Reasoning about Autonomous Processes in an Estimated-Regression Planner *

Drew V. McDermott
Yale University
Computer Science Department
drew.mcdermott@yale.edu

Abstract

We examine the issues that arise in extending an estimated-regression (ER) planner to reason about autonomous processes that run and have continuous and discrete effects without the planning agent's intervention (although the planner may take steps to get processes running). An ER planner is a classical planner that searches situation space, using as a heuristic numbers derived from a backward search through a simplified space, summarized in the *regression-match graph.* Extending the planner to work with processes requires it to handle objective functions that go beyond the traditional step count or cumulative step cost. Although regressing through process descriptions is no more difficult than regressing through standard action descriptions, figuring out how good an action recommended by the regression-match graph really is requires "plausibly projecting" the subtree suggested by the action, which often requires forcing actions to be feasible. The resulting algorithm works well, but still suffers from the fact that regression-match graphs can be expensive to compute.

Keywords: Planning and scheduling with complex domain models, domain-independent (almost) classical planning, constraint reasoning for planning and scheduling, planning with resources.

Introduction

Classical planning assumes that only the agent executing the plan has any effect on the world. However, there are many problems in which *processes* can be set in motion, either by the agent's actions or by forces of nature, and then proceed while the agent does other things. The other things the agent does may influence or terminate these processes. For example, an agent may plan to fly to San Francisco. The flight is a process that takes a predictable amount of time, but if a storm (another process) occurs, the flight may slow down.

In the 2002 International Planning Competition, a limited form of process called a *durative action* was introduced. These are actions that take a predictable amount of time computed when the action begins. Nothing can change that time. Although there are domains in which this idealization is useful, it doesn't seem all that natural. It is not hard (Fox & Long 2001) to define duratives in terms of a more general process model. The question is whether planners can reason in terms of that more general model.

This paper is about extensions to the Optop planner (McDermott 1996; 1999; 2002) to handle autonomous processes, which, as we shall see, also requires that it handle a broader range of objective functions than in the past. Optop is an *estimated-regression (ER) planner*, which searches through a space of plan prefixes, using as a heuristic estimator a number derived by doing a search back from the goal in a relaxed space in which deletions are ignored. (The HSP family of planners (Bonet, Loerincs, & Geffner 1997; Bonet & Geffner 2001) are also ER planners.) One key difference between Optop and other recent planners is that it does not instantiate all action schemas in advance in all possible ways, and then work with the corresponding ground action instances. This frees it to work in domains involving numbers and other infinite sets.[1]

As a simple example, figure 1 shows a domain with one process schema, describing what happens when a bathtub is filled. We use an extension of the PDDL notation (McDermott 1998; Fox & Long 2001), in which `:process` definitions are allowed by analogy with `:action` definitions. A problem in the domain might involve the goal of floating your boat. A solution would require one to turn the tub on, wait for it to fill, then put the boat in. If there are multiple tubs, the optimal plan is the one that uses the tub that fills up the fastest, assuming that one wants to minimize time. One might want to minimize some other metric, such as the weighted difference of time and tub surface area.

A solution to a one-boat problem in this domain will look like

(turn-on u); (wait); (float the-boat)

where u is the chosen tub, and `the-boat` is the given boat. If we want to require that no overflow occur, the plan will also have a `turn-off` step.

We will assume that concurrency is *interleaved* (Chandy

*This work was supported by DARPA/CoABS under contract number F30602-98-0168. Mark Burstein supplied ideas for an earlier version of this paper.

[1] Another difference is that it is written in Lisp, not C or C++. Although the use of Lisp has made it possible for a very complex algorithm to evolve faster than would be possible in any other language, performance has suffered.

```
(define (domain tub)
   (:requirements :fluents :processes)

   (:types Boat Tub - Obj)

   (:functions (volume ?tub - Tub)
               (faucet-rate ?tub - Tub) - Float
               (water-in ?tub - Tub) - (Fluent Float))

   (:predicates (floating ?b - Boat ?tub - Tub)
                (faucet-on ?tub - Tub)
                (overflowing ?tub - Tub))

   (:action (float ?b - Boat ?tub - Tub)
      :precondition (=~ (water-in ?tub) (volume ?tub))
      :effect (floating ?b ?tub))

   (:action (turn-on ?tub - Tub)
      :effect (faucet-on ?tub))

   (:action (turn-off ?tub - Tub)
      :effect (not (faucet-on ?tub)))

   (:process (filling ?tub - Tub)
      :condition (faucet-on ?tub)
      :effect
         (and (when (< (water-in ?tub) (volume ?tub))
                 (derivative (water-in ?tub)
                             (faucet-rate ?tub)))
               (when (>= (water-in ?tub) (volume ?tub))
                  (and (derivative (water-in ?tub) 0.0)
                       (overflowing ?tub))))))

(define (problem tub-prob-1)
   (:domain tub)
   (:objects tub1 - Tub my-boat - Boat)
   (:facts (= (faucet-rate tub1) 1.0)
           (= (volume tub1) 10.0))
   (:init (current-value (water-in tub1) 0.0))
   (:goal (exists (tub - Tub)
             (and (floating my-boat tub)
                  (not (overflowing tub)))))
   (:metric minimize (total-time)))
```

Figure 1: Simple domain involving processes

& Misra 1988; Roscoe 1998), in that no two actions occur at exactly the same time. If several bathtubs are turned on at the beginning of the plan, we model that by assigning an arbitrary sequence to the agents' actions and treating consecutive actions as separated by an infinitesimal slice of time. Remember that this applies only to the actions of turning on the faucets; the fillings of the bathtubs occur in parallel.

