Subgoal Search For Complex Reasoning Tasks

Anonymous Author(s)
Affiliation
Address
email

Abstract

Humans excel in solving complex reasoning tasks through a mental process of moving from one idea to a related one. Inspired by this, we propose Subgoal Search (kSubS) method. Its key component is a learned subgoal generator, an analog of human intuition, that produces a diversity of subgoals that are both achievable and closer to the solution. Using subgoals reduces the search space and induces a high-level search graph suitable for efficient planning. In this paper, we implement kSubS using a transformer-based subgoal module coupled with the classical best-first search framework. We show that a simple approach of generating k-the step ahead subgoals is surprisingly efficient on three challenging domains: two popular puzzle games, Sokoban and the Rubik's Cube, and an inequality proving benchmark INT. kSubS achieves strong results including state-of-the-art on INT within a modest computational budget.

1 Introduction

Reasoning is often regarded as a defining property of advanced intelligence [30, 14]. When confronted with a complicated task, humans' thinking process often moves from one idea to a related idea, and the progress is made through milestones, or *subgoals*, rather than through atomic actions that are necessary to transition between subgoals [12]. During this process, thinking about one subgoal can lead to a possibly diverse set of subsequent subgoals that are conceptually reachable and make a promising step towards the problem's solution. This intuitive introspection is backed by neuroscience evidence [15], and in this work, we present an algorithm that mimics this process. Our approach couples a deep learning generative subgoal modeling with classical search algorithms to allow for successful planning with subgoals. We showcase the efficiency of our method on the following complex reasoning tasks: two popular puzzle games Sokoban and the Rubik's Cube, and an inequality theorem proving benchmark INT [46], achieving the state-of-the-art results in INT and competitive results for the remaining two.

The deep learning revolution has brought spectacular advancements in pattern recognition techniques and models. Given the hard nature of reasoning problems, these are natural candidates to provide search heuristics [4]. Indeed, such a blend can produce impressive results [35, 36, 29, 1]. These approaches seek solutions using elementary actions. Others, e.g. [23, 26, 18], utilize variational subgoals generators to deal with long-horizon visual tasks. We show that these ideas can be pushed further to provide algorithms capable of dealing with combinatorial complexity.

We present Subgoal Search (SubS) method and give its practical implementations: MCTS-kSubS and BF-kSubS. SubS consists of the following four components: planner, subgoal generator, a low-level policy, and a value function. The planner is used to search over the graph induced by the subgoal generator and is guided by the value function. The role of the low-level policy is to prune the search tree as well as to transition between subgoals. In this paper, we assume that the generator predicts subgoals that are k step ahead (towards the solution) from the current state, and to emphasize this we henceforth add k to the method's abbreviation. MCTS-kSubS and BF-kSubS differ in the choice of

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

the search engine: the former uses Monte-Carlo Tree Search (MCTS), while the latter is backed by Best-First Search (BestFS). We provide two sets of implementations for the generator, the low-level policy, and the value functions. The first one uses transformer architecture ([40]) for each component, while the second utilizes a convolutional network for the generator and the value function, and the classical breadth-first search for the low-level policy. This lets us showcase the versatility and effectiveness of the approach.

The subgoal generator lies at the very heart of Subgoal Search, being an analog of human intuition and implementation of reasoning with high-level ideas. To be useful in a broad spectrum of contexts, the generator should be implemented as a learnable (generative) model. As a result, it is expected to be imperfect and (sometimes) generate incorrect predictions, which may turn the search procedure invalid. Can we thus make planning with learned subgoals work? In this paper, we answer this question affirmatively: we show that the autoregressive framework of transformer-based neural network architecture [41] leads to superior results in challenging domains.

We train the transformer with the objective to predict the k-th step ahead. The main advantages of this subgoal objective are simplicity and empirical efficiency. We used expert data to generate labels for supervised training. When offline datasets are available, which is the case for the environments considered in this paper[1], such an approach allows for stable and efficient optimization with high-quality gradients. Consequently, this method is often taken when dealing with complex domains (see e.g. [33, 43]) or when only an offline expert is available[2]. Furthermore, we found evidence of out-of-distribution generalization.

Finally, we formulate the following hypothesis aiming to shed some light on why kSubS is successful: we speculate that subgoal generation may alleviate errors in the value function estimation. Planning methods based on learning, including kSubS, typically use imperfect value function-based information to guide the search. While traditional low-level search methods are susceptible to local noise, subgoal generation allows for evaluations of the value functions at temporally distant subgoals, which improves the signal-to-noise ratio and allows a "leap over" the noise.

