Domain Model Acquisition in the Presence of Static Relations in the LOP System

Peter Gregory
Digital Futures Institute,
School of Computing,
Teesside University, UK
p.gregory@tees.ac.uk

Stephen Cresswell
The Stationery Office
St. Crispins, Duke Street,
Norwich, NR3 1PD, UK
stephen.cresswell@tso.co.uk

Abstract

This paper addresses the problem of domain model acquisition from only action traces when the underlying domain model contains static relations. Domain model acquisition is the problem of synthesising a planning domain model from example plan traces and potentially other information, such as intermediate states.

The *LOCM* and *LOCM2* domain model acquisition systems are highly effective at determining the dynamics of domain models with only plan traces as input (i.e. they do not rely on extra inputs such as predicate definitions, initial, final and intermediate states or invariants). Much of the power of the *LOCM* family of algorithms comes from the assumption that each action parameter goes through a transition. One place that this assumption is too strong is in the case of static predicates.

We present a new domain model acquisition algorithm, LOP, that induces static predicates by using a combination of the generalised output from *LOCM2* and a set of optimal plans as input to the learning system. We observe that static predicates can be seen as restrictions on the valid groundings of actions. Without the static predicates restricting possible groundings, the domains induced by *LOCM2* produce plans that are typically shorter than the true optimal solutions. LOP works by finding a minimal static predicate for each operator that preserves the length of the optimal plan.

Introduction

Modelling is well known as a bottleneck in the development of solutions to difficult combinatorial problems. The research field of automated modelling focuses on the task of constructing formal descriptions of problems automatically, often using solution data as input. Automated model acquisition is an active research area in constraint programming, general game playing and computer security (e.g. (O'Sullivan 2010; Bessiere et al. 2014; Björnsson 2012; Aarts, De Ruiter, and Poll 2013)).

Domain model acquisition is automated modelling in a planning context: it is the problem of learning planning domain models from example data. The *LOCM* family of domain model acquisition systems (Cresswell, McCluskey, and West 2009; Cresswell and Gregory 2011; Cresswell, McCluskey, and West 2013) learn planning domain models from collections of plans. In comparison to other systems of the same type, these systems require only a minimal amount of information in order to form hypotheses: they only require plan traces, where other systems require state information.

Copyright © 2015, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

```
(:action drive-truck-benchmark
 :parameters (?t - truck ?d - driver ?l1 - loc ?l2 - loc)
 :precondition (and
                    (at ?t ?l1)
                    (conn ?l1 ?l2)
                    (driving ?d ?t))
 :effect (and
                    (not (at ?t ?l1)) (at ?t ?l2)))
(:action drive-truck-locmii
 :parameters (?t - truck ?d - driver ?l1 - loc ?l2 - loc)
 :precondition (and
                    (at ?t ?l1)
                    (driving ?d ?t ?l1))
 :effect (and
                    (not (at ?t ?l1)) (at ?t ?l2)
                    (not (driving ?d ?t ?l1))
                    (driving ?d ?t ?l2)))
```

Figure 1: The Driverlog drive action as encoded manually in the benchmark domain (top) and by the *LOCM2* system (bottom). Note the lack of a static precondition in the *LOCM2* version means that there are no restrictions on truck movement.

However, the *LOCM* family of algorithms only learn the dynamic aspects of the domain (i.e. state changes that occur due to action application). This is problematic since many domains use static relations to restrict the possible actions. Consider the Driverlog domain, where the road map is encoded as a binary static predicate. Figure 1 shows both the benchmark and *LOCM2* versions of the drive action. The conn predicate encodes the road map in the hand coded benchmark. This is not modelled in the induced drive action. In this paper, we present an approach that can discover this type of predicate.

Some of these static relations could be inferred given extra information about intermediate states. However, it is more desirable to be able to infer these static relations using only the minimal information available to the *LOCM* system, since intermediate state information may not always be available. In this paper, we extend the *LOCM* system in order to detect static relations. The key assumption in our approach is that the input is drawn from optimal goal-directed

plans.

Our approach is to compare the optimal input plans with the optimal plans found using the induced domain model of *LOCM2*. Assuming that *LOCM2* has detected the dynamics of the problem correctly, then if the induced plan is shorter, then this is a good clue to the fact that some static relation has gone undetected. Static conditions supporting the length of the input plans can then be hypothesised. We call our system *LOP* (standing for *LOCM* with Optimised Plans). We present results that show *LOP* is effective in discovering static relations in the standard planning benchmark domains. We go on to show that even when the restriction of optimality is removed, and we learn with sub-optimal goal-directed plans, *LOP* can still usefully discover static relations.

Background

Since our approach is based on the *LOCM* family of algorithms, we now briefly present these algorithms, before discussing the issues involved in detecting static predicates using only plan traces.

