Available online at www.sciencedirect.com

SCIENCE DIRECT®

Artificial Intelligence 155 (2004) 93–146

Artificial Intelligence

www.elsevier.com/locate/artint

Lifelong Planning A*

Sven Koenig [a,*], Maxim Likhachev [b], David Furcy [c]

[a] *Computer Science Department, USC, Los Angeles, CA 90089, USA*
[b] *School of Computer Science, CMU, Pittsburgh, PA 15213, USA*
[c] *College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA*

Received 15 April 2002; received in revised form 9 October 2003

Abstract

Heuristic search methods promise to find shortest paths for path-planning problems faster than uninformed search methods. Incremental search methods, on the other hand, promise to find shortest paths for series of similar path-planning problems faster than is possible by solving each path-planning problem from scratch. In this article, we develop Lifelong Planning A* (LPA*), an incremental version of A* that combines ideas from the artificial intelligence and the algorithms literature. It repeatedly finds shortest paths from a given start vertex to a given goal vertex while the edge costs of a graph change or vertices are added or deleted. Its first search is the same as that of a version of A* that breaks ties in favor of vertices with smaller g-values but many of the subsequent searches are potentially faster because it reuses those parts of the previous search tree that are identical to the new one. We present analytical results that demonstrate its similarity to A* and experimental results that demonstrate its potential advantage in two different domains if the path-planning problems change only slightly and the changes are close to the goal.

Keywords: A*; Continual planning; Heuristic search; Heuristic search-based planning; Incremental search; Lifelong planning; Plan reuse; Replanning; Symbolic STRIPS-style planning

1. Overview

Artificial intelligence has investigated search methods that allow one to solve path-planning problems in large domains. Most of the research on search methods has studied how to solve one-shot path-planning problems. However, many artificial intelligence

* Corresponding author.
E-mail address: skoenig@usc.edu (S. Koenig).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2003.12.001

systems have to adapt their plans continuously to changes of the world or their models of the world. In these cases, the original plan might no longer apply or might no longer be good. In this case, one needs to replan for the new situation [1]. Examples of practical significance include the aeromedical evacuation of injured people in crisis situations [2] and air campaign planning [3]. Similarly, one needs to solve a series of similar path-planning problems if one wants to perform a series of what-if analyses or if the cost of planning operators, their preconditions, or their effects change over time because they are learned or refined. Consequently, search is often a repetitive process. In this situation, many artificial intelligence systems replan from scratch, that is, solve the path-planning problems independently. However, this can be inefficient in large domains with frequent changes and thus severely limits their responsiveness or the number of what-if analyses that they can perform, which is often unacceptable. This problem becomes even more severe when changes occur during planning. Fortunately, the changes to the path-planning problems are usually small. For example, planes might no longer be able to land on a particular airfield for the aeromedical evacuation example. This suggests that a complete recomputation of the best plan can be wasteful since some of the previous search results can be reused. This is what incremental search methods do. Notice that the terminology is unfortunately somewhat problematic since the term "incremental search" also refers to both on-line search and search with limited look-ahead [4].

Although incremental search methods are not widely used in artificial intelligence, different researchers have developed incremental versions of uninformed search methods, mostly in the algorithms literature. Incremental search methods, such as DynamicSWSF-FP [5], reuse information from previous searches to find shortest paths for series of similar path-planning problems potentially faster than is possible by solving each path-planning problem from scratch. Heuristic search methods, such as A* [6], on the other hand, are widely used in artificial intelligence. They use heuristic knowledge in the form of approximations of the goal distances to focus the search and find shortest paths for path-planning problems potentially faster than uninformed search methods.

In this article, we develop Lifelong Planning A* (LPA*), a replanning method that is an incremental version of A*.[1] We chose its name in analogy to "lifelong learning" [7] because it reuses information from previous searches. (Other researchers use the term continual planning for the same concept.) LPA* repeatedly finds shortest paths from a given start vertex to a given goal vertex in a given graph as edges or vertices are added or deleted or the costs of edges are changed, for example, because the cost of planning operators, their preconditions, or their effects change from one path-planning problem to the next. LPA* generalizes both DynamicSWSF-FP and A* and promises to find shortest paths faster than these two search methods individually because it combines their techniques. It is easy to understand, easy to analyze, and easy to optimize. Its first search is the same as that of a version of A* that breaks ties among vertices with the same f-value in favor of smaller g-values but the subsequent searches are potentially faster because it reuses those parts of the previous search tree that are identical to the new search tree, and

[1] The artificial intelligence planning literature actually distinguishes between replanning and plan reuse. Replanning attempts to retain as many plan steps of the previous plan as possible. Plan reuse does not have this requirement. Strictly speaking, LPA* is a plan reuse method rather than a replanning method.

uses an efficient method for identifying these parts. This can reduce the search time if large parts of the search trees are identical, for example, if the path-planning problems change only slightly and the changes are close to the goal. LPA* can also handle changes to the graph during its search and can be extended to inadmissible heuristics, more efficient tie-breaking criteria, and nondeterministic graphs [8].

In the following, we first describe the path-planning problems that LPA* solves. Second, we explain why it is possible to take advantage of information from previous searches. Third, we describe LPA* and how it takes advantage of this information, both in the abstract and for a concrete example. Fourth, we prove properties about its behavior, in particular its correctness, its close similarity to A*, its efficiency in terms of vertex expansions, and several other properties that help one to understand how it operates. Fifth, we explain how to optimize it. Finally, we evaluate it experimentally and apply it to both simple route planning and symbolic planning.

2. Notation

Lifelong Planning A* (LPA*) solves the following path-planning problems: It applies to path-planning problems on known finite graphs whose edge costs increase or decrease over time. (Such cost changes can also be used to model edges or vertices that are added or deleted.) 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 $s \in S$. $0 < c(s, s') \leqslant \infty$ 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. We use $g^*(s)$ to denote the start distance of vertex $s \in S$, that is, the cost of a shortest path from s_{start} to s. The start distances satisfy the following relationship:

$$g^*(s) = \begin{cases} 0 & \text{if } s = s_{start}, \\ \min_{s' \in pred(s)}(g^*(s') + c(s', s)) & \text{otherwise.} \end{cases} \quad (1)$$

To motivate and test LPA*, we use a special case of these search problems that is easy to visualize. We apply LPA* to route planning in known eight-connected gridworlds with cells whose traversability changes over time. They are either traversable (with cost one) or untraversable. LPA* always determines a shortest path between two given cells of the gridworld, knowing both the topology of the gridworld and which cells are currently untraversable. This is a special case of the path-planning problems on eight-connected gridworlds whose edge costs are either one or infinity. As an approximation of the distance between two cells, we use the maximum of the absolute differences of their x and y coordinates. These heuristics are for eight-connected gridworlds what Manhattan distances are for four-connected gridworlds.

3. Lifelong Planning A*: overview

Path-planning problems can be solved with traditional graph-search methods, such as breadth-first search, if they update the shortest path every time some edge costs change. They typically neither take advantage of available heuristics nor reuse information from previous searches. The following example, however, shows that taking advantage of these sources of information can potentially be beneficial individually and even more beneficial when they are combined.

Consider the gridworlds of size 15×20 shown in Fig. 1. The original gridworld is shown on top and the changed gridworld is shown at the bottom. We assume that one can squeeze through diagonal obstacles, which is simply an artifact of how we generated the underlying graphs from the gridworlds. The traversability of only a few cells has changed. In particular, three blocked cells became traversable (namely, A6, D2, and F5) and three traversable cells became blocked (namely, B1, C4, E3). Thus, two percent of the cells changed their status but the obstacle density remained the same. The figure shows the shortest paths in both cases. The shortest path changed since one cell (C4) on the original shortest path became blocked. The new shortest path is one step longer than the old one.

Once the start distances of all cells are known, one can easily trace back a shortest path from the start cell to the goal cell by always greedily decreasing the start distance, starting at the goal cell. This is similar to how A* traces the shortest path back from s_{goal} to s_{start} using the search tree it has constructed. Thus, we only need to determine the start distances. The start distances are shown in each traversable cell of the original and changed gridworlds. Those cells whose start distances in the changed gridworld have changed from the corresponding ones in the original gridworld are shaded gray.

We investigate two different ways of decreasing the search effort for determining the start distances for the changed gridworld.

- First, some start distances have not changed and thus need not be recomputed. This is what DynamicSWSF-FP [5] does. DynamicSWSF-FP, as originally stated, searches from the goal vertex to all other vertices and thus maintains estimates of the goal distances rather than the start distances. It is a simple matter of restating it to search from the start vertex to all other vertices. Also, to calculate a shortest path from the start vertex to the goal vertex not all distances need to be known even for uninformed search methods. To make DynamicSWSF-FP more efficient and thus avoid biasing our experimental results in favor of LPA*, we changed the termination condition of DynamicSWSF-FP so that it stops immediately after it is sure that it has found a shortest path from the start vertex to the goal vertex. The modified version of DynamicSWSF-FP is an incremental version of breadth-first search.
- Second, heuristic knowledge, in the form of approximations of the goal distances, can be used to focus the search and determine that some start distances need not be computed at all. This is what A* [6] does.

We demonstrate that the two ways of decreasing the search effort are orthogonal by developing LPA* that combines both of them and thus is potentially able to replan faster than either DynamicSWSF-FP or A*.

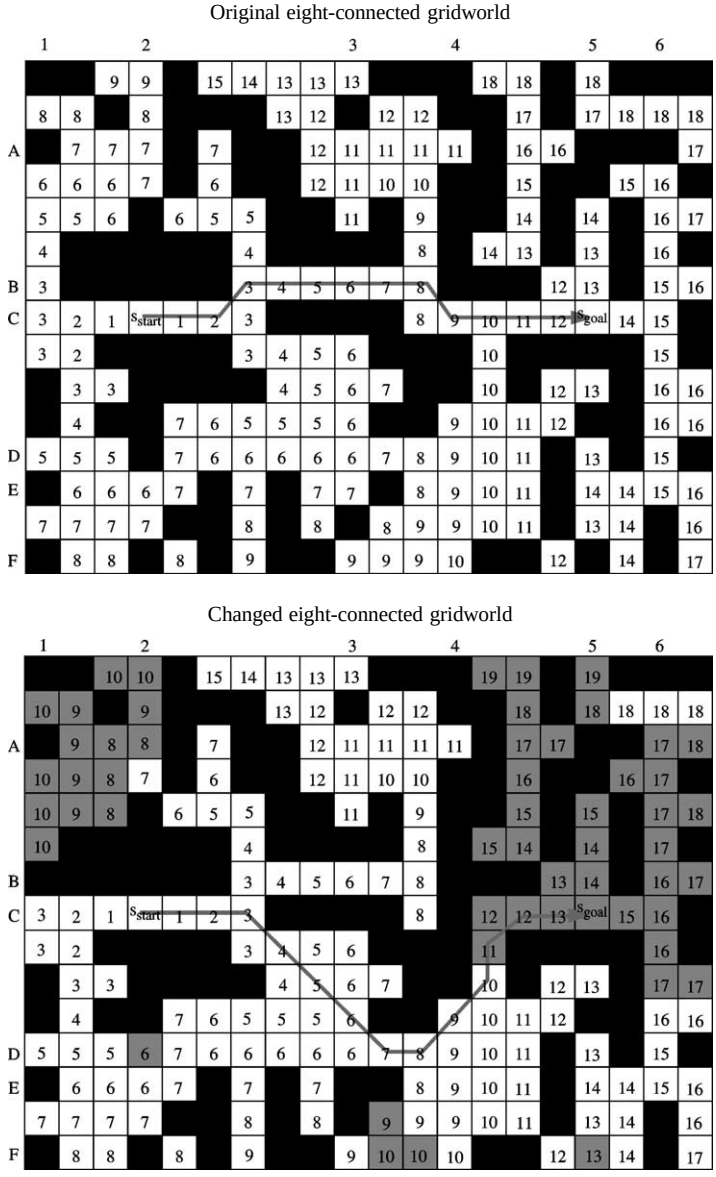

Fig. 1. Simple gridworld.

