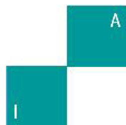
Inteligencia Artificial, 27(74) (2024), 227-242
doi: 10.4114/intartif.vol27iss74pp227-242



INTELIGENCIA ARTIFICIAL

http://journal.iberamia.org/

Automated feature extraction for planning state representation

Oscar Sapena[1,A], Eva Onaindia[1], Eliseo Marzal[1]

[1] Valencian Research Artificial Intelligence Institute, Universitat Politècnica de València, Camino de Vera s/n, Valencia, 46022, Spain.
[A] ossaver@upv.es

Abstract Deep learning methods have recently emerged as a mechanism for generating embeddings of planning states without the need to predefine feature spaces. In this work, we advocate for an automated, cost-effective and interpretable approach to extract representative features of planning states from high-level language. We present a technique that builds up on the objects type and yields a generalization over an entire planning domain, enabling to encode numerical state and goal information of individual planning tasks. The proposed representation is then evaluated in a task for learning heuristic functions for particular domains. A comparative analysis with one of the best current sequential planner and a recent ML-based approach demonstrate the efficacy of our method in improving planner performance.

Resumen Los métodos de aprendizaje profundo han surgido recientemente como un mecanismo para generar embeddings de estados de planificación sin la necesidad de predefinir espacios de características. En este trabajo, abogamos por un enfoque automatizado, eficiente en costes e interpretable para extraer características representativas de los estados de planificación a partir de un lenguaje de alto nivel. Presentamos una técnica que se basa en los tipos de objetos y permite una generalización sobre todo un dominio de planificación, posibilitando la codificación de información numérica del estado y de los objetivos de tareas de planificación individuales. La representación propuesta se evalúa mediante una tarea de aprendizaje de funciones heurísticas para dominios específicos. Un análisis comparativo con uno de los mejores planificadores secuenciales actuales y con un enfoque reciente basado en aprendizaje automático demuestra la eficacia de nuestro método para mejorar el rendimiento de los planificadores.

Keywords: AI planning, Feature extraction, State representation, Neural Networks, Machine Learning, Heuristic functions.

Palabras Clave: Planificación en IA, Extracción de características, Representación de estados, Redes Neuronales, Aprendizaje Automático, Funciones heurísticas.

1. Introduction

Automated Planning is the deliberation process devoted to selecting and organizing actions with the aim to accomplish an objective [1]. One common strategy for solving planning tasks is to apply forward heuristic search in a state space where nodes represent planning states and the algorithm finds a sequence of actions starting from the initial state of the task and ending in a final state that satisfies the goals. Heuristic search algorithms use a function $h(S)$ to estimate the cost-to-go of an optimal path that reaches a goal state from the planning state S. This paper presents an approach to automatically learn an efficient heuristic function for a given planning domain.

ISSN: 1137-3601 (print), 1988-3064 (on-line)
©IBERAMIA and the authors

Heuristic search is a powerful and robust technique to solve planning tasks as stated in the results of the International Planning Competitions[1] (IPC), which has led to very significant contributions in the design of domain-independent heuristics for planning over the past years. The most successful heuristic functions for planning rely on the delete-relaxation of the planning task, which consists in ignoring the delete effects of actions, and this amounts to assuming that once a variable takes a certain value, that value can always be used to satisfy preconditions of actions or goals. Based on the delete relaxation, various heuristics have been designed such as the additive heuristic h^{add}, and the max heuristic h^{max} [2], as well as the successful Fast Forward heuristic h^{FF} [3]. There also exist numerous heuristics based on ideas other than delete relaxation like the causal graph heuristic h^{CG} [4]; the h^m heuristic family based on critical paths [5]; or landmark-based heuristics like h^{LM-CUT} [6, 7, 8].

With the tremendous success in the area of supervised Machine Learning (ML) achieved with Neural Networks (NN), an interest in learning heuristic functions with NN has recently emerged. The NN input for learning planning heuristic functions is a set of training samples of the form (S, h) where S is some kind of representation of a planning state and h is the estimated distance (cost-to-go) from S to the goal state of a planning task, typically measured in the number of actions. Hence, the elements involved in the construction of the sample set are a planning domain, which defines a set of predicates and operators; a planning task, which describes the initial state and the goal condition; and the state search space of the task that results from the successive application of the domain operators in the initial state of the task until the goal condition is reached. Additionally, an ad-hoc solver that provides the value h for each state S is needed.

Depending on the generalization capabilities of the learned heuristic, three categories can be distinguished [9] :

1. **Instance-specific** (IS) heuristics are specifically tailored to solving one instance of a domain, i.e., a particular planning task. They only generalize over the state space of that task. Since the predicate symbols of the domain and the object symbols of the task are known, the set of facts that define a planning state is fully determined by the elements of the task. Therefore, any planning state S can be represented by a fixed number of variables.

2. **Domain-specific** (DS) heuristics aim to generalize over instances of the same domain, so they generalize over states, goals, and object sets. In this case, the learning algorithm needs to handle problems of different sizes. Consequently, unlike the IS heuristics, we cannot use the facts that define a planning state as its representation in the sample set. This leads to the need of creating state abstractions so that the planning states of all of the tasks are encoded under a single feature representation.

3. **Domain-independent** (DI) heuristics. Under this setting, we have various domains and a varying number of training tasks for each domain. Addressing this type of heuristics requires a general-purpose feature extractor and powerful ML models with huge expressive power.

There are hardly any investigations on the learning of DI heuristics with neural methods, except for a technique that uses Hypergraph Networks [10] or Graph Neural Networks (GNNs) [11]. These works, however, show limited success in DI experiments and more competitive performance in domain-dependent scenarios.

Most works fall within the category of IS heuristics [12, 13, 14, 15] where planning states are directly represented using the raw SAS+ encoding state vectors [16]; i.e., using multi-valued state variables, and no feature extraction is needed. In these works, the focus is primarily on analyzing the performance of different NN architecture hyperparameters and sampling techniques in the training data generation [17].

Our contribution focuses on the design of DS heuristic functions. We will apply them to different domains and show that they can be of great help to improve the performance of current planners in many cases. The paper is structured as follows: the next section is devoted to presenting the related work on different approaches that address DS heuristics; section 3 formalizes a planning task; section 4 describes the proposed technique for feature extraction and section 5 the numerical representation of the states. Section 6 shows the strategy followed for learning the heuristic function for a given domain

[1] `https://www.icaps-conference.org/competitions/`

and the section 7 presents the experimental results. The last section presents the conclusions and future work.

