Adaptive Planner Scheduling with Graph Neural Networks

Tengfei Ma,[1] Patrick Ferber,[2] Siyu Huo,[1] Jie Chen,[1*] Michael Katz[1*]
[1]IBM Research [2]University of Basel
Tengfei.Ma1@ibm.com, patrick.ferber@unibas.ch
{siyu.huo, chenjie, Michael.Katz1}@us.ibm.com

Abstract

Automated planning is one of the foundational areas of AI. Since a single planner unlikely works well for all tasks and domains, portfolio-based techniques become increasingly popular recently. In particular, deep learning emerges as a promising methodology for online planner selection. Owing to the recent development of structural graph representations of planning tasks, we propose a graph neural network (GNN) approach to selecting candidate planners. GNNs are advantageous over a straightforward alternative, the convolutional neural networks, in that they are invariant to node permutations and that they incorporate node labels for better inference.

Additionally, for cost-optimal planning, we propose a two-stage adaptive scheduling method to further improve the likelihood that a given task is solved in time. The scheduler may switch at halftime to a different planner, conditioned on the observed performance of the first one. Experimental results validate the effectiveness of the proposed method against strong baselines, both deep learning and non-deep learning based.

Introduction

Automated planning is one of the foundational areas of Artificial Intelligence research (Russell and Norvig 1995). Planning is concerned with devising goal-oriented policies executed by agents in large-scale state models. Since planning is intractable in general (Chapman 1987) and even classical planning is PSPACE-complete (Bylander 1994), a single algorithm unlikely works well for all problem domains. Hence, surging interest exists in developing *portfolio*-based approaches (Seipp et al. 2012; Vallati 2012; Cenamor, de la Rosa, and Fernández 2013; Seipp et al. 2015), which, for a set of planners, compute an offline schedule or an online decision regarding which planner to invoke per planning task. While offline portfolio approaches focus on finding a single invocation schedule that is expected to work well across all planning tasks, online methods *learn* to choose the right planner for each given task. Most online methods use handcrafted features for learning (Cenamor, de la Rosa, and Fernández 2016).

Recent advances in deep learning has stimulated increasing interest in the use of deep neural networks for online portfolio selection, alleviating the effort of handcrafting features. A deep neural network may be considered a machinery for learning feature representations of an input object without the tedious effort of feature engineering. For example, convolutional neural networks (CNN) take the raw pixels as input and learn the feature representation of an image through layers of convolutional transformations and abstractions, which result in a feature vector that captures the most important characteristics of the image (Krizhevsky, Sutskever, and Hinton 2012). A successful example in the context of planning is *Delfi* (Katz et al. 2018), which treats a planning task as an image and applies CNN to predict the probability that a certain planner solves the task within the time limit. Delfi won the first place in the Optimal Track of the International Planning Competition (IPC) 2018.

As planning tasks admit state transition graphs that are often too big to fit in any conceivable size memory, several other graphs were developed to encode the structural information. Two prominent examples are the problem description graph (Pochter, Zohar, and Rosenschein 2011), for a *grounded* task representation, and the abstract structure graph (Sievers et al. 2017), for a *lifted* representation. Both graphs are used in classical planning for computing structural symmetries (Sievers et al. 2017; Domshlak, Katz, and Shleyfman 2012). The most important use of structural symmetries is search space pruning, considerably improving the state-of-the-art. The lifted structural symmetries are also found useful for faster grounding and mutex generation (Röger, Sievers, and Katz 2018).

Owing to the development of these structural graphs, we propose a graph neural network (GNN) approach to learn the feature representation of a planning task. A proliferation of GNN architectures emerged recently (Bruna et al. 2014; Defferrard, Bresson, and Vandergheynst 2016; Li et al. 2016; Kipf and Welling 2017; Hamilton, Ying, and Leskovec 2017; Gilmer et al. 2017). They have two advantages over CNNs for graph inputs. First, GNNs address the limitation of images that are not invariant to node permutation. Second, GNNs incorporate node and edge attributes that produce a richer representation than does the image surrogate of the graph adjacency matrix alone. In this work, we explore the use of two representative GNNs—graph convolutional networks (Kipf and Welling 2017) and gated graph neural networks (Li et al. 2016). The former is *convolutional*, which

