Solving Hard AI Planning Instances Using Curriculum-Driven Deep Reinforcement Learning

Dieqiao Feng, **Carla P. Gomes** and **Bart Selman**
Department of Computer Science
Cornell University
{dqfeng, gomes, selman}@cs.cornell.edu

Abstract

Despite significant progress in general AI planning, certain domains remain out of reach of current AI planning systems. Sokoban is a PSPACE-complete planning task and represents one of the hardest domains for current AI planners. Even domain-specific specialized search methods fail quickly due to the exponential search complexity on hard instances. Our approach based on deep reinforcement learning augmented with a curriculum-driven method is the first one to solve hard instances within one day of training while other modern solvers cannot solve these instances within any reasonable time limit. In contrast to prior efforts, which use carefully handcrafted pruning techniques, our approach automatically uncovers domain structure. Our results reveal that deep RL provides a promising framework for solving previously unsolved AI planning problems, provided a proper training curriculum can be devised.

1 Introduction

Deterministic, fully observable planning is a key domain for artificial intelligence. In its full generality, AI planning encompasses general theorem proving, where proofs can be viewed as plans leading from a set of basic axioms to the theorems to be proved. Planning is well known to be a very challenging computational problem: finding proofs in a strong first-order mathematical theory which encodes basic arithmetic is undecidable [Gödel, 1931] and plan-existence is PSPACE-complete for propositional STRIPS planning [Bylander, 1994]. Domain-independent planners, such as BLACKBOX [Kautz and Selman, 1998], IPP [Koehler *et al.*, 1997], and FF [Hoffmann, 2001] among many others, were built for solving general planning tasks given by an initial state, goal state, and a set of plan operators [Vallati *et al.*, 2015]. Though these planners have greatly enlarged the set of feasible planning tasks, one major shortcoming of these planning systems is that they may do well on one problem domain but poorly on another, which has prevented a wider use of AI planning systems. This situation is in contrast to the development of SAT/SMT solvers, which also tackle a combinatorial search task, but have found wide applicability in, for example, hardware and software verification [Järvisalo *et al.*, 2012]. An alternative approach to general AI planning is to develop domain-specialized solvers, e.g., Sokolution for solving Sokoban planning problems, as discussed below. The specialized solvers utilize handcrafted domain-specific knowledge to prune the search space. Clearly, an effective domain-independent approach is preferable. Our learning framework presented here provides a path towards such domain independence. In particular, we will use a machine learning framework to automatically uncover domain-specific problem structure during the solution process.

Recent advances in the deep learning community inspired methods of augmenting search with deep neural networks using deep reinforcement learning (RL). In the game domain, AlphaGo [Silver *et al.*, 2016] as the first Go program to beat professional players in 2016 and its more general and newest version AlphaZero [Silver *et al.*, 2017] achieved a higher Elo rating and dominated the state-of-the-art Chess program Stockfish. One key question about the success of deep RL in these combinatorial (logical) domains is *whether the game setting is a required component for success.* RL requires a reward signal. In a game setting, this signal comes from the ultimate win/loss result from playout. For a game, we can train the deep nets in a self-play approach. In such an approach, the initial deep net starts off playing at a very low level (essentially random play). But in self-play against an equally weak player (using a copy of the trained network), the system will see a mixture of wins and losses and thus gets a useful reward signal about the utility of states. The system can then slowly improve its level of play through repeated rounds of self-play.

The core challenge in a single-agent setting, such as AI planning, where we want to solve unsolved problem instances is: *how do we get any positive reward signal during training?* This is because a positive feedback signal would require a valid plan to the goal but that is exactly what we are looking for. In fact, the problem instances we will solve here require very subtle chains of several dozens to even hundreds of steps. A random exploration will never "accidentally" encounter a successful chain. Our solution is to devise a series of training instances (a "curriculum") that slowly builds up to the full, previously unsolved problem instance. We will see below how such an approach is general and surprisingly effective. Curriculum based training has earlier been proposed

in [Bengio *et al.*, 2009] as a strategy for partitioning training data for incremental training of hard concepts. A novel aspect of our setting is that at each level of our curriculum training, we use what was learned at the previous level to obtain new training data to reach the next level.

Given the PSPACE-completeness of many interesting planning tasks, it is widely assumed (unless P = PSPACE) that developing a solver capable of solving effectively any arbitrary instance is infeasible. Moreover, due to the significant overhead of training deep neural networks, we do not aim to compete on running time with finely tuned specialized solvers on small problem instances. In this work, we therefore focus on planning instances that are right beyond the reach of current state-of-the-art specialized solvers. We will show for the first time how such instances can be tackled successfully within a deep RL framework. Specifically, we will show AI planning instances on which our deep learning strategy outperforms the best previous combinatorial search methods. We will also provide insights about what problem structure deep nets capture during the learning process.