The *LOCM* Algorithm

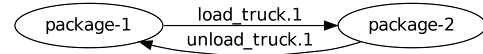
LOCM (Cresswell, McCluskey, and West 2009) is a system for learning domain models from example action sequences. Its distinguishing feature is that it uses no other information besides the action sequences - i.e. no information about types, predicates, initial or final states. This is possible because it is based on restricting assumptions about the form of the domain model.

The assumptions of *LOCM* are:

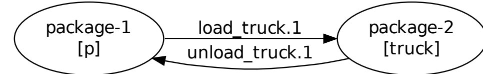
1. Each invocation of a planning operator causes a specific state change in each of the objects given as arguments.
2. The behaviour of each object is described by a single FSM.
3. Objects are grouped into sorts, and objects of the same sort are described by identical state machines.
4. Each argument position of each action always takes objects of the same sort.
5. Each transition appears only once in the FSM.

The domain model construction has two aspects:

Firstly, the action sequences are analysed to obtain a simple state machine for each sort. This is achieved by simply creating dummy start and end states for each transition, where a transition is identified uniquely by an action name and argument position. The sequences of transitions experienced by individual objects are then tracked through example plan sequences. For each pair of transitions occurring consecutively for an object, unify the end state of the first transition with start state of the second transition. By repeating this process for all objects and all example plans, the consequential unification of states causes the transitions to be grouped into a set of state machines. Each state machine represents the behaviour of a single sort, and the process also reveals which objects belong to each sort. E.g. for a package in Driverlog, the following machine is revealed, comprising two transitions and two states.



Secondly, the action sequences are further analysed to establish whether a given state for a given sort has a temporary association with another object. If so, the state is qualified with a parameter which records the association. For the Driverlog packages, they have an association with a place in one state, and a truck in the other state.



It is then possible for the learned model to be translated into the STRIPS fragment of PDDL. Each state is represented by a PDDL predicate having its associated object as first argument, with further arguments formed from state parameters. Operators are constructed from the transitions and their parameters, using the binding constraints discovered between action parameters and state parameters. It is also possible to output PDDL task descriptions for the example plans by describing the initial and final states using the synthesised predicates.

While the very simple underlying model representation is sufficient for many benchmark domains, one of the assumptions of *LOCM* leads to a significant limitation. As the behaviour of each sort is described by a single state machine, it is not possible to describe a sort which has separate independent aspects to its state. E.g. in Driverlog the sort *truck* has the transitions drive_truck.1, board_truck.2, disembark_truck.2, load_truck.2, unload_truck.2. The fact that load_truck.2 and unload_truck.2 can be interleaved arbitrarily with the other transitions masks out the state machine formed from the other three transitions.

The *LOCM2* Algorithm

The *LOCM2* system (Cresswell and Gregory 2011) overcomes some of the limitations of the original *LOCM* by generalising the underlying representation to allow a sort to be represented by multiple state machines, each containing a subset of the full transition set for the sort. In the Driverlog example, a separate machine is formed with only drive_truck.1, board_truck.2, disembark_truck.2 transitions, and this captures behaviour that was missed in *LOCM*. Another example is 4-operator blocksworld. Here there are some transitions which only change the state of the top of a block, and other transitions which change the state of both the top and bottom of the block. *LOCM2* analysis of this domain produces separate state machines for top and bottom of a block.

The generalised *LOCM2* representation is still readily translatable into PDDL-STRIPS, as each transition still only occurs once within each FSM.

```
(:action drive-truck-benchmark
 :parameters (?t - truck ?d - driver ?l1 - loc ?l2 - loc)
 :precondition (and
                   (at ?t ?l1)
                   (conn ?l1 ?l2)
                   (driving ?d ?t))
 :effect (and
                   (not (at ?t ?l1)) (at ?t ?l2)))

(:action jump-new-move
 :parameters (?from - location ?over - location ?to - location)
 :precondition (and
                       (move-ended)
                       (IN-LINE ?from ?over ?to)
                       (occupied ?from)
                       (occupied ?over)
                       (free ?to)
                    )
 :effect (and
                 (not (move-ended))
                 (not (occupied ?from))
                 (not (occupied ?over))
                 (not (free ?to))
                 (free ?from)
                 (free ?over)
                 (occupied ?to)
                 (last-visited ?to)
                 (increase (total-cost) 1)))
```

Figure 2: The Driverlog drive action as encoded manually in the benchmark domain (top) and the jump-new-move action from the Peg Solitaire domain. Note the static relations (conn and IN-LINE respectively)

Defining Static Relations

One task which was previously considered (Cresswell, McCluskey, and West 2013) of limited utility is that of PDDL problem generation from the input plans. In order to generate PDDL problems, the initial and goal states need to be known for each FSM. In *LOCM* this information is easy to find: the first and last transition in each object's transition sequence determines the initial and goal states in the PDDL translation.

