A Proposal to Generate Planning Problems with Graph Neural Networks

Carlos Núñez-Molina, Pablo Mesejo, Juan Fernández-Olivares
Universidad de Granada, Spain
ccaarlos@ugr.es, pmesejo@decsai.ugr.es, faro@decsai.ugr.es

Abstract

In the field of Automated Planning, much effort has been devoted to learning planning domains from data. On the other hand, the task of generating planning problems for a given domain has received less attention. In most cases, these problems need to be created by hand or produced by hard-coded, domain-specific generators, which is a time-consuming endeavor. Having a large set of planning problems is useful for two main reasons. Firstly, they provide data to train Machine Learning methods (e.g., for learning planning heuristics and HTN domains) and, secondly, they can be used as benchmark problems in planning competitions. In this preliminary work, we propose a domain-independent method to generate planning problems for any typed STRIPS domain. We formulate problem generation as a Markov Decision Process and propose the use of Graph Neural Networks, which are trained with Reinforcement Learning in order to generate valid, quality (i.e., difficult to solve) and diverse problems. We hope our approach will enable the effortless generation of large sets of planning problems with the qualities desired by the user, thus providing an alternative to existing problem generation methods.

Introduction

Throughout the years, many works (Shen and Simon 1989; Pasula, Zettlemoyer, and Kaelbling 2007; Segura-Muros, Pérez, and Fernández-Olivares 2021) have tried to automatically learn planning domains from data, in order to alleviate the burden on domain designers. On the other hand, the task of generating planning problems for a given domain has received less attention. In most cases, they need to be created by hand or produced by hard-coded generators tailored to a specific domain. Having a large set of planning problems is useful for two main reasons. Firstly, many approaches that apply Machine Learning (ML) (Alpaydin 2021) to AP require training data in the form of planning problems and their solutions, such as those for learning planning heuristics (Shen, Trevizan, and Thiébaux 2020) and HTN domains (Hogg, Munoz-Avila, and Kuter 2008). Secondly, they can be used as benchmarks in planning competitions to compare the performance of different automated planners.

In this preliminary work, we design a method to automatically generate planning problems for any typed STRIPS domain. We formulate problem generation as a Markov Decision Process (MDP), in which a problem is not generated all at once but by executing a sequence of actions. We propose the use of Graph Neural Networks (GNNs) (Battaglia et al. 2018) to direct this generation process, i.e., select the best action to execute at each step in order to generate problems with the desired qualities. GNNs are a family of neural networks designed to work with relational data, represented as graphs. They receive a graph as input and perform iterative *message-passing computations* where the information about nodes (encoded as vector embeddings), and possibly also about edges, is propagated through the graph. Then, they output a prediction for every node, edge and/or the entire graph, depending on the required task. Thanks to their suitability for relational representations, GNNs have been successfully applied to different AP tasks (Shen, Trevizan, and Thiébaux 2020; Ma et al. 2020; Silver et al. 2020).

We will resort to Reinforcement Learning (RL) (Sutton and Barto 2018) for training the GNNs in our approach because it enables agents to learn without labeled examples, just from environment feedback (i.e., rewards). In our case, this means no example planning problems will be required to train our approach. The problems obtained with our method must exhibit the following properties:

- **Validity.** Valid problems must meet two criteria. Firstly, the initial state of the problem must represent a **consistent** (possible) situation of the world (e.g., an object can only be in a single place at the same time). These consistency constraints do not appear in the domain description and must be provided separately. Secondly, valid problems must be **solvable**, i.e., there must exist at least one valid plan from the initial state to the goal.
- **Quality.** This property depends on the problem characteristics desired by the user and, thus, must be defined by them. In this work, we will use difficulty as the only quality metric, i.e., the goal is to generate problems as hard to solve as possible (in terms of planning time) by an automated planner. This will encourage our generative method to search for problems with particular properties that make them challenging for a planner. In order to prevent really large problems, we will limit the maximum number of objects and atoms in the initial state.
- **Diversity.** The proposed method must generate problems very different from one another. More formally, the generated problems must be representative of the entire

problem subspace that satisfies the validity and quality requirements commented in the two previous points.