Fig. 2 shows in gray those cells whose start distances each of the four search methods recomputes. (To be precise: It shows in gray the cells that each of the four search methods expands.) During the search in the original gridworld, DynamicSWSF-FP computes the same start distances as breadth-first search during the first search and LPA* computes the same start distances as A*. (This is only guaranteed if the search methods break ties

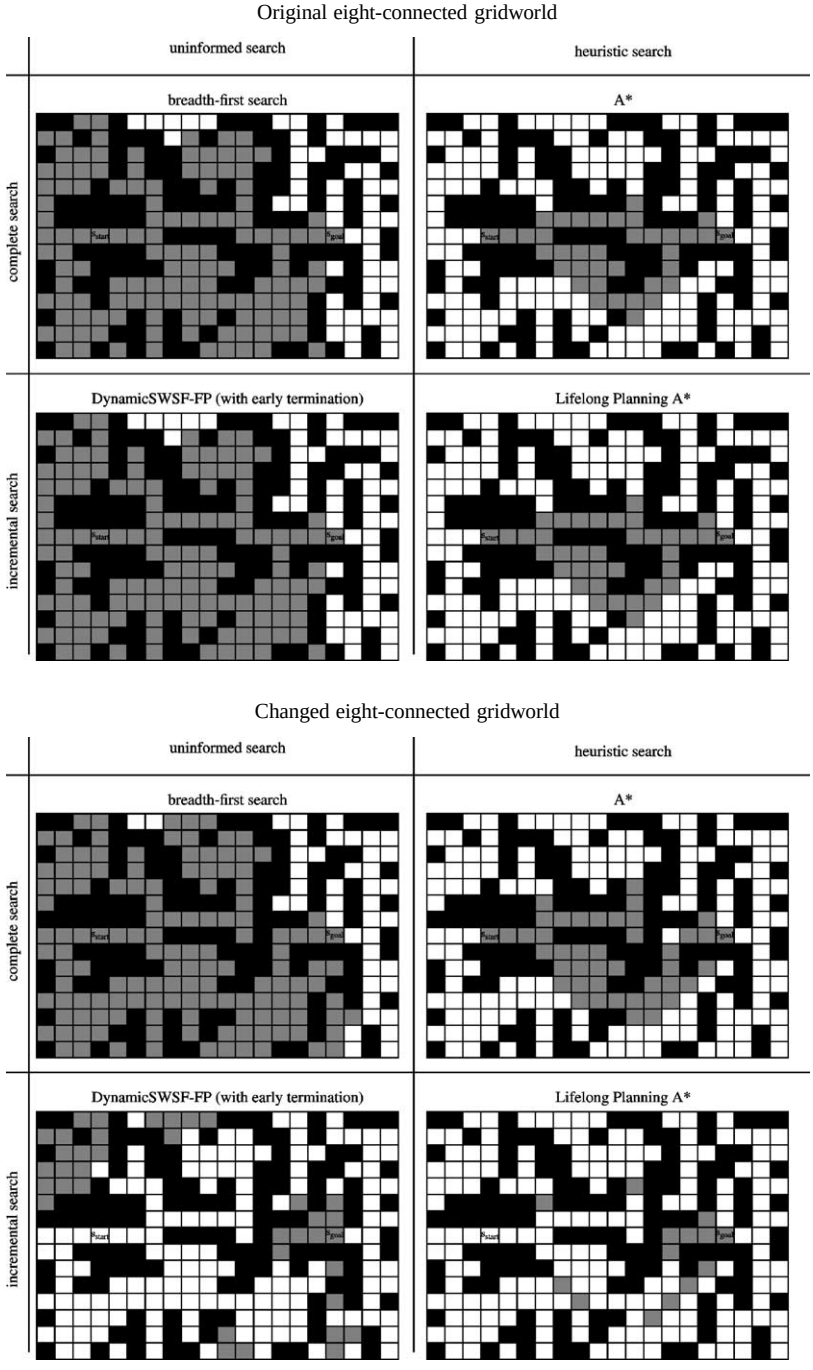

Fig. 2. Performance of search methods in the simple gridworld.

suitably.) During the search in the changed gridworld, however, both incremental search (DynamicSWSF-FP) and heuristic search (A*) individually decrease the number of start distances that need to be recomputed compared to breadth-first search, and together (LPA*) decrease this number even more. Note that LPA* updates only a subset of those start distances that are incorrect (either because they have changed or never been calculated). We will prove this property in the analytical section.

To illustrate the behavior of LPA*, we use the route-planning example in the eight-connected gridworld shown in Figs. 3–5. The cells are either traversable or blocked, and their traversability changes over time. LPA* always determines a shortest path from start

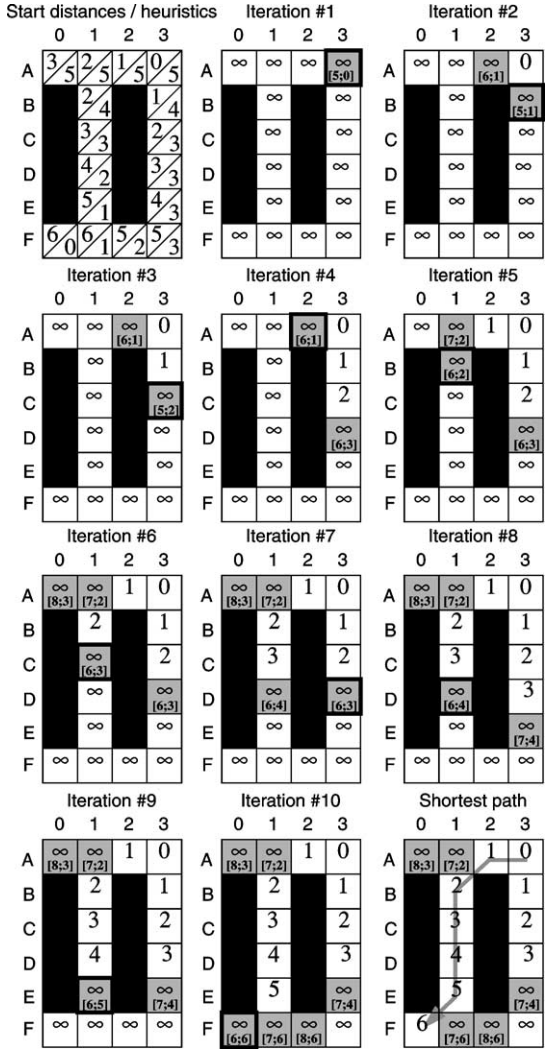

Fig. 3. An example: first search.

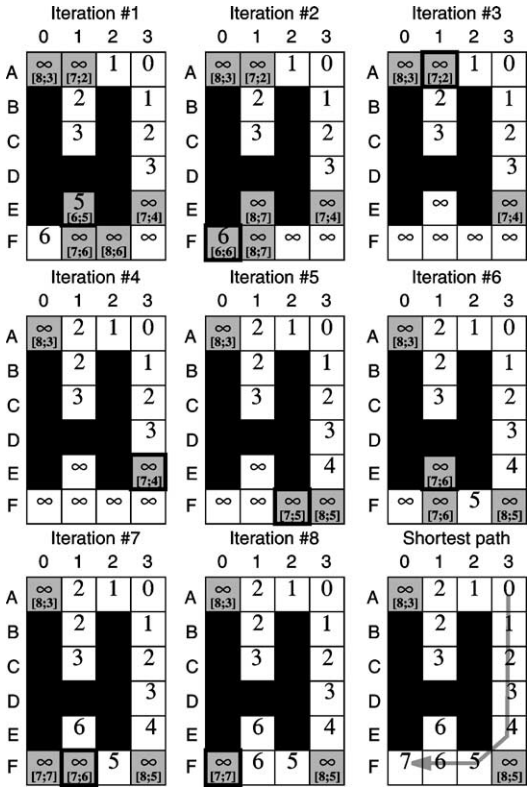

Fig. 4. An example: second search.

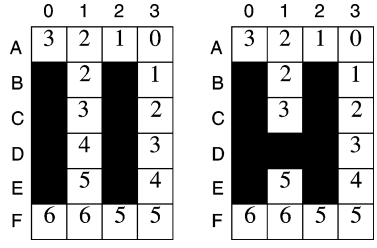

Fig. 5. An example: principle behind LPA*.

cell A3 to goal cell F0. The upper left gridworld in Fig. 3 shows the true start distances in the upper left corners of the cells and the heuristics in their lower right corners.

We first illustrate the main principle behind LPA*. LPA* maintains two estimates of the start distance of each cell, namely a g-value and an rhs-value. The g-values directly correspond to the g-values of an A* search. The rhs-values are one-step lookahead values based on the g-values and thus potentially better informed than the g-values. Their name comes from DynamicSWSF-FP where they are the values of the right-hand sides (rhs) of

grammar rules. The rhs-value of the start cell is zero. The rhs-value of any other cell is the minimum over all of its neighbors of the g-value of the neighbor and the cost of moving from the neighbor to the cell in question. Consider, for example, the g-values given in the left gridworld in Fig. 5. The rhs-value of cell A0 is three, namely the minimum of the g-value of cell A1 plus one and the g-value of cell B1 plus one. Thus, the g-value of cell A0 equals its rhs-value. We call such cells locally consistent. This concept is important because all g-values are equal to the respective start distances iff all cells are locally consistent.

Now assume that one is given the g-values in the left gridworld in Fig. 5, and it is claimed that they are equal to the start distances. There are at least two different approaches to verify this. One approach is to perform a complete search to determine the start distances and compare them to the g-values. Another approach is to check that all cells are locally consistent, that is, that their g-values are equal to their rhs-values, which is indeed the case. Thus, the g-values are indeed equal to the start distances. Both approaches need the same amount of time to confirm this. Now assume that cell D1 becomes blocked as shown in the right gridworld in Fig. 5, and it is claimed that the g-values in the cells remain equal to the start distances. Again, there are at least two different approaches to verify this. One approach is to perform again a complete search to determine the start distances and compare them to the g-values. The second approach is again to check that all cells are locally consistent. 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 blockage status of at least one neighbor of the corresponding cell has changed. Thus, one needs to check only whether the cells close to changes in the gridworld remain locally consistent, that is, cells C1 and E1. It turns out that cell C1 remains locally consistent (its g-value and rhs-value are both three) but cell E1 has become locally inconsistent (its g-value is five but its rhs-value is now six). Thus, not all g-values are equal to the start distances. (This does *not* mean that all g-values except the one of cell E1 are equal to the start distances.) Note that the second approach now needs less time than the first one. Furthermore, the second approach provides a starting point for replanning. One needs to work on the locally inconsistent cells since all cells need to be locally consistent in order for all g-values in the cells to be equal to the start distances. Locally inconsistent cells thus provide a starting point for replanning. However, LPA* does not make every cell locally consistent. Instead, it uses heuristics to focus its search and updates only the g-values that are relevant for computing a shortest path. This is the main principle behind LPA*.

Iterations 1–10 in Fig. 3 trace the behavior of the first search of LPA*. Each gridworld shows the g-values of the cells at the beginning of an iteration. LPA* maintains a priority queue that always contains exactly the locally inconsistent cells. These are the cells whose g-values LPA* potentially needs to update to make them locally consistent. The priorities of the cells in the priority queue are pairs that are compared according to a lexicographic ordering. The first component of the key roughly corresponds to the f-value used by A*, and the second component roughly corresponds to the g-value used by A*. Cells in the priority queue are shaded and their keys are given below their g-values. LPA* always recalculates the g-value of the cell ("expands the cell") with the smallest key in the priority queue (shown with a bold border in the figure). This is similar to A* that always expands the cell with the smallest f-value in the priority queue. The initial g-values are all infinity. LPA* always removes the cell with the smallest key 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 is sure that it has found a shortest path. LPA* expands the cells in the same order during the first search as an A* search that breaks ties among cells with the same f-value in favor of smaller g-values. 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 F0 (6) via E1 (5), D1 (4), C1 (3), B1 (2), and A2 (1) to A3 (0), as shown in the bottom right gridworld. Moving in the opposite direction then results in a shortest path from cell A3 to cell F0.

Now assume that cell D1 becomes blocked. Iterations 1–8 in Fig. 4 trace the behavior of the second search of LPA*. Note that the new blockage changes only three start distances, namely the ones of cells D1, E1, and F0. 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 the form of the g-values) rather than adapting previous plans, at the cost of larger memory requirements. In particular, not only can the g-values be used to determine a shortest path but they can also be more easily reused than the shortest paths themselves. The number of cells in our example is too small to result in a large advantage over an A* search but in the experimental section we will report more substantial savings in larger gridworlds.

