Beyond A^*: Better Planning with Transformers via Search Dynamics Bootstrapping

Lucas Lehnert[1], **Sainbayar Sukhbaatar**[1], **Paul Mcvay**[1], **Michael Rabbat**[1], **Yuandong Tian**[1]

[1]FAIR at Meta

While Transformers have enabled tremendous progress in various application settings, such architectures still lag behind traditional symbolic planners for solving complex decision making tasks. In this work, we demonstrate how to train Transformers to solve complex planning tasks and present **Searchformer**, a Transformer model that optimally solves previously unseen Sokoban puzzles 93.7% of the time, while using up to 26.8% fewer search steps than standard A^* search. Searchformer is an encoder-decoder Transformer model trained to predict the *search dynamics* of A^*. This model is then fine-tuned via expert iterations to perform fewer search steps than A^* search while still generating an optimal plan. In our training method, A^*'s search dynamics are expressed as a token sequence outlining when task states are added and removed into the search tree during symbolic planning. In our ablation studies on maze navigation, we find that Searchformer significantly outperforms baselines that predict the optimal plan directly with a 5–10× smaller model size and a 10× smaller training dataset. We also demonstrate how Searchformer scales to larger and more complex decision making tasks like Sokoban with improved percentage of solved tasks and shortened search dynamics.

Correspondence: {lucaslehnert, yuandong}@meta.com

1 Introduction

Over the past few years, Transformer-based architectures (Vaswani et al., 2017) have demonstrated impressive performance in different tasks, including holding conversations at the human level (Shuster et al., 2022; OpenAI, 2022, 2023; Touvron et al., 2023), high-quality image understanding (Caron et al., 2021; Oquab et al., 2024; Assran et al., 2023) and video generation (Singer et al., 2023), multi-modal generation (Girdhar et al., 2023; Radford et al., 2021), and code completion (Roziere et al., 2023; OpenAI, 2021). By training these architectures with a huge amount of data, the resulting models, such as Large Language Models (LLMs), can generalize well in real-world use cases.

Despite these successes, Transformer-based architectures and LLMs still struggle when it comes to solving planning and reasoning tasks. Previous studies (Momennejad et al., 2023; Valmeekam et al., 2023a,b) demonstrate that LLMs fall short in multi-step planning tasks (Valmeekam et al., 2023b) or when performing higher-order reasoning (Momennejad et al., 2023; Fan et al., 2020).

In recent years, various methods have been proposed to improve the performance of Transformers in reasoning and planning tasks. One common and effective approach is to simulate the human thinking process and produce intermediate "thoughts" before outputting a response. Chain-of-Thought (CoT) prompting (Wei et al., 2022) encourages the model to predict the intermediate steps and to "think" step by step. Tree-of-thoughts (ToT) uses a branching strategy and critics to generate different thought paths before picking the best one (Yao et al., 2023). While these techniques are often effective, there are studies showing that in many cases, they may lead to worse performance, for example due to self-enforcing (Huang et al., 2023). Furthermore, techniques effective on one dataset may not work well on others due to changes in the type of reasoning involved (e.g., spatial reasoning vs. mathematical reasoning vs. common-sense reasoning). How to enable Transformers or LLMs to plan, solve multi-step decision making tasks, and perform different types of reasoning still remains elusive and an active area of research.

These methods stand in sharp contrast with traditional symbolic planning and search techniques. While such techniques may not exhibit the language understanding capabilities of LLMs trained on internet-scale datasets,

they are well-defined and can solve planning tasks that are far more complex than the tasks state-of-the-art LLMs can solve (Momennejad et al., 2023; Valmeekam et al., 2023a). Additionally, there often exist formal guarantees for solutions computed by these traditional methods, because symbolic planning algorithms often follow a well-defined rule-based search procedure.

Our work

In this paper, we demonstrate how to train Transformers to solve complex planning tasks and present *Searchformer*, a Transformer model that computes an optimal plan in fewer search steps than symbolic planning algorithms such as A^* search (Russell and Norvig, 2021, Chapter 3) in multi-step planning tasks such as maze navigation and solving Sokoban puzzles. To achieve this, we introduce *search dynamics bootstrapping*, a method that first trains a Transformer model to imitate the search procedure of A^* and then fine-tunes the model to find an optimal plan with fewer search steps. *so this is like a ground-up transformer, no LLM pretrained, no strings attached !*

In the first step a Transformer model is trained to imitate A^* search. Here, we generate synthetic datasets by running A^* search in randomly generated planning task instances. While A^* is executed, we log the performed computation and optimal plan into a sequence of words, called *tokens*. The resulting training dataset then contains an execution trace of A^* and encodes information about the *search dynamics* of A^* itself. A Transformer model is then trained to generate these token sequences along with an optimal plan for any given planning task.

In the second step, we show that our *Searchformer* trained with search-augmented sequences—sequences that include A^*'s execution trace—can be further improved via expert iteration (Gulcehre et al., 2023), a method that encourages the Transformer to generate fewer search steps while still generating optimal solutions. This procedure leads to a neural planning algorithm that is implicitly encoded in the Transformer's network weights, and finds an optimal plan with high probability in fewer search steps than standard A^* search. Specifically, when solving Sokoban puzzles, our models solve 93.7% of all test tasks while performing search steps that are on average 26.8% shorter than A^* search. This paves the way to use Transformers to move beyond traditional symbolic planning algorithms.

To better understand how the used training data and number of model parameters influence performance of the resulting models, we present a set of ablation studies. We train models on both *solution-only sequences* (token sequences including task description and final plan) and *search-augmented sequences* (token sequences including task description, search tree dynamics, and final plan) and compare their performance. Despite the fact that search-augmented sequences are thousands of tokens long, search-augmented models (that are trained on search-augmented sequences) can still learn to generate the correct search dynamics and optimal plans with high probability. Further, compared to solution-only models (that are trained on solution-only sequences), search-augmented models generate an optimal plan more often on unseen tasks with ten times fewer training sequences, highlighting the power of including A^*'s search dynamics into the training process of Transformer models. Since the search dynamics can be included into the training data simply by through logging when generating synthetic training sequences, our study provides a data-efficient way to improve the reasoning power of Transformer models. We also find that solution-only models do not improve with more parameters, showing that the type of training data used plays a critical role in this setting.

2 Related Work

While existing work (Trinh et al., 2024; Ruoss et al., 2024) leverages synthetic datasets to learn strong policies for reasoning, our study is fundamentally different in this regard. We focus on improving the reasoning capability embedded in a Transformer's weights, while algorithms such as AlphaZero (Silver et al., 2018), MuZero (Schrittwieser et al., 2020) and AlphaGeometry (Trinh et al., 2024) treat general neural network models (including Transformers) as black-boxes, and use existing symbolic planning techniques to improve the neural network itself. For example, Silver et al. (2017) use MCTS as a policy improvement operator to update the neural network's weights. In contrast, the search dynamics bootstrapping method in this work uses a Transformer model to generalize towards more efficient search patterns and improve the model itself. A planning algorithm is only used to initially train a Transformer model.

(a) Maze navigation task

(b) Tokenization of a planning task and its solution
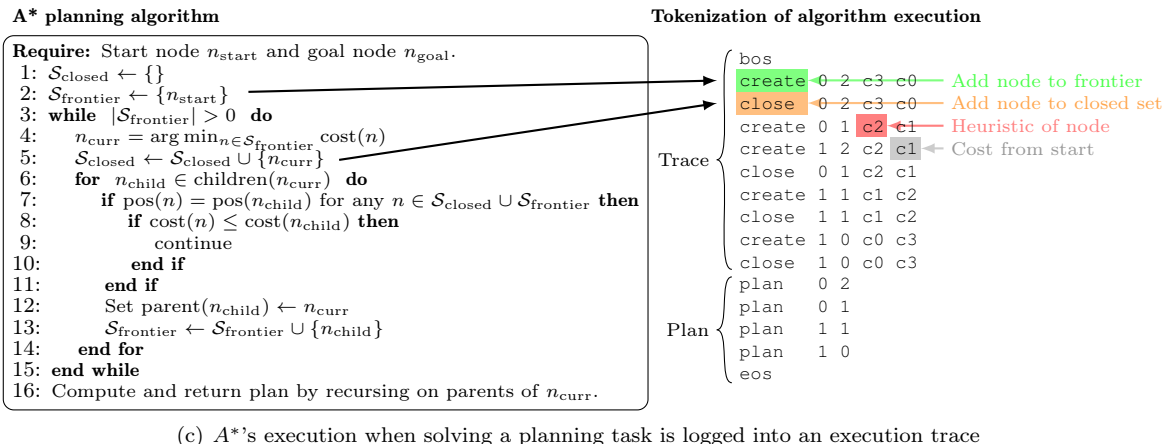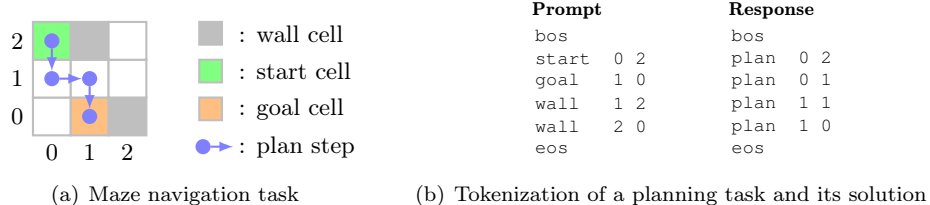
(c) A^*'s execution when solving a planning task is logged into an execution trace

Figure 1 Expressing a planning task in token sequences. (a): A 3×3 maze navigation task where an agent must navigate from the start cell to the goal cell without entering a wall cell. For each transition, a cost of one is given and the optimal plan is the shortest path from start to goal. (b): The 3×3 maze navigation task is expressed as a prompt token sequence (left panel) and the optimal plan is expressed as a response token sequence (right panel). The start and end of a response sequence is indicated by a beginning-of-sequence token, **bos**, and an end-of-sequence tokens, **eos**. The numbers indicate x, y coordinates. (c): The search dynamics of the A^* algorithm (left panel) is logged into an execution trace represented by a token sequence (right panel). For maze tasks, coordinates are encoded with integer tokens. The next two numbers costs associated with that node (letter "c" is added to differentiate them from coordinate numbers). Each token is mapped to a unique non-negative integer and this integer is processed in the neural network.

