Solve Sokoban with AlphaZero

Group 3 TDDE19 Project Report

Daniel Purgal (Team Leader)
danpu323@student.liu.se

Johan Christiansson
johch241@student.liu.se

Priyansh Gupta
prigu857@student.liu.se

August Gustafsson
auggu628@student.liu.se

Anders Luong
andlu434@student.liu.se

Jamie Mersh
jamme483@student.liu.se

Daniel Nyström
danny931@student.liu.se

Abstract

This project explores the use of the AlphaZero reinforcement learning algorithm to solve Sokoban, a PSPACE-complete puzzle game. The approach leverages a combination of Monte Carlo Tree Search (MCTS) and a neural network. The implementation includes optimizations to MCTS such as cycle detection and softlock prevention to improve efficiency. While the results demonstrate that the approach can solve smaller boards, the trained models struggled with medium and complex boards, failing to outperform state-of-the-art solvers.

1 Introduction

This section will introduce the problem and discuss some possible solutions.

1.1 Problem description

The project focuses on adapting AlphaZero, a reinforcement learning (RL) algorithm originally designed for games like chess and Go, to the domain of Sokoban puzzles. Sokoban is a puzzle video game where the player pushes boxes around a grid, aiming to place all boxes onto designated goal tiles while navigating obstacles and avoiding getting stuck. While simpler in complexity compared to games like chess, it remains a challenging, PSPACE-complete problem, especially for large puzzle instances. Solving Sokoban is a common challenge in the International Planning Competition (IPC), making it an interesting problem. Traditional Sokoban solvers aim to solve any puzzle, whereas this project seeks to develop a specialized model for each puzzle instance. The goal is to determine whether an AlphaZero-based model, which excels in games with a single, well-defined setup, can be applied to a puzzle game like Sokoban. This will be investigated by training a model for a specific instance of a Sokoban board and see if it solves a board faster than state-of-the-art Sokoban solvers.

1.2 Sokoban Solvers

Current state-of-the-art Sokoban solvers vary in quality and features. There is no single algorithm that will solve every single Sokoban board in an optimal and efficient way. Therefore, one approach might solve a specific board very well while performing suboptimally on a different board compared to other solvers. Some currently developed approaches for solving Sokoban boards are:

- Search algorithms. Algorithms that search all or a limited amount of combination of moves in hope to solve the board. Example of search algorithms are breadth-first search, depth-first search and A*.
- Reinforcement learning. Attempts to maximize cumulative rewards for an agent, often using different approximation heuristics. Common reinforcement learning algorithms are Q-learning and temporal difference learning.
- Precomputed pattern databases. A database of irrelevant paths and deadlocks for each board. Requires handcrafted databases.

Because of the varying effectiveness of different algorithms, the idea of the project is to explore if a neural network approach with an implementation of AlphaZero could outperform the current state-of-the-art Sokoban solvers.

1.3 Possible implementation

Training a model that outperforms state-of-the-art approaches could prove to be difficult. The model needs to be able to see and train on a lot of different states to learn to predict sensible moves. This means many, potentially highly time-consuming, simulations of Sokoban games that need to be done, while still solving the board before classical approaches.

One method for speeding up the simulation of Sokoban games is to use Monte Carlo Tree Search (MCTS). An explanation of MCTS is given in (Cheerla, 2018). This search algorithm tries to balance the trade-off between efficiency and finding an optimal solution, and constitutes a cornerstone of AlphaZero. Optimal search algorithms exist, but work by performing an exhaustive search, which is not viable in practice due to the large number of states that a Sokoban board can have. However, a normal implementation of MCTS, which relies on simulating random moves until the game is decided and updating a search tree with win/loss ratios, would not be efficient either. This is due to Sokoban being a puzzle that only has one possible solution state, which makes win/loss ratios not applicable as a metric for how good an action is for a given state. A solution to this problem is using a modified MCTS where heuristic values are saved in the search tree's nodes instead of win/loss ratios. The implementation and background can be seen in more detail in section 2.1.

Using a neural network approach that then tries to learn heuristic values for a specific Sokoban board has the potential of being better than normal heuristics. For example, in a board state where the agent is far from solving the puzzle, the neural network can extract hidden patterns or features that indicate the state has high potential for progress, even if traditional heuristics might deem it less promising. This allows the modified MCTS to make more informed decisions, potentially accelerating convergence toward a solution.

2 Methods

This section describes the implemented methods and algorithms in this project.

