ECAI 2024
U. Endriss et al. (Eds.)
© 2024 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution Non-Commercial License 4.0 (CC BY-NC 4.0).
doi:10.3233/FAIA240978

NeSIG: A Neuro-Symbolic Method for Learning to Generate Planning Problems

Carlos Núñez-Molina[a,b,*], **Pablo Mesejo**[a,b] **and Juan Fernández-Olivares**[a,b]

[a]University of Granada, Spain
[b]Andalusian Institute of Data Science and Computational Intelligence (DaSCI)
ORCID (Carlos Núñez-Molina): https://orcid.org/0000-0003-1450-7323, ORCID (Pablo Mesejo): https://orcid.org/0000-0001-9955-2101, ORCID (Juan Fernández-Olivares): https://orcid.org/0000-0002-7391-882X

Abstract. In the field of Automated Planning there is often the need for a set of planning problems from a particular domain, e.g., to be used as training data for Machine Learning or as benchmarks in planning competitions. In most cases, these problems are created either by hand or by a domain-specific generator, putting a burden on the human designers. In this paper we propose NeSIG, to the best of our knowledge the first domain-independent method for automatically generating planning problems that are valid, diverse and difficult to solve. We formulate problem generation as a Markov Decision Process and train two generative policies with Deep Reinforcement Learning to generate problems with the desired properties. We conduct experiments on three classical domains, comparing our approach against handcrafted, domain-specific instance generators and various ablations. Results show NeSIG is able to automatically generate valid and diverse problems of much greater difficulty (15.5 times more on geometric average) than domain-specific generators, while simultaneously reducing human effort when compared to them. Additionally, it can generalize to larger problems than those seen during training.

1 Introduction

Automated Planning (AP) [9] is a subfield of Artificial Intelligence devoted to providing goal-oriented, deliberative behaviour to both physical and virtual agents, e.g. robots and video game automated players. An automated planner receives as input the description of the planning task to solve, containing the environment dynamics, initial state and goal. It then carries out a search process in order to find a plan (sequence of actions) which achieves the task goal starting from its initial state. Planning tasks are usually described in a declarative, first-order logic (FOL) language such as PDDL (Planning Domain Definition Language) [10]. The PDDL description consists of a planning domain, containing the environment dynamics, and a planning problem/instance, containing the initial state and goal to achieve. This encoding allows the reuse of the same planning domain for planning tasks with different initial state and/or goal but which share their environment dynamics.

Throughout the years, many works [26, 20, 24] have tried to automatically learn planning domains from data, in order to alleviate the burden on domain designers. Nevertheless, the task of generating planning problems has received less attention. In most cases, they need to be created by hand or produced by hard-coded, domain-specific instance generators, which requires great human effort. Having a large set of planning problems is useful for several reasons. The main one is that many approaches which apply Machine Learning (ML) to AP use planning problems as training data, such as those for learning heuristics [25] and hierarchical, HTN domains [13]. Additionally, problems are used as benchmarks in planning competitions [29] and are useful for domain validation [17], i.e., ensuring the planning domain faithfully represents the environment dynamics.

In this work, we address the problem of automatically generating planning instances for a particular PDDL domain provided by the user, in order to reduce human effort. Generated problems must exhibit three desirable properties: **validity**, **diversity** and **quality**. A problem is considered valid if it is solvable and its initial state is consistent, i.e., it describes a possible initial state of the world. Diversity refers to the fact that generated problems should all be different from each other. Finally, the quality of a problem is defined by the user. In this work, we will use difficulty as our only quality measure, i.e., the goal is to generate problems as hard to solve as possible.

As our main contribution, we propose **NeSIG** (*Neuro-Symbolic Instance Generator*) which, to the best of our knowledge, is the first domain-independent method for automatically generating planning problems that are at the same time valid, diverse and of good quality (i.e., difficult to solve). NeSIG receives as inputs the PDDL description of the domain, a set of consistency constraints which generated problems must satisfy, the maximum size of the problems to generate and a list with the predicates and object types that can appear in the problem goals. Then, it leverages this information to learn to generate valid, diverse and difficult problems for the domain provided as input. A remarkable feature of our method is that **it does not need to observe a single example problem** from the domain and, thus, can be regarded as a *data-free* generative model for planning problems. Finally, we have limited our scope to typed-STRIPS domains with existential and negative preconditions.[1]

Consistency constraints describe the set of properties that problem initial states must satisfy in order to be considered consistent, i.e., so that they describe a possible initial state of the world in the

* Corresponding Author. Email: ccaarlos@ugr.es.

[1] We have used *Lifted PDDL* as our parser, which only provides support for this subset of ADL.

system modeled by the PDDL domain at hand. An example consistency constraint would be "*an object cannot be at two places at the same time*". Since planning problems are often crafted by hand, this consistency information is not encoded in the PDDL domain description. Instead, it is the duty of the human designer to create problems representing possible, *reasonable* situations of the world. In the case of handcrafted, domain-specific generators, consistency constraints are hard-coded into their problem-generation procedure, so that inconsistent problems are never generated. In other words, consistency information must always be present when generating problems in one way or another. Therefore, in order to alleviate the burden on human designers, we propose a novel semi-declarative language that combines FOL and Python-like syntax so that the consistency constraints required by NeSIG can be encoded in an intuitive and interpretable manner, thus reducing human effort when compared to manual problem design and handcrafted generators.