In this work we require *LOCM2* generated PDDL problems. The problem of generating PDDL problem instances in *LOCM2* is complicated by the fact that objects may now be represented by multiple state machines. It can be the case that transitions have not been seen for all of the FSMs representing an object. In this case, we generate PDDL with no state for machines of this type. This has the negative consequence of restricting the possible transitions for objects with this restriction.

Static Relations in Planning

Static relations are often thought of by their semantic interpretations: road maps in Driverlog, successor relations in Zenotravel and Freecell, for example. This is seen in Figure 2 for the Driverlog and Peg Solitaire domains. We now present an alternative purely syntactic interpretation of static relations that is critical to our work.

Defining Static Relations Static relations can be seen as restrictions on the groundings of each operator. Instead of thinking of how the objects are related, we now think of which combinations of ground operator parameters are valid. The static relations in a domain can be defined as, for each operator, a table of all the valid groundings for that operator. Because of this, the static relations for any domain

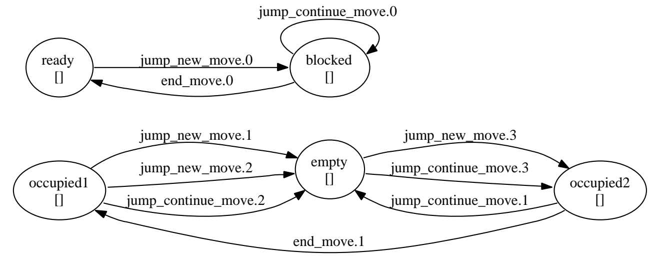


Figure 3: The derived state machines for the Peg Solitaire domain. This is a faithful representation of the original domain dynamics.

can be encoded as a single relation per operator.

This observation allows us to construct our hypothesis space of potential static relations. Each operator can be seen to have a static relation associated with it, and we have to identify the minimal subset of parameters that permit exactly the valid groundings of an operator.

We can now present definition of the minimal static relation identification problem:

Definition 1 (Minimal Static Relation Identification). Given an operator template:

$$\mathrm{op}(p_1, ..., p_n)$$

We define a minimal static relation as the subset $P_m \subseteq p_1, ..., p_n$ such that all $p_m \in P_m$ are members of static preconditions of op.

Thus, this problem is simply to identify the parameters which play a part in the static relation. In reality, there could be multiple static relations in the precondition. However, there is no loss of generality, as the cross product of these relations forms what we have defined as the minimal static relation. Factoring these smaller relations is an interesting problem from a practical standpoint, which we return to later in this work.

Universal Static Relations Some statics are defined for specific instances and are different in different instances depending on the problem objects. These include the road maps in Driverlog and the locations of the hoists in Depot, for example. In contrast to these types of static relations, there are also those that hold in all instances. For example, in Peg Solitaire, all instances use the same underlying board. In Zeno Travel, the static relation that encodes the successor relation for the fuel levels is the same in all problem instances. If a static relation holds between all instances, we call that a *universal* static relation. When finding static relations, it will clearly be useful to identify which are universal relations.

Issues in Detecting Statics The key issue in detecting statics using plan traces as input is that there are domains which look very similar, but which vary in the static relations. For example, Blocksworld and Freecell, in which the goal in both is to rearrange stacks of objects. In Blocksworld there are no static relations, in Freecell there are static relationships between the ranks and colours of the cards.

We can hypothesise two extremes:

- Anything goes: There are no static preconditions - e.g. blocksworld.
- Over-cautious: The only action instantiations permitted are exactly the ones that have occurred somewhere in the training set. This means that for every action schema like `move(?thing,?from,?to)`, we add a precondition `poss_move(?thing,?from,?to)`, and then create a static `poss_move` fact for each distinct observed action instance.

In order to address this issue we use additional information about the plans, other than simply the plan traces. Namely, we identify a set of optimal plans to be used in the learning phase. In the following sections, we detail exactly how we use optimal plans to discover static relations.

Detecting Static Relations

The state machine at the bottom of Figure 3 shows the dynamics of the position sort in the Peg Solitaire domain. The state occupied1 is the state in which a position is occupied by a peg, and there is no current move to continue. From this state, either the peg at the position is removed by starting a new move, or by another peg jumping over it. Both of these cases leave the position in the state empty. To become occupied again, a peg can be moved from elsewhere into the position. The PDDL generated by *LOCM2* is equivalent to the original PDDL, except that the static information is not present (the important static information is that the three locations in the parameters of the jump actions are in a line). Cresswell, McCluskey, and West discuss an approach to extracting static information in *LOCM*, but this relies on manually providing a hint.

