

# Online Planner Selection with Graph Neural Networks and Adaptive Scheduling

Tengfei Ma,<sup>1</sup> Patrick Ferber,<sup>3,4</sup> Siyu Huo,<sup>1</sup> Jie Chen,<sup>1,2\*</sup> Michael Katz<sup>1\*</sup>

<sup>1</sup>IBM Research, <sup>2</sup>MIT-IBM Watson AI Lab, <sup>3</sup>University of Basel, <sup>4</sup>Saarland University  
 {Tengfei.Ma1, siyu.huo, Michael.Katz1}@ibm.com, patrick.ferber@unibas.ch, chenjie@us.ibm.com

## Abstract

Automated planning is one of the foundational areas of AI. Since no single planner can work well for all tasks and domains, portfolio-based techniques have become increasingly popular in recent years. 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.

The code is available at [https://github.com/matenure/GNN\\_planner](https://github.com/matenure/GNN_planner).

## 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; Howe et al. 1999; 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 have 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; Sievers et al. 2019a), 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 2018 International Planning Competition (IPC).

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; 2019b) for a *lifted* representation. Both graphs are used in classical planning for computing structural symmetries (Sievers et al. 2019b; 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 learning 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

\*Contact authors

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

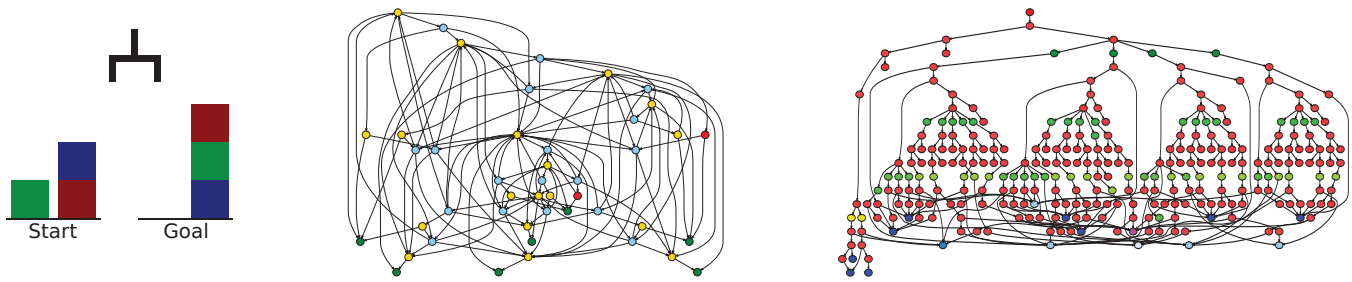


Figure 1: An example planning task (left) with its grounded graph representation (middle) and the lifted one (right). The task, *blocksworld*, uses a gripper to rearrange a set of blocks from an initial configuration to the goal configuration. The coloring of the graph nodes indicate node labels. For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

2017; Gilmer et al. 2017; Veličković et al. 2018). 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 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 across layers/steps, similar to that between a CNN and a recurrent neural network.

GNNs 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.

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. 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 algo-

gorithms 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  $A^*$  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 allocates a time budget to each planner and assigns the same allocation for all tasks, offline. Prominent examples include Fast Downward Stone Soup (FDSS) (Helmert et al. 2011) and Fast Downward Cedalion (Seipp, Sievers, and Hutter 2018).

Another approach uses supervised machine learning to predict an appropriate planner for a task. The predictive model requires for each task a feature representation, often handcrafted (Howe et al. 1999; Cenamor, de la Rosa, and Fernández 2016), including for example 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.

For cost-optimal planning, an online approach is a meta-search in the space of solution set preserving problem mod-

ifications (Fuentetaja et al. 2018), aiming at finding task formulations that a planner would work well on. The resulting planner, *MSP*, also ranked 12th in IPC 2018.

Yet another cost-optimal planner performing online selection is Delfi, which treats a planning task as an image and selects a planner from the portfolio 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 an image, and a CNN is used to perform image classification. Delfi won IPC 2018.

Note that graph representations have also been used for probabilistic planning (Toyer et al. 2018), which is applied to a specific domain. In such a setting, action schemas are shared among the input tasks, but it is unclear how the approach can be adapted to domain-independent settings.

## Graph Construction

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. 2019b); 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 are ignored when being converted to an image.

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.

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  $f : \mathcal{G} \times \Theta \rightarrow [0, 1]^D$  that computes the failure 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  $\theta$  that minimizes the cross-entropy loss function

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

## Graph Representation Learning

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  $\alpha_v$  denotes the attention weight, scoring in a sense the importance of the contribution of each node to the overall representation of the graph.