4. Lifelong Planning A*: details

So far, we have given some intuition about how LPA* works. We now explain the details. LPA* is an incremental version of A* that applies to the same finite path-planning problems as A*. It shares with A* the fact that it uses nonnegative and consistent heuristics $h(s)$ [9] that approximate the goal distances of the vertices s to focus its search. Consistent heuristics obey the triangle inequality $h(s_{goal}) = 0$ and $h(s) \leqslant c(s, s') + h(s')$ for all vertices $s \in S$ and $s' \in succ(s)$ with $s \neq s_{goal}$. For example, the heuristics that we used in the context of the gridworlds, (namely the maximum of the absolute differences of the x and y coordinates of a cell and the goal cell) are consistent. LPA* reduces to a version of A* that breaks ties among vertices with the same f-value in favor of smaller g-values when LPA* is used to search from scratch and to a version of DynamicSWSF-FP that applies to path-planning problems and terminates earlier than the original version of DynamicSWSF-FP (as described above) when LPA* is used with uninformed (that is, zero) heuristics. These statements assume that A* and DynamicSWSF-FP break ties among vertices with the same f-values suitably.[2]

[2] To be precise: LPA* differs from DynamicSWSF-FP only in the calculation of the priorities for the vertices in the priority queue (line {01} in the pseudo code in Fig. 6) and the termination condition {09}. DynamicSWSF-

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 = ∅;
{03} for all s ∈ S rhs(s) = g(s) = ∞;
{04} rhs(s_start) = 0;
{05} U.Insert(s_start, [h(s_start); 0]);
procedure UpdateVertex(u)
{06} if (u ≠ s_start) rhs(u) = min_{s'∈pred(u)}(g(s') + c(s', u));
{07} if (u ∈ U) U.Remove(u);
{08} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{09} while (U.TopKey() ≺ CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal))
{10}    u = U.Pop();
{11}    if (g(u) > rhs(u))
{12}       g(u) = rhs(u);
{13}       for all s ∈ succ(u) UpdateVertex(s);
{14}    else
{15}       g(u) = ∞;
{16}       for all s ∈ succ(u) ∪ {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);
```

Fig. 6. Lifelong Planning A*.

4.1. Lifelong Planning A*: the variables

LPA* maintains an estimate $g(s)$ of the start distance $g^*(s)$ of each vertex s. The initial search of LPA* calculates the g-values of each vertex in exactly the same order as A*. LPA* then carries the g-values forward from search to search. LPA* also maintains a second kind of estimate of the start distances. The rhs-values are one-step lookahead values (based on the g-values) that always satisfy the following relationship (Invariant 1) according to Lemma A.1 in Appendix A:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start}, \\ \min_{s' \in pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases} \quad (2)$$

A vertex is called locally consistent iff its g-value equals its rhs-value. This concept is similar to satisfying the Bellman equation for undiscounted deterministic sequential decision problems [10]. If all vertices are locally consistent then all of their g-values satisfy

FP calculates the key of vertex s as $k(s) = \min(g(s), rhs(s))$. LPA* calculates the same key when it is used with uninformed heuristics. In that case, the first and second components of the key are identical and only the first component needs to be used. The termination condition of the original version of DynamicSWSF-FP is "while $(U \neq \emptyset)$".

$$g(s) = \begin{cases} 0 & \text{if } s = s_{start}, \\ \min_{s' \in pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases} \quad (3)$$

A comparison to Eq. (1) shows that all g-values are equal to their respective start distances. Thus, the g-values of all vertices equal their start distances iff all vertices are locally consistent. This concept is important because one can then trace back a shortest path from s_{start} to any vertex u by always moving from the current vertex s, starting at u, to any predecessor s' that minimizes $g(s') + c(s', s)$ until s_{start} is reached (ties can be broken arbitrarily). However, LPA* does not make every vertex locally consistent. Instead, it uses the heuristics to focus the search and updates only the g-values that are relevant for computing a shortest path.

A* maintains an OPEN and a CLOSED list. The CLOSED list allows A* to avoid vertex reexpansions. LPA* does not maintain a CLOSED list since it uses local consistency checks to avoid vertex reexpansions. The OPEN list is a priority queue that allows A* to always expand a fringe vertex with a smallest f-value. LPA* also maintains a priority queue for this purpose. Its priority queue always contains exactly the locally inconsistent vertices (Invariant 2) according to Lemma A.2. The keys of the vertices in the priority queue roughly correspond to the f-values used by A*, and LPA* always recalculates the g-value of the vertex ("expands the vertex") in the priority queue with the smallest key. This is similar to A* that always expands the vertex in the priority queue with the smallest f-value. By expanding a vertex, we mean executing {10–16} (numbers in brackets refer to line numbers in Fig. 6). The key $k(s)$ of vertex s is a vector with two components:

$$k(s) = \big[k_1(s); k_2(s)\big], \quad (4)$$

where $k_1(s) = \min(g(s), rhs(s)) + h(s)$ and $k_2(s) = \min(g(s), rhs(s))$ {01}. The priority of a vertex in the priority queue is always the same as its key (Invariant 3) according to Lemma A.3. Keys are compared according to a lexicographic ordering. For example, a key $k(s)$ is less than or equal to a key $k'(s)$, denoted by $k(s) \dot{\leqslant} k'(s)$, iff either $k_1(s) < k'_1(s)$ or $(k_1(s) = k'_1(s)$ and $k_2(s) \leqslant k'_2(s))$. The first component of the keys $k_1(s)$ corresponds directly to the f-values $f(s) := g^*(s) + h(s)$ used by A* because both the g-values and rhs-values of LPA* correspond to the g-values of A* and the h-values of LPA* correspond to the h-values of A*.[3] The second component of the keys $k_2(s)$ corresponds to the g-values of A*. LPA* always expands the vertex in the priority queue with the smallest k_1-value, which corresponds to the f-value of an A* search, breaking ties in favor of the vertex with the smallest k_2-value, which corresponds to the g-value of an A* search. This is similar to A* that always expands the vertex in the priority queue with the smallest f-value, breaking ties towards smallest g-values. The resulting behavior of LPA* and A* is also similar. The keys of the vertices expanded by LPA* are nondecreasing over time according to Theorem 1. This is similar to A* where the f-values of the expanded vertices are also nondecreasing over time (since the heuristics are consistent), and—if A* breaks ties among vertices with the same f-values in favor of smaller g-values—$[f(s); g(s)]$ is also nondecreasing over time (since

[3] It turns out that using only the first component of the keys as priority is insufficient to imply Theorem 4 and thus insufficient to guarantee the efficiency of LPA* in terms of vertex expansions.

all children of an expanded vertex have strictly larger g-values than the expanded vertex itself).

4.2. Lifelong Planning A*: the algorithm

LPA* is shown in Fig. 6. The main function Main() first calls Initialize() to initialize the path-planning problem {17}. Initialize() sets the initial g-values of all vertices to infinity and sets their rhs-values according to Eq. (2) {03–04}. Thus, initially s_{start} is the only locally inconsistent vertex and is inserted into the otherwise empty priority queue with a key calculated according to Eq. (4) {05}. This initialization guarantees that the first call to ComputeShortestPath() performs exactly an A* search, that is, expands exactly the same vertices as A* in exactly the same order, provided that A* breaks ties among vertices with the same f-values suitably. Note that, in an actual implementation, Initialize() only needs to initialize a vertex when it encounters it during the search and thus does not need to initialize all vertices up front. This is important because the number of vertices can be large and only a few of them might be reached during the search. LPA* then waits for changes in edge costs {20}. To maintain Invariants 1–3 if some edge costs have changed, it calls UpdateVertex() {23} to update the rhs-values and keys of the vertices potentially affected by the changed edge costs as well as their membership in the priority queue if they become locally consistent or inconsistent, and finally recalculates a shortest path {19} by calling ComputeShortestPath(), that repeatedly expands locally inconsistent vertices in order of their priorities {10}.

A locally inconsistent vertex s is called locally overconsistent iff $g(s) > rhs(s)$. When ComputeShortestPath() expands a locally overconsistent vertex {12–13}, then it sets the g-value of the vertex to its rhs-value {12}, which makes the vertex locally consistent. A locally inconsistent vertex s is called locally underconsistent iff $g(s) < rhs(s)$. When ComputeShortestPath() expands a locally underconsistent vertex {15–16}, then it simply sets the g-value of the vertex to infinity {15}. This makes the vertex either locally consistent or overconsistent. If the expanded vertex was locally overconsistent, then the change of its g-value can affect the local consistency of its successors {13}. Similarly, if the expanded vertex was locally underconsistent, then it and its successors can be affected {16}. To maintain Invariants 1–3, ComputeShortestPath() therefore updates the rhs-values of these vertices, checks their local consistency, and adds them to or removes them from the priority queue accordingly {06–08}.

LPA* expands vertices until s_{goal} is locally consistent and the key of the vertex to expand next is no less than the key of s_{goal}. This is similar to A* that expands vertices until it expands s_{goal} at which point in time the g-value of s_{goal} equals its start distance and the f-value of the vertex to expand next is no less than the f-value of s_{goal}. 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 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) according to Theorem 5. This is similar to what A* can do if it does not use backpointers.

5. Analytical results

We now present some properties of LPA* that provide insight into how it works and show that it terminates, is correct, similar to A*, and efficient in terms of vertex expansions. The proofs of all theorems are given in Appendix A.

One of the most fundamental theorems for explaining the operation of LPA* is the next one about the order in which LPA* expands vertices.

Theorem 1. *The keys of the vertices that* ComputeShortestPath() *selects for expansion on line* {10} *are monotonically nondecreasing over time until* ComputeShortestPath() *terminates.*

Theorem 1 allows one to prove several properties of ComputeShortestPath(). For example, consider a locally consistent vertex whose key is less than U.TopKey(), that is, the smallest key of any locally inconsistent vertex. Its g-value can change only when it is expanded again. Consequently, its key cannot increase and must remain less than U.TopKey() since U.TopKey() is monotonically nondecreasing according to Theorem 1. Thus, the vertex cannot be expanded again. The next theorem proves that this remains true for locally consistent vertices whose keys are less than or equal to U.TopKey().

Theorem 2. *Let* $k =$ U.TopKey() *during the execution of line* {09}. *If vertex* s *is locally consistent at this point in time with* $k(s) \dot{\leqslant} k$, *then it remains locally consistent until* ComputeShortestPath() *terminates.*

Now assume that ComputeShortestPath() expands a locally overconsistent vertex. ComputeShortestPath() sets the g-value of the vertex to its rhs-value {12}. This does not change its rhs-value nor its key but makes it locally consistent. Consequently, the vertex satisfies the conditions of Theorem 2 and thus remains locally consistent until ComputeShortestPath() terminates, which proves the next theorem.

Theorem 3. *If a locally overconsistent vertex is selected for expansion on line* {10}, *then it is locally consistent the next time line* {09} *is executed and remains locally consistent until* ComputeShortestPath() *terminates.*

5.1. Termination and correctness

Theorem 3 implies that ComputeShortestPath() expands any locally overconsistent vertex at most once until it terminates. Now assume that ComputeShortestPath() expands a locally underconsistent vertex. ComputeShortestPath() sets the g-value of the vertex to infinity {15}. This makes the vertex either locally consistent or overconsistent. Since the g-value of a vertex changes only when it is expanded, the vertex cannot become locally underconsistent before it is expanded again. Thus, if the vertex is expanded again, it is expanded as locally overconsistent and, as just argued, is then not expanded again until ComputeShortestPath() terminates. Thus, ComputeShortestPath() expands each vertex at most twice and therefore terminates.

Theorem 4. ComputeShortestPath() *expands each vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and thus terminates.*

All theorems stated so far hold for the termination condition of ComputeShortestPath() {09} and the modified termination condition "while U is not empty". ComputeShortestPath() with the latter termination condition terminates when all vertices are locally consistent and thus when the g-values of all vertices equal their start distances. In this case, one can trace back a shortest path from s_{start} to any vertex s'' by always moving from the current vertex s, starting at s'', to any predecessor s' that minimizes $g(s') + c(s', s)$ until s_{start} is reached (ties can be broken arbitrarily). However, the modified termination condition expands too many vertices since one only needs to find a shortest path from s_{start} to s_{goal}. For example, Theorem 2 shows that, if the goal vertex is locally consistent during the execution of line {09} and its key is less than or equal to U.TopKey(), then it remains locally consistent until ComputeShortestPath() terminates. Thus, its g-value no longer changes. The g-value of the goal vertex equals its start distance after ComputeShortestPath() with the modified termination condition terminates. Thus, it was equal to its start distance since its last expansion. This implies that the g-value of the goal vertex also equals its start distance after ComputeShortestPath() with the actual termination condition {09} terminates. Furthermore, one can show that, if the goal vertex is locally consistent during the execution of line {09} and its key is less than or equal to U.TopKey(), that is, after ComputeShortestPath() with the actual termination condition {09} terminates, then one can find a shortest path from s_{start} to s_{goal} in exactly the same way as stated for the modified termination condition, which proves the next theorem.

Theorem 5. *After* ComputeShortestPath() *terminates, one can trace back a shortest path from s_{start} to s_{goal} by always moving from the 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*).

*5.2. Similarity to A**

In Section 4, we pointed out strong algorithmic similarities between LPA* and A*. The next theorems show additional similarities between LPA* and A*.

Theorem 4 already showed that ComputeShortestPath() expands each vertex at most twice. This is similar to A*, that expands each vertex at most once. Thus, ComputeShortestPath() returns after a number of vertex expansions that is at most twice the number of vertices.

The next three theorems show that ComputeShortestPath() expands locally overconsistent vertices in a way very similar to how A* expands vertices. The next theorem, for example, shows that the first component of the key of a locally overconsistent vertex at the time ComputeShortestPath() expands it is the same as the f-value of the vertex. The second component of its key is its start distance.

Theorem 6. *Whenever* ComputeShortestPath() *selects a locally overconsistent vertex s for expansion on line* {10}*, then its key is $k(s) \doteq [f(s); g^*(s)]$.*

Theorem 1 showed that ComputeShortestPath() expands vertices in order of monotonically nondecreasing keys. Thus, Theorem 6 implies that ComputeShortestPath() expands locally overconsistent vertices in order of monotonically nondecreasing f-values and vertices with the same f-values in order of monotonically nondecreasing start distances. A* has the same property provided that it breaks ties in favor of vertices with smaller start distances.

Theorem 7. ComputeShortestPath() *expands locally overconsistent vertices s with finite f-values in the same order as* A* (*possibly except for vertices with the same $[f(s); g^*(s)]$ keys*)*, provided that* A* *always breaks ties among vertices with the same f-values in favor of vertices with the smallest start distances and in case of remaining ties expands s_{goal} last.*

Note, however, that most of the vertices expanded by A* are usually not expanded by ComputeShortestPath(). The next theorem shows that ComputeShortestPath() expands at most those locally overconsistent vertices whose f-values are less than the f-value of the goal vertex and those vertices whose f-values are equal to the f-value of the goal vertex and whose start distances are less than or equal to the start distance of the goal vertex. A* has the same property provided that it breaks ties in favor of vertices with smaller start distances. (Theorem 11 points out a related similarity of LPA* and A*.)

Theorem 8. ComputeShortestPath() *expands at most those locally overconsistent vertices s with $[f(s); g^*(s)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$.*

The next theorem shows that the search tree of LPA* contains the search tree of A*. This is not surprising since LPA* finds shortest paths and every search method that finds shortest paths has to expand at least the vertices that A* with the same heuristics expands, except possibly for some vertices whose f-values are equal to the f-value of the goal vertex [9].

Theorem 9. LPA* *shares with* A* *the following property for s_{goal} and all vertices s that* A* *expands* (*possibly except for vertices with $[f(s); g^*(s)] = [f(s_{goal}); g^*(s_{goal})]$*)*, provided that* A* *always breaks ties among vertices with the same f-values in favor of vertices with the smallest start distances and its g-values are assumed to be infinity if* A* *has not calculated them*: *The g-values of these vertices s equal their respective start distances after termination and one can trace back a shortest path from s_{start} to them by always moving from the current vertex s', starting at s, to any predecessor s'' that minimizes $g(s'') + c(s'', s')$ until s_{start} is reached* (*ties can be broken arbitrarily*)*.*

5.3. Efficiency

We now show that LPA* can expand fewer vertices than suggested by Theorem 4. The next theorem shows that LPA* is efficient because it performs incremental searches and

thus calculates only those g-values that have been affected by cost changes or have not been calculated yet in previous searches.

Theorem 10. ComputeShortestPath() *does not expand any vertices whose g-values were equal to their respective start distances before* ComputeShortestPath() *was called.*

Our final theorem shows that LPA* is efficient because it performs heuristic searches and thus calculates only the g-values of those vertices that are important to determine a shortest path. Theorem 8 has already shown how heuristics limit the number of locally overconsistent vertices expanded by ComputeShortestPath(). The next theorem generalizes this result to all locally inconsistent vertices expanded by ComputeShortestPath().

Theorem 11. ComputeShortestPath() *expands at most those vertices* s *with* $[f(s); g^*(s)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$ *or* $[f_{old}(s); g_{old}(s)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$, *where* $g_{old}(s)$ *is the g-value and* $f_{old}(s) = g_{old}(s) + h(s)$ *is the f-value of vertex* s *directly before the call to* ComputeShortestPath().

More informed heuristics are larger and thus $[f(s); g^*(s)]$ and $[f_{old}(s); g_{old}(s)]$ are larger. This implies that fewer vertices s satisfy $[f(s); g^*(s)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$ or $[f_{old}(s); g_{old}(s)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})] \doteq [g^*(s_{goal}), g^*(s_{goal})]$ and can get expanded by ComputeShortestPath() according to the previous theorem.

Note, however, that incremental search is not more efficient than search from scratch in the worst case [11]. Replanning with LPA* can best be understood as transforming the A* search tree of the old search problem to the A* search tree of the new one. This results in some computational overhead since parts of the old A* search tree need to be undone. It also results in computational savings since other parts of the old A* search tree can be reused. The larger the overlap between the old and new A* search trees, the more efficient replanning with LPA* tends to be compared to using A* to create the new search tree from scratch. To be more precise: It is not only important that the trees are similar but most start distances of its vertices have to be the same as well. LPA* can be less efficient than A* if the overlap between the old and new A* search trees is small. Note also that LPA* needs about the same amount of memory as A* since it needs to remember the previous search tree. Therefore, the search trees need to fit in memory, which is a realistic assumption, for example, when searching maps in robotics, computer gaming, or network routing, in addition to the application discussed in the second part of this article.

6. Optimizations of Lifelong Planning A*

There are several ways of optimizing LPA*, including modifying the termination condition of ComputeShortestPath() {09}. As stated, ComputeShortestPath() terminates when the goal vertex is locally consistent and its key is less than or equal to U.TopKey(). However, ComputeShortestPath() can also terminate when the goal vertex is locally overconsistent and its key is less than or equal to U.TopKey(). To understand why this is so, assume that the goal vertex is indeed locally overconsistent and its key is less than

or equal to U.TopKey(). Then, its key must be equal to U.TopKey() since U.TopKey() is the smallest key of any locally inconsistent vertex. Thus, ComputeShortestPath() could expand the goal vertex next, in which case it would set its g-value to its rhs-value. The goal vertex then becomes locally consistent according to Theorem 3, its key is less than or equal to U.TopKey(), and ComputeShortestPath() thus terminates. At this point in time, the g-value of the goal vertex equals its start distance. Thus, ComputeShortestPath() can already terminate when the goal vertex is locally overconsistent and its key is less than or equal to U.TopKey(). In this case, the goal vertex is not expanded. Its rhs-value equals its start distance but its g-value is not updated and thus does not equal its start distance. However, the procedure for tracing back a shortest path from the start vertex to the goal vertex does not depend on the g-value of the goal vertex and thus can be used unchanged. If the rhs-value of the goal vertex is infinity then there is no path from the start vertex to the goal vertex. This optimization avoids expanding all vertices whose keys are the same as the key of s_{goal}, which could potentially be a large number of vertices.

In the following, we describe several other simple ways of optimizing LPA* that do not change which vertices LPA* expands or in which order it expands them. The resulting version of LPA* is shown in Fig. 7.

- A vertex sometimes is removed from the priority queue and then immediately reinserted with a different key. For example, a vertex can be removed on line {07} and then be reentered on line {08}. In this case, it is often more efficient to leave the vertex in the priority queue, update its key, and only change its position in the priority queue {08'}.
- When UpdateVertex() on line {13} computes the rhs-value for a successor of a locally overconsistent vertex it is unnecessary to take the minimum over all of its predecessors. It is sufficient to compute the rhs-value as the minimum of its old rhs-value and the sum of the new g-value of the locally overconsistent vertex and the cost of moving from the locally overconsistent vertex to the successor {19'}. The reason is that only the g-value of the locally overconsistent vertex has changed. Since it decreased, it can only decrease the rhs-value of the successor.
- When UpdateVertex() on line {16} computes the rhs-value for a successor of a locally underconsistent vertex, the only g-value that has changed is the g-value of the locally underconsistent vertex. Since it increased, the rhs-value of the successor can only be affected if its old rhs-value was based on the old g-value of the locally underconsistent vertex. This can be used to decide whether the successor needs to be updated and its rhs-value needs to be recomputed {26'}.
- The second and third optimizations concern the computations of the rhs-values of the successors after the g-value of a vertex has changed. Similar optimizations can be made for the computation of the rhs-value of a vertex after the cost of one of its incoming edges has changed {38',43'}.
- Finally, we introduce new variables $p(s)$ that satisfy the invariants $rhs(s) = g(p(s)) + c(p(s), s)$ for all vertices s to avoid some calculations. For example, we can now write "if ($s \neq s_{start}$ AND $p(s) = u$)" {24'} instead of the more cumbersome similar "if ($s \neq s_{start}$ AND $rhs(s) = g(u) + c(u, s)$)".

The pseudocode uses the following functions to manage the priority queue: U.Top() returns a vertex with the smallest priority of all vertices in priority queue U. U.TopKey() returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey() returns $[\infty; \infty]$.) U.Insert(s, k) inserts vertex s into priority queue U with priority k. U.Update(s, k) changes the priority of vertex s in priority queue U to k. (It does nothing if the current priority of vertex s already equals 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 = ∅;
{03'} for all s ∈ S
{04'}   rhs(s) = g(s) = ∞;
{05'}   p(s) = NULL;
{06'} rhs(s_start) = 0;
{07'} U.Insert(s_start, [h(s_start); 0]);
procedure UpdateVertex(u)
{08'} if (g(u) ≠ rhs(u) AND u ∈ U) U.Update(u, CalculateKey(u));
{09'} else if (g(u) ≠ rhs(u) AND u ∉ U) U.Insert(u, CalculateKey(u));
{10'} else if (g(u) = rhs(u) AND u ∈ U) U.Remove(u);

procedure ComputeShortestPath()
{11'} while (U.TopKey() ≺ CalculateKey(s_goal) OR rhs(s_goal) > g(s_goal))
{12'}  u = U.Top();
{13'}  if (g(u) > rhs(u))
{14'}    g(u) = rhs(u);
{15'}    U.Remove(u);
{16'}    for all s ∈ succ(u)
{17'}      if (rhs(s) > g(u) + c(u, s))
{18'}        p(s) = u;
{19'}        rhs(s) = g(u) + c(u, s);
{20'}        UpdateVertex(s);
{21'}  else
{22'}    g(u) = ∞;
{23'}    for all s ∈ succ(u) ∪ {u}
{24'}      if (s ≠ s_start AND p(s) = u)
{25'}        p(s) = arg min_{s'∈pred(s)} (g(s') + c(s', s));
{26'}        rhs(s) = g(p(s)) + c(p(s), s);
{27'}        UpdateVertex(s);
procedure Main()
{28'} Initialize();
{29'} forever
{30'}  ComputeShortestPath();
{31'}  Wait for changes in edge costs;
{32'}  for all directed edges (u, v) with changed edge costs
{33'}    c_old = c(u, v);
{34'}    Update the edge cost c(u, v);
{35'}    if (c_old > c(u, v))
{36'}      if (rhs(v) > g(u) + c(u, v))
{37'}        p(v) = u;
{38'}        rhs(v) = g(u) + c(u, v);
{39'}        UpdateVertex(v);
{40'}    else
{41'}      if (v ≠ s_start AND p(v) = u)
{42'}        p(v) = arg min_{s'∈pred(v)} (g(s') + c(s', v));
{43'}        rhs(v) = g(p(v)) + c(p(v), v);
{44'}        UpdateVertex(v);
```