Using *LOCM2* to Identify Static State Parameters

One method by which we can identify static relations is through a small modification to the *LOCM* analysis itself. Recall *LOCM* identifies state parameters by finding temporary associations, as described in the Background section. If it is the case that the association holds for the entire plan in all example traces then we form the hypothesis that this state parameter in fact describes a static relationship. Examples of when this occurs is in the Logistics domain and the Depots domain.

In effect, the existing *LOCM* analysis misreports some static relations as dynamic relations. Our extra check of the transition sequences divides the state parameters into those that are certainly dynamic, and the new class that seem to represent static relations. Of course, it could be that the state parameters are indeed dynamic relations, and that the input plans simply do not contain examples in which the state parameter changes. As is the case in any learning technique, overfitting to the input data is a potential problem.

A particular advantage to finding static relations in this way is that you require no more information than *LOCM2* uses. However, there are systematic cases where this analysis is insufficient to detect all static relations. The state parameters in *LOCM* state machines represent one-to-one and one-to-many relations. Therefore the limit to which the analysis detailed in this section can detect static relations does not extend to static many-to-many relations.

We now present an algorithm for detecting static relations of all types, which can be used as a post-processing step after *LOCM* analysis.

The *LOP* Algorithm

We now present the *LOP* algorithm that we use to detect static relations. The algorithm is sketched as follows:

1. Run *LOCM2* on both the optimal and suboptimal training data, constructing the *LOCM2* model of the domain dynamics.
2. Identify minimal subsets of each action's parameters that 'preserve' the optimal length of all plans.
3. Split each relation into a minimal partition that preserve optimal plan lengths.
4. Identify the universal static relations.
5. Return the valid ground relations and the problem-specific templates.

LOP Assumptions

The *LOP* algorithm relies on additional assumptions that we have discussed in the text. We now make these assumptions explicit:

1. The *LOCM2* system discovers the correct dynamics of the domain. It is necessary to trust the input from *LOCM2* being correct.
2. It is possible to identify the subset of the input plans that are optimal. These actions are assumed to be unit cost.

Preserving Optimality

All stages of the *LOP* algorithm rely on testing whether or not optimality is preserved for a set of input plans. We now define exactly what we mean by *preserving optimality*. Consider the following optimal input plan:

```
(drive truck loc1 loc2)
(drive truck loc2 loc3)
```

Now, consider the drive action and PDDL problem fragment induced by *LOCM2*:

```
(:action drive
 :parameters (?t - truck ?l1 ?l2 - loc)
 :precondition (at ?t ?l1)
 :effect (and (not(at ?t ?l1))
              (at ?t ?l2))
)
(:init (at truck loc1))
(:goal (at truck loc3))
```

Note that the dynamics of the drive action are correct: the truck moves from the start location to the destination location. However, if we solve this output using an optimal planner we find the following plan:

```
(drive truck loc1 loc3)
```

Algorithm 1 The Preserve Optimality testing algorithm.

Require: $\mathcal{M}$: an output domain model from *LOCM2*
Require: $\mathcal{T}$: a set of *LOCM2* generated problems
Require: P : a subset of the parameters for each operator
  function preserveOptimality
    for $t \in \mathcal{T}$ **do**
      optP $\leftarrow$ optimal input plan for t
      dom $\leftarrow$ $\mathcal{M}$ plus statics defined by P
      prob $\leftarrow$ t plus ground statics defined by optP
      optT $\leftarrow$ optimal solution to dom, prob
      if length(optT) $<$ length(optP) **then**
        return false
      end if
    end for
    return true
  end function

Algorithm 2 Minimal static relation hypothesis algorithm.

  function MSR
    for o : operators **do**
      msr(o) $\leftarrow$ MSRo(o)
    end for
    return msr
  end function
  function MSRo(o : operator)
    minS $\leftarrow$ parameters(o)
    for $p \in$ minS **do**
      minS′ $\leftarrow$ (minS $\setminus \{p\}$)
      if preserveOptimality(minS′) **then**
        minS $\leftarrow$ minS′
      end if
    end for
    return minS
  end function

This plan is shorter than the input plan which we *knew* to be optimal. Therefore, we say that the drive action does not *preserve optimality*. In order to restore the optimality we add static relations both in the action preconditions and the corresponding ground predicates in the initial state. For example, the set of all parameters yields the following action and problem fragment:

```
(:action drive
 :parameters (?t - truck ?l1 ?l2 - loc)
 :precondition (and (at ?t ?l1)
                (drive_static ?t ?l1 ?l2)
 :effect (and (not(at ?t ?l1))
              (at ?t ?l2))
)
(:init (at truck loc1)
       (drive_static truck1 loc1 loc2)
       (drive_static truck1 loc2 loc3)
)
(:goal (at truck loc3))
```

The important thing to note is that in order to test whether optimality is preserved, we are required to edit the domain by adding a predicate corresponding to the parameter set, and we are also required to edit the problem instance to add the instantiated statics as per the input plan. Algorithm 1 presents the algorithm for testing if optimality is preserved for a particular parameter set of the operators. In essence, it checks that each input plan preserves optimality for that parameter set.

