A LOW-LEVEL PATH FUNCTION

The low-level path function (see LL_PATH, Algorithm 4) computes a path from the starting state to the goal state in the environment using low-level actions. However, it is responsible not only for returning the path but also for checking false positive errors of the verifier. Specifically, the verifier can accept an unreachable state in Algorithm 3 and then wrongly include it in the solution path. Thus, LL_PATH has to construct a low-level path and confirm that every step on the way is achievable.

Algorithm 4 Low-level path

```
function LL_PATH(s, parents)
    ▷ parents is the dictionary of parent nodes in the subgoal tree. (S,C) ∈ parents means that
C is a subgoal for state S
    path ← []
    while s in parents.KEYS() do
        subgoal_path ← GET_PATH(parents[s], s)                    ▷ see Algorithm 2
        if subgoal_path = [] then return False                    ▷ mistake of the verifier
        path ← concatenate(subgoal_path, path)
        s ← parents[s]
    return path
```

B TRAINING DETAILS

B.1 ARCHITECTURES

INT and Rubik's cube. All components of AdaSubS utilize the same architecture. Specifically, we used mBart, a transformer from the HuggingFace library (see Liu et al. (2020b)). To make training of the model and the inference faster we reduced the number of parameters: we used 45M learned parameters instead of 680M in the original implementation. We used 6 layers of encoder and 6 layers of decoder. The dimension of the model was set to 512 and the number of attention heads to 8. We adjusted the size of the inner layer of position-wise fully connected to 2048. During the inference, we used beam search with width 16 for INT and width 32 for Rubik's Cube. Our implementation of the model follows (Czechowski et al., 2021, Appendix B.1)

Sokoban. We used four convolutional neural networks: the subgoal generator, conditional low-level policy, value, and the verifier. They all share the same architecture with a different last layer, depending on the type of output. Each model had 7 convolutional layers with kernel size (3,3) and 64 channels. Conditional low-level policy and verifier need two Sokoban boards as an input, so for these networks we concatenate them (across the last, depth dimension) and we treat two boards as one tensor. For the value function on top of a stack of convolutional layers there is a fully connected layer with 150 outputs representing 150 distances to the goal or. CLLP has analogous final layers with the one exception that there are only two classes: determining if a subgoal is possible to reach by CLLP or not. Network used for generation of subgoals returns two outputs: distribution over possible modifications of a given state, and prediction whether a modified state is a good subgoal. First output is obtained with a fully connected layer, second with global average pooling followed by fully connected layer. Generation of a single subgoal is realised as a sequence of calls to this network. We start from a given state and iteratively apply modifications with high probability assigned by the first head of the network, until the second head predict that no more iterations are needed. (see also Appendix G.1)

B.2 TRAINING PIPELINE

To ensure fair comparison with Czechowski et al. (2021) we followed their settings of training pipeline.

INT and Rubik's Cube. To train the models we used the training pipeline from the HuggingFace library Liu et al. (2020b). We trained our models from scratch without using any pretrained checkpoints. The size of the training batch was 32, the dropout was set to 0.1, and there was no label smoothing. We used the Adam optimizer with the following parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. We applied warm-up learning schedule with 4000 warm-up steps and a peak learning rate of $3 \cdot 10^{-4}$. For inference in INT, we used temperature 1 and for Rubik's Cube to 0.5 (the optimal value was chosen experimentally).

Sokoban. For the training of all networks we used a supervised setting with the learning rate 10^{-4} and trained for 200 epochs. We used Adam optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$.

B.3 DATASETS

For dataset used to train all the network see Appendix D.

C COMPUTATIONAL BUDGET ANALYSIS

The default metric of the graph size that we use for comparisons counts all the states visited during the search, both high-level subgoals and intermediate states passed by the CLLP. It is a good estimate of the number of steps the algorithm takes to solve the given problem. For completeness, in this section, we analyze the total number of calls to every learned component of the pipeline for AdaSubS and the baseline kSubS.