*Corresponding authors

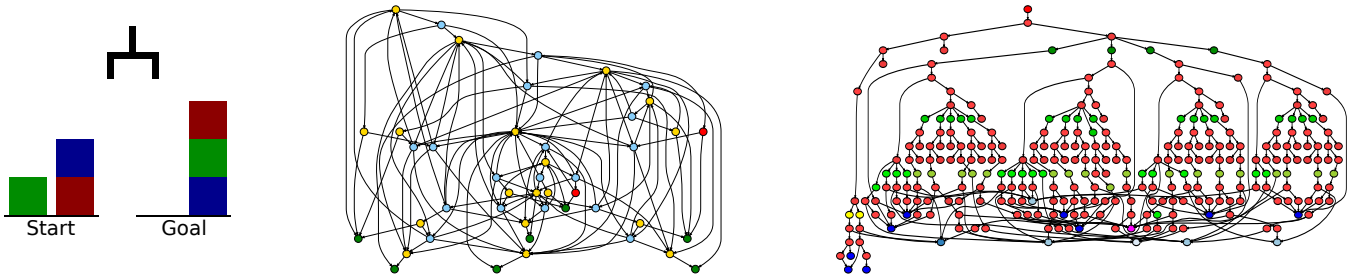

Figure 1: An example planning task (left) with its grounded graph representation (middle) and the lifted one (right). The task, *blocksworld*, uses a gripper to rearrange a set of blocks from an initial configuration to the goal configuration. The coloring of the graph nodes indicate node labels. For more details, see the section "Graph Construction."

extends convolution filters for image patches to graph neighborhoods. The latter is *recurrent*, which treats the representation of a graph node as a dynamical-system state that can be recurrently updated through neighborhood aggregation. A key difference between the two is whether network parameters are shared cross layers, similar to that between a CNN and a recurrent neural network.

With the use of GNNs, we in addition consider the problem of cost-optimal planning, whose goal is to solve as many tasks as possible, each given a time limit, with cost-optimal planners. We propose a two-stage adaptive scheduling approach that enhances the likelihood of task solving within the time limit, over the usual approach of using a single planner for the whole time span. The proposal is based on the observation that if a planner solves a given task in time, its execution is often rather quick. Hence, we divide the time interval in two equal halves and determine at the midpoint whether to change the planner, should it be still running at that time. Experimental results show that the proposed adaptive scheduling consistently increases the number of solved tasks, compared with the use of a single planner.

Planning and Planner Selection

Planning algorithms generally perform reachability analysis in large-scale state models, which are implicitly described in a concise manner via some intuitive declarative language (Russell and Norvig 1995; Ghallab, Nau, and Traverso 2004). One of the most popular approaches to classical planning in general and to cost-optimal planning in particular is state-space heuristic search. The key to this approach is to automatically derive an informative *heuristic function* h from states to scalars, estimating the cost of the cheapest path from each state to its nearest goal state. The search algorithms then use these heuristics as search guides. If h is *admissible* (that is, it never overestimates the true cost of reaching a goal state), then search algorithms such as A^* (Hart, Nilsson, and Raphael 1968) are guaranteed to provide a cost-optimal plan.

Over the years, many admissible heuristics were developed to capture various aspects of planning tasks; see, e.g., (Edelkamp 2001; Helmert et al. 2014; Helmert and Domshlak 2009; Haslum, Bonet, and Geffner 2005). Further, search pruning techniques (Wehrle and Helmert 2014; Shleyfman et al. 2015) were developed to reduce the search effort, producing sophisticated search algorithms (Edelkamp, Kissmann, and Torralba 2015; Gnad and Hoffmann 2015). All these techniques can be used interchangeably. Moreover, most of them are highly parameterized, allowing to construct many possible cost-optimal planners.

Because of the intractability of planning, a single planner unlikely works well across all possible domains. Some planners excel on certain tasks, while some on others. However, given a task, it is unclear whether a particular planner works well on the task without actually running it. With a large number of planners, especially in resource constrained situations, it is infeasible to try all of them until a good one is found. Hence, it is desirable to predict the performance of the planners on the task and select the best performing one.

One approach of making such a selection is machine learning based on handcrafted features (Cenamor, de la Rosa, and Fernández 2016), which include the number of actions, objects, predicates in the planning task, and the structure of the task's causal graph. This approach worked reasonably well in practice for non-optimal planning, winning the first place in IPC 2014. However, even the updated version, whose portfolio included top performing planners at IPC 2018 (e.g., the one presented by (Katz and Hoffmann 2014)), performed poorly in this competition, ranked only 12th.

To the best of our knowledge, the only existing cost-optimal planner that is based on online planner selection is Delfi (Katz et al. 2018). Delfi treats planning tasks as images and selects a planner through training a CNN to predict which planner solves the given task in time. Specifically, a planning task is formulated as a certain graph, whose adjacency matrix is converted to a black-and-white image, which in turn is resized to 128×128, becoming grayscale. A CNN is used to perform image classification.

Two versions of Delfi were submitted to IPC 2018, differing in the way the planning task is represented. Delfi1 works on the lifted representation of the task, based on PDDL's abstract structure graph (ASG) (Sievers et al. 2017); whereas Delfi2 works on the grounded representation, based on the problem description graph (PDG) (Pochter, Zohar, and Rosenschein 2011). Both graphs have additional features (e.g., node labels), which however are ignored when being converted to an image.