2. Related Work

This section describes approaches that learn domain-specific heuristics, also referred to as *per-domain* heuristics in the literature. The principal idea in the design of DS heuristics is to train the NN with small-size instances and learn a function that generalizes to larger instances of the domain. This line of research falls within what is known as Generalized Planning and, more specifically, Learning of Generalized Heuristics. In this work, we will focus on methods that specifically learn heuristic functions via neural methods, as they have shown superior performance over other techniques. Therefore, a suitable planning state encoding that is representative of any instance within a domain and that can be used as the NN input is needed.

Broadly speaking, a representative set of features of a planning state can be obtained with any of these two trends: either (1) using deep learning techniques that automatically extract structure from high-level data or (2) using automated feature engineering methods that exploit the domain structure. Although there is not yet a standard neural network that suits well for planning problems, the approaches that fall within trend (1) are mostly based on graph convolution to learn state embeddings like Graph-Neural Networks [18, 10, 19] or Neuro-Logical Machines [20]. Overall, all these cited approaches show to be competitive with baseline models using planning heuristics or state-of-the-art implementations of classical planners in terms of success rate (number of solved problems). There are rare indications though of the quality of the obtained plans with the exception of the heuristic h^{HTG} [10], which reports small deviations from the optimal solutions. More interestingly, some works report the inability of NN-based heuristics to outperform classical planning engines in transport-like domains wherein a tight coupling between the different objects of a planning task exists [18, 19, 20].

The second trend comprises approaches that apply feature engineering methods to extract features from planning states. Most symbolic approximations for computing general policies work on abstractions of the planning states in order to derive patterns. In this line, it has been proven that is possible to identify a set of *boolean* and *numerical features* for a generalized planning problem that encompasses a family of planning instances [21]. Subsequently, some authors found that numerical features, which represent state features that capture the number of objects satisfying a certain first-order logic property in a state, generalize naturally to a whole domain [22]. The concept language that is used in [22] to build the features allows expressing objects and relations between objects out of the domain predicates but a relation between objects that is implicitly encoded (e.g., in the available actions) is not expressible. The D2L approach [23] uses the domain description, a set of planning problems and their solutions, and computes a set of common features representative of the states. D2L obtains a pool of expressive features but its time-consuming algorithm allows training only on small amounts of data and does not generalize well to large-size problems.

An interesting approach presented in [9] proposes creating an *object graph* from a pair (initial-state, goal-state) and then extracting a fixed domain-specific number of features as the number of occurrences of certain subgraphs in the object graph. This approach achieves good results in success rate and plan length but it is only tested in two domains of the IPC. There also exist approaches that learn generalized heuristics in the absence of symbolic action models (planning domains) using only an input predicate vocabulary [24]. This work creates a canonical abstraction that represents a concrete state using only abstraction predicates without information about object names.

3. Background

Our approach builds upon the elements that make up the PDDL specification of a planning domain [25, 26]. Specifically, we define a domain Δ as a tuple $\langle T, P, OP \rangle$:

- A set T that comprises all the types specified in the domain such that every object of a planning problem belongs to one type of T.

- A set of *predicates* P; each predicate $p \in P$ has a name and a list of arguments $arg(p)$. We represent predicates in the form $name(a_1, a_2, \ldots, a_n)$, $n \geq 0$, where each $a_i \in arg(p)$ is a variable of a given type in T.
- A set of operators or action schemes OP. Each operator $op \in OP$ is a tuple containing the following elements:
  - $name(op)$: operator name.
  - $arg(op)$: list of arguments, i.e., a typified list of variables.
  - $pre(op)$, $add(op)$ and $del(op)$: sets of predicates that represent the preconditions, positive effects and negative effects of the operator, respectively. All the arguments of these predicates correspond to operator arguments.

A problem Π of a domain Δ is a tuple $\langle O, I, G\rangle$, where:

- O is a set of objects. Each object $o \in O$ belongs to a concrete type in T.
- I is a state representing the initial situation. Any state is a set of facts, that is, predicates where each argument is instantiated by an object of a compatible type.
- G is the set of facts representing the problem goal.

The starting point of our proposal is the calculation of the possible stages for each type in T. It is a static analysis that is carried out from the domain definition and that is valid for any problem of the domain. This technique is described in Section 4. In Section 5 we describe how the numerical representation of a state is obtained based on the calculated stages.

4. Stages of object types

The stages of an object type are conceptually similar to the *state invariants* [27, 28, 29], which are logical formulas that denote a common property that must be true in all the states of a domain. The stages of a type is a mutually exclusive and exhaustive set of feature sets that provide a full representation of any object of such type in a state.

Definition 1 (Features of a type) *Given a domain* $\Delta = \langle T, P, OP\rangle$*, we define* $features(t)$ *as the set of features of each type* $t \in T$*. A feature of a type* t *is a predicate* p *where one of its arguments* $a_i \in \arg(p)$ *is a variable of type* t*.*

Features are represented as predicates showing only the variable of the argument of type t the feature refers to, and representing the rest of the predicate arguments with the symbol $*$. In the Blocksworld domain [30], for example, the *block* type has the following features: {*clear(x), holding(x), ontable(x), on(x, *), on(*, x)*}.

Our method for extracting the stages of objects of a type t follows the next steps:

1. Construction of a transition graph to classify the $features(t)$.
2. Discover mutex relationships in $features(t)$.
3. Construction of the **basic stages** of type t by combining features that are not pairwise mutex.
4. Combination of basic stages among different types to generate a more comprehensive set of stages, which we call **extended stages**.

All these steps are described in detail in the following subsections.

4.1. Transition graph and feature classification

Inspired in TIM (The Type Inference Module) [27], we construct the transition rules for each typed argument of the action schemes in OP. As a result, for each type $t \in T$, we obtain a set of transition rules over *features(t)*.

Given an operator $op \in OP$ with an argument of type t, the corresponding transition rule has the form $del'(op) \longrightarrow add'(op)$, where $del'(op)$ (or $add'(op)$) is the subset of negative (or positive) effects of *op* which contain that argument. Transition rules represent, then, the features that an object of type t needs to lose in order to gain new ones.

