SOLVING PLANNING PROBLEMS WITH DEEP REINFORCEMENT LEARNING AND TREE SEARCH

BY

VICTOR GE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Svetlana Lazebnik

ABSTRACT

Deep reinforcement learning methods are capable of learning complex heuristics starting with no prior knowledge, but struggle in environments where the learning signal is sparse. In contrast, planning methods can discover the optimal path to a goal in the absence of external rewards, but often require a hand-crafted heuristic function to be effective. In this thesis, we describe a model-based reinforcement learning method that bridges the middle ground between these two approaches. When evaluated on the complex domain of Sokoban, the model-based method was found to be more performant, stable and sample-efficient than a model-free baseline.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Svetlana Lazebnik, for her wisdom and guidance, and for teaching the course that introduced me to the field of computational artificial intelligence. I would also like to express gratitude to the innumerable scientists and engineers who laid the groundwork for this field, and without whom my thesis would not have been possible. Finally, I would like to thank my parents for supporting me in my research efforts and in my decision to attend graduate school.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

A2C	Advantage Actor-Critic
DFS	Depth-First Search
DQN	Deep Q-Network
ExIt	Expert Iteration
IL	Imitation Learning
MCTS	Monte-Carlo Tree Search
MDP	Markov Decision Process
RL	Reinforcement Learning

CHAPTER 1: INTRODUCTION

Reinforcement learning is a generalization of the supervised learning paradigm, in which the underlying distribution of the data shown to the decision-making agent is non-stationary, and some aspects of the data may be occluded from the perspective of the agent. Model-free reinforcement learning methods are capable of achieving superhuman performance in environments where precise control and high reaction speeds are of paramount importance [1] [2] [3]. However, in environments where long-term planning and careful deliberation are important, it becomes necessary to augment reinforcement learning with search-based methods such as Monte-Carlo Tree Search [4] [5] [6]. In this paper, we first discuss a few preliminary concepts in the fields of artificial intelligence and optimal planning. We then describe a baseline method that imitates an oracular solver and a baseline that uses model-free reinforcement learning, as well as a recent method called Expert Iteration. We demonstrate that the Expert Iteration method achieves better performance than the baseline techniques when applied to the classic puzzle game of Sokoban.

1.1 SOKOBAN

Sokoban is a challenging one-player puzzle game in which the goal is to navigate a gridworld maze and push box-like objects onto goal tiles. A Sokoban puzzle is considered solved when all boxes are positioned on top of goals. The difficulty of Sokoban puzzles emerges from its constraints: the player can move in all 4 cardinal directions, but cannot pass through the walls of the maze or boxes. The player can push adjacent boxes when there is an empty space behind the box, but cannot pull boxes. Figure 1.2 shows an example of a solution path for a puzzle with 2 boxes, and figure 1.1 explains the meaning of the visual icons.

Despite its simple ruleset, Sokoban is an incredibly complex game for which no general solver exists. It can be shown that Sokoban is NP-Hard [7] and PSPACE-complete [8]. Sokoban has an enormous state space that makes it inassailable to exhaustive search meth-

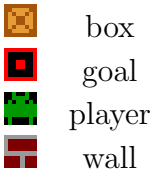


Figure 1.1: 8x8 sprites used to visually represent the basic elements of a Sokoban puzzle

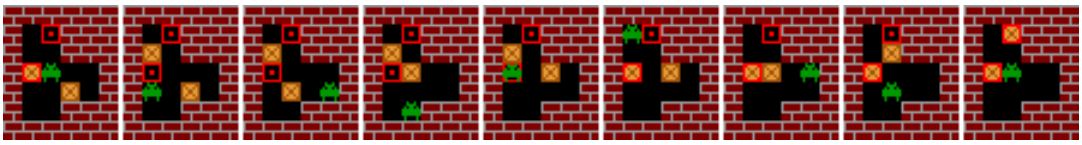


Figure 1.2: From left to right, a sequence of steps taken to solve the first puzzle in the Microban collection. Although the puzzle starts with one box on top of a goal, the player must temporarily move the box off the goal in order to maneuver the second box into a position where it can be guided to a different goal.

ods. In the XSokoban puzzle set, the branching factor can be as high as 136, the solution depth ranges from 97 to 674 box pushes, and the median search space size is roughly 10^{18} [9]. Junghanns et al. showed that an automated planner that does not exploit domain-specific knowledge requires 90 seconds to solve a simple 2-box puzzle, whereas a human would immediately intuit the solution to the puzzle. An efficient automated solver for Sokoban must have strong heuristics so that it is not overwhelmed by the number of possible game states.

Another reason why Sokoban is difficult is that moves in the game are sometimes non-reversible, which may lead to situations where the solution is not reachable from the current game state. When this occurs, the game is said to be in a state of **deadlock** [10]. This property distinguishes Sokoban from other puzzles such as sliding puzzles or Rubik's Cube, where moves are always reversible. In the general case, detecting whether a Sokoban puzzle is deadlocked is as hard as solving the puzzle, which makes deadlock detection NP-Hard as well [11].

1.2 RELATED WORK

The uniquely challenging properties of the Sokoban puzzle make it a rich domain for research, with hundreds of papers published on the topic. For the sake of brevity, we discuss only a few notable papers that deal with this domain. Weber et al. designed an actor-critic architecture that learns to interpret the trajectories generated by an imperfect model of the game environment, which outperforms a model-free agent that has the same number of parameters [12]. Our work is different because we assume that a perfect model of the environment is available, but similar in that we avoid encoding domain-dependent prior knowledge in the game-playing agent. Junghanns et al. enhanced the IDA* algorithm with several domain-specific heuristics, and were able to solve 57 out of 90 puzzles in the complex, human designed XSokoban collection [13]. In our paper, we progressively learn a heuristic, starting with zero knowledge of the domain, although the kinds of puzzles that we use

are smaller in size and less complex. Botea et al. formulate an abstract representation of Sokoban based on rooms and tunnels, which gives a potentially exponential reduction in the search space [14].

The capacity to learn domain-specific heuristics in a manner that generalizes to as many problems as possible is an important milestone for artificial general intelligence. Silver et al. achieved a major breakthrough towards this goal by training an actor-critic agent to predict the results of a Monte-Carlo Tree Search, attaining superhuman performance in Go starting from a tabula-rasa state [4]. Anthony et al. came up with a similar idea, called Expert Iteration, that involves iterating between imitation learning and MCTS. The resulting procedure outperforms the policy gradient algorithm on the board game Hex [6]. We use a similar methodology to the two aforementioned works, although we test whether this approach can be extended from the context of adversarial, zero-sum games to the graph search setting. Arfaee et al. take a machine learning approach to bootstrapping a non-admissible heuristic, which can be combined with the IDA* algorithm to find near-optimal solutions to a variety of search problems [15].

CHAPTER 2: BACKGROUND

In this section, we discuss preliminary concepts that are necessary for understanding the baseline methods and the Expert Iteration algorithm.

2.1 MARKOV DECISION PROCESS

Markov decision processes (MDP) describe a class of problems that satisfy the Markov property - that is, the behavior of the process depends on the current state, and is independent of all preceding states that the process has been in. Formally, an MDP is defined as a 5-tuple (S, A, P, R, γ), where S is a finite set of states, A is a finite set of actions, $P(s, a, s') = \text{Prb}(s'|s, a)$ is a transition function that gives the probability of reaching state s' from s by taking action a, $R(s, a)$ is a scalar reward function, and $\gamma \in [0, 1]$ is a discount factor.

Sokoban can be characterized as an MDP because it is finite in its state and action space, discrete, and fully observable. Sokoban is also deterministic, which is not necessarily true for all MDPs. Technically, Sokoban is an infinite-horizon MDP, but for the sake of practicality we assume that after 100 timesteps the puzzle is deadlocked and terminate the episode. Our version of Sokoban is therefore a finite-horizon MDP.

For a fixed room size $h \times w$, we enumerate all potential Sokoban states as the set of arrays $S = \{0, 1\}^{h \times w \times 4}$, where a 1 respectively indicates the presence of a box, goal, player or wall in a room position. Note that not all such states are valid, for instance in the case of 2 incompatible objects coexisting in the same room position or 2 players in the same puzzle. To reconcile with this fact, we can define the probability of transitioning from a valid to an invalid state to always be zero. We define the action space as the set $A = \{up, right, down, left\}$. Alternatively, we could define an action as a single box push, in which case the action space would be variable and dependent on the current state. We define the reward function as:

$$R(s, a) = \begin{cases} +1 & \text{if the puzzle is solved} \\ +0.1 & \text{if a box is pushed onto a goal} \\ -0.1 & \text{if a box is pushed off a goal} \\ -0.01 & \text{otherwise} \end{cases} \tag{2.1}$$

Finally, we set the discount factor to be $\gamma = 0.99$.

