# Utilizing Automatically Inferred Invariants in Graph Construction and Search

**Maria Fox and Derek Long**
University of Durham, UK
D.P.Long@dur.ac.uk,Maria.Fox@dur.ac.uk

**Abstract**

In this paper we explore the relative importance of persistent and non-persistent mutex relations in the performance of Graphplan-based planners. We also show the advantages of pre-compiling persistent mutex relations. Using TIM we are able to generate, during a pre-processing analysis, all of the persistent binary mutex relations that would be inferred by Graphplan during graph construction. We show how the efficient storage of, and access to, these pre-processed persistent mutexes yields a modest improvement in graph construction performance. We further demonstrate that the process by which these persistent mutexes are identified can, in certain kinds of domain, allow the exploitation of binary mutex relations which are inaccessible to Graphplan. We present *The Island of Sodor*, a simple planning domain characterizing a class of domains in which certain persistent mutexes are present but are not detectable by Graphplan during graph construction. We show that the exploitation of these hidden binary mutexes makes problems in this kind of domain trivially solvable by STAN, where they are intractable for other Graphplan-based planners.

## Introduction

Graphplan (Blum & Furst 1995) has become an important example of efficient classical planning technology. The algorithm operates by constructing a data structure, the planning graph, which is a compressed form of the reachable states in a planning problem, and then by searching backwards using an iterated depth-first search to explore the reachable states and find a plan. An important elements in this algorithm is that the compression of reachable states ensures that the plan graph is feasible to construct, even for quite complex planning problems. The price paid for the compression is that the data structure loses differentiation between pairs of reachable states, by taking the union of all the collections of propositions in states reachable by equal length plans. This differentiation is not entirely lost, however, since the algorith maintains a collection of *binary mutex* relations between pairs of facts showing which pairs cannot be true together in the same state (that is, they are mutually exclusive). The algorithm uses these fact mutex relations to induce similar mutex relations between actions which might be applied to progress from one state to another. This paper explores the nature of the binary mutex relations constructed by Graphplan-based planners and differentiates between *persistent* and *non-persistent* mutex relations.

It has been observed before (Smith & Weld 1999; Long & Fox 1999) that mutex relations constructed by Graphplan behave monotonically – they can first occur between a pair of facts at the first stage at which the two facts are both reachable, and will then either remain mutex at each subsequent level, or else lose the mutex relationship with one another. Two facts can never start non-mutex and subsequently become mutex since if they can occur in the same state then they will always be able to appear in the same state, if only by virtue of persistence of that state. *Persistent* mutexes are mutual exclusion relations which can be identified to hold between pairs of propositions, or pairs of actions, and which do not get removed as the graph develops. These relations capture aspects of the logical structure of the domain. They include permanent mutex relations indicating conflicts between the preconditions and effects of actions (referred to here as PAD-mutexes). *Non-persistent* mutexes express mutual exclusion relations which decay as the graph lengthens and capture the temporal requirements of solutions to given goal combinations. That is, certain pairs of facts can only occur in the same state once some combination of events has been brought about, so that each fact may be made separately true much sooner than the combination can be achieved: this situation can yield a non-persistent mutex relation between these facts between the first state at which both are separately achievable and the state at which they may both be achieved together.

We have performed a number of experiments to de-

termine the relative significance of these different types of relation. Our experiments indicate that persistent mutexes tend to predominate, often by a very large margin, in the standard benchmark domains. We constructed a version of STAN in which only persistent mutexes were built during graph construction, and found that failure to even construct non-persistent mutexes had virtually no effect on planning performance on many problems in these domains. We found that PAD-mutex relations are not the critical ones in determining the solvability (or otherwise) of problems. The most significant class of mutex relations appears to be the class of persistent, non-PAD, mutex relations.

