Solving Sokoban with Reinforcement Learning

Omar Tounsi, Joseph Lin, Jihyo Park

Purdue University
otounsi@purdue.edu
lin1223@purdue.edu
park1384@purdue.edu

Abstract

This project explores the application of reinforcement learning (RL) algorithms to the Sokoban puzzle game and evaluates their performance against traditional search-based algorithms, such as A* and breadth-first search (BFS). We implemented several RL algorithms, including Monte Carlo, Temporal Difference (TD) learning, REINFORCE, and Actor-Critic. Our results show that while some RL algorithms demonstrate potential in solving Sokoban puzzles, A* and BFS proved superior, consistently solving puzzles that RL methods could not. Moreover, even for puzzles solvable by both RL and non-RL approaches, A* and BFS outperformed RL algorithms in terms of speed and memory use. The code for our implementation is publicly available on GitHub (Tounsi, Lin, and Park 2024).

Introduction

Sokoban is an intriguing classic puzzle game due to its simple rules yet significant computational complexity. The player's task is to push boxes to designated goal cells within a confined two-dimensional grid. Despite its straightforward mechanics, solving Sokoban puzzles is computationally intensive; the problem is proven to be NP-hard and PSPACE-complete (Dor and Zwick 1999), indicating that the difficulty escalates rapidly with the size and arrangement of obstacles and boxes. This complexity makes Sokoban an ideal testbed for evaluating various problem-solving techniques. Traditional approaches to solving Sokoban have predominantly utilized non-reinforcement learning (non-RL) methods, including classical search algorithms and heuristic-driven strategies. For instance, the Rolling Stone solver (Junghanns and Schaeffer 2001) combines search-based methods with domain-specific heuristics to efficiently solve puzzles. In recent years, RL has emerged as a promising avenue for tackling complex decision-making problems, including games like Sokoban. RL algorithms learn policies through environmental interactions, receiving feedback in the form of rewards, and gradually improving performance. Notably, the forward-backward reinforcement learning approach (Shoham and Elidan 2021) has demonstrated efficacy in solving Sokoban by training a backward-looking agent.

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

This paper aims to compare RL methods with non-RL techniques for solving Sokoban puzzles. By evaluating various methods across puzzles of differing difficulty levels, we seek to elucidate the strengths and limitations of each approach.

Background

Game Rules and Mechanics

Sokoban is a puzzle game set in a grid-like warehouse environment. The objective of the game is to position all boxes within the warehouse onto designated storage locations (i.e. box goals), achieving a solved state. Each puzzle presents a unique arrangement of walls, boxes, box goals, and the player's starting position, requiring strategic movement and careful planning to complete. The warehouse is depicted as a grid where each square represents either a floor section or a wall. Walls act as static obstacles, while floor squares provide the pathways for movement. Some floor squares are marked as storage locations, indicating the target destinations for the boxes. The player can move horizontally or vertically one square at a time, provided the destination square is an empty floor section. The player cannot pass through walls or occupy the same square as a box. To interact with boxes, the player must walk up to one and push it to the square immediately beyond. This action is subject to three critical constraints:

1. The square beyond the box must be an empty floor square or a storage location.
2. A box cannot be pushed into a wall, another box, or any other obstacle.
3. Boxes cannot be pulled.

Each puzzle ensures that the number of boxes matches the number of storage locations, and the challenge lies in pushing all the boxes onto these storage locations in the least number of steps. The game is solved only when every storage location is occupied by a box. The simplicity of Sokoban's rules is deceptive, as the game poses significant challenges. Improper moves can easily lead to unsolvable states, requiring players to anticipate the outcomes of their actions. These constraints make Sokoban not only a test of spatial reasoning but also an intricate exercise in problem-solving.

Sokoban Markov Decision Process

To model Sokoban as a Markov Decision Process (MDP), we formalize its components—states, actions, transition dynamics, rewards, and termination criteria—based on the game mechanics. To facilitate experimentation, we implemented a custom Sokoban environment using the `Gymnasium` framework. This section describes how the game is structured as an MDP and provides the theoretical foundation necessary for RL algorithms applied to Sokoban.

State space. A state in Sokoban is defined by the agent's position and the positions of all boxes. To encode this state, we use a flattened integer representation of the entire grid, with each cell represented by one of the following values:

- '0': Empty space.
- '1': Wall.
- '2': Goal.
- '3': Box.
- '4': Box on a goal.
- '5': Agent.
- '6': Agent on a goal.

