Computing Parameter Domains as an Aid to Planning

Alfonso Gerevini
Dip. di Elettronica per l'Automazione
Università di Brescia
via Branze 38, 25123 Brescia, Italy
email: gerevini@bsing.ing.unibs.it

Lenhart Schubert
Dept. of Computer Science
University of Rochester
Rochester, NY 14627-0226 USA
email: schubert@cs.rochester.edu

Abstract

We show that by inferring parameter domains of planning operators, given the definitions of the operators and the initial and goal conditions, we can often speed up the planning process. We infer parameter domains by a polynomial-time algorithm that uses forward propagation of sets of constants occurring in the initial conditions and in operator postconditions. During planning parameter domains can be used to prune operator instances whose parameter domains are inconsistent with binding constraints, and to eliminate spurious "clobbering threats" that cannot, in fact, be realized without violating domain constraints. We illustrate these applications with examples from the UCPOP test suite and from the Rochester TRAINS transportation planning domain.

Introduction

We are concerned here with improving the performance of "well-founded" domain-independent planners – planners that permit proofs of soundness, completeness, or other desirable theoretical properties. A state-of-the-art example of such a planner is UCPOP (Penberthy & Weld 1992; Barrett *et al.* 1994), whose intellectual ancestry includes STRIPS (Fikes & Nilsson 1971), TWEAK (Chapman 1987), and SNLP (McAllester & Rosenblitt 1991). Such planners unfortunately do not perform well at present, in comparison with more practically oriented planners such as SIPE (Wilkins 1988), PRS (Georgeff & Lansky 1987), or O-Plan (Currie & Tate 1991).

However, there appear to be ample opportunities for bringing well-founded planners closer to practicality. We recently obtained order-of-magnitude speedups for UCPOP operating on problems in the UCPOP test suite (among others) (Schubert & Gerevini 1995). These speedups were achieved by improving the plan- and flaw-selection strategies used by UCPOP. At the end of the cited paper we suggested some *preprocessing* methods as an additional way of speeding up domain-independent planners. In particular, we suggested the use of parameter domain precomputation, and precomputation of state constraints, using this information for search pruning during planning.

Herein we follow up the first of these suggestions. We propose a method of precomputing parameter domains based on propagating sets of constants forward from the initial conditions.[1] The process is iterative, but the algorithm runs within a time bound that is polynomial in the size of the problem specification. We show how to use parameter domain information in a UCPOP-style planner, illustrating its usefulness for accelerating planning.[2]

Our examples are drawn from the UCPOP test suite as well as from the TRAINS transportation planning world developed in Rochester (Allen & Schubert 1991; Allen *et al.* 1995). The comparisons we make use our own improved search strategies as baseline, i.e., we are interested in *additional* speedups obtainable by use of parameter domains, above those obtainable with the most effective search strategies. Our plan-selection strategy uses S+OC – the number of steps in a plan plus the number of open conditions still to be established – as a heuristic measure for UCPOP's A* search of the plan space. Our flaw- selection strategy, which we term ZLIFO, prefers "zero commitment" plan refinements to others, and otherwise uses a LIFO (stack) discipline. Zero commitment refinements are logically necessary ones: they either eliminate a plan altogether because it contains an irremediable flaw, or they add a unique step or unique causal link (from the initial state) to establish an open condition that cannot be established in any other way. ZLIFO is a variant of another flaw selection strategy called "least cost flaw repair" (LCFR) that was previously studied by Joslin and Pollack (1994). As evidence that the effectiveness of using parameter domains is not dependent on some

[1]We hope that the notion of a "parameter domain", as a set of admissible bindings (constants), will cause no confusion with the notion of a *planning* domain, as a specified set of operators, along with constraints on admissible initial conditions and goal conditions.

[2]While the techniques we shall describe are applicable to other planners, our focus is on UCPOP because it is well-known and the Lisp code is readily available. The system can be obtained via anonymous ftp from cs.washington.edu.

peculiarity of our search strategy, we also show some results for LCFR with use of parameter domains.

Precomputing Parameter Domains

How can parameter domains help?

In our previous experimentation with UCPOP strategies, we found that UCPOP goal regression often hypothesized steps that were doomed to be abandoned eventually, because they stipulated impossible parameter bindings. A clear example of this occurred in the Molgen domain, as encoded in the UCPOP test suite. The goal of the test problem is

```
(and (bacterium ?b) (molecule ?m) (contains IG ?m)
     (contains IG ?m) (contains ?m ?b) (pure ?b)),
```

where `?b` and `?m` are existentially quantified variables. We are using the abbreviations `IG, EE, JE, L` for insulin-gene, e-coli-exosome, junk-exosome, and linker; and `E, J, A1` for e-coli, junk, and antibiotic-1. Now, `(bacterium ?b)` and `(molecule ?m)` can be established only with the `*start*` operator, i.e., with the initial conditions, and thus will not be instantiated to "bizarre" values. (The initial conditions supply `E` and `J` as the only instances of `bacterium`, and `IG, EE, JE`, and `L` as the only instances of `molecule`.) On the other hand, the remaining goals turn out to match the effects of various instances of the `ligate, transform`, and `screen` operators of Molgen, as follows:

```
(contains IG ?m): (ligate IG ?m), (transform IG ?m)
(contains ?m ?b): (ligate ?m ?b), (transform ?m ?b)
(pure ?b):        (screen ?b ?y ?z)
```