The following are the transition rules associated with each Blocksworld domain operator for the *block* type:

`pickup(x)`: {*clear(x), ontable(x)*} $\longrightarrow$ {*holding(x)*}
`putdown(x)`: {*holding(x)*} $\longrightarrow$ {*clear(x), ontable(x)*}
`unstack(x, *)`: {*on(x, *), clear(x)*} $\longrightarrow$ {*holding(x)*}
`unstack(*, x)`: {*on(*, x)*} $\longrightarrow$ {*clear(x)*}
`stack(x, *)`: {*holding(x)*} $\longrightarrow$ {*on(x, *), clear(x)*}
`stack(*, x)`: {*clear(x)*} $\longrightarrow$ {*on(*, x)*}

We represent the transition rules in a directed graph where the nodes represent features and the edges represent the transitions. Figure 1 shows the transition graph for the *block* type.

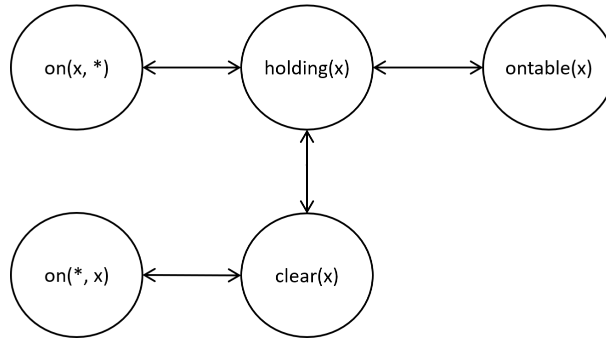


Figure 1: Transition graph for the *block* type in the Blocksworld domain.

It is possible that the left (or right) part of a transition rule is an empty set, meaning that one or more features can be obtained (or lost) without the need to lose (or gain) others. For example, in the Driverlog[2] domain, we have the following transition rule for type *location*: {} $\longrightarrow$ {*at-truck(*, x)*}. This means that a truck can arrive at location x without x losing any feature. We represent these empty sets by a node named *null* in the transition graph (see Figure 2).

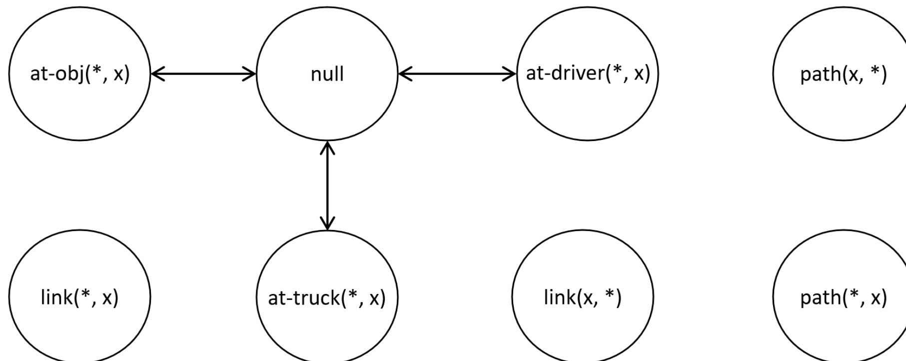


Figure 2: Transition graph for the *location* type in the Driverlog domain.

[2]https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume20/long03a-html/JAIRIPC.html

Once the transition graph for a type t is built, we classify *features(t)* into:

- *static(t)*: features that are never acquired or lost. They are defined in the initial state of the problem and cannot be modified through actions. These features are easily detected in the transition graph since they are isolated nodes. This is the case of the features *link(x, *)*, *link(*, x)*, *path(x, *)* and *path(*, x)* in Figure 2. They represent existing roads and paths in the problem and there are no actions to remove them or build new ones.

- *attributes(t)*: features that do not involve any feature when they are acquired or lost. They can be easily identified in the transition graph because they are nodes whose only adjacent is the *null* node. For example, *at-obj(*, x)*, *at-driver(*, x)* and *at-truck(*, x)* are attributes of the *location* type in Driverlog.

- *properties(t)*: features that, when acquired or lost, involve the acquisition or loss of another feature. These, in turn, are divided into the following categories:

  - *permanent(t)*: features that only involve itself when they are acquired or lost. They are isolated nodes in the transition graph with a self-loop. In Figure 3 it can be seen that *at(x, *)* and *available(x)* are permanent properties for type *rover* in the Rovers[3] domain. A rover, for example, can move between waypoints, but it is always in one of them.
  - *multiple(t)*: multiple instances of these properties can coexist in the same state. They can be identified by a path in the transition graph that goes from the *null* node to that property. *have-rock-analysis(x, *)*, *have-soil-analysis(x, *)*, *calibrated(*, x)* and *have-image(x, *)* are multiple properties for a rover. For example, a rover can have multiple images from different objectives and be taken with different modes at a certain moment.
  - *transient(t)*: properties that, when they are lost, cannot be acquired again. They are nodes that are not part of any cycle[4] in the transition graph and that are not reachable from the *null* node. In Figure 4, nodes *at-soil-sample(x)*, *have-soil-analysis(*, x)*, *at-rock-sample(x)* and *have-rock-analysis(*, x)* are transient properties. Note that, for example, when a soil sample is taken from a waypoint, that sample can no longer be taken from that place.
  - *reversible(t)*: properties that can be obtained cyclically. They are nodes in the transition graph that belong to a cycle[4] and that is not reachable from the *null* node. All properties of the *block* type are reversible (Figure 1).

4.2. Mutex features

The next step is to discover features that cannot coexist in the same state. We define mutex as a pair of properties that are mutually exclusive. In most approaches [27, 28, 29], the calculation is problem-dependent and is subject to the particular objects of the initial state of a concrete problem. However, some works focus on extracting mutex solely from the domain description [31, 32]. We use the inference algorithm of this last work to obtain the set of pairwise mutex features in a given domain.

From the set of features of a given type t, *static(t)*, *attributes(t)* and *permanent(t)* cannot be mutex since they do not interact with other features. Neither the *multiple(t)*, because many instances of these properties can coexist at the same time. Therefore, only *transient(t)* and *reversible(t)* properties can be pairwise mutex.

For the *block* type in the Blocksworld domain (Figure 1) we obtain six mutex relationships between reversible properties: {(*on(x,*)*, *holding(x)*), (*ontable(x)*, *holding(x)*), (*on(x,*)*, *ontable(x)*), (*on(*,x)*, *holding(x)*), (*clear(x)*, *holding(x)*), (*on(*,x)*, *clear(x)*)}. For the *location* type in the Driverlog domain (Figure 2) and for the *rover* type in the Rovers domain (Figure 3), we don't get any mutex. For the *waypoint* type in the Rovers domain, we find two mutex relationships between transient properties: {(*at-soil-sample(x)*, *have-soil-analysis(*,x)*), (*at-rock-sample(x)*, *have-rock-analysis(*,x)*)}

[3] https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume20/long03a-html/JAIRIPC.html
[4] We only consider cycles of length greater than 1, so self-loops are not taken into account here.

