On the Role of Ground Actions in Refinement Planning

Håkan L. S. Younes and **Reid G. Simmons**
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
{lorens, reids}@cs.cmu.edu

Abstract

Less than a decade ago, the focus in refinement planning was on partial order planners using lifted actions. Today, the currently most successful refinement planners are all state space planners using ground actions—i.e. actions where all parameters have been substituted by objects. In this paper, we address the role of ground actions in refinement planning, and present empirical results indicating that their role is twofold. First, planning with ground actions represents a bias towards early commitment of parameter bindings. Second, ground actions help enforce joint parameter domain constraints. By implementing these two techniques in a least commitment planner such as UCPOP, together with using an informed heuristic function to guide the search for solutions, we show that we often need to generate far fewer plans than when planning with ground action, while the number of explored plans remains about the same. In some cases a vast reduction can also be achieved in the number of explored plans.

Introduction

The principle of least commitment in planning is intuitively appealing. It states that one should delay commitment of action orderings and variable bindings until such commitments become necessary in order to resolve consistency threats. This strategy leads to a reduced branching factor in the search space (Weld 1994). A least commitment planner must keep track of ordering and binding constraints, and enforce consistency on these, which adds to the complexity of the planning algorithm. The hope, however, is that the reduced branching factor will make up for the added complexity in consistency enforcement. Successful implementations of least commitment planners, such as SNLP (McAllester & Rosenblitt 1991) and UCPOP (Penberthy & Weld 1992), demonstrated the promise of this approach a decade ago, but since then the research focus in refinement planning has shifted to state space planners using ground (fully instantiated) actions.

The abandonment of the least commitment principle can, to a great extent, be credited to the success of Graphplan (Blum & Furst 1995). Graphplan constructs a planning graph, which consists of alternating levels of ground atoms and actions. The planning graph proves to be extremely useful in guiding the search for solutions to planning problems. Many of the currently most successful refinement planners are state space planners using the planning graph, either explicitly (e.g. FF (Hoffmann & Nebel 2001)) or implicitly (e.g. HSP (Haslum & Geffner 2000)), to extract domain independent heuristics that can be used in hill-climbing or A^* search.

The idea of extracting heuristics from the problem representation is not new. McDermott (1996) introduced greedy regression-match graphs as a means for estimating the effort of achieving goals. McDermott's planning system, Unpop, is a state space planner, but works with lifted actions (actions that contain variables). Although the heuristic is similar to that used in HSP, Unpop does not appear to be competitive. Bonet & Geffner (2001) suggest that the main reason for this is that HSP works with ground actions, and thus avoids having to deal with variable bindings and matching operations in the planning phase.

Nguyen & Kambhampati (2001) recently showed that the efficiency of partial order planners can be dramatically improved by using planning graphs to estimate the cost of achieving goals. Their planner, REPOP, is based on UCPOP, but uses ground actions. The choice of working with ground actions appears almost incidental, and there is no discussion on the possible effect this choice may have had on the results. From their comparison with a version of UCPOP using ground actions, it is quite clear that ground actions alone do not help much. As suggested by Bonet & Geffner (2001), however, the use of an informed heuristic may not be enough either. Still, little has been said about the role that ground actions play in refinement planning.

We attempt to shed some light on this issue. To do so, we have implemented a partial order planner, based on UCPOP, that can work with either ground or lifted actions. The planner uses a version of the additive heuristic proposed by Bonet, Loerincs, & Geffner (1997) to estimate the cost of achieving goals, extended to work with partially instantiated goals.

We have identified two important roles that ground actions play. First, the use of ground actions corresponds to an early commitment of parameter bindings in the case when lifted actions are used. Second, by using ground actions we enforce joint constraints on the parameter domains of actions in a plan. Both these roles enable the planner to identify inconsistencies at an earlier stage, thereby reducing the need for backtracking. We show that by enforcing joint parameter domain constraints when planning with lifted actions, and implementing a bias towards early commitment of parameter bindings, we can find solutions exploring about as many nodes in the search space as when using ground actions. The benefit of using lifted actions is that the branching factor is reduced, which results in fewer generated nodes. We show that in some cases the reduction in generated nodes can be significant. More importantly though, we show that in certain cases planning with lifted actions can result in a vast reduction in the number of explored nodes as well.

Partial Order Refinement Planning

A planning problem consists of a set of initial conditions, a set of goals to be achieved, and a set of operators. The initial conditions and the goals contain no variables. An operator is a schematic representation of an action available to the planner, and consists of a set of preconditions that must hold when an instance of the operator is applied, and a set of effects. An instantiated operator that is part of a plan is called an action. If all operator parameters are substituted by objects when instantiating an operator, we get a ground action. If the action has unbound parameters, it is called a lifted action.

