Using Static Graphs in Planning Domains to Understand Domain Dynamics

Wickler, G. (2013). Using Static Graphs in Planning Domains to Understand Domain Dynamics. In Proceedings of Knowledge Engineering for Planning and Scheduling (KEPS 2013), Part of 23rd International Conference on Autmated Planning and Scheduling (ICAPS) Rome, Italy, June 2013.

Using Static Graphs in Planning Domains to Understand Domain Dynamics*

Gerhard Wickler
Artificial Intelligence Applications Institute
University of Edinburgh
Edinburgh, Scotland

Abstract

This paper describes a method for analyzing STRIPS-like planning domains by identifying static graphs that are implicit in the set of operators defining a planning domain. A graph consisting of nodes and possibly directed edges is a common way to construct representations for many problems, including computational problems and problems of reasoning about action. Furthermore, there may be objects or properties related to the nodes of such a graph that may be modified by the operators in ways restricted by the graph. The formal definition of *shift operators* over *static graphs* as a domain feature is the main contribution of this paper.

Such an analysis could be used for verification and validation of the planning domain when it is presented to the domain author who may or may not agree with the result of the analysis. The method described relies on domain features that can also be extracted automatically, and it works on domains rather than problems, which means the result is problem-independent. However, if problems are given further analysis may be perfomed. The method has been evaluated using a small number of planning domains drawn from the international planning competition.

Introduction

Specifying a planning domain and a planning problem in a formal description language defines a search space that can be traversed by a state-space planner to find a solution plan. It is well known that this specification process, also known as *problem formulation* [Russell and Norvig, 2003], is essential for enabling efficient problem-solving though search [Amarel, 1968]. However, the most efficient representation is often hard to understand, verify and maintain. One way to ensure the correctness of a problem specification is to enforce consistency. Obviously this does not guarantee correctness, but it may highlight problems to the knowledge engineer.

Consistency can be enforced if the representation contains some redundancy. We have described a set of domain features [Wickler, 2011] that can be used to assist during the verification and validation of planning domains by exploiting information implicit in the planning domain. The features are: *domain types*, *relation fluency*, *inconsistent effects* and *reversible actions*. These features can be efficiently and automatically extracted from a planning domain. If the planning domain also contains an explicit specification of these features then these represent redundant information that can be compared and used to enforce consistency, by which we mean that the feature values specified by the knowledge engineer should be the same as the ones that can be automatically extracted. Hopefully, this will lead to a planning domain that is in line with what the knowledge engineer intended to represent.

*This work has been sponsored by the Engineering and Physical Sciences Research Council (UK) under grant number EP/J011800/1. The University of Edinburgh and research sponsors are authorized to reproduce and distribute reprints and online copies for their purposes notwithstanding any copyright annotation hereon.

Knowledge Engineering

Knowledge engineering (KE) for planning domains is a topic that has received relatively little attention in the AI planning community and much work is still needed in the area. However, with planners becoming more efficient, the problems they can solve in reasonable time are becoming larger, and so the planning domains may be larger and more complex to engineer.

KE Methodology

The design process for planning domain models is similar to the knowledge engineering process followed for other types of software models. The baseline phases for planning domains described in [Vaquero *et al.*, 2011] are the following:

1. requirements specification
2. knowledge modeling
3. model analysis
4. deployment to planner
5. plan synthesis
6. plan analysis and post-design

The work described in this paper focusses on the third phase, model analysis, which includes the verification and validation of the planning domain model.

One of the most advanced systems in this area is GIPO [Simpson, 2007] which includes support for the complete KE process, including model analysis. GIPO goes well beyond simple syntactic checks, verifying the consistent use

of a type hierarchy and predicate templates, as well as more advanced features such as invariants. It also includes a tool that visualizes domain dynamics for the knowledge engineer. Another graphical system supporting KE for planning domains is itSIMPLE [Vaquero *et al.*, 2007]. Static support for model analysis is mostly visual, using multiple views which can also be interpreted as a kind of redundancy.

The target output in most KE systems for planning is the Planning Domain Definition Language (PDDL) [Fox and Long, 2003], which has become a de-facto standard for specifying STRIPS-like planning domains and problems with various extensions. PDDL allows for the specification of some auxiliary information about a domain, such as types, but this information is optional.

Domain Features

Amongst the features mentioned above, domain types have received significant attention in the planning literature. A rigorous method for problem formulation in the case of planning domains was presented in [McCluskey and Porteous, 1997]. In the second step of their methodology types are extracted from an informal description of a planning domain. Types have been used as a basic domain feature in TIM [Fox and Long, 1998]. Their approach exploits functional equivalence of objects to derive a hierarchical type structure. This work has later been extended to infer generic types such as mobiles and resources that can be exploited to optimize plan search [Coles and Smith, 2006].