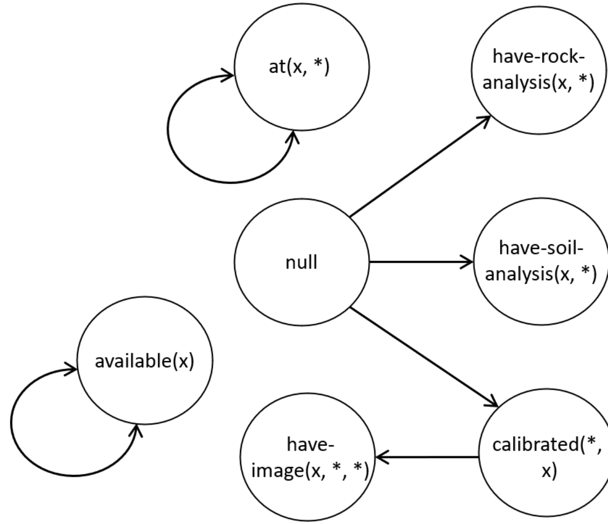


Figure 3: Transition graph for the *rover* type in the Rovers domain. For simplicity, in the examples of Rovers domain, we omit the six static features of the *rover*: *can-traverse(x, *, *)*, *equipped-for-soil-analysis(x)*, *equipped-for-rock-analysis(x)*, *equipped-for-imaging(x)*, *store-of(*, x)* and *on-board(*, x)*.

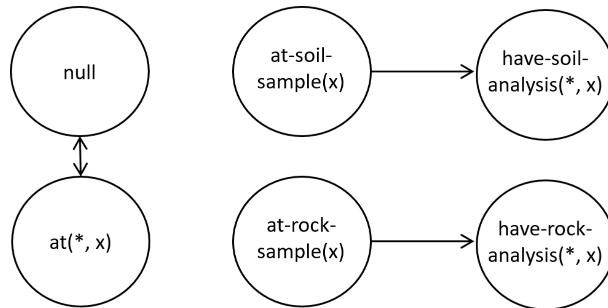


Figure 4: Transition graph for the *waypoint* type in the Rovers domain.

4.3. Basic stages

This section is devoted to presenting the formalization and computation of the basic stages of an object type.

A stage of an object type t is a set of pairwise non-mutex features from $features(t)$ that unambiguously define one situation wherein an object of type t can be in any state. Let $s = \{f_1, \ldots, f_n\}$ be a stage of a type t where $f_i \in features(t)$. By substituting the argument of type t in each feature f_i by a particular object x of type t, the set s perfectly describes a possible situation of x in any state.

For example, consider the stage $s = \{ontable(x), on(*, x)\}$ associated with the type *block* in the *blocksworld* domain. This stage expresses that an object x of type block is on the table and has a block on top. These two properties are sufficient to unmistakably define one situation in which an object of type block can be found.

Definition 2 (Basic stages of an object type t**)** *Given a domain* $\Delta = \langle T, P, OP \rangle$*, the set of basic stages of a type* $t \in T$*, stages(t), is the set of all non-mutex combinations of features from features(t) considering that:*

- *Permanent properties hold on every stage.*
- *Transient and reversible properties must be always included in a stage if they are not mutex with the rest of features in the stage.*

For example, in Blocksworld domain we cannot have a stage that includes only the property *ontable(x)*: features that do not appear in a stage are assumed not to hold (negation by failure). Therefore, no block will ever be in that stage, since a block on the table must be clear or have another block on top of it. According to these constraints, the basic stages of a block are the following ones:

stages(block) = {{*on(*, x)*, *ontable(x)*}, {*on(*, x)*, *on(x, *)*}, {*clear(x)*, *ontable(x)*}, {*clear(x)*, *on(x, *)*}, {*holding(x)*}}

For type rover in the Rovers domain (see Figure 3), we only have permanent and multiple properties. Permanent ones, *at(x, *)* and *available(x)*, are included in all the stages while, with multiples, we do all possible combinations (including them or not in the stages).

4.3.1. Extended stages

The stages of an object x of type t represent a set of possible situations in which x can be. But they do not link this information with that of other objects (of the same or different type). For example, in the Blocksworld domain, one block x can be in the stage {*clear(x)*, *on(x, *)*}, i.e. clear and on another block (which we will call y). But we ignore if block y is on the table or, in turn, on another block.

Extended stages extend the basic stages by combining information about the stages of several objects, particularly those participating in the remaining arguments of the features. In this way, we increase the number of stages while representing possible situations with a higher level of detail.

Definition 3 (Set of extended stages of an object type t) *Given a domain $\Delta = \langle T, P, OP \rangle$, the set of extended stages of a type $t \in T$, stages'(t), is the set resulting from combining each stage $s \in stages(t)$ with every stage $s' \in stages(t')$, avoiding mutex features, where $t' \in T$ is the type of one ungrounded parameter of a feature in s.*

Let's take, for example, the stage s = {*clear(x)*, *on(x, *)*} in the Blocksworld domain. We can see that the second parameter of *on(x, *)* is not grounded. We ground it now with another block y: s = {*clear(x)*, *on(x, y)*}. We analyze then which stages of y are not mutex with s:

- {*on(*, y)*, *ontable(y)*}: this stage is not mutex with s, so we obtain the following extended stage: {*clear(x)*, *on(x, y)*, *ontable(y)*}. That is, block x is clear and on another block y which is on the table.

- {*on(*, y)*, *on(y, *)*}: this stage is also not mutex with s. Then, we obtain another extended stage: {*clear(x)*, *on(x, y)*, *on(y, *)*}. Here, a new ungrounded parameter appears. However, to avoid infinite recursion, we stop doing combinations when all the parameters of the features in s are already instantiated.

- {*clear(y)*, *ontable(y)*}: *clear(y)* is mutex with *on(x, y)*, so this stage is not valid to generate an extended stage.

- {*clear(y)*, *on(y, *)*}: same as above.

- {*holding(y)*}: this property is also mutex with *on(x, y)*.