Given an MDP, we wish to find the policy, $\pi(s, a) = \text{Prb}(a|s)$, that when followed, maximizes the expected discounted reward $V^{\pi}(s)$. We can make this complex problem more amenable by using the elegant Bellman equation to formulate $V^{\pi}(s)$ in a recursive fashion:

$$V^{\pi}(s) = \sum_{a \in A} \pi(s, a)[R(s, a) + \gamma \sum_{s'} P(s, a, s')V^{\pi}(s')] \tag{2.2}$$

When the state space is small, it is possible to solve MDPs using tabular methods such as value iteration [16] or policy iteration [17]. However, for problems of practical interest, the state space is exponentially large, so it becomes necessary to use methods based on function approximation and reinforcement learning [18].

2.2 REINFORCEMENT LEARNING

Reinforcement learning (RL) is a broad approach for solving MDPs when the transition model and reward function are unknown, the environment may be partially observable, and the state space is large or possibly infinite. Since the RL agent is not endowed with a priori knowledge, it must learn by sequentially interacting with its environment, using its observations of the environment and the rewards/penalties it receives to shape its policy. RL agents infer functions to guide their decisions based on their lived experience. Value functions estimate the future reward the agent is expected to receive, assuming that it starts from a given configuration of the environment. Policy functions map from observed states to actions, and are either deterministic or stochastic. Actor-critic methods learn both a value function and a policy function.

Reinforcement learning methods can be subdivided into two categories. Model-free RL methods learn a value function and/or a policy function based on repeated interactions with the environment. Model-based RL methods additionally learn an internal model of the transition dynamics of the environment. Approximate models of the environment tend to exhibit a trade-off between accuracy and efficiency. As a result, model-free methods are the default choice for most RL systems, unless a robust simulator of the environment is available. However, a drawback of model-free methods is that they are not sample-efficient, requiring millions of frames to exceed human-level proficiency on classic Atari games [1]. In contrast, model-based methods have the potential to perform look-ahead and determine the optimal sequence of actions to reach a goal.

An important problem that lies at the core of reinforcement learning is the conflict between exploration and exploitation. RL agents must strike a balance between exploring unknown

facets of the environment and optimizing the strategies that they have discovered to work well. The exploration-exploitation dilemna is exacerbated in environments with sparse rewards, where an agent must make a series of correct decisions before receiving a reward. Ostrovski outlined a taxonomy of Atari games based on their level of exploration difficulty [19]. The Deep Q-Network (DQN) algorithm, a standard model-free RL method, was able to attain human-level performance on just 1 of the 7 Atari games with sparse rewards [1], as classified by the taxonomy. Even in environments with dense rewards, the exploration problem is non-trivial - imagine a trail of low-reward breadcrumbs leading the agent in one direction, while a high-reward goal lies hidden in the opposite direction. Model-based RL methods are a promising way of dealing with the exploration problem, by using exhaustive search to uncover sparse rewards [5] or a learned model to encourage exploration of novel aspects of the environment [19] [20].

2.3 MONTE-CARLO TREE SEARCH

Monte-Carlo Tree Search (MCTS) is an algorithm for efficiently exploring search trees that uses stochastic simulations to evaluate states in a domain-general way [21]. The advantages of MCTS over the standard minimax algorithm are that it is more tractable, does not require a utility function, and handles stochastic environments more naturally. MCTS has been successfully applied to adversarial games such as Go [22], Chess [23] and Hex [6], as well as general search problems such as chemical retrosynthesis [24]. In its purest form, the MCTS algorithm consists of 4 steps:

1. Selection - starting from the root and moving downwards, choose a leaf node in a way that balances the need to explore unknown states and the need to pursue the most promising moves.

2. Expansion - if the selected node is not terminal, create one or more child nodes, each of which corresponds to a move in the environment.

3. Simulation - follow a random policy in the selected state until a terminal point is reached.

4. Backpropogation - update all parents of the selected node to take the results of the simulation into account.

The selection step is usually performed by successively picking child nodes which maximize a formula that comprimises between the empirical value of the node based on limited

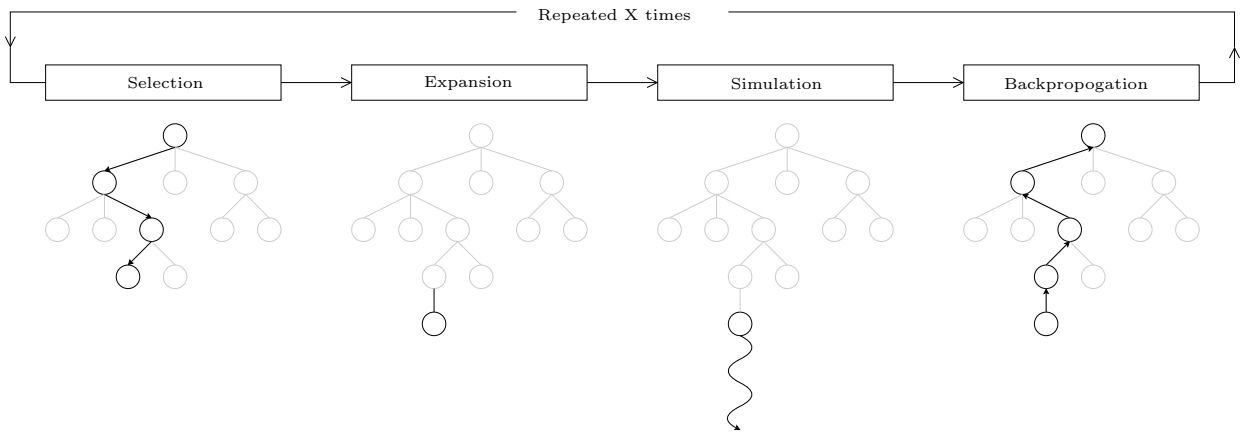


Figure 2.1: Core loop of the MCTS algorithm. As shown in the graphic, the MCTS algorithm creates search trees of variable depth and variable arity.

observations and its potential value. Most often, the selection formula is based on the Upper Confidence Bound algorithm [25], but other formulas are possible, including formulas that approximate the Gaussian distribution [21] and formulas that estimate the value of individual actions [26]. MCTS implementations can also vary in how the simulation step is performed. For some applications, random simulations have too much variance to be a useful estimate of the value of a state. One solution to this problem is to run simulations according to an informed policy, which can be hand-designed [27] or learned [22]. Another solution is to learn a value function that accurately estimates the value of a state, which then can either be linearly combined with the simulation outcome [22], or used to replace the simulation step altogether [4].

2.4 IMITATION LEARNING

One of the main challenges in reinforcement learning is that RL agents must learn from a non-stationary data distribution. RL agents can only learn about states they have visited before, but certain states may remain out of reach until the agent becomes more proficient at navigating its environment. Imitation learning avoids this chicken-and-egg dilemna by treating the problem of learning an optimal policy as a supervised learning task. In imitation learning (IL), an agent learns a policy function by mimicking the decision that an expert makes in a given state. The expert policy can be based on recordings of human experts completing a task, or an algorithmic expert such as an optimal planner with complete knowledge of the environment.

The crucial difference between imitation learning and supervised learning is that in IL,

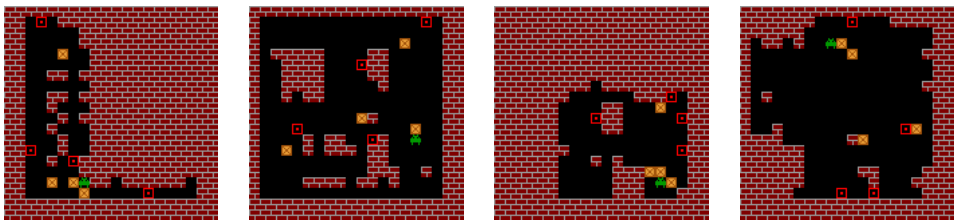
the states encountered during test time may deviate from the kinds of states that the agent is trained on. Since the agent is trained on expert trajectories, it may not know how to deal with situations that a novice could end up in. This is sometimes referred to as the **data mismatch problem** [28]. One technique that resolves this problem is DAGGER (Dataset Aggregation), an iterative algorithm for training deterministic policies [29]. In each iteration of DAGGER, a policy is trained from the current dataset of expert moves, then the dataset is expanded by asking the expert to make decisions in trajectories generated by the policy. Another strategy is to initialize a policy model by training it to predict expert moves via imitation learning, then refine the model using policy gradient reinforcement learning [22]. Although imitation learning resembles supervised learning, it can be combined with more dynamic methods so that the resulting policy is optimized to perform well in its environment, not so that it blindly follows the expert policy.