UCPOP will happily regress on these actions. Yet two of them are doomed to fail, perhaps after a great deal of effort expended on trying to satisfy their preconditions. In particular, examination of the constants that can "flow into" the `transform` operator from the initial conditions and other Molgen operators shows that the first argument is restricted to domain {`EE, JE`} and the second is restricted to {`E, J`}. Consequently the instance `(transform IG ?m)` above is unrealizable, as its first argument `IG` is not in {`EE, JE`}. (Note that distinct constants denote distinct entities according to the unique-names assumption made by UCPOP.) For slightly more subtle reasons, the instance `(ligate ?m ?b)` is also unrealizable: it is the result of a match between `(contains ?m ?b)` and a "*when*-clause" (conditional effect) of the `ligate` operator, whose preconditions can only be reached if the second parameter (here, `?b`) lies in {`IG, JE, EE`} – yet `?b` is also restricted to the set {`E, J`}, as a result of the goal condition `(bacterium ?b)`, which (as noted before) is only achievable through the initial conditions.

Note that elimination of action candidates as above increases the number of "zero commitment" plan refinements that can be made. In the example, we are left with exactly one action for each of the three goals, and so the ZLIFO and LCFR strategies will prefer to regress on these goals rather than regressing on `(bacterium ?b)` and `(molecule ?m)` – which would prematurely make arbitrary choices of `?b` and `?m` from the initial state.

Description of the algorithm

In any completed plan, each precondition of each action must be instantiated by an effect of some earlier action. So the values of the variables of the action can only be values that can be "produced" by earlier actions, starting with the initial "action", `*start*`. Moreover, where a variable occurs in more than one precondition of a given action, only those values are possible which can be produced by instantiation of *all* these preconditions.

Our algorithm *find-parameter-domains* is based on these observations. Beginning in the initial state, it "propagates" positive atomic predications to all possible operator preconditions. For a propagated ground atom, if the atom matches an operator precondition, the algorithm adds the constants in that ground atom to the *individual domains* of the parameters they were unified with. These individual domains are particular to specific preconditions. For instance, the individual domain of `?x` for an operator with preconditions `(on ?x ?y), (clear ?x)` will in general be distinct for these two preconditions.

As soon as we have nonempty individual domains for all parameters in all preconditions of an operator, we form the *intersection* of the individual domains of each parameter of the operator. For example, if `(on ?x ?y)` has (so far) been matched by `(on A B)` and `(on B C)`, and `(clear ?x)` has (so far) been matched by `(clear A)` and `(clear Table)`, then the individual domain of `x` will be {`A,B`} in the first precondition and {`A,Table`} in the second. Thus (assuming there are no other preconditions) the intersected domain of `?x` will be {`A`} at this point. If later `(clear B)` is also matched against `(clear ?x)`, the intersected domain of `?x` will grow to {`A,B`}. When both `?x` and `?y` have nonempty intersected domains, the effects (postconditions) of the operator can in turn be propagated, with `?x` and `?y` "bound" to their intersected domains.

The propagated effects are again matched against all possible operator preconditions, and when a variable "bound" to an intersected domain is successfully unified with a variable in a precondition, it passes its intersected domain to the *individual* domain of that precondition-variable (via a union operation). This can again lead to growth of the intersected domains of the operator whose precondition was matched, the effects of that operator may then be propagated, and so on. The individual domains and intersected domains grow monotonically during the propagation process, and in the end represent the desired parameter domains of the operators.

Before presenting the algorithm a little more formally, we note that the parameter domains will sometimes be "too large", including values that would be found to be impossible if a more detailed state space exploration were conducted. However, all that is required for soundness in our use of the domains is that they not be "too small" (i.e., that they contain all parameter values that can actually occur in the problem under consideration). Of course, to be of practical use the parameter domains of an operator should exclude some of the constants occurring in the problem specification, particularly those for which it is intuitively obvious that they are of the wrong sort to fill particular argument slots of the operator. This has turned out to be the case for all problem domains we have so far experimented with.