We formulate problem generation as a Markov Decision Process (MDP) [27], in which a problem is generated in a series of steps. Firstly, the initial state of the problem is generated by successively adding objects and atoms to an initially empty state (or some other state predefined by the user). Then, a sequence of domain actions (i.e., the actions present in the planning domain) are executed from the generated initial state to arrive at a goal state, where a subset of the atoms are selected to form the problem goal, according to the goal types and predicates provided by the user. We use Deep Reinforcement Learning (RL) [27] to train two generative policies, one for generating the problem initial state and one for the goal. These policies are learned by Neural Logic Machines (NLMs) [4], a neuro-symbolic deep neural network architecture capable of inductive learning and logic reasoning.

We test our method on three classical planning domains, *blocksworld*, *logistics* and *sokoban*, comparing the problems generated by NeSIG with those obtained by handcrafted, domain-specific generators and several ablations. Results show NeSIG obtains valid and diverse problems of much greater difficulty than the alternative approaches (15.5 times more than domain-specific generators on geometric average), while requiring little human effort, as our approach removes the need to design the problems manually or program a handcrafted generator. Additionally, NeSIG exhibits remarkable generalization abilities when tested on larger problems than those seen during training: in *logistics*, it generates harder problems than the handcrafted generator for up to twice the training size whereas, in *blocksworld* and *sokoban*, it outperforms the domain-specific generators for all test sizes considered.

2 Related work

Several works have proposed domain-independent methods for planning problem generation but, to the best of our knowledge, none of them have been able to generate problems that are simultaneously valid, of good quality and diverse. [6] 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 atoms of s_g, which constitutes the goal g, and returns the planning problem (s_i, g). Although the problems obtained are always solvable, they may not exhibit the other properties (consistency, quality and diversity), as they are generated at random. [8] also employs a random-walk approach but, unlike the previous work, it 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. [16] 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 problems of good quality. However, it only works for domains with a single, predefined goal and for which there exist an *inverse transition model*, i.e., for every action a that transitions from state s to s' an inverse action a' that goes from s' to s must exist, 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 paper. [14] proposes a method for obtaining diverse and difficult planning tasks with different causal graphs. This work generates complete tasks (i.e., domain-problem pairs) whereas NeSIG generates planning problems for the particular domain provided by the user. [28] proposes Autoscale, a method for obtaining valid and diverse problems with graded difficulty for their use in planning competitions. However, unlike our method, Autoscale does not generate problems on its own. Instead, it relies on domain-specific instance generators, selecting a set of problems with graded difficulty among the ones they generate. Therefore, Autoscale can be considered complementary to our approach, as it could be used to select problems among those NeSIG generates.

3 Background

3.1 Planning task representation

A planning task is a tuple formed by a planning domain and a planning problem (also known as an instance). Both the domain and problem are often represented in a formal FOL-based language such as PDDL [10]. In PDDL, the domain encodes the existing object types, predicates and actions available to solve the task, detailing for each action its parameters (variables), preconditions (conditions which must be true for the action to be applicable) and effects (how the action modifies the state). This information is encoded in *lifted* form, i.e., in terms of FOL variables which can be instantiated (*grounded*) on objects. On the other hand, the PDDL problem encodes the objects present in that particular instance, the set of atoms which are true at the initial state, and the goal to achieve, represented as a FOL formula (usually as just a conjunction of atoms).

3.2 Neural Logic Machines

A Neural Logic Machine (NLM) [4] is a deep neural network capable of learning from FOL data and performing logic reasoning. An NLM receives as input a set of predicates grounded on a set of objects. Then, it sequentially applies first-order rules to obtain a different set of output predicates instantiated on the same objects. Input predicates are represented as binary tensors containing the truth value for each grounding of the predicate on the set of objects. Given some input predicate p, if $p(o_i, o_j, o_k)$ is true (where i, j, k represent object indexes), then its associated tensor will contain a value of 1 at the (i, j, k) position. Output predicates and those inferred internally by the NLM are also represented as tensors, but they contain real values between 0 and 1. The NLM operates with these tensors by using neural modules that approximate boolean rules (and, or, not) and quantifications ($\forall$ and $\exists$), being expressive enough to realize a set of Horn clauses. Therefore, NLMs are more expressive than alternative architectures such as Graph Neural Networks [1], which is why they are used in this work.