The distinction between rigid and fluent relations [Ghallab *et al.*, 2004] is common in AI planning and will be discussed below. Inconsistent effects of different actions are exploited in the GraphPlan algorithm [Blum and Furst, 1995] to define the mutex relation. However, this is applied to pairs of actions (i.e. fully ground instances of operators) rather than operators. Reversible actions, as a domain feature, are not related to regression of goals, meaning this feature is unrelated to the direction of search (forward from the initial state or regressing backwards from the goal). A formal treatment of reversibility of actions (or operators) does not appear to feature much in the AI planning literature, despite the fact that reversible actions are common in planning domains. However, in generic search problems they are a common technique used to prune search trees [Russell and Norvig, 2003].

Preprocessing of planning domains is a technique that has been used to speed up the planning process [Dawson and Siklossy, 1977]. Perhaps the most common preprocessing step is the translation of the STRIPS (function-free, first-order) representation into a propositional representation. An informal algorithm for this is described in [Ghallab *et al.*, 2004, section 2.6]. A conceptual flaw in this algorithm (highlighted by the analysis of inconsistent effects) was described in [Wickler, 2011].

Static and Fluent Relations

A domain feature that is useful for the analysis of planning domains concerns the relations that are used in the definition of the operators. The set of predicates used here can be divided into static (or rigid) relations and fluent (or dynamic) relations, depending on whether atoms using this predicate can change their truth value from state to state.

Definition 1 (static/fluent relation) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of operators and let $P = \{P_1, \ldots, P_{n(P)}\}$ be a set of all the predicate symbols that occur in these operators. A predicate $P_i \in P$ is **fluent** iff there is an operator $O_j \in O$ that has an effect that uses the predicate P_i. Otherwise the predicate is **static**.*

The algorithm for computing the sets of fluent and static predicate symbols is trivial and hence, we will not list it here.

There are at least two ways in which this information can be used in the validation of planning problems. Firstly, if the domain definition language allowed the domain author to specify whether a relation is static or fluent then this could be verified when the domain is parsed. This might highlight problems with the domain. This use requires only a planning domain to be provided. Secondly, if a planning problem that uses additional relations is given, these could be highlighted or simply removed from the initial state.

Type Information

Many planning domains include explicit type information. In PDDL the `:typing` requirement allows the specification of typed variables in predicate and operator declarations. In problem specifications, it allows the assignment of constants or objects to types. If nothing else, typing tends to greatly increase the readability of a planning domain. However, it is not necessary for most planning algorithms to work, and the analysis techniques described here require neither a problem nor explicit types to be given.

Type Consistency The simplest kind of type system often used in planning is one in which the set of all constants C used in the planning domain and problem is divided into disjoint types T. That is, each type corresponds to a subset of all constants and each constant belongs to exactly one type. This is the kind of type system we will look at here.

Definition 2 (type partition) *A **type partition** $\mathcal{P}$ is a tuple $\langle C, T, \tau \rangle$ where:*

- *C is a finite set of $n(C) \geq 1$ constant symbols $C = \{c_1, \ldots, c_{n(C)}\}$,*
- *T is a set of $n(T) \leq n(C)$ types $T = \{t_1, \ldots, t_{n(T)}\}$, and*
- *$\tau : C \rightarrow T$ is a function defining the type of a given constant.*

A type partition divides the set of all constants that may occur in a planning problem into a set of equivalence classes. However, constants are only necessary to define the types' extension. As we shall show, the intension is implicit in the operators that constitute the planning domain alone. The availability of a type partition can be used to limit the space of world states that may be searched by a planner. In general, a world state in a planning domain can be any subset of the powerset of the set of ground atoms over predicates P with arguments from C.

Definition 3 (type function) *Let $P = \{P_1, \ldots, P_{n(P)}\}$ be a set of $n(P)$ predicate symbols with associated arities $a(P_i)$ and let $T = \{t_1, \ldots, t_{n(T)}\}$ be a set of types. A **type function** for predicates is a function*

$$arg_P : P \times \mathbb{N} \to T$$

which, for a given predicate symbol P_i and argument number $1 \leq k \leq a(P_i)$ gives the type $arg_P(P_i, k) \in T$ of that argument position.

This is the kind of type specification we find in PDDL domain definitions as part of the definition of predicates used in the domain, provided that the typing extension of PDDL is used. The type function is defined by enumerating the types for all the arguments of each predicate.