The environment's state space is defined using the `Box` observation space in `Gymnasium`, with bounds `low=0` and `high=6`, and a shape of (height $\times$ width), where `height` and `width` correspond to the dimensions of the grid. The flattening of the grid ensures compatibility with function approximators, such as neural networks, which operate on vector inputs.

Including the entire grid in the state representation, rather than solely the positions of the agent and boxes, is important for modeling Sokoban. For example, to detect whether a box has been pushed into an unwinnable position, it is necessary to check whether it is surrounded by two or more obstacles. This assessment requires knowledge of the entire grid configuration. Similarly, checking whether a box has been pushed into a goal or whether the game has been won both rely on knowledge of the global grid.

This representation offers several advantages:

- It encapsulates all necessary information about the environment, enabling accurate reward computation and termination checks.
- Its compatibility with function approximators facilitates the application of RL algorithms.

However, it also presents challenges:

- The size of the state space grows exponentially with the grid size and the number of boxes, leading to computational challenges for large puzzles.
- Flattening the grid discards spatial locality, potentially complicating learning for algorithms that could otherwise leverage the grid's structure.

Overall, this representation attempts to strike a balance between expressiveness and computational feasibility.

Action space. The agent can perform four possible actions: Up, Left, Down, Right. The action space is modeled using the `Discrete` space provided by the `Gymnasium` framework, which represents these four actions as integers.

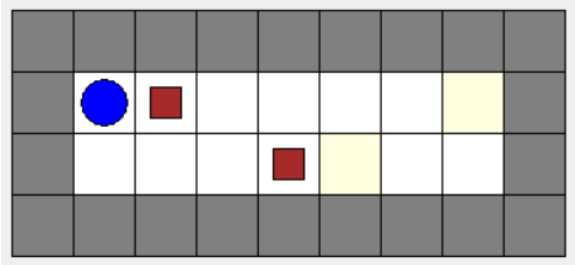

Figure 1: This 4x9 puzzle is encoded as an array of 36 integers.

Transition dynamics. The environment's transition dynamics determine how the agent's actions modify the state. These dynamics include:

- Agent Movement: The agent moves in the intended direction unless obstructed by a wall or box.
- Box Movement: If the agent pushes a box, the box moves one step in the same direction, provided the destination cell is free (either empty or a goal).
- Constraints: If an action is invalid (e.g., moving into a wall or pushing a box into another box or wall), the state remains unchanged.

The transition function is deterministic and fully observable, ensuring that the next state is uniquely determined by the current state and the chosen action.

Reward function. The reward function is designed to guide the agent toward solving the puzzle efficiently while penalizing invalid or counterproductive actions. Rewards are assigned as follows:

- +100: Winning the game, by pushing the last unplaced box onto a goal. This serves as the ultimate objective and ensures that the agent prioritizes completing the level.
- +1: Pushing a box onto a goal (without winning). This is a smaller intermediate reward to encourage incremental progress towards solving the puzzle.
- -0.01: Any valid move that does not involve pushing a box. This discourages unnecessary movements and encourages the agent to find shorter solutions.
- -1: Attempting an invalid move, such as moving into a wall. This is to dissuade the agent from repetitive behaviors that do not lead to progress.
- -10: Pushing a box into an irrecoverable position, such as a corner surrounded by walls. This discourages actions that make the puzzle unsolvable.

This reward structure was designed with the following considerations:

- Progressive Rewards: The structure ensures incremental reinforcement for making progress (e.g., pushing boxes onto goals) while penalizing states that lead to failure (e.g., irrecoverable box positions).
- Balancing Exploration and Exploitation: The step penalty prevents the agent from wandering aimlessly,

while the small positive rewards for intermediate progress ensure it continues exploring toward the goal.

- Encouraging Problem Solving: By assigning significant rewards to the ultimate goal, the system incentivizes prioritization of actions that contribute to the long-term objective.

The reward system has the following pros and cons:

- Goal-Oriented Learning. Rewards are aligned with the puzzle's objectives, guiding the agent effectively.
- Efficiency Incentive. The step penalty encourages solutions with fewer moves.
- Prevention of Failures. The box-stuck penalty ensures that the agent learns to avoid irreversible mistakes.
- Sparse Rewards. The agent may struggle to discover paths leading to significant rewards.
- Hyperparameter Sensitivity. The relative magnitudes of rewards and penalties must be carefully tuned to balance exploration and exploitation effectively.

