Proceedings of the Thirty-Sixth International Conference on Automated Planning and Scheduling (ICAPS 2026)

Self-Improvement for Fast, High-Quality Plan Generation

Robert Gieselmann, Henrike von Huelsen, Mihai Samson, Marie-Christine Meyer, Dariusz Piotrowski, Oleksandr Radomskyi, Justin Okamoto, Turan Gojayev, Michael Painter, Gavin Brown, Federico Pecora, Jeremy L. Wyatt

Amazon

{robgie, vohenrik, samsonmf, mariemea, piotrod, radomole, justmoto, otugojay, painterm, brorgav, fpecora, jeremywy}@amazon.com

Abstract

Generative models trained on synthetic plan data are a promising approach to generalized planning. Recent work has focused on finding any valid plan, rather than a high-quality solution. We address the challenge of producing high-quality plans, a computationally hard problem, in sub-exponential time. First, we demonstrate that, given optimal data, a decoder-only transformer can generate high-quality plans for unseen problem instances. Second, we show how to self-improve an initial model trained on sub-optimal data. Each round of self-improvement combines multiple model calls with graph search to generate improved plans, used for model fine-tuning. An experimental study on four domains: Blocksworld, Logistics, Labyrinth, and Sokoban, shows on average a 30% reduction in plan length over the source symbolic planner, with over 80% of plans being optimal, where the optimum is known. Plan quality is further improved by inference-time search. The model's latency scales sub-exponentially in contrast to the satisficing and optimal symbolic planners to which we compare. Together, these results suggest that self-improvement with generative models offers a scalable approach for high-quality plan generation.

1 Introduction

Generative models have emerged as a useful tool for automated planning, providing a mechanism for retrieving and generalizing previously observed action sequences. Autoregressive models like transformers can quickly generate complete plan sequences that are then validated against formal verifiers like VAL (Howey, Long, and Fox 2004). Recent work (Rossetti et al. 2024b) has shown that this approach can achieve high completion rates across classical planning domains when trained on large datasets of valid plans. These generative models enable a form of *generalized planning* (Hu and De Giacomo 2011): once trained, they solve new problems within the same domain without requiring per-instance search.

Prior work such as Rossetti et al. (2024b) has focused primarily on plan validity rather than optimality. Generating optimal plans is computationally much harder than finding a valid solution. For instance, while Blocksworld admits linear-time algorithms, optimal planning in this domain is NP-hard (Slaney and Thiébaux 2001; Helmert 2003). This computational barrier makes it infeasible to generate large optimal training datasets, fundamentally limiting generative models to the quality of their training data.

To overcome this quality ceiling, we present *SiGPlan* (**S**elf-**i**mprovement for **G**eneralized **Plan**ning), a framework that iteratively refines a generative model to produce near-optimal plans without requiring optimal training data. Unlike search-based self-improvement frameworks such as AlphaGo (Silver et al. 2016), *SiGPlan* is designed to leverage the rapid candidate plan synthesis that generative models excel at. Starting from a model pretrained on sub-optimal plans, our self-improvement loop alternates between (1) discovering improved plans by combining model predictions with search, and (2) model finetuning on these improved labels. *SiGPlan* automatically derives a near-optimal, domain-specific generalized planner from sub-optimal training data.

We first establish an upper bound on generative model performance by training on optimal plans for Blocksworld, achieving a 78.9% optimality rate for problems with up to 100 blocks. We then evaluate *SiGPlan* on four domains: Blocksworld, Logistics, Labyrinth, and Sokoban. Across all domains, *SiGPlan* substantially improves upon the initial model, achieving plan length reductions of 39.3% on Blocksworld and 38.0% on Labyrinth, demonstrating that self-improvement can break through the supervised learning quality ceiling. Moreover, *SiGPlan* is computationally efficient, solving problems in seconds while optimal symbolic planners may require hours or days.

In summary, our main contributions are:

- An empirical analysis establishing the performance of generative models trained on optimal plans for Blocksworld with up to 100 blocks.
- *SiGPlan*, a self-improvement framework that iteratively refines generative planning models to approach optimal performance without requiring optimal training data.
- A benchmark across four domains demonstrating that self-improvement enables models to substantially exceed their initial training data quality, producing generalized planners with near-optimal efficiency.
- A runtime analysis showing that learned models achieve orders-of-magnitude speedups over optimal symbolic planners while maintaining near-optimal plan quality.

2 Background and Related Work

Automated Planning in PDDL. Automated planning (Nau, Ghallab, and Traverso 2004) finds action sequences that transform an initial state into one satisfying goal conditions. The Planning Domain Definition Language (PDDL) (McDermott et al. 1998; Fox and Long 2003) provides a standardized formalism for planning problems. Classical PDDL domains consist of predicates, object types, and operators with preconditions and effects. Problem instances include objects, an initial state, and goal specifications. States are sets of atoms (propositional statements combining predicates with object tuples) that hold true in specific configurations. PDDL has become the standard representation in planning research and serves as the foundation for competitions like the International Planning Competition[1]. We use PDDL as our main formalism, providing a general recipe to tokenize planning problems for training and sequence generation with transformer models.

Sequence Modeling with Transformers. Transformers (Vaswani et al. 2017) are a neural network architecture that have revolutionized sequence modeling across multiple domains. They leverage self-attention mechanisms to capture dependencies between all positions simultaneously, enabling efficient parallel computation. Multi-head attention allows joint attention to information from different representation subspaces, while position encodings preserve sequential information. The Generative Pretrained Transformer (GPT) (Radford et al. 2019) is designed for sequence generation, predicting sequences autoregressively by feeding generated tokens back as input. It scales effectively through parallelizable training via causal masking and inference speedup via kv-caching. The GPT architecture underpins modern large language models and has been adopted in domains where applications can be formulated as sequence prediction, e.g., offline reinforcement learning (Janner, Li, and Levine 2021) or Boolean satisfiability (Pan et al. 2025).

Pre-trained LLMs for Planning. Recent work has explored leveraging pre-trained language transformers for automated planning. Pallagani et al. (2023) rely on an LLM that was pretrained for coding tasks and finetuned it for optimal planning in several PDDL benchmark domains. They demonstrated that for small-scale problems (e.g., 2-5 blocks in Blocksworld), their approach outperformed the base LLM in both completion rate and plan quality. Moreover, the model generated plans significantly faster than an optimal symbolic solver. Other research has focused on LLMs for planning from natural language descriptions, including autoformalizing problems into PDDL (Liu et al. 2023), generating world models (Guan et al. 2023; Katz et al. 2024), and synthesizing domain-specific planning policies as Python code (Silver et al. 2024). Despite these promising results, there remains skepticism about large pre-trained transformers due to the high inference costs and limited reliability (Kambhampati et al. 2024).

