Structure and Randomness in Planning and Reinforcement Learning

Konrad Czechowski*§, Piotr Januszewski*†§, Piotr Kozakowski*§,
Łukasz Kuciński‡ and Piotr Miłoś‡
* Faculty of Mathematics, Informatics and Mechanics, University of Warsaw
† Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology
‡ Institute of Mathematics, Polish Academy of Science
k.czechowski@mimuw.edu.pl, piotr.januszewski@pg.edu.pl, p.kozakowski@mimuw.edu.pl,
lkucinski@impan.pl, pmilos@impan.pl

***Abstract*—Planning in large state spaces inevitably needs to balance the depth and breadth of the search. It has a crucial impact on the performance of a planner and most 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.

***Index Terms*—reinforcement learning, MCTS, planning, deep learning**

I. 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 (MCTS). Planning is a powerful concept and an important policy improvement mechanism. However, in many interesting problems, the search state 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 critical element of STS is *multi-step expansion*, which performs a fixed-length rollout during the expansion phase. In contrast to random rollouts often used by MCTS, *multi-step expansion* utilizes a neural network to guide action selection and adds visited nodes to the search tree. *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 [1]. Google Research Football is an advanced, physics-based simulator of football, introduced recently in [2]. It has been designed to offer a diverse set of challenges for testing reinforcement learning 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 IV is devoted to experimental results.

II. Background and Related Work

A classical introduction to reinforcement learning can be found in [3]. In contemporary research, the line between model-free and model-based methods is often blurred. An early example is [4], where MCTS plays the role of an 'expert' in DAgger [5], a policy learning algorithm. In a series of papers [6], [7] culminating with AlphaZero, the authors developed a system combining elements of model-based and model-free

Appendix, code of our methods and hyper-parameters configuration files used in our experiements can be found in https://github.com/shoot-tree-search/sts

§Equal contribution, alphabetic order of authors

methods that master the game of Go (and others). Similar ideas were also studied in [8]. In [9], planning and model-free learning were brought together to solve combinatorial environments. [10] successfully integrated model learning with planning in the latent space. A recent paper [11] suggests further integration of model-free and model-based methods via utilizing internal planner information to calculate more accurate estimates of the Q-function. [12] presents an expansion phase much similar to ours. The crucial algorithmic difference is the aggregate backpropagation (Section III). In a similar vein, [13] propose a framework blending tree search and Monte-Carlo simulations in a smooth way. Both [12] and [13] differ from our work, as they do not use learned value functions, resorting to heuristics and/or long rollouts. As most of these works, we use the model-based reinforcement learning paradigm, in which the agent has access to a true model of the environment.

Searching and planning algorithms are deeply rooted in classical computer science and classical AI, see e.g. [14] and [15]. Traditional heuristic algorithms such as A* [16] or GBFS [17] are widely used. The Monte Carlo Tree Search algorithm, which combines heuristic search with learning, led to breakthroughs in the field, see [18] for an extensive survey. Similarly, [19] bases on the classical BFS to build a heuristic search mechanism with theoretical guarantees. In [20] the authors utilize 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 [21]. 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 [22] and Scrabble [23].

Moreover, Monte Carlo rollouts are often used as a part of model predictive control, see [24]. As suggested by [25], [26], 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. [27] applied their Model Predictive Path Integral control algorithm [28], the approach based on stochastic sampling of trajectories, to the problem of controlling a fifth-scale Auto-Rally vehicle in an aggressive driving task.

Some works aim to compose planning modules into neural network architectures, see e.g., [29], [30]. [31], recent work on model-based Atari, has shown the possibility of sample efficient reinforcement learning with an explicit visual model. [32] 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 [1], [33], [34].

III. Methods

Reinforcement learning (RL) is formalized using Markov decision processes (MDP) [3]. An MDP is defined as $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$, where $\mathcal{S}$ is a state space, $\mathcal{A}$ is a set of actions available to an agent, P is the transition kernel, r is reward function and $\gamma \in (0, 1)$ is the discount factor. An agent policy, $\pi : \mathcal{S} \mapsto P(\mathcal{A})$, maps states to distribution over actions. An object central to the MDP formalism is the value function $V^\pi : \mathcal{S} \mapsto \mathbb{R}$ associated with policy π $V^\pi(s) := \mathbb{E}_\pi \left[\sum_{t=0}^{+\infty} \gamma^t r_t | s_0 = s\right]$, where r_t denotes the stream of rewards, assuming that the agent operates with policy π (which is denoted as $\mathbb{E}_\pi$) and that at $t = 0$ the system starts from s. The objective is to find a policy, which achieves maximal value. In our experiments, we utilize neural architectures to parametrize value and policy functions. In the further part of the paper, we will denote them by $\mathbf{V}_\theta$ and π_ϕ where θ and ϕ are parameters of networks. In this work we concentrate on planning methods, which in each step search a subspace of the state-space $\mathcal{S}$ to render a robust decision.

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)
    # Notation:
    # leaf: state of the environment
    # path, rollout: sequences of
    #   triples (state, action, reward).
    for 1...C do
        path, leaf ← SELECT(state)
        rollout, leaf ← EXPAND(leaf)
        UPDATE(path, rollout, leaf)
    return CHOOSE_ACTION(state)
```

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 reinforcement learning (RL) 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.

a) Random Shooting: In this section, we present two instantiations of Algorithm 1, which use Monte Carlo rollouts

[§]Appendix can be accessed under following link https://github.com/shoot-tree-search/sts/blob/master/appendix.pdf

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
    # Function CALCULATE_TARGET is
    # described in Appendix A.1.
    return CALCULATE_TARGET(episode)
```

to evaluate state-action pairs: Random Shooting and Bandit Shooting, see Algorithm 3 and Algorithm 4, respectively.