In the rest of this paper I will explain how ER planners work, and how to extend them to reason about autonomous processes.

Estimated-Regression Planning

Optop is a "state-space" planner. More precisely, the space it searches is a space of *plan prefixes*, each of which is just a sequence of steps, feasible starting in the initial situation, that might be extendable to a solution plan. In the context of considering a particular plan prefix, the term *current situation* will be used to refer to the situation that would obtain if these steps were to be executed starting in the initial situation. Each search state consists of:

1. a plan prefix P
2. the resulting current situation
3. a score

The original scoring metric was the length of P + the *estimated completion effort* for P, which is an estimate of the cost of the further steps that will be needed to get from the current situation to a situation in which the goal is true.

The estimated completion effort for P is obtained by constructing a *regression-match* graph, which can be considered a "subgoal tree with loops." The nodes of the tree are divided into *goal nodes* and *reduction nodes*. A goal node G is a literal to be made true.[2] If G is not true in the current situation, then below it are zero or more reduction nodes, each corresponding to an action that might achieve it. A reduction (node) is a triple $\langle G, M, \{P_1, \ldots, P_k\}\rangle$, where G is a goal node, M is an action term, and each P_i is a goal node. Ignoring the possibility of variables in goals, in a reduction the conjunction of the P_i are sufficient to make M feasible and G one of M's effects. There is a *reduction edge* from any goal node G to every reduction with G as its first component, and a *link edge* to P_i from a reduction whose third component includes the goal node P_i. If P_i occurs in a reduction, it is said to be a *subgoal* of that reduction.

The structure is not a tree because there can be multiple paths from one goal node to another through different reductions, and there can be a path from a goal node to itself. What we are interested in are certain cycle-free subgraphs, called "reduction trees," because they correspond to (apparently) feasible ways of achieving goals. The size of the subgraph below a goal node gives an estimate of the difficulty of achieving it. The estimate neglects step ordering, and destructive and constructive interactions among steps, but taking those phenomena into account is an exponential process, whereas the regression-match graph tends to be of polynomial size (as a function of the size of the problem and the size of the solution).

Optop avoids variables in goal nodes by its treatment of conjunctive goals. Given a conjunctive goal $H_1 \wedge \ldots \wedge H_n$, Optop finds *maximal matches* between it and the current situation, defined as substitutions that, roughly, eliminate all the variables while making as many of the H_i true in the current situation as possible (McDermott 1999). However, in domains with numbers, an unsatisfied H_i may have to be left with a variable. In that case, the exact meaning of a reduction becomes slightly harder to state (McDermott 2002).

Optop builds the regression-match graph by creating an artificial goal node `top` as the root of the "tree," then maximally matching the end goal to the current situation, producing a set of reductions, each of the form $\langle \texttt{top}, \texttt{done}, \{P_i\}\rangle$. It then examines every P_i in every reduction, finds actions that would achieve it, and maximally matches their preconditions to produce reductions for P_i. The process normally just keeps running until no more goal nodes have been generated, although there is a depth limit on the graph to avoid runaway recursions in perverse domains.

Having generated the graph, its "leaves" are reductions all of whose subgoals are true in the current situation. We call these *feasible* reductions. The action of such a reduction, called a *recommended action*, is feasible in the current situation, and is therefore a candidate for extending the current plan prefix.

We assign *completion effort estimates* to every node of the graph by assigning effort 0 to feasible reductions, assigning effort ∞ to all other nodes, then recursively updating the efforts according to the rules:

$$\mathit{eff}(G) = \begin{cases} 0 & \\ \quad \text{if } G \text{ is true} & \\ \quad \text{in the current situation} & \\ \min_{R \in \mathit{reductions}(G)} \mathit{eff}(R) & \\ \quad \text{otherwise} & \end{cases}$$

$$\mathit{eff}(\langle G, M, \{P_i\}\rangle) = 1 + \sum_i \mathit{eff}(P_i)$$

until all the values have stabilized.

As I said above, we are interested in extracting *reduction trees* from the regression-match graph. A reduction tree is a rooted tree, whose root, which coincides with the artifical `top` node, we will call $E_{\texttt{top}}$. Each node E of the tree has a goal $G(E)$ and a reduction $R(E)$. If $G(E)$ is true in the current situation, then $R(E) = \phi$; otherwise, for some reduction $\langle G, M, \{P_1, \ldots, P_k\}\rangle$ in the regression-match graph, $R(E) = \langle G, M, \{E_1, \ldots, E_k\}\rangle$, where $G(E_i) = P_i$, and where each P_i does not occur as the goal of any ancestor reduction in the tree.[3]

[2] It also contains a set of constraints on its free variables, if it has any, but space limits prevent me from saying more, except for some brief remarks in the Conclusions.

