Using Regression-Match Graphs to Control Search in Planning

Drew McDermott

January 11, 1999

Abstract

Classical planning is the problem of finding a sequence of actions to achieve a goal given an exact characterization of a domain. An algorithm to solve this problem is presented, which searches a space of *plan prefixes,* trying to extend one of them to a complete sequence of actions. It is guided by a heuristic estimator based on *regression-match graphs,* which attempt to characterize the entire subgoal structure of the remaining part of the problem. These graphs simplify the structure by neglecting goal interactions and by assuming that variables in goal conjunctions should be bound in such a way as to make as many conjuncts as possible true without further work. In some domains, these approximations work very well, and experiments show that many classical-planning problems can solved with very little search.

1 Definition of the Problem

The *classical planning* problem is to generate a sequence of actions that make a given proposition true, in a domain in which there is perfect information about the initial state of the world and the effects of every action. Problems of this type are of practical interest, for instance in tightly controlled domains such as manufacturing, and many algorithms have been proposed for solving them. However, none of them have been applied in practical domains.[1] The main reason is that all interesting classes of classical-planning problems are intractable [11], and therefore all planning algorithms must resort to search. However, there is hope that for some kinds of problems there are algorithms that do well enough in spite of the intractability. Some recent algorithms succeed by searching nontraditional spaces [4, 21]. In this paper, the focus is on improving the performance of classical-planning algorithms by finding improved heuristic estimators for controlling search in a traditional space.

I will assume that world states (henceforth called *situations*) are described as collections of atomic propositions, and that actions[2] are described using PDDL, the Planning Domain Definition Language [25], developed for the AIPS-98 planning competition. This is a descendant of the University

[1] Most applications of "practical planning" algorithms such as Sipe [38] and O-plan [9] have made good use of the plan-management capacities of these systems, but not much use of their plan-search capacities.

[2] A note on terminology: I reserve the term *operator* to refer to transitions in the space of plans, as explained below; transitions between world states are called *actions.* Analogously, I use the term *state* to refer to the state of a search process, and reserve *situation* to refer to the state of the world, considered as a set of atomic propositions. A *proposition* is a fact; an *atomic proposition* is the proposition denoted by an atomic formula or its negation. An *atomic formula* is one consisting of a predicate followed by several arguments, which in this paper will be written as $(P\ a_1\ a_2\ \ldots a_n)$. Please note that the existence of *propositional logic*, in which the objects of study are unanalyzed propositions, does not mean that propositions cannot have more complex structure in other frameworks.

of Washington notation, which is based on Pednault's Action Description Language [30]. These notations have a flavor like the Strips notation [13], and share its essential weakness, which is that it is good with propositions and bad with numbers and geometry.

Tables 1 through 3 show the PDDL definitions for a domain that has been used for one of the experiments reported below. The notation tends to be Lispish, with atomic formulas and terms being written (*f* —*args*—). Actions are described as terms whose arguments are variables, prefixed by question marks. The function in such a term is called an *action functor*. Each action has a *precondition* which must be true for the action to be feasible, and several *effects* which will occur in the next situation if the action is executed. Effects are defined by the following recursive definition:

1. *p*: Proposition *p* becomes true in the next situation. (*p* must not have functor `not`, `when`, `forall` or `and`.)
2. (`not` *p*): Proposition *p* becomes false in the next situation.
3. (`change` *f e*): The value *f* changes to the value of expression *e*. I will explain exactly what this means in Section 3.1.
4. (`when` *p e*): If proposition *p* is true in the *current situation,* then effect *e* occurs.
5. (`and` $e_1 \ldots e_n$): All of the effects e_i occur.
6. (`forall` (—*vars*—) *e*): Every effect obtained by substituting objects for the variables *vars* in *e* occurs.

Effects of the first two kinds are called *literal effects*, in parallel with the usual definition of a *literal* as an atomic formula or the negation of an atomic formula.

Note that all the free variables in an effect must be bound in preconditions, either in a proposition from the `:precondition` field of an action definition, or in some *p* from a clause (`when` *p*`...`) that governs the effect in question. All variables must occur as `:parameters` of the action being defined, or be explicitly quantified in the `:vars` field of the action or in a `forall`. Note that when variables are bound they are given a type. Some types, such as `integer`, are defined in all PDDL theories. Others, like `key`, are specific to this domain.

So, according the definition of `put_down` in Table 2, if the robot is carrying key `?k`, then, if the robot is at an arbitrary location $\langle ?i, ?j \rangle$, then one effect of (`put_down ?k`) is that `?k` is now located at $\langle$`?i,?j`$\rangle$. If the robot is located in two places, `?k` will be in two places; if the robot is not anywhere, the key will not be anywhere. Of course, we arrange things so that these pathological cases never happen.

There is no way to use existential quantifiers in action definitions, or to indicate that an action has disjunctive effects.

Table 3 completes the domain definition by defining adjacency. Two intersections are adjacent if they differ by 1 in either the *x* or *y* coordinate. The proposition (`equation` e_1 e_2) just means that e_1 and e_2 are equal. The planner is able to solve simple equations such as (`equation (+ ?i 1) t`), binding `?i` to 4. The proposition (`bounded-int` *i l h*) is true for all integers *i* between *l* and *h*. The planner can solve such goals if *l* and *h* are known. In this case, *l* and *h* are simple arithmetic functions of the *domain variable* `coord_lim`, which is bound to 5 in the domain definition.

This action formalism has two valuable properties. One is that it is easy to compute the effect of an action sequence. Given a complete description of an initial situation S_0, it is easy to generate a complete description of the situation after executing $\langle A_1, \ldots, A_n \rangle$ starting from that initial situation. I call this situation the *result* of executing the action sequence, written $result(S_0, \langle A_1, \ldots, A_n \rangle)$.

It is also easy, as I will discuss below, to go the other way: given a proposition and an action sequence, to infer what must be true before the sequence is executed in order for the proposition to be true afterward.

I can now give a formal definition of a *planning problem:* It is a tuple $\langle \mathcal{A}, \mathcal{I}, G \rangle$, where

- $\mathcal{A}$ is a *domain definition,* in the format exemplified above;

```
(define (domain grid)
   (:requirements :typing :expression-evaluation
                  :conditional-effects :existential-preconditions)
   (:types coord - integer shape key agent direction - object)
   (:domain-variables (coord_lim 5) - integer)
   (:predicates
      (legal_coord ?i - integer)
      (locked ?i ?j - integer)
      (loc_shape ?i ?j - integer ?c - shape)
      (key_shape ?k - key ?c - shape)
      (at ?a - (either agent key) ?i ?j - integer)
      (carrying ?a - agent ?b - key)
      (adjacent ?i ?j ?i1 ?j1 - integer ?d - direction)
      (consecutive ?i ?j  - integer))
   (:constants left right up down - direction
               triangle circle diamond - shape
               robot - agent)

   (:action move
       :parameters (?dir - direction)
       :vars (?i ?j ?i1 ?j1 - integer)
       :precondition (and (at robot ?i ?j)
                          (adjacent ?i ?j ?i1 ?j1 ?dir)
                          (not (locked ?i1 ?j1)))
       :effect (and (not (at robot ?i ?j))
                    (at robot ?i1 ?j1)))

   (:action unlock
       :parameters (?i1 ?j1 - integer)
       :vars (?i ?j - integer)
       :precondition (and (at robot ?i ?j)
                          (exists (?dir - direction)
                              (adjacent ?i ?j ?i1 ?j1 ?dir))
                          (exists (?k - key ?c - shape)
                             (and (carrying robot ?k)
                                  (loc_shape ?i1 ?j1 ?c)
                                  (key_shape ?k ?c))))
       :effect (when (locked ?i1 ?j1)
                     (not (locked ?i1 ?j1))))
   ...
```

Table 1: Grid-World Definition — Part 1

- $\mathcal{I}$ is an *initial situation description*, a *complete* description of all true atomic formulas. We will use a "closed-world assumption" to keep these descriptions finite: any atomic formula not mentioned is assumed to be false;
- G is a *problem goal*, a proposition to be made true. This is a conjunction of literals (atomic formulas or their negations), possibly containing free variables.

A *solution* to a planning problem $\langle \mathcal{A}, \mathcal{I}, P \rangle$ is a sequence $\langle A_1, \ldots, A_n \rangle$ of variable-free action

```
...
(:action pick_up
    :parameters (?k - key)
    :vars (?i ?j - integer)
    :precondition (and (at ?k ?i ?j)
                       (at robot ?i ?j))
    :effect (and (not (at ?k ?I ?J))
                 (carrying robot ?k)
                 (forall (?k1 - key)
                    (when (carrying robot ?k1)
                          (and (not (carrying robot ?k1))
                               (at ?k1 ?i ?j))))))

(:action put_down
    :parameters (?k - key)
    :precondition (carrying robot ?k)
    :effect (and (not (carrying robot ?k))
                 (forall (?i ?j - integer)
                    (when (at robot ?i ?j)
                          (at ?k ?i ?j))))))
```

Table 2: Grid-World Definition — Part 2

terms, such that in the situation resulting from executing the sequence starting in $\mathcal{I}$, some instance of P is true.

2 Means-Ends Analysis

The historically dominant framework for solving planning problems is *refinement search.* A refinement search goes on in a space of *potential plans.* A potential plan is a partial sketch of a plan, which can be filled out by applying various *planning operators.* Different planners use different notions of potential plan, and different operators for transforming one potential plan into another. What makes it a "refinement" search is that a potential plan can be thought of as defining a set of plans — its *completions* —, and operators can be thought of as narrowing these sets. When potential plan P_1 is transformed into potential plan P_2, that corresponds to moving from the set of completions of P_1 to the set of completions of P_2, a subset. The search stops when a potential plan is found all of whose completions are solutions to the problem[20].

In this paper, I will be discussing a very simple refinement search space for classical planning. Potential plans are just *plan prefixes,* that is, sequences of actions that the planner is trying to extend to a solution plan. The novel contribution is a method of computing an estimate of how much work is required to finish a plan. Most of the recent research in this area has been in a quite different paradigm, using search states that consist of networks of partially ordered steps. Search control in this paradigm consists mainly of deciding, using local criteria, which "flaw" in the current partial plan to fix. [15, 33]. There has been practically no work on heuristic estimators for comparing potential plans. As a consequence, these planners often search through thousands of plans to solve seemingly simple problems.

The alternative search space I describe is much closer to the space searched by the Prodigy planner [36, 34], in that it is based on *means-ends analysis,* a classic search technique first embodied in the GPS system [26, 10]. The principal difference is that Prodigy, like GPS and Strips [13], uses as

```
(define (addendum adjacent-def)
   (:domain grid)
   (:axiom
      :vars (?i ?j ?i1 - integer)
      :implies (adjacent ?i ?j ?i1 ?j right)
      :context (and (equation (+ ?i 1) ?i1)
                    (legal_coord ?i)
                    (legal_coord ?i1)))
   (:axiom
      :vars (?i ?j ?i1 - integer)
      :implies (adjacent ?i ?j ?i1 ?j left)
      :context (and (equation (- ?i 1) ?i1)
                    (legal_coord ?i)
                    (legal_coord ?i1)))
   (:axiom
      :vars (?i ?j ?j1 - integer)
      :implies (adjacent ?i ?j ?i ?j1 up)
      :context (and (equation (+ ?j 1) ?j1)
                    (legal_coord ?j)
                    (legal_coord ?j1)))
   (:axiom
      :vars (?i ?j ?j1 - integer)
      :implies (adjacent ?i ?j ?i ?j1 down)
      :context (and (equation (- ?j 1) ?j1)
                    (legal_coord ?j)
                    (legal_coord ?j1)))
   (:axiom
      :vars (?i - integer)
      :implies (legal_coord ?i)
      :context (bounded-int ?i (- 1 coord_lim) (- coord_lim 1))))
```

Table 3: Grid-World Definition — Part 3

a search state an ordered pair containing a plan prefix and a goal structure (called the "head plan" and "tail plan" respectively by the Prodigy group). There are two sorts of operators: those that add steps to the prefix, and those that commit to a particular action for achieving an outstanding goal. In my framework, the goal structure is generated anew at each state, and represents (in a sense made precise below) *all possible* ways of achieving the original goal. Prefix lengthening is the only search operator that is used, that is, the only operation that actually moves the planner through the space of partial plans, hopefully toward a solution.

Partial-order planners have the advantage that, when goals interact only weakly, they can be planned for independently and the results combined. But they have a big disadvantage, namely, that they cannot keep a complete description of any intermediate situation that will arise during the course of plan execution. Hence it is not possible to compare intermediate situations to the goal description to look for directions in which to move. The best they can do is to compare individual propositions that are created by steps proposed so far. Sometimes these comparisons give information that is too "local," so that the planner flounders a lot in trying to decide what to do next. The planner I describe invests a large effort in working out these comparisons, so that for many domains it has a good picture of the next move to make.

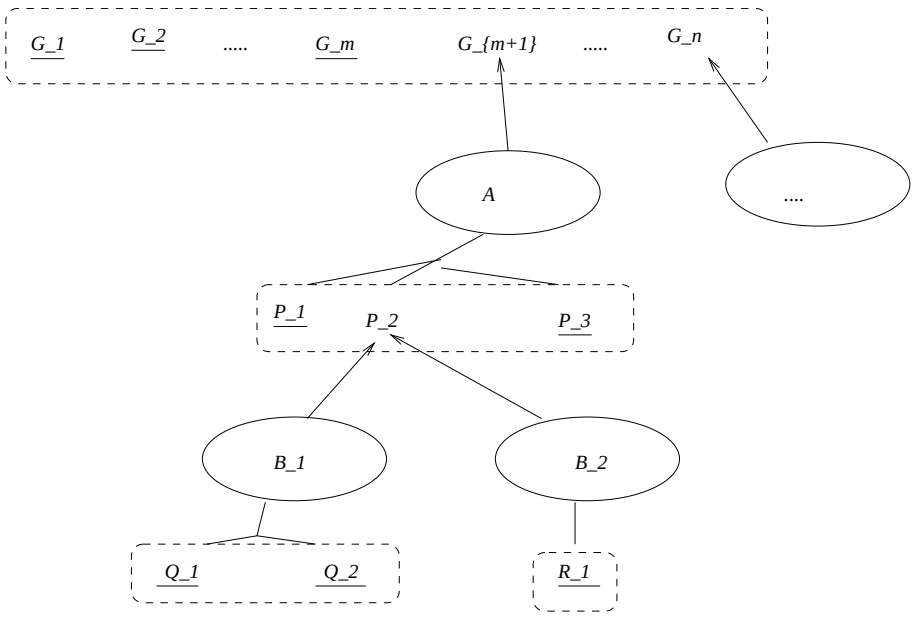


Figure 1: Analysis of Actions and Goals

2.1 Plan-Prefix-Space Search

Suppose the planner is trying to complete the plan prefix $\langle A_1, A_2, \ldots, A_k \rangle$. I will use the term *current situation* to denote the situation that obtains after $\langle A_1, A_2, \ldots, A_k \rangle$ is executed beginning in the initial situation. Suppose the goal description is $G_1 \wedge \ldots \wedge G_n$. If we *match* this goal against the current situation, we may discover that (say) $G_1 \wedge \ldots G_m$ are true in that situation, and $G_{m+1} \wedge \ldots \wedge G_n$ are false. (In Figure 1, goals true in the current situation are underlined.) We can say that $\{G_{m+1}, \ldots, G_n\}$ is the set of *differences* between the current situation and the goal description. (For now, I will ignore any variables that might occur in the goals; pretend there aren't any.) This notion of matching to find differences goes all the way back to GPS [26]. If there are no differences, then the current plan prefix is a solution to the problem. Otherwise, it would seem that a reasonable idea for improving the plan prefix is to find an action that achieves one of the goals (G_{m+1}, say) and tack it on to the end. The planner knows exactly which actions achieve which goals, because every action has a well defined list of literals that it adds. This idea is called *means-ends analysis,* and in this form was first developed by the Strips group [13].