The simplest version of Algorithm 3, the so-called flat Monte Carlo [22], [23], does not use a policy π_ϕ (instead rollouts are uniformly sampled) nor a value function $\mathbf{V}_\theta$ (just truncated sum of rewards $\hat{G} = \sum_{k=1}^{H} \gamma^{k-1} r_k$). The value function is neither used in the experiments with the pre-trained PPO policy in Section IV-A. Bandit Shooting, presented in Algorithm 4, is a Multi-armed Bandits variant of Random Shooting and uses PUCT [7] rule to improve exploration and thus achieve more reliable evaluations of actions.

b) MCTS: Monte Carlo Tree Search (MCTS) is a family of methods, that iteratively and explicitly build a search tree, see [18]. 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 [6]). In this work, we refer to the latter version, using value networks, as MCTS. Finally, `UPDATE` backpropagates these values from the leaf up the tree. After planning, `CHOOSE_ACTION` chooses an action to take by sampling from the empirical child visitation distribution, sharpened by a predefined temperature parameter τ. This is consistent with MuZero, as used in the Atari domain [10]. A basic variant of MCTS is presented in Algorithm 5. More details are provided in Appendix A.5.

c) 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

Algorithm 3 Random Shooting Planner

```
function SELECT(state)
    s ← state
    a ~ π_φ(s, ·)
    leaf, r ← model.STEP(s, a)
    # Here path is a single transition.
    path ← [(s, a, r),]
    return path, leaf
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[0]
    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 arg max_a Q(s, a)
```

