FAST AND PRECISE: ADJUSTING PLANNING HORIZON WITH ADAPTIVE SUBGOAL SEARCH

Anonymous authors
Paper under double-blind review

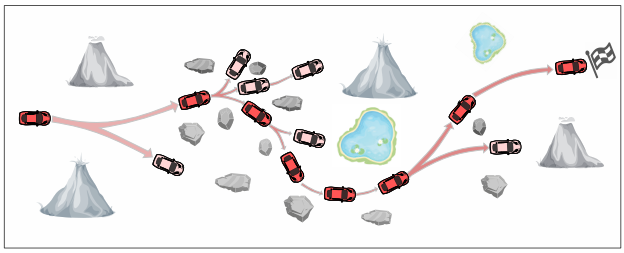
ABSTRACT

Complex reasoning problems contain states that vary in the computational cost required to determine a good action plan. Taking advantage of this property, we propose Adaptive Subgoal Search (AdaSubS), a search method that adaptively adjusts the planning horizon. To this end, AdaSubS generates diverse sets of subgoals at different distances. A verification mechanism is employed to filter out unreachable subgoals swiftly, allowing to focus on feasible further subgoals. In this way, AdaSubS benefits from the efficiency of planning with longer subgoals and the fine control with the shorter ones, and thus scales well to difficult planning problems. We show that AdaSubS significantly surpasses hierarchical planning algorithms on three complex reasoning tasks: Sokoban, the Rubik's Cube, and inequality proving benchmark INT.

1 INTRODUCTION

When solving hard problems, people often try to decompose them into smaller parts that are typically easier to complete Hollerman et al. (2000). Similarly, *subgoal search methods* aim to solve complex tasks by considering intermediate subgoals that advance towards the goal. Besides their intuitive appeal, such approaches offer many practical advantages. Most notably, they allow for deeper search within a smaller computational budget and reduce the negative impact of approximation errors. *Subgoal search* methods powered by deep learning have shown promising results for continuous control tasks, such as robotic arm manipulation Nair & Finn (2020); Jayaraman et al. (2019); Nasiriany et al. (2019); Fang et al. (2019); Pertsch et al. (2020b); Liu et al. (2020a); Zhang et al. (2020) and navigation Kim et al. (2019); Stein et al. (2018); Gao et al. (2017); Savinov et al. (2018). Recently, Czechowski et al. (2021) showed that the usage of a subgoal generator can significantly improve search efficiency on discrete domains with high combinatorial complexity.

This paper builds upon the following observation: many complex reasoning problems contain states that vary in complexity, measured by the computational cost required to determine a good action plan. To illustrate this, imagine driving a car. When traversing a narrow, winding street, it is crucial to focus on the closest events: the next turn, the next car to avoid, etc. However, after entering a straight, empty street, it is enough to think about reaching its far end. This suggests that careful balancing of the subgoal distance is desirable: choose longer subgoals, if possible, to advance faster towards the goal, and choose shorter subgoals to power through the harder states. Hence, the question arises whether it is possible and, if so, how to incorporate this adaptive subgoal generation procedure into subgoal search methods. In this paper, we answer this question affirmatively.



An illustrative example of adaptive planning. The planner may choose long-distance subgoals in the easier areas (e.g. the left most part) and use short distances in the hard areas (e.g. middle part).

We propose a novel planning algorithm *Adaptive Subgoal Search* (AdaSubS), which adaptively selects amongst subgoals with different horizons. Our method benefits both from the efficiency of

planning with longer subgoals and from the reliability of the shorter ones. AdaSubS *prioritizes further distances*, retracting to shorter ranges only when stuck. Additionally, we introduce a *verifier network*, which assesses whether the proposed subgoal is valid and reachable. The verifier helps to efficiently discard faulty subgoals, which are common and more costly to detect in longer horizons. AdaSubS is a data-driven algorithm whose key components are implemented as learnable deep models. In most cases, we use general-purpose transformer architectures to model subgoal generators and the verifier networks. We train those models on offline data.

We show the effectiveness of AdaSubS in three challenging domains: Sokoban, Rubik's Cube, and the inequality theorem prover INT Wu et al. (2020). AdaSubS significantly surpasses hierarchical planning algorithms and sets new state-of-the-art on INT.

Our main contributions are:

1. We propose Adaptive Subgoal Search (AdaSubS), a new algorithm that adjusts the planning horizon to take into account the varying complexity of the state space.
2. We present a comprehensive study of adaptive methods, showing that they typically outperform non-adaptive ones. Amongst these, AdaSubS is the best choice across environments and planning budgets.
3. We also observe a strong indication of out-of-distribution generalization. AdaSubS trained on the proof of length 15 in INT (longest considered in the literature so far) retains more than 50% of its performance when the proof length is increased two-fold.

The code of our method is available at https://github.com/AdaptiveSubgoalSearch/adaptive_subs.

2 Related work

The combination of planning algorithms with deep learning is an active area of research. It provided impressive results e.g. in automated theorem proving Polu & Sutskever (2009), games of chess and Go Silver et al. (2017), Atari benchmark Schrittwieser et al. (2019) and video compression Mandhane et al. (2022).