ScratchPad (Nye et al., 2021) train neural networks on execution traces of reasoning tasks, Yang et al. (2022), Pallagani et al. (2022), and Ruoss et al. (2024) train models to predict the optimal solution or an optimal action directly. In contrast, we find more efficient LLM-based planning algorithms by predicting the search dynamics of a planning algorithm, which is the execution trace when the solver attempts to find the optimal solution, which is a harder problem than predicting an optimal decision making strategy.

Our work also bears some similarity with neuro-symbolic systems (e.g., Neural Turing Machines (Graves et al., 2014), Neural Programming Architecture (Cai et al., 2017)), which build differentiable architectures to mimic the functionality of existing symbolic systems. However, these methods use dedicated components (e.g., explicit memory components, built-in recursion), while Searchformer uses a general Transformer architecture. By using long contexts and positional embeddings (Chen et al., 2023; Peng et al., 2023) to predict an optimal plan, Searchformer performs well in solving different complex planning tasks. Ultimately, our work is a step towards building more general neural architectures that automatically learn a reasoning and planning mechanism without prior human knowledge.

Using Transformer architectures to solve complex sequential decision making has been studied in prior work in a reinforcement learning (RL) setting (Chen et al., 2021; Janner et al., 2021; Laskin et al., 2023). Decision Transformer (Chen et al., 2021) predicts the next optimal action given a trajectory segment that is annotated with total return values, while Trajectory Transformer (Janner et al., 2021) focuses on learning a world model that predicts future states, actions, and rewards. Similarly, in Algorithm Distillation (Laskin et al., 2023) a Transformer architecture is trained to predict the next action given a trajectory segment of trial and error interactions as context. In contrast, we incorporate how a planning algorithm arrives at a final result into a

Transformer models. MCTSNet (Guez et al., 2018) also attempts to learn the search procedure itself, but still hard-codes the MCTS search procedure into neural network, which leads to quadratic backpropagation overhead and can only deal witih up to 500 rollouts, while our approach can deal with much longer search execution trace. We demonstrate that Transformers can not only imitate a symbolic planning algorithm but also improve past the algorithm it was initially trained on via fine-tuning.

3   Problem Setup

We consider two domains: maze navigation tasks (Figures 1(a)) and solving Sokoban puzzles (Figure 2). In maze navigation, the goal is to find the shortest path in an n-by-n maze between a start location and a goal location without passing through wall cells. Figure 2 depicts a Sokoban puzzle where the worker can move up, down, left, or right to push each box onto a dock (crossed grid cells). The worker cannot pull a box away from a wall cell. To solve the puzzle, each box must be moved onto a dock in as few steps as possible. An incorrect move may immediately lead to a dead end and thus careful planning is needed to solve the task. Furthermore, each state in a Sokoban puzzle consists of every possible combination of a box and worker position. Consequently, state spaces are large making a Sokoban puzzle computationally more difficult to solve for algorithms such as A^* search.
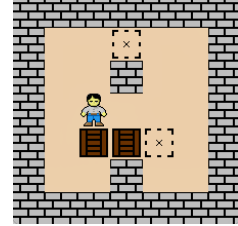

Figure 2   Example Sokoban puzzle[1]**.** The prompt, A^* execution trace, and optimal plan for this task is illustrated in Figure 10 in Appendix A.3. For Sokoban, we only use a non-deterministic A^* implementation.

3.1   Generating execution traces of A^* search.

The A^* algorithm computes an optimal plan by manipulating two sets of nodes:

- A frontier set containing the current search frontiers;
- A closed set containing all searched nodes.

In the maze example in Figure 1(a), each node corresponds to an empty (non-wall) grid cell. For each node, the algorithm computes a heuristic value and a cost from start value. At any given iteration, which node is searched next is determined by the content of the frontier and closed sets as well as the heuristic and cost from start values (Figure 1(c), left panel). The A^*'s execution trace is collected by tracking all insertion operations into the frontier and closed set along with heuristic and cost from start values. The right panel in Figure 1(c) illustrates the resulting trace for the maze example shown in Figure 1(b). Each row corresponds either to an insertion of a node into the frontier, indicated by a `create` token, or to moving a node into the closed set, indicated by a `close` token. Each node is represented by its (x, y) position in the maze as well as the two cost tokens. The resulting plan is then appended to this trace. This trace is constructed such that given any prefix of this trace the next token can be predicted correctly. For the maze datasets, A^* uses the Manhattan distance to the goal location as a heuristic. In Sokoban, A^* first matches every box to the closest dock and then computes the sum of all Manhattan distances between each box and dock pair.

For each experiment, we generate two token sequence variants, as illustrated in Figure 1:

- *Search-augmented sequence*. This variant contains sequences of the format `<prompt><trace><plan>`, where the `<trace>` part encodes A^* execution trace and `<plan>` encodes the optimal plan.

- *Solution-only sequence*. This variant omits A^*'s exeuction trace and only contains sequences of the format `<prompt><plan>`.

For example, the 3×3 maze navigation task (Figure 1(a)) can be mapped into a prompt and plan sequence (Figure 1(b)). The `<trace>` part is constructed by running A^* on each planning task and logging the performed computations into a token sequence (Figure 1(c)). While the optimal plan in both cases can be fairly short,

---

[1]This level image was composed using image icons from https://github.com/morenod/sokoban (accessed 2023-11-21).

the corresponding execution traces can be quite long, in particular when A^*'s heuristic underestimates the cost to reach the goal state and often misleads the search to explore in-feasible paths.

As illustrated in Figure 1, all token sequences are synthetically generated and use a special vocabulary. Furthermore, every model is trained from scratch—no natural language dataset is used for training. Consequently, the resulting models are specifically trained to only predict sequences that outline optimal plans for a set of different planning tasks. After training, the model's output is parsed and evaluated if it contains a correct and optimal solution plan.

3.2 Training a Transformer model

To train a Transformer model, we randomly generate a set of planning tasks together with their corresponding token sequences. In each dataset, each task is unique and the test set is constructed such that it does not contain any duplicate of the training set. For example, a maze task that occurs in the training dataset does not occur in the test dataset and no task is repeated. With this experiment design, we hope to gain insight into how Transformers can be used to solve planning tasks.

By including intermediate computation steps, the Transformer model is trained to effectively imitate the computation performed by the A^* algorithm. This approach to training a neural network is similar to Procedure Cloning (Yang et al., 2022) where a neural network is not only presented with input-output pairs (in our case task prompts and optimal plans), but also with examples of how to "think" about a particular task

In our experiments, we train an adaptation of the encoder-decoder T5 architecture (Raffel et al., 2020) and use Rotary Position Embeddings (RoPE) (Su et al., 2023). More details and hyper-parameters about the models we use can be found in Appendix A.2. The encoder processes the <prompt> part of a training sequence, and the decoder processes either a <trace><plan> formatted sequence to train a search-augmented model or only a <plan> formatted sequence when training a solution-only model. Depending on the model variant, each network is trained to maximize the cross-entropy between the distribution of the decoder generations and the distribution of sampling a corresponding sequence from the training dataset. Gradient updates are computed using teacher forcing. Appendix A.1 describes our optimization setup in more detail.

3.3 Moving past algorithm imitation via search dynamics bootstrapping

To reduce the number of search steps generated by a search-augmented model during inference, we implement a method to shift the distribution with which the decoder generates execution traces. First, a search-augmented model is trained to imitate the search dynamic of our non-deterministic A^* implementation. This non-deterministic A^* implementation breaks cost ties randomly and randomizes the order with which child nodes are expanded. By doing so, we do not decrease the efficiency of A^* search itself and merely change the order with which different nodes are searched while still respecting A^*'s heuristic and cost calculations.

Using a non-deterministic implementation randomizes the sequence generation process and induces additional variance into the length of each execution trace. The resulting search-augmented model will then approximate the probability distribution with which the training sequences were generated.

Once a model is trained to imitate the search dynamics of non-deterministic A^* search, it is used to generate a *new* training dataset consisting of shorter token sequences. This new dataset is constructed by using the trained search-augmented model to generate multiple token sequences for each training prompt. In this step, we only use the training dataset for bootstrapping and not the test dataset. Each generated sequence is parsed and checked if it ends in an optimal plan. If this is the case and the sequence is also shorter than the corresponding sequence contained in the original training dataset, then this shortened sequence is included in the new short sequence training dataset. If the generated sequence does not end in an optimal plan or is longer, then the sequence from the original training dataset is re-used.