To sum up, our contributions are:

1. We propose Subgoal Search method with two transformer-based implementations (MCTS-kSubS, BF-kSubS), and one, simpler, convolutional-based implementation. We demonstrate that our approach requires a relatively little search or, equivalently, is able to handle bigger problems. Interestingly, we observe evidence of out-of-distribution generalization.
2. We show that a transformer-based autoregressive model learned with a simple supervised objective to produce k-th step ahead subgoal already achieves a strong performance.
3. We study the negative influence of value function errors on planning and show that using subgoal planning might mitigate this problem.

We provide the code of our method and experiment settings at `https://github.com/subgoal-search/subgoal-search`.

2 Related work

In classical AI, reasoning is often achieved by *search* ([30]). Search rarely can be exhaustive and a large body of algorithms and heuristics has been developed over the years, [30, Section 3.5]. It is hypothesized that progress can be achieved by combining search with learning [4]. Among notable successful examples of such a blend are Alpha Zero [34], or solving Rubik's cube using a learned function to guide A* search (see [1]).

An eminent early example of using goal-directed reasoning is PARADISE algorithm ([44]). In deep learning literature, [19] was perhaps the first work implementing subgoal planning. This was followed by a large body of work on planning with subgoals in the latent space for visual tasks ([18, 24, 26, 23, 16, 8, 27]) or landmark-based navigation methods ([31], [20], [11], [37], [47]).

[1]The dataset for INT or Sokoban can be easily generated or are publicly available. For the Rubik's Cube, we use random data or simple heuristic (random data are often sufficient for robotic tasks and navigation.)

[2]For example, the INT engine can easily generate multiple proves of random statements, but *cannot* prove a given theorem.

The tasks considered in the aforementioned studies are often quite forgiving when it comes to small errors in the subgoal generation. This can be contrasted with complex reasoning domains, in which small changes to a given state may drastically change its meaning or even render it invalid. Thus, neural networks may struggle to generate semantically valid states ([4, Section 6.1]). Assuming that this problem was solved, a generated subgoal still remains to be assessed. The exact evaluation may, in general, require exhaustive search or access to an oracle (in which case the original problem is essentially solved). Consequently, it is unlikely that a simple planner (e.g., one unrolling independent sequences of subgoals [8])) will either produce an informative outcome or could be easily improved using only local changes via gradient descent [24], or cross-entropy method (CEM) [23, 26, 27]. Existing approaches based on more powerful subgoal search methods, on the other hand, have their limitations. [10] is perhaps the closest to our method and uses an MCTS to search the subgoal-induced graph. However, it uses a predefined (not learned) predicate function as a subgoal generator, limiting applicability to the problems with available high-quality heuristics. Learned subgoals are used in [25], a method of hierarchical planning of subgoal sequences. That said, the subgoal space needs to be relatively small for this method to work (or crafted manually to reduce cardinality).

More generally, concepts related to goals and subgoals percolated to reinforcement learning early on, leading, among others, to prominent ideas like hindsight [17], hierarchical learning [39, 6] or the Horde architecture [38]. Recently, with the advent of *deep* reinforcement learning, these ideas have been resurfacing and scaled up to deliver their initial promises. For example, [42] implements ideas of [6] and a very successful hindsight technique [3] is already considered to be a core RL method. Further, a recent paper [28] utilizes a maximum entropy objective to select achievable goals in hindsight training.

3 Method

Our method is designed for problems, which can be formulated as a search over a graph with a known transition model. Formally, let G be a directed graph and $\tilde{S} \subset G$ be the set of success states. We assume that, during the solving process, the algorithm can, for a given node g, determine the edges starting at g and check if $g \in \tilde{S}$.

Subgoal Search method consists of four components: planner, subgoal generator, low-level policy, and a value function. The planner, coupled with a value function, is used to search over the graph induced by the subgoal generator. Namely, for each selected subgoal, the generator allows for sampling the candidates for the next subgoals. Only these reachable by the low-level policy are used. The procedure continues until the solution is found or the computational budget is exhausted.

In this paper we provide BestFS- and MCTS- backed implementations of kSubS. Algorithm 1 presents BF-kSubS (see Algorithm 6 in Appendix A.1 for MCTS-kSubS), while the listing for low-level policy and the subgoal generator is given in Algorithms 2 and Algorithm 3, respectively.