This way, we obtain 10 extended stages from the 5 basic ones:

stages'(block) = {{*ontable(x)*, *on(y, x)*, *clear(y)*}, {*ontable(x)*, *on(y, x)*, *on(*, y)*}, {*on(y, x)*, *on(x, z)*, *clear(y)*, *ontable(z)*}, {*on(y, x)*, *on(x, z)*, *on(*, y)*, *ontable(z)*}, {*on(y, x)*, *on(x, z)*, *clear(y)*, *on(z, *)*}, {*on(y, x)*, *on(x, z)*, *on(*, y)*, *on(z, *)*}, {*clear(x)*, *ontable(x)*}, {*clear(x)*, *on(x, y)*, *ontable(y)*}, {*clear(x)*, *on(x, y)*, *on(y, *)*}, {*holding(x)*}}

5. Numerical state representation

The numerical representation of a state consists of counting, for each type, how many objects of that type are in each stage. This representation has the advantage that it has a fixed size for a given domain, regardless of the problem, equal to the sum of the number of stages (or extended stages, depending on our needs) of each type: $\sum_{t_i \in T} |stages'(t_i)|$. This makes it suitable to be used in machine learning techniques.

However, there are certain state characteristics that our formalism is not able to capture. While we can express that one block is on top of another block or that a rover is in a waypoint, we cannot identify the block or the waypoint. The identity of the objects is also partially lost when grouping in a single numerical value several objects that share the same stage.

5.1. Goal stages

A problem goal can also be represented numerically using our proposed technique, but it must be taken into account that the objectives only show partial information about a state. Therefore, an object could be in several different stages and fulfill the objective in all of them. For example, if we want a given block x to be on the table, and the problem goal does not specify if it should have another block on, then x can be in three different stages in the goal state: {*ontable(x)*, *on(y, x)*, *clear(y)*}, {*ontable(x)*, *on(y, x)*, *on(*, y)*} and {*clear(x)*, *ontable(x)*}. We solve this problem by defining the goal stages, which allow classifying the situation of an object in the goal state in a unique way:

Definition 4 (Goal stages of an object type t) *Given a domain $\Delta = \langle T, P, OP \rangle$, the set of goal stages of a type $t \in T$, goalStages(t), is any possible subset of $stages'(t)$, not including static features.*

This way, we will have a goal stage only with the property *ontable(x)* and we will classify x in this stage. Note that the empty set is also included in the goal stages, since an object may not appear in the goals. Static features are not included in these stages since they are never problem goals (cannot be modified through actions).

6. Self-learned heuristic based on the numerical state representation

We have designed a heuristic function which exploits the numerical state representation to model the problems. The objective is to demonstrate that this representation can be useful to improve the performance of current planners. This heuristic is computed through a neural network (NN) automatically trained with random problems.

6.1. NN architecture

We used a fully connected feed-forward NN and supervised learning for training. The configuration of hyperparameters that returned the best results follows this setup: one hidden layer with 256 neurons using a rectified linear unit (ReLU) activation function, and two dropout layers with probability of 10% (between the hidden layer and the input and output layers).

Input layer The input vector of our NN is a coding of the current state S (state being evaluated) and goals G of a given task $\Pi = \langle O, I, G \rangle$ of a domain $\Delta = \langle T, P, OP \rangle$. For each type $t \in T$, we generate all possible tuples (s_i, gs_j), where $s_i \in$ *stages'(t)* and $gs_j \in$ *goalStages(t)*. This way, we can represent information about the objects of type t that are in the stage s_i in the current state S and that must be in the stage gs_j in the goal state. For simplicity, we will call $O_{t,i,j} \in O$ the set of objects of type t, which are in stage s_i in S and must be in the goal stage sg_j. For each tuple (s_i, gs_j), we define three input neurons in the input layer:

- Number of objects in $O_{t,i,j}$ whose goals are satisfied in S (i.e. all the goals in which the object appears as a parameter are fulfilled in the current state).

- Number of objects in $O_{t,i,j}$ whose goals are not satisfied in S.
- A distance estimate to reach the goals of the objects in $O_{t,i,j}$ from S.

To obtain this last value, we use an admissible estimate: the makespan of an optimal parallel solution plan from S, calculated as the first level of the RPG (relaxed planning graph), where all goals of these objects are simultaneously achieved. This is known as the h_{max} heuristic [2]. This heuristic is not used in current planners, as it is not very informative. However, we are just interested in getting a measure that does not overestimate the cost of achieving the goals.

In some domains, the number of input neurons is quite small. We have addressed this problem by finding the object types with the fewest number of stages and dividing the objects of these types into several groups. The objects are randomly distributed among the groups. The size of the input layer can be calculated then as $3 * \sum_{t_i \in T} |stages'(t_i)| * |goalStages(t_i)| * numGroups(t_i)$, where $numGroups(t_i)$ is the number of groups into which we have divided the objects of type t_i.

Output layer The output layer represents the estimated distance (heuristic value) to reach the goals from a given state. This value is binary encoded: a value of h is represented by h ones followed by zeros. This representation has given us better results than using a single neuron to encode the integer value (or real if we normalize) or to set the neuron number h to one and leave the rest zeroes. For this layer, we used a sigmoid activation function. The size of this layer corresponds to the largest heuristic value obtained from the training samples.

6.2. Training

To generate training data for supervised learning in a given domain, we need a dataset of different problems and an algorithm for extracting pairs (S, h) from each problem, where S is a state (search node) and h is the distance from S to the goal.

We implemented a random problem generator for the tested domains. For obtaining the training samples, we designed an incremental algorithm that starts from simple problems and scales up to problems of a greater size. The varying sizes were chosen with the aim to handle problems of similar difficulty to those in the International Planning Competitions. This way, we defined 10 different sized problems per domain, increasing the number of objects linearly from one size to the next one.

The smallest size samples were obtained by generating 1000 problems that were then solved using BFS (Breadth First Search). These optimal solutions are easy to compute for such small problems. A solution plan $\pi = \{a_1, \ldots, a_n\}$ is a sequence of actions that leads from the initial to the goal state. From π, we can extract training samples from the sequence of states resulting of applying its actions to the initial state:

- $S_0 = I$, and its distance to the goal, h_0, is equal to n (number of actions in π).
- S_1 is the result of applying action a_1 to the previous state, S_0. Its distance is $h_1 = h_0 - 1$.
- And so on.

Once we have the samples for problems of size 1, we normalize the input and train a NN for that size (we will refer to this network as NN_1) using 20% of the samples for testing. We use the Adam optimizer on the cross-entropy loss function with a batch size of 256. The training process runs for 80 to 640 epochs, and we select the model that maximizes the accuracy on the validation data.

A trained NN_i network is then used as a heuristic function to generate the training samples of size $i+1$. Specifically, our planner nnPLAN uses a weighted A* search with evaluation function $f(S) = g(S)+2 \cdot h(S)$, where $h(S)$ is the value returned by the NN for state S.