In the field of hierarchical planning, majority of deep-learning-based methods has focused on visual domains Kim et al. (2019); Nasiriany et al. (2019); Pertsch et al. (2020a); Nair & Finn (2020); Jayaraman et al. (2019); Fang et al. (2019); Pertsch et al. (2020b) or on landmark-based navigation methods Savinov et al. (2018); Liu et al. (2020a); Gao et al. (2017); Stein et al. (2018); Zhang et al. (2020). This body of work often relies on variational autoencoders for the compression of visual observations and uses planning mechanisms suitable for continuous control settings.

There exist many approaches to hierarchical planning utilizing different temporal distances. Kim et al. (2019) and Pertsch et al. (2020b) use hierarchical variational models to learn the temporal structure of tasks by reconstructing the visual state sequences. Pertsch et al. (2020a); Parascandolo et al. (2020); Jurgenson et al. (2020) recursively construct a plan by generating subgoals in the middle between the existing ones. These works have been shown to work on domains with limited combinatorial complexity. They often assume being able to generate the middle state, and thus, arguably, unlikely to cover harder scenarios.

Recently, Czechowski et al. (2021) has shown how combinatorially complex domains can be treated with a hierarchical planning method. Their approach shares similarities with our Adaptive Subgoal Search; however, it cannot address variable environment variable complexity. By using the mechanism of adaptive selection of the subgoal generation distance and verifier, we successfully tackle this problem, confirmed by significant performance gains over Czechowski et al. (2021).

Our verifier is based on an idea similar to Cobbe et al. (2021). They propose a verifier network that is used to assess the correctness of sequences sampled from the language model. Kurutach et al. (2018) uses a verification mechanism to approximate distances between states and optimize plans in latent space with gradient descent in continuous control domains. Functionally, our verifier resembles an affordance function from Ahn et al. (2022), which is used there to score the feasibility of hypothetical robotic plans.

3 METHOD

In this work, we propose Adaptive Subgoal Search (AdaSubS), a subgoal-based search algorithm designed to solve tasks that can be formulated as a search over a graph with a known transition model. AdaSubS is the best choice stemming from a careful study of methods based on the principle of mixing subgoal distances, see Section 4.4 for their definitions and empirical comparisons.

AdaSubS (see Algorithm 1) utilizes the following key components: *subgoal generators*, *verifier*, *conditional low-level policy* (CLLP), and *value function*. These components are implemented using trained neural networks (see Appendix B). To solve a task, AdaSubS iteratively builds a tree of subgoals reachable from the initial state until the target state is reached or the search budget is depleted. In each iteration, it chooses a node in the tree that is expanded by one of the generators. The chosen generator creates a few subgoal candidates, i.e., states expected to be a few steps closer to the target than the current node. For each of them, we use the verifier and CLLP to check whether they are valid and reachable within a few steps. For the correct subgoals, we compute their value function, place them in the search tree, and the next iteration follows.

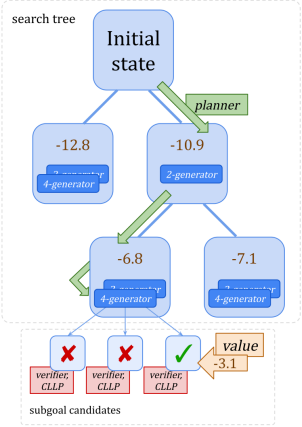


Figure 1: An example iteration of the search performed by AdaSubS.

AdaSubS follows the general structure of Best-First Search BestFS (BestFS); thus the key design decision is how we prioritize nodes to expand and choose generators to produce subgoals. We defer the answer to these questions after providing details of the algorithm components (see also Appendix G).

Subgoal generators. Subgoal generator, or more precisely k-subgoal generator, takes a state as input and returns a diverse set of new candidate states expected to be k step closer to the solution. The key trade-off, which AdaSubS needs to address, is that further subgoals, i.e., those for higher values of k, advance faster towards the target but are also increasingly harder to generate and verify. We typically use a few (e.g., 3) generators.

Conditional low-level policy (CLLP). CLLP returns a path of low-level actions between two states (see Algorithm 2). CLLP calls iteratively a conditional low-level policy network (PN). PN takes as input the current and target states and returns an action. It is possible that CLLP is not able to reach the target, in which case an empty sequence is returned. The role of CLLP is two-fold: it serves as a mechanism allowing AdaSubS to transition between subgoals, and together with the verifier network, it is used in the subgoal verification algorithm (see Algorithm 3).

Verifier. The verifier network is used in the verification algorithm (see Algorithm 3), to answer the following binary classification question: given a starting state and a goal state, is it possible to reach the latter from the former using conditional low-level policy? Computationally, the evaluation of the verifier network is faster than CLLP. However, since verifier is a binary classifier, we expect two types of error to occur: accept invalid subgoals or reject valid subgoals. To control these error levels, the verification algorithm accepts (resp. rejects) a subgoal if the verifier network values are above (resp. below) a certain threshold. In the remaining case, the algorithm falls back on CLLP to decide whether to keep or discard a given subgoal.

Value function. The value function is a neural network that estimates the negative distance between the current state and the goal state. This information is later used by the planner when selecting the next node to expand.

Adaptive Subgoal Search. The particular way in which AdaSubS chooses nodes to expand and a generator to produce subgoals (see highlighted lines in Algorithm 1) implements an adaptive mechanism adjusting the planning horizon. The key difficulty to tackle is that further subgoals, despite being capable of advancing the search faster, are more likely to be faulty. Nevertheless, we assume an optimistic approach prioritizing the longer distances (e.g., higher k). If the search using the long steps is stuck, the planner retracts and expands the most promising, high-value nodes with closer, more conservative subgoals. The verifier network helps in mitigating the risks of this strategy, as it allows for the rejection of faulty subgoals efficiently. This way, by traversing easier parts using

fast long-distance subgoals and conservative ones in harder parts, AdaSubS adapts to the variable complexity of the environment.