Termination criteria. The environment terminates in one of two cases:

- Success: All boxes are placed on their respective goals. This corresponds to the agent solving the puzzle and receiving the maximum reward.
- Failure: The agent gets a boxes stuck, making it impossible to complete the puzzle. In this case, the game terminates, and the agent incurs the maximum penalty.

Discount factor γ. The discount factor, denoted as γ, is a critical parameter in determining the importance of future rewards relative to immediate rewards. In our Sokoban environment, γ is set to 0.99. Sokoban puzzles often involve intricate sequences of moves to achieve sub-goals, such as maneuvering boxes toward goals without creating deadlocks. This high discount factor ensures the agent considers the rewards of placing multiple boxes in succession, rather than focusing on short-term step penalties. A significant tradeoff is slower convergence as the agent must evaluate more distant rewards.

Initial state distribution The initial state distribution in the Sokoban environment is determined by the puzzle being solved. Each puzzle has one and only one starting state, defined by the initial positions of the agent, boxes, walls, and goals on the grid.

Size of the State Space

The state space of a Sokoban puzzle represents all possible configurations of the agent and box positions on the grid. Understanding the size of the state space is critical to appreciating the computational complexity of solving Sokoban puzzles. For a grid of size $n \times n$ with k boxes, the size of the state space is determined by the number of ways to position the agent and the k boxes on the grid.

- **Choosing Positions:** To place the $k+1$ entities (the agent and k identical boxes) on the n^2 grid cells, we first choose $k+1$ distinct positions. This is given by the binomial coefficient:

$$\binom{n^2}{k+1} = \frac{n^2!}{(k+1)!(n^2-k-1)!}.$$

- **Assigning the Agent:** Among the chosen $k+1$ positions, one is assigned to the agent, and the remaining k positions are occupied by the identical boxes. There are exactly $k+1$ ways to assign the agent to one of these positions.
- **Total Number of States:** Combining the above, the total number of valid states in the Sokoban puzzle is:

$$\text{Total states} = \binom{n^2}{k+1} \times (k+1).$$

- **Asymptotic Growth:** For large grids ($n \to \infty$), the binomial coefficient $\binom{n^2}{k+1}$ grows approximately as $(n^2)^{k+1}$. Including the linear factor $k+1$, the total number of states scales asymptotically as:

$$O(n^{2(k+1)}).$$

- **Implications:** The exponential growth of the state space highlights the inherent computational difficulty of Sokoban. Even for relatively small grid sizes and modest numbers of boxes, the vast state space poses significant challenges for exact or approximate solutions.

Challenges of the MDP Representation

Our Sokoban MDP presents some challenges that can significantly impact the performance and efficiency of RL algorithms. Below, we discuss the key difficulties associated with our MDP representation.

- **Reward Sparsity:** The reward function is inherently sparse, as the agent only receives positive rewards for solving the puzzle ($+100$) or placing a box on a goal ($+1$). Most actions yield negative rewards (e.g., -0.01 for each step), providing limited feedback for learning. This sparsity makes it challenging for agents to learn meaningful behaviors, especially in the absence of intermediate rewards that guide exploration toward the goal.
- **High Dimensional State Space:** As we have seen, the state space size grows exponentially with the number of boxes and the grid size. For a grid with $n \times n$ cells and k boxes, the number of possible states is approximately $O(n^{2(k+1)})$. This large state space makes it computationally expensive to explore and store all possible states and actions, requiring algorithms to generalize effectively across unseen states.
- **Exploration Challenges:** Sokoban puzzles often have long sequences of actions with minimal immediate rewards. Without sufficient exploration, agents can become stuck in suboptimal regions of the state space. Techniques like epsilon-greedy exploration help, but they cannot fully mitigate the difficulty of finding efficient trajectories in high-dimensional spaces.

Related Work

Single-Agent Policy Tree Search With Guarantees

Single-Agent Policy Tree Search With Guarantees (Orseau et al. 2018) presents two innovative tree search algorithms designed for single-agent decision-making problems, particularly Sokoban puzzles. The key contributions of this paper include a best-first enumeration algorithm and a sampling-based tree search algorithm. The best-first enumeration algorithm employs a cost function to guide the search and provides a theoretical upper bound on the number of nodes expanded before reaching a goal state, which is especially advantageous in sparse reward environments like Sokoban. The sampling-based tree search algorithm estimates the expected number of nodes expanded before achieving a set of goal states, making it suitable for scenarios involving multiple solution paths. Both algorithms utilize a policy-guided approach, where a neural network trained via Asynchronous Advantage Actor-Critic (A3C) provides action probabilities that guide the search process. Using this neural policy, the authors effectively combine heuristic planning with RL.