Fig. 7. Lifelong Planning A* (optimized version).

Also, we have not included two optimizations in the pseudocode because they make it somewhat messy. One optimization is to initialize the data structures of vertices only when the vertices are encountered during the search rather than up front in Initialize(). The other optimization is to continue the while-loop of ComputeShortestPath() only if the heuristic

value of the vertex with the smallest key in the priority queue is finite. This is similar to A* that can terminate if it is about to expand a vertex with an infinite f-value. The second optimization was not used in the experimental evaluation of LPA*.

7. Extensions of Lifelong Planning A*

The costs of edges can change during replanning. In this case, it can be more efficient to take the changed edge costs into account before ComputeShortestPath() terminates than to wait until it does. This requires one to modify ComputeShortestPath() so that it continues to maintain Invariants 1–3, which can be done by processing all edges with changed edge costs before the while loop in ComputeShortestPath() iterates, by copying lines {21–23} and inserting them directly after line {16} into the while loop. In this case, Theorem 5 continues to hold but some of the other theorems might not, including Theorem 4. For example, a vertex that has already been expanded twice and thus is locally consistent can, after each change of edge costs, again become locally inconsistent and thus be expanded up to two more times. On the other hand, a vertex that is locally inconsistent can, after a change of edge costs, become locally consistent and thus might not get expanded at all.

8. Experimental evaluation

We now compare breadth-first search, A*, DynamicSWSF-FP, and the optimized version of LPA* experimentally. We use DynamicSWSF-FP with the same optimizations that we developed for LPA*, to avoid biasing our experimental results in favor of LPA*. We study two versions of A*, namely one that breaks ties among vertices with the same f-value in favor of vertices with smaller g-values (A* version 1), just like LPA*, and one that breaks ties among vertices with the same f-value in favor of vertices with larger g-values (A* version 2), which tends to result in fewer vertex expansions. The priority queues of all search methods were implemented as binary heaps. Since all search methods determine shortest paths, we need to compare their total search time until a shortest path has been found. To this end, we measure their actual runtimes *ti* (in milliseconds), run on a Pentium 1.7 MHz PC. Since the runtimes are machine dependent, they make it difficult for others to reproduce the results of our performance comparison. We therefore also use two performance measures that both correspond to common operations performed by the search methods and thus heavily influence their runtimes, yet are machine independent: the total number of vertex expansions *ve* (that is, updates of the g-values, similar to backup operations of dynamic programming for sequential decision problems), and the total number of heap percolates *hp* (exchanges of a parent and child in the heap). Note that we count two vertex expansions, not just one vertex expansion, if LPA* expands the same vertex twice, to avoid biasing our experimental results in favor of LPA*.

We performed experiments with four-connected gridworlds of size 51×51 with directed edges between adjacent cells. We use the Manhattan distances as heuristics for the cost of a shortest path between two cells for both A* and LPA*, that is, the sum of the absolute differences of their x- and y-coordinates. We generate one hundred gridworlds. The start

and goal cells are drawn with uniform probability from all cells for each gridworld. All edge costs are either one or two with uniform probability. We then change each gridworld five hundred times in a row by selecting 0.6 percent of the edges (with replacement) and assigning them random costs. After each change, the search methods recompute a shortest path. Fig. 8 reports the average over the one hundred gridworlds for each search method and the three performance measures (per replanning episode). Both versions of A* perform about equally well; the tie-breaking rule does not make a difference in our gridworlds.