Since all of the main components are deep neural networks, their evaluation time dominates the computational budget. Tables 3, 2 and 4 present the number of calls to each component per 1000 episodes. That indicates which component consumes the largest part of the computational budget. The results are presented for different numbers of beams (see Appendix G.1) used for sampling predictions from the subgoal generators – the only component that outputs a set of predictions. The default number of beams was 16 for Sokoban and INT, and 32 for the Rubik's Cube (see Appendix F for the complete list of the parameters).

As the tables show, AdaSubS not only solves more problems within smaller search graphs but also calls each component fewer times, which results in faster inference.

In the Rubik's Cube, the calls to the generators dominate the computations. However, when using smaller beams, this number can be significantly reduced while preserving the high success rate. In all the environments, AdaSubS is less sensitive to reducing the number of beams than kSubS in terms of performance. This is the case since in AdaSubS every single generator creates less subgoal candidates (see Tables 7-9), thus it does not require wide beam search. Therefore, by reducing the number of beams, AdaSubS can provide strong results within a much shorter time.

In the Rubik's Cube and Sokoban, using the verifier in AdaSubS significantly reduces the number of calls to the low-level policy. However, in INT it is not the case. In most cases when kSubS fails to find a solution, at some point it cannot create any valid subgoal, thus the search ends early. AdaSubS does not suffer from this issue, since it uses more generators. Thus, it counts the calls even from hard instances that require much larger graphs.

As shown in Table 5, if we count the calls only for the tasks solved by both methods, AdaSubS offers an advantage. Therefore, AdaSubS indeed provides better results within a smaller computational budget compared to kSubS.

Environment	Rubik's Cube					
Variant	kSubS (32 beams)	kSubS (4 beams)	AdaSubS (32 beams)	AdaSubS (8 beams)	AdaSubS (4 beams)	AdaSubS (2 beams)
Success rate	98.8	97.1	99.2	99.2	99	98.5
Generator calls	6 085 504	852 424	8 872 512	2 205 296	1 244 344	680 702
Verifier calls	0	0	277 266	275 662	311 086	340 351
Policy calls	1 330 328	1 526 116	352 883	350 877	395 804	446 320
Value calls	259 381	285 899	163 566	162 859	181 828	197 567
Total calls	7 675 213	2 664 439	9 666 227	2 994 694	2 133 062	1 664 940
Wall-time	24h	13h 38m	26h 39m	13h 43m	13h 9m	10h 9m

Table 2: Comparison of the number of calls to generator, verifier, policy, and value networks for different number of beams (width of beam search in subgoal generation)) for Rubik's Cube environment.

Environment	Sokoban					
Variant	kSubS (16 beams)	kSubS (8 beams)	kSubS (4 beams)	AdaSubS (16 beams)	AdaSubS (8 beams)	AdaSubS (4 beams)
Success rate	84.4	84.6	82.3	94	94	94.1
Generator calls	2 500 192	1 281 368	746 812	3 389 456	1 692 096	848 764
Verifier calls	0	0	0	211 841	211 512	212 191
Policy calls	4 576 807	4 693 216	5 554 468	248 120	247 785	247 993
Value calls	183 056	187 337	216 409	81 829	81 716	81 929
Total calls	7 260 055	6 161 921	6 301 280	3 931 246	2 233 109	1 390 877
Wall-time	13h 33m	10h 50m	9h 20m	15h 6 m	10h 56m	8h 31m

Table 3: Comparison of the number of calls to generator, verifier, policy, and value networks for different number of beams (width of beam search in subgoal generation) for Sokoban environment.

Environment	INT				
Variant	kSubS (16 beams)	kSubS (4 beams)	AdaSubS (16 beams)	AdaSubS (8 beams)	AdaSubS (4 beams)
Success rate	90.7	89.7	96	96	95.3
Generator calls	107 472	26 008	362 560	166 032	76 356
Verifier calls	0	0	67 980	62 262	57 267
Policy calls	378 125	366 805	801 345	738 000	659 545
Value calls	6 906	6 682	14 053	13 012	11 928
Total calls	492 503	399 495	1 245 938	979 306	805 096
Wall-time	4h 15m	3h 22m	12h 10m	8h 51m	86 38m

Table 4: Comparison of the number of calls to generator, verifier, policy, and value networks for different number of beams (width of beam search in subgoal generation) for INT environment.