Definition 4 (type consistency) *Let $\langle C, T, \tau \rangle$ be a type partition. Let $P_i \in P$ be a predicate symbol and let $c_1, \ldots, c_{a(P_i)} \in C$ be constant symbols. The ground first-order atom $P_i(c_1, \ldots, c_{a(P_i)})$ is **type consistent** iff $\tau(c_k) = arg_P(P_i, k)$. A world state is **type consistent** iff all its members are type consistent.*

Thus, for a given predicate P_i there are $|C|^{a(P_i)}$ possible ground instances that may occur in world states. Clearly, the set of type consistent world states is a subset of the set of all world states. The availability of a set of types can also be used to limit the actions considered by a planner.

Definition 5 (type function) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of $n(O)$ operator names with associated arities $a(O_i)$ and let $T = \{t_1, \ldots, t_{n(T)}\}$ be a set of types. A **type function** for operators is a function*

$$arg_O : O \times \mathbb{N} \to T$$

which, for a given operator symbol O_i and argument number $1 \leq k \leq a(O_i)$ gives the type $arg_O(O_i, k) \in T$ of that argument position.

Again, this is exactly the kind of type specification that may be provided in PDDL where the function is defined by enumeration of all the arguments with their types for each operator definition.

Definition 6 (type consistency) *Let $\langle C, T, \tau \rangle$ be a type partition. Let $O_i(v_1, \ldots, v_{a(O_i)})$ be a* STRIPS *operator defined over variables $v_1, \ldots, v_{a(O_i)}$ with preconditions precs(O_i) and effects effects(O_i), where each precondition/effect has the form $P_j(v_{P_j,1}, \ldots, v_{P_j,a(P_j)})$ or $\neg P_j(v_{P_j,1}, \ldots, v_{P_j,a(P_j)})$ for some predicate $P_j \in P$. The operator O_i is **type consistent** iff:*

- *all the operator variables $v_1, \ldots, v_{a(O_i)}$ are mentioned in the positive preconditions of the operator, and*
- *if $v_k = v_{P_j,l}$, i.e. the kth argument variable of the operator is the same as the lth argument variable of a precondition or effect, then the types must also be the same: $arg_O(O_i, k) = arg_P(P_j, l)$.*

The first condition is often required only implicitly (see [Ghallab *et al.*, 2004, chapter 4]) to avoid the complication of "lifted" search in forward search.

Derived Types The above definitions assume that there is an underlying type system that has been used to define the planning domain (and problems) in a consistent fashion. We shall continue to assume that such a type system exists, but it may not have been explicitly specified in the PDDL definition of the domain. We shall now define a type system that is derived from the operator descriptions in the planning domain.

Definition 7 (type name) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of* STRIPS *operators. Let P be the set of all the predicate symbols used in all the operators. A **type name** is a pair $\langle N, k \rangle \in (P \cup O) \times \mathbb{N}$.*

A type name (a predicate or operator name and an argument number) can be used to refer to a type in a derived type system. There usually are multiple names to refer to the same type. The basic idea behind the derived types is to partition the set of all type names into equivalence classes. If a planning problem is given, the constants occurring in such a problem can then be assigned to the different equivalence classes, thus treating each equivalence class as a type.

Definition 8 (O-type) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of* STRIPS *operators over operator variables $v_1, \ldots, v_{a(O_i)}$ with conds(O_i) = precs$(O_i) \cup$ effects(O_i) and all operator variables mentioned in the positive preconditions. Let P be the set of all the predicate symbols used in all the operators. An **O-type** is a set of type names. Two type names $\langle N_1, i_1 \rangle$ and $\langle N_2, i_2 \rangle$ are in the same O-type, denoted $\langle N_1, i_1 \rangle \equiv_O \langle N_2, i_2 \rangle$, iff one of the following holds:*

- *$N_1(v_{1,1}, \ldots, v_{1,a(N_1)})$ is an operator with precondition or effect $N_2(v_{2,1}, \ldots, v_{2,a(N_2)}) \in$conds$(N_1)$ which share a specific variable: $v_{1,i_1} = v_{2,i_2}$,*
- *$N_2(v_{2,1}, \ldots, v_{2,a(N_2)})$ is an operator with precondition or effect $N_1(v_{1,1}, \ldots, v_{1,a(N_1)}) \in$conds$(N_2)$ which share a specific variable: $v_{1,i_1} = v_{2,i_2}$, or*
- *there is a type name $\langle N, j \rangle$ such that $\langle N, j \rangle \equiv_O \langle N_1, i_1 \rangle$ and $\langle N, j \rangle \equiv_O \langle N_2, i_2 \rangle$.*

Definition 9 (O-type partition) *Let (s_i, g, O) be a* STRIPS *planning problem. Let C be the set of all constants used in s_i. Let $T = \{t_1, \ldots, t_{n(T)}\}$ be the set of O-types derived from the operators in O. Then we can define the function $\tau : C \to T$ as follows:*

$$\tau(c) = t_i : \forall R(c_1, \ldots, c_{a(R)}) \in s_i : (c_j = c) \Rightarrow \langle R, j \rangle \in t_i$$

Note that $\tau(c)$ is not necessarily well-defined for every constant mentioned in the initial state, e.g. if a constant is used in two relations that would indicate different derived types (which rely only on the operator descriptions). In this case the O-type partition cannot be used as defined above. However, if appropriate unions of O-types are taken then this results in a new type partition for which $\tau(c)$ is defined. In the worst case this will lead to a type partition consisting of a single type. Given that this approach is always possible, we shall now assume that $\tau(c)$ is always defined.

Definition 10 *Let $T = \{t_1, \ldots, t_{n(T)}\}$ be the set of O-types for a given set of operators O and let $P = \{P_1, \ldots, P_{n(P)}\}$ be the predicates that occur on operators from O. We can easily define type functions arg_P and arg_O as follows:*

$$arg_P(P_i, k) = t_i : \langle P_i, k \rangle \in t_i \text{ and}$$
$$arg_O(O_i, k) = t_i : \langle O_i, k \rangle \in t_i$$

Proposition 1 *Let (s_i, g, O) be a* STRIPS *planning problem and let $\langle C, T, \tau \rangle$ be the O-type partition derived from this problem. Then every state that is reachable from the initial state s_i is type consistent.*

To show this we first show that the initial state is type consistent. Since the definition of τ is based on the argument positions in which they occur in the initial state, this follows trivially.

Next we need to show that every action that is an instance of an operator in O is type consistent. All operator variables must be mentioned in the positive preconditions according to the definition of an O-type. Furthermore, if a precondition or effect share a variable with the operator, these must have the same type since $\equiv_O$ puts them into the same equivalence class.

Finally we can show that, if action a is applicable in a type consistent state s, the resulting state $\gamma(s, a)$ must also be type consistent. Every atom must come either from s in which case it must be type consistent, or it comes from a positive effect, which, given the type consistency of a means it must also be type consistent. ■

This shows that the type system derived from the operator definitions is indeed useful as it creates a state space of type consistent states. Note that the definition of the type system does not require the initial state that is part of the problem, but uses only the operators. The initial state is only necessary for the proposition, as is spans a state-space about which we can make a claim in the above proposition.

Advanced Features

In this section we shall define some more abstract features that can be used to achieve an understanding of a planning domain that is, perhaps, more human-like. The formal definition of these features represent the main contribution of this paper.

Static Graphs

We have now formally defined a type system that we can derive from a planning domain, and we have defined what it means for a predicate used in a planning domain to be static. Together, these two features form the basis for the static graphs that we will identify in a given planning domain. We shall use the dock worker robots (DWR) domain defined in [Ghallab *et al.*, 2004] to illustrate the concepts defined in this section.

Many planning domains fall into the general category of transportation domains. That is, they define a network of locations that are connected by paths that can be traversed by vehicles. Often there are other types of movable objects that need to be brought into a given configuration, defined by the goal of a planning problem over such a domain. The path network that is part of the planning problem is usually fixed, meaning it cannot be changed by actions in the domain. It forms a graph that can be analyzed independent from the state of the world, i.e. the location of vehicles and other movable objects.

The following definition attempts to capture this notion:

Definition 11 (static graph relation) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of* STRIPS *operators. Let $P = \{P_1, \ldots, P_{n(P)}\}$ be a set of $n(P)$ predicate symbols with associated arities $a(P_i)$ used in all the operators. Then we say that P_i is a* ***static graph relation*** *if and only if:*

- *P_i is a static relation;*
- *P_i is a binary relation: $a(P_i) = 2$; and*
- *the two arguments of P_i are of the same (derived) type: $arg_{P_i}(1) = arg_{P_i}(2)$.*

Note that this definition relies only on information that can be computed from the planning domain specification, i.e. no planning problem and no hint from the knowledge engineer as to what relation might define a graph is required. In the DWR domain, the only relation that satisfies these conditions is the `adjacent`-relation, and this the relation that defines the network of locations between which the robots can move. Given a static graph relation, it is straight-forward to define the graph that is defined by this relation.

Definition 12 (static graph) *Let (s_i, g, O) be a* STRIPS *planning problem and let $\langle C, T, \tau \rangle$ be the O-type partition derived from this problem. Let P_i be a static graph relation for the set of operators O. Then P_i defines a* ***static graph*** *$G_{P_i} = (V, E)$ consisting of nodes (vertices) V and directed edges E, where:*

- *$V = \{c \in C | \tau(c) = arg_{P_i}(1) = arg_{P_i}(2)\}$; and*
- *$E = \{(c, c') | P_i(c, c') \in s_i\}$.*

Thus, in the DWR example, the `adjacent`-relation defines a graph that consists of nodes that correspond to instances of the type `location` and the edges are defined by the initial state.

Note that the definition of a static graph applies to a planning problem. However, the analysis which relations can define a static graph is problem independent, and therefore does not have to be re-computed for each problem. It can be computed from the set of operators defining the domain.

To exploit this analysis for verification and validation, the knowledge engineer would have to explicitly specify which relations are meant to represent static graphs. Furthermore, a relation specified as a static graph relation could have more properties that can be easily verified: it could be reflexive, symmetric, and/or transitive, or it could even be specified to represent a tree structure.

Finally, note that the idea of static graphs is based on location networks as found in the DWR domain, but static graph could represent many things, and thus this technique is more general and not only applicable to transportation domains.

Node-Fixed Types

Given a static graph, it is often possible to identify other types that represent objects or properties in the domain which have a fixed relation to nodes in a static graph.

Definition 13 (node-fixed type) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of* STRIPS *operators and P_i be a static graph relation for this domain, where t_i is the*

node type for this relation. A type $t_j \neq t_i$ represents a ***node-fixed type*** *if and only if:*

- *there exists a static binary relation P_j; and*
- *P_j has one argument of type t_i and the other of type t_j.*

The intuition for node-fixed types should be fairly obvious: these are objects that cannot move between nodes in a static graph. In the DWR example, where the only static graph is defined by the `adjacent`-relation and the nodes are of type `location`, there are two node-fixed types: a `crane` belongs to a location and `pile` (of containers) is attached to a location. Thus, any operator that has a crane or a pile as one of its parameters is implicitly located by these objects.

In general, one might expect the relation P_j to be functional if the node-fixed type represents a physical object that can only be at one node. This is not necessarily the case, however, and the node-fixed type may represent a property (e.g. a colour) in which case instances can be associated with multiple nodes. Similarly, the relation P_j may be functional in the other direction, e.g. if there is only ever one crane at a given location. Once a node-fixed type has been identified the functional properties of the defining relation can be computed fairly easily for a given planning problem, which will define the graph. Also, since the relation must be static (by definition), these functional properties do not change in the state space spanned the problem. However, since this requires a planning problem to be given we shall not go into it.

Finally, the intuition behind node-fixed types may location-based, but nothing in the definition requires such a view, and thus this technique may be applied to any type of static graph.

Shift Operators

Given a static graph relation that defines a static graph for a given planning problem, there are often operators that shift objects or properties from one node in the static graph to a neighbouring node. This is a basic way in which the underlying state can be changed. Of course, there is usually more to such an operator than the simple shifting of an object or property.

Definition 14 (shift operator) *Let O be a planning operator with positive preconditions $p^p_1, \ldots, p^p_{n(p^p)}$, positive effects $e^p_1, \ldots, e^p_{n(e^p)}$ and negative effects $e^n_1, \ldots, e^n_{n(e^n)}$, where each precondition and positive/negative effect is a first-order atom. Let P_i be a static graph relation for the domain containing O, where t_i is the node type for this relation. Then we say that O is a* ***shift operator wrt. node type*** *t_i if and only if:*

- *O has a precondition $P_i(v, v')$;*
- *O has a precondition p^p_s that has an argument v (or v');*
- *O has a negative effect that is equal to this precondition p^p_s; and*
- *O has a positive effect that is equal to the precondition except where the argument v (or v') occurs, where the effect must have the value v' (or v respectively).*

Again, note that the definition of a shift operator relies solely on the definition of the operator, i.e. no planning problem is required. However, once a planning problem is given, this analysis step can be used to compute which objects or properties can be shifted to which other node in a static graph.

To illustrate this definition, we shall look at the `move` operator defined for the DWR domain. This operator is defined as follows:

```
(:action move
  :parameters (?r ?fr ?to)
  :precondition (and (adjacent ?fr ?to)
    (at ?r ?fr) (not (occupied ?to)))
  :effect (and (at ?r ?to) (occupied ?to)
    (not (occupied ?fr)) (not (at ?r ?fr))))
```

The intended meaning should be fairly obvious (to a human): the operator moves `?r`, a `robot`, from the `location ?fr` to the `location ?to`. Note that the operator has two parameters that have the node type (`location`) for the static graph defined by the `adjacent`-relation. The verify that `move` is a shift operator with respect to node type `location`, we simply have to find preconditions and effects according to the definition:

- the precondition `(adjacent ?fr ?to)` uses the predicate that defines the static graph, thus defining the edge along which this operators shifts;
- the precondition `(at ?r ?fr)` uses the variable `?fr` as its second argument, thus defining the node from which the shifting takes place;
- `(at ?r ?fr)` is also a negative effect of the operator; and finally
- the positive effect `(at ?r ?to)` is equal to the deleted precondition except for the position of the second argument which is replaced by the other node given in the first precondition, thus defining the node to which a shift takes place.

The algorithm that identifies shift operators follows the same procedure and simply applies the definition, attempting to find preconditions and effects that satisfy all the conditions. This is expressed in the following pseudo code.

function is-shift-op(O, P_i)
  for every $P_i(v, v') \in p^p_1, \ldots, p^p_{n(p^p)}$ **do**
    for every $P_j(v_1, \ldots, v_k) \in p^p_1, \ldots, p^p_{n(p^p)}$ **do**
      $i_v \leftarrow i_v$ such that $v_{i_v} = v$
      if i_v is undefined **continue**
      if $P_j(v_1, \ldots, v_k) \notin e^n_1, \ldots, e^n_{n(e^n)}$ **continue**
      for every $P_j(x_1, \ldots, x_k) \in e^p_1, \ldots, e^p_{n(e^p)}$ **do**
        for $i_e \in 1 \ldots k$ **do**
          if $i_v = i_e \wedge x_{i_e} \neq v'$ **next** $P_j(x_1, \ldots, x_k)$
          if $i_v \neq i_e \wedge x_{i_e} \neq v x_{i_e}$ **next** $P_j(x_1, \ldots, x_k)$
        return true

The algorithm takes two parameters, an operator and a static graph relation. It returns true if and only if the given operator is a shift operator wrt. the argument type of the

given predicate. Note that this requires the type to be known and well-defined, but as shown above, this kind of type can be derived from the planning domain. The algorithm then loops over all the precondition to find one that represents an edge in the graph. Then it loops over the preconditions again to find one that represents a candidate for a shifted property. A necessary condition here is that the node from which we are shifting occurs in the property precondition. Given such a candidate, the algorithm tests whether the property is deleted by the operator, another necessary condition. Finally, the algorithm tests whether a positive effect exists that represents the shifted property, which is true if it agrees with the property precondition in all arguments except for the one representing the static graph node, which must be the node to which we are shifting for the effect.

Domain Analysis

The above definitions and algorithm show how we can understand a planning domain in terms of static graphs that will be encoded in the planning problem. As already mentioned, the analysis of the domain (without the problem) lets us identify which relations represent edges and which types represent nodes in such a static graph. Given the planning problem itself, we may perform an analysis of the graph itself, where the results of this analysis will be valid for every world state that is reachable from the initial state.

One specific property that might be interesting in such a static graph in light of the shift operators just defined, is whether the graph is fully connected, or if it is not, which nodes are reachable from a given node. If we know that a node is reachable from a node in the initial state, and we know that a property that we can shift with a given operator holds in the initial state, then we know that this property can be achieved in any node reachable from the node in the initial state. In other words, we have a reachability condition that can be evaluated in constant time. Of course, planning graph analysis [Blum and Furst, 1995; Hoffmann and Nebel, 2001] gives us the same information and more, but at a much higher computational cost. This type of analysis is very useful to guide search, but whether it contributes to a better understanding of the problem is a different question, and a better understanding is what aids knowledge engineering.

There is another important difference between our analysis and the planning graph techniques just mentioned. While the latter give very good information to a heuristic search planner, it is hard to understand the information contained in a planning graph from a knowledge engineering point of view. The technique described in this paper is specifically aimed at the knowledge engineer, supporting a more intuitive way of understanding how a given set of operators can manipulate a world state.

In fact, the analysis can be used to help the knowledge engineer even more. Once an operator has been identified as shifting a property across a network of nodes, the next question is what other conditions there are that make the generic problem difficult. Clearly, if the shifting was all that is going on in a domain, this would not be a hard problem. So the remaining preconditions and effects of a shift operator must somehow encode the difficult part.

For example, the the `move` operator defined for the DWR domain constitutes a shift operator that shifts the location of the robot as given by the `at` along edges defined by the `adjacent relation`. Thus, for a given problem, it is easy to compute all the possible locations at which a robot may be in the state space. Also, paths to these locations are easy to compute. However, the other conditions defining the `move` operator specify that movement is only possible to an unoccupied location. Thus, the problem becomes hard, because an optimal solution involves collision avoidance in time. Interestingly, if one continued this line of reasoning it should be easy to see that problems involving one robot are easy, as the `occupied` relation can simply be disregarded, that is, it can dropped from the planning domain. This type of reasoning is very much in line with the kind of analysis shown in [Amarel, 1968], and this is what we hope to (eventually) achieve with this work.