The problem, of course, is that an action that achieves G_{m+1} may not be feasible in the current situation. In Figure 1, action A achieves the goal conjunct, but has preconditions that are not all true. The obvious tack to take at this point is to "recurse," and repeat the same operations on the preconditions of A. Some will be differences, which the planner can achieve by proposing actions, which in turn have preconditions, and so on. The tree obtained by considering all possible actions and matches is called the *regression-match graph.* I will be more precise later about exactly what it consists of and how it is constructed, and why it is a graph and not a tree. If we continue to pretend it's a tree for a moment, we can see that its "leaves" are conditions that are true now, or that are unachievable by any action.[3]

The regression-match graph is interesting for several reasons:

- Its size serves as an estimate of how hard it is to complete the current plan prefix.

[3] In some domains, the tree may be infinite or unmanageably large, and a depth cutoff can be imposed, in which case some goals are leaves because they appear too deep; they behave like unachievable goals for our purposes. See Section 3.3.

- If the tree below a goal bottoms out in unachievable subgoals, then that is evidence that the goal is impossible to achieve.
- Actions just above the "leaves" of the graph are likely to be those that are feasible in the current situation, and relevant to achieving the overall goal.

Looking again at Figure 1, we can see that the graph structure suggests two possible ways to achieve G_{m+1}: First B_1, then A; or first B_2, then A. Either way, we can estimate that achieving this conjunct will take 2 steps. If the graph is completed with similar analyses for the other conjuncts $G_{m+2}, \ldots$, then we will get estimates for how hard it is to achieve those goals, and other possible first steps might emerge.

Of course, the planner has no way of knowing which action is best to try as the first step toward solving its problem, B_1, B_2, or one of the actions that occur in the analysis of the other goals. But it can try them all. That is, if the initial search state is the empty plan prefix $\langle\rangle$, then the possible next states include $\langle B_1 \rangle$ and $\langle B_2 \rangle$. See Figure 3. Each state corresponds to a different "current situation," either that obtaining after B_1 is executed, or that obtaining after B_2 is executed. In each situation we can repeat the construction of the regression-match graph. This exercise will result in finding new estimated efforts for the goals, and new feasible actions. It may reveal unforeseen side effects. For example, B_1 might delete G_1, thus taking away part of what it gains. So in the state $\langle B_1 \rangle$, the effort of reachieving G_1 will be added into the total effort, causing $\langle B_2 \rangle$ to become more attractive.

It is common to label this kind of planner a "state-space search," and it is, but let me hasten to point out that the word "state" in this phrase means "world state," or "situation." I normally use the word "state" to refer to search states, and in this sense the phrase "state-space search" is tautologous. In my terminology, *search states* are *plan prefixes* of the form $\langle A_1, \ldots, A_k \rangle$. Each such prefix corresponds to a world *situation,* namely that obtaining after A_1, A_2, $\ldots$, A_k are executed starting in the initial situation. Hence the planner is for all intents and purposes a situation-space searcher.

I should also point out that the arrows in Figure 1 may appear to go the wrong way, from children to parent nodes. The reason is that they are intended to reflect the flow of time and causality, rather than dependency. I will continue to refer to the nodes at the tails of the arrows as the *children* of the nodes at the heads. If node N_2 can be reached by following zero or more child arcs from N_1 then N_2 is a *descendant* of N_1, and N_1 is an *ancestor* of N_2. In these definitions, N_1 and N_2 may be goal conjunctions, cohorts, or goal literals, and the arcs may be labeled with action terms or maximal matches, or, in the case of an arc from a cohort to its elements, left unlabeled.

3 Formal Treatment

Now let's be more precise about how the regression-graph is defined. It has a layered, tripartite structure. There are three types of nodes, and a given type of node is always connected to the same types of neighbors, and by the same kinds of edges. Two of the edge categories are nontrivial, and I will explain them before describing the overall graph structure. The two nontrivial categories are *regression* edges and *match* edges.

3.1 Regression

The *regression of a proposition P through an action A*, written $[A]^R(P)$ is the weakest condition Q such that if Q is true before A is executed, then P will be true afterward. This condition is easy to compute given our Strips-style action formalism [30]. It depends on the action definitions of a particular theory, and when that is important I will write $[A]^R_{theory}P$ to relativize it.

We can always write the regression of P as

$$[A]^R(P) = \neg P \wedge [A]^{\overline{R}}(P) \vee P \wedge [A]^{\underline{R}}(P)$$

where $[A]^{\overline{R}}(P)$ is the weakest precondition that causes P to become true after A when it is false before; and $[A]^{\underline{R}}(P)$ is the weakest precondition that keeps P true after A when it is true before. We call $[A]^{\overline{R}}(P)$ the *causation precondition* for P before A, and $[A]^{\underline{R}}(P)$ the *preservation precondition* for P before A [29, 30]. Until Section 3.7, I will focus only on causation preconditions.

When P is a literal, to compute $[A]^{\overline{R}}_T(P)$, it suffices to take the definition of A in T, and examine all the literal effects. If some literal effect unifies with P with unifier θ, then $\theta(R)$ is the desired condition, where R is the precondition of A. That is, if A is feasible at all, then P will be caused by it. If an effect is of the form $C \Rightarrow E$, then the algorithm is applied recursively to E, taking $R \wedge C$ as the relevant precondition. C is not necessary to the feasibility of A, but is necessary to A's causing P; in Pednault's terminology, it is a *secondary precondition*.

The PDDL language allows the specification of numerical effects using the `change` notation, which can effect the truth value of formulas of the form `(fluent-test (`r e_1 e_2`))`, where r is a an inequality ($<$, $>$, $\leq$, $\geq$, $\neq$, or $=$) and e_1 and e_2 are arithmetic expressions. If f occurs in e_1 or e_2, then the effect

`(change` f e`)`

can potentially make the inequality true. We get the usual precondition, plus the formula

`(fluent-test` r e'_1 e'_2`)`

which is obtained by substituting e for f in the original goal. In these goals, f is a *fluent term* such as `(water_in jug2)`, that is, a term whose value can change from situation to situation.

3.2 Matching

The second concept we need in order to understand regression-match graphs is matching. In the simple examples above, I left variables out of the propositions in the graph. But in general, when we take a goal and regress through an action term with variables, then the result will have variables, even if the goal did not. For example, in the grid world, the result of computing

$$[(\texttt{pick\_up ?k2})]^{\overline{R}}((\texttt{at k1 2 } -3))$$

is

$$(\texttt{carrying robotk1}) \wedge (\texttt{at robot 2 } -3) \wedge (\texttt{at ?k2 2 } -3)$$

In other words, one way to cause `k1` to be at $\langle 2, -3 \rangle$ is to be at that location and pick up some other key `?k2`.

A planner can handle the subgoal `(at ?k2 2 -3)` in one of several ways. One way is to treat `?k2` as an *unknown*, a global variable that is "solved for" during the course of the remaining planning process. Typically, it gets bound when the planner decides what effect of what step to identify with this goal. If a step causes, e.g., `(at key14 2 -3)` to become true, then the planner can achieve the later goal `(at ?k2 2 -3)` by binding `?k2` everywhere it occurs to `key14`.

The only problem with that idea is that it forces the planner to do nothing with the goal when it is first produced. In particular, it can have no idea how difficult it is to achieve it compared to other subgoals it might have adopted instead.

Another approach is to avoid variables by substituting variables in all meaningful ways as early as possible, either during the formulation of the problem, as SATPLAN does [21], or during the construction of the planning graph that Graphplan uses [5]. This technique runs the risk of generating many irrelevant atomic formulas.

An attractive alternative is to guess likely values of the variable as soon as it occurs in a goal. Suppose that the planner sees that `key14` is at $\langle 2, -3 \rangle$ in the current situation. Then it is a plausible

guess that as further plans unfold `key14` will remain there, and hence remain a candidate for binding to `?k2`. Hence it is reasonable to estimate that the number of steps required to achieve `(at ?k2 2 -3)` is zero.

In general, the idea is to bind variables in such a way as to make as many conjuncts in a goal true as possible. That is, whenever a goal arises of the form

$$P_1(?\mathbf{x}) \wedge P_2(?\mathbf{x}) \wedge \ldots \wedge P_k(?\mathbf{x})$$

where `?x` represents all the variables that occur in the conjunction, the planner should find bindings of `?x` to constants so that as many of the P_i as possible are made true. The remaining conjuncts are *differences* between the goal and the current situation. One estimate of the effort required to achieve the conjunction is the effort required to achieve the differences resulting from binding the variables.

To be more precise, define the *hit set* for a substitution θ in a set of formulas $P =$

$$\{P_1(?\mathbf{x}), P_2(?\mathbf{x}), \ldots, P_k(?\mathbf{x})\}$$

with respect to situation S to be the set of all $P_i(?\mathbf{x})$ such that $\theta(P_i(?\mathbf{x}))$ is variable-free and true in S. I'll write this as $hit(\theta, P, S)$. Following standard terminology, I'll use the word *ground* to mean "variable-free."

Now define a *match substitution* for the set $P = \{P_1(?\mathbf{x}), P_2(?\mathbf{x}), \ldots, P_k(?\mathbf{x})\}$ with respect to situation S to be a substitution θ that binds (some or all of) the variables `?x` to constants so that the conjunction is split into two disjoint parts: P_{true} and P_{false} such that:

1. P_{true} is the hit set for θ with respect to S;
2. $P_{false} = P - P_{true}$ is a set of conjuncts, not necessarily ground, that have no true instances in S;
3. for any proper subset $\theta' \subset \theta$, the hit set for θ' in P is a proper subset of P_{true}.

A *maximal match* of $P = P_1(?\mathbf{x}) \wedge P_2(?\mathbf{x}) \wedge \ldots \wedge P_k(?\mathbf{x})\}$ with situation S is then a substitution μ that assigns constants to all the variables `?x`, such that there is a match substitution $\theta \subseteq \mu$. θ will differ from μ only if $\theta(P_{false})$ contains free variables; μ must bind them all to whatever objects make sense, as I discuss shortly. The *difference set* of the maximal match, written $diff(\mu, P, S)$ is the set $\mu(P_{false}) = \mu(P - hit(\theta, P, S))$.

For example, consider the goal $P =$`(at robot ?i ?j)` $\wedge$ `(at key15 ?i ?j)`, which might arise as a precondition of the action `(pick_up key15)`. Suppose the robot is currently at location $\langle 2, 3\rangle$, and `key15` is currently at location $\langle 5, 6\rangle$. There are two maximal matches:

Substitution	*Differences*
`i=2,j=3`	`(at key15 2 3)`
`i=5,j=6`	`(at robot 5 6)`

Differences give rise to subgoals. In the example, two ways of achieving P are proposed: Get `key15` to $\langle 2, 3\rangle$, or get the robot to `5,6`. Of course, the first of these is silly if the reason to achieve it is to pick up `key15`, but further analysis will be required to decide that.

In the example, the substitution μ is identical to θ. They differ when there are free variables that occur in P_{false}, which must be bound by μ. The formal definition allows these variables to be bound to arbitrary constants, which works fine for formal purposes. However, in practice we usually get great benefit out of the following optimization. Suppose that $P_i(?\mathbf{y})$ is a conjunct of P_{false}, where `?y` represents all the variables of `?x` that are left unbound by θ. Some constants will make no sense when plugged in as arguments to P_i. The resulting differences will be goals like `carrying(robot,robot)`, or `at(key15,2,key13)`. We avoid all such absurdities in practice by keeping track of the types of arguments for predicates and action functors, and plugging in only objects of the appropriate types.

Note that we never bind a variable in such a way as to make a conjunct false. A binding is added to θ only if it makes some conjunct true; and a binding is added to μ only if there are conjuncts containing the binding's variable which are false for every way the variable could be bound.

3.3 Regression-Match Graphs

We are now in a position to be precise about regression-match graphs. A *regression-match graph* for a goal G (a conjunction of literals, possibly containing variables) is a tuple $\langle C, L, H, E\rangle$ with the following properties:

- C is a collection of *goal conjunction nodes,* each labeled with a conjunction of literals, possibly containing variables (I distinguish nodes from their labels only to allow two nodes to have the same label; in what follows I will relax the distinction and just use the term "goal conjunction");
- There is a node $\in C$ labeled with G;
- L is a collection of *goal literals*, each containing no variables;
- H is a collection of *cohorts,* each a collection of goal literals;
- E is a collection of *edges*, $\subseteq (C \times H) \cup (H \times L) \cup (L \times C)$, some labeled and some unlabeled;
- for every $g \in C$, and for every maximal match μ of g with S, $\mu(g) \in H$, and edge $\langle g, \mu(g)\rangle \in E$, labeled with μ;
- for every $\mu(g) \in H$, if p is a conjunct of $\mu(g)$, then $p \in L$ and there is an edge $\langle \mu(g), p\rangle \in E$;
- for every $p \in L$ such that p is not true in S, and for every n-argument action functor A, if

$$[(A\ ?v_1\ \ldots\ ?v_n)]^{\overline{R}}(p)\ =\ g_1 \vee \ldots \vee g_k$$

  in disjunctive normal form, where the $?v_j$ are distinct variables, and g_i is a conjunction of literals that is not identically **false**, then there is a goal conjunction $\in C$ labeled with g_i, and $\langle p, g_i\rangle \in E$, labeled with $(A ?v_1 \ldots ?v_n)$. Each such goal conjunction is called a *reduction* of p.
- C, L, H, and E have no elements not required by a finite number of applications of the previous rules in this list.

Obviously the regression-match graph for a goal is unique, up to renaming of the variables in goal conjunctions.

Let me pause to insert an example, drawn from the grid world. In the initial situation, the robot and key `K` are at $\langle 0, 0\rangle$, and the goal is to get `K` to location $\langle 1, 0\rangle$. The initial goal conjunction consists of a single variable-free goal, so it has a trivial maximal match and just one cohort. Goal conjunctions are indicated by dotted lozenges drawn around a group of literals. Cohorts are indicated by edges from their elements (which are goal literals) to the goal conjunction they derive from, with edges labeled with the maximal matches used to derive them. In this simple example, the initial goal contains just one ground conjunct, so there is an empty substitution labeling the arc to it from the goal literal that constitutes its only cohort. There is just one action term such that $[A]^{\overline{R}}$(`(at K 1 0)`) is not identically **false**, namely `(put_down ?v)`. The regression yields a single goal conjunction. As at the top, it contains no variables, so it has just one cohort, consisting of the two goal literals `(at robot 1 0)` and `(carrying robot K)`. (Formulas are abbreviated to avoid clutter.)

Interesting things happen at the next layer. First observe the subgraph attached to `(carrying robot K)`. There is a unique action term that leads to this conclusion, `(pick_up ?v)`. The regression of `(carrying robot K)` through this action yields the goal conjunction

`(at robot ?i ?j)`$\wedge$ `(at K ?i ?j)`

This goal conjunction maximally matches the current situation just one way: $\{\mathtt{i} = 0, \mathtt{j} = 0\}$. The resulting cohort has two goal literals that are true in the current situation. The effort required to achieve them is 0.

We can propagate estimates of the efforts for all nodes in the graph using the following recursive definition of the *estimated effort* of goal conjunctions and goal literals:

- If p is a goal literal, and p is true in S, then $EE(p) = 0$.