Graph Construction

In this work, we reuse the graphs built by Delfi, incorporating additionally node labels. Figure 1 shows a classical planning example, *blocksworld*, with its two graphs. For illustrative purpose only the three-block version is shown; the problem is NP-hard (Gupta and Nau 1992).

For the construction of ASGs, planning tasks correspond to abstract structures, which include actions, axioms, the initial state, and the goal. Nodes are labeled by their types; e.g., action, axiom, predicate, and object. Edges encode the inclusion hierarchy of the abstract structures.

For the construction of PDGs, there are nodes for all task facts, variables, actions, conditional effects, and axioms. Each node type has a separate label, further divided by the action cost in the case of action nodes, and whether the fact is initially true and required in the goal, in the case of facts. Edges connect facts to their variables, actions to their conditional effects, conditional effects to their effect facts, condition facts to their conditional effects, precondition facts to their actions and axioms, and actions and axioms to their unconditional effect facts.

Planner Selection with Graph Neural Nets

Given a portfolio of planners, we model the selection problem as predicting the probability that each planner fails to solve a given task in time. Then, the planner with the lowest probability is selected for execution. Denote by G a task, $\mathcal{G} = \{G\}$ the space of tasks, and D the size of the portfolio. Parameterized by $\theta \in \Theta$, the problem amounts to learning a D-variate function $f : \mathcal{G} \times \Theta \to [0, 1]^D$ that computes the probabilities for all planners in the portfolio.

Let $S = \{(G, y)\}$ be the set of task-label pairs for training, where $y \in \{0, 1\}^D$ is the ground-truth labeling vector, whose element y_j denotes the fact whether planner j fails to solve the task in time:

$$y_j = \begin{cases} 0, & \text{if execution time of } j \text{ does not exceed } T, \\ 1, & \text{otherwise.} \end{cases} \quad (1)$$

Then, the learning amounts to finding the optimal parameter θ that minimizes the cross-entropy loss function

$$L(\theta) = -\sum_{(G,y)\in S,\ j=1}^{D} y_j \log f_j(G, \theta) + (1 - y_j) \log(1 - f_j(G, \theta)).$$

Learning Graph Representations

Since a planning task is formulated as a graph, we write $G = (V, E)$, where V is the node set and E is the edge set. For calculus, the function f requires a vectorial representation h_G of the graph G. Deep learning uses deep neural networks to compute this vector, rather than handcrafting. In our work, the design of f consists of three steps:

1. Parameterize the vectorial representation h_v for all nodes $v \in V$.
2. Form the graph representation as a weighted combination of h_v:
$$h_G = \sum_{v \in V} \alpha_v h_v, \quad (2)$$
where α_v denotes the attention weight, scoring in a sense the importance of the contribution of each node to the overall representation of the graph. These weights depend on the node representations h_v.
3. Parameterize f as a feedforward neural network, taking h_G as input:
$$f(G, \theta) = \text{sigmoid}(W_{\text{logit}}^{\top} h_G). \quad (3)$$
The parameter set θ thus includes the parameter matrix W_{logit} and all the parameters in h_v and α_v. Note the use of the sigmoid activation function, such that each element of the output vector is an independent probability. This is in contrast to the multiclass classification scenario, where the softmax activation function is used such that the output is a probability distribution.

Graph neural networks differ in the parameterizations of the node representation h_v and possibly the attention weight α_v. In this work, we consider two types of GNNs: graph convolutional networks and gated graph neural networks.

Graph Convolutional Networks (GCN)

GCN (Kipf and Welling 2017) generalizes the convolution filters for image patches to graph neighborhoods. Whereas an image patch contains a fixed number of pixels, which may be handled by a fixed-size filter, the size of a node neighborhood varies. Hence, the convolution filter for graphs uses a parameter matrix to transform each node representation computed from the past layer, and linearly combines the transformed representations with certain weights based on the graph adjacency matrix.

Specifically, let t be the layer index, orient the node representations $h_v^{(t)}$ as row vectors, and stack them to form the matrix $H^{(t)}$. A layer of GCN is defined as

$$H^{(t+1)} = \sigma(\widehat{A} H^{(t)} W^{(t)}).$$

Here, $W^{(t)}$ is the parameter matrix, $\widehat{A}$ is a normalization of the adjacency matrix A, and σ is an activation function (e.g., ReLU). Clearly, the combination weights are nothing but the rows of $\widehat{A}$. The normalization is defined as

$$\widehat{A} = \widetilde{D}^{-\frac{1}{2}} \widetilde{A} \widetilde{D}^{-\frac{1}{2}},\ \widetilde{A} = A + I,\ \widetilde{D} = \text{diag}(d_i),\ d_i = \sum_k \widetilde{A}_{ik}.$$