To the best of our knowledge, we propose the first domain-independent method for generating planning problems with all the three properties presented above. Nevertheless, it is important to note that this work is preliminary and only presents our proposal, leaving the actual implementation and experimentation for future work.

Related Work

Several works have proposed domain-independent methods for planning problem generation but, to this date, none of them have been able to generate problems that are simultaneously valid, of good quality and diverse. (Fern, Yoon, and Givan 2004) proposes a random-walk approach to generate planning problems. It randomly creates an initial state s_i and executes n actions at random to arrive at state s_g. Then, it selects a subset of the predicates of s_g, which constitutes the goal g, and returns the planning problem (s_i, g). Although the problems generated with this method are always solvable, they may not exhibit the other properties (consistency, quality and diversity), as they are generated at random. (Fuentetaja and De la Rosa 2012) also employs a random-walk approach but uses semantics-related information, provided by the user, to guarantee the consistency of the problems obtained. Thus, this method always generates valid problems but provides no guarantees about their diversity or quality, since they are also generated at random. (Marom and Rosman 2020) follows a different approach. It starts from a predefined goal state and performs a backward search for the initial state. The problems obtained are used to learn a planning heuristic. The proposed method estimates its uncertainty and uses this value to search for problems with the right difficulty for training the heuristic. Hence, this method is able to obtain valid and quality problems. However, it only works for domains with a single goal and for which there exists an *inverse transition model*, i.e., for every action a that transitions from state s to s' must exist an inverse action a' that goes from s' to s, which needs to be provided to the method.

Finally, it is worth to mention several works that address a similar problem to the one tackled in this work. (Katz and Sohrabi) addresses the problem of generating planning tasks whose causal graph matches a graph given as input. This work generates complete planning tasks (planning domain and problem) whereas our method generates planning problems for a given domain. (Torralba, Seipp, and Sievers 2021) proposes a method for selecting a set of planning problems with some desired properties. This method does not generate planning problems and, thus, requires a problem generator as input. For this reason, it can be seen as a complementary method to the one proposed in this work.

Proposed Method

In this section we describe our proposal, shown in Figure 1a, corresponding to a method that receives a typed STRIPS domain encoded in PDDL, along with a procedure to check the consistency of the problems generated and a list of predicates to consider for the goal, and outputs a set of valid, diverse and quality PDDL problems for that domain. We describe the problem formulation, the state representation, how GNNs can be used to direct problem generation and, finally, the RL method proposed for their training.

Problem Formulation as MDP

We propose to generate problems (s_i, g) via an iterative process which first generates the initial state s_i and, then, the problem goal g. The initial state generation phase starts at an empty state (with no objects or atoms) and successively adds new objects (constants) and atoms to construct s_i. Then, the goal generation phase starts at s_i and successively executes the actions present in the planning domain to arrive at a goal state s_g. Finally, g is obtained as a subset of the atoms which are true in s_g. The subset of predicates considered for the goal must be specified by the user.

This entire process is depicted in Figure 1b and a handcrafted example is given in Appendix B. It can be formulated as an MDP (S, A, app, r, t):

- S is the state space of the MDP. In our case, states correspond to (incomplete or fully-generated) planning problems (s_{ic}, s_{gc}). The subindex c (which comes from *current*) is used to denote that the initial state s_{ic} and goal state s_{gc} may not be completely generated yet.
- A is the action space, while $app : S \times A \to \{0, 1\}$ is the applicability function which determines if an action can be executed at a state or not. The set of applicable actions A_{app} is different for the initial state and goal generation phases. In the initial state generation phase, A_{app} corresponds to adding a new atom to the initial state. The objects this new atom is instantiated on can be already present in the initial state or not. If they are not, we refer to them as *virtual* objects, and are added to the initial state along with their corresponding atom. Thus, instantiating atoms on virtual objects is the mechanism we use to add new objects to the state. In the goal generation phase, A_{app} is the subset of actions in the planning domain for which their preconditions are met at the current state.
- $r : S \times A \to \mathbb{R}$ is the reward function, accounting for the validity and quality of the problems. Since all the generated problems are solvable (as s_g is obtained by executing domain actions from s_i), we only need to consider the consistency aspect of the validity. To do so, the user must provide a procedure that receives the (incomplete) initial state s_{ic} associated with an MDP state and returns if it is consistent or not. If the agent selects an action that would result in an inconsistent state, it receives a negative reward as penalty. Once a problem has been completely generated, it is solved with an automated planner. Then, the agent receives a final reward directly proportional to the resolution difficulty of the problem, e.g., the planning time or number of nodes expanded by the planner.
- $t : S \times A \to S$ is the transition function. In our problem, t is deterministic and returns the next state s' resulting from executing an applicable action at the current state s.