CHAPTER 3: METHODS

3.1 PUZZLE GENERATION

A Sokoban solver that functions at human-level ought to have heuristics that work for all possible puzzle configurations, and not just a single puzzle. To this end, we implement a puzzle generation procedure that is efficient and highly customizable. The puzzle generator creates puzzles that have a wild variance in their difficulty levels, which ensures that samples drawn from the generation procedure are diverse and faithfully represent the overall probability distribution of puzzles. Our puzzle generation algorithm is adapted from Weber et al. [12], who in turn were influenced by Taylor et al. [30] and Murase et al. [31].

At a high level, the puzzle generation algorithm can be divided into three phases - room generation, reverse-playing and puzzle scoring. In the first phase, a fixed-size room is created and initialized so that all positions contain a wall object. A random walk is then performed over the bounded 2D space of the room. At each position covered by the random walk, a template pattern is used to carve out a small space in the room. In the end, the first phase will have created the contours of a single, connected maze. The next phase begins by placing the player and goals in random, distinct spaces in the room. A single box is placed on each goal. Random depth-first search (DFS) is applied to 'play' the puzzle in reverse, so that the player can pull nearby boxes instead of pushing them. The DFS ends after all reachable puzzle states are explored, or once the number of visited puzzle states exceeds a maximum limit.



prob = 0.15 steps = 120	prob = 0.15 steps = 600	prob = 0.85 steps = 120	prob = 0.85 steps = 600

Figure 3.1: Examples of generated Sokoban puzzles under various parameter settings. Unless otherwise noted, the default parameter settings listed in Table 3.1 are used, except with $h = w = 20$. "prob" and "steps" are shorthand for the parameters NEW_DIRECTION_PROB and WALK_STEPS, respectively.

Name	Explanation	Value
MAX_ROOM_TRIES	Limit on how many mazes are generated.	∞
MAX_POSITION_TRIES	Limit on how many player/goal/box positions are tried for each maze.	1
MAX_ACTION_DEPTH	Maximum depth of DFS.	300
TOTAL_POSITIONS	Limit on number of distinct puzzle states DFS should find.	10^5
NEW_DIRECTION_PROB	Probability of changing direction after each move in random walk.	0.35
WALK_STEPS	Number of moves in random walk.	$1.5 * (h + w)$
ALLOW_BOX_ON_GOAL	Whether it is acceptable to generate puzzles with a box on top of a goal.	False
HEIGHT (h)	Height of puzzle.	10
WIDTH (w)	Width of puzzle.	10
NUM_BOXES	Total number of boxes/goals.	4

Table 3.1: Default parameters for puzzle generation procedure.

In the last stage, all states visited by DFS are scored by a heuristic function [12]:

$$\text{RoomScore} = \text{BoxSwaps} \times \sum_i \text{BoxDisplacement}_i \tag{3.1}$$

In the above function, BoxSwaps represents the number of times the player stops pulling a box and starts pulling a different one. BoxDisplacement represents the Manhattan distance between the starting and final position of each box. The scoring function can be configured to return 0 whenever the player of a box ends up on top of a goal. The puzzle generator returns the state that maximizes this function if the maximum score is above 0, otherwise it repeats the second phase. If the number of times the second phase is re-attempted exceeds a limit, it will repeat the first phase.

Figure 3.1 shows a few examples of generated Sokoban puzzles with different room generation parameters. In general, a small probability of changing the walk direction tends to result in long, snaky corridors, while a large probability produces round rooms that resemble Gaussian blobs. Setting a small number of walk steps gives simple and compact rooms, while using a larger number results in more interconnected and spacious mazes.

For more details on the puzzle generation procedure, we invite the reader to consult the pseudocode on the following page, or the appendix of [12].

Algorithm 3.1

```
procedure GeneratePuzzle
    for i from 1 to MAX_ROOM_TRIES do
        puzzle ← HEIGHT × WIDTH room filled with walls
        ▷ use random walk to carve out a maze
        position ← random position in interior of room
        direction ← random direction in {up, right, down, left}
        for k from 1 to walk_steps do
            carve out random pattern centered around current position
            if position + direction in room interior then
                position ← position + direction
            if random() < NEW_DIRECTION_PROB then
                direction ← random direction in {up, right, down, left}
        ▷ play puzzle in reverse using random DFS
        for j from 1 to MAX_POSITION_TRIES do
            player ← random empty space in room
            for n from 1 to NUM_BOXES do
                goals[n] ← random empty space in room
            frontier ← new Stack initialized with (puzzle, player, 0)
            visited ← {(puzzle, player)}
            while |visited| < TOTAL_POSITIONS ∧ |frontier| > 0 do
                puzzle, player, depth ← frontier.pop()
                if depth ≥ MAX_ACTION_DEPTH then
                    continue
                for direction, do_pull ∈ {up, right, down, left} × {T, F} do
                    if can move player in direction then
                        if do_pull ∧ puzzle[player − direction] = box then
                            move box to player's position
                            if moved box is different from last box then
                                increment box_swap count for puzzle
                        player ← player + direction
                        if (puzzle, player) ∉ visited then
                            frontier.push((puzzle, player, depth + 1))
                            visited ← visited ∪ {(puzzle, player)}
            puzzle, player ← argmax_{puzzle, player∈visited} Score(puzzle, player)
            best_score ← Score(puzzle, player)
            if best_score > 0 then return Sokoban(puzzle, player, goals)
procedure Score(puzzle, player)
    if ¬ ALLOW_BOX_ON_GOAL then
        if player or box on top of any goal then return 0
    for n from 1 to NUM_BOXES do
        box_dist[n] = L_1 distance of original and final position of box n
    return box_swaps × ∑_n box_dist[n]
```

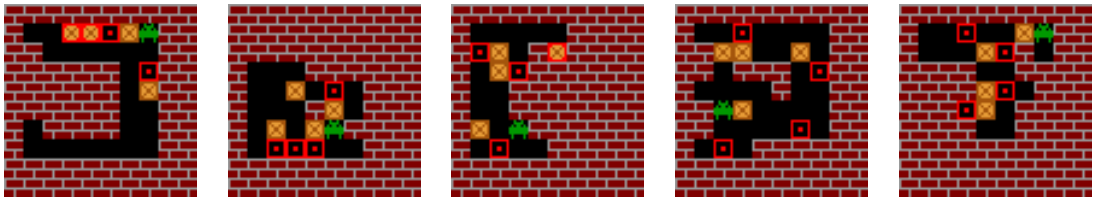


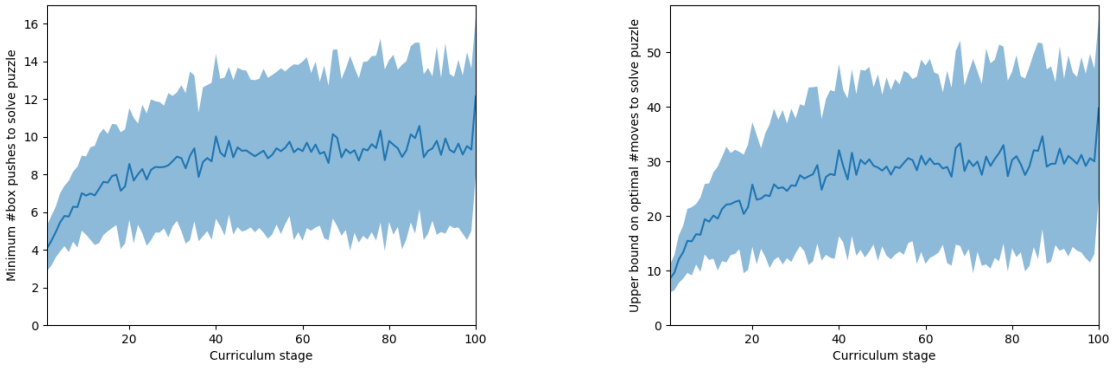
Figure 3.2: Examples of generated puzzles at selected stages.

3.2 CURRICULUM LEARNING

Curriculum learning is a technique for supervised training of complex models, in which examples are presented to the model in gradually increasing order of conceptual difficulty [32]. Hierarchical models such as deep neural networks may especially benefit from curriculum learning, because each successive layer of a neural network represents features at increasing levels of abstraction. Bengio et al. hypothesize that curriculum learning acts similarly to unsupervised pre-training, a strategy in which layers of the neural network are trained one at a time [32] [33]. Empirically, for non-convex models, curriculum learning has been proved to increase the rate of convergence of training and improve the quality of the final model.

There are a few reasons why one might want to employ curriculum learning when training an agent to solve Sokoban levels, starting from zero knowledge of the environment. The initial behavior of a bootstrapping agent approximates a random walk, which means the time for the agent to reach some goal is exponentially proportional to the distance from the goal to the starting point. In Sokoban, the complexity of the puzzle increases dramatically as more box-goal pairs are introduced. Even when a box is on top of a goal, it can still be entangled with the solution path of other boxes and contribute to the complexity of the puzzle, as in Figure 1.2. Furthermore, a random agent is likely to carelessly push a box into a corner and deadlock the puzzle, rendering it unsolvable. There is almost zero probability that a random agent will be capable of solving an average Sokoban puzzle.

Given the above reasons, we use curriculum learning for all of our experiments. Our curriculum is divided into 100 stages. At each stage of the curriculum, we configure the puzzle generator with different pre-defined parameter settings. Our curriculum modifies the generation parameters in the following way: the MAX_ACTION_DEPTH parameter is linearly annealed from 10 to 300, and the TOTAL_POSITIONS parameter is increased from 10^2 to 10^5 in a log-uniform manner. In addition, the ALLOW_BOX_ON_GOAL parameter is set to `true` for all stages other than the final stage. In the first stage, it is feasible to solve some of the generated puzzles using a random walk, and in the final stage, the generator



(a) (b)

Figure 3.3: Plots showing the average and standard deviation for two metrics of puzzle difficulty for each stage in the curriculum.

uses the parameter settings listed in Table 3.1.

After each iteration of training, we use the following 3 conditions to decide whether or not to allow the agent to progress to the next curriculum stage:

1. The percentage of puzzles solved (solve rate) must be over 50%.
2. At least 10 iterations have passed.
3. The absolute difference between the solve rate for this iteration and the average solve rate over the last 10 iterations must be less than 1%, or the solve rate is over 90%.

These 3 conditions jointly ensure that the difficulty of generated puzzles is appropriate for the current problem-solving capabilities of the agent. Condition (1) prevents the solve rate from approaching 0%, a situation we want to avoid because there would be no useful signal for the agent to learn from. Conditions (2) and (3) ensure that the agent's learning progress has stabilized before it continues to the next stage. The last 2 conditions also prevent the agent from blithely advancing through the stages if its performance suddenly decreases upon reaching a new stage.

Figure 3.3 empirically demonstrates that on average, the difficulty of puzzles increases as a function of the curriculum stage. To obtain this data, we first generate 100 different puzzles for each curriculum stage. We then run A* with the min-matching heuristic, described in Section 3.4, to determine the optimal number of box pushes required to solve each puzzle. The A* algorithm minimizes box pushes and not player moves, as there is no good admissible heuristic for estimating the optimal solution length in terms of player moves. However, we

can find an upper bound on the minimal number of moves by calculating the shortest distance that the player can travel in between box pushes, and summing these distances. We obtain a tighter upper bound by running A* 100 times with different random seeds, and then returning the smallest estimated move count out of all runs.

In summary, curriculum learning is vital to training agents for the Sokoban domain, because the capability of the game-playing agent and the difficulty of puzzles must be kept in balance. If the proportion of puzzles the agent is able to solve approaches the extrema of 0% or 100%, the agent will cease to learn effectively. This problem appears in generative adversarial networks, where training can be destabilized if the generator or discriminator outpaces the other network [34] [35]. 2-player symmetric games do not face this problem, because agents can be trained using self-play.

3.3 A*

A* is a graph traversal algorithm that can find the shortest path between two nodes in a weighted, directed graph. A* is an informed search algorithm, because it can make use of a heuristic to find the target node more efficiently. This property differentiates A* from search algorithms such as depth-first search and Dijkstra's algorithm - in fact, A* reduces to Dijkstra's algorithm when a trivial heuristic $h(n) = 0$ is used. The efficiency of A* depends heavily on the accuracy of the heuristic that is used. A* search is only guaranteed to find the optimal path if the heuristic is **admissible**, that is, if for every node the heuristic returns a lower bound on the optimal distance to the target node. The tighter this lower bound is, the fewer nodes A* must visit to find the optimal path. Given two admissible heuristics h_1 and h_2, if for every node $h_2(n) \geq h_1(n)$, then h_2 is said to **dominate** h_1.

Sokoban can be recast as a finite graph traversal problem, because it is deterministic, fully observable and its action space and transition dynamics are discrete. In theory, Sokoban puzzles can always be solved with A*, although in practice it can take an exponential amount of time and computing resources to do so. Junghanns et al. demonstrated that even when equipped with a powerful, application-dependent heuristic, the IDA* algorithm was not able to solve 33 out of 90 challenging puzzle instances in a reasonable amount of time [13]. In this next section, we describe a series of admissible heuristics for the Sokoban domain that can be used to augment the A* algorithm, many of which were taken from [13].

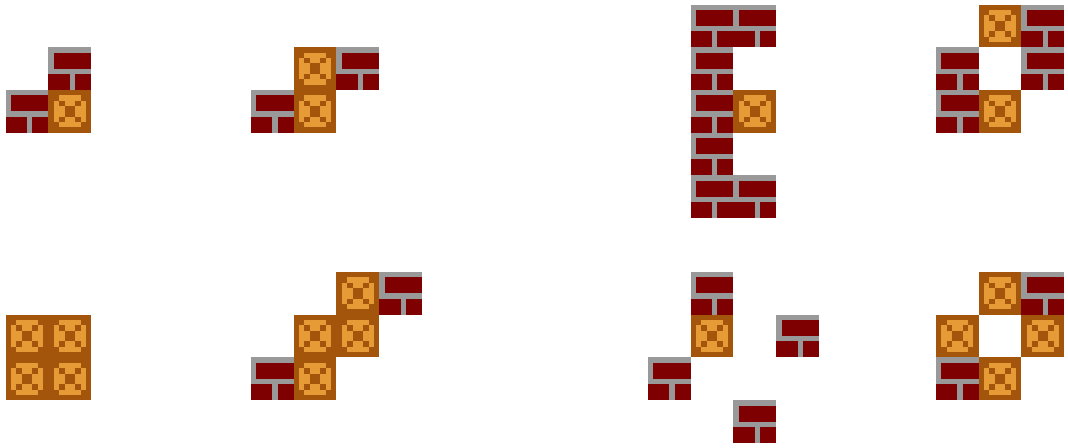
Relaxation	remaining-boxes	Manhattan-dist	min-matching
Unconstrained movement	✓	✓	✓
Multiple boxes on goal	✓	✓	
Non-goal moves are free	✓		

Table 3.2: From right to left, each heuristic progressively allows more relaxation of the game rules. From left to right, each heuristic dominates the previous one.

3.3.1 Domain-Specific Heuristics

We use a slightly different representation of the Sokoban environment for the optimal solver than for the bootstrapping agents. For the A2C and ExIt agents, the action space is defined as the set of 4 movements in the cardinal directions that the player can take. For the A* agent, the action space is defined as the set of all box pushes that the player can perform. This divergence is necessary because the vast majority of Sokoban solvers in the literature use the latter representation, so there are no known efficient heuristics for estimating the optimal number of player moves to solve a puzzle.

The simplest heuristic for the Sokoban domain is to count the number of boxes that have yet to be pushed to a goal. This heuristic, which we call the remaining-boxes heuristic, can be thought of as a relaxation of the game rules where movement is not impeded by walls or boxes, and a move is not counted until a box is transported on top of a goal. A more practical heuristic is to measure the number of pushes it would take to move each box to the nearest goal, and then sum these distances. We refer to this heuristic as the Manhattan-distance heuristic, and it is equivalent to a relaxation of the rules where multiple boxes can be pushed to the same goal and objects do not exclusively occupy tiles. To make the previous heuristic more realistic, we can match every box to a different goal, and return the minimum total Manhattan distance out of all matchings. This is known as the minimum-matching heuristic [13]. Our implementation of this heuristic uses the Hungarian algorithm for perfect bipartite matching [36]. This algorithm runs in $O(n^3)$ time, where n is the number of boxes, but because our experiments are limited to 4 boxes, it is essentially a constant time operation. These 3 heuristics form a kind of hierarchy, in which the minimum-matching heuristic dominates the Manhattan-distance heuristic, which in turn dominates the remaining-boxes heuristic.



(a) Static deadlock (b) Dynamic deadlock

Figure 3.4: 4 simple examples of each type of deadlock.

3.3.2 Deadlock Detection

An orthogonal heuristic to the 3 heuristics described in the previous section is to use pattern matching to detect simple situations in which deadlock must be present. This deadlock-detection heuristic returns ∞ when the input puzzle contains a pattern known to cause deadlock, and 0 otherwise. We can combine the information from multiple admissible heuristics by taking the maximum value out of all heuristics for a given puzzle state, thereby creating a new admissible heuristic. The deadlock-detection heuristic in tandem with the min-matching heuristic can make A* search orders of magnitude more efficient, although these performance gains are still not enough to solve the most challenging Sokoban puzzles [10].

To establish an intuition for how the deadlock-detection heuristic might work, we can consider a simple example of a pattern that necessitates deadlock, which is a box trapped in a corner. Any puzzle that contains this pattern clearly must be unsolvable. However, if the box is also on top of a goal, then there may still be hope for solving the puzzle. We can differentiate between two types of deadlock patterns, depending on how they are affected by the presence of goals. **Static deadlock** is the case where one or more boxes are completely immobile, and cannot be nudged from its current position no matter how the other boxes are moved around. If a box is on top of a goal, then testing whether it is in static deadlock is fruitless. **Dynamic deadlock** is the case where a box cannot be moved outside of a fixed zone. Dynamic deadlock is only applicable when there are no goals inside of the zone. Static

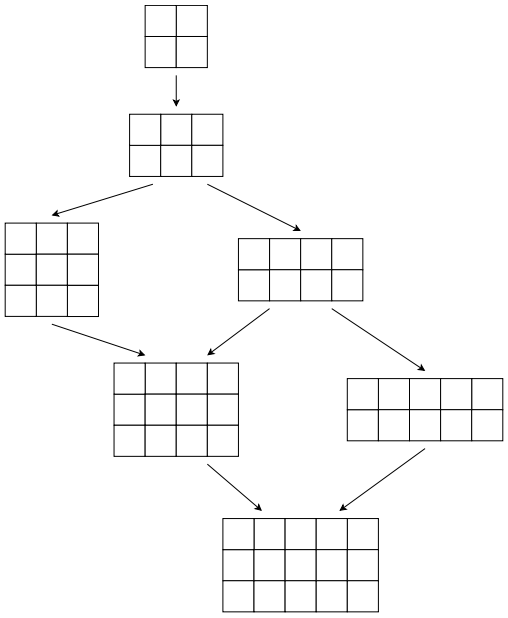
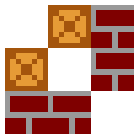


Figure 3.5: Topological ordering of pattern sizes. We skip the 1x1, 1x2 and 1x3 cases, because there are no deadlock patterns of that size.

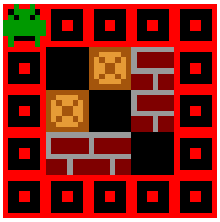
deadlock is a special case of dynamic deadlock, although static deadlock is more lenient in the conditions under which it can be applied.

In our implementation of deadlock detection, we essentially compare every 3×5 window of the puzzle against a pattern database. We generate different pattern databases for static and dynamic deadlock. The naive way to generate such a database is to iterate over an exponential set of states and test whether each state is in deadlock. In the worse case, one may need to exhaust the search space in order to ascertain that the state is not in deadlock. We advocate for a more scalable approach that exploits the fact that if more boxes or walls are added to a puzzle that is in static or dynamic deadlock, the puzzle will continue to be deadlocked. With this principle in mind, we can start by finding the smallest deadlock patterns, and then as we work our way up, cull the patterns that already contain a known deadlock pattern.

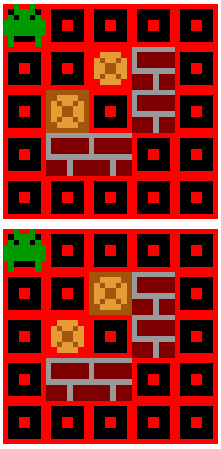
The details of our deadlock table generation method are as follows: First, we define a topological ordering of pattern sizes. A pattern size s_1 is the parent of a pattern size s_2 if and only if s_1 fits inside the area of s_2, and there are no other pattern sizes s_3 such that s_3 fits inside s_2 and s_1 fits inside s_3. We also ignore symmetric pattern sizes. Figure 3.5 illustrates an example of this ordering. We iterate over all pattern sizes from 2×2 up to 3×4 and 2×5 in topological order. For a pattern size $r \times c$, we enumerate over all $3^{r \cdot c}$ configurations of spaces, walls and boxes. We prune configurations if, for some pattern size $r' \times c', r' \leq r, c' \leq c$, the configuration contains a known deadlock pattern of that size. We



(a) Candidate deadlock pattern



(b) Auxiliary puzzle for dynamic deadlock



(c) Auxiliary puzzles for static deadlock

Figure 3.6: Sample auxiliary puzzles for both static and dynamic deadlock. By attempting to solve the auxiliary puzzles, we can see that the pattern is a dynamic deadlock pattern, but not a static one.

also prune configurations that contain more than 4 boxes. If the configuration cannot be pruned, then we check whether it is a deadlocked pattern. We do this by embedding the pattern in a Sokoban puzzle of size $(r+2) \times (c+2)$, and then trying to solve this auxiliary puzzle with A*. In the case of dynamic deadlock, the auxiliary puzzle is set up so that the puzzle can only be solved if all boxes can be pushed out of the boundaries of the pattern. In the case of static deadlock, we create auxiliary puzzles for each box which are only solvable if the box can be moved in any direction. Figure 3.6 displays concrete examples of each type of auxiliary puzzle. For the auxiliary puzzles, we modify the rules of Sokoban so that if a box is pushed outside of the boundaries of the puzzle, it disappears. Once this process is finished, we end up with a relatively small set of deadlock patterns which can be readily checked by hand. The final step is to enumerate over all $3^{3 \cdot 5}$ puzzle states and check whether any rotation and/or reflection of the deadlock patterns can be found in the puzzle state. If so, we add the state to the deadlock table.

3.4 IMITATION LEARNING FROM A*

One of our baseline agents is a purely supervised model that does not interact with the game environment, but instead learns to mimic an oracular decision-maker. The oracle is an A* agent that uses backtracking search to find a near-optimal series of moves to solve a puzzle. We first generate 10,000 puzzles using the default generator parameters in Table

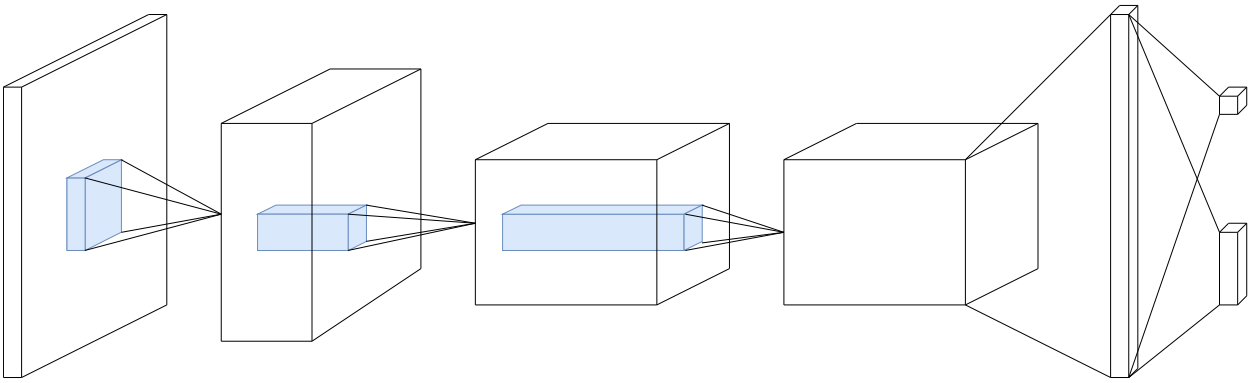


Figure 3.7: Convolutional actor-critic neural network.

3.1. We run A* with the min-matching heuristic to find an optimal series of box pushes to solve each puzzle. We use the procedure described in Section 3.2 to convert from a minimal sequence of box pushes to a near-minimal sequence of player moves, using 10 different random seeds. For each move sequence, we select 5 random state-action pairs to use as training data. We also record the number of additional moves needed to solve the puzzle at each state. The imitation learning agent is trained to output the same action that the A* oracle took at each state, and to output the value of the state, as determined by the reward function used by the Expert Iteration agent (see Section 3.5).