We also performed experiments with four-connected gridworlds of size 51×51 with obstacles. We again use the Manhattan distances as heuristics for the cost of a shortest path between two cells, generate one hundred gridworlds, and draw the start and goal cells with uniform probability from all cells for each gridworld. Each cell is blocked with 20 percent probability. Blocked cells have neither incoming nor outgoing edges but there exist edges from unblocked cells to adjacent unblocked cells. Their costs are one. We then change each gridworld five hundred times in a row by randomly selecting eight unblocked cells and making them blocked, and randomly selecting eight blocked cells and making them unblocked. Thus, the obstacle density remains unchanged but about 0.6 percent of the cells change their blockage status. After each of the changes, the search methods recompute a shortest path. Fig. 9 reports the average over the one hundred gridworlds for each search method and the three performance measures (per replanning episode). A* version 2 outperforms A* version 1 in these gridworlds because there are often multiple shortest

	uninformed search		heuristic search		
complete search	Breadth-First Search			A* Version 1	(A* Version 2)
	ve =	1240.04	ve =	307.93	(255.58)
	hp =	5232.67	hp =	2021.92	(2059.81)
	ti =	0.249	ti =	0.083	(0.077)
incremental search	DynamicSWSF-FP			LPA*	
	ve =	104.91	ve =		23.71
	hp =	491.08	hp =		212.43
	ti =	0.036	ti =		0.015

Fig. 8. Comparison of search methods in gridworlds with random edge costs.

	uninformed search		heuristic search		
complete search	Breadth-First Search			A* Version 1	(A* Version 2)
	ve =	1124.23	ve =	241.77	(103.33)
	hp =	3612.74	hp =	1003.50	(820.79)
	ti =	0.226	ti =	0.064	(0.040)
incremental search	DynamicSWSF-FP			LPA*	
	ve =	91.47	ve =		15.56
	hp =	482.87	hp =		137.68
	ti =	0.039	ti =		0.018

Fig. 9. Comparison of search methods in gridworlds with random obstacles.

paths and a large number of cells on these paths have f-values that are equal to the f-value of the goal cell. A* version 1 expands all of these cells, whereas A* version 2 expands only those cells on one of the shortest paths. Thus, it appears to be a disadvantage that LPA* breaks ties in the same way as A* version 1. However, the fact that LPA* finds all shortest paths during the first planning episode speeds up replanning when some of them get blocked, and LPA* outperforms even A* version 2 in the long run. This suggests that tie-breaking might become less important as the number of replanning episodes increases.

Both tables confirm the observations made in Section 3. Each of the three performance measures is improved when going from an uninformed to a heuristic search and from a complete to an incremental search, although this is not guaranteed in general. LPA* outperforms the other search methods according to all performance measures. Thus, combining lifelong and heuristic searches can indeed speed up replanning. Note, however, that the exact number of vertex expansions and heap percolates depends on low-level implementation and machine details, for example, how the graphs are constructed from the gridworlds and in which order successors are generated when vertices are expanded. Similarly, the differences in runtime depend on the instruction set of the processor, the optimizations performed by the compiler, and the data structures used for the priority queues. For example, LPA* needs more time per vertex expansion than both versions of A* but the resulting difference in runtime could potentially be decreased in favor of LPA* by optimizing LPA* by "unrolling" its code into code for the first iteration and code for all subsequent iterations and then deleting all unnecessary code from the code for the first iteration. Similarly, LPA* needs fewer heap percolates than both versions of A* but the resulting difference in runtime can be decreased in favor of A* by using buckets to implement the priority queues rather than heaps. For example, the runtime of A* decreased from 0.083 and 0.077 milliseconds to 0.035 milliseconds in the experiment of Fig. 8 when we implemented A* with buckets and a simple FIFO tie-breaking strategy within buckets.

We also performed more detailed experiments that compare LPA* with the two versions of A*. We use again four-connected gridworlds with directed edges between adjacent cells, as in the first experiment. We report the probability that the cost of the shortest path changes to ensure that the edge cost changes indeed change the shortest path sufficiently often. A probability of 33.9 percent, for example, means that the cost of the shortest path changes on average after 2.96 planning episodes. For each experiment, we report the runtime (in milliseconds) averaged over all first planning episodes (#1) and over all planning episodes (#2). We also report the speedup of LPA* over A* version 2 in the long run (#3), that is, the ratio of the runtimes of A* version 2 and LPA* averaged over all planning episodes. Since LPA* expands the same vertices during the first search as A* version 1 but expands them more slowly, its first search is always slower than that of A* version 1, which in turn often expands more vertices and then is slower than A* version 2. During the subsequent searches, however, LPA* often expands fewer vertices than both versions of A* and is thus faster than them. We therefore also report the replanning episode after which the average total runtime of LPA* is smaller than the one of A* version 2 (#4), in other words, the number of replanning episodes that are necessary for one to prefer LPA* over A* version 2. For example, if this number is one, then LPA* solves one planning problem and one replanning problems together faster than A* version 2.

Experiment 1. In the first experiment, the size of the gridworlds is 101×101. We change the number of edges that get assigned random costs before each planning episode. Fig. 10 shows our experimental results. The smaller the number of edges that get reassigned random costs, the less the search space changes and the larger the advantage of LPA* in our experiments. The average runtime of the first planning episode of LPA* tends to be larger than the one of both versions of A* but the average runtime of the following planning episodes tends to be so much smaller (if the number of edges that get reassigned random costs is sufficiently small) that the number of replanning episodes that are necessary for one to prefer LPA* over A* is one. Although our tabulated results do not show this, the average runtime of LPA* can also be larger than the one of A*, for example, if a larger number of edges change their cost.

Experiment 2. In the second experiment, the number of edges that get reassigned random costs before each planning episode is 0.6 percent. We change the size of the square gridworlds. Fig. 11 shows our experimental results. The smaller the gridworlds, the larger the advantage of LPA* in our experiments, although we were not able to predict this effect. This is an important insight since it implies that LPA* does not scale well in our gridworlds (although part of this effect could be due to the fact that more edges get reassigned random costs as the size of the gridworlds increases and this time is included in the runtime averaged over all planning episodes). We therefore devised the third experiment.

edge cost changes	path cost changes	A* version 1	A* version 2	LPA*			
		#1 and #2	#1 and #2	#1	#2	#3	#4
0.2%	3.0%	0.302	0.299	0.386	0.029	10.370×	1
0.4%	7.9%	0.340	0.336	0.419	0.067	5.033×	1
0.6%	13.0%	0.365	0.362	0.453	0.108	3.344×	1
0.8%	17.6%	0.410	0.406	0.499	0.156	2.603×	1
1.0%	20.5%	0.373	0.370	0.434	0.174	2.126×	1
1.2%	24.6%	0.414	0.413	0.476	0.222	1.858×	1
1.4%	28.7%	0.470	0.468	0.539	0.282	1.657×	1
1.6%	32.6%	0.504	0.500	0.563	0.332	1.507×	1
1.8%	32.1%	0.479	0.455	0.497	0.328	1.384×	1
2.0%	33.8%	0.401	0.394	0.433	0.315	1.249×	1

Fig. 10. Experiment 1.

gridworld size	path cost changes	A* version 1	A* version 2	LPA*			
		#1 and #2	#1 and #2	#1	#2	#3	#4
51×51	7.3%	0.083	0.077	0.098	0.015	5.032×	1
76×76	10.7%	0.206	0.201	0.258	0.050	3.987×	1
101×101	13.0%	0.348	0.345	0.437	0.104	3.315×	1
126×126	16.2%	0.681	0.690	0.789	0.220	3.128×	1
151×151	17.7%	0.917	0.933	1.013	0.322	2.900×	1
176×176	21.5%	1.499	1.553	1.608	0.564	2.753×	1
201×201	22.9%	1.781	1.840	1.898	0.682	2.696×	1

Fig. 11. Experiment 2.

80% of edge cost changes are $\leqslant$ 25 cells away from the goal

gridworld size	path cost changes	A* version 1	A* version 2	LPA*			
		#1 and #2	#1 and #2	#1	#2	#3	#4
51 × 51	13.5%	0.091	0.084	0.115	0.014	6.165×	1
76 × 76	23.9%	0.195	0.189	0.245	0.028	6.661×	1
101 × 101	33.4%	0.302	0.295	0.375	0.048	6.184×	1
126 × 126	42.5%	0.691	0.696	0.812	0.084	8.297×	1
151 × 151	48.5%	0.864	0.886	0.964	0.114	7.808×	1
176 × 176	55.7%	1.308	1.353	1.450	0.156	8.683×	1
201 × 201	59.6%	1.613	1.676	1.733	0.202	8.305×	1

80% of edge cost changes are $\leqslant$ 50 cells away from the goal

gridworld size	path cost changes	A* version 1	A* version 2	LPA*			
		#1 and #2	#1 and #2	#1	#2	#3	#4
51 × 51	8.6%	0.092	0.086	0.115	0.017	5.138×	1
76 × 76	15.7%	0.195	0.190	0.247	0.039	4.822×	1
101 × 101	23.2%	0.310	0.304	0.378	0.072	4.235×	1
126 × 126	31.3%	0.696	0.702	0.812	0.130	5.398×	1
151 × 151	36.2%	0.875	0.896	0.959	0.173	5.166×	1
176 × 176	44.0%	1.331	1.372	1.458	0.242	5.664×	1
201 × 201	48.3%	1.636	1.689	1.742	0.313	5.398×	1

80% of edge cost changes are $\leqslant$ 75 cells away from the goal

gridworld size	path cost changes	A* version 1	A* version 2	LPA*			
		#1 and #2	#1 and #2	#1	#2	#3	#4
76 × 76	12.1%	0.201	0.196	0.250	0.047	4.206×	1
101 × 101	17.5%	0.312	0.306	0.391	0.088	3.499×	1
126 × 126	26.0%	0.699	0.703	0.818	0.175	4.012×	1
151 × 151	28.8%	0.881	0.893	0.972	0.225	3.978×	1
176 × 176	36.8%	1.331	1.370	1.438	0.319	4.301×	1
201 × 201	40.1%	1.670	1.728	1.790	0.408	4.236×	1

Fig. 12. Experiment 3.

Experiment 3. In the third experiment, the number of edges that get reassigned random costs before each planning episode is again 0.6 percent. We change both the size of the square gridworlds and how close the edges that get reassigned random costs are to the goal cell. 80 percent of these edges leave cells that are close to the goal cell. Fig. 12 shows our experimental results. Now, the advantage of LPA* no longer decreases with the size of the gridworlds. The closer the edge cost changes are to the goal cell, the larger the advantage of LPA* in our experiments. This is an important insight since it suggests to use LPA* when most of the edge cost changes are close to the goal cell.

To summarize, in some situations LPA* is more efficient than A* not only in terms of vertex expansions but also in terms of runtime. However, these situations need to get characterized better. Also, the efficiency of LPA* and A* depends on low-level implementation and machine details, and the results of the comparison thus might have been different for different implementations or hardware environments. For example, LPA* needs more than one replanning episode to outperform A* if the number of edges that get reassigned random costs before each planning episode is less than 1.0 percent and

does not outperform A* at all if the number of edges that get reassigned random costs before each planning episode is 1.0 percent or more in the experiment of Fig. 10 when we implemented A* with buckets and a simple FIFO tie-breaking strategy within buckets but left the implementation of LPA* unchanged. One problem of making fair comparisons is that A* and LPA* perform very different basic operations and thus cannot be compared using proxies, such as the number of vertex expansions. Another problem is that the search spaces of incremental search methods can be relatively small (for example, when searching maps for computer gaming) and their scaling properties are thus less important than implementation and machine details. Therefore, we are only willing to conclude from our experiments that incremental heuristic search is a promising technology that needs to get investigated further.

9. An application to symbolic planning

Obvious applications of LPA* include search in the context of transportation or communication networks, for example, route planning for cars under changing traffic conditions and for packages on computer networks with changing load conditions. For example, in "most of today's commercial routers, this recomputation is done by deleting the current SPT [shortest-path tree] and recomputing it from scratch by using the well known Dijkstra algorithm" [12] although it has recently been discovered in the networking literature that DynamicSWSF-FP can be used to update routing tables as the congestion of links changes [12,13]. In this section, however, we apply LPA* to more complex path-planning problems, namely to symbolic planning problems. LPA* applies to replanning problems where edges or vertices are added or deleted, or the costs of edges are changed, for example, because the cost of planning operators, their preconditions, or their effects change from one path-planning problem to the next. We first describe how to apply LPA* to symbolic planning and then present experimental results. Our goal is not to develop a full scale symbolic replanner but rather to evaluate LPA* in an additional domain and provide some insight into its properties.

*9.1. Heuristic search-based replanning with Lifelong Planning A**

Heuristic search-based planners perform a heuristic forward or backward search in the space of world states to find a path from the start vertex to a goal vertex. They were introduced in [14] and [15] and are now very popular. Several of them entered the second planning competition at AIPS-2000, including HSP 2.0 [16], FF [17], GRT [18], and AltAlt [19].

Many heuristic search-based planners solve STRIPS-planning problems with ground planning operators. We use LPA* in the same way. Such STRIPS-planning problems consist of a set of propositions P that are used to describe the states and planning operators, a set of ground planning operators O, the start state $I \subseteq P$, and the partially specified goal $G \subseteq P$. Each planning operator $o \in O$ has a cost $cost(o) > 0$, a precondition list $Prec(o) \subseteq P$, an add list $Add(o) \subseteq P$, and a delete list $Delete(o) \subseteq P$. The STRIPS-planning problem induces a path-planning problem that consists of a set of states (vertices)

2^P, a start state I, a set of goal states $\{X \subseteq P \mid G \subseteq X\}$, a set of actions (directed edges) $\{o \in O \mid Prec(o) \subseteq s\}$ for each state $s \subseteq P$ where action o transitions from state $s \subseteq P$ to state $s - Delete(o) + Add(o) \subseteq P$ with cost $cost(o)$. All paths (plans) from the start state to any goal state are solutions of the STRIPS planning problem. The shorter the path, the higher the quality of the solution.