If an invalid action, i.e., one that produces an inconsistent state, is selected, then $s' = s$.

State Representation as Hypergraph

Since problem generation is guided by GNNs, we need to encode the MDP states (s_{ic}, s_{gc}) in a suitable representation for them. Thus, we choose to represent states as *labeled directed hypergraphs*. This hypergraph contains one node for every object/constant in s_{ic} and s_{gc} (the initial and goal states contain the same set of objects). Each node is labelled according to the type of the object it represents. Additionally, for every atom $a(o_1, o_2, ..., o_n)$ present in s_{ic} or s_{gc}, the hypergraph will contain a directed (hyper)edge $e(o_1, o_2, ..., o_n)$ linking the objects (nodes) $o_1, o_2, ..., o_n$ that appear in the atom in order. Each edge has a label which encodes the predicate type of the associated atom and whether such atom appears in s_{ic} or s_{gc} (unlike objects, an atom can appear in the initial state, goal state or both).

Generative Policies as GNNs

We propose to use two different stochastic RL policies for guiding the generation process, one for initial state generation and the other for goal generation, both implemented as GNNs. More specifically, we propose to implement the policies as *ACR-GNNs* (Barceló et al. 2020), a special type of GNN which is as expressive as FOC_2 logic, a fragment of first-order logic. In addition, they must be adapted to handle our state representation as labeled directed hypergraphs, by using the extension to GNNs proposed in (Ståhlberg, Bonet, and Geffner 2021, Algorithm 1).

Initial state generation policy. This policy is used to iteratively generate s_i starting from an empty state $s_0 = (\_, \_)$, with no objects or atoms. Our approach is heavily inspired by the method employed in (You et al. 2018) to generate molecular graphs. At each step, our policy receives the current state $(s_{ic}, \_)$, encoded as a hypergraph, and selects a new atom to add (see Figure 1c). Firstly, it adds one virtual object (node) of each existing type to the graph. These nodes make possible to add predicates instantiated on objects which are not present in the state and, thus, add new objects to the state, as explained previously. After adding the atom selected by the policy, the nodes representing virtual objects which do not appear in the new atom are deleted. Once the virtual objects have been added, an ACR-GNN is used to compute an embedding e_i for each node n_i in the graph and also an embedding e_G for the whole graph G. Then, the policy uses these embeddings to select an action representing an atom to add to the state. This action is composed of three different components, which are predicted in the following order:

1. **Predicate type.** A multi-layered perceptron (MLP) receives e_G and returns a probability distribution over the existing predicate types. This distribution is sampled to select the type of the atom to add.
2. **Objects to instantiate the predicate on.** We use a Recurrent Neural Network (RNN) to iteratively decide which objects to instantiate the predicate on. At the beginning, the RNN hidden state is initialized with the predicate type. Then, at each step, the RNN separately receives each e_i and outputs a probability p_i for each node, representing how likely that node is to be selected as the next object to instantiate the predicate on. This probability distribution is masked so that objects of type incompatible with the predicate are given zero probability. Next, the probability distribution is sampled to select an object o_k and the RNN hidden state is updated with the embedding e_k of the associated node. This process is repeated a times, where a is the arity of the predicate.
3. **Termination condition.** Finally, another MLP receives e_G and outputs a termination condition probability. If true, the initial state generation phase ends and $s_i = s_{ic}$. Then, the goal generation phase starts.

