Finding State Similarities for Faster Planning

Christian Fritz
Department of Computer Science, University of Toronto,
Toronto, Ontario. CANADA.
fritz@cs.toronto.edu

Abstract

In many planning applications one can find actions with overlapping effects. If for optimally reaching the goal all that matters is within this overlap, there is no need to consider all these actions – for the task at hand they are equivalent. Using this structure for speed-up has previously been proposed in the context of least commitment planning. Of a similar spirit is the approach for improving best-first search based planning we present here: intuitively, given a set of start states, reachable from the initial state, we plan in parallel for all of them, exploiting the similarities between them to gain computational savings. Since the similarity of two states is problem specific, we explicitly infer it by regressing all relevant entities, goal, heuristic function, action preconditions and costs, over the action sequences considered in planning. If the resulting formulae mention only fluents whose values the two states have in common, it suffices to evaluate the formulae in one of them. This leads to computational savings over conventional best-first search.

Introduction

Consider the oft-used trip planning problem, where a user needs to book a flight, a hotel, and a rental car for specific dates. A conventional forward search based planner, even one which is heuristically guided, would go about enumerating the possible combinations of flight, hotel, and car booking actions, not realizing that these are independent tasks which can be optimized separately as long as there are no constraints between them. A planner that can exploit this independence can gain exponential speed-up, as has been shown in the past in particular in the context of least commitment planning (Friedman & Weld 1996).

However, many of the top performers at the International Planning Competition are planners based on best-first forward search. We show that above idea can be extended and applied also to forward search based planners, including heuristic search based ones, and that in fact more than just similar action effects can be exploited. Intuitively, also similar non-effects of actions can be exploited.

The intuitive idea of the approach we present and test empirically is as follows: imagine there are N actions possible in the initial state $A_1, \ldots, A_N$. Each of these defines a sub-search tree explored separately during search, but the difference between the states reached by these actions may actually not be that significant. In fact, the difference may not even be relevant to the given goal, feasibility of future actions, and, if provided, metric function. If this is the case, the resulting states could as well be considered in parallel, allowing for great computational savings.

But how do we know which aspects of a state, i.e. which state variables, are relevant? Let us consider partial-order planning, which seems more amenable to least commitment planning. One of the main differences to forward search is that search begins with the goal, choosing actions that satisfy parts of the goal in a backward chaining manner. Thus, for each chosen action we know its *purpose*. When there are two actions with overlapping effects, say `BookFlight(United)` and `BookFlight(Delta)`, whose particular effects beyond the commonality are irrelevant to the purpose, these can be treated jointly as one abstract action (cf. (Friedman & Weld 1996)).

In this paper we present a very similar approach for improving forward search. It works by fully expanding the first few levels of the search tree and henceforth planning for all the resulting states in parallel. We call these states *start states*. Parallel search is enabled by the use of regression. To *progress* a state over an action, means to modify the state according to the effects of the action. This is what most planners deploy for evaluating conditions in possible future states. *Regressing* a condition over an action in turn does the inverse. It produces a formula, describing all states from which the given action leads to a state where the condition holds. Regression reveals the relevance-structure of the problem. Note that the use of regression is not limited to backward chaining- and partial-order planning. Also when enumerating possible action sequences in a forward fashion, we can use regression to reason about what will be true after executing these actions.

We use this in our approach. To test, for instance, whether the goal is reached by an action sequence $[\alpha_1, \alpha_2]$ we regress the goal condition over the action sequence and evaluate the resulting formula in the current state. This can also be done for all other relevant entities, like action preconditions, costs, and even a heuristic function. Note that the regression of a formula is independent of the state in which the result is meant to be evaluated in, allowing us to evaluate it in many

states, in our case, in all start states. This alone would not yet lead to computational savings, since the evaluation of the formula can be complex. However, now that we have a concise formula describing what is relevant to solving the given planning problem, we can determine and exploit similarities between all considered start states. If the formula does not mention any of the state variables on which two particular start states disagree (i.e. give different values to), then it suffices to evaluate the formula in one of them, the (truth-)value with respect to the other will be the same! This can lead to computational savings, as we show empirically.

The approach we present here works with any action language for which regression can be defined, including STRIPS and ADL. Here we use the situation calculus, which we review in the next section. We then present our approach, followed by empirical results, and a discussion.

