Heuristic Search-Based Replanning

Sven Koenig **David Furcy** **Colin Bauer**
College of Computing
Georgia Institute of Technology
Atlanta, GA 30312-0280
{skoenig, dfurcy, colinos}@cc.gatech.edu

Abstract

Many real-world planning problems require one to solve a series of similar planning tasks. In this case, replanning can be much faster than planning from scratch. In this paper, we introduce a novel replanning method for symbolic planning with heuristic search-based planners, currently the most popular planners. Our SHERPA replanner is not only the first heuristic search-based replanner but, different from previous replanners for other planning paradigms, it also guarantees that the quality of its plans is as good as that achieved by planning from scratch. We provide an experimental feasibility study that demonstrates the promise of SHERPA for heuristic search-based replanning.

Introduction

Artificial intelligence has investigated planning techniques that allow one to solve planning tasks in large domains. Most of the research on planning has studied how to solve one-shot planning problems. However, planning is often a repetitive process, where one needs to solve a series of similar planning tasks, for example, because the actual situation turns out to be slightly different from the one initially assumed or because the situation changes over time. In these cases, the original plan might no longer apply or might no longer be good and one thus needs to replan for the new situation (desJardins *et al.* 1999). Examples of practical significance include the aeromedical evacuation of injured people in crisis situations (Kott, Saks, & Mercer 1999) and air campaign planning (Myers 1999). Similarly, one needs to solve a series of similar planning tasks if one wants to perform a series of what-if analyses or if the cost of operators, their preconditions, or their effects changes over time because they are learned or refined (Wang 1996).

Most planners solve each planning task from scratch when they solve a series of similar planning tasks. Since planning is time-consuming, this severely limits their responsiveness or the number of what-if analyses that they can perform. To enhance their performance, we are developing replanning methods that reuse information from previous planning episodes to solve a series of similar planning tasks much faster than is possible by solving each planning task from scratch. In particular, we present our Lifelong Planning A* (LPA*), an incremental version of A*, and demonstrate how it can be extended to heuristic search-based replanning, resulting in the SHERPA replanner (Speedy HEuristic search-based RePlAnner). Replanning methods have, of course, been studied before. Examples include case-based planning, planning by analogy, plan adaptation, transformational planning, planning by solution replay, repair-based planning, and learning search-control knowledge. These methods have been used as part of systems such as CHEF (Hammond 1990), GORDIUS (Simmons 1988), LS-ADJUST-PLAN (Gerevini & Serina 2000), MRL (Koehler 1994), NoLimit (Veloso 1994), PLEXUS (Alterman 1988), PRIAR (Kambhampati & Hendler 1992), and SPA (Hanks & Weld 1995). One difference between SHERPA and the other replanners is that SHERPA is so far the only replanner for the currently most popular planning paradigm, namely heuristic search-based planning. A second difference between SHERPA and the other replanners is that SHERPA does not only remember the previous plans but also the previous plan-construction processes. Thus, it has more information available for replanning than even PRIAR, that stores plans together with explanations of their correctness, or NoLimit, that stores plans together with substantial descriptions of the decisions that resulted in the solution. Finally, a third difference between SHERPA and the other replanners is that the quality of the plans of SHERPA is as good as that achieved by planning from scratch. This prevents SHERPA from separating replanning into two phases, as is often done by hierarchical or partial-order replanners, namely one phase that determines where the previous plan fails and one phase that uses slightly modified standard planning techniques to replan for those parts. Instead, SHERPA identifies quickly which parts of the previous plan-construction processes cannot be reused to construct the new plan and then uses an efficient specialized replanning method to plan for these parts.

Our LPA* has not been described before in the artificial intelligence planning literature. In the following, we therefore first present LPA* and illustrate its behavior on a simple gridworld problem. We originally developed it for route-planning tasks for mobile robots in both known and unknown terrain (Koenig & Likhachev 2001), including the route-planning tasks solved by Dynamic A* (Stentz 1995).

The main contribution of this paper is to extend LPA* to heuristic search-based replanning. We describe the resulting SHERPA replanner and then present an experimental feasibility study that demonstrates for three randomly chosen domains from previous AIPS planning competitions that SHERPA can replan much faster than planning from scratch and provides some insight into when SHERPA works especially well.

Lifelong Planning A*

Our Lifelong Planning A* (LPA*) is a search method that repeatedly determines shortest paths between two given vertices as the edge costs of a graph change. The term lifelong planning, in analogy to lifelong learning (Thrun 1998), refers to LPA*'s reusing information from previous plan-construction processes to avoid the parts of the new plan-construction process that are identical to the previous ones.

Notation and Definitions

Our LPA* solves the following search tasks: It applies to finite graph search problems on known graphs whose edge costs can increase or decrease over time. S denotes the finite set of vertices of the graph. $succ(s) \subseteq S$ denotes the set of successors of vertex $s \in S$. Similarly, $pred(s) \subseteq S$ denotes the set of predecessors of vertex $s \in S$. $0 < c(s, s') \leq \infty$ denotes the cost of moving from vertex s to vertex $s' \in succ(s)$. LPA* always determines a shortest path from a given start vertex $s_{start} \in S$ to a given goal vertex $s_{goal} \in S$, knowing both the topology of the graph and the current edge costs. Since LPA* always determines a shortest path, it also finds a plan for the changed planning task if one exists (that is, it is complete).