Goal generation policy. This policy iteratively generates s_g starting from s_i. At each step, it receives the current state (s_i, s_{gc}) and selects a domain action to execute (see Figure 1d). To do so, it follows a very similar process to the initial state generation policy. Firstly, it uses an ACR-GNN to compute e_i, for every n_i, and e_G, this time without adding virtual objects to the state. Then, the policy uses these embeddings to select the action to execute. This action is composed of three different components, predicted in the following order:

1. **Action type.** An MLP receives e_G and returns a probability distribution over the existing domain actions, which is sampled to select the action to execute. This probability distribution is masked so that non-applicable actions, i.e., those for which their preconditions are not met at the current state, are never chosen.
2. **Action parameters.** We use an RNN to iteratively decide which objects to ground, i.e., instantiate, the action on. At the beginning, the RNN hidden state is initialized with the selected action. Then, at each step, the RNN separately receives each e_i and outputs a probability p_i for each node, representing how likely that node is to be selected as the next object to instantiate the action on. This probability distribution is masked so that objects of type incompatible with the corresponding action parameter are given zero probability. Next, the probability distribution is sampled to select an object o_k and the RNN hidden state is updated with the embedding e_k of the associated node. This process is repeated n times, where n is the number of action parameters.
3. **Termination condition.** Finally, another MLP receives e_G and outputs a termination condition probability. If true, the goal generation phase ends and the current goal state s_{gc} is output as s_g. Then, the user-defined subset of true atoms in s_g is selected to obtain g and the final problem (s_i, g) is returned.

Policy Training with RL

We propose to use RL to train the entire generative system, composed of the initial state and goal generation policies, in an end-to-end fashion. More specifically, we plan to use the state-of-the-art RL algorithm known as Proximal Policy Optimization (PPO) (Schulman et al. 2017). Since PPO requires a *critic* module that predicts the value $V(s)$ of the

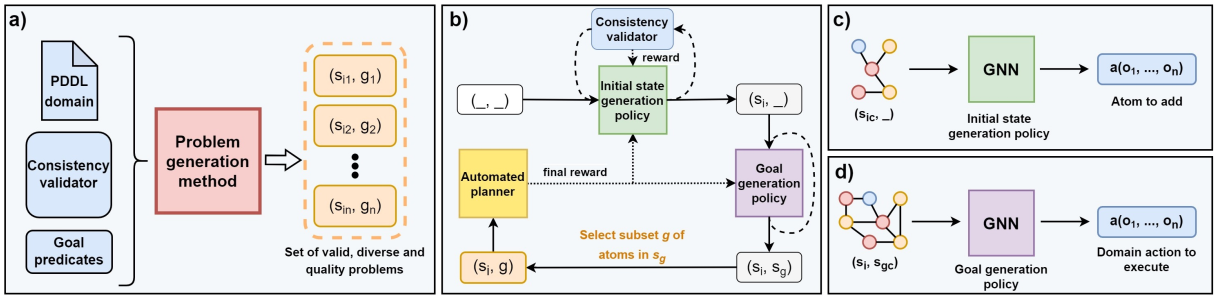

Figure 1: **Proposed method for problem generation. a) Method overview.** It receives the PDDL domain, a procedure for checking the consistency of the problems generated and the predicates to consider for the goal, and outputs a set of planning problems. **b) Process followed to generate a single problem.** Dashed lines represent the application of several actions whereas dotted lines are used to indicate the reward signal, used to train the generative policies with RL. **c) Initial state generation policy.** It receives a state $(s_{ic}, \_)$ corresponding to a partially-generated initial state and outputs the next atom to add. **d) Goal generation policy.** It receives a state (s_i, s_{gc}) representing a complete initial state but a partially-generated goal state and outputs the next (grounded) domain action to execute.

current state s, we will augment each policy with an additional MLP that receives e_G and ouputs $V(s)$.

The loss function being optimized by PPO includes an entropy term which encourages policies with high entropy, i.e., highly stochastic policies. This ensures sufficient exploration by the policies. Additionally, this entropy term should also boost the diversity of the problems generated. This is because highly stochastic policies should produce very different problems in distinct generation episodes.

Nonetheless, encouraging high-entropy policies may not be enough to ensure effective exploration in some domains. For example, let us assume a planning domain where an agent must traverse a grid while collecting colored keys to open doors of the same color. In this domain, there exist *interesting* problems (in our case, problems which are difficult to solve) corresponding to particular configurations of keys and doors, e.g., problems where the keys must be collected in a specific order and problems with dead ends. These problems seem unlikely to be uncovered by performing random exploration, no matter how stochastic the policies are. Were this be the case, we will resort to exploration schemes based on *intrinsic motivation* (Chentanez, Barto, and Singh 2004), where exploration is directed towards states with interesting properties such as *novelty*.