For INT and Rubik's Cube, we use transformer models (see Appendix B.1) in all components other than the planner. The setting for Sokoban is different (based on convnets), and for clarity, we postpone its description till the end of this section. While transformers could also be used in Sokoban, we show that a simplified setup already achieves strong results. This showcases that Subgoal Search is general enough to work with different design choices. For each environment we collect expert data, see Section 4.2, on which learned models are trained.

Subgoal generator. Formally, it is a mapping $\rho\colon \mathcal{S} \to \mathbf{P}(\mathcal{S})$, where $\mathbf{P}(\mathcal{S})$ is a family of probability distributions over the environment's state space $\mathcal{S}$. More precisely, let us define a trajectory as a sequence of state-action pairs $(s_0, a_0), (s_1, a_1), \ldots, (s_n, a_n)$, with $(s_i, a_i) \in \mathcal{S} \times \mathcal{A}$, where $\mathcal{A}$ stands for the action space and n is the trajectory's length. We will say that the generator predicts *k-step ahead* subgoals, if at any state s_ℓ it aims to predict $s_{\min(\ell+k,n)}$. We show, perhaps surprisingly, that this simple objective is an efficient way to improve planning, even for small values of k, i.e. $k \in \{2, 3, 4, 5\}$.

Operationally, the subgoal generator takes as input an element of the $\mathcal{S}$ and returns *subgoals*, a set of new candidate states expected to be closer to the solution, and is implemented by Algorithm 3. This procedure is parameterized by three constants: C_3, governing the number of subgoals to be generated, C_4, the number of beams in beam search, and C_5, a probability threshold which controls how plausible (according to the model) the subgoal set is. In INT and Rubik's Cube, we model

subgoal generator network (ρ in Algorithm 3) as transformer. It is trained in a supervised way on the expert data, with training examples being: s_ℓ (input) and $s_{\min(\ell+k,n)}$ (label), see Appendix D for details. Other possible approaches to modelling a subgoal generator are discussed in Section 5.

Algorithm 1 Best-First Subgoal Search

Require: C_1 max number of nodes
V value function network
SOLVED predicate of solution

```
function SOLVE(s_0)
    T.PUSH((V(s_0), s_0))          ▷ T is priority queue
    paths[s_0] ← []                ▷ paths is dict of lists
    seen.ADD(s_0)                  ▷ seen is set
    while 0 < LEN(T) and LEN(seen) < C_1 do
        _, s ← T.EXTRACT_MAX()
        subgoals ← SUB_GENERATE(s)
        for s' in subgoals do
            if s' in seen then continue
            seen.ADD(s')
            actions ← GET_PATH(s, s')
            if actions.EMPTY() then
                continue
            T.PUSH((V(s'), s'))
            paths[s'] ← paths[s] + actions
            if SOLVED(s') then
                return paths[s']
    return False
```

Algorithm 2 Low-level conditional policy

Require: C_2 steps limit
π low-level conditional 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 []
```

Low-level (conditional) policy. Formally, it is a mapping $\pi\colon \mathcal{S} \times \mathcal{S} \to \mathcal{A}^*$. It provides a sequence of actions on how to reach a subgoal starting from a given initial state. Operationally, it may return an empty sequence if the subgoal cannot be reached within C_2 steps, see Algorithm 2. This is used as a pruning mechanism for the *subgoals* set in Algorithm 1.

In INT and Rubik's Cube, we implement low-level policy network (π in Algorithm 2) as a transformer (see Appendix B.1 for details). Similarly to the subgoal generator, it is trained using expert data in a supervised fashion, i.e. for a pair $(s_\ell, s_{\min(\ell+i,n)})$, with $i \leq k$, its objective is to predict a_ℓ.

Algorithm 3 Subgoal generator

Require: C_3 number of subgoals
C_4 number of beams in sampling
C_5 target probability
ρ subgoal generator network