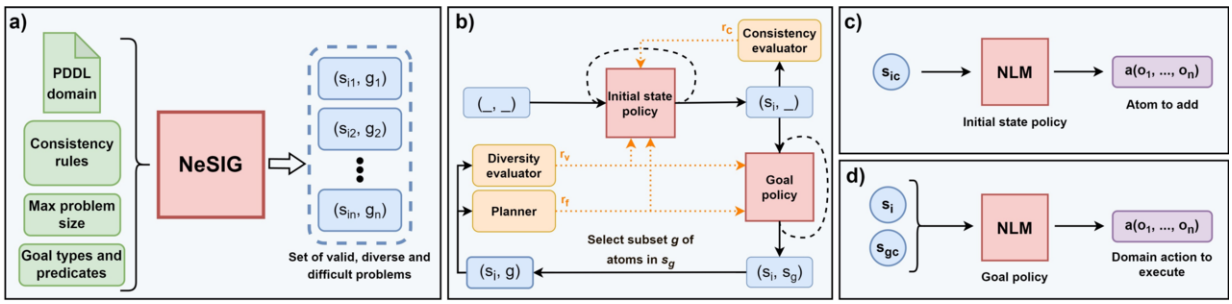

Figure 1: **NeSIG. a) Architecture overview.** NeSIG receives as inputs a PDDL domain, several consistency rules and some extra information (maximum problem size and goal types and predicates). It then trains two generative policies with RL (see subfigure b) so that they learn to generate valid, diverse and difficult problems for the domain provided as input. **b) Policy training with RL.** Dashed lines represent the application of several MDP actions, corresponding to adding an atom to the initial state in the case of the initial state policy (see subfigure c), or executing a domain action in the goal state in the case of the goal policy (see subfigure d). Dotted lines indicate the reward signal, accounting for the consistency r_c, diversity r_v and difficulty r_f of the problems generated. **c) Initial state policy.** It receives an MDP state $(s_{ic}, \_)$ corresponding to a partially-generated initial state and selects the next atom to add to s_{ic}. **d) Goal policy.** It receives an MDP state (s_i, s_{gc}) representing a complete initial state but a partially-generated goal state and selects the next domain action to execute in s_{gc}.

4 Neuro-Symbolic instance generation

In this section we describe our method, shown in Figure 1. NeSIG takes as inputs a PDDL planning domain, a set of consistency rules generated problems must satisfy and some extra information, corresponding to the maximum size of the problems to generate and a list with the predicates and object types which can appear in the problem goals.[2] It then learns to generate problems for that particular domain so that they are valid, diverse and difficult to solve (see Figure 1a). Problems are generated via an iterative process that first generates the problem initial state by sequentially adding objects and atoms to some predefined (often empty) state and, then, executes domain actions from the initial state generated to arrive at the goal state, where the problem goal is obtained according to the goal predicates and types specified by the user. We now discuss how validity, diversity and difficulty are defined and measured, present our novel MDP formulation of problem generation and explain how we leverage Deep RL to learn to generate problems with the desired properties.

4.1 Problem properties

4.1.1 Validity

This property can be decomposed into two sub-properties: **solvability** and **consistency**. A problem is considered solvable if there exists at least one valid plan that achieves the problem goal starting from its initial state, i.e., which *solves* the problem. By design, every problem generated by NeSIG is solvable, since the goal of a problem is generated by executing applicable domain actions from its initial state. A problem is considered *consistent* if its initial state represents a possible initial situation (state) within the system modeled by the planning domain, in other words, if it *makes sense*. An example consistency rule would be "*an object cannot be at two places at the same time*." Consistency constraints arise from the semantics of the domain and, as previously explained, are not encoded in its PDDL description so they need to be provided separately. Additionally, they depend on human interpretation and preferences. Going back to our previous example, some user could consider a state where one object is at two different places ($at(o, p_1)$, $at(o, p_2)$) at the same time to be consistent, and that choice would be completely valid as there is nothing in the PDDL domain that forbids it.

Due to the sequential nature of our proposed method, in which problem initial states are generated by incrementally adding objects and atoms to an initially empty state or some other state provided by the user, we distinguish between **continuous** and **eventual** consistency. A continuous consistency rule is one which must be *continuously* satisfied throughout the entire initial state generation process. In order to make a continuous-inconsistent state consistent again we would need to remove some atom(s) and/or object(s) from the state, which is forbidden in our method. For this reason, NeSIG never adds objects or atoms which result in continuous-inconsistent states. An example continuous consistency rule would be "*an object o cannot be at two places p_1, p_2 at the same time*", i.e., $at(o, p_1), at(o, p_2)$ is forbidden. If this constraint is not met, we would need to remove either $at(o, p_1)$ or $at(o, p_2)$ from the initial state which, as previously stated, is forbidden. On the other hand, eventual consistency rules are those which must be *eventually* satisfied once the initial state has been completely generated, but do not need to be met at each step of the generation process. An eventual-inconsistent state can be made consistent if some particular combination of object(s) and/or atom(s) are added to it. Therefore, eventual consistency is only checked at the end of the initial state generation process. An example eventual consistency rule would be "*the initial state must contain at least one object of type t*". If this constraint is not met, we can simply add an object of type t to the state to make it eventual-consistent.

These consistency rules are encapsulated in a consistency evaluator that provides two methods. The first one returns whether the state resulting from adding some atom (and optionally some objects) to the current state is continuous-consistent or not. The second method receives a completely-generated initial state and checks whether it is eventual-consistent or not. Although consistency rules must be provided by a human designer on a per-domain basis, doing so is often much simpler than devising a procedure for generating a diverse set of consistent problems, i.e., programming an instance generator. To reduce human effort even further, we have designed a novel, semi-declarative language for describing consistency rules. It allows the construction of first-order logic (FOL) formulas (with counting quantifiers) expressing conditions about state objects and

[2] Additionally, NeSIG may also take as input the list of object types that can be added to the problem initial state during generation. Nonetheless, this is completely optional and is only used for improving NLM efficiency.

atoms. For example, the consistency rule "*the initial state must contain at least 3 objects of type city*" can be concisely expressed as $TE(x, type(x, city)) >= 3$, where TE stands for *There Exists* and x is a variable. These formulas are then automatically evaluated, and their truth value is stored in a Python boolean variable. Therefore, we can encode consistency rules using either standard Python, FOL or a combination of them. This choice is transparent to NeSIG and does not impact training. We provide the consistency rules for each domain in the Appendix [18], showing how our semi-declarative language makes possible to represent consistency constraints in an interpretable manner with just a few lines of code.