The authors benchmark their methods on 1,000 computer-generated Sokoban levels. The results demonstrate that the proposed algorithms are competitive with state-of-the-art domain-independent planners that rely on heuristic search. The policy-guided algorithms efficiently navigate large state spaces and are adaptable to the complexities of Sokoban, showcasing their potential for solving similar single-agent search problems. The theoretical guarantees provided for node expansion further strengthen the robustness and reliability of the algorithms in practical applications.

Although the paper makes a significant contribution to the field, there are areas for critique. One notable strength is the innovative methodology that combines RL and heuristic search, providing a novel approach to the combinatorial challenges of Sokoban. In addition, the paper offers detailed descriptions of the algorithms, including theoretical underpinnings and practical implementations, which add to its clarity and accessibility. The extensive benchmarking of Sokoban puzzles also underscores the practical utility of the proposed methods.

However, the reliance on pre-trained policies in the document introduces a potential limitation. The quality of the neural network trained through A3C directly affects the efficiency of the search, and poorly trained policies could degrade performance. Furthermore, the discussion of the generalization of these methods to other domains is limited, leaving open questions about their applicability beyond Sokoban. Computational overhead is another potential drawback, as the integration of policy-guided search with tree expansion may become demanding for larger state spaces or more complex environments. Finally, the paper does not explore hybrid approaches, such as integrating reward shaping or adaptive heuristics, which could further enhance search efficiency.

In conclusion, the paper represents a strong contribution to single-agent search problems, particularly in solving complex puzzles like Sokoban. Its innovative combination of policy-guided search with theoretical guarantees enhances the robustness and efficiency of tree search methods. However, future work could focus on addressing computational overhead, improving generalization to other domains, and exploring hybrid approaches to further optimize the methodology. Despite these limitations, this work marks a significant advance in the intersection of RL and heuristic search methodologies.

Sokoban: Enhancing General Single-Agent Search Methods

This paper (Junghanns and Schaeffer 2001) is a seminal work that thoroughly explores the challenges of solving Sokoban puzzles through non-reinforcement learning approaches. In their study, the authors focus on traditional search methods such as BFS, DFS, A*, and iterative deepening A* (IDA*), demonstrating how these general problem solving algorithms can be adapted and extended to handle the severe state-space explosion characteristic of Sokoban. The complexity of the puzzle ensures that brute-force search is expensive, prompting the authors to seek more effective ways of nagivating the solution space.

A key contribution of their work is the use of domain-specific heuristics and pruning techniques. The authors highlight the importance of recognizing "deadlock states". Deadlock states are configurations where a box is pushed into an irrecoverable position, such as a corner that is not a goal. They made sure that the search can immediately discard these branches to minimize the search time. They also draw upon pattern databases, which precompute the cost of placing subsets of boxes onto their corresponding goals. Using these databases during the search provides strong lower bounds on the number of moves needed to solve the puzzle, guiding the algorithm away from states that cannot lead to efficient solutions. The authors showed that combining general search methods with engineered heuristics significantly increases performance over basic BFS/DFS methods, allowing them to solve moderately complex Sokoban puzzles within reasonable computational limits.

However, the authors emphasize that even the best-informed heuristic search methods still face scaling issues as the number of boxes and the size of the grid increase. Since the puzzle's state space grows exponentially, it requires even more sophisticated heuristics, more aggressive pruning, and greater computational resources. This observation sets the stage for reinforcement learning, which could learn representations and policies that mimic or even surpass engineered heuristics without needing domain engineering.

Therefore, this paper essentially serves as an anchor point for Sokoban research and a benchmark for comparing future techniques and algorithms. Their methodologies and findings showed the strengths and weaknesses of non-RL approaches. By demonstrating how these search algorithms can be adapted and enhanced for a domain as challenging as Sokoban, they provide insight into deeper consideration of how newer methods like RL could build on the results gained through traditional methods.

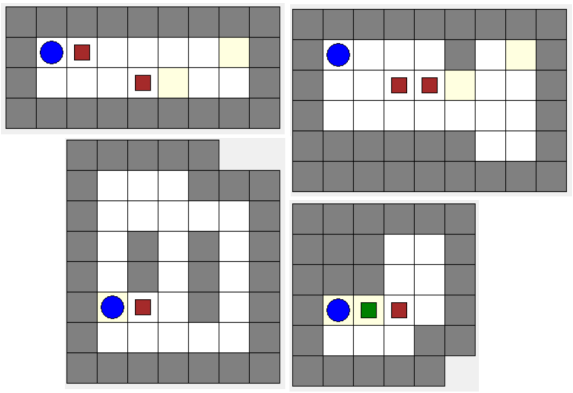

Figure 2: Levels 1-4 (clockwise). Easy puzzles.