Algorithm 1 presents a simple implementation of this approach. The nodes in the search tree are placed in a max-priority queue T with keys, being the pairs (k, v) of the next subgoal distance and its estimated value, sorted in the lexicographical order. In this way, Algorithm 1 uses the highest k possible, searching with the BestFS strategy over values. This might get stuck if for the given k all generated subgoal are invalid (faulty or unreachable), in which case we expand for shorter distances. If successful, we go back to generating with the highest value of k. After reaching the target states AdaSubS reconstructs the path of subgoals and fills it with low-level actions, see function LL_PATH in Algorithm 4.

3.1 Training objectives

The components of AdaSubS are trained using a dataset of offline trajectories of subsequent states and actions: $(s_0, a_0), \ldots, (s_{n-1}, a_{n-1}), s_n$. We do not assume that they are perfect; for some of our environments, even randomly generated trajectories turn out to be sufficient. Details on how the data is collected for each domain can be found in Section 4.1 and Appendix B.

Provided with such data, we train the k-generators to map s_i onto s_{i+k}. The value function is trained to map s_i onto $(i - n)$. CLLP is trained to map (s_i, s_{i+d}) onto a_i for every $d \leq k_{\text{max}}$ (k_{max} is the maximal distance of the subgoal generators used).

We note that AdaSubS still works, albeit much worse, if we disable verifier network (e.g. by setting $\texttt{t}_{\texttt{hi}} = 1$ and $\texttt{t}_{\texttt{lo}} = 0$ in Algorithm 3). However, it is a useful setup to gather a dataset of subgoals and their reachability verified by CLLP. This dataset is used to train the verifier network. For details see Appendix D.

For INT and Rubik's Cube, we use transformer models for all the key components. For the Sokoban, we utilize convolutional networks, for details see Appendix B.

Algorithm 1 Adaptive Subgoal Search

Requires:
- C_1 max number of nodes
- V value function network
- $\rho_{k_0}, \ldots, \rho_{k_m}$ subgoal generators
- SOLVED predicate of solution

```
function SOLVE(s_0)
    T ← ∅
        ▷ priority queue with lexicographic order
    parents ← {}
    for k in k_0, ..., k_m do
        T.PUSH(((k, V(s_0)), s_0))
    seen.ADD(s_0)                    ▷ seen is a set
    while 0 < LEN(T) and LEN(seen) < C_1 do
        (k, _), s ← T.EXTRACT_MAX()
        subgoals ← ρ_k(s)
        for s' in subgoals do
            if s' in seen then continue
            if not IS_VALID(s, s') then
                continue
            seen.ADD(s')
            parents[s'] ← s
            for k in k_0, ..., k_m do
                T.PUSH(((k, V(s')), s'))
            if SOLVED(s') then
                return LL_PATH(s', parents)
                    ▷ get low-level path, see Alg. 4
    return False
```

Algorithm 2 Conditional low-level policy

Requires:
- C_2 steps limit
- π conditional low-level policy network
- M model of the environment

```
function GET_PATH(s_0, subgoal)
    step ← 0
    s ← s_0
    action_path ← []
    while step < C_2 do
        action ← π.PREDICT(s, subgoal)
        action_path.APPEND(action)
        s ← M.NEXT_STATE(s, action)
        if s = subgoal then
            return action_path
        step ← step + 1
    return []
```

Algorithm 3 Verification algorithm

Requires:
- v verifier network
- $\texttt{t}_{\texttt{hi}}$ upper threshold
- $\texttt{t}_{\texttt{lo}}$ lower threshold

```
function IS_VALID(s, s')
    if v(s, s') > t_hi then return True
    else if v(s, s') < t_lo then return False
    return GET_PATH(s, s') ≠ []
```

4 EXPERIMENTS

We empirically demonstrate the efficiency of Adaptive Subgoal Search on three complex reasoning domains: Sokoban, Rubik's Cube, and the inequality proving benchmark INT Wu et al. (2020). We demonstrate that AdaSubS is the best choice in a family of adaptive methods. Interestingly, even weaker methods in this class also outperform non-adaptive baselines. Finally, we show that AdaSubS has strong out-of-distribution generalization properties on INT.

As the performance metric, we use the success rate, defined as the fraction of solved problem instances. The computational budget is defined as the graph size, i.e., the number of nodes visited during the search and evaluated with a neural network (subgoal generator, value function, verifier, or conditional low-level policy). In Appendix C we provide details concerning the number of neural network calls and wall-time evaluations.

4.1 EXPERIMENTAL DOMAINS AND DATASETS

Sokoban is a puzzle in which the goal is to push boxes on target locations. It is a popular testing ground for classical planning methods Lipovetzky & Geffner (2012), and deep-learning approaches Guez et al. (2019); Miłoś et al. (2019). Sokoban is considered to be hard Fern et al. (2011) due to its combinatorial complexity. Finding a solution for a given Sokoban board is an NP-hard problem. In our experiments we used 12×12 Sokoban boards with four boxes.

Rubik's Cube is a celebrated 3D puzzle with over 4.3×10^{18} possible configurations. Recently Agostinelli et al. (2019); Czechowski et al. (2021) have developed methods for solving Rubik's Cube using neural networks.