In this work we have been interested in identifying what kind of mutex relations Graphplan-based planners benefit from the most and whether there are inferrable mutual exclusions that are not accessible to the Graphplan graph-construction process. An initial hypothesis was that the persistent non-PAD mutex relations could be pre-compiled into an efficient representation, to avoid their repeated recomputation in the graph construction phase, and that this would yield an improvement in graph construction performance. Since profiling shows that over fifty per cent of resources are typically spent in graph construction it appeared that this could be quite significant. A second hypothesis was that a class of domains could be constructed in which the pre-compilation of these mutex relations could make binary mutexes accessible which would not otherwise be accessible to Graphplan-based planners. We constructed a range of experiments to test these hypotheses, the results of which are reported in detail below. In summary, we have been able to demonstrate that the use of pre-compiled mutex relations, whilst having only a limited advantage for graph construction, can have a massive impact on search, making the difference between problems being solvable or not solvable in a certain class of domains.

## Pre-compiling Persistent Mutex Relations

Persisent mutex relations correspond to logically invariant properties of the domain. We are able to pre-compile the non-PAD mutexes using the TIM system (Fox & Long 1998), which automatically infers a collection of invariants from STRIPS domain descriptions using a collection of static analysis techniques. As we describe in (Fox & Long 1998) TIM is able to infer four different kinds of invariant: *identity invariants* (for example, no two objects can be *at* the same place at the same time), *unique state invariants* (every object must be in at most one place at any one time), *state membership invariants* (every object must be in at least one place at any one time) and *resource invariants* (there are exactly 4 surfaces in the 3-blocks world). The first two of these correspond to *intensional* representations of collections of persistent mutex relations. For example, the (indicative) identity invariant above is an intensional representation of the collection of mutex relations that would be inferred to hold between *at(a,d)* and *at(b,d)*, *at(a,d)* and *at(c,b)*, etc., for distinct $a, b$ and $c$ within a layer of the graph.

The inferred invariants can be used to compile, during the pre-processing stage, a look-up table of persistent mutual exclusions that can be consulted very efficiently to determine persistent mutex relationships during the graph construction process. The identification of these mutexes as persistent means that there is never any need to reconsult the look-up table with respect to a given pair of propositions, or actions, once they have been identified as persistently mutex. In standard Graphplan there is no distinction between persistent non-PAD and non-persistent mutex relations. Thus Graphplan builds, at each layer of the graph, an extensional representation of both the invariant and time-dependent structures of the domain without distinguishing between the two forms of binary relation. This means that the same persistent mutex relations are repeatedly re-inferred, which is unnecessarily expensive. Analysis of the standard domains (Gripper, Logistics, Mystery, etc.) revealed that Graphplan infers all of the mutexes that follow from each of the four forms of invariant inferred by TIM, so it might be concluded that there is little, other than a small saving in graph construction, to be obtained from deriving them during pre-processing. However this is, in fact, not the case.

In our experiments we have observed that non-persistent mutex relations can decay prematurely during the Graphplan graph construction process. A simple example of this is a domain in which there are three cities, two of which have packages located at them and the third of which has a truck located at it. If the goal is to have both of the packages at the third location, then the shortest plan involves six steps: three sequential moves, two sequential loads and two parallel unloads. It is impossible for the goal to be achieved in fewer steps. However, Graphplan infers that the goals can be achieved together by level 5 of the graph. This is because the preconditions for the two packages to be both at the same destination by level 5 are that they both be in the truck at level 4 and that the truck be at the destination. There is a 3 step plan which will get the truck, loaded with either package, to the destination. There is also a 4 step plan which will have the truck loaded with both pack-

ages, although not at the destination. This means that Graphplan will not find any mutex relations between any *pair* of the preconditions required for the packages to be unloaded at their destination by level 5. The problem lies in the fact that the binary mutex relation between the packages both being at the destination by level 5 rests on a *ternary* "mutex" relation between the facts *in(package1,truck)*, *in(package2,truck)* and *at(truck,destination)*, which Graphplan does not detect. This requires Graphplan to begin searching for a plan from level 5 when it is impossible for one to be found until level 6 has been constructed. Graphplan will discover the mutex relation by searching the graph and failing to find a plan, which is a much more expensive route to finding the relation than to construct it from the outset. This example is very simple and the penalty for Graphplan is very small, but in larger examples we have found that Graphplan invests significant effort in finding mutex relations which it failed to discover by construction. Our observation that premature decay can occur posed the question of whether non-PAD persistent mutexes might also apparently decay in the Graphplan plan graph. Graphplan treats these mutexes as though they can decay so it seemed in principle possible, even though we were unable to identify a benchmark domain in which this occurred. This led us to work on the construction of a domain which would force Graphplan to lose non-PAD persistent mutexes and have to resort to search to rediscover them.

The Island of Sodor domain, described below, provides an example of such a domain. Because of its exploitation of the pre-compiled mutex relations derived from TIM's invariants STAN finds problems in this kind of domain trivial, although they are intractable for other GraphPlan-based planners.

## Storing and Accessing Persistent Mutex Relations

In order to infer invariants TIM first constructs finite state machines, called *property spaces*, which capture the behaviour of the distinct types of objects that can traverse the states within them. The process by which this is achieved is described in detail in (Fox & Long 1998) and sumarised very briefly here.

Central to the analysis conducted by TIM is the notion of a *property*, which is formed by combining a predicate and an index to one of its argument positions. This property is said to *belong* to values which can occupy the corresponding position in some valid proposition in the domain. TIM begins its analysis by projecting the behaviour of each operator schema for each argument that the schema contains. That is, the

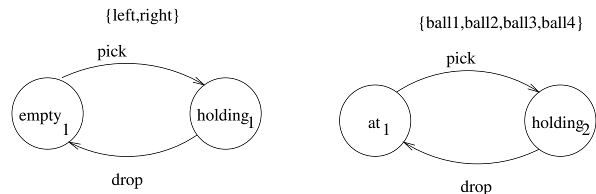


Figure 1: FSMs built by TIM for the Gripper domain.

preconditions and postconditions of each schema are examined and, for each argument, the properties that the object has in the pre- and postconditions of the schema are identified. An operator therefore generates *rules* for each of its arguments, indicating how the properties of the objects to which they refer change as a consequence of application of the operator. A rule will specify the properties the object loses and which it gains. There can also be properties which merely *enable* the transition, without being affected by it. Some rules allow properties to be gained without a corresponding loss, or to be lost without a corresponding gain: these rules are called *attribute rules*. Attribute rules and the properties they affect are ignored in the analysis discussed here, although they can have other roles in analyses carried out by TIM. Properties which are only affected by non-attribute rules defined small finite-state machines (FSMs) in which the states are bags of properties held by objects in reachable states of the planning domain. The rules are the transitions between these states. Subsets of the objects in the planning domain will make transitions on FSMs according to whether they start, in the initial state, in one of the legal states associated with the FSM. A simple example of FSMs built by TIM for the Gripper domain (a simple domain in the AIPS'98 competition data set) is shown in Figure 1.

It can now be observed that two propositions are persistently mutex if they are associated with properties that appear in the same FSM but their predicates never occur as properties of the same object in the same state. Any object which traverses an FSM cannot simultaneously have two properties that appear in the FSM, but never in the same state. For example, in Gripper, balls can make the transition from being *at* a room to being *held* in a gripper ($holding_2$), but the properties $at_1$ and $holding_2$ can never be simultaneously true of any one ball. Thus, *at* and *holding* are persistently mutex when they are talking about the same ball.

During the pre-processing analysis performed by TIM a collection of persistent mutex relations is con-

```
create a square matrix, M, of lists, initially all empty,
        with dimension equal to the number of
        dynamic predicates in the language;

for each property space, P,
        for each pair of properties in P, p and q,
                if [p, q] is not included in any
                                state in P
                then    split p into its predicate,
                                pp, and its index, pi;
                        split q into its predicate,
                                qq, and its index, qi;
                        add (pi, qi) to the list
                                at M[pp, qq];
                        add (qi, pi) to the list
                                at M[qq, pp];
```

Two facts, $A = a(x_1, ..., xn)$ and $B = b(y_1, ..., y_m)$, are permanently mutex if:

not $(A = B)$ **and** for some entry, $(i, j)$, in $M[a, b]$, $x_i = y_j$

Figure 2: The pseudo-code for generation and use of the mutex matrix in TIM.

structed. This collection is stored in a matrix and when STAN compares two propositions to determine whether they are mutex this matrix is indexed into by the labels of the two propositions being compared. Each cell in the matrix stores a list of pairs, each pair containing the corresponding arguments that must be the same for the two propositions to be mutex. There is a list because there may be alternative ways in which two propositions can be mutex. The arguments referenced by a pair might be in the same position in the two propositions, as in the pair $(1, 1)$ (this will be the case, for example, for mutex relations inferred from identity invariants), but they need not be. For example, as discussed above, the properties $at_1$ and $holding_2$ are persistently mutex (in the Gripper domain) if they refer to the same ball. Thus, the matrix indexed at $[at, holding]$ contains the pair $(1, 2)$, since the ball appears in the first argument position in *at* and in the second position in *holding*. On the other hand, packages in Logistics can be either $at_1$ or $in_1$, and these states are mutually exclusive. TIM infers the pair $(1, 1)$ for the matrix entry indexed at $[at, in]$ in this case. Thus, comparing two propositions to see if they are mutex involves consulting the matrix and then comparing the arguments at the designated positions. Figure 2 contains a pseudo-code algorithm for constructing the matrix from the property spaces built by TIM in the first phase of its analysis.

The construction of the matrix is done during the preprocessing phase and requires time $O(n^4)$ in the worst case, where $n$ is the size of the domain encoding. This follows because the number of properties, identified by TIM, is linear in the size of the domain encoding and, in the worst case, for each of the $O(n^2)$ pairs of properties there is at most $O(n^2)$ work to determine whether they appear in the same bag within their identified property space. In practice the process is very fast, requiring only a few miliseconds. This is because there are generally few properties in each space, dramatically reducing the cost of the analysis. For example, if the size of the property spaces is bounded by a constant (which seems typical) the analysis only requires time linear in the size of the domain encoding. This analysis is not unrealistic since, as domain complexity increases, it is typical that type differentiation improves and the individual behaviours remain at a similar level of complexity.

Others have examined the automatic generation of domain invariants, including (Gerevini & Schubert 1998; 1996a; 1996b; Morris & Feldman 1989) and (Kelleher & Cohn 1992). Some of this work has been exploited in the construction of data structures encoding domains in preparation for planning: (Gerevini & Schubert 1998) and (Gerevini & Schubert 1996a) in particular explore this. Graphplan has been seen as a different way of constructing domain invariants, using an extensional rather than intensional representation. The work presented here can be seen, in part, as a comparison of the effectiveness of Graphplan's inference strategy and that used by TIM.

## The Island of Sodor

It is straightforward to construct domains in which objects can enter collections of states, characterised by groups of properties, in which any pair of properties can appear together, in a state, but the whole group cannot appear together in any state. This situation gives rise to multiple mutual exclusions between collections of the properties in the group, even though there are no binary mutex relations between the properties in these collections. The simplest example of this situation is the one depicted in Figure 3. Here, objects can move between the states $[p]$ and $[q, r]$, $[p]$ and $[q, s]$ and $[p]$ and $[r, s]$ but the properties $q$, $r$ and $s$ are ternary mutex. The ternary mutex relation obscures the fact that, starting from an initial state in which no object has more than one $p$-derived property, no object that traverses this state space can have more than one $p$-derived property at any time, since $p$-derived properties are always exchanged for pairs of the other properties. In fact, Graphplan is unable to

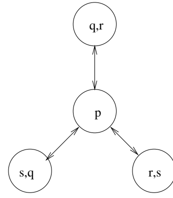


Figure 3: The presence of ternary mutex relations.

maintain the binary mutual exclusion between distinct occurrences of $p$ in the fact layers built as the graph develops because it is unable to recognise the effect of this ternary mutex relation. In order to experiment with this example we designed the Island of Sodor domain, the simplest domain we could identify that exhibits the ternary mutual exclusions just described.

The Island of Sodor[1] is a STRIPS planning domain in which a number of engines transport cargos between locations on a rail network. The rail network is a simple topography of one-way rails. The locations include the *Top Station*, the *quarry*, the *docks*, *Gordon's Hill* and several others. Engines can be involved in transportation tasks or they can be in the process of being cleaned, refuelled and rewatered. There are only sufficient personnel available to enable any two of these operations to be performed for each engine at any time, so an engine can be cleaned and refuelled, cleaned and rewatered or refuelled and rewatered. When these operations are in progress the engine is not recorded as being *at* any of the locations on the rail network. It is taken off-line for these maintenance operations and brought back on-line again at their conclusion. Figure 4 presents seven of the nine operators of this domain and Figure 5 shows an example initial state of the rail network.

The domain was constructed to demonstrate the way in which certain persistent mutex relations are lost in the graph construction phase of Graphplan. The operator schemas consist of a *move* schema which moves engines from $a$ to $b$, three maintenance schemas, *clean-and-refuel*, *clean-and-rewater* and *refuel-and-rewater*, which take engines off-line in order to *clean* and *refuel*, *clean* and *rewater* or *refuel* and *rewater* them (respectively) and three corresponding *recommission* schemas, which take engines from maintenance to being on-line at any location (and therefore *at* that location). In the initial state engines are placed at the various stations on the rail network, or are off-line. In the plan

[1] The Island of Sodor is the setting of a famous series of children's books, involving anthropomorphic steam engines.

```
(:action move
   :parameters (?engine ?from ?to)
   :precondition (and (inServiceAt ?engine ?from)
                      (rail-link ?from ?to))
   :effect (and       (inServiceAt ?engine ?to)
                      (not (inServiceAt ?engine ?from))))

(:action load
   :parameters (?engine ?cargo ?location)
   :precondition (and (inServiceAt ?engine ?location)
                      (at ?cargo ?location))
   :effect (and       (in ?cargo ?engine)
                      (not (at ?cargo ?location))))

(:action unload
   :parameters (?engine ?cargo ?location)
   :precondition (and (inServiceAt ?engine ?location)
                      (in ?cargo ?engine))
   :effect (and       (at ?cargo ?location)
                      (not (in ?cargo ?engine))))

(:action clean-and-refuel
   :parameters (?engine)
   :precondition (inServiceAt ?engine top-station)
   :effect (and       (refuelling ?engine)
                      (cleaning ?engine)
                      (not (inServiceAt ?engine top-station))))

(:action clean-and-rewater
   :parameters (?engine)
   :precondition (inServiceAt ?engine top-station)
   :effect (and       (rewatering ?engine)
                      (cleaning ?engine)
                      (not (inServiceAt ?engine top-station))))

(:action refuel-and-rewater
   :parameters (?engine)
   :precondition (inServiceAt ?engine top-station)
   :effect (and       (rewatering ?engine)
                      (refuelling ?engine)
                      (not (inServiceAt ?engine top-station))))
(:action recommission
   :parameters (?engine)
   :precondition (and (refuelling ?engine)
                      (cleaning ?engine))
   :effect (and       (inServiceAt ?engine top-station)
                      (not (refuelling ?engine))
                      (not (cleaning ?engine))))
```

Figure 4: The Island of Sodor operators schemas. Only one of the operators for recommissioning is given here - the other two are symmetric with this one, but using the other pairs of services available to an engine.

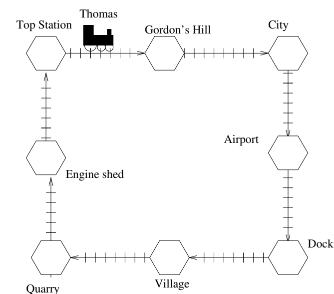


Figure 5: The Island of Sodor - a typical configuration

Figure 6: The initial section of the plan graph obtained from an Island of Sodor instance in which there is a single engine and adjacent locations *Top-Station* and *Gordons-Hill*.

graph constructed from an initial state in which there is a single engine, *Thomas*, at the *Top-Station*, the fact pairs *at(Thomas,Top-Station)* and *cleaning(Thomas)*, *at(Thomas,Top-Station)* and *rewatering(Thomas)* and *at(Thomas,Top-Station)* and *refuelling(Thomas)* will be binary mutex at fact layer 1. No other pairs of facts will be mutex at this level. The initial section of Figure 6 depicts the state of the plan-graph at layer 1. At the next level the fact *at(Thomas,Top-Station)* can be obtained by re-commissioning following cleaning and refuelling. The facts *refuelling(Thomas)*, *rewatering(Thomas)* and *cleaning(Thomas)* can be maintained by no-ops, as shown in Figure 6. Finally, at layer 3, *at(Thomas,Gordons-Hill)* can be obtained by a move from the Top Station (assuming that they are adjacent in the initial state), and *at(Thomas,Top-Station)* can be obtained by re-commissioning following refuelling and rewatering. Now the two *at* relations are binary non-mutex. Graphplan has lost the persistent mutual exclusion between *at* relations because the three-way relationship between *cleaning*, *rewatering* and *refuelling* hides the binary exclusion that holds between the (implicit) compound elements *cleaning-and-rewatering* and *inServiceAt*, *refuelling-and-rewatering* and *inServiceAt* and *cleaning-and-refuelling* and *inServiceAt*. STAN cannot recognise these hidden binary relations either, but STAN has access to the graph-independent invariant structure of the domain which allows it to retain the knowledge that all pairs of distinct *at* relations are binary mutex.

## Empirical Results

We performed a range of experiments on a collection of benchmark domains (the AIPS-98 competition set) and on our Island of Sodor domain in order to test our initial hypotheses. All experiments were performed under Linux RedHat 6.0 on a 500 MHz Celeron PC with 128 Mb of RAM. We constructed four versions of STAN for comparison: STAN is our most recent release (version 4). STAN- is STAN without the pre-compilation of persistent mutexes. STAN-NP does not construct non-persistent mutex relations, leaving it to the searching phase to identify these constraints. STAN+PAD constructs only PAD-mutexes, leaving search to identify all other constraints.

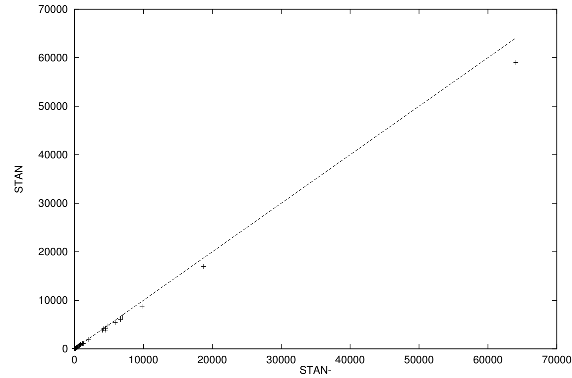


Figure 7: Modest advantage obtained from precompiled persistent mutexes in the AIPS-98 competition Logistics and Mystery problems.

We compared STAN and STAN-, on a collection of problems from the AIPS-98 competition domains, to determine the extent to which exploitation of precompiled persistent mutexes affects the performance of the graph construction phase. Figure 7 shows that a modest, but consistent, advantage is obtained in two standard benchmark domains. Tests were carried out on other domains showing similar performance. We have not been able to find any problems in the benchmark domains for which there is a significant impact on overall performance deriving from the graph construction saving. STAN- uses the conventional Graphplan strategy of recomputing persistent non-PAD mutexes at each layer in the graph. Since Graphplan is able to construct, extensionally, all of the mutex relations available for extraction from the TIM invariants the modest scale of the saving obtained by STAN is unsurprising. There is no possible advantage to be gained over STAN- during search. The saving that is obtained is explained by the work that no longer needs to be done on recomputation of this class of mutex relations. However, the intensional representation of these mutexes must still be built, so that the basic overhead incurred by their construction is similar in both STAN- and STAN.

Figure 8 demonstrates the effects of not building non-persistent mutex relations during graph construction. We compared STAN-NP with STAN on the same benchmark domains and problems. For most of the problems considered we found that no significant disadvantage was suffered by STAN-NP. For example, all of the competition Gripper problems remained solvable with no noticeable effect on performance. Examination of the relative proportions of persistent to

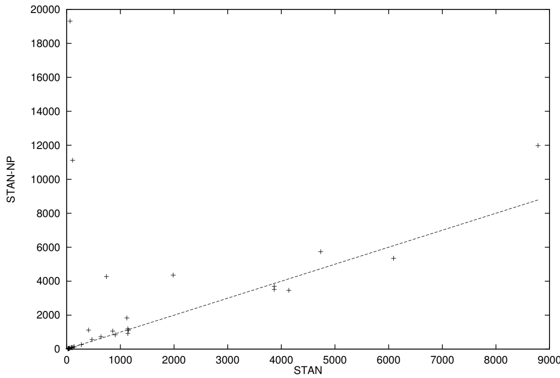


Figure 8: STAN compared with STAN-NP showing that performance remains broadly similar when non-persistent mutexes are not inferred in the construction phase.

non-persistent mutexes constructed by STAN revealed that non-persistent mutexes make up only a very small proportion of the inferred mutex relations, typically accounting for fewer than five per cent of the whole mutex collection. It therefore seems plausible that the extra search overhead incurred by not building these exclusions during graph construction could be insignificantly small. However, in a few cases (two problems in Logistics and six in Mystery) the failure to build non-persistent mutexes made problems unsolvable. This is very difficult to explain, given that the relative proportions of non-persistent mutexes remain as low as in the other problems. Further, we encountered one problem (Logistics problem 5 from round 2) which STAN-NP solved faster than STAN having constructed *fewer* non-persistent mutexes during search than STAN constructed, despite STAN having built the accessible non-persistent mutexes during graph construction. STAN identified a further 519 mutexes in the search phase, finding a plan in 59 seconds, whilst STAN-NP identified only 433 and found the same plan in 33 seconds. It must be noted that the extra mutexes are binary bad goal sets found by Kambhampati's EBL/DDB mechanism (Kambhampati 1999) which we implemented in (both versions of) STAN. Non-binary bad goal sets are ignored. Figure 9 indicates the number of non-persistent mutex relations that are built during the search phase by both STAN and STAN-NP.

As can be seen in the table, the number of non-persistent mutexes found during search by STAN-NP is very small - indeed, much smaller than the number of non-persistent mutexes built by STAN during the graph construction phase. Thus, much of the work done by STAN in this respect is wasted, because many irrelevant mutexes are constructed for each important one. STAN-NP often benefits from not having to construct these irrelevant mutexes, as can be seen from the performance figures. Our implementation of STAN-NP was not optimized for taking advantage of the reduced obligation during graph construction so it is possible to anticipate a much greater saving than the figures already suggest. Since STAN is finding additional non-persistent mutexes during search in four of the Logistics problems (although none of the Mystery or Gripper problems) it can be concluded that there is scope for improving the coverage, by Graphplan-based planners, of the non-persistent mutexes during the construction phase. As a final observation it is interesting to note that the performance of STAN-NP on identifying unsolvable problems is significantly worse than that of STAN. This suggests that building non-permanent mutexes during graph construction is of particular importance in identifying unsolvable problems without search.

| Problem | STAN | SBs | STAN-NP | SBs |
|---|---|---|---|---|
| 1.1 | 138 | 0 | 134 | 13 |
| 1.2 | 1148 | 0 | 1110 | 33 |
| 1.3 | 5456 | 0 | – | – |
| 1.4 | 16973 | 0 | – | – |
| 1.5 | 858 | 42 | 1064 | 86 |
| 1.7 | 3867 | 1 | 3517 | 35 |
| 1.11 | 1139 | 0 | 1196 | 8 |
| 1.17 | 6093 | 0 | 5347 | 13 |
| 2.1 | 38 | 0 | 37 | 8 |
| 2.2 | 38 | 0 | 37 | 11 |
| 2.3 | 100 | 0 | 92 | 16 |
| 2.4 | 4140 | 26 | 3464 | 43 |
| 2.5 | 59002 | 519 | 32869 | 433 |

Figure 9: STAN compared with STAN-NP on Logistics problems from rounds 1 and 2. The third and fifth columns indicate the number of non-persistent mutexes found, by STAN and STAN-NP respectively, during the search phase.

Figure 10 shows the effects of only building PAD-mutex relations during graph construction, leaving all other constraints to be discovered during the search phase. As might be expected, the performance of STAN+PAD is very poor, except on trivial problems where it seems not to suffer significantly. Of the 29 Mystery problems solved (or proved unsolvable) by STAN only 12 were solved (or proved unsolvable) by STAN+PAD. Of the 13 Logistics problems only 4 were solved by STAN+PAD. It is of interest to observe that the number of mutex relations (of all non-PAD kinds) identified during search is consistently very low. This does not imply that the thousands of mutexes built

| Problem | STAN | STAN+PAD | Search binaries |
|---|---|---|---|
| Log 1.1 | 138 | 27618 | 207 |
| Log 2.1 | 38 | 40 | 58 |
| Log 2.2 | 38 | 41 | 55 |
| Log 2.3 | 100 | 268 | 76 |
| Myst 1 | 23 | 22 | 15 |
| Myst 3 | 104 | 86 | 11 |
| Myst 7* | 36 | 37 | 0 |
| Myst 9 | 87 | 128 | 55 |
| Myst 11 | 50 | 507 | 80 |
| Myst 18* | 1140 | 821 | 0 |
| Myst 19 | 639 | 1974 | 103 |
| Myst 25 | 25 | 23 | 6 |
| Myst 26 | 471 | 997 | 147 |
| Myst 27 | 131 | 118 | 16 |
| Myst 28 | 31 | 72 | 62 |
| Myst 29 | 62 | 56 | 14 |

Figure 10: STAN+PAD on Logistics and Mystery problems. The starred problems are provably unsolvable.

by STAN in its construction phase are useless. Indeed, these constrain search sufficiently to make some problems solvable that are not solvable when the search is unconstrained. Even in cases where STAN+PAD solves the problem the inference of mutexes in the construction phase, by STAN, reduces the size of the search space that must be explored by STAN to discover the same information. Our observations lead us to conclude that PAD-mutexes have a less pronounced effect on performance than non-PAD persistent mutexes, since the inclusion of the latter has such a dramatically positive effect.

Figure 11 demonstrates the advantages obtained by STAN over STAN- in the Island of Sodor domain. The advantage is derived from the pre-compilation of non-PAD mutex relations which prevents them from being obscured by the ternary mutexes, exhibited by the domain, which cannot be identified by either planner. In STAN- the binary mutex relations that persist between pairs of *at* goals are lost early in the graph construction process, so that STAN- is quickly submerged in search. On the other hand, STAN is always able to access the pre-compiled information that distinct *at* goals are persistently binary mutex.

## Conclusions

| Problem | STAN- | STAN |
|---|---|---|
| 1 | – | 166 |
| 2 | 45 | 38 |
| 3 | 534 | 78 |
| 4 | 45 | 19 |
| 5 | – | 33 |
| 6 | 60 | 15 |
| 7 | 9008 | 24 |
| 8 | 584 | 19 |
| 9 | – | 746 |
| 10 | – | 315 |
| 11 | – | 427 |
| 12 | – | 352 |

Figure 11: STAN compared with STAN- on problems in the Island of Sodor domain.

## References


Blum, A., and Furst, M. 1995. Fast Planning through Plan-graph Analysis. In *IJCAI*.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *JAIR* 9.

Gerevini, A., and Schubert, L. 1996a. Accelerating Partial Order Planners: Some Techniques for Effective Search Control and Pruning. *JAIR* 5:95–137.

Gerevini, A., and Schubert, L. 1996b. Computing Parameter Domains as an Aid to Planning. In *AIPS-96*.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proceedings of AAAI-98, Madison, WI.*

Kambhampati, S. 1999. Improving Graphplan's search with EBL and DDB techniques. In *Proceedings of IJCAI-99.*

Kelleher, G., and Cohn, A. 1992. Automatically Synthesising Domain Constraints from Operator Descriptions. In *Proceedings ECAI92.*

Long, D., and Fox, M. 1999. The efficient implementation of the plan-graph in STAN. *JAIR* 10.

Morris, P., and Feldman, R. 1989. Automatically Derived Heuristics for Planning Search. In *Proceedings of the 2nd Irish Conference on Artificial Intelligence and Cognitive Science, School of Computer Applications, Dublin City University.*

Smith, D., and Weld, D. 1999. Temporal Graphplan with mutual exclusion reasoning. In *Proceedings of IJCAI-99.*