Evaluation

The methodology for evaluating the technique described above was chosen as follows. We used the DWR domain [Ghallab *et al.*, 2004] to develop the technique in terms of definitions and algorithms. This was possible because the DWR domain does encode a static network of locations along which a robot can move. There are also other operators that do not represent shift operations. Thus, this domain was used to provide a first correctness check.

To properly evaluate the technique we have applied it to a small number of other planning domains. To avoid any bias we used only planning domains that were available from third parties, namely from the international planning competition. Since the algorithm works on domains and the results have to be interpreted manually only a limited number of experiments was possible. Note that a knowledge engineer using the approach described here to ensure the consistency of the domain they are developing would not need to perform a manual analysis. This manual analysis is only necessary for this evaluation as the features we are looking at were not defined with the domains. Random domains are not suitable as they cannot be expected to encode meaningful knowledge.

The domains used for the evaluation were the simple STRIPS versions of the following domains: movie, gripper, logistics, mystery, mprime, and grid. The first step towards an analysis of these domains was an analysis identifying static and fluent relations and derived types, as none of the domains had explicit given types as part of the domain definition. However, this will not be described here.

Static Graphs

The first domain in our analysis, the movie domain, is a very simple domain that is almost propositional. All the predicates used are unary, effectively specifying the types of objects that exist. Clearly, there is no static graph encoded here and indeed, our algorithm does not identify any static graph relation.

The second domain, gripper, is more promising as it contains a robot that can move objects between room. Thus

the same intuition as for the DWR domain applies, and one might expect to find at least a network of locations as a static graph in this domain. However, the domain is defined to allow movement of the robot from any room to another; the path taken by the robot is abstracted away. No other static graph is identified by our algorithm in this domain.

The same issue occurs in the logistics domain. Again, the connections between the different types of places are not explicit, meaning no static graph can be identified. This is unfortunate as the domain has different types of locations (cities and airports), which would have provided an interesting challenge for our intuition.

The next domain, mystery, is perhaps the most interesting case here. This domain is not based on robots that have to transport objects between locations, and it is indeed a mystery what is going on in this domain at first glance. Our domain analysis identifies three different predicates that define static graphs for this domain. The first predicate, `orbits` is perhaps the most obvious as it can be seen as location-related. The other static graph relations identified are `eats` and `attacks`, which are somewhat similar and can be interpreted in a meaningful way. Note that none of the three relations identified here would be expected to be symmetric, so this is quite different from the adjacency used as the intuition behind the approach.

The mprime domain is really a variation of the mystery domain and the analysis of static graph relations yields the same three relations as a result.

The final domain used for the evaluation is the grid domain which is a classic transportation domain in which the network of locations is explicitly specified. And indeed, our domain analysis finds one static graph relation for this domain, namely the relation `conn` which, not surprisingly, corresponds directly to the adjacency relation in the DWR domain.

Node-Fixed Types

The movie domain which does not contain a static graph, cannot contain node-fixed types. The same is true for the gripper and the logistics domain.

More interesting is the mystery domain, which had three static graph relations. The first of these, `orbits`, gives us one node-fixed type. Looking at the type definition, we can see that this is the type used for both arguments of the `orbits`-relation, which must hold by definition. Another place where this type is used is the unary `plant` predicate, which gives us an idea of type of object this is meant to be. Finally, the type is also used as the second argument of the `harmony`-relation. The other static graph relations, `eats` and `attacks`, also define one node-fixed type each: `eats` gives us a type that is also used in the unary predicate `food`, whereas `attacks` reveals a node-fixed type that is not used in a unary predicate, but only as the second argument in the `locale`-relation.

The node-fixed types for the mprime domain are the same as the ones for the mystery domain. What is perhaps interesting here is that there is a surprising amount of static information in both these domains, which might make a domain analysis before planning useful.

Finally, the grid domain is less surprising here, rendering just one node-fixed type which is used in, amongst other, two unary relations, namely `at-robot` and `place`. The latter can be interpreted as the node type, of course.

Shift Operators

Of course, none of the first three domains, movie, gripper and logistics contains shift operators since there also do not contain static graphs.

The mystery domain is again interesting in that two of the static graph relations have operators that shift properties along the edges. Firstly, the operator `succumb` shifts `harmony(?v,?s1)` along `orbits(?s1,?s2)` to `harmony(?v,?s2)`, and secondly, the operator `feast` shifts `craves(?v,?n1)` along `eats(?n1,?n2)` to `craves(?v,?n2)`.

And this is where the mprime domain differs from the mystery domain significantly. In the mprime domain there is a shift operator also for the final static graph relation: operator `drink` shifts `locale(?n2,?l21)` along `attacks(?l21,?l22)` to `locale(?n2,?l22)`.