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 ← arg max_a (Q(s, a) + c_puct π_φ(s, a) U(s, a))
    leaf, r ← model.STEP(s, a)
    # Here path is a single transition.
    path ← [(s, a, r),]
    return path, leaf
function EXPAND(leaf)
    Same as in Algorithm 3.
function UPDATE(path, rollout)
    Same as in Algorithm 3.
function CHOOSE_ACTION(s)
    return arg max_a N(s, a)
```

added to the search tree. Note a crucial difference between STS and vanilla MCTS using random rollouts: in contrast to the latter, STS adds visited nodes to the tree, so the explored paths can easily be branched out during later planning passes. We call this mechanism *multi-step expansion*. Intuitively, *multi-step expansion* tilts slightly the search towards DFS. Its advantage comes from making "faster advances" towards the solution, though possibly at risk of missing some promising nodes. Our experiments support these intuitions, suggesting a sweet spot around $H = 10$ (see experiments in Table IV-B and

Algorithm 5 MCTS, additionally uses tree structure $tree$, requires exploration weight c_{puct} and action sampling temperature τ

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

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 SELECT_CHILD(s)
  $U(s, a) \leftarrow \sqrt{\sum_{a'} N(s, a')}/(1 + N(s, a))$
  $a \leftarrow \arg\max_a (Q(s, a) + c_{puct} \pi_\phi(s, a) U(s, a))$
  return a

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

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)
  Same as in Algorithm 5.

function CHOOSE_ACTION(s)
  Same as in Algorithm 5.

function UPDATE(path, rollout, leaf)
  # This is equivalent to calling
  # MCTS.UPDATE() for each node in
  # the rollout and the 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
    $N(s, a) \leftarrow N(s, a) + c$
    $Q(s, a) \leftarrow \frac{W(s,a)}{N(s,a)}$
    $s' \leftarrow s$

Appendix A.9.1.)

A similar method was proposed in [12], with two crucial differences. First, [12] uses hard-coded heuristics while we embed STS into the RL training. Second, [12] only the last estimate of value is backpropagated. Our UPDATE backpropagates aggregate value estimates calculated on the rollout of *multi-step expansion*. We weight this update by the rollout length (hence $N(s, a) \leftarrow N(s, a) + c$). We consider our method more natural and, importantly, it presents better experimental results; see also Section A.9.4.

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 IV. 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.

Equipping STS with additional exploration mechanism (see Appendix A.4, A.5, and A.6), can guarantee that every state-action pair will be visited infinitely often. Combining this with Robbins-Monro conditions for learning rate, implies the convergence in tabular case, following the classical tabular Q-learning convergence theorem (see [35]).

IV. Experiments

We tested the spectrum of algorithms presented in Section III on the Google Research Football and Sokoban domains. Those tasks present numerous challenges, which evaluate

various properties of planning algorithms. Our experiments support the hypothesis that STS builds a more efficient search tree. This hypothesis is substantiated by measuring the tree size in isolation (see Table II), the average depth of the tree (see Appendix A.9.1), and comparison to AlphaGo (see Appendix A.9.2). For the training details, a list of hyper-parameters and network architectures, see Appendix A.1, A.2 and A.3, respectively.

We note that in all comparisons, we set the parameters C, H so that MCTS and STS perform the same number of node expansions during each `PLANNER` call; see Algorithm 1. This ensures the same computational budget (i.e. the number of neural network evaluations) and similar memory usage.

In Appendix A.10, we present two thought experiments, supported by formal proofs, where we argue that STS can better handle certain errors in value functions by using the *multi-step expansion*. We show that STS can get quicker to the regions with accurate values during planning and that it is less prone to entering 'decoy' paths. The errors are inevitable during training and when using function approximators.

A. Google Research Football

Google Research Football (GRF) is an environment recently introduced in [2]. 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. As such, it requires both tactical and strategical decision-making. This makes it a suitable benchmark for planning algorithms. A part of GRF is the Football Academy consisting of 11 scenarios highlighting various tactical difficulties, see [2, Table 10] for description. Due to its diversity, the GRF Academy is an excellent testing ground of planning methods listed in Section III, including STS. GRF provides several state representations, including the internal game representation, as well as visual observations. 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.9.

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

Fig. 1. Example from the Google Football League

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 over at least 20 episodes per environment. In Table I we compare STS with various other methods. To the best of our knowledge, no prior work has evaluated model-based methods on Google Research Football. Hence, we provide two baselines: model-free PPO results, reported by the authors of GRF, and model-based AlphaZero implemented by us, with a minor environment-specific modification. To efficiently deal with the large action space, we use a Q-network $Q(s, a)$ instead of a value network $V(s)$ to evaluate all actions at the same time when expanding a leaf. We provide an ablation on this architectural choice in Appendix A.9.

We report the median of the solved rates in at least three runs with different seeds.

a) 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 representations and, as a consequence, two different architectures (MLP and Conv.). The latter used a uniform policy (flat) or a pretrained policy (PPO). For all variants, we set $C = 30$ passes and a planning horizon $H = 10$. More details can be found in Appendix A.1.

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 pretrained 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 have also tested whether mixing the policy with Dirichlet noise (see [7]) and sampling actions 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).

b) 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 and is the best or close to the best on the remaining 3. One can observe that in the academy environments Corner, Counterattack easy and hard, Pass and shoot with keeper, Run to score with keeper, and Single goal vs. lazy, the difference between STS and MCTS, as well as most of the other methods, is substantial. See Table I for exact results. We stress that STS Conv. easily outperforms any other method in one-to-

TABLE I
COMPARISON OF SELECTED ALGORITHMS ON GRF. ENTRIES ARE ROUNDED SOLVED RATES. PPO RESULTS COME FROM [2].

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	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	0.90	**0.87**	0.80	0.73	0.93	**1.00**	0.87	0.70	0.90	0.37	0.67
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	0.87	0.47	0.73	0.60	**1.00**	**1.00**	0.90	0.80	0.93	**1.00**	0.60
	Conv.	0.97	0.41	0.81	0.44	0.97	**1.00**	0.94	0.69	**1.00**	0.91	0.00
MCTS	Conv.	0.81	0.50	0.31	0.31	0.99	**1.00**	0.45	0.89	0.70	0.00	0.00
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**

one comparison across all academies. STS MLP achieves a close second place. These results provide further evidence that STS gives a boost in environments requiring long-horizon planning. This stands in sharp contrast to MCTS, which was not able to achieve impressive results in the considered time budget. We have 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. Our findings are consistent with the recent recommendations of [36, Section 3.2] for the model-free setting. In the STS experiments, we used approx. 0.8M training samples (median).

One possible explanation for the significant performance gain of STS in comparison to MCTS could be that STS visits more states in the same number of planning passes. In Appendix A.9 we show experiments comparing STS with (a) a variant of MCTS with random rollouts in the fashion of AlphaGo [6], and (b) a variant of MCTS exploring the same path from a leaf as STS, but without adding these nodes to the tree. Both variants perform significantly worse than full STS. This shows that the advantage of STS stems partly from changing the shape of the search tree to a more efficient one.

More details about the experiments can be found in Appendix A.8. In Appendix A.9, we provide an extensive set of 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.

B. 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 the irreversibility of certain actions. A typical example is pushing a box into a corner, although 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 PSPACE-complete, see e.g. [37]. Due to these challenges, the game is often used to test reinforcement learning and planning methods.

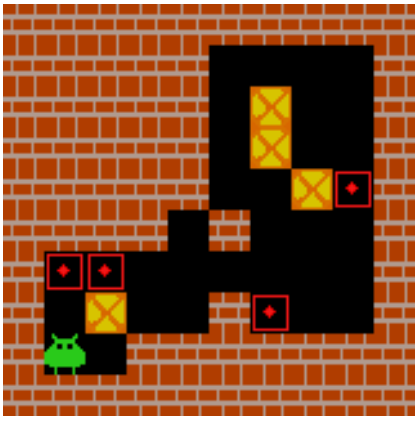


Fig. 2. Example $(10, 10)$ Sokoban board with 4 boxes. Boxes (yellow) are to be pushed by agent (green) to designed spots (red). The optimal solution in this level has 37 steps.

We use procedurally generated Sokoban levels, as proposed by [1]. The agent is rewarded with 1 by putting a box into a designated spot and additonally with 10 when all 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 [9] and Appendix A.5.

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 (trained on MCTS targets) and varied the number of passes C and the depth of multi-step expansion H, such that $H \cdot C$ remains constant. In

TABLE II
COMPARISON OF MCTS AND STS ON SOKOBAN. THE FULL TABLE IS AVAILABLE IN THE APPENDIX (TABLE 4).

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

this way, we ensure a fair comparison because the same computing power is used. In Table II we present statistics that measure the computational cost of planning. In this table, C, H are parameters in Algorithm 1, "S. rate" is the ratio of solved boards, N_p is the average number of planning passes, $N_t (= N_p \cdot H)$ is the average number of nodes in a planning tree, and N_g is the average number of game states observed until the solution is found.

Arguably, the most important of these is N_t, which denotes the total size of the planning tree used to find the solution. In the two presented scenarios, there is a sweet spot for the choice of H. For this choice, the number of tree nodes, N_t, is the smallest, and even more importantly, we observe an increase in the solved rate. This might 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.

To better understand where the differences in performance stem from, we evaluated MCTS with 50 passes and STS with 10 passes and 5 steps of multi-step expansion, both without the loop avoidance mechanism. We found that 68% of boards is solved by both MCTS and STS, 10% only by STS, and 2.1% only by MCTS. On several examples, we observed that STS could recover better than MCTS from errors in the value function, which are relatively localized in the state space, even though they might be quite significant in value. This can be attributed to multi-step expansion, which exits from the erroneous region more quickly by correcting biased value function estimates with deep search (note that deep paths, while introducing more variance, will stronger discount the biased value function). The downside is that sometimes STS is overoptimistic, pushing into dead-end states. We include the detailed analysis on the example room in Appendix A.7.3.

Methods based on random shooting perform poorly for Sokoban: we evaluated Bandit Shooting (Algorithm 4), which struggled to exceed 5% solved rate. Only when the difficulty of the 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.

In the experiments shown here, we have focused on the performance of STS in low computational budgets. In Appendix A.10.1, we also present results for a ten times higher budget: $H \cdot C = 500$.

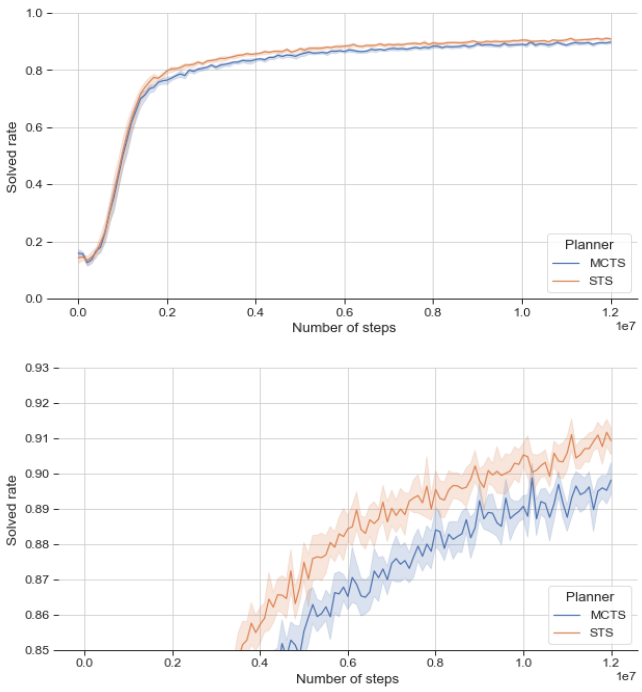


Fig. 3. Learning curve for the Sokoban domain. Left figure shows full results, right one inspects the same data for limited interval of values on the y axis. The results are averaged over 10 runs, shaded areas shows 95% confidence intervals. The x axis is the number of collected samples.

V. 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 the 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 more randomness into planning. Having empirically verified the efficiency of this extension in many challenging scenarios, we argue that it could be included in a standard MCTS toolbox.

ACKNOWLEDGMENTS

The work of Konrad Czechowski, Piotr Januszewski, Piotr Kozakowski and Piotr Miłoś was supported by the Polish National Science Center grant UMO-2017/26/E/ST6/00622. This research was supported by the PL-Grid Infrastructure. Our experiments were managed using https://neptune.ai. We would like to thank the Neptune team for providing us access to the team version and technical support.

REFERENCES

[1] S. Racanière, T. Weber, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. Battaglia, D. Hassabis, D. Silver, and D. Wierstra, "Imagination-augmented agents for deep reinforcement learning," in *NIPS*, 2017.

[2] K. Kurach, A. Raichuk, P. Stańczyk, M. Zając, O. Bachem, L. Espeholt, C. Riquelme, D. Vincent, M. Michalski, O. Bousquet, and S. Gelly, "Google research football: A novel reinforcement learning environment," *arXiv preprint arXiv:1907.11180*, 2019.

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[4] X. Guo, S. P. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline monte-carlo tree search planning," in *NIPS*, 2014.

[5] S. Ross and J. A. Bagnell, "Reinforcement and imitation learning via interactive no-regret learning," *CoRR*, vol. abs/1406.5979, 2014. [Online]. Available: http://arxiv.org/abs/1406.5979

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Van Den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, 2017.

[7] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 1144, pp. 1140–1144, 2018.

[8] T. Anthony, Z. Tian, and D. Barber, "Thinking fast and slow with deep learning and tree search," in *NIPS*, 2017.

[9] P. Miłoś, Ł. Kuciński, K. Czechowski, P. Kozakowski, and M. Klimek, "Uncertainty-sensitive learning and planning with ensembles," *arXiv preprint arXiv:1912.09996*, 2019.

[10] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[11] J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, T. Pfaff, T. Weber, L. Buesing, and P. 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. [Online]. Available: https://openreview.net/forum?id=SkeAaJrKDS

[12] D. J. N. J. Soemers, C. F. Sironi, T. Schuster, and M. H. M. Winands, "Enhancements for real-time monte-carlo tree search in general video game playing," in *IEEE Conference on Computational Intelligence and Games, CIG 2016, Santorini, Greece, September 20-23, 2016*. IEEE, 2016, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CIG.2016.7860448

[13] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, ser. Lecture Notes in Computer Science, H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, Eds., vol. 4630. Springer, 2006, pp. 72–83. [Online]. Available: https://doi.org/10.1007/978-3-540-75538-8_7

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[15] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

[16] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[17] J. E. Doran and D. Michie, "Experiments with the graph traverser program," *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, vol. 294, no. 1437, pp. 235–259, 1966.

[18] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[19] L. Orseau, L. Lelis, T. Lattimore, and T. 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.*, 2018, pp. 3205–3215.

[20] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the rubik's cube with deep reinforcement learning and search," *Nature Machine Intelligence*, vol. 1, no. 8, pp. 356–363, 2019. [Online]. Available: https://doi.org/10.1038/s42256-019-0070-z

[21] B. Abramson, "Expected-Outcome: A General Model of Static Evaluation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1990.

[22] M. L. Ginsberg, "GIB: Imperfect information in a computationally challenging game," *Journal of Artificial Intelligence Research*, 2001.

[23] B. Sheppard, "World-championship-caliber Scrabble," *Artificial Intelligence*, 2002.

[24] E. F. Camacho and C. B. Alba, *Model predictive control*. Springer Science & Business Media, 2013.

[25] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," in *Advances in Neural Information Processing Systems*, 2018, pp. 4754–4765.

[26] A. Nagabandi, G. Kahn, R. S. Fearing, and S. 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*. IEEE, 2018, pp. 7559–7566. [Online]. Available: https://doi.org/10.1109/ICRA.2018.8463189

[27] G. Williams, P. Drews, B. Goldfain, J. M. Rehg, and E. A. Theodorou, "Aggressive driving with model predictive path integral control," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2016, pp. 1433–1440.

[28] G. Williams, A. Aldrich, and E. Theodorou, "Model Predictive Path Integral Control using Covariance Variable Importance Sampling," sep 2015. [Online]. Available: http://arxiv.org/abs/1509.01149

[29] J. Oh, S. Singh, and H. Lee, "Value prediction network," in *NIPS*, 2017.

[30] G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson, "Treeqn and atreec: Differentiable tree planning for deep reinforcement learning," *CoRR*, vol. abs/1710.11417, 2017.

[31] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, R. Sepassi, G. Tucker, and H. Michalewski, "Model-based reinforcement learning for atari," *CoRR*, vol. abs/1903.00374, 2019. [Online]. Available: http://arxiv.org/abs/1903.00374

[32] S. Gu, T. P. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in *ICML*, 2016.

[33] R. Pascanu, Y. Li, O. Vinyals, N. Heess, L. Buesing, S. Racanière, D. P. Reichert, T. Weber, D. Wierstra, and P. Battaglia, "Learning model-based planning from scratch," *CoRR*, vol. abs/1707.06170, 2017.

[34] A. Guez, M. Mirza, K. Gregor, R. Kabra, S. Racanière, T. Weber, D. Raposo, A. Santoro, L. Orseau, T. Eccles, G. Wayne, D. Silver, and T. 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*, 2019, pp. 2464–2473. [Online]. Available: http://proceedings.mlr.press/v97/guez19a.html

[35] J. N. Tsitsiklis, "Asynchronous stochastic approximation and q-learning," *Machine learning*, vol. 16, no. 3, pp. 185–202, 1994.

[36] M. Andrychowicz, A. Raichuk, P. Stanczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem, "What matters in on-policy reinforcement learning? A large-scale empirical study," *CoRR*, vol. abs/2006.05990, 2020. [Online]. Available: https://arxiv.org/abs/2006.05990

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

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 using targets calculated by `CALCULATE_TARGET`. In shooting experiments we use "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. In MCTS and STS in GRF experiments (see Section A.5 for details) we use "tree action-values" targets, similar to the one used in Hamrick et al. (2020); Miłoś et al. (2019).

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. (2019)) and the OpenAI Baselines (Dhariwal et al. (2017)) 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 and substantial amount of tuning experiments (> 3000).

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.

Parameter	Sokoban Shooting	Sokoban MCTS	Sokoban STS	Google Research Football Shooting	Google Research Football MCTS	Google Research Football 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]	0.0	0.0	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	2.5e−4	2.5e−4	2.5e−4	1.0e−4	1.0e−3	1.0e−3
Batch size	32	32	32	64	64	64
Target function[9]	*rew2goT*	*treeT*	*treeT*	*rew2goT*	*treeT*	*treeT*

[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] Softmax temperature. 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.
[9] Indicates how training targets (`CALCULATE_TARGET` in Algorithm 2) are obtained. *rew2goT* and *treeT* corresponds "reward-to-go" and "tree action-values" described in Section A.1.

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

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`
  $P(s,a) \leftarrow (1 - c_{noise})\pi_\phi(s,a) + c_{noise}D$
  $U(s,a) \leftarrow \sqrt{\sum_{a'} N(s,a')}/(1 + N(s,a))$
  $a \leftarrow \arg\max_a (Q(s,a) + c_{puct}P(s,a)U(s,a))$
  $s', r \leftarrow model.\text{STEP}(s,a)$
  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 Silver et al. (2018)) 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. (2019), 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. (2017). 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. □

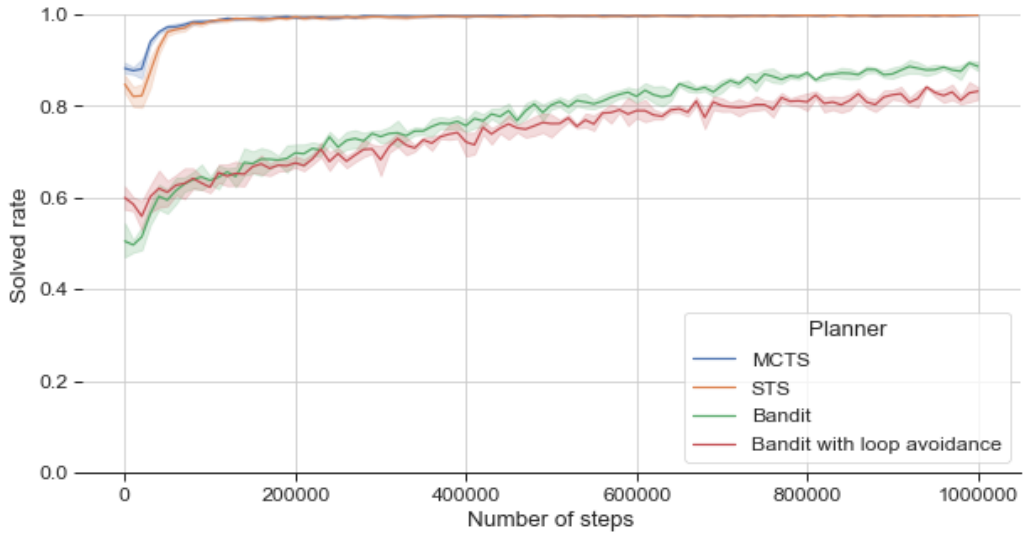


Figure 4: Sokoban on simpler boards: training curves for MCTS, STS and Bandit Shooting with and without loop avoidance. Mean over 5 seeds with shaded regions representing 95% confidence intervals.

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 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.

[1] Our Sokoban code is fully compatible with Racanière et al. (2017).

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

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 4. 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.7.3 CORRECTING BIASED VALUE FUNCTIONS ESTIMATES WITH DEEP SEARCH

To generate value function heatmaps we evaluated the pre-trained MCTS value function for each possible agent position in a room. Figures 5 and 6 present two chosen rooms with their corresponding VF heatmaps. Specifically, the room in Figure 6 was solved by the STS with 10 passes and 5 steps of multi-step expansion and wasn't solved by the MCTS with 50 passes, both without the avoid loops mechanism. We include movies of both agents in this room in the code repository: `https://github.com/shoot-tree-search/sts/tree/master/movies`.

Because the value function is biased, it makes MCTS stuck in states with overestimated value. See Figure 6, in this room MCTS gets stuck in the bottom-left region. However, with a deeper tree, STS can get unstuck quicker and still find a solution. Remember that the search statistics (i.e., W, N, Q) are accumulated for states of the environment (see Appendix A.5). As this overestimated region gets searched deeper the bias in the value function gets discounted more and the agent figures out there are no rewards in reality. At some point, other actions will have a higher value and the agent has a chance to get unstuck and explore other parts of the room. That being said, it should be noted that this "potential well" will still attract the agent, make its planning paths distracted, even when it gets unstuck. STS is less vulnerable to this effect and is able to solve this room despite high bias in the value function.

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

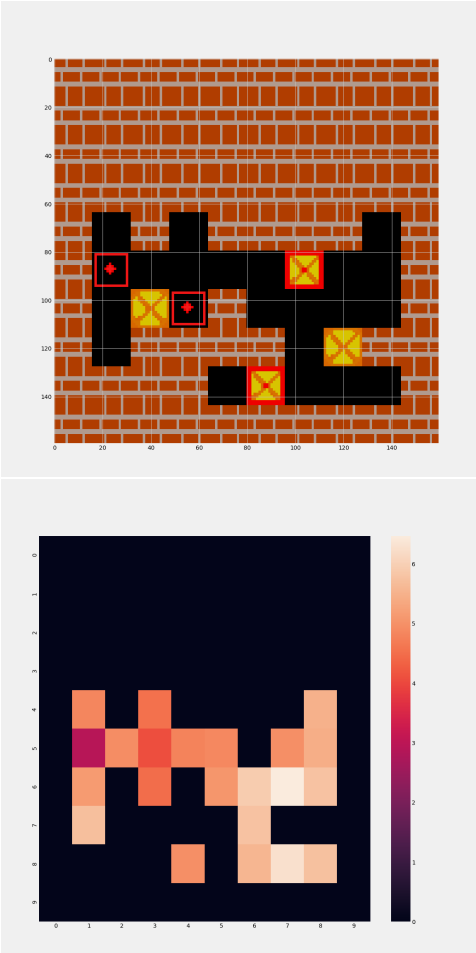


Figure 5: Sokoban value function heatmap, brighter means higher value estimate.

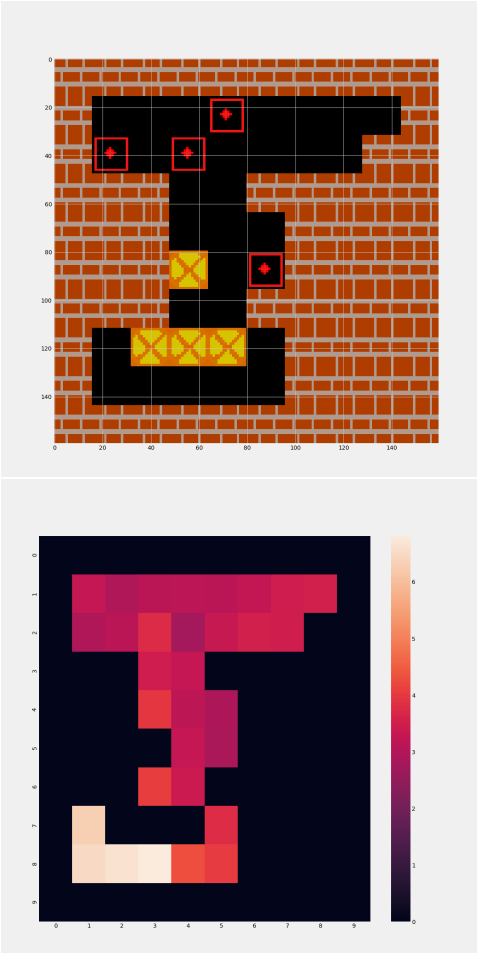


Figure 6: Sokoban value function heatmap, brighter means higher value estimate.

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).

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 Kurach et al. (2019) 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 Kurach et al. (2019), 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 Kurach et al. (2019, Table 1).

Figure 7 shows training curves for our STS agent on Google Research Football. Training was run for 3 days until convergence. 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. Moreover, to smooth the curves, data points are averaged in the windows of 10000 steps.

A.8.1 Shooting methods

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.

A.8.2 MCTS and STS experiments

Apart from *multi-step expansion* we introduced another simple 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 12. This mechanism is similar to recent recommendations of (Andrychowicz et al., 2020, Section 3.2) given in the model-free setting.

A.9 Ablations and analysis

This section is devoted to analysis of various aspects of STS and comparisons to MCTS.

A.9.1 Analysis of the tree depth

Recall that our hypothesis is that the benefits of *multi-step expansion* come from tilting the search towards DFS. While it is hard to formally prove this statement we were able to pin-point this effect in Sokoban and Google Football experiments, see Figure 8 and Figure 9 respectively.

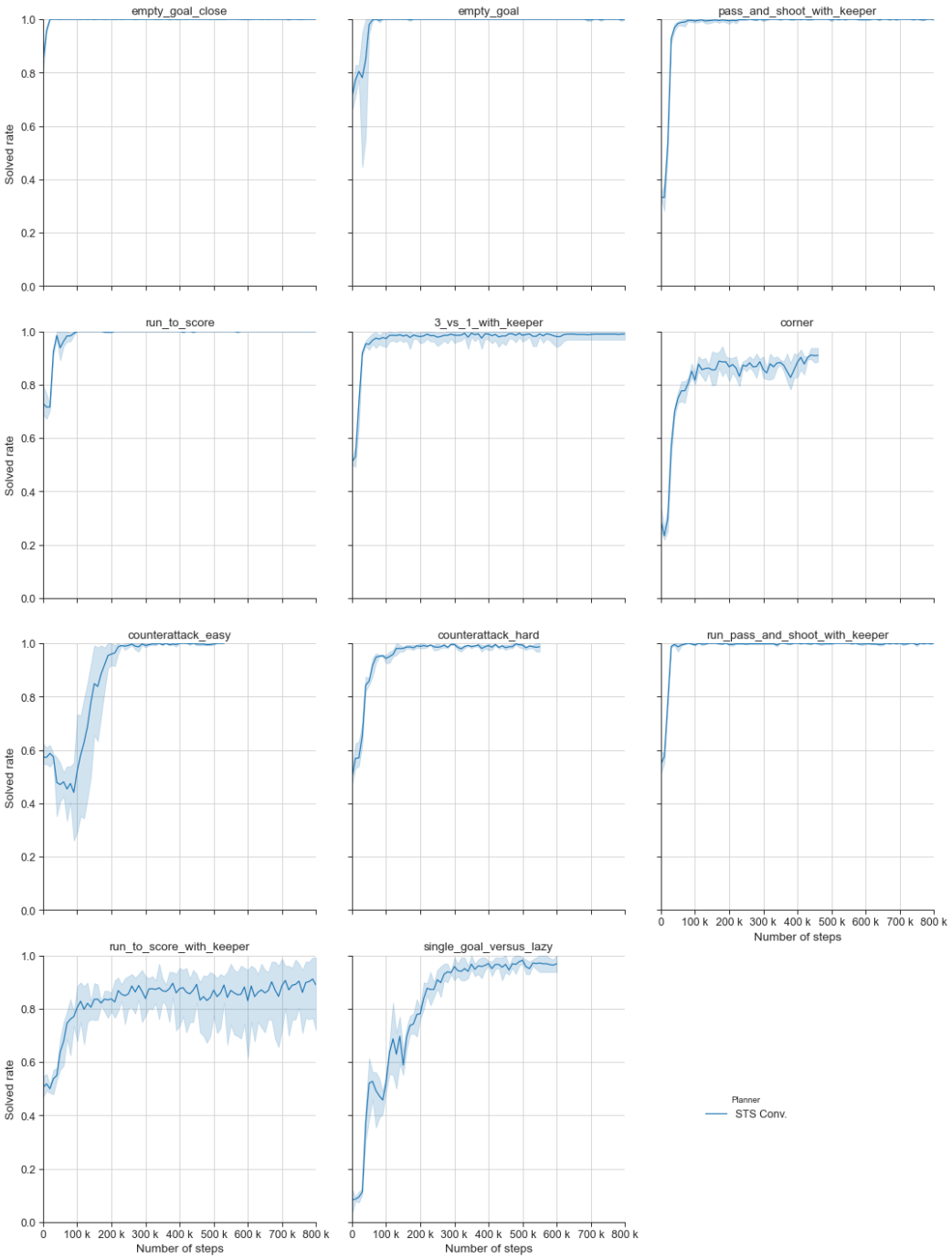


Figure 7: Google Research Football training curves for STS on GRF. Mean over 3 seeds with shaded regions representing 95% confidence intervals.

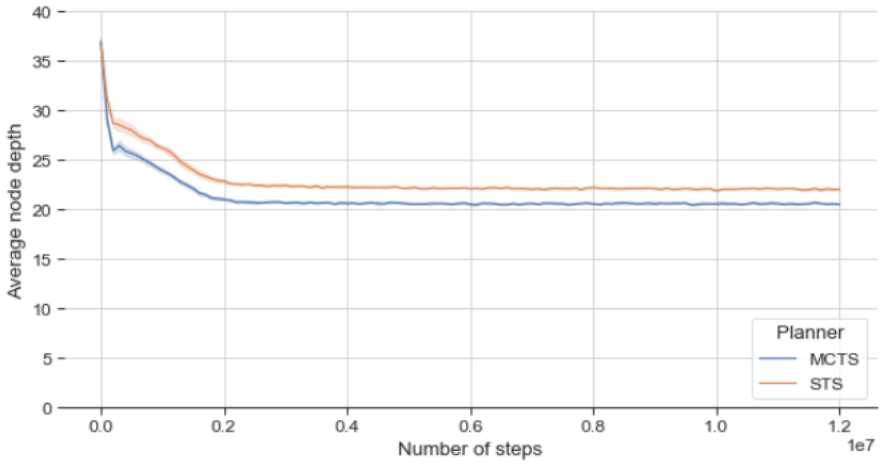


Figure 8: Search depth of MCTS and STS on Sokoban.

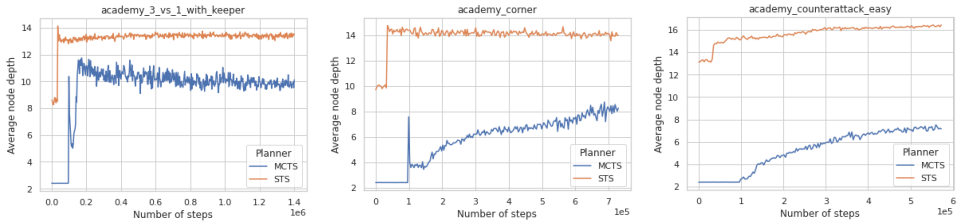


Figure 9: Search depth of MCTS and STS on three Football Academy tasks.

A.9.2 Comparison to AlphaGo

We run two additional experiments with rollout-based evaluation using different policies. These are meant to provide more evidence to support our hypothesis that the benefit of STS is due to the *multi-step expansion* mechanism.