Environment	INT	
Variant	kSubS	AdaSubS
Generator calls	93 472	102 112
Verifier calls	0	19 146
Policy calls	328 485	300 120
Value calls	6 206	5 597
Total calls	428 163	426 975

Table 5: Comparison of the number of calls to generator, verifier, policy, and value networks for problems solved by both methods for INT environment.

D DATASETS AND DATA PROCESSING

Sokoban. To collect offline data for Sokoban we used an MCTS-based RL agent from Miłoś et al. (2019). Namely, the dataset consisted of all successful trajectories obtained by the agent: 154000 trajectories for 12x12 boards with four boxes. We use 15% of states from each trajectory to create the training dataset $\mathcal{D}$. We performed the split of dataset $\mathcal{D}$ into two parts of equal size: $\mathcal{D}_1$ and $\mathcal{D}_2$. The former was used to train the subgoal generators and conditional low-level policy, while the latter was used to train the verifier network. This split mitigates the possibility of the verifier's over-optimism concerning the probability of achieving subgoals by CLLP.

INT. We represent both states and actions as strings. For states, we used an internal INT tool for such representation. For actions, we concatenate one token representing the axiom and the arguments for this axiom (tokenized mathematical expressions) following Czechowski et al. (2021).

To generate the dataset of successful trajectories we used the configurable generator of inequalities from the INT benchmark (see Wu et al. (2020)). We adjusted it to construct trajectories of length 15 with all available axioms. The dataset used for our experiments consisted of $2 \cdot 10^6$ trajectories.

Rubik's Cube. To construct a single successful trajectory we performed 20 random permutations on an initially solved Rubik's Cube and took the reverse of this sequence. Using this procedure we collected 10^7 trajectories.

D.1 DATASET FOR VERIFIER

The verifier answers the question of whether a given subgoal is reachable by the CLLP. Thus, the dataset for training this component cannot be simply extracted from the offline trajectories.

To get the training samples for the Rubik's Cube and INT, we run AdaSubS without the verifier. That is, we set $\mathtt{t_{hi}} = 1$ and $\mathtt{t_{lo}} = 0$, which essentially means that the reachability of all the subgoal candidates is checked solely by CLLP. During the searching, the generators create subgoal candidates, which are then verified by CLLP. Therefore, after working on some problem instances, we obtain a reach dataset of valid and not valid subgoals, marked by CLLP.

For the experiments in Sokoban, the limitation of the size of the offline dataset is an important factor for the final performance. Therefore, to ensure a fair comparison of AdaSubS with baselines, we do not generate additional solutions. Instead, we split the dataset as described above into $\mathcal{D}_1$ and $\mathcal{D}_2$ and used only $\mathcal{D}_2$ to generate data for the verifier. From every trajectory in $\mathcal{D}_2$, we sample some root states. For every such state, we use the subgoal generators to predict subgoal candidates. Then, CLLP checks the validity of each of them and we include them in the verifier training dataset.

After collection, it is essential to balance the dataset. Easy instances with short solutions provide fewer datapoints than hard tasks that require a deep search. Thus, it may happen that a substantial fraction of data collected this way comes from a single instance, reducing the diversity of the dataset. We observed such issues, particularly in the INT environment. To prevent this, during the collection of the data for INT, we limit the datapoints that can be collected from a single problem instance to at most 100. This way, we collected about $5 \cdot 10^6$ training samples for the verifier for each domain.

E BASELINES

As baselines, we use BestFS and BF-kSubS.

BestFS 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).

Similarly to AdaSubS, BestFS iteratively expands the graph of visited states by choosing nodes with the highest value and adding its children to the priority queue. However, instead of using children from the subgoal tree, it uses direct neighbors in the environment space. In other words, we use a single policy network to generate neighbor subgoals in the distance of 1 action from a given node and treat it as a new subgoal. One can implement BestFS by replacing the call to a subgoal generator ρ_k in Algorithm 1 with ρ_{BFS}.

