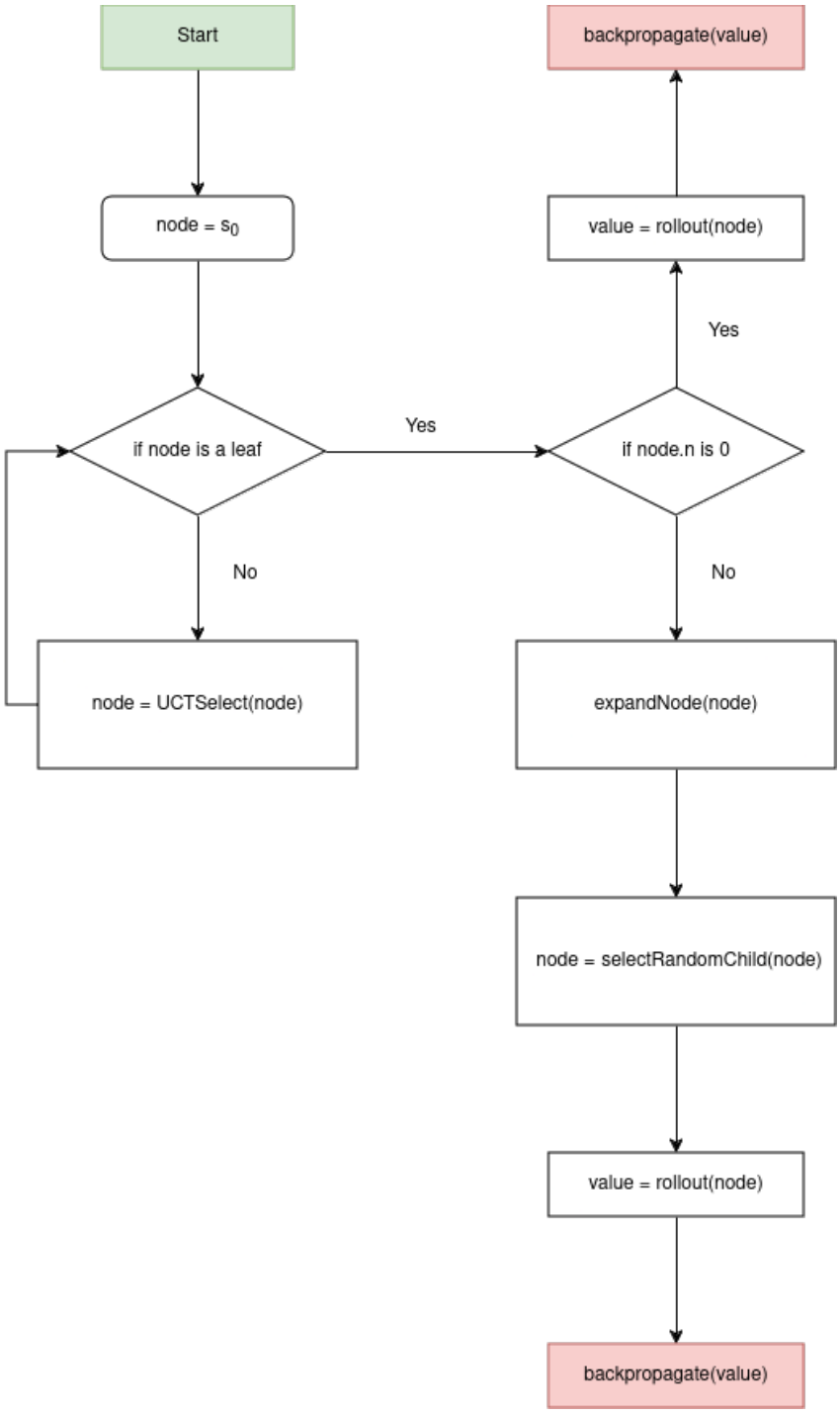


Figure B.1: The search tree for Tic-Tac-Toe is built following this state diagram.