The preceding sketch of our method is an oversimplification since preconditions and effects of UCPOP operators may be particular to a *when*-clause. In this case we compute individual domains and intersected domains separately for each *when*-clause. For example, consider the following schematic representation of an operator:

```
(define (operator op1)
   :parameters (?x ?y)
   :precondition (and P1 P2)
   :effect (and E1 E2
                (when P'E')
                (when P"E") )),
```

where all conditions starting with *P* or *E* denote atomic formulas that may involve ?x and ?y. We can think of this operator as consisting of a *primary when-clause* whose preconditions *P1* and *P2* must always be satisfied and whose effects *E1* and *E2* are always asserted, and two *secondary when-clauses* whose respective preconditions *P'* and *P"* may or may not be satisfied, and when they are, the corresponding effects *E'* and *E"* are asserted. Here our algorithm would maintain individual domains for ?x and ?y for each of preconditions *P1, P2, P'*, and *P"*, and it would maintain intersected domains for ?x and ?y for the primary *when*-clause and each of the two secondary clauses. The intersected domains for the secondary clauses would be based on the individual domains of ?x and ?y not only relative to *P'* and *P"*, but also on those relative to *P1* and *P2*, since (as noted) the primary preconditions must hold for the operator to have any of its effects, including conditional effects.

The algorithm is outlined below. W is a list of (names of) *when*-clauses whose effects are to be propagated. Individual and intersected parameter domains are initially nil, except that the intersected domain of any parameter that does not appear in the preconditions (and corresponding primary preconditions) of a given *when*-clause is T (the universal domain) relative to that *when*-clause. Unification in step 2(a) is as usual, except that when an effect variable v is unified with a constant c in a precondition, the unification succeeds, with unifier v = c, just in case c is an element of the intersected domain of v (for the relevant *when*-clause. The given *inits* (initial conditions) and *goals* (which may be omitted, i.e., nil) are treated as an operator `*start*` with no preconditions and an operator `*end*` with no effects. Variables in *goals* are treated like operator parameters. We use the terms "parameters" and "variables" interchangeably here.

Algorithm:
find-parameter-domains(*operators,inits,goals*)

1. Initialize W to the initial conditions, so that it contains just the (primary) *when*-clause of `*start*`.
2. Repeat steps (a–c) until W = nil:
   (a) Unify the positive effects of all *when*-clauses in W with all possible operator preconditions, and mark the preconditions successfully matched in this way as "matched". (This marking is permanent.) Augment the *individual* domain of each matched precondition variable with a certain set C of constants, defined as follows. If the precondition variable was unified with a constant c, then C = {c}; if it was unified with an effect variable, then C is the *intersected* domain of that effect variable (relative to the *when*-clause to which the effect belongs).
   (b) Mark those *when*-clauses as "propagation candidates" that have all their preconditions (including corresponding primary preconditions) marked as "matched" and that involve at least one variable for which some relevant individual domain was augmented in step (a).
   (c) Reset W to nil. For all *when*-clauses that are propagation candidates, compute new intersected domains for their variables. If an intersected domain of a *when*-clause is thereby enlarged, and all intersected domains for the *when*-clause are now nonempty, then add the *when*-clause to W.
3. Further restrict intersected domains using equative preconditions of form (EQ u v), i.e., form a common intersected domain if both u and v are variables. If u is a constant and v is a variable, reduce the intersected domain of v by intersecting it with {u}; similarly if u is a variable and v is a constant. If the equation belongs to a primary *when*-clause, use it to reduce the intersected domains of u and v (whichever are variables) in the secondary clauses as well.
4. Return the intersected domains as the parameter domains, producing a sequence of lists with each list of form
   $(op\ (x_1\ a_1\ b_1\ c_1\ ...)\ (x_2\ a_2\ b_2\ c_2\ ...)\ ...)$,
   where each operator *op* appears at least once. If *op* has *k* conditional effects, there will be $k + 1$ successive lists headed by *op*, where the first provides the parameter domains for the primary effects of *op* and the rest provide the parameter domains for the conditional effects (in the order of appearance in the UCPOP definition of *op*).

Note that we do not match or propagate *negative* conditions. The problem with negative conditions is that a very large number of them may be implicit in the initial conditions, given the use of the Closed World Assumption in UCPOP. For instance, in a world of n blocks, with at most $O(n)$ on-relations (assuming that a block can only be on one other block), we necessarily have $O(n^2)$ implicit `(not (on ...))` relations. In fact, the individual variable domains of negative preconditions or goals can really be infinitely large. For instance, given an empty initial state and a `(paint-red ?x)` operation with precondition `(not (red ?x))` and effect `(red ?x)`, we can achieve `(red` c`)` for infinitely many constants c. Perhaps negative conditions could be effectively dealt with by maintaining anti-domains for them, but we have not explored this since in practice ignoring negative conditions seems to cause only minimal "domain bloating". (We have proved that no actual domain elements can be *lost* through neglect of some preconditions.)

Our use of EQ-conditions could be refined by making use of them during the propagation process, and NEQ-conditions could also be used. However, doing so would probably have marginal impact. Finally, we do not at present handle universally quantified preconditions or effects.