INT is a benchmark for automated theorem proving proposed by Wu et al. (2020). It consists of a generator of mathematical inequalities and a tool for proof verification. An action (proof step) in INT is a string containing an axiom and a specification of its input entities, making the action space effectively infinite and thus challenging search algorithms.

To collect offline trajectories datasets for Rubik's Cube, we generate random paths of length 20 starting from the solved cube and take them in reversed order. For INT we use the generator provided by Wu et al. (2020). For Sokoban, we use the expert data generated by a reinforcement learning agent Miłoś et al. (2019). Detailed information is contained in Appendix D.

4.2 PROTOCOL AND BASELINES

Our protocol consists of three stages. In the first one, an offline dataset is prepared, see Section 4.1 and Appendix D. Secondly, we use this dataset to train the learnable components of AdaSubS: the family of subgoal generators, verifier network, and value network, see Section 3.1. Evaluation is the final step in which the algorithm's performance, measured by the success rates, is calculated.

As baselines, we use BestFS and kSubS. The former is a well-known class of search algorithms (including A^*), which, among others, performs strongly on problems with high combinatorial complexity Pearl (1984), achieves state-of-the-art results in theorem proving Polu & Sutskever (2009), and strong results on Rubik's Cube Agostinelli et al. (2019); Czechowski et al. (2021).

kSubS is the first general learned hierarchical planning algorithm shown to work on complex reasoning domains Czechowski et al. (2021) (called BF-kSubS there), attaining strong results on Sokoban and Rubik's Cube, and INT. kSubS can be view as a non-adaptive version of AdaSubS realized by a suitable hyperparameters choice: a single subgoal generator and inactive verifier ($\mathtt{t_{lo}} = 0$,$\mathtt{t_{hi}} = 1$).

For details on the hyperparameter choice for our method and the baselines see Appendix F.

4.3 MAIN RESULTS: IN- AND OUT- OF DISTRIBUTION PERFORMANCE

AdaSubS shows overall strong in- and out- of distribution performance. The results for the former regime are presented in Figure 2, which shows that AdaSubS is able to make use of search capacity in the most effective manner, practically dominating other methods across the graph size spectrum. Taking a closer look at the low computational budgets, one can observe that AdaSubS achieves significantly positive success rates while the competing methods struggle. Perhaps the most striking

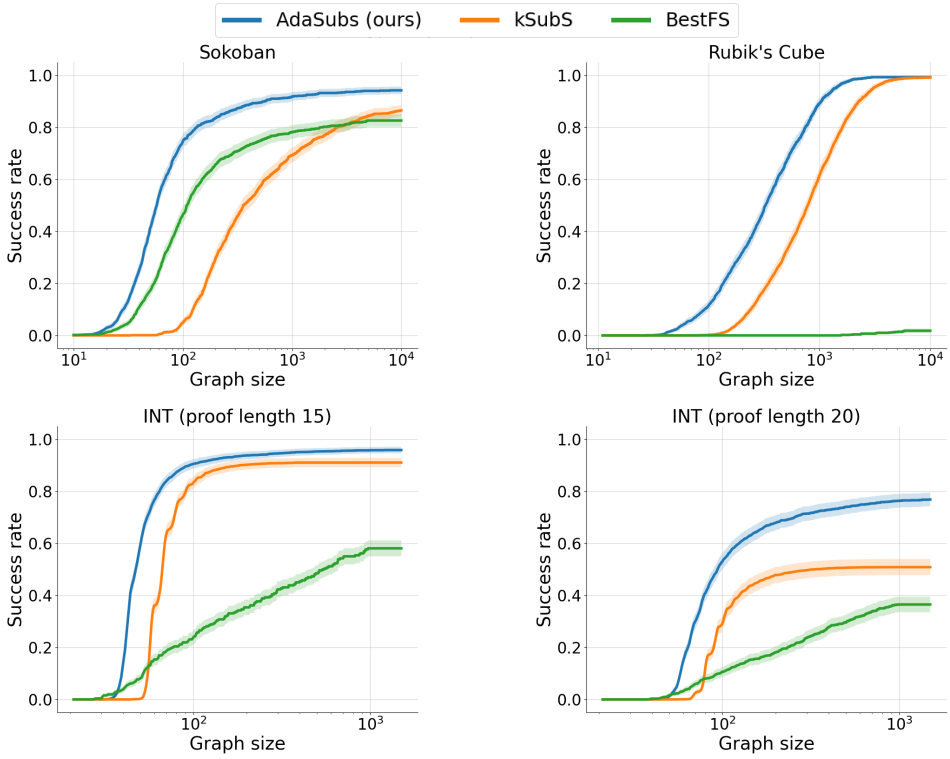


Figure 2: The success rates of AdaSubS, kSubS, and BestFS expressed in terms of graph size. The figure in the bottom right shows the out-of-distribution performance of methods evaluated on INT with the proof length 20 but trained on length 15. The remaining figures present the in-distribution performance. Shaded areas indicate 95% confidence intervals.

difference is observed for INT, where at the budget of 50 nodes AdaSubS achieves around 60% success rate, while kSubS has the success rate close to zero and BestFS does not exceed 10%. This is particularly impressive since the budget of 50 nodes is only slightly larger than three times the considered proof length. To summarize, AdaSubS performs well in low computational regimes, which can be helpful in systems that need to solve search problems under compute or memory constraints.