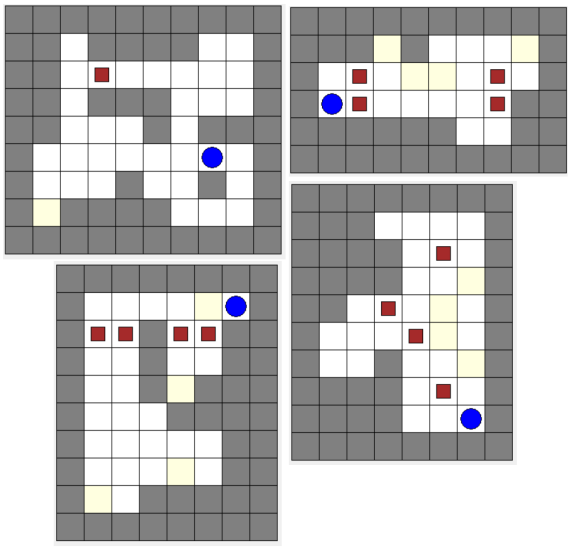

Figure 3: Levels 5-8 (clockwise). Hard puzzles.

Methodology

We implemented four RL algorithms—Monte Carlo Policy Optimization, TD Learning, REINFORCE, and Actor-Critic—to solve Sokoban puzzles. To evaluate the efficacy of these RL approaches, we compared their performance metrics with those of traditional search-based algorithms, including BFS, A* with the Manhattan heuristic, and A* with the Dijkstra heuristic. The evaluation was conducted on a set of eight Sokoban puzzles of varying difficulty. Key metrics analyzed include the time taken to solve each puzzle, memory usage during computation, and the number of actions required to arrive at a solution. In this section, we discuss our algorithm implementations.

Monte Carlo Policy Optimization

The Monte Carlo (MC) policy optimization algorithm estimates state-action values $Q(s, a)$ and refines a policy through iterative sampling of full episodes from the environment. Below, we detail the specific features and hyperparameters used in this implementation.

- **Every-Visit Monte Carlo:** Our implementation uses an every-visit approach to update the action-value function $Q(s, a)$. For each state-action pair encountered during an episode, the return is calculated and used to update $Q(s, a)$, averaged over all visits.
- **Epsilon-Greedy Exploration:** The agent employs an epsilon-greedy exploration strategy, with a decaying exploration rate (ϵ). Initially, $\epsilon = 0.9$, and it decays by a factor of $\epsilon_{\text{decay}} = 0.995$ after each step, down to a minimum value of $\epsilon_{\text{min}} = 0.1$. This encourages exploration in the early stages and exploitation of the learned policy as training progresses.
- **Loop Prevention:** During each episode, the algorithm maintains a set of visited states to detect and penalize loops. If a state is revisited, the episode is terminated, and a significant negative reward (-10) is assigned to discourage such trajectories.
- **Policy Updates:** After each episode, the policy is updated based on the $Q(s, a)$ values. For each state, the action with the highest $Q(s, a)$ value becomes the new action for that state in the policy:

$$\pi(s) \leftarrow \arg\max_a Q(s, a)$$

- **Termination and Convergence:** The algorithm halts when the policy does not change for a specified number of consecutive episodes (1000) or when the maximum number of episodes (100000) is reached.

Temporal Difference Learning

The TD Learning algorithm iteratively updates the state-action value function $Q(s, a)$ based on sampled state transitions, without requiring complete trajectories. Below, we outline the specific features and hyperparameters of the implementation.

- **Online Updates with TD(0):** Our implementation uses TD(0), where the value of $Q(s, a)$ is updated immediately after each state-action transition. The update rule is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$

  Here, α is the learning rate (which we set to 0.01), r is the immediate reward, γ is the discount factor, and s' is the next state.
- **Epsilon-Greedy Exploration:** Same as Monte Carlo.
- **Loop Prevention:** Same as Monte Carlo.
- **Policy updates:** Same as Monte Carlo.
- **Termination and Convergence:** Same as Monte Carlo.

REINFORCE

The REINFORCE algorithm directly optimizes the policy by adjusting its parameters in the direction that maximizes the expected cumulative reward. Below, we outline the specific features and hyperparameters used in our implementation.