Background

The situation calculus is a logical language for specifying and reasoning about dynamical systems (Reiter 2001). In the situation calculus, the *state* of the world is expressed in terms of *fluents* (set $\mathcal{F}$), functions and relations relativized to a *situation* s, e.g., $F(\vec{x}, s)$. A situation is a *history* of the primitive actions a performed from a distinguished initial situation S_0. The function $do(a, s)$ maps an action and a situation into a new situation thus inducing a tree of situations rooted in S_0. For readability, action and fluent arguments are generally suppressed. Also, $do(a_n, do(a_{n-1}, \ldots do(a_1, s)))$ is abbreviated to $do([a_1, \ldots, a_n], s)$ or $do(\vec{a}, s)$ and $do([\,], s) \stackrel{\text{def}}{=} s$. In this paper we only consider finite sets of actions, $\mathcal{A}$.

A basic action theory in the situation calculus, $\mathcal{D}$, comprises four *domain-independent foundational axioms*, and a set of *domain-dependent axioms*. Details of the form of these axioms can be found in (Reiter 2001). Included in the domain-dependent axioms are the following sets:

Initial State, S_0: a set of first-order sentences relativized to situation S_0, specifying what is true in the initial state.

Successor state axioms: provide a parsimonious representation of frame and effect axioms under an assumption of the completeness of the axiomatization. There is one successor state axiom for each fluent, F, of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$, where $\Phi_F(\vec{x}, a, s)$ is a formula with free variables among $\vec{x}, a, s$. $\Phi_F(\vec{x}, a, s)$ characterizes the truth value of the fluent $F(\vec{x})$ in the situation $do(a, s)$ in terms of what is true in situation s. These axioms can be automatically generated from effect axioms (e.g. add and delete lists).

Action precondition axioms: specify the conditions under which an action is possible. There is one axiom for each action $a \in \mathcal{A}$ of the form $Poss(a(\vec{x}), s) \equiv \Pi_a(\vec{x}, s)$ where $\Pi_a(\vec{x}, s)$ is a formula with free variables among $\vec{x}, s$.

Regression

The *regression* of a formula ψ through an action a is a formula ψ' that holds prior to a being performed if and only if ψ holds after a is performed. In the situation calculus, one step regression is defined inductively using the successor state axiom for a fluent $F(\vec{x})$ as above:

$$\mathcal{R}[F(\vec{x}, do(a, s))] = \Phi_F(\vec{x}, a, s) \quad | \quad \mathcal{R}[\neg\psi] = \neg\mathcal{R}[\psi]$$
$$\mathcal{R}[\psi_1 \wedge \psi_2] = \mathcal{R}[\psi_1] \wedge \mathcal{R}[\psi_2] \quad | \quad \mathcal{R}[(\exists x)\psi] = (\exists x)\mathcal{R}[\psi]$$

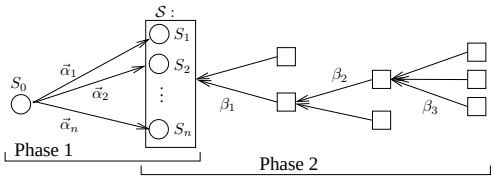


Figure 1: Schematic overview of our approach.

We use $\mathcal{R}[\psi, \alpha]$ to denote $\mathcal{R}[\psi(do(\alpha, s))]$, and $\mathcal{R}[\psi, \vec{\alpha}]$ to denote the repeated regression over all actions in the sequence $\vec{\alpha}$ (in reverse order). The resulting formula has a free variable s of sort situation. Intuitively, it is the condition that has to hold in s in order for ψ to hold after executing $\vec{\alpha}$ (i.e. in $do(\vec{\alpha}, s)$). It is predominantly comprised of the fluents occurring in the conditional effects of the actions in $\vec{\alpha}$. Due to the Regression Theorem (Reiter 2001) we have that $\mathcal{D} \models \psi(do(\vec{\alpha}, s)) \equiv \mathcal{R}[\psi, \vec{\alpha}]$ for all situations s.

Regression is a purely syntactic operation. Nevertheless, it is often beneficial to simplify the resulting formula for later evaluation. Regression can be defined in many action specification languages. In STRIPS, regression of a literal l over an action a is defined based on the add and delete lists of a: $\mathcal{R}^{\text{STRIPS}}[l] = \text{FALSE}$ if $l \in \text{DEL}(a)$ and $\{l\} \setminus \text{ADD}(a)$ otherwise. Regression in ADL was defined in (Pednault 1989).

Notation: We use α to denote arbitrary but explicit actions and S to denote arbitrary but explicit situations, that is $S = do(\vec{\alpha}, S_0)$ for some explicit action sequence $\vec{\alpha}$. Further $\vec{\alpha} \cdot \alpha$ denotes the result of appending action α to the sequence $\vec{\alpha}$.