At the far end of the computational budget spectrum, AdaSubS still performs the best, achieving above 90% performance in each environment ($\sim$ 95% for INT, 100% for Rubik's Cube, and 93% for Sokoban). Importantly, when success rates are high, and consequently the absolute differences between methods' results seem to be low, it is instructive to think about failure rates instead. For instance, in the case of INT (the proof length 15) the failure rate of kSubS is 9%, almost twice the failure rate of AdaSubS. For more results on low and high budgets, see Tables 10-12 in Appendix H.4.

For the out-of-distribution analysis, we used INT, an environment designed to study this phenomenon. Namely, we investigate how methods trained on the proofs of length 15 perform on problems with longer proofs, see Figure 3. The length 15 is the longest considered in the literature Czechowski et al. (2021), and we go much further, studying the proof lengths up to 28. AdaSubS retains more than 50% of its performance, suffering a relatively slow decay of 3.5% (on average) per one step of the proof. This stands in stark contrast to kSubS, which already loses half of its performance at the proofs of length 21. AdaSubS not only outperforms kSubS at each difficulty level but also achieves the most significant advantage on the hardest problems. As an additional study, we provide a full profile of the success rate with respect to the graph size for the proof length 20; see bottom right corner of Figure 2. Adaptive Subgoal Search performs much better than the baselines, with the biggest advantage for large budgets.

The performance boost of AdaSubS over the baselines stems from two components: the verifier and the adaptativeness of subgoal selection. The former allows to assign a bigger fraction of a

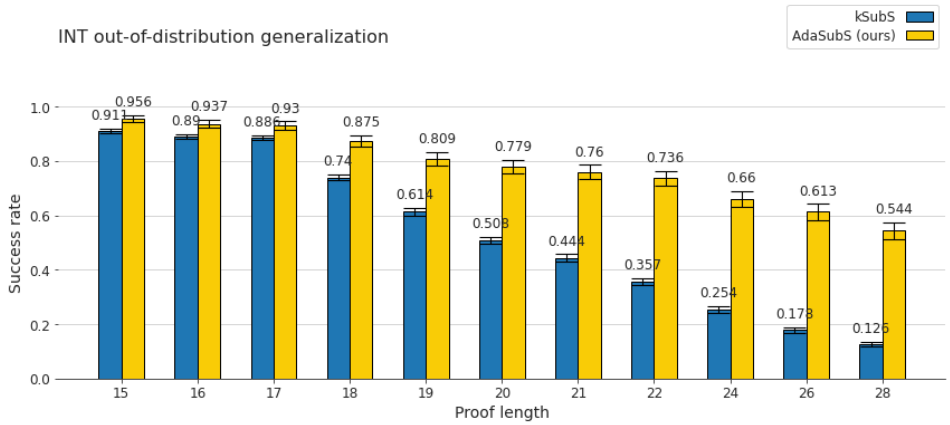


Figure 3: Out-of-distribution performance of AdaSubS and kSubS for long proofs in INT with budget of 5000 nodes. Both methods were trained on proofs of length 15. Error bars correspond to 95% confidence intervals.

computational budget on search, by recovering a part of it from CLLP. This could in principle already provide significant gain to the method. However, as shown in Table 1 and Tables 10-12 in Appendix H.4, verifier helps, but only to a limited degree. Consequently, the majority of the improvement stems from the algorithmic novelty offered by the adaptive procedure. The adaptivity mechanism in AdaSubS creates this interesting dynamic that incentives the method to be optimistic about choosing the subgoal distances while providing the safety net in case this optimism fails. How it works in practice can be nicely seen in Sokoban, where AdaSubS uses 8-subgoals 91.8% of the time, 4-subgoals 7.4% of the time, and 2-subgoals the remaining 0.8% of the time[1].

As a final note, Figure 2 can be used to infer the computational budget required for achieving a certain success rate. Additionally, the ratio of success rate to the graph size can measure the efficiency of the chosen budget, while the derivative of the success rate with respect to graph size provides the marginal utility of the increase in the budget.

4.4 Developing adaptive search methods

In this section, we present a comprehensive empirical study of four adaptive methods. This lets us draw two main conclusions: adaptive search methods outperform non-adaptive ones, and *Longest-first* (on which AdaSubS is based) is the most efficient adaptive procedure. We present full results for INT, the hardest environment, and shortened results for Rubik and Sokoban, see Table 1. The complete set of numerical results, along with an extended discussion, can be found in Appendix H.

The four adaptive methods presented in this section are implemented using the search method[2]. Their adaptivity mechanism is defined by setting the way the subgoals are generated and the order in which the states are processed. This happens in two distinguished lines in Algorithm 1 and changing them determines how the search prioritizes various distances.

Specifically, we designed and tested the following methods: *MixSubS*, *Iterative mixing*, *Strongest-fist*, *Longest-fist*. Each method uses a set of n generators $\rho_{k_1}, \ldots, \rho_{k_n}$ trained to produce subgoals on different distances $k_1 < \ldots < k_n$ (recall Section 3.1 for training details). A more detailed description of the methods (and pseudocodes) can be found in Appendix H.

[1]Additionally, the 8-generator, the 4-generator, and the 2-generator generate subgoals that are on average 6.9, 3.9, and 2.0 steps away, respectively. For AdaSubS parameters see Table 7 in Appendix F.