LPA* performs a forward search in the space of world states using the consistent h_{max}-heuristic that was first developed in the context of HSP [16]. The heuristic values are calculated by solving a relaxed version of the planning problem, where one recursively approximates (by ignoring all delete lists) the cost of achieving each goal proposition individually from the given state and then combines the estimates to obtain the heuristic value of the given state. In the following, we explain the calculation of the heuristic values in detail. We use $g_s(p)$ to denote the approximate cost of achieving proposition $p \in P$ from state $s \subseteq P$, and $g_s(o)$ to denote the approximate cost of achieving the preconditions of planning operator $o \in O$ from state $s \subseteq P$. HSP defines these quantities recursively. It defines for all $s \subseteq P$, $p \in P$, and $o \in O$ (the minimum of an empty set is defined to be infinity):

$$g_s(p) = \begin{cases} 0 & \text{if } p \in s, \\ \min_{o \in O \mid p \in Add(o)} [cost(o) + g_s(o)] & \text{otherwise,} \end{cases} \tag{5}$$

$$g_s(o) = \max_{p \in Prec(o)} g_s(p). \tag{6}$$

Then, the heuristic value $h_{max}(s)$ of state $s \in S$ can be calculated as $h_{max}(s) = \max_{p \in G} g_s(p)$. These heuristics are consistent and thus allow LPA* to find shortest plans.

Unfortunately, LPA* cannot be used completely unchanged for heuristic search-based replanning. There are three issues that need to be addressed, resulting in SHERPA (Speedy HEuristic search-based RePlAnner) [20]. Fig. 13 shows the unoptimized version of SHERPA that can be optimized as outlined in Section 6.

- First, the pseudocode shown in Fig. 6 initializes all vertices 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 vertices and edges only when they are encountered during the search.
- Second, the pseudocode iterates over all predecessors of a vertex to determine its rhs-value on line 6 in Fig. 6. However, it is difficult to determine the predecessors of vertices for symbolic planning. (Switching the search direction does not help since LPA* and thus SHERPA sometimes needs to iterate over all predecessors and sometimes over all successors of a vertex.) We address this issue as follows: Whenever a vertex is expanded, SHERPA generates all of its successors and for each of them remembers that the expanded vertex is one of its predecessors. Thus, at any point in time, SHERPA has those predecessors of a vertex 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 vertex (instead of all of them) when it calculates the rhs-value of the vertex. This does not change the calculated rhs-value since the g-values of the other predecessors are infinite.

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 vertex.

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 planning 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$(destination(e))$;

Fig. 13. The SHERPA replanner.

- Third, the pseudocode assumes that there is only one goal vertex. However, there are often many goal states in symbolic planning if the goal is only partially specified. We address this issue by removing the successors of all vertices that satisfy the goal condition and then merging all vertices that satisfy the goal condition into one new vertex, called s_{goal}.

9.2. An example of heuristic search-based replanning

In the miconic (elevator) domain, the f floors of a building are served by an elevator. Initially, p people are either in the elevator or waiting for it on randomly selected floors. The goal is to get each person to his or her destination floor. The elevator can move from any floor to any other floor in one step, whether it is empty or not. There is no limit on the number of people that can be in the elevator at any time.

The planning domain contains the following operators:

- The elevator moves from floor f_i to floor f_j with $i \neq j$.
- Person p_k boards the elevator on floor f_i provided that the elevator is currently on floor f_i and floor f_i is the origin of person p_k.
- Person p_k gets off the elevator on floor f_i, provided that person p_k is in the elevator, the elevator is currently on floor f_i, and floor f_i is the destination of person p_k.

A problem instance is defined by f, p, a start state (the initial location of each person and the initial location of the elevator) and a goal condition (the destination floor of each person). We apply SHERPA to a problem instance with $p = 2$ people (Paul and Sally) and $f = 3$ floors. In the start state, Paul has boarded the elevator on the third floor and Sally is waiting on the first floor. The goal condition requires Paul to be on the first floor and Sally to be on the third floor.

Fig. 14 shows the search graph generated by SHERPA when it uses search from scratch with the h_{max} heuristic to solve the planning problem. Expanded vertices are shown in grey with a solid outline in the figure. The numbers in circles indicate the order of vertex expansions. Generated but not expanded vertices are shown in white with a dashed outline. Keys of the locally inconsistent vertices are shown in the lower right corner. The shortest plan is to move the elevator directly to the first floor, let Paul exit and Sally enter the elevator (in any order), move the elevator directly to the third floor, and let Sally exit the elevator. We now remove the ground operator that corresponds to the elevator moving from the first floor directly to the third floor. This deletes several edges from the state space, including one that is part of the plan. The edges deleted from the search graph are shown dashed in the figure. Consequently, SHERPA needs to replan. Fig. 15 (left) shows the search graph generated by SHERPA when it uses search from scratch with the same heuristic to solve the new planning problem. The shortest plan now is to move the elevator directly to the first floor, let Paul exit and Sally enter the elevator (in any order), move the elevator first to the second and then to the third floor, and let Sally exit the elevator. Fig. 15 (right) shows the search graph generated by SHERPA when it uses incremental search with the same heuristic to solve the new planning problem, resulting in the same shortest plan.

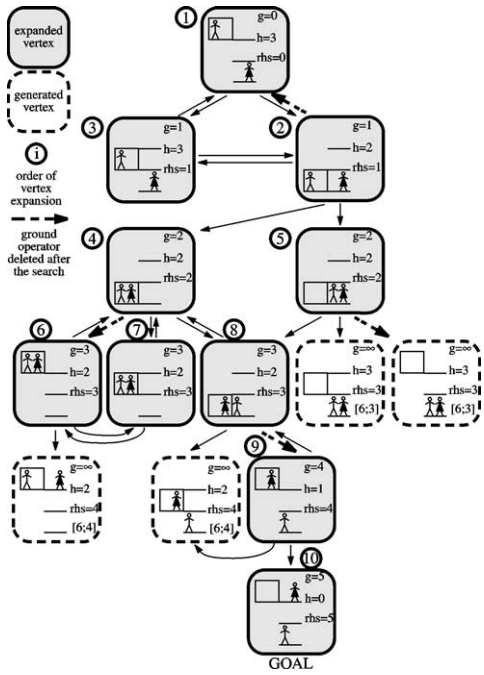

Fig. 14. First search (with search from scratch).

Although the incremental search expands three vertices twice, it performs 33 percent fewer expansions than a search from scratch.

Inadmissible heuristics allow HSP to solve search problems in large state spaces by trading off runtime and the plan-execution cost of the resulting plan. SHERPA uses LPA* with consistent heuristics. While we have extended LPA* to use inadmissible heuristics and still guarantee that it expands every vertex at most twice, it turns out to be difficult to make incremental search more efficient than search from scratch with the same inadmissible heuristics, although we have had success in special cases. This can be explained as follows: The larger the heuristics are, the narrower the A* search tree and thus the more efficient A* is. On the other hand, the narrower the A* search tree, the more likely it is that the overlap between the old and new A* search trees is small and thus the less efficient LPA* is.

9.3. Experimental evaluation of heuristic search-based replanning

In the following, we compare SHERPA against search from scratch. 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 savings percentage is defined to be $100(y - x)/y$ [21]. 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

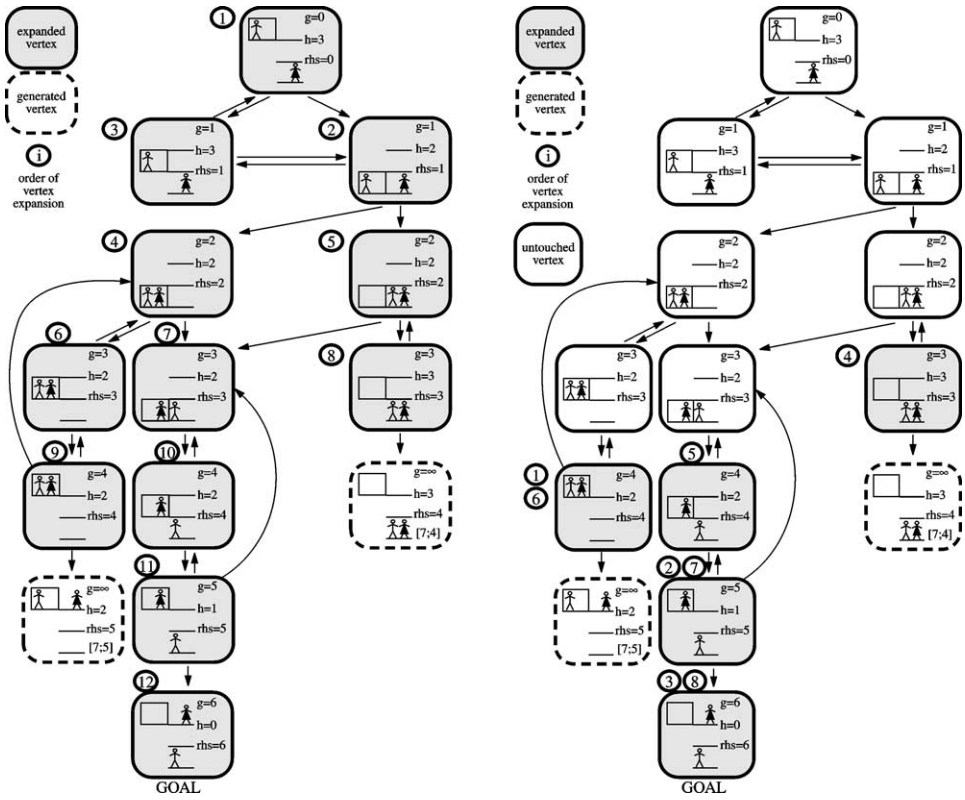

Fig. 15. Second search with search from scratch (left) and incremental search (right).

number of vertex expansions to measure the computational effort of SHERPA. This is justified because our earlier experiment showed that both performance measures were well correlated. As before, 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 incremental search. At this point in time, we don't have results about the runtimes available since we would need very clean code to obtain meaningful results but the software system is rather large.

We used the code of HSP 2.0 [16] to implement SHERPA. 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 path-planning problem. We then randomly 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. Note that deleting a ground planning operator deletes several edges from the state space graph and thus changes the graph substantially. We then used SHERPA twice to solve the resulting modified path-planning problem: one time it used

Domains		Deleted Edges (%)			Sample Size	Average Savings Percentage
		minimum	maximum	average		
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

Fig. 16. Savings percentages of SHERPA over repeated A* searches.

incremental search and the other time it searched from scratch. Since the h_{max}-heuristic depends on the available planning operators, we decided to let SHERPA continue to use the heuristic for the original path-planning problem when it solved the modified one because this enables SHERPA to cache the heuristic values. Caching the heuristic values benefits incremental search and search from scratch equally since computing the heuristics is very time-consuming. No matter whether SHERPA used incremental search or search from scratch, it always found the same plans for the modified path-planning problems and the plans were optimal, which is consistent with our theoretical results about LPA*. Fig. 16 lists the percentage of edges deleted from the state space graph, the number of modified path-planning problems that were solvable, and the savings percentages averaged over all cases where the resulting path-planning problems were solvable and thus the original plan-construction process could indeed be reused. Since the state spaces are large, we approximated the percentage of edges deleted from the state space graph with the percentage of edges deleted from the cached part of the graph. We used a paired-sample z test at the one-percent significance level to confirm that the incremental searches of SHERPA indeed outperform searches from scratch significantly.

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

- Figs. 17–19 show that the savings percentages tend to increase with the size of the three domains. (Figs. 20 and 21 show the same trend.) This is a desirable property since search is time-consuming in large domains and the large savings provided by incremental searches are therefore especially important. The savings percentages in the gripper domain appear to level off at about eighty percent, which is similar to

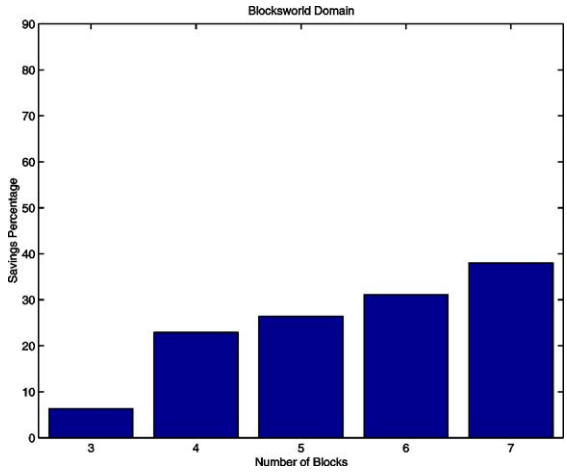

Fig. 17. Blocksworld: average savings percentage as a function of the domain size.