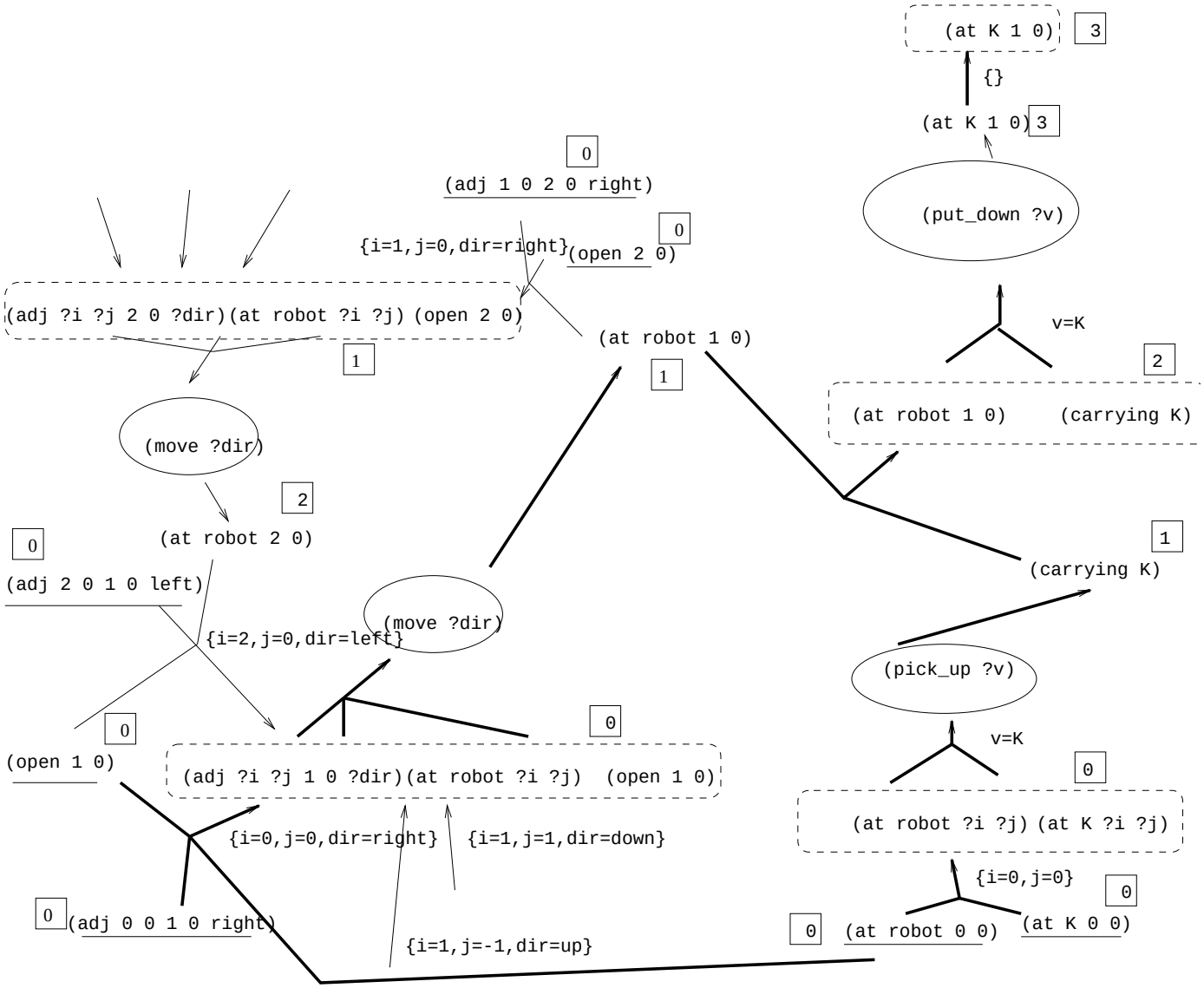


Figure 2: A Regression-Match Graph for a Grid-World Goal

- If p is a goal literal, and p is not true in S, then

$$EE(p) = 1 + \min_{g \in reductions(p)} EE(g)$$

- If g is a goal conjunction, then

$$EE(g) = \min_{h \in cohorts(g)} \sum_{p \in h} EE(p)$$

- If applying the previous rules repeatedly does not assign an effort, then the effort is ∞.

By this definition, the estimated effort of `(carrying robot K)` is 1.
On the other branch, `(at robot 1 0)` is achieved by `(move ?dir)`, yielding the goal conjunction

`(adjacent ?i ?j 1 0 ?dir)` $\wedge$ `(at robot ?i ?j)` $\wedge$ `(open 1 0)`

There are four maximal matches with the current situation:

Substitution	*Differences*
{i=0,j=0,dir=right}	*None*
{i=2,j=0,dir=left}	`(at robot 2 0)`
{i=1,j=1,dir=down}	`(at robot 1 1)`
{i=1,j=-1,dir=up}	`(at robot 1 -1)`

To avoid clutter, only the first two of the resulting cohorts are shown in the figure. The first consists of three goal literals that are all true, so there are no differences, and the estimated effort of the goal conjunction is 0, and the estimated effort of `(at robot 1 0)` is 1. However, the other cohort demonstrates an interesting phenomenon. The only difference obtained if `dir=left` is `(at robot 2 0)`. It gets regressed through `(move ?dir)` in a very similar fashion to the way `(at robot 1 0)` was regressed, and we once again get four maximal matches. This time only one is shown: {i=1,j=0,dir=right}, with difference `(at robot 1 0)`. The intuition is that if the robot were at $\langle 1, 0\rangle$, it could get to $\langle 2, 0\rangle$ by going right. Of course, this tactic makes no sense. But no harm is done. The effort for `(at robot 1 0)` is 1, so the effort for the cohort is 1, the effort for `(at robot 2 0)` is 2, and this value plays no part in determining the value of `(at robot 1 0)`.

Although the regression-match graph often contains cycles, there is an important class of acyclic subgraphs that can be extracted. A *stratified subgraph* is obtained by selecting a single cohort for each goal conjunction, and a single reduction for each goal literal, in such a way that the resulting subgraph contains no cycles. More precisely, a stratified subgraph of a goal literal or goal conjunction G in a regression-match graph $\langle C, L\rangle$ is a subgraph $\langle C', L', H', E'\rangle$ such that

- $C' \subseteq C$, $L' \subseteq L$, $H' \subseteq H$, $E' \subseteq E$, and $G \in C'$;
- for every $g \in C'$, either (a) for every maximal match μ of g with S (the current situation), some conjunct of $\mu(g)$ is an ancestor of g in $\langle C, L\rangle$; or (b) there is a single maximal match μ of g with S and for every conjunct p of $\mu(g)$, $p \in L'$;
- for every $p \in L'$ such that p is not true in S, either there is an n-argument action functor define A, such that
$$[A(?v_1, \ldots, ?v_n)]^{\overline{R}}(p) = g_1 \vee \ldots \vee g_k$$
in disjunctive normal form, where the $?v_j$ are distinct variables, and some g_i is a non-identically-**false** conjunction of literals, in which case one such $g_i \in C'$; or (b) for every such A and v_j, all the g_i are identically false;
- E' consists of exactly the edges both ends of which are in $C' \cup H' \cup L'$.

By *minimal* I mean that if there is another subgraph $\langle C'', L'', H'', E''\rangle$ with these properties, and $C'' \subseteq C'$, $L'' \subseteq L'$, $H'' \subseteq H'$, $E'' \subseteq E'$, then $C'' = C'$, $L'' = L'$, $H'' = H'$, and $E'' = E'$. In Figure 2, one stratified subgraph has been indicated with heavier lines.

A stratified subgraph all of whose leaves are goal literals that are true in S, the current situation, is called a *coherent subgraph*. The subgraph in the figure is also coherent. It should be obvious that there will be a coherent subgraph of the top-level goal conjunction if and only if its estimated effort is $< \infty$. If the estimated top-level effort is ∞, then every stratified subgraph has at least one leaf node that is a goal conjunction with effort ∞, and so there are no coherent subgraphs. Note that a leaf node of a stratified subgraph is not necessarily a leaf node of the original graph.

You can think of a coherent subgraph it as being a "plan sketch" for how to achieve the top-level goal. If an action A occurs in a coherent subgraph with precondition goal conjunction G, and there is a maximal match μ that makes G true in the current situation, then $\mu(A)$ is said to be *an initial step* of the subgraph. In the figure, `(move right)` and *(pick_up K)* are the initial steps of the indicated coherent subgraph. Either one is feasible as a first step in an action sequence. What we hope is that at least one of them *makes sense* as a first step; i.e., that the sequence can then be continued until

the problem is solved. That's true for `(pick_up K)`; whether it's true for `(move right)` depends on whether we count sequences such as ⟨(`moveright`), (`moveleft`), (`pick_upK`), (`moveright`)⟩ as solutions or not.

The actions *allowed by a regression-match graph* are all the actions that occur as initial steps of some coherent subgraph of the top-level goal conjunction. If a regression-match graph has no coherent subgraphs, then no actions are allowed by the graph, and it is a good guess (but not always true) that its top-level goal conjunction is infeasible. The actions allowed by the graph are the only ones my planning algorithm considers.

The *estimated effort* of a stratified subgraph is obtained in the obvious way: The effort of a goal literal is 0 if it is true in S, ∞ if it is an untrue leaf, and otherwise 1+ the effort of its only reduction. The effort of a goal conjunction is ∞ if it is a leaf, otherwise the effort of its only cohort.

In general, the effort of a goal literal is equal to 1+ the efforts of its lowest-effort reductions, unless the literal is true in the current situation, or all its reductions have infinite effort. Any reduction whose effort is finite, and as low as any other reduction of that goal literal is called an *effective* reduction of that goal literal. The others are *ineffective*, and do not influence the estimated effort for the goal literal. We can similarly divide the cohorts of a goal conjunction into an effective and an ineffective category; the effective ones are just those with finite effort less than or equal to the efforts of all the others. An *effective subgraph* of a regression-match graph is one obtained by choosing exactly one effective cohort for the top goal conjunction, then one effective reduction for each untrue element of that cohort, then one effective cohort for each reduction, and so on. It is easy to see that if the estimated effort for the top goal conjunction is finite, then at least one such graph exists, and is a coherent subgraph. The subgraph indicated in Figure 2 is effective. If we completed the diagram in Figure 2, we would see coherent but ineffective subgraphs, such as the one involving the action sequence ⟨..., `(move up)`, `(move right)`, `(move right)`, `(move down)`, `(move left)`, ...⟩.

For every action allowed by the graph, there is an *estimated effort* associated with that action, which is just the lowest effort of any coherent subgraph for which the action is an initial step. If $\mu(A)$ is allowed by the graph, and occurs in an effective subgraph, then it is said to be *favored by* the graph. If the action is favored by the graph, this value is the same as the estimated effort of the overall graph.

Note that if a literal is not added by any action, then it will always have effort ∞. For example, suppose it is true in the current situation that `(at robot 0 0)`, and the goal `(at robot 2 0)` occurs in the regression-match graph. Regressing through the action `(move ?dir)`, we obtain the goal conjunction

`(adjacent ?i ?j 2 0 ?dir)` ∧ `(at robot ?i ?j)` ∧ `(open 2 0)`

This goal has eight maximal matches:

Substitution	*Differences*
{i=1,j=0,dir=right}	`(at robot 1 0)`
{i=3,j=0,dir=left}	`(at robot 3 0)`
{i=2,j=1,dir=down}	`(at robot 2 1)`
{i=2,j=-1,dir=up}	`(at robot 2 -1)`
{i=0,j=0,dir=right}	`(adjacent 0 0 2 0 right)`
{i=0,j=0,dir=left}	`(adjacent 0 0 2 0 left)`
{i=0,j=0,dir=down}	`(adjacent 0 0 2 0 down)`
{i=0,j=0,dir=up}	`(adjacent 0 0 2 0 up)`

But the last four of these are absurd, because there is no way to make $\langle 0,0\rangle$ adjacent to $\langle 2,0\rangle$ from any direction. The estimated effort for the cohort resulting from each match is therefore ∞. In the implementation, such impossible goals are weeded out at a very early stage (see Section 5), and do not actually appear in the graph.

In some domains it may be necessary to impose a depth cutoff on a regression-match graph because the whole graph is infinite, or too large to be manageable. A *depth-limited* regression-match graph with depth limit d is then defined as the largest subgraph of the whole graph in which the depth of the shortest path from the root (G) to any goal conjunction is $\leq d$. We define the depth of a path as the number of goal literals it contains. The depth-limited graph can contain some goal literals with no reductions because they would be too depp. The definition of estimated effort (EE) is the same for depth-limited subgraphs. The definition of stratified subgraph has to change slightly:

- for every $p \in L'$ such that p is not true in S, either p is at depth d; or one of the cases (a) and (b) listed above holds, i.e., p has at most one reduction.

In cases where no natural numbers are involved, the regression-match graph is guaranteed to be finite, and, by an argument analogous to that of [5] for planning graphs, to have size polynomial in the number of symbols in the domain, provided we have a bound on the arity of all the predicates involved. This bound doesn't mean much unless the graphs are of manageable size in practice, but, as illustrated in Section 6, they usually are.

3.4 Search Space

Now I can be more formal about the search space my planner uses. The order in which it searches the space is the topic of Section 3.5.

To define a search space, we need to define three things: the initial state, the operators, and the success criterion. I said in Section 1 that the state space is the set of all action sequences, so you might expect the initial state to be the empty sequence. However, in order to get the details to work out, we actually define the space so that sequences always end in an action allowed by some regression-match graph. That is, if the sequence $\langle A_1, \ldots, A_{n-1}, A_n \rangle$ is in the space, then A_n is allowed by the regression-match graph for the goal with respect to the situation resulting from $\langle A_1, \ldots, A_{n-1} \rangle$.

So: The initial state is the set of all singleton sequences $\langle A \rangle$, where A is some action allowed by the regression-match graph for the goal with respect to the initial situation.[4]

The operators take an action sequence

$$\langle A_1, \ldots, A_n \rangle, \text{ and extend it to } \langle A_1, \ldots, A_n, A_{n+1} \rangle$$

where A_{n+1} is allowed by the regression-match graph for the problem goal with respect to

$$result(\mathcal{I}, \langle A_1, \ldots, A_n \rangle).$$

The success criterion is that a sequence result in a situation where the top-level goal is true.

This is essentially a situation-space search, because the main feature of an action sequence that we care about is the situation that results from it. In particular, if two sequences result in the *same* situation, then it is wasteful to attempt to extend both of them. The second time a sequence is seen, it should be discarded. See Figure 3, in which transitions that cause discards are drawn with a dotted line. (Of course, exactly which transitions cause discards depends on the order in which situations are encountered.) The data structure required to test for repeated occurrence of a situation is described in Section 5.

The only remaining piece of the puzzle to supply is the heuristic evaluation function, the "score" we assign to each search state. If the state is $\langle A_1, \ldots, A_{n-1}, A_n \rangle$, then its score is $n - 1$ plus the estimated effort of A_n in the regression-match graph for the problem goal in the situation $result(\mathcal{I}, \langle A_1, \ldots, A_{n-1} \rangle)$. (Recall that the estimated effort of a feasible action is the effort of the cheapest coherent subgraph of which it is an initial step.)

[4] To be technically precise, we must allow for the case where the goal is true in the initial situation. We can stipulate that in that case the search space has just one state: the empty action sequence.

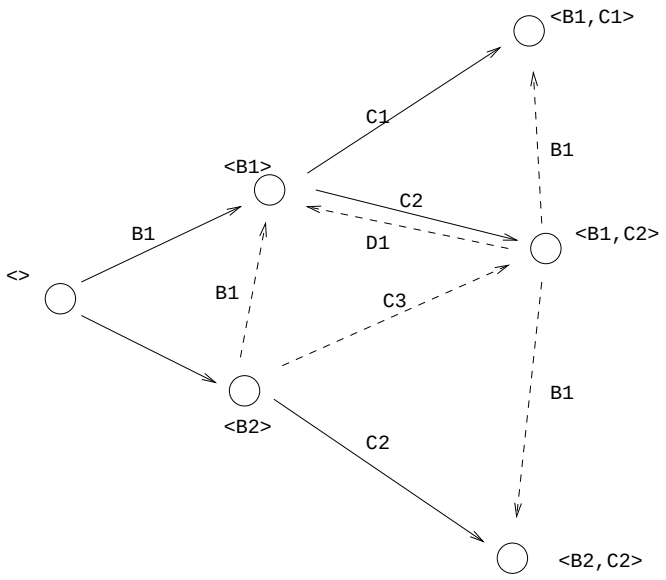


Figure 3: Situation-Space Search

3.5 Search Strategies

Now that we have a search space, we have to choose a strategy to use in exploring it. The search space gives the "legal moves" the planner can make; the strategy decides which ones to try. I have experimented with two strategies: best-first search, and hill-climbing search with restarts.

Best-first search is implemented in the usual way: The program keeps a queue of plan prefixes. It chooses the prefix $\langle A_1, \ldots, A_n \rangle$ with the lowest score (estimated effort), builds the regression-match graph for it, finds all the actions $B_1, B_2, \ldots, B_k$ allowed by the graph, plus their efforts, and generates k new prefixes $\langle A_1, \ldots, A_n, B_j \rangle$ for $1 \leq j \leq k$. These are sorted and merged back onto the queue, and the process is repeated. It terminates as soon as an action sequence is found whose result is a situation in which the problem goal is true.