[2]Adaptivity may also be implemented using the subgoal generator. We considered various approaches in this category, however, they did not perform better than the non-adaptive baseline kSubS, see Appendix H.3. We speculate that assessing the state difficulty is a hard learning problem; thus, it might be easier to handle it by a search mechanism.

INT		Small budget (50 nodes)		Large budget (1000 nodes)	
		with verifier	without	with verifier	without
BestFS		-	1.7%	-	36.7%
kSubS	$k = 4$	2.2%	0.1%	82.4%	83.0%
	$k = 3$	4.0%	0.2%	89.6%	90.7%
	$k = 2$	2.1%	0.5%	89.8%	91.7%
	$k = 1$	0.0%	0.0%	34.7%	46.0%
MixSubS	$k = [4, 3, 2]$	0.0%	0.0%	94.6%	95.0%
	$k = [3, 2, 1]$	0.0%	0.0%	92.2%	92.9%
	$k = [3, 2]$	17.0%	14.8%	92.2%	93.5%
Iterative mixing	iterations $= [1, 1, 1]$	32.0%	30.1%	87.0%	88.6%
	iterations $= [10, 1, 1]$	43.0%	44.8%	95.1%	96.0%
	iterations $= [4, 2, 1]$	54.0%	52.1%	93.6%	95.5%
Strongest-first		39.5%	40.8%	88.5%	89.8%
Longest-first		59.0%	51.5%	95.7%	95.5%

Rubik (with verifier)	400 nodes	6000 nodes
BestFS	0.0%	1.8%
kSubS	28.8%	98.6%
MixSubS	49.1%	99.2%
Iterative mixing	50.6%	99.1%
Strongest-first	33.4%	99.0%
Longest-first	58.0%	99.2%

Sokoban (small budget, 100 nodes)	with verifier	without
BestFS	-	45.9%
kSubS	26.0%	4.7%
MixSubS	52.7%	37.7%
Iterative mixing	64.5%	52.6%
Strongest-first	54.6%	41.9%
Longest-first	72.2%	63.4%

Table 1: *(left)* Results for the INT environment. For each case, except specified otherwise, the distances of subgoal generators are $k = [3, 2, 1]$. *(right)* Shortened results for Rubik and Sokoban, for complete results see Table 11 and Table 12. The results were obtained on at least 1000 problems each, which yields $\pm 3\%$ Bernoulli 95% confidence intervals.

- *MixSubS* is the simplest approach, in which for each processed state we generate one subgoal from each generator ρ_{k_i} ($\texttt{subgoals} \leftarrow \cup_{j=1}^{n} \rho_{k_j}(\texttt{s})$). In each iteration, *MixSubS* chooses a state with the highest value estimation $V(s)$ to process.
- *Iterative Mixing* is similar to *MixSubS* and allows for advanced schedules of generators to be used. In the consecutive iterations, the i-th generator is used to expand l_i nodes before switching to the next generator. This allows us to flexibly prioritize the better generators, but at the cost of tuning tuning additional hyperparameters $l_1, \ldots, l_n$. For these reasons, it is not practical, but useful as a reference point.
- *Strongest-first* uses one generator at a time ($\texttt{subgoals} \leftarrow \rho_{k_\ell}(\texttt{s})$), where k_ℓ is the longest distance not previously used in $\texttt{s}$. In each iteration, *Strongest-first* chooses a state with the highest value estimation $V(s)$ to process.
- *Longest-first* prioritizes long subgoals over the whole search procedure. Only if the queue does not contain any nodes with the highest k, it uses subgoals of lower distances. The nodes are processed in the order of their value estimation $V(s)$.

The high-level conclusion, resulting from our empirical evaluation, is that the performance of methods is roughly ordered as follows: *Longest-first* > *Iterative mixing* > *MixSubS* > *Strongest-first* > *kSubS* > *BestFS*.

In more detail, already the simple *MixSubS* works better than the non-adaptive baselines. In particular, it can outperform the maximum of performances of kSubS for each k. This is in line with the intuition that our mixing mechanism can elicit benefits of various distances while avoiding their drawbacks. We conjecture that whenever a single generator begins to struggle, the search advances with the help of another generator, allowing for stable progress. *Iterative mixing* is able to exhibit strong performance; however, it needs tedious schedule tuning for each domain.

Strongest-first and *Longest-first* implement bias towards longer distances. Even though they are quite similar, they display a large performance difference. We speculate that when *Strongest-first* encounters an area with falsely large value estimates, it wastes a lot of compute to examine it with all subgoal distances. On the other hand, *Longest-first* first explores other areas before using shorter subgoals and thus is able to avoid this problem. We stress that these effects are far from being obvious; however, they occur robustly across our test scenarios.

4.5 Quality of AdaSubS components

The effectiveness of AdaSubS depends on the quality of its trainable components, below we present analysis for generators and verifiers.

Generator: k trade-off. The quality of generators deteriorates when k is increased. In Sokoban, nearly 90% of subgoals created with the 4-generator are valid (according to CLLP), however at the same time for the 16-generator, this statistic drops to 50%. Similarly, in Rubik's Cube, about 82% of subgoals proposed by the 4-generator are valid, while the 3-generator has over 99% accuracy. On the other hand, longer distances are beneficial for search as they allow to build sparser search trees and achieve a solution faster. It turns out that the optimal choice of k depends on the search budget. It pays off to be optimistic (i.e., choose long distances) for small budgets, while more prudent choices get the upper hand when more compute is available. Crucially, AdaSubS with multiple generators successfully resolves the trade-off, outperforming every single generator, see Figure 4.