4.1.2 Diversity

This property measures how different generated problems are from each other. In order to measure diversity, we automatically extract a set of interpretable features that describes the objects and atoms of each problem and their relationships. We say that two objects are *connected* if they are instantiated on the same atom, regardless of position. Based on this idea, we define the sets of *connection features* c_μ and c_σ. $c_\mu[t_i][p][t_j]$ encodes how many objects of type t_j, on average, each object of type t_i is connected to through atoms of predicate type p. Analogously, $c_\sigma[t_i][p][t_j]$ contains the standard deviation instead of the mean number of connections. For example, a value $c_\mu[city][in][location] = 3$ means that each city contains (atom in) an average of three locations, whereas $c_\sigma[city][in][location] = 2$ means that the standard deviation between the number of locations in each city is 2 (i.e., not every city contains the same number of locations). In total, we extract 7 groups of features, corresponding to the number of objects of each type in the problem and, separately for the initial state and goal, the number of atoms of each predicate type, c_μ and c_σ. They are divided by their sum so that, for each problem, features in each group add up to one. Then, we calculate the pairwise problem distance as the absolute difference between their feature vectors, dividing distances by $7 * 2 = 14$ to normalize them to the $[0, 1]$ range. Finally, the diversity of a problem is equal to its average distance to all the problems in the set (excluding itself).

4.1.3 Difficulty

In this work, we measure the quality of a problem by its difficulty. In other words, our goal is to generate problems which are as hard to solve by a planner as possible (in addition to being consistent and diverse). We have chosen difficulty as our quality measure because it plays a central role in AP, where great effort has been devoted to studying problem difficulty [3] and developing efficient algorithms for solving difficult problems [2]. We measure difficulty as the number of nodes a particular planner needed to expand to solve the problem. Since this measure depends on the planner employed, we calculate problem difficulty with a different set of planners at training and test time, to evaluate whether NeSIG is able to generate problems which are challenging for different planners.

4.2 Problem generation as MDP

We propose to generate problems of the form (s_i, g), where s_i is the problem initial state and g is the goal, via an iterative process which first generates s_i and then g. The initial state generation phase starts either from an empty state (with no objects or atoms) or from some predefined state provided by the user. Then, at each step, a new atom is added to the initial state and, optionally, one or more new objects. Once s_i has been completely generated, the goal generation phase begins if the state meets the eventual consistency constraints. Otherwise, the problem is discarded. Starting from s_i, the goal generation phase successively executes the actions available in the domain to arrive at another state, known as the goal state s_g. Finally, the goal g is obtained by selecting a subset of the atoms in s_g, according to the goal predicates and object types specified by the user. For instance, in the *blocksworld* domain, problem goals only contain atoms of the form *on(block,block)* by design. This entire process is depicted in Figure 1b and a handcrafted example is provided in the Appendix. It can be formulated as an undiscounted, finite-horizon MDP (S, A, app, t, r):

- S is the state space of the MDP. In our case, states correspond to (incomplete or fully-generated) planning problems $s = (s_{ic}, s_{gc})$. We use the subindex c (*current*) to denote when the initial state s_{ic} and goal state s_{gc} may not be completely generated yet.
- A is the action space, while $app : S \times A \rightarrow \{0, 1\}$ 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 s_{ic} which preserves the continuous consistency constraints (see Section 4.1.1). The objects this new atom is instantiated on can already be present in s_{ic} or not. If they are not, we refer to them as *virtual* objects, and are added to s_{ic} alongside their corresponding atom. For example, if the applicable action $add\ ontable(b1)$ is selected and the object $b1$ does not exist in s_{ic}, then both the atom $ontable(b1)$ and the object $b1$ will be added to s_{ic}. Thus, instantiating atoms on virtual objects is the mechanism we use to add new objects to the problem. 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 goal state s_{gc}. Additionally, we add a *termination action* end to A_{app}. When end is applied during the initial state generation phase, $s_i = s_{ic}$ is fixed and, if s_i is eventual-consistent, the goal generation phase starts. Otherwise, the MDP episode concludes. When end is applied during the goal generation phase, $s_g = s_{gc}$ is fixed, so the problem (s_i, g) is returned and the episode concludes. In order to control problem size, we set a maximum number of actions for each generation phase so, if this number is reached, end is executed and the corresponding phase concludes.
- $t : S \times A \rightarrow S$ is the transition function. In our setting, t is deterministic and returns the next MDP state (i.e., problem) resulting from executing an applicable action at the current state.
- $r : S \times A \rightarrow \mathbb{R}$ is the reward function. In our setting, there are three different reward sub-types accounting for problem consistency, difficulty and diversity. At the end of the initial state generation phase, a consistency reward $r_c = -1$ is given if s_i is eventual-inconsistent[3], as a form of penalization. At the end of the goal generation phase, problems receive a difficulty reward r_f equal to the logarithm of their difficulty, and a diversity reward r_v equal to their diversity. In every other situation, r_c, r_f and r_v are all 0. Finally, the (total) reward is calculated as follows:

$$r = r_c + min\Big(\frac{r_v}{\theta}, 1\Big) \cdot r_f \tag{1}$$

where $\theta \in [0, 1]$ is a hyperparameter known as the *diversity threshold*. We now explain the rationale behind Equation 1. MDP

[3] Since actions resulting in continuous-inconsistent states are never executed, we know s_i meets the continuous consistency constraints.

trajectories resulting in eventual-inconsistent problems will receive a reward $r = -1$ in their last sample, since the difficulty and diversity of an inconsistent problem is 0. For trajectories resulting in consistent problems, the reward (for the last sample) will be equal to r_f scaled down by a factor $min(r_v/\theta, 1)$, which depends on the diversity: if $r_v \geq \theta$, then $r = r_f$ whereas, if $r_v < \theta$, r_f will be scaled down up to a minimum of $r = 0$, in case $r_v = 0$. This reward function r balances problem consistency, diversity and difficulty. By maximizing it, we hope NeSIG will learn to generate consistent problems with a diversity close to θ (since diversity values r_v larger than θ do not increase r and values lower than θ reduce r considerably) and as difficult to solve as possible.