2.1 MCTS

MCTS is a search algorithm that is used by the AlphaZero algorithm to choose the best action for the agent. Firstly, there are hyperparameters to determine the amount of rollouts, that is, the number of simulations performed before choosing the next best action. In this modified MCTS for Sokoban, where simulations are not played out until the game is finished, there are also hyperparameters used to specify the maximum simulation depth and the state value heuristic to be used.

For each rollout, a sensible move from the current state is selected based on two factors: the move that can be made, and how high the Upper Confidence bounds applied to Trees (UCT) is. The latter selects a move that balances exploitation and exploration. Afterward, a simulation is made from this specific move. The simulation goes through several sensible moves until one of the following criteria is met:

- A solution is found.
- There are no more sensible moves for the current path.
- The maximum depth (hyperparameter) is reached.

Each move creates a new node to the search tree, which is a representation of a game state, and a heuristic is used to calculate a heuristic value of that particular state. The aim is to minimize the heuristic value, but since the heuristic value was chosen to be a negative value, the score has to be maximized. This is the best score for that rollout, which is then used to update the rest of the tree to determine which move should be done by the agent. This was done using two different approaches.

The first approach recursively updates the heuristic score of all parent nodes with the mean score of all child nodes, and then selects the node with the most visits. This approach works fairly well due to how UCT balances exploration and exploitation, as nodes with a good score will get exploited more. This approach has two main issues. First, the score is a mean of all the child nodes, which will not accurately represent how good the node is for when the model is later trained on this score. Secondly, in some instances the best node is not selected; suppose a node has one child node with a great heuristic value, but the remaining have terrible values, the average will still be poor. In such instances, the best move is unlikely to be used.

The second approach attempts to fix the updates of the approach above. It works by updating the parent nodes with the best heuristic value of the child nodes and then selecting the node with best value. This initially had problems since multiple nodes might have the same heuristic value, which could cause the agent to use multiple moves to reach a state that could be reached in fewer steps. A depth penalty existed that was supposed to handle this, but it proved not useful when the simulations simulated to a set depth each time. It was then expanded upon to take into account when the heuristic value was improved, which fixed the issue.

2.1.1 Cycle detection

Initially, unnecessary computations were causing slower rollout times. This slowdown was determined to be caused by cyclic paths, leading to already visited states being re-explored.

In order to address this issue, a class was implemented to prevent cyclical paths. The main component being a dictionary where box positions serve as keys, and the corresponding values store the states previously visited by the agent. The mechanism checks proposed moves to ensure they do not lead to previously visited states. Before executing a move, the current box positions are matched against the dictionary keys. If no matching key is found, the current position is added to the dictionary. Afterward, the agent's new position is added to the list of visited positions. If the new position already exists in the list, the move is invalidated and not executed.

Cycles were handled in two ways, depending on if the cycle was detected in a simulation or when executing a real move. A cycle in a simulation resulted in the simulation ending early. On the other hand, when a cycle was detected in a real move, the agent backtracked to the latest position where another move could be taken that would not result in a cycle. Figure 1 shows a scenario where the agent is stuck in a cycle and has to backtrack. The agent will follow the red arrows backwards and then make the move with the purple arrow, which results in a new state shown in Figure 2. The agent is shown by the green figure, boxes by brown squares, and goals by blue flags. Red arrows are the agent's taken path, while the purple arrow is the move it will do after backtracking. Figure 2 has no red arrows because the box configuration has never been seen before, thus no positions have been stored before.

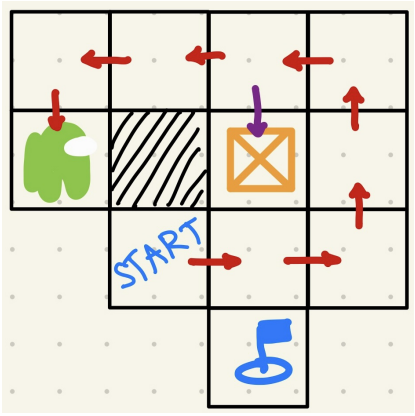

Figure 1: Scenario where the agent is encountering a cycle.

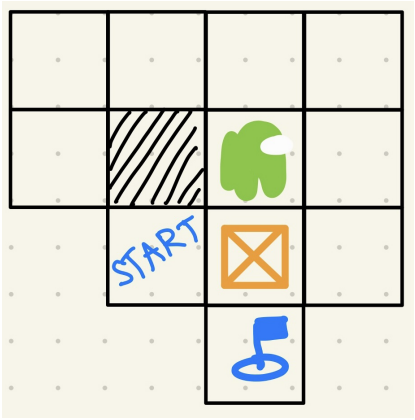

Figure 2: New state after backtracking.

2.1.2 Node structure optimization

While cycle detection sped up the MCTS significantly, further analysis identifies that the operations involving the tree nodes were still slow, which slowed down the entire MCTS process. To address this, the node (search tree) structure tried to be optimized by using a dictionary-based structure instead of using the implemented object-based structure.

In this approach, the nodes in the tree were represented as dictionary entries, with keys as the current board and values representing relevant data, such as the possible actions, the number of visits, accumulated score, and child nodes. A new class, Hashable, was introduced to store the current board as a key, as the previously used NumPy array was non-hashable. The use of dictionaries as nodes was intended to enable constant-time complexity for tree operations, such as key lookup and updating a node. This was expected to significantly accelerate operations like backpropagation and node expansion, enhancing the overall performance of MCTS.

However, this approach presented several challenges. It boosted memory usage and increased the complexity of the implementation. Additionally, difficulties were encountered in implementing the parent-child hierarchy with dictionaries, which prevented correct execution of the program. As a result, no performance gain was achieved, and the approach failed in providing any benefit. Given the issue with executing the implementation and the marginal performance improvements from the other optimizations, this dictionary-based approach was abandoned in favor of the simpler and more efficient object-based solution.

2.1.3 Parallelization with Multithreading

To further optimize MCTS, there were attempts to introduce multithreading to parallelize the rollouts. By utilizing multithreading, MCTS would be able to explore multiple branches of the tree concurrently and distribute the computational workload across different threads. However, there were some challenges with deep copying the complex environment, as it was not possible to efficiently copy the environment for each thread. To overcome this, an effort was made to optimize the node structure itself, described in 2.1.2. Since that approach also did not provide the desired results, the idea of multithreading was finally dropped.

2.2 Heuristics

The following heuristics were introduced:

- Number of boxes currently not in goal states.
- The minimum Manhattan distance between boxes and goal states.
- The minimum Manhattan distance between the agent and boxes.
- Penalizing boxes that are not able to move freely.
- An AlphaZero inspired trained model.

These heuristics could be combined with each other and also individually weighted with specified values.

2.3 Softlock detection

Because boxes can only be pushed and not pulled, boxes can be pushed into locations from where they cannot be recovered. The boxes become deadlocked, and the board cannot be solved. In these states, the agent can still make moves, but the board state and all following board states are unwinnable. We thus end up in a so-called softlocked state, where it is necessary to reset the board or backtrack to a previous non-softlocked state. These scenarios not only result in unnecessary computation of long branches of softlocked states in the game tree but also render boards unsolvable without an almost exhaustive search of the state space. Thus, the prevention of two types of softlocking scenarios was implemented as described below. However, it should be noted that there exist a lot more types that are not detected.

2.3.1 Avoid box stuck in corner

A clear softlocked situation is when a box is pushed and then stuck in a corner formed by two walls neighboring diagonally, as shown in Figure 3. To prevent this, push actions that result in a box in a corner are filtered out when the available moves of the agent are retrieved. These push actions are detected by iteratively checking the surrounding spaces of the box's new position for a corner.

2.3.2 Avoid deadlock between boxes

The other type of softlocking that is prevented is when two boxes become deadlocked, and cannot be moved, if they are horizontally or vertically aligned and there is a wall in front of both boxes, as shown in Figure 4. Similar to the corner prevention, this is detected and prevented by iteratively checking the surrounding spaces of a box's new position during push actions. Push actions with these types of wall and box arrangements are excluded from the available moves.

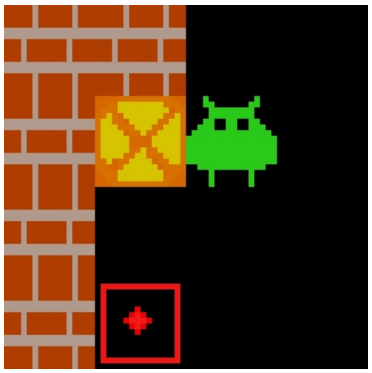

Figure 3: An example of a softlocked state caused by a box stuck in a corner.

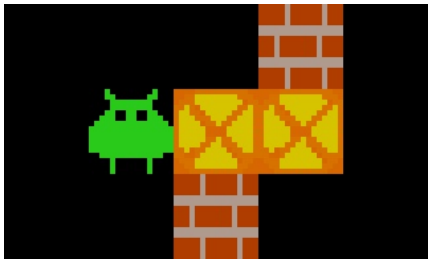

Figure 4: An example of a softlocked state caused by boxes deadlocking each other when aligned with walls.