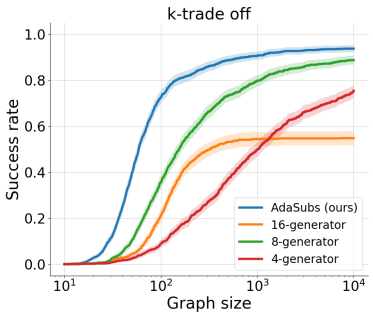


Figure 4: Comparison of success rates for different subgoal generators for Sokoban. AdaSubS-k describes using a single generator with distance k.

Verifier: precision and recall For each subgoal, the verifier outputs a classification probability p, which is used to accept the subgoal (if $p > t_{hi}$), reject the subgoal (if $p < t_{lo}$) or to pass to the further verification (if $p \in [t_{lo}, t_{hi}]$) by CLLP. For the acceptance task, we require high precision, as one false positive can lead to a failure of the whole search procedure. This leads to high value of the corresponding parameter t_{hi} (e.g. for Sokoban $t_{hi} = 0.99$, which corresponds to the precision of 97%). For the rejection task, we do not want to reject true positives, i.e., we aim for high recall. In this case, errors usually increase the search cost; the corresponding parameter t_{lo} is set to $t_{lo} = 0.1$ for Sokoban (this gives the recall of 99%). Additionally, for the verifier to be useful, we want to avoid passing subgoals to the further verification $p \in [t_{lo}, t_{hi}]$. It turns out that for the selected thresholds, the verifier is able to assess 70% states without the CLLP assistance.

5 Limitations and future work

Determinism, access to model dynamics Our main focus is combinatorially complex domains. There are many applications of interest in which we can assume access to underlying dynamics and determinism (for example Automated Theorem Proving). Nevertheless, it is an interesting future direction to adjust our method to stochastic domains and learned models.

Reliance on the offline data In our experiments, we need offline datasets of successful trajectories. We leave for future work developing an algorithm based on the expert iteration paradigm.

Path optimization The goal of our algorithm is to find any path leading to the solution. In many real-world problems, it is also important to find a short path or one with a high reward.

Combine with recursive search methods In some domains, one can generate useful subgoals for long distances and recursively split the problem Pertsch et al. (2020a); Parascandolo et al. (2020); Jurgenson et al. (2020). It would be interesting to propose an algorithm that automatically detects when such an approach is possible and combine two ways (our and recursive) of generating subgoals.

6 Conclusions

We study planning methods that adapt to the local complexity of a solved problem. We concentrate on the adaptive selection of the subgoal distance realized by mixing various subgoal generators. We show that methods based on this principle outperform non-adaptive counterparts They can tackle complex reasoning tasks as demonstrated on Sokoban, the Rubik's Cube, and INT. Our main algorithm, AdaSubS, is the best of the tested choices across all the environments and search budgets. Interestingly, AdaSubS exhibits strong out-of-distribution generalization capability, retaining much of its performance for harder problem instances on INT than it was trained for.

REFERENCES

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.

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Keerthana Gopalakrishnan, Karol Hausman, Alex Herzog, Daniel Ho, Jasmine Hsu, Julian Ibarz, Brian Ichter, Alex Irpan, Eric Jang, Rosario Jauregui Ruano, Kyle Jeffrey, Sally Jesmonth, Nikhil J Joshi, Ryan Julian, Dmitry Kalashnikov, Yuheng Kuang, Kuang-Huei Lee, Sergey Levine, Yao Lu, Linda Luu, Carolina Parada, Peter Pastor, Jornell Quiambao, Kanishka Rao, Jarek Rettinghouse, Diego Reyes, Pierre Sermanet, Nicolas Sievers, Clayton Tan, Alexander Toshev, Vincent Vanhoucke, Fei Xia, Ted Xiao, Peng Xu, Sichun Xu, and Mengyuan Yan. Do as i can, not as i say: Grounding language in robotic affordances, 2022. URL https://arxiv.org/abs/2204.01691.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Konrad Czechowski, Tomasz Odrzygóźdź, Marek Zbysiński, Michał Zawalski, Krzysztof Olejnik, Yuhuai Wu, Łukasz Kuciński, and Piotr Miłoś. Subgoal search for complex reasoning tasks. *Advances in Neural Information Processing Systems*, 34:624–638, 2021.

Kuan Fang, Yuke Zhu, Animesh Garg, Silvio Savarese, and Li Fei-Fei. Dynamics learning with cascaded variational inference for multi-step manipulation. *arXiv preprint arXiv:1910.13395*, 2019.

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

Wei Gao, David Hsu, Wee Sun Lee, Shengmei Shen, and Karthikk Subramanian. Intention-net: Integrating planning and deep learning for goal-directed autonomous navigation. In *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings*, volume 78 of *Proceedings of Machine Learning Research*, pp. 185–194. PMLR, 2017. URL http://proceedings.mlr.press/v78/gao17a.html.

Arthur Guez, Mehdi Mirza, Karol Gregor, Rishabh Kabra, Sébastien Racanière, Théophane Weber, David Raposo, Adam Santoro, Laurent Orseau, Tom Eccles, et al. An investigation of model-free planning. In *International Conference on Machine Learning*, pp. 2464–2473. PMLR, 2019.