We additionally train the imitation learning agent to recognize simple deadlock states. To do this, we run DFS on 1,000 of the puzzles generated earlier. We then apply the deadlock-detection heuristic to the puzzle states visited by DFS and randomly select one deadlocked state for each puzzle. The imitation learning agent is trained to output a value of 0 and to not prioritize any particular action for the deadlocked states.

We split the training data so that 8,000 puzzles are used for training and 2,000 puzzles are set aside as a validation set. The imitation learning agent is trained for 20,000 batches, with a batch size of 128 training examples. After 20k steps, performance on the validation set stops improving. The total loss is an unweighted sum of the cross-entropy error of action probabilities and the mean square error of predicted values. During training, we apply a

	Imitation learning	A2C	ExIt
Weight init.	Truncated normal	Orthogonal init.	Truncated normal
Regularization	Weight decay	Policy entropy + Gradient clipping	Weight decay
Optimizer	Adam	RMSProp	Adam

Table 3.3: Differences in the training regime for each method.

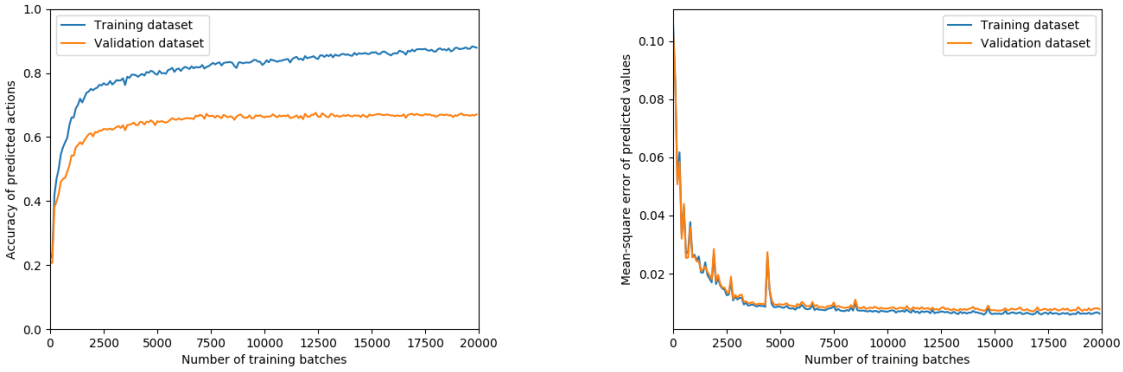


Figure 3.8: Training progress of imitation learning agent.

random number of rotations and reflections to the puzzle state before adding it to the mini-batch. This trick effectively increases the size of our dataset by a factor of 8. Table 3.3 shows more details about the training regime.

We use the same actor-critic neural network model across all experiments, depicted in figure 3.7. The model consumes an RGB image of the puzzle state and outputs a scalar value estimate along with a softmax distribution over actions. It is a standard architecture that consists of 3 convolutional layers with kernel sizes 8×8, 4×4, 3×3, output channels of 32, 64, 64, and strides of 4, 2, 1 [37]. The final convolutional layer feeds into a fully connected layer with a dimensionality of 512, followed by 2 linear operations to produce the action and value output.

3.5 EXPERT ITERATION

Expert Iteration (ExIt) is a novel technique for bootstrapping policies that interleaves between imitation learning and in-depth planning [6]. ExIt maintains two types of policies - an apprentice policy that generates fast, heuristic-driven decisions based on the current state, and an expert policy that uses lookahead to converge upon a more accurate decision. The behavior of the apprentice and expert policy are intertwined - the apprentice gradually learns to generalize the expert policy, and the expert uses the apprentice as a heuristic to guide its own decisions. When a perfect model of the environment is available, ExIt can be an effective alternative to model-free reinforcement learning methods, which are often plagued with high variance and slow convergence. Variations of the ExIt strategy have shown strong performance on the board games Hex [6] and Go [4].

In our implementation of ExIt, we use an actor-critic neural network as the apprentice

policy, and Monte-Carlo Tree Search as the expert policy. We refer to this hybrid method as MCTS-ExIt. The apprentice neural model reads in an RGB image of the puzzle, and outputs a softmax distribution over the next action to take, along with a scalar prediction of the cumulative reward the agent will receive. We use a modified reward function that encourages the agent to solve puzzles faster while avoiding deadlock. The reward function yields 1 if the agent eventually solves the puzzle and 0 if it cannot, with -0.01 subtracted for each step the agent must take to reach the solution. This linear discounting ensures that there is no discontinuity between the case where it takes 99 steps to solve the puzzle and the case where the agent is unable to find a solution after 100 steps elapse. The MCTS expert differs from canonical MCTS implementations mainly in two ways - it stores information in edges instead of nodes, and it uses the actor-critic model to evaluate leaf nodes instead of running a simulation to reach a terminal state. Our MCTS implementation iterates between 4 steps, with the following nuances:

1. Selection - starting from the root node, successively pick child nodes that maximize the equation:

$$\frac{1}{N(s,a)} \sum_{s' \in \text{subtree}(s,a)} V(s') + c * P(s,a) * \frac{1}{1+N(s,a)} \sum_{a'} N(s,a') \tag{3.2}$$

   where $N(s,a)$ is the number of times the state-action pair has been selected, subtree(s,a) is the set of all nodes under the root pointed to by edge a in the node s, $V(s')$ is the estimated value of a node, c is equal to $1/\sqrt{2}$, and $P(s,a)$ is a prior value that indicates the probability the actor-critic model would choose the action in the given state. We also maintain a transposition table for visited puzzle states, and avoid selecting nodes whose corresponding state has been visited before.

2. Expansion - Populate the child list of the selected leaf node with 4 unexpanded nodes. Initially, the root node is unexpanded.

3. Evaluation - Run the actor-critic network to determine the value of $V(s)$ and $P(s,a)$ for the selected node. We delay the evaluation step until 8 nodes need to be evaluated, using the placeholder value of $P(s,a) = 0.25$ in the meantime. This allows us to leverage the parallelism of the GPU to improve the efficiency of this bottleneck step.

4. Update - After the node is evaluated, trace backwards from the node to the root and update each node along the way with $V(s)$.

After 256 nodes have been evaluated, a move is chosen based on the visitation frequencies of the edges of the root node. There are two design decisions for how we could use this information to choose a move, both of which are investigated in our experiments. The **deterministic** ExIt agent picks the move with the highest visitation count. The **stochastic** ExIt agent chooses a move probabilistically, with each move weighted by its selection count. An argument in favor of the deterministic agent is that it is less likely to make catastrophic decisions that lead to deadlock. The stochastic agent is motivated by the observation that cycles, where the agent repeatedly loops through the same set of moves, are a common failure case of deterministic Sokoban agents [38]. After the agent makes a move, it reuses the portion of the search tree that corresponds to the move - the child node pointed to by the edge that represents the played move becomes the new root node.

For each timestep of puzzle playing, we note the puzzle state, the reward that the agent eventually receives, and the likelihood that each move was selected during tree search. We record at most 50 state-reward-action triples for each puzzle. This sampling method slightly reduces the correlations in the sequence of observations, which is beneficial because such correlations have been noted to destabilize the training of agents in the RL setting [1]. We maintain a queue of the last 500,000 data points, which plays a similar role to the Experience Replay buffer used by DQN [1]. The actor-critic model is trained to predict the scalar reward that the MCTS expert will achieve and the probability distribution over moves selected by the search procedure, given a snapshot of the puzzle state. We apply a random number of reflections and/or rotations to the puzzle state to induce the model to generalize better. Furthermore, we do not penalize the model for mispredicting the move distribution in the cases where the puzzle was not solved.

We use the curriculum learning scheme described in section 3.2. In each iteration, we generate 100 puzzle instances for the MCTS expert to solve, train the actor-critic apprentice for 10,000 batches of size 128, and then update the curriculum stage.