2.4 AlphaZero

The main part of the project was to implement an adaption of the AlphaZero algorithm, which was adapted for Sokoban. AlphaZero was developed by

DeepMind and was first described in the paper *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm* by (Silver et al., 2017). However, the exact algorithm was not made publicly available, instead general descriptions of AlphaZero were published while precise implementation details were left out. Therefore, the group had to create an adaption of AlphaZero based on publicly available information, primarily from the article *AlphaZero Explained* (Cheerla, 2018).

From the group's knowledge, AlphaZero works and serves as a layer on top of MCTS. This is to improve decision-making by learning a heuristic of itself, with the goal to surpassing a given initial heuristic. In the group's implementation, AlphaZero was trained on MCTS heuristic scores to evaluate states and predict a value head. The value function acted as a dynamic heuristic that improved over time. The architecture for the neural network that was implemented by the group is shown in Figure 5.

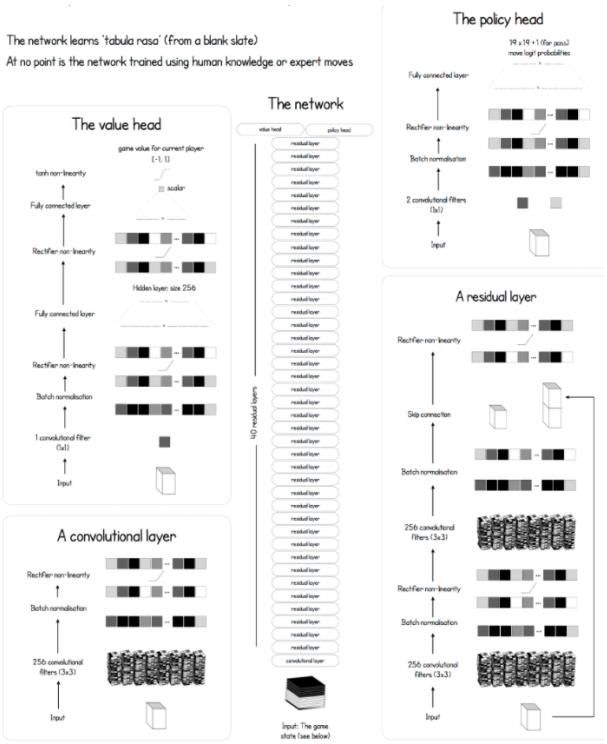

Figure 5: AlphaZero architecture. Source from (Cheerla, 2018)

Some notable layers in the neural network are a Rectified Linear Unit layer, batch normalization layer, convolutional filters, and skip connections. These layers are described as:

- Residual Layer. A residual layer in neural networks uses skip connections to add the input of a layer directly to its output to prevent a vanishing gradient. This design allows deep networks to train more effectively.
- Rectified Linear Unit (ReLU). A popular activation function that made deep neural networks possible by mitigating the vanishing gradient problem.
- Batch normalization. Normalizes input to a mean of 0 and a variance of 1. This is to stabilize input between layers and allows for faster converge and reduce reliance on weight initialization.
- Convolutional filter. Primarily used for feature extraction such as edge detection of matrix like data. Commonly used for feature extraction of images using Convolutional Neural Networks.
- Skip connection. For a skip connection, the input before specified layers is combined with the output of said layers. This helps with the vanishing gradient problem and feature propagation.

Through backpropagation, MCTS updates the neural network with scores from each rollout. Two different methods for training the model were tested. These methods are described in Section 2.5. After training for the network on a specific board, a model is saved, which can then be run as a heuristic for a new run of the same board.

2.5 Using the model

During the course of the project, two different methods were used to train the model. Initially, the model was trained using the rollout scores obtained during MCTS as target values. As input, the model was given the board state, which had been passed through an embedding layer.

Due to poor initial results with the aforementioned approach, an alternative method was proposed. Instead of the model being trained to predict scores, it was instead taught to predict how many steps were required to reach a certain heuristic value. The model was still given the board state as input, but also the heuristic value of one of the descendants of that state with a lower heuristic value than the current state's. The goal of the model was to predict how many steps were required to reach the descendant's heuristic value. As an example of this method, consider the tree given in Figure 6.