```
function SUB_GENERATE(s)
    subgoals ← ∅
    (states, probs) ← BEAM_SEARCH(ρ, s; C_3, C_4)
                ▷ (states, probs) is sorted wrt probs
    total_p ← 0
    for state, p ∈ (states, probs) do
        if total_p > C_5 then break
        subgoals.ADD(state)
        total_p ← total_p + p
    return subgoals
```

Value function. Formally, it is a mapping $V\colon \mathcal{S} \to \mathbb{R}$, that assigns to each state a value related to its distance to the solution, and it is used to guide the search (see Algorithm 1 and Algorithm 6). For INT and Rubik's Cube, we use a transformer architecture, which is trained on expert data, to predict for each state s_ℓ its negated distance to the goal $\ell - n$, where n denotes the end step of a trajectory that s_ℓ belongs to.

Planner. This is the engine that we use to search the subgoal-induced graph. In this paper, we use BestFS (Algorithm 1) and MCTS (Algorithm 6 in Appendix A.1). The former is a classic planning method, which maintains a priority queue of states waiting to be explored, and greedily (with respect to their value) selects elements from it (see, e.g., [30]). MCTS is a search method that iteratively and explicitly builds a search tree, using (and updating) the collected node statistics (see, e.g., [5]). In this paper, we use an AlphaZero-like [33] algorithm for single-player games.

Sokoban setup. Thanks to a low branching factor and small values of k that we consider in this paper, the setup for Sokoban can be simplified. For the neural network used by the subgoal generator (ρ in Algorithm 3) we use a convolutional-network based architecture. Accordingly, we replace the beam search of Algorithm 4 with an alternative approach, better suited for the chosen architecture, see Appendix B.2. The value function used in this implementation comes from a separate MCTS training, for details see Appendix C.1. For a low-level policy, we simply use breadth-first search, instead of Algorithm 2.

4 Experiments

In this section, we empirically demonstrate the efficiency of MCTS-kSubS and BF-kSubS, in particular, we show that they vastly outperform their standard ("non-subgoal") counterparts. As a test ground, we consider three challenging domains: Sokoban, Rubik's Cube, and INT. All of them, require non-trivial reasoning. The Rubik's Cube is a well-known 3D combination puzzle. Sokoban is a complex video puzzle game known to be NP-hard and thus a challenging testbed for planning methods. INT [46] is a recent theorem proving benchmark.

4.1 Training protocol and baselines

Our experiments consist of three stages. First, we collect domain-specific expert data, see Section 4.2. Secondly, we train the subgoal generator, low-level conditional policy, and value function networks using the data and targets described in Section 3. For more details see Appendix D. Eventually, we evaluate the planning performance of MCTS-kSubS and BF-kSubS, details of which are presented below. In the second step, we use supervised learning, which makes our setup stable with respect to network initialization, see details in Appendix D.1.3.

As baselines, we use BestFS and MCTS (single player reimplementation of AlphaZero) operating on elementary actions. In INT and Rubik's Cube both algorithms utilize policy network (trained with behavioral cloning, on the same dataset, which we used to train kSubS). Note that policy network can be seen as a variant of subgoal generator for $k = 1$, since we consider deterministic transition function between states (the distribution over actions can be viewed as a distribution over states). This representation allowed us to reuse the same codebase for baseline search methods and kSubS and ensures a fair comparison. For more details see Appendix H.

4.2 Search Domains and Datasets

Sokoban is a single-player complex game in which a player controls an agent that is able to push boxes across the board. The goal is to place all boxes on target locations solely by pushing them, without crossing any obstacles or walls. Sokoban has recently been used to test the boundaries in RL [13]. Sokoban is known to be hard [9], mainly due to its combinatorial complexity and the existence of irreversible states. Formally, deciding if a given Sokoban board is solvable is a PSPACE-complete problem. [7]

To collect the expert dataset, we use a separate training using an MCTS agent [22] (with implementation made publicly available by the authors). Our dataset consists of all successful trajectories occurring during the training. These are suboptimal, especially in the early phases of the training or for harder boards.