Subsequently, the search-augmented model is fine-tuned on the new short sequence training dataset. To distinguish from the search-augmented model trained on the original dataset, we call this new model *Searchformer*. This procedure can then be repeated by using the resulting fine-tuned model to generate the next even shorter sequence dataset and then using this dataset to further fine-tune the Searchformer model. In Section 4.4 we demonstrate that this procedure does in fact reduce the number of search steps

performed during inference while further improving performance. At this point, the Searchformer model no longer imitates A^* search and has instead discovered a new way of solving a planning problem using fewer search steps.

4 Experiments

In our experiments, we use a deterministic and a non-deterministic variant of A^* search to generate each sequence dataset.

Deterministic A^* dataset

In the first case, sequences are generated by executing A^* in a deterministic fashion (by ordering child nodes and breaking equal cost ties in the frontier set deterministically). Consequently, given a task prompt, the optimal plan and A^* execution trace is unique. A Transformer model trained on this data only has to learn which token to predict next. Evaluating such a model is simple, because the generated token sequences need to exactly match the token sequences generated by A^*.

Non–deterministic A^* dataset

In practice, there can be multiple plans that solve the same task optimally and a deterministic implementation of A^* search will only compute one optimal solution. To diversify the dataset and include different search sequences, we generate sequences by executing A^* search in a non-deterministic fashion: in each run, we randomly shuffle the order with which child nodes are expanded and break equal cost ties randomly. Consequently, the generated <trace><plan>-formatted sequences vary between different executions. The resulting plans may differ as well but are still optimal. This modification does not decrease the efficiency of A^* search, but merely forces our implementation to use different search sequences that still correspond to executing the pseudo-code listed in Figure 1(c). Other randomization strategies—which do influence A^*'s efficiency—are also possible (e.g., randomizing cost values at each step) and are left for future works.

Statistics

Figure 3 presents an overview of the token sequence length used for each model type for the Sokoban dataset and each maze datasets. The length of the generated A^* execution traces grows with the grid size of each maze. In Sokoban, the task with the largest state space, execution traces are the longest. However, the solution plans for Sokoban are amongst the shortest.
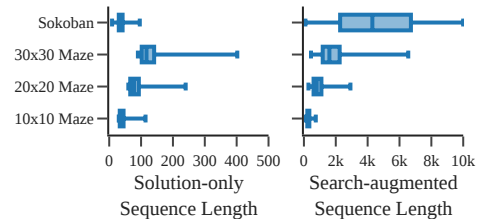

Figure 3 Training Sequence Length Comparison. The left panel plots the length of the solution-only sequences and the right panel plots the length of the search-augmented sequences, excluding the start and end of sequence tokens bos and eos. The whiskers indicate the range of all sequence lengths and the box plots indicate the 25%, 50%, and 75% quantiles. Because we focus on planning in complex sequential decision making tasks, the token sequences are multiple orders of magnitude longer than usual token sequences used to train LLMs—especially when A^* execution traces are included in the responses. For example, fine-tuning of the Llama 2 model on human preference data is performed with sequences that are on average 600 tokens long (Touvron et al., 2023).

4.1 Evaluation Criteria

For each task, one model may generate a search sequence ending either in an optimal plan, a correct but sub-optimal plan, or an invalid plan. For a plan to be valid, we not only check whether the final solution sequence is feasible, but also verify the output format. The search-augmented models must generate a token sequence of the format <trace><plan>, while solution-only models must generate a token sequence of the format <plan> only. Generated sequences violating such constraints—including sequences that do not predict an end-of-sequence token eos—are omitted.

4.1.1 Measuring plan optimality

We evaluate whether the output plan is optimal using one of three criteria:

- *Exact-match criterion.* For each task, if the generated sequence from a trained model matches the output of deterministic A^* exactly, it is labelled as correct, otherwise labelled as incorrect. This is only used to evaluate supervised cloning of deterministic A^*.

- *Any-optimal-64 criterion.* For each task, we sample 64 responses from a trained model. They are parsed and then evaluated to test if the final plan is feasible and optimal, regardless of the generated `<trace>` part. If any of the 64 plans is feasible and optimal, then the task is labelled as correct.

- *SWC score.* To further measure the sub-optimalness of the resulting plans, we also report the *Success Weighted by Cost (SWC)* (Wu et al., 2019), a statistic that factors in how close the cost l_i of the best predicted correct plan (over 64 trials) is to the optimal plan cost l_i^*, averaged across all n test tasks and weighted by a binary variable $c_i \in \{0, 1\}$:

$$\text{SWC} := \frac{1}{n} \sum_{i=1}^{n} c_i \frac{l_i^*}{\max\{l_i, l_i^*\}}.$$

  When computing the SWC score, the binary variable c_i is set to one if a correct plan was found and zero otherwise. This SWC score lies between zero and one. If all generated sequences end in an optimal plan, then this value is one.

4.1.2 Measuring search dynamics length

Given the plan is correct or optimal, we evaluate the length of sequence dynamics in terms of number of tokens using one of the two criteria:

- *Average-on-optimal length.* For each task, we sample 64 responses from a trained model and compute averaged search dynamics length for sequences that lead to an optimal plan.

- *ILR score.* To further measure how much improvement of the model-generated search dynamics against A^* planning, we report the *Improved Length Ratio of Search Dynamics* (ILR) score. Specifically, for each test task i, we compute the ratio between the length t_i of the shortest generated search dynamics sequence and the A^* token sequence length t_i^*. We then average across all test tasks while only including ratios for tasks for which either an optimal or a correct (and potentially sub-optimal) plan was found, as specified by the binary variable c_i. The corresponding measures are thus called *ILR-on-optimal* and *ILR-on-solved.* The ILR is defined as

$$\text{ILR} := \frac{1}{n} \sum_{i=1}^{n} c_i \frac{t_i^*}{t_i}.$$

  The ILR measure can take non-negative values and values above one indicate that the model generates shorter search dynamics than the A^* reference. Consequently, if the numbers lie significantly above one, then the model has found a more efficient way to search a task's state space to compute an optimal plan.

Unless indicated otherwise, each experiment is repeated five times and each figure plots averages across all repeats. Error bars are used to indicate the Standard Error of Measurement (SEM).
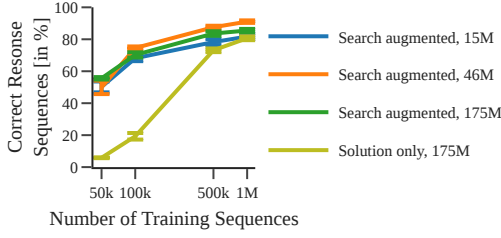
4.2 Maze Navigation

In the first experiment set, we train a set of encoder-decoder Transformer models to predict optimal plans for 30×30 mazes. We vary the training dataset size and model size (the number of optimized parameters) between different training runs and evaluate each model on the test tasks generated using the same hyper-parameters (e.g., maze size).
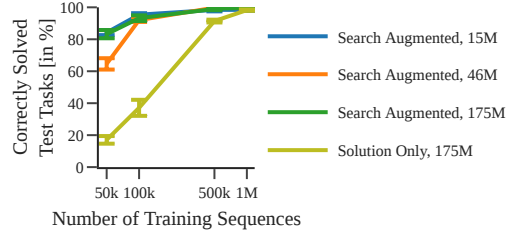
Evaluation with the exact-match criterion

Figure 4(a) plots for how many test tasks a correct response was generated. Both solution-only and search-augmented models are trained on the deterministic A^* dataset and follow the exact-match criterion. One

(a) Deterministic Prediction

(b) Non-deterministic Prediction

Figure 4 Predicting intermediate computation steps leads to robust performance in the low data regime. Each model was trained to predict optimal plans for 30×30 maze navigation tasks. The decoder of the search-augmented models were trained on sequences of the format `<trace><plan>` whereas the decoder of the solution-only models were trained on sequence of the format `<plan>`. For each model, the number of free parameters (indicated in millions of parameters with "15M", "46M", and "175M") is varied by changing the hyper-parameters of the encoder-decoder architecture. Appendix A.2 describes the used architecture and hyper-parameters in more detail.
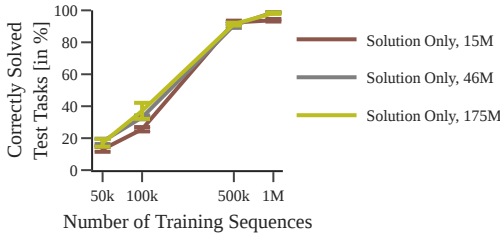

Figure 5 Solution-only model performance. Performance of the solution-only models is primarily influenced by the number of training sequences. Increasing a model's size does not always improve performance.
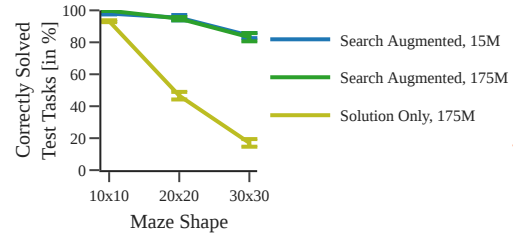

Figure 6 Performance across different task difficulties. Increasing task difficulty from left to right degrades performance of the solution-only models significantly. Here, each model was trained on 50,000 sequences.

can observe that the solution-only model is outperformed by most search-augmented models. Only for large enough training datasets, the solution-only model matches the performance of the worst search-augmented model. In the low training data regime (100,000 training sequences and less), performance of the solution-only model degrades significantly, while the performance of each search-augmented model stays relatively high.