[3] Reduction trees are actually DAGs, but the planner ignores that fact, and treats two identical subtrees at different places in the "tree" as if they were different. So in practice reduction "trees" might as well really be trees.

The completion effort estimate for a reduction tree is defined by

$$eff(E) = \begin{cases} 0 & \\ \quad \text{if } G(E) \text{ is true} & \\ \quad \text{in the current situation} & \\ eff(R(E)) & \\ \quad \text{otherwise} & \end{cases}$$

$$eff(\langle G, M, \{E_i\}\rangle) = 1 + \sum_i eff(E_i)$$

A *reduction tree for* recommended action A is a reduction tree that includes A in a feasible node. A *minimal reduction tree for* A is a reduction tree for A that has minimal completion effort estimate. As an example, consider a simple blocks world with one action `(move` x y`)`. The initial situation has `(on C A)`, `(on A table)`, and `(on B table)`. The goal is `(and (on A B) (on B C))`. The reduction nodes of the regression-match graph are:

$\langle$`top, done,` $\{$`(on A B), (on B C)`$\}\rangle$[2]
$\langle$`(on A B), (move A B),` $\{$`(clear A)`$\}\rangle$[2]
$\langle$`(clear A), (move C table),` $\{\}\rangle$[1]
$\langle$`(on B C), (move B C),` $\{\}\rangle$[1]

The numbers in brackets are the completion-effort estimates. The recommended actions are `(move B C)` and `(move C table)`. The following tree is a reduction tree for both actions:

(on A B)
    Do (move A B)
        (clear A)
            Do (move C table)
(on B C)
    Do (move B C)

Optop will try both recommended actions, and will realize that `(move B C)` is a mistake as soon as it recomputes the regression-match graph on the next iteration of its outer search loop.

A recent improvement to Optop is to have it edit regression-match graphs for each plan prefix rather than recompute them each time, by copying the graph for the prefix's predecessor, discarding and re-matching all nodes with effort ≤ 1, on the grounds they are most likely to yield new recommended actions. The resulting graph is not quite as accurate as the one obtained by recomputing from scratch, but it usually contains significantly fewer nodes. Occasionally the edited graph recommends actions that are not actually feasible; in that case Optop discards it and regenerates the graph from scratch. In the "convoys" domain to be described in section , reusing regression-match graphs cuts the total number of goal nodes, and therefore maximal matches, by about 45%.

Another improvement in Optop is to avoid maintaining an index of all the facts true in a situation; instead, each situation contains an index of the *differences* between it and the initial situation. This device is very useful for domains with large initial situations.

Processes and Objective Functions

We now describe the extensions to Optop to handle autonomous processes and objective functions. Both rely on the concept of *fluent*, a term whose value varies from situation to situation. Fluents have been in PDDL from the beginning, but are just now beginning to be noticed, notably in the AIPS02 competition (Fox & Long 2001). The value of a fluent can be of any type, but we assume from here on that they all have numerical values, some integer (the *discrete* fluents), some floating-point (the *continuous* fluents). A *primitive fluent* is a term like `(water-in tub21)`; a nonprimitive fluent is the arithmetic combination of primitive fluents.

We formalize a *process* as an action-like entity which has a `:condition` field instead of a `:precondition`; whenever the condition field is true, the process is *active*. There are three fields specifying the effects of the process. The `:start-effect` and `:stop-effect` fields specify what happens when the process becomes active or ceases to be active. The `:effect` field specifies what happens at every point in time when the process is active. For this to be useful we require a new kind of effect:

`(derivative` q d`)`

which means that the derivative of continuous (primitive) fluent q is d. Although "derivative" sounds like we might be able to reason about arbitrary differential equations, for now we will assume that d is always a constant, i.e., that all fluent changes are linear.

An *objective function* is a measurement of the cost of a plan, a value to be minimized. We will take this to be the value of some given fluent in the situation when the end goal is achieved. It turns out that the extensions to Optop to handle objective functions and those to handle autonomous processes are closely related. Both extensions require the planner to abandon "step count" as a measure of the cost of a plan. We can still use it as a crude heuristic for managing the regression-match graph, but once the planner has found a feasible action, it must extract from the graph an "expected continuation" of the plan, and use that to generate a more precise estimate of the value of the action.

Let's look at an example (see figure 2). A problem in this domain involves convoys moving through a road network. The speed a convoy can attain degrades quadratically with the number of convoys it has to share its current road segment with. There are no unsolvable problems in the domain (assuming the road network is connected), but there are a lot of decisions about which way to send a convoy and how many convoys to put on a road segment at the same time.

Clearly, a solution to a problem in this domain will consist of issuing a set of orders to all the convoys that have to get somewhere, interspersed with "waits" in which time is allowed to pass. Exactly what the planner is waiting for will be explained later.