Best-first search often works quite well, especially for the sort of "toy" problems that are found in the literature. But what I observe in many cases is that estimated effort is a better local estimator than a global one. That is, it often correctly ranks the possible next actions, without necessarily comparing those moves accurately with competitors from a completely different part of the space. If it says that $\langle A_1, \ldots, A_n, B_1 \rangle$ is better than $\langle A_1, \ldots, A_n, B_2 \rangle$, then it's often right. When it reports that $\langle A'_1, \ldots, A'_m \rangle$, a random competitor found on the queue, is just as good as $\langle A_1, \ldots, A_n, B_1 \rangle$, it's often wrong. The reason is that the inaccuracies in estimated effort, are often balanced across near relatives in the space. If one is weighting one subproblem too heavily, the other probably is, too. But for distant relatives the errors tend to depend on the wrong factors. Typically errors in effort estimates tend to be correlated with the length of plan prefixes. The shorter the prefix, the further it is from a solution, the larger the effective subgraph, and the greater the error tends to be. The problem is especially severe if short prefixes look too good. Then a best-first search tends to degenerate into a breadth-first search.

We can usually avoid these problems by the use of hill climbing, in which we discard all but the locally best successors to the current plan prefix, pick one randomly, and pursue it. The *locally best* successors of a search state are those that are better than all the other successors. There may be more than one. We give up on this branch of the search space only when a state has no successors.

In that case we pick at random one of the states that was locally best at some previous state and restart the search from there.[5]

For all the experiments reported in this paper, I used a hybrid search scheme, in which the planner searched best-first until it decided that approach was ineffective, then switched to hill climbing with restart. The details appear in Section 5.

In practice, whenever either search strategy is used, Unpop is supplied with a bound on plan length. This serves two roles: it allows the system to discard plans that are too long, and it can be used as a bound on the depth of the regression-match graph. The latter is the more important bound, because where the depth cuts off influences how big the search space really is and how expensive it is to explore one search state. Applying the maximum plan length as a depth bound is a generous constraint, because the estimated effort extracted from a graph is usually larger than its depth.

In hill-climbing mode, the bound on plan length is often useful for limiting search for reasons having nothing to do with the depth cutoff on the regression-match graph. Without the length limit, Unpop can get itself into searching an infinite branch containing no solutions; it keeps lengthening the plan hoping to get closer. Giving Unpop a length limit may sound like giving it too much of a hint, but it isn't, for the following reason. We have to give it a limit on the number of plans to search anyway, because if the problem has no solution the planner can go for a long time before exhausting the entire search space. In all the experiments, the plan-length bound was set to $\frac{1}{2}B$, where B is the plan-number bound. Unpop uses up a search state for every step it adds to a partial plan. Therefore, if it is distracted by an incorrect partial plan that is discarded only when it reaches length $L \approx \frac{1}{2}B$, then after giving up on that branch it has only another $\frac{1}{2}B$ partial plans to try, so if it's lost it's almost sure to fail. Hence this policy prevents Unpop from using the plan-length bound to skip unpromising plans and go for the correct part of the search space.

3.6 Plan Incoherence

The algorithm as presented so far has an attention-deficit disorder. At each state in the search space, it recomputes the entire regression-match graph, and gives equal weight to all the favored actions. Suppose that the planner is in a part of the search space where there are two overall goals, neither of which is true. The regression-match graph will have two parts, one for each goal. If the planner tries an action that helps achieve Goal 1, then in the graph computed in the resulting situation the subgraph for Goal 1 will be a bit smaller, and the subgraph for Goal 2 will typically be the same size or even a bit larger. If the planner takes an action from the subgraph for Goal 2, then it runs the risk of making that subgraph smaller while the graph for Goal 1 gets larger. This risk is especially great using hill-climbing-with-restart search, because the planner cannot backtrack to undo the damage until it has explored a large portion of the zone of the situation space resulting from the bad action.

An example occurs in the "fridge" problem from the University of Washington corpus (see Section 6.2). To fix a refrigerator, it is necessary to turn the refrigerator off, remove the backplane, change the compressor, reattach the backplane, and turn the refrigerator back on. The problem is difficult because removing the backplane requires unscrewing four screws, and the refrigerator can't be turned on until all four screws are back in place. Let's focus on what happens after one screw has been unscrewed, or, more formally, when the current situation is the result of

$$\langle \texttt{(unscrew s1)} \rangle$$

Figure 4 is a simplified diagram of the situation, in which there are just three screws. Screw `s1` is unscrewed and `s2` remains to be unscrewed. In the regression-match graph, there are three possible

[5] In the earlier report on this work[24], I described this process in terms of Ginsberg and Harvey's *limited-discrepancy search* [16]. The current description is more accurate given what the program actually does in practice. For those familiar with limited-discrepancy search, the following may explain: My program searches the subspace of discrepancy 0. Although I programmed it to follow up with a search of the subspaces of discrepancy 1, 2, 3, etc., it never actually found a solution in any but the subspace of discrepancy 0. It may be of interest that when there is a solution to a planning problem, there is almost always a solution of discrepancy 0, even if it's not optimal.

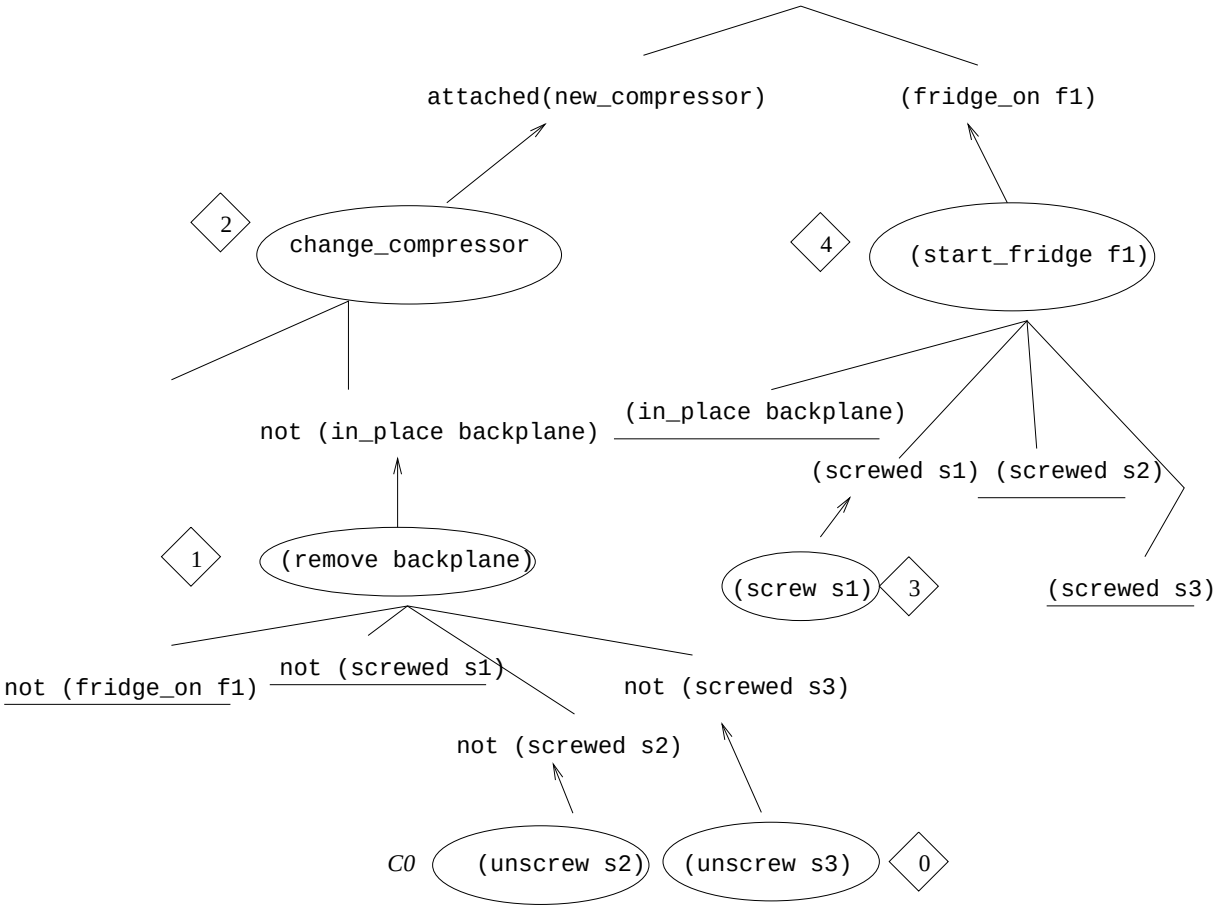


Figure 4: The Infamous "Fridge" Problem

initial steps: `(unscrew s2)` and `(unscrew s3)` to achieve a precondition of `(remove backplane)`, and `(screw s1)`, to achieve a precondition of `(start_fridge f1)`. (The actual terms are different, as explained in Section 6.2.)

The algorithm as presented so far will have no reason to favor `(unscrew s2)` and `(unscrew s3)` over `(screw s1)`. In the simplified version, the prefix

$$\langle \texttt{(unscrew s1)}, \texttt{(screw s1)} \rangle$$

will be rejected quickly because it repeats the previous situation. But there is no way to reject prefixes such as

$$\langle \texttt{(unscrew s1)}, \texttt{(unscrew s2)}, \texttt{(screw s1)} \rangle$$

which will become an option after `(unscrew s2)` is added to the prefix. If there are N screws, then there are 2^N situations to be explored in which various combinations of screws are unscrewed. Unpop will explore a large subset of them. A person can see that if you've just unscrewed some of the screws, then you should keep unscrewing them. Unless there is a good reason, you shouldn't switch from one goal to another.

One way to diagnose this problem is to recall that coherent subgraphs of the regression-match graph correspond to sketches of plans for achieving subgoals, and to observe that in the absence of any advice to the contrary, the planner should pursue sequences of actions drawn from one coherent

subgraph rather than jumping back and forth between subgraphs. Put another way, the planner should pretend it is actually executing a plan (even though it is really only projecting possible plans).

As I described in Section 3.4, a search state is a sequence of actions

$$\langle A_1, \ldots, A_n, A_{n+1} \rangle$$

where A_{n+1} is allowed by the regression-match graph for the problem goal with respect to

$$result(\mathcal{I}, \langle A_1, \ldots, A_n \rangle)$$

Because A_{n+1} is *allowed,* there is a coherent subgraph such that it occurs in that subgraph with only true goals as preconditions. Let's look a little more closely at the information extractable from that coherent subgraph. For clarity, forget A_{n+1}'s position in the plan prefix, and relabel it as C_0, an arbitrary step allowed by the regression-match graph, and focus on a minimal-effort coherent subgraph $\mathcal{C}$ of which it is an initial step. (As it happens, Figure 4 shows exactly one coherent subgraph.) In Figure 4, I have labeled `unscrew(s2)` as C_0. Define the following partial order on action occurrences in that subgraph:

Definition 1 If action occurrences C_1 and C_2 occur in $\mathcal{C}$, then C_1 is $before_{\mathcal{C},C_0}$ C_2 if and only if either $C_1 \neq C_2$ and

- C_1 is a descendant of C_2; or
- There is an action occurrence C' in $\mathcal{C}$ such that C_0 and C_1 are descendants of C' and C_2 is not. (C' may be identical to C_0 or C_1.)

Note that C_1 and C_2 are action occurrences, and so may be distinct even if they refer to the same action.

Now we can define the "incoherence" of an action:

Definition 2 If $\mathcal{C}$ is a coherent subgraph, and C_0 and C_1 occur in it, then the *incoherence* of action occurrence C_1 with respect to C_0 in $\mathcal{C}$ is the number of action occurrences C such that

$$C_0 \ before_{\mathcal{C},C_0} \ C \ before_{\mathcal{C},C_0} \ C_1$$

If A_0 is an action allowed by a regression-match graph, then the *incoherence* of action A *after* A_0 is then defined as follows. Let

$$G(A) = \{\langle \mathcal{C}, C_0 \rangle | \mathcal{C} \textit{is a coherent subgraph in which } C_0 \textit{ is an initial step and an occurrence of } A_0\}$$

Then the *incoherence* of A with respect to A_0 is

$$\min_{\langle \mathcal{C}, C_0 \rangle \in G(A)} \ \min_{\text{occurrences } C_1 \text{ of } A} \textit{incoherence of } C_1 \text{ with respect to } C_0 \textit{ in } \mathcal{C}$$

In Figure 4, the incoherence of each step with respect to C_0 is written next to the step inside a diamond. In this case, no two steps have the same incoherence, but if C_0 had had two sibling steps (say, `unscrew(s3)` and `unscrew(s4)`), then they both would have had incoherence 0, and the parent, `remove(backplane)`, would have had incoherence 2.

The incoherence of an action is essentially its distance from an initial step of a coherent subgraph. The plan embodied in a coherent subgraph can be thought of as: "Do the initial step, then all the steps of incoherence zero (either feasible siblings or, if there aren't any, the immediate ancestor of the initial step), then all the steps of the next level of incoherence, and so forth." If after taking the initial step there is a choice among followup actions, it's better to take an action with incoherence 0 than an action with incoherence 1, if one is available.

Hence we can use incoherence to differentiate between two plan-prefix extensions that have the same estimated effort. If one has lower incoherence, we favor it. More formally, we can amend the definition of the heuristic estimator:

If the state is $\langle A_1, \ldots, A_{n-1}, A_n \rangle$, then its score is an ordered pair $\langle n-1+E, H \rangle$, where E is the estimated effort of A_n in the regression-match graph for the problem goal in the situation $result(\mathcal{I}, \langle A_1, \ldots, A_{n-1} \rangle)$; and H is the incoherence of A_n after A_{n-1} in the regression-match graph for the problem goal in the situation $result(\mathcal{I}, \langle A_1, \ldots, A_{n-2} \rangle)$. (If $n < 2$ then $H = 0$.) Two scores $S_1 = \langle E_1, H_1 \rangle$ and $S_2 = \langle E_2, H_2 \rangle$ are compared lexicographically. That is, $S_1 < S_2$ if and only if $E_1 < E_2$ or $E_1 = E_2$ and $H_1 < H_2$.

The graph of Figure 4 is constructed for the plan prefix ⟨`(unscrew s1)`⟩. In that graph, the incoherence of `(unscrew s3)` after `(unscrew s2)` is 0, while the incoherence of `(screw s1)` after `(unscrew s2)` is 3. Hence the score of

⟨`(unscrew s1)`, `(unscrew s2)`, `(unscrew s3)`⟩

is $\langle 2 + E, 0 \rangle$, where E is the estimated effort of `(unscrew s3)` in the graph obtained after adding the action `(unscrew s2)` to the prefix[6]; while the score of

⟨`(unscrew s1)`, `(unscrew s2)`, `(screw s1)`⟩

is $\langle 2 + E, 3 \rangle$, which is inferior.

Computing incoherence exactly is expensive. As described in Section 5, in practice we need only to approximate it.

3.7 Preservation Preconditions

In Section 3.1, I pointed out that the regression of P through A could be analyzed in terms of causation and preservation preconditions. So far I have described an algorithm that deals only in causation conditions. In the framework of this algorithm, there is no problem with such "secondary" preconditions. In building the regression-match graph, we don't care whether the goal conjunctions come from primary or secondary preconditions.

Things are different when we turn to thinking about preservation. Consider the "briefcase problem," due to Ed Pednault, whose formalization appears in Tables 4 and 5.

The algorithm as presented so far will build a regression-match graph in which `(at paycheck1 home)` has effort 0 (being true already), and the only way to achieve `(at briefcase2 office)` is to `move` it there. Unfortunately, in the resulting situation, the goal `(at paycheck1 home)` is false, and the only way to make it true is to `move` the briefcase home again. The algorithm now detects a loop in the space of situations, and halts, reporting failure.

The correct solution is, of course,
⟨`(take_out paycheck1 briefcase2), (move briefcase2 home office)`⟩. The problem is to find it. The only reason to perform the `take_out` action is to make the goal `(at paycheck1 home)` true, but at the time it is executed the goal is already true, so the motivation for performing it is obscure.

The solution is to wait until the one-step plan

⟨`(move briefcase2 home office)`⟩

has been proposed, then realize that one way to achieve the (now false) goal `(at paycheck1 home)` is to take the paycheck out of the briefcase *before* the last step. More generally, if A_n is the last step of a plan prefix $\langle A_1, A_2, \ldots, A_{n-1}, A_n \rangle$, then one possible step in plan space to achieve a goal G that is true after A_{n-1} is to move to the plan $\langle A_1, A_2, \ldots, A_{n-1}, B \rangle$, where B achieves $[A_n]^{\underline{R}}(G)$, the preservation precondition of G before A_n. Note that we must *discard* A_n, because it may not be feasible after B. Presumably, if it is feasible and relevant at that point, the algorithm will propose it again on the next iteration.

This still isn't quite general enough, because B may be relevant to achieving $[A_n]^{\underline{R}}(G)$, but not feasible. But addressing this issue is the whole purpose of the regression-match graph. All we need to do is build a piece of the graph for $[A_n]^{\underline{R}}(G)$, but build it relative to the situation that

[6] $E = 6$ if you take the graph seriously, which you shouldn't.

```
(define (domain briefcase-world)
  (:requirements :strips :equality :typing :conditional-effects)

  (:types place)
  (:constants B P D)
  (:predicates (at ?thing - object
                   ?l - place)
               (in ?thing - object))

  (:action mov-b
      :parameters (?m ?l - place)
      :precondition (and (at B ?m) (not (= ?m ?l)))
      :effect (and (at b ?l) (not (at B ?m))
                   (when (in P)
                     (and (at P ?l) (not (at P ?m))))
                   (when (in D)
                     (and (at D ?l) (not (at D ?m))))))

  (:action take-out
      :parameters (?x)
      :precondition (in ?x)
      :effect (not (in ?x)))

  (:action put-in
      :parameters (?x - object ?l - place)
      :precondition (and (at ?x ?l) (at B ?l) (not (= ?x B)))
      :effect (in ?x)))
```

Table 4: The Briefcase Domain

```
(define (problem get-paid)
  (:domain briefcase-world)
  (:objects home office bank - place)
  (:init (at B home) (at P home) (at D home) (in P))
  (:goal (and (at B home) (at D office) (at P bank))))
```

Table 5: PDDL Definition of Briefcase Problem

obtains after A_{n-1}. This extension requires that different pieces of the graph be relative to different situations. In other words, we must extend goal literals, goal conjunctions, and cohorts so that they are pairs of the form $\langle goal, situation \rangle$, where the *situation* is normally the current situation (i.e., that obtaining after execution of the current plan prefix), but will become a previous situation whenever a preservation goal is generated.

To make this idea work, we must provide a method for computing preservation preconditions from action definitions. The key is to notice that each definition provides a specification (a necessary and sufficient condition) for when it *deletes* a proposition. Hence to ensure that the action *fails* to delete a proposition, we must simply find the negation of that specification, and express it as a disjunction of conjunctions. Each such conjunction is a sufficient condition for the preservation of the given goal.

In most cases, this kind of complexity is not necessary. In the given example, the only way that `(move briefcase2 office)` can delete `(at paycheck1 home)` is if `(in paycheck1 briefcase2)` is true. Hence ¬`(in paycheck1 briefcase2)` is the desired preservation precondition, and the algorithm generates the goal literal

$$\langle \neg\texttt{(in paycheck1 briefcase2)}, \mathcal{I} \rangle$$

(an ordered pair in the new scheme). This goal literal is achieved by the action `(take_out paycheck1 briefcase2)`, given a causation precondition that yields the goal conjunction

$$\langle \texttt{(in paycheck1 briefcase)}, \mathcal{I} \rangle$$

which has estimated effort zero. Next the algorithm considers the plan prefix

$$\langle \texttt{(take\_out paycheck1 briefcase2)} \rangle$$

and then solves the problem by rediscovering the `move` action.

There is one remaining complexity. Suppose that the paycheck and briefcase had started off at the office instead of at home; in other words, that the initial situation had had

$$\texttt{(at briefcase2 office)} \wedge \texttt{(at paycheck1 office)}$$

true. Now the algorithm would have first generated the plan prefix

$$\langle \texttt{ (move briefcase2 home)} \rangle$$

to get the paycheck home, and then

$$\langle \texttt{(move briefcase2 home), (move briefcase2 office)} \rangle$$

to get the briefcase back to the office. But it will refuse to consider this prefix any further, because it appears to return the entire world to its initial situation without accomplishing anything. The preservation machinery doesn't get a chance to try to insert a new action before the last one.

To avoid this problem, the system does not discard situation-repeating plan prefixes immediately, but allows only search-space moves that discard the last step of such a prefix. In the experiments reported in Section 6, however, this feature was turned off. In those experiments, preservation preconditions never occurred, and it speeded things up to avoid considering plan prefixes that repeated previously encountered situations.

4 Limitations

Using regression-match graphs does not solve all classical-planning problems. There are a variety of reasons.

First, the method is incomplete, because sometimes solving a problem requires actions that are not allowed by the regression-match graph. The idea of bindings variables using maximal matches is only a heuristic, and it is not hard to contrive examples where the correct bindings are missed. I expected some such examples to occur in practice, but they never did. In all the experiments described in Section 6, as far as I know this heuristic never failed.

Of course, when no variables are involved and actions do not have context-dependent effects, as in the artificial domains of Section 6.5, the regression-graph technique *is* complete. This is so obvious that I won't glorify it with a formal statement and proof.

The algorithms I have described have two classical limitations: they don't worry about finding optimal plans, and they are not very good at inferring that a problem has no solutions. The former issue is one that is often dealt with ambivalently in the literature. Officially plan optimality is not part of the classical-planning problem, but unofficially some of the research (e.g., [32, 21]) has been driven by attempts to avoid redundant plan steps. Unpop shares this ambivalence. On one hand, its heuristic estimator explicitly takes plan length into account — the value of a plan prefix is its length plus the estimated effort of achieving the goal by extending it. On the other hand, this estimator is

often systematically inaccurate. And in hill-climbing-with-restart mode it sticks with a plan prefix as long as it can, which often means fixing plan blemishes by adding steps instead of backtracking to a point where the blemish doesn't happen.

The other issue, proving that a plan doesn't exist, is more fundamental. Unpop, like many search systems, is oriented entirely towards finding a solution. If no solution exists, the only way it can tell that is by exploring its space exhaustively. In practice, however, reporting that a problem can't be found in a certain amount of time may be almost as useful as reporting that it can't be solved at all.

On some problems Unpop takes a very long time, even though the space searched doesn't seem that large. Closer investigation showed that the system was doing a lot of unifications while searching for maximal matches. As with other deductive systems, the order in which conjuncts appear in an action definition can have a big impact on how efficiently the conjunction can be handled. It would be nice if a preoprocessor could find and correct such problems, but for now some tuning is necessary. In general, however, very little rewriting is required. For the experiments reported here, *no* rewriting is required; some of the domains from other researchers are expressed in a very unnatural way for Unpop, but I avoided any revisions. For example, in the Fridge domain, the actions to remove and attach the backplane of a refrigerator are written to include all the screws involved: `(remove-backplane b1 f1 s3 s4 s1 s2)`, for instance. This is because it was originally produced for a planner that couldn't handle context-dependent effects. That means that there are 24 (= 4!) different equivalent actions, one for each permutation of the screws. It is a real nuisance for Unpop to think about all these, but it does.

Finally, the regression-match graph often provides an inaccurate estimate of the effort required to achieve a goal. The major reason for this limitation is that the mechanism doesn't count desctructive effects of plan steps. It doesn't distinguish between steps that achieve one goal and delete another, and steps that achieve one goal while preserving another. Because so much of the research in the planning literature is concerned with this kind of negative interaction, it may be a mystery how Unpop works at all. The answer is that in many cases it is acceptable to handle deletion by waiting for it to be projected. After the planner has added an action that deletes a goal, the newly constructed regression-match graph will contain a nonzero estimate for the cost of reachieving it. In best-first mode, the result is often a polyonimal amount of backtracking. For example, in the blocks world, if the system chooses a destination for a block that covers up a block that is supposed to be clear, then on the next iteration it will backtrack and try a different place. There is only a bounded list of places to try. However, in some problems this technique is not satisfactory. Without the incoherence fetaure, Unpop thrashes badly on problems like the "Fridge" problem in which a long sequence of goals must be deleted that are true now and must be true in the end. In hill-climbing-with-restart mode the poor handling of deletion means that the planner's estimate of the actual work remaining can steer it in the wrong direction for a long time.

Another source of inaccuracy in effort estimates is the failure to count multiple occurrences of subgraphs properly. Consider a domain in which a robot is able to carry two objects, and it has to a long distance to reach them. The estimated effort will count the steps of the trip twice. To alleviate this problem, I experimented with a version of the algorithm that returned a multiset of actions instead of a single number. This multiset was defined as follows, letting $AS(p)$ be the "action set" for a node of the regression-match graph:

- If p is a goal literal, and p is true in S, then $AS(p) = \{p\}$;
- If p is a goal literal, and p is not true in S, then

  $$AS(p) = p + smallest_{g \in reductions(p)} AS(g)$$

  where the "+" sign means to add an occurrence of p to the multiset.
- If g is a goal conjunction, then

  $$AS(g) = |smallest_{h \in cohorts(g)} \cup_{p \in h} AS(p)|$$

where $\cup$ refers to multiset union: the number of occurrences of x in $S_1 \cup S_2$ is the max of the number of occurrences in S_1 and the number of occurrences in S_2.

- If the previous rules do not assign an action set, then assign a multiset in which every action occurs infinitely often.

(Compare the definition of EE in Section 3.3.) Now instead of estimated effort we can take the size of the multiset that is associated with the top node. The hope was that if a subgraph occurred more than once its actions would be counted just once in the action set. In the example I just gave, the sequence of motion actions would give rise to a multiset of "moves," which would be counted just once.

This scheme didn't work as well as I hoped. It tends to underestimate the effort about as often as the numerical scheme overestimates. More subtly, it has little effect on the *differential* powers of effort estimates. That is, if the numerical scheme rates two successor plans about the same, so will the multiset scheme. This lack of discrimination is especially important in hill climbing.

5 Implementation

Unpop is implemented using the Nisp macro package [23] on top of Harlequin Common Lisp. Because Nisp is a macro package, it affects only the compile times for programs, and not execution times (except to the extent that its presence alters the behavior of the garbage collector).

Problems and domains are presented to Unpop using the PDDL formalism [25]. For each domain there is stored:

1. Lists of all the object types and constant symbols associated with the theory.
2. For each predicate and action functor, the type constraints on its arguments.
3. An index of all the action definitions in the theory. Each action definition specifies the preconditions and effects of an action.
4. An index of all the deductive rules associated with the theory. These are used for backward chaining, Prolog-style.
5. A list of facts that are true in every situation. (These involve only constant symbols, and cannot be changed by actions.)
6. Indexes for maintaining unique copies of propositions and actions. This enables them to be EQ-tested and EQ-hashed when necessary. I use the term *occasion* to refer to a "uniquified" proposition of this kind.
7. A list of domain variables, such as the dimensions of the grid.
8. An action-difference table, which for each action stores all its regressions.
9. An "achievability table," which specifies for each predicate whether it can be altered by any action.

PDDL supplies a simple inheritance mechanism for domain theories, so that one theory can be defined to be equal to another theory, with the addition of further predicates, rules, actions, or whatever.

The *action-difference table* corresponds to the *operator-difference table* of GPS [26]. It is built up incrementally. Every time Unpop computes the regression of an action, it caches it in the table, so the next lookup will be much faster. This table does not depend on the particular problem being solved, so it is saved from run to run.

The "indexes" in the list, and the action-difference table, are implemented as discrimination trees on symbolic expressions [8, 28]. Each node discriminates on a particular position in the expression (`CAR`, `CADR`, etc.) and partitions the expressions it is storing into buckets depending on whatever

content it finds at that position. When a bucket gets to be too large, it is further discriminated. These indexes are used to fetch rules, propositions, or action definitions that unify with a particular goal. They are efficient enough that in practice 95% of the unifications the system tries succeed.

To define a problem, you must tell Unpop three things: the domain, the initial situation, and goal. An initial situation defines a *situation space,* defined as the set of all situations that can be reached by taking sequences of legal actions starting in the initial situation. A situation space is represented by a data structure that specifies, among other things, a *situation index* that enables Unpop to tell very quickly whether a newly generated situation has been encountered previously. (See Figure 3.) This index is also implemented as a discrimination tree, whose nonleaf nodes are labeled with occasions. If the label is occasion C, then the node has two subnodes, one containing all the situations stored in the node in which C is true, the other containing all those in which C is false. The contents of nodes are further discriminated whenever they contain five or more situations, provided there exists a C that is not true in all the situations or false in all of them.

In an earlier version of Unpop, instead of this situation-index mechanism, situation spaces were represented as lists of situations, which were linearly searched. For certain sorts of abstract problems (such as the artificial domains mentioned in Section 6.5), thousands of situations can be generated, and Unpop would spend most of its time deciding if a newly computed situation had been seen before. The situation index reduces this time drastically.

In Figure 3, the dotted arcs are drawn based on the assumption that a situation is first seen when the planner explores one of the shortest paths to it. In fact, especially in hill-climbing-with-restart mode, the planner may well encounter a situation first when it is pursuing the longer path. In this case it does not discard the shorter path when it is found, but continues to work on it.

The heart of Unpop is the module for computing regression-match graphs, which calls two main subroutines: one to compute regressions and one to compute maximal matches. I'll describe the graph manager itself first.

The implemented system differs from what I've described here in various ways. Mathematically it's convenient to talk in terms of goal conjunctions, cohorts, and goal literals. In practice, it's more convenient to keep track of two things:

1. The *goal literals*
2. For each goal literal, the set of all its *reductions.*

The reduction of a goal literal g is a pair $\langle A, l \rangle$, where A is an action and l is a list of goal literals. The reduction indicates that there is an action definition for action term $A_v = a(?v_1, \ldots, ?v_n)$ and a substitution μ such that $[A_v]^{\overline{R}}(g) = c$ and μ is a maximal match of c with the current situation, and $A = \mu(A_v)$ and $l = \mu(c)$. That is, having computed the maximal match, we combine it with the action definition and discard it.

The regression-match graph is built breadth-first. I experimented with depth-first schemes, but in some domains they visit the same goal literal repeatedly at different depths. In domains in which a depth cutoff is necessary, it matters what depth a node is encountered at, so it's best to encounter it first at its least depth. A breadth-first scheme makes that more likely.

After the graph has been built, a second pass through it computes effective efforts and feasible actions. A *feasible action specification* consists of the following:

1. A (uniquified) action term;
2. A plan prefix. (This is always just the current plan prefix, unless the preservation machinery of Section 3.7 is in use, when it might be some prefix of the current prefix.)
3. The effective effort of the action (the minimal effort of any coherent subgraph of which it is an initial step; see Section 3.3).
4. The *subsequent-steps table* of the feasible action, which is a table giving, for each action that occurs in the tree, an estimate of its incoherence after this feasible action (see Section 3.6).

A few more words about incoherence are in order. Recall that the incoherence of action A_1 after action A_0 is the least number of actions between A_0 and A_1 in any coherent subgraph of which A_0 is an initial step. Unfortunately, there is no way to compute this number without enumerating all the coherent subgraphs, which would be very expensive. Instead, the Unpop system computes the following approximation: the least number of "incoherence layers" between A_0 and A_1 in any coherent subgraph, where an "incoherence layer" is a set of actions that all have the same incoherence. That is, it assigns 0 to all feasible steps in pursuit of immediate siblings of the purpose of A_0, or, if there aren't any, A_0's immediate successor action. It then assigns 1 to all feasible steps that are successors of siblings. These numbers can be computed simply by computing siblings and successors at each level, as the estimated efforts are being propagated up the regression-match graph. In practice, the main role of incoherence measures is to distinguish steps with incoherence 0 from steps with incoherence > 0, and this approximation agrees with the exact definition in what it assigns the value 0 to. In fact, above some threshold (the `MAX-INCOHERENCE*` parameter), the system does not distinguish among different incoherence values.

I experimented with best-first and hill-climbing search schemes. For most tests I settled on a hybrid scheme that runs best-first until the search tree has gotten to be too bushy, then switches to hill climbing on the most promising branch, restarting randomly as describe in Section 3.4. For many problems such a hybrid approach does almost as well as the better of the other two schemes would do. The criterion of "bushiness" is the "obesity" of the queue of search states, defined as the number of search states that occur at approximately the same depth, and with approximately the same score, as the current search state. When this number exceeds `FAT-THRESH*`, usually set to 10, the system switches to hill-climbing mode.

The "inner loop" of the whole Unpop system is the algorithm for computing maximal matches. This operation is in itself NP-hard in the worst case, although in practice that doesn't seem to matter much. The maximal-match algorithm works as follows. Given a goal of the form $g1_1 \wedge g_2 \wedge \ldots \wedge g_k$, it recurses into two subcases: finding matches for which g_1 is in the hit set P_{true}, and matches for which it is in P_{false}. (See Section 3.2.) The former are obtained by finding all insances of g_1 that are true in the current situation, and the associated substitutions. For each such substitution θ, the match finder is called recursively with the goals $\theta(g_2) \wedge \ldots \wedge \theta(g_k)$.

The other recursive branch is somewhat more interesting. The match finder is looking for any match θ of $g_2 \wedge \ldots \wedge g_k$ such that $\theta(g_1)$ has *no* true instances in the current situation. If g_1 is variable-free, the this branch can be pruned if g_1 has true instances; if g_1 has no true instances, then it is added to a list of differences. Otherwise, when it has variables and true instances, it must be passed down the recursion as an element of an "avoid" list, a list of goals that must not be satisfiable when all variables are bound. Every time a substitution is found, the avoid list must be checked. If an element becomes variable-free and true, the branch is pruned; if variable-free and false, the instantiated goal is added to the difference list. Otherwise, it is retained on the "avoid" list.

The recursion ends when the matcher runs out of conjuncts. At that point it has a list of known differences (ground literals that are false in the current situation), and the "avoid" list, literals with variables that must have no true instances in the current situation. If any element of the "avoid" list has true instances, the branch is pruned. Otherwise, the only remaining task is to instantiate these variables. The matcher does this by finding the type constraints for all the predicates mentioned in the difference list and avoid list, and finding all true instances of them. In the example described at the beginning of Section 3.2, the system requires maximal matches of

`(carrying robot k1)` $\wedge$ `(at robot 2 -3)` $\wedge$ `(at ?k2 2 -3)`

Suppose that the first two conjuncts are true, but there is no key at $\langle 2, -3 \rangle$. Then the only branch of the maximal-match search that is not pruned bottoms out with difference list

{`(carrying robot k1)`, `(at robot 2 -3)`}

and "avoid" list

```
{(at ?k2 2 -3)}
```

The type constraint on the `at` predicate is

```
(at ?a - (either agent key) ?i ?j - coord)
```

So the set of maximal matches is the set $\{\texttt{k} = x\}$ for all x such that[7]

$$(\texttt{agent}\ x) \vee (\texttt{key}\ x)$$

There is one last optimization in the maximal-match finder. Before a literal is added to the "avoid" list, it is checked to verify that it is achievable. A literal is *achievable* if there is any rule that mentions its predicate in an effect. Once a predicate has been checked by searching through the rules, its achievability is cached so it can be checked quicker the next time. The maximal matcher prunes any branch of its recursion that requires putting an unachievable formula in the difference list. That's how the absurd matches described at the end of Section 3.3 are eliminated.

6 Results

Unpop has been run on a wide variety of planning problems, including the Blocks World, the Grid World described in Section 1, a corpus of problems from the University of Washington, the "Mystery" domains from the AIPS competition, the "Rocket" problem of [5], and the artificial domains of [3]. On some of these it exhibits exponential behavior, but on many of them its behavior is polynomial, especially when you measure the number of plans (search states) tried. In this section I will summarize these results. The data reported below were obtained by running on a 300 MHz Dell Pentium-II workstation with 128MBytes of primary memory, using Windows NT as the operating system and Harlequin Common Lisp as the programming language. The running times below include garbage-collection times, because I don't think it makes sense to talk about Lisp run times without including garbage collection.[8] If someone is interested in how long this algorithm will take to solve a problem they care about, they need to know how long it will actually take, not how long it would take in an ideal world where garbage collection was free. It is no doubt true that when and if this style of algorithm becomes useful for practical applications, it will have to be rewritten in a clumsier but more efficient language such as C++. However, there are many factors that would make it run faster if so translated; there is no reason to single out garbage collection.

All the experiments were run using the hybrid search scheme outlined above. The maximum number of plans considered varied from problem to problem, and, as explained in Section 3.5, the maximum plan length was always one-half of the maximum number of plans. Like any other refinement planner, Unpop must consider at least $N + 1$ partial plans when it succeeds in finding a solution of length N. ($N + 1$ because the empty plan it starts with is counted.) In the tables below, the "Search" column gives the number of plans generated that are not on the path to the solution. Therefore, in the case where it finds a solution, the "Search" number is the total number examined $-(N + 1)$. When it doesn't find a solution, the "Search" is the total number examined, which is usually equal to the bound it was given; however, in some cases it runs out of plans to try much sooner because the regression-match graph can detect that the problem is unsolvable.

In all the experiments below the system was run with the same parameter settings, with one exception described in Section 6.5. The settings are:

`CONSIDER-PRESERVATION*` *Value:* **false**

> This turns off the preservation-precondition machinery described in Section 3.7. In many domains it just slows things down by causing the system to artificially consider undoing its last action in order to look for ways to achieve a preservation precondition before it.

[7] Actually, if you check Section 3.2, you'll see that the context makes the first disjunct of this goal absurd.

[8] Because of a memory leak, either in my code or in the Harlequin system, the Lisp had to be restarted periodically to avoid thrashing. It was never allowed to grow to more than 80Mb.

`COUNT-INCOHERENCE*` *Value:* **true**

This turns on the incoherence machinery of Section 3.6.

`MAX-INCOHERENCE*` *Value:* 3

Plan-prefix extensions with incoherence 3 or greater are counted the same.

`MAX-MERGED-OUT*` *Value:* 5 *except in Section 6.5*, where it had value 20.

`MAX-HYB-BRANCHING*` *Value:* 20

In some domains the planner finds a large number (> 20) of feasible actions in a state. Such a huge branching factor makes one false step deadly. To avoid such a fate, we limit the number of feasible actions generated as preconditions of a single action to be $\leq$ `MAX-MERGED-OUT*`, and limit the number used to generate successor states to be $\leq$ `MAX-HYB-BRANCHING*`. If the limits are exceeded, the actions are sorted using their estimated efforts and incoherences, and only the best are retained.

`SCRAMBLE-SUCCESSORS*` *Value:* **true**

The list of favored actions is scrambled before being sorted and added to the search queue. This prevents the planner from being harmed by or profiting from a lucky coincidence in the ordering of actions tried.

`HC-BACKTRACK-DEPTH-FIRST*` *Value:* **false**

In hill-climbing mode, when a state has no successors and this flag is **true**, the planner backs up to the chronologically most recent untried locally best branch. If it's **false**, it does a random restart from an arbitrary ancestor node, again picking some locally best branch. The planner tends to do better on the average when this flag is **true**, but (a) I don't know why, and (b) when it does worse, it can do a lot worse, because it must explore a bad part of the space exhaustively. So I made it **false**.

`FAT-THRESH*` *Value:* 9

As explained in Section 5, once the search-state queue's "obesity' exceeds this threshold, the system switches from best-first search to hill climbing.

`MAX-HYB-QUEUE-LENGTH*` *Value:* 100

In the hybrid search algorithm, there is no point in letting the search-state queue get to be long, because if it's long it's probably obese, and once the program gets into hill-climbing mode it rarely examines more than a small fraction of the states on the queue. Once the queue has more than `MAX-HYB-QUEUE-LENGTH*` states, the once past that horizon are discarded.

All of the problems discussed in this section are accessible from my web page:

`http://www.cs.yale.edu/users/mcdermott.html`

6.1 The Grid World

A typical problem in this world is as specified by Figure 5. The shapes are keys. The squares with holes in them represent locks. Initially, all intersections with locks are locked. The robot can open a locked intersection $\langle i, j \rangle$ by standing next to it with a key of the same shape as the lock, and executing the action `open(`i`,`j`)`. The goal is to get the diamond-shaped key `DK` to location $\langle 3, 0 \rangle$. To do that requires unlocking intersection $\langle 3, 1 \rangle$, which is locked with a circular key. It does no good to try to use a triangular key, because the only one is trapped inside a ring of triangular-locked intersections. Hence the robot must use `dk` to open $\langle -4, 0 \rangle$, $\langle -3, 1 \rangle$, or $\langle -3, -1 \rangle$, carry the circular key `ck` to where the intersection $\langle 3, 1 \rangle$ can be unlocked, then go back and retrieve `dk`. (The robot can carry only one key at a time.) The optimal plan has 43 steps. Unpop did not find it in the five runs reported here, but did find a 48-step plan:

```
((move right) (move up) (move right)
 (pick_up dk) (move left) (move left) (move left)
```

Problem	*Optimal*	*Unpop's behavior*		
		Length	*Search*	*Time*
Grid World	43	48	30	214
		48	30	205
		48	30	194
		48	30	196
		48	30	197

Times are in seconds

Table 6: Grid World Results

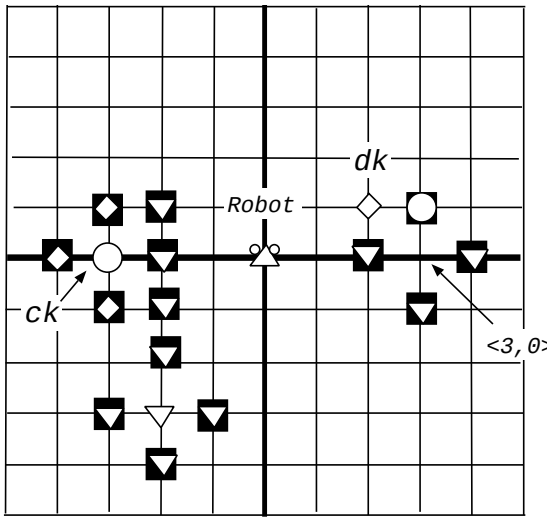


Figure 5: The Grid World

```
(move up) (move left) (move left)
(unlock -3 1) (move down) (move down)
(pick_up ck) (move up) (move up)
(move right) (move right) (move right) (move right) (move right) (move right)
(unlock 3 1) (move down) (move down)
(put_down ck) (move up)
(move left) (move left) (move left) (move left)
(move up) (move left) (move left)
(move down) (move down)
(pick_up dk) (move up) (move up)
(move right) (move right) (move right) (move right) (move right) (move right)
(move down) (move down)
(put_down dk))
```

The results of 5 runs are shown in Table 6. In each case the maximum number of plans explored was set at 200, and the maximum plan length to 100.

Unpop switches from best-first to hill-climbing mode after finding about a 25-step plan (in the usual case). This plan is optimal "so far" (meaning that it could have been extended to an optimal plan), but the search had gotten to be too bushy, so Unpop gave up on exploring all the possibilities. This class of problems is a good example of the global inaccuracies that can occur in estimated efforts. The robot must go back and forth several times to achieve its goals; the regression-match graph contains several pointers to the substructure for these trips; but there is no way to tell how many occurrences of the substructure will actually need to occur in the final plan. It depends on factors like what the robot needs to carry when it moves. For instance, when the robot has just executed `(pick_up ck)`, it must unlock the lock at location $\langle 3, 1\rangle$, then move `dk` to $\langle 3, 0\rangle$. Both goals require a trip from the second quadrant to the first quadrant; the second goal requires the robot to be holding `dk`. This second goal appears simple, because the robot is still at the same location as `dk`. However, it can't pick `dk` up without dropping `ck`. As it moves back to the right carrying `ck`, the goal to get `ck` to $\langle 3, 1\rangle$ looks easier and easier, while the goal to pick up `dk` looks harder and harder. Consequently, the estimated effort stays constant. After $\langle 3, 1\rangle$ has been unlocked, the robot must go back to get `dk`. Now the goal starts to look more and more difficult, as the planner is forced to realize step by step that it's going to have to retrace the steps it's taking. If the planner stayed in best-first-search mode, it would have to prove that all the alternative side trips, themselves overoptimistically assessed, would do no better if pursued. In hill-climbing mode, the system just plods ahead looking at the locally best successor plan, and solves the problem with almost no search, albeit with a few suboptimalities.

Problems like those in the grid world are difficult for traditional planners. A system like Ucpop [37] or Prodigy [36] has trouble because it represents only a single goal structure in its partial-plan representation. Because it has no way of knowing from which direction to approach an intersection, when it is looking for a path of length n it has to consider $O(4^n)$ alternative goal reductions before getting to goals that can be satisfied by feasible actions.

6.2 The University of Washington Corpus

The University of Washington Planning Group has a corpus of planning problems that have been attempted or solve by their planner (Ucpop) and planners from other institutions. It is accessible from the URL

http://www.cs.washington.edu/research/projects/ai/www/ucpop.html

These may be compared with the versions at my web site to see exactly what changes were made in order to get Unpop to run on them. In each case, the problem specification was edited as little as possible. The major change was to introduce argument types for predicates and actions. In most cases, this was straightforward and enhanced the clarity of the specification. Without these changes, the maximal-match finder would have produced way too many matches, as explained in Section 3.2. On some of the domains minor bugs had to be fixed to make the problems solvable. In some, such severe bugs were found that it was impossible to figure out the intent. Timing results for sample problems in most of the remaining domains are given in Table 7, in increasing order of difficulty. Some of these problems are so easy that it was not necessary for Unpop to be run multiple times; for others there was more variety. A detailed comparison with UCPOP is not possible, because the corpus does not include systematic performance data; my impression is that UCPOP cannot solve the Strips, Fridge, and Flat-tire problems in a reasonable amount of time.

The "Ferry" and "Robot" problems are toys. The "Molgen" domain is inspired by the work of [31]. But the problem in the corpus ("rat-insulin") is probably not representative of problems solved by the real Molgen planner. The "Monkey" problem("monkey-test3" in the corpus) involves a monkey, some boxes, some bananas, and so forth. The goal is to get some bananas and a glass of water.

Problem	*Optimal*	*Unpop's behavior*			
		Bound	*Length*	*Search*	*Time*
Ferry	7	20	7	2	0.2
Robot	7	30	7	2	0.2
Molgen	10	80	10	13	1.6
Monkey	12	80	13	5	3.5
Flat tire	19	300	∞	301	14.6
			35	213	10.0
			27	205	8.1
			29	66	2.6
			21	41	1.7
Blocks	8	100	10	19	10.3
			15	1	5.5
			26	20	19.1
			15	12	12.8
			10	1	4.8
Prodigy	13	100	20	19	3.7
			30	21	4.2
			20	10	2.9
			20	5	2.4
			18	5	1.6
Fridge	13	80	13	37	19.8
			13	13	6.8
			13	26	9.4
			13	11	5.6
			13	1	3.1
Strips	14	100	17	22	54.6
			19	29	64.2
			18	19	46.6
			17	42	79.3
			17	14	38.6

Times are in seconds

Table 7: Results for University of Washington Corpus

The "Flat Tire" problem ("fixit" in the corpus) was originally due to Stuart Russell. To fix a flat tire, it is necessary to take tools out of a car's trunk (or "boot," as the problem so quaintly says), use them in various straightforward ways, then put them away (along with the bad tire) and close the boot. Unpop quickly switches from best-first search to hill climbing with restarts. Every time it happens to choose to close the boot, it can't reopen it again without repeating a state, so it does a random restart. It usually finds a solution eventually, but it can take a long time; in one of the five runs it failed (indicated by a value of ∞ for solution length). Note that it had to be given a plan-number bound of 300 to avoid giving up prematurely.

The blocks world contains a single action (`puton` *block destination origin*), with the usual axiomatization, involving a `clear` predicate and a restriction that every block have at most one block on top of it (except for a big block called "`table`"). The problem in Table 7 is the most difficult for Unpop: Starting from a situation in which five blocks are stacked up in increasing order ("`B5` on `B4` on `B3` on `B2` on `B1` on the table"), achieve the following goal:

`(on B3 B2)` $\wedge$ `(on B4 B3)` $\wedge$ `(on B5 B4)` $\wedge$ `(on B1 B5)`

which requires undoing every goal but the last before reachieving all of them, with no hint about where `B2` must be in the end. This problem is not in the corpus, but is harder than any that appears there.

The "Prodigy" problem is another version of the blocks world, closer to Nilsson's [27] specification, in which there are separate actions of the form `unstack(`*block, off-block*`)`, `stack(`*block, on-block*`)`, `pick_up(`*block*`)`, etc. The one for which results are reported is "prodigy-p22" in the corpus.

I discussed the "Fridge problem" (corpus label: "fixb") in Section 3.6, in connection with the incoherence mechanism. That mechanism sharply reduces the search required to solve the problem, although the system still has a high probability of choosing the wrong action when it must reattach the backplane before putting the screws back. The problem is that it has a 0.8 chance of choosing to put a screw back. In hill-climbing mode, once it has gone down this path it has to explore it to a dead end before giving up and trying another path.

The "Strips" problem ("move-boxes-1" in the corpus) involves a robot pushing boxes from room to room, as the Shakey robot [12] did. Solving the problem does not require a lot of search, at least not in limited-discrepancy mode, but it does take a lot of time. Obviously, the reason is that the maximal matcher is doing a lot of work. The number of unifications done for the Strips problem is as high as 648,000, of which more than 90% are successful. This is about 9000 per search state. For comparison, in the Monkey problem there are about 1100 unifications per search state.

6.3 The Mystery World

The "Mystery World" was designed for the AIPS-98 Planning Competition. It is actually three domains, called Mystery, Mystery-Prime, and Mystery-Two. In each the world consists of a planar graph of locations, each having zero or more cargo items, zero or more vehicles, and some amount of fuel. The possible actions are to load a cargo item onto a vehicle at the same location, to move a vehicle from a location to an adjacent one, and to unload a cargo item from a vehicle. An item can't be loaded unless there is room for it on the vehicle, and a vehicle can't move unless there is fuel for it at the location. One trip takes one unit of fuel. Mystery-prime differs from Mystery in that any location with at least two units of fuel can "leak" a unit to any other location. Mystery-two is the same as Mystery-prime except that fuel can leak only between adjacent locations. See Table 8 and 9.

Note the use of the `change` construct of PDDL to express changes in numerical quantities (and the `:functors` field to declare the *fluent functors* `fuel` and `space`). These constructs allow for a succinct statement of what the `leak` action accomplishes, and allows the regression system to determine, for instance, that in Mystery-two you can make `(> (fuel n22) 0)` by finding a node `?n1` such that

`(conn ?n1 n22)` $\wedge$ `(> (fuel ?n1) 1)` $\wedge$ `(> (+ (fuel n22) 1) 0)`

In the AIPS competition, the true nature of the domain was concealed by calling locations "foods," vehicles "pleasures," cargo items "pains," and so forth. Numbers were simulated with a clumsy system of constants and special relations. (None of the contestants could handle the `change` construct.) Since Unpop *can* handle numbers, there seemed no point in expressing the domain in an unnatural way.

Tables 10 and 11 give the results for Mystery and Mystery-prime. (The results for Mystery-two were not qualitatively different from those from Mystery-prime.) There were 35 problems in each domain, and they were the same for each domain. The planner was given a length bound of 30 for each problem (no problem required a plan more than 16 steps long), and a search-space size of 60. After searching 60 partial plans with no solution, the search was aborted. In at least one case the planner could have found a plan of length 33 if allowed to continue (although the optimal plan was of length 12, so this isn't so brilliant). Note that for unsolvable problems X-7 (size 264), Y-3 (size 373), and X-18 (size 383), Unpop did no search at all, but inferred from the regression graph constructed at the first step that no action would get it anywhere.

```
(define (domain mystery)
   (:requirements :typing :existential-preconditions :conditional-effects
                  :expression-evaluation :fluents)
   (:types physob node - object
           vehicle cargo - physob)
   (:predicates
       (conn ?n1 ?n2 - node)
       (loc ?v - physob ?n - node)
       (aboard ?c - cargo ?v - vehicle))
   (:functors
       (fuel ?n - node)
       (space ?v - vehicle)
       - (fluent integer))
   (:action load
       :parameters (?c - cargo ?v - vehicle)
       :vars (?n - node)
       :precondition (and (loc ?c ?n)
                          (loc ?v ?n)
                          (fluent-test (> (space ?v) 0)))
       :effect (and (not (loc ?c ?n))
                    (aboard ?c ?v)
                    (change (space ?v) (- (space ?v) 1))))
   (:action move
       :parameters (?v - vehicle ?n1 ?n2 - node)
       :precondition (and (loc ?v ?n1)
                          (conn ?n1 ?n2)
                          (fluent-test (> (fuel ?n1) 0)))
       :effect (and (not (loc ?v ?n1))
                    (loc ?v ?n2)
                    (change (fuel ?n1) (- (fuel ?n1) 1))))
   (:action unload
       :parameters (?c - cargo ?v - vehicle)
       :vars (?n - node)
       :precondition (and (aboard ?c ?v)
                          (loc ?v ?n))
       :effect (and (not (aboard ?c ?v))
                    (loc ?c ?n)
                    (change (space ?v) (+ (space ?v) 1)))))
```

Table 8: "Mystery" Domain

Any plan that is legal in Mystery is legal in Mystery-two, and any plan legal in Mystery-two is legal in Mystery-prime, so it's reasonable to assume that if a planner can solve a problem in the Mystery domain, it can also solve it in the other two. However, the search space changes, and there are cases where Unpop (and the AIPS contestants) fail to solve a Mystery-prime problem even though they solved it for the simpler Mystery domain.

The actual problems are too big to display here. See the web site referred to earlier for a complete list. In tables 10 and 11 I have arranged the problems in order of increasing size (as measured by the number of symbols in their PDDL definitions). Obviously, this is only a rough measure of how

```
(define (domain mystery-prime)
   (:extends mystery)
   (:action leak
      :parameters (?n1 ?n2 - node)
      :precondition (fluent-test (> (fuel ?n1) 1))
      :effect (and (change (fuel ?n1) (- (fuel ?n1) 1))
                   (change (fuel ?n2) (+ (fuel ?n2) 1)))))

(define (domain mystery-two)
   (:extends mystery)
   (:action leak
      :parameters (?n1 ?n2 - node)
      :precondition (and (conn ?n1 ?n2)
                         (fluent-test (> (fuel ?n1) 1)))
      :effect (and (change (fuel ?n1) (- (fuel ?n1) 1))
                   (change (fuel ?n2) (+ (fuel ?n2) 1)))))
```

Table 9: "Mystery-prime" and "Mystery-two" Domains

difficult the problems were.

Each line of the table gives numbers for a single problem. "Best AIPS" gives the length of the shortest plan found by any contestant at the AIPS competition, and the time required to find that plan. If the length is ∞, no plan was found (which may or may not mean that no plan exists). If the length is "—," then the problem was not attempted during the competition. (The "Y" problems were from Round 2 of the competition, and so were treated as being Mystery-prime problems only.) No time is given for the cases where no AIPS contestants could find a solution to a problem, because not all programs specified a time for problems they couldn't solve, and also because some programs just gave up after a while (as Unpop does) while others were able to *prove* there was no solution, thus making comparisons between them meaningless.

Obviously, the times Unpop takes on these problems is usually larger than the time taken by the contestants. However, most of the AIPS contestants are C programs, whereas Unpop is written in Lisp. When Unpop can solve a problem, it usually does very little search, as shown by the fact that the number of partial plans considered is only a bit larger than the length of the plan found. Insisting that it try 61 plans before declaring failure is probably too conservative; if the threshold had been set at 30 the results would not have changed, but the run times for the case where it failed to find a plan would have been half as long.

For very large problems Unpop's directedness pays off. Of the 16 largest problems in the Mystery-prime domain, Unpop was able to find solutions for 13, whereas all the AIPS contestants together were able to solve 3. I don't know if the three problems Unpop couldn't solve have solutions. Unpop took up to 5 minutes to solve the largest of these problems, and conceivably the AIPS contestants could have solved them with that amount of time. However, my impression is that they all ran out of memory space due to their insistence on advance instantiation of all terms.

Figures 6 and 7 plot the time per "plan-symbol" for these two domains. This number is just the ratio of running time by the product of problem size (number of symbols) times number of plans considered[9] If the number is constant, then all the time Unpop spends can be accounted for by an increase in problem size and number of plans considered. Although there is considerable noise (because number of symbols is only an approximate measure of how hard each problem is), the ratio appears to grow linearly for both of these domains. Because the number of plans considered grows

[9] Remember that, when a solution is found, this number is "Search" + solution length + 1.

Problem	*Size*	*Best AIPS*		*Unpop*		
		Length	*Time*	*Length*	*Search*	*Time*
X-25	92	4	0.10	4	0	0.4
X-1	107	5	0.04	5	0	0.3
X-28	118	7	0.06	9	3	1.4
Y-5	130	—	—	∞	61	12.8
X-27	158	5	4.3	9	2	3.8
X-11	161	7	0.4	11	2	1.4
X-12	162	∞	—	∞	61	20.1
X-29	175	4	0.11	4	1	0.9
X-9	182	8	0.16	8	2	3.3
X-4	188	∞	—	∞	61	8.2
X-5	207	∞	—	∞	61	31.6
Y-2	208	—	—	8	5	3.7
X-3	228	4	0.20	4	0	2.1
X-2	228	9	0.41	10	1	5.0
X-26	236	6	1.78	6	2	6.0
X-16	240	∞	—	∞	3	3.6
Y-1	255	—	—	4	0	5.3
Y-4	258	—	—	4	0	2.2
X-7	264	∞	—	∞	1	0.3
X-30	265	12	5.64	14	1	20.8
X-8	326	∞	—	∞	61	104.2
X-19	327	8	0.87	6	1	11.8
X-21	352	∞	—	∞	61	47.9
X-20	356	10	56.5	7	1	22.5
X-15	369	∞	—	6	1	17.3
Y-3	373	—	—	∞	1	0.8
X-17	376	4	17.8	5	1	13.1
X-23	377	∞	—	∞	61	101.9
X-18	383	∞	—	∞	1	2.3
X-6	384	∞	—	∞	61	177.4
X-24	385	∞	—	∞	61	33.5
X-10	485	8	9.1	∞	61	123.1
X-22	515	∞	—	∞	61	302.4
X-13	521	16	1.79	16	44	370.1
X-14	548	∞	—	18	23	162.1

Table 10: Results for "Mystery" Domain

almost linearly with solution length, the conclusion is that, for solvable problems, running time is growing proportionally to the cube of problem size.

6.4 The Rocket Problem

This problem, drawn from [5], is very hard for Unpop. You are given two rockets and N cargo objects, all in London. Any amount of cargo can be loaded onto a rocket, but the rocket can be flown only once. Some subset of the objects must go to New York and some subset to Paris. Although considerable variation in the order of steps is possible, there is essentially only one solution to each such problem: Load the objects destined for Paris into one rocket; load those destined for New York

Problem	*Size*	*Best AIPS*		*Unpop*		
		Length	*Time*	*Length*	*Search*	*Time*
X-25	92	4	0.1	4	1	0.5
X-1	107	5	3.7	5	0	0.4
X-28	118	7	79.7	11	1	1.6
Y-5	130	6	0.5	8	1	4.0
X-27	158	8	3.2	7	2	2.8
X-11	161	8	1.8	11	0	2.9
X-12	162	9	4.5	12	1	8.0
X-29	175	5	2.3	4	1	1.5
X-9	182	8	1.9	8	1	13.5
X-4	188	9	0.8	9	1	3.9
X-5	207	11	8.1	17	2	19.2
Y-2	208	7	2.5	9	2	7.0
X-3	228	4	0.9	4	0	5.9
X-2	228	9	6.5	10	1	7.5
X-26	236	7	13.3	14	0	16.4
X-16	240	11	5.2	13	1	25.2
Y-1	255	4	7.3	4	1	10.1
Y-4	258	4	8.4	4	2	5.6
X-7	264	5	1.6	5	0	4.0
X-30	265	∞	—	12	2	17.7
X-8	326	7	2.8	10	2	52.5
X-19	327	∞	—	6	2	24.7
X-21	352	7	1.1	11	2	22.1
X-20	356	∞	—	17	2	62.8
X-15	369	∞	—	6	0	14.6
Y-3	373	∞	—	13	0	18.8
X-17	376	4	7.1	5	0	12.3
X-23	377	∞	—	18	0	55.0
X-18	383	∞	—	∞	61	27.4
X-6	384	∞	—	∞	61	313.9
X-24	385	∞	—	15	2	24.8
X-10	485	∞	—	19	1	79.0
X-22	515	∞	—	16	1	135.7
X-13	521	∞	—	15	2	89.3
X-14	548	∞	—	∞	61	289.2

Table 11: Results for "Mystery-prime" Domain

into the other; fly each rocket; unload in Paris; unload in New York. If N objects are to be taken, the optimal solution takes $2N + 2$ steps.

Regression-match graphs are little help with this problem, because you get almost zero information about progress from a single step. The problem is that getting each cargo object to its destination is trivial. The hard part is realizing that exactly one rocket can go to London and one to Paris. Unpop's performance is summarized in Table 12. Unpop eventually stumbles on the right plan, but it must search a huge number of plans (on the average about 4^N) that are essentially the same except for step ordering. The planner was given a bound of 100 plans to try, so it can't solve a problem of size greater than 3.

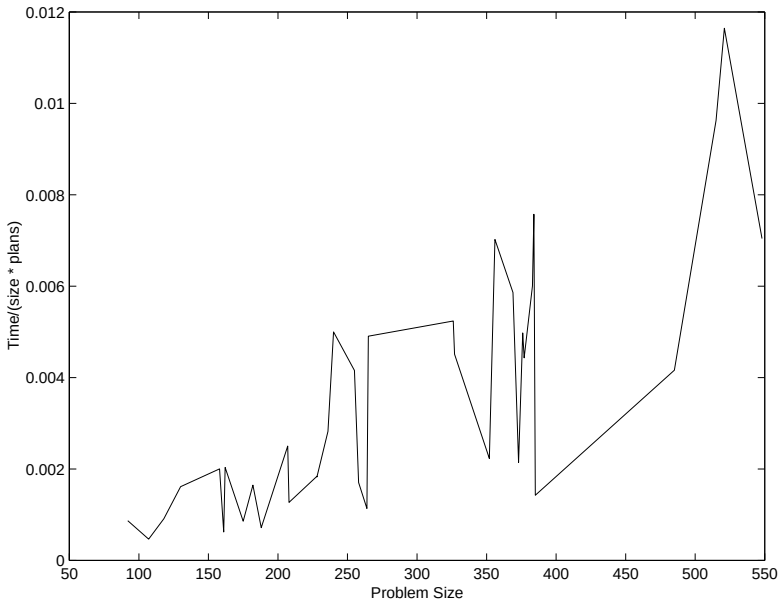


Figure 6: Mystery-domain Ratio of $\frac{\text{time}}{\text{size} \times plans}$

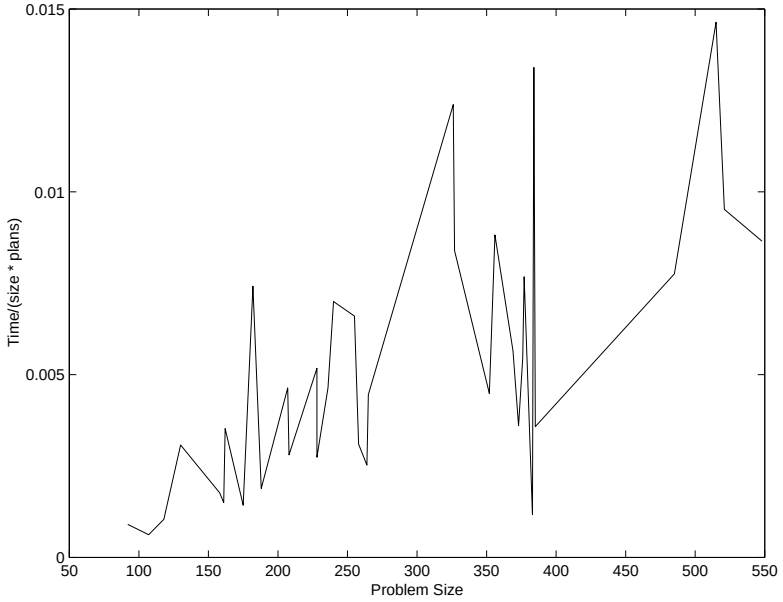


Figure 7: Mystery-prime Ratio of $\frac{\text{time}}{\text{size} \times plans}$

6.5 Artificial Domains

I tested Unpop on most of the problem domains described in [3]. None of these domains used variables in any nontrivial way, so there was always exactly one maximal match for every goal conjunction. However, they do involve lots of action interactions, so the order of steps is important. In [3], the total-order planners usually do very poorly on these problems, becoming exponential for all but the

Size	*Optimal*	*Unpop's Behavior*		
(No. of cargo items)		*Length*	*Search*	*Time*
1	3	3.4	0.8	0.3
2	6	6.0	4.4	1.1
3	8	8.0	22.2	4.0
4	10	∞	101.0	15.2

All numbers averaged over 5 runs. Times are in seconds
For problem of size N, optimal solution is size $2N+2$; Unpop finds it.