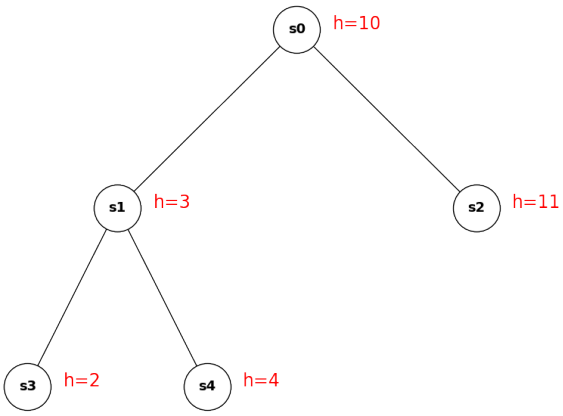

Figure 6: A rudimentary tree. The nodes represent board states and their names are written within them. Next to each node, the heuristic value for the state is given.

For the root node, named s0, in the tree above, the following training samples are obtained:

- (s0, 10) -> 0 (from s0)
- (s0, 3) -> 1 (from s1)
- (s0, 2) -> 2 (from s3)
- (s0, 4) -> 2 (from s4)

Note especially that s2 does not yield a training sample since its heuristic value is higher than s0's. This way of generating samples is then performed for all nodes in the tree. When the trained model was used as a heuristic, board states were evaluated based on how many steps the network predicted to be required to reach a heuristic value of zero.

2.6 Gym environment

The project was built on the open-source gym environment from OpenAI (Brockman et al., 2016). On top of the environment, the wrapper gym-sokoban was used to get the logics of a Sokoban game (Schrader, 2018). Initially, there was an additional wrapper, MCTS-Sokoban, which was used to perform actions using MCTS (bkuschner, 2020). This was later removed due to compatibility issues and critical bugs, such as the board flickering in certain unknown circumstances, making it totally broken. All bugs encountered during the project caused by the environment can be seen in Appendix A.1.

In order to get a graphical visualization of the agent, the library PyGame (Community, 2000) was added to the wrapper. PyGame served as a representation of the Sokoban board and was updated after an action was performed by the agent.

2.7 Evaluation

To evaluate the performance of our solver and implementation, it was compared against the statistics of selected state-of-the-art solvers discussed in Section 2.7.1. Due to technical reasons, it was not possible to run these state-of-art solvers on the same boards. Therefore, the available performance metrics and documented results for these solvers were used. The comparisons were targeted towards metrics such as ability to solve different difficulty boards, algorithms used, and the time taken to solve these boards. This evaluation highlights the strength and limitations of the approach compared to existing solutions.

2.7.1 Considered solvers

Two state-of-the-art solvers are selected to compare with our model, providing benchmarks for performance and highlighting its strengths and weaknesses. These solvers were chosen for their efficiency and recognition within the Sokoban research community, offering valuable insights into our model's capabilities.

- **Sokolution**: a fast, memory-efficient Sokoban solver in C++ by Florent Diedler. Sokolution uses bitsets to minimize memory usage, handling levels containing at most 256 floor squares. It supports both optimal (BFS, A*, IDA) and non-optimal (Greedy, DFS) algorithms, with default bidirectional greedy search for fast solutions. Key features include advanced heuristics, dynamic deadlock detection with pre-calculated deadlock position, PI-Corral pruning and tunnel/goal area macros (Diedler, 2017).

- **Curry**: Curry is a curriculum learning-based Sokoban solver that approaches the problem by first solving easier subcases and progressively increasing the difficulty. It uses only the game's winning condition, all boxes on targets, and does not depend on any human-designed features, heuristics or deadlock detectors, differentiating it from traditional solvers. Curry solves subcases with fewer boxes incrementally, and uses the solved parts to help solve more complex levels. Although computationally much lighter than the rest, it is not as strong as Sokolution, but it can still solve many XSokoban levels, with a time limit of one hour (Shoham, 2021).

3 Results

This section describes the results of our investigations. The parameters used for training were set to:

- training time: 25 minutes
- learning rate: 0.001
- beta values: (0.9, 0.999)
- weight decay: 0.01
- embedding dimension: 16
- number of residual blocks: 10
- number of channels: 64
- maximum rollouts: 100
- maximum depth: 30
- maximum steps 120

No noticeable change was seen in the trained model on any board when adjusting parameters.

3.1 Small board

The small board, seen in Figure 7, could be solved by using MCTS with different heuristics. Two different MCTS implementations were used, one where if a solution was found in a simulation, the agent took that path, and the other where no such shortcut was used. Let us call the first method MCTS-short, and the other MCTS-long.