Rubik's Cube is a classical 3-dimensional classic puzzle. It is considered challenging due to the fact that the search space has more than 4.3×10^{18} possible configurations. Similarly to Sokoban, Rubik's Cube has been recently used as a testing ground for RL methods [1].

To collect the expert dataset, we generate random paths of length 30 starting from the solved cube and take them backward. These backward solutions are highly sub-optimal (optimal solutions are proven to be shorter than 26).

INT: Inequality Benchmark for Theorem Proving. INT provides a generator of inequalities, which produces mathematical formulas along with a corresponding proof. Proofs are represented as sequences of consecutively applied mathematical axioms (there is a total number of 18 axioms). Each action is represented as a tuple containing a chosen axiom and its input entities. Action space in this problem can be very large, reaching up to 10^6 elements, which significantly complicates planning.

INT is equipped with the generator of mathematical proofs [46, Section 3.3]. It generates a path by randomly applying axioms starting with a trivial statement. The path taken backward constitutes the proof of its final statement. The path length is an important hyperparameter regulating the difficulty – we use $5, 10, 15$. The proofs generated as follows are not guaranteed to be optimal.

Hyperparameters of BF-kSubS used in specific domains are shown in Table 1. For more details on datasets see Appendix C.

	INT	Sokoban	Rubik
k	3	4	4
C_1	400	5000	1500
C_2	4	4	7
C_3	4	4	3
C_4	16	NA	32
C_5	1	0.98	1

Table 1: BF-kSubS hyperparameters.

4.3 Main results

In this section, we present our most important finding: that MCTS-kSubS and BF-kSubS perform significantly better than the respective methods not using subgoals, including state-of-the-art on INT. Using Subgoal Search enables for more efficient search and consequently scales up to problems with higher difficulty.

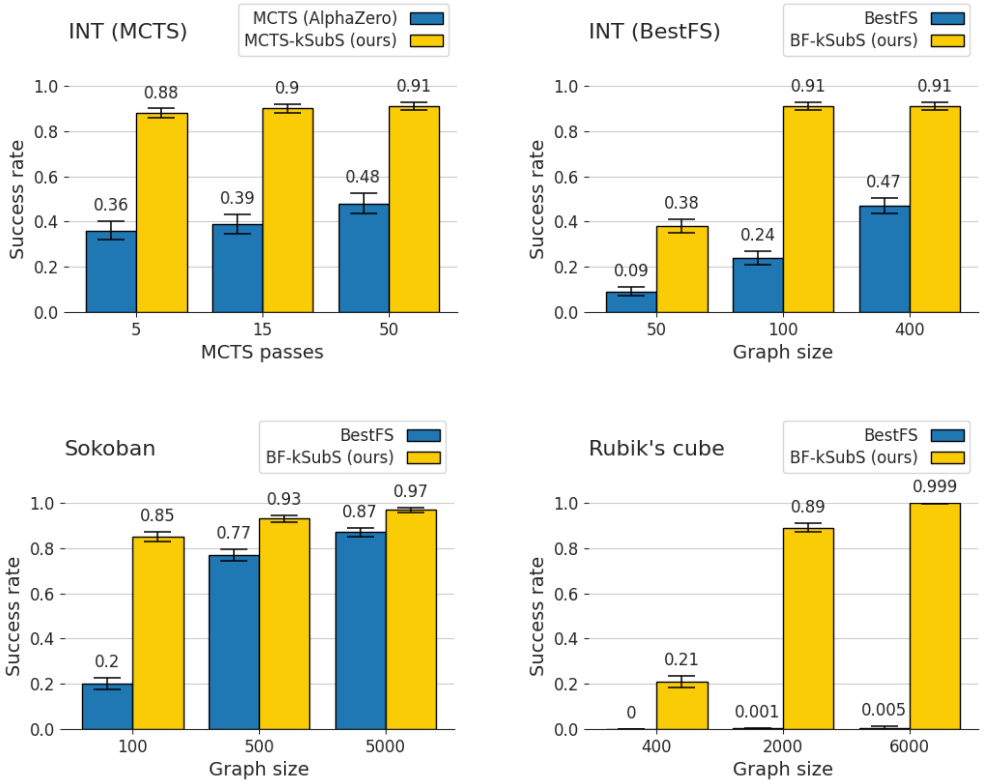

Figure 1: The performance of Subgoal Search. (top, left) compared on INT (with the proof length 15) to AlphaZero. (top, right) BF-kSubS consistently achieves high performance, even for small computational budges. (bottom, left) similarly on Sokoban (board size 12x12 with 4 boxes) the advantage of BF-kSubS is prominent for small budget. (bottom, right) BestFS fails to solve Rubik's Cube.

In Figure 1, we present the performance of Subgoal Search. We measure the *success rate* in function of the *search budget* used. The success rate metric is measured on 1000 instances of the given problem (which results in confidence intervals within ± 0.03 range). For BF-kSubS the search budget is referred to as *graph size* and includes the total number of nodes visited by the algorithm. More precisely, these are subgoals visited in Algorithm 1 and nodes on paths connecting them (given by Algorithm 2). For MCTS, we report *MCTS passes*, see details in Appendix A.1.

Below we discuss the results separately for each domain.

INT. The difficulty of the problems in INT increases fast with the proof length (denoted by L in [46]) and the number of accessible axioms (denoted by K in [46]). W used $K = 18$; all of available axioms. We observed, that BF-kSubS scales to proofs of length 10 and 15, which are significantly harder than 5 considered in [46], see Table 2. The same holds for MCTS-kSubS, see Appendix A.2. We check also that MCTS-kSubS vastly outperforms the AlphaZero algorithm, see Figure 1 (top, left).[3]

An MCTS agent was also evaluated on INT by [46]. The Implementation provided there, guided by the value function realized by Graph Neural Network, obtained 92% solve rate on the proofs of length 5. We ensured that our baseline is stronger - it solves over 99% of the problems on the same problem set.

	Proof length	5		10		15	
	Method	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)
Graph size	50	0.82	**0.99**	0.47	**0.97**	0.09	**0.38**
	100	0.89	**0.99**	0.64	**0.99**	0.24	**0.91**
	200	0.92	**0.99**	0.67	**0.99**	0.35	**0.91**
	400	0.93	**0.99**	0.72	**0.99**	0.47	**0.91**

Table 2: INT success rates for various problems difficulties and graphs sizes.

Sokoban. Using BF-kSubS allows for significantly higher success rates rates within the same computational budged, see Table 4.3. Our solution works well for 12×12 (with 4 boxes) boards commonly used in recent deep-learning research [13, 2] but also scales to much harder boards - 16×16 and 20×20. Importantly, we observe that already for small computational budget (graph size 100). BF-kSubS obtains higher success rates than the expert we used to create the dataset (these are 78%, 67%, 60% for the respective board sizes).