This example progresses to a point where each child node of the root was visited precisely once to demonstrate the application of the UCT formula. Figure B.8 shows the tree's current state, while Table B.1 summarizes the outcomes of the nine simulations initiated from the root node's nine children. Again, the selection phase begins at the root node, but this time, there are no children with a visit count of zero requiring immediate selection. Instead, the UCT formula is employed to identify the most promising child node for further

Node Index	1	2	3	4	5	6	7	8	9
Visit Counts	1	1	1	1	1	1	1	1	1
Node Values	0	-1	1	1	-1	0	1	0	-1

Table B.1: The table contains the statistics of the root's children after the first ten iterations of the MCTS.

exploration. For convenience the UCT formula, as detailed in Equation 3.2.5.1, is recalled here:

$$UCT(node_i) = \bar{X}_i + c \times \sqrt{\frac{2\ln(n_p)}{n_i}}.$$

Thus, the UCT value for the first child node is calculated using the current value of node 1, which is 0, the exploration constant $c = 2$, a parent visit count of 10, and the node's visit count of 1. This calculation yields a value of approximately 4.29. Table B.2 summarizes the UCT values for the remaining nodes following a similar calculation.

Node Index	1	2	3	4	5	6	7	8	9
UCT	4.29	3.29	5.29	5.29	3.29	4.29	5.29	4.29	3.29

Table B.2: The values in Table B.1 are used to calculate the UCT scores above.

Since nodes $3, 4, 7$ tie in UCT score, one is selected randomly. As this arbitrarily chosen node has a visit count larger than zero, it enters the expansion phase. From there, MCTS continues as described above. It cycles through four phases repeatedly until termination. Once MCTS concludes, it returns the move associated with the most visited child node of the root node. In response, the environment generates a new state that includes the opponent's move. Based on this new state, a fresh MCTS is initiated to determine the agent's next move.

At this point, it is crucial to highlight a specific feature of the chosen state space. The environment only presents the player with board configurations where it is the agent's turn, which accounts for approximately half of the states in our state space. A similar observation applies to the actions. Despite this, all states are still included in the state space because they are essential for the MCTS to model the environment. Effectively, MCTS exploits its knowledge about the game's dynamics to model the environment, more precisely the opponent's moves, to compute the next action.

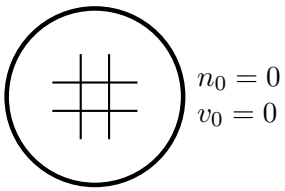


Figure B.2: At the beginning of the MCTS, the search tree only consists of a root node.

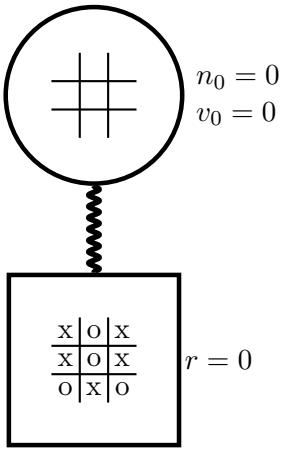


Figure B.3: The first random rollout of the MCST is visualized using a squiggly line. The terminal node representing the outcome of the simulation is not added to the tree.

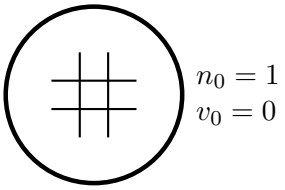


Figure B.4: After the first simulation phase, the root's value is updated during backpropagation.

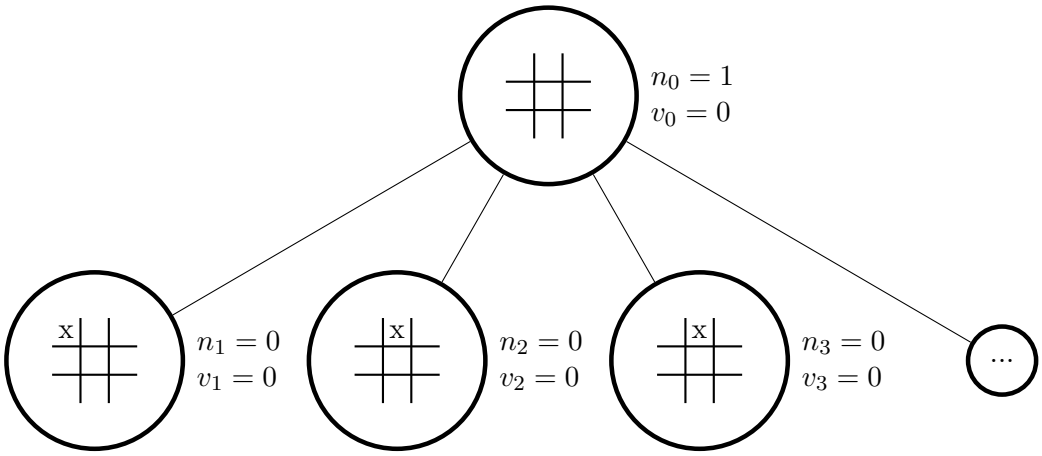


Figure B.5: During the first expansion phase, nodes representing every accessible state are added. However, for clarity in visualization, not all states are shown. The node containing $\cdots$ indicates states that are not displayed.