The last domain, the grid domain, shows itself as being similar to the DWR domain again. It is the move operator that can be used to shift the robot from one place to another. What is perhaps interesting here is that the property that is shifted in this domain is represented by a unary predicate: `at-robot`. This is of course a valid representational choice and, perhaps fortunately, the definitions of static graphs and shift operators are sufficiently general to catch this special case.

Conclusions

The work described in this paper builds on the domain analysis in terms of features described in [Wickler, 2011]. The general approach in which this work can be used is similar to the previous analysis: knowledge engineering could be given the option to specify additional, redundant knowledge that can then be used by the automatic analysis to ensure consistency of the representation. As for previous work, the analysis is of the domain, that is, a set of operators, not of a planning problem. Thus, the result of the analysis is valid for all problems referring to the analyzed domain and will not have to be repeated.

Further analysis of a planning problem, based on the results of the domain analysis, may be performed and may give further insights into the problem. In fact, it is often the case that those aspects of the initial state of a planning problem that are given by the static relations have a degree of reusability and may not change across a range of problems. For example, in a real dock the topology is fixed and will not change from one problem to the next.

The major difference between previous work and the technique described here is that the analysis is in terms of a feature that may or may not be present in a planning domain, namely, static graphs that are more or less explicit in the representation. GIPO and itSIMPLE are systems that support the whole KE process, but neither performs an analysis in terms of shift operators based on static graphs as we have defined

it here. A graph consisting of nodes and edges is a very generic way of describing an aspect of a problem, and thus we can hope to find this feature in many domains, but if not present, this analysis obviously cannot aid the knowledge engineering for such a domain.

Also, the analysis algorithms and the underlying definitions rely on certain representational choices that are commonly used when formally representing knowledge, but there may be alternatives that we have not considered that may make the analysis fail despite the presence of static graphs.

The DWR domain that was used to develop the ideas underlying this work highlights some of the issues with the representation that interfere with our analysis. For example, our algorithm does not find that the move operator also shifts the `occupied`-relation together with the `at`-relation. This is simple because the domain uses occupied as a negative precondition, which is not captured by the definition. The definitions could of course be adapted to this case, but it also raises the question why this representational choice was made and whether its negation, the `free`-relation, would not be a better choice for the domain.

Similarly, the exploitation of invariants may lead to domain specifications for which our analysis fails.

Another interesting result from the evaluation is that most static graphs were identified in domains that are not classic transportation domains. While this shows that the analysis does indeed apply to other classes of problems, it is not clear what these classes are. However this was never our aim; our approach is inspired by one problem class, but the definitions do not make reference to specific concepts associated with this class.

What the type of domain analysis presented in this paper is trying to achieve is a more human-like understanding of a planning domain without reverting to using the names of symbols as a clue to their meaning or even the comments in a PDDL. A graph may be just one way to achieve this. If we could identify other sub-problems that are common in planning domains and allow for a fast (polynomial time) analysis, then it may just be possible to identify what exactly it is that makes a planning problem that uses the domain difficult, or which interaction of features is the hard part of the generic problem. This is future work, of course.

References

Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence 3*, pages 131–171. Elsevier/North-Holland, 1968.

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1636–1642. Morgan Kaufmann, 1995.

Andrew Coles and Amanda Smith. Generic types and their use in improving the quality of search heuristics. In *Proc. 25th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2006)*, 2006.

Clive Dawson and Laurent Siklossy. The role of preprocessing in problem-solving systems. In *Proc. 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 465–471. Morgan Kaufmann, 1977.

Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

Maria Fox and Derek Long. PDDL2.1 : An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning*. Morgan Kaufmann, 2004.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

T.L. McCluskey and J.M. Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95:1–65, 1997.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

R.M. Simpson. Structural domain definition using GIPO IV. In *Proc. 2nd Int. Competition on Knowledge Engineering for Planning and Scheduling*, 2007.

Tiago Stegun Vaquero, V. Romero, F. Tonidandel, and J.R. Silva. itSIMPLE 2.0: An integrated tool for designing planning environments. In Mark Boddy, Maria Fox, and Sylvie Thiébaux, editors, *Proc. 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 336–343, 2007.

Tiago Stegun Vaquero, José Reinaldo Silva, and J. Christopher Beck. A brief review of tools and methods for knowledge engineering for planning and scheduling. In Roman Barták, Simone Fratini, Lee McCluskey, and Tiago Stegun Vaquero, editors, *Proc. Knowledge Engineering for Planning and Scheduling (KEPS)*, pages 7–14, 2011.

Gerhard Wickler. Using planning domain features to facilitate knowledge engineering. In Roman Barták, Simone Fratini, Lee McCluskey, and Tiago Stegun Vaquero, editors, *Proc. Knowledge Engineering for Planning and Scheduling (KEPS)*, pages 39–46, 2011.