	Board Size	12 x 12		16 x 16		20 x 20	
	Method	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)	BestFS	BF-kSubS (ours)
Graph size	50	0.02	**0.69**	0.01	**0.47**	0.00	**0.31**
	100	0.20	**0.85**	0.10	**0.74**	0.04	**0.64**
	1000	0.81	**0.95**	0.80	**0.89**	0.73	**0.88**
	5000	0.87	**0.97**	0.89	**0.93**	0.87	**0.91**

Table 3: Sokoban success rates for various board sizes (each with 4 boxes).

Rubik's Cube BF-kSubS solves nearly 100% of cubes, BestFS is close to 0%, see Figure 1 (bottom, right). This is perhaps the most striking example of the advantage of using subgoal generator instead of low-level actions. We present possible explanation in Appendix J. We also include there the comparison with [1], a recent work on the Rubik's Cube, we note however their scope is to obtain nearly optimal solutions using high computational budges.

Out-of-distribution generalization is considered to be the crucial ability to make progress in hard combinatorial optimization problems [4] and automated theorem proving [46]. The INT benchmark has been specifically designed to benchmark this phenomenon. We checked how the Subgoal Search, method trained on proofs on length 10, generalizes favourably to longer problems, see Figure 3.

It might be sometimes hard to compare computational budgets between various algorithms and their versions. In Appendix E we measure that BF-kSubS and MCTS-kSubS offer very practical benefits in terms of wall-time, sometimes as much as 7 times.

We provide examples of solutions and generated subgoals in Appendix G. The preceding sections are devoted to a more detailed analysis of various aspects of our method.

[3] We use our implementation of AlphzaZero. In particular, it obtains stronger results than ones reported in [46] for $L = 5$. ($L = 10$ and larger are not considered in [46].

4.4 Analysis of k (subgoal distance) parameter

The subgoals are trained to be k steps ahead. In principle, having higher k should make planning easier. However, as k increases, the quality of the generator might drop; thus, the overall effect is uncertain. For Sokoban increasing k generally helps, but the benefits saturate around $k = 4$, see Figure 2. We note that higher k increases computational demand from low-level conditional policy search.

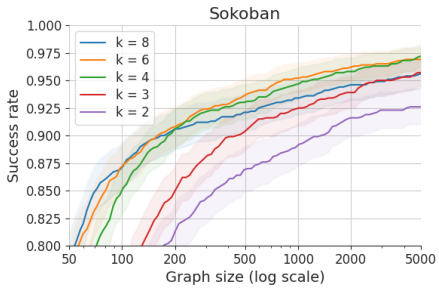

Figure 2: Comparison of success rates w.r.t k.

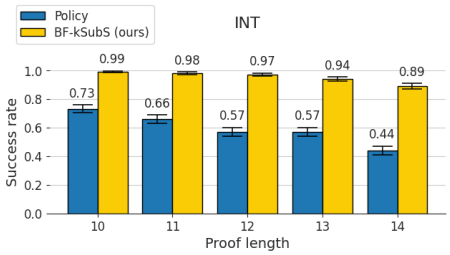

Figure 3: Out-of-distribution generalization to longer proofs. We compare with the Behavioral Cloning agent (Policy) studied in [46].

4.5 Quality of subgoals

The learned subgoal generator is likely to be imperfect (especially in hard problems). We study this on smaller 10×10 boards of Sokoban, for which the true distance $dist$ to the solution can be found with the Dijkstra algorithm. In Figure 4, we study $\Delta := dist_{s_1} - dist_{s_2}$, where s_1 is a sampled state and s_2 is a subgoal generated from s_1. Ideally, the histogram should concentrate on $k = 4$ used in training. We see, however, that in more than 60% subgoals lead to an improvement, see Figure 4.

We checked that in INT and Rubik's Cube about 50% of generated subgoals can be reached by the low-level conditional policy in Algorithm 2.

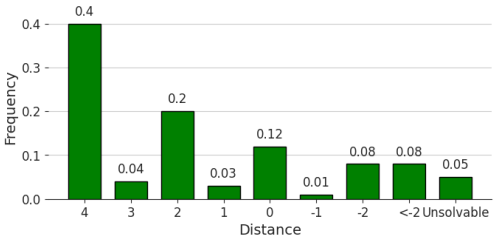

Figure 4: Histogram of Δ. Note that 12% of subgoals increases the distance, with 5% leading to irreversible "dead states" present in Sokoban.

4.6 Value errors

There are a few possible explanations for the success of our method. Below we discuss one of them. We hypothesize that using subgoals mitigates errors of value estimates, inherently present when using learned models. Both BestFS and MCTS choose where to expand search graph (or tree) based on the value function predictions. If on the path leading to the solution, value estimates are not monotonic, it will likely slow down the process of approaching the goal state. Intuitively, if we consider a path in higher temporal resolution i.e. one with consecutive states spaced k actions apart, the value estimates become more monotonic. We studied this on proofs generated by INT, the probability that value decreases when moving ℓ steps changes from 0.32 when $\ell = 1$ down to only 0.02 for $\ell = 4$.