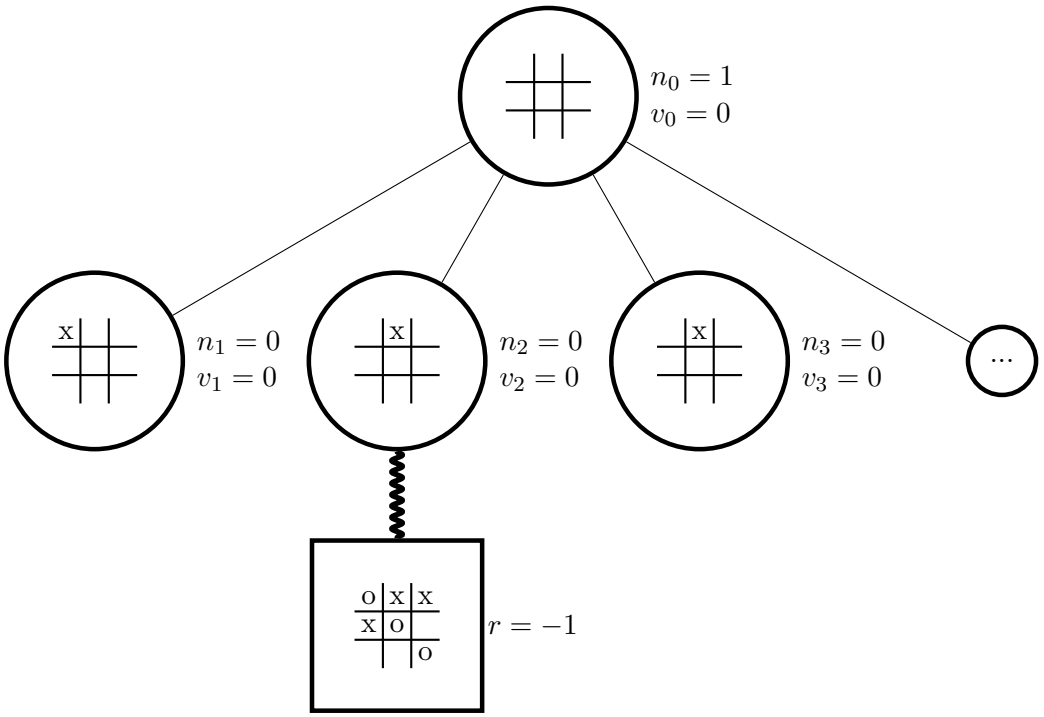


Figure B.6: During the second simulation phase, the MCTS discovers a losing state and receives a reward of -1.

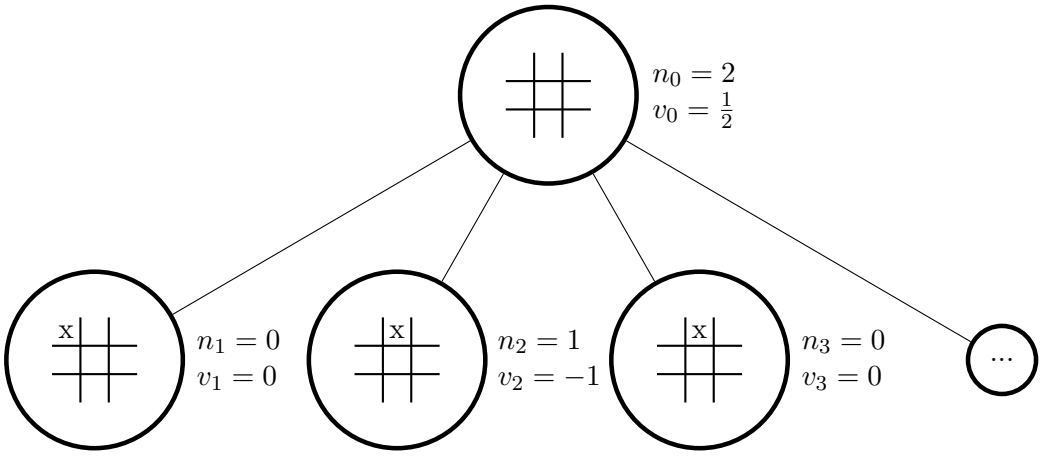


Figure B.7: Two nodes are updated during the second backpropagation phase.