Correctness and tractability

In keeping with the remarks in the previous section, we will call an algorithm for computing parameter domains *correct* if the domains it computes subsume all possible parameter values that can actually occur (in a given primary or secondary *when*-clause) if we consider all possible sequences of operator applications starting at the given initial state.

The point is that this property will maintain the soundness of a planning algorithm that uses the precomputed parameter domains to prune "impossible" actions (as well as spurious threats) from a partially constructed plan. We assert the following

Theorem 1. *The* `find-parameter-domains` *algorithm is correct for computing parameter domains of UCPOP-style sets of operators, initial conditions, and (possibly) goal conditions.*

Proof sketch: [3] A preliminary step is to establish termination, using the monotonic growth of domains and the finiteness of the set of constants involved. We then need to show that if there exists a valid sequence $A_0A_1...A_n$ of actions (operator instances) starting with A_0 = `*start*`, and if A_n is an instance of `Op`, then the bindings that the parameters of `Op` received in instance A_n are eventually added to the relevant intersected domains of `Op` (where "relevant" refers to the *when*-clauses of `Op` whose preconditions are satisfied at the beginning of A_n). This can be proved by induction on n. The basis case ($n = 1$) is straightforward, and in the non-basis case we make use of the fact that every primary and secondary precondition of `Op` that holds at the beginning of A_n was established by one of the actions in $A_0A_1...A_{n-1}$. For each such precondition, if it was established by A_i, and in this process a parameter `?x` of `Op` was bound to a constant `c`, we use the induction hypothesis to show that `c` appears in the relevant intersected domain of the parameter of A_i unified with `?x`. (For $A_i = A_0$, no appeal to intersected domains is needed.) Also all preconditions (including parameter-free ones) must be shown to be marked "matched" eventually, before we can conclude that the intersected domain(s) of the parameters of `Op` relative to the instantiated *when*-clauses of `Op` will eventually contain the bindings of these parameters in A_n. Some slight further complications are added by the use of `EQ`-conditions at the end of *find-paramer-domains*.□

[3]For details see (Gerevini & Schubert forthcoming).

Next we formally state our tractability claim for the algorithm, as follows (with some tacit assumptions mentioned in the proof sketch).

Theorem 2. *Algorithm* `find-parameter-domains` *can be implemented to run in* $O(mn_pn_e(n_p+n_e))$ *time in the worst case, where* m *is the number of constants in the problem specification,* n_p *is the combined number of preconditions for all operators (and goals, if included), and* n_e *is the combined number of operator effects (including those of* `*start*`*).*

Proof sketch:[3] For any particular intersected domain of a particular operator, there can be at most m updates of this domain. Each such update can cause all of the effects of the *when*-clause to which the intersected domain belongs to be propagated. A bound on this number is n_e. Each propagated effect may then be unified with n_p preconditions. Thus an intersected domain update may cause $O(mn_en_p)$ unifications. Assuming that predicates have at most a fixed number of arguments, the number of intersected domains is $O(n_e)$, yielding an overall bound of $O(mn_e^2n_p)$ on the number of unifications. This can also be shown to bound the cost of individual domain updates, assuming appropriate handling of incremental set unions.

Next we look at the cost of attempted intersected domain updates. There can be one attempt for each relevant individual domain update, and each relevant individual domain may be updated m times. So there are at most $O(mn_p)$ attempts to update one intersected domain. The cost per added constant in such attempts is $O(n_p)$ using appropriate methods for incremental set intersection. Since (as before) there are no more than $O(n_e)$ intersected domains, the total intersected domain updating cost is $O(mn_en_p^2)$. Thus the combined costs yield a bound of $O(mn_pn_e(n_p+n_e))$.□

Using Parameter Domains for Accelerating a Planner

We have already used the example of Molgen to motivate the use of precomputed parameter domains in planning, showing how such domains may allow us to prune non-viable actions from a partial plan.

More fundamentally, they can be used each time the planner needs to unify two predications involving a parameter, either during goal regression or during threat detection. (In either case, one predication is a (sub)goal and the other is an effect of an action or an initial condition.) If the unifier is inconsistent with a parameter domain, it should count as a failure even if it is consistent with other binding constraints in the current (partial) plan. And if there is no inconsistency, we can use the unifier *to intersect and thus refine the domains* of parameters equated by the unifier.

For example, suppose that G = (at ?x ?y) is a precondition of a step in the current plan, and that E = (at ?w ?z) is an effect of another (possibly new) step, where ?x, ?y, ?w and ?z are parameters which have no binding constraints associated with them in the current plan. Assume also that the domains of the parameters are:

```
?x : {Agent1, Agent2, Agent3}  ?y : {City1, City2}
?w : {Agent1, Agent2}          ?z : {City3, City4}
```

The unification of G and E gives the binding constraints {?x = ?w, ?y = ?z}, which are not viable because the parameter domains of ?y and of ?z have an empty intersection.
On the other hand, if the domain of ?z had been {City2, City3, City4}, then the unification of G and E would have been judged viable, and the domains of the parameters would have been refined to:

```
?x : {Agent1, Agent2}     ?y : {City2}
?w : {Agent1, Agent2}     ?z : {City2}
```

Thus parameter domains can be incrementally refined as the planning search progresses; and the narrower they become, the more often they lead to pruning.