Another problem is over-optimism, a search method would be misguided into states with erroneously positive values. To illustrate this consider $S(s)$, the set of all states within distance 3 from s and, additionally, having the same distance to the solution as s. Intuitively, $S(s)$ are similar states with respect to the difficulty of solving. Unlikely, the variance of value function prediction for $S(s)$ is equal to 0.84 (averaged over different s). For comparison, the average increase of value for moving one step closer to the solution is only 0.33. We found that with probability 86% there exists a suboptimal state in $S(s)$, i.e. having a higher value than the best immediate neighbour of s (which by properties of the game will be closer to solution in Sokoban). If instead of the neighbours we

consider states closer by 4 step (say given by a subgoal generator), then the probability of the error mentioned above drops to 38%. For more details see Appendix F.1.

5 Limitations and future work

In this section, we list some limitations of our work and suggest further research directions.

Reliance on expert data In this version, we use expert data to train learnable models. As kSubS improves the performance, we speculate that training akin to AlphaZero can be used, i.e. in a planner-learner loop without any outside knowledge.

Optimality and completness kSubS searches over a reduced state space, which might produce suboptimal solutions or even fail to find them. This is arguably unavoidable if we seek an efficient method for complex problems.

Subgoals definition We use simple k-step ahead subgoals, which is perhaps not always optimal. Our method can be coupled with other subgoal paradigms. Unsupervised detection of landmarks (see e.g. [47]) seems an attractive option.

More environment In future work, we plan to test kSubS on more environments to understand its strengths and weaknesses better. In this work, we generate subgoals in the state space, which might be limiting for tasks with high dimensional input (e.g., visual).

Reliance on a model of the environment We use a perfect model of the environment, which is a common practice for some environments, e.g., INT. Extending kSubS to use learned (imperfect) models is an important future research direction.

Determinism Our method requires the environment to be deterministic.

6 Conclusions

We propose Subgoal Search, a search algorithm based on subgoal generator. We present two practical implementations MCTS-kSubS and BF-kSubS meant to be effective in complex domains requiring reasoning. We confirm that indeed our implementations excel in Sokoban, Rubik's Cube, and inequality benchmark INT. Interestingly, a simple k step ahead mechanism of generating subgoals backed up by transformer-based architectures perform surprisingly well. This evidence, let us hypothesise, that our methods (and related) can be further scaled up to even harder reasoning tasks.

6.1 Potential Negative Societal Impacts

We investigate a general-purpose search algorithm. The progress in domains we use as testing grounds (Automated Theorem Proving, combinatorially complex puzzles) arguably has relatively low societal risk. One of the potential applications of ATP in real-word is to prove the correctness of software [32], which could be used both to fix or to compromise it.

Our method can be potentially used to improve algorithms in various domains, including SAT-solving, chemical retrosynthesis planning, Combinatorial Optimization, and automated code synthesis. Analysis of the potential impacts of development in these fields is out of the scope of this paper.

References

[1] 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.

[2] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

[3] Marcin Andrychowicz, Dwight Crow, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural*

Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA, pages 5048–5058, 2017.

[4] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

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

[6] Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In Stephen Jose Hanson, Jack D. Cowan, and C. Lee Giles, editors, *Advances in Neural Information Processing Systems 5, [NIPS Conference, Denver, Colorado, USA, November 30 - December 3, 1992]*, pages 271–278. Morgan Kaufmann, 1992.

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

[8] 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.

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

[10] Thomas Gabor, Jan Peter, Thomy Phan, Christian Meyer, and Claudia Linnhoff-Popien. Subgoal-based temporal abstraction in Monte-Carlo Tree Search. In *IJCAI*, pages 5562–5568, 2019.

[11] 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*, pages 185–194. PMLR, 2017.

[12] Timothy Gowers. *The importance of mathematics*. Springer-Verlag, 2000.

[13] 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*, pages 2464–2473. PMLR, 2019.

[14] Demis Hassabis, Dharshan Kumaran, Christopher Summerfield, and Matthew Botvinick. Neuroscience-inspired artificial intelligence. *Neuron*, 95(2):245–258, 2017.

[15] 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.

[16] 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.

[17] Leslie Pack Kaelbling. Learning to achieve goals. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 1094–1099. Morgan Kaufmann, 1993.

[18] 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, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 11566–11575, 2019.

[19] 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, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 8747–8758, 2018.

[20] 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*, pages 6259–6270. PMLR, 2020.

[21] 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, 2020.

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

[23] 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.

[24] 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, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14814–14825, 2019.

[25] 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.

[26] 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, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[27] 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, editors, *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*, pages 969–979. PMLR, 2020.

[28] Silviu Pitis, Harris Chan, Stephen Zhao, Bradly C. Stadie, and Jimmy Ba. Maximum entropy gain exploration for long horizon multi-goal reinforcement learning. 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*, pages 7750–7761. PMLR, 2020.

[29] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

[30] Stuart Russell and Peter Norvig. Artificial intelligence: A modern approach. ed. 3. 2010.

[31] 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.

[32] Johann M Schumann. *Automated theorem proving in software engineering*. Springer Science & Business Media, 2001.

[33] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

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

[35] 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.

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

[37] 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*, pages 213–222. PMLR, 2018.

[38] Richard S. Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M. Pilarski, Adam White, and Doina Precup. Horde: a scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In Liz Sonenberg, Peter Stone, Kagan Tumer, and Pinar Yolum, editors, *10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011), Taipei, Taiwan, May 2-6, 2011, Volume 1-3*, pages 761–768. IFAAMAS, 2011.

[39] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211, 1999.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[42] Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Manfred Otto Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. *ArXiv*, abs/1703.01161, 2017.

[43] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, Nov 2019.

[44] David Wilkins. Using patterns and plans in chess. *Artif. Intell.*, 14(2):165–203, 1980.

[45] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.

[46] 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.

[47] Lunjun Zhang, Ge Yang, and Bradly C. Stadie. World model as a graph: Learning latent landmarks for planning. *CoRR*, abs/2011.12491, 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] In the Section 5
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] In Section 6.1
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] We include the URL to the code repository in Section 1.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix sections B, D and C.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] In Appendix K
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] For one of domains we generate expert data with implementation of MCTS agent published with work [22], as noted in Section 4.2.
   (b) Did you mention the license of the assets? [N/A] The implementation noted above is made available on github, without any licence.
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We release our code, the URL is in Section 1.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]