4.3 Learning to generate problems with RL

We use two different policies for guiding problem generation. One policy generates the initial state s_i of each problem, whereas the other generates its goal g. Each policy is encoded by a separate NLM (see Section 3.2).

At each step, the corresponding NLM receives information about the current MDP state. In the case of the initial state policy, it receives a tensor representation of the atoms and objects in the current initial state s_{ic}. This set of objects contains both the actual objects in s_{ic} and the new, virtual objects that can be added to the state alongside the next atom. The set of virtual objects is automatically inferred from the predicate information encoded in the PDDL domain. In the case of the goal policy, the NLM receives as input a concatenation of the tensor representations of the initial state s_i and current goal state s_{gc}. Since no new objects can be added during the goal generation phase, no virtual objects are used. Additionally, both NLMs receive as extra information the percentage of actions executed in the corresponding phase (relative to the maximum number of actions allowed), for each object its type and whether it is virtual or not, the total number of objects of each type, and the total number of atoms of each predicate type in the initial state and, for the goal policy NLM, also in the goal state.

The output of the NLM is represented as a new set of atoms, where each atom is associated with a different MDP action $a \in A$, corresponding to either a new atom to add to s_{ic} (for the initial state policy) or a domain action to apply to s_{gc} (for the goal policy), in addition to the termination action end. The NLM outputs a real value for each atom (action) in this set. Then, we mask out inapplicable actions $a \notin A_{app}$, corresponding to either atoms that violate the continuous consistency constraints (for the initial state policy) or domain actions whose preconditions are not met at s_{gc} (for the goal policy). Finally, we apply the softmax function to obtain a probability distribution over applicable actions $a \in App$, from which we sample the action to execute at the current MDP state.

In order to train the initial state and goal policies, we resort to the Deep RL algorithm Proximal Policy Optimization (PPO) [23]. Since PPO is an actor-critic algorithm, we need to employ an additional critic NLM for each policy, whose sole purpose is to evaluate the current MDP state. The two policies are trained simultaneously in an end-to-end fashion. The initial state policy receives rewards accounting for problem consistency, diversity and difficulty. On the other hand, the reward signal the goal policy receives accounts for diversity and difficulty but not consistency, since the consistency of a problem is independent of its goal g and, thus, of the goal policy. In order to calculate the PPO advantages, we use the Generalized Advantage Estimation (GAE) [22] method. However, we found the best λ value to be equal to 1, which is equivalent to simply calculating advantages with the n-step returns (i.e., not using GAE). Moreover, we use a policy entropy bonus as proposed in [23] to encourage sufficient exploration, in addition to the diversity reward.

5 Experimentation

In this section we detail our experimental setup and analyze the results of our experiments, where we compare NeSIG against alternative approaches. Our full code and data can be found in GitHub [19]. We have made available a Docker image for easy deployment, as we intend for our method to become a staple tool in the AP community.

5.1 Experimental setup

We perform experiments on a set of diverse and well-known planning domains: *blocksworld*, *logistics* and *sokoban*. In *blocksworld*, a set of stackable blocks needs to be re-assembled with a gripper. *Logistics* represents a transportation task where a set of packages needs to be delivered across locations and cities using airplanes and trucks. *Sokoban* is a challenging puzzle where several boxes must be pushed to their goal locations. In *blocksworld* and *logistics*, the initial state generation state starts from an empty state s_i with no objects or atoms. In *sokoban*, s_i initially describes an empty NxM map with no robots, walls or boxes, which will be added at generation time. The PDDL description for each domain can be found in the Appendix.

We train NeSIG separately on each domain, performing 5000 training steps using Adam [15] with a learning rate of 10^{-3}. Each experiment is run on 25 threads of an AMD EPYC 7742 CPU and one Nvidia A100 GPU, although our method can be trained on consumer-grade GPUs since only 8 GBs of VRAM are needed. In each training step, we generate 25 problems by executing up to 15 initial state actions (i.e., adding a maximum of 15 atoms to s_i) and up to 60 goal actions in *blocksworld* and *logistics*. For *sokoban*, we execute up to 75 goal actions, as this domain is more challenging than the others, and use a map of size 5x5. Every 250 training steps, we perform one validation epoch, where 100 problems are generated and the reward r of each problem is obtained using Equation 1. We calculate the *validation score* of the model as the average problem reward and, once training concludes, we load the model checkpoint with the best validation score for testing. The complete list of hyperparameters is provided in the Appendix. We use almost identical values for each domain so as to show our method needs little hyperparameter tuning.