The first step in getting an ER planner to handle processes is to get it to regress through process descriptions. Given a goal of the form `(< (dist convoy-13 dest-2) 3.0)`, it sees that the process instance `(rolling convoy-13` $pt1$ `dest-2)` would change the derivative of

```
(define (domain agent-teams)
   (:types Agent Order)
   (:predicates
       (told ?a - Agent ?r - Order))
   (:action
        (tell ?a - Agent ?r - Order)
    :effect
       (and (forall (?r-old - Order)
               (when (told ?a ?r-old)
                  (not
                    (told ?a ?r-old))))
            (told ?a ?r)))

(define (domain convoys)
   (:requirements :fluents :processes)
   (:extends agent-teams)
   (:types Convoy - Agent Point - Obj)
   (:parameters (temporal-grain-size 1.0)
                (temporal-scope 10.0e10)
                - Float)
   (:predicates
       (at cv - Convoy pt - Point)
       (between cv - Convoy
                pt1 pt2 - Point)
       (connected pt1 pt2 - Point))
   (:functions
       (dist cv - Convoy pt - Point)
       - (Fluent Float)
       (traffic pt1 pt2 - Point)
       - (Fluent Integer)
       (base-speed cv - Convoy)
       (geo-dist pt1 pt2 - Point)
       - Float
       (to-go pt1 pt2 - Point)
       - Order)
   (:facts
       (forall (cv - Convoy pt1 pt2 - Point)
          (<- (feasible cv (to-go pt1 pt2))
              (at cv pt1)))))
```

```
(:process (rolling ?cv - Convoy
                   ?pt1 ?pt2 - Point)
  :condition
      (and (told ?cv (to-go ?pt1 ?pt2))
           (connected ?pt1 ?pt2)
           (or (at ?cv ?pt1)
               (between ?cv ?pt1 ?pt2))
           (> (dist ?cv ?pt2) 0))

  :start-effect
     (and (not (at ?cv ?pt1))
          (between ?cv ?pt1 ?pt2)
          (increase (traffic ?pt1 ?pt2)
                    1))

  :effect
        (and (derivative
               (dist ?cv ?pt1)
               (/ (base-speed ?cv)
                  (sq (fl-v (traffic
                              ?pt1
                              ?pt2))))
             (derivative
               (dist ?cv ?pt2)
               (/ (- (base-speed ?cv))
                  (sq (fl-v (traffic
                              ?pt1
                              ?pt2)))))

  :stop-effect
     (and (not (between cv ?pt1 ?pt2))
          (at cv ?pt2)
          (decrease (traffic ?pt1 ?pt2)
                    1)
          (forall (pt3 - Point)
             (when (connected pt2 pt3)
                (assign
                   (dist cv pt3)
                   (geo-dist pt2
                             pt3))))))
```

Note: `(sq (fl-v` f`))` *is the square of the value of fluent* f.

Figure 2: Convoys domain

`(dist convoy-13 dest-2)`, where $pt1$ is as yet unchosen. To make this instance active requires that all of the following be achieved (assuming that a convoy's base speed is positive):

```
*(told convoy-13 (to-go pt1 dest-2))
*(connected pt1 dest-2)
*(or (at convoy-13 pt1)
     (between convoy-13 pt1 dest-2))
*(> (dist pt1 dest-2) 0)
```

Optop, as usual, tries to find a $pt1$ that makes as many of these conjuncts true as possible. If `convoy-13` is `at` a point connected to `dest-2`, then one obvious choice is to bind $pt1$ to that point. The only unsatisfied goal would be `(told convoy-13 (to-go` $pt1$ `dest-2))`, which becomes a new goal node in the regression-match graph. If there is no such $pt1$, then for every point connected to `dest-2`, we get a maximal match with two unsatisfied goals, which gives rise to a reduction with two subgoal nodes, one to get to the candidate jumping-off point, and one to tell the convoy to go to `dest-2`.

The novelty in the reduction we get in this case is that it does not contain an action term, but a process term. The idea is that achieving the subgoals will cause the process to become active, after which simply waiting will cause the supergoal `(< (dist convoy-13 dest-2) 3.0)` to become true.

A Theory of Waiting

Actually, "simply waiting" is not so simple. The planner can't just wait for the process it created to terminate, because other processes are going on simultaneously. These processes may have useful effects that the planner should take advantage of (or, of course, deleterious effects that should cause the planner to backtrack in plan-prefix space). Other active processes may even cause the parameters of this process to change, thus changing the derivates of the fluents it affects.

Hence whenever an action is executed, Optop must construct a *process complex* attached to the resulting situation, with the following information:

1. All the active processes in this situation.
2. For each primitive fluent affected by active processes, a *process model* explaining how it changes over time.
3. A list of *triggers,* each a process model for a fluent (not necessarily primitive), plus a list of effects that will result when the fluent becomes non-negative.
4. The *trigger-time specification*, which specifies the next time point at which one or more trigger fluents will become non-negative, plus the effects that will then occur.

How the process complex is built is discussed below.

The trigger-time specification is needed when the planner is contemplating waiting for something to happen, instead of taking another action. In that case, time is advanced to the trigger time, all the primitive fluents are changed in accordance with their process models, and a new situation is produced in which the expected events have occurred.