- **Neural Network Policy Representation:** The policy is represented as a neural network, referred to as the Policy Network. The network consists of:
  - An input layer with dimensions matching the flattened observation space of Sokoban (n^2).
  - A hidden layer with 128 neurons and ReLU activation.
  - An output layer with dimensions matching the discrete action space (4), followed by a Softmax activation to produce a probability distribution over actions.
- **Trajectory Collection:** In each episode, the agent generates a trajectory by interacting with the environment:
  - At each timestep, the policy network outputs probabilities for the four actions.
  - The action is sampled from the resulting categorical distribution.
  - Rewards and log probabilities of the chosen actions are recorded for later optimization.
- **Policy Gradient Update:** After completing an episode:
  - The returns G_t for each timestep are computed using:

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$$

    where γ is the discount factor.
  - The returns are normalized to improve training stability.
  - The policy network is updated to maximize the log-probability of actions weighted by their returns:

$$\nabla J(\theta) \approx \frac{1}{T} \sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) G_t$$

    This is implemented by minimizing the loss:

$$L = -\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) G_t$$

- **Learning Rate (α):** $\alpha = 0.001$, controlling the speed of policy updates.
- **Loop Prevention:** A mechanism penalizes revisiting states within an episode, discouraging repetitive actions and ensuring efficient exploration.
- **Termination and Convergence:** The algorithm terminates once the reward exceeds a specified threshold (80) or when the maximum number of episodes (1000) is reached.

Actor-Critic

The Actor-Critic algorithm combines policy optimization with value function approximation, allowing for more stable updates by leveraging both a policy network (the actor) and a value network (the critic). Below, we outline the specific features and adaptations used in our implementation.

- **Neural Network Representation:** The algorithm uses two neural networks:
  - **Policy Network (Actor):** This network outputs a probability distribution over the discrete action space (4). It consists of:
    * An input layer with dimensions matching the flattened observation space of Sokoban (n^2).
    * A hidden layer with 128 neurons and ReLU activation.
    * An output layer with a Softmax activation to produce the action probabilities.
  - **Value Network (Critic):** This network predicts the value of the current state ($V(s)$) and consists of:
    * An input layer with dimensions matching the flattened observation space.
    * A hidden layer with 128 neurons and ReLU activation.
    * An output layer producing a single scalar value, representing $V(s)$.
- **Trajectory Collection:** In each episode, the agent generates a trajectory by interacting with the environment:
  - At each timestep, the policy network outputs probabilities for the actions.
  - An action is sampled from the resulting categorical distribution.
  - The value network predicts $V(s)$ for the current state.
  - Rewards, log probabilities of actions, and predicted values are recorded for later optimization.

  Loop prevention is implemented by penalizing revisits of previously visited states within the same episode, encouraging efficient exploration.
- **Policy and Value Updates:** After completing an episode:
  - The returns G_t for each timestep are computed:

$$G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$$

    The returns are normalized to improve stability:

$$G_t \leftarrow \frac{G_t - \mu}{\sigma + 1e - 5}$$

    where μ and σ are the mean and standard deviation of the returns.
  - The policy network is updated using the advantage function:

$$A_t = G_t - V(s_t)$$

    The actor loss is minimized:

$$L_{\text{actor}} = -\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) A_t$$

  - The value network is updated to minimize the mean squared error of the value predictions:

$$L_{\text{critic}} = \sum_{t=1}^{T} (G_t - V(s_t))^2$$

    Gradient descent is performed separately on both networks using the Adam optimizer.

- **Learning Rate** (α): $\alpha = 0.001$, for both actor and critic.
- **Loop Prevention:** Same as REINFORCE.
- **Termination and Convergence:** Same as REINFORCE.

Search Method Solvers

We use a publicly available repository (xbandrade 2023) as our reference for solving Sokoban with search algorithms. This repository implements the following search algorithms to solve Sokoban puzzles, each employing distinct strategies to navigate the puzzle state space:

BFS uses a queue to manage states to be explored. At each iteration, the algorithm dequeues a state and considers all possible moves. If a move leads to a new valid state that has not been visited, it is enqueued for further exploration. To prevent redundant computations, a set tracks all previously visited states. The algorithm checks each new state for puzzle completion. If a solution is found, the path and depth are returned. BFS ensures the shortest path is found but can be memory-intensive since it stores all states at the current depth level.