Incorporating Parameter Domains into UCPOP

The preceding consistency checks and domain refinements can be used in a partial-order, causal-link planner like UCPOP as follows. Given a goal (open condition) G selected by UCPOP as the next flaw to be repaired, we can

1. restrict the set of the operators that UCPOP would use for establishing G to those having an effect E matching G which does not violate the parameter domains (precomputed in E and possibly refined in G) and the binding constraints of the plan;
2. restrict the set of the steps already in the plan that the planner would reuse for establishing G to those having an effect matching G which does not violate the (possibly refined) parameter domains and the binding constraints of the plan.

Moreover, given a *potential* threat where P is the protected condition and Q is the threatening effect, parameter domains can be used to detect that the threat is actually spurious, because matching Q against P violates the (possibly refined) domain constraints of a parameter in P or G. Thus we can often

3. reduce the number of threats that are generated by the planner when a new causal link is introduced into the plan (this happens when an open condition is established either by reusing a step or by introducing a new one);
4. recognize that a threat on the list of the flaws to be processed is redundant, allowing its elimination.[4]

These four uses of parameter domains cut down the search space without loss of viable solutions, since the options that are eliminated cannot lead to a correct, complete plan.

We have incorporated these techniques into UCPOP (version 2.0), along with our earlier improvements to the plan and goal selection strategies. Parameter domains are handled through an extension of the "VARSET" data structure (Weld 1994) to include the domains of the variables (parameters), and by extending the unification process to implement the filtering discussed above.[5] We now describe our experiments with this enhanced system.

Experimental Results

First we should mention that the CPU times needed by our implementation of *find-parameter-domains* are negligible for the problems we have looked at. They were 10 msec or less for many problems in the UCPOP test suite (when running compiled Allegro CL 4.2 on a SUN 20), 20 msec for two problems (Fixa from the fridge repair domain and Fixit from the flat tire domain), and 30msec on the TRAINS world problems described below.

Our initial tests were based on several problems taken from UCPOP's test suite. While relatively easy problems such as Sussman-anomaly, Fixa, Test-Ferry, and Tower-Invert4 showed no further improvement beyond what is achievable with our S+OC and ZLIFO strategies, somewhat harder problems such as Hanoi1 (with 3 disks), Fix3, and especially Molgen showed significant improvements. The "rat-insulin" problem for Molgen was solved with the aid of parameter domains while generating/visiting only half as many plans as without parameter domains.

These initial experiments suggested to us that the most promising application of precomputed parameter domains would be for nontrivial problems that involved a variety of *types* of entities and relationships, and significant amounts of goal chaining (i.e.,

[4]Note that since parameter domains are incrementally refined during planning, even if we use 3. during the generation of the threats, it is still possible that one of these becomes spurious after it has been added to the flaw list.

[5]In the current implementation new threats are filtered only when the protected condition is established by a step already in the plan.

TRAINS problems	without domains		with domains	
	Plans	Time	Plans	Time
Trains1	4097/2019	13.7	297/238	1.4
Trains2	17482/10907	80.6	1312/1065	7.16
Trains3	31957/19282	189.8	3885/3175	25.1

Table I: Plans generated/visited and CPU-time (secs) for UCPOP with and without parameter domains in the TRAINS domain using the ZLIFO strategy.

TRAINS problems	without domains		with domains	
	Plans	Time	Plans	Time
Trains1	1093/597	8.1	265/194	2.3
Trains2	>50000	>607	>50000	>534
Trains3	>50000	>655	>50000	>564

Table II: Plans generated/visited and CPU-time (secs) for UCPOP with and without parameter domains in the TRAINS domain using the LCFR strategy.

with each successive action establishing preconditions for the next). From this perspective, the TRAINS transportation planning world (Allen & Schubert 1991; Allen *et al.* 1995) struck us as a natural choice for further experimentation, with the additional advantage that its design was independently motivated by research at Rochester into mixed-initiative problem solving through natural-language interaction.

The version we encoded involves four cities (Avon, Bath, Corning, Dansville) connected by four tracks in a diamond pattern, with a fifth city (Elmira) connected to Corning by a fifth track. The available resources, which are located at various cities, consist of a banana warehouse, an orange warehouse, an orange juice factory, three train engines (not coupled to any cars), 4 boxcars (suitable for transporting oranges or bananas), and a tanker car (suitable for transporting orange juice). Goals are typically to deliver oranges, bananas, or orange juice to some city, requiring engine-car coupling, car loading and unloading, engine driving, and possibly OJ-manufacture. The UCPOP formalization is shown in Figure 1.