Therefore, although the regression-match graph contains reductions that appear to depend on waiting for a particular process to reach some conclusion, when it comes time to extend the plan prefix, and one of those reductions is chosen, what actually happens is that the planner just adds a `wait` to the plan prefix, advancing to the current situation's trigger time. After that it recomputes the regression-match graph (by editing the old one if possible), and proceeds just as it does when taking an ordinary action.

Choosing The Next Action

Because autonomous processes can run in parallel, the time taken by a group of them is usually less than the sum of the times of each process in the group. Obviously, if we want to take advantage of this fact, we must be able to incorporate "total time taken" (makespan) into the objective function used by the planner. This requires a radical change to the way Optop works.

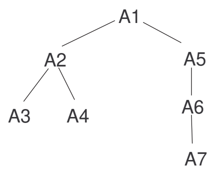
The original Optop used the step counts derived from the regression-match graph directly to provide an estimate of the work remaining if a particular action is chosen to execute next. But if we allow an arbitrary fluent as an objective function (and we might as well), the regression-match graph will not provide an estimate of it. We must go one step further, and, for each recommended action derive a *plausible projection* from a minimal reduction tree for that action. A plausible projection is obtained by "projecting" (simulating the execution of) the reduction tree bottom up, starting with the recommended action. The resulting projection terminates in a final state (which we hope is also a goal state), and the planner can measure the objective function there.

Unfortunately, the plausible-projection algorithm turns out to be quite tricky. There are two main issues:

1. In what order should the planner execute the actions in the reduction tree?
2. What should it do if the next action is not feasible?

The first issue is hard for obvious reasons: figuring out the order in which to do a set of apparently relevant steps is often the critical problem for a planner. I have implemented a simple depth-first, left-to-right, post-order traversal of the tree. That is, no action is executed until all the actions below it in the tree have been executed, and after executing an action, the next one to try is the leftmost unexecuted descendent of its parent (which may be the parent itself).

For instance, in the following schematic reduction tree (showing actions but no subgoals):



the algorithm projects the actions in order `A3, A4, A2, A7, A6, A5, A1.`

Here's where the second issue raises its head. Because we've picked an arbitrary order, it can easily happen that, e.g., in the situation after `A4` is executed, `A7` is not feasible. A related, but easier, issue is that the purpose of `A7` may already have been achieved, either because `A7` occurred earlier in the tree, or the purpose was achieved by other means. In either of these cases, the plausible projector just skips the redundant action.

The hard case is when an action (or process's) goal is not true, and the action is not feasible (or the process is not active). In such a case, the plausible projector must *force* the action to be feasible or the process to exist. It does this by calling the maximal matcher to find the missing pieces of the action or process's condition, then adds those missing pieces to the current projected situation.

Note that adding a missing piece can result in a physically impossible situation. For example, if a process requires an object B to be at a different location L' than L, the place it is projected to be, the feasibility forcer will cheerfully assert that it is at L'. From then on, the plausible projector will assume the object is in two places, at least until it projects a move to L. This tactic may sound odd, but it is in keeping with the overall idea of using a relaxed space in which deletions are ignored to evaluate plans. During plausible projection, the planner performs deletions when it projects an action, but also tosses in force-feasibility assumptions when it needs to.

We can now summarize the planning algorithm as shown in figure 3. The algorithm will return `FAILED` if it runs out of plan prefixes, but in practice that's unlikely; it usually runs forever if a problem is unsolvable.

```
search-for-plan(prob, metric)
   let Q = queue of scored plan prefixes,
              initially containing only the empty prefix
     (while Q is not empty
       (Remove the plan prefix P with minimum score from Q;
        If P is a solution to the problem, return it
        Compute a regression-match graph relating the goal of prob
           to the situation reached after P, editing the regression-
           match graph of P's predecessor if possible;
        For each action A recommended by the graph,
           Let R = some minimal reduction tree for A
             (Plausibly project R;
              Evaluate the metric in the resulting situation;
              Put the new prefix P+A on Q with that value as its
                score));
     return 'FAILED)
```

Figure 3: The overall Optop algorithm

Computing Process Models

During both plausible projection and "real" projection (i.e., the sort that happens when the planner extends a plan prefix by one step), the planner must build a process model, by examining every process definition and finding active and "dormant" instances. To explain this distinction, I will first note that the condition of a process definition can be separated into discrete and continuous parts. The continuous parts are inequalities on real-valued fluents; the discrete parts are all other formulas. If we put the condition in disjunctive normal form, and collect all the disjuncts whose discrete conjuncts are the same, the condition will look like this:

$$\begin{aligned}
&((d_{11} \wedge d_{12} \wedge \ldots \wedge d_{1n_1}\\
&\quad \wedge((c_{111} \wedge c_{112} \wedge \ldots \wedge c_{11k_{11}})\\
&\qquad \vee(c_{121} \wedge \ldots \wedge c_{12k_{12}})\\
&\qquad \vee \ldots\\
&\qquad \vee(c_{1m_1 1} \wedge \ldots \wedge c_{1m_1 k_{1m_1}})))\\
&\ \vee(d_{21} \wedge \ldots \wedge d_{2n_2}\\
&\quad \wedge((c_{211} \wedge \ldots \wedge c_{21k_{21}})\\
&\qquad \vee \ldots\\
&\qquad \vee(c_{2m_2 1} \wedge \ldots \wedge c_{2m_2 k_{2m_2}})))\\
&\ \vee \ldots\\
&\ \vee(d_{p1} \wedge \ldots \wedge d_{pn_p}\\
&\quad \wedge((c_{p11} \wedge \ldots \wedge c_{p1k_{p1}})\\
&\qquad \vee \ldots\\
&\qquad \vee(c_{pm_p 1} \wedge \ldots \wedge c_{pm_p k_{pm_p}})))
\end{aligned}$$

I'll use the notation d_i to refer to the conjunction of discrete conjuncts $d_{i1} \wedge \ldots \wedge d_{in_i}$. Similarly, c_i refers to the disjunction of continuous conjuncts associated with d_i; c_{ij} is then the j'th conjunction in that disjunction. A process instance is *dormant* if one of the d_i corresponding to it is true, but no corresponding c disjunct is true. A process instance that is neither active nor dormant is *comatose*.

Theorem: If the planning agent takes no action, then no process instance changes state (within the set $\{$*active, dormant, comatose*$\}$) until some c_i corresponding to a true d_i changes truth value.

Proof: None of the d_i can change truth value until some process starts or stops. But there must be an earliest starting or stopping event, so the d_i stay the same until then. Hence the only thing that could make a process start or stop is for a c_i to change truth value; furthermore, the corresponding d_i must be true, or c_i's truth value would be irrelevant. QED

Hence the planner needs to include in each process complex a model of how each changing primitive fluent is changing. For every active or dormant process, it must also include in the process model, for each fluent mentioned in c_i for a true d_i, a model of how it is changing and when c_i will become true. It does *not* need to do anything similar for comatose process instances.

In order to compute the trigger time, the planner must compute the earliest time at which some c_{ij} will become true. It approximates this computation by computing

$$max_{1 \leq q \leq k_{im_i}} \textit{trigger-time}(c_{ijq})$$

That is, it finds the earliest time before which each of these conjuncts will have become true. (It ignores the possibility that some may have become false again by that point.)

The two activities of building the fluent-change models and computing when c_{ij} will become true are largely independent and subject to different computational constraints. In the current implementation, all fluent-change models must be linear. The subroutine that predicts when c_{ij} will become true requires only that the fluents it mentions change monotonically. It does a binary search among all the fluents mentioned in all the c_i for the one that reaches its threshold first.

Both of these modules can be generalized. Indeed, there is no reason why the system should not be able to handle arbitrary differential equations. The idea would be to call an external solver (presumably some specialized Fortran program), telling it to simulate the differential equations numerically and stop when a list of thresholds is crossed. For reasons discussed below, this is not the top priority for making the planner practical.

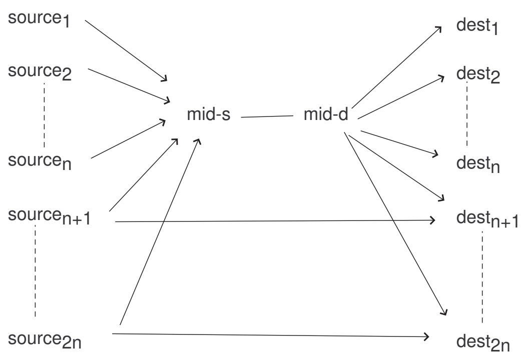


Figure 4: Schematic structure of "convoys" domain of size n

Results

The new version of Optop has been tried on a variety of domains, including some of those involving durative actions from the IPC3 problem suite (available at `http://www.dur.ac.uk/d.p.long/competition.html`). In those problems, durative actions are translated into processes, so that every action takes two steps: start the action, wait for it to stop.

A more interesting test of the planner is for processes that cannot be simulated by durative actions, such as the convoy domain described earlier. The examples we will examine for the rest of this section are a scalable test suite of stylized road graphs in this domain. The problem of size n in this suite has $2n$ convoys, which start at $2n$ different starting points ("sources") and must get to $2n$ different destinations ("dests"). The graph connects sources $1, \ldots, 2n$ to point `mid-s`. `mid-s` connects only to `mid-d`, which then connects to dests $1, \ldots, 2n$. In addition, source $n + k, 0 < i \leq 2n$, connects directly to dest $n+k$. Hence all of convoys $1, \ldots, n$ must go through the bottleneck `mid-s—mid-d`, but convoys starting at points $n + 1, \ldots, 2n$ can bypass the bottleneck. See figure 4.