As an example of regression, consider a formula stating "at Union Square and cash $=$ \$10". Regressing this formula over an action sequence ["take subway to Times Square", "buy ice cream"] yields a condition "cash $=$ \$15", when a subway ride costs \$2 and an ice cream \$3, say.

Forward Search Regression Planning

To give a formal account of our approach, we assume the planning domain is described as a basic action theory $\mathcal{D}$.

Figure 1 shows our approach schematically. In Phase 1, we generate a small set of action sequences $\vec{\alpha}_1, \ldots, \vec{\alpha}_n$ feasible in the initial state described by S_0. We refer to the resulting situations as *start situations* and denote them $S_i = do(\vec{\alpha}_i, S_0)$, and their set by $\mathcal{S}$. We refer to the tree spanned by them as the *pre-tree*. In Phase 2, we then perform planning for all these situations in parallel, only exploring one search tree for all start situations together. When Phase 2 finds a goal for this abstract state, the result can be combined with the best start sequence, which can be easily determined from additional information returned in Phase 2.

To better distinguish between action sequences considered in the two phases, we use $\vec{\alpha}$ to refer to action sequences of Phase 1, and $\vec{\beta}$ for those considered in Phase 2.

Phase 1: Pre-computation

The set of start situations can be generated in a variety of ways, for instance, using a depth limited breadth-first search or A^* search with a small expansion limit. Our approach is independent of the generation and shape of the pre-tree, as long as it includes a prefix for each feasible action sequence.

Algorithm 1 Pre-compute matrix M (Phase 1)

Input: situations $\mathcal{S}$, ground fluents $\mathcal{F}_g$, theory $\mathcal{D}$
Output: matrix M

1: $n \Leftarrow |\mathcal{S}|$; $m \Leftarrow |\mathcal{F}_g|$
2: M $\Leftarrow$ new matrix(n, m)
3: **for** $i \Leftarrow 1$ to n **do** *// For each start situation*
4: **for** $j \Leftarrow 1$ to m **do** *// and each fluent*
5: M$[i, j] \Leftarrow v$ with v s.t. $\mathcal{D} \models F_j(S_i) = v$
6: **end for** *// compute the assigned value.*
7: **end for** *// Then, determine*
8: **for** $i \Leftarrow 1$ to n **do** *// for each start situation*
9: **for** $j \Leftarrow 1$ to m **do** *// and each fluent*
10: *equals* $\Leftarrow$ new empty set *// the set of*
11: **for all** $i' \Leftarrow 1$ to n **do** *// all other situations*
12: **if** $[i, j] = [i', j]$ **then** *// with the same fluent value*
13: add i' to equals *// and store it*
14: **end if**
15: **end for**
16: M$[i, j] \Leftarrow$ (M$[i, j]$, *equals*) *// in the matrix.*
17: **end for**
18: **end for**
19: **return** M

We study the effects of using different cardinalities of start situations in the empirically results section and, as we will see, guessing a good value is not that hard.

We assume complete knowledge of the initial state of the world, described in S_0, and a finite number of ground fluents over a finite number of objects. We refer to the set of all ground fluents $\{F_1, \ldots, F_m\}$ as $\mathcal{F}_g$. For each situation in $\mathcal{S}$ we can compute the values of all ground fluents, effectively progressing the initial state over the actions in the sequence describing that situation. Further, we can group start situations by fluents on whose value they agree.

Algorithm 1 formalizes this pre-computation. It consults the action theory $\mathcal{D}$ to determine the (truth-)value for formulae, by testing entailment ($\models$). The algorithm returns a matrix, M, whose rows denote start situations, and columns denote ground fluents. Hence $[i, j]$ denotes the cell corresponding to the i-th start situation S_i and j-th ground fluent F_j. The entries of the matrix are of the form $(v, \{i'_1, \ldots, i'_k\})$, where v is the value of F_j in S_i, i.e. $\mathcal{D} \models F_j(S_i) = v$, and $\{i'_1, \ldots, i'_k\}$ is a possibly empty set of indices. The latter indicates all other start situations that assign the same value v to fluent F_j. This set is the key to the speed-up provided by our approach.

Phase 2: Parallel Planning

With the matrix M in place, we can start the actual planning in Phase 2.[1] In this second phase, we plan in parallel for all considered start situations.

While our approach is conceptually compatible with a number of forward search algorithms, we here only consider A^* search-based planning, and assume that the metric is defined in terms of positive action costs.

The first difference between conventional A^* and the variant we present here, is that our algorithm deploys regression to reason about the state of the world as opposed to progression, as outlined earlier. The second difference is that we evaluate each relevant function or relation in all start states rather than just one initial state. This is made possible by the use of regression and sped up by exploiting the lists of situations with coinciding fluent values, stored in M.

[1] For parsimony, we assume no plan can be found in Phase 1.

Algorithm 2 Regression A^* Parallel Search (Phase 2)

Input: $\mathcal{D}, \mathcal{S}, \text{M}, \mathit{Goal}(s), \mathit{Cost}(a, s), H(s)$
Output: a plan for some $S \in \mathcal{S}$ or "failure" if none exists

1: OPEN $\Leftarrow \left[(\vec{g}, \vec{\infty}, [\,], \{1, \ldots, n\})\right]$
2: *plan* $\Leftarrow (\infty, [\,], 0)$
3: **repeat**
4: $(\vec{g}, \vec{h}, \vec{\beta}, I) \Leftarrow$ get-and-remove first element from OPEN
5: *plan*, $I' \Leftarrow$GOALTEST$((\vec{g}, \vec{h}, \vec{\beta}, I), \mathcal{D}, \mathcal{S}, \text{M}, \mathit{Goal}(s), \mathit{plan})$
6: $(g^*, \vec{\beta}^*, i^*) \Leftarrow$ *plan*
7: **if** $g^* < \min_{i \in I'}(\vec{g}[i] + \vec{h}[i])$ **then** *// found plan is optimal*
8: **return** *plan*
9: **else** *// there may still be a better plan, continue*
10: $succ \Leftarrow$ EXPAND$((\vec{g}, \vec{h}, \vec{\beta}, I'), \mathcal{D}, \mathcal{S}, \text{M}, \mathit{Cost}(a, s), H(s))$
11: insert-sorted $succ$ into OPEN
12: **end if**
13: **until** OPEN= [ ]
14: **return** "no plan exists"

Besides the basic action theory $\mathcal{D}$, the algorithm receives as input the set of start situations $\mathcal{S}$, the matrix M, the goal formula $\mathit{Goal}(s)$, the cost function $\mathit{Cost}(a, s)$ defining the cost of performing action a in situation s, and an admissible heuristic function $H(s)$. The latter two are expressed in terms of (regressable) formulae in $\mathcal{D}$. The heuristic defines for a situation s a lower bound on the minimal costs accrued by any action sequence that connects s with a goal state. The algorithm outputs a sequence of actions $\vec{\beta}^*$, and an index i such that: $\vec{\beta}^*$ is executable in S_i, the reached situation $do(\vec{\beta}^*, S_i)$ satisfies the goal, and is optimal in the sense that the costs accumulated by executing its actions are minimal. By assumption, all start situations are executable in S_0.

The algorithm is specified in Algorithm 2. It uses an *open list*, OPEN, containing elements of the form $(\vec{g}, \vec{h}, \vec{\beta}, I)$. Here I is a subset of all start situation indices, $\vec{\beta}$ is an action sequence possible in all situations contained in I, and $\vec{g}$ and $\vec{h}$ are vectors of values, one for $i \in I$. The vector $\vec{g}$ states the accumulated costs, $\vec{h}$ the heuristic values. The i-th vector element of $\vec{g}$, for instance, denotes the costs accumulated when $\vec{\beta}$ is executed in S_i. Similarly the value $\vec{h}[i]$, the i-th element of the second vector, is such that $\mathcal{D} \models H(do(\vec{\beta}, S_i)) = \vec{h}[i]$. The open list is initialized with a single element $(\vec{g}, \vec{\infty}, [\,], \{1, \ldots, n\})$, denoting the empty sequence and $\vec{g}$ containing the costs accumulated by the actions of the start situations. The last element in the tuple is a set of all indices i of start situations, S_i, for which this action sequence is feasible ($\mathcal{D} \models \mathit{executable}(do(\vec{\beta}, S_i))$). Initially this set includes all indices $1, \ldots, n$, since the empty sequence is trivially executable in all start situations. The algorithm then repeatedly expands the first element of the open list until an optimal goal is found, where new search nodes are inserted into the open list (Line 11) sorted by their minimal f value defined as $f = \min_{i \in I}(\vec{g}[i] + \vec{h}[i])$.

So far, the algorithm looks like conventional A^* search.

Algorithm 3 GOALTEST

Input: $(\vec{g}, \vec{h}, \vec{\beta}, I), \mathcal{D}, \mathcal{S}, \mathrm{M}, \mathit{Goal}(s), \mathit{plan}$
Output: the best plan so far, its costs, and a pruned set I

1: $G \Leftarrow \mathcal{R}[\mathit{Goal}(s), \vec{\beta}]$ *// Regress the goal over $\vec{\beta}$.*
2: $I' \Leftarrow$ HOLDSIN$(G, I, \mathcal{S}, \mathrm{M})$ *// Test it in I.*
3: **if** $I' \neq \emptyset$ **then** *// If it holds for some $I' \subseteq I$,*
4: $i^* \Leftarrow \mathrm{argmin}_{i \in I'}(\vec{g}[i])$ *// get best new plan and*
5: $(g', \vec{\beta}', i') \Leftarrow \mathit{plan}$ *// get best previous plan.*
6: **if** $\vec{g}[i^*] < g'$ **then** *// If better than previous plan*
7: **return** $(g^*, \vec{\beta}, i^*), I \setminus I'$ *// return it,*
8: **else** *// otherwise*
9: **return** $(g', \vec{\beta}', i'), I \setminus I'$ *// keep old one.*
10: **end if** *// In both cases: prune goal states from I,*
11: **else** *// otherwise*
12: **return** plan, I *// nothing to do.*
13: **end if**

Algorithm 4 EXPAND

Input: $(\vec{g}, \vec{h}, \vec{\beta}, I), \mathcal{D}, \mathcal{S}, \mathrm{M}, \mathit{Cost}(a, s), H(s)$
Output: all feasible successors of $(\vec{g}, \vec{h}, \vec{\beta}, I)$

1: successors $\Leftarrow [\,]$
2: **for all** $\beta' \in \mathcal{A}$ **do** *// For all actions*
3: $P \Leftarrow \mathcal{R}[\mathit{Poss}(\beta', s), \vec{\beta}]$ *// regress the preconditions*
4: $I' \Leftarrow$ HOLDSIN$(P, I, \mathcal{S}, \mathrm{M})$ *// and evaluate them.*
5: **if** $I' \neq \emptyset$ **then** *// If β' is possible for at least one $i \in I$*
6: $C \Leftarrow \mathcal{R}[\mathit{Cost}(\beta', s), \vec{\beta}]$ *// regress the costs,*
7: $\vec{c} \Leftarrow$ EVALIN$(C, I', \mathcal{S}, \mathrm{M})$ *// evaluate,*
8: $\vec{g}' \Leftarrow \vec{g} \oplus \vec{c}$ *// and add them.*
9: $H \Leftarrow \mathcal{R}[H(s), \vec{\beta} \cdot \beta']$ *// Also regress the heuristic*
10: $\vec{h}' \Leftarrow$ EVALIN$(H, I', \mathcal{S}, \mathrm{M})$ *// and evaluate it.*
11: append $(\vec{g}', \vec{h}', \vec{\beta} \cdot \beta', I')$ to successors
12: **end if**
13: **end for**
14: **return** successors

The difference is in the auxiliary methods GOALTEST and EXPAND and the functions they use, which we will explain next. Later we also elaborate on the particularities of Line 11, given the format of the elements of the open list.

The function GOALTEST returns a tuple representing the best plan found so far or $(\infty, [\,], 0)$ otherwise. The returned tuple $(g, \vec{\beta}, i)$ is such that executing $\vec{\beta}$ in S_i achieved the goal with total costs g. The function begins by regressing the goal condition over the corresponding action sequence of the first open list element, and tests, using the function HOLDSIN described below, for which start situations the resulting formula holds. For all these, the minimum is determined and compared to the best previously found plan. If it is better, this new plan is returned, otherwise the old one is kept.

Another function used in our Regression A^* Parallel Search algorithm is EXPAND (Alg. 4). This function receives the first element of the open list, $(\vec{g}, \vec{h}, \vec{\beta}, I)$, and returns the list of all its feasible successors. Feasibility is defined in terms of the set of start situations indicated in I for which $\vec{\beta}$ is feasible. As long as an action β' is possible according to one of them, a successor will be produced. To test this, the preconditions of β' are regressed over the action sequence thus far, and the HOLDSIN function is used to determine the subset of I of indices for which they hold.

Algorithm 5 HOLDSIN

Input: $\psi, I, \mathcal{S}, \mathrm{M}$
Output: a set of indices $\mathit{holds} \subseteq I$ in which ψ holds

1: $\mathit{holds} \Leftarrow \emptyset$
2: **while** $I \neq \emptyset$ **do**
3: $\mathit{same} \Leftarrow I$
4: $i \Leftarrow$ get-and-remove one element from I *// For S_i*
5: **for all** $F \in$ FLUENTSIN(ψ) **do** *// and all fluents*
6: $(v, \mathit{equals}) \Leftarrow \mathrm{M}[i, F]$ *// get the equals lists*
7: $\mathit{same} \Leftarrow \mathit{same} \cap \mathit{equals}$ *// and intersect them.*
8: **end for**
9: $I \Leftarrow I \setminus \mathit{same}$ *// Prune the result from I.*
10: **if** $\mathcal{D} \models \psi(S_i)$ **then** *// If ψ holds in S_i*
11: $\mathit{holds} \Leftarrow \mathit{holds} \cup \mathit{same} \cup \{i\}$ *// append the indices.*
12: **end if**
13: **end while**
14: **return** holds

Algorithm 6 EVALIN

Input: $\psi, I, \mathcal{S}, \mathrm{M}$
Output: a vector of values, one for each element in I

1: $\mathit{values} \Leftarrow$ new vector of length $|\mathcal{S}|$
2: **while** $I \neq \emptyset$ **do**
3: $\mathit{same} \Leftarrow I$
4: $i \Leftarrow$ get-and-remove one element from I *// For S_i*
5: **for all** $F \in$ FLUENTSIN(ψ) **do** *// and all fluents*
6: $(v, \mathit{equals}) \Leftarrow \mathrm{M}[i, F]$ *// get the equals lists*
7: $\mathit{same} \Leftarrow \mathit{same} \cap \mathit{equals}$ *// and intersect them.*
8: **end for**
9: $I \Leftarrow I \setminus \mathit{same}$ *// Prune the result from I,*
10: get val s.t. $\mathcal{D} \models \psi(S_i) = \mathit{val}$ *// compute the value for ψ*
11: $\mathit{values}[i] \Leftarrow \mathit{val}$ *// and set it for i*
12: **for all** $i' \in \mathit{same}$ **do** *// as well as for all pruned indices.*
13: $\mathit{values}[i'] \Leftarrow \mathit{val}$
14: **end for**
15: **end while**
16: **return** values

If the resulting set is non-empty, then also the action costs are regressed and evaluated for all start situations in I using another function EVALIN. This function, which works similarly to HOLDSIN, returns the values for a given formula with respect to a set of start situation indices. The result is a vector of values. This vector is added to the accumulated costs so far, vector $\vec{g}$, to determine the new accumulated costs vector $\vec{g}'$. Similarly, the heuristic function is regressed over the new sequence formed by appending the considered action to the existing sequence. Again, the value vector is obtained from EVALIN. Finally, a new successor is added: $\vec{g}'$ is the new accumulated costs vector, $\vec{h}'$ contains the new heuristic values, $\vec{\beta} \cdot \beta'$ is the new action sequence, and I' contains all start situation indices for which it is feasible.

Let us now turn to the key to our computational savings: The functions HOLDSIN and EVALIN (Alg. 5 and 6). Since these work very similarly, we describe them together and only in abstract terms. Both are given a formula ψ to evaluate to a (truth-)value for each start situation indicated in a given set I. They also both receive the matrix computed in Phase 1. Both algorithms iterate over I, pruning it as much as possible as they go along. For each element $i \in I$, for each fluent F mentioned by the formula ψ, the matrix is con-

sulted to retrieve all start situations that give the same value to F as S_i. The resulting sets are intersected and the result is removed from I. This can be done because if for a start situation S_j all fluents mentioned in ψ are assigned the same value as by S_i, we do not need to evaluate ψ twice and can thus remove j from I. This is the core of our computational savings, which we demonstrate in the next section.

When Phase 2 terminates, a tuple $(g^*, \vec{\beta}^*, i^*)$ is returned and the action sequence described by the situation $do(\vec{\beta}^*, S_{i^*})$ is an optimal plan. Our algorithm preserves both the completeness and optimality of A^*.

Theorem 1. (Completeness) If there exists a sequence of actions $\vec{\alpha}$ s.t. $\mathcal{D} \models executable(do(\vec{\alpha}, S_0)) \wedge Goal(do(\vec{\alpha}, S_0))$, then our algorithm will find and return such a sequence.

Theorem 2. (Optimality) Any action sequence returned by our algorithm is optimal in the sense that it minimizes the costs accumulated by its execution.[2]

Intuitively, the theorems hold because (a) all computed g and h values stated in the open list are correct with respect to their corresponding start situations, (b) the list is sorted according to the minimum $\min_{i \in I}(\vec{g}[i] + \vec{h}[i])$, where I is the list of start situations for which the action sequence is feasible, and (c) these sets, I, correctly denote all feasible action sequences that can be built by appending the action sequence stated in the open list element to any start situation. Completeness and optimality are then shown based on the admissibility of the heuristic, analogous to the proof for conventional A^* search. Roughly, item (a) regards the correctness of HOLDSIN and EVALIN and follows from Reiter's Regression Theorem, and the correctness of the progression performed in the pre-tree. The latter is trivial given the assumption of complete knowledge and a finite set of ground fluents. Item (b) and (c) follow by construction of our algorithms, and for (c), again by correctness of HOLDSIN.

Empirical Results

We compared a preliminary implementation of our approach to a planner based on conventional A^* search. We used problems from the Trucks and the Zenotravel domain from the International Planning Competition. In both domains we experimented with different expansion depths for the pre-tree, resulting in different numbers of start situations. In all graphs, the expansion time and expanded nodes for this tree are included in the respective numbers for our approach.

In the Trucks domain we tested using uniform-cost search for both the baseline and our Phase 2 planner, by applying the zero-heuristic. Figure 2(a) shows the running time of the conventional best-first search baseline approach and our approach for various expansion depths of the pre-tree. Recall that both approaches produce an optimal plan. For readability we sorted the test cases by the running time of the A^* search baseline approach. The results show that our approach is particularly beneficial on large problems. Not visible in the graph is that for small problems a low number of start situations is better, while for larger problems the initial overhead for a large number of start situations pays off. The speed-up is explained by the reduced number of expanded nodes, shown in Figure 2(b). This number is, however, only of limited indication of the benefit of our approach. Only when the start situations have similarities that can be exploited does this lead to a practical speed-up. Such similarities depend on the domain, problem, and heuristic function, but also on the number of considered start situations.

In the Zenotravel domain we tested with several hand-coded, problem specific heuristics, the results of which we present jointly, for parsimony. In this domain, we did not at first achieve good results. Investigating the reason for this showed that there were fewer similarities than in Trucks and that the branches of the search tree in Phase 2 quickly got "thin", in the sense that the action sequences were only feasible for very few start situations, i.e. I got small. We hence added an adaptive functionality to our approach: when the pruning of I, done in HOLDSIN and EVALIN, is low, our search algorithm switches to conventional A^* search. The resulting adaptive algorithm enabled us to sometimes also obtain speed-ups in the Zenotravel domain, shown in 2(c), and almost never perform worse than the baseline approach.

Discussion

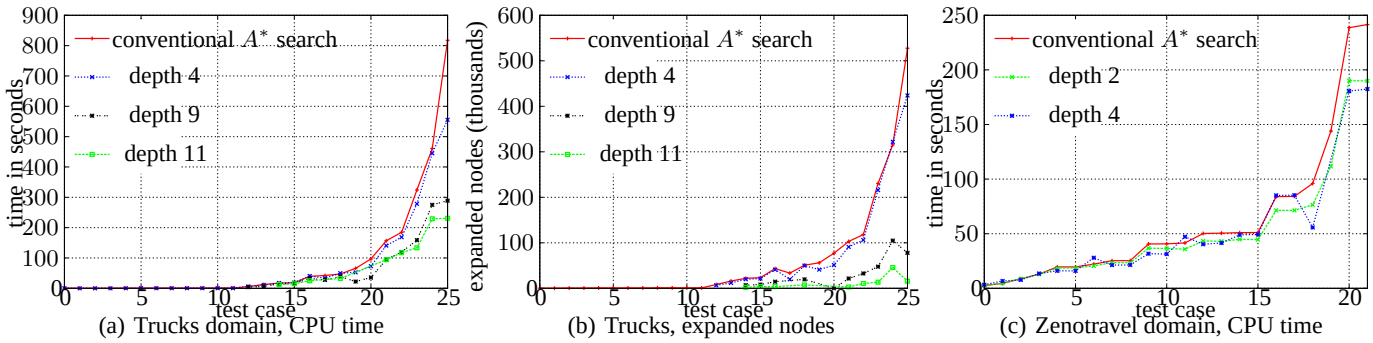
We have presented a novel approach for exploiting structure in planning problems that works by planning in parallel for several initial states, exploiting their problem specific similarities to gain computational savings. Our approach preserves the completeness and optimality of A^* search. Preliminary empirical results show that when states exhibit similarities, our approach can exploit them to realize computational savings. This seems particularly true for large problems. More experiments need to be run in future work to better characterize domains and problems for which the approach works well.

With respect to future work, first and foremost, the uninformed and in-adaptive choice of start situations seems to leave room for improvement. We imagine that an adaptive approach which dynamically adds and removes start situations during Phase 2, could perform significantly better, in particular on small problems. This also brings back the question as to how many start situations to use. Theoretically, the more start situations we consider, the greater the possible speed-up. But higher cardinalities also prolong pre-computation and node expansion. We hope to establish theoretical insights on this trade-off in the future. Also, a better implementation with optimized data structures is required in order to compare the approach to state-of-the-art planners.

In future work we may also consider planning under uncertainty. It is conceivable that there are similarities between the possible physical states of a belief state. This relates to similar work by Boutilier, Reiter, & Price (2001), who define symbolic dynamic programming using regression.

The main limitation of our approach when compared to state-of-the-art heuristic search based planners, is the requirement for the heuristic to be defined in terms of a regressable formula in the action theory (cf. (Reiter 2001) regarding the regressability of formulae). This does not seem

[2] A formal definition of optimality for planning in the situation calculus is provided by Lin (1999).



(a) Trucks domain, CPU time (b) Trucks, expanded nodes (c) Zenotravel domain, CPU time

Figure 2: Experimental results for various problems of the Trucks and Zenotravel domain.

immediately possible for popular heuristics, which are often defined algorithmically. Finding and applying successful heuristics that are compatible with our approach is another topic of future work. One such heuristic might be pattern databases (Edelkamp 2001). Also heuristics which are themselves already computed in a regression fashion as described, for instance, by McDermott (1999), Bonet & Geffner (2001), and Refanidis & Vlahavas (2001)), may be more amenable to our approach.

Our approach is similar in spirit to work by Friedman & Weld (1996), regarding least commitment action selection in partial-order planning. The idea presented there is to support open sub-goals by abstract actions, which represent sets of physical actions whose common effects satisfy the sub-goal, and only later refine them to concrete actions. This can lead to exponential speed-ups.

Perhaps the most recent and most closely related work is by Cushing & Bryce (2005). Cushing & Bryce consider the problem of spanning planning graphs, as it is done by many state-of-the-art heuristic search based planners, in order to compute heuristic values. To speed up the repeated generation of this graph for several states, referred to as "source states", they also exploit similarities between these states. Their empirical results compare to ours: while degrading performance is possible, on difficult problems often significant speed-ups can be realized. Besides the different application to plan graph generation rather than planning itself, their approach also contrasts with ours by their limitation to reachability analysis. In particular, it does not allow to reason about action costs or similar metric functions, while our approach works with any metric function that can be expressed as a regressable formula. Also, the use of progression commits the planning graph to a fixed set of source states, while with regression, as used in our approach, it is technically possible to add and remove states at a later time.

Nebel, Dimopoulos, & Koehler (1997) propose a method for a-priori detecting features of a planning domain which are irrelevant and can be ignored in planning, allowing for great speed-ups. Our approach is similar in that irrelevant features do not show up in the regression of the relevant formulae and thus any set of states only differing on these irrelevant features could be treated jointly. However, our approach only exploits this for the chosen set of start situations, while Nebel, Dimopoulos, & Koehler are able to benefit from such irrelevance at any level of the search tree.

Finally, our approach differs from bi-directional search (e.g. (Rosenschein 1981)), which also combines forward search and regression, as we do not search backwards from the goal, but regress over a given action sequence, explored during forward search.

Acknowledgments I would like to thank Jorge Baier for a fruitful discussion and helpful suggestions regarding the algorithm, and the anonymous reviewers for their constructive comments. I gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada.

References

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

Boutilier, C.; Reiter, R.; and Price, B. 2001. Symbolic dynamic programming for first-order MDPs. In *Proc. of the 17th International Joint Conference on Artificial Intelligence*, 690–700.

Cushing, W., and Bryce, D. 2005. State agnostic planning graphs. In *Proceedings of the 20th National Conference on Artificial Intelligence*, 1131–1138.

Edelkamp, S. 2001. Planning with pattern databases. In *Proceedings of the 6th European Conference on Planning*, 13–24.

Friedman, M., and Weld, D. S. 1996. Least-commitment action selection. In *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems*, 86–93.

Lin, F. 1999. Search algorithms in the situation calculus. *Logical Foundations of Cognitive Agents: Contributions in Honor of Ray Reiter* 213–233.

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

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In *Proceeding of the 4th European Conference on Planning*, 338–350.

Pednault, E. 1989. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceeding of the 6th International Conference on Principles of Knowledge Representation and Reasoning*, 324–332.

Refanidis, I., and Vlahavas, I. 2001. The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research* 15:115–161.

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

Rosenschein, S. J. 1981. Plan synthesis: A logical perspective. In *Proc. of the 7th International Joint Conference on Artificial Intelligence*, 331–337.