We selected Sokoban planning as our AI planning task because of its extreme difficulty for AI planners [Fern *et al.*, 2011] [Lipovetzky, 2013]. Moreover, Sokoban instances have a regular 2-D input shape that is well-suited for convolutional neural networks. Such 2-D structure can also be found in many other AI planning that involve scheduling and transportation style problems. However, Sokoban is much more challenging in computational terms. Sokoban is a single-player game, created in 1981, in which, given a set of boxes and equal number of goal locations, a player needs to push all boxes to goal squares without crossing walls and boxes. Figure 1 shows a typical instance. The player can only move horizontally or vertically onto empty squares. Despite its apparent conceptual simplicity, it quickly became clear that one could create very hard instances with highly intricate and long solutions (if solvable at all). Analyzing the computational complexity of Sokoban is non-trivial but the question was finally resolved by Culberson in 1997, who proved the problem to be PSPACE-complete [Culberson, 1997] [Hearn and Demaine, 2005]. We will show below that the harder Sokoban instances lie far beyond general AI planners but also quickly are beyond the reach of specialized Sokoban solvers. All modern state-of-the-art solvers are based on a combinatorial search framework augmented with intricate handcrafted pruning rules and dead-end detection techniques.

Our framework learns and solves a single hard Sokoban instance at a time. This is an important choice in our setting. We want *the deep net to uncover the underlying structure of the combinatorial space* that is directly relevant to the hard — previously unsolved — instance under consideration. This approach mimics conflict-driven clause learning (CDCL) [Marques-Silva and Sakallah, 1999] for solving Boolean satisfiability problem (SAT). In SAT solving, the clauses are learned *during the processing of a single instance.* In this setting, the learned clauses are optimally relevant to the problem instance at hand. Another potential advantage of our framework is that all the parameters of the deep neural network are focused on the layout of the given input instance and its corresponding search space. Though some general knowledge about Sokoban, e.g., that pushing a box to a corner leads to a dead-end state, can be learned from one instance and generalized to others, we show that our training setup can also discover this kind of knowledge efficiently and generalize well across its search space. In addition, the deep learning framework can now uncover very specialized problem structure that helps tailor the search for the solution to the specific problem instance at hand. Examples of such structure can be a certain placement of a subset of boxes from which the goal state cannot be reached. The search mechanism can now eliminate any exploration action sequences that lead to such a placement. This is analogous to the pruning provided by learned clauses in SAT solvers. The learned clauses are specific to the SAT instance under consideration.

As we discussed above, we need to devise a way to obtain a proper training signal for solving AI planning problems. Since the input instance might be extremely hard and therefore cannot directly provide any positive reward signal, we incorporate the idea of *curriculum learning* and construct simpler subcases derived from the original challenge problem. In our Sokoban domain, a natural choice is to randomly select smaller subsets of initial boxes and goal squares while leaving all walls unchanged. In particular, our learning procedure starts from exploring 2-box subcases and gradually increases the number of boxes after the success rate of finding a solution increases to a certain threshold. We show that knowledge learned from subcases with smaller numbers of boxes can generalize successfully to subcases with larger numbers of boxes.

Solving the Sokoban planning task is a combinatorial search problem and we will utilize AlphaZero-style Monte Carlo tree search in reinforcement learning for exploring the search space more efficiently. The only domain knowledge we use during learning is computing valid pushes from a state and building the state transition table, and the input of the neural network is the current raw 2-D board state. Intricate handcrafted techniques in modern solvers like dead-end detection are not used.

Our experiments reveal that our curriculum-driven deep reinforcement learning framework can surpass traditional specialized solvers for a large set of instances from benchmark datasets such as XSokoban and Sasquatch. The deep network helps the Monte Carlo tree search explore the search space more effectively and offers significant generalization to unseen states. In addition, the growth of running time when the complexity of the instances in the curriculum increases is near polynomial instead of exponential.

We will also provide a number of other insights into the learning process. Of particular interest is the observation that when training the deep net on harder tasks in the curriculum, its performance on easier instances degrades. This form of "catastrophic forgetting" makes the stronger networks less robust, even when better at solving harder instances. It would be an interesting research direction to devise a curriculum-driven approach that does not show degradation on easier tasks while still reaching maximal effectiveness on the original problem.