Table 1 shows the performance of Optop on convoy problems of various sizes. In each case, Optop finds an optimal plan embodying the following scheme: Start all the convoys moving. Numbers 1 to n go to `mid-s`. Numbers $n+1$ to $2n$ go directly to their corresponding `dests`. When numbers 1 to n arrive, they are sent one-by-one to `mid-d`. (Numbers $n + 1$ to $2n$ are still en route, of course.) As each convoy arrives at `mid-d` it is sent to its respective destination. The distance from each source to each destination is 30 km; the `mid-s—mid-d` bottleneck is 10 km. Each convoy travels at 10 km/hr. Hence the optimal plan takes $3 + n - 1$ hours. Figure 5 shows the solution found by Optop for the problem of size 3 (i.e., with 6 convoys).

The times required to find solutions to problems of different sizes are given in table 1.[4] Optop does no search at all in this domain; every plan prefix it considers is part of its final answer. The reason is that the regression-match graph plus the plausible projector allow Optop to look all the way to the end of the problem from the very beginning. The price paid for this guidance is the cost of constructing the regression-match graph. The size of the graph is shown in the column labeled *G-Nodes*, which gives the total number of goal nodes generated in the course of solving the problem. This doesn't count nodes carried over from one search state to a successor. The number of maximal matches (in the column headed *Matches*) is greater than the number of goal nodes because "old nodes" (those carried over) may need to be re-matched. The time per step for the output increases rapidly, but the time per goal node and per maximal match rises more slowly. The biggest single reason for Optop's less than stellar performance is that most of the matches it performs on a plan prefix return identical results to the matches on the same goal nodes in the prefix's predecessor. More research is needed on recognizing when this work does not have to be redone.

The other parts of the program, including the plausible projector and the process-complex constructor, may sound complicated, but actually run quite fast; most of the time is spent in the regression-match-graph builder.

I should point out that the rate of growth of the regression-match graph with problem size (indicated in the *Nodes/step* column of table 1) is essentially the same phenomenon that occurs in many planners that transform all problems into propositional form by finding all possible instantiations of action definitions. The principal difference is that these planners pay the price just once, before beginning the search for a solution.

Related Work

The idea of modeling processes in terms of their beginnings and endings is not novel. For an elegant logical treatment, see (Reiter 2001). The notion that in simulating processes one can jump to the next moment at which something qualitative changes is also fairly standard, and was exploited in a planning context by (Atkin, Westbrook, & Cohen 1999).

Several recent planners handle durative actions, or actions with costs, or both (Smith & Weld 1999; Do & Kambhampati 2001). I am not aware of any domain-independent planner that handles truly autonomous processes as well as arbitrary objective functions. There has been a significant amount of work in areas such as *contingent* planning, where a planner must deal with ignorance about the world, including the effects of some of its actions. But such issues are completely orthogonal to the issues discussed here.

A preliminary report on this work appeared in the AIPS 2002 workshop on multiagent planning (McDermott & Burstein 2002).

Conclusions and Future Work

My conclusions contain good news and bad news. The good news is that estimated-regression planners do indeed have the flexibility to generalize to a wide variety of domains,

[4]Bear in mind that if Optop were recoded in C++ the run times might be significantly lower; but their growth as a function of problem size would presumably be similar.

Time 0
```
   Action: (tell convoy-6
                (to-go source-6
                       dest-6))
   Start: ((rolling convoy-6
                  source-6 dest-6))
   Action: (tell convoy-3
                (to-go source-3
                       mid-s))
   Start: ((rolling convoy-3
                  source-3 mid-s))
   Action: (tell convoy-2
                (to-go source-2
                       mid-s))
   Start: ((rolling convoy-2
                  source-2 mid-s))
   Action: (tell convoy-1
                (to-go source-1
                       mid-s))
   Start: ((rolling convoy-1
                  source-1 mid-s))
   Action: (tell convoy-5
                (to-go source-5
                       dest-5))
   Start: ((rolling convoy-5
                  source-5 dest-5))
   Action: (tell convoy-4
                (to-go source-4
                       dest-4))
    Start: ((rolling convoy-4
                  source-4 dest-4))
   Action: WAIT
```
Time 3601
```
   Stop: ((rolling convoy-3
                  source-3 mid-s)
          (rolling convoy-1
                  source-1 mid-s)
          (rolling convoy-2
                  source-2 mid-s))
   Action: (tell convoy-3
                (to-go mid-s
                       mid-d))
   Start: ((rolling convoy-3
                  mid-s mid-d))
   Action: WAIT
```
Time 7202
```
   Stop: ((rolling convoy-3
                   mid-s mid-d))
   Action: (tell convoy-2
                 (to-go mid-s mid-d))
   Start: ((rolling convoy-2
                   mid-s mid-d))
   Action: (tell convoy-3
                 (to-go mid-d dest-3))
   Start: ((rolling convoy-3
                   mid-d dest-3))
   Action: WAIT
```
Time 10801
```
   Stop: ((rolling convoy-6
                   source-6 dest-6)
          (rolling convoy-5
                   source-5 dest-5)
          (rolling convoy-4
                   source-4 dest-4))
   Action: WAIT
```
Time 10802
```
   Stop: ((rolling convoy-3
                   mid-d dest-3)
          (rolling convoy-2
                   mid-s mid-d))
   Action: (tell convoy-1
                 (to-go mid-s mid-d))
   Start: ((rolling convoy-1
                   mid-s mid-d))
   Action: (tell convoy-2
                 (to-go mid-d dest-2))
   Start: ((rolling convoy-2
                   mid-d dest-2))
   Action: WAIT
```
Time 14403
```
   Stop: ((rolling convoy-1
                   mid-s mid-d)
          (rolling convoy-2
                   mid-d dest-2))
   Action: (tell convoy-1
                 (to-go mid-d dest-1))
   Start: ((rolling convoy-1
                   mid-d dest-1))
   Action: WAIT
```
Time 18004
```
   Stop: ((rolling convoy-1
                   mid-d dest-1))
```