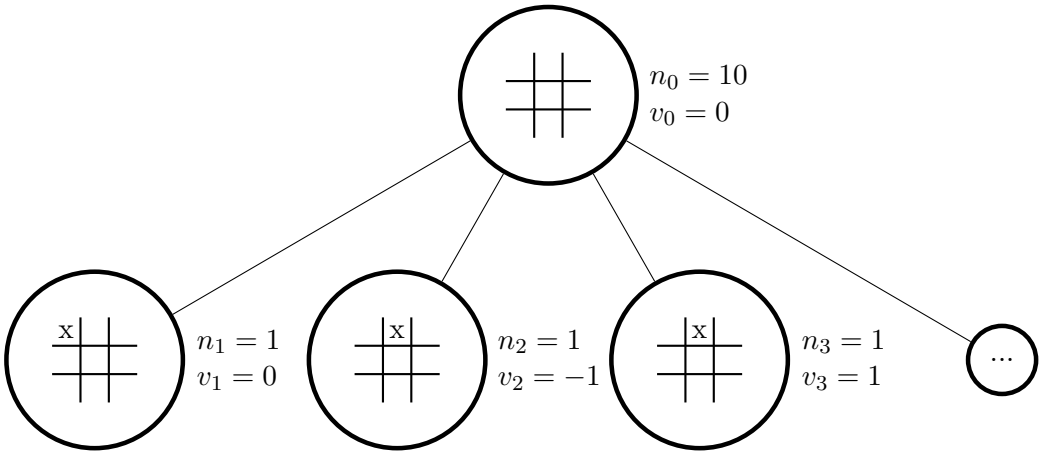


Figure B.8: After the first then iterations, the state of the search tree might look like this.

B.2 Alternating Vales during Backpropagation

An explanation of why alternating the values obtained from the rollout is necessary during backpropagation begins with considering the MCTS in Figure B.9, where MCTS is used to plan the next move for the board shown in the root node. Suppose during the selection phase, the MCTS algorithm selects the double-circled node as a leaf node. Additionally, assume this node has a visit count of zero, prompting MCTS to perform a random simulation. This simulation ends immediately as the state is already terminal, and the resulting win translates to a reward of one. Consequently, $node_3$ is updated. Afterward, $node_2$ receives an update with a value of negative one, which reduces its estimated value, decreasing the UCT score utilized in the next selection phase. This reduction makes further exploration of $node_2$ less likely, justified by the assumption that the agent is competing against an adversary. Thus, the algorithm deems it unlikely that the opponent would select a move, allowing an easy win for the player. As a result, it makes sense that $node_2$'s UCT score decreases. After the update of $node_2$, $node_1$ is updated with a value of 1, reflecting the promising outcome obtained after the selection of $node_1$ in the previous iteration. Consequently, increasing its value during the update supports the algorithm's strategy to prioritize promising leads.

To clarify further, refer to Figure B.10. Assume that during the selection phase, $node_4$ is chosen, and the simulation terminates immediately as the node represents a terminal state. Notably, $node_4$ assigns a reward of 1 despite representing a losing scenario for player X. This is because the simulation at $node_4$ begins from the perspective of player O, for whom this node signifies a winning outcome. During backpropagation, this reward of 1 is negated before updating the value of $node_3$. As a result, MCTS becomes less likely to select $node_3$ in subsequent selection phases. This strategic adjustment aims to steer the algorithm away from moves directly resulting in an opponent's victory.

Summing up, the negation of rewards mirrors the changing perspectives between the player and the opponent, ensuring that the MCTS accounts for the optimal moves of both parties. Within this framework, the nodes at the first level of the tree, directly beneath the root, represent the agent's decisions. From this point, every alternate level reflects the agent's choices, while the intervening levels are associated with the opponent's decisions.