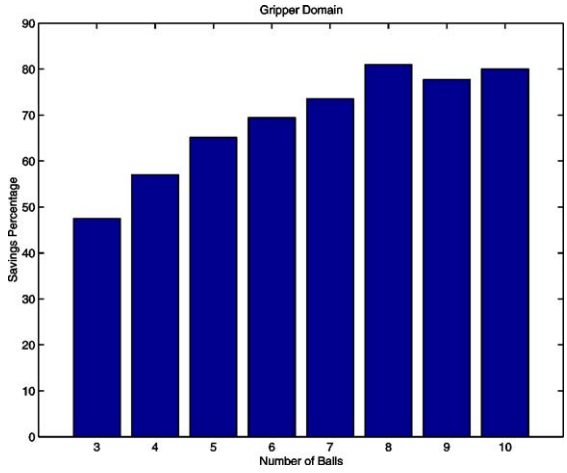

Fig. 18. Gripper: average savings percentage as a function of the domain size.

the savings percentages that [21] reports for PRIAR, a symbolic replanning method, and better than the savings percentages that [21] reports for SPA, another symbolic replanning method. 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.

- Fig. 20 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 path-planning problem. Note that the savings percentages become less reliable as the distance of the deleted ground planning operator to the goal increases because the number of shortest plans in the sample with length larger than n quickly decreases as n increases. The savings percentages decrease as the distance of the deleted ground

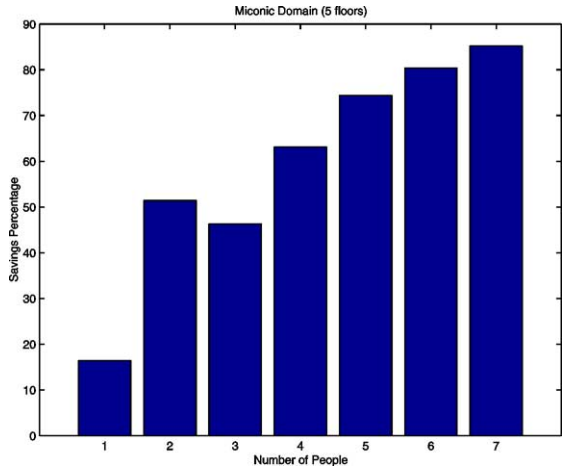

Fig. 19. Miconic: average savings percentage as a function of the domain size.

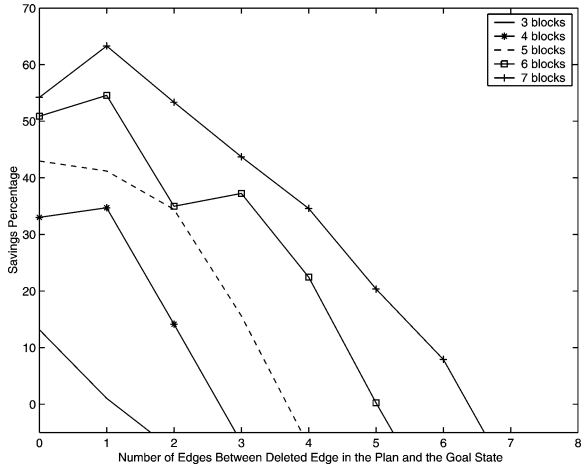

Fig. 20. Blocksworld: average savings percentage as a function of the distance of the deleted edge from the goal.

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.

- Fig. 21 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 planning task

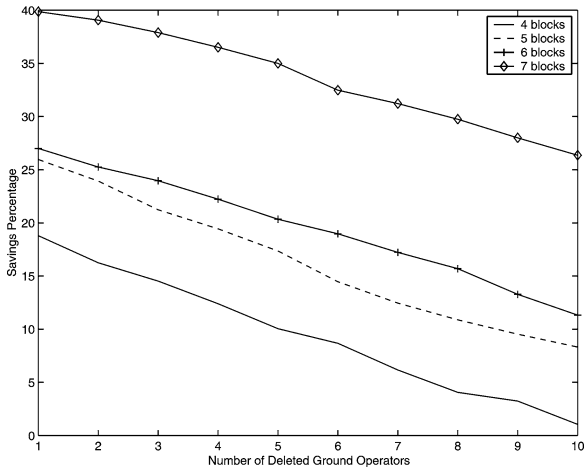

Fig. 21. Blocksworld: average savings percentage as a function of the dissimilarity of the planning tasks.

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 $n = 1 \ldots 10$, 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 $n = 1 \ldots 10$ 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.

10. Related research

A variety of search methods from artificial intelligence, algorithm theory, and robotics share with LPA* the fact that they find solutions to series of similar path-planning problems potentially faster than is possible by solving each path-planning problem from scratch. The idea of incremental search has also been studied in the context of dynamic constraint satisfaction [22–24] and constraint logic programming problems [25]. In the following, however, we focus on path-planning problems:

Symbolic replanning. Symbolic replanning methods from artificial intelligence include case-based planning, planning by analogy, plan adaptation, transformational planning, planning by solution replay, repair-based planning, and learning search-control knowledge. These replanning methods have been used as part of systems such as

CHEF [26], GORDIUS [27], LS-ADJUST-PLAN [28], MRL [29], NoLimit [30], PLEXUS [31], PRIAR [32], and SPA [21]. NoLimit, for example, accelerates a backward-chaining nonlinear planner that uses means-ends analysis, SPA accelerates a causal-link partial-order planner, PRIAR accelerates a hierarchical nonlinear planner, and LS-ADJUST-PLAN accelerates a planner that uses planning graphs. A difference between LPA* and the other replanners is that LPA* 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. Another difference between LPA* and the other replanners is that the quality of the plans of LPA* is as good as the plan quality achieved by using it to search from scratch whereas the quality of the plans of the other replanners can be worse than the plan quality achieved by using them to search from scratch. A third difference between LPA* and some other replanners is that LPA* does not separate replanning into two phases, namely one phase that determines where the previous plan fails and another phase that uses slightly modified standard search methods to replan for those parts. Instead, LPA* 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.

Incremental search. Incremental search methods solve dynamic shortest path problems, that is, path problems where shortest paths have to be determined repeatedly as the topology of a graph or its edge costs change [33]. Thus, they differ from symbolic replanning methods in that they find shortest paths. A number of incremental search methods have been suggested in the algorithms literature [34–45] and, to a much lesser degree, the artificial intelligence literature [46]. They are all uninformed but differ in their assumptions, for example, whether they solve single-source or all-pairs shortest path problems, which performance measure they use, when they update the shortest paths, which kinds of graph topology and edge costs they apply to, and how the graph topology and edge costs are allowed to change over time [47]. If arbitrary sequences of edge insertions, deletions, or weight changes are allowed, then the dynamic shortest path problems are called fully dynamic shortest path problems [48]. LPA* is an incremental search method that solves fully dynamic shortest path problems but, different from the incremental search methods cited above, uses heuristics to focus its search and thus combines two different techniques to reduce its search effort.

Incremental heuristic search. The incremental search method most similar to LPA* is (focussed) D* from robotics [49]. We believe that D* is the first truly incremental heuristic search method. It plans routes for mobile robots that move in initially unknown terrain towards given goal coordinates by searching from the goal coordinates towards the current coordinates of the robots. We have extended LPA* to solve the same path-planning problems as D*, resulting in our D* Lite [50]. This was our original motivation for developing LPA*. D* Lite implements the same navigation strategy as D* but is simpler. For example, it has more than thirty percent fewer lines of code (without any coding tricks), uses only one tie-breaking criterion when comparing priorities, and does not need nested if-statements with complex conditions that occupy up to

three lines each which makes it easier to understand, analyze, optimize, and extend. Furthermore, the theoretical results presented in this article allow us to show a strong similarity of D* Lite to A* and characterize its behavior much better than is currently possible for D*, for which only its correctness has been proven.

Researchers have now started to investigate alternative ways of making A* incremental and thus alternatives to LPA* (personal communication from Peter Yap in 2003), partly by extending idea that have previously been explored in the context of uninformed search [51].

11. Conclusions

Incremental search methods find optimal solutions to series of similar path-planning problems potentially faster than is possible by solving each path-planning problem from scratch. They do this by using information from previous search episodes to speed up later searches. In this article, we developed LPA*, an incremental version of A*, and applied it to route planning and symbolic planning. LPA* applies to path-planning problems where one needs to find shortest paths repeatedly as edges or vertices are added or deleted, or the costs of edges are changed, for example, because the cost of planning operators, their preconditions, or their effects change from one path-planning problem to the next. LPA* builds on previous results from parsing theory and theoretical computer science, namely DynamicSWSF-FP [5]. We modified DynamicSWSF-FP to search from the start vertex to the goal vertex and to stop immediately after it is sure that it has found a shortest path, in which case it becomes an incremental version of breadth-first search. LPA* and DynamicSWSF-FP then both maintain estimates of the start distances of the vertices, use a priority queue to determine in which order to update these estimates, and compute shortest paths based on them. LPA* uses the same notion of local consistency as DynamicSWSF-FP, which it extends by focusing the search. Just like A*, it uses consistent heuristics in the form of approximations of the goal distances of the vertices. Consequently, LPA* combines the advantages of DynamicSWSF-FP (incremental search) and A* (heuristic search) and is thus potentially more efficient than both of them individually. The simplicity of LPA* allowed us to prove various properties about it that demonstrated its efficiency in terms of vertex expansions and showed a strong similarity to A*, which makes it easy to understand, easy to analyze, easy to optimize, and easy to extend. LPA* needs more time per vertex expansion than A* but we were able to show experimentally that LPA* is more efficient than A* in some situations not only in terms of vertex expansions but also in terms of runtime, especially if the path-planning problems change only slightly and the changes are close to the goal. We hope that our analytical and experimental results about LPA* will eventually provide a strong foundation for developing further incremental heuristic search methods and speeding up various artificial intelligence methods. As a first step in this direction, we have applied our LPA* to heuristic search-based replanning, resulting in our SHERPA. LPA* can also be used to develop a simplified version of D* [49], a robot navigation method for unknown terrain [50]. Besides developing a full scale symbolic replanner, it is future work to understand LPA* better, characterize the exact conditions when it is more efficient than A* in terms of runtime, and compare it to search

methods other than breadth-first search, A*, and DynamicSWSF-FP in studies similar to [52,53]. From the results presented in this paper, we are only willing to conclude that incremental heuristic search seems to have an advantage over alternative search methods in some situations and thus is a promising technology that needs to get investigated further. Clearly, we need to improve our understanding of incremental search, including when to prefer incremental search over alternative search methods and which incremental search methods to use, since it is currently unclear how its runtime depends on properties of the search problems as well as low-level implementation and machine details and thus whether it has advantages in situations that are important in practice.

Acknowledgements

Thanks to Anthony Stentz for his support. Without him, this research would not have been possible. Thanks to Peter Yap, Rob Holte, and Jonathan Schaeffer for interesting insight into the behavior of LPA*. Thanks also to Craig Tovey for helpful discussions and to Colin Bauer for helping us to apply LPA* to symbolic planning. This research was performed while the authors were at Georgia Institute of Technology. The Intelligent Decision-Making Group is partly supported by NSF awards to Sven Koenig 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, agencies, companies or the US government.

Appendix A. The proofs

In the following, we prove the theorems stated in the article for the version of LPA* shown in Fig. A.1. All line numbers in the appendix refer to this version of LPA*. The theorems then also hold for the unoptimized version of LPA* stated in the main article since it is a special case where initially $g(s) = \infty$ for all vertices s. This initialization allows for a more efficient implementation since the rhs-value of the start vertex is zero, all other rhs-values are known to be infinity, and the start vertex is known to be the only locally inconsistent vertex and thus the only vertex in the priority queue. More importantly, this initialization allows LPA* to avoid having to iterate over all vertices in Initialize() since the start vertex is the only vertex in the priority queue initially and the other vertices can thus be initialized only after they have been encountered during the search. This is important because the number of vertices can be large and only a few of them might be reached during the search.

All theorems hold no matter how the g-values are initialized by the user before Main() is called. Unless stated otherwise, all theorems also hold not matter whether the termination condition of line {08} or the alternative termination condition "while U is not empty" is used. The heuristics need to be nonnegative and consistent.

In the following, we use $k(u)$ as a shorthand to denote the value returned by CalculateKey(u) and call it the key of vertex $u \in S$. We will show that the key of any vertex

The g-values are initialized by the user before Main() is called.

