Comparing State-of-the-art Graph Neural Networks and Transformers for General Policy Learning

Nicola J. Müller [1,3], **Pablo Sánchez**[1], **Jörg Hoffmann**[1,2], **Verena Wolf**[1,2], **Timo P. Gros**[1,2,3] *

[1]Saarland University, Saarland Informatics Campus
[2]German Research Center for Artificial Intelligence (DFKI)
[3]Center for European Research in Trusted Artificial Intelligence (CERTAIN)
{nmueller, sanchez, hoffmann, wolf, timopgros}@cs.uni-saarland.de, {verena.wolf, timo_philipp.gros}@dfki.de

Abstract

Graph Neural Networks (GNNs) have recently emerged as a powerful mechanism within the Artificial Intelligence (AI) research community, proving especially effective in a variety of applications from molecular structure prediction to enhancing recommender systems. In the realm of AI planning, the concept of general policy learning — which aims at creating agents capable of solving any instance within a specific domain — has gained significant attention. So far, the pursuit of general policies has often involved the use of custom-built GNN architectures tailored to unique graph representations of planning problems. These custom approaches, while effective, are heavily dependent on the construction of their underlying graph representation, which can limit their applicability and scalability. In this paper, we explore the feasibility of achieving similar successes in general policy learning using standard GNNs and Transformers, which have been extensively tested and researched; the latter are additionally not constrained by specific graph representations. Our findings indicate that while state-of-the-art GNNs and Transformers are generally suitable for general policy learning, their performance does not yet match that of the more specialized, custom-built GNN architectures previously developed in the field.

Introduction

Recently, *graph neural networks* (GNNs) have emerged as a powerful tool within the Artificial Intelligence (AI) research community. To name just a few applications, GNNs have demonstrated remarkable effectiveness in predicting molecular structures (Gilmer et al. 2017), facilitating drug discovery (Bongini, Bianchini, and Scarselli 2021), and enhancing recommender systems (Fan et al. 2019).

AI planning, at its core, is about devising sequences of actions, so-called *plans*, to achieve specific goals. This process is typically applied across various *instances* of a planning *domain*, aiming to identify the optimal sequence of actions that transitions from the instances' clearly defined start state to meet its established goal conditions.

The remarkable successes of GNNs have not gone unnoticed within the AI planning community. Consequently, researchers started to leverage graph neural networks as *general policies*.[1]

Toyer et al. presented the pioneering approach, utilizing a CNN-inspired architecture to compute general policies for probabilistic planning (2018). Their architecture features modules specific to predicates and action schemas. Sharma et al. introduced an expressive graph representation of RDDL states, addressing the capture of long-range pairwise relationships between nodes (2023). Rivlin, Hazan, and Karpas utilized a novel architecture combining layers of both GNNs and so-called Transformers, addressing several classical planning problems through training with deep reinforcement learning (2020). Furthermore, Ståhlberg, Bonet, and Geffner developed a relational GNN architecture for learning general value functions for classical planning problems (2022a; 2022b). Their work puts focus on the expressive limits of GNNs and proposed the integration of derived atoms to overcome these limitations. Further, they investigate how variations in their GNN's structure impacts the ability to solve different domains.

However, each of these approaches employs a custom-built GNN architecture based on their custom-built graph representation. Simultaneously, there is a growing research

*This work was partially supported by the German Research Foundation (DFG) under grant No. 389792660, as part of TRR 248, see https://perspicuous-computing.science, by the German Research Foundation (DFG) - GRK 2853/1 "Neuroexplicit Models of Language, Vision, and Action" - project number 471607914, and by the European Regional Development Fund (ERDF) and the Saarland within the scope of (To)CERTAIN.

[1]Essentially, a *policy* can be understood as a trained *agent* that possesses the capability to compute a plan for a specific instance of a domain. The ambition behind a general policy goes a step further: creating an agent capable of solving any instance of a specific domain.

community studying GNNs, establishing state-of-the-art architectures. These well-examined architectures have already addressed some of the issues that custom-built GNNs for general policy learning have to face, e.g., failing to capture relationships between distant nodes (Wu et al. 2021; Li et al. 2021), or being unable to distinguish certain non-isomorphic graphs (Rampášek et al. 2022; Morris et al. 2019).

Lately, Horcik and Šír proposed various graph representations for classical planning states and delved into the expressive capabilities of state-of-the-art GNNs by quantifying the number of states indistinguishable by the policies (2024). However, they do not utilize these GNNs to investigate plans.

With this paper, we aim at bridging the gap between these two research strands. Rather than crafting a more advanced and specialized custom GNN along with a specialized graph representation, we explore the potential of standard GNNs that have undergone extensive testing and research. Starting from scratch, we introduce an intuitive graph representation of classical planning states that is both expressive and architecture-agnostic. We refrain from customizing or specializing this graph representation; instead, we assess the ability of standard GNNs to embody general policies using this graph representation.

Additionally, we consider the widely successful *Transformer* architecture (Vaswani et al. 2017) as an alternative to GNNs. Transformers, in contrast to GNNs, do not depend on any graph representation. Therefore, while the performance of GNNs relies on the underlying graph structure, Transformers do not have such a strong restriction but can directly be applied to the states' representation. Thus, their performance is not limited by a (possibly misconstructed) graph representation.

Our investigations reveal that state-of-the-art GNNs and Transformers are viable for general policy learning, yet they have not achieved the performance levels of the custom-built GNNs by Ståhlberg, Bonet, and Geffner (2022a).

The contribution of our paper is outlined as follows. First, we introduce graph neural networks and focus specifically on the state-of-the-art architectures that we will utilize in our analysis. Additionally, we discuss the limitations of GNNs and subsequently introduce Transformers. We then present our general graph representation for classical planning states. Afterward, we analyze the proposed approaches in our experiments and conclude our work.

Graph Neural Networks and Transformers

In this section, we lay the foundation for the remainder of the paper, starting with an overview of Graph Neural Networks, detailing the specific GNN models employed in our study, and concluding with an exploration of Transformer architectures.

Graph Neural Networks

A *graph neural network* (Scarselli et al. 2008) represents a parameterized function f over graphs that is invariant to graph isomorphism and can be applied to graphs of any size. Given an undirected graph G, with nodes $V(G)$ and edges $E(G)$, a GNN computes for every node $v \in V(G)$ an embedding h_v capturing information regarding the local structure of v in G. The computation of a node embedding $h_v^{(l)}$ consists of L so-called *layers* l, where the initial embedding $h_v^{(0)}$ corresponds to the node's feature vector as specified by the graph G. The computation of a single layer can be divided into two steps:

1. AGGREGATE: The embeddings $h_u^{(l)}$ of all nodes u in the so-called 1-hop neighborhood $N(v)$, i.e., $\forall u \in V(G) : (u, v) \in E(G)$, are aggregated.
2. COMBINE: The aggregated embeddings $h_u^{(l)}$ and the node's current embedding[2] $h_v^{(l)}$ are combined to produce an updated embedding $h_v^{(l+1)}$.

Thus, these two steps jointly can be written as the general GNN update rule

$$h_v^{(l+1)} = \textsc{Combine}(h_v^{(l)}, \textsc{Aggregate}(\{\{u \in \mathcal{N}(v)\}\})), \tag{1}$$

where $\{\{\dots\}\}$ denotes a multiset. The update is applied to each node $v \in V(G)$ simultaneously and for each layer l iteratively. Essentially, the choice of AGGREGATE and COMBINE functions is what differentiates most GNN architectures.

After computing the final node embeddings $h_v^{(L)}$, a prediction of the function f is made for a node v by passing the embedding $h_v^{(L)}$ to another neural network, such as a *multi-layer perceptron* (MLP). For making graph-level predictions, we apply the so-called READOUT operation,

$$h_G = \textsc{Readout}(\{\{h_v^{(L)} | v \in G\}\}), \tag{2}$$

where the embeddings $h_v^{(L)}$ of all nodes $v \in V(G)$ are aggregated into a global embedding h_G before being passed to the final MLP. There are several choices for the readout operation, such as taking the maximum, the mean, or the sum across each dimension, where the latter has been shown to yield the most expressive function approximator (Xu et al. 2018), and thus will be used in our experiments.

Using a GNN, we can learn a general policy by defining a suitable graph representation for planning states and then training the GNN to, for any given state s, either predict an action a or state value $V(s)$, corresponding to the cost of reaching a goal state s_g from s. In the latter case, we learn a *general value function*, which induces a general policy when we always take the actions leading to the successor states s' with the lowest state value $V(s')$.

State-of-the-art GNNs

In the following, we will introduce the three state-of-the-art GNN architectures used in our experiments.

Graph Convolutional Network. The graph convolutional network (GCN) (Kipf and Welling 2016) is a simple but widely used architecture utilizing the update rule

$$h_v^{(l+1)} = \text{ReLU}(W \cdot \text{MEAN}(\{\{h_u^{(l)} | u \in \mathcal{N}(v) \cup \{v\}\}\})), \tag{3}$$

[2]The current embedding is given by the node features for the first layer and the value from the former layer otherwise.

where the aggregation function corresponds to the element-wise mean operator, and the combination function corresponds to a linear neural network layer parameterized by the weight matrix W, followed by the non-linear ReLU activation function.

Graph Attention Network v2. The *graph attention network v2* (GATv2) (Brody, Alon, and Yahav 2021) utilizes a dynamic attention mechanism, enabling it to learn which neighbors of a node contain the most relevant information. Its update rule can be written as

$$h_v^{(l+1)} = \text{ReLU}\left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \alpha_{v,u} \cdot W h_u^{(l)}\right), \quad (4)$$

where the normalized attention coefficients $\alpha_{u,v}$ are computed as

$$\alpha_{v,u} = \frac{\exp(a^\top \text{LeakyReLU}(W \cdot \left[h_v^{(l)} || h_u^{(l)}\right]))}{\sum_{u' \in \mathcal{N}(v) \cup \{v\}} \exp(a^\top \text{LeakyReLU}(W \cdot \left[h_v^{(l)} || h_{u'}^{(l)}\right]))}, \quad (5)$$

and a corresponds to a learned weight vector. We can further improve performance and stabilize learning by applying multi-head attention, where the GATv2 update is computed K times in parallel using separate weight matrices W^k and weight vectors a^k, and the results are concatenated

$$h_v^{(l)} = \Big\Vert_{k=1}^{K} h_v^{k,(l)}. \quad (6)$$

Whereas GCN assigns equal importance to all neighbors during aggregation, GATv2's attention coefficients $\alpha_{v,u}$ determine the importance of each neighbor node u's embedding $h_u^{(l)}$ for the update of node v, such that the updated embedding $h_v^{(l+1)}$ can be computed as a weighted sum over all $h_u^{(l)}$ and $h_v^{(l)}$.

Graph Isomorphism Network. The expressivity of GNNs has been shown to be limited to that of the Weisfeiler-Leman (WL) test, meaning that they cannot distinguish certain types of non-isomorphic graphs (Morris et al. 2019). However, many GNNs are even less expressive. Xu et al. introduced the *graph isomorphism network* (GIN) (2018) and have proven that it as expressive as possible, i.e., as expressive as the (WL) test. Its update rule corresponds to

$$h_v^{(l+1)} = \text{MLP}^{(l)}\left(\left(1 + \epsilon^{(l)}\right) \cdot h_v^{(l)} + \sum_{u \in \mathcal{N}(v)} h_u^{(l)}\right), \quad (7)$$

where ϵ is a fixed scalar. Further, GIN introduces a special readout operator

$$h_G = \Big\Vert_{l=0}^{L} \sum_{v \in V(G)} h_v^{(l)}, \quad (8)$$

where the embeddings $h_v^{(l)}$ computed in *every* layer l are summed up and then concatenated into a global embedding h_G. This might be beneficial, as embeddings from earlier layers may contain important structural information not included in the last layer. Note that in contrast to this architecture, GNNs usually only use the last layer and not all layers to compute the global embedding.

GNN Limitations

Since every application of the GNN update rule lets information travel to nodes from their 1-hop neighborhood, the number of GNN layers L determines the distance that information can travel in any given graph. Specifically, information can only flow between two nodes along up to L hops (Wu et al. 2021). Restricting information flow to local neighborhoods provides GNNs with a strong inductive bias, making learning very efficient (Wu et al. 2020). However, if there are important relationships between distant nodes in a graph, the GNN may be unable to capture them. Depending on the graph representation, this can prevent general policies from solving large instances, e.g., Ståhlberg, Bonet, and Geffner observed that their GNN architecture could not solve some instances of domains requiring the computation of distances exceeding the number of layers L (2022b). Increasing L to a large value can address such issues, but, in general, it leads to deteriorating performance, as, with increasing depth, all nodes tend to aggregate information from the same neighborhoods, causing all embeddings to converge to the same value (so-called over-smoothing). Further, increasing L exponentially increases the number of neighbors which every node considers, meaning each fixed-size node embedding needs to process an exponentially increasing amount of information, likely leading to information loss (so-called over-squashing) (Alon and Yahav 2020).

Another limitation of GNNs for general policy learning is that, due to their limited expressivity, they may be unable to distinguish certain states with different optimal state values or actions (Horcik and Šír 2024).

Transformers

GNNs are popular for general policy learning because they can process graphs of any size, allowing a trained GNN to solve instances of any size. However, *recurrent neural networks (RNNs)* (Hochreiter and Schmidhuber 1997) and *Transformers* (Vaswani et al. 2017) can also handle variable-sized input data, where the latter has emerged as the superior architecture in recent years (Lin et al. 2022). We note that since classical planning states are relational structures and not graphs (Ståhlberg, Bonet, and Geffner 2022a), we are not restricted to using GNNs for general policy learning and, thus, should also investigate the benefits of utilizing Transformers.

Transformers represent parameterized functions f over sequences which are divided into separate inputs x_i, so-called tokens. Given N tokens x_i, a Transformer computes an embedding h_i for every x_i, which are subsequently used to predict f. Similarly to the GNN, the embedding computation consists of L layers, where each comprises two main steps:

1. ATTENTION: The embedding $h_i^{(l)}$ of token x_i is updated

by aggregating the embeddings $h_j^{(l)}$ of *all* other tokens x_j.

2. POSITION-WISE FEED-FORWARD NETWORK: The embedding $h_i^{(l+1)}$ is further updated by passing it to a feed-forward network.

ATTENTION. The Transformer's attention mechanism is commonly referred to as *self-attention*, and involves three steps: First, we compute a query q_i, key k_i, and value vector v_i for each embedding $h_i^{(l)}$ using learnable linear transformations. Second, we multiply the query q_i of each embedding $h_i^{(l)}$ with the keys k_j of *all* embeddings $h_j^{(l)}$, and then apply the softmax function to obtain normalized attention coefficients

$$\alpha_{i,j} = \frac{exp(q_i k_j)}{\sum_{j'=0}^{N-1} exp(q_i k_{j'})}, \tag{9}$$

modeling the pair-wise importances of the $h_j^{(l)}$ to $h_i^{(l)}$. Third, we compute the updated embeddings $h_i^{(l+1)}$ as weighted sums over the value vectors v_j of *all* embeddings $h_j^{(l)}$

$$h_i^{(l+1)} = \sum_{j=0}^{N-1} \alpha_{i,j} \cdot v_j. \tag{10}$$

In practice, self-attention is computed for all embeddings simultaneously using matrix multiplications

$$\text{ATTENTION}(Q, K, V) = softmax\left(\frac{QK^\top}{\sqrt{d_k}}\right) V, \tag{11}$$

where dividing the attention scores by the root of the queries' and keys' dimension d_k has been shown to increase performance. Similarly to the GATv2 architecture (6), we can compute the more expressive multi-head attention by applying multiple self-attention heads in parallel and concatenating the results.

POSITION-WISE FEED-FORWARD NETWORK. We further update each embedding $h_i^{(l+1)}$ by separately giving them to the same *feed-forward network* (FFN) consisting of two fully connected layers and a non-linear activation function[3]. To reduce the chance of overfitting, a dropout layer (Srivastava et al. 2014) is applied, which randomly deactivates a fixed percentage of the FFN's neurons during training.

Predicting f. Both sub-layers of the Transformer layer have residual connections around them (He et al. 2016), followed by a normalization layer such as layer normalization (Ba, Kiros, and Hinton 2016), which stabilizes training. After computing the embedding $h_i^{(L)}$ of each token x_i, we make a prediction of the function f by summing all token's embeddings across each dimension and passing the result to an MLP. We can pass a classical planning state either directly to a Transformer by appropriately encoding the ground atoms true in the state as tokens, or we can reuse an existing graph representation by using the nodes feature vectors as tokens, disregarding the graph's connectivity. In our experiments, we will do the latter to enable a fair comparison between GNNs and Transformers.

[3]In this case, the feed-forward network is an MLP, but we here stick to the more general term of FFN used in the original paper (Vaswani et al. 2017)

State-of-the-art Transformers

We now introduce the Transformer-based architectures utilized.

Performer. The *Performer* (Choromanski et al. 2020) architecture reduces the quadratic time and space complexity of self-attention by computing a linear approximation of the softmax attention matrix. This linear attention mechanism ensures that our Transformer-based general policies can scale to large instances efficiently.

General, Powerful, Scalable Graph Transformer. The *general, powerful, scalable graph Transformer* (Rampášek et al. 2022) (GPS) processes a graph in every layer by passing it in parallel to a GNN and Transformer layer, and then combines the resulting node embeddings using an MLP. This allows GPS to attain both the GNN's strength of capturing structural information from nodes' local neighborhoods and the Transformer's strength of capturing long-range relationships between nodes. GPS can be used with any GNN and Transformer layer, where if the latter uses a linear attention mechanism, the overall complexity of GPS is linear in the number of nodes and edges, making it more efficient than previous Graph Transformer approaches. In our experiments, we will use GCN as the GNN layer and Performer as the Transformer layer. Note that, given GPS's nature as a hybrid architecture, its performance might be influenced by the quality of the provided graph representation, even with the flexibility offered by the Transformer component.

Comparing Transformers and GNNs

It is interesting to compare Transformers and GNNs on a theoretical level as both architectures have many similarities. The Transformer's self-attention is similar to the GNN's aggregation, with the main difference being that self-attention exchanges information between the embeddings of all tokens, whereas GNN aggregation exchanges information only within each node's local neighborhood. Hence, Transformers can model arbitrary pair-wise relationships between their input tokens, independent from the distance of the corresponding nodes. In a nutshell, a Transformer applied to graphs can be seen as a GNN where all nodes are assumed to be connected to each other, i.e., information can flow between any possible nodes. A drawback of using Transformers is that because they make only small assumptions about their input's structure, they typically require more training data than architectures with stronger inductive biases, such as GNNs (Lin et al. 2022).

Graph Representation

This section introduces our graph representation for classical planning states represented in the *planning domain definition language* (PDDL). We seek a simple graph representation that contains all necessary information while avoiding

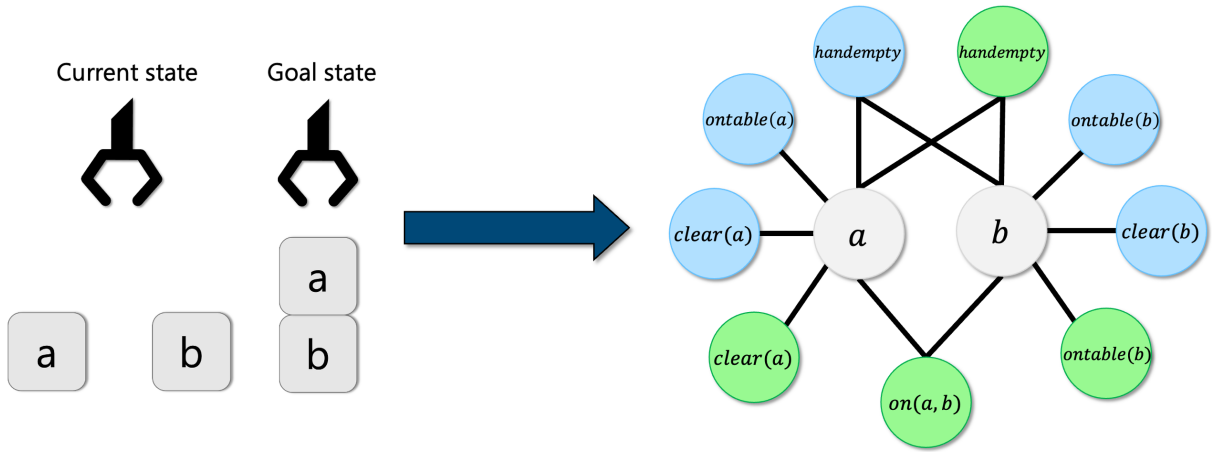

Figure 1: Example of our graph representation for the Blocks World Domain. Nodes for ground atoms true in the current state are blue and nodes for ground atoms in the goal state are green.

expressivity issues and inducing excessively large graphs. Further, the graph representation (or node representation, respectively) should be compatible with any GNN or Transformer architecture, to enable a fair comparison in our experiments. Figure 1 shows an example where a graph is constructed for a pair of current and goal states from the Blocks World domain.

Nodes. First, we add a node for every ground atom true in the current state to the graph. We repeat this for every ground atom true in the goal state. Further, we add a node for every object of the instance. Each node is assigned a feature vector $[i, t, n, a_0, ..., a_m]$ that encodes the corresponding ground atom or object and has a dimensionality of $3 + m$, where m is the maximum arity of the domain's atoms:

1. i: unique ID to prevent symmetries in the graph.
2. t: integer representing the type of the node, corresponding to either object, ground atom, or goal ground atom.
3. n: integer representing an object's name or a ground atom's predicate symbol.
4. a_i: integers representing the names of a ground atom's arguments, entries are set to -1 if atom's arity is smaller than m, all are set to -1 for object nodes.

Edges. The inclusion of nodes for both the ground atoms and objects allows to simply insert edges between ground atoms and their arguments. Nodes corresponding to ground atoms with an arity of 0 are connected to the nodes of all objects since they represent global properties potentially relevant to all objects. Note that the GNN does not need to infer the states' ground atoms through the graph's local structures, as they are directly encoded in the nodes. Thus, the edges serve only to guide the exchange of embeddings during the aggregation step. This also enables us to utilize Transformers by passing the node features as tokens.

Comparison to Similar Graph Structures. Although the relational GNN architecture by Ståhlberg, Bonet, and Geffner does not explicitly construct a graph, its layers pass messages between ground atoms and their arguments in a similar way to how a GNN, applied to our graph representation, would exchange embeddings between neighboring nodes (2022a; 2022b). By explicitly constructing graphs for given classical planning states we disconnect the graph representation from the neural network architecture, and thus we can easily apply different GNN and Transformer architectures.

Our graph representation is also akin to the object-atom binary structure introduced in Horcik and Šír (2024), with a significant difference: we abstain from using edge features due to their lack of support across all GNN architectures and their incompatibility with existing Transformer architectures employing linear attention mechanisms (Rampášek et al. 2022).

We can directly apply our graph representation to domains with atoms that have an arity higher than 2, whereas the graph representation of Rivlin, Hazan, and Karpas would require a pre-processing step that decomposes such atoms to lower arity atoms (2020).

Empirical Evaluation

We now compare the previously introduced GNN and Transformer architectures for general policy learning w.r.t. their performance as general policies.

Setup

We compare the 5 different architectures on the Blocks-clear, Gripper, Visitall, Parking-behind, and Satellite domains.

Datasets. The influence of dataset construction merits its own thorough examination. As a starting point, we adhere to the data generated by Ståhlberg, Bonet, and Geffner (2022a). The training and validation data sets were created by performing random walks from the instances' initial states s_0, and then computing an optimal plan from each visited state s_i.

Domain	Training	Validation	Test
Blocks-clear	$[6-26]$	$[22-31]$	$[27-44]$
Gripper	$[22-48]$	$[52-60]$	$[65-138]$
Visitall	$[133-662]$	$[563-962]$	$[966-1959]$
Parking-behind	$[63-85]$	$[94-95]$	$[94-115]$
Satellite	$[30-134]$	$[142-202]$	$[155-333]$

Table 1: Numbers of nodes of graphs in training, validation, and test sets. Intervals correspond to minimum and maximum number of nodes.

Each data sample then consists of the current state s_i, goal state s_g, and state value $V^*(s_i)$ which computes the minimal cost of reaching s_g from s_i. The test data sets consist of PDDL instance files.

The sizes of the instances used are measured by their number of objects and increase from training over validation to test sets. Table 1 displays the sizes of the resulting graphs. Hence, we train the GNNs and Transformers to imitate the optimal planner on small and medium-sized instances, such that the networks learn to generalize the knowledge, enabling them to solve even larger instances after training.

Training. Our models' parameters are learned by minimizing the standard mean-squared error loss $[V(s) - V^*(s)]^2$ using the Adam optimizer (Kingma and Ba 2014). We employ a learning rate schedule that, starting from a high learning rate of 0.001, decays the learning rate by a factor of 0.5 whenever the validation loss does not decrease for 25 epochs. We stop the training when the validation loss does not decrease for 50 epochs. During testing, a general policy is induced by expanding states s and taking the actions leading to successors s' with the lowest predicted state value $V(s')$. As done in (Ståhlberg, Bonet, and Geffner 2022b), we avoid cycles by keeping a history of previously visited states and disregarding them as possible successors.

Hyperparameters. For each architecture, we conduct a grid search over the most impactful hyperparameters: the number of layers (2, 4), the dimension of hidden layers (32, 64, 128), the dropout percentage (0.1, 0.5), and the number of attention heads (1, 2). We report the hyperparameters used for each domain in the Appendix. Each configuration is trained using three random seeds, and the model with the lowest validation error is returned.

Performance

Table 2 shows the number of solved test instances of the five considered architectures on all domains. Additionally, the number of optimally solved instances, i.e., solved with the shortest possible plan, is displayed in parentheses. In our evaluation, nearly all architectures fully solve the test sets of Blocks-clear and Gripper, with all discovered plans on Gripper being optimal. On Visitall, the Transformer-based Performer and GPS architectures can solve 11 and 10 out of 12 test instances, respectively, whereas the GNN architectures fall behind by only solving up to 8 test instances. Given that the states in Visitall's test set induce large graphs of up to 1959 nodes, we suspect that the Transformer-based architectures' ability to capture long-range dependencies between nodes gives them an advantage over the GNN architectures on this domain. On the Parking-behind domain, GPS solves all 32 test instances, where only one instance was not solved optimally. GIN follows closely by solving 31 instances, of which only 12 were solved optimally. Surprisingly, GCN, GATv2, and Performer can only solve up to 18 instances. On Satellite, GPS performs best by solving 16 out of the 20 test instances. We suspect that this domain is particularly challenging because it has the highest number of atoms, inducing more complex graphs than the other domains.

Conclusion. Regarding the three tested GNN architectures, we do not observe one architecture having a clear advantage. However, it is noteworthy that the simple GCN's performance was better than/competitive with that of the more sophisticated GIN and GATv2 on most domains. The Performer solves only a few test instances on the Parking-behind and Satellite domains, likely due to overfitting, whereas the hybrid GPS architecture performs best overall. This suggests that combining Transformers and GNNs can introduce inductive biases to increase learning efficiency while retaining the ability to capture long-range relationships and unlimited expressiveness.

Related Work

To circumvent the limited expressivity of standard GNNs, Morris et al. introduced so-called k-GNNs, which are based on the k-dimensional WL test, making them strictly more expressive (2019). Ståhlberg, Bonet, and Geffner recently proposed the R-GNN architecture, which computes an efficient approximation of 3-GNNs (2024). The increased expressivity of R-GNN allowed learning of general policies

Domain (#Instances)	GCN	GIN	GATv2	PERFORMER	GPS
Blocks-clear (11)	10 (2)	**11** (1)	**11** (6)	**11** (3)	**11** (6)
Gripper (39)	**39** (39)	**39** (39)	**39** (39)	**39** (39)	**39** (39)
Visitall (12)	8 (1)	7 (1)	8 (0)	**11** (1)	10 (1)
Parking-behind (32)	18 (7)	31 (12)	7 (2)	14 (2)	**32** (31)
Satellite (20)	11 (0)	3 (0)	4 (0)	1 (0)	**16** (0)

Table 2: Total number of solved test instances and number of optimally solved instances (parantheses).

that previous approaches could not compute. Compared to our approach, which relies on off-the-shelf architectures, R-GNN represents an architecture tailored to the planning application.

Another compelling line of work is approaches aiming to learn domain-independent heuristics. The GOOSE architecture (Chen, Thiébaux, and Trevizan 2024) uses graph representations for the lifted representation of planning tasks. This avoids the high computational cost of processing large graphs induced by grounded representations, thus allowing for efficient planning. Recently, Chen, Trevizan, and Thiébaux introduced WL-GOOSE, which does not rely on a GNN but instead computes features from given graphs using the WL test and passes them to classical machine learning techniques to compute a heuristic (2024). The advantage of this approach is that it requires fewer parameters than GNN-based approaches, leading to faster training. While both of these approaches need to be combined with a planner, general policies, as we use them in this paper, can directly be applied to any domain instance once they have been trained.

The application of Transformers to planning tasks is gaining increasing attention. Stein et al. (2024) were the first to introduce a tool for automatically translating PDDL problem specifications into benchmarks for large language model (LLM) planners, enabling them to effectively evaluate the planning capabilities of LLMs. Their experiments showed that the LLMs performed well on some domains and poorly on others. In comparison, our approach trains a small transformer model from scratch for the planning application instead of applying a large Transformer model trained for natural language processing. The works of Lehnert et al. (2024), and Rossetti et al. (2024) investigated training LLM Transformer architectures from scratch to predict plans autoregressively. Although they demonstrated the ability to generate valid plans for many unseen tasks, these approaches require tremendous training data and computational resources. In contrast, our approach builds a plan iteratively by executing predicted actions and observing new states, allowing for more efficient learning.

Conclusion & Future Work

In this paper, we demonstrated that general policies can be learned using standard, rather than custom-built, graph neural networks. We also explored Transformers as an alternative to GNNs, which do not require the construction of a graph representation. In our experiments, both approaches successfully learned general policies, with Transformers outperforming GNNs.

However, none of the approaches we considered could match the performance of the policies derived from the specialized architecture of Ståhlberg, Bonet, and Geffner (2022a). We contend that with further research, standard GNNs and Transformers have the potential to surpass the performance of bespoke GNNs, although significant work remains to achieve this goal.

We have several ideas on potentially beneficial approaches for training GNNs and Transformers in the context of classical planning, which we outline here.

Dataset Generation. The dataset currently in use comprises plans from states reached via random walks starting from the initial state. Instead, we propose to compute the optimal plan from the initial state and then perform random walks from all visited states. This approach will discover additional, so far not included states. Further, we could iteratively repeat this process (planning from the set of discovered states, random walks starting on all states visited when following the plan to discover additional states). Note that this idea bears resemblance to the concept of the training procedure of ASNets (Toyer et al. 2018).

Another option would be to explore even further by following sub-optimal plans (instead of optimal plans only) to accrue additional states that can be used for dataset generation. We particularly believe this approach would facilitate scaling to larger instances, including those so large that traditional planning becomes impractical.

Further Transformers Investigation. The Transformer architectures used in this paper have been relatively general. We plan to enhance our methods by incorporating inductive biases, for example, by utilizing separate layers for different atoms, incorporating information about applicable actions into the input layer, or basing our predictions on the trajectory of visited states rather than solely on the current state. Note that this differs from custom-building GNNs; rather than constructing a graph and customizing its processing, we tailor specific components, as is commonly done in state-of-the-art Transformer research.

Training Procedure. Rather than relying on standard supervised learning, we recognize advantages in employing offline reinforcement learning (RL) techniques, such as Actor-Critics, where a policy and a value function are learned jointly. Additionally, the novel concept of *Decision Transformers* (Chen et al. 2021) presents a promising direction for future exploration since it has been shown to be more data-efficient than other offline RL techniques and to learn optimal policies from sub-optimal data.

Domain-general Policies. We have already conducted initial experiments training a general policy not only to solve all instances within a specific domain but to address multiple domains. Our preliminary experiments suggest that this approach could enhance performance within individual domains, as the agent may generalize across similar tasks and leverage this knowledge for superior performance. However, these results are very preliminary and not yet ready for formal reporting, which is why we will explore them further in future work.

References

Alon, U.; and Yahav, E. 2020. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205*.

Ba, J. L.; Kiros, J. R.; and Hinton, G. E. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bongini, P.; Bianchini, M.; and Scarselli, F. 2021. Molecular generative graph neural networks for drug discovery. *Neurocomputing*, 450: 242–252.

Brody, S.; Alon, U.; and Yahav, E. 2021. How attentive are graph attention networks? *arXiv preprint arXiv:2105.14491*.

Chen, D. Z.; Thiébaux, S.; and Trevizan, F. 2024. Learning Domain-Independent Heuristics for Grounded and Lifted Planning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 20078–20086.

Chen, D. Z.; Trevizan, F.; and Thiébaux, S. 2024. Return to Tradition: Learning Reliable Heuristics with Classical Machine Learning. *arXiv preprint arXiv:2403.16508*.

Chen, L.; Lu, K.; Rajeswaran, A.; Lee, K.; Grover, A.; Laskin, M.; Abbeel, P.; Srinivas, A.; and Mordatch, I. 2021. Decision transformer: Reinforcement learning via sequence modeling. *Advances in neural information processing systems*, 34: 15084–15097.

Choromanski, K.; Likhosherstov, V.; Dohan, D.; Song, X.; Gane, A.; Sarlos, T.; Hawkins, P.; Davis, J.; Mohiuddin, A.; Kaiser, L.; et al. 2020. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*.

Fan, W.; Ma, Y.; Li, Q.; He, Y.; Zhao, E.; Tang, J.; and Yin, D. 2019. Graph neural networks for social recommendation. In *The world wide web conference*, 417–426.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*, 1263–1272. PMLR.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation*, 9(8): 1735–1780.

Horcik, R.; and Šír, G. 2024. Expressiveness of Graph Neural Networks in Planning Domains. In *34th International Conference on Automated Planning and Scheduling*.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kipf, T. N.; and Welling, M. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

Lehnert, L.; Sukhbaatar, S.; Mcvay, P.; Rabbat, M.; and Tian, Y. 2024. Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping. *arXiv preprint arXiv:2402.14083*.

Li, G.; Müller, M.; Ghanem, B.; and Koltun, V. 2021. Training graph neural networks with 1000 layers. In *International conference on machine learning*, 6437–6449. PMLR.

Lin, T.; Wang, Y.; Liu, X.; and Qiu, X. 2022. A survey of transformers. *AI open*, 3: 111–132.

Morris, C.; Ritzert, M.; Fey, M.; Hamilton, W. L.; Lenssen, J. E.; Rattan, G.; and Grohe, M. 2019. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, 4602–4609.

Rampášek, L.; Galkin, M.; Dwivedi, V. P.; Luu, A. T.; Wolf, G.; and Beaini, D. 2022. Recipe for a general, powerful, scalable graph transformer. *Advances in Neural Information Processing Systems*, 35: 14501–14515.

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized planning with deep reinforcement learning. *arXiv preprint arXiv:2005.02305*.

Rossetti, N.; Tummolo, M.; Gerevini, A.; Putelli, L.; Serina, I.; Chiari, M.; and Olivato, M. 2024. Learning General Policies for Planning through GPT Models. In *34th International Conference on Automated Planning and Scheduling*.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2008. The graph neural network model. *IEEE transactions on neural networks*, 20(1): 61–80.

Sharma, V.; Arora, D.; Singla, P.; et al. 2023. SymNet 3.0: exploiting long-range influences in learning generalized neural policies for relational MDPs. In *Uncertainty in Artificial Intelligence*, 1921–1931. PMLR.

Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1): 1929–1958.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022a. Learning general optimal policies with graph neural networks: Expressive power, transparency, and limits. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 32, 629–637.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2022b. Learning generalized policies without supervision using gnns. *arXiv preprint arXiv:2205.06002*.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2024. Learning General Policies for Classical Planning Domains: Getting Beyond C $_2$. *arXiv preprint arXiv:2403.11734*.

Stein, K.; Fišer, D.; Hoffmann, J.; and Koller, A. 2024. AutoPlanBench: Automatically generating benchmarks for LLM planners from PDDL. *arXiv preprint arXiv:2311.09830*.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action schema networks: Generalised policies with deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Wu, Z.; Jain, P.; Wright, M.; Mirhoseini, A.; Gonzalez, J. E.; and Stoica, I. 2021. Representing long-range context for graph neural networks with global attention. *Advances in Neural Information Processing Systems*, 34: 13266–13279.

Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; and Philip, S. Y. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1): 4–24.

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*.

Hyperparameters

Domain	GCN	GIN	GATv2	PERFORMER	GPS
Blocks-clear	$(4, 128, 0.5, \text{None})$	$(2, 64, 0.5, \text{None})$	$(2, 32, 0.5, 1)$	$(2, 32, 0.5, 1)$	$(4, 32, 0.1, 1)$
Gripper	$(4, 64, 0.5, \text{None})$	$(4, 64, 0.1, \text{None})$	$(2, 32, 0.1, 1)$	$(4, 64, 0.5, 1)$	$(4, 32, 0.5, 2)$
Visitall	$(2, 64, 0.1, \text{None})$	$(4, 64, 0.1, \text{None})$	$(2, 128, 0.1, 1)$	$(2, 128, 0.1, 1)$	$(2, 64, 0.1, 2)$
Parking-behind	$(2, 128, 0.5, \text{None})$	$(2, 32, 0.1, \text{None})$	$(2, 32, 0.1, 2)$	$(2, 32, 0.5, 2)$	$(4, 64, 0.1, 2)$
Satellite	$(2, 128, 0.5, \text{None})$	$(4, 32, 0.1, \text{None})$	$(2, 64, 0.1, 2)$	$(4, 64, 0.5, 1)$	$(2, 64, 0.5, 1)$

Table 2: Hyperparameters of the best-performing policies on each domain. Tuples consist of the number of layers, size of hidden layers, dropout percentage, and number of attention heads.