In partial order planning, a plan is represented by a set of actions $\mathcal{A}$, a set of causal links $\mathcal{L}$ indicating dependencies among actions, a set of ordering constraints $\mathcal{O}$ defining a partial order of the actions, and a set of binding constraints $\mathcal{B}$ on the action parameters ($\mathcal{B} = \emptyset$ if ground actions are used). A causal link, $a_i \xrightarrow{q} a_j$, represents a commitment by the planner that precondition q of action a_j is to be fulfilled by an effect of action a_i.

A refinement planner works by adding elements to a plan in order to remove *flaws* in the plan. A flaw can be either an *open condition*, $\xrightarrow{q} a_i$, representing a precondition q of an action a_i that has not yet been linked to the effect of another action in the plan, or an *unsafe link*, $a_i \xrightarrow{q} a_j$, whose condition q can be unified with the negation of an effect of an action that could possibly be ordered between a_i and a_j. The set of flaws of a plan π is the union of open conditions and unsafe links: $\mathcal{F}(\pi) = \mathcal{OC}(\pi) \cup \mathcal{UL}(\pi)$.

An open condition, $\xrightarrow{q} a_i$, of a plan $\langle \mathcal{A}, \mathcal{L}, \mathcal{O}, \mathcal{B} \rangle$ can be resolved by linking the effect p of an existing or new action a_k to q. The resulting plan has actions $a_k \cup \mathcal{A}$, causal links $a_k \xrightarrow{q} a_i \cup \mathcal{L}$, and ordering constraints $a_k \prec a_i \cup \mathcal{O}$. In addition, unless ground actions are used, the *most general unifier* (MGU) of p and q is added to the binding constraints $\mathcal{B}$. MGU of two atomic literals p and q is the smallest set of variable substitutions needed to unify p and q.

An unsafe link, $a_i \xrightarrow{q} a_j$, threatened by the effect p of action a_k can be resolved in three ways:

- *Demotion*: Order a_k before a_i.
- *Promotion*: Order a_k after a_j.
- *Separation*: Add variable bindings that make p and $\neg q$ non-unifiable.

Separation is of course not available when ground actions are used.

A planning problem can be represented as a plan π_{init} with $\mathcal{L} = \mathcal{B} = \emptyset$, and with two actions:

- a_0 without preconditions, and effects corresponding to the problem's initial conditions. The action a_0 is always ordered before all other actions in a plan.
- a_∞ with preconditions corresponding to the goals of the problem, and no effects. The action a_∞ is always ordered after all other actions in a plan.

The partial order planner proceeds by nondeterministically choosing a flaw in π_{init} to work on. Given a flaw, there are often several refinements that can be applied to repair the flaw. One of the refinements is chosen (nondeterministically), and is then applied to the current plan to produce a new plan π'.[1] The refinement process repeats with π' until there are no remaining flaws in which case π' is a solution to the planning problem.

In contrast to total order planners, which maintain a total order of the actions in a plan, partial order planners defer decisions on action orderings until such orderings become necessary in order to resolve conflicts. In addition, unless ground actions are used, variable bindings are kept at a minimum by only adding the most general unifier when linking effects to preconditions. This way, the planner adheres to the principle of least commitment. For a thorough introduction to least commitment planning, we refer the reader to (Weld 1994).

Heuristic Search Planning

Although described in terms of nondeterministic choice in the previous section, any actual implementation of a partial order planner uses some kind of search representing the nondeterministic choice of refinements as branching points in the search space. Our implementation uses the A^* algorithm (Hart, Nilsson, & Raphael 1968).

The A^* algorithm requires a search node evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of getting to n from the start node, and $h(n)$ is the estimated remaining cost of reaching a goal node. In our case, a search node is a plan, and we consider the cost of a plan to be the number of actions in it. For a plan $\pi = \langle \mathcal{A}, \mathcal{L}, \mathcal{O}, \mathcal{B} \rangle$, we use $g(\pi) = |\mathcal{A}|$. The heuristic cost $h(\pi)$ of completing the plan π (i.e. resolving all flaws in π) should be an estimate of the number of actions we need to add to π in order to make it complete.

Heuristic Plan Cost

The original implementations of SNLP and UCPOP used $h(\pi) = |\mathcal{F}(\pi)|$ as heuristic cost function—i.e. the number of flaws in a plan. Schubert & Gerevini (1995) consider alternatives for $h(\pi)$, and present empirical data showing that just counting the open conditions ($h(\pi) = |\mathcal{OC}(\pi)|$) often gives better results. A big problem, however, with using the

[1] Note that the choice of which refinement step to take is a backtracking choice point, but not the choice of which flaw to work on (Weld 1994).

number of open conditions as an estimate on the number of actions that needs to be added is that it assumes a uniform cost for open conditions. It ignores the fact that some open conditions can be linked to existing actions (thus requiring no additional actions), while other open conditions can be resolved only by adding a whole chain of actions (thus requiring more than one action).

Recent work in heuristic search planning has resulted in more informed heuristic cost functions. Heuristic costs are often derived from a *relaxed planning graph*. A planning graph consists of alternating layers of ground literals and actions. The topmost layer consists of the ground literals in the initial conditions of a planning problem. The next layer consists of all ground actions that are applicable given the initial conditions. What follows is a second layer of ground literals, consisting of the union of literals from the previous literal layer and the literals produced by the actions in the preceding action layer. The layers keep alternating like this until no more literals are added. The planning graph also records if two actions are mutually exclusive at a specific layer—information that is left out in the relaxed planning graph.

Nguyen & Kambhampati (2001) use a heuristic for their partial order planner REPOP that counts the number of actions needed to resolve the open conditions based on the serial planning graph, without counting actions that are already in the plan. By not counting actions already in the plan, they account for reuse, but their heuristic can easily overestimate the possible reuse of steps since it does not take the current ordering constraints into account. Their heuristic cost estimator is reported to give significantly better results than $h(\pi) = |\mathcal{OC}(\pi)|$ on several planning problems.

Our planner uses the additive heuristic first proposed by Bonet, Loerincs, & Geffner (1997) in the context of planning as real-time search. While not taking reuse of actions other than a_0 into account, the heuristic has worked very well for us. We present here a slight modification of the heuristic, so that it can be used to estimate the cost of negative and disjunctive preconditions, conditional effects, and also the cost of achieving partially instantiated literals as well as ground literals.

The cost of a set of goals, S, is

$$h_{\mathrm{add}}(S) = \sum_{q \in S} h_{\mathrm{add}}(q).$$

The cost of a conjunction of literals is

$$h_{\mathrm{add}}(\bigwedge_i q_i) = \sum_i h_{\mathrm{add}}(q_i),$$

and we define the cost of a disjunction of literals to be

$$h_{\mathrm{add}}(\bigvee_i q_i) = \min_i h_{\mathrm{add}}(q_i).$$

Given a literal q, let $\mathcal{GA}(q)$ be the set of ground actions occurring in the relaxed planning graph having an effect p that unifies with q. The cost of q is

$$h_{\mathrm{add}}(q) = \begin{cases} 0 & \text{if } q \text{ holds initially} \\ \min_{a \in \mathcal{GA}(q)} h_{\mathrm{add}}(a) & \text{if } \mathcal{GA}(q) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$

A positive literal q holds initially if it is part of the initial conditions. A negative literal $\neg q$ holds initially if q is not part of the initial conditions (the closed world assumption). Finally, the cost of an action a is

$$h_{\mathrm{add}}(a) = 1 + h_{\mathrm{add}}(\mathit{Prec}(a)),$$

where $\mathit{Prec}(a)$ is the set of preconditions of action a.

As heuristic cost function we use $h(\pi) = h_{\mathrm{add}}(\mathcal{OC}(\pi))$. The cost of ground literals can be efficiently computed while constructing the relaxed planning graph. We take conditional effects into account in the cost computation. If the effect q is conditioned by p in action a, we add $h_{\mathrm{add}}(p)$ to the cost of achieving q with a. We only need to compute the cost for ground literals once, and the matching of partially instantiated literals to ground literals can be implemented quite efficiently, leaving little overhead for evaluating plans using the proposed heuristic.

Estimating Remaining Effort

Not only do we want to find plans consisting of few actions, but we also want to do so exploring as few plans as possible. Schubert & Gerevini (1995) suggest that the number of open conditions can be useful as an estimate of the number of refinement steps needed to complete a plan. We take this idea a bit further.

When computing the heuristic cost of a literal, we also record the estimated effort of achieving the literal. A literal that is achieved through the initial conditions has estimated effort 1 (corresponding to the work of adding a causal link to the plan). If the cost of a literal comes from an action a, the estimated effort for the literal is the estimated effort for the preconditions of a plus 1 for linking to a. Finally, the estimated effort of a set of literals is the sum of the estimated effort of each individual literal.

The estimated effort is used as a tie-breaker between two plans π and π' in case $f(\pi) = f(\pi')$, which tends to lower the number of plans explored before finding a solution.

Early Commitment of Parameter Bindings

We now turn to the role of ground actions in refinement planning. In this section we discuss how early commitment of parameter bindings can be implemented as a flaw selection strategy, and in the next section we discuss joint parameter domain constraints.

Flaw Selection Strategies

There are two types of flaws: Unsafe causal links (threats) and open conditions. The original versions of both SNLP and UCPOP worked on unsafe links first, trying to eliminate any threats before working on an open condition. If there were no unsafe links, the most recently added open condition was selected (corresponding to a LIFO order). Peot & Smith (1993) suggest several strategies for delaying threat-removal, the most effective strategies being to delay all but the threats for which there is at most one possible repair, or to delay threats that can be resolved using separation (separable threats). Joslin & Pollack (1994) generalize this strategy to include open conditions as well. Their "least-cost flaw repair" (LCFR) strategy chooses the flaw for which the least number of refinements exist. Schubert & Gerevini (1995) propose the "zero-commitment last in first out" (ZLIFO) strategy for selecting open conditions in case there are no non-separable threats (ZLIFO delays resolution of all separable threats). ZLIFO assigns highest priority to open conditions that cannot be achieved at all, or that can only be achieved in one unique way. If no high priority open conditions exist, ordinary LIFO order is used. The rationale behind ZLIFO is that no choice is involved in the linking of a prioritized open condition, so it can be seen as a zero-commitment refinement. Still, any refinement tends to constrain the search space by adding variable bindings and causal links, which helps reduce the number of generated and explored plans.

Early Commitment through Flaw Selection

We too use a flaw selection strategy that sometimes delays the resolution of a threat to a causal link. Before resolving any threats, we choose to link *static* open conditions. A static open condition q_{static} is a literal that involves a predicate that occurs in the initial conditions, but not in the effects of any operator. This means that a static open condition always has to be linked to an effect of a_0. Because the initial conditions contain no variables, linking q_{static} to an initial condition will cause all free variables in q_{static} to get bound to a specific object. Linking static preconditions before working on any other flaws thus represents a bias towards early commitment of parameter bindings. This resembles the search strategy inherent in planners using ground actions, but we can avoid binding all parameters of an action in one single step and thereby achieve a reduction in branching factor. In the absence of static preconditions and unsafe causal links, open conditions are selected in LIFO order. This ordering of open conditions tends to maintain focus on completely resolving one goal in a depth-first manner before trying to resolve any other goals—a strategy that often is beneficial (cf. (Gerevini & Schubert 1996a)).[2]

While our flaw selection strategy may seem contrary to the spirit of least commitment, we have found that it can help reduce the number of explored search nodes. We believe the main reason for this is that with more variables bound to specific objects, it becomes easier to detect potential conflicts. Early detection of conflicts is important, because it helps the planner avoid going deep down dead-end branches.

Joint Parameter Domain Constraints

When planning with ground actions, it is common practice to first find all complete instantiations of the available operators, discarding those having preconditions that cannot possibly be achieved. We can, for instance, discard any instantiation with static preconditions that are not fulfilled by the initial conditions. As an example, consider the `drive` operator from the logistics domain (Figure 1). When instantiating this operator, we can immediately discard any instan-

```
(:action drive
  :parameters (?truck ?s ?d ?city)
  :precondition (and (truck ?truck)
                     (at ?truck ?s)
                     (in-city ?s ?city)
                     (in-city ?d ?city)
                     (not (= ?s ?d)))
  :effect (and (at ?truck ?d)
               (not (at ?truck ?s))))
```

Figure 1: Operator in the logistics domain for moving a truck between two locations.

tiation that binds the `?truck` parameter to an object that is not a truck.

The relaxed planning graph can be used to further restrict the set of ground actions that can ever be part of a plan, because only a ground action appearing in the planning graph can possibly have its preconditions achieved. Again, consider the `drive` operator from the logistics domain. This is the only operator available for moving trucks. As can be seen, the preconditions of the operator assert that the two locations are in the same city, which means that a truck starting out at a location in Boston, for example, can never be moved to a location in Pittsburgh. We can derive this information from the relaxed planning graph and, in many cases, the result can be a significant reduction in the number of feasible initiations of an operator. For the logistics-a problem used later in the evaluation section, the number of ground actions is reduced from 11,558 when pruning is based only on unachievable preconditions, to 210 after considering the relaxed planning graph.

A ground action can effectively be seen as a joint domain constraint on the parameters of the corresponding operator. By pruning the set of ground actions as described above, we obtain tighter parameter domain constraints. For example, for the logistics-a problem we obtain the following possible joint assignments to the parameters of the `drive` operator:

$$\begin{aligned}&\langle \texttt{?truck}, \texttt{?s}, \texttt{?d}, \texttt{?city}\rangle \in \\ &\quad \{\langle \texttt{bos-truck}, \texttt{bos-po}, \texttt{bos-airport}, \texttt{bos}\rangle \\ &\quad \langle \texttt{bos-truck}, \texttt{bos-airport}, \texttt{bos-po}, \texttt{bos}\rangle \\ &\quad \langle \texttt{pgh-truck}, \texttt{pgh-po}, \texttt{pgh-airport}, \texttt{pgh}\rangle \\ &\quad \langle \texttt{pgh-truck}, \texttt{pgh-airport}, \texttt{pgh-po}, \texttt{pgh}\rangle \\ &\quad \langle \texttt{la-truck}, \texttt{la-po}, \texttt{la-airport}, \texttt{la}\rangle \\ &\quad \langle \texttt{la-truck}, \texttt{la-airport}, \texttt{la-po}, \texttt{la}\rangle\}\end{aligned}$$

We can make use of such joint parameter domain constraints when planning with lifted actions. Every time we add an action to a plan, we also add the joint parameter domain constraints of the corresponding operator to the binding constraints of the plan. When adding other binding constraints, we update the parameter domain constraints accordingly. For example, if we bind the `?truck` parameter of a `drive` action to `pgh-truck`, we keep only the tuples of the action's parameter domain constraints with `pgh-truck` in the position for `?truck`, which would leave us

[2] Pollack, Joslin, & Paolucci (1997) argue, however, that much of the benefit of ZLIFO can be attributed to the delayed resolution of separable threats.

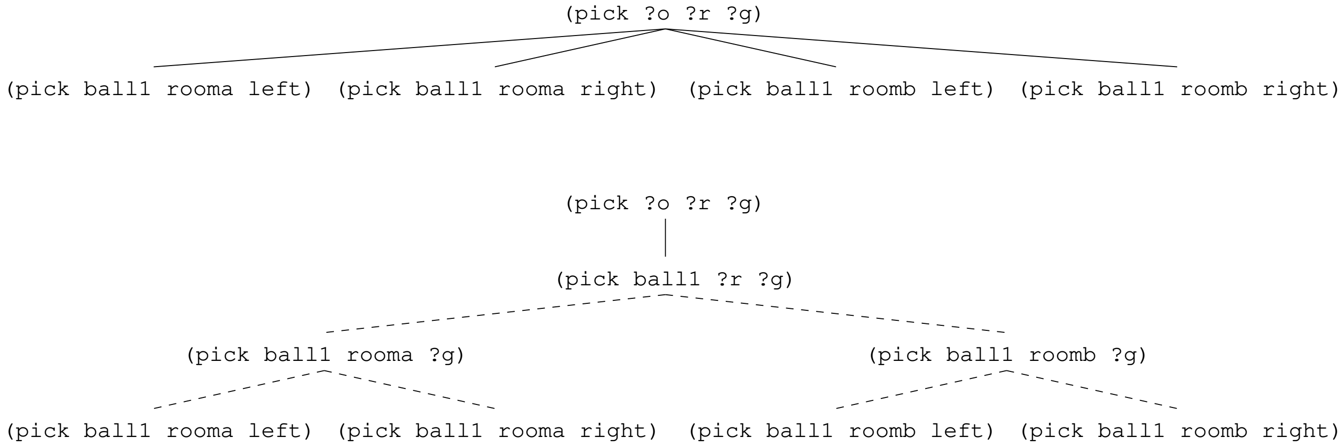

Figure 2: Tree representing the instantiation of the operator `(pick ?o ?r ?g)` needed to support the open condition `(carry ball1)` in the gripper domain, when using ground actions (top) and lifted actions (bottom). Solid lines represent instantiations, and dashed lines represent linking of static open conditions.

with the constraint

$$\begin{aligned}&\langle \texttt{?truck}, \texttt{?s}, \texttt{?d}, \texttt{?city}\rangle \in\\ &\quad \{\langle \texttt{pgh-truck}, \texttt{pgh-po}, \texttt{pgh-airport}, \texttt{pgh}\rangle\\ &\quad\ \langle \texttt{pgh-truck}, \texttt{pgh-airport}, \texttt{pgh-po}, \texttt{pgh}\rangle\}.\end{aligned}$$

This immediately gives us a binding for the `?city` parameter, viz. `pgh`. Inequality constraints can be propagated in a similar way to further restrict the joint parameter domain constraints. Our empirical tests show that this constraint propagation, together with our flaw selection strategy that biases towards early commitment of parameter bindings, makes planning with lifted actions comparable to planning with ground actions.

The idea of inferring parameter domains from the problem description, and making use of these domain constraints to restrict the search space for partial order planners, was tested by Gerevini & Schubert (1996b), who reported it to be very useful. They do not consider *joint* domain constraints for actions as we do, but rather compute separate domains for predicate parameters. In the example above, they would not be able to conclude that `?city` should be bound to `pgh` once `?truck` is bound to `pgh-truck`. By recoding joint domain constraints, we are able to restrict possible bindings even more than they can. In fact, our heuristic cost function already enforces the type of domain constraints on predicate parameters that they use. The cost of an atomic literal is simply ∞ if it is unachievable because of some parameter bindings.

Empirical Results

In this section, we demonstrate the effectiveness of the two proposed techniques—linking static preconditions before resolving threats and enforcing joint parameter domain constraints—both in isolation and combined. Our experiments indicate that these two techniques represent the power of ground actions to a great extent, and when used together they yield similar results as when planning with ground actions.

When instantiating all ground actions, we eliminate those that have static preconditions not present in the initial conditions. It is sufficient to make this check in the preprocessing stage, so we will never have to link any static open conditions when using ground actions. When planning with lifted actions, we use the static open conditions to represent the choice of binding parameters to specific objects. This can lead to a reduced branching factor, as illustrated by the "instantiation trees" in Figure 2 for an example from the gripper domain. The operator `(pick ?b ?r ?g)` has static preconditions `(ball ?b)`, `(room ?r)`, and `(gripper ?g)`, and is used to achieve the goal `(carry ball1)`. We assume that there are two rooms (`rooma` and `roomb`) and two grippers (`left` and `right`) in the world model. With ground actions, we end up generating four instantiations in one refinement step (upper part of Figure 2), while with lifted actions we instantiate one parameter per refinement step resulting in a tree structure with lower branching factor at each level (lower part of Figure 2). If the search is focused (e.g. if we choose to work on the plan where `?r` is bound to `rooma` and it turns out to be a good choice), we will not have to generate the entire tree in the latter case, while in the former case we have no choice but to generate all instantiations at once.

Because static preconditions are used to implement a bias towards early parameter bindings when using lifted actions, they cannot be eliminated as is the case when planning with ground actions. This will inevitably result in additional refinement steps for which there will be no counterpart when using ground actions. To account for this fact in the empirical evaluation, we subtract one from both the number of generated and explored plans every time a static open condition is processed for which there is at least one refinement. The rationale behind this counting scheme is as follows: When choosing to add a ground action to resolve an open condition, we generate in one refinement step a new plan for each

Problem	heuristic	static	domains	all	ground
gripper-8	3,729 / 3,572	566 / 441	576 / 441	566 / 441	1,089 / 441
gripper-10	26,089 / 25,852	965 / 784	986 / 784	965 / 784	1,958 / 784
gripper-12	*	1,529 / 1,280	1,565 / 1,280	1,529 / 1,280	3,224 / 1,280
gripper-20	*	6,204 / 5,514	6,348 / 5,514	6,204 / 5,514	14,386 / 5,514
rocket-ext-a	57,278 / 47,435	34,498 / 29,150	57,273 / 47,435	34,493 / 29,150	27,983 / 20,147
rocket-ext-b	*	36,468 / 30,438	*	36,468 / 30,438	26,461 / 18,670
logistics-a	*	7,004 / 4,745	1,107 / 827	498 / 338	559 / 345
logistics-b	*	12,446 / 8,138	8,226 / 6,363	633 / 446	734 / 415
logistics-c	*	12,705 / 8,321	12,164 / 9,978	718 / 497	701 / 397
logistics-d	*	>31,000†	*	2,524 / 1,639	3,167 / 1,494

Table 1: Shows the effect of different techniques on the number of generated/explored plans. The h_{add} heuristic is used in all cases. The baseline ("heuristic") uses the standard flaw ordering and no domain constraints; "static" links static open conditions before resolving threats; "domains" enforces joint parameter domain constraints; "all" uses both these techniques; and "ground" uses ground actions. A star (*) means that more than 100,000 nodes were generated, and a dagger (†) means that the planner ran out of memory on a machine with 256Mb of RAM after generating at least the indicated number of nodes.

usable ground action. When working with lifted actions, we instead add only one action, and later commit to one, or a few, parameter bindings at a time for each static preconditions we link. This corresponds to not counting internal nodes in the "instantiation tree" of an action.

Table 1 shows the results of our test runs on problems from three different domains. All tests were run with $h(\pi) = h_{\text{add}}(\mathcal{OC}(\pi))$ as the heuristic cost function. We wanted to test the effectiveness of the proposed techniques both separately and combined. The baseline (labeled "heuristic" in the table) uses lifted actions without giving priority to static open conditions, and without enforcing joint parameter domain constraints. As can be seen in the table, this configuration solved the least number of problems, and was by far the worst on the problems it did solve.[3] Linking static open conditions before working on any other flaws ("static") improves the results significantly compared to the baseline. Only the logistics-d problem remained unsolved.

The enforcing of joint parameter domain constraints ("domains") gives the best result in the the logistics domain, where using the relaxed planning graph helps reduce the number of instantiated actions tremendously. For the gripper and rocket domains, the relaxed planning graph does not cause any additional pruning of the joint parameter domains, in which case just linking static preconditions first is more effective.

When combining the two techniques ("all"), the performance for all domains is comparable to when using ground actions. In the gripper domain, the number of explored nodes is exactly the same, while the number of generated nodes is only half of what it is when using ground actions.

The results in the logistics domain, when combining the two techniques, are comparable to the results from using ground actions. For most problems the number of explored nodes is slightly larger in the lifted case, but the number of generated nodes is slightly smaller.

[3] We also tried using $h(\pi) = |\mathcal{OC}(\pi)|$ as heuristic cost function, but then we were not able to solve any of the problems before generating 100,000 nodes.

The performance of the planner in the rocket domain when using lifted actions does not quite match up to the performance in the ground case. In both the logistics domain and the rocket domain, the main disadvantage of using lifted actions is that the heuristic cost function often underestimates the cost of achieving an open condition. The reason for this is that each open condition is matched to a ground literal without considering interactions between open conditions of the same action. For example, the logistics domain has an action `(unload ?o ?v ?l)` with preconditions `(in ?o ?v)` and `(at ?v ?l)`. The heuristic function estimates the cost of these two open conditions independently, which can lead to different matchings for the parameter `?v`. When planning with ground actions, all open conditions are fully instantiated so the matching to ground literals are unique. It would be possible to modify the heuristic function so that it matches sets of open conditions simultaneously, making sure that the same parameter is not matched with two different objects. There would be a higher overhead in the heuristic evaluation of a plan, but the result should be a much closer match in the number of explored nodes.

When it comes to running time, using ground actions was faster in all cases. Even in the gripper domain, where the performance measured in number of generated/explored nodes point in favor of lifted actions, the running time when using ground actions was 38-45% of what it was when using lifted actions. This can be attributed to the overhead associated with looking up and updating variable bindings, and to the matching of open conditions to ground literals in the heuristic function. We have not put much effort into optimizing these parts of the code, however, and we firmly believe that the running time can be improved significantly by using more sophisticated data structures for recoding variable bindings.

Still, in some cases the reduction in number of generated nodes is so significant that the running time when using lifted actions actually becomes lower than it is when using ground actions. Figure 3 shows an example of such a case where the domain is the four operator blocks world, and the

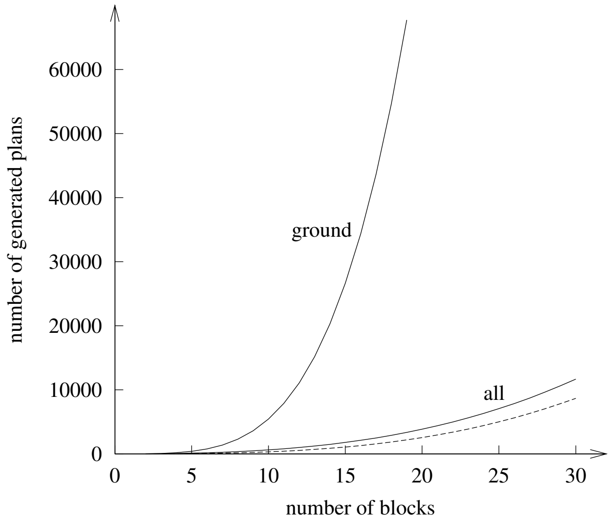

Figure 3: Shows the number of generated plans using ground actions and lifted actions (all) for blocks world problems of different size. The dashed curve is the number of explored plans, which is the same in both cases.

problem is to stack n blocks in one tower, with all the blocks initially on the table.[4] The number of generated nodes grows dramatically as a function of n in the ground case, while the number of generated nodes when using lifted actions remains within a factor two of the number of explored nodes (which is the same in both cases). The planner runs out of memory for problems with more than 19 blocks when using ground actions, while it comfortably solves problems with up to 30 blocks when using lifted actions. In terms of running time, the advantage grows for lifted actions with increasing numbers of blocks. The planner is up to a factor 1.7 faster when using lifted actions, compared with using ground actions.

While a reduction in the number of generated nodes is important, a reduction in the number of explored nodes is likely to give even greater performance improvements. One would expect that there are domains where planning with lifted actions could lead to a significant reduction in the number of explored nodes. So far, we have mostly been interested in showing how we can search the same number of plans when using lifted actions as when using ground actions. Because we are linking static open conditions first, we cannot expect to search many fewer plans when using lifted action. It also means that we need to keep the static open conditions while we are planning, although the enforcement of joint parameter domain constrains actually makes them superfluous.

The grid world domain (McDermott 1999) is a good candidate domain for which planning with lifted actions could be beneficial. The domain represents a robot that is able to move between adjacent positions in a grid. A problem will typically require the robot to move between two positions that are not adjacent, in which case the robot has to move over some intermediate positions. When planning with ground actions, the planner will have to commit to which intermediate positions to use at the point when an action is added to a plan, but if we use lifted actions we can defer such a decision. This can have a tremendous impact on performance. Of course, if we link static open conditions first, we will commit to a particular intermediate position at an early stage anyway, and the potential benefit is lost. If we instead use a low-commitment flaw selection strategy such as LCFR, in combination with enforcement of joint parameter domain constraints, then we can eliminate static preconditions. For the grid world problem illustrated in Figure 4, this strategy is superior. While none of the other strategies presented so far are able to solve this problem before generating more than 100,000 plans (and exploring over 60,000), LCFR with domain constraints solves the problem generating only 3,704 plans and exploring 3,084. We also tried using LCFR with ground actions, but the result was just as bad as when using the standard flaw selection strategy.

[4] In the encoding of the blocks world domain that we use there are no static preconditions, so the node count has not been adjusted.

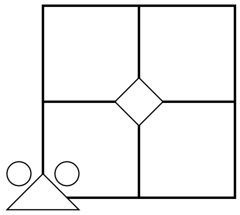

Figure 4: A grid world, with a robot in the lower left corner and a diamond-shaped key in the center. The problem is to have the robot move the key to the top right corner of the grid, and then return to the lower left corner.

Discussion

This paper has shown that the power of ground actions in refinement planning can be attributed mainly to two factors—early commitment of parameter bindings and enforcement of joint parameter domain constraints. We have implemented similar strategies in a planner using lifted actions, and showed that the performance becomes comparable to that of a planner using ground actions. Ground actions still work better in some domains, but this can be explained by the fact that our heuristic function evaluates open conditions of an action separately without making sure that the same parameter is matched to the same object in different open conditions.

By giving priority to static open conditions over other flaws, we implement a bias towards early commitment of parameter bindings. While somewhat in conflict with the least commitment principle, we have shown that this flaw selection strategy can improve performance significantly. One reason for this is that the more parameters that are bound to objects, the more accurate our heuristic function becomes. In addition, with tighter constraints on the parameters, it becomes easier to detect inconsistencies. We have experimented with more low-commitment flaw selection strategies that delay linking of static open conditions, but a big problem that we have observed with these strategies is that inconsistencies tend to pass undetected for a long time, which can cause large irrelevant parts of the search space to be explored. Inequality constraints (arising through separation) are particularly hard to deal with. We may have the constraints `?g1` $\neq$ `?g2`, `?g1` $\neq$ `?g3`, and `?g2` $\neq$ `?g3` in addition to some domain constraints for the parameters, and the only way to detect that no consistent assignment exists for them may be to solve the corresponding constraint satisfaction problem. With parameters bound to objects, it becomes trivial to determine whether an equality or inequality constraint holds. It also disables separation in many cases, which can be a contributing factor to the improved performance.

We have shown, however, that a low-commitment flaw selection strategy such as LCFR, used in combination with enforcement of joint parameter domain constraints, sometimes can lead to a vast reduction in the number of explored nodes when planning with lifted actions. While the main contribution of this paper has been to indicate where the power of using ground actions in refinement planning lies, we have also presented results suggesting that there are situations when planning with lifted actions is beneficial.

For future research, we would like to further investigate the impact that the flaw selection strategy has on the performance of partial order planners. We have developed several novel flaw selection strategies, for example one that ranks open conditions based on heuristic cost, and the initial results are promising.

References

Blum, A. L., and Furst, M. L. 1995. Fast planning through planning graph analysis. In Mellish, C. S., ed., *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, 1636–1642. Montreal, Canada: Morgan Kaufmann Publishers.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 714–719. Providence, RI: AAAI Press.

Gerevini, A., and Schubert, L. 1996a. Accelerating partial-order planners: Some techniques for effective search control and pruning. *Journal of Artificial Intelligence Research* 5:95–137.

Gerevini, A., and Schubert, L. 1996b. Computing parameter domains as an aid to planning. In Drabble, B., ed., *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, 94–101. Edinburgh, Scotland: AAAI Press.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems*, 140–149. Breckenridge, CO: AAAI Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Joslin, D., and Pollack, M. E. 1994. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1004–1009. Seattle, WA: AAAI Press.

McAllester, D. A., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 634–639. Anaheim, CA: AAAI Press.

McDermott, D. 1996. A heuristic estimator for means-ends analysis in planning. In Drabble, B., ed., *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, 142–149. Edinburgh, Scotland: AAAI Press.

McDermott, D. 1999. Using regression-match graphs to control search in planning. *Artificial Intelligence* 109(1–2):111–159.

Nguyen, X., and Kambhampati, S. 2001. Reviving partial order planning. In Nebel, B., ed., *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 459–464. Seattle, WA: Morgan Kaufmann Publishers.

Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B.; Rich, C.; and Swartout, W., eds., *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 103–114. Cambridge, MA: Morgan Kaufmann Publishers.

Peot, M. A., and Smith, D. E. 1993. Threat-removal strategies for partial-order planning. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*. Washington, DC: AAAI Press.

Pollack, M. E.; Joslin, D.; and Paolucci, M. 1997. Flaw selection strategies for partial-order planning. *Journal of Artificial Intelligence Research* 6:223–262.

Schubert, L., and Gerevini, A. 1995. Accelerating partial order planners by improving plan and goal choices. In *Proceedings of the Seventh International Conference on Tools with Artificial Intelligence*, 442–450. Herndon, VA: IEEE Computer Society Press.

Weld, D. S. 1994. An introduction to least commitment planning. *AI Magazine* 15(4):27–61.