Jeffrey R Hollerman, Leon Tremblay, and Wolfram Schultz. Involvement of basal ganglia and orbitofrontal cortex in goal-directed behavior. *Progress in brain research*, 126:193–215, 2000.

Dinesh Jayaraman, Frederik Ebert, Alexei A. Efros, and Sergey Levine. Time-agnostic prediction: Predicting predictable video frames. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=SyzVb3CcFX.

Tom Jurgenson, Or Avner, Edward Groshev, and Aviv Tamar. Sub-goal trees a framework for goal-based reinforcement learning. In *International Conference on Machine Learning*, pp. 5020–5030. PMLR, 2020.

Taesup Kim, Sungjin Ahn, and Yoshua Bengio. Variational temporal abstraction. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 11566–11575, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/b5d3ad899f70013367f24e0b1fa75944-Abstract.html.

Thanard Kurutach, Aviv Tamar, Ge Yang, Stuart J. Russell, and Pieter Abbeel. Learning plannable representations with causal infogan. In Samy Bengio, Hanna M. Wallach,

Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pp. 8747–8758, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/08aac6ac98e59e523995c161e57875f5-Abstract.html.

Nir Lipovetzky and Hector Geffner. Width and serialization of classical planning problems. In *ECAI 2012*, pp. 540–545. IOS Press, 2012.

Kara Liu, Thanard Kurutach, Christine Tung, Pieter Abbeel, and Aviv Tamar. Hallucinative topological memory for zero-shot visual planning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pp. 6259–6270. PMLR, 2020a. URL http://proceedings.mlr.press/v119/liu20h.html.

Yinhan Liu, Jiatao Gu, Naman Goyal, Xian Li, Sergey Edunov, Marjan Ghazvininejad, Mike Lewis, and Luke Zettlemoyer. Multilingual denoising pre-training for neural machine translation. *CoRR*, abs/2001.08210, 2020b. URL https://arxiv.org/abs/2001.08210.

Amol Mandhane, Anton Zhernov, Maribeth Rauh, Chenjie Gu, Miaosen Wang, Flora Xue, Wendy Shang, Derek Pang, Rene Claus, Ching-Han Chiang, et al. Muzero with self-competition for rate control in vp9 video compression. *arXiv preprint arXiv:2202.06626*, 2022.

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

Suraj Nair and Chelsea Finn. Hierarchical foresight: Self-supervised learning of long-horizon tasks via visual subgoal generation. 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=H1gzR2VKDH.

Soroush Nasiriany, Vitchyr Pong, Steven Lin, and Sergey Levine. Planning with goal-conditioned policies. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 14814–14825, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/c8cc6e90ccbff44c9cee23611711cdc4-Abstract.html.

Giambattista Parascandolo, Lars Buesing, Josh Merel, Leonard Hasenclever, John Aslanides, Jessica B Hamrick, Nicolas Heess, Alexander Neitz, and Theophane Weber. Divide-and-conquer monte carlo tree search for goal-directed planning. *arXiv preprint arXiv:2004.11410*, 2020.

Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., 1984.

Karl Pertsch, Oleh Rybkin, Frederik Ebert, Shenghao Zhou, Dinesh Jayaraman, Chelsea Finn, and Sergey Levine. Long-horizon visual planning with goal-conditioned hierarchical predictors. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020a. URL https://proceedings.neurips.cc/paper/2020/hash/c8d3a760ebab631565f8509d84b3b3f1-Abstract.html.

Karl Pertsch, Oleh Rybkin, Jingyun Yang, Shenghao Zhou, Konstantinos G. Derpanis, Kostas Daniilidis, Joseph J. Lim, and Andrew Jaegle. Keyframing the future: Keyframe discovery for visual prediction and planning. In Alexandre M. Bayen, Ali Jadbabaie, George J. Pappas, Pablo A. Parrilo, Benjamin Recht, Claire J. Tomlin, and Melanie N. Zeilinger (eds.), *Proceedings of the 2nd Annual Conference on Learning for Dynamics and Control, L4DC 2020, Online Event, Berkeley, CA, USA, 11-12 June 2020*, volume 120 of *Proceedings of Machine Learning Research*, pp. 969–979. PMLR, 2020b. URL http://proceedings.mlr.press/v120/pertsch20a.html.

Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. corr abs/2009.03393 (2020), 2009.

Nikolay Savinov, Alexey Dosovitskiy, and Vladlen Koltun. Semi-parametric topological memory for navigation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=SygwwGbRW.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *ArXiv*, abs/1911.08265, 2019.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv*, abs/1712.01815, 2017.

Gregory J. Stein, Christopher Bradley, and Nicholas Roy. Learning over subgoals for efficient navigation of structured, unknown environments. In *2nd Annual Conference on Robot Learning, CoRL 2018, Zürich, Switzerland, 29-31 October 2018, Proceedings*, volume 87 of *Proceedings of Machine Learning Research*, pp. 213–222. PMLR, 2018. URL http://proceedings.mlr.press/v87/stein18a.html.

Yuhuai Wu, Albert Jiang, Jimmy Ba, and Roger Grosse. Int: An inequality benchmark for evaluating generalization in theorem proving. *arXiv preprint arXiv:2007.02924*, 2020.

Lunjun Zhang, Ge Yang, and Bradly C. Stadie. World model as a graph: Learning latent landmarks for planning. *CoRR*, abs/2011.12491, 2020. URL https://arxiv.org/abs/2011.12491.