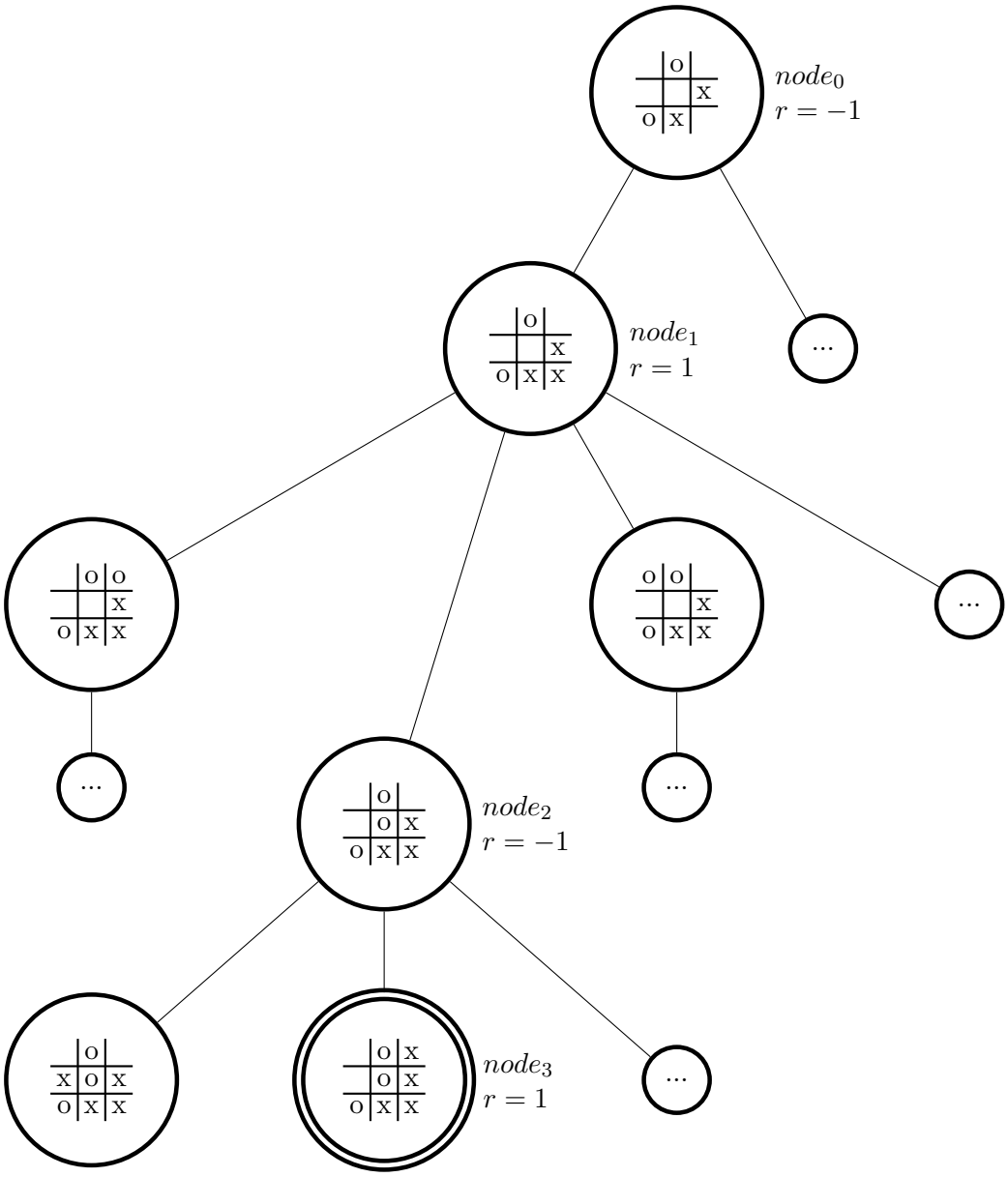


Figure B.9: The double-encircled node selected by the selection phase represents a win for player X. The values used for the updates during backpropagation are indicated next to the nodes.