Tables I,II,III and IV report experimental results from the TRAINS domain. The initial state of Trains3 is the same as that of Trains1 except that `oj-fac1` and `e3` are at `Corning` instead of `Elmira`. The initial state of Trains2 is the same as that of Trains3 except that the connections from `Corning` to `Bath` and from `Dansville` to `Corning` are disabled (say, for maintenance).)

Each table gives the number of plans generated/visited by the planner and the CPU-time (seconds) required to solve the problems.[6]

We ran UCPOP with and without parameter domains on these problems, using S+OC (mentioned pre-

[6]The systems were compiled under Allegro CL 4.2, with settings (space 0) (speed 3) (safety 1) (debug 0), and run on a SUN 20. The CPU-time includes the Lisp garbage collection (it is the time given in the output by UCPOP).

```
(define (operator mv-engine)
   :parameters (?eng ?city1 ?city2 ?track ?car)
                 ; ?car is a "hidden" parameter
   :precondition (:and (engine ?eng) (at ?eng ?city1)
                          (connects ?track ?city1 ?city2))
   :effect (:and (at ?eng ?city2) (:not (at ?eng ?city1))
                 (when (coupled ?eng ?car)
                          (:and (at ?car ?city2)
                                (:not (at ?car ?city1))))) )

(define (operator ld-oranges)
   :parameters (?ors ?car ?city)
   :precondition (:and (oranges ?ors) (boxcar ?car) (empty ?car)
                          (at ?ors ?city) (at ?car ?city) )
   :effect (:and (:not (empty ?car)) (in ?ors ?car)
                 (:not (at ?ors ?city))) )

(define (operator ld-bananas)
   :parameters (?bas ?car ?city)
   :precondition (:and (bananas ?bas) (boxcar ?car) (empty ?car)
                          (at ?bas ?city) (at ?car ?city) )
   :effect (:and (:not (empty ?car)) (in ?bas ?car)
                 (:not (at ?bas ?city))) )

(define (operator ld-oj)
   :parameters (?oj ?car ?city)
   :precondition (:and (oj ?oj) (tanker-car ?car) (empty ?car)
                          (at ?oj ?city) (at ?car ?city) )
   :effect (:and (:not (empty ?car)) (in ?oj ?car)
                 (:not (at ?oj ?city))) )

(define (operator make-oj)
   :parameters (?o ?fac ?city)
   :precondition (:and (oranges ?o) (oj-fac ?fac) (at ?o ?city)
                           (at ?fac ?city) )
   :effect (:and (oj ?o) (:not (oranges ?o))) )

(define (operator unload)
   :parameters (?comm ?car ?city)
   :precondition (:and (in ?comm ?car) (at ?car ?city))
   :effect (:and (:not (in ?comm ?car)) (empty ?car)
                 (at ?comm ?city)) )

(define (operator couple)
   :parameters (?eng ?car ?city)
   :precondition (:and (engine ?eng) (car ?car) (loose ?car)
                          (at ?eng ?city) (at ?car ?city) )
   :effect (:and (coupled ?eng ?car) (:not (loose ?car))) )

(define (operator uncouple)
   :parameters (?eng ?car)
   :precondition (coupled ?eng ?car)
   :effect (:and (loose ?car) (:not (coupled ?eng ?car))) )

INITIAL STATE FOR Trains1:
( (city avon) (city bath) (city corning) (city dansville) (city elmira)
  (track tr1) (track tr2) (track tr3) (track tr4) (track tr5)
  (connects tr1 avon bath) (connects tr1 bath avon)
  (connects tr2 bath corning) (connects tr2 corning bath)
  (connects tr3 avon dansville) (connects tr3 dansville avon)
  (connects tr4 dansville corning) (connects tr4 corning dansville)
  (connects tr5 corning elmira) (connects tr5 elmira corning)
  (engine e1) (engine e2)(engine e3) (car bc1) (car bc2) (car bc3)
  (car bc4) (car tc1) (boxcar bc1) (boxcar bc2) (boxcar bc3)
  (boxcar bc4) (tanker-car tc1) (oranges ors1) (bananas bas1)
  (oj-fac oj-fac1) (empty bc1) (empty bc2) (empty bc3)
  (empty bc4) (empty tc1) (loose bc1) (loose bc2) (loose bc3)
  (loose bc4) (loose tc1) (at e1 avon) (at bas1 avon) (at bc1 bath)
  (at bc2 bath) (at bc3 dansville) (at tc1 corning) (at ors1 corning)
  (at e2 elmira) (at e3 elmira) (at bc4 elmira) (at oj-fac1 elmira) )

GOAL OF Trains1: (:exists (oranges ?x) (at ?x bath))
GOAL OF Trains2 AND Trains3: (:exists (oj ?x) (at ?x dansville))

PARAMETER-DOMAINS FOR LD-OJ:
?oj={ors1} ?car={tc1} ?city={elmira corning dansville bath avon}
```

Figure 1: Rochester TRAINS transportation domain, and an example of parameter domains.