Proposed Experimentation

Once our method has been implemented, it will be used to generate problems for different planning domains, such as those used in ICAPS planning competitions. Our method will be compared against baseline methods that follow our problem generation approach but, instead of using the generative policies, select actions at random in order to construct s_i, g or both, as in (Fern, Yoon, and Givan 2004; Fuentetaja and De la Rosa 2012). This way, it will be possible to assess how effective the initial state and goal generation policies are at guiding the generation process, as opposed to simply picking actions at random.

We will evaluate the problems generated by our approach and the baseline methods in terms of their validity, quality and diversity. To evaluate problem validity, we only need to measure how often the methods select an invalid action and, to evaluate quality, we only need to solve the generated problems with an off-the-shelf planner and measure planning times or number of expanded nodes. Diversity, however, is harder to evaluate. One straightforward, though time-consuming, approach is to manually inspect the problems generated by the different methods. An alternative approach consists of using a metric that measures similarity or distance between problems, such as graph edit distance (Gao et al. 2010).

Conclusions

In this preliminary work, we have proposed a domain-independent method for generating planning problems for typed STRIPS domains. We propose to formulate the problem generation process as an MDP and use RL to train policies that learn to generate valid, quality and diverse problems. These generative policies will be implemented as GNNs which receive as input a (partially-generated) problem, encoded as a hypergraph, and select the next action to apply in the generation process. Once implemented, our method will be used to generate problems for several planning domains. These problems will be compared with the ones obtained by baseline methods, which select actions at random.

We hope our approach will provide a domain-independent and easy-to-use alternative for planning problem generation, thus replacing time-consuming, manual problem design and domain-specific generators. If successful, our method will enable the effortless generation of large amounts of valid, quality and diverse planning problems, which will be greatly useful to train ML/RL methods for AP and as benchmark problems in planning competitions.

Appendix A: Problem Generation for Active Learning

The purpose of the method proposed in this work is to facilitate the generation of large numbers of valid, diverse and quality planning problems, regardless of their intended use. Nevertheless, if our goal is to employ these problems as data for training ML techniques (e.g., planning heuristics) that learn to solve problems of the same domain, then our method can be made more suitable for this specific case.

To achieve this, we can resort to the Active Learning (Settles 2009) paradigm, in which the ML method being trained has control over its own training data, resulting in better sample efficiency. In order to integrate this paradigm into our problem generation method, we can substitute the automated planner, in charge of solving the problems generated and measuring their difficulty, with the ML technique being trained. This technique will receive the problems generated, perform several training iterations on them, and output a metric measuring the quality of the problems used as training data (e.g., the training loss/error). Then, this quality metric will be used as feedback for the generative policies, which will try to optimize it when generating the next batch of problems to train the ML technique. As this process repeats, the problem generation method will learn to generate problems of ever-increasing difficulty to train the ML technique, what should result in an increased sample efficiency when compared to the "standard" problem generation approach proposed in this work (using an automated planner to solve the problems generated).

Another benefit of this approach is that it will allow us to more easily evaluate the quality and diversity of the problems generated. To do so, we can compare the sample efficiency of the ML technique (e.g., how many training samples it needs to achieve a particular loss/error threshold) when trained on problems generated by our method versus those generated by the random baselines. If it is more sample efficient when trained on problems obtained with our approach, that will mean our problem generation method is able to produce better problems (in terms of quality and diversity) than those obtained by the random baselines, thus providing more useful information to train the ML technique. In addition to the baseline methods, we will also consider the comparison of our approach with other active learning methods, like the one proposed in (Marom and Rosman 2020), where the problems generated are used to learn a planning heuristic.