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  $\theta$  thus includes the parameter matrix  $W_{\text{logit}}$  and all the parameters in  $h_v$  and  $\alpha_v$ .

Graph neural networks differ in the parameterizations of the node representation  $h_v$  and possibly the attention weight  $\alpha_v$ . 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  $\sigma$  is an activation function (e.g., ReLU). The normalization is defined as

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

Using an initial feature matrix  $X$  (which, for example, can be defined based on node labels by using one-hot encoding) 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  $w_{\text{gate}}$  is a parameter vector. Hence, the overall parameter set for the model  $f$  by using the GCN architecture is

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

### 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:

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

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

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

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  $\alpha_v$  may be computed in the same manner as (4). Therefore, the overall parameter set for the model  $f$  by using GG-NN is

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

### Variants

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

## 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  $g : \mathcal{G} \times [D] \times \Theta \rightarrow [0, 1]^D$ , where  $[D]$  denotes the set of integers from 1 to  $D$ , and parameterize it as

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

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

Compare this model with  $f$  in (3). First, we introduce an additional parameter matrix  $W_{\text{fail}}$  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,

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

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	2,294 / 145	
# Nodes, min / max / mean / median	6 / 47,554 / 2,056 / 580	51 / 238,909 / 3,001 / 1,294
Node degree, 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  $j$ , hoping for 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. 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. Summary of the data set is given in Table 1.

We use the same portfolio as did Delfi (Katz et al. 2018), which contains 17 base planners, as it is convenient to compare with the image-based CNN approach. See the cited reference for details of the base planners. The tasks unsolvable by any of these planners within the time limit  $T = 1800s$  are ignored in the construction of the data set.

We prepare two sets of training/validation splits. The first set reuses the split in Delfi, which conforms to the competition setting where a single model is used for evaluation. On the other hand, to reduce the bias incurred by a single model, for another set we randomly generate 20 splits (with an approximately 9:1 ratio). In ten of them, tasks from the same domain are not separated, whereas in the other ten, they may. We call the former scenario *domain-preserving split* whereas the latter *random split*.

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. Table 1 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. The plot indicates that the lifted version is much smaller for the tasks with the largest grounded graphs.

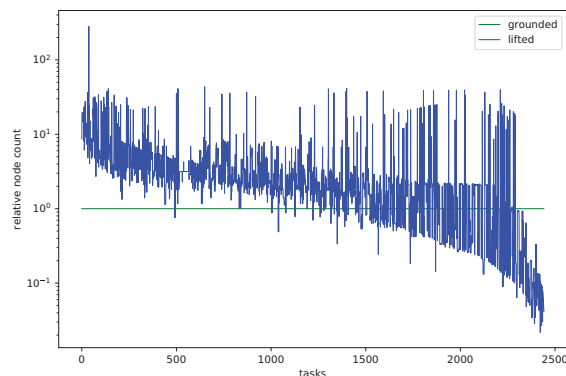


Figure 2: Node counts of the lifted graphs relative to that of the grounded ones (tasks sorted in increasing order of grounded size).

### 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  $w_{gate}$  in (4) by using  $D$  separate copies, one for each planner (see the subsection “Variants”).

We used eight CPU cores and one GPU for training. The consumed memory was approximately 5–10GB. The training of one model with one data split took approximately 10 minutes.

### Effectiveness of Graph Neural Networks

**Single planner; Delfi split:** We compare the performance of several types of methods, as summarized in Table 2. In addition to the coverage—the percentage of solved tasks—the column “eval. time” presents the time needed for selecting a planner, which includes, for example, the time to convert a planning task to a graph 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  $T = 1800s$ .

The first two methods are weak baselines. As the name suggests, “random planner” uniformly randomly selects a

Table 2: Percentage of solved tasks in the test set and average evaluation time of the method. Delfi split; single planner.

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
Enhanced features + random forest	82.1%	0.51s
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	<b>87.6%</b>	9.41s
Graph based, GG-NN, grounded	77.9%	14.53s
Graph based, GG-NN, lifted	81.4%	11.44s

planner, whereas “single planner for all tasks” uses the one 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 (Torralla et al. 2017). None of these methods is portfolio based and hence no time is needed for planner selection. Still, they are rather competitive for cost-optimal planning.

The next one is a machine learning approach based on enhanced features developed by (Fawcett et al. 2014). With these handcrafted features, any standard, non-deep, machine learning model may be applied. We use random forest, reportedly the best model experimented by the authors of the referenced paper. In addition to the percentage coverage, time to compute the features is reported in the table.

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, they also require less evaluation time. One reason is that lifted graphs

are less expensive to construct, albeit being larger on average.

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

**Single Planner; multiple splits:** We additionally report in Table 3 the results of multiple splits, for the lifted graphs, following (Sievers et al. 2019a). See the top three rows. Similar to the above observations, GCN consistently works better than GG-NN and it also outperforms CNN. Moreover, one generally obtains better performance by using random splits, compared with domain-preserving ones.

Table 3: Percentage of solved tasks in the test set (lifted version). Multiple splits; single planner.

	Domain-preserv.		Random	
	Mean	Std	Mean	Std
Image based, CNN	82.1%	6.6%	86.1%	5.5%
Graph based, GCN	<b>85.6%</b>	<b>5.5%</b>	<b>87.2%</b>	<b>3.5%</b>
Graph based, GG-NN	76.6%	5.8%	74.4%	2.7%
Adaptive, GCN	<b>91.1%</b>	<b>3.8%</b>	<b>92.1%</b>	<b>3.2%</b>
Adaptive, GG-NN	83.0%	5.8%	86.6%	2.0%

## Effectiveness of Adaptive Scheduling

We first verify the motivation of using more than one planner: frequently a planner solves a task rather quickly, if it ever does. In Figure 3 we plot a curve for each of the deep learning methods regarding the number of solved tasks as time evolves (to avoid cluttering, only those with the lifted version 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.

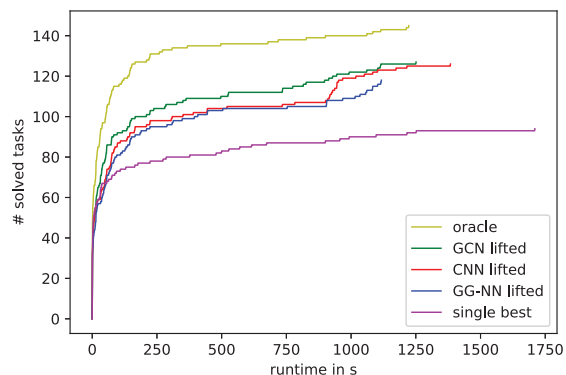


Figure 3: Number of solved tasks with respect to time. “Oracle” is the ceiling performance obtained by always selecting the fastest planner for each task. Curves for the grounded graphs are not shown to avoid cluttering.

**Two planners; Delfi split:** Based on this observation, 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. For a straightforward variant, we also consider a two-planner fixed scheduling, whereby two planners with the smallest failure probability are selected for execution, each given a timeout limit  $T/2$ .

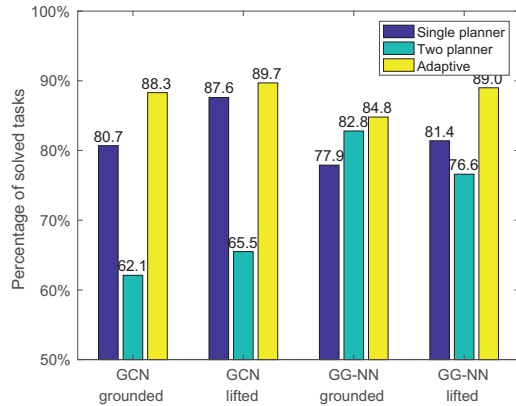


Figure 4: Percentage of solved tasks in the test set. Delfi split; two planners.

Figure 4 reports the results for the Delfi split. 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%. On the other hand, two-planner fixed scheduling is generally less competitive than even one-planner. This result is not surprising, because if the single-planner model  $f$  is not sufficiently accurate, one does not expect that selecting two may help. The adaptive model  $g$ , on the other hand, adapts to the case when the first planner fails and hence is more useful.

**Two planners; multiple splits:** For multiple splits, the results are reported in the bottom two rows of Table 3. One similarly sees that adaptive scheduling consistently improves over single-planner scheduling.

**Multiple planners; Delfi split; simple greedy:** The success of adaptive scheduling raises much interest in the use of more than one planner. A straightforward extension of the adaptive model  $g$  is to select more than two planners; but as the number of planners increases, the model becomes more and more complex and challenging to train. Instead, we explore offline, non-machine learning selection methods.

In Table 4 we consider the following approach:  $k$  planners are selected (independent of the task) and each is allotted  $T/k$  time. The first selected planner is the best performer on the training+validation set in the allotted time. The next planners are chosen in the same manner on the portion of the set not solved by previously selected planners. The top four rows of Table 4 report the results. For reference, the next four rows report the ceiling performance were the planners selected from the test set.

If  $k = 2$  planners are used, the coverage is 85.5%, lower than that achievable by the proposed adaptive scheduling approach, 89.7% (see Figure 4). If one is willing to use more planners, a better result is achieved with  $k = 3$ . However, the coverage does not increase monotonically with  $k$ . Moreover, the performance appears to be much sensitive to the change of  $k$ . Hence, it is challenging to find the best  $k$ .

Table 4: Percentage of solved tasks in the test set. Offline method. Delfi split. Compare with performance in Figure 4.

Method	Solved
(Greedy) Best 2 planners from train set	85.5%
(Greedy) Best 3 planners from train set	92.4%
(Greedy) Best 4 planners from train set	89.7%
(Greedy) Best 5 planners from train set	87.6%
(Greedy oracle) Best 2 planners from test set	93.8%
(Greedy oracle) Best 3 planners from test set	93.8%
(Greedy oracle) Best 4 planners from test set	93.1%
(Greedy oracle) Best 5 planners from test set	92.4%
Fast Downward Stone Soup	92.4%

**Multiple planners; multiple splits; FDSS:** A more sophisticated offline planner selection approach is Fast Downward Stone Soup (Helmert et al. 2011), wherein the selection of planners and their allotted time are optimized according to a certain score function. Its result is reported in the last row of Table 4. This result is achieved with four planners. The coverage coincides with the above greedy approach when taking  $k = 3$ .

## 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 CNNs for selecting candidate planners. Our 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.

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. An avenue of future research is to investigate more efficient and scalable training approaches.

We have seen that the use of multiple planners beyond two may improve the performance. Another line of future work is to extend the proposed adaptive scheduling to more

than two planners. In this case, the training set construction becomes more and more complex and the model may be challenging to train. Additionally, a principled approach of setting the right number of planners is to be developed.

## Acknowledgments

Patrick Ferber was funded by DFG grant 389792660 as part of TRR 248 (see <https://perspicuous-computing.science>).

## References

- 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. 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 *ICAPS*.
- Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2015. BDDs strike back (in AI planning). In *AAAI*.
- Edelkamp, S. 2001. Planning with pattern databases. In *ICAPS*.
- Fawcett, C.; Vallati, M.; Hutter, F.; Hoffmann, J.; Hoos, H. H.; and Leyton-Brown, K. 2014. Improved features for runtime prediction of domain-independent planners. In *ICAPS*.
- Franco, S.; Lelis, L. H. S.; and Barley, M. 2018. The complementary2 planner in the ipc 2018. In *IPC-9 planner abstracts*.
- Fuentetaja, R.; Barley, M.; Borrajo, D.; Douglas, J.; Franco, S.; and Riddle, P. 2018. The Meta-Search Planner (MSP) at IPC 2018. In *IPC-9 planner abstracts*.
- 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 *ICAPS*.
- Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *NIPS*.
- Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *AAAI*.
- Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What’s the difference anyway? In *ICAPS*.
- Helmert, M.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M. 2011. Fast Downward Stone Soup. In *IPC 2011 planner abstracts*.
- 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.
- Howe, A. E.; Dahlman, E.; Hansen, C.; Scheetz, M.; and von Mayrhauser, A. 1999. Exploiting competitive planner performance. In *ICAPS*.
- Katz, M., and Hoffmann, J. 2014. Mercury planner: Pushing the limits of partial delete relaxation. In *IPC-8 planner abstracts*.
- Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online planner selection for cost-optimal planning. In *IPC-9 planner abstracts*.
- 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*.
- Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting problem symmetries in state-based planners. In *AAAI*.
- Röger, G.; Sievers, S.; and Katz, M. 2018. Symmetry-based task reduction for relaxed reachability analysis. In *ICAPS*.
- 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 *ICAPS*.
- Seipp, J.; Sievers, S.; Helmert, M.; and Hutter, F. 2015. Automatic configuration of sequential planning portfolios. In *AAAI*.
- Seipp, J.; Sievers, S.; and Hutter, F. 2018. Fast Downward Cedalion. In *AAAI*.
- Shleyfman, A.; Katz, M.; Helmert, M.; Sievers, S.; and Wehrle, M. 2015. Heuristics and symmetries in classical planning. In *AAAI*.
- 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*.
- Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019a. Deep learning for cost-optimal planning: Task-dependent planner selection. In *AAAI*.
- Sievers, S.; Röger, G.; Wehrle, M.; and Katz, M. 2019b. Theoretical foundations for structural symmetries of lifted pddl tasks. In *Proc. ICAPS 2019*.
- Torralba, A.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient symbolic search for cost-optimal planning. *AIJ* 242.
- Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action schema networks: Generalised policies with deep learning. In *AAAI*.
- Vallati, M. 2012. A guide to portfolio-based planning. In *MIWAI*.
- Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph attention networks. In *ICLR*.
- Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In *ICAPS*.