There are eight possible combinations of the parameters of the drive action (all subsets of the parameters, including the empty set if there is no static relation). The minimal static relation for the drive action contains simply the two locations: in another domain, the truck may form part of the minimal static relation, an example of this type of is in the Rovers domain where each rover has its own roadmap. The algorithm for discovering minimal static relations (or MSRs) is presented in Algorithm 2 which attempts to remove each parameter in turn from a candidate relation, adding it back in if it leads to shorter optimal solutions.

Depending on the current MSR hypotheses for each operator, Algorithm 2 may return different results. For this reason, the algorithm is run repeatedly until a fix-point is reached, when the MSRs do not change following an iteration.

Splitting The Static Relation

Once a minimal static relation is found for each operator, we test if it can be divided into multiple smaller static relations. As an example, the Freecell domain has the operator:

$$\text{move}(?\text{card}, ?\text{cols}, ?\text{ncols}, ?\text{cells}, ?\text{ncells})$$

Where there is a static relation between cols and ncols and between cells and ncells. The minimal static relation will be across all four of these parameters, where it would clearly be beneficial to identify two separate relations.

In order to split the relation, we need to check the possible partitions of the parameters in the minimal static relation. Given a candidate partition of the parameters, we can use the same approach as when finding the minimal static relation, and verify that the candidate partition preserves optimality in the induced tasks.

Checking every partition is computationally prohibitive for actions with a large number of parameters. For this reason, we propose a pruning technique based on dominated partition refinements. Suppose that we test a certain partition of the minimal static relation that fails to preserve optimality within the input plans. Dividing the partition still further will not restore that optimality, and so no refinements of that partition need to be tested. A description of the algorithm is given in Algorithm 3.

Testing for Universality

As a final step, we test the static relations for universality. In each input plan, we only have access to a subset of the ground relations defined in the real problem, since we can only observe those as revealed by the actions performed. If a static relation is universal, then all instances share the same groundings for that relation. This means that for universal statics, the input plans provide us with a much fuller picture of the relation.

Algorithm 3 Minimal static relation splitting algorithm. In this algorithm rank refers to the rank of the partition (the size of the largest block minus the total number of blocks).

```
function MSR_Split(P : a partition of the MSR)
    if preserveOptimality(P) = false then
        return ⊥
    end if
    min ← P
    for P′ ∈ refinement(P) do
        minP′ ← MSR_Split(P′)
        if evaluate(minP′) < evaluate(min) then
            min ← minP′
        end if
    end for
    return min
end function
function evaluate(P : a partition of the MSR)
    if P = ⊥ then
        return ∞
    else
        return rank(P)
    end if
end function
```

Intuitively, we collect all of the instances of the static relations found in the earlier stages of *LOP* and combine the groundings across instances. When we combine these ground instances there are two possible results: the combined relation preserves optimality (in this case we accept the relation as universal), otherwise the combination of the ground relation leads to shorter than optimal plans (in this case, of course, we reject the universality of the relation). The algorithm is presented as Algorithm 4.

Evaluation

In this section, we demonstrate the effectiveness of the *LOP* domain acquisition system on the planning benchmarks. The experimental setup is as follows: we used the Fast Downward (Helmert 2006) planner, with the LMCut heuristic (Helmert and Domshlak 2009) with a fifteen minute cutoff to find optimal plans to as many of the benchmark planning problems as possible. These form the optimal training plans needed for the static analysis. For benchmark instances which then have fewer than 10 optimal plans to learn from, we generate ten additional problems by sampling states in a random walk and choosing random pairs of states as the initial and goal states.

Experiment 1: Optimal *LOP* Plans

In this experiment, we learn the initial *LOCM2* state machines using both optimal and suboptimal plans. This leads to a greater coverage of transitions, and hence machines that match the domain much more closely. However, we also have a set of plans which we know to be optimal and we use these plans for the *LOP* analysis.