2 Related Work

Search in combinatorial domains has been studied extensively in AI, in areas such as planning, decision making, and reasoning. For NP-complete tasks, successful SAT solvers WalkSAT [Selman *et al.*, 1992] and the CDCL framework [Marques-Silva and Sakallah, 1999] have been built to efficiently uncover structure of the input problem and demonstrate near polynomial scaling on many industrial SAT domains. The key insight of their success is the ability of the algorithm to learn problem invariants and reshape the search space by avoiding entering subtrees which do not contain a solution. Most planning tasks are harder than SAT and usually are at least PSPACE-complete. Graphplan [Blum and Furst, 1997] and FF are general planners accepting formal languages such as PDDL [Fox and Long, 2003]. However, as reported in [Welle, 2003], a major shortcoming of these general planners is that they may do well on one problem domain but poorly on another.

The enhancement of planning with learning [Fern *et al.*, 2011] has been investigated extensively in the past. Directed by the current goal, [Abel *et al.*, 2015] prune away irrelevant actions. In each state, [Rosman and Ramamoorthy, 2012] exploit the usefulness of each action by learning action priors. For Sokoban-specialized solving, modern solvers utilize intricate domain-dependent techniques such as subtle dead-end detection, duplicate positions pruning, lower bound calculation, and no influence move detection [Junghanns and Schaeffer, 2001]. While all of these techniques offer efficiency improvements over general planning for Sokoban, good representations of states, tight heuristic functions as well as dead-end detection methods are handcrafted, which requires a careful inspection of domain structure and heavy utilization of domain knowledge.

In recent years, deep neural networks have achieved promising results in many domains. The most exciting result in the combinatorial domain is AlphaZero which utilizes deep reinforcement learning to automatically discover domain structure of two-player games like Chess and Go. Key to its success is the self-play learning strategy, which starts with two weak players and gradually collects useful learning signals by self-play and gradually improves the ability of the players. Previously, it was not clear how to develop such "curriculum-driven" strategy in the planning domain. Because unlike in the game domain, where learning signals (wins/losses) are available for any pair of players of roughly equal strengths, including very weak and random players, in the planning domain, the agent will initially fail to reach the goal state at every attempt and therefore cannot bootstrap its learning process.

Deep neural networks have also been used to help tackle Sokoban problems. [Weber *et al.*, 2017] augment deep reinforcement learning with an imagination component, and [Groshev *et al.*, 2018] use imitation learning to learn from successful Sokoban plays and generalize reactive policies to unseen instances. However, their performance is nowhere close to state-of-the-art specialized Sokoban solvers.

Our approach offers two major advantages over prior approaches: (1) our approach solves hard benchmark instances that are out of reach of specialized Sokoban solvers; (2) no domain-specific knowledge is needed during learning and no extra data, e.g., manually provided solutions, are required. Our idea of training on similar, but easier subcases out from the original instance can be adapted to other planning domains.

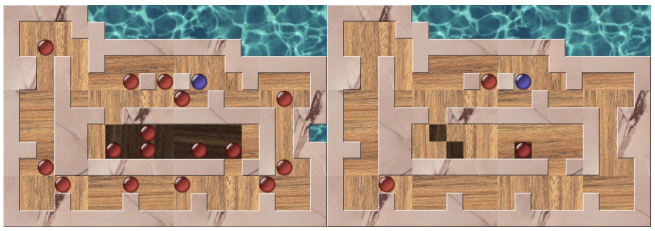

Figure 1: The instance XSokoban_29 (left panel) and one of its 3-box subcase (right panel). The blue circle is the location of the player (or "pusher"), red circles are boxes, and cells with dark background are goal squares. Light colored squares form walls. The player has to push the boxes onto goal squares.

3 Formal Framework

3.1 Model

Given a Sokoban instance $\mathcal{I}$, the preprocessing phase computes the set of all possible pushes $\mathcal{A}$. A deep neural network $(\boldsymbol{p}, v) = f_\theta(s)$ with parameters θ takes the board state s as input and outputs a vector of action probability $\boldsymbol{p}$ with components $\boldsymbol{p}_a = \Pr(a|s)$ for each push action $a \in \mathcal{A}$, and a scalar value v indicating the estimated number of remaining steps to the goal from state s. The left figure of Figure 1 shows the original instance XSokoban_29 from the benchmark dataset XSokoban. The input to the network is a $6 \times H \times W$ image stack consisting of 6 features planes while H and W are the height and width of the corresponding Sokoban instance. Feature planes represent walls, empty goal squares, boxes on empty squares, boxes on goal squares, player-reachable cells, and player-reachable cells on goal squares respectively.

The effort of solving a Sokoban instance can be divided into two parts:

1. The player moves to the correct position adjacent to a box for pushing.
2. The player pushes the box.

In our experiment, we use the set of valid pushes instead of valid moves as the action set since the number of pushes in a solution is significantly smaller than the number of moves. (One move is moving one square over for the player (the pusher).) In other words, the plan length is generally far shorter in terms of number of pushes vs. number of moves. To model the set of valid pushes at each state requires keeping track of the reachable cells that are next to blocks for the player. Illegal pushes are masked out by setting their probabilities to zero, and re-normalising the probabilities for remaining moves.

For each instance $\mathcal{I}$, we set a maximum allowable pushes or "steps" $\mathcal{I}_{\max}$ during the learning phase. $\mathcal{I}_{\max}$ indicates the maximum number of steps of a single plan that the algorithm is allowed to explore during learning. The model will

be forced to stop after $\mathcal{I}_{\max}$ pushes. Setting such a threshold can help avoid infinite meaningless loops when exploring. The remaining-step estimator v is also normalized to the interval $[0, 1]$ to fit better into the neural network framework. Notice that if $\mathcal{I}_{\max}$ is set smaller than the length of the shortest solution plan then the model will never find any solution. In our experiments, we start with $\mathcal{I}_{\max} = 500$ and double it whenever learning fails after a long run.

The learning framework consists of multiple iterations, and each iteration contains three parts:

1. Initial board generation phase: we randomly generate 500 initial boards according to our curriculum-driven strategy described in subsection 3.2.
2. Exploration phase: the model searches for solutions (plans) for these boards with Monte Carlo tree search (MCTS) driven by the policy/value network trained so far. For more details see subsection 3.3.
3. Training phase: we train the neural work with learning signals collected from the exploration phase. This part will be further illustrated in subsection 3.4.

3.2 Curriculum-driven Strategy

For hard Sokoban instances the deep RL setup may fail to find any solution and thus gets no useful training signal. Our curriculum strategy is based on two insights: (1) construct simpler subcases that are more likely to be solved by the current trained model; (2) the constructed subcases should share similar structure information with the original instance to enhance knowledge generalization from a series of subcases to the original problem instance.

Learning starts by choosing a small subset of initial boxes and goal squares to form a subcase for exploration and training while leaving wall locations unchanged. Figure 1 (right) shows one such example. Three boxes and goal squares are randomly selected from the initial ones. The resulting subcase requires a much simpler plan. Specifically, assume the input Sokoban instance has n boxes and goal squares, in each iteration we randomly select $m \leq n$ boxes and goal squares and gradually increase m after a certain level of performance has been reached at each level. Compared with the original problem, solutions of the subcases will be shorter and, most importantly, easier to find with MCTS and the deep net trained so far. By solving a collection of m box subcases for each value of m, we effectively train a distance function and action model that can handle m box subproblems for a range of initial and goal placements. This level of generality is important because we do not know in advance on which goal square, any particular box from the initial state will end up. Moreover, because we start with a 2 box subcase, MCTS with a randomly initialized deep net can still find a solution path, and thus a positive reward signal. By slowly increasing the subcase size (and difficulty), deep RL can continue to obtain a positive reward signal and incrementally improve the trained net to handle increasingly complex scenarios, ultimately leading to a solution to to the original instance, when $m = n$.

In the experiment section we will show that it is necessary for the model to learn to a certain accuracy rate on m-box subcases before jumping to $(m + 1)$-box subcases. Specifically, if the model jumps to $(m + 1)$-box scenarios before it reaches a high success rate on m-box scenarios, one potential danger is that the performance of learning will abruptly degrade and the model might be no longer able to find any solution for $(m + 1)$-box subcases.

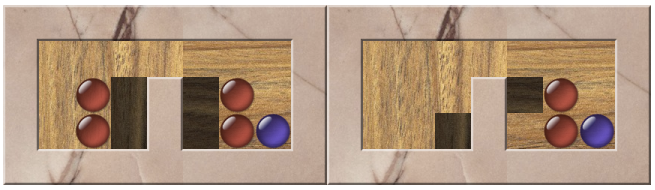

Figure 2: One example showing that subcases are not necessarily solvable even though the original instance has a solution.

To decide on when to increase m, one possible measure to use would be the solution rate reached at that level. However, somewhat counterintuitively, even if the original problem instance is solvable, certain subcases may not have solutions. This is due to a hidden complexity of the Sokoban domain: we know that the boxes need to reach the goal squares but we don't know exactly which box should go to which goal square. So, even though our subproblems use a strict subset of the boxes and goal squares, we may accidentally generate an unsolvable subproblem. Figure 2 gives an example. In this case, the probability of generating a solvable 2-box subcase is only $\frac{1}{2}$, since we need to guarantee that the number of boxes and goal squares be the same in each room. In general, it is difficulty to compute the probability that a random subcase is solvable. Therefore, using the success rate at a certain level is not a robust criterion. We use an alternative way to decide when to increment m. Specifically, we increment m when the success rate of finding a solution has not improved over a certain number of iterations. We use 5 iterations in our experiments. Our experiments show that this strategy works well in practice.

3.3 Monte Carlo Tree Search

We search for solutions (plans) of the Sokoban m-box subinstances ($m \leq n$) using an AlphaZero-style Monte Carlo tree search (MTCS) guided by the deep net trained so far. As we will see, MTCS works well but other search techniques may also be worth exploring in future work. In MTCS, at each state s, we compute $(\boldsymbol{p}, v) = f_\theta(s)$ and create a root node R which contains the state s. Multiple Monte Carlo rounds will be performed from R to calculate the best move in s. Each round consists of three components:

- Selection: start from the root node R, which contains the state s, and select successive child nodes which maximize a utility function until a leaf node L, the goal, or a dead-end is reached. A leaf node is any node that has never been evaluated by the neural network before. If the goal or a dead-end is reached then we jump to the backpropagation phase otherwise the expansion phase.
- Expansion: compute the set of all valid pushes $\mathcal{A}$ from the state s_L of the node L. Unlike traditional MCTS followed by a roll-out which simulates multiple random

plays, we evaluate $(\boldsymbol{p}, v) = f_\theta(s_L)$ with the neural network and use v as the estimated evaluation for the backpropagation phase.

- Backpropagation: use v of s_L to update information of the nodes on the path from R to L. Set v to 0 if s_L is the goal state or 1 if s_L is a dead-end. Assume the state-observation-action trajectory from R to L is $s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_l} s_l = s_L$ where l is the length of the trajectory, we update $Q_{\text{new}}(s_i, a_{i+1})$ to

$$\frac{Q(s_i, a_{i+1}) \cdot N(s_i, a_{i+1}) + \min(v + \frac{l-i}{\mathcal{I}_{\max}}, 1)}{N(s_i, a_{i+1}) + 1},$$

where $Q(s, a)$ is the mean action value averaged from previous rounds and $N(s, a)$ is the visit count.

To select child nodes, we choose $a_t = \text{argmax}_a Q(s_{t-1}, a) + U(s_{t-1}, a)$ using a variant of the PUCT algorithm where

$$U(s, a) = \text{cput} \cdot \frac{\sqrt{1 + \sum_b N(s, b)}}{1 + N(s, a)} \cdot \boldsymbol{p}_a,$$

and cput is a constant balancing exploration and exploitation.

After 1600 rounds have been performed, we choose a move either greedily or proportionally with respect to the visit count at the root state s. This procedure proceeds until a dead-end or the goal is reached, or the maximum allowable pushes $\mathcal{I}_{\max}$ have been performed. Notice we don't utilize any advanced dead-end detection algorithm used in previous modern solvers. Instead, we only detect dead-ends when no valid pushes from the state are available, e.g., all boxes are pushed into corners and are no longer movable.

To construct learning signals for the training phase, we collect all states on paths explored by the Monte Carlo tree search, and use the probability proportional to the visit count as the improved probability distribution $\boldsymbol{\pi}$ for training. For value prediction, if the leaf node is the goal state then we use the distance to the leaf node as the new label u. Otherwise, either a loop or a dead-end is reached and we set u to 1 for all nodes on the path.

3.4 Training

We use 5 GPUs to train the network and each iteration contains 1000 epochs with mini-batch 160 in total. Unlike [Mnih *et al.*, 2013] and [Silver *et al.*, 2017] who maintain an extra memory pool to save training episodes, we directly train on data collected from the current iteration. Specifically, the network parameters θ are adjusted by gradient descent on the loss function that sums over a mean-squared loss and a cross-entropy loss

$$l = (u - v)^2 - \boldsymbol{\pi} \log(\boldsymbol{p}) + c \cdot \|\theta\|^2,$$

where c is the constant to control the impact of weight decay. After the training phase, new parameters of the network are used to guide the Monte Carlo tree search in the next iteration.

For this paper, we did not perform a detailed hyper parameter study to select the best network structure for our problem setting. We used vanilla ResNet [He *et al.*, 2016] with 8 residual blocks as the network setting for all experiments.

Sokoban instance	Our method	Sokolution	FF
XSokoban_29	9.1h	Failed	Failed
Sasquatch_29	Failed	Failed	Failed
Sasquatch_30	Failed	Failed	Failed
Sasquatch3_18	14.9h	Failed	Failed
Sasquatch7_48	23.4h	1.0h	Failed
Grigr2001_2	22.1h	Failed	Failed

Table 1: Performance comparison. Sokoban instances are selected from standard datasets and are marked as "Solved by none". Time limit for all solvers are extended to 24 hours if the option is available. All solvers are running on the same CPU cores while our method utilizes additional 5 GPUs. The conversion from Sokoban into STRIPS format is shown in [Welle, 2003].

It is an indication of the promise of our general framework that we already obtained good results with standard ResNet. The overall performance can likely be further improved with careful hyper parameter tuning.

4 Experiments

Here we report our experiments on XSokoban, the de facto standard test suite in the academic literature on Sokoban solver programming, as well as other large test suites[1]. We pick instances that are marked as "Solved by none of the four modern solvers". The time limit for the statistics of previous benchmark is usually 10 minutes. We extend the time limit to 24 hours if the option is available and retest all solvers on these hard instances. We do the test on our method, state-of-the-art Sokoban-specialized solver Sokolution, and domain-independent general planner FF. Table 1 shows the performance of each solver on the selected instances.

4.1 Scaling Comparison

Since our framework utilizes extra GPU resources, to gain more insights about the difference between the ways that our framework and traditional search-based algorithms handle hard instances, we evaluate the scaling performance of each solver on subcases with gradually increasing difficulty. We use XSokoban_29, which contains 16 boxes, for illustration purposes, as shown in Figure 3. The running time for FF and Sokolution clearly show exponential growth and FF can no longer solve any m-box subcase for $m \geq 8$. Sokolution significantly outperforms our method for small-box subcases. We believe this is mainly due to the heavy overhead of the training of neural networks. As the size of subcases increases, our method shines both in running time and scaling performance. Note that our method spends almost no extra time jumping from 15-box subcases to the original 16-box instance. That's because the prediction of the network learned from 15-box subcases is highly accurate on the 16-box instance and the model is already capable of solving the original problem.

[1] Sokoban datasets available at http://sokobano.de/wiki/index.php?title=Solver_Statistics

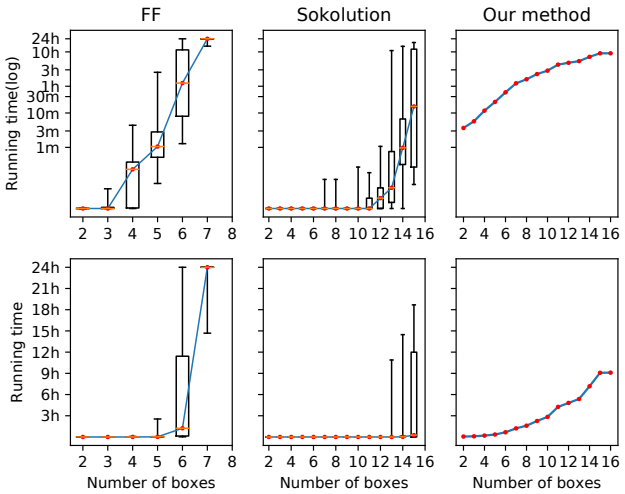

Figure 3: Scaling performance comparison between FF, Sokolution and our method on instance XSokoban_29. For m-box subcases, we randomly generated 19 subcases for FF and Sokolution, and plot their boxplot according to running time. The running time in the top three figures is in logarithmic scale while the bottom three are in linear scale. For our method, we plot the total time needed for the algorithm to achieve 95% success rate for each m.

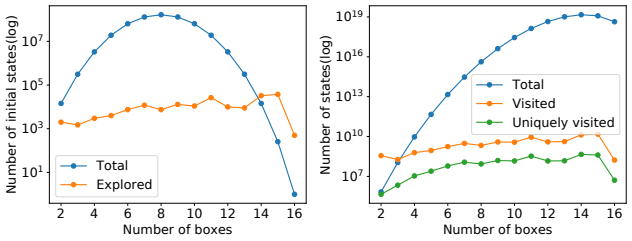

Figure 4: The state statistics during learning. The left figure shows the total number of initial states and the number of explored initial states by the Monte Carlo tree search. The right figure shows the total number of all possible states, all states, and unique states explored by the Monte Carlo tree search. The y-axis of both figures is plotted in logarithmic scale.

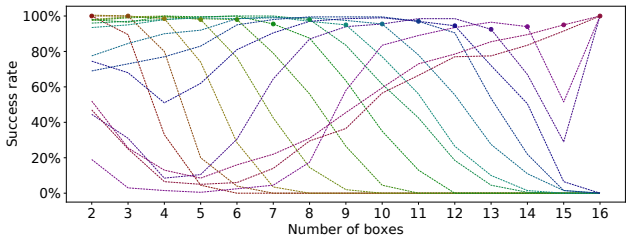

Figure 5: Success rate of different models on different subcases. For each $n \in [2, 16]$, we extracted the model $\mathcal{N}_n$ when the algorithm reached 95% success rate for the first time on n-box subcases. Each curve represents a model $\mathcal{N}$, and for each curve there is a corresponding circle on it whose x-coordinate n indicates the model $\mathcal{N}_n$. The x-axis represents each m-box subcase, and for each model $\mathcal{N}_n$ we randomly generated 500 m-box subcases and tested its success rate on these subcases.

4.2 Exploration Efficiency

We now show that the network can efficiently extract knowledge when exploring and generalize to unseen states. For the same XSokoban_29 instance, we plot the state efficiency by comparing the number of seen states during the exploration phase and the total number of possible states in Figure 4. In the left figure, for each m, the total number of initial states is $\binom{16}{m}^2$. The number of explored states almost remains at the same magnitude as m increases. This implies the capability of the neural network to efficiently extract structure information of the combinatorial search space and generalize its knowledge to unseen search spaces. The right figure shows the comparison between total possible board states and those explored by the Monte Carlo tree search.

Also notice that for subcases with $m \leq 3$, the model needs to see almost all possible board states before jumping to next stages. This implies generalization does not start for small-box subcases and the model needs to explore every possible combination of board states to understand the underlying structure. As m increases and the combinatorial space grows, generalization starts to shine and help the Monte Carlo tree search stay around the most promising search space.

4.3 Forgetting during Curriculum Learning

One surprising phenomenon in curriculum-driven learning is that the networks may start to forget previously learned structure as the learning proceeds. As seen in Figure 5, as the number of boxes increases, the success rate of small-box subcases gradually decreases. Specifically, we see that the curves trained on higher numbers of boxes drop off to the left, i.e., the performance on cases with fewer boxes becomes worse. On the other hand, the ability to solve increasingly hard cases (more boxes) through the curriculum-driven training shows that knowledge learned from m-box subcases can be useful in finding solutions to m'-box subcases where $m' > m$. The curve shows that the model learned from 13-box subcases is already capable of solving the original instance with 16 boxes. This implies that an ensemble of knowledge from small-box subcases can work together to provide enough guidance for finding a solution of the original, unsolved problem instance.

Note that catastrophic forgetting has been previously observed in the context of training deep neural networks. Specifically, deep nets can gradually or abruptly forget previously learned knowledge upon learning new information. This is an important issue to consider because humans typically do not show such catastrophic forgetting when increasing their proficiency on a task. For example, a chess player reaching grand master level will not suddenly start lose to a beginner player. An interesting research challenge is to develop training curricula that prevent catastrophic forgetting for deep RL.

4.4 Knowledge Extraction from the Network

We now show how accurate both value prediction and probability prediction are compared with ground truth provided

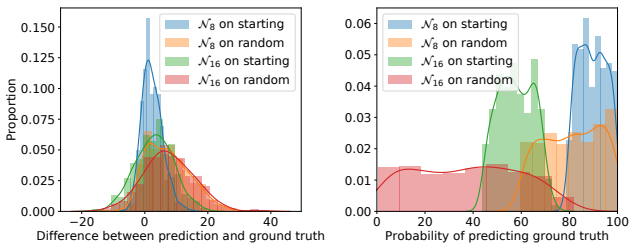

Figure 6: The accuracy of network prediction compared with ground truth. The left figure shows the difference between value prediction which indicates the remaining steps to the goal and the ground truth. The right figures shows the confidence of the network about the ground truth action. We test both on $\mathcal{N}_8$ and $\mathcal{N}_{16}$ that are trained after 8-box subcases and the original problem.

by optimal Sokoban solvers. We also want to test whether the network can generalize to absolutely unseen board states or just learns well on states that are frequently visited by the Monte Carlo tree search. For this experiment, we test on m-box subcases of XSokoban_29 where $m = 8$. We randomly select 500 initial 8-box subcases and do some random pushes on them to generate the set of starting states which are supposed to be frequently seen by the Monte Carlo tree search. And we also generate 500 board states whose boxes are randomly selected from all possible locations of the board. These states are supposed to be hardly explored. All test states are guaranteed to be solvable and dead-end free.

As shown in figure 6, we see that the utility function captures the distance to the goal for 8-box subcases surprisingly well for states where the 8 boxes are close to the initial 16-box setup. When we consider subcases with the 8 boxes initially placed randomly, we see the utility function degrade. So the learning does focus on states close to the states that may occur as legal intermediate states which are heavily explored and exploited by reinforcement learning.

When we consider the 16-box learned network, we see an analogous phenomenon but overall less accurate in terms of both utility and policy compared with 8-box scenarios. In fact, the policy for 8-box subcases for randomly placed boxes becomes worst, though still way better than random policy. This means that the Monte Carlo tree search is no longer focused enough to find the goal state in 8-box scenarios. This explains the forgetting curve as discussed earlier.

5 Conclusion

We presented a framework based on deep RL for solving hard combinatorial planning problems in the domain of Sokoban. A key challenge in the application of deep RL in a single agent setting is the lack of a positive reinforcement signal since our goal is to solve previously unsolved instances that are beyond existing combinatorial search methods. We showed how a curriculum-driven deep RL approach can successfully address this challenge. By devising a sequence of increasingly complex sub problems, each derived from the original instance, we can incrementally learn an approximate distance to goal function that can guide MCTS to solving the original problem instance.

We showed the effectiveness of our learning based planning strategy by solving hard Sokoban instances that are out of reach of previous search-based solution techniques, including methods specialized for Sokoban. We could uncover plans with over two hundred actions, where almost any deviation from the plan would lead to an unrecoverable state. Since Sokoban is one of the hardest challenge domains for current AI planners, this work shows the potential of curriculum-based deep RL for solving hard AI planning tasks. In future work, we hope to extend these techniques to boost theorem proving methods to find intricate mathematical proofs consisting of long sequences of inference steps to assist in mathematical discovery.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by the Center for Human-Compatible AI, an NSF Expeditions in Computing award (CCF-1522054), an AFSOR award (FA9550-17-1-0292), and an ARO DURIP award (W911NF-17-1-0187) for the compute cluster used in our experimental work.

References

[Abel *et al.*, 2015] David Abel, David Ellis Hershkowitz, Gabriel Barth-Maron, Stephen Brawner, Kevin O'Farrell, James MacGlashan, and Stefanie Tellex. Goal-based action priors. In *Twenty-Fifth International Conference on Automated Planning and Scheduling*, 2015.

[Bengio *et al.*, 2009] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[Blum and Furst, 1997] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[Bylander, 1994] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[Culberson, 1997] Joseph Culberson. Sokoban is pspace-complete. *University of Alberta, Technical Report, TRID-ID TR97-02*, 1997.

[Fern *et al.*, 2011] Alan Fern, Roni Khardon, and Prasad Tadepalli. The first learning track of the international planning competition. *Machine Learning*, 84(1-2):81–107, 2011.

[Fox and Long, 2003] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[Gödel, 1931] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

[Groshev *et al.*, 2018] Edward Groshev, Aviv Tamar, Maxwell Goldstein, Siddharth Srivastava, and Pieter

Abbeel. Learning generalized reactive policies using deep neural networks. In *2018 AAAI Spring Symposium Series*, 2018.

[He *et al.*, 2016] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[Hearn and Demaine, 2005] Robert A Hearn and Erik D Demaine. Pspace-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, 2005.

[Hoffmann, 2001] Jörg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22(3):57–57, 2001.

[Järvisalo *et al.*, 2012] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international sat solver competitions. *Ai Magazine*, 33(1):89–92, 2012.

[Junghanns and Schaeffer, 2001] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Improving the search with relevance cuts. *Theoretical Computer Science*, 252(1-2):151–175, 2001.

[Kautz and Selman, 1998] Henry Kautz and Bart Selman. Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS98 Workshop on Planning as Combinatorial Search*, volume 58260, pages 58–60, 1998.

[Koehler *et al.*, 1997] Jana Koehler, Bernhard Nebel, Jörg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an adl subset. In *European Conference on Planning*, pages 273–285. Springer, 1997.

[Lipovetzky, 2013] Nir Lipovetzky. *Structure and inference in classical planning*. PhD thesis, Universitat Pompeu Fabra, 2013.

[Marques-Silva and Sakallah, 1999] Joao P Marques-Silva and Karem A Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[Mnih *et al.*, 2013] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[Rosman and Ramamoorthy, 2012] Benjamin Rosman and Subramanian Ramamoorthy. What good are actions? accelerating learning using learned action priors. In *2012 IEEE International Conference on Development and Learning and Epigenetic Robotics (ICDL)*, pages 1–6. IEEE, 2012.

[Selman *et al.*, 1992] Bart Selman, Hector J Levesque, David G Mitchell, et al. A new method for solving hard satisfiability problems. In *Aaai*, volume 92, pages 440–446. Citeseer, 1992.

[Silver *et al.*, 2016] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

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

[Vallati *et al.*, 2015] Mauro Vallati, Lukas Chrpa, Marek Grześ, Thomas Leo McCluskey, Mark Roberts, Scott Sanner, et al. The 2014 international planning competition: Progress and trends. *Ai Magazine*, 36(3):90–98, 2015.

[Weber *et al.*, 2017] Théophane Weber, Sébastien Racanière, David P Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomenech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, et al. Imagination-augmented agents for deep reinforcement learning. *arXiv preprint arXiv:1707.06203*, 2017.

[Welle, 2003] James Welle. A comparison of parallelism in current ai planning systems. 2003.