Using an initial feature matrix X (which, for example, can be defined based on node labels) as the input $H^{(0)}$, a few graph convolutional layers produce a sophisticated representation matrix $H^{(T)}$, whose rows are treated as the final node representations h_v. Orient them back as column vectors; then, the attention weights are defined by using a feedforward layer

$$\alpha_v = \text{sigmoid}(w_{\text{gate}}^{\top} [h_v^{(T)}; h_v^{(0)}]), \quad (4)$$

where w_{gate} is a parameter vector. Hence, the overall parameter set for the model f by using the GCN architecture is

$$\theta = \{W_{\text{logit}}, w_{\text{gate}}, W^{(0)}, W^{(1)}, \ldots, W^{(T-1)}\}.$$

Gated Graph Neural Networks (GG-NN)

The architecture of GG-NN (Li et al. 2016) is recurrent rather than convolutional. In this architecture, the node representation is treated as the state of a dynamical system and the gated recurrent unit (GRU) is used to update the state upon a new input message:

$$h_v^{(t+1)} = \mathrm{GRU}(h_v^{(t)}, m_v^{(t+1)}).$$

The message $m_v^{(t+1)}$ is an aggregation of the transformed states for all the neighboring nodes of v. Specifically, denote by $\mathrm{in}(v)$ and $\mathrm{out}(v)$ the sets of in-neighbors and out-neighbors of v, respectively, and let W_{in} and W_{out} be the corresponding parameter matrices shared by all graph nodes and recurrent steps. The message is then defined as

$$m_v^{(t+1)} = \sum_{u \in \mathrm{in}(v)} W_{\text{in}}^\top h_u^{(t)} + \sum_{u' \in \mathrm{out}(v)} W_{\text{out}}^\top h_{u'}^{(t)}.$$

Similar to GCN, GG-NN may use the initial features for each node as the input $h_v^{(0)}$ and produce $h_v^{(T)}$ as the final node representation h_v, through T recurrent steps. Thus, the attention weights α_v may be computed in the same manner as (4). Therefore, the overall parameter set for the model f by using GG-NN is

$$\theta = \{W_{\text{logit}}, w_{\text{gate}}, W_{\text{in}}, W_{\text{out}}, \text{ and parameters of GRU}\}.$$

Discussions

The original GCN architecture proposed by (Kipf and Welling 2017) defines representations for only the nodes but not the overall graph. Here, we use the unified framework (2) to define the graph representation, which essentially is a global pooling of the node representations with weights. The definition of the weights (4) is inspired by the attention mechanism (Bahdanau, Cho, and Bengio 2015; Vaswani et al. 2017) popularly used nowadays for sequence models. This definition is also simpler than that of the original GG-NN architecture (Li et al. 2016).

One variant of the attention weights in (4) is that the parameter vector w_{gate} may not be shared by the planners. In other words, for each planner j, a separate parameter $w_{\text{gate},j}$ is used to compute the attention weights $\alpha_{v,j}$ and subsequently the graph representation $h_{G,j}$ and the predictive model $f_j(G, \theta) = \mathrm{sigmoid}(W_{\text{logit},j}^\top h_{G,j})$. In this manner, node representations are still shared by different planners, but not the graph one. Such an approach may be used to increase the capacity of the model f, which sometimes works better than using a single w_{gate}.

The original GG-NN architecture incorporates also edge labels by using a different pair of parameter matrices $W_{\text{in},e}, , W_{\text{out},e}$ for each edge type e. This idea may easily be used to extend GCN: Replace the parameter matrix $W^{(t)}$ by $W_e^{(t)}$ for different types of edges when performing convolution. In our context, however, no edge labels exist and hence in the preceding subsection we present only the simplest version of GG-NN.

Adaptive Scheduling

When the goal is to solve a given task within a time limit T (but not how quickly it is solved), one may try a second planner if she "senses" that the selected one unlikely completes in time. Such a scenario may occur when the model f described in the preceding section is insufficiently accurate. Then, we offer a second chance to reevaluate the probability that the currently invoked planner cannot complete within the rest of time allowance, versus the probability that a separate planner fails to solve the task in this time span. If the former probability is lower, we have no choice but to continue the execution of the current planner; otherwise, we switch to the latter one. The intuition comes from the observation that if a planner solves a task in time, often it completes rather quickly. Hence, the remaining time may be sufficient for a second planner, should its failure/success be accurately predicted.

To formalize this idea, we set the time of reevaluation at $T/2$. We learn a separate model g that predicts the probabilities that each planner fails to solve the task before timeout, conditioned on the fact that the current planner p needs more time than $T/2$. We write the function $g : \mathcal{G} \times [D] \times \Theta \to [0, 1]^D$, where $[D]$ denotes the set of integers from 1 to D, and parameterize it as

$$g(G, p, \theta_g) = \mathrm{sigmoid}(W_{\text{logit}}^\top h_G + W_{\text{fail}}^\top e_p),$$

where $e_p \in \{0, 1\}^D$ is the one-hot vector whose pth element is 1 and 0 for others.

Compare this model with f in (3). First, we introduce an additional parameter matrix W_{fail} to capture the conditional fact. Second, the graph representation h_G reuses that in f. In other words, the two models f and g share the same graph representation but differ in the final prediction layer.

Training Set

One must construct a training set S_g for learning the model g. One approach is to reuse all the graphs in the training of the model f. For every such graph G, we pick the planners p whose execution time exceeds $T/2$ and form the pairs (G, p). For each such pair, we further construct the ground-truth labeling vector $z \in \{0, 1\}^D$ to form the training set $S_g = \{(G, p, z)\}$.

The construction of the labeling vector follows this rationale: For any planner j different from p, because the time allowance is only $T/2$, straightforwardly $z_j = 0$ if j solves the task in time less than $T/2$; otherwise, $z_j = 1$. On the other hand, when j coincides with p, the continued execution of j gives a total time allowance T. Hence, $z_j = 0$ if j solves the task in time less than T and otherwise $z_j = 1$. To summarize,

$$z_j = \begin{cases} 0, & \text{if } j = p \text{ and execution time of } j \text{ is} \le T, \\ 1, & \text{if } j = p \text{ and execution time of } j \text{ is} > T, \\ 0, & \text{if } j \ne p \text{ and execution time of } j \text{ is} \le T/2, \\ 1, & \text{if } j \ne p \text{ and execution time of } j \text{ is} > T/2. \end{cases}$$

The size of the training set S_g constructed in this manner may be smaller, but more likely greater, than that of S, depending on the performance of the planners on each task. In

Table 1: Summary of data set.

	Grounded	Lifted
# Graphs, train/val/test	2,008 / 286 / 145	
# Nodes, min/max/mean/median	6 / 47,554 / 2,056 / 580	51 / 238,909 / 3,001 / 1,294
Edge-Node ratio, min/max/mean/median	0.88 / 10.65 / 3.54 / 3.28	1.04 / 1.82 / 1.49 / 1.47
# Node labels	6	15

practice, we find that $|S_g|$ is a few times of $|S|$. Such a size does not incur substantially more expense for training.

With the training set defined, the loss function is

$$L_g(\theta_g) = -\sum_{(G,p,z)\in S_g,}\sum_{j=1}^{D} z_j \log g_j(G,p,\theta_g) + (1-z_j)\log(1-g_j(G,p,\theta_g)).$$

Two-Stage Scheduling

We now have two models f and g. In test time, we first evaluate f and select the planner p with the lowest predicted probability for execution. If it solves the task before halftime $T/2$, we are done. Otherwise, at halftime, we evaluate g and obtain a planner j with the lowest predicted probability. If $j = p$, we do nothing but to let the planner continue the job. Otherwise, we terminate p and invoke j, expecting a successful execution.

Experiments

Data Set and Portfolio

To evaluate the effectiveness of the proposals, we prepare a data set composed of historical and the most recent IPC tasks.[1] Specifically, the historical IPC tasks form the training and validation sets, whereas those of the year 2018 form the test set. A small amount of tasks are ignored, the reason of which is explained in the next paragraph. When performing the training/validation split, tasks in the same domain are not separated in two sets. Hence, we randomly select a few domains to form the validation set, such that its size is approximately 10% of that of the training set. Summary of the resulting data set is given in Table 1.

We use the same portfolio as did (Katz et al. 2018), which makes it convenient to compare with the image-based CNN approach. The tasks unsolvable by any of the planners in the portfolio within the time limit $T = 1800$s are ignored in the construction of the data set. In particular, some of these tasks occur in IPC 2018.

Each task in the data set has two graph versions, grounded and lifted, as explained earlier. For each version, the size of the graphs has a very skewed distribution (whereas the sparsity distribution is relatively flat), with some graphs being particularly large. The table suggests that the lifted version is generally larger than the grounded one. However, because the distribution is rather skewed, we plot additionally in Figure 2 the individual graph sizes to offer a complementary view. In this figure, the tasks are sorted according to the size of the grounded graph. Then, for each task, the sizes of the graphs are normalized. The blue curve to the far right end indicates that the lifted version is much smaller for the tasks with the largest grounded graphs.

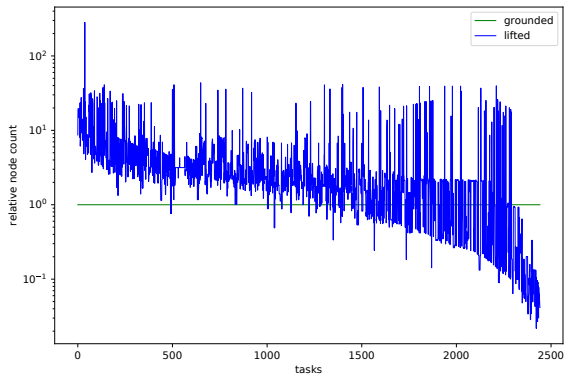

Figure 2: Node counts of the lifted graphs relative to that of the grounded ones.