Algorithm 4 Universal static relation test. We assume a $\mathrm{preserveOptimality}$ function that works with an input relation here.

```
function UniversalStatics(R : a set of relations)
    U ← ∅
    for r ∈ R do
        if preserveOptimality(r) then
            U ← U ∪ {r}
        end if
    end for
    return U
end function
```

Experiment 2: Only Suboptimal Plans

In this experiment, we treat goal-directed plans that are not proven to be optimal the same way as we treat optimal plans in our previous analyses. For this, we use the LAMA planner (Richter and Westphal 2010), with a 120 second time cutoff, and we take the first plan produced. This experiment is to demonstrate whether or not our strong assumption of optimality is strictly necessary, or whether goal-directed plans suffice for detecting statics.

Discussion

Table 1 shows the results of both experiments. Underneath the *LOCM2* label, there are two columns, the first indicates whether or not *LOCM2* was able to correctly discover the dynamics of the domain. If not, this is typically due to one of the *LOCM* assumptions being violated. Where possible, we correct these flaws, so that we can still perform the *LOP* analysis. The results for the first experiment, only using an optimal planner, are shown in the three columns underneath the '*LOP* Optimal' heading. The results for the second experiment, using a satisficing planner, are shown in the three columns underneath the '*LOP* Satisficing' heading. The column that counts the number of errors counts either an incorrect minimal static relation, a misidentified universal relation or a missing static relation as an error. Note that there are no false positive results; *LOP* never reports a static that isn't there. Note also that the satisficing results are effectively the same as the optimal results.

Generally the results are very positive: of the domains that have static relations and had valid dynamic output from *LOCM2*, nine out of twelve had their static relations discovered error free. *LOP* can be seen as a *LOCM*-like system in that it produces an overly general result when incorrect. We now present a discussion of some of the more interesting results:

Driverlog In the Driverlog domain, the structure of both statics are detected correctly. These are the roads that the trucks drive on and the path that the drivers can walk on. The paths are incorrectly identified as universal statics. We are unsure whether this happens because the path maps are completely consistent with the input data, or that the path maps are very dense, leading to many alternative route.

Domain	Operators			Input Plans		*LOCM2*		*LOP* Optimal			*LOP* Satisficing		
	#	#Static	#Univ.	#Opt	#Sub	Valid	#St.	#St.	#Uni.	#Er.	#St.	#Uni.	#Er.
AoP Freecell	8	3	3	60	60	✓	1	2	2	0	2	2	0
Blocks	4	0	0	28	35	✓	0	0	0	0	0	0	0
Depot	4	2	0	17	19	✓	2	0	0	0	0	0	0
Driverlog	6	2	0	13	15	✓	0	2	1	1	2	1	1
Freecell[1]	10	10	10	14	16	✓*	8	-	-	-	3	1	6
Grid	5	2	2	12	15	✓	0	2	2	0	2	2	0
Gripper	3	0	0	7	20	✓	0	0	0	0	0	0	0
Logistics	6	1	0	20	28	✓	1	0	0	0	0	0	0
Miconic	4	4	2	141	92	✓	0	2	0	2	2	0	2
Mystery[2]	3	3	0	16	19	✓*	0	3	0	0	3	0	0
Parking	4	0	0	12	20	✓	0	0	0	0	0	0	0
Peg Solitaire	3	2	2	16	20	✓	0	2	2	0	2	2	0
Rovers	9	9	0	15	15	✗	-	-	-	-	-	-	-
Satellite	5	4	0	17	18	✗	-	-	-	-	-	-	-
Scanalyzer	4	4	4	15	13	✗	-	-	-	-	-	-	-
Sokoban	3	3	0	18	20	✓	0	3	3	0	3	3	0
Storage	5	5	2	15	14	✓	0	5	2	0	5	2	0
TPP	4	4	4	16	25	✗	-	-	-	-	-	-	-
Visitall	2	2	2	10	20	✓*	0	2	2	0	2	2	0
Zenotravel[2]	5	3	3	12	13	✓*	0	3	3	0	3	3	0

Table 1: Table of results running *LOP* on a collection of benchmark domains. Headings refer to # (number of operators) #Static (number of operators with static precondition) #Univ. (number of operators with universal preconditions) #Opt (number of optimal training plans) #Sub (number of suboptimal training plans) Valid (did *LOCM2* output complete dynamics) a star means the input had to be modified to meet the *LOCM2* assumptions #St. (number of operators for which statics are found) #Uni. (number of operators for which universal statics were found) #Er. (the number of errors made in all stages of *LOP*).

Peg Solitaire In the Peg Solitaire domain, *LOP* recovers all of the static information and determines the universal relations that encode the positions that make the rows.