Problem difficulty is calculated as the average number of nodes expanded by one or more planners to solve the problem. We employ the planners provided by FastDownward (FD) [11]. During training, we solve each problem with LAMA-first [21] using up to 500 MB of memory and 5 minutes of planning time, setting a difficulty of 10^6 for problems that could not be solved under those limits. At test time, we use LAMA-first, lazy-greedy search with the FF heuristic [12] and lazy-greedy with the additive heuristic [2] with a memory limit of 8 GB and time limit of 30 minutes, setting a difficulty of 10^8 for terminated problems. We use different planners for training and testing to evaluate whether NeSIG can generate problems that are challenging for several planners. In the Appendix, we provide experiments with optimal planners. Finally, for efficiency purposes, we generate small problems during training and then evaluate the generalization abilities of NeSIG by generating larger problems at test time (see Figure 2).

Several methods are compared to NeSIG in our experiments. First, we employ ablations where either s_i (*random-init* models), s_g (*random-goal* models) or both (*random-both* models) are generated

Table 1: **Same test-size experiment results.** The table compares the problems generated by NeSIG, several ablations (*random-init*, *random-goal* and *random-both* models) and the domain-specific generator (*ad hoc* model) in *blocksworld*, *logistics* and *sokoban*. For each domain and model, we generate 100 test problems with the same maximum size D used during training, corresponding to 15 max atoms in s_i. In *sokoban*, we use a map size of 5x5. We evaluate the consistency, difficulty, diversity and generation time of the test problems generated, showing for each property its mean value and standard deviation ($\pm$) across 5 random seeds. Since the ad hoc models do not require training, we use a single initial random seed (which will be used to deterministically obtain the seed to generate each problem), which is why their std values are always 0. Consistency is measured as the percentage of problems that meet the eventual consistency rules. Difficulty is measured as the mean number of nodes the test planners needed to expand to solve the problems. When calculating the mean difficulty and diversity, we do not consider inconsistent problems. Time refers to the total generation time (in seconds) needed to generate the whole set of 100 test problems.

Property	Blocksworld NeSIG	random-init	random-goal	random-both	ad hoc
Consistency	.986±.01	.14±.022	.986±.012	.14±.022	1.0±.0
Difficulty	325±36	94±24	36±4	30±2	83±0
Diversity	.025±.002	.026±.014	.033±.006	.026±.014	.024±.0
Time	26±4	11±2	22±1	9±0	4±0

Property	Logistics NeSIG	random-init	random-goal	random-both	ad hoc
Consistency	.998±.004	.254±.047	.994±.008	.254±.047	1.0±.0
Difficulty	76±4	14±2	13±1	5±1	16±0
Diversity	.196±.01	.222±.009	.167±.003	.25±.007	.264±.0
Time	28±3	14±1	26±1	13±1	4±0

Property	Sokoban NeSIG	random-init	random-goal	random-both	ad hoc
Consistency	1.0±.0	.994±.005	.998±.004	.994±.005	1.0±.0
Difficulty	2.4e5±7.6e4	5±0	1.8e4±1.7e4	6±1	1.2e3±0
Diversity	.016±.001	.007±.0	.013±.001	.007±.0	.016±.0
Time	221±17	327±8	255±36	333±4	1019±0

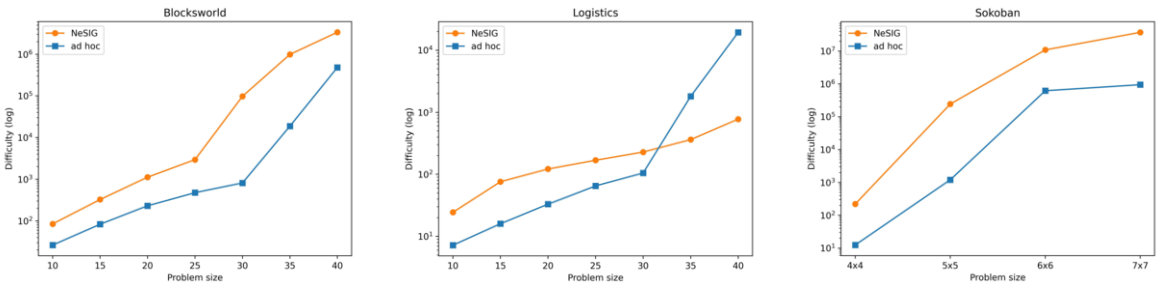

Figure 2: **Problem size generalization results.** The plots show the mean difficulty (in log scale) obtained by NeSIG across five different seeds, when tested on larger (and smaller) problems than those seen during training. We also plot the problem difficulty of the domain-specific generators (*ad hoc* models) for comparison purposes. In *blocksworld* and *logistics*, problem size is measured as the maximum number of atoms allowed in the initial state s_i. In *sokoban*, it is measured by the map size NxM. The maximum number of initial state and goal actions used by NeSIG for each problem size, along with the parameters of the *ad hoc* models, are detailed in the Appendix.

by executing random actions $a \in A_{app}$. We note our *random-both* model is equivalent to the method proposed in [8], which also generates s_i and s_g at random. We do not compare with Autoscale [28] since it leverages domain-specific generators to obtain problems of graded difficulty, often by gradually incrementing their size, whereas our goal is instead to maximize problem difficulty given a limit on their size. For this reason, we directly utilize the ad hoc, domain-specific generators (*ad hoc* models) used in the International Planning Competitions (IPCs) [29], choosing their parameter values to maximize problem diversity (see Appendix for the exact values). Nonetheless, the *sokoban* generator allowed for little flexibility (e.g., problems of size 5x5 could not have more than two boxes), so we have implemented our own based on a trial and error strategy which obtains s_i by placing objects at random on the grid, randomly moves boxes to obtain g, makes sure g can be achieved from s_i and, otherwise, discards the problem and starts again. For a fair comparison with NeSIG, we discard generated *blocksworld* and *logistics* problems with size smaller than $D-2$, where D is the maximum problem size, measured as the maximum number of atoms in s_i.

5.2 Analysis of results

Table 1 compares the problems generated by NeSIG, its ablations and the domain-specific generators (*ad hoc* models) using the same problem size for training and testing. It can be observed that NeSIG successfully learns to generate problems according to the user-defined consistency rules, as it seldom generates inconsistent problems in *blocksworld* and *logistics*, and actually achieves perfect consistency (100%) in *sokoban*. Additionally, NeSIG generates problems that are significantly more difficult than those from domain-specific generators: 3.9 times in *blocksworld*, 4.75 times in *logistics*, and 200 times in *sokoban*, for a total (geometric) average of 15.5 times more difficulty. Despite this, NeSIG achieves only 8% less diversity than the domain-specific generators on geometric average, which is surprising considering that the random procedure followed by the latter results in highly diverse problems. This is a remarkable result, as it means that our proposed method does not need to sacrifice diversity in order to increase problem difficulty, e.g., by learning to only generate a certain type of problem, thus effectively learning to balance difficulty

and diversity. Moreover, by leveraging parallel GPU computation, we can generate 100 problems with NeSIG in only half a minute for *blocksworld* and *logistics*, and in less than four minutes for *sokoban*. Finally, we observe that NeSIG consistently achieves similar results across runs, as indicated by the low standard deviations.

We now turn our attention to the ablation models. It can be observed that using a random policy for initial state generation (*random-init* and *random-both* models) severely degrades consistency in *blocksworld* and *logistics*. This shows that, for domains with complex consistency rules such as *blocksworld* and *logistics*, a trained (i.e., non-random) generative policy is needed to reliably generate consistent initial states. Additionally, ablations also significantly impair problem difficulty, although the effect of each policy ablation depends on the particular domain considered. In *blocksworld*, it is more important to train the goal policy than the initial state policy, since *random-init* achieves better difficulty than *random-goal*. In *sokoban*, the opposite case happens, as *random-goal* achieves several orders of magnitude better difficulty than *random-init*. Finally, in *logistics* the two policies seem to be equally important, as both ablations attain similar difficulty.

The *random-both* model represents the full ablation where no policy is trained, thus obtaining the worst results among all models. However, an important advantage of this model over NeSIG and the other ablations is that it does not require any type of training so, as long as consistency rules are provided (which can be done easily using our proposed consistency language), it can be quickly applied to generate problems for any (typed-STRIPS) planning domain. Although problems generated with this approach are easier to solve than those from the *ad hoc* models, problem difficulty can often be easily raised by incrementing problem size, just as Autoscale and *ad hoc* models do. Therefore, for cases where increasing problem size is acceptable or problem difficulty is not a concern, the *random-both* model offers a general and low-effort alternative to domain-specific generators, serving as a side contribution of our work.

Figure 2 shows the difficulty obtained by NeSIG when tested on problems of different size than those used during training. We also plot the difficulty of domain-specific generators for comparison purposes. In *logistics*, NeSIG successfully generalizes to problems up to twice the size of those seen in training, beating the *ad hoc* model in terms of difficulty. However, for sizes 35 and 40, there is a sudden spike in the difficulty of problems from the *ad hoc* model, which manages to outperform NeSIG. Our hypothesis is that the patterns learned by NeSIG about which problem features result in high difficulty do not apply to problems with more than 30 atoms. Therefore, in order for our method to generalize past this point, it should be trained on larger problems. In *blocksworld* and *sokoban*, NeSIG displays even better generalization abilities, obtaining several times more difficulty (note the logarithmic Y-axis in Figure 2) than the domain-specific generators for every problem size tested.

In conclusion, NeSIG is able to generate consistent problems with high difficulty and diversity, successfully generalizing to problems several times larger than those seen during training. These are remarkable results, especially taking into consideration that our method is domain-independent, whereas *ad hoc* models have been tailored to each particular domain and leverage extensive domain knowledge. For example, the *blocksworld* generator uses an ad hoc formula to make sure that every consistent state has the same probability of being generated. In *logistics*, the *ad hoc* model obtains the goal by randomly shuffling the packages in the initial state, knowing in advance that such a goal will always be achievable. In *sokoban*, the original ad hoc generator employed a complex procedure that allowed for little flexibility (e.g., problems of size 5x5 could not have more than two boxes). Our new *sokoban* generator does not have this limitation but, in exchange, it is very slow (e.g., it needs 1019 seconds to generate 100 problems of size 5x5, as shown in Table 1). When compared to ad hoc generators, NeSIG requires little prior knowledge, as it only receives as inputs the maximum problem size, the types and predicates that can appear in goals, and the set of properties (consistency constraints) that initial states must satisfy. Moreover, with our proposed semi-declarative language, these consistency constraints can be easily and intuitively encoded (see Appendix for concrete examples), thus reducing human effort even further.

6 Conclusion

In this work we introduced NeSIG, to the best of our knowledge the first domain-independent method for the automatic generation of planning problems that are simultaneously valid, diverse and difficult to solve. We formulated problem generation as an MDP, training two policies with Deep RL to generate problems with the desired properties. Both policies were encoded by NLMs, a neuro-symbolic deep neural network architecture capable of working with FOL data.