Training Details

For the training of the neural networks, we use the Adam optimizer (Kingma and Ba 2015) with learning rate 0.001. We slightly tune other hyperparameters: the number of layers in GCN and steps in GG-NN is selected from $\{2, 4, 6\}$ and the dimension of the node representations $h_v^{(t)}$ is chosen from $\{100, 150, 200\}$. Meanwhile, we also tune the architecture through experimenting with a variant of the attention weights: replace the parameter vector w_{gate} in (4) by using D separate copies, one for each planner (see discussions in an earlier section). Because of memory limitation, graphs with more than $100,000$ nodes are ignored for training.

Effectiveness of Graph Neural Networks

We compare the performance of several types of methods, as summarized in Table 2. In addition to the golden criterion, the percentage of solved tasks, the column "eval. time" is the time needed for selecting a planner, which includes the time to convert a planning task to a graph, that to convert the graph file format, and that to evaluate the neural network model. This time is overhead and hence for any reasonable method, it should not occupy a substantial portion of the overall time allowance $T = 1800$s.

The first two methods are weak baselines. As the name suggests, "random planner" uniformly randomly selects a planner, whereas "single planner for all tasks" uses the one

[1] https://ipc2018.bitbucket.io/

Table 2: Percentage of solved tasks in the test set and average evaluation time of the method.

Method	Solved	Eval. Time
Random planner	60.6%	0
Single planner for all tasks	64.8%	0
Complementary2	84.8%	0
Planning-PDBs	82.0%	0
Symbolic-bidirectional	80.0%	0
Image based, CNN, grounded	73.1%	11.00s
Image based, CNN, lifted	86.9%	3.16s
Graph based, GCN, grounded	80.7%	23.15s
Graph based, GCN, lifted	**87.6%**	9.41s
Graph based, GG-NN, grounded	77.9%	14.53s
Graph based, GG-NN, lifted	81.4%	11.44s

that solves the most number of tasks in the training set. Neither method takes time to perform selection. The percentage of solved tasks for the random method is the expected value.

The next three are state-of-the-art planing systems, not based on deep learning. These systems are the top performers of IPC 2018, second only to Delfi. Both Complementary2 and Planning-PDBs perform A^* search with heuristic guidances based on sophisticated methods for pattern databases creation (Franco, Lelis, and Barley 2018; Martinez et al. 2018). Symbolic-bidirectional, on the other hand, is a baseline entered into the competition by the organizers. As the name suggests, it runs a bidirectional symbolic search, with no heuristic guidance (Torralba et al. 2017). These three methods are not portfolio based and hence no time is needed for planner selection; however, there are rather competitive for cost-optimal planning.

Followed are deep learning methods: the two CNNs come from Delfi and the GCNs and GG-NNs are our proposal. For each network architecture, the performance of using grounded/lifted inputs are separately reported.

The results in Table 2 show that the planners in the portfolio have good qualities: with close to twenty planners, even a random choice can solve more than 60% of the tasks. Meanwhile, the state-of-the-art methods, even though not based on deep learning, set a high bar. Delfi, based on CNN, yields a better result for the lifted graphs, but not so much for the grounded ones. Further, one of our GNNs (GCN on lifted graphs) achieves the best performance, whereas the other three GNNs outperform CNN on grounded graphs.

Using either CNNs or GNNs, it appears consistently that the lifted graphs yield better results than do the grounded ones. Moreover, for the same neural network architecture, they also require less evaluation time. One reason is that lifted graphs are less expensive to construct, albeit being larger on average. More discussions in this regard are given in a later subsection.

We confirm from the table that for all deep learning methods, the time for selecting a planner is negligble compared with the allowed time for executing the planner.

We further plot a curve for each method regarding the number of solved tasks as time evolves; see Figure 3. To avoid cluttering, only a few are shown. The *oracle* curve on the top is the ceiling performance obtained by always selecting the fastest planner for each task. For all curves, one sees a trend of diminishing increase, indicating that most of the tasks are solved rather quickly.

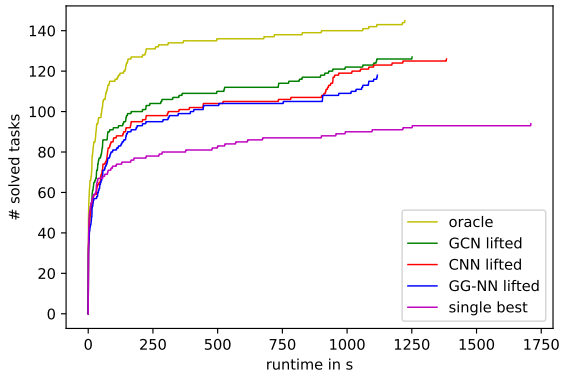

Figure 3: Number of solved tasks with respect to time. "Oracle" is the ceiling performance obtained by always selecting the fastest planner for each task. Curves for the grounded graphs are not shown to avoid cluttering.

Effectiveness of Adaptive Scheduling

The oracle curve in Figure 3 confirms our intuition for adaptive scheduling: because often a planner solves a task rather quickly, allowing halftime for a second chance suffices for an alternative planner to complete the task. Hence, we compare the performance of the single selection with that of adaptive scheduling.

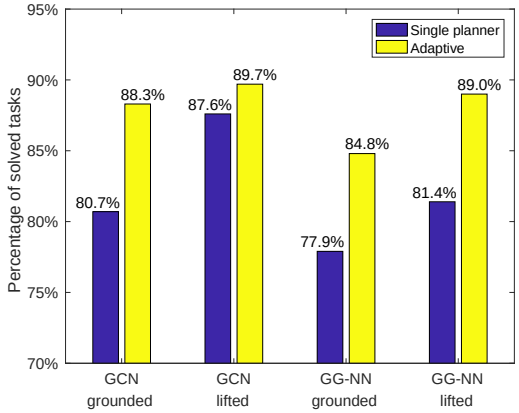

Figure 4: Percentage of solved tasks.

Figure 4 reports the results. One sees that adaptive scheduling consistently increases the percentage of solved tasks on both GNN architectures and both graph versions. It pushes the best performance 87.6% seen in Table 2 to the new best 89.7%.

For a finer grained analysis, we focus on the tasks for which adaptive scheduling changes the planner at halftime and divide them in four groups, according to whether the

original/new planner solves the task. The gain of adaptive scheduling must come from a higher number of tasks solvable by the new but not the original planner, than that solvable by the original but not the new one. Table 3 lists these numbers. Adaptive scheduling changes the fate of quite a few tasks and indeed in more cases than other, the new planner successfully solve the task but the original one cannot.

Table 3: Fine grained analysis of adaptive scheduling. In each subtable, tasks for which adaptive scheduling changes the planner in the second half are divided in four groups.

GCN, grounded

Original planner	New planner	# Tasks
solvable	solvable	8
solvable	unsolvable	2
unsolvable	solvable	13
unsolvable	unsolvable	11

GCN, lifted

Original planner	New planner	# Tasks
solvable	solvable	1
solvable	unsolvable	7
unsolvable	solvable	10
unsolvable	unsolvable	7

GG-NN, grounded

Original planner	New planner	# Tasks
solvable	solvable	5
solvable	unsolvable	6
unsolvable	solvable	16
unsolvable	unsolvable	4

GG-NN, lifted

Original planner	New planner	# Tasks
solvable	solvable	8
solvable	unsolvable	5
unsolvable	solvable	16
unsolvable	unsolvable	7

Discussions on Grounded versus Lifted Graphs

Throughout the paper, we referred to the graphs built from ASGs (that represent PDDL tasks) as the lifted version, while those from PDGs the grounded version, which is customary in the planning community. Experimental results in the prior subsections suggest that the former version yields consistently better results than the latter. It is, however, important to understand the distinctions between these two versions, since tradeoffs exist in construction and characteristics.

While in most cases, PDDL describes tasks in the lifted manner through parameterizing actions and predicates, occasionally the PDDL may already be pre-grounded, with all actions and predicate parameters instantiated. In such cases, the lifted version might be comparable to in size, or even larger than, the grounded one. One additional reason is the existence of static predicates in the lifted version but not the grounded one. These predicates are often used to encode roadmaps or arithmetics and they can become quite large. However, they do not serve any purpose on the grounded level, since the truth values of these predicates do not change. Thus, on the grounded level, static predicates are safely removed.

Beyond the difference in sizes, as the grounded version encodes grounded actions, which are connected to facts in their preconditions and effects, such causal relations between facts are encoded explicitly in the grounded version. Further, the grounded version is more structured. In the case of planning tasks without conditional effects, grounded versions are bipartite graphs, where all edges are from fact to non-fact nodes, or in the reverse direction. In the case of conditional effects, these versions are tripartite graphs, where nodes are partitioned into facts, effect condition nodes, and the rest. Lifted versions have a different structure. Since nodes encode abstract structures and edges correspond to the hierarchy among the abstract structures, lifted versions are directed acyclic graphs. These structural differences may attribute to the predictive power of the learning models.

Conclusion

Graphs encode the structural information of a planning task. In this work, we have proposed a graph neural network approach for online planner selection. This approach outperforms Delfi, the winner of the Optimal Track of IPC 2018, which treats a planning task as an image and applies convolutional neural networks for selecting candidate planners. The appealing results are owing to the representation power of GNNs that address the lack of permutation invariance and the negligence of node-labeling information in CNNs.

We have also proposed an adaptive scheduling approach to compensate the inaccuracy of a single predictive model, through offering a chance for switching planners at halftime, conditioned on the performance of the previously selected one. Such an adaptive approach consistently increases the number of solved tasks, leading to a new state-of-the-art.

Overall, it appears that the lifted graph version is advantageous over the grounded one, because of consistently better performance. However, on average they are larger in size and some are particularly enormous. Moreover, the size distribution is highly skewed in both versions. These factors impose substantial challenges for the batch training of the neural networks. As a compromise, we cut some overly large graphs. An avenue of future research is to investigate more efficient and scalable training approaches.

Another line of future work is to use GNNs and to extend the idea of adaptive scheduling for other types of problems, including satisficing planning, agile planning, and cost-bounded planning.

References

Bahdanau, D.; Cho, K.; and Bengio, Y. 2015. Neural machine translation by jointly learning to align and translate. In *ICLR*.

Bruna, J.; Zaremba, W.; Szlam, A.; and LeCun, Y. 2014. Spectral networks and locally connected networks on graphs. In *ICLR*.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *AIJ* 69(1–2):165–204.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2013. Learning predictive models to configure planning portfolios. In *ICAPS 2013 Workshop on Planning and Learning*, 14–22.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2016. The IBaCoP planning system: Instance-based configured portfolios. *JAIR* 56:657–691.

Chapman, D. 1987. Planning for conjunctive goals. *AIJ* 32:333–377.

Defferrard, M.; Bresson, X.; and Vandergheynst, P. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced symmetry breaking in cost-optimal planning as forward search. In *Proc. ICAPS 2012*, 343–347.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2015. BDDs strike back (in AI planning). In *Proc. AAAI 2015*, 4320–4321.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. ECP 2001*, 84–90.

Franco, S.; Lelis, L. H. S.; and Barley, M. 2018. The complementary2 planner in the ipc 2018. In *IPC-9 planner abstracts*, 30–34.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *ICML*.

Gnad, D., and Hoffmann, J. 2015. Beating LM-cut with h^{max} (sometimes): Fork-decoupled state space search. In *Proc. ICAPS 2015*, 88–96.

Gupta, N., and Nau, D. S. 1992. On the complexity of blocks-world planning. *AIJ* 56(2–3):223–254.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *NIPS*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proc. AAAI 2005*, 1163–1168.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, 162–169.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM* 61(3):16:1–63.

Katz, M., and Hoffmann, J. 2014. Mercury planner: Pushing the limits of partial delete relaxation. In *IPC-8 planner abstracts*, 43–47.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online planner selection for cost-optimal planning. In *IPC-9 planner abstracts*, 55–62.

Kingma, D. P., and Ba, J. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS*.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. 2016. Gated graph sequence neural networks. In *ICLR*.

Martinez, M.; Moraru, I.; Edelkamp, S.; and Franco, S. 2018. Planning-PDBs planner in the ipc 2018. In *IPC-9 planner abstracts*, 63–66.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting problem symmetries in state-based planners. In *Proc. AAAI 2011*, 1004–1009.

Röger, G.; Sievers, S.; and Katz, M. 2018. Symmetry-based task reduction for relaxed reachability analysis. In *Proc. ICAPS 2018*, 208–217.

Russell, S., and Norvig, P. 1995. *Artificial Intelligence — A Modern Approach*. Prentice Hall.

Seipp, J.; Braun, M.; Garimort, J.; and Helmert, M. 2012. Learning portfolios of automatically tuned planners. In *Proc. ICAPS 2012*, 368–372.

Seipp, J.; Sievers, S.; Helmert, M.; and Hutter, F. 2015. Automatic configuration of sequential planning portfolios. In *Proc. AAAI 2015*, 3364–3370.

Shleyfman, A.; Katz, M.; Helmert, M.; Sievers, S.; and Wehrle, M. 2015. Heuristics and symmetries in classical planning. In *Proc. AAAI 2015*, 3371–3377.

Sievers, S.; Röger, G.; Wehrle, M.; and Katz, M. 2017. Structural symmetries of the lifted representation of classical planning tasks. In *ICAPS 2017 Workshop on Heuristics and Search for Domain-independent Planning*, 67–74.

Torralba, A.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient symbolic search for cost-optimal planning. *AIJ* 242:52–79.

Vallati, M. 2012. A guide to portfolio-based planning. In *Proc. MIWAI 2012*, 57–68.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is all you need. In *NIPS*.

Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In *Proc. ICAPS 2014*, 323–331.