Table 12: Results for Blum and Furst's "Rocket Problem"

simplest classes. In addition, with `MAX-MERGED-OUT*` set to 5, as for all the more "realistic" problems, Unpop was unable to solve many of the artificial problems, because these problems have unusually high branching factors. The larger problems typically had 15 unrelated goals, whose first steps were by design equally attractive. Consequently, I set the parameter to 20 for these experiments. In the interests of space, I'll comment on only two of the domains, D^1S^1, in which its behavior is acceptable, and D^mS^2*, for which it is hopelessly exponential.

In D^1S^1, there are 15 actions, `A1` through `A15`, and 30 propositions, `I1`, ..., `I15`, `G1`, ..., `G15`. All the `I`k are true initially. Each `A`k requires `I`k to be true, adds `G`k and deletes `I`$(k-1)$. A problem of size n consists of a random selection of the `G`'s. Within any contiguous sequence of `G`'s that happens to be included, the corresponding `A`'s to achieve them must be in numerical order. This sounds like it might be difficult for a total-order planner (as it was for the ones Barrett and Weld tried), but in actuality Unpop can take but one false step before having to backtrack. That is, if a plan prefix cannot be extended to a complete plan, Unpop will realize it immediately after producing it. The performance is graphed as a function of problem size in Figure 8. For this problem, the bound on number of plans searched was set to 500, and the bound on plan length was set to 250. Oddly enough, Unpop actually does worst for problems of medium size in this problem class. I have not succeeded in explaining this phenomenon, although it is very repeatable. For comparison the dotted line shows the behavior of a partial-order planner on the same problem, from [3]. My total-order planner does not succeed in being linear, but it's not exponential either.

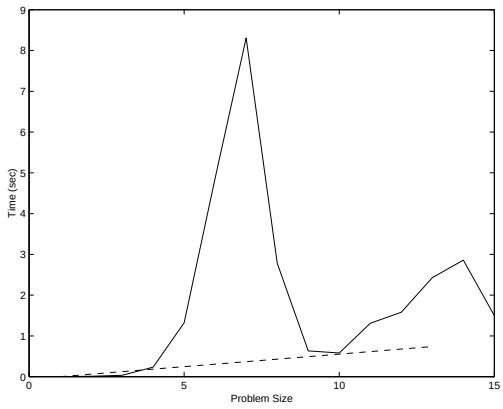
In the domain D^mS^2*, there are 13 actions, two of the form `A`k1 and `A`k2 for $k = 1, \ldots, 6$, and a special action `A*`. There are 7 goals, with `G`k achieved by `A`k2, and `G*` achieved only by `A*`. The action `A`k2 requires precondition `P`k. This precondition is achieved only by `A`k1, which also requires `I`k as a precondition. Unfortunately, each `A`k deletes `P`j for $j < k$. The action `A*` deletes every goal except `G*`, plus all of the `I`k conditions. Initially all the `I`k's are true. A problem consists of a random sample of `G`k's, plus `G*`. A solution is a sequence of `A`k1's in decreasing k order, followed by `A*`, followed by all of the `A`k2's in increasing k order. For example, to achieve `G4`, `G5`, `G6`, and `G*`, you must execute

$$\langle \texttt{A61, A51, A41, A*, A42, A52, A62} \rangle$$

Unpop's behavior is shown in Figure 9. It is quite exponential, just like the planners in [3]. It also does not find the optimal plan, but adds in pointless occurrences of `A`k2. The plan it finds for the example problem is

$$\langle \texttt{A61, A62, A51, A52, A41, A*, A42, A52, A62} \rangle$$

The reason is that its coherence heuristic causes it to try to achieve `G`k as soon as possible. Because it's in hill-climbing mode, it never undoes this decision.



Times are in tenths of a second

Figure 8: Graph of Results for D^1S^1

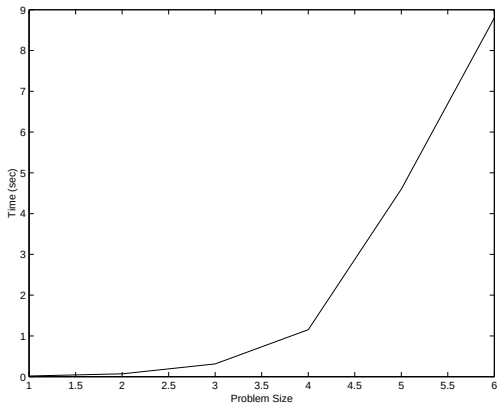
7 Relation to Previous Work

The present work derives from an attempt to simplify the GPS control structure, which, as described by [10], is rather arcane and complex. The idea of cashing out all matching operations to generate a graph structure linking top-level goals to feasible actions first appeared in [7], Section 5.7, where the phrase "operator-difference tree" was used for what I now call the "regression-match graph." However, I did not at the time appreciate the need for clear definition of the match layers of the graph.

The most recent planning work that is related to the Unpop algorithm is the Prodigy planner of Carbonell and Veloso[36, 14], and especially its incarnation as the "FLECS" commitment strategy.[35]. Kambhampati[18] introduced a similar framework in the context of partial-order planning.

What these papers have in common is that they model plans as collections of goals and subgoals. Alternative goals and subgoals are reached only by switching to another part of the search space (i.e., backtracking). More recent algorithms have begun to represent explicit alternatives the way Unpop does. See [17, 19]. However, all of these previous works omit the idea of matching as a way of zeroing in on relevant actions. On the plus side, they do a better job than Unpop in reasoning about destructive interactions among subplans.

The currently most successful (and fashionable) approaches to planning are based on the idea of avoiding variables by reasoning only about fully instantiated action and proposition terms. Graphplan [5] constructs a "planning graph" containing all propositions that could conceivably become



Times are in tenths of a second

Figure 9: Graph of Results for $D^m S^2*$

true as the result of a series of actions. The resulting structure is extremely useful; it would be nice to incorporate some of these ideas into Unpop, and I have some suggestions below. The original Graphplan had trouble with context-dependent effects, but this has been fixed in recent work by Köhler et al. [22] and by Anderson et al. [2]. SATPLAN [21] treats planning as a satisfiability problem. This requires representing all possible propositions and actions at all possible times, which sounds unlikely to work, but can be made to work with ingenious coding tricks. Both of these approaches currently dominate Unpop. The idea of solving a problem with a huge amount of search by an incredibly optimized search engine may or may not win out over doing less search with a slower program. As shown in Section 6, the space greed of these algorithms begin to catch up with them, when a more plodding approach just keeps on going.

Since the original paper on Unpop appeared [24], a similar idea was independently discovered by Bonet, Loerincs, and Geffner [6]. Their version avoids matching by working with fully instantiated propositions from the beginning, much as SATPLAN does. However, it uses the same idea of estimating the effort required to achieve a goal by constructing tree of subgoals all the way to currently true propositions before taking a step. The resulting program performed well in the AIPS-98 Planning Competition.

8 Conclusions and Future Work

Total-order planning is more promising than its critics have implied. Although it doesn't solve everything, it has one big advantage over some of the other approaches to classical planning: it

represents a *current situation* in exact detail, which allows the planner to compare that situation to the goal description, and produce estimates of how much work remains to be done. In some cases, the regression-match graph proposed here produces an excellent estimate; in other cases, it's not so good. When it works, it typically allows a planner to avoid search almost completely, at the cost of a polynomial-size computation at each step through plan space. The moral of the story is that we should be looking for better heuristic estimators to guide the search through plan space.

Of course, no algorithm will work well on all planning problems, because planning is NP-complete [11]. Unpop can be expected to work well whenever the negative effects of inserting a step too early are revealed quickly. In that case, its ability to look far ahead into the the structure of subgoals and actions gives it an excellent estimate of the difficulty of the problem and the potential of the feasible actions. I believe many of the problems in the literature are like this. Unpop doesn't do so well when the the precise order of steps matters and the problems with bad orderings are not revealed until they are almost complete. It is not designed to handle problems with this sort of "combination lock" flavor, such as the Rocket problem.

The currently most attractive alternatives to the approach proposed here are algorithms based on planning graphs [5, 22] and algorithms based on propositional satisfiability [21]. So far those approaches have had to sacrifice expressivity in order to allow their algorithms to work. By contrast, the approach embodied in my planner handles a larger subset of the PDDL language, including some simple numerical reasoning. In principle, it can operate in any domain in which there is a reasonable notion of *regression,* the inference of a weak precondition for a goal.

There are plenty of interesting research directions suggested by this work. Because the hill-climbing strategy often finds suboptimal plans, it might be possible to find a better plan by restarting the planner using the length of the first plan found as a bound on plan length; this idea could be expanded into a branch-and-bound algorithm. The rewriting technique of [1] might also be applicable.

One main direction for future work is to make management of the regression-match graph incremental. Currently the graph is rebuilt before the selection of every planning operator. For domains of interesting size, such as the grid world and mystery world described above, the graphs are fairly large and change only slightly after each action. After some preliminary design, I think it would be cost-effective to represent the graph as a growing data structure with edges labeled by the situations in which they are present. The label system would work in such a way that when a successor situation were created, it would automatically inherit all the edges of its predecessors, unless overriden by the incremental changes due to the effects of the action leading to the situation.

The key element that would make the scheme work is an efficient method for finding all the maximal matches that must be redone as the result of additions and deletions. It turns out to be fairly easy to characterize the set of literals a change in whose status would affect a given maximal match. The regression-match graph must be supplemented with an efficient lookup table for finding those points after every action.

The other main direction is to improve Unpop's blind spot with respect to negative interactions among goals. There are two main cases where its blindness causes it serious problems, both involving two sibling goals G_1 and G_2 from the same conjunction:

- The effective subgraph for G_1 relies on true literals that are going to be deleted by the actions in the subgraph for G_2.
- The actions in the subgraph for G_2 deletes G_1.

It would not be too tricky to detect these two situations during the feasible-action-computation phase described in Section 5. It would take time, but only a polynomial amount (because the program wouldn't do anything like trying all possible orderings of the two sibling subgraphs). However, even a polynomial amount of time may be too much, unless the system gains a significant amount of search control. It is impossible to be sure if the investment is worth it without a detailed study.

Acknowledgements: This work was supported in part by the Advanced Research Projects Agency of the U.S. Department of Defense, administered through ONR contract N00014-93-1-1235.

References

[1] José Luis Ambite and Craig A. Knoblock. Planning by rewriting: efficiently generating high-quality plans. In *Proc. Fourteenth National Conference on Artificial Intelligence*, 1997.

[2] Corin Anderson, Dan Weld, and David Smith. Conditional effects in Graphplan. In *Proc. of the 4th Int. Conf. AI Planning Systems*, 1998.

[3] Anthony Barrett and Daniel S. Weld. Partial-order planning: evaluating possible efficiency gains. *Artificial Intelligence* , 67(1):71–112, 1994.

[4] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proc. Ijcai*, 1995.

[5] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence 1–2*, 90:279–298, 1997.

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

[7] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence*. Addison-Wesley, 1985.

[8] Eugene Charniak, Christopher Riesbeck, Drew McDermott, and James Meehan. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, 1987. second edition.

[9] Ken Currie and Austin Tate. O-plan: the open planning architecture. *Artificial Intelligence* , 52(1):49–86, 1991.

[10] George W. Ernst and Allen Newell. *GPS: A Case Study in Generality and Problem Solving*. Academic Press, 1969.

[11] Kutluhan Erol, Dana Nau, and V.S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. In Drew McDermott and James Hendler, editors, *Artificial Intelligence* **76**, *Special Issue on Planning and Scheduling*, pages 75–88. NIL, 1995.

[12] Richard Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence* , 3(4):349–371, 1972.

[13] Richard Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*, pages 189–208, 1971.

[14] Eugene Fink and Manuela Veloso. Prodigy planning algorithm. Technical Report 94-123, CMU School of Computer Science, 1994.

[15] Alfonso Gerevini and Lenhart K. Schubert. Accelerating partial-order planners: some techniques for effective search control and pruning. *J. of Art. Intell. Res* , 5:95–137, 1996.

[16] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proc. Ijcai95*, pages 607–613, 1995.

[17] David Joslin and Martha Pollack. Passive and active decision postponement in plan generation. In *Proc. 3rd European Workshop on Planning*, 1995.

[18] Subbarao Kambhampati. Universal classical planner: an algorithm for unifying state-space and plan-space planning. In *Proc. AAAI-95*, 1995.

[19] Subbarao Kambhampati. On the role of disjunctive representations and constraint propagation in refinement planning. In *Proc. Conf. on Knowledge Representation and Reasoning*, 1996.

[20] Subbarao Kambhampati, Craig A. Knoblock, and Qiang Yang. Planning as refinement search: a unified framework for evaluating design tradeoffs in partial-order planning. In Drew McDermott and James Hendler, editors, *Artificial Intelligence* **76**, *Special Issue on Planning and Scheduling*, pages 167–238. NIL, 1995.

[21] Henry A. Kautz, David McAllester, and Bart Selman. Encoding Plans in Propositional Logic. In *Proc. KR-96*, 1996.

[22] Jana Koehler, Bernhard Nebel, J. Hoffmann, and Y. Dimopoulos. Extending Planning Graphs to an Adl Subset. In *Proc. European Conference on Planning*, 1997.

[23] Drew McDermott. Revised Nisp Manual. Technical Report 642, Yale Computer Science Department, 1988.

[24] Drew McDermott. A Heuristic Estimator for Means-ends Analysis in Planning. In *Proc. International Conference on AI Planning Systems* **3**, pages 142–149, 1996.

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

[26] Allen Newell and Herbert Simon. Gps: a program that simulates human thought. In *Lernende Automaten*, pages 279–293. R. Oldenbourg KG. Reprinted in Feigenbaum and Feldman 1963, 1961.

[27] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[28] Peter Norvig. *Paradigms of Artificial Intelligence Programming:Case Studies in Lisp*. Morgan Kaufmann, 1992.

[29] Edwin Peter Dawson Pednault. *Toward a Mathematical Theory of Plan Synthesis*. PhD thesis, 1986.

[30] Edwin Peter Dawson Pednault. Adl: Exploring the middle ground between Strips and the situation calculus. In *Proc. Conf. on Knowledge Representation and Reasoning* **1**, pages 324–332, 1989.

[31] Mark Stefik. Planning with constraints. *Artificial Intelligence* , 16(2):111–139, 1980.

[32] Gerald J. Sussman. *A Computer Model of Skill Acquisition*. American Elsevier Publishing Company, 1975.

[33] Reiko Tsuneto, Dana Nau, and James Hendler. Plan-refinement strategies and search-space size. In *Proc. Fourth European Conf. on Planning (ECP-97)*, 1997.

[34] Manueal Veloso and P. Stone. "Flecs: Planning with a Flexible Commitment Strategy. *J. of Art. Intel. Res* , 3:25–52, 1995. ".

[35] Manuela Veloso and James Blythe. Linkability: examining causal link commitments in partial-order planning. In *Proc. AIPS-94*, 1994.

[36] Manuela Veloso and Jaime Carbonell. Derivational analogy in PRODIGY: automating case acquisition, storage, and utilization. *Machine Learning* , 10:249–278, 1993.

[37] Daniel Weld. An introduction to least-commitment planning. *AI Magazine*, 1994.

[38] David Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann Publishers, Inc, 1988.