This result is surprising, because the search-augmented models must generate a sequence of the format `<trace><plan>`, which is on average more than 10 times longer than the corresponding sequence of the format `<plan>` generated by the solution-only models. Furthermore, even the smallest search-augmented model with 15 million (15M) parameters significantly outperforms the much larger solution-only model with 175 million (175M) parameters.

Evaluation with the any-optimal-64 criterion

When trained on non-deterministic A^* data, there could be multiple different optimal paths for one task, and in practice one would only use the model to compute an plan, the any-optimal-64 metric is more appropriate. Note that using the any-optimal-64 criterion results in higher absolute numbers compared to exact-match, because to score one point under this criterion only one generated response needs to be correct.

Figure 4(b) plots the any-optimal-64 criterion on the test set as a function of the number of non-deterministic A^* training sequences. We can observe a similar pattern: even the smallest search-augmented models outperform solution-only model, especially for small training set. Moreover, we found that model size only impacts the performance of each of the search-augmented models when using very small training datasets (50,000 training sequences). For larger training dataset sizes no significant difference is found. Increasing the number of parameters of the solution-only models does not significantly improve their performance in the low data regime (Figure 5).

Performance across different task difficulty levels

Figure 6 illustrates how a task's difficulty influences the performance of each model. Here, we focus again on the dataset generated by non-deterministic A^*, and consider the number of correctly solved test tasks as a function of maze size. The larger the maze, the larger the task's state space and the more computation is required to find an optimal solution plan. While the solution-only model reaches close to the performance of search-augmented model for the small 10×10 mazes, its performance drops rapidly as the task becomes more challenging. In contrast, the search-augmented models maintain their high accuracy as the task becomes harder, even for its smallest model size. Appendix A.5 presents A full comparison across all maze sizes.

Overall, while the solution-only models learn to predict an optimal plan if the used training dataset is large and diverse enough, search-augmented models perform significantly better in the low data regime and scale better to more difficult tasks. The search-augmented models reach higher performance because they can perform on-demand computation during inference. More specifically, the search-augmented models can control the search dynamics for a grounded reasoning chain that leads to an optimal plan, while the solution-only models have to infer direct correlations between a task description and its optimal plan through supervised learning where many of such correlations can be spurious and unreliable in the test time.
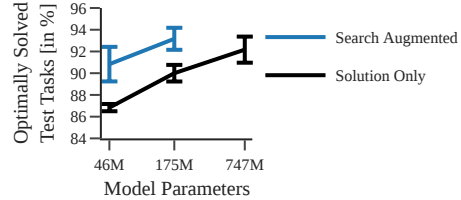

Figure 7 **Performance comparison with the Sokoban dataset.** Percentage of correctly answered test prompts for each model size. All models see 100,000 training sequences. The plot averages percentage values across three different seeds and evaluates each model on the same 200 test tasks. Test tasks are not included in the training dataset.

4.3   Solving Sokoban puzzles

To test if similar results can be obtained on a different and more complex task with a different tokenization pattern, we generate a planning dataset for Sokoban puzzles. Here, A^*'s execution traces are generated non-deterministically and expressed as a token sequence as outlined in Figure 10.

Figure 7 plots how often each model generated a correct optimal plan for each test task. As before, by training on execution traces, the search-augmented models outperform the solution-only models. Even increasing the parameterization of a solution-only model to 747 million parameters only leads to a marginal performance improvement. On average, this 747 million parameter solution-only model is still outperformed slightly by a smaller 175 million parameter search-augmented model. This experiment confirms the findings found for mazes on a dataset with more complex planning tasks that use a different tokenization method.

4.4   Searchformer: Improving search dynamics via bootstrapping

In this last experiment, we investigate how the search-augmented model can be iteratively improved to compute an optimal plan while predicting fewer search steps. Here, our goal is to shorten the length of the search trace while still producing an optimal solution.

We start out with the smallest search-augmented model trained on the non-deterministic A^* Sokoban dataset and use it to generate a new shorter sequence training dataset as outlined in Sec. 3.3. For each Sokoban puzzle in the training data, we generated 32 different `<trace><plan>` sequences by sampling tokens from the Transformer's output distribution and include the shortest generation (measured in tokens) if it contains an optimal plan.

Subsequently, we fine-tune the search-augmented model on this newly created training data (by running an additional 10,000 training steps) to obtain the first Searchformer model. Using this Searchformer model, we subsequently generate another short sequence dataset and repeat the fine-tuning procedure. This newly generated dataset is then used to further fine-tune the previously obtained Searchformer model for another 10,000 training steps. In total, we repeat this improvement step three times.

Figure 8 illustrates how the sequence lengths generated by the Searchformer model's are iteratively shortened by our bootstrapping method. With every improvement step, the length of the generated traces—the number
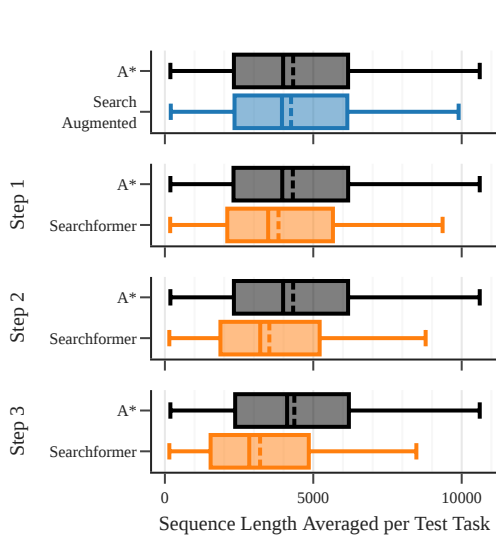
(a) Comparison of distribution of search dynamics length between A^* search and our models. Each test task contributes one data point of averaged length on trials that lead to optimal plans (i.e., *average-on-optimal length* (Sec. 4.1.2)).
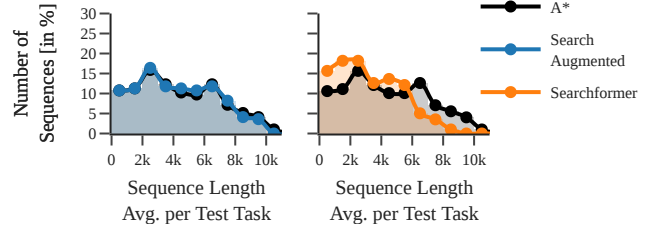
(b) Distribution of average-on-optimal length.

(c) Distribution of lengths of individual sequences that lead to an optimal plan.

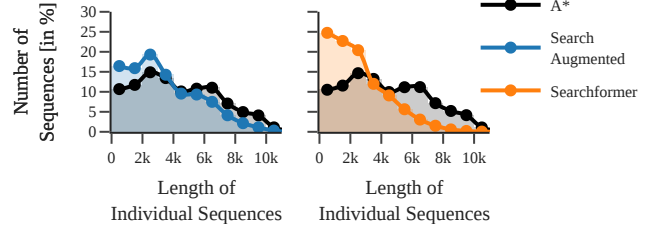**Figure 8 Improvement of search dynamics length on optimal plans via bootstrapping in Sokoban.** After three improvement steps, our Searchformer model reduced the generated search dynamics token length to an average of 3094—a 26.8% reduction in total from 4222, the average A^* search length. With each improvement step, the Searchformer models generated shorter and shorter token sequences when computing an optimal plan. (a): Comparison of search dynamics length (in terms of number of tokens) between A^* search and our models (search-augmented in blue, Searchformer step 1-3 in orange), on the test subset in which our models yield optimal plans. Here, for each test task, we average the lengths of sequences that ends in an optimal plan, out of 64 trials (i.e., average-on-optimal length (Sec. 4.1.2)). The box plots show the distribution of average-on-optimal lengths, in which the left boundary, mid solid line, right boundary represents the 25%, 50% and 75% quantiles. Dotted lines are means and whiskers show the range. (b): Histograms of average-on-optimal lengths. (c): Histograms of the length of individual sequences that lead to optimal plan. Because it is more likely for each model to generate an optimal plan for easy tasks with fewer search steps, the distributions are skewed towards the left—towards shorter sequence lengths. The data plotted in this figure aggregates sequence lengths across three experiment repeats.

Table 1 Test set performance in the Sokoban tasks. Over 200 unseen test Sokoban tasks, we report percentage of solved, percentage of optimal solutions, Success weighted by cost (SWC) and Improved Length Ratio (ILR) w.r.t A^* reference, averaged over sequences ending in an optimal/correct plan. Please check Sec. 4.1 for more detailed definitions of optimality metrics (Sec. 4.1.1) and search length metrics (Sec. 4.1.2). The result shows that continue fine-tuning Searchformer yields shorter search dynamics length and at the same time improved percentage of solved/optimally solved task. All values are averages with errors indicating SEM.