A remarkable feature of our method is that it does not require a training dataset of example problems. Instead, it only receives as inputs the PDDL domain description and a set of consistency constraints generated problems must satisfy, along with some extra information (maximum problem size and the types and predicates that are allowed in goals). Therefore, NeSIG requires less prior knowledge than handcrafted, domain-specific generators such as those often used in the IPCs. Moreover, we proposed a semi-declarative language for encoding consistency constraints in an intuitive and interpretable manner, further reducing human effort.

We tested NeSIG on three classical domains, comparing our approach against domain-specific generators and several ablations. Results show NeSIG successfully generates valid problems which are as diverse as those from domain-specific generators but considerably more difficult (15.5 times more on geometric average). Additionally, it showcases impressive generalization abilities, as it generates harder problems than the ad hoc generator in *logistics* for up to twice the training size, and surpasses the difficulty of domain-specific generators in *blocksworld* and *sokoban* for all test sizes considered. In light of the results obtained, we believe our work establishes a new state of the art in planning problem generation and hope it will prove useful to the Automated Planning community.

We note the choice of consistency constraints and quality metric to optimize depends on user preferences regarding the type of problems to generate. Therefore, in future work, we plan to harness the flexibility of NeSIG by generating problems according to different user preferences (e.g., maximizing plan length instead of planning difficulty), where hard constraints will be represented as consistency rules and soft constraints numerically as part of the reward function. We will also extend the expressivity of our method beyond typed STRIPS, e.g., by generating PDDL2.1 [7] problems with numeric fluents. Additionally, we also plan to explore several applications of our method. A few examples would be *automated curriculum generation*, i.e., generating problems of just the right difficulty for efficiently training an AI agent to solve a particular set of tasks, and *adversarial problem generation*, i.e., exploring the weaknesses of planning algorithms by generating problems that are challenging for a particular planner.

Acknowledgements

This work has been partially funded by the Grant PID2022-142976OB-I00, funded by MICIU/AEI/ 10.13039/501100011033 and by "ERDF/EU", as well as the Andalusian Regional predoctoral grant no. 21-111- PREDOC-0039 and by "ESF Investing in your future".

We want to express our deep gratitude to Masataro Asai, for his suggestion to use NLMs in our work; Simon Stahlberg, for providing the implementation of Graph Neural Networks used in a previous version of this work; Mauro Vallati and the rest of authors of [5], in addition to Sergio Jimenez Celorrio, for their advice on how to measure problem difficulty; Jiayuan Mao and the rest of authors of [4], for their helpful advice on NLMs; and, finally, Christian Muise and the FastDownward (FD) community, for their invaluable help on the use of the FD planning system.

References

[1] P. Barceló, E. V. Kostylev, M. Monet, J. Pérez, J. Reutter, and J.-P. Silva. The logical expressiveness of graph neural networks. In *8th International Conference on Learning Representations (ICLR 2020)*, 2020.

[2] B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

[3] E. Cohen and J. C. Beck. Problem difficulty and the phase transition in heuristic search. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[4] H. Dong, J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou. Neural logic machines. *arXiv preprint arXiv:1904.11694*, 2019.

[5] C. Fawcett, M. Vallati, F. Hutter, J. Hoffmann, H. H. Hoos, and K. Leyton-Brown. Improved features for runtime prediction of domain-independent planners. In *Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014.

[6] A. Fern, S. W. Yoon, and R. Givan. Learning domain-specific control knowledge from random walks. In *ICAPS*, pages 191–199, 2004.

[7] M. Fox and D. Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[8] R. Fuentetaja and T. De la Rosa. A planning-based approach for generating planning problems. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[9] M. Ghallab, D. Nau, and P. Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[10] P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise. An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187, 2019.

[11] M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[12] J. Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22 (3):57–57, 2001.

[13] C. Hogg, H. Munoz-Avila, and U. Kuter. Htn-maker: Learning htns with minimal additional knowledge engineering required. In *AAAI*, pages 950–956, 2008.

[14] M. Katz and S. Sohrabi. Generating data in planning: Sas planning tasks of a given causal structure. *HSDIP 2020*, page 41, 2020.

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

[16] O. Marom and B. Rosman. Utilising uncertainty for efficient learning of likely-admissible heuristics. In *ICAPS*, volume 30, pages 560–568, 2020.

[17] T. L. McCluskey, T. S. Vaquero, and M. Vallati. Engineering knowledge for automated planning: Towards a notion of quality. In *Proceedings of the Knowledge Capture Conference*, pages 1–8, 2017.

[18] C. Núñez-Molina, P. Mesejo, and J. Fernández-Olivares. NeSIG: A neuro-symbolic method for learning to generate planning problems. *arXiv preprint*, 2024. Full version of this paper.

[19] C. Núñez-Molina, P. Mesejo, and J. Fernández-Olivares. Code and data for "NeSIG: A neuro-symbolic method for learning to generate planning problems". GitHub, 2024. URL https://github.com/ari-dasci/S-PlanningProblemGeneration.

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

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

[22] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

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

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

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

[26] W. M. Shen and H. A. Simon. Rule creation and rule learning through environmental exploration. In *IJCAI*, pages 675–680. Morgan Kaufmann, 1989.

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

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

[29] M. Vallati, L. Chrpa, M. Grześ, T. L. McCluskey, M. Roberts, S. Sanner, et al. The 2014 international planning competition: Progress and trends. *Ai Magazine*, 36(3):90–98, 2015.