A Heuristic Estimator for Means-Ends Analysis in Planning

Drew McDermott*
Yale Computer Science Department
51 Prospect Street, P.O. Box 808285
New Haven, CT 06520-8285
e-mail: mcdermott@cs.yale.edu
phone: 203-432-1281 fax: 203-432-0593

Abstract

Means-ends analysis is a seemingly well understood search technique, which can be described, using planning terminology, as: *keep adding actions that are feasible and achieve pieces of the goal.* Unfortunately, it is often the case that no action is both feasible and relevant in this sense. The traditional answer is to make *subgoals* out of the preconditions of relevant but infeasible actions. These subgoals become part of the search state. An alternative, surprisingly good, idea is to recompute the entire subgoal hierarchy after every action. This hierarchy is represented by a *greedy regression-match graph.* The actions near the leaves of this graph are feasible and relevant to a sub...subgoals of the original goal. Furthermore, each subgoal is assigned an estimate of the number of actions required to achieve it. This number can be shown in practice to be a useful heuristic estimator for domains that are otherwise intractable.

Keywords: planning, search, means-ends analysis

Reinventing GPS

Means-ends analysis is one of the oldest ideas in AI. It was named and studied by Newell, Shaw, and Simon in the 1950s, and was the key idea behind the General Problem Solver (GPS) (Newell & Simon 1961; Ernst & Newell 1969) In the late sixties, Fikes, Nilsson, and Raphael embodied the idea in their planner, Strips (Fikes & Nilsson 1971). It is still an important technique today, especially as embodied the Prodigy planner (Fink & Veloso 1994)).

As used by planners, means-ends analysis can be described thus: We are given a set of *action specifications,* an *initial situation,* and a *goal-situation description.* The problem is to find a sequence of actions that, if carried out starting in the initial situation, would get to a situation that satisfies the goal description. Action specifications define the meanings of action terms by specifying preconditions, addlists, and deletelists in the usual way. For example, we might define the action `take_out(?x,?b)` thus:

Action:	`take_out(?x,?b)`
Preconditions:	`in(?x,?b)` $\wedge$ `exposed(?b)`
Effects:	*Del:* `in(?x,?b)`
	Add: `exposed(?x)`

That is, if `?x` is in container `?b`, and `?b` is "exposed" (not inside anything), then the result of executing `take_out(?x,?b)` is that `?x` ceases to be inside `?b`, and becomes exposed.

In a nutshell, the idea behind means-ends analysis is to build action sequences by continually adding actions whose addlists contain conjuncts that correspond to pieces of the goal-situation description. I will use the term *plan prefix* for an action sequence that the planner is trying to extend to be a solution to a planning problem. The search space is the set of all plan prefixes. The search begins with the empty prefix. The goal is a sequence such that executing it gets the world to a situation that satisfies the goal-situation description. If I is the initial situation, then let $r(I,s)$ be the situation resulting from executing action sequence s starting in I. If the search has reached prefix s, then s can be extended to s,A, where A is an action such that: (1) A is feasible (has all preconditions satisfied) in $r(I,s)$; (2) some conjunct in A's addlist occurs in the goal-situation description and is not already true in $r(I,s)$.

The main bug with this idea is, of course, that for almost all interesting problems, we can't hope to satisfy both condition (1) and condition (2) on the action that extends a plan prefix. That is, many actions that would add a goal conjunct are not feasible in $r(I,s)$. The solution that has been adopted since Newell, Shaw, and Simon invented means-ends analysis is to make search states more complicated, by keeping track of a hierarchy of subgoals as well as a plan prefix. In this paper I consider the following alternative: Keep search states simple, just sequences of actions,

*This work by supported by ARPA and administered by ONR under Contract Number N00014-93-1-1235

and try harder to find feasible and relevant actions to add. However, instead of cautiously back-chaining in tiny steps, back-chain all the way to feasible actions at every search state.

For example, suppose we are given the action `take_out` as defined above, plus the initial situation:

`in(b1,b2)` `in(b2, b3)` `exposed(b3)`

plus the goal `exposed(b1)`. The action `take_out(b1,b2)` is relevant (the goal occurs in its addlist), but it isn't feasible. So we introduce a subgoal `exposed(b2)`, which traditionally becomes part of the search state. What I am proposing is that we explore the subgoal immediately, back-chaining until we can verify that there is an action, `take_out(b2,b3)`, which is feasible and leads (by reversing the series of back chains) to the goal. We tack this action onto the sequence, then *discard the subgoal structure,* regenerating it from scratch in the situation resulting from executing `take_out(b2,b3)`. In this situation, we'll discover that `take_out(b1,b2)` is feasible and achieves the original goal.

Greedy Regression-Match Graphs

Let me be more precise about this back-chaining system. We start with the given overall goal G, and we grow a tree of subgoals. On alternate layers of the tree, we match a goal to the current situation to find differences, then back-chain through actions to find subgoals, then match again, and so forth. By "matching" a goal $g_1 \wedge g_2 \wedge \ldots \wedge g_k$ I mean computing a substitution θ such that $\theta(g_i)$ is true in the current situation for as many g_i as possible. More formally, define a *match* of a conjunction of goal literals $G = g_i \wedge \ldots \wedge g_k$ to situation S to be a substitution θ that binds the free variables of G (and no other variables). Define $\mathit{hit\_set}(G, \theta, S)$ of a match θ to be the set $\{g_i : \theta(g_i)$ is true in $S\}$. A *maximal match* is then a match θ such that there is no θ' with $\mathit{hit\_set}(\theta) \subset \mathit{hit\ set}(\theta')$. The *difference set* of a maximal match θ is defined as $\theta(G \setminus \mathit{hit\_set}(G, \theta, S))$, and written $\mathit{difference\_set}(G, \theta, S)$. Note that hit sets are defined as subsets of the original goal conjuncts, but difference sets are defined as subsets of the conjuncts after variable substitution. Difference sets correspond to goals that remain to be achieved, although it may well be that goals not in the difference set, which are satisfied in the current situation, may be made false by some of the actions that achieve goals in the difference set.

Now we can define the structure that specifies what relevant actions are. This is the *greedy regression-match graph*, defined in terms of two kinds of nodes: (1) *c-nodes:* Conjunctions of literals that represent goals, possibly containing variables; and (2) *l-nodes:* Ground literals that represent elements of difference sets. There are edges from l-nodes to c-nodes and from c-nodes to l-nodes. The first kind are labeled with substitutions; an edge from l-node l to c-node c is labeled with θ only if θ is a maximal match such that $l \in \mathit{difference\_set}(c, \theta, S)$. ($S$ is the current situation.) The edge from c-node c to l-node l is labeled with an action term A only if A causes l to become true if c is true in the current situation S, that is, if the *regression of l through A* is c, or, in symbols, $c = [A]^R(l)$.

The greedy regression-match graph is a graph because l-nodes are not duplicated, so that there can be multiple paths, and even cycles, between nodes. The graph is "greedy" because it considers only *maximal* matches. In the interest of brevity, I'll often drop the word "greedy."

The regression-match graph is built starting with the original goal, G, which is matched to the current situation, yielding a maximal match θ with difference set D. Each element of D becomes a new l-node, which is added to the graph, connected to G by an edge labeled with θ. An l-node is then regressed through all action terms of the form $a(v_1, \ldots, v_k)$, where the v_i are new variables, yielding a c-node with some, but not necessarily all, of the variables replaced by l-node terms. The instantiated action term then labels the link from the c-node to the l-node. Then the process repeats. A regression-match graph for our simple example is shown in Figure 1. The l-nodes appear as unboxed literals; c-nodes are in dotted lozenges; actions are in ovals. Each c-node is connected to a group of l-nodes by an edge labeled with a maximal match. In general a c-node will be connected to zero or more such groups; in the figure, there's exactly one per c-node. The l-nodes that are made true by the match are underlined. Each of the others is connected to zero or more c-nodes by edges labeled with a actions; in the figure, there's exactly one action for each such l-node.[1]

In Figure 1, c-nodes are labeled with numbers in boxes. These are *estimated effort* numbers, which are an estimate of how many actions it will take to achieve the main goal. They are computed as follows: The effort of a c-node is the sum of the efforts of the l-nodes in the difference set with least effort, or

$$\mathit{effort}(c) = \min_{\theta \in mm(c,S)} \sum_{l \in \mathit{difference\_set}(c,\theta,S)} \mathit{effort}(l)$$

where $mm(c, S)$ is the set of all maximal matches between c and S. In particular, the effort assigned to a

[1] The direction of the arrows in the graph may seem counterintuitive.The arrows point toward the top c-node rather than away from it in order to make the causal direction explicit. Nonetheless, when I talk of the "children" of a node, I mean the nodes that point *toward* it.

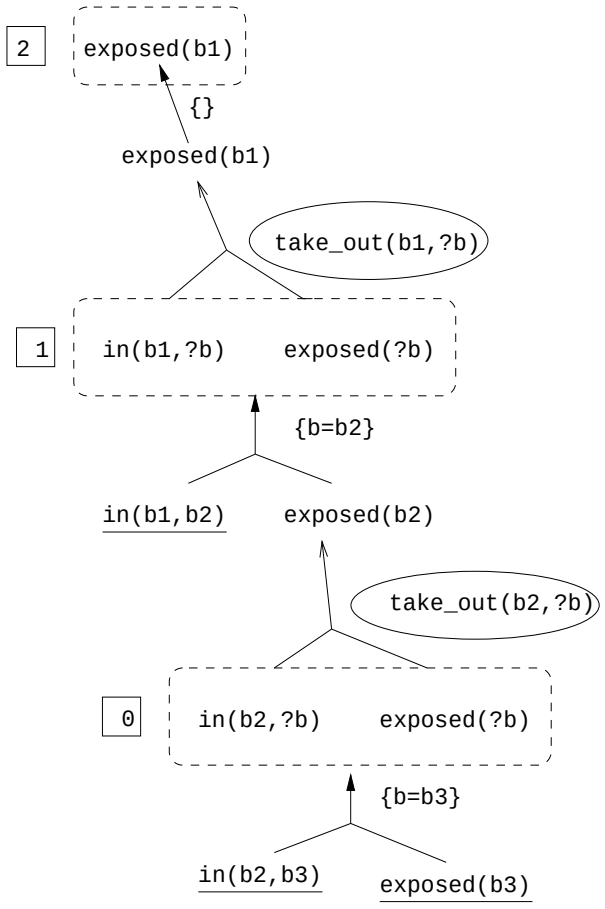

Figure 1: A Simple Greedy-Regression Match Graph

c-node that matches the current situation with difference set ϕ is 0. The effort assigned to an l-node counts the actions required to achieve it, and so is obtained by adding one to the effort of the precondition of the action whose precondition has the least effort. In symbols,

$$\mathit{effort}(l) = 1 + \min_A \mathit{effort}([A]^R(l))$$

Because estimated efforts are defined by minimizing over the children of a c-node or l-node, they depend on only a subset of those children, namely, those whose scores actually are minimal. Call these the *effective* children of that node. Define the *effective subgraph* of the regression-match graph to be the subgraph obtained by selecting just the effective children at each node, but removing all children of a node that are also its ancestors, thus ensuring that the effective subgraph is a DAG. Such cycles will exist only if the estimated effort of the top node is ∞. When the estimated effort is less than ∞, the "leaves" of the effective subgraph will be c-nodes that are true in the current situation. If the estimated effort is ∞, then there will also be at least one "impossible leaf," either an l-node that is achieved by no action, or a c-node all of whose difference sets have at least one l-node that is an ancestor of the c-node.

We can compute the effective subgraph as we build the entire regression-match graph. For each l-node and c-node, as we accumulate their subnodes, we just keep track of the minimal ones. It is always possible to tell locally whether a child is also an ancestor. We also keep track of the *actions favored by* the greedy regression-match graph, defined as those whose preconditions are leaf c-nodes in the effective subgraph (and hence are true in the current situation).

The regression-match graph has two valuable features: the estimated-effort numbers give us a heuristic estimate for the current plan; and the allowed actions are the "feasible and relevant" actions we seek. We exploit these features by embedding them in a search algorithm with the following search space:

Search Space RM:

- *Initial state:* The empty plan prefix $\langle\rangle$.
- *Operators:* Let s be the current plan prefix. Let $S = r(I, s)$ be the current situation. Compute the greedy regression-match graph for the goal G with respect to S. The set of operators is then the set of all action terms in the tree whose precondition c-nodes have estimated effort 0.
- *Heuristic evaluation function:* Score state s as $\mathit{length}(s)+\ \mathit{estimated\_effort}(G)$. The estimated effort is read off from the regression-match graph.

Action:	`unlock(?i1,?j1)`
Preconditions:	`at(robot,?i,?j)`
	`adjacent(?i,?j,?i1,?j1,`
	`?direction)`
	`carrying(robot,?key)`
	`loc_shape(?i1,?j1,?s)`
	`key_shape(?key,?s)`
Effects:	`locked(?i1,?j1)`
	⇒ *Del:* `locked(?i1,?j1)`
	Add: `open(?i1,?j1)`

Table 1: Manhattan World Action Definitions — 1

Action:	`move(?direction)`
Preconditions:	`at(robot,?i,?j)`
	`adjacent(?i,?j,?i1,?j1,`
	`?direction)`
	`open(?i1,?j1)`
Effects:	*Del:* `at(robot,?i,?j)`
	Add: `at(robot,?i1,?j1)`

Table 2: Manhattan World Action Definitions — 2

- *Goal state:* A plan prefix s for which the goal G is true in $r(I, s)$.

We need a more realistic example, and for that we turn to what I call the "Manhattan world," a large grid of intersections a robot can move through. The robot can carry one object at a time. Some of the intersections are locked, and can be opened only with a key of the same shape. If the robot is standing next to a locked intersection $\langle i, j \rangle$with a key of that shape, the action `unlock(`i, j`)` causes it to become unlocked. The robot can move to a neighboring intersection if it is unlocked. (See Tables 1 and 2.) It can pick objects up and put them down (Tables 3 and 4).

The construct $p \Rightarrow e$ indicates a conditional effect; the effect e occurs if p is true just before the action is executed. (This and other aspects of my notation are taken from (Penberthy & Weld 1992).) For example, if the robot is already carrying an object `?k1`, picking up `?key` will cause it to let go of `?k1`.

Figure 2 shows a problem in this domain. The shapes represent keys; the black squares with white shapes inscribed represent locks, which are initially all locked. The shapes must match for a key to open a

Action:	`pick_up(?key)`
Preconditions:	`at(?key,?i,?j)`
	`at(robot,?i,?j)`
Effects:	*Del:* `at(?key,?i,?j)`
	Add: `carrying(robot,?key)`
	`carrying(robot,?k1)`
	⇒ *Del:* `carrying(robot,?k1)`
	Add: `at(?k1,?i,?j)`

Table 3: Manhattan World Action Definitions — 3

Action:	`put_down(?k)`
Preconditions:	`carrying(robot,?k)`
Effects:	*Del:* `carrying(robot,?k)`
	`at(robot,?i,?j)`
	⇒ *Add:* `at(?k,?i,?j)`

Table 4: Manhattan World Action Definitions — 4

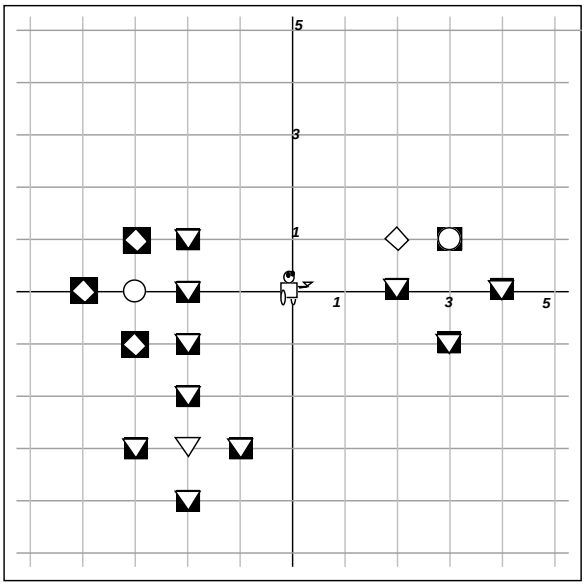

Figure 2: The Manhattan World

lock. The problem is to get `dk`, the diamond-shaped key at $\langle 2, 1 \rangle$, to location $\langle 3, 0 \rangle$. The optimal plan has 43 steps. People find the problem quite simple, but one reason for that is that they condense sequences of motions to single steps. Automated planners can't do that, and they tend to get blown away by the combinatorics. (See Section "Results and Related Work.")

Let's look at how the problem responds to greedy regression-match graphs. The goal is `at(dk,3,0)`. The goal is false in the current situation, so it becomes the only element of the difference set obtained by matching. So we regress it through all possible action terms, `move(?dir)`, `unlock(?i,?j)`, `pick_up(?k)`, and `put_down(?k)`. The first two yield identically false preconditions, but the last two give rise to two nontrivial c-nodes:

1 `put_down(dk)`:
`at(robot,3,0)` ∧ `carrying(robot,dk)`

2 `pick_up(?k)`:
`at(robot,3,0)` ∧ `at(?k,3,0)`
∧ `carrying(robot,dk)`

The first of these has one maximal match:

$\theta = \{\}$, with difference set
{`at(robot,3,0)`, `carrying(robot,dk)`}

The c-node for the second action, `pick_up(?k)` has three maximal matches, corresponding to three different values for `?k`, `dk`, `tk` (the triangular key), or `ck` (the circular key)[2]. Each match has a difference set that includes `at(robot,3,0)` and `carrying(robot,dk)`, plus one other literal, either `at(dk,3,0)` for $\theta = \{\texttt{k} = \texttt{dk}\}$, `at(tk,3,0)` for $\theta = \{\texttt{k} = \texttt{tk}\}$, or `at(ck,3,0)` for $\theta = \{\texttt{k} = \texttt{ck}\}$.

This fragment of the regression-match graph is just the tip of a big iceberg. The whole graph has about 730 l-nodes, and takes about 25 seconds to generate on a Sun Microsystems SparcStation 2. The total estimated effort is 36, which is off by 7 from the optimum. Because we must rebuild this large graph every time we take a step through the search space, it might seem as if this algorithm makes no sense at all. Instead of searching through the space of situations in a forward way, we are back-chaining on each iteration; doesn't that amount to searching backward through exactly the same space? The answer is No: the backward search is not through situations, but through literals. Roughly speaking, if there are n literals, then there are 2^n possible situations. A straight situation-space search for the problem of Figure 2 is, as discussed in Section "Results and Related Work", out of the question.

Search Space RM may be considered a nondeterministic algorithm. To implement it on a computer, we must specify a search strategy, that is, a specification of which operator to apply when more than one is possible. I have experimented with two strategies: best-first search and limited-discrepancy search (Harvey & Ginsberg 1995). The former just keeps a queue of plans that have been generated but not extended, and works on the plan with the least estimated effort. The limited-discrepancy strategy is a modification of depth-first search in which branches that are not locally optimal according to the heuristic estimate are postponed. It is based on the observation that the estimated effort is a better measure of relative merit of search states than absolute merit. For toy domains, where plans are no longer than length 15 or so, the best-first strategy usually results in finding an optimal plan. But for more realistic domains, such as the Manhattan world, the best-first strategy spends an exponential amount of time ruling out all promising alternatives before extending the main line. The limited-discrepancy approach will often find a reasonable plan, but not the optimal one. Results appear in Section "Results and Related Work."

[2]Domain-specific argument constraints for `pick_up` are used to ensure that only keys are considered as bindings for `?k`.

Technicalities, Limitations and Possible Enhancements

In this section I'll fill in some details on how the algorithm works. I call the implemented algorithm "Unpop." Space does not permit a detailed discussion of algorithms for matching, regression, and search. The regression algorithm just works through action definitions in the obvious way. The matching algorithm is a straightforward combinatorial algorithm that considers all possible hit sets. Note that the problem of finding all maximal matches for a formula is itself potentially expensive, because there may be an exponential number of candidate matches to consider. It may take a long time to consider them, and if an exponential number get through that will make the regression-match graph too big. In practice, this has not been a problem. (For discussion of the exponentialities that *are* problematic, see Section "Results and Related Work")

I do need to make one crucial remark about the search space. On top of all the other search strategies I have discussed, my algorithm is doing a situation-space search, and so must cope with the problem of encountering the same situation repeatedly. I have implemented the simplest possible tactic. A table of situations is kept, and whenever a situation is created, the program does a linear search through the table to see if it has been encountered before. If so, it is not explored again, unless the new path to the situation is shorter than the old, in which case the program treats it as a new situation.

The example action specfications in the "box" domain made use of the construct $p \Rightarrow e$, which conditionalizes the effect e based on the *secondary precondition* p. The precondition is called "secondary" (Pednault 1989) because the feasibility of the action being defined does not depend on its being true. The effect e is called *conditional* or *context-dependent.* The greedy regression-match algorithm has no trouble when e is of the form *Add:* q. The secondary precondition becomes part of the c-node created when a goal conjunct matching e is regressed through this action specification. But we must extend the algorithm to handle the case when e is of the form *Del:* q. In that case, p is called a *preservation precondition.* Intuitively, there will be circumstances where p must be achieved before an action A in order to prevent A from deleting e. The classic example is due to Pednault (Pednault 1989): If a briefcase is carried from one place to another, then an object moves to its location if and only if the object is in the briefcase.

It is not obvious how to include preservation preconditions in a system that adds actions only at the end of a plan, because the whole idea is to add actions before a plan step to make sure that step doesn't have some effect. My solution doesn't handle every case, but handles a surprising number of them. I allow preservation goals to arise only before the last step. In essence, I have Unpop treat "preserve across last step in the current plan prefix" as an odd kind of action for achieving an l-node g, applicable only when g was actually true before the last step. Its precondition is just the preservation precondition for g before the last step, and this precondition, expressed in disjunctive normal form, gives rise to c-nodes, one per disjunct. However, for this tactic to work, we have to change c-nodes and l-nodes so that they stipulate where in the action sequence they are to be made true. Furthermore, if an action favored by the regression-match graph is adopted, and that action is to occur in a situation prior to the current one, then all actions after that point are discarded and replaced by the new action. Of course, they are likely to be rediscovered and reappended.

It's traditional in papers like this to prove soundness, completeness, or optimality of a planning algorithm.. But the soundness of my algorithm is so obvious it isn't worth stating formally; the algorithm produces a plan only when it results in a situation in which the goal description is satisfied. Alas, the algorithm is neither complete nor optimal, so there are no theorems there either. On the other hand, the recent history of research in automated planning tends to have a depressing surplus of completeness results and shortage of heuristic estimators. If you really want completeness, you could plug the Unpop heuristic estimator into a complete goal-directed planning framework such as that of (Fink & Veloso 1994).

Results and Related Work

The program has no trouble with the standard "toy" problems in the literature, where solutions are plans with about five or six steps. My main test domain has been the Manhattan world. Before I discuss how well the program works, let me pause to note how poorly all previous general-purpose systems perform on this problem. There are three classes of algorithm to consider: a blind situation-space search, a Strips/Prodigy-style algorithm (Fikes & Nilsson 1971; Fink & Veloso 1994), and a partial-order (or "nonlinear") planner (McAllester & Rosenblitt 1991; Weld 1994).

A straight situation-space search in this domain is out of the question. There are about 100 possible destinations for the robot if it doesn't pick anything up, but as soon as it reaches the diamond key it gets another 100 positions for that key, for a situation space of 10,000. But there are three diamond-shaped locks, each of which can be open or closed, which is 8 more situation classes, for a total of 80,000. Now we can get at the circular key, which multiplies it by 100 again, times two for the circular lock, so we're up to 16,000,000. Adding one more key would multiply it by 100 again, and so forth.

The Strips/Prodigy family of planners is similar to mine, but differ in that they build the subgoal hierarchy incrementally. For Prodigy, the hierarchy is a tree of conjunctive goals; for Strips, it's a subgoal stack, a single branch through that tree. There are two big headaches that these systems share. First, because they can't represent a disjunction in their subgoal trees, there is an exponential blowup in the number of possible trees; for each regression-match graph there is a large number of subgoal trees, obtained by (in essence) transforming the graph into disjunctive normal form, that is, raising all ORs to the top and calling each disjunct a separate search state. Second, there is no heuristic estimator for these subgoal hierarchies. There is no way of knowing that carrying the circular key to $\langle 3, 1\rangle$ (see Figure 2) from the west is better than carrying it from the south. In the regression-match graph, we work all the way to actions that are feasible now, and verify that the estimated effort coming from the west is lower.

Strips and Prodigy come out even worse when the possibility of using the triangular key at $\langle -2, -3\rangle$ arises. They have no way of knowing that getting the triangular key is impossible, without exploring the entire search space. The greedy regression match graph contains an l-node `carrying(robot,tk)`, but its effective subgraph has a cycle, and so has estimated effort ∞.

Partial-order planners do even worse in this domain, if that's possible. They represent attempted solutions as a pure hierarchy of subgoals, with no action sequence and hence no current situation. Hence they are unable to realize that if you have committed to getting to $\langle 3, 0\rangle$ via $\langle 2, 0\rangle$, there is no point in trying to get to $\langle 2, 0\rangle$ via $\langle 3, 0\rangle$. The number of possible partial-order plans in this domain is astronomical.

With these comparisons in mind, let's look at how the regression-match approach compares. For large Manhattan-world problems such as that of Figure 2, I used the limited-discrepancy strategy, for reasons described above. Because it's behavior is randomized, it behaves differently on different runs, but it always takes about 30 minutes, explores around 60 plan pre-

Size	Num	Time	Size	Num	Time
1	2.0	.02	9	20.6	1.95
2	3.2	.07	10	25.2	2.95
3	4.2	.12	11	32.6	4.69
4	5.2	.23	12	37.4	7.06
5	7.2	.38	13	42.8	8.03
6	10.2	.58	14	52.8	10.76
7	10.8	.82	15	66.2	10.37
8	14.2	1.24			

Table 5: Unpop's Behavior on D^1S^1
(Num = number of plan prefixes explored
Run = run time)

fixes (30 seconds per prefix), and finds a plan within 5 of optimal. Unpop has also been run on a suite of problems created by the University of Washington AI Group. Many of these problems involve context-dependent effects. Unpop can solve all of them, and only exhibits exponential behavior on one (which I discuss below).

For a more systematic comparison of Unpop with previous planners, I ran it in best-first mode on some of the artificial problem spaces of (Barrett & Weld 1994). In most of these problem spaces, the number of plan prefixes examined by the planner grew linearly or quadratically with the size of the problem. The run time grew faster, because as problems grow, the size of the regression-match graph grows, too. Even so, on some of the domains, Unpop's behavior was polynomial,although it was never linear. Table 5 shows Unpop's run-time in the domain Barrett and Weld call D^1S^1 as the problem size increases. The actions in this domain are of the form:

Action:	$\mathtt{A}_i$
Preconditions:	$\mathtt{I}_i$
Effects:	*Del:* $\mathtt{I}_{i-1}$
	Add: $\mathtt{G}_i$

A typical problem begins with an initial situation in which all of $\mathtt{I}_i$ are true for $i = 1$ to 13; and the goal is to make $\mathtt{G}_i$ true for some random set of i between 1 and 13. Solutions are sequences that contain an $\mathtt{A}_i$ for every $\mathtt{G}_i$, and such that $\mathtt{A}_j$ never comes after $\mathtt{A}_{j+1}$ for any j. Unpop's heuristic estimator cannot "see" this interaction, but after one false step it becomes quite clear. That is, whenever $\mathtt{A}_{j+1}$ is added to the plan prefix, the goal $\mathtt{A}_j$ becomes impossible. Hence Unpop will try at most $12 - l$ blind alleys on a plan prefix of size l.