Training Planning Generative Models from Scratch. Smaller, domain-specialized transformers have recently emerged as effective alternatives for automated plan generation. *PlanGPT* (Rossetti et al. 2024b) uses a large corpus of valid plans generated by a domain-independent planner to train domain-specific GPT transformers from scratch. The authors developed a specialized tokenization scheme representing actions as sequences of predicate and object tokens, achieving notable results across domains, such as 100% plan completion on unseen Blocksworld test problems with up to 20 blocks. In follow-up work, Rossetti et al. (2024a) integrated an action validator into the sequence generation pipeline to prevent infeasible plans and thereby improved completion rates. Tummolo et al. (2024) introduce an additional processing step which repairs invalid plans by using them as the initialization for a local planning algorithm. Fritzsche, Gestrin, and Seipp (2025) proposed a symmetry-aware contrastive learning objective to enhance *PlanGPT*'s ability to generalize to unseen configurations, especially problem instances with numbers of objects that were not seen during training.

PlanGPT is neither trained on optimal data nor evaluated in terms of the length of the generated plans. To address this gap, we devise a method that reduces plan lengths over time through an iterative process that alternates between search and learning.

Self-Improvement via Search. The idea of combining search and learning in a loop of self-improvement dates back to Arthur Samuel's checkers program (Samuel 1959). Jabbari Arfaee, Zilles, and Holte (2011) applied this concept to classical planning with heuristic search. Their method alternates between graph search and training a shallow neural network heuristic, where the current model guides search while solved instances update the model for subsequent iterations. Self-improvement with deep neural networks gained widespread attention with *AlphaGo* (Silver et al. 2016), which integrated Monte Carlo Tree Search with deep policy and value networks. Groshev et al. (2018) used a neural network as a learned policy and heuristic for generalized planning. Under the name *leapfrogging*, they introduce a system where model-guided A* search provides training labels for a curriculum of increasingly harder problem instances. In contrast, our work centers around autoregressive generative models, while focusing on improving plan optimality while maintaining fast runtimes. Recently, numerous studies have adopted search-based self-improvement frameworks to enhance transformers (Lehnert et al. 2024; Zhang et al. 2024; Chen et al. 2024). Our work applies this strategy to automated planning, leveraging the efficiency of smaller specialized models as demonstrated by *PlanGPT* (Rossetti et al. 2024b).

3 Training an Optimal Plan Generator

Our aim is to train a generative model that can generate (near-)optimal plans on unseen PDDL problem instances from the same distribution as the training data. In this section, we show that a generative model trained on optimal plans can generate optimal solutions for the majority of unseen test instances, and near-optimal solutions for the rest.

[1] https://ipc2023-classical.github.io/

3.1 Problem Definition

We consider single-agent planning problems defined over a finite set of objects $\mathcal{O}$ and predicates $\mathcal{P}$. Each predicate $p \in \mathcal{P}$ has arity arity(p). The set of grounded atoms is $\mathcal{X} = \bigcup_{p \in \mathcal{P}} \{ p(o_1, \ldots, o_{\text{arity}(p)}) \mid o_i \in \mathcal{O} \}$. The state space is $\mathcal{S} = 2^{\mathcal{X}}$, where each state $s \in \mathcal{S}$ is the set of propositions that are true in that world configuration. The action space $\mathcal{A}$ consists of all grounded operators $a = \langle \text{pre}(a), \text{add}(a), \text{del}(a) \rangle$, where $\text{pre}(a)$, $\text{add}(a)$ and $\text{del}(a)$ are subsets of $\mathcal{X}$ defining preconditions, add, and delete effects, respectively. The deterministic transition function is $F(s, a) = (s \setminus \text{del}(a)) \cup \text{add}(a)$ which applies when $\text{pre}(a) \subseteq s$. The goal space $\mathcal{G} \subseteq \mathcal{S}$ consists of partial states $g \in \mathcal{G}$, where a state s satisfies a goal g if $g \subseteq s$. A valid plan is a sequence of actions $\tau = (a_0, \ldots, a_{T-1})$ such that, starting from an initial state $s_0 \in \mathcal{S}$ and applying transitions $s_{t+1} = F(s_t, a_t)$, the resulting state s_T satisfies $g \subseteq s_T$ for a given goal $g \in \mathcal{G}$. Among all valid plans, we define the optimal solution as the one that minimizes the plan length.

We define $\mathcal{I} = \mathcal{S} \times \mathcal{G}$ as the space of problem instances, where each instance $(s_0, g) \in \mathcal{I}$ consists of an initial state $s_0 \in \mathcal{S}$ and a goal $g \in \mathcal{G}$. These problem instances are generated from an unknown distribution $P_{\mathcal{I}}$. Given a planning domain, our objective is to learn a fast plan generator that generalizes across problems from $P_{\mathcal{I}}$ while minimizing plan length. To achieve this, we assume access to a training set of problem instances $\mathcal{D}_{\mathcal{I}}$ generated by sampling from $P_{\mathcal{I}}$.

3.2 Generative Model

We train a generative model π which maps from a starting state $s_0 \in \mathcal{S}$, goal $g \in \mathcal{G}$, and action history $a_{<t} \in \mathcal{A}^t$ to a distribution over next actions: $\pi(a_t | s_0, g, a_{<t})$. Following Rossetti et al. (2024b), π is implemented as a decoder-only transformer (Radford et al. 2019) that operates at the token level. The model generates action sequences token-by-token through iterative forward passes, handling the combinatorially large space of grounded actions with a fixed-size vocabulary. For instance, in Blocksworld, the grounded action `unstack b1 b2` (unstacking block `b1` from block `b2`) is represented as a token sequence: the `unstack` operator token followed by tokens for `b1` and `b2`. The model operates autoregressively, appending each generated token to its input sequence after every prediction step.

The vocabulary is constructed from PDDL domain and problem files, comprising predicate names, object identifiers, types, and special delimiter tokens. For each domain, we define a maximum number of objects per type. Figure 1 illustrates the tokenization scheme during inference on a Blocksworld problem instance. The PDDL problem file (1a) is translated into a sequence of tokens (1b). Special tokens such as `[startofproblem]`, `[startofplan]`, and `[endofplan]` structure the input. The model then generates a plan as a sequence of action tokens, terminated by `[endofplan]` (1c). Generated plans are validated by splitting sequences into actions and verifying syntactic and semantic validity using VAL (Howey, Long, and Fox 2004).

During training, the model is optimized using the standard cross-entropy loss for next-token prediction.

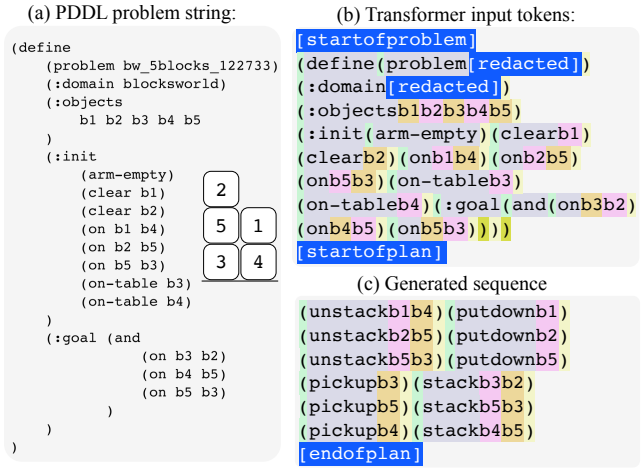

Figure 1: Illustration of our tokenization scheme.

3.3 Training from Optimal Plan Data

Obtaining optimal training data with domain-independent classical solvers is usually prohibitively expensive. However, for the Blocksworld domain, a domain-specific optimal solver was introduced by Slaney and Thiébaux (2001). Blocksworld is a common benchmark domain where blocks must be rearranged from an initial configuration to a goal configuration by stacking and unstacking one block at a time. We generate a training dataset of 999,270 Blocksworld problem instances of 3 to 100 blocks with optimal plans computed by the Slaney-Thiébaux solver. To satisfy the assumptions of the solver, problem instances are generated such that all objects appear in the goal specification. To establish whether efficient generation of (near-) optimal plans scales to larger problems, we evaluate performance of the trained model (Section 3.2) on three held-out test sets: 1000 problems each with 3-10 blocks, 11-25 blocks, and 26-100 blocks, respectively. During inference, we generate plans in batches of size 10 and report the best solution.

We compare the generative model against *Fast Downward* (FD) (Helmert 2006), a state-of-the-art domain-independent solver, in an optimal setting with a 20-minute timeout. To quantify plan quality (among solved instances), we report the percentage of optimal plans and the regret, which we define as the percentage length increase over the optimal solution: $\frac{\text{cost} - \text{cost}_{\text{optimal}}}{\text{cost}_{\text{optimal}}} \cdot 100\%$ (defined as 0% when $\text{cost}_{\text{optimal}} = 0$).

The results in Table 1 show that our model generates optimal or near-optimal plans across all three test sets (3-100 blocks). Even for the most challenging set with 26-100 block problem instances, the model maintained 97.5% completion rate. While the ratio of optimally solved instances dropped from almost 100% for less than 25 blocks to 39% for problems with 26-100 blocks, even the non-optimally solved problems were solved with less than 2% regret on average.

Notably, for the test sets with more than 10 blocks, the model shows a significantly lower runtime than *Fast-Downward*. While *Fast Downward* could not solve any problem instance with over 25 blocks within 20 minutes, the model solved almost all problem instances with an average

	3-10 blocks	11-25 blocks	26-100 blocks
Generative Model (GPT2 transformer)			
Compl. (%)	99.80	100.00	97.50
Optimal (%)	100.0	98.70	39.18
Regret (%)	0.00 (± 0.00)	0.06 (± 0.02)	1.65 (± 0.06)
Runtime (s)	0.60 (± 0.01)	1.85 (± 0.02)	9.53 (± 0.15)
FD-optimal (<20min)			
Compl. (%)	100.00	44.30	0.00
Runtime (s)	0.48 (± 0.06)	116.70 (± 10.76)	-

Table 1: Results of scalability analyses for 3 Blocksworld test sets containing 1000 unique problems each. For regret and runtime we report mean ± std. error.

runtime of under 10 secs. This underlines the prohibitively long runtime of classical (optimal) solvers, which grows exponentially with the problem complexity.

The analysis shows that even for complex problems, the transformer model was able to generate a high percentage of optimal plans (78.9% for 3-100 blocks), and near-optimal otherwise. The bottleneck, however, is computing optimal training data at scale, which is often infeasible in practice. This highlights the need for a method that can initially train on sub-optimal data and improve plan quality via self-refinement, which we address in the following section.

4 A Self-improving Plan Generator

We introduce *SiGPlan* (**S**elf-**i**mprovement for **G**eneralized **Plan**ning), a self-improving planning framework that bootstraps from sub-optimal solvers. As shown in Figure 2, our approach has three components: pretraining, plan improvement via graph search, and model finetuning.

We train an initial generative model π^0 from scratch on a dataset of sub-optimal plans ($\mathcal{D}_{\text{pretrain}}$) generated by an off-the-shelf, domain-independent planner (Figure 2a). We then begin our self-improvement loop. For m problem instances drawn randomly from $\mathcal{D}_{\mathcal{I}}$, we sample candidate plans from the current model to construct state graphs, then extract the shortest valid plans from these graphs via search (Figure 2b). The collection of improved plans forms a new dataset $\mathcal{D}^i_{\text{finetune}}$, which we use to finetune the generative model from the previous iteration. We choose finetuning over training from scratch as it requires significantly fewer new plans per iteration which is critical since generating these improved plans is computationally expensive due to numerous model calls. Preliminary experiments confirmed finetuning provides a better trade-off between plan quality improvement and computational cost. *SiGPlan* runs for n_{loop} steps, with each iteration i using model π^{i-1} to guide graph construction and extract improved training labels.

Next, we detail our framework's building blocks: the generative model, graph search, and model finetuning.

4.1 Graph Construction

A critical step within *SiGPlan* is to compute better plans for model finetuning. In summary, we first sample plans from our generative model and combine these to a graph representation of states and actions. To obtain improved training

Algorithm 1: Self-improvement Loop

Input: Problem set $\mathcal{D}_{\mathcal{I}}$, initial data $\mathcal{D}_{\text{pretrain}}$, number of problem instances for finetuning m, number of candidate plans per problem n, self-improvement iterations n_{loop}
Output: π

```
# Pretrain model on suboptimal data
π^0 ← pretrain(D_pretrain)
for i in 1 ... n_loop do
    # Sample subset of m problem instances
    D^i_I ← sample_problems(D_I, num=m)
    # Generate set of candidate plans for each problem
    {T_j} ← sample_plans(D^i_I, π^{i-1}, num=n)
    # Compute states (compile) and filter valid plans
    {T̂_j} ← compile_and_filter(D^i_I, {T_j})
    # Create set of state graphs
    {G_j} ← construct_graphs({T̂_j})
    # Search graphs and extract shortest plan data
    D^i_finetune ← search_and_extract_data({G_j})
    # Finetune plan generator
    π^i ← finetune(π^{i-1}, D^i_finetune)
end for
return π
```

At each iteration $i{=}1, \ldots, n_{\text{loop}}$, we start by drawing m problem instances uniformly from $\mathcal{D}_{\mathcal{I}}$. The resulting problem subset is denoted by $\mathcal{D}^i_{\mathcal{I}}$. For each problem $(s_0, g) \in \mathcal{D}^i_{\mathcal{I}}$, we use our most recent generative model π^{i-1} to synthesize n candidate plans $\{\tau_k\}_{k=1}^n$, where each $\tau_k{=}\{a_0, a_1, \ldots, a_{T_k-1}\}$ is a sequence of actions with individual length T_k. This sampling procedure can be performed efficiently by leveraging batch computation on GPUs. For each problem, the autoregressive generation stops either if the `[endofplan]` token is generated or a maximum token limit is exhausted. Generated sequences that do not follow the correct syntax of PDDL are discarded. The set of remaining plans for each problem instance $j \in \{1, \ldots, m\}$ is denoted by $\mathcal{T}_j$.

We continue by *compiling* the candidate plans in $\mathcal{T}_j$, using the transition model $s_{t+1}{=}F(s_t, a_t)$ to translate them into state-action sequences $\hat{\tau}_k{=}(s_0, a_0, s_1, a_1, \ldots, s_{T_k})$. During this process, we filter out invalid plans and retain only those satisfying the following conditions:

1. Actions a_t fulfill the operator preconditions at every step in the plan, i.e., $\text{pre}(a_t) \subseteq s_t$
2. The plan achieves the goal g, i.e., $g \subseteq s_{T_k}$, where $s_{T_k} = F(s_{T_k-1}, a_{T_k-1})$.

We collect all valid plans across the n candidates for each problem $j \in \{1, \ldots, m\}$ into a set $\hat{\mathcal{T}}_j$.

Finally, we construct a set of state graphs $\{G_j\}_{j=1}^m$, one per problem instance. Each directed graph $G_j = (V_j, E_j)$ is constructed such that V_j contains all unique states appearing in $\hat{\mathcal{T}}_j$, and E_j contains all unique state-action-state transitions present in $\hat{\mathcal{T}}_j$.

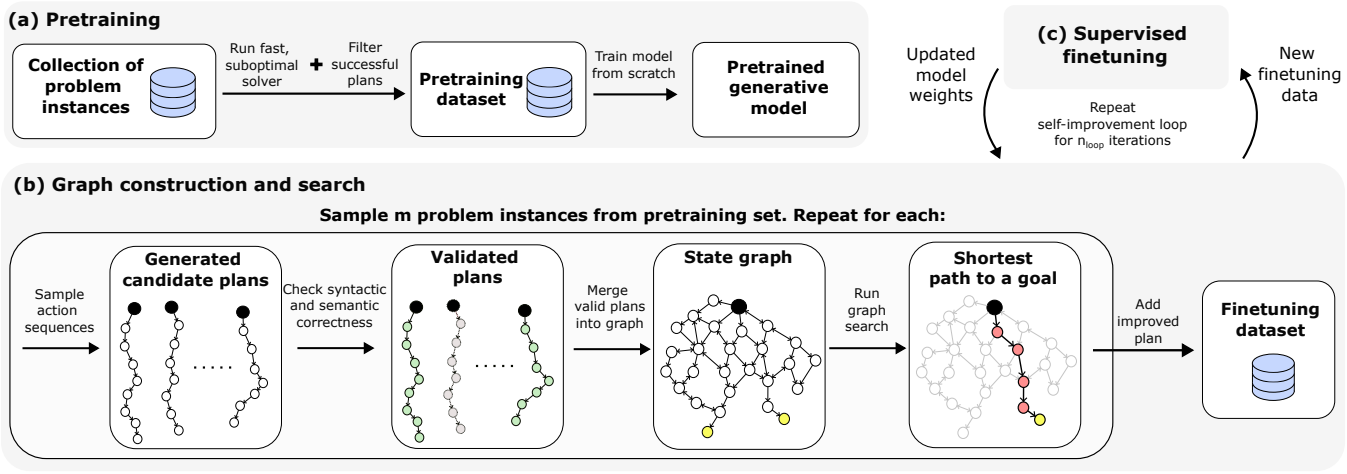

Figure 2: Overview of *SiGPlan*. (a) The generative model is pretrained on plans generated by a domain-independent planner. (b) For a subset of m problem instances, we sample candidate plans from our model, construct state graphs, and compute the shortest valid plans on these graphs. A finetuning dataset is created based on the best plan found for each problem in the subset. (c) The model is finetuned on the new data. Self-improvement iterates between (b) and (c) for a fixed number of steps.

4.2 Plan Harvesting and Finetuning

After obtaining the graphs $\{G_j\}$, we extract new plan labels to improve our model. For each graph G_j, we apply breadth-first search (BFS) to compute the shortest sequence of actions from the initial state to a goal. A new finetuning dataset $\mathcal{D}^i_{\text{finetune}}$ is created based on the shortest paths extracted from $\{G_j\}$ for all m problems. If no plan can be extracted from G_j for a problem instance j (i.e., when our generative model cannot synthesize valid plans), j is excluded from $\mathcal{D}^i_{\text{finetune}}$.

To prevent performance regression over iterations, we maintain a global best solution cache across all iterations. For each problem in $\mathcal{D}^i_{\text{finetune}}$, if a shorter or equally short plan was found in previous iterations or during pretraining, we use that plan instead. This ensures $\mathcal{D}^i_{\text{finetune}}$ contains only the shortest plans observed so far for each solved problem.

Finally, we update the previous iteration model π^{i-1} on $\mathcal{D}^i_{\text{finetune}}$ to obtain a new model π^i via supervised learning on the (problem, plan) pairs. To avoid overfitting on this finetuning data, we use a lower learning rate and fewer gradient updates compared to the pretraining phase.

5 Experiments

We benchmark *SiGPlan* to demonstrate how it effectively enhances plan quality when starting from a model initially trained on suboptimal data. We evaluate on four domains:

- **Blocksworld**: Introduced in Section 3.3. Here we use a more general setting where not all blocks are required to appear in the goal specification. Optimal planning in Blocksworld is NP-hard (Slaney and Thiébaux 2001).
- **Logistics**: A transportation domain where packages are delivered between airports and cities using trucks and airplanes. We consider instances with 1-50 cities, city sizes 1-5, 1-50 packages, 1-10 airplanes, and 1 truck per city.
- **Labyrinth**: A navigation domain where each grid cell represents an intersection with different connectivity patterns (e.g., L-shaped or I-shaped). The agent can move between connected cells, as well as shift entire rows or columns left/right/up/down, where cells pushed beyond one edge are reinserted at the opposite edge. We consider grid sizes of 3×3 and 4×4 cells.
- **Sokoban**: A puzzle domain where an agent pushes boxes to designated target locations while navigating around walls. We consider grid sizes of 5×5 to 14×14 cells with 1-10 boxes and 0-10 walls. Solving Sokoban is PSPACE-complete (Culberson 1998).

Initial Datasets and Pretraining. For each domain, we generate $|\mathcal{D}_{\mathcal{I}}|=|\mathcal{D}_{\text{pretrain}}|=10^5$ problem instances and compute corresponding suboptimal plans using *FD-LAMA-first* (Richter and Westphal 2010). We pretrain a generative model π^0 on $\mathcal{D}_{\text{pretrain}}$ for 60 epochs on Blocksworld, 30 on Logistics, and 100 on Labyrinth and Sokoban. The checkpoint with the lowest validation loss (computed on 128 held-out problems) is selected as the starting point for *SiGPlan*. We use 32 NVIDIA A100 GPUs (40 GB) for training and 100 parallel GPU workers for plan sampling. Blocksworld and Labyrinth completed 15 iterations within one day, Logistics within two days, and Sokoban within 3.5 days.

Hyperparameters. For all domains, we run $n_{\text{loop}}=15$ self-improvement iterations. At each iteration i, we sample $m=2000$ unique problem instances and construct graphs using $n=200$ generated plans per instance. For Logistics and Labyrinth, we use a transformer softmax temperature of 1, while for Blocksworld and Sokoban, a temperature of 2 provided a better trade-off between plan diversity and validity. We finetune on $\mathcal{D}^i_{\text{finetune}}$ for 30 epochs, empirically determined to improve the model without overfitting. The final checkpoint provides the model for the next iteration.

Evaluation Set and Methods. We evaluate on 1000 held-out problems per domain. These instances are sampled from

Method	Blocksworld		Logistics		Labyrinth		Sokoban	
	Compl. (%)	Plan length	Compl. (%)	Plan length	Compl. (%)	Plan length	Compl. (%)	Plan length
SiGPlan (n_{loop}=15, N=10)	99.40	41.86 (± 0.57)	99.30	146.12 (± 3.20)	98.70	15.50 (± 0.23)	94.40	118.79 (± 2.97)
SiGPlan (n_{loop}=15, N=10^3)	99.90	40.91 (± 0.55)	99.70	145.25 (± 3.16)	99.30	14.80 (± 0.20)	98.60	121.98 (± 2.96)
SiGPlan+BFS (n_{loop}=15, N=10^3)	99.90	40.86 (± 0.55)	99.70	145.20 (± 3.16)	99.30	14.61 (± 0.18)	98.60	121.91 (± 2.95)
π^0 (N=10)	99.80	68.98 (± 1.25)	99.30	161.85 (± 3.59)	96.70	25.01 (± 0.54)	94.60	134.26 (± 3.41)
π^0 (N=10^3)	99.90	59.85 (± 1.01)	99.90	157.10 (± 3.47)	100.00	18.13 (± 0.25)	99.20	132.52 (± 3.25)
π^0+BFS (N=10^3)	99.90	57.19 (± 0.94)	99.90	156.62 (± 3.46)	100.00	17.52 (± 0.23)	99.20	130.69 (± 3.17)
FD-LAMA-first (<20min)	100.00	79.43 (± 1.50)	100.00	165.32 (± 3.42)	100.00	25.78 (± 0.50)	100.00	149.05 (± 3.62)
FD-LAMA-anytime (<20min)	100.00	44.85 (± 0.72)	100.00	160.93 (± 3.57)	100.00	16.25 (± 0.32)	100.00	143.43 (± 3.84)
FD-optimal (<48h)	63.00	30.12 (± 0.48)	16.90	27.31 (± 1.48)	100.00	12.82 (± 0.13)	48.90	47.03 (± 1.08)

Table 2: Comparison of performance across domains. Plan length reports the mean (± standard error) for those problems which the respective method solved. N indicates the number of candidate plans generated per problem.

Method	Blocksworld		Logistics		Labyrinth		Sokoban	
	Optimal plans	Plan length	Optimal plans	Plan length	Optimal plans	Plan length	Optimal plans	Plan length
SiGPlan (n_{loop}=15, N=10)	495 / 625	30.83 (± 0.50)	95 / 169	28.29 (± 1.55)	619 / 955	15.24 (± 0.23)	406 / 468	47.14 (± 1.14)
SiGPlan (n_{loop}=15, N=10^3)	600 / 625	30.24 (± 0.49)	95 / 169	28.25 (± 1.55)	657 / 955	14.51 (± 0.20)	439 / 468	46.76 (± 1.12)
SiGPlan+BFS (n_{loop}=15, N=10^3)	606 / 625	30.20 (± 0.48)	95 / 169	28.25 (± 1.55)	662 / 955	14.37 (± 0.18)	442 / 468	46.73 (± 1.12)
π^0 (N=10)	76 / 625	46.42 (± 1.00)	62 / 169	30.31 (± 1.76)	284 / 955	24.80 (± 1.54)	195 / 468	52.24 (± 1.32)
π^0 (N=10^3)	103 / 625	40.83 (± 0.80)	63 / 169	29.80 (± 1.71)	357 / 955	17.77 (± 0.25)	247 / 468	50.12 (± 1.23)
π^0+BFS (N=10^3)	116 / 625	39.39 (± 0.76)	63 / 169	29.76 (± 1.71)	389 / 955	17.22 (± 0.24)	258 / 468	49.89 (± 1.22)
FD-LAMA-first (<20min)	62 / 625	52.47 (± 1.22)	47 / 169	31.21 (± 1.83)	246 / 955	25.35 (± 0.51)	118 / 468	57.25 (± 1.45)
FD-LAMA-anytime (<20min)	595 / 625	30.50 (± 0.51)	106 / 169	28.51 (± 1.62)	635 / 955	15.95 (± 0.32)	421 / 468	47.13 (± 1.16)
FD-optimal (<48h)	625 / 625	30.14 (± 0.48)	169 / 169	27.31 (± 1.48)	955 / 955	12.82 (± 0.13)	468 / 468	46.46 (± 1.11)

Table 3: Comparison of performance **on the intersection of problems for which all methods found a solution**. 'Optimal plans' gives the ratio between the number of problems for which the respective method found optimal plans, and the number of problems in the intersective set (e.g., out of 625 Blocksworld problems solved by all methods in the table).

the set of problems solved by *FD-LAMA-first* in less than 20 minutes. The following baselines are considered:

- *SiGPlan*: Our finetuned generative model after n_{loop} iterations of self-improvement
- π^0: The generative model pretrained on $\mathcal{D}_{\text{pretrain}}$.
- *FD-LAMA-first*: *Fast Downward* with LAMA-first configuration. This symbolic planner generated $\mathcal{D}_{\text{pretrain}}$.
- *FD-LAMA-anytime*: *Fast Downward* with LAMA anytime planning configuration (20min time limit).
- *FD-optimal*: *Fast Downward* with LM-Cut heuristic for optimal planning (48h time limit).

Test-time compute. For *SiGPlan* and π^0, we evaluate two inference settings: N=10 and N=10^3, where N is the number of candidate plans generated per problem instance. We report results for the shortest valid plan among the N candidates. Additionally, we evaluate *SiGPlan*+BFS and π^0+BFS, which augment the model predictions with graph construction and BFS (see Section 4.2).

Benchmark Results. Table 2 shows *SiGPlan*'s performance for all domains. Our method achieves high completion rates (98.6-99.9%) across all tested domains, compared to *FD-optimal* which solves 16.9-100.0% of problems within its 48-hour timeout. Despite allowing *FD-LAMA-anytime* a maximum compute time of 20 minutes, *SiGPlan* still achieves lower average plan cost across all domains. *SiGPlan* generates shorter plans compared to the pretrained generative model in all domains. Notably, plan lengths are reduced by 39.3% in the Blocksworld domain (from 68.98 to 41.86 with N=10), and by 38.0% for Labyrinth (from 25.01 to 15.50 with N=10). *SiGPlan* (N=10^3) improves both completion rates and plan quality across all domains, showing benefits from increased sampling. *SiGPlan*+BFS (N=10^3) yields only marginal improvements in plan length compared to *SiGPlan* (N=10^3), i.e., the additional search components provide minimal benefits beyond what the finetuned model already achieves. The benefits of search are, however, larger for the pretrained model, as shown by the plan length improvements of π^0+BFS (N=10^3) over π^0 (N=10^3) — e.g., a reduction from 132.52 to 130.69 in Sokoban. These findings support that *SiGPlan* scales to complex problems where optimal planners are intractable, while maintaining both high completion and short plans.

Convergence Behavior. Figure 3 shows average plan length and completion rates obtained with *SiGPlan* (N=10) over self-improvement iterations on 1000 validation instances. Across all domains, *SiGPlan* demonstrates rapid improvement in plan quality during the early iterations, with the most substantial gains occurring within the first 5 loop iterations. For Blocksworld and Labyrinth, the mean plan length decreases sharply from the pretrained model (69.7 to 41.8 and 25.5 to 15.6, respectively), with convergence stabilizing around iterations 8-10. Logistics exhibits more gradual but consistent model improvement throughout all 15 iterations (161.0 to 145.7). Sokoban shows initial improvement from the pretrained model (127.8) to iteration 6 (116.4), ultimately reaching 112.5 by iteration 15. Notably, completion

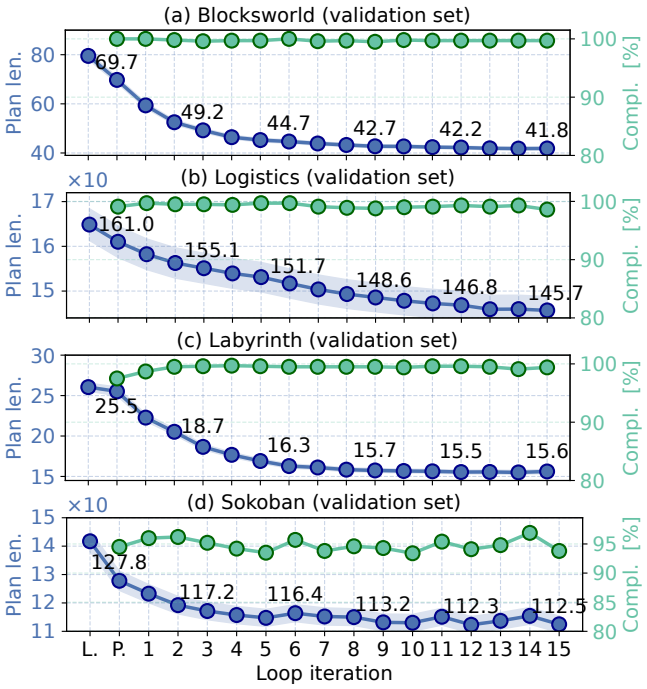

Figure 3: Plan length (mean ± std. error) and completion rates over self-improvement iterations (on 1000 unseen validation instances). 'L.' refers to FD-LAMA-first and 'P.' to the pretrained model.

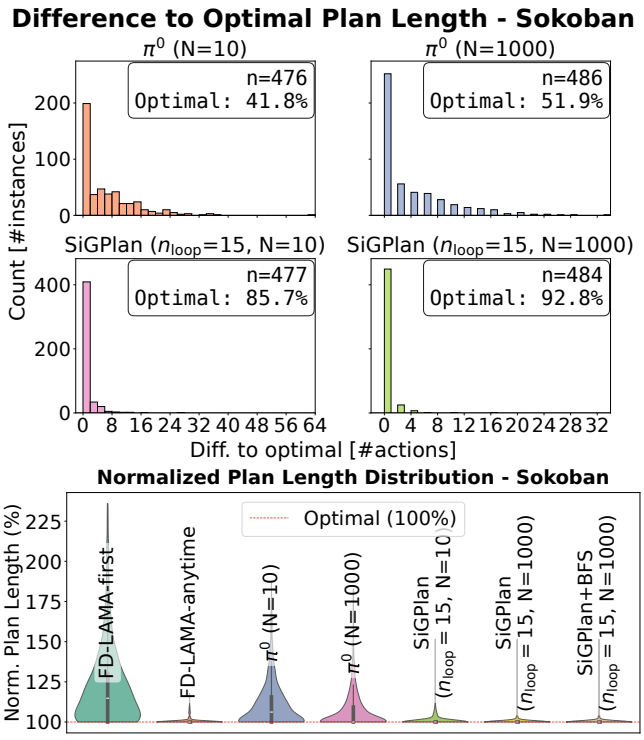

Figure 4: Plan length metrics for Sokoban. Histograms show the distribution of plan length differences to optimal. Violin plots show the distribution of normalized plan length, i.e., the plan length as a percentage of the optimal plan.

rates remain stable at near 100% for Blocksworld, Logistics, and Labyrinth throughout, indicating that the iterative refinement process improves plan quality without degradation.

How close are we to optimal? To obtain optimal baseline data, we ran *FD-optimal* for 48 hours on all test sets. Table 3 evaluates plan optimality on the subset of test problems solved by all methods. Using *SiGPlan* (n_{loop}=15, N=10^3), we optimally solve 600/625 (96%) of the Blocksworld problems, 95/169 (56.2%) of Logistics problems, 657/955 (68.8%) of Labyrinth problems, and 439/468 (93.8%) of the Sokoban problems. When adding BFS at inference time, the number of optimal plans increases, e.g. to 606/625 (97%) for Blocksworld and 442/468 (94.4%) for Sokoban. This substantially outperforms both the pretrained model π^0 and the classical *FD-LAMA-first* planner. Notably, the mean plan lengths produced by *SiGPlan* (N=10^3) closely approach those of *FD-optimal*: 30.24 vs. 30.14 for Blocksworld, 28.25 vs. 27.31 for Logistics, and 46.76 vs. 46.46 for Sokoban. Figure 4 shows that most plans generated by *SiGPlan* (N=10^3) in Sokoban are either optimal or few actions longer than the corresponding optimal plans. These results demonstrate that *SiGPlan* drives the model toward near-optimal and often optimal solutions.

Runtime Comparison. Table 4 presents the average test runtimes across domains. As shown, *SiGPlan* (N=10) requires on average only a few seconds of runtime. Despite requiring only a fraction of the time budget afforded to classical planners, *SiGPlan* produces shorter plans across all domains (Table 2), making it a compelling approach for fast and high-quality plan generation. Figure 5 compares the solving times of π^0 (N=10) and *SiGPlan* (n_{loop}=15, N=10) against *FD-optimal* on Blocksworld. For this comparison, we ran *FD-optimal* for 5 days on the test set, solving 643 instances. Both generative models solve problems in 0.1-10 seconds with consistent runtimes, while *FD-optimal* requires anywhere from few seconds to over 10^5 seconds (>27 hours). *SiGPlan* demonstrates similar runtime characteristics to π^0, indicating that the improvements in plan quality come without additional computational cost at inference time. This highlights a key advantage of learned models: they provide near-optimal solutions at a fraction of the computational cost required by optimal symbolic planners.

Method	Blocksw.	Logistics	Labyrinth	Sokoban
SiGPlan (n_{loop}=15, N=10)	0.49	2.15	0.54	4.08
SiGPlan (n_{loop}=15, N=10^3)	21.21	112.91	27.81	407.66
FD-LAMA-first (<20min)	0.39	7.32	270.15	12.70
FD-LAMA-anyt. (<20min)	985.82	765.71	935.28	796.18
FD-optimal (<48h)	13311.40	16670.74	3083.48	9762.96

Table 4: Average solving times in seconds on test sets across different domains (computed on solved instances only).

Statistical Analysis. While Table 2 contains the descriptive statistics comparing our methods on the two perfor-

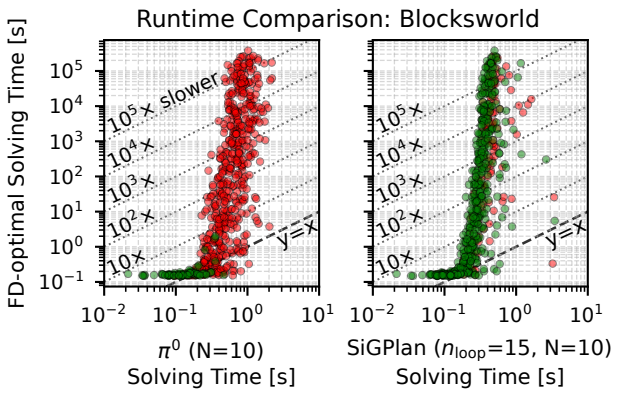

Figure 5: Runtime comparison between *FD-optimal* and the generative models. Green dots represent instances where the model generated a plan with same length as *FD-optimal*.

mance metrics plan length and completion rate, inferential statistics is necessary to establish which of the observed differences are meaningful rather than due to random variation. We therefore conducted a statistical analysis (on commonly solved instances) comparing six specific conditions: (1) π^0 vs. *SiGPlan* with $N{=}10$, (2) π^0 vs. *SiGPlan* with $N{=}10^3$, (3) π^0 with $N{=}10$ vs. $N{=}10^3$, (4) *SiGPlan* with $N{=}10$ vs. $N{=}10^3$, (5) *SiGPlan* ($N{=}10^3$) vs. *SiGPlan+BFS* ($N{=}10^3$), and (6) *SiGPlan+BFS* ($N{=}10^3$) vs. *FD-LAMA-anytime*.

For **Blocksworld**, all six comparisons of plan length were highly significant after Bonferroni-corrections ($p{<}0.001$). *SiGPlan* reduced mean plan length from 69.1 to 41.8 at $N{=}10$, and from 60.1 to 40.9 at $N{=}10^3$. Adding BFS improved plan quality significantly, but with a small effect size (*SiGPlan*: 40.90 vs. *SiGPlan+BFS*: 40.85). *SiGPlan+BFS* substantially outperformed *FD-LAMA-anytime* (40.9 vs. 44.9). Completion rates were consistently high (99.4–100%) with no significant differences.

For **Logistics**, plan length differences were again highly significant ($p{<}0.001$). *SiGPlan* yielded mean reductions of 15.9 actions with $N{=}10$ (161.4 to 145.5) and 11.6 actions with $N{=}10^3$ (155.9 to 144.4). Increasing candidate plans to $N{=}10^3$ reduced plan length by 5.5 for π^0 and by 1.1 for *SiGPlan*. Adding BFS improved plan quality significantly, but with a very small effect size (*SiGPlan*: 144.37 vs. *SiGPlan+BFS*: 144.33), and *SiGPlan+BFS* substantially outperformed *FD-LAMA-anytime* (144.3 vs. 159.8). Completion rates showed no significant differences.

For **Labyrinth**, all differences in plan length are highly significant ($p{<}0.001$) as well. *SiGPlan* reduced mean plan length from 24.8 to 15.2 with $N{=}10$, and from 17.8 to 14.5 with $N{=}10^3$. Adding BFS improved plan quality significantly, but with a small effect size (*SiGPlan*: 14.51 vs. *SiGPlan+BFS*: 14.37 actions), while *SiGPlan+BFS* substantially outperformed *FD-LAMA-anytime* (14.4 vs. 15.9 actions). In addition, completion rates varied meaningfully (96.7–100%), with $N{=}10^3$ increasing completion from 96.7% to 100% ($p{<}0.001$), and *SiGPlan* improving completion from 96.7% to 98.7% ($p = 0.022$) with $N{=}10$.

Plan length for **Sokoban** was again significantly different across all comparisons ($p{<}0.001$). *SiGPlan* reduced mean plan length by 14.6 with $N{=}10$ (130.5 vs. 115.9) and by 9.4 with $N{=}10^3$ (124.4 vs. 115.0). For π^0, increasing N from 10 to 1000 reduced plan length by 6.2, and by 0.9 for *SiGPlan*. Adding BFS improved plan quality significantly but only very slightly (*SiGPlan*: 115.04 vs. *SiGPlan+BFS*: 114.99). Lastly, *SiGPlan+BFS* again outperformed *FD-LAMA-anytime* (115.0 vs. 133.6). For completion rates, we found a significant effect of $N{=}10^3$ vs. $N{=}10$ for both π^0 and *SiGPlan* ($p{<}0.001$). *FD-LAMA-anytime* achieved significantly higher completion than *SiGPlan+BFS* ($p{<}0.001$), though all rates were high (94.4-100%).

6 Complexity Analysis

Plan validation runs in $\mathcal{O}(T{\times}M)$ time, where T is the maximum plan length and M is the maximum number of precondition and effect atoms per action. **Graph construction** inserts each unique transition from valid plans using hash table lookups, running in time linear in the total number of transitions with memory complexity $\mathcal{O}(D{\times}N)$, where D is the maximum rollout length and N is the number of generated plans. **BFS** explores the graph from the initial state in $\mathcal{O}(V{+}E)$, where V is the number of states and E the number of transitions. For graphs constructed from at least one valid plan, every graph contains at least one goal state, and BFS's exhaustive exploration in order of increasing depth guarantees that the shortest path is found, making it both complete and optimal with respect to the constructed graph.

7 Discussion & Conclusions

Finding efficient algorithms for optimal and near-optimal planning has been a long-standing goal in AI. We presented *SiGPlan*, a self-improving generalized planning system that combines transformer models with symbolic graph search to iteratively improve plan quality, scaling favorably compared to traditional symbolic solvers.

The resulting system produces optimal plans for 72.8% of the benchmark problems solved by *FD-optimal*, with average per-domain latencies ranging from 0.49 seconds (Blocksworld) to 4.08 seconds (Sokoban). Inference-time scaling increases this proportion to 80.8%, with latencies ranging from 21.21 seconds (Blocksworld) to 407.66 seconds (Sokoban), compared to 13311.40 seconds and 9762.96 seconds for *FD-optimal*. Unlike *FD-optimal*, model latency scales polynomially with the combined context and plan length. *SiGPlan* represents an important step toward computationally efficient near-optimal planning.

Future Work. Despite strong performance, several limitations remain. First, our approach requires an initial dataset of solutions, which may be hard to obtain for complex problems (e.g., Sokoban remains PSPACE-complete). Second, our exploration strategy is essentially on-policy, so convergence to optimal plans requires optimal actions to lie within the pretrained policy's support. Future work could incorporate off-policy exploration via A* or MCTS, and establish probabilistic guarantees for completeness or optimality.

Acknowledgments

The authors would like to thank Marc Toussaint, Alexander Melkozerov, Aaron Parness, Sara Anwar, and Michael Zhang for their valuable support and contributions to this work.

References

Chen, G.; Liao, M.; Li, C.; and Fan, K. 2024. AlphaMath Almost Zero: Process Supervision without Process. In *Neural Information Processing Systems*, volume 37.

Culberson, J. 1998. Sokoban is PSPACE complete. *Proceedings of the International Conference on Fun with Algorithms*, 65–76.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61–124.

Fritzsche, M.; Gestrin, E.; and Seipp, J. 2025. Symmetry-Aware Transformer Training for Automated Planning. *arXiv preprint arXiv:2508.07743*.

Groshev, E.; Goldstein, M.; Tamar, A.; Srivastava, S.; and Abbeel, P. 2018. Learning generalized reactive policies using deep neural networks. In *Intl. Conf. on Automated Planning and Scheduling*, volume 28, 408–416.

Guan, L.; Valmeekam, K.; Sreedharan, S.; and Kambhampati, S. 2023. Leveraging Pre-trained Large Language Models to Construct and Utilize World Models for Model-based Task Planning. In *Neural Information Processing Systems*.

Helmert, M. 2003. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2): 219–262.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Int. Res.*, 26(1): 191–246.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th IEEE Intl. Conf. on Tools with Artificial Intelligence*, 294–301. IEEE.

Hu, Y.; and De Giacomo, G. 2011. Generalized planning: Synthesizing plans that work for multiple environments. In *IJCAI Proceedings-Intl. Joint Conf. on Artificial Intelligence*.

Jabbari Arfaee, S.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artificial Intelligence*, 175(16): 2075–2098.

Janner, M.; Li, Q.; and Levine, S. 2021. Offline Reinforcement Learning as One Big Sequence Modeling Problem. In *Neural Information Processing Systems*, volume 34.

Kambhampati, S.; Valmeekam, K.; Guan, L.; Verma, M.; Stechly, K.; Bhambri, S.; Saldyt, L. P.; and Murthy, A. B. 2024. Position: LLMs can't plan, but can help planning in LLM-modulo frameworks. In *Intl. Conf. Machine Learning*.

Katz, M.; Kokel, H.; Srinivas, K.; and Sohrabi, S. 2024. Thought of Search: Planning with Language Models Through The Lens of Efficiency. In *Neural Information Processing Systems*, volume 37.

Lehnert, L.; Sukhbaatar, S.; Su, D.; Zheng, Q.; Mcvay, P.; Rabbat, M.; and Tian, Y. 2024. Beyond a*: Better planning with transformers via search dynamics bootstrapping. *arXiv preprint arXiv:2402.14083*.

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023. LLM+P: Empowering Large Language Models with Optimal Planning Proficiency. *arXiv preprint arXiv:2304.11477*.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the planning domain definition language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Nau, D.; Ghallab, M.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 1558608567.

Pallagani, V.; Muppasani, B.; Srivastava, B.; Rossi, F.; Horesh, L.; Murugesan, K.; Loreggia, A.; Fabiano, F.; Joseph, R.; Kethepalli, Y.; et al. 2023. Plansformer Tool: Demonstrating Generation of Symbolic Plans Using Transformers. In *IJCAI*.

Pan, L.; Ganesh, V.; Abernethy, J.; Esposo, C.; and Lee, W. 2025. Can Transformers Reason Logically? A Study in SAT Solving. In *Intl. Conf. on Machine Learning*.

Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I.; et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8): 9.

Richter, S.; and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Rossetti, N.; Tummolo, M.; Gerevini, A. E.; Olivato, M.; Putelli, L.; and Serina, I. 2024a. Enhancing GPT-based planning policies by model-based plan validation. In *Intl. Conf. on Neural-Symbolic Learning and Reasoning*. Springer.

Rossetti, N.; Tummolo, M.; Gerevini, A. E.; Putelli, L.; Serina, I.; Chiari, M.; and Olivato, M. 2024b. Learning general policies for planning through GPT models. In *Intl. Conf. on Automated Planning and Scheduling*, volume 34, 500–508.

Samuel, A. L. 1959. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3): 210–229.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587): 484–489.

Silver, T.; Dan, S.; Srinivas, K.; Tenenbaum, J. B.; Kaelbling, L.; and Katz, M. 2024. Generalized planning in pddl domains with pretrained large language models. In *AAAI Conf. on artificial intelligence*.

Slaney, J.; and Thiébaux, S. 2001. Blocks World revisited. *Artificial Intelligence*, 125(1): 119–153.

Tummolo, M.; Rossetti, N.; Gerevini, A. E.; Olivato, M.; Putelli, L.; and Serina, I. 2024. Integrating classical planners with gpt-based planning policies. In *Intl. Conf. of the Italian Association for Artificial Intelligence*, 315–329. Springer.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. *Neural Information Processing Systems*.

Zhang, D.; Zhoubian, S.; Yue, Y.; Dong, Y.; and Tang, J. 2024. ReST-MCTS*: LLM Self-Training via Process Reward Guided Tree Search. *arXiv preprint arXiv:2406.03816*.