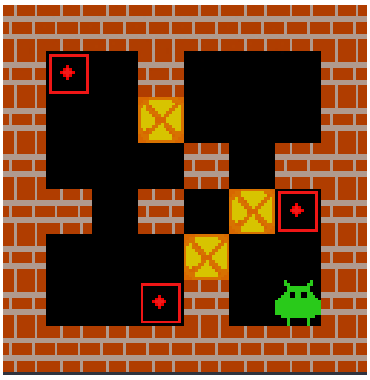

Figure 7: Initial state of the small board.

When using MCTS-short with distance to goal from boxes as heuristic, the board was solved in 2 to 10 seconds, depending on randomness. With the trained model, this took around 50 seconds. On the other hand, when using MCTS-long, the model solved the board in about 300 to 400 seconds.

Solving the board with the trained model and without MCTS gave more unpredictable results. The board was solved sometimes, but whether the trained model was able to do so varied greatly, and it was only possible with help of another heuristic. This was done by scaling the trained model heuristic by 0.8 and adding 0.2 from the distance to goal from boxes heuristic.

3.2 Medium board

Medium boards, such as the one shown in Figure 8, were not solvable with the trained model, nor any variants of MCTS. However, MCTS with distance to goal from boxes as heuristic was able to get 2 or 3 boxes to goals reliably, as can be seen in Figure 9. This differs from the trained model that only ever managed to move 1 or 2 boxes to goals. This can be seen in Figure 10, where at the end of 2 minutes of going back and forth, the agent pushes boxes further from any goal. This was also the case when combining the model heuristic with other heuristics, though it performed better by pushing boxes closer to individual goals, as seen in Figure 11.

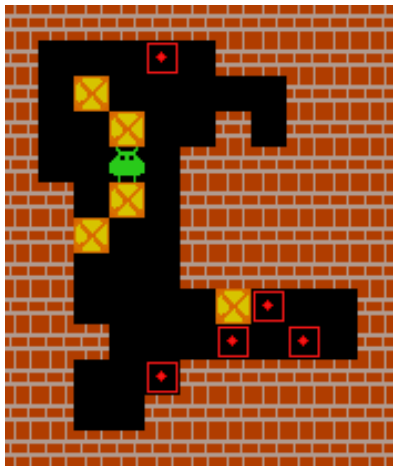

Figure 8: Initial state of the medium board.

3.3 Hard board

Larger and harder boards were also tried, such the board is shown in Figure 12. These boards required a very specific order to be solved, such as handling a box far away from the start position to have a solvable board. These boards were not solvable by any of the tried methods, such as MCTS variants and the trained model, with or without help. No box was ever pushed to any goal in these boards, as the agent pushed boxes into a one way corridor, as seen in Figure 13. The trained model performed the same, if not worse, to MCTS.

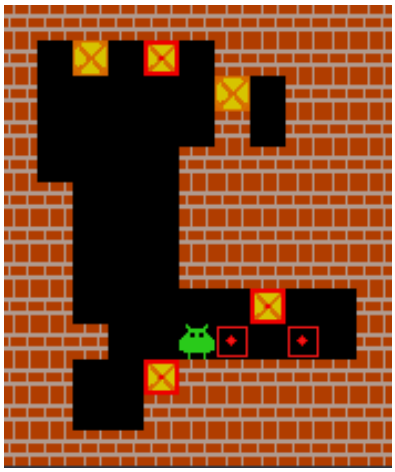

Figure 9: Best state of the medium board after 2 minutes of playing. Moves taken by MCTS with distance to goal from boxes as heuristic.

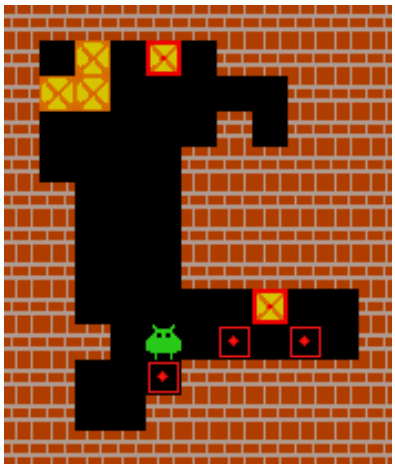

Figure 10: Best state of the medium board after 2 minutes of playing. Moves taken by the trained model heuristic.

4 Discussion

The results were consistent with the expectations of the group. As of now, there is no universal solution for solving Sokoban boards. The expectation of the group was that the idea of training a model to solve the board would fall in the same category; sometimes an optimal solution would be possible, but not always. However, the group was conceptually uncertain whether a model trained on patterns learned from MCTS could solve boards that MCTS itself could not. The idea of training a model was to get a general sense of how good a certain board state is, but we were not confident that this would lead to any actual improvement over regular MCTS. This was somewhat confirmed by the results because no board that was not solvable directly by MCTS, was solvable by the corresponding trained model.