Freecell and AoP Freecell Figure 4 shows the state machine derived by *LOCM2* for the number type that maintains which column is next available. This type exists only as a symmetry-breaking measure, ensuring that only exactly one column of cards can begin at any one time. In the original benchmark domain, a collection of static predicates describe the successor function for these numbers. The *LOCM2* induced domain encodes this information as a dynamic relation described by the state machine in Figure 4

Freecell is interesting in that the optimal version of *LOP* takes too long to complete (over 24 hours), but the suboptimal version does complete, albeit with several mistakes. All of these mistakes are with respect to the underlying number system. The detected statics are defining the card orders. It is possible that with more example plans, a more complete model would have been discovered.

AoP Freecell is an alternative encoding of Freecell, derived by *LOCM* from logs of people playing Freecell. This has been discussed in detail in (Cresswell, McCluskey, and West 2013), and (Cresswell and Gregory 2011) shows that *LOCM2* produces a correct encoding of the domain's dynamics. Perhaps due to the simplicity of the more natural set of actions and objects used, both versions of *LOP* are able to discover the static relations in this domain.

Miconic Miconic encodes both universal and non-universal static relations. The universal statics encode an

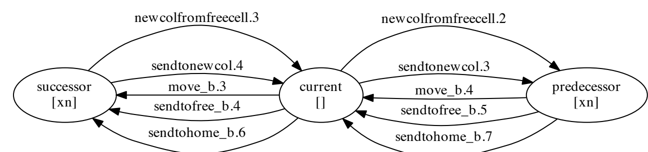


Figure 4: The *LOCM2* encoding of a propositionally encoded number. What was encoded as a static relation in the original benchmark is encoded dynamically in the *LOCM2* derived domain.

[1]The plans for the benchmark Freecell domain often have duplicate objects in actions. This violates the *LOCM* assumption that each object makes a single transition per action. This arises due to the fact that integers are encoded as objects and are used to number the cards and the spaces. We simply provide two sets of objects, one for counting cards and the other for counting positions, both mirroring the original number objects.

[2]In Zeno Travel, *LOCM2* unfortunately fails to correctly induce the dynamics of the domain. This violates one of the *LOP* assumptions. *LOCM2* fails to identify the fuel level state parameter in the plane's state machine. In order to demonstrate the performance of *LOP* we correct this flaw in the *LOCM2* output manually. In the Mystery domain, there is an equivalent problem with the *LOCM2* analysis, fixed in the same way.

ordering relation over the floors. The origin and destinations of the passengers are encoded as statics. *LOP* successfully detects the origin and destinations of the passengers (in the board and depart actions) but fails to detect the relations between the different floors (in the up and down actions). This failure is due to the fact that all that happens by allowing more groundings for the up and down actions is that an up action can now move the lift down (and vice versa). This will never lead a shorter plan, and so even an empty static relation for up and down will preserve optimality.

Miconic demonstrates an example of when *LOP* fails to discover a static relation. However, it is difficult to know if this presents a meaningful problem: since all floors are accessible at all times, it can be argued that the only reason to have separate up and down actions is as a presentational device for the plan.

Zeno Travel Zeno Travel encodes the possible transitions of the fuel level of the planes as universal static relations. These restrict the groundings of the fly, zoom and refuel operators. *LOP* successfully induces these relations in their scope and universality, and can be said in a way to improve on the original domain. In the original domain, the zoom operator:

$$\mathrm{zoom}(\mathrm{plane}, \mathrm{loc1}, \mathrm{loc2}, \mathrm{f3}, \mathrm{f2}, \mathrm{f1})$$

encodes that the plane in question uses two units of fuel. The preconditions encode this as the two predicates $(\mathrm{next\ f3\ f2}), (\mathrm{next\ f2\ f1})$. The static relation identified by *LOP* is between the two parameters f3 and f1 and ignores f2. This may initially be viewed as a mistake, but on closer inspection, the universal relation identified encodes the $\mathrm{n_plus_two}$ relation. The parameter f2 is, in fact, redundant.

Satellite and Rovers Satellite fails in the *LOCM2* analysis as it contains dynamic many-to-many relations, which cannot be represented in a *LOCM* state machine. The particular relation is added by the take-image operator:

$$(\mathrm{have_image\ ?direction\ ?mode})$$

There is a similar problem in Rovers, where images can also be taken using different camera modalities. In order to avoid this type of problem, the direction and the mode objects would need to be combined to form a single direction_mode object.

Summing up, our empirical analysis demonstrates that the *LOP* algorithm is effective at discovering static relations for a wide range of problems. We have also demonstrated domains in which *LOCM2* fails to discover the correct domain dynamics, possibly suggesting new research direction.

Related Work

Within the planning literature, there are many domain model acquisition systems. These systems each have varying levels of detail in their input observations. *LOCM*-derived systems use a minimal amount of input (only plan traces) whereas most other systems use predicates, initial and goal states and possibly intermediate states. The comprimise is that the target language in *LOCM* is simpler than many other systems. The *Opmaker2* system (McCluskey et al. 2009; Richardson 2008) learns models in the target language of OCL (McCluskey and Porteous 1997) and requires a partial domain model, along with example plans as input. The ARMS system (Wu, Yang, and Jiang 2007), can learn STRIPS domain models with partial or no observation of intermediate states in the plans, but does at least require predicates to be declared. The LAMP system (Zhuo et al. 2010) can target PDDL representations with quantifiers and logical implications. Systems that learn planning models in the presence of noisy and incomplete data (Mourao et al. 2012) have also been studied. Other types of learning include (Mehta, Tadepalli, and Fern 2011) considering the task of learning a single state space with no input plans, but with a system by which an oracle can validate plan hypotheses. A form of transfer learning has been considered (Zhuo et al. 2011) where action schema are constructed via a combination of analysing exisiting domains and using web queries to match operator names.

Outside of the planning literature, model acquisition is also of interest. In the constraint programming literature (O'Sullivan 2010; Bessiere et al. 2014), for example, constraint model acquisition is performed as an interactive task between the domain modeller and the acquisition system. In general game playing (Björnsson 2012) it has been observed that in addition to being of use in learning game rules, model acquisition systems can be of use in translating between formalisms. In the area of computer security, automata learning (Aarts, De Ruiter, and Poll 2013) has aided in learning models of system protocols, for example.

Conclusions and Future Work

Domain model acquisition is useful whenever a knowledge engineer has access to a controllable system that acts, but for which the structure of those actions has not been formally specified. Simple observation can provide a means to discover this formal structure.

We have presented a solution to the problem of domain model acquisition under the presence of static relations, relying on only plan traces as input. The *LOP* algorithm relies on the quality of the input plans serving as a guide to refining an overly general model generated by the *LOCM2* system. We have presented the *LOP* algorithm as using optimal plans as input. However, as our empirical analysis shows, even sub-optimal goal-directed plans are typically sufficient for the detection of the types of statics present in the benchmark domains. In fact, due to the relative efficiency of satisficing planners, the sub-optimal version of the system has better domain coverage than the optimal version.

There will always remain a trade-off between the amount of information contained in the input to domain model acquisition systems, and the expressiveness of the target language for which models can be learnt. For example, the *LOCM* family of algorithms at present cannot correctly learn domains which contain dynamic many-to-many relationships (such as a dynamic road network, for example). Our future efforts will be to explore this boundary even more closely.

References

Aarts, F.; De Ruiter, J.; and Poll, E. 2013. Formal Models of Bank Cards for Free. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, 461–468. IEEE.

Bessiere, C.; Coletta, R.; Daoudi, A.; Lazaar, N.; Mechqrane, Y.; and Bouyakhf, E. H. 2014. Boosting Constraint Acquisition via Generalization Queries. In *European Conference on Artificial Intelligence*.

Björnsson, Y. 2012. Learning Rules of Simplified Boardgames by Observing. In *European Conference on Artificial Intelligence*, 175–180.

Cresswell, S., and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In *International Conference on Automated Planning and Scheduling*, 42 – 49.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2009. Acquisition of object-centred domain models from planning examples. In Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds., *International Conference on Automated Planning and Scheduling*. AAAI.

Cresswell, S.; McCluskey, T.; and West, M. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review* 28(2):195 – 213.

Helmert, M., and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *International Conference on Automated Planning and Scheduling*, 162–169.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

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

McCluskey, T. L.; Cresswell, S. N.; Richardson, N. E.; and West, M. M. 2009. Automated acquisition of action knowledge. In *International Conference on Agents and Artificial Intelligence (ICAART)*, 93–100.

Mehta, N.; Tadepalli, P.; and Fern, A. 2011. Autonomous learning of action models for planning. In *Advances in Neural Information Processing Systems*.

Mourao, K.; Zettlemoyer, L.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS Operators from Noisy and Incomplete Observations. In *Uncertainty in Artifical Intelligence*, 614 – 623.

O'Sullivan, B. 2010. Automated Modelling and Solving in Constraint Programming. *AAAI Conference on Artificial Intelligence* 1493–1497.

Richardson, N. E. 2008. *An Operator Induction Tool Supporting Knowledge Engineering in Planning*. Ph.D. Dissertation, School of Computing and Engineering, University of Huddersfield, UK.

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

Wu, K.; Yang, Q.; and Jiang, Y. 2007. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *The Knowledge Engineering Review* 22(2):135–152.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artificial Intellgence* 174:1540–1569.

Zhuo, H.; Yang, Q.; Pan, R.; and Li, L. 2011. Cross-Domain Action-Model Acquisition for Planning via Web Search. In *International Conference on Automated Planning and Scheduling*, 298–305.