In each experiment, the rollout was truncated after 10 steps to ensure fair comparison with STS. The return after the last step of the rollout was approximated using the Q-value network. This value and rewards collected were used to calculate the leaf's value in the same way as in AlphaGo. In experiments, we tested two strategies for generating rollouts:

1. Actions sampled from the prior policy - the same setup as in AlphaGo, except for rollout truncation and the choice of the policy. AlphaGo used a policy pretrained on expert data. Since we do not have access to such data for Google Football, we instead used the prior policy trained over the course of the algorithm.
2. Actions chosen deterministically, to maximize $Q(s,a) + c_{PUCT} * \pi(a|s)$. Q was computed by a neural network and π is the probability given by the trained prior distribution. We recall that this setup is equivalent to STS except for the crucial fact that STS adds the expanded leaves to the search tree.

We observed that strategy 1. performed very poorly, which highlights the importance of using neural networks for leaf evaluation. Strategy 2. performed significantly better but still worse than STS. We believe that these results strengthen the evidence that the advantage of STS stems from the algorithmic reasons by building a more efficient search tree. Note the tasks on which the two evaluated strategies performed the worst, i.e. counterattack_easy, counterattack_hard, single_goal_versus_lazy, are those with the longest lengths of a successful episode. This supports the argument that STS better handles problems with long planning horizons.

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
MCTS	Conv.	0.81	0.50	0.31	0.31	0.99	**1.00**	0.45	0.89	0.70	0.00	0.00
	Conv.+r.r.	0.91	0.09	0.00	0.03	0.00	**1.00**	**1.00**	**1.00**	0.00	0.06	0.00
	Conv.+d.r.	**1.00**	**0.81**	0.06	0.35	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	0.94	0.25
STS	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: Comparison of STS with leaf evaluation using a Q-value network and MCTS with different leaf evaluations: Q-value network only (Conv.), network + deterministic rollout (Conv.+d.r.), network + random rollout using the prior policy (Conv.+r.r.). In all experiments we used the same convolutional network architecture. The reported results are the median solved rates across 3 runs.

A.9.3 Ablations of STS design choices

The ablations were performed on three environments from GRF Academy: *corner*, *counterattack hard* and *empty goal*, see Figure 12. 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 12):

- `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 12. It uses parameters described in Table 3 in the Google Research Football STS column.

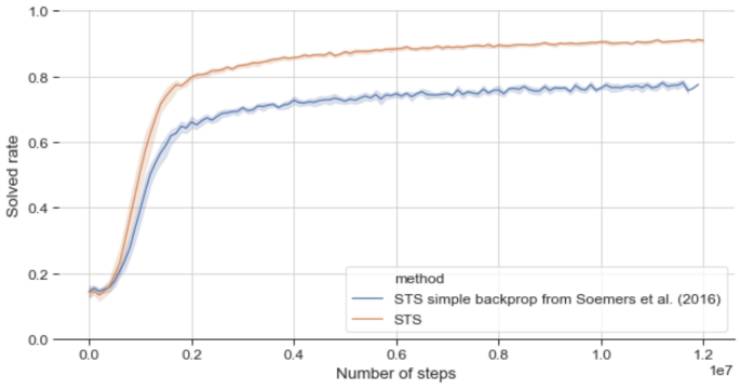
A.9.4 Ablation - backprop weight

This ablation aims to verify if there is benefit of the aggregate backprop implemented in `UPDATE` in Algorithm 6. We compare to the method proposed by Soemers et al. (2016), which backpropagates only the last value. We observed a clear advantage of STS on the Sokoban domain, see Figure A.9.4.

A.10 Multi-step expansion analysis on toy problems

First, consider an MDP presented at the top of Figure 11. 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.

Second, consider an MDP shown at the bottom of Figure 11. It illustrates the case when the errors are "pseudo-random". In this MDP all rewards are 0 except the marked edges, where they are $-a, a > 0$.



Starting from s_0, the agent can move only to the right. 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. 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, 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). In Lemma A.10.1, we show that $p_1 > p_H$ for $H \geq 2$, and in fact $p_H \to 0$ when $H \to +\infty$.

Lemma A.10.1. *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 $p_1 = 0.56, p_2 = 0.46, p_4 = 0.35, p_8 = 0.41, p_{16} = 0.21$. Note that p_1/p_H is as high as 3 for $H = 16$ and quite natural choice of a and ϵ_i.

A.10.1 High computational budgets

STS is predominantly meant to improve the search for modest computational budgets. When the budget gets bigger, any search becomes more exhaustive, and the benefits are likely to diminish. To our pleasant surprise, see Figure 10, we still observe certain advantages of STS in later phases of training, even though MCTS behaves better initially. This observation might suggest another interesting line of inquiry, namely, developing methods for adaptive breadth/depth balancing (e.g. via changing H).

A.11 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.

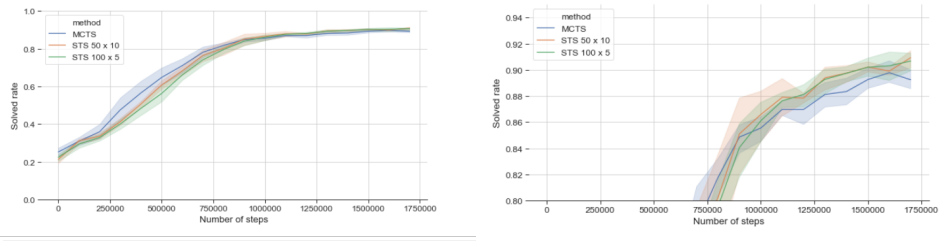


Figure 10: Experiment with big computational budget $C \cdot h = 500$ on Sokoban.

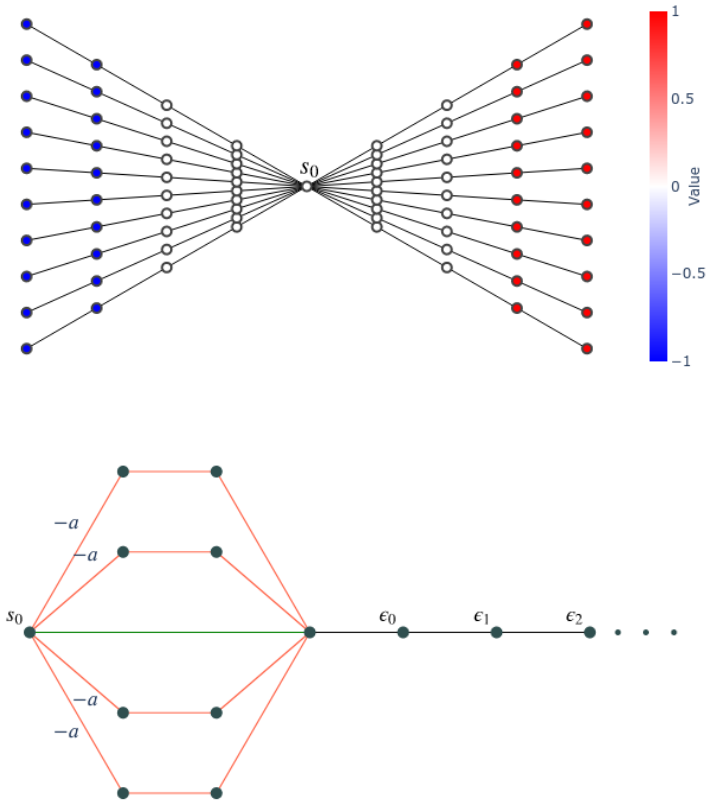


Figure 11: Visualization of the toy environments.

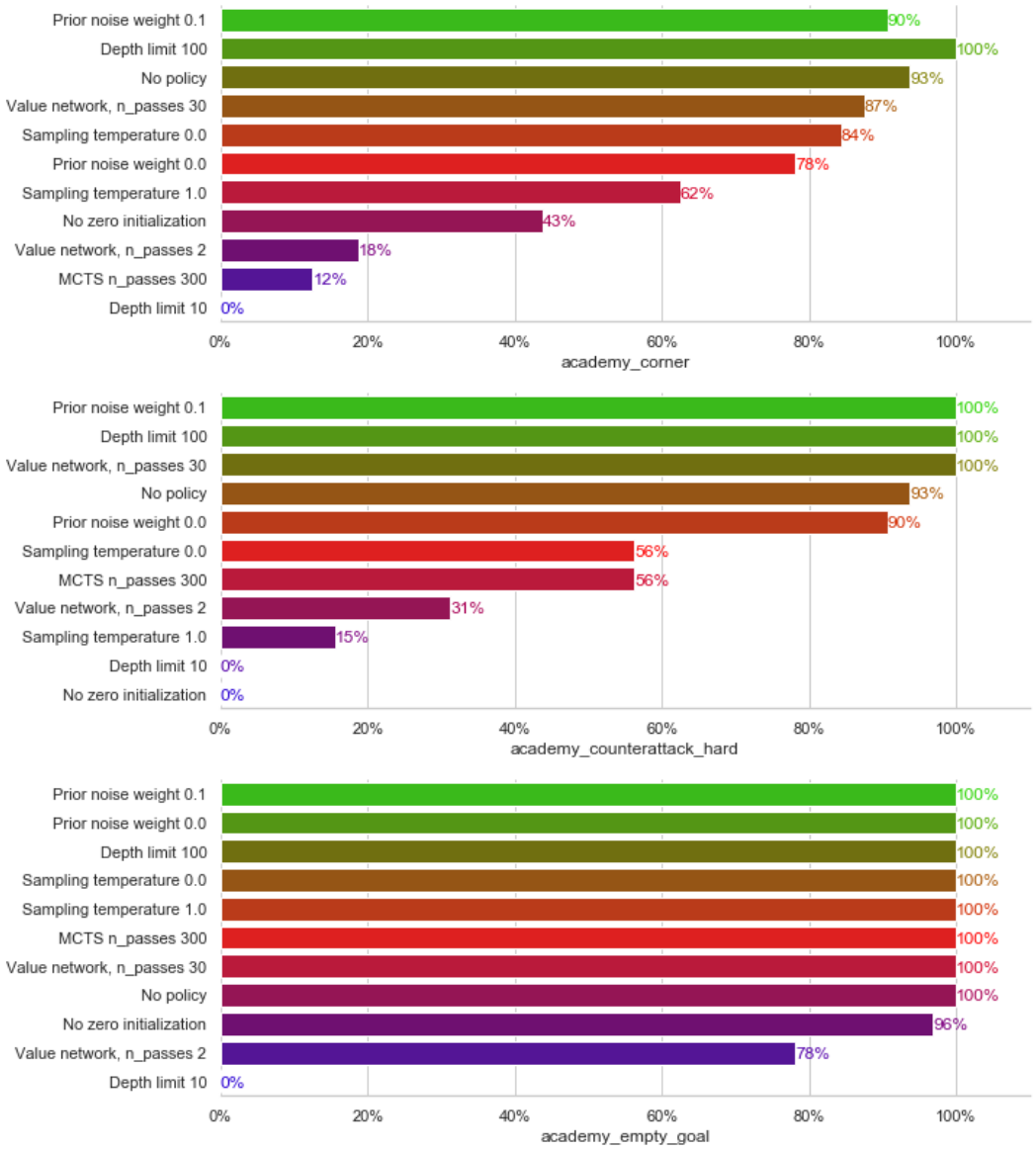


Figure 12: Ablations performed GRF Academy environments: *corner*, *counterattack hard* and *empty goal*.