ρ_{BFS} works in the following way. First, it uses a trained policy network to generate actions to investigate. Specifically, for INT we use beam search to generate high probability actions (it is necessary as for INT we represent actions as sequences, following Czechowski et al. (2021)). Then, it uses these actions to get a state that follows a given action (note that all our environments are deterministic). Finally, we treat returned states as our new subgoals, which are easily found in one step by the low-level policy.

BF-kSubS is the first general learned hierarchical planning algorithm shown to work on complex reasoning domains Czechowski et al. (2021), attaining strong results on Sokoban and Rubik's Cube and state-of-the-art results on INT. BF-kSubS is a special case of AdaSubS with the following hyperparameters choice: a single subgoal generator and inactive verifier (with $\mathtt{t_{lo}} = 0$ and $\mathtt{t_{hi}} = 1$) in Algorithm 3).

F HYPERPARAMETERS

Environment	Sokoban	Rubik's Cube	INT
learning rate	10^{-4}	$3 \cdot 10^{-4}$	$3 \cdot 10^{-4}$
learning rate warmup steps	-	4000	4000
batch size	32	32	32
kernel size	[3, 3]	-	-
weight decay	10^{-4}	-	-
dropout	-	0.1	0.1

Table 6: Hyperparameters used for training.

Environment	Sokoban		
Method	kSubS	MixSubS	AdaSubS (ours)
number of subgoals	4	1	1
number of beams	16	16	16
beam search temperature	1	1	1
k-generators	8	[8, 4, 2]	[8, 4, 2]
number of steps to check (C_2)	10	[10, 6, 4]	[10, 6, 4]
max steps in solution check	-	18	18
max nodes in search tree (C_1)	5000	5000	5000
acceptance threshold of verifier (t_{hi})	-	0.99	0.99
rejection threshold of verifier (t_{lo})	-	0.1	0.1

Table 7: Hyperparameters used for evaluation in the Sokoban environment.

Environment	the Rubik's Cube		
Method	kSubS	MixSubS	AdaSubS (ours)
number of subgoals	3	1	1
number of beams	32	32	32
beam search temperature	0.5	0.5	0.5
k-generators	4	[4, 3]	[4, 3, 2]
number of steps to check (C_2)	4	[4, 3]	[4, 3, 2]
max steps in solution check	-	4	4
max nodes in search tree (C_1)	5000	5000	5000
acceptance threshold of verifier (t_{hi})	-	0.995	0.995
rejection threshold of verifier (t_{lo})	-	0.0005	0.0005

Table 8: Hyperparameters used for evaluation in the Rubik's Cube environment.

Most of the hyperparameters, both for training and evaluation, follow from Czechowski et al. (2021).

The most important parameter of AdaSubS is the set of k-generators to use and the number of subgoals each of them generate. Based on experimental results, we have chosen generators of 8, 4, and 2 steps for Sokoban, 4, 3, and 2 steps for the Rubik's Cube, and 4, 3, 2, and 1 step for INT. In the first two domains, the longest generator match the one used for kSubS. In INT, AdaSubS allows using even longer subgoals than kSubS.

For kSubS, we used k equal to 8, 4, and 4 for Sokoban, Rubik's Cube, and INT, respectively. The last two were chosen from Czechowski et al. (2021), as they performed the best. For Sokoban, our experiments showed that 8-generator performs better than the 4-generator proposed in Czechowski et al. (2021).

Environment	INT		
Method	kSubS	MixSubS	AdaSubS (ours)
number of subgoals	4	2	3
number of beams	16	16	16
beam search temperature	1	1	1
k-generators	3	[3, 2, 1]	[3, 2, 1]
number of steps to check (C_2)	3	[3, 2, 1]	[3, 2, 1]
max steps in solution check	-	5	5
max nodes in search tree (C_1)	400	400	400
acceptance threshold of verifier (t_{hi})	-	1	1
rejection threshold of verifier (t_{lo})	-	0.1	0.1

Table 9: Hyperparameters used for evaluation in the INT environment.