Params.	Model	Solved (%)	Optimal (%)	SWC	ILR-on-solved	ILR-on-optimal
45M	Solution only	90.3 ±1.0	86.8 ±0.3	0.890 ±0.009	−	−
	Search augmented	92.5 ±1.0	90.8 ±1.6	0.924 ±0.011	0.908 ±0.020	0.919 ±0.019
	Searchformer, step 1	95.5 ±1.0	93.5 ±1.0	0.953 ±0.010	1.054 ±0.025	1.062 ±0.015
	Searchformer, step 2	96.0 ±0.5	93.4 ±0.6	0.957 ±0.005	1.158 ±0.025	1.181 ±0.012
	Searchformer, step 3	95.5 ±0.8	93.7 ±1.6	0.953 ±0.009	1.292 ±0.044	1.343 ±0.067
175M	Solution only	95.7 ±0.2	90.0 ±0.8	0.949 ±0.003	−	−
	Search augmented	95.2 ±0.9	93.2 ±1.0	0.949 ±0.010	0.925 ±0.010	0.933 ±0.011
757M	Solution only	96.5 ±0.1	92.2 ±1.2	0.958 ±0.002	−	−

of search steps—decreases (Figure 8(a)). When computing an optimal plan, the final Searchformer model generates search dynamics sequences that are on average 26.8% shorter than the sequences generated with A^* search. Consequently, the Searchformer model found a more efficient way to plan a complicated task and go beyond A^*'s search, even for unseen test puzzles. In Figure 8(b) we can observe that the search-augmented model generates sequences that on average match the sequences generated with A^* search in length. The Searchformer models generate shorter sequences resulting in a distribution that is skewed towards shorter sequence lengths.

We find that on average each model produces an optimal plan more often for tasks that can be solved with fewer search steps. Generating an optimal plan for more complex tasks is possible with each model but requires the model to be prompted more often. This is illustrated in Figure 8(c) which plots the sequence length distribution for individual sequences. The sequence length distribution of both models is skewed towards shorter sequences and the models do not match the A^* sequence length distribution.

As reported in Table 1, fine-tuning the model resulted in a significant performance improvement, reducing the rate of incorrect and non-optimal solutions by 40% and 30% respectively. After three steps of fine-tuning, the 45M Searchformer shows similar to a 757M model that learns to predict a plan (task solution) only. The comparison using the more fine-grained SWC measures, which also take sub-optimal plans into consideration, shows a similar trend. See Sec. 4.1 for definitions of evaluation measures.

Reduced length of search dynamics with Searchformer

Lastly, we evaluate to which extend the search-augmented and Searchformer models reduced the number of generated search steps to compute a correct plan. This analysis is different from the results presented in Figure 8 because here we focus on computing length ratios between traces generated with each model and traces generated with A^* search using Improved Length Ratio (ILR) measure.

This improvement can be observed in Table 1 for each Searchformer model where with every improvement step the model generates shorter and shorter execution traces leading to higher ILR scores. Furthermore, the scores for the two search-augmented models are slightly below one, indicating that the search-augmented models tend to perform slightly more search steps than A^* search, while fine-tuned models has a ILR score higher than 1, showing more efficient search dynamics, in particular for those that lead to optimal plans. For example, A^* search dynamics is $\sim 34.3\%$ longer than that produced by Searchformer after 3 steps of fine-tuning. More details about model's performance characteristics are presented in Appendix A.4.

5 Conclusion, limitations, and future work

Prior work (Momennejad et al., 2023; Valmeekam et al., 2023a) has found that LLMs, which are based on Transformer models, struggle with solving complex decision making tasks. Searchformer demonstrates that with appropriate training data, Transformers can in fact solve complex planning tasks. Moreover, Searchformer

robustly follows the intermediate steps—the execution trace—of a symbolic planner and improves (in terms of trace length) beyond the human-crafted rule-based planning strategy it was initially trained on. Compared to solution-only models that directly predict a solution, our search-augmented models require fewer training sequences and scale better to more difficulty tasks.

Limitations

Currently, Searchformer is trained on the execution traces of A^* to learn a complex planning strategy. However, the trace length may grow exponentially in the length of an optimal plan (see Figure 3), and training on the resulting token sequence data become computationally very expensive. In fact, the presented experiments use token sequences that are significantly longer than the sequences used to train LLMs such as Llama 2 (Touvron et al., 2023).[2]

Future work

One way to mitigate this limitation is to use curriculum learning: starting from simple tasks with reasonably long execution traces, train and fine-tune the Searchformer to reduce the trace length, and then adapt the improved model to more complex tasks. Another possibility is to integrate better heuristics or value functions into A^* search, similar to Monte Carlo Tree Search (MCTS) (Coulom, 2006), to cap the maximal depth the search algorithm explores. Integrating hierarchical planning methods and temporal abstractions (Sutton et al., 2023, 1999; Dieterich, 2000; Hafner et al., 2022) are another avenue. This would equip the resulting model with the ability to abstract over multiple time steps and states to find an optimal plan using fewer computation steps.

We hope that our study informs further research about how to build Transformer-based models that can robustly solve decision making tasks and mitigate the limitations previously outlined by Momennejad et al. and Valmeekam et al..

6   Broader Impact

Our work focuses on symbolic planning tasks and uses synthetic datasets for training. While the tasks we explored in this paper can be easily solved with simple symbolic solvers, it is important to study the effectiveness of neural networks on such tasks. Neural network based Large Language Models have shown impressive performance on non-symbolic tasks, such as following complex natural language instructions, but still fail on relatively simple symbolic tasks like adding numbers. While the results in this paper are encouraging, they are still far from being competitive with symbolic solvers.

Acknowledgment

---

[2]The 175 million parameter model trained on the Sokoban dataset uses 64 GPUs with 32GB of memory.

References

Mahmoud Assran, Quentin Duval, Ishan Misra, Piotr Bojanowski, Pascal Vincent, Michael Rabbat, Yann LeCun, and Nicolas Ballas. Self-supervised learning from images with a joint-embedding predictive architecture. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15619–15629, 2023.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611*, 2017.

Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 9650–9660, 2021.

Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

Angela Fan, Thibaut Lavril, Edouard Grave, Armand Joulin, and Sainbayar Sukhbaatar. Accessing higher-level representations in sequential transformers with feedback memory. *CoRR*, abs/2002.09402, 2020. https://arxiv.org/abs/2002.09402.

Rohit Girdhar, Mannat Singh, Andrew Brown, Quentin Duval, Samaneh Azadi, Sai Saketh Rambhatla, Akbar Shah, Xi Yin, Devi Parikh, and Ishan Misra. Emu video: Factorizing text-to-video generation by explicit image conditioning. *arXiv preprint arXiv:2311.10709*, 2023.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Arthur Guez, Théophane Weber, Ioannis Antonoglou, Karen Simonyan, Oriol Vinyals, Daan Wierstra, Rémi Munos, and David Silver. Learning to search with mctsnets. In *International conference on machine learning*, pages 1822–1831. PMLR, 2018.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, Wolfgang Macherey, Arnaud Doucet, Orhan Firat, and Nando de Freitas. Reinforced self-training (rest) for language modeling, 2023.

Danijar Hafner, Kuang-Huei Lee, Ian Fischer, and Pieter Abbeel. Deep hierarchical planning from pixels. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. https://openreview.net/forum?id=wZk69kjy9_d.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet, 2023.

Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem. In *Advances in Neural Information Processing Systems*, 2021.

Michael Laskin, Luyu Wang, Junhyuk Oh, Emilio Parisotto, Stephen Spencer, Richie Steigerwald, DJ Strouse, Steven Stenberg Hansen, Angelos Filos, Ethan Brooks, maxime gazeau, Himanshu Sahni, Satinder Singh, and Volodymyr Mnih. In-context reinforcement learning with algorithm distillation. In *The Eleventh International Conference on Learning Representations*, 2023. https://openreview.net/forum?id=hy0a5MMPUv.

Ilya Loshchilov and Frank Hutter. SGDR: stochastic gradient descent with restarts. *CoRR*, abs/1608.03983, 2016. http://arxiv.org/abs/1608.03983.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

Ida Momennejad, Hosein Hasanbeig, Felipe Vieira, Hiteshi Sharma, Robert Osazuwa Ness, Nebojsa Jojic, Hamid Palangi, and Jonathan Larson. Evaluating cognitive maps and planning in large language models with cogeval, 2023.

MongoDB Inc. MongoDB. https://www.mongodb.com/. Accessed: 2024-01-23.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021.

OpenAI. Openai codex, 2021. https://openai.com/blog/openai-codex.

OpenAI. Openai: Introducing chatgpt, 2022. https://openai.com/blog/chatgpt.

OpenAI. Gpt-4 technical report, 2023.

Maxime Oquab, Timothée Darcet, Théo Moutakanni, Huy V. Vo, Marc Szafraniec, Vasil Khalidov, Pierre Fernandez, Daniel HAZIZA, Francisco Massa, Alaaeldin El-Nouby, Mido Assran, Nicolas Ballas, Wojciech Galuba, Russell Howes, Po-Yao Huang, Shang-Wen Li, Ishan Misra, Michael Rabbat, Vasu Sharma, Gabriel Synnaeve, Hu Xu, Herve Jegou, Julien Mairal, Patrick Labatut, Armand Joulin, and Piotr Bojanowski. DINOv2: Learning robust visual features without supervision. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. https://openreview.net/forum?id=a68SUt6zFt.

Vishal Pallagani, Bharath Muppasani, Keerthiram Murugesan, Francesca Rossi, Lior Horesh, Biplav Srivastava, Francesco Fabiano, and Andrea Loreggia. Plansformer: Generating symbolic plans using transformers, 2022.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 18–24 Jul 2021. https://proceedings.mlr.press/v139/radford21a.html.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. http://jmlr.org/papers/v21/20-074.html.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Anian Ruoss, Grégoire Delétang, Sourabh Medapati, Jordi Grau-Moya, Li Kevin Wenliang, Elliot Catt, John Reid, and Tim Genewein. Grandmaster-level chess without search, 2024.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 4 edition, 2021.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

Kurt Shuster, Jing Xu, Mojtaba Komeili, Da Ju, Eric Michael Smith, Stephen Roller, Megan Ung, Moya Chen, Kushal Arora, Joshua Lane, Morteza Behrooz, W.K.F. Ngan, Spencer Poff, Naman Goyal, Arthur Szlam, Y-Lan Boureau, Melanie Kambadur, and Jason Weston. Blenderbot 3: a deployed conversational agent that continually learns to responsibly engage. *ArXiv*, abs/2208.03188, 2022. https://api.semanticscholar.org/CorpusID:251371589.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550 (7676):354–359, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419): 1140–1144, 2018. doi: 10.1126/science.aar6404. https://www.science.org/doi/abs/10.1126/science.aar6404.

Uriel Singer, Adam Polyak, Thomas Hayes, Xi Yin, Jie An, Songyang Zhang, Qiyuan Hu, Harry Yang, Oron Ashual, Oran Gafni, Devi Parikh, Sonal Gupta, and Yaniv Taigman. Make-a-video: Text-to-video generation without text-video data. In *The Eleventh International Conference on Learning Representations*, 2023. https://openreview.net/forum?id=nJfylDvgzlq.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.

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

Richard S Sutton, Marlos C Machado, G Zacharias Holland, David Szepesvari, Finbarr Timbers, Brian Tanner, and Adam White. Reward-respecting subtasks for model-based reinforcement learning. *Artificial Intelligence*, 324:104001, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024. doi: 10.1038/s41586-023-06747-5. https://doi.org/10.1038/s41586-023-06747-5.

Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, and Subbarao Kambhampati. On the planning abilities of large language models – a critical investigation, 2023a.

Karthik Valmeekam, Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change), 2023b.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017. http://arxiv.org/abs/1706.03762.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022.

Yi Wu, Yuxin Wu, Aviv Tamar, Stuart Russell, Georgia Gkioxari, and Yuandong Tian. Bayesian relational memory for semantic visual navigation, 2019.

Mengjiao (Sherry) Yang, Dale Schuurmans, Pieter Abbeel, and Ofir Nachum. Chain of thought imitation with procedure cloning. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 36366–36381. Curran Associates, Inc., 2022.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

A Appendix

A.1 Training encoder–decoder Transformers to predict optimal plans

For this work, we consider encoder-decoder Transformers (Raffel et al., 2020) and process each prompt—the task specification—with an encoder network to obtain an embedding of a planning task. Using the decoder, the resulting embedding is then decoded into a response of the format `<trace><plan>` or the format `<plan>`. We refer to a model predicting sequences of the format `<trace><plan>` as a *search-augmented* model and a model predicting sequences of the format `<plan>` as solution-only models.

A.1.1 Encoder-decoder architecture

A token sequence is processed by a neural network by first indexing the set of all tokens contained in a specific dataset. This indexing is used to map any token sequence to a set of integers. Formally, we denote a prompt as a sequence of integers $x_{1:n} = (x_1, ..., x_n)$. An encoder Transformer neural network with network weights $\boldsymbol{\theta}_{\mathrm{enc}}$ is a function $f_{\boldsymbol{\theta}_{\mathrm{enc}}}$ mapping a tokenized prompt $x_{1:n}$ of arbitrary length n to a sequence of n real-valued vectors:

$$f_{\boldsymbol{\theta}_{\mathrm{enc}}} : x_{1:n} \mapsto \boldsymbol{z}_{1:n} \tag{1}$$

Each vector $\boldsymbol{z}_i$ in this sequence of vectors $\boldsymbol{z}_{1:n}$ has the same dimension. The decoder network is then used to generate a response auto-regressively: Starting with a specific beginning-of-sequence token `bos` to cue the decoder, a sequence is recursively built by first mapping the start token `bos` to a probability distribution over next-tokens. This probability distribution is stored as a vector $\boldsymbol{p}_1$ whose dimension is equal to the size of the vocabulary. The next token is then generated by sampling from this distribution and the sampled token is appended to the response sequence. Subsequently the two-token response sequence is fed into the decoder again to compute the next probability vector $\boldsymbol{p}_2$ and sample the next token. This procedure is repeated until an end-of-sequence token `eos` is sampled. While only the last computed probability vector is needed to sample the next token, the decoder network simultaneously predicts a sequence of next token probability vectors $\boldsymbol{p}_{1:m}$ given an input sequence $y_{1:m}$. Furthermore, this prediction is conditioned on the encoder output $\boldsymbol{z}_{1:n}$. The decoder Transformer neural network with weight parameters $\boldsymbol{\theta}_{\mathrm{dec}}$ is therefore a function

$$g_{\boldsymbol{\theta}_{\mathrm{dec}}} : \boldsymbol{z}_{1:n}, y_{1:m} \mapsto \boldsymbol{p}_{1:m}. \tag{2}$$

The encoder network $f_{\boldsymbol{\theta}_{\mathrm{enc}}}$ and decoder network $g_{\boldsymbol{\theta}_{\mathrm{dec}}}$ internally both use a number of stacked causal attention layers to form an encoder-decoder Transformer (Vaswani et al., 2017) as outlined in more detail in Appendix A.2. We denote a concatenation of all encoder parameters $\boldsymbol{\theta}_{\mathrm{enc}}$ and decoder parameters $\boldsymbol{\theta}_{\mathrm{dec}}$ with the vector $\boldsymbol{\theta}$.

A.1.2 Training with teacher forcing

An encoder-decoder architecture is optimized to generate responses that follow the distribution of a training dataset by minimizing the cross-entropy between the training data distribution $p_{\mathcal{D}}$ over prompt-response pairs (x_n, y_m) and the distribution $p_{\boldsymbol{\theta}}$ with which the encoder-decoder model is generating responses. This cross-entropy loss objective

$$H(p_{\mathcal{D}}, p_{\boldsymbol{\theta}}) = \mathbb{E}_{\mathcal{D}} \left[-\log p_{\boldsymbol{\theta}}(y_{1:m} | x_{1:n}) \right] \tag{3}$$

$$= \mathbb{E}_{\mathcal{D}} \left[-\sum_{i=1}^{m-1} \log p_{\boldsymbol{\theta}}(y_{i+1:m} | y_{1:i}, x_{1:n}) \right], \tag{4}$$

where line (4) follows from the auto-regressive generation procedure described before. Within the same planning dataset, different prompt-response pairs can have different prompt and response lengths. To emphasize shorter response sequences during training, we re-weigh each sample resulting in the loss objective

$$L(\boldsymbol{\theta}) = \frac{1}{D} \sum_{d=1}^{D} \frac{1}{m_d - 1} \sum_{i=1}^{m_d - 1} \log p_{\boldsymbol{\theta}}(y_{i+1:m_d}^d | y_{1:i}^d, x_{1:n_d}^d), \tag{5}$$

where the first summation averages over all D prompt-response pairs of the training dataset. In Equation (5) the super-script d indexes individual prompt-response pairs in the training dataset. This average is an
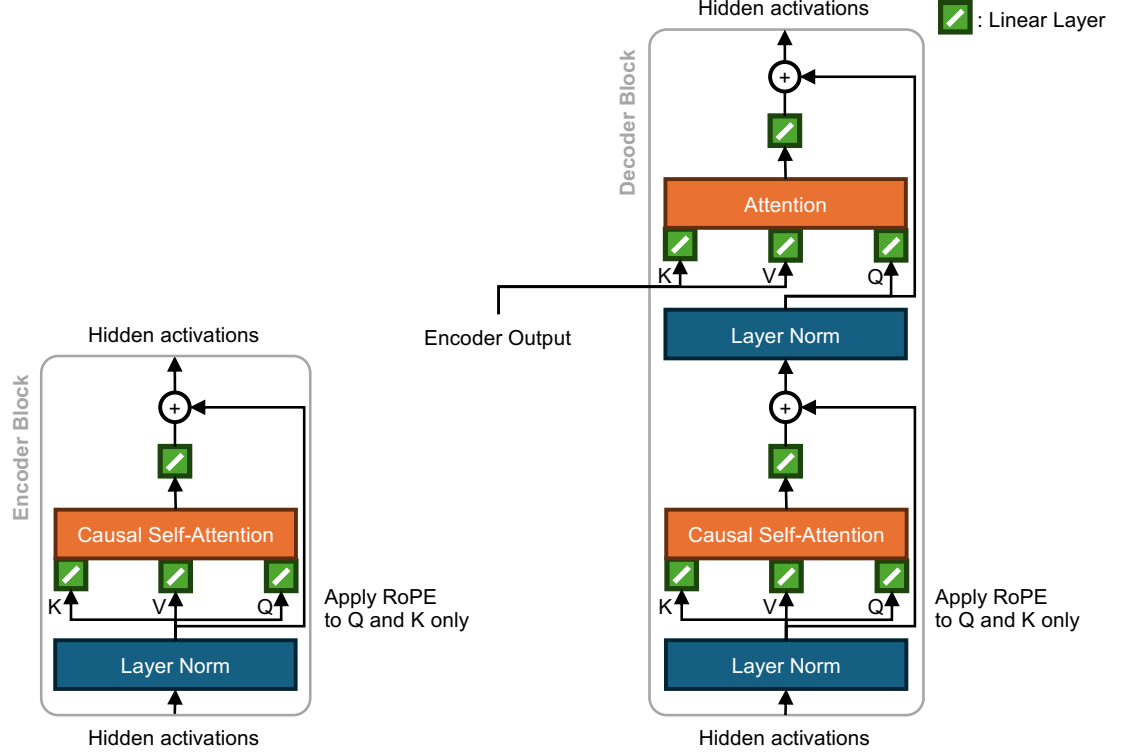
Figure 9 **Attention blocks architecture used in the encoder-decoder Transformer architecture.** The encoder network consists of a set of feed-forward layers where each layer processes hidden activations in parallel with a set of encoder attention blocks (left diagram). Similarly, the decoder network consists of a set of feed-forward layers composed of a number of decoder attention blocks (right diagram). The number of blocks in each layer is referred to as the number of heads. Token sequences are first mapped into integer sequences using a look-up dictionary. Then, these sequences are fed through a PyTorch (Paszke et al., 2019) `torch.nn.Embedding` layer to map the integer sequence into a sequence of hidden activation vectors. After the last decoder layer, hidden activations are mapped through a linear layer into a sequences of logits vectors to compute the next token probability vectors $\boldsymbol{p}_{1:m}$.

empirical approximation of the expectation in Equation (3) for a finite i.i.d. sample of size D. This loss objective is optimized using gradient descent (Goodfellow et al., 2016, Chapter 10).

A.2 Network architecture and hyper-parameters

The encoder-decoder Transformer first maps every token to a one-hot vector of the same dimension as the token vocabulary space. These one-hot vectors are then projected through a linear embedding layer into a set of vectors.

The encoder then consists of multiple feed-forward layers and each layer consists of multiple encoder blocks (left part of Figure 9). The output of these layers is then mapped through another linear layer to project the hidden activations into a tensor of the correct shape. The decoder also consists of multiple feed-forward layers and each layer also consists of multiple decoder blocks (right part of Figure 9) processing the hidden activations in parallel. As illustrated in Figure 9, the decoder network is conditioned on the encoder by processing the encoder's output tensor directly in the second attention map. Furthermore, each tokens position is encoded by applying RoPE embeddings (Su et al., 2023) as indicated in Figure 9. We did not use dropout in our architecture.

Table 2 lists the used architecture hyper-parameter and Table 3 lists the hyper-parameters used for optimizing each model. All experiments were implemented in PyTorch 2.0 (Paszke et al., 2019) and default parameters were used unless reported here otherwise.

17

Table 2 Architecture Hyper-parameters. The same parameters were used in both the encoder and decoder network. The number of heads indicates how many attention blocks are used in one layer. Layer dimension indicates the dimension of the feature vectors processed through each attention block (dimension of K, V, and Q in Figure 9). All models used a RoPE frequency of 10000.

Parameter	15M param. model	46M param. model	175M param. model	747M param. model
Layers	6	8	9	16
Heads	3	4	4	12
Layer dim.	64	96	192	96

Table 3 Optimization hyper-parameters. Every model was optimized using AdamW (Loshchilov and Hutter, 2019) with setting $\beta_0 = 0.9$ and $\beta_1 = 0.99$. Initially, the learning rate was linearly interpolated: Starting at zero and then increasing linearly to the value listed below until step 2000. Then a cosine learning rate schedule was followed (Loshchilov and Hutter, 2016). to the between zero at the first training step and the listed value below for the

Parameter	Model	Maze Tasks	Sokoban Puzzels
Learning rate	15M	$2.5 \cdot 10^{-4}$	$2.5 \cdot 10^{-4}$
	46M	$7.5 \cdot 10^{-5}$	$7.5 \cdot 10^{-5}$
	175M	$5.0 \cdot 10^{-5}$	$5.0 \cdot 10^{-5}$
	747M	$-$	$5.0 \cdot 10^{-5}$
Batch size	all	16	64
Training steps	all	400000	80000

```
Prompt   ⎧ bos
         ⎪ worker 2 3
         ⎪ box    2 4
         ⎪ box    3 4
         ⎪ dock   1 3
         ⎨ dock   4 4
         ⎪ wall   0 0
         ⎪ wall   0 1
         ⎪ ...
         ⎪ wall   6 6
         ⎩ eos
```

```
Trace            ⎧ bos
(2583 tokens)    ⎪ create worker 2 3 box 2 4 box 3 4 c0   c3
                 ⎨ close  worker 2 3 box 2 4 box 3 4 c0   c3
                 ⎪ ...
                 ⎪ create worker 5 4 box 2 3         c10 c3
                 ⎩ close  worker 2 1                 c12 c0
```

```
Plan     ⎧ plan 2 3
         ⎪ plan 1 3
         ⎪ plan 1 4
         ⎪ plan 1 5
         ⎪ plan 2 5
         ⎪ plan 2 4
         ⎪ plan 3 4
         ⎨ plan 2 4
         ⎪ plan 2 3
         ⎪ plan 2 2
         ⎪ plan 1 2
         ⎪ plan 1 1
         ⎪ plan 2 1
         ⎩ eos
```

Figure 10 Token sequence example for Sokoban This figure lists the token sequence for the Sokoban level depicted in Figure 2.

A.3 Dataset generation

All datasets were generated by first randomly sampling a task and then executing A^* to obtain an optimal plan. Maze tasks were generated first by randomly selecting 30-50% of all cells to be wall cells. Then a start and goal location was randomly selected and A^* was executed to obtain an optimal plan. If the plan had a length of at least the mazes width or height (e.g. for 10×10 mazes the optimal plan needs to contain at least

(a) Plan Sequence Length

(b) A^* Execution Trace Length

Figure 11 Sequence length distribution for each dataset. The training and test sets are designed such that their sequence lengths match closely.

10 steps), then the task was added into the dataset. For Sokoban, a 7×7 grid was sampled and two additional wall cells were added as obstacles to the interior of the map. Then two docks, boxes, and the worker locations were randomly placed. If the sampled task is solvable by A*, then the task was admitted to the dataset.

Due to the large volume of generated data, all data was stored in and transformed with a MongoDB (MongoDB Inc.) instance and map-reduce techniques. Furthermore, when sampling tasks, the dataset was constructed to reject duplicate tasks ensuring that training and test data and each prompt is distinct. Once each task and trace dataset was generated, each execution trace is converted into prompt and response token sequences, as illustrated in Figure 1(c) and Figure 11. Because the Sokoban tasks are very complex and difficult to solve for A*, the resulting token sequences were partially very long and reached almost 100000 tokens. Due to GPU memory requirements, the Sokoban dataset was further sliced to only include sequences of with at most 10000 tokens. Figure 11 compares the sequence length distribution for each dataset. During training, each dataset was also sorted and sliced to only contains the reported number of training sequences. Furthermore, each model was evaluated on the same test tasks within each task dataset. The test dataset contains plans and traces that are of comparable length to the training data (Figure 11).

A.4   Searchformer performance analysis

In Sec. 3.3, each search-augmented and Searchformer model is evaluated by generating 64 token sequences for each Sokoban test task. For the same test task the same model can generate sequences that end in an
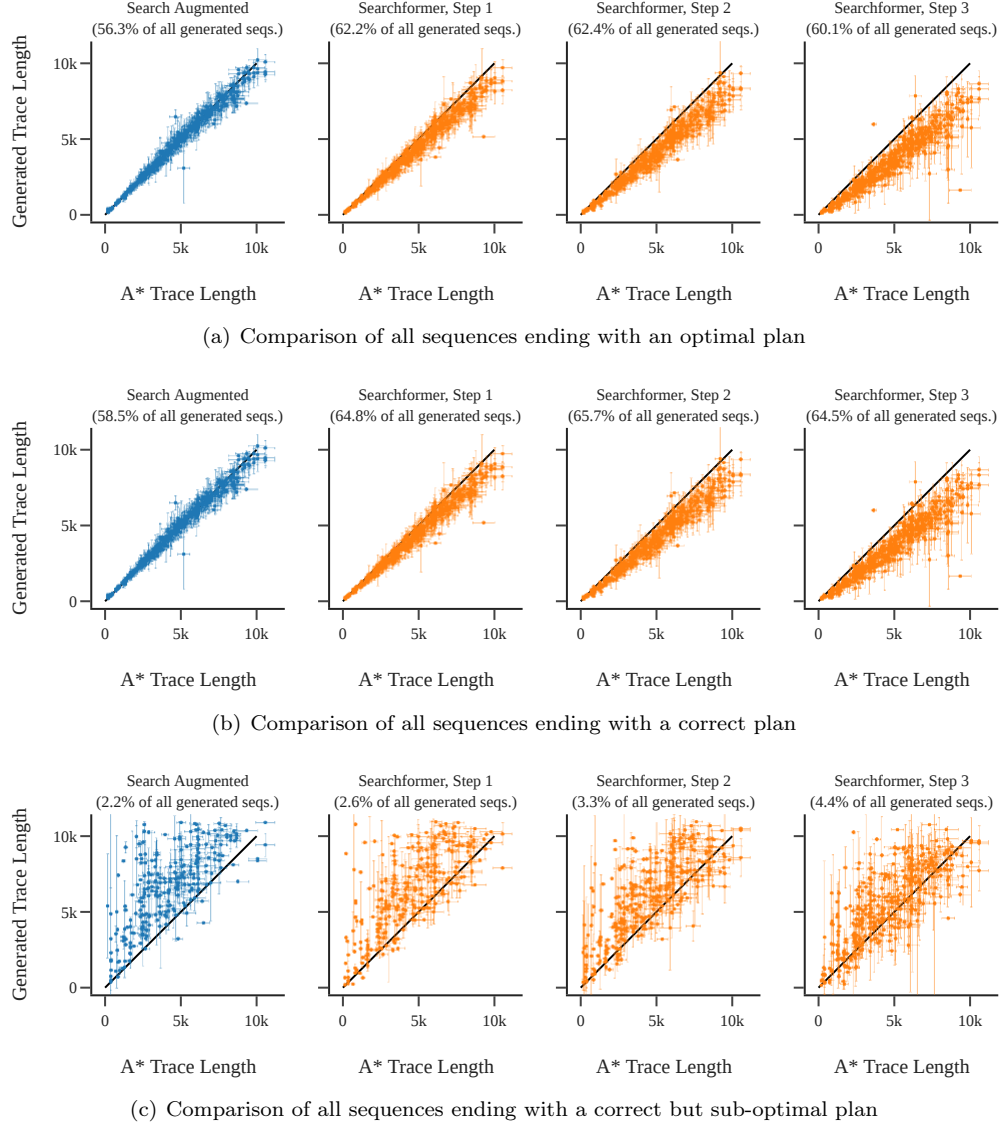
(a) Comparison of all sequences ending with an optimal plan

(b) Comparison of all sequences ending with a correct plan

(c) Comparison of all sequences ending with a correct but sub-optimal plan

Figure 12 Sequence length comparison between sequences generated with A^* search and sequences generated with each model. Each dot in each scatter plot corresponds to one specific test task. On the x-axis, we plot the average token sequence length when A^* search is used. On the y-axis, we plot the average token sequence length when each model is used. Error bars indicate standard deviations. Percentages indicate the fraction of the generated sequences that are included in each scatter plot. (a): Sequence length comparison for all test prompts for which an optimal plan was generated using the corresponding model. (b): Sequence length comparison for all test prompts for which a correct plan was generated using the corresponding model. This plot aggregates across sequences ending in an optimal plan and sequences ending in a correct but sub-optimal plan. (c): Sequence length comparison for all test prompts for which a correct but sub-optimal plan was generated using the corresponding model. This plot only aggregates across sequences ending in a correct but sub-optimal plan. Sequences ending in an optimal plan were withheld.

optimal plan, an incorrect plan, or a correct but sub-optimal plan. In Figure 12 we compare the length of the generated sequences with the length of sequences generated when using A^* search for each case. The percentages in each panel's caption list how many out of all generated sequences end in an optimal plan, a correct plan (that can be optimal or sub-optimal), and a correct but sub-optimal plan.

In the left panel of Figure 12(a) and Figure 12(b), points are centered around the diagonal axis indicating that the search-augmented models do approximately match the A^* search algorithm in terms of token sequence lengths. Figure 12(a) and Figure 12(b) further confirm the results presented in Sec. 3.3: With every improvement step, the points move down and below the diagonal line. This highlights that the improved Searchformer models generate token sequences that are shorter than sequences generated with A^* search. The Searchformer has found a method of searching a task to compute an optimal plan in fewer search steps than A^* search uses.

Figure 12(c) illustrates what happens when each model generates a correct but sub-optimal plan. Here, the search-augmented model, that is trained to imitate A^*, generates trace sequences that are significantly longer than the sequences generated with A^*. This suggests that the model struggles in computing a correct plan and generates too many search steps, ultimately leading in finding a correct but sub-optimal plan. Similarly, the Searchformer models also generate sequences that are longer than the sequences generated with A^* search. Despite these inefficiencies, our bootstrapping method is still able to improve the model and bring the average sequence length closer to the length of sequences generated with A^* search (right most panel in Figure 12(c)). While we would desire the trace length to be low in either case, we found that each model generates a correct but sub-optimal plan with less than 5% chance. Consequently, Figure 12(b) shows that the final Searchformer model still generates a correct plan with on average fewer search steps than A^* search. Statistically, the differences between Figure 12(a) and Figure 12(b) are marginal.

A.5 Supplemental maze experiments

(a) Deterministic Plan Prediction
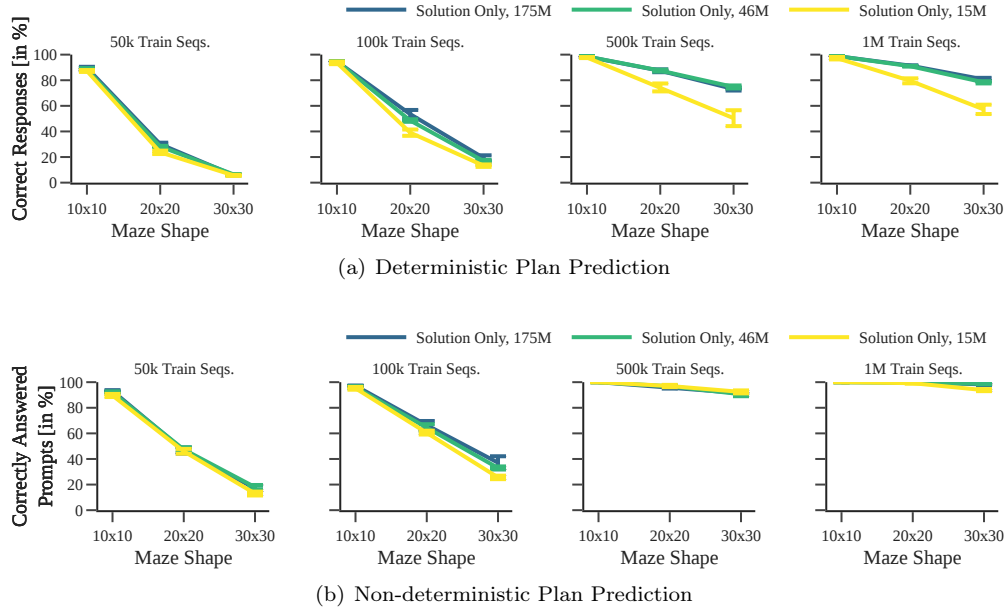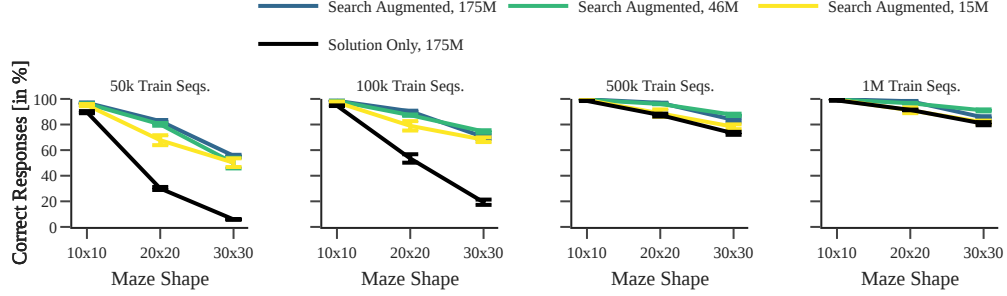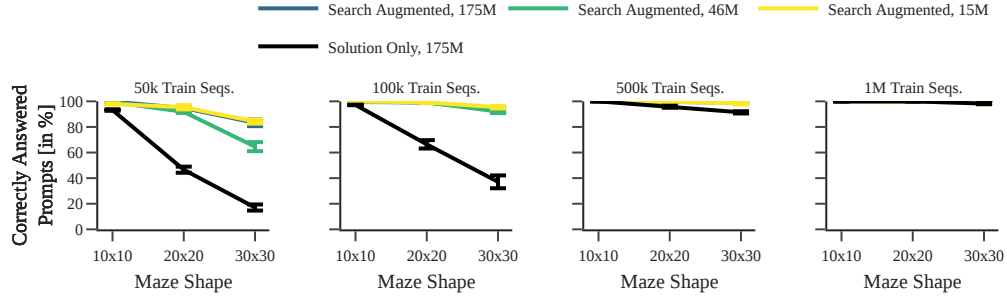
(b) Non-deterministic Plan Prediction

Figure 13 Optimal plan prediction performance for solution-only models. Each panel plots the percentage of correctly answered prompts averaged across five different seeds.

Figure 13 and Figure 14 plot how often a correct answer was provided by each model for all studied maze shape, training dataset size, and model parameterization combinations.

(a) Deterministic Prediction

(b) Non-deterministic Prediction

Figure 14 Optimal plan prediction performance for search-augmented models. Training an encoder-decoder Transformer to imitate A* execution traces significantly boosts performance. (a): Percentage of correctly generated responses. Note that the plan only models only need to generate a few hundred token long plan to answer a prompt correctly. The trace plan model must generate a much longer A* execution trace correctly to correctly answer a prompt. This requires the trace plan models to generate response sequences that are hundreds or thousands of tokens long (*cf.* Figure 3). If one token is incorrectly predicted, the response is counted as incorrect. (b): Percentage of correctly generated plans. Each model was used to generate multiple responses for each prompt. If one of the generated responses contains an optimal plan, the test prompt is counted as correctly answered.