With this method, we can progressively scale up to larger problem sizes. This approach has, however, a limitation: NN_i is not an admissible heuristic for problems of size $i+1$, which yields suboptimal training samples (suboptimal distances to the goal). Thus, a trained network NN_{i+1} will produce suboptimal plans.

7. Experimental evaluation

We have selected six classical planning domains to evaluate the performance of our heuristic function: Blocksworld, Driverlog, Floortile, Freecell, Rovers and Termes. They are all domains used in the IPCs and good representatives of the possible scenarios that we can find. Stages in Blocksworld and Freecell, for example, describe fairly accurately the search state as well as the goal to reach, maintaining information about interactions between the subgoals. Freecell has the added difficulty of dead-ends (goals are unreachable from certain states). On the other hand, stages in Driverlog, Floortile or Rovers include very few properties, so objects are similarly classified in very different tasks. In Termes domain, we obtain very few stages and goal stages (since most features are static). To equate the number of input neurons to those obtained in the other domains, we have divided the objects of types *numb* and *position* into 16 groups.

To evaluate the performance of our NN-based heuristic, we compared our planner, which we call nnPLAN, with:

- The sequential planner LAPKT-DUAL-BFWS [33], which is currently one of the best state-of-the-art planners. It was the runner-up of the Satisficing Track in the 2018 International Planning Competition and in the Sparkle Planning Challenge 2019.
- The GOOSE planner [11], using its domain-dependent heuristic based on GNNs (which offers better performance than the domain-independent one). This way we can compare ourselves with other heuristic learned automatically using machine learning techniques.
- Our planner, replacing the NN-based heuristic with h^{FF}. This setting, which we will call ffPLAN, allows comparing both heuristics on equal terms.

We generated 250 random problems (25 per size) for each domain. We have limited the time to one minute, to compare the quality of the first solutions, and to five minutes, to study how the quality improves over time. We run the tests in a i7-8750H 2,20 GHz CPU with 16 Gb. of memory, using only a single CPU core. No GPU has been used except for GOOSE, which is more computationally demanding. We have run it on a single NVIDIA GeForce RTX 3090 GPU.

The first step of nnPLAN is to check the size of the task, according to the number of defined objects. Then, for a problem of size i, nnPLAN uses NN_i as heuristic to guide the search. NN_i is the network trained with samples of size i, so this is the most adequate net to solve the problem at hand. We did run some tests to check the effects of using a NN of a size different than the problem size and we verified that using NN_i is also suitable for problems of a size smaller than i. However, it becomes less and less informative as the size of the problem grows over i. For example, NN_{10}, trained with problems of maximum 30 blocks, gives a good performance in *Blocksworld* until problems of 42 blocks, being unable to solve some tasks of 51 blocks in less than 15 minutes. Table 1 shows the maximum number of objects defined for each problem size.

Domain	Size 1	Size 10
Blocksworld	3 blocks	30 blocks
Driverlog	5 cities, 1 truck, 1 driver, 3 objects	50 cities, 10 trucks, 10 drivers, 30 objects
Freecell	3 cards/suit	12 cards/suit
Floortile	4x2 tiles, 1 robot	7x5 tiles, 3 robot
Rovers	1 rover, 3 waypoints	10 rovers, 30 waypoints
Termes	3x3 positions, 2 heights	5x5 positions, 4 heights

Table 1: Maximum number of objects for sizes 1 and 10 in each domain. This number grows roughly linearly from size 1 to 10.

Table 2 shows the coverage of the planners in the six domains. In one minute, our approach solves 1,380 out of 1,500 problems, increasing to 1,402 problems within five minutes. In comparison, using the h^{FF} heuristic in our planner results in solving 232 and 191 fewer problems, respectively. The performance of LAPKT-DUAL-BFWS is more comparable to nnPLAN, with a shortfall of 53 and 20 problems, respectively. GOOSE demonstrates lower overall performance, though it achieves similar coverage in specific domains such as Freecell, Rovers, and Termes.

As expected, nnPLAN performs very well on Blocksworld and Freecell, as the stages describe the state of the problem and its goals quite accurately. In Driverlog, on the other hand, we manage to solve fewer problems (with a one-minute deadline) than LAPKT-DUAL-BFWS. The Termes domain is a special case: the stages do not offer much information, but the division of the objects into groups is very helpful for the network. The positions are distributed in squares on a 2D board, so their division into groups allows each square to be better identified. This makes the performance in this domain also good.

Table 2: Percentage of solved problems in 1 and 5 minutes of a total of 250 problems per domain.

	nnPLAN		ffPLAN		LAPKT-DUAL		GOOSE	
Domain	1 min.	5 min.	1 min.	5 min.	1 min.	5 min.	1 min.	5 min.
Blocksw.	100%	100%	73.2%	81.2%	88.8%	98.8%	36.7%	46.8%
Driverlog	97.2%	100%	88%	93.2%	100%	100%	54.8%	60.8%
Floortile	60%	61.2%	59,6%	61,6%	53.2%	54.8%	11.2%	14.8%
Freecell	100%	100%	93.2%	97.6%	100%	100%	99.2%	99.6%
Rovers	100%	100%	99.2%	100%	100%	100%	94%	97.2%
Termes	94.8%	100%	46%	50.8%	89.2%	99.2%	85.2%	92.4%
Average	92%	93.5%	76.5%	80.7%	88.5%	92.1%	63.5%	68.6%

Regarding plan quality, in Figures 5, 6 and 7 we compare the average plan length (number of actions) of the solution plans of ffPLAN, LAPKT-DUAL-BFWS and GOOSE, respectively, in relation to nnPLAN. A value of 1 means both get plans of equal length. A value of 1.5, for example, implies that the planner being compared calculates plans 1.5 times longer than nnPLAN.

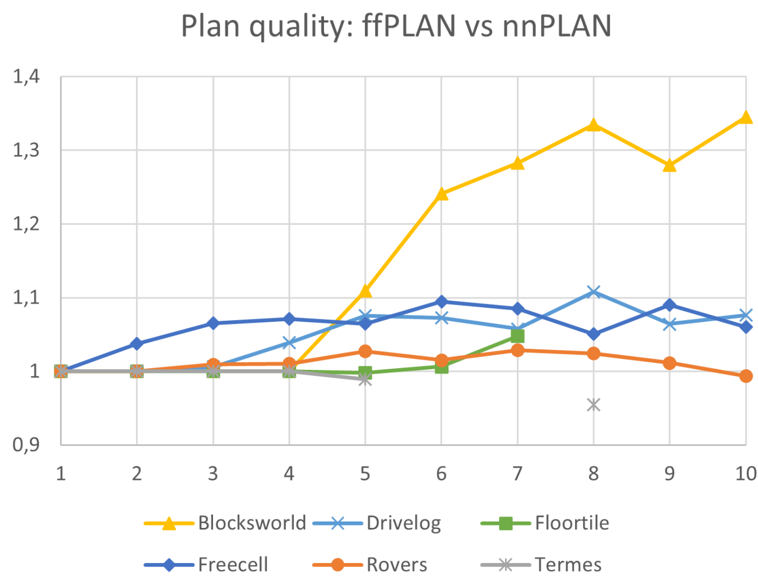


Figure 5: Quality of the plans generated by ffPLAN compared to nnPLAN for the 10 problem sizes defined in each of the domains within a deadline of 5 min.

As shown in Figure 5, the NN-based and the h^{FF} heuristics produce plans of comparable quality. While ffPLAN tends to generate longer plans, particularly in the Blocksworld domain, it shows a slight advantage in the Termes domain.

Regarding to LAPKT-DUAL-BFWS and nnPLAN, both produce solutions of similar quality for the smallest problems (until size 5). From there, the differences become noticeable:

- Blocksworld: this is the domain where nnPLAN stands out the most. LAPKT-DUAL-BFWS finds

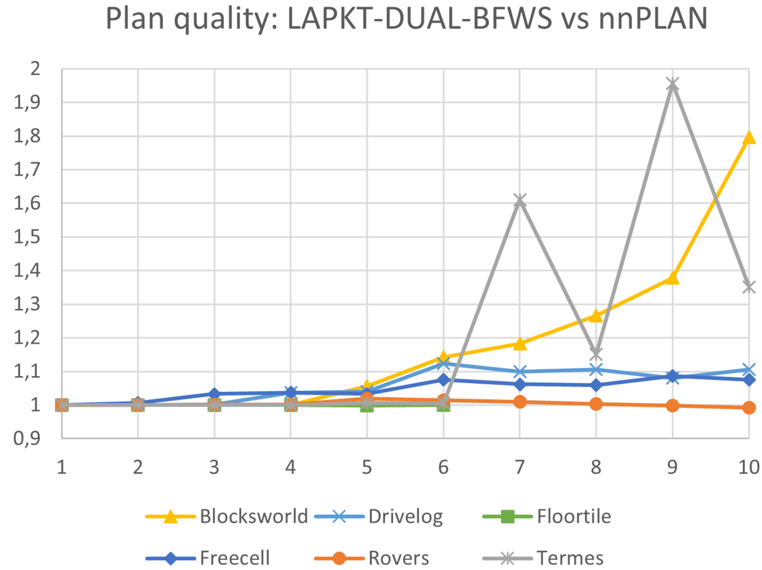


Figure 6: Quality of the plans generated by LAPKT-DUAL-BFWS compared to nnPLAN for the 10 problem sizes defined in each of the domains within a deadline of 5 min.

solutions 1.8 times longer with a five-minute time limit. And the difference increases as the size of the problem grows.

- Termes: quality also stands out in this domain, with LAPKT-DUAL-BFWS generating plans twice as long. But the sawtooth shape of the graph suggests that, for even problem sizes, the distribution in groups of the positions is not so effective.
- Driverlog: despite not being the most suitable domain to handle with our numerical representation, nnPLAN achieves slightly better quality plans.
- Freecell: this domain works very well in nnPLAN. But the improvements in quality are small since the numerous dead-ends prevent big differences in the quality of the solutions.
- Rovers: LAPKT-DUAL-BFWS is particularly efficient in this domain and produces slightly shorter plans (0.99%) than our approach.
- Floortile: the quality of both planners is practically identical in this domain, due both to the dead-ends and to the fact that hardly any problem beyond size 6 is solved.

Regarding GOOSE, although it is trained with optimal plans, the quality of the generated plans is not good, especially on the Blocksworld, Driverlog and Termes domains.

The results obtained in these domains reveal that our proposed numerical representation can be very useful to enhance the performance of current planners. However, preprocessing time must be considered, that includes sample generation and training a neural network for each problem size, and requires several hours of computation time.

8. Conclusions

This work presents a technique based on object types that yields a generalization over an entire planning domain and allows us to numerically encode state and goal information about individual planning tasks. We propose a novel mechanism to classify the features of the object types, unambiguously representing the possible situations of a given object among a collection of stages, both in the current state and in the goals.

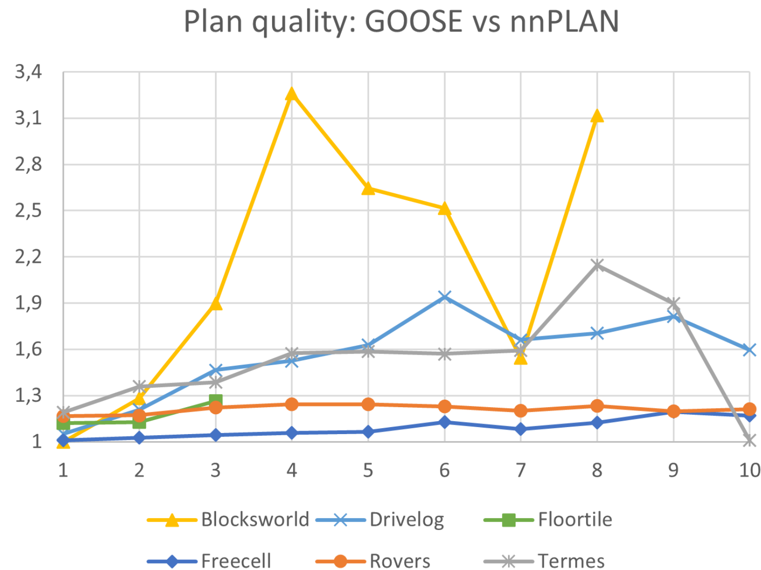


Figure 7: Quality of the plans generated by GOOSE compared to nnPLAN for the 10 problem sizes defined in each of the domains within a deadline of 5 min.

We also show an example of self-learned heuristic function, based on neural networks, which exploits this representation to model the problems. We implemented a new planner, nnPLAN, that uses this heuristic to guide an A* search toward a solution. Results show that nnPLAN outperforms one of the best current sequential planners, LAPKT-DUAL-BFWS, in six tested domains. They also show that the NN-based heuristic solves many more problems and generates shorter plans in these domains than the successful Fast Forward heuristic h^{FF}. This reveals that the proposed feature extraction technique can be very useful to enhance the performance of current planners. It is not only applicable to the development of new heuristics, but it can also be interesting for the preprocessing, classification, and analysis of the characteristics of the problems.