The news about Unpop is not all good. It tends to do poorly on problems where a goal literal g that is true in the current situation is sure to be deleted by an action that must be taken, but not right away. Suppose the action becomes inevitable for all plans beginning with plan prefix P, but that there are exponentially many ways to lengthen it before the action gets added. All of these extensions will look more attractive than the ones that add that action, because they will count g as already achieved. One domain that has this property is the "fridge" domain in the University of Washington collection. There are four screws that must be unscrewed in order to service a refrigerator. But the final goal specifies that the screws must all be screwed. Hence Unpop tries very hard to find a way to avoid unscrewing all the screws, and considers all possible ways of unscrewing two or three before finally exploring plan prefixes in which all four are unscrewed. Another example is Barrett and Weld's (Barrett & Weld 1994) domain $D^mS^{2\star}$.

Yet another example is the "Rocket" domain discussed by (Blum & Furst 1995). In this domain a rocket can only be used once, a fact expressed by having the action of flying a rocket delete the precondition `has-fuel(`*rocket*`)`, which is not added by any action. Unpop considers moving cargo to two different destinations by flying the same rocket, and once again will try all possible permutations of cargo and rockets before finally flying the rocket and realizing that the plan prefix just can't be extended to a solution.

I am currently exploring ways to fix this problem with Unpop. It is encouraging that the regression-match graph usually signals in advance that a deleterious action is inevitable, when that action occurs as the only way to achieve some l-node. The main roadblock to exploiting that information is finding a good way to estimate the cost of repairing the damage done by the action. The cost depends on the situation when the action is executed, which differs from the current situation.

Two other planners have used data structures similar to regression-match graphs to look for exactly this kind of information. One is the system of Smith and Peot (Smith & Peot 1993), which used a data structure called an "operator graph" that is essentially a regression-match graph without the matches. The planner of (Blum & Furst 1995) used a data structure called a "planning graph" that is similar in spirit to a regression-match graph, but "looks forward" from the current situation all the way to the end rather than "looking back" from goals all the way to the current situation, as the regression-match graph does. Graphplan is blindingly fast for some classes of problem, including the Rocket problem. However, Graphplan apparently does not handle context-dependent effects, and it's not clear how to extend it to do so.

Conclusions

I have presented a new way of thinking about means-ends analysis, in which an exhaustive subgoal analysis, based on *greedy regression-match graphs,* is repeated at each search state, rather than being spread over a number of search steps. This sounds like a bad idea, but it yields two benefits: a heuristic estimator of the number of actions required to get to a problem solution from the current state; and a set of actions that are good candidates for the next step to take. The resulting situation-space search algorithm searches many fewer states than traditional planners on a large class of problems, although it takes longer than usual per state.

A key lesson is that results such as those of (Barrett & Weld 1994) on the inferiority of total-order planners versus partial-order planners may not apply when the planners are given more accurate heuristic estimators. The fact that a search space is exponential matters less if the searcher can avoid looking at most of it.

References

Barrett, A., and Weld, D. S. 1994. Partial-order planning: evaluating possible efficiency gains. *Artificial Intelligence* 67(1):71–112.

Blum, A. L., and Furst, M. L. 1995. Fast planning through planning graph analysis. In *Proc. Ijcai.*

Ernst, G. W., and Newell, A. 1969. *GPS: A Case Study in Generality and Problem Solving.* Academic Press.

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

Fink, E., and Veloso, M. 1994. Prodigy planning algorithm. Technical Report 94-123, CMU School of Computer Science.

Harvey, W. D., and Ginsberg, M. L. 1995. Limited discrepancy search. In *Proc. Ijcai95*, 607–613.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. AAAI* **9**, 634–639.

Newell, A., and Simon, H. 1961. Gps: a program that simulates human thought. In *Lernende Automaten*, 279–293. R. Oldenbourg KG. Reprinted in Feigenbaum and Feldman 1963.

Pednault, E. P. D. 1989. Adl: Exploring the middle ground between Strips and the situation calculus. In *Proc. Knowledge Representation Conf*, 324–332.

Penberthy, J. S., and Weld, D. S. 1992. Ucpop: A sound, complete, partial order planner for Adl. *KR-92.*

Smith, D. E., and Peot, M. A. 1993. Postponing threats in partial-order planning. In *Proc. AAAI* **11**, 500–506.

Weld, D. 1994. An introduction to least-commitment planning. *AI Magazine.*