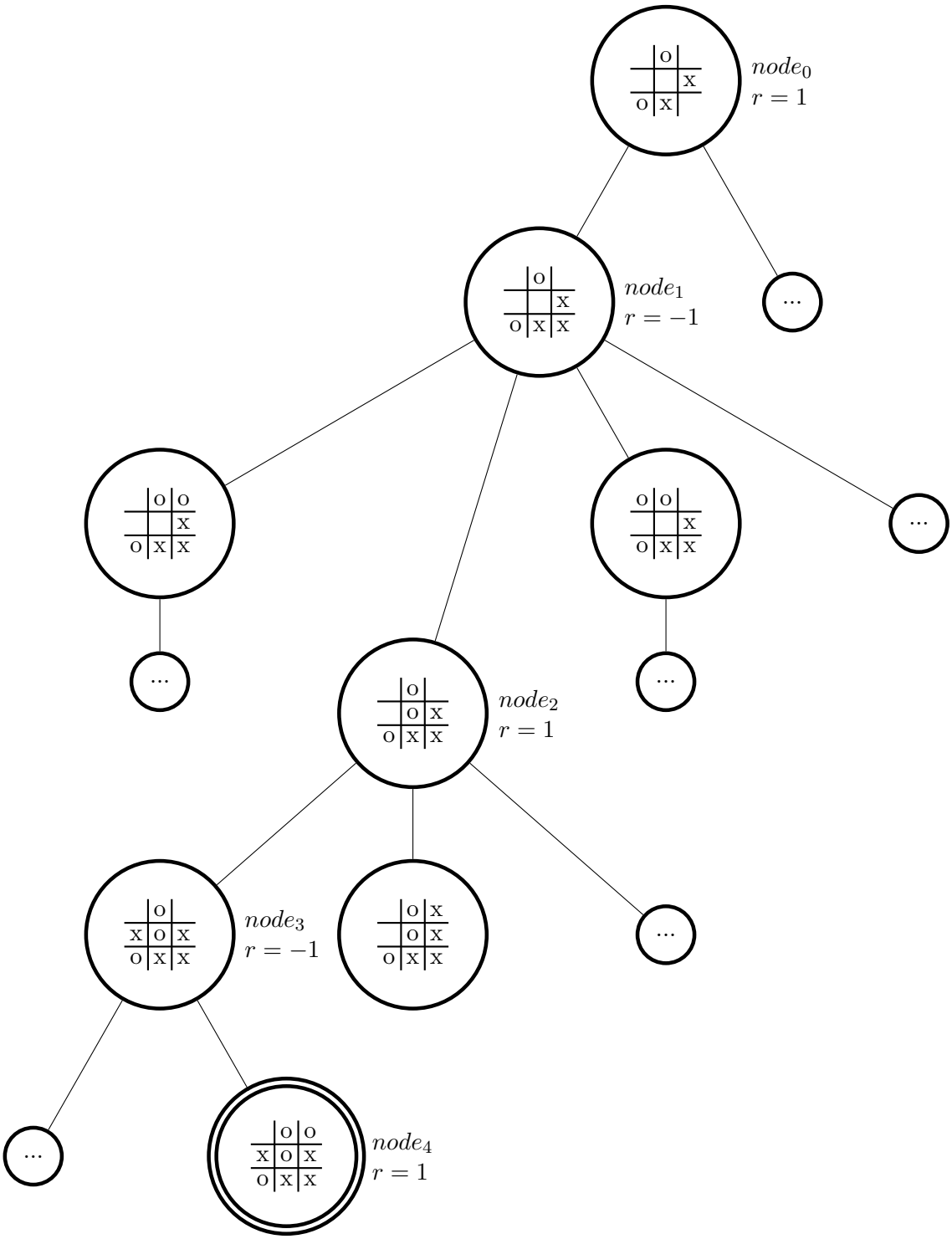


Figure B.10: The double-encircled node selected by the selection phase represents a loss for player X. The values used for the updates during backpropagation are indicated next to the nodes.

Declaration of Originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor .

Title of work (in block letters):

SCHOKOBAN USING MONTE CARLO TREE SEARCH FOR SOLVING SOKOBAN

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
KRÖGER	PAUL

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the Citation etiquette information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work .
- I am aware that the work may be screened electronically for plagiarism.
- I have understood and followed the guidelines in the document *Scientific Works in Mathematics.*

Place, date:

VIENNA JULY 18, 2024

Signature(s):

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.