Times are in seconds; they are not exact multiples of 3600 (= 1 hr) because arithmetic in trigger-time finder is not exact.

Figure 5: Plan found for 6 convoys ($n = 3$)

.

Size	*Steps*	*G-Nodes*	*Matches*	*Time*	*T/step*	*T/node*	*T/match*	*Nodes/step*
1	4	178	340	3455	864	19	10	44
2	8	913	1637	17614	2201	19	10	114
3	12	2334	4076	49578	4131	21	12	194
4	16	4715	8107	112550	7034	23	13	294
5	20	8278	14094	242650	12132	29	17	414
6	24	13251	22409	478194	19924	36	21	515

Steps is the number of steps in the plan (not including `waits`). *G-Nodes* is the number of goal nodes constructed. *Matches* is the number of maximal matches that were performed. *Time* is the planner's running time in milliseconds. *T/step* is the time per step in the solution plan. *T/match* is the time per maximal match.

Table 1: Results for the convoy domain

including the web-service domain described in (McDermott 2002), and the sort of autonomous processes described here.

The bad news is that building the regression-match graph is too expensive. As far as I can tell, this conclusion has nothing to do with processes per se, but is merely the observation that for large problems Optop can't keep up with planners based on Graphplan (Blum & Furst 1995), i.e., most "modern" planners. I remain hopeful, however, that ways can be found to speed up the process of building or editing the regression-match graph. The reason to pursue this idea is that the Graphplan trick of reasoning about all possible feasible actions is just not going to work when actions can have real-valued parameters.

The simple Convoys domain can be made more realistic in several ways. One could distinguish between paved and unpaved roads, provide predictable weather events that influence the passability of the roads, make the interference between convoys less heavy-handed, and so forth. Most of these changes would make it harder to construct a series of test problems; and would also probably make the domain easier, by eliminating the symmetry it now contains.

I can't claim to have solved the problem of optimally coordinating processes. For one thing, Optop handles only *search-based* optimization, not *parameter-based* optimization. That is, given a choice between various actions, it will often pick the one that yields the best final situation. But if an action has an uninstantiated free variable, it uses a local constraint solver to find a value for it that satisfies all the known constraints, without regard to the impact of that value on the value of the objective function in the final situation. It would be nice if some values could be left as free variables, so that in the final situation one had something to optimize. Alas, there is no obvious way to do this and still be able to run the plausible projector. Of course, if the parameter space is multidimensional, finding an optimal point in it is hard no matter how complex the interaction is between the optimizer and the planner.

References

Atkin, M.; Westbrook, D.; and Cohen, P. R. 1999. Capture the Flag: Military simulation meets computer games. In *Proceedings of AAAI Spring Symposium Series on AI and Computer Games*, 1–5.

Blum, A. L., and Furst, M. L. 1995. Fast planning through planning graph analysis. In *Proc. Ijcai*, volume 14, 1636–1642.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2).

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A fast and robust action selection mechanism for planning. In *Proc. AAAI-97*.

Chandy, K. M., and Misra, J. 1988. *Parallel program design : a foundation*. Addison-Wesley.

Do, M., and Kambhampati, S. 2001. Sapa: A Domain-independent Heuristic Metric Temporal Planner. In *Proc. ECP-01*.

Fox, M., and Long, D. 2001. Pddl. 2.1: An Extension to PDDL for Expressing Temporal Planning Domains available at `http://www.dur.ac.uk/d.p.long/pddl2.ps.gz`.

McDermott, D., and Burstein, M. 2002. Extending an estimated-regression planner for multi-agent planning. In *Proc. AAAI Workshop on Planning by and for Multi-Agent Systems*.

McDermott, D. 1996. A Heuristic Estimator for Means-ends Analysis in Planning. In *Proc. International Conference on AI Planning Systems*, 142–149.

McDermott, D. 1998. The Planning Domain Definition Language Manual. Technical Report 1165, Yale Computer Science. (CVC Report 98-003).

McDermott, D. 1999. Using Regression-match Graphs to Control Search in Planning. *Artificial Intelligence* 109(1–2):111–159.

McDermott, D. 2002. Estimated-regression planning for interactions with web services. In *Proc. AI Planning Systems Conference 2002*.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press.

Roscoe, A. 1998. *The theory and practice of concurrency*. Prentice Hall.

Smith, D. E., and Weld, D. S. 1999. Temporal planning with mutual exclusion reasoning. In *Proc. Ijcai*, 326–337.