A* with Manhattan Heuristic starts by initializing a priority queue, where states are prioritized based on cost. The cost of a state is calculated as the sum of the cost incurred so far and an estimated cost to reach that state. The Manhattan heuristic estimates the remaining cost by summing the horizontal and vertical distances of all boxes from their respective goals. At each step, the algorithm dequeues the state with the lowest priority and evaluates all possible moves from the current player position. For each valid move, the algorithm calculates the new cost using the Manhattan heuristic and updates the priority queue. This approach efficiently explores states likely to lead to a solution while avoiding redundant computations. However, the Manhattan heuristic assumes linear paths to goals without obstacles, which is not always the case in Sokoban puzzles.

A* with Dijkstra's Heuristic uses a more exhaustive calculation for the remaining costs. Here, the estimated cost is computed using Dijkstra's algorithm to find the shortest possible paths to goals, considering potential obstacles. This heuristic provides a more precise estimation of the cost compared to Manhattan distance, albeit at the expense of additional computational overhead. Like the Manhattan-based implementation, the algorithm initializes a priority queue and evaluates states based on their combined actual and estimated costs. For each move, the algorithm updates the priority queue with the new state and its corresponding costs. The Dijkstra heuristic offers improved accuracy in estimating remaining costs, making the algorithm less likely to explore unproductive paths. However, its computational demands are higher, especially for puzzles with complex layouts or a large number of boxes.

Experimental Results

Non-RL Algorithms Results

The results in Tables 1-3 demonstrate the effectiveness of A* with heuristics compared to BFS. In terms of time to converge, A* with Manhattan heuristic consistently outperformed both BFS and A* with Dijkstra heuristic, solving puzzles significantly faster in all but one case. A* with Dijkstra heuristic also showed better time efficiency than BFS but lagged slightly behind the Manhattan heuristic. The enhanced efficiency of A* with heuristics underscores the value of incorporating domain-specific information into the search process. Memory usage further highlights the advantages of heuristic-guided search. Both variants of A* used considerably less memory than BFS, with the Manhattan heuristic achieving the best memory efficiency overall. The difference was particularly pronounced in more complex puzzles, where BFS required several magnitudes more memory than its heuristic-based counterparts. In terms of the number of actions required to solve puzzles, BFS has an edge over A*, since it explores the state space more and is thus able to find more optimal ways of solving the puzzle. Overall, the results confirm that heuristic-guided search methods, particularly A* with the Manhattan heuristic, are superior to BFS in solving Sokoban puzzles efficiently, but not necessarily optimally: they achieve faster convergence and lower memory consumption at the expense of not finding the best solution.

Puzzle	BFS	A* (Manhattan)	A* (Dijkstra)
1	0.00316	0.00589	0.00635
2	0.05551	0.00653	0.00876
3	0.00112	0.00694	0.00613
4	0.00105	0.00971	0.00679
5	0.07946	0.01361	0.04977
6	5.53024	0.02344	0.06463
7	0.33402	0.00787	0.01622
8	19.35767	0.01292	0.06613

Table 1: Time to converge to a solution (in seconds) for non-RL algorithms.

Puzzle	BFS	A* (Manhattan)	A* (Dijkstra)
1	0.03	0.02	0.03
2	0.98	0.02	0.01
3	0.02	0.01	0.01
4	0.01	0.01	0.01
5	0.12	0.09	0.06
6	3.86	0.17	0.09
7	3.03	0.02	0.02
8	23.56	0.08	0.09

Table 2: Memory usage (in MB) for non-RL algorithms.

RL Algorithms Results

The results in Tables 4-6 indicate that MC and TD emerged as the most effective RL algorithms. They were the only methods to solve a majority of the puzzles. Furthermore, both MC and TD consistently utilized the least memory. Their performance highlights the strength of simpler, tabular RL methods. In contrast, our AC and REINFORCE implementations struggled to solve most puzzles. Even in instances where they succeeded, these algorithms required significantly more memory than MC and TD. Overall, the results suggest that while MC and TD are more suitable for

Puzzle	BFS	A* (Manhattan)	A* (Dijkstra)
1	9	10	10
2	15	15	15
3	5	5	5
4	14	14	14
5	53	57	53
6	44	50	50
7	26	30	30
8	31	53	53

Table 3: Number of actions to solve puzzles for non-RL algorithms.

Puzzle	MC	TD	REINFORCE	Actor-Critic
1	3.64	9.51	12.50	46.89
2	-	52.35	-	-
3	4.24	18.95	-	11.79
4	0.64	1.70	-	-
5	-	-	-	-
6	-	-	-	-
7	74.84	60.42	-	-
8	-	-	-	-

Table 4: Time to converge to a solution (in seconds) for RL algorithms. "-" indicates the algorithm did not converge within the given constraints.

Sokoban puzzles under the current experimental conditions, AC and REINFORCE face challenges in handling the domain's complexity and computational demands.

Discussion

The objective of this project was to compare the performance of RL algorithms with non-RL algorithms on a set of Sokoban puzzles. Non-RL algorithms demonstrated superior solvability, consistently solving all puzzles in the benchmark set. In contrast, RL algorithms struggled with more complex puzzles. Other than puzzle solvability, three primary metrics are considered: time to solve a puzzle, memory usage, and the number of actions required to reach a solution.

Time Efficiency: Non-RL algorithms were considerably faster than RL algorithms across all puzzles. BFS and A* with both heuristics converged to solutions in seconds, even for the more challenging puzzles. RL algorithms, however, required more time to learn policies capable of solving even the simpler puzzles.

Memory Usage: Non-RL algorithms also exhibited greater efficiency in memory usage. BFS and A*, particularly with the Manhattan heuristic, demonstrated minimal memory overhead while maintaining their ability to solve puzzles. In contrast, RL algorithms, especially those employing neural networks such as Actor-Critic and REINFORCE, required substantially more memory due to the storage of policy networks, value functions, and training trajectories. The memory requirements of RL algorithms scaled poorly with the complexity of the puzzle.

Number of Actions: While RL algorithms occasionally

Puzzle	MC	TD	REINFORCE	Actor-Critic
1	0.38	0.48	75.83	88.24
2	-	2.66	-	-
3	0.31	0.65	-	69.12
4	0.11	0.07	-	-
5	-	-	-	-
6	-	-	-	-
7	8.26	5.28	-	-
8	-	-	-	-

Table 5: Memory usage (in MB) for RL algorithms.

Puzzle	MC	TD	REINFORCE	Actor-Critic
1	9	11	9	10
2	-	30	-	-
3	5	5	-	5
4	14	14	-	-
5	-	-	-	-
6	-	-	-	-
7	37	33	-	-
8	-	-	-	-

Table 6: Number of actions to solve puzzles for RL algorithms.

matched non-RL algorithms in terms of the number of actions required for simpler puzzles, non-RL algorithms generally produced more optimal solutions for complex puzzles. The deterministic nature of BFS and A* ensured that they identified the shortest paths, whereas RL algorithms often discovered suboptimal policies that required additional actions to solve the same puzzle.

Conclusion

The non-RL algorithms outperformed RL algorithms in all evaluated metrics—time efficiency, memory usage, and the optimality of solutions. This highlights the limitations of RL algorithms in environments with sparse rewards and high state-space complexity, such as Sokoban, and underscores the suitability of traditional search-based algorithms for such tasks.

A key limitation in our approach was how we designed our reward function. Although we introduced intermediate incentives, the reward structure did not sufficiently capture all variations of Sokoban. For instance, it did not fully account for subtle deadlock configurations or partial progress that brings boxes closer to goals without achieving immediate milestones. By engineering the reward function more carefully, such as by adding rewards for moving boxes into productive positions, and penalizing the moving of boxes out of goals, the agent could learn more efficiently, and not run into infinite loops.

Moreover, the exploration strategies used were relatively simple. Methods like count-based exploration, curiosity-driven rewards, or hierarchical RL could encourage agents to systematically discover and exploit valuable states. Similarly, integrating domain-specific insights into the RL training process like heuristic guidance for initial exploration

may help closing the gap between the heuristic search methods and the more flexible RL approaches.

In summary, our results highlight the strengths of classical search methods in handling Sokoban puzzles and emphasize the limitations of RL techniques in this domain. By improving the reward function and adopting more sophisticated exploration strategies, we could achieve better results in solving Sokoban with RL.

References

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

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

Orseau, L.; Lelis, L. H. S.; Lattimore, T.; and Weber, T. 2018. Single-Agent Policy Tree Search With Guarantees. arXiv:1811.10928.

Shoham, Y.; and Elidan, G. 2021. Solving Sokoban with forward-backward reinforcement learning. arXiv:2105.01904.

Tounsi, O.; Lin, J.; and Park, J. 2024. sokoban-rl. https://github.com/omartounsi7/sokoban-rl.

xbandrade. 2023. sokoban-solver-generator. https://github.com/xbandrade/sokoban-solver-generator.