T-TRAINS	without domains		with domains	
problems	Plans	Time	Plans	Time
T-Trains1	3134/2183	17.2	505/416	3.4
T-Trains2	5739/4325	37.3	3482/2749	27.3
T-Trains3	17931/13134	130.4	11962/9401	105.1

Table III: Plans generated/visited and CPU-time (secs) for UCPOP with and without parameter domains in the "typed" TRAINS domain using the ZLIFO strategy.

T-TRAINS	without domains		with domains	
problems	Plans	Time	Plans	Time
T-Trains1	3138/2412	31.5	1429/1157	14.5
T-Trains2	>50000	>1035	>50000	>1136
T-Trains3	>50000	>976	>50000	>962

Table IV: Plans generated/visited and CPU-time (secs) for UCPOP with and without parameter domains in the "typed" TRAINS domain using the LCFR strategy.

viously) for plan selection and both the ZLIFO strategy (Schubert & Gerevini 1995), and the LCFR (least cost flaw selection) strategy (Joslin & Pollack 1994) for flaw selection.[7]

The results of Table I and II show that using parameter domains can give very significant improvements in performance. For example, the use of parameter domains provided an 11-fold speedup for Trains2. In this particular problem the speedup (on all metrics) was the result of pruning 1482 plans (more than half of those generated) during the search, and recognizing 305 unsafe conditions as redundant. Evidently, the effect of this pruning is amplified by an order of magnitude in the overall performance, because of the futile searches that are cut short.

Intuitively, the use of parameter domains to constrain planning is analogous to using type constraints on the parameters (although parameter domains also take account of initial conditions). It is therefore of interest to see whether adding type constraints can provide similar efficiency gains as the use of parameter domains. Tables III and IV show the results of solving T-Trians1–3, where these use a "typed" version of TRAINS (T-TRAINS); the operators have been slightly changed by adding new preconditions stating the types of the parameters involved. For example, we added the preconditions `(engine ?eng)` and `(car ?car)` to the operator `uncouple`.

It is interesting to note that parameter typing gives only modest improvements in the absence of parameter-domains, and significant deterioration in their presence. This is understandable since the mere fact that certain type-preconditions are set up does not immediately lead to any pruning. On the contrary, it may lead to some overhead because the additional open conditions must be dealt with.

[7]For lack of space the strategies are not described here.

Conclusions and Further Work

We have developed and implemented a tractable algorithm for precomputing parameter domains of planning operators, relative to given initial conditions. We have shown how to use the precomputed domains during the planning process, using them to prune non viable actions and bogus threats, and updating them dynamically for maximum effect.

The idea of using precomputed parameter domains to constrain planning was apparently first proposed in a technical report by Goldszmidt *et al.* (1994).[8] This contains the essential idea of accumulating domains by forward propagation from the initial conditions. However, the algorithm sketched there assumes that this propagation can be done in a single sweep from the initial conditions to the goals; by contrast, our algorithm allows for a cyclic operator graph and for conditional effects. Also (to our knowledge at the time of writing), their algorithm remains to be theoretically analyzed and experimentally tested.

Judging from the examples we have experimented with, our techniques are well-suited to nontrivial problems that involve diverse types of objects, relations and actions, and significant logical interdependencies among the steps needed to solve a problem. Speedups by a factor of around 10 were observed for some problems in the TRAINS transportation planning world. Though our implementation is aimed at a UCPOP-style planner, essentially the same techniques would be applicable to many other planners.

We also found the parameter domain precomputations to be a very useful debugging aid. In fact, the domain precomputation for our initial formulation of the TRAINS world immediately revealed several errors. For instance, the domain of the `?eng` parameter of `mv-engine` turned out to contain oranges, bananas, and an OJ factory, indicating the need for a type constraint on `?eng`. (Without this, transportation problems would have been solvable without the benefit of engines and trains!) Another immediately apparent problem was revealed by the parameter domains for `?city1` and `?city2` in `mv-engine`: the domain for `?city1` excluded Elmira, and that for `?city2` excluded Avon. The obvious diagnosis was that we had neglected to assert both

`(connected c1 c2)` and `(connected c2 c1)`

for each track connecting two cities. Furthermore, the parameter domains can quickly identify unreachable operators and goals in some cases. For instance, without the `make-oj` operator, the computed domains show that the `ld-oj` operator is unreachable, and that a goal like `(and (oj ?oj) (at ?oj Bath))` (getting

[8]We thank one of the anonymous referees for drawing this to our attention, and David Smith for making the report available to us.

some orange juice to Bath) is unattainable (the parameter domain for `?oj` will be empty).