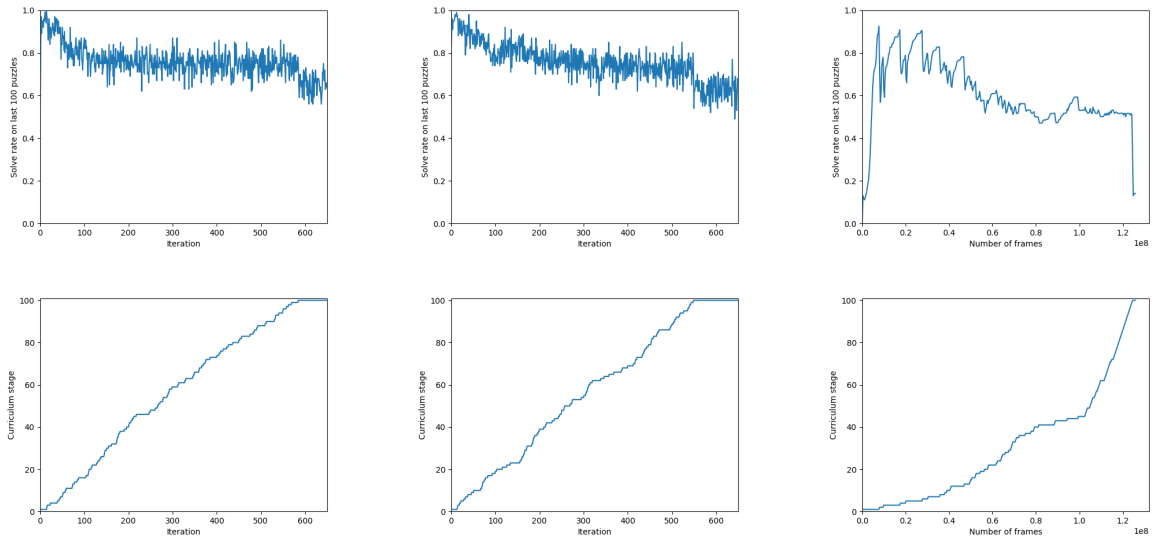
CHAPTER 4: RESULTS

In our experiments, we train neural network heuristics starting from zero knowledge using Expert Iteration and curriculum learning. We test the deterministic and stochastic variants of MCTS-ExIt. We also compare against two baselines: a model-free reinforcement learning algorithm called Advantage Actor-Critic (A2C), and an imitation learning baseline, described in Section 3.4.

A2C is a synchronous version of the A3C reinforcement learning algorithm [37]. We use the OpenAI implementation of A2C, as found in the `baselines` Python library [39]. We retain most of the default hyperparameters, except that we use 64 actors instead of 16. For the purpose of curriculum learning, we define an iteration as equivalent to 320,000 total timesteps across all agents. The A2C agent is trained until it reaches the last curriculum stage and obtains at least 1.5 million timesteps of experience in the last stage.

Figure 4.1 shows how the ExIt and RL methods gradually acquire the ability to solve puzzles. We observe that the deterministic and stochastic versions of ExIt are nearly identical in their performance. We can also see that the ExIt methods are more robust to perturbations in the environment. Whenever the curriculum stage is advanced, the performance of the A2C method drops precipitously, followed by a gradual upward climb. In contrast, the ExIt methods are more stable, and only experience a sharp drop in performance when the curriculum stage changes from 99 to 100. There is a discontinuity in puzzle difficulty from stage 99 to 100, as the boolean flag that specifies whether boxes are allowed to initially reside on goals is changed from `true` to `false`. The A2C method does not cope well with this sudden change, and its solve rate plummets to the rate that it had at the beginning of training.

At a first glance, it may seem like the methods are not making progress because on a macroscopic scale, the solve rates decrease over time. However, the decrease in performance can be attributed to the increased difficulty of higher curriculum stages - the average decrease in solve rate when the curriculum stage changes is -0.0026 for deterministic ExIt and -0.019 for stochastic ExIt, and the average increase in solve rate from the start to the end of a curriculum stage is +0.0035 for deterministic ExIt and +0.018 for stochastic ExIt. As counterintuitive as it may be, this proves that the ExIt methods are learning to solve the puzzles, despite the overall downward trend in the progress graph.



(a) Deterministic ExIt (b) Stochastic ExIt (c) A2C

Figure 4.1: Learning progress of the bootstrapping methods.

4.1 MCTS NEURAL HEURISTIC

We also test how effective the different neural network models are when used in the evaluation step of MCTS. To do this, we generate a test dataset of 1,000 puzzles using the puzzle generator with default parameters, described in Section 3.1. We run MCTS to decide what action take at each timestep, re-using relevant parts of the search tree in between moves. For the stochastic ExIt heuristic, the action is chosen probabilistically, and for the other heuristics, the chosen action is always the one with the highest visitation count. The number of rounds that MCTS is allowed to run for each move is equal to successive powers of two, starting from 2 and ending at 1024. We do not delay the evaluation step when the number of rounds that MCTS is allowed to run for is less than 64. Finally, we reshape the range of the value function of the A2C model from [-1, 1] to [0, 1].

The results of this experiment are shown in Figure 4.2. The ExIt heuristics clearly outperform the two baseline heuristics, and the deterministic and stochastic variants of ExIt perform comparably well. We speculate that even though the imitation learning method had access to information that the bootstrapping methods were not privileged to have, it failed to perform well because of overfitting or the data mismatch problem.

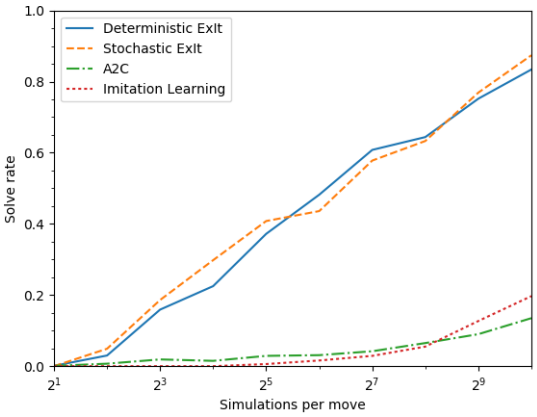


Figure 4.2: Efficiency of various neural network actor-critic models when used as a heuristic in MCTS.

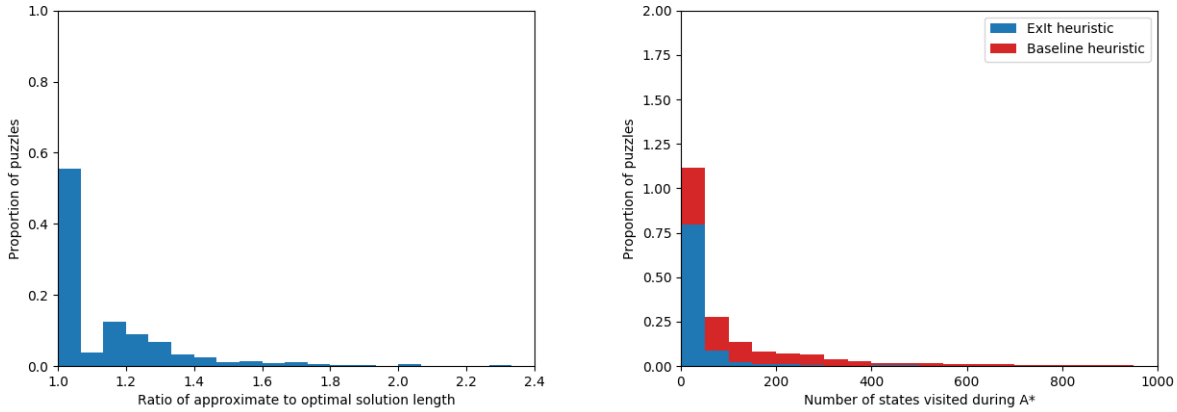
4.2 A* NEURAL HEURISTIC

One point of contention against the results of the previous section is that the ExIt models perform well as a heuristic for MCTS only because they are trained using MCTS as the expert function. In this section, we show that a model trained using MCTS-ExIt can be effectively used as a heuristic for the A* pathfinding algorithm. We re-use the test dataset of 1,000 puzzles described in the previous section. We derive an inadmissible heuristic from the deterministic ExIt model by inverting its value function:

$$h_{ExIt}(n) = (1 - V(n)) * 100 \tag{4.1}$$

We compare the ExIt heuristic against a hand-crafted admissible heuristic that is equal to the maximum of the min-matching heuristic, dynamic deadlock detection heuristic and static deadlock detection heuristic. To do this, for each puzzle in the test dataset, we run A* with both heuristics 10 times, and record the average solution length and the average number of states visited during A*. The results of this experiment are shown in Figure 4.3. For the second graph, we only show the cases where less than 1000 states are visited, omitting the long tail of the distribution.