Our future work is focused on analyzing further features that better represent the situation of an object in a given state and extending the proposed representation to include numeric variables and temporal actions.

Acknowledgements

This work has been partially supported by the project I+D+i AEI PID2021-127647NB-C22 funded by MICIU/AEI/10.13039/501100011033 and by FEDER, UE.

References

[1] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier, 2004. `doi:10.1016/B978-1-55860-856-6.X5000-5`.

[2] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001. `doi:10.1016/S0004-3702(01)00108-4`.

[3] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res.*, 14:253–302, 2001. `doi:10.1613/jair.855`.

[4] Malte Helmert. A planning heuristic based on causal graph analysis. In *International Conference on Automated Planning and Scheduling*, pages 161–170, 2004.

[5] Patrik Haslum, Blai Bonet, and Hector Geffner. New admissible heuristics for domain-independent planning. In *AAAI*, pages 1163–1168. The MIT Press, 2005.

[6] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In Dieter Fox and Carla P. Gomes, editors, *AAAI Conference on Artificial Intelligence*, pages 975–982, 2008.

[7] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*. AAAI, 2009.

[8] Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In Craig Boutilier, editor, *IJCAI*, pages 1728–1733, 2009.

[9] Otakar Trunda and Roman Barták. Deep Learning of Heuristics for Domain-independent Planning. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *ICAART*, pages 79–88. SCITEPRESS, 2020.

[10] William Shen, Felipe W. Trevizan, and Sylvie Thiébaux. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *ICAPS*, pages 574–584, 2020.

[11] Dillon Z. Chen, Sylvie Thiébaux, and Felipe Trevizan. Learning Domain-Independent Heuristics for Grounded and Lifted Planning. In *ICAPS*, 2024.

[12] Patrick Ferber, Malte Helmert, and Jörg Hoffmann. Neural network heuristics for classical planning: A study of hyperparameter space. In *ECAI*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2346–2353. IOS Press, 2020.

[13] Liu Yu, Ryo Kuroiwa, and Alex S. Fukunaga. Learning Search-Space Specific Heuristics Using Neural Network. In *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, pages 1–8, 2020.

[14] Patrick Ferber, Florian Geißer, Felipe W. Trevizan, Malte Helmert, and Jörg Hoffmann. Neural Network Heuristic Functions for Classical Planning: Bootstrapping and Comparison to Other Methods. In *ICAPS*, pages 583–587, 2022.

[15] Stefan O'Toole, Miquel Ramírez, Nir Lipovetzky, and Adrian R. Pearce. Sampling from Pre-Images to Learn Heuristic Functions for Classical Planning (Extended Abstract). In *15th International Symposium on Combinatorial Search, SOCS*, pages 308–310, 2022.

[16] Christer Bäckström and Bernhard Nebel. Complexity Results for SAS+ Planning. *Comput. Intell.*, 11:625–656, 1995.

[17] Rafael Vales Bettker, Pedro Minini, André G. Pereira, and Marcus Ritt. Understanding Sample Generation Strategies for Learning Heuristic Functions in Classical Planning. *J. Artif. Intell. Res.*, 80:243–271, 2024. URL: `https://doi.org/10.1613/jair.1.15742`, `doi:10.1613/JAIR.1.15742`.

[18] O. Rivlin, T. Hazan, and E. Karpas. Generalized planning with deep reinforcement learning. In *Bridging the Gap Between AI Planning and Reinforcement Learning (PRL @ ICAPS) – Workshop at ICAPS 2022*, 2020. `doi:10.48550/arXiv.2005.02305`.

[19] Simon Ståhlberg, Blai Bonet, and Hector Geffner. Learning generalized policies without supervision using gnns. In *International Conference on Principles of Knowledge Representation and Reasoning KR 2022*, 2022.

[20] Clement Gehring, Masataro Asai, Rohan Chitnis, Tom Silver, Leslie Pack Kaelbling, Shirin Sohrabi, and Michael Katz. Reinforcement Learning for Classical Planning: Viewing Heuristics as Dense Reward Generators. In *ICAPS 2022, Singapore*, pages 588–596. AAAI Press, 2022.

[21] Blai Bonet and Hector Geffner. Features, projections, and representation change for generalized planning. In Jérôme Lang, editor, *IJCAI*, pages 4667–4673, 2018.

[22] Guillem Francès, Augusto B. Corrêa, Cedric Geissmann, and Florian Pommerening. Generalized potential heuristics for classical planning. In *IJCAI 2019*, pages 5554–5561, 2019.

[23] Guillem Francès, Blai Bonet, and Hector Geffner. Learning General Planning Policies from Small Examples Without Supervision. In *AAAI*, pages 11801–11808, 2021.

[24] Rushang Karia and Siddharth Srivastava. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *AAAI 2021*, pages 8064–8073. AAAI Press, 2021.

[25] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language, 1998.

[26] Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.*, 20:61–124, 2003. URL: `https://doi.org/10.1613/jair.1129`, `doi:10.1613/JAIR.1129`.

[27] Maria Fox and Derek Long. The Automatic Inference of State Invariants in TIM. *J. Artif. Intell. Res.*, 9:367–421, 1998. URL: `https://doi.org/10.1613/jair.544`, `doi:10.1613/JAIR.544`.

[28] Alfonso Gerevini and Lenhart K. Schubert. Discovering state constraints in DISCOPLAN: some new results. In *AAAI*, pages 761–767, 2000.

[29] Jussi Rintanen. Schematic invariants by reduction to ground invariants. In *AAAI Conference on Artificial Intelligence*, pages 3644–3650, 2017.

[30] John K. Slaney and Sylvie Thiébaux. Blocks World revisited. *Artif. Intell.*, 125(1-2):119–153, 2001. `doi:10.1016/S0004-3702(00)00079-5`.

[31] Malte Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009. `doi:10.1016/j.artint.2008.10.013`.

[32] Daniel Fišer. Lifted fact-alternating mutex groups and pruned grounding of classical planning problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(06):9835–9842, 2020. `doi:10.1609/aaai.v34i06.6536`.

[33] N. Lipovetzky and H. Geffner. A polynomial planning algorithm that beats LAMA and FF. In *ICAPS*, volume 27(1), pages 195–199, 2017.