As shown in the results, small boards that were

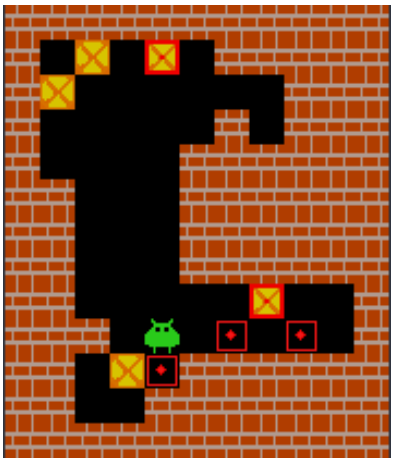

Figure 11: Best state of the medium board after 2 minutes of playing. Moves taken by the trained model heuristic combined with distance to goal from boxes heuristic.

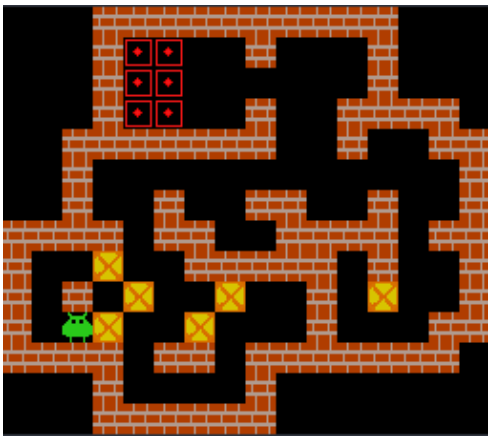

Figure 12: Initial state of the hard board.

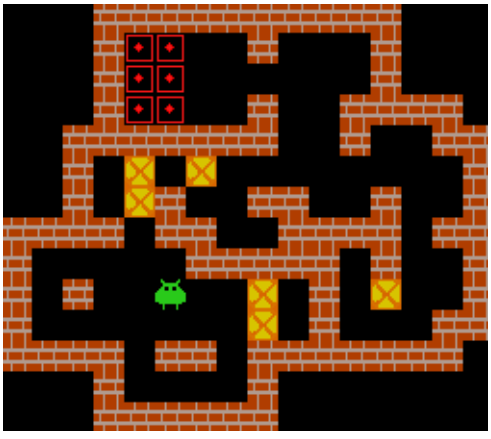

Figure 13: Best state of the hard board after 2 minutes of playing. Moves taken by MCTS with distance to goal from boxes as heuristic.

solvable directly by MCTS could be solved using the trained model, but any more complicated boards that were not solvable by MCTS directly were not solvable by the model. The machine learning approach of our project is a relatively unexplored area when it comes to solving Sokoban boards. Because of this, there are currently no papers that directly explore this type of Sokoban solver. Compared to current algorithms-based solutions, which can be seen in Table 1, our results are quite poor due to the inability to solve any kind of harder boards or outperform other solvers in any meaningful way. Therefore, the group believes that the approach of training a model is unreliable and provides no noticeable benefit compared to other methods and should not be considered for a state-of-the-art solver for Sokoban unless major improvements are made.

The group suspects that the reason for the poor performance is due to the nature of Sokoban itself. The order of how boxes need to be pushed is very important for solving most Sokoban boards, especially advanced boards. A single incorrect push can make the board unsolvable, even without the player's knowledge. Therefore, integrating an accurate softlock detection is almost as difficult as solving the game itself. Training on a softlocked board could potentially cause the model to learn suboptimal or undesirable patterns, which is what the group suspects is the main reason for the poor results seen in Section 3. Another potential pitfall is that the cycle detection and prevention breaks off simulations early, sometime 5 moves in. This could be the cause of invariant results, even though parameters were changed. However, removing cycle detection and prevention would cause a drastic decrease in performance, slowing down training by a big factor.

4.1 State-of-the-art solver vs. Alphazero model

Comparison of our model against state-of-art sokoban solvers, Sokolution and Curry, highlights distinct approaches and various trade-offs. While our model follows an AlphaZero-inspired framework with MCTS, cycle- and softlock detection, it fails in generalization and handling of complex cases, especially medium and hard boards. Sokolution uses both optimal(BFS, A*, IDA) and non-optimal (Greedy, DFS) algorithms and many precomputed optimizations to achieve robust performance even on hard boards, solving easier level faster compared to other solvers. While curry is conceptually innovative, its curriculum learning approach makes it computationally inefficient, as solving a complex board required considerable runtime. Table 1 presents a concise comparison between our model and both state-of-art-solvers.