Overall, A* is able to get away with visiting fewer states using the ExIt heuristic, compared to the hand-crafted baseline, although the solutions it finds with the ExIt heuristic are sometimes suboptimal in terms of the number of times a box is pushed. Figure 4.3(a) quantifies the suboptimality of the ExIt heuristic. Surprisingly, A* with the ExIt heuristic finds an optimal solution for 55.4% of the puzzles. The ExIt heuristic actually estimates the number of player moves to complete the puzzle, so it greatly overestimates the solution

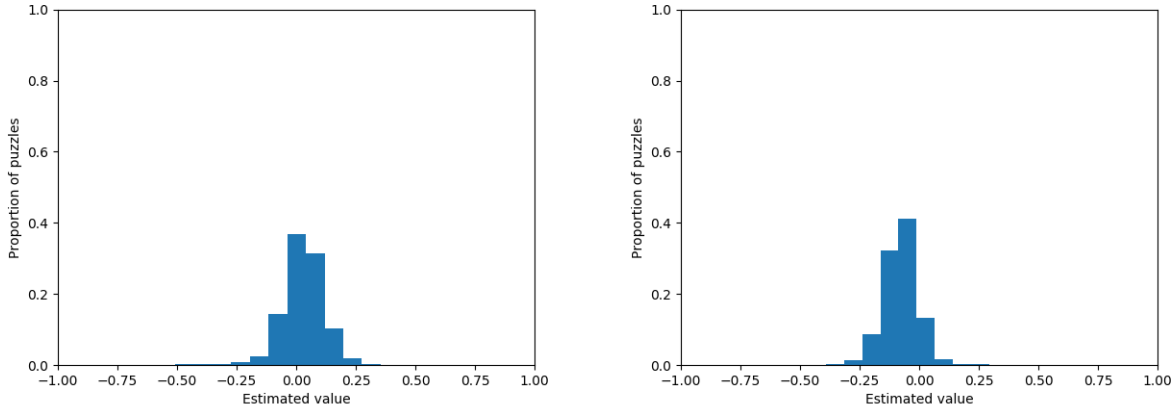


(a) (b)

Figure 4.3: Comparison of hand-crafted and neural A* heuristics.

length with respect to the number of box pushes. Figure 4.3(b) compares the efficiency of both heuristics. For 90.8% of the puzzles, A* visits fewer states with the ExIt heuristic compared to the hand-crafted heuristic. In many cases, the ExIt heuristic can almost directly guide the A* algorithm to the goal state.

4.3 APPROXIMATE DEADLOCK DETECTION



(a) Deterministic ExIt (b) Stochastic ExIt

Figure 4.4: Estimated value for deadlocked states.

In this section, we demonstrate that the ExIt models have learned to recognize a subset of deadlocked states. We generate 9,984 deadlocked puzzles using the procedure described

in Section 3.4. Next, we run a foward pass of the deterministic and stochastic ExIt models for each puzzle. Figure 4.4 shows two histograms for the estimated value of the deadlocked puzzles. In general, the estimated values seem to cluster close to 0, suggesting that the ExIt models have the capacity to recognize some deadlock states. The estimated values for the stochastic ExIt model tend to lean slightly more negative, perhaps because it has learned to avoid accidentally transitioning into a deadlock state.

CHAPTER 5: CONCLUSION

We have implemented two reinforcement learning methods for learning domain-specific heuristics from scratch - a model-free baseline and a recent model-based technique called MCTS-ExIt. We also developed an imitation learning method that mimics an A* oracle. We compare the performance of these methods on the complex domain of the Sokoban puzzle game. In general, we found that the model-based method is more stable to changes in the environment and achieves a higher solve rate at the end of training.

The model-free A2C algorithm and model-based MCTS-ExIt algorithm have relative advantages and disadvantages. Over the course of 650 iterations, the MCTS-ExIt algorithm was exposed to 65,000 different puzzles. In contrast, the A2C algorithm observed at least 1,259,840 puzzles over the course of its training. This confirms the general wisdom that model-based RL methods are more sample efficient than their model-free counterparts; the model-based method was able to extract more information from individual puzzles and generalize from a limited amount of information. The main benefit of model-free methods is that they do not require a perfect simulation of the environment, which is convenient when the environment has a high degree of randomness, hidden information, or other complicating factors that make it difficult to accurately model.

5.1 FUTURE WORK

One avenue for future research is to investigate alternate choices for the expert policy of ExIt. For instance, Groshev et al. explore how to learn a reactive policy that imitates execution traces of the A* search algorithm. [38]. Their method learns a neural heuristic for the Sokoban domain, similar to our work.

REFERENCES

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[2] V. Firoiu, W. F. Whitney, and J. B. Tenenbaum, "Beating the world's best at super smash bros. with deep reinforcement learning," *arXiv preprint arXiv:1702.06230*, 2017.

[3] OpenAI, "More on dota 2," https://blog.openai.com/more-on-dota-2/, 2017.

[4] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton et al., "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[5] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline monte-carlo tree search planning," in *Advances in neural information processing systems*, 2014, pp. 3338–3346.

[6] T. Anthony, Z. Tian, and D. Barber, "Thinking fast and slow with deep learning and tree search," in *Advances in Neural Information Processing Systems*, 2017, pp. 5366–5376.

[7] D. Dor and U. Zwick, "Sokoban and other motion planning problems," *Computational Geometry*, vol. 13, no. 4, pp. 215–228, 1999.

[8] J. Culberson, "Sokoban is pspace-complete," 1997.

[9] T. Virkkala, "Solving sokoban," Ph.D. dissertation, Masters thesis, University Of Helsinki, 2011.

[10] A. Junghanns and J. Schaeffer, "Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock," in *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer, 1998, pp. 1–15.

[11] T. Schaul, "Evolving a compact concept-based sokoban solver," *Master's thesis, École Polytechnique Fédérale de Lausanne*, 2005.

[12] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li et al., "Imagination-augmented agents for deep reinforcement learning," *arXiv preprint arXiv:1707.06203*, 2017.

[13] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing general single-agent search methods using domain knowledge," *Artificial Intelligence*, vol. 129, no. 1-2, pp. 219–251, 2001.

[14] A. Botea, M. Müller, and J. Schaeffer, "Using abstraction for planning in sokoban," in *International Conference on Computers and Games*. Springer, 2002, pp. 360–375.

[15] S. J. Arfaee, S. Zilles, and R. C. Holte, "Learning heuristic functions for large state spaces," *Artificial Intelligence*, vol. 175, no. 16-17, pp. 2075–2098, 2011.

[16] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, pp. 679–684, 1957.

[17] R. A. Howard, *Dynamic programming and Markov processes*. Wiley for The Massachusetts Institute of Technology, 1964.

[18] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[19] G. Ostrovski, M. G. Bellemare, A. v. d. Oord, and R. Munos, "Count-based exploration with neural density models," *arXiv preprint arXiv:1703.01310*, 2017.

[20] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. De Turck, and P. Abbeel, "Vime: Variational information maximizing exploration," in *Advances in Neural Information Processing Systems*, 2016, pp. 1109–1117.

[21] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.

[22] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot et al., "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[23] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel et al., "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[24] M. Segler, M. Preuß, and M. P. Waller, "Towards" alphachem": Chemical synthesis planning with tree search and deep neural network policies," *arXiv preprint arXiv:1702.00020*, 2017.

[25] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[26] S. Gelly and D. Silver, "Combining online and offline knowledge in uct," in *Proceedings of the 24th international conference on Machine learning*. ACM, 2007, pp. 273–280.

[27] M. Świechowski and J. Mańdziuk, "Self-adaptation of playing strategies in general game playing," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 4, pp. 367–381, 2014.

[28] J. Schulman, "Berkeley cs 294, lecture: Dagger and friends," October 2015, 2015.10.5.dagger.pdf. [Online]. Available: http://www.rll.berkeley.edu/deeprlcourse-fa15/docs/

[29] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 627–635.

[30] J. Taylor and I. Parberry, "Procedural generation of sokoban levels," in *Proceedings of the International North American Conference on Intelligent Games and Simulation*, 2011, pp. 5–12.

[31] Y. Murase, H. Matsubara, and Y. Hiraga, "Automatic making of sokoban problems," in *Pacific Rim International Conference on Artificial Intelligence*. Springer, 1996, pp. 592–600.

[32] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*. ACM, 2009, pp. 41–48.

[33] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[34] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[35] T. Karras, T. Aila, S. Laine, and J. Lehtinen, "Progressive growing of gans for improved quality, stability, and variation," *arXiv preprint arXiv:1710.10196*, 2017.

[36] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics (NRL)*, vol. 2, no. 1-2, pp. 83–97, 1955.

[37] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1928–1937.

[38] E. Groshev, A. Tamar, S. Srivastava, and P. Abbeel, "Learning generalized reactive policies using deep neural networks," *arXiv preprint arXiv:1708.07280*, 2017.

[39] P. Dhariwal, C. Hesse, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Openai baselines," 2017.