The Algorithm

Our LPA* is shown in Figure 1. It finds a shortest path by maintaining an estimate $g(s)$ of the start distance of each vertex s, that is, the distance from s_{start} to vertex s. It also maintains rhs-values, a second kind of estimates of the start distances. The rhs-values are one-step lookahead values based on the g-values and thus potentially better informed than the g-values. They always satisfy the following relationship:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start} \\ \min_{s' \in pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases}$$

A vertex is called locally consistent iff its g-value equals its rhs-value. This concept is important because the g-values of all vertices equal their start distances iff all vertices are locally consistent. However, LPA* does not make every vertex locally consistent after some of the edge costs have changed. First, LPA* uses heuristics to focus the search and determine that some start distances need not be computed at all because they are irrelevant for recalculating a shortest path from s_{start} to s_{goal} (heuristic search), similar to A* (Pearl 1985). The heuristics $h(s)$ estimate the goal distance of vertex s and need to be consistent, that is, obey the triangle inequality. Second, LPA* does not recompute the start distances that have been computed before and have not changed

The pseudocode uses the following functions to manage the priority queue: U.TopKey() returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey() returns $[\infty; \infty]$.) U.Pop() deletes the vertex with the smallest priority in priority queue U and returns the vertex. U.Insert(s, k) inserts vertex s into priority queue U with priority k. Finally, U.Remove(s) removes vertex s from priority queue U.

```
procedure CalculateKey($s$)
{01} return $[\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))]$;
procedure Initialize()
{02} $U = \emptyset$;
{03} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{04} $rhs(s_{start}) = 0$;
{05} U.Insert($s_{start}$, CalculateKey($s_{start}$));
procedure UpdateVertex($u$)
{06} if ($u \neq s_{start}$) $rhs(u) = \min_{s' \in pred(u)}(g(s') + c(s', u))$;
{07} if ($u \in U$) U.Remove($u$);
{08} if ($g(u) \neq rhs(u)$) U.Insert($u$, CalculateKey($u$));
procedure ComputeShortestPath()
{09} while (U.TopKey() $\dot{<}$ CalculateKey($s_{goal}$) OR $rhs(s_{goal}) \neq g(s_{goal})$)
{10}   $u$ = U.Pop();
{11}   if ($g(u) > rhs(u)$)
{12}     $g(u) = rhs(u)$;
{13}     for all $s \in succ(u)$ UpdateVertex($s$);
{14}   else
{15}     $g(u) = \infty$;
{16}     for all $s \in succ(u) \cup \{u\}$ UpdateVertex($s$);
procedure Main()
{17} Initialize();
{18} forever
{19}   ComputeShortestPath();
{20}   Wait for changes in edge costs;
{21}   for all directed edges $(u, v)$ with changed edge costs
{22}     Update the edge cost $c(u, v)$;
{23}     UpdateVertex($v$);
```

Figure 1: Lifelong Planning A*

(incremental search), similar to DynamicSWSF-FP (Ramalingam & Reps 1996). LPA* thus recalculates only very few start distances.

LPA* applies to the same graph search problems as A*. The initial search of LPA* even calculates the g-values of each vertex in exactly the same order as A*, provided that A* breaks ties between vertices with the same f-values suitably. In general, both search methods maintain a priority queue. The priority queue of LPA* always contains exactly the locally inconsistent vertices. The priority of vertex s in the priority queue is always

$$k(s) = [k_1(s); k_2(s)],$$

a vector with two components where $k_1(s) = \min(g(s), rhs(s)) + h(s)$ and $k_2(s) = \min(g(s), rhs(s))$. The priorities are compared according to a lexicographic ordering. For example, priority $k(s)$ is smaller than or equal to priority $k'(s)$, denoted by $k(s) \dot{\leq} k'(s)$, iff either $k_1(s) < k'_1(s)$ or ($k_1(s) = k'_1(s)$ and $k_2(s) \leq k'_2(s)$). LPA* recalculates the g-values of vertices s ("expands the vertices") by executing lines {10-16}) in the priority queue in the order of increasing first priority components, which correspond to the f-values of an A* search, and vertices with equal first priority components in order of increasing second priority components, which correspond to the g-values of an A* search. Thus, it expands vertices in a similar order to an A* search that expands vertices in the order of increasing f-values and vertices with equal f-values that are on the same branch of the search tree in order of increasing g-values.