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 = ∅;
{03} rhs(s_start) = 0;
{04} for all s ∈ S UpdateVertex(s);

procedure UpdateVertex(u)
{05} if (u ≠ s_start) rhs(u) = min_{s'∈pred(u)}(g(s') + c(s', u));
{06} if (u ∈ U) U.Remove(u);
{07} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));

procedure ComputeShortestPath()
{08} while (U.TopKey() ≺ CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal))
{09}    u = U.Pop();
{10}    if (g(u) > rhs(u))
{11}       g(u) = rhs(u);
{12}       for all s ∈ succ(u) UpdateVertex(s);
{13}    else
{14}       g(u) = ∞;
{15}       for all s ∈ succ(u) ∪ {u} UpdateVertex(s);

procedure Main()
{16} Initialize();
{17} forever
{18}    ComputeShortestPath();
{19}    Wait for changes in edge costs;
{20}    for all directed edges (u, v) with changed edge costs
{21}       Update the edge cost c(u, v);
{22}       UpdateVertex(v);
```

Fig. A.1. Lifelong Planning A* (version used in the proofs).

in the priority queue is its priority. Thus, U.TopKey() returns the vertex in the priority queue with the smallest key. However, the key is thus defined for all vertices, while the priority is only defined for the vertices in the priority queue. The subscript $b(u)$ denotes the value of a variable directly before vertex u is expanded, that is, directly before line {09} is executed. Similarly, the subscript $a(u)$ denotes the value of a variable after vertex u is expanded, that is, directly before line {08} is executed again.

Lemma A.1. *The rhs-values of all vertices $u \in S$ always satisfy the following relationship*:

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

Proof. Initialize() initializes the rhs-values so that they satisfy the relationship. The right-hand side of the relationship can then change for a vertex only when the cost of one of its incoming edges changes or the g-value of one of its predecessors changes. This can happen on lines {11}, {14} and {21}. In all of these cases, UpdateVertex() updates the potentially affected rhs-values so that they continue to satisfy the relationship. □

Lemma A.2. *The priority queue contains exactly the locally inconsistent vertices every time line* {08} *is executed.*

Proof. Initialize() initializes the priority queue so that it contains exactly the locally inconsistent vertices. The local consistency of a vertex can then only change when its g-value or its rhs-value changes.

The rhs-value can change only on line {05}. UpdateVertex() then adds the vertex to the priority queue or deletes it from the priority queue, as necessary, immediately afterwards on lines {06-07}. Thus, the theorem continues to hold.

The g-value can change only on lines {11} and {14}.

Whenever ComputeShortestPath() updates the g-value of a locally overconsistent vertex on line {11}, then the g-value of the vertex is set to its rhs-value. The vertex thus becomes locally consistent and is correctly removed from the priority queue. Thus, the theorem continues to hold.

Whenever ComputeShortestPath() updates the g-value of a locally underconsistent vertex on line {14}, then the local consistency of the vertex can change. ComputeShortestPath() then calls UpdateVertex() immediately afterwards on line {15}, which adds the vertex to the priority queue or deletes it from the priority queue, as necessary. Thus, the theorem continues to hold. □

Lemma A.3. *The priority of each vertex* $u \in U$ *is equal to* $k(u)$.

Proof. Whenever a vertex u is inserted into the priority queue, its priority equals its key $k(u)$. Its key can then change only when its g-value or rhs-value changes. This can happen on lines {05}, {11} and {14}. Line {05} can update the rhs-value of a vertex. If vertex u remains locally inconsistent, it is reinserted into the priority queue with priority $k(u)$. Line {11} updates the g-value of a vertex but the vertex is no longer in the priority queue. Finally, line {14} updates the g-value of a vertex u. Directly afterwards, line {15} calls UpdateVertex(u) which updates its rhs-value. If the vertex remains locally inconsistent, it is reinserted into the priority queue with priority $k(u)$. Thus, the relationship continues to hold. □

Lemma A.4. *Assume that vertex* u *has key* $k_{b(u)}(u)$ *and is selected for expansion on line* {09}. *If vertex* v *is locally consistent at this point in time but locally inconsistent the next time line* {08} *is executed, then the new key* $k_{a(u)}(v)$ *of vertex* v *satisfies* $k_{a(u)}(v) \dot{>} k_{b(u)}(u)$ *the next time line* {08} *is executed.*

Proof. Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line {09}. Vertex v is locally consistent at this point in time but locally inconsistent the next time line {08} is executed.

The local consistency of vertex v can only change if its g-value changes or its rhs-value changes. Its rhs-value can change only when the cost of one of its incoming edges changes or the g-value of one of its predecessors changes. The edge costs do not change in ComputeShortestPath(). The g-value of vertex v does not change either. Only the g-value of vertex u changes and the two vertices must be different since vertex u is initially

in the priority queue and thus locally inconsistent whereas vertex v is locally consistent. Consequently, vertex u must be a predecessor of vertex v, and the rhs-value of vertex v changes when the g-value of vertex u changes. We distinguish two cases:

Case one: Vertex u was locally overconsistent. Thus, $g_{b(u)}(u) > rhs_{b(u)}(u)$. The assignment on line {11} decreases the g-value of vertex u since $g_{a(u)}(u) = rhs_{b(u)}(u) < g_{b(u)}(u) \leqslant \infty$. This can affect the rhs-value of vertex v only if $rhs_{a(u)}(v) = g_{a(u)}(u) + c(u, v)$. In this case, the rhs-value of vertex v decreased. Its rhs-value must now be less than its g-value since it was locally consistent before and thus its rhs-value was equal to its g-value, which did not change. Formally, $rhs_{a(u)}(v) < rhs_{b(u)}(v) = g_{b(u)}(v) = g_{a(u)}(v)$. Putting it all together, it holds that

$$\begin{aligned}
k_{a(u)}(v) &\doteq \big[\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(v); \min(g_{a(u)}(v), rhs_{a(u)}(v))\big] \\
&\doteq \big[rhs_{a(u)}(v) + h(v); rhs_{a(u)}(v)\big] \\
&\doteq \big[g_{a(u)}(u) + c(u, v) + h(v); g_{a(u)}(u) + c(u, v)\big] \\
&\dot{>} \big[g_{a(u)}(u) + h(u); g_{a(u)}(u)\big] \\
&\doteq \big[rhs_{b(u)}(u) + h(u); rhs_{b(u)}(u)\big] \\
&\doteq \big[\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(u); \min(g_{b(u)}(u), rhs_{b(u)}(u))\big] \\
&\doteq k_{b(u)}(u).
\end{aligned}$$

We used during the derivation the fact that $c(u, v) + h(v) \geqslant h(u)$ since the heuristics are consistent, and the fact that $g_{a(u)}(u) + c(u, v) > g_{a(u)}(u)$ since the edge cost $c(u, v)$ is positive and the g-value $g_{a(u)}(u)$ is finite.

Case two: Vertex u was locally underconsistent. Thus, $g_{b(u)}(u) < rhs_{b(u)}(u) \leqslant \infty$. The assignment on line {14} increases the g-value of vertex u from a finite value to infinity. This can affect the rhs-value of vertex v only if $rhs_{b(u)}(v) = g_{b(u)}(u) + c(u, v)$. In this case, the rhs-value of vertex v increased. Its rhs-value must now be larger than its g-value since it was locally consistent before and thus its rhs-value was equal to its g-value, which did not change. Formally, $rhs_{a(u)}(v) > rhs_{b(u)}(v) = g_{b(u)}(v) = g_{a(u)}(v)$. Putting it all together, it holds that

$$\begin{aligned}
k_{a(u)}(v) &\doteq \big[\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(v); \min(g_{a(u)}(v), rhs_{a(u)}(v))\big] \\
&\doteq \big[g_{a(u)}(v) + h(v); g_{a(u)}(v)\big] \\
&\doteq \big[rhs_{b(u)}(v) + h(v); rhs_{b(u)}(v)\big] \\
&\doteq \big[g_{b(u)}(u) + c(u, v) + h(v); g_{b(u)}(u) + c(u, v)\big] \\
&\dot{>} \big[g_{b(u)}(u) + h(u); g_{b(u)}(u)\big] \\
&\doteq \big[\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(u); \min(g_{b(u)}(u), rhs_{b(u)}(u))\big] \\
&\doteq k_{b(u)}(u).
\end{aligned}$$

We used during the derivation the fact that $c(u, v) + h(v) \geqslant h(u)$ since the heuristics are consistent, and the fact that $g_{b(u)}(u) + c(u, v) > g_{b(u)}(u)$ since the edge cost $c(u, v)$ is positive and the g-value $g_{b(u)}(u)$ is finite. □

Lemma A.5. *If a locally overconsistent vertex u with key $k_{b(u)}(u)$ is selected for expansion on line* {09}*, then it is locally consistent the next time line* {08} *is executed and its new key $k_{a(u)}(u)$ satisfies $k_{a(u)}(u) = k_{b(u)}(u)$.*

Proof. Assume that a locally overconsistent vertex u is selected for expansion on line {09}. Thus, $\infty \geqslant g_{b(u)}(u) > rhs_{b(u)}(u)$. Its g-value is then set to its rhs-value on line {11} ($g_{a(u)}(u) = rhs_{b(u)}(u)$) and it thus becomes locally consistent. If u is not a successor of itself, then its rhs-value does not change and it thus remains locally consistent. If u is a successor of itself, then the call to UpdateVertex() on line {12} does not change its rhs-value either and it thus remains locally consistent. This follows directly from the definition of the rhs-values if vertex u is the start vertex. Otherwise, it holds that $rhs_{b(u)}(u) = \min_{v \in pred(u)}(g_{b(u)}(v) + c(v,u)) = g_{b(u)}(w) + c(w,u)$ for some vertex $w \neq u$. (Otherwise $rhs_{b(u)}(u) = g_{b(u)}(u) + c(u,u) \geqslant g_{b(u)}(u)$ which would be a contradiction.) Thus, $g_{a(u)}(u) + c(u,u) = rhs_{b(u)}(u) + c(u,u) > rhs_{b(u)}(u) = g_{b(u)}(w) + c(w,u) = g_{a(u)}(w) + c(w,u)$ and consequently $rhs_{a(u)}(u) = \min(g_{a(u)}(w) + c(w,u), g_{a(u)}(u) + c(u,u)) = g_{a(u)}(w) + c(w,u) = rhs_{b(u)}(u) = g_{a(u)}(u)$, which proves the first part of the theorem. Then,

$$\begin{aligned}
k_{a(u)}(u) &\doteq \left[\min(g_{a(u)}(u), rhs_{a(u)}(u)) + h(u); \min(g_{a(u)}(u), rhs_{a(u)}(u))\right] \\
&\doteq \left[rhs_{a(u)}(u) + h(u); rhs_{a(u)}(u)\right] \\
&\doteq \left[rhs_{b(u)}(u) + h(u); rhs_{b(u)}(u)\right] \\
&\doteq \left[\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(u); \min(g_{b(u)}(u), rhs_{b(u)}(u))\right] \\
&\doteq k_{b(u)}(u). \qquad \square
\end{aligned}$$

Lemma A.6. *Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line* {09}*. If vertex v is locally inconsistent at this point in time and remains locally inconsistent the next time line* {08} *is executed, then the new key $k_{a(u)}(v)$ of vertex v satisfies $k_{a(u)}(v) \dot{\geqslant} k_{b(u)}(u)$ the next time line* {08} *is executed.*

Proof. Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line {09}. Vertex v is locally inconsistent at this point in time and remains locally inconsistent the next time line {08} is executed. Since vertex u is expanded instead of vertex v, it holds that $k_{b(u)}(v) \dot{\geqslant} k_{b(u)}(u)$. We consider four cases:

Case one: The key of vertex v does not change. Then, it holds that $k_{a(u)}(v) \doteq k_{b(u)}(v) \dot{\geqslant} k_{b(u)}(u)$.

Case two: The key of vertex v changes, and $v = u$. Vertex $u = v$ was locally underconsistent. (Had it been locally overconsistent, then it would have been locally consistent after its expansion according to Lemma A.5, which violates our assumptions.) The g-value of vertex $v = u$ is then set to infinity and thus $g_{a(u)}(u) \geqslant g_{b(u)}(u)$. Since no other g-value changes, the rhs-value can only change if vertex $v = u$ is a successor of itself. However, it is guaranteed not to decrease since the g-value does not decrease. Thus, it holds that $rhs_{a(u)}(u) \geqslant rhs_{b(u)}(u)$. Putting it all together,

$$\begin{aligned} k_{a(u)}(v) &\doteq k_{a(u)}(u) \\ &\doteq \left[\min(g_{a(u)}(u), rhs_{a(u)}(u)) + h(u); \min(g_{a(u)}(u), rhs_{a(u)}(u))\right] \\ &\dot{\geqslant} \left[\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(u); \min(g_{b(u)}(u), rhs_{b(u)}(u))\right] \\ &\doteq k_{b(u)}(u). \end{aligned}$$

Case three: The key of vertex v changes, $v \neq u$, and vertex u was locally overconsistent. The g-value of vertex v does not change since $v \neq u$. Thus, $g_{a(u)}(v) = g_{b(u)}(v)$. Since the key of vertex v changes, its rhs-value changes and thus vertex v is a successor of vertex u. Vertex u was locally overconsistent and thus $g_{b(u)}(u) > rhs_{b(u)}(u)$. The assignment on line {11} decreases the g-value of vertex u since $g_{a(u)}(u) = rhs_{b(u)}(u) < g_{b(u)}(u) \leqslant \infty$.

This decrease can affect the rhs-value of vertex v only if $rhs_{a(u)}(v) = g_{a(u)}(u) + c(u, v) = rhs_{b(u)}(u) + c(u, v) = \min(g_{b(u)}(u), rhs_{b(u)}(u)) + c(u, v)$. This equality implies both that $rhs_{a(u)}(v) \geqslant \min(g_{b(u)}(u), rhs_{b(u)}(u))$ (since $c(u, v) > 0$) and $rhs_{a(u)}(v) + h(v) = \min(g_{b(u)}(u), rhs_{b(u)}(u)) + c(u, v) + h(v) \geqslant \min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(u)$. (We used during the derivation of the last inequality the fact that $c(u, v) + h(v) \geqslant h(u)$ since the heuristics are consistent.) