Structure and randomness in planning and reinforcement learning

Piotr Januszewski
University of Warsaw & Gdansk University of Technology
piotr.januszewski@pg.edu.pl

Konrad Czechowski
University of Warsaw
k.czechowski@mimuw.edu.pl

Piotr Kozakowski
University of Warsaw
p.kozakowski@mimuw.edu.pl

Łukasz Kuciński
Polish Academy of Sciences
lkucinski@impan.pl

Piotr Miłoś
Polish Academy of Sciences
pmilos@impan.pl

Abstract

Planning in large state spaces inevitably needs to balance depth and breadth of the search. It has a crucial impact on their performance and most planners manage this interplay implicitly. We present a novel method *Shoot Tree Search (STS)*, which makes it possible to control this trade-off more explicitly. Our algorithm can be understood as an interpolation between two celebrated search mechanisms: MCTS and random shooting. It also lets the user control the bias-variance trade-off, akin to $TD(n)$, but in the tree search context.

In experiments on challenging domains, we show that STS can get the best of both worlds consistently achieving higher scores.

1 Introduction

Classically, reinforcement learning is split into model-free and model-based methods. Each of these approaches has its strengths and weaknesses: the former often achieves state-of-the-art performance, while the latter holds the promise of better sample efficiency and adaptability to new situations. Interestingly, in both paradigms, there exists a non-trivial interplay between structure and randomness. In the model-free approach, Temporal Difference (TD) prediction leverages the structure of function approximators, while Monte Carlo (MC) prediction relies on random rollouts.

Model-based methods often employ planning, which counterfactually evaluates future scenarios. The design of a planner can lean either towards randomness, with random rollouts used for state evaluation (e.g. random shooting), or towards structure, where a data-structure, typically a tree or a graph, forms a backbone of the search (e.g. Monte Carlo Tree Search). Planning is a powerful concept and an important policy improvement mechanism. However, in many interesting problems, the search space is prohibitively large and cannot be exhaustively explored. Consequently, it is critical to balance the depth and breadth of the search in order to stay within a feasible computational budget. This dilemma is ubiquitous, though often not explicit.

The aim of our work is twofold. First, we present a novel method: Shoot Tree Search (STS). The development of the algorithm was motivated by the aforementioned observations concerning structure, randomness, and dilemma between breadth and depth of the search. It lets the user control the depth and breadth of the search more explicitly and can be viewed as a bias-variance control method. STS itself can be understood as an interpolation between MCTS and random shooting. We show experimentally that, on a diverse set of environments, STS can get the best of both worlds. We also provide some toy environments, to get an insight into why STS can be expected to perform well. The

Preprint. Under review.

critical element of STS, *multi-step expansion*, can be easily implemented on top of many algorithms from the MCTS family. As such, it can be viewed as one of the extensions in the MCTS toolbox.

The second aim of the paper is to analyze various improvements to planning algorithms and test them experimentally. This, we believe, is of interest in its own right. The testing was performed on the Sokoban and Google Research Football environments. Sokoban is a challenging combinatorial puzzle proposed to be a testbed for planning methods by Racanière et al. [26]. Google Research Football is, an advanced, physics-based simulator of football, introduced recently in Kurach et al. [19]. It has been designed to offer a diverse set of challenges for testing RL algorithms.

The rest of the paper is organized as follows. In the next section, we discuss the background and related works. Further, we present details of our method. Section 4 is devoted to experimental results.

2 Background and related work

The introduction to reinforcement learning can be found in Sutton and Barto [33]. In contemporary research, the line between model-free and model-based methods is often blurred. An early example is Guo et al. [15], where MCTS plays the role of an 'expert' in DAgger (Ross and Bagnell [27]), a policy learning algorithm. In the series of papers Silver et al. [31, 32], culminating in AlphaZero, the authors developed a system combining elements of model-based and model-free methods that master the game of Go (and others). Similar ideas were also studied in Anthony et al. [3]. In Miłoś et al. [21], planning and model-free learning were brought together to solve combinatorial environments. Schrittwieser [29] successfully integrated model learning with planning in the latent space. A recent paper, Hamrick et al. [16], suggests further integration model-free and model-based methods via utilizing internal planner information to calculate more accurate estimates of the Q-function.

Searching and planning algorithms are deeply rooted in classical computer science and classical AI, see e.g. Cormen et al. [7] and Russell and Norvig [28]. Traditional heuristic algorithms such as A* (Hart et al. [17]) or GBFS (Doran and Michie [10]) are widely used. The Monte Carlo Tree Search algorithm, which combines heuristic search with learning, led to breakthroughs in the field, see Browne et al. [4] for an extensive survey. Similarly, Orseau et al. [24] bases on the classical BFS to build a heuristic search mechanism with theoretical guarantees. In Agostinelli et al. [2] the authors utilise the value-function to improve upon the A* algorithm and solve Rubik's cube.

Monte Carlo rollouts are known to be a useful way of approximating the value of a state-action pair [1]. Approaches in which the actions of a rollout are uniformly sampled are often called flat Monte Carlo. Impressively, Flat Monte Carlo achieved the world champion level in Bridge [12] and Scrabble [30].

Moreover, Monte Carlo rollouts are often used as a part of model predictive control, see Camacho and Alba [5]. As suggested by Chua et al. [6], Nagabandi et al. [22], they offer several advantages, including simplicity, ease of parallelization. At the same time, they reach competitive results to other (more complicated) methods on many important tasks.

Some works aim to compose a planning module into neural network architectures, see e.g., Oh et al. [23], Farquhar et al. [11]. Kaiser et al. [18], recent work on model-based Atari, has shown the possibility of sample efficient reinforcement learning with an explicit visual model. Gu et al. [13] uses model-based methods at the initial phase of training and model-free methods during 'fine-tuning'. Furthermore, there is a body of work that attempts to learn a planning module, see Pascanu et al. [25], Racanière et al. [26], Guez et al. [14].

3 Methods

A Generic Planner, presented in Algorithm 1, gives a unified view on all methods analyzed in the paper: Random Shooting, MCTS and, our novel approach, STS. By a suitable choice of functions SELECT, EXPAND, UPDATE and CHOOSE_ACTION, we can recover each of these methods (see description below).

Typically, a planner is a part of a training process, see Algorithm 2. In a positive feedback loop, the planner improves the quality of data used for training of the value function V_θ and a policy π_ϕ.

Conversely, the policy and value function might further improve planning. Implementation details of Algorithm 2 are provided in Appendix A.1.

Below we give a detailed description of the planning methods considered in the papers.

Algorithm 1 Generic Planner, defines required constants, variables and objects used in further algorithms

Require:	C	planning passes
	H	planning horizon
	γ	discount factor
Use:	$N(s,a)$	visit count
	$W(s,a)$	total action-value
	$Q(s,a)$	mean action-value
	$\mathbf{V}_\theta$	value function
	π_ϕ	policy
	$model$	environment simulator

```
# Initialize N, W, Q to zero
function PLANNER(state)
    for 1 ... C do
        path, leaf ← SELECT(state)
        rollout, leaf ← EXPAND(leaf)
        UPDATE(path, rollout, leaf)
    return CHOOSE_ACTION(state)
```

Algorithm 2 Training loop, additionally requires environment env

```
# Initialize parameters of V_θ, π_φ
# Initialize replay_buffer
repeat
    episode ← COLLECT_EPISODE
    replay_buffer.ADD(episode)
    B ← replay_buffer.BATCH
    Update V_θ, π_φ using B and SGD
until convergence
function COLLECT_EPISODE
    s ← env.RESET
    episode ← []
    repeat
        a ← PLANNER(s)
        s', r ← env.STEP(a)
        episode.APPEND((s, a, r, s'))
        s ← s'
    until episode is done
    return CALCULATE_TARGET(episode)
```

Random Shooting In this section we present two instantiations of Algorithm 1, which use Monte Carlo rollouts to evaluate state-actions pairs: Random Shooting and Bandit Shooting, see Algorithm 3 and Algorithm 4, respectively.

Algorithm 3 Random Shooting Planner

```
function SELECT(state)
    s ← state
    a ~ π_φ(s, ·)
    s', r ← model.STEP(s, a)
    return (s, a, r), s'
function EXPAND(leaf)
    s_0 ← leaf
    rollout ← (s_k, a_k, r_{k+1})_{k=0}^{H-1}
     where s_{k+1}, r_{k+1} ← model.STEP(s_k, a_k)
     and a_k ~ π_φ(s_k, ·)
    return rollout, s_H
function UPDATE(path, rollout, leaf)
    Ĝ ← Σ_{k=1}^{H} γ^{k-1} r_k + γ^H V_θ(leaf)
     where r_k ∈ rollout
    s, a, r ← path
    quality ← r + γ * Ĝ
    W(s,a) ← W(s,a) + quality
    N(s,a) ← N(s,a) + 1
    Q(s,a) ← W(s,a) / N(s,a)
function CHOOSE_ACTION(s)
    return argmax_a Q(s,a)
```

The simplest version of Algorithm 3, the so-called flat Monte Carlo [12, 30], does not use a policy π_ϕ (instead rollouts are uniformly sampled) nor a value function $\mathbf{V}_\theta$ (just truncated sum of rewards $\hat{G} = \sum_{k=1}^{H} \gamma^{k-1} r_k$). Bandit Shooting, presented in Algorithm 4, is a Multi-armed Bandits variant of Random Shooting and uses PUCT [32] rule to improve exploration and thus achieve more reliable evaluations of actions.

Algorithm 4 Bandit Shooting Planner, additionally requires exploration weight c_{puct}

```
function SELECT(state)
    s ← state
    U(s,a) ← sqrt(Σ_{a'} N(s,a')) / (1 + N(s,a))
    a ← argmax_a (Q(s,a) + c_puct π_φ(s,a) U(s,a))
    s', r ← model.STEP(s, a)
    return (s, a, r), s'
function EXPAND(leaf)
    The same as in Algorithm 3.
function UPDATE(path, rollout)
    The same as in Algorithm 3.
function CHOOSE_ACTION(s)
    return argmax_a N(s,a)
```

MCTS Monte Carlo Tree Search (MCTS) is a family of methods, that iteratively and explicitly build a search tree, see Browne et al. [4]. It follows the schema of Algorithm 1. `SELECT` traverses down the tree, according to an in-tree policy, until a leaf is encountered. `EXPAND` grows the tree by adding the leaf's children. The values of these new nodes are estimated, usually with the help of a rollout policy in a similar vein as Random Shooting Planner, or via the value network V_θ (see Silver et al. [31]). Finally, `UPDATE` backpropagates these values from the leaf up the tree. A basic variant of MCTS is presented in Algorithm 5. More details are provided in Appendix A.5.

Algorithm 5 MCTS, additionally uses tree structure $tree$.

function SELECT(`state`)
  $s \leftarrow$ `state`
  `path` $\leftarrow []$
  while s belongs to $tree$ **do**
    $a \leftarrow$ CHOOSE_ACTION(s)
    $s', r \leftarrow tree[s][a]$
    `path`.APPEND((s, a, r))
    $s \leftarrow s'$
  return `path`, s

function EXPAND(`leaf`)
  for $a \in \mathcal{A}$ **do**
    $s', r \leftarrow model$.STEP(`leaf`, a)
    $tree$[`leaf`][a] $\leftarrow (s', r)$
    W(`leaf`, a) $\leftarrow r + \gamma * \mathbf{V}_\theta(s')$
    N(`leaf`, a) $\leftarrow 1$
    Q(`leaf`, a) $\leftarrow W$(`leaf`, a)
  return $[]$, `leaf`

function UPDATE(`path`, `rollout`, `leaf`)
  `quality` $\leftarrow \mathbf{V}_\theta$(`leaf`)
  for $s, a, r \leftarrow$ `reversed`(`path`) **do**
    `quality` $\leftarrow r + \gamma *$ `quality`
    $W(s, a) \leftarrow W(s, a) +$ `quality`
    $N(s, a) \leftarrow N(s, a) + 1$
    $Q(s, a) \leftarrow \frac{W(s,a)}{N(s,a)}$

function CHOOSE_ACTION(s)
  return $\text{argmax}_a\, Q(s, a)$

Shoot Tree Search Shoot Tree Search (STS) extends MCTS in a novel way, by redesigning the expansion phase, see Algorithm 6. Given a leaf and a planning horizon, H, the method expands H consecutive vertices starting from the leaf. Each new node is chosen according to the in-tree policy and is added to the search tree.

Algorithm 6 Shoot Tree Search

function EXPAND(`leaf`)
  $s \leftarrow$ `leaf`
  `rollout` $\leftarrow []$
  for $1 \dots H$ **do**
    MCTS.EXPAND(s)
    $a \leftarrow$ CHOOSE_ACTION(s)
    $s', r \leftarrow tree[s][a]$
    `rollout`.APPEND((s, a, r))
    $s \leftarrow s'$
  return `rollout`, s

function SELECT(`state`)
  `The same as in Algorithm 5.`

function CHOOSE_ACTION(s)
  `The same as in Algorithm 5.`

function UPDATE(`path`, `rollout`, `leaf`)
  $s' \leftarrow$ `leaf`
  $c \leftarrow 1$
  `quality` $\leftarrow 0$
  for $s, a, r \leftarrow$ `reversed`(`path` + `rollout`) **do**
    if $s' \in$ `path` **then**
      `v` $\leftarrow 0$
    else
      `v` $\leftarrow \mathbf{V}_\theta(s')$
      $c \leftarrow c + 1$
    `quality` $\leftarrow c * r + \gamma *$ (`quality` + `v`)
    $W(s, a) \leftarrow W(s, a) +$ `quality`
    $N(s, a) \leftarrow N(s, a) + c$
    $Q(s, a) \leftarrow \frac{W(s,a)}{N(s,a)}$
    $s' \leftarrow s$

STS can be viewed as a sophisticated version of Random Shooting applied to MCTS. In this interpretation, STS interpolates between the two methods. We demonstrate empirically that the change introduced by STS is essential to solving challenging RL domains; see Section 4. We note that $H = 1$ corresponds to MCTS.

Interestingly, in some of our experiments, we identified that the tree traversal performed during `SELECT` was the computational bottleneck. The cost of building the search tree is quadratic with respect to its depth. STS allows to significantly reduce this cost since a single tree traversal adds

not one but H new nodes. To get this computational benefit we tweak `UPDATE` to backpropagate all values from `leaf` and `rollout` in one pass. A more formal analysis of computational gains is presented in Lemma A.6.1.

4 Experiments

We tested the spectrum of algorithms presented in Section 3 on the Sokoban and Google Research Football domains. Those tasks present numerous challenges, which evaluate various properties of planning algorithms. In this work, we assume access to a model (which is used by the planning algorithm). Using learned models is an exciting research topic left for further work. The training details, a list of hyper-parameters and network architectures are presented in appendices A.1, A.2 and A.3 respectively.

We present two thought experiments, where we argue that STS can better handle certain errors in value functions by using the *multi-step expansion* (parametrized by H; recall that $H = 1$ corresponds to MCTS). The errors are inevitable during training and when using function approximators.

First, consider an MDP presented at the top of Figure 1. It showcases the situation when the errors are systematic: in the vicinity of the starting state s_0, the estimates of the value function are biased (for simplicity set to 0 and shown as white vertices), while the values in the area surrounding terminal states are accurate (shown as color vertices). This example is an exaggeration. However, something similar can happen in practice, when information is propagated with TD-like methods or the environment has an "easy" region, which is hard to find. Under these circumstances, STS, given large enough H, will be able to reach accurate values (color vertices) within a few passes. On the contrary, MCTS would explore the whole uncertain area (white vertices) in a breadth-first fashion.

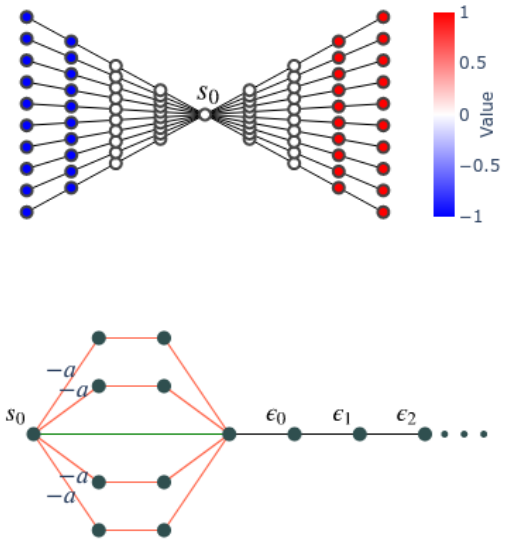

Figure 1: Toy environments. Full size in Appendix A.9

Second, consider an MDP shown in at the bottom of Figure 1. It illustrates the case when the errors are "pseudo-random". In this MDP all rewards are 0 except the marked edges, where they are $-a, a > 0$. The optimal path is highlighted in green and the "decoy" paths are shown in red. The perfect value function is 0 in every state, however we assume that the current noisy value estimates equal to ϵ_i on the "tail" part of the diagram. Let p_H be the probability of entering into the decoy branches. In Lemma A.9.1, we show that $p_1 > p_H$ for $H \geq 2$, and in fact $p_H \to 0$ when $H \to +\infty$ (under mild assumptions). The ratio p_1/p_H, depends on the ratio of a to the noise ϵ_i. In Appendix A.9, we show that p_1/p_H can be as high as 3 for $H = 16$ and quite natural choice of a and ϵ_i; we also present there the formal proof the lemma.

4.1 Sokoban

Sokoban is a well-known combinatorial puzzle, where the agent's goal is to push all boxes (marked as yellow, crossed squares) to the designed spots (marked as squares with a red dot in the middle), see Figure 2. Additionally, to the navigational challenge, Sokoban's difficulty is attributed to

Scenario	C	H	S. rate	N_p	N_t	N_g
av. loops	256	1	95.2%	1224	1224	716
	64	4	96.5%	299	1194	830
	16	16	95.7%	114	1822	1333
	4	64	89%	62	3960	1491
no av. loops	256	1	84.5%	1497	1497	376
	32	8	88.4%	185	1483	409
	2	128	65.3%	36	4589	967

Table 1: Comparison on evaluation of MCTS and STS. C, H are parameters in Algorithm 1. S. rate is the ratio of solved boards, $N_p, N_t (= N_p \cdot H), N_g$ are the average number of passes, tree nodes and game states observed until the solution is found. Full table is available in Table 4.

the irreversibility of certain actions. A typical example is pushing a box into a corner, though there are multiple less apparent cases. The environment's complexity is formalized by the fact that, deciding whether a level of Sokoban is solvable or not, is NP-hard, see e.g. Dor and Zwick [9]. Due to these challenges, the game is often used to test reinforcement learning and planning methods.

We use procedurally generated Sokoban levels, as proposed by Racanière et al. [26]. The agent is rewarded with 1 by putting a box into a designated spot and additonally with 10 when all the boxes are in place. We use Sokoban with the board of size $(10, 10)$, 4 boxes, and the limit of 200 steps. We use an MCTS implementation with transposition tables and a loop avoidance mechanism, see Miłoś et al. [21] and Appendix A.5.

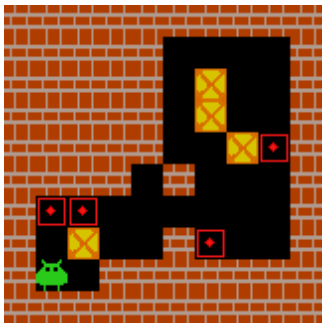

Figure 2: Example $(10, 10)$ Sokoban board with 4 boxes. Boxes (yellow) are to be pushed by agent (green) to designed spots (red). The optimal solution in this level has 37 steps.

In the first experiment, we evaluated the planning capabilities of STS in isolation from training. To this end, we used a pre-trained value function and varied the number of passes C and the depth of multi-step expansion H, such that $H \cdot C$ remains constant. In Table 1 we present quantities (N_p, N_t, N_g), which measure planning costs. In two presented scenarios, there is a sweet spot for the choice of H. For this choice, the number of tree nodes, N_t, which is the most important metric, is the smallest. Interestingly, we observe an increase in the solved rate. This may possibly be explained by the fact that the number of distinct visited game states, N_g, grows. This suggests that STS explores more aggressively and efficiently.

In the second line of experiments, we analyzed the training performance (see Algorithm 2). For MCTS we used $C = 50$ passes per step, while for STS we considered $C = 10$ passes with multi-step expansion $H = 5$. The learning curve for STS dominates the learning curve for MCTS, which persists throughout training, see Figure 3. Since the difficulty of Sokoban levels increases progressively, the achieved improvement is substantial, even though in absolute terms, it may seem small.

Shooting methods perform poorly for Sokoban: we evaluated Bandit Shooting (Algorithm 4), which struggled to exceed 5% solved rate. Only when the difficulty of boards was significantly reduced, to the board size of $(6, 6)$ with 2 boxes, this method achieved results above 90%. Our shooting setup included applying loop avoidance improvements. This feature is highly effective in the case of MCTS (and STS) but did not bring much improvement for shooting methods. Details are provided in Appendix A.7.2.

We conclude with a conjecture that for domains with combinatorial complexity, tree methods (MCTS or STS) significantly outperform shooting methods, and STS offers some benefits over MCTS.

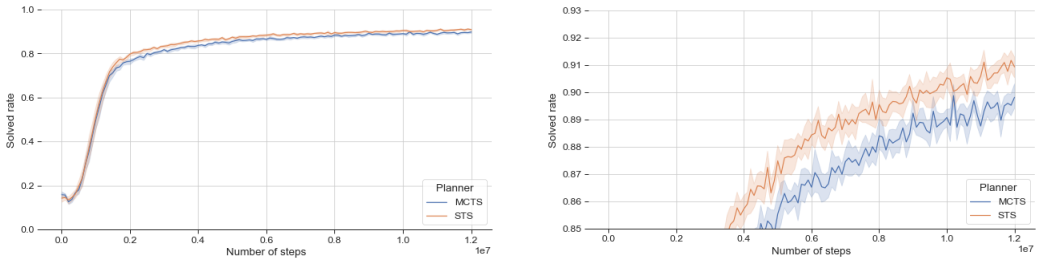

Figure 3: Learning curve for Sokoban domain. Left figure shows full results, right one inspects the same data for limited interval of values on the y axis. The results are averaged over 10 runs, shaded areas shows 95% confidence intervals. The x axis is the number of collected samples.

4.2 Google Research Football

Google Research Football (GRF) is an environment recently introduced in Kurach et al. [19]. It is an advanced, physics-based simulator of the game of football. It is designed to offer a set of challenges for testing RL algorithms. At the same time, it is highly-optimized and open-sourced. GRF is modeled after popular football (a.k.a. soccer) video games, fun and engaging for humans. As such, it requires both tactical and strategical decision-making. This makes it an interesting benchmark for planning algorithms. A part of GRF is the Football Academy consisting of 11 scenarios highlighting various tactical difficulties, see Kurach et al. [19, Table 10] for description. Due to its diversity, the GRF Academy is an excellent testing ground of planning methods listed in Section 3, including STS. GRF provides several state representations, including internal game representation as well as visual observation. We tested both of them: the former was processed with an MLP architecture, while the latter with a convolutional neural network. Details are provided in Appendix A.1.

One feature which makes GRF hard (and thus interesting) for planning is its action space, which is relatively large (19 actions). From the perspective of the design of a low budget planner, this can be viewed as a challenge.

A GRF Academy episode is considered finished after 100 steps or when the goal is scored by the agent. The game is stochastic, hence we report the solved rates. In Table 2 we compare various methods including STS and the baseline PPO policy provided by [19] for a selected four academy environments (one easy and three hard). We report the median of the solved rates in at least three runs with different seeds. Table 5 contains results for all GRF Academy environments.

Figure 4: Example from the Google Football League

Random shooting For each of the Random Shooting and Bandit Shooting planners (Algorithm 3 and Algorithm 4, respectively), we performed two batches of experiments: with and without training. The former used two different state representation and, as a consequence, two different architectures (MLP and Conv.). The latter used a uniform policy (flat) or a pre-trained policy (PPO). For all the variants, we set $C = 30$ passes and a planning horizon $H = 10$. More details can be found in Appendix A.1.

Method		Corner	Counterattack hard	Empty goal	Pass and shoot with keeper
PPO [19]		0.10	0.65	0.90	0.65
Random Shooting	flat	0.00	0.10	0.00	0.05
	PPO	0.10	0.30	1.00	0.25
	MLP	0.74	-	-	0.81
Bandit Shooting	flat	0.10	0.00	0.05	0.05
	PPO	0.05	0.80	1.00	0.55
	MLP	0.60	-	-	0.90
	Conv.	0.41	0.44	0.97	0.94
MCTS Conv.		0.13	0.56	1.00	-
STS	MLP	0.78	0.97	1.00	0.94
	Conv.	0.81	1.00	1.00	1.00

Table 2: Summary of selected algorithms' performance on GRF. Entries correspond to a solved rate. Results for the whole GRF Academy are presented in Table 5.

The flat version cannot solve GRF Academy tasks. This is rather unsurprising and confirms that it is a challenging test suite. The Bandit Shooting algorithm generally offers a better performance both when using the pre-trained policy or training from scratch. This indicates that bandit-based exploration results in more reliable estimates of action values. Bandit Shooting Conv. experiments are better than the baseline in 6 cases and worse in 4. This shows that, at least in some environments, planning can improve performance. We also tested whether mixing the policy with Dirichlet noise (see [32]) and sampling an action to take in an environment can impact exploration and training performance. Nevertheless, the results were inconclusive (see Appendix A.4 for details). It can be seen that the *corner* scenario is particularly challenging: the baseline scores

on the lower end of the spectrum, the shooting algorithms rather underperformed and the training was quite unstable. The results improved significantly under the STS algorithm. In the Shooting experiments, we used approx. 1.5M training samples (median).

STS and MCTS STS achieves state-of-the-art results on the GRF Academy and significantly outperforms other methods. For STS we used $C = 30$ passes with $H = 10$ and for MCTS we set corresponding $C = 300$.

STS Conv. completely solves 8 out of 11 academy environments, see full results in Table 5, and is the best on the remaining 3, except for *run to score with keeper* (where STS achieves 97% and is beaten only by PPO), see Table 2 and Table 5. STS MLP achieves a close second place, except for *run pass and shoot with keeper* (it achieves 97% and other than STS Conv. is only beaten by Bandit Shooting with PPO) and *run to score with keeper* (94%). These results provide further evidence that STS gives a boost in environments requiring long-horizon planning. This stands in sharp contrast with MCTS, which was not able to achieve impressive results in the considered time budget. We found that exploration was a challenge in GRF Academy environments. Namely, training often got stuck in disadvantageous regions of the state space, which was caused by unfavorable random initialization of the value function. To deal with it, the last layer of the value function neural network was initialized to 0. We suspect this zero-initialization method might be useful in other domains as well. In the STS experiments we used approx. 0.8M training samples (median).

More details can be found in Appendix A.8, including ablations. They indicate that *multi-step expansion* of STS blends well with various elements of the MCTS toolbox as well as demonstrate the impact of the aforementioned zero-initialization.

5 Conclusions and further work

In this paper, we introduced a new algorithm, Shoot Tree Search. STS aims to explicitly address the dilemma between depth and breadth search in large state spaces. That touches upon interesting issues of using randomness and structure in search algorithms. The core improvement is *multi-step expansion*, which may be used to control the depth of search and inject into planning more randomness via random multi-step expansions. Having empirically verified the efficiency of this extension in many challenging scenarios, we argue that it could be included in a standard MCTS toolbox.

In future work, we want to address dynamical (online) change of planner parameters. In some initial experiments, we observed that varying the temperature of the shooting policy, during Random Shooting planning rollout, may improve overall exploration and result in better coverage of the state space.

There are many interesting follow-up research directions involving STS. One of them concerns the automatic choice of the multi-step expansion depth, H, during training. This could not only improve the performance of the method, but also alleviate the necessity for fine-tuning this additional hyper-parameter. Another, quite natural extension of this work is to use learned models. As a research question, this typically splits into two sub-problems: learn an accurate model, or adjust the planner to accommodate for the model's deficiencies. An exciting research avenue, is related to a multi-agent version of GRF. This constitutes an open challenge both for planning and learning models.

It is interesting to study STS itself. Historically, the fusion of different multi-step estimates (such as TD(λ) or GAE) lead to significant improvements, and it is only natural to ask if a similar advancement can be reached here. Moreover, STS could be combined with different statistical tree search methods, where statistics other than the expected value (e.g. $\max$) are stored and updated (see e.g. Agostinelli et al. [2]). The method could also be augmented with uncertainty estimation (e.g. in the spirit of Miłoś et al. [21]) to strengthen the exploration, and consequently the algorithm.

Another tempting option would be to combine ideas of this work with other search methods. We conjecture that similar benefits would be observed for the A* algorithm with learned heuristics (akin to Agostinelli et al. [2]) or LevineTS (see Orseau et al. [24]).

Broader Impact

Evaluation of broader impact is not applicable to our work.

In more detail, our research aims to develop efficient planning algorithms. Such algorithms may have huge impact in the future for creating truly intelligent systems. However, at the moment our studies involve fundamental algorithmic properties without immediate real-world applications. Our finding are mostly useful for other reinforcement learning researchers.

Acknowledgments and Disclosure of Funding

This research was supported by the PL-Grid Infrastructure. We extensively used the Prometheus supercomputer, located in the Academic Computer Center Cyfronet in the AGH University of Science and Technology in Kraków, Poland. The work of Konrad Czechowski, Piotr Kozakowski, Piotr Januszewski and Piotr Miłoś was supported by the Polish National Science Center grants UMO-2017/26/E/ST6/00622. We managed our experiments using https://neptune.ai. We would like to thank the Neptune team for providing us access to the team version and the technical support.

References

[1] Bruce Abramson. Expected-Outcome: A General Model of Static Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1990. ISSN 01628828. doi: 10.1109/34.44404.

[2] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1 (8):356–363, 2019. doi: 10.1038/s42256-019-0070-z. URL `https://doi.org/10.1038/s42256-019-0070-z`.

[3] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *NIPS*, 2017.

[4] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[5] Eduardo F Camacho and Carlos Bordons Alba. *Model predictive control*. Springer Science & Business Media, 2013.

[6] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, pages 4754–4765, 2018.

[7] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[8] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. `https://github.com/openai/baselines`, 2017.

[9] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[10] James E Doran and Donald Michie. Experiments with the graph traverser program. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 294(1437): 235–259, 1966.

[11] Gregory Farquhar, Tim Rocktäschel, Maximilian Igl, and Shimon Whiteson. Treeqn and atreec: Differentiable tree planning for deep reinforcement learning. *CoRR*, abs/1710.11417, 2017.

[12] Matthew L. Ginsberg. GIB: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, 2001. ISSN 10769757. doi: 10.1613/jair.820.

[13] Shixiang Gu, Timothy P. Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *ICML*, 2016.

[14] Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Theophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, Greg Wayne, David Silver, and Timothy P. Lillicrap. An investigation of model-free planning. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pages 2464–2473, 2019. URL http://proceedings.mlr.press/v97/guez19a.html.

[15] Xiaoxiao Guo, Satinder P. Singh, Honglak Lee, Richard L. Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *NIPS*, 2014.

[16] Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Tobias Pfaff, Theophane Weber, Lars Buesing, and Peter W. Battaglia. Combining q-learning and search with amortized value estimates. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=SkeAaJrKDS.

[17] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[18] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H. Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, Ryan Sepassi, George Tucker, and Henryk Michalewski. Model-based reinforcement learning for atari. *CoRR*, abs/1903.00374, 2019. URL http://arxiv.org/abs/1903.00374.

[19] Karol Kurach, Anton Raichuk, Piotr Stańczyk, Michał Zając, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, and Sylvain Gelly. Google research football: A novel reinforcement learning environment. *arXiv preprint arXiv:1907.11180*, 2019.

[20] Karol Kurach, Anton Raichuk, Piotr Stańczyk, Michał Zając, Olivier Bachem, Lasse Espeholt, Carlos Riquelme, Damien Vincent, Marcin Michalski, Olivier Bousquet, and Sylvain Gelly. Google research football. https://github.com/google-research/football, 2019.

[21] Piotr Miłoś, Łukasz Kuciński, Konrad Czechowski, Piotr Kozakowski, and Maciek Klimek. Uncertainty-sensitive learning and planning with ensembles. *arXiv preprint arXiv:1912.09996*, 2019.

[22] Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation, ICRA 2018, Brisbane, Australia, May 21-25, 2018*, pages 7559–7566. IEEE, 2018. doi: 10.1109/ICRA.2018.8463189. URL https://doi.org/10.1109/ICRA.2018.8463189.

[23] Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. In *NIPS*, 2017.

[24] Laurent Orseau, Levi Lelis, Tor Lattimore, and Theophane Weber. Single-agent policy tree search with guarantees. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 3205–3215, 2018.

[25] Razvan Pascanu, Yujia Li, Oriol Vinyals, Nicolas Heess, Lars Buesing, Sébastien Racanière, David P. Reichert, Theophane Weber, Daan Wierstra, and Peter Battaglia. Learning model-based planning from scratch. *CoRR*, abs/1707.06170, 2017.

[26] Sébastien Racanière, Theophane Weber, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning. In *NIPS*, 2017.

[27] Stéphane Ross and J. Andrew Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *CoRR*, abs/1406.5979, 2014. URL http://arxiv.org/abs/1406.5979.

[28] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[29] Antonoglou Ioannis Hubert Thomas Simonyan Karen Sifre Laurent Schmitt Simon Guez Arthur Lockhart Edward Hassabis Demis Graepel Thore Lillicrap Timothy Silver David Schrittwieser, Julian. Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. *arXiv preprint arXiv:1911.08265*, 2019. URL `https://arxiv.org/abs/1911.08265`.

[30] Brian Sheppard. World-championship-caliber Scrabble. *Artificial Intelligence*, 2002. ISSN 00043702. doi: 10.1016/S0004-3702(01)00166-7.

[31] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 2017.

[32] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 1144:1140–1144, 2018.

[33] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

A.1 Training details

We provide the code of our methods and hyper-parameters configuration files in `https://github.com/shoot-tree-search/sts`.

The training loop follows the logic of Algorithm 2. We use a distributed setup with 30 workers and a replay buffer of size 30000. We perform 1000 optimizer updates on batches of transitions whenever all workers collect and store one full episode. During batch sampling, we ensured an equal amount of examples from solved and unsolved episodes. In GRF and Sokoban experiments, each episode was limited to 100 and 200 time steps, respectively.

A value function approximator, $\mathbf{V}_\theta$, is trained via the MSE loss and "reward-to-go" targets $\sum_{i=t+1}^{T} \gamma^{i-t-1} r_i$, where T is the terminal time-step in an episode. Q-function approximator, used by MCTS and STS in GRF experiments (see Section A.5 for details), is trained via the MSE loss and tree action-values targets, similar to the one used in Hamrick et al. [16], Miłoś et al. [21].

Policy, π_ϕ, is trained using the cross-entropy loss. As targets, we use one-hot encoded actions chosen in the environment for Random Shooting and the empirical distribution of actions chosen in the root during the planning for Bandit Shooting, MCTS, and STS.

The total loss is a weighted sum of the value function (or the Q-function) loss, the policy loss (weighted by $1\mathrm{e}{-2}$ in Random Shooting and Bandit Shooting, and $1\mathrm{e}{-3}$ in MCTS and STS), and a regularizing, l_2 term (weighted by $1\mathrm{e}{-6}$).

A pre-trained PPO policy in Shooting methods was obtained using a script included in the Google Research Football repository (see Kurach et al. [20]) and the OpenAI Baselines (Dhariwal et al. [8]) PPO2 implementation.

A.2 Hyper-parameters

Table 3 presents hyper-parameters used in our experiments. These were based on hyper-parameters previously proposed in the literature, e.g., Miłoś et al. [21], and a certain amount of tuning.

	Sokoban			Google Research Football		
Parameter	Shooting	MCTS	STS	Shooting	MCTS	STS
Number of passes C	48	50	10	30	300	30
Planning horizon H	5	1	5	10	1	10
Discounting γ	0.99	0.99	0.99	0.95 / 0.99[1]	0.99	0.99
Exploration weight c_{puct}	10.0[2]	-	-	1.0 / 2.5[3]	1.0	1.0
Policy π_ϕ temperature[4]	2.0	-	-	2.0	1.0	1.0
Action sampling temp. τ	-	-	-	0.3[5]	0.3	0.3
Dirichlet parameter α	-	-	-	0.03[5]	0.3	0.3
Noise weight c_{noise}	-	-	-	0.1[5]	0.1	0.1 / 0.3[6]
Depth limit `depth limit`[7]	-	-	-	-	30	30
VF zero-initialization[8]	no	no	no	no	yes	yes
Optimizer	RMS	RMS	RMS	RMS	Adam	Adam
Learning rate	$2.5\mathrm{e}{-4}$	$2.5\mathrm{e}{-4}$	$2.5\mathrm{e}{-4}$	$1.0\mathrm{e}{-4}$	$1.0\mathrm{e}{-3}$	$1.0\mathrm{e}{-3}$
Batch size	32	32	32	64	64	64

[1] All $\gamma = 0.99$ except for Shooting experiments with a uniform and a pre-trained PPO policy, where $\gamma = 0.95$.
[2] Applies only to Bandit Shooting.
[3] $c_{puct} = 1.0$ for Bandit Shooting with a uniform and a pre-trained PPO policy and $c_{puct} = 2.5$ for Bandit Shooting with a trained policy.
[4] MCTS and STS in Sokoban does not use policy, see Section A.5 for details.
[5] Applies only to Bandit Shooting with additional exploration mechanisms, see Section A.4.
[6] $c_{noise} = 1.0$ for STS Conv. and $c_{noise} = 0.3$ for STS MLP.
[7] The maximum number of nodes visited in a single planning pass, see Section A.5.
[8] If the last layer of a value function neural network was initialized to 0, see Section A.8.2.

Table 3: Default values of hyper-parameters used in our experiments.

A.3 Network architectures

In GRF experiments we use two different state representations: 'simple115' and 'extended' (see Section A.8). In the former case, we use an MLP architecture with two hidden layers of 64 neurons, while in the latter case, we use 4 convolutional layers with 16, 3x3, filters, zero-padding and stride 2, followed by a dense layer of 64 neurons. In both cases, two heads, corresponding to a value function (or Q-function for MCTS and STS) and policy, follow.

In Sokoban experiments, we use 5 convolutional layers of 64, 3x3, filters with zero-padding and stride 1, followed by a dense layer of 128 neurons and heads corresponding to a value function and policy (policy is used only for Shooting methods).

In all the cases, we use the ReLU non-linearity. We use the standard Keras initialization schemes, except for MCTS and STS in GRF experiments, see Section A.8.2.

A.4 Bandit Shooting

Algorithm 7 Bandit Shooting Planner with additional exploration mechanisms, requires exploration weight c_{puct}, action sampling temperature τ, noise weight c_{noise} and Dirichlet distribution parameter α

function SELECT(`state`)
  $s \leftarrow$ `state`
  $P(s,a) \leftarrow (1 - c_{noise})\pi_\phi(s,a) + c_{noise}D$
  $U(s,a) \leftarrow \sqrt{\sum_{a'} N(s,a')}/(1 + N(s,a))$
  $a \leftarrow \text{argmax}_a(Q(s,a) + c_{puct}P(s,a)U(s,a))$
  $s', r \leftarrow model.\text{STEP}(s,a)$
  return $(s,a,r),\ s'$

function EXPAND(`leaf`)
  `The same as in Algorithm 3.`

function UPDATE(`path`, `rollout`)
  `The same as in Algorithm 3.`

function CHOOSE_ACTION(s)
  $a \sim \text{softmax}\left(\frac{1}{\tau}\log N(s,\cdot)\right)$
  return a

Algorithm 7 describes Bandit Shooting with additional exploration mechanisms: mixing the policy with Dirichlet noise (as in [32]) and action sampling with temperature τ in CHOOSE_ACTION(s). The noise variable D is sampled from the Dirchlet distribution $Dir(\alpha)$ each time when PLANNER is called (see also Algorithm 2).

A.5 MCTS

In our experiments, we used various implementations of MCTS. The reasons were two-fold. First, some implementation details fit better Sokoban and some GRF. Second, we wanted to check in various cases that the multi-step expansion is beneficial, see Section A.6.

In Sokoban experiments, we used the MCTS implementation similar to the one in Miłoś et al. [21], containing a loop avoidance mechanism and transposition tables. The loop avoidance mechanism alters `SELECT` and `CHOOSE_ACTION` (see Algorithm 5) so that the selected `path` does not contain repetitions of states. The transposition tables are a rather standard technique, which proposes to accumulate search statistics (i.e., W, N, Q) for states of the environment (rather than for the nodes of the search tree, as it happens in the standard case).

In GRF, we used our custom implementation of MCTS based on the one in Silver et al. [31]. It uses leaf evaluation with Q-function and policy networks. The Q-function is used to evaluate all children of a given node at once (instead of separately invoking value function $\mathbf{V}_\theta$ in `UPDATE`). The policy network is considered to be 'prior' for choosing actions, similarly as in `SELECT` in Algorithm 7. Dirichlet noise, parameterized by α and c_{noise}, is mixed with the prior in the root and action sampling with temperature τ is used to choose action on the real environment, similarly as in Bandit Shooting with additional exploration mechanisms in Section A.4. Additionally, we put a limit, `depth limit`, on the maximum number of nodes visited in a single STS pass.

A.6 STS

We tested STS with two MCTS setups described in Section A.5. In both the cases we observed substantial experimental improvements as reported in Section A.7 and Section A.8. This alone, in our view, provides enough evidence that the *multi-step expansion* is a useful method.

Apart from this, STS offers practical computational benefits, which are analyzed below.

A.6.1 Computational benefits of STS

We distinguish three types of computational costs in MCTS (see Algorithm 5):

1. Traversing down the search tree (performed in SELECT and EXPAND).
2. Backpropagation of values and counts update (handled by UPDATE).
3. Evaluation of heuristics (value network $\mathbf{V}_\theta$, or Q-function and policy as described in Section A.5)

In large GRF experiments, we found that it was the first cost that dominated the remaining two. The reason is that the cost of building a search tree is quadratic to its depth. The use of *multi-step expansion* significantly reduces this cost as several nodes are added during single tree traversal. In our case, these benefits allowed for much smoother experimenting with GRF and are, arguably, a step towards developing more efficient planners. We expect this might be practically useful (i.e., costs 1 and 2 are dominant) when the search size is large, or the heuristic evaluation is relatively cheap compared to the environment step. This is the case in some of our GRF experiments. The GRF simulator is rather complex and slower than small MLP networks.

The following simple lemma offers some theoretical analysis.

Lemma A.6.1. *Assume that STS and MCTS build the same tree $\mathcal{T}$, starting from the root state s_0. Denote the number of nodes in $\mathcal{T}$ as C and the number of nodes to be added at a single multi-step expansion of STS as H. Then the number of steps in $\mathcal{T}$ performed by STS will be lower compared to MCTS by a factor in $[\frac{h-1}{2}, h]$.*

Proof. Lets consider h consecutive nodes $s_1, \ldots, s_h$ in the search tree added in a single EXPAND step during STS search. In STS, the number of steps, C_{STS}, in the tree during SELECT and EXPAND is equal to $h+d$, where d is distance between s_0 and s_1 in $\mathcal{T}$. To add the same set of nodes during MCTS search, one need h separate calls to SELECT and EXPAND. The total number of steps performed is $C_{MCTS} = \sum_{k=0}^{h-1} d + k + 1 = hd + h\frac{h-1}{2}$. Clearly,

$$\frac{h-1}{2} C_{STS} \leq C_{MCTS} \leq h C_{STS}.$$

Similar calculation hold for the costs of backpropagation. □

A.7 Sokoban experiments

For a description of Sokoban see Section 4.1. In our experiments, we used inputs of dimension $(x, x, 7)$, where (x, x) is the size of the board ($(10, 10)$ in most cases) and 7 is one-hot encoding of the state of a given cell (enumerated as follows: wall, empty, target, box_target, box, player, player_target). In most experiments, we used 4 boxes. The agent is rewarded with 1 by putting a box into a designated spot and additionally with 10 when all the boxes are in place[1]. The action space consists of four movement directions (up, down, right, left).

A.7.1 Evaluation experiments

In Table 4 we show full details of the evaluation experiment (which complements Table 1). Recall that in this experiment, we evaluated the planning capabilities of STS in isolation from training. To this end, we used a pre-trained value function and varied the number of passes C and the depth

[1] Our Sokoban code is fully compatible with Racanière et al. [26].

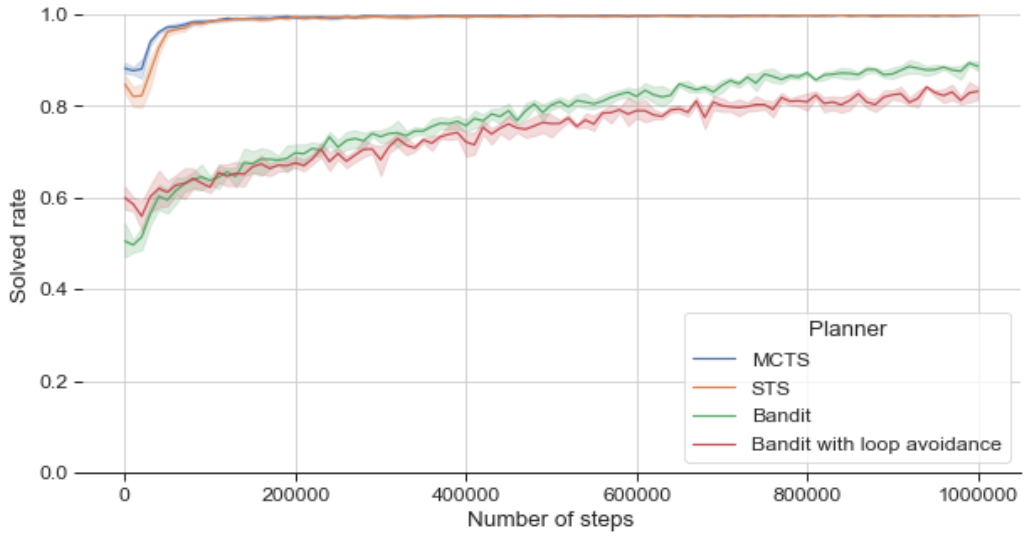

Figure 5: Sokoban on simpler boards: training curves for MCTS, STS and Bandit Shooting with and without loop avoidance. Mean over 5 seeds with shaded regions representing 95% confidence intervals.

of multi-step expansion H, such that $H \cdot C$ remains constant. In Table 4, we present quantities (N_p, N_t, N_g), which measure planning costs for finding a solution (the average number of passes, tree nodes and game states observed, respectively, until the solution is found). We run experiments with and without the loop avoidance mechanism (see Section A.5). We observe that there is a sweet spot for the choice of H. It is evident for the 'no avoid loop' case, $C = 32, H = 8$. For this choice, the number of tree nodes, N_t, which is the most important metric, is the smallest. Interestingly, we observe a significant increase in the solved rate. This may be explained by the fact that the number of distinct visited game states, N_g, grows. This suggests that STS explores more aggressively and efficiently. For bigger H, we observe a further increase of the solved rate until some point, though at the cost of much bigger N_t.

In experiments with the avoid loop mechanism, there is a similar effect for $C = 64, H = 4$, though more subtle, probably because results are already quite strong. Moreover, we observe a more significant drop in performance as H increases (when planning resembles more shooting methods).

The values presented in Table 4 are averages over more than 5000 boards.

A.7.2 MCTS and Shooting on simpler boards

We found the Bandit Shooting method underperforming on Sokoban. As a sanity test, we tested a simpler setting with smaller boards of size $(6, 6)$ and two boxes. Learning curves are presented in Figure 5. MCTS and STS experiments quickly learn to solve over 99% of boards. Bandit Shooting experiment showed stable but much slower progress. We also evaluated the version of Bandit Shooting, with additional loop avoidance, see Section A.5. This mechanism was beneficial for MCTS and STS but failed to bring improvements for the shooting algorithms.

A.8 Google Research Football experiments

For a description of Google Research Football see Section 4.2. A Google Research Football academy environment is considered solved when an agent scores a goal. Reported results correspond to solved rates over 20 episodes in case of Shooting methods with an uniform and a pre-trained policy and around 30 episodes in case of all other methods. Results for MCTS, STS, and Shooting methods with the trained policy are medians of at least three training runs. During evaluations we disabled Dirichlet noise and action sampling (in Bandit Shooting Expl., MCTS and STS).

Scenario	C	H	S. rate	N_p	N_t	N_g
avoid loops	256	1	95.2%	1224	1224	716
	128	2	95.9%	569	1137	728
	64	4	96.5%	299	1194	830
	32	8	95.9%	173	1385	1040
	16	16	95.7%	114	1822	1333
	8	32	93.4%	79	2527	1528
	4	64	89%	62	3960	1491
	2	128	80%	52.7	6754	1207
no avoid loops	256	1	84.5%	1497	1497	376
	128	2	86.3%	724	1448	332
	64	4	87.8%	385	1541	370
	32	8	88.4%	185	1483	409
	16	16	89.5%	110	1754	539
	8	32	89.9%	84	2690	882
	4	64	85.2%	68	4463	1300
	2	128	65.3%	36	4589	967

Table 4: Evaluation of various STS settings on Sokoban

Google Research Football offers two major mode of observations: 'simple115' and 'extracted' (also called the super mini-map).

The simple115 state representation is consists of coordinates of players, players' movement directions, the ball position, a ball movement direction, a one-hot encoding of ball ownership, a one-hot encoding of which player is active. This totals in a vector of length 115.

The extracted state representation consists 4 stacked layers of size $(72, 96)$. Layers contain one-hot encoding of spatial positions of game entities. These are (on the subsequent layers): players on the left team, players on the right team, the ball, and the active player.

We note that even though the extracted representation contains 'less information' than simple115, it has been reported in [19] to generate better results.

In our experiments, we use the so-called checkpoint rewards, which provide an additional signal for approaching the goal area. Details can be found in [19], where they were introduced and used in large-scale experiments.

The action space in GRF consists of 19 actions representing high-level football behaviors (e.g. "Short Pass"), see [19, Table 1].

Figure 6 shows selected training curves on Google Research Football, the best from each family of our methods: Shooting, MCTS and STS. Figure 7 shows all training curves for our methods on Google Research Football. On the y-axis is the solved rate calculated as described above in Section A.8. On the x-axis is the number of real steps in the environment (planning steps in the simulator are not added). Curves are mean over 3 training runs with different seeds and shaded regions represent 95% confidence intervals (exceptionally for Bandit Shooting we report just one run). Moreover, to smooth the curves, data points are averaged in the windows of 10000 steps.

A.8.1 Shooting methods

Table 5 presents our methods performance in all Google Research Football academies.

Tuning c_{puct} turned out to be the most important one to make Bandit Shooting work, see Algorithm 4. In a nutshell, it needs to be adjusted to scale of rewards (value function) in a given environment. In our experiments we found $c_{puct} = 2.5$ to work best.

Using additional Dirichlet noise, $c_{noise} > 0$, and action sampling on the real environment, $\tau > 0$ (see Algorithm 7) resulted in inferior results with an exception of the "Counterattack hard" scenario.

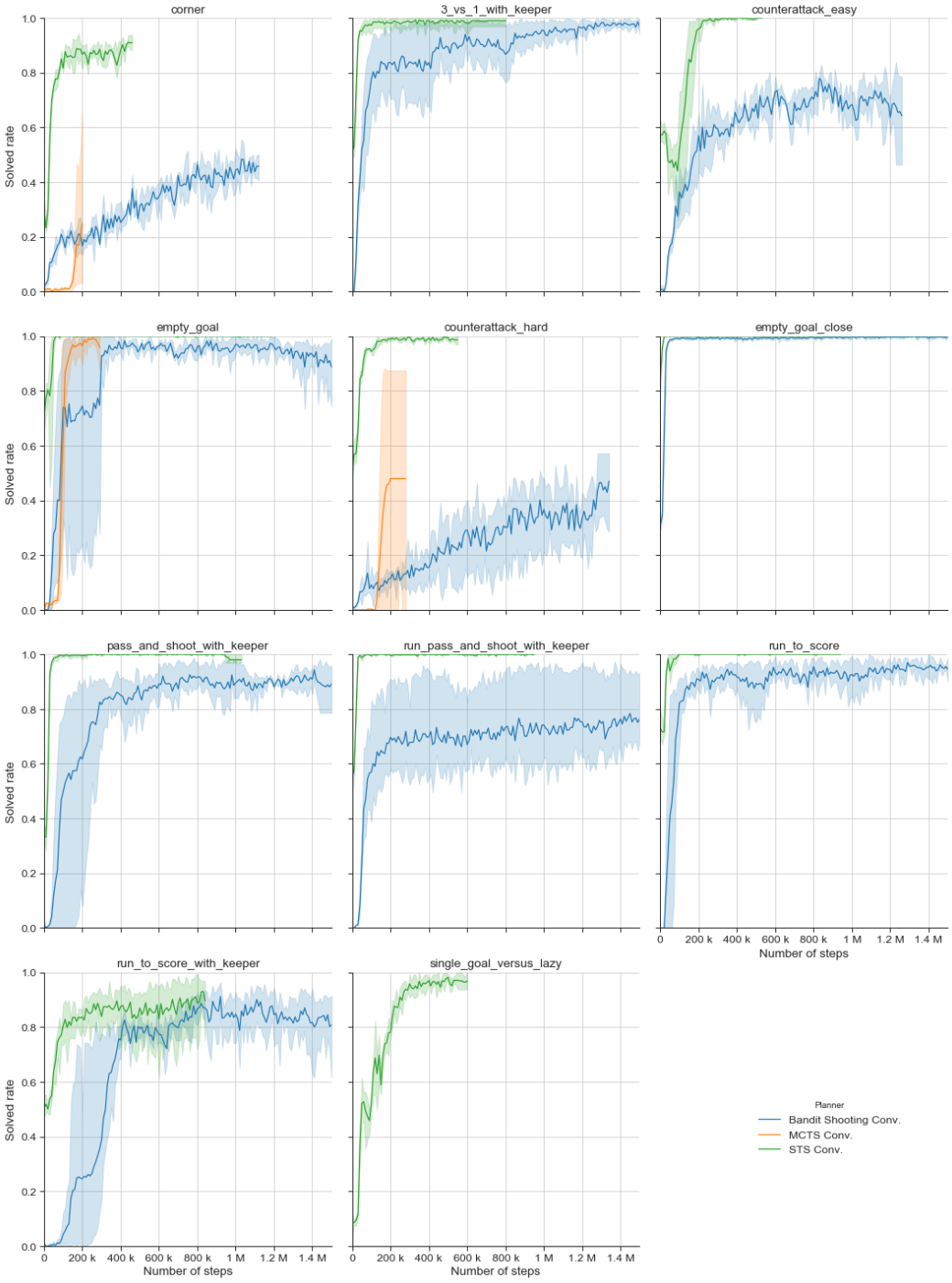

Figure 6: Google Research Football training curves for three best methods on GRF. Mean over 3 seeds with shaded regions representing 95% confidence intervals.

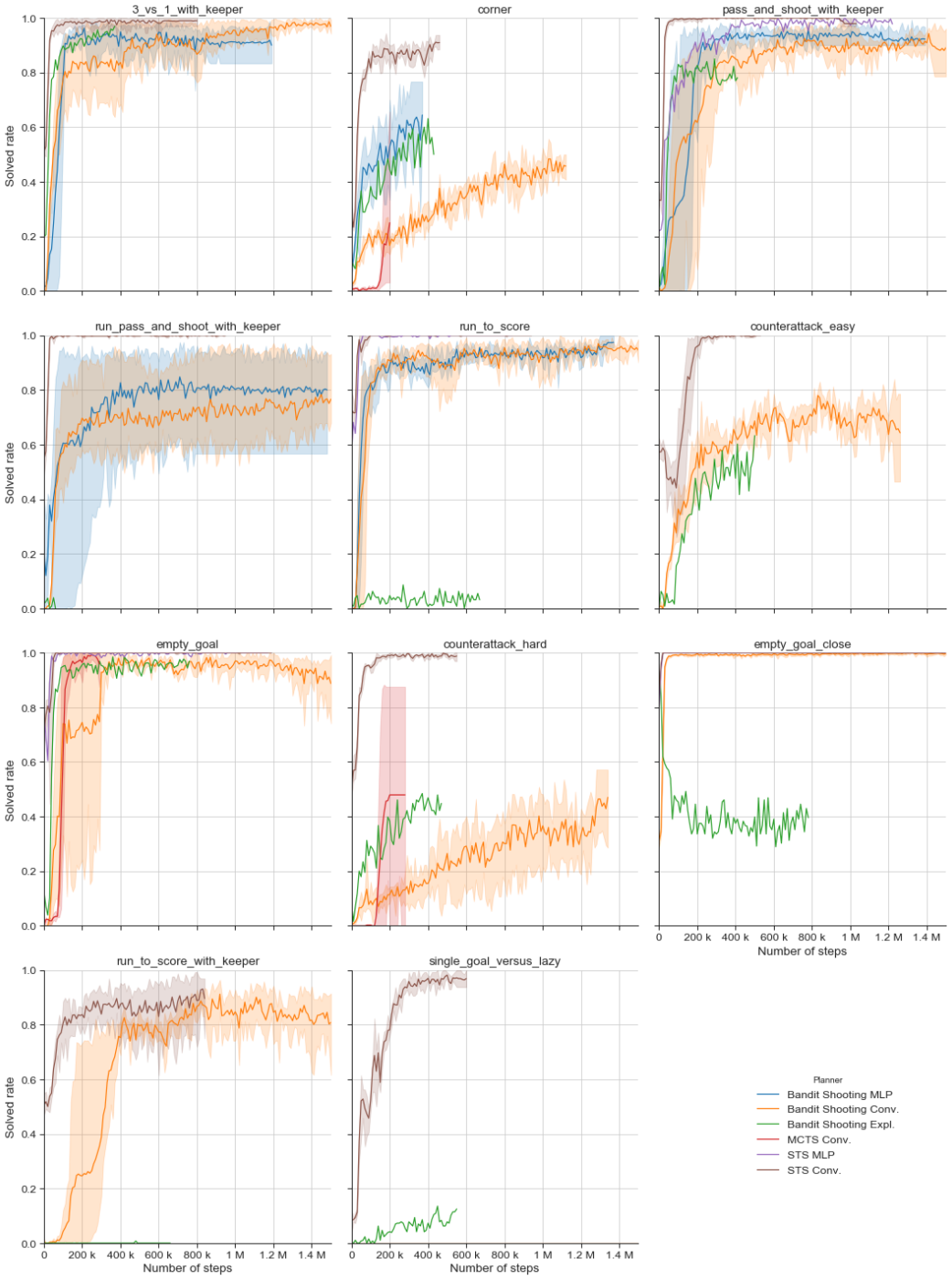

Figure 7: Google Research Football training curves for all methods on GRF. Mean over 3 seeds with shaded regions representing 95% confidence intervals.

Method		3 vs. 1 with keeper	Corner	Counterattack easy	Counterattack hard	Empty goal	Empty goal close	Pass and shoot with keeper	Run pass and shoot with keeper	Run to score	Run to score with keeper	Single goal versus lazy
PPO [19]		0.90	0.10	0.70	0.65	0.90	1.00	0.65	0.90	0.90	1.00	0.90
Random Shooting	flat	0.10	0.00	0.05	0.10	0.00	0.95	0.05	0.10	0.00	0.00	0.00
	PPO	0.45	0.10	0.10	0.30	1.00	1.00	0.25	0.80	0.80	0.20	0.30
	MLP	0.85	0.74	-	-	-	-	0.81	0.58	0.74	-	-
Bandit Shooting	flat	0.20	0.10	0.00	0.00	0.05	0.35	0.05	0.05	0.05	0.00	0.00
	PPO	1.00	0.05	0.95	0.80	1.00	1.00	0.55	1.00	0.85	0.45	0.60
	MLP	0.93	0.60	-	-	-	-	0.90	0.90	1.00	-	-
	Conv.	0.97	0.41	0.81	0.44	0.97	1.00	0.94	0.69	1.00	0.91	0.00
	Expl.	1.00	0.53	0.50	0.66	1.00	0.00	0.81	0.34	0.00	0.00	0.09
MCTS Conv.		-	0.13	-	0.56	1.00	-	-	-	-	-	-
STS	MLP	1.00	0.78	1.00	0.97	1.00	1.00	0.94	0.97	1.00	0.94	0.94
	Conv.	1.00	0.81	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.97

Table 5: Summary of methods performance on GRF. Entries correspond to rounded solved rates over at least 20 episodes per environment. Results for Shooting methods with the trained policy, MCTS and STS are reported as median of at lest three training runs.

A.8.2 MCTS and STS experiments

Apart from *multi-step expansion* we introduced another simple novel method, which might be of interest to the general public. Namely, before starting training, we set the weights of the last layer of the Q-value neural network to 0 (see Section A.3 for a detailed description of architectures). We observed that this significantly improved the training stability due to better exploration (and avoiding suboptimal strategies at the early stages of training). See 'No zero initialization' on Figure 8.

A.8.3 Ablations

The ablations were performed on three environments from GRF Academy: *corner*, *counterattack hard* and *empty goal*, see Figure 8. The first two environments are difficult, while the last one is easy. The following parameters or settings were subject to analysis (they correspond to the labels in Figure 8):

- `prior noise weight`: a weight in the mixture of Dirichlet noise and the prior.
- `depth limit`: the maximum number of nodes visited in a single STS pass.
- `sampling temperature`: temperature for sampling the actions on the real environment.
- `MCTS n_passes 300`: this corresponds the standard MCTS setting with $H = 1$ (MCTS) and $C = 300$
- `Value network n_passes`: value network is used instead of Q-function. Note that `n_passes` $= 2$ matches roughly the Q-function version in terms of visited states (recall, see Section 5, that Q-function evaluates all children at once and that number of actions in GRF is 19).
- `No policy`: instead of a learned policy network, a uniform policy is used.
- `No zero initialization`: the last layer of the value function neural network was not initialized to 0 (see description at the beginning of Section A.8.2).

The default setup (denoted as `Prior noise weight 0.1`) is always positioned at the top in Figure 8. It uses parameters described in Table 3 in the Google Research Football STS column.

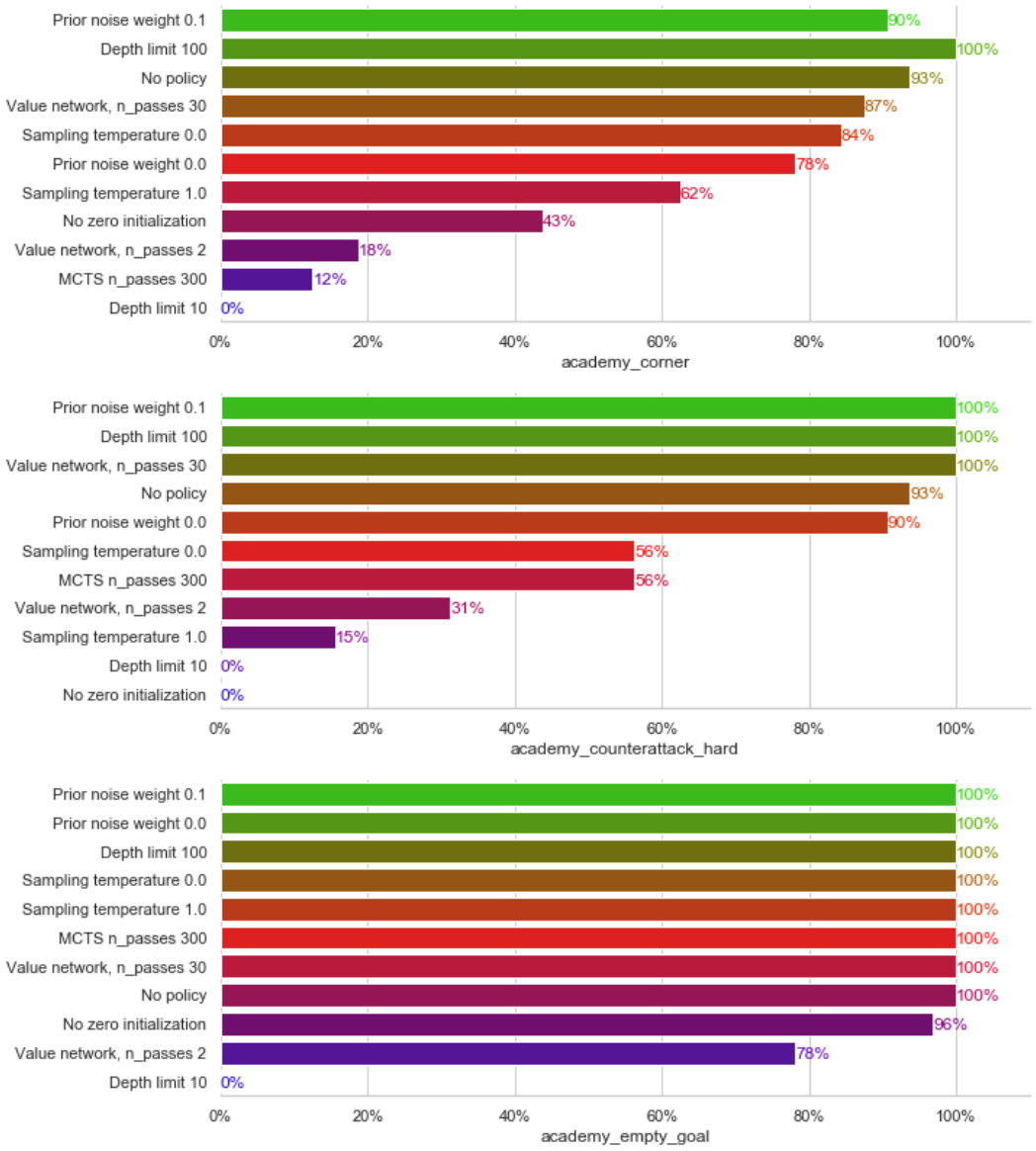

Figure 8: Ablations performed GRF Academy environments: *corner*, *counterattack hard* and *empty goal*.

A.9 Analysis, toy envs

See Figure 9 for bigger versions of the toy environments introduced in Section 4. Recall the one at the bottom of the figure. It has been crafted to highlight the ability of STS to handle "pseudo-random" errors in an approximated value function. Starting from s_0, the agent can move only to the right. The rewards for all transitions are 0 except for marked edges, where they are $-a, a > 0$. Clearly, the optimal value function is 0 in each state. Nevertheless, we assume that the estimates on the tail is not yet perfect and are ϵ. In this example, we assume that the errors arise in interactions of many factors, thus can be modeled as i.i.d. centered random variables ϵ_i such that $\mathbb{E}|\epsilon_i| < +\infty$.

The optimal path is, going over the green edge and later over the tail, is accompanied by several 'decoy' paths (marked in orange). They will not be entered unless errors on the tail have accumulated below $-a$. We denote the probability of such an event by p_H, where H is the number of steps in the multi-step expansion ($H = 1$ corresponds to MCTS). Under the assumptions above we haves

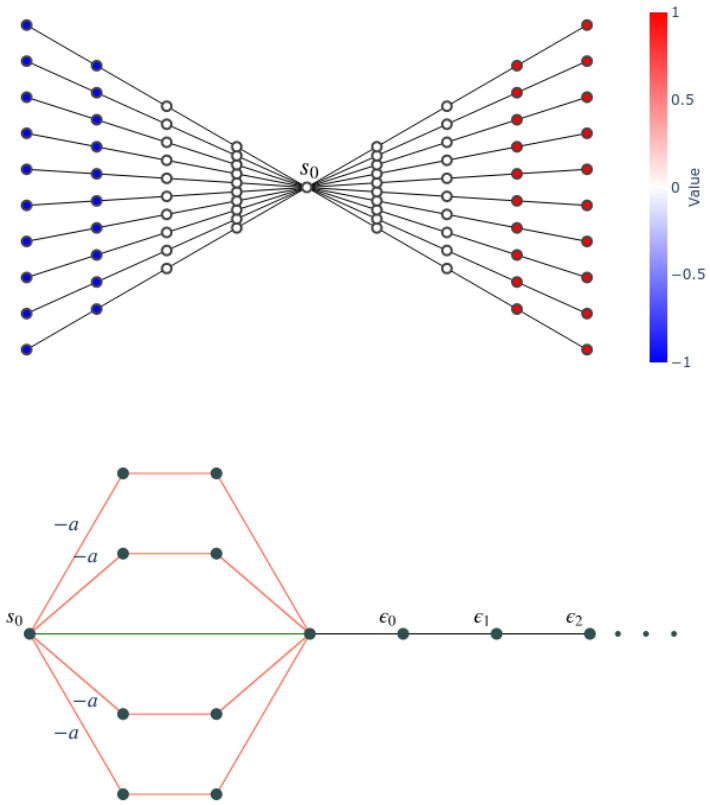

Figure 9: Full size visualization of the toy environments.

Lemma A.9.1. *Under the above assumptions* $p_1 > p_H$ *and* $p_H \to 0$.

Proof. Assume that for the first $\ell \geq 2$ steps of the search tree was unfolded via the middle (green) edge and further via the tail. The state-action value estimated by the MCTS/STS is thus $q_\ell = (\epsilon_0 + \ldots + \epsilon_{\ell-2})/\ell$. Consequently,

$$p_H = \mathbb{P}(\exists_{k \in \mathbb{N}} q_{kH} < -a).$$

The claims follow from the fact $q_\ell \to 0$ a.s., which itself is the consequence of the strong law of large numbers. □

As the lemma serves mainly the illustrative purpose we used the i.i.d. assumption, which can be easily weakened. As a test we simulate the case $\epsilon_i \sim \mathcal{N}(0, 1)$ and $a = 0.3$. In this case $p_1 = 0.56, p_2 = 0.46, p_4 = 0.35, p_8 = 0.41, p_{16} = 0.21$.

A.10 Infrastructure used

We ran our experiments on clusters with servers typically equipped with 24 or 28 CPU cores and 64GB of memory. A typical experiment was 72 hours long (the timeout set on the clusters), which was enough for most experiments. Experiments that did not converge during this time were resumed.

During the project, we run more than 10k experiments.