If $g(s_{goal}) = \infty$ after the search, then there is no finite-cost path from s_{start} to s_{goal}. Otherwise, one can trace back a shortest path from s_{start} to s_{goal} by always moving from the

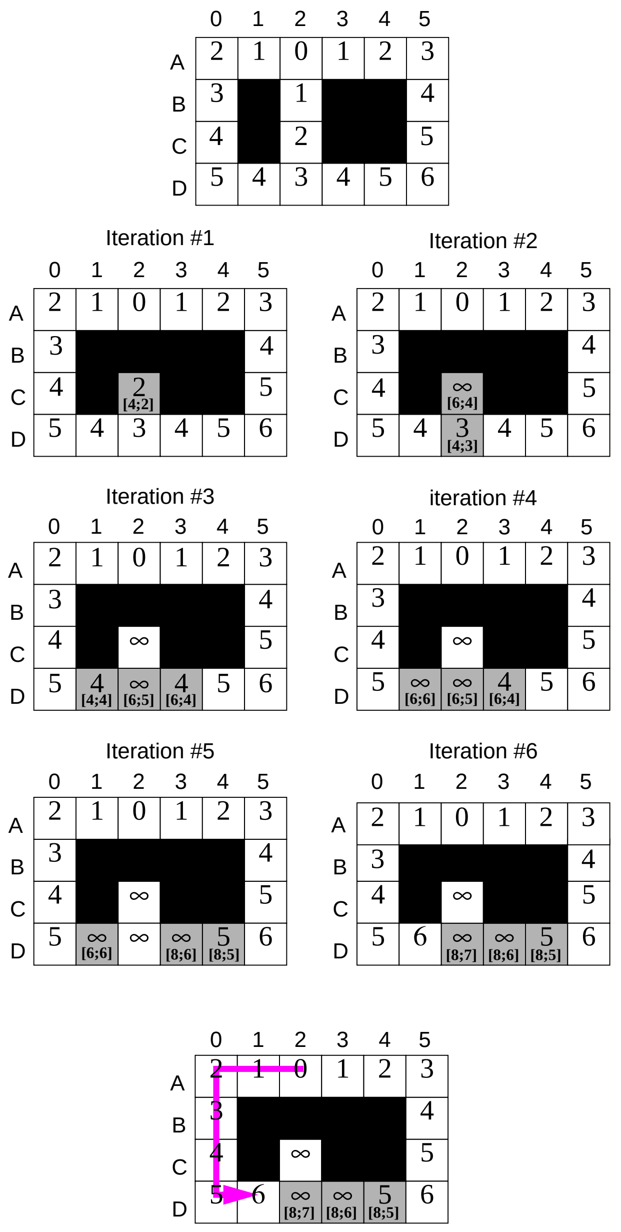

Figure 2: An Example

current vertex s, starting at s_{goal}, to any predecessor s' that minimizes $g(s') + c(s', s)$ until s_{start} is reached (ties can be broken arbitrarily).

A more detailed and formal description of LPA* , its comparison to A*, and proofs of its correctness, completeness, and other properties can be found in (Likhachev & Koenig 2001).

An Example

To describe the idea behind our LPA* and its operation, we use a path-planning example in a known four-connected gridworld with cells whose traversability can change over time, see Figure 2. They are either traversable or blocked. LPA* always determines a shortest path from cell A2 to cell D1. This is a special case of a graph search problem on four-connected grids whose edge costs are either one or infinity. As an approximation of the distance between two cells, we use their Manhattan distance.

Assume that one is given the g-values in the top gridworld and it is claimed that they are equal to the correct start distances. One way to verify this is to check that all cells are locally consistent, that is, that their given g-values are equal to their rhs-values (one-step lookahead values), which is indeed the case. Now assume that cell B2 becomes blocked, as shown in the gridworld of iteration one, and it is claimed that the g-values in the cells remain equal to the correct start distances. Since the g-values remain unchanged, each g-value continues to be equal to the corresponding rhs-value unless the rhs-value has changed which is only possible if the neighbors of the corresponding cell have changed. Thus, one needs to check only whether the cells close to changes in the gridworld remain locally consistent, that is, cells A2 and C2. It turns out that cell C2 is now locally inconsistent and thus that not all g-values are equal to the correct start distances. To change this, one needs to work on the locally inconsistent cells since all cells need to be locally consistent in order for all g-values to be equal to the correct start distances. Locally inconsistent cells thus provide a starting point for replanning.

Iterations 1-6 trace the behavior of LPA* if ComputeShortestPath() is called in this situation. Each gridworld shows the g-values at the beginning of an iteration, that is, when ComputeShortestPath() executes Line 9. At every point in time, exactly the locally inconsistent cells are in the priority queue. In the gridworlds, these cells are shaded and their priorities are given below their g-values. LPA* always removes the cell with the smallest priority from the priority queue. If the g-value of the cell is larger than its rhs-value, LPA* sets the g-value of the cell to its rhs-value. Otherwise, LPA* sets the g-value to infinity. LPA* then recalculates the rhs-values of the cells potentially affected by this assignment, checks whether the cells become locally consistent or inconsistent, and (if necessary) removes them from or adds them to the priority queue. It then repeats this process until it has found a shortest path from the start cell to the goal cell, which requires it to recalculate some start distances but not all of them. One can then trace back a shortest path from the start cell to the goal cell by starting at the goal cell and always greedily decreasing the start distance. Any way of doing this results in a shortest path from the start cell to the goal cell. Since all costs are one, this means moving from D1 (6) via D0 (5), C0 (4), B0 (3), A0 (2), and A1 (1) to A2 (0), as shown in the bottom gridworld.

Note that the blockage of cell B2 changes only five start distances, namely the ones of C2, D1, D2, D3, and D4. This allows LPA* to replan a shortest path efficiently even though the shortest path from the start cell to the goal cell changed completely. This is an advantage of reusing parts of previous plan-construction processes (in form of the g-values) rather than adapting previous plans. In particular, the g-values can easily be used to determine a shortest path but are more easily reusable than the shortest paths themselves. The number

The pseudocode uses the following functions to manage the priority queue: U.TopKey() returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey() returns $[\infty; \infty]$.) U.Pop() deletes the vertex with the smallest priority in priority queue U and returns the vertex. U.Insert(s, k) inserts vertex s into priority queue U with priority k. Finally, U.Remove(s) removes vertex s from priority queue U.

The pseudocode assumes that s_{start} does not satisfy the goal condition (otherwise the empty plan is optimal). Furthermore, s_{goal} is a special symbol that does not correspond to any state.

```
procedure CalculateKey($s$)
{01} return $[\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))]$;
procedure Initialize()
{02} $rhs(s_{start}) = 0$;
{03} $g(s_{start}) = \infty$;
{04} $h(s_{start}) =$ the heuristic value of $s_{start}$;
{05} $pred(s_{start}) = succ(s_{start}) = \emptyset$;
{06} $operators = \emptyset$;
{07} $U = \emptyset$;
{08} U.Insert($s_{start}$, CalculateKey($s_{start}$));
procedure UpdateVertex($u$)
{09} if ($u \neq s_{start}$) then $rhs(u) = \min_{e \in pred(u)}(g(source(e)) + cost(e))$;
{10} if ($u \in U$) then U.Remove($u$);
{11} if ($g(u) \neq rhs(u)$) then U.Insert($u$, CalculateKey($u$));
procedure ComputeShortestPath()
{12} while (U.TopKey() $\dot{<}$ CalculateKey($s_{goal}$) OR $rhs(s_{goal}) \neq g(s_{goal})$)
{13}   $u =$ U.Pop();
{14}   if ($u$ is expanded for the first time AND $u \neq s_{goal}$) then
{15}     for all ground planning operators $o$ whose preconditions are satisfied in $u$:
{16}       if ($o \notin operators$) then
{17}         $operators = operators \cup \{o\}$;
{18}         $edges(o) = \emptyset$;
{19}       $s =$ the vertex that results from applying $o$;
{20}       if (vertex $s$ satisfies the goal condition) then $s = s_{goal}$;
{21}       if ($s$ is encountered for the first time) then
{22}         $rhs(s) = g(s) = \infty$;
{23}         $h(s) =$ the heuristic value of $s$;
{24}         $pred(s) = succ(s) = \emptyset$;
{25}       Create a new edge $e$;
{26}       $source(e) = u$;
{27}       $destination(e) = s$;
{28}       $cost(e) =$ the cost of applying $o$;
{29}       $edges(o) = edges(o) \cup \{e\}$;
{30}       $pred(s) = pred(s) \cup \{e\}$;
{31}       $succ(u) = succ(u) \cup \{e\}$;
{32}   if ($g(u) > rhs(u)$) then
{33}     $g(u) = rhs(u)$;
{34}     for all $e \in succ(u)$: UpdateVertex($destination(e)$);
{35}   else
{36}     $g(u) = \infty$;
{37}     UpdateVertex($u$);
{38}     for all $e \in succ(u)$ with $destination(e) \neq u$: UpdateVertex($destination(e)$);
procedure Main()
{39} Initialize();
{40} forever
{41}   ComputeShortestPath();
{42}   Wait for changes in operator costs;
{43}   for all ground planning operators $o \in operators$ with changed operator costs:
{44}     for all $e \in edges(o)$:
{45}       $cost(e) =$ the (new) cost of applying $o$;
{46}       UpdateVertex($destination(e)$);
```

Figure 3: The SHERPA Replanner

of cells in our example is too small to result in a large advantage over an A* search from scratch but larger gridworlds result in substantial savings (Koenig & Likhachev 2001).

The SHERPA Replanner

Heuristic search-based planners were introduced by (McDermott 1996) and (Bonet, Loerincs, & Geffner 1997) and have become very popular since then. For example, several heuristic search-based planners entered the AIPS-2000 planning competition and exhibited very good performance, including HSP 2.0 (Bonet & Geffner 2000), GRT (Refanidis & Vlahavas 1999), FF (Hoffmann 2000), and AltAlt (Srivastava *et al.* 2000). In the following, we show how to extend our LPA* to symbolic planning with heuristic search-based planners that perform a forward search using A* with consistent heuristics, resulting in the SHERPA replanner (Speedy HEuristic search-based RePlAnner). SHERPA applies to replanning tasks where edges or states are added or deleted, or the costs of edges are changed, for example, because the cost of operators, their preconditions, or their effects change from one planning task to the next. With a small change, SHERPA can also be used if the goal description or the heuristic change.

Unfortunately, LPA* cannot be used completely unchanged for heuristic search-based replanning. There are three issues that need to be addressed.

- First, the pseudocode shown in Figure 1 initializes all states of the state space up front. This is impossible for symbolic planning since the state space is too large to fit in memory. We address this issue by initializing states and operators only when they are encountered during the search.
- Second, the pseudocode iterates over all predecessors of a state to determine the rhs-value of the state on Line 6 of Figure 1. However, it is difficult to determine the predecessors of states for symbolic planning. We address this issue as follows: Whenever a state is expanded, SHERPA generates all of its successors and for each of them remembers that the expanded state is one of its predecessors. Thus, at any point in time, SHERPA has those predecessors of a state available that have been expanded at least once already and thus have potentially finite g-values. We then change the pseudocode to iterate only over the cached predecessors of the state (instead of all of them) when it calculates the rhs-value of the state. This does not change the calculated rhs-value since the g-values of the other predecessors are infinite.
- Third, the pseudocode assumes that there is only one goal state. However, there are often many goal states in symbolic planning if the goal is only partially specified. We address this issue by merging all states that satisfy the goal condition into one new state, called s_{goal}, and then removing their successors.

Experimental Results

In the worst case, replanning is no more efficient than planning from scratch (Nebel & Koehler 1995). This can happen in the context of SHERPA if the overlap of the old and new A* search trees is minimal, for example, if the cost of an edge close to the root of the old search tree increases and there is no suitable detour that can be used instead. In many situations, however, SHERPA can exploit the fact that large parts of the plan-construction processes are identical in order to decrease its planning effort significantly over planning from scratch. In the following, we provide an experimental feasibility study that shows that SHERPA indeed reduces the planning effort substantially compared to planning from scratch. Our experiments assume that the planning task changes slightly over time. Thus, SHERPA can use the results of the previous planning task for replanning.

Replanners are commonly evaluated using the savings percentage. If x and y denote the computational effort of replanning and planning from scratch, respectively, then the

Domains		Deleted Edges (%)			Sample	Average Savings
		minimum	maximum	average	Size	Percentage
blocksworld	(3 blocks)	5.3	25.0	7.5	348	6.3
blocksworld	(4 blocks)	1.3	25.0	3.9	429	22.9
blocksworld	(5 blocks)	0.4	10.0	2.1	457	26.4
blocksworld	(6 blocks)	0.2	4.5	1.2	471	31.1
blocksworld	(7 blocks)	0.1	2.7	0.7	486	38.0
gripper	(3 balls)	1.2	22.4	8.2	340	47.5
gripper	(4 balls)	0.8	21.7	7.2	349	57.0
gripper	(5 balls)	0.6	21.8	5.8	367	65.1
gripper	(6 balls)	0.5	21.8	5.6	361	69.4
gripper	(7 balls)	0.5	21.9	5.2	358	73.4
gripper	(8 balls)	0.3	22.0	4.6	368	81.0
gripper	(9 balls)	0.3	21.8	4.3	374	77.7
gripper	(10 balls)	0.2	21.6	4.5	356	80.0
miconic	(5 floors, 1 person)	1.8	11.1	3.5	229	16.3
miconic	(5 floors, 2 people)	1.7	7.0	3.5	217	51.4
miconic	(5 floors, 3 people)	1.7	5.3	3.4	166	46.3
miconic	(5 floors, 4 people)	1.7	4.9	3.2	162	63.1
miconic	(5 floors, 5 people)	1.6	4.4	2.9	158	74.4
miconic	(5 floors, 6 people)	1.5	4.2	2.8	159	80.4
miconic	(5 floors, 7 people)	1.5	3.9	2.6	119	85.2

Table 1: Speedup of SHERPA over Repeated A* Searches

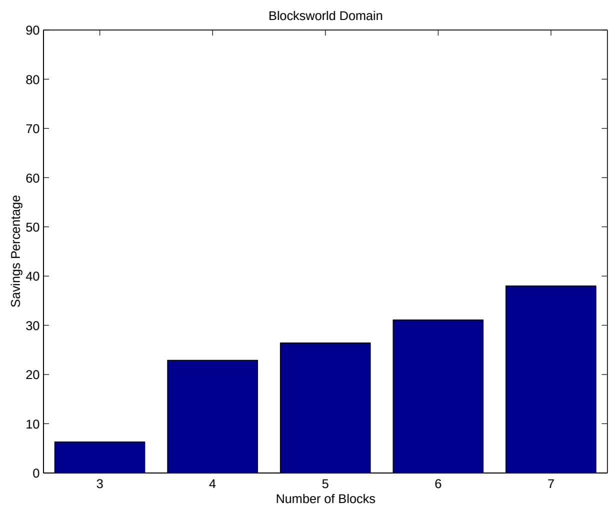

Figure 4: Blocksworld: Average Savings Percentage as a Function of Domain Size

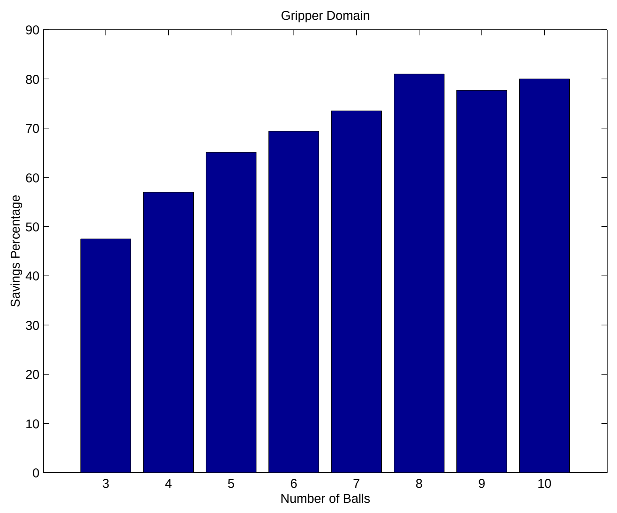

Figure 5: Gripper: Average Savings Percentage as a Function of Domain Size

savings percentage is defined to be $100(y - x)/y$ (Hanks & Weld 1995). Consequently, we use the savings percentage to evaluate SHERPA, which means that we evaluate SHERPA relative to its own behavior in generating plans from scratch or, equivalently, relative to an A* search with the same heuristic and tie-breaking behavior. When calculating the savings percentage, we use the number of vertex expansions (that is, updates of the g-values) to measure the computational effort of SHERPA. This is justified because the number of vertex expansions heavily influences the runtime of SHERPA. We do not use the runtime directly because it is implementation and machine dependent and thus makes it difficult for others to reproduce the results of our performance comparison. We count two vertex expansions if SHERPA expands the same vertex twice when it performs an incremental search, to avoid biasing our experimental results in favor of replanning.

We used the code of HSP 2.0 (Bonet & Geffner 2000) to implement SHERPA and performed our experiments with the consistent h_{max}-heuristic. We used three randomly chosen domains from previous AIPS planning competitions, namely the blocksworld, gripper, and miconic (elevator) domains of different sizes. In each of these domains, we repeated the following procedure 500 times. We randomly generated a start state and goal description, and used SHERPA to solve this original planning task. We then ran-

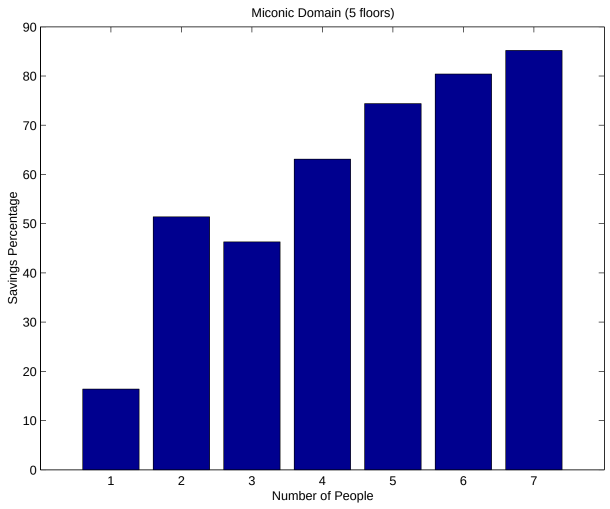

Figure 6: Miconic: Average Savings Percentage as a Function of Domain Size

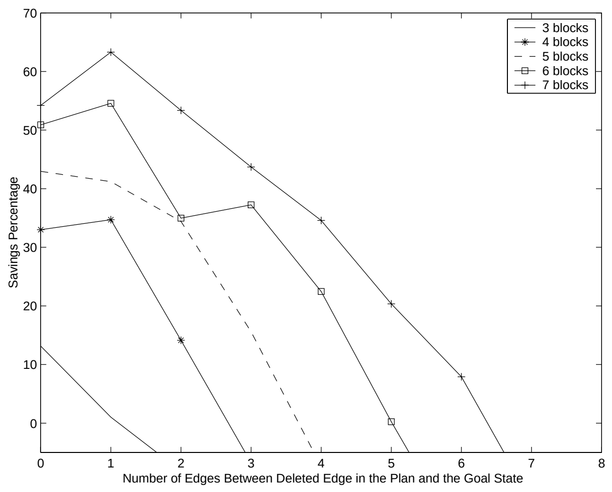

Figure 7: Blocksworld: Average Savings Percentage as a Function of Distance of the Deleted Edge from the Goal

domly selected one of the ground planning operators that were part of the returned plan and deleted it from the planning domain. Thus, the old plan can no longer be executed and replanning is necessary. Deleting a ground planning operator deletes several edges from the state space and thus changes the state space substantially. We then used SHERPA twice to solve the resulting modified planning task: one time it replanned using the results from the original planning task and the other time it planned from scratch. Since the h_{max}-heuristic depends on the available operators, we decided to let SHERPA continue to use the heuristic for the original planning task when it solved the modified one because this enables SHERPA to cache the heuristic values. Caching the heuristic values benefits replanning and planning from scratch equally since computing the heuristics is very time-consuming. No matter whether SHERPA replanned or planned from scratch, it always found the same plans for the modified planning tasks and the plans were optimal, thus verifying our theoretical results about LPA*. We tabulated the number of vertex expansions, together with how many edges were deleted from the state space and whether this made the resulting planning task unsolvable. Table 1 shows the results, averaged over all cases where the resulting planning tasks were solvable and thus the original plan-construction process could indeed be reused. The table reports the percentage of edges deleted from the state spaces, the number of modified planning tasks that are solvable, and the savings percentages averaged over them. Since the state spaces are large, we approximated the percentage of edges deleted from the state space with the percentage of edges deleted from the cached part of the state space. We used a paired-sample z test at the one-percent significance level to confirm that replanning with SHERPA indeed outperforms planning from scratch significantly.

In the following, we interpret the collected data to gain some insight into the behavior of SHERPA.

- Figures 4, 5 and 6 show that the savings percentages tend to increase with the size of the three domains. (Figures 7 and 8 show the same trend.) This is a desirable property of replanning methods since planning is time-consuming in large domains and the large savings provided by replanning are therefore especially important. The savings percentages in the gripper domain appear to level off at about eighty percent, which is similar to the savings percentages that (Hanks & Weld 1995) report for PRIAR and better than the savings percentages that they report for SPA. The savings percentages in the other two domains seem to level off only for domain sizes larger than what we used in the experiments but also reach levels of eighty percent at least in the miconic domain.
- Figure 7 shows how the savings percentages for the blocksworld domain change with the position of the deleted ground planning operator in the plan for the original planning task. Note that the savings percentages become less reliable as the distance of the deleted ground planning operator to the goal increases. This is so because the number of shortest plans in the sample with length larger than or equal to n quickly decreases as n increases. The savings percentages decrease as the distance of the deleted ground planning operator to the end of the plan increases. They even become negative when the deleted ground planning operator is too close to the beginning of the plan, as expected, since this tends to make the old and new search trees very different.
- Figure 8 shows that the savings percentages for the blocksworld domains degrade gracefully as the similarity of the original and modified planning tasks decreases, measured using the number of ground planning operators deleted at the same time. In other words, SHERPA is able to reuse more of the previous plan-construction process the more similar the original and modified planning tasks are, as expected. We repeated the following procedure 500 times to generate the data: We randomly generated a start state and goal description, and solved the resulting

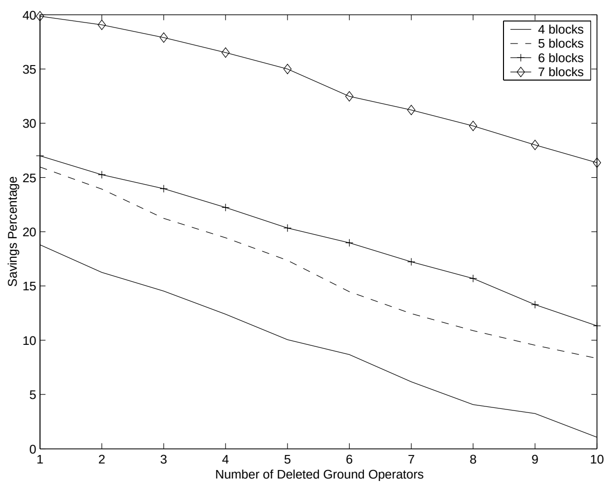

Figure 8: Blocksworld: Average Savings Percentage as a Function of Dissimilarity of the Planning Tasks

planning task from scratch using SHERPA. We call the resulting search graph G and the resulting plan P. We then generated a random sequence of 10 different ground operators. The first ground operator was constrained to be part of plan P to ensure the need for replanning. For each $n = 1 \dots 10$, we then deleted the first n ground operators in the sequence from the planning domain and used SHERPA to replan using search graph G. We discarded each of the 500 runs in which the planning task became unsolvable after all 10 ground operators had been deleted from the domain. Finally, we averaged the savings percentages over all remaining planning problems with the same number $n = 1 \dots 10$ of deleted ground operators.

We used this experimental setup in the blocksworld domain for each problem size ranging from 3 to 7 blocks. Note that we omitted the results for planning tasks with three blocks. Because its state space is so small, most planning tasks are unsolvable after 10 ground planning operators are deleted.

Future Work

We used small domains in our feasibility study to be able to average over a large number of runs. Furthermore, SHERPA finds optimal plans which also limited the size of the domains we could plan in. Since our feasibility study was so successful, we intend to test SHERPA with the admissible h^2-heuristic that is more informed than the h_{max}-heuristic. We also intend to scale it up to larger problems by adapting it to the way A* is used by HSP 2.0. HSP 2.0 demonstrates convincingly that A* is capable of planning in large domains by trading off plan quality and planning effort. We therefore intend the next version of SHERPA, like HSP 2.0, to weigh the h-values more than the g-values and use the inadmissible h_{add}-heuristic (Bonet & Geffner 2001). Future work also includes extending our experimental characterization of the strengths and weaknesses of SHERPA.

Conclusions

In this paper, we have described a fast replanning method for symbolic planning that resulted from extending our LPA* to heuristic search-based replanning, resulting in the SHERPA replanner. SHERPA applies to replanning tasks where edges or states are added or deleted, or the costs of edges are changed, for example, because the cost of operators, their preconditions, or their effects change from one planning task to the next. SHERPA is not only the first heuristic search-based replanner but, different from previous replanners for other planning paradigms, it also guarantees that the quality of its plans is as good as that achieved by planning from scratch. Our experimental results show that SHERPA indeed reduces the planning effort substantially compared to planning from scratch.

Acknowledgments

We thank Blai Bonet and Hector Geffner for making the code of HSP 2.0 available to us. Blai Bonet also patiently answered all of our questions about the code. We thank Sylvie Thiébaux and John Slaney for making their blocksworld planning task generator available to us. We thank Joerg Hoffmann for making his PDDL planning problem files available to us. We thank Maxim Likhachev for helping to develop LPA* and proving its correctness. Finally, we thank the anonymous reviewers for their suggestions, some of which we will address in future publications. The Intelligent Decision-Making Group is partly supported by NSF awards under contracts IIS-9984827, IIS-0098807, and ITR/AP-0113881 as well as an IBM faculty partnership award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations and agencies or the U.S. government.

References

Alterman, R. 1988. Adaptive planning. *Cognitive Science* 12(3):393–421.

Bonet, B., and Geffner, H. 2000. Heuristic search planner 2.0. *Artificial Intelligence Magazine* 22(3):77–80.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence – Special Issue on Heuristic Search* 129(1):5–33.

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism. In *Proceedings of the National Conference on Artificial Intelligence*, 714–719.

desJardins, M.; Durfee, E.; Ortiz, C.; and Wolverton, M. 1999. A survey of research in distributed, continual planning. *Artificial Intelligence Magazine* 20(4):13–22.

Gerevini, A., and Serina, I. 2000. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling*, 112–121.

Hammond, K. 1990. Explaining and repairing plans that fail. *Artificial Intelligence* 45:173–228.

Hanks, S., and Weld, D. 1995. A domain-independent algorithm for plan adaptation. *Journal of Artificial Intelligence Research* 2:319–360.

Hoffmann, J. 2000. FF: The fast-forward planning systems. *Artificial Intelligence Magazine* 22(3):57–62.

Kambhampati, S., and Hendler, J. 1992. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence* 55:193–258.

Koehler, J. 1994. Flexible plan reuse in a formal framework. In Bäckström, C., and Sandewall, E., eds., *Current Trends in AI Planning*. IOS Press. 171–184.

Koenig, S., and Likhachev, M. 2001. Incremental A*. In *Proceedings of the Neural Information Processing Systems*.

Kott, A.; Saks, V.; and Mercer, A. 1999. A new technique enables dynamic replanning and rescheduling of aeromedical evacuation. *Artificial Intelligence Magazine* 20(1):43–53.

Likhachev, M., and Koenig, S. 2001. Lifelong Planning A* and Dynamic A* Lite: The proofs. Technical report, College of Computing, Georgia Institute of Technology, Atlanta (Georgia).

McDermott, D. 1996. A heuristic estimator for means-ends analysis in planning. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling*, 142–149.

Myers, K. 1999. Cpef: A continuous planning and execution framework. *Artificial Intelligence Magazine* 20(4):63–69.

Nebel, B., and Koehler, J. 1995. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence* 76(1–2):427–454.

Pearl, J. 1985. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Ramalingam, G., and Reps, T. 1996. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21:267–305.

Refanidis, I., and Vlahavas, I. 1999. GRT: a domain-independent heuristic for STRIPS worlds based on greedy regression tables. In *Proceedings of the European Conference on Planning*, 346–358.

Simmons, R. 1988. A theory of debugging plans and interpretations. In *Proceedings of the National Conference on Artificial Intelligence*, 94–99.

Srivastava, B.; Nguyen, X.; Kambhampati, S.; Do, M.; Nambiar, U.; Nie, Z.; Niganda, R.; and Zimmerman, T. 2000. AltAlt: Combining graphplan and heuristic state search. *Artificial Intelligence Magazine* 22(3):88–90.

Stentz, A. 1995. The focussed D* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1652–1659.

Thrun, S. 1998. Lifelong learning algorithms. In Thrun, S., and Pratt, L., eds., *Learning To Learn*. Kluwer Academic Publishers.

Veloso, M. 1994. *Planning and Learning by Analogical Reasoning*. Springer.

Wang, X. 1996. *Learning Planning Operators by Observation and Practice*. Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh (Pennsylvania).