The hyperparameters for training the verifier network remain the same as for other components. The thresholds t_{hi} and t_{lo} were chosen with a grid-search for every domain. Though some of the chosen values may seem tight, they efficiently reduce the amount of required computations (see Appendix C). For instance, the rejection threshold for the Rubik's Cube of $\texttt{t}_{\texttt{lo}} = 0.0005$ seems to be very low, but it is enough to reject more than 85% of non-valid subgoals.

The parameter C_1 (see Algorithm 1) controls the number of high-level nodes in the search tree. It is lower than the actual graph size that we use for comparisons since it counts neither the intermediate states visited by CLLP nor the subgoals that turned out to be invalid. That hyperparameter was chosen so that it allows all the evaluated algorithms to reach the graph size values used for comparison in Figure 2 and others in Section 4.

G COMPONENTS OF THE ADASUBS

G.1 SUBGOAL GENERATORS

The main purpose of the subgoal generator is to propose subgoal candidates in every iteration of the planner loop. That is, given the current state s of the environment it should output other states that are a few steps closer to the goal g.

We train a k-generator by extracting training data from successful trajectories. Let $s_0, s_1, \ldots, s_n$ be a trajectory that leads to the goal s_n. For every state s_i we train the k-generator network to output the state s_{i+k}, exactly k steps ahead. Provided with a dataset of trajectories, this is a supervised objective. Clearly, a state that is k steps ahead does not need to be exactly k steps closer to the solution, especially if the trajectories include noise or exploration. However, it is guaranteed to be at most k steps closer, which allows setting reliable limits for the reachability checking.

In a simple approach, k is a hyperparameter that needs to be fixed, as proposed by Czechowski et al. (2021). However, this is a strong limitation if the environment exhibits the variable complexity problem. Therefore, AdaSubS instead uses a set of generators, trained for different values of k. This way, the planner can adjust the expansion to match the local complexity of the problem. Additionally, training a set of generators can be easily parallelized.

For our generators, we use the transformer architecture. The input state is encoded as a sequence of tokens, as described in (Czechowski et al., 2021, Appendix C). The network produces another sequence of tokens on the output, which is then decoded to a subgoal state. The output sequence is optimized for the highest joint probability with beam search – the consecutive tokens are sampled iteratively and a fixed number of locally best sequences passes to the next iteration. This way, the generator allows sampling of a diverse set of subgoals by adjusting the beam width and sampling temperature. The exact number of the subgoals that the generators output are given in Appendix F.

As noted in Section 3.1, for the Sokoban environment instead of transformers we use simple convolutional networks. In this domain, the subgoal is created by a sequence of changes to the input state. The generator network is trained to predict the probability of changing for every pixel. Then, the subgoals are obtained as a sequence of modifications that maximize the joint probability. For simplicity, in AdaSubS we use beam search for all the domains, including Sokoban.

G.2 CONDITIONAL LOW-LEVEL POLICY (CLLP)

When we want to add a subgoal candidate to our search tree, we need to check whether it is reachable from the current state. This can be done using CLLP – a mapping that given a state and a subgoal produces a sequence of actions that connects those configurations, or claim there is no. Specifically, the policy network, given the state and subgoal, iteratively selects the best action and executes it until the subgoal is reached or a threshold number of steps is exceeded, as shown in Algorithm 2.

CLLP is trained to imitate the policy that collected the training data. For every pair of states s_i, s_j that are located at most d steps from each other, it is trained to predict the action a_i, taken in the state s_i. Such action may not be optimal but usually it leads closer to s_j. The threshold d controls the range of the policy, as it is trained to connect states that are at most d steps away. Thus, it is essential to set the hyperparameter d to a value that is greater than the distances of all the generators used.

G.3 VERIFIER

To check whether a k-subgoal is reachable with the conditional policy, we need to call it up to k times. If we decide to use generators with long horizons, it becomes a significant computational cost. To mitigate this issue, we use the verifier that estimates the validity of a subgoal candidate in a single call. During the search, the generated subgoal candidates are evaluated by the verifier. For each of them, it estimates whether they are valid and outputs its confidence. If the returned confidence exceeds a fixed threshold, we do not run the costly check with the conditional policy. We perform such a check only in case the verifier is uncertain (see Algorithm 3).

At the end of the search, when a solving trajectory is found, we need to find the paths between all the pairs of consecutive subgoals that were omitted due to the verifier (see Algorithm 4). Since the length of the final trajectory is usually much smaller than the search tree, that final check requires much less computations.

It should be noted that the verifier estimates validity with respect to the conditional policy that is used. In case a valid subgoal is generated but the policy cannot reach it for some reason, it cannot be used to build the search tree anyway, for no solution that uses it can be generated in the final phase. Thus, the verifier should be trained to predict whether the CLLP that is used can reach the subgoal, rather than whether it is reachable by an optimal policy.

To train the verifier, we run our pipeline on some problem instances. All the subgoals created by the generators are validated with CLLP. This way, eventually we obtain a dataset of reachable and unreachable subgoal candidates. We train the verifier network to fit that data. Unlike for the other components, training the verifier does not require access to any trajectories, only to a number of problem instances.

G.4 VALUE FUNCTION.

The value function $V : \mathcal{S} \to \mathbb{R}$ estimates the negative distance between the current state s and the goal state g. During the search, this information is used to select the most promising nodes to expand. For every trajectory $s_0, \ldots, s_n$ in the dataset it is trained to output $i - n$ given s_i. We opted for a simple training objective but any value function can be used in the algorithm.

H DEVELOPING ADAPTIVE SEARCH METHODS

There are many natural ways to incorporate adaptivity to the subgoal search pipeline. We experimented with several designs to find one that gives strong results in any domain. Here we provide detailed description of all the tested variants and the numerical results of their evaluation in our environments. Their implementations can be found in Section H.2.

An adaptive algorithm should adjust the complexity of the proposed subgoals to the local complexity of the environment in the neighbourhood of the processed state. This can be realized with the following two approaches:

- Use adaptive planner that provided a list of k-generators, in every step selects the most promising node and a generator to expand it.
- Use adaptive subgoal generator that instead of proposing fixed-distance subgoals learns to automatically adjust the distance.

H.1 ADAPTIVE PLANNERS

When implementing the adaptivity with the planner, we need to specify a list of k-generators $\rho_{k_0}, \ldots, \rho_{k_m}$. In every iteration, the algorithm will select a node to expand and generators from the list that will create the new subgoals. This way, it can directly control the complexity of the subgoals and adapt to the current state and progress of the search.

MixSubS. Given a list of trained k-generators, a simple approach is to call all of them each time a node is expanded. In every iteration, we choose the node with the highest value in the tree and add subgoals proposed by each generator ρ_{k_0} to ρ_{k_m}. See Algorithm 5 for the implementation.

Observe that in the easy areas of the environment the search will progress fast, since the furthest subgoal will most likely have the highest value, so it will be chosen as the next node to expand. On the other hand, in the hard parts the shortest generators are more likely to provide subgoals that advance towards the target at least a step.

This method already achieve superior results compared to single generators, both on small and large budget. In the Rubik environment, it even reaches 100% solved cubes. MixSubS offer the advantage of planning with different horizons, but at the same time, it produces many unnecessary nodes in the easy areas, where taking only long steps suffices to solve the task. Additionally, one may want to prioritize the generators that perform better, which cannot be done with this method.

Iterative mixing. In this approach, we specify a number of iterations l_i for each generator ρ_{k_i}. We use ρ_{k_0} to expand the highest-valued nodes in the first l_0 iterations. Then, we use ρ_{k_1} to expand the best nodes in the following l_1 iterations and the procedure follows for the consecutive generators. After finishing with the last one, we start again from the beginning. See Algorithm 6 for the implementation.

This algorithm offers the flexibility of specifying the exact number of iterations for each generator, which forms an explicit prioritization. It can resemble some of the listed algorithms for carefully chosen l_i. However, tuning the number of iterations requires much more effort than the other parameter-free algorithms do. Therefore, we experimented with another two mixing approaches that in every iteration select the generator automatically.

Strongest-first. Another natural implementation of the planner is to choose the node with the highest value and expand it with the longest generator that was not used there yet. See Algorithm 7 for the implementation. While this greedy approach maintain clear advantage over single generators, it is outperformed by most of the mixing methods, even the simple mixes. We hypothesize that this method is more sensitive to the errors of the value function – if the search enters an area that the value function estimates too optimistically, it spends too much time trying to exploit it.

Longest-first (used by AdaSubS). This method in every iteration selects the longest generator that has at least one node to expand and highest-valued node for that generator in the queue. This way, it explicitly prioritizes using the longest generators and turns to the shorter only when the search is stuck. See Algorithm 8 for the implementation. As shown in the tables below, this method outperforms all other designs, in all the environments and within all budget constraints. It prioritizes

the better generators, but does not require specifying any additional hyperparameters. Therefore, we consider it the best mixing algorithm and use in AdaSubS as the default planner.

H.2 Adaptive planners implementations

In this section we provide the implementations of the planners. The lines highlighted in blue indicate the differences with the AdaSubS code. All the methods require specifying the list of generators $\rho_{k_0}, \dots, \rho_{k_m}$. The Iterative mixing planner additionally requires a list of iterations $l_0, \dots, l_m$.

Algorithm 5 MixSubS

```
function SOLVE(s_0)
    T ← ∅                                              ▷ priority queue
    parents ← {}
    T.PUSH((V(s_0), s_0))
    seen.ADD(s_0)
    while 0 < LEN(T) and LEN(seen) < C_1 do
        _, s ← T.EXTRACT_MAX()
        subgoals ← {ρ_{k_1}(s), ..., ρ_{k_m}(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
            T.PUSH((V(s'), s'))
            if SOLVED(s') then
                return LL_PATH(s', parents)
    return False
```

Algorithm 6 Iterative mixing

```
function SOLVE(s_0)
    T_{k_i} ← ∅                                        ▷ m + 1 priority queues
    parents ← {}
    for k in k_0, ..., k_m do
        T_k.PUSH((V(s_0), s_0))
    seen.ADD(s_0)
    cnt ← 0                                            ▷ Iterations counter
    id ← 0                                             ▷ Current generator id
    while 0 < LEN(T) and LEN(seen) < C_1 do
        if cnt = l_id or LEN(T_{k_id}) = 0 then
            id ← (id + 1)%(m + 1), cnt ← 0
        cnt ← cnt + 1
        _, s ← T_{k_id}.EXTRACT_MAX()
        subgoals ← ρ_{k_id}(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_k.PUSH((V(s'), s'))
            if SOLVED(s') then
                return LL_PATH(s', parents)
    return False
```

Algorithm 7 Strongest-first

```
function SOLVE(s_0)
    T ← ∅    ▷ priority queue with lexicographic order
    parents ← {}
    for k in k_0, ..., k_m do
        T.PUSH(((V(s_0), k), s_0))
    seen.ADD(s_0)
    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(((V(s'), k), s'))
            if SOLVED(s') then
                return LL_PATH(s', parents)
    return False
```

Algorithm 8 Longest-first

```
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)
    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)
    return False
```

H.3 ADAPTIVE GENERATORS

A k-generator is trained to propose subgoals that should be exactly k steps ahead. However, instead of matching a fixed distance, it can opt for long subgoals when the next steps are clear and short when difficulties appear, or both if it is not certain.

Implementing this idea requires changing the training of the generator. Given a training trajectory, for each state s_i we need to select the target state $s_{t(i)}$ that should be the output of the generator. We tested a few methods that select this target.

Longest-reachable We use the low-level conditional policy to estimate the local complexity around s_i. Specifically, we choose $s_{t(i)}$ to be the furthest state on the trajectory such that it is reachable from s_i with the CLLP and so do all its predecessors. In other words, we check whether CLLP starting in s_i can reach s_{i+1}, s_{i+2}, etc. When we find the first state s_j that is not reachable, we set $t(i)$ to be $j-1$.

Intuitively, this approach makes the generator learn to output subgoals as distant as possible, but still reachable for CLLP. However, this way the targets are selected on the borderline of reachability, which may lead to too hard subgoals in some cases.

Sampling-reachable To make the target state selection more robust, we modify the reachability verification. Instead of greedily following the best action determined by CLLP probabilities, in every step we sample the action. This way, we are more likely to take suboptimal actions, so the selected target should be reachable with higher confidence.

Secondary-reachable Another method of making more robust selection is to follow the action with the lowest probability that exceeds a fixed threshold, e.g. 25%. Intuitively, we follow the action that CLLP consider as good, but is less certain than in case of the highest-ranked. Therefore, a subgoal reached in this way should be reachable with even higher confidence when following the greedy actions.

Our experiments show that the adaptive generators trained according to those designs perform well in the environments we consider. For instance, all the methods reach nearly 90% solve rate on Sokoban. However, none of them provide better results than the kSubS baseline. Therefore in this work we focus on planner-based adaptivity and leave tuning the adaptive generators pipeline for future work.

H.4 BENCHMARKING RESULTS

Tables 10-12 show the numerical results achived by the adaptive planners described in section H.1, compared to baselines: BestFS and kSubS. For some of the methods a few variants are provided. In each table, the longest-first, strongest-first and iterative mixing methods use the same set of generators: $[3, 2, 1]$ for INT, $[4, 3, 2]$ for Rubik, and $[8, 4, 2]$ for Sokoban. Our main algorithm, Adaptive Subgoal Search, uses the longest-first planner and the verifier network.

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%

Table 10: INT benchmark

Rubik					
		Small budget (400 nodes)		Large budget (6000 nodes)	
		with verifier	without	with verifier	without
BestFS		-	0.0%	-	1.8%
kSubS	k=4	28.8%	24.5%	98.6%	98.8%
	k=3	19.3%	18.6%	95.6%	95.4%
	k=2	8.2%	4.5%	99.0%	95.8%
	k=1	0.5%	0.5%	76.5%	76.5%
MixSubS	k=[4,3,2]	29.1%	20.9%	99.1%	100.0%
	k=[4,3]	49.1%	45.1%	99.2%	100.0%
Iterative mixing	iterations=[1,1,1]	33.5%	23.0%	99.2%	100.0%
	iterations=[10,1,1]	50.6%	43.6%	99.1%	99.9%
	iterations=[4,2,1]	48.4%	41.2%	99.2%	100.0%
Strongest-first		33.4%	27.1%	99.0%	99.9%
Longest-first		58.0%	52.4%	99.2%	100.0%

Table 11: Rubik benchmark

		Sokoban			
		Small budget (100 nodes)		Large budget (5000 nodes)	
		with verifier	without	with verifier	without
BestFS		-	45.9%	-	82.6%
kSubS	k=16	13.7%	5.1%	60.5%	63.5%
	k=8	26.0%	4.7%	85.6%	84.4%
	k=4	8.2%	2.6%	68.1%	65.5%
	k=2	1.4%	0.7%	40.0%	38.3%
MixSubS	k=[8,4,2]	52.7%	37.7%	91.7%	90.2%
	k=[16,8,4]	55.6%	44.9%	89.1%	89.0%
Iterative mixing	iterations=[1,1,1]	52.7%	37.7%	91.7%	90.2%
	iterations=[10,1,1]	68.3%	58.6%	92.5%	92.1%
	iterations=[4,2,1]	64.5%	52.6%	93.5%	93.2%
Strongest-first		54.6%	41.9%	92.0%	90.8%
Longest-first		72.2%	63.4%	93.4%	93.6%

Table 12: Sokoban benchmark

I INFRASTRUCTURE USED

We performed experiments using two types of hardware: with and without access to GPUs. In the former, we used nodes equipped with a single Nvidia V100 32GB card or Nvidia RTX 2080Ti 11GB. Each such node had 4 CPU cores and 168GB of RAM. In the latter, we used nodes equipped with Intel Xeon E5-2697 2.60GHz CPU with 28 cores and 128GB RAM.

Each transformer model was trained on a single GPU node for 3 days. Sokoban models were trained on CPU nodes (due to the small size of the models).