Of course, running the planner itself can also be used for debugging a formalization, but planning is in general far more time-consuming than our form of preprocessing (especially if the goal we pose happens to be unachievable in the formalization!), and the trace of an anomalous planning attempt can be quite hard to interpret, compared to a listing of parameter domains, obtained in a fraction of a second.

Some possible extensions to our techniques include (i) handling of universally quantified preconditions and effects, as permitted by UCPOP; (ii) handling of special "facts", i.e. lisp-evaluated predications; (iii) more "intelligent" calculation of domains, by applying a constraint propagation process to the sets of ground predications that have been matched to the preconditions of an operator; this can be shown to yield tighter domains, though at some computational expense. Blum and Furst (1995) recently explored a similar idea, but rather than computing parameter domains, they directly stored sets of ground atoms that could be generated by one operator application (starting in the initial state), two successive operator applications, and so on, and then used these sets of atoms (and exclusivity relations among the atoms and the actions connecting them) to guide the regressive search for a plan. This provided dramatic speedups, although the algorithm they describe does not allow for conditional effects.

We are also working on the second preprocessing technique mentioned at the outset, namely the inference of state constraints from operator specifications. One useful form of constraint is implicational (e.g., `(implies (on ?x ?y) (not (clear ?y)))`), and another is single-valuedness conditions (e.g., `(on ?x ?y)` may be single-valued in both `?x` and `?y`). We conjecture that such constraints can be tractably inferred and used for further large speedups in domain-independent, well-founded planning.

Acknowledgements

The detailed comments of the referees are gratefully acknowledged. The work was supported in part by Rome Lab contract F30602-91-C-0010 and NATO Collaborative Research Grant CRG951285. Some of the work (by AG) was carried out at IRST, 38050 Povo (TN), Italy.

References

Allen, J., and Schubert, L. 1991. The TRAINS project. Tech. Rep. 382, Dept. of Computer Science, Univ. of Rochester, Rochester, NY.

Allen, J.; Schubert, L.; Ferguson, G.; Heeman, P.; Hwang, C.; Kato, T.; Light, M.; Martin, N.; Miller, B.; Poesio, M.; and Traum, B. 1995. The TRAINS project: A case study in building a conversational planning agent. *Experimental and Theoretical Artificial Intelligence* 7:7–48.

Barrett, A.; Golden, K.; Penberthy, S.; and Weld, D. 1994. UCPOP user's manual. Tech. Rep. 93-09-06, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98105.

Blum, A., and Furst, M.L. 1995. Fast planning through planning graph analysis. In *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence (IJCAI-95)*, 1636–1642, Montreal, CA: Morgan Kaufmann.

Chapman, D. 1987. Planning for conjunctive goals. *Artificial Intelligence* 32(3):333–377.

Currie, K., and Tate, A. 1991. O-Plan: The open planning architecture. *Artificial Intelligence, 51*(1).

Fikes, R., and Nilsson, N. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Gerevini, A., and Schubert, L. 1996. Computing parameter domains as an aid to planning. To appear as Dept. of Computer Science Tech. Rep., University of Rochester, Rochester, NY 14627-0226.

Georgeff, M., and Lansky, A. 1987. Reactive reasoning and planning. In *Proc. of the 6th Nat. Conf. of the Am. Assoc. for Artificial Intelligence(AAAI-87)*, 677–682. Seattle, WA: Morgan Kaufmann.

Goldszmidt M.; Darwiche, A.; Chavez, T.; Smith, D.; and White, J. 1994. Decision-theory for crisis management. Tech. Rep. RL-TR-94-235, Rome Laboratory.

Joslin, D., and Pollack, M. 1994. Least-cost flaw repair: a plan refinement strategy for partial-order planning. In *Proc. of the 12th Nat. Conf. of the Am. Assoc. for Artificial Intelligence (AAAI-94)*, 1004–1009. Seattle, WA: Morgan Kaufmann.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. of the 9th Nat. Conf. on Artificial Intelligence (AAAI-91)*, 634–639. Anheim, Los Angeles, CA: Morgan Kaufmann.

Penberthy, J., and Weld, D. 1992. UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B.; Rich, C.; and Swartout, W., eds., *Proc. of the 3rd Int. Conf. on Principles of Knowledge Representation and Reasoning (KR92)*, 103–114. Boston, MA: Morgan Kaufmann.

Schubert, L., and Gerevini, A. 1995. Accelerating partial order planners by improving plan and goal choices. In *Proc. of the 7th IEEE Int. Conf. on Tools with Artificial Intelligence*, 442–450. Herndon, Virginia: IEEE Computer Society Press.

Weld, D. 1994. An introduction to least commitment planning. *AI Magazine* 15(4):27–62.

Wilkins, D. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm.* San Mateo, CA: Morgan Kaufmann.