Table 1: State-of-the-art solver vs. Alphazero model

Aspect	**AlphaZero Model**	**Sokolution (Florent Diedler)**	**Curry (Yaron Shoham)**
Programming Language	Python	C++	C
Algorithms	AlphaZero with MCTS	BFS, A*, greedy algorithm	Curriculum learning
Optimization	Cycle and softlock detection	Precomputed deadlock and penalty positions	No deadlock or cycle detection implemented
Results	Solves easy boards but fails on medium/ hard boards	solves hard boards reliably	Solve easy levels faster, takes significant time on hard levels (30+ hours)

4.2 Improvements

This section describes potential future implementations that could be done to improve the results of the project.

4.2.1 MCTS

The MCTS algorithm could be improved by extending the current softlock detection solution. There are many unwinnable states in Sokoban that are hard to detect as a softlock for both a human player and an agent. Some common examples of this are boxes that are pushed together or a box getting pushed in to a long corridor with a dead end. Extending MCTS to handle such softlocks could reduce the amount of unwinnable, and therefore,

useless states being explored and trained on. In theory, this could improve the performance of the trained model.

Another possible improvement could be to implement more heuristics. A restriction of the current solution is that the heuristics are too simple for Sokoban which, in turn, makes any difficult board more problematic to solve. More advanced heuristics that are more dedicated to the game of Sokoban could be more suitable and improve the final results. An example would be a heuristic that deeply accounts for softlocks and player positioning, which in turn makes the model learn what moves not to make.

4.2.2 Training

Something that was unexplored during the project was the idea of different approaches to training the model. The current method ran a Sokoban board as is, in hope of eventually finding a solution or being able to train a model that could learn important features of the board. Instead, a different approach that might yield better results could be training a model to predict the amount of steps required to move between two different board states, instead of steps to a heuristic value. Such a model could be trained by pairs of parent and child states encountered during MCTS. Then, by passing a solved version of the board along with the current board state to such a model, one would obtain a heuristic estimate of how far away from a solution a board configuration is.

Training would also benefit from having the node structure optimization mentioned in Section 2.1.2, as well as having multithreading mentioned in Section 2.1.3. These changes would make training faster, which in turn means cycle detection and prevention could be removed, eliminating an early stop in simulations, without having a big impact on performance as it is now.

5 Conclusion

In this project, we investigated whether an AlphaZero-inspired model could be applied to Sokoban, a PSPACE-complete puzzle game. Many different implementations were tried, and none gave better results than state-of-the-art solvers. While small boards were solvable with our implementation of MCTS, the trained model was unable to the same.

Some of the key findings are:

- MCTS could solve small boards, but struggled with medium and hard boards.
- The trained model performed inconsistently and often pushed boxes into softlocks, performing worse than MCTS
- No boards not solvable by MCTS were solvable by the model.

Due to the complexity of Sokoban, a lot of challenges arose. One wrong move can create softlocks that renders the board unsolvable. The efficiency of MCTS was also drastically reduced due to simple heuristics and premature cycle preventions.

For future improvements, an expanded softlock detection and more sophisticated heuristics could be beneficiary. Additionally, optimizations in parallelization and deeper simulations may also enhance performance.

References

Sagar Batel bkuschner. 2020. Mcts-sokoban. https://github.com/bkuschner/MCTS-Sokoban.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym.

Nikhil Cheerla. 2018. Alphazero explained. [Online; accessed 14-November-2024].

Pygame Community. 2000. Pygame - python game development library. https://www.pygame.org. Version 2.1.3, accessed on November 19, 2024.

Florent Diedler. 2017. Sokolution solver. Accessed: 2025-01-06.

Max-Philipp B. Schrader. 2018. gym-sokoban. https://github.com/mpSchrader/gym-sokoban.

Yaron Shoham. 2021. Curry. Accessed: 2025-01-06.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

A Appendix

A.1 Bugs

- Sokoban gym environment called the wrong step function.
- Sokoban gym environment had wrong library versions that were too new and unusable.

- Sokoban gym environment had no way to revert states, which meant a new environment needed to be made with Sokoban environment as parent.
- MCTS Sokoban gym environment changed the board and created blocks when the agent moved sometimes. Our testing concluded that it happened when the agent moved onto a goal marker.