Appendix B: Problem Generation Example

In this appendix we provide a simple, handcrafted example of our problem generation method that illustrates how a single planning problem is created from start to finish. For this example, we will use *blocksworld* as our domain and, at each step (state), we will assume a random action is chosen from the set of applicable actions A_{app}. In the initial state generation phase, an action $a \in A_{app}$ corresponds to adding an atom to the (current) initial state s_{ic}, where this atom is obtained by instantiating a domain predicate (on, $ontable$, $handempty$, $holding$ and $clear$ in blocksworld) on objects of the correct type ($block$ type in blocksworld, as all predicates are only instantiated on objects of this type). These objects can be in s_{ic} or not. In the latter case, we call them $virtual$ objects and they are added to s_{ic} along with their corresponding atom. In the goal generation phase, an action $a \in A_{app}$ corresponds to executing a domain action ($stack$, $unstack$, $pickup$ and $putdown$ in blocksworld), in the (current) goal state s_{gc}, modifying the atoms in s_{gc} according to the *add* and *delete* effects of a. The selected action a must be grounded, i.e., instantiated, on objects of the correct type ($block$ type in blocksworld, as all actions are applied to objects of this type) present in s_{gc} and, also, its preconditions must be true in s_{gc}.

Additionally, we will assume we randomly choose when to stop generating the states s_{ic} and s_{gc}. In a real scenario, the generative policies would be in charge of both selecting the next action to execute and when to stop generating s_{ic} and s_{gc}. We represent the states of the MDP, corresponding to (incomplete or fully-generated) planning problems, as a tuple (s_{ic}, s_{gc}). We represent s_{ic} and s_{gc} as another tuple (O, P), where O is a set containing the objects present in the state (with their respective types), and P is a set containing the atoms of the state. We now detail the process followed to generate the example problem:

1. The generation process starts at an empty state $s_0 = (\_, \_)$ and the initial state generation phase begins. The selected action is $add\ ontable(o1)\ to\ s_{ic}$, where $o1$ is an object of type $block$. Since $o1$ corresponds to a virtual object, we also need to add it to the state. Thus, the resulting state is $s_1 = (\ (\{block\ o1\},\ \{ontable(o1)\}),\ \_)$, which corresponds to a consistent state as it does not violate any consistency rule for the blocksworld domain.
2. The action $add\ on(o2, o1)\ to\ s_{ic}$ is selected, where $o2$ is a virtual object of type $block$. The resulting state is $s_2 = (\ (\{block\ o1, block\ o2\},\ \{ontable(o1), on(o2, o1)\}),\ \_)$, which corresponds to a consistent state.
3. The action $add\ on(o1, o2)\ to\ s_{ic}$ is selected. The resulting state would be $s'_3 = (\ (\{block\ o1, block\ o2\},\ \{ontable(o1), on(o2, o1), on(o1, o2)\}),\ \_)$. However, the state s'_3 is not consistent (since a block cannot be simultaneously on top of and under another block), so the atom $on(o1, o2)$ will not be added to the state and $s_3 = s_2$.
4. The action $add\ clear(o2)\ to\ s_{ic}$ is selected and the resulting state is $s_4 = (\ (\{block\ o1, block\ o2\},\ \{ontable(o1), on(o2, o1), clear(o2)\}),\ \_)$, which corresponds to a consistent state.
5. The action $add\ handempty()\ to\ s_{ic}$ is selected and the resulting state is consistent. Assume the initial state generation phase concludes at this step. If that is the case, the initial state has been completely generated, i.e., $s_i = s_{ic}$, and the goal generation phase starts from s_i, i.e., $s_{gc} = s_i$. Thus, the next state is actually $s_5 = (\ (\{block\ o1, block\ o2\},\ \{ontable(o1), on(o2, o1), clear(o2), handempty()\}), (\{block\ o1, block\ o2\}, \{ontable(o1), on(o2, o1), clear(o2), handempty()\}))$.
6. As we are now in the goal generation phase, the set of applicable actions A_{app} corresponds to the domain ac-

tions whose preconditions are met in s_{gc}. Assume the action *apply* $unstack(o2, o1)$ *to* s_{gc} is selected. Then, the current goal state s_{gc} is modified with the effects of the selected action. Thus, the next state is $s_6 = (\,(\{block\ o1, block\ o2\}, \{ontable(o1), on(o2, o1), clear(o2), handempty()\}), (\{block\ o1, block\ o2\}, \{ontable(o1), holding(o2), clear(o1)\}))$.

7. The action *apply* $putdown(o2)$ *to* s_{gc} is selected and the resulting state is $s_7 = (\,(\{block\ o1, block\ o2\}, \{ontable(o1), on(o2, o1), clear(o2), handempty()\}), (\{block\ o1, block\ o2\}, \{ontable(o1), clear(o1), clear(o2), handempty(), ontable(o2)\}))$. Assume the goal generation phase concludes at this step, i.e., $s_g = s_{gc}$. If that is the case, the goal g must be obtained by selecting from s_g the subset of goal predicates given by the user. Assume we only consider for g the predicates *on* and *ontable*. Then, the goal is $g = \{ontable(o1), clear(o1), clear(o2), ontable(o2)\}$ and the problem generation method outputs the problem (s_i, g), which is shown in Listing 1.

Listing 1: Example problem generated with our method.

```
(define (problem
    example_blocksworld_problem)

(:domain blocksworld)

(:objects o1 o2 - block)

(:init (ontable o1) (on o2 o1)
       (clear o2) (handempty))

(:goal (ontable o1) (clear o1)
       (clear o2) (ontable o2))
)
```

Acknowledgements

This work is being partially funded by the Andalusian Regional Project PYC20-RE-049-UGR and the Spanish National Project RTI2018-098460-B-I00 with FEDER funds.

References

Alpaydin, E. 2021. *Machine learning*. MIT Press.

Barceló, P.; Kostylev, E.; Monet, M.; Pérez, J.; Reutter, J.; and Silva, J.-P. 2020. The logical expressiveness of graph neural networks. In *ICLR*.

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

Chentanez, N.; Barto, A.; and Singh, S. 2004. Intrinsically motivated reinforcement learning. *Advances in neural information processing systems*, 17.

Fern, A.; Yoon, S. W.; and Givan, R. 2004. Learning Domain-Specific Control Knowledge from Random Walks. In *ICAPS*, 191–199.

Fuentetaja, R.; and De la Rosa, T. 2012. A Planning-Based Approach for Generating Planning Problems. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Gao, X.; Xiao, B.; Tao, D.; and Li, X. 2010. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1): 113–129.

Hogg, C.; Munoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *AAAI*, 950–956.

Katz, M.; and Sohrabi, S. ???? Generating Data In Planning: SAS Planning Tasks of a Given Causal Structure. *HSDIP 2020*, 41.

Ma, T.; Ferber, P.; Huo, S.; Chen, J.; and Katz, M. 2020. Online planner selection with graph neural networks and adaptive scheduling. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 5077–5084.

Marom, O.; and Rosman, B. 2020. Utilising Uncertainty for Efficient Learning of Likely-Admissible Heuristics. In *ICAPS*, volume 30, 560–568.

Pasula, H. M.; Zettlemoyer, L. S.; and Kaelbling, L. P. 2007. Learning symbolic models of stochastic domains. *Journal of Artificial Intelligence Research*, 29: 309–352.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Segura-Muros, J. Á.; Pérez, R.; and Fernández-Olivares, J. 2021. Discovering relational and numerical expressions from plan traces for learning action models. *Applied Intelligence*, 51(11): 7973–7989.

Settles, B. 2009. Active learning literature survey.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning domain-independent planning heuristics with hypergraph networks. In *ICAPS*, volume 30, 574–584.

Shen, W. M.; and Simon, H. A. 1989. Rule Creation and Rule Learning Through Environmental Exploration. In *IJCAI*, 675–680. Morgan Kaufmann.

Silver, T.; Chitnis, R.; Curtis, A.; Tenenbaum, J.; Lozano-Perez, T.; and Kaelbling, L. P. 2020. Planning with learned object importance in large problem instances using graph neural networks. *arXiv preprint arXiv:2009.05613*.

Ståhlberg, S.; Bonet, B.; and Geffner, H. 2021. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. *arXiv preprint arXiv:2109.10129*.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Torralba, A.; Seipp, J.; and Sievers, S. 2021. Automatic instance generation for classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 31, 376–384.

You, J.; Liu, B.; Ying, Z.; Pande, V.; and Leskovec, J. 2018. Graph convolutional policy network for goal-directed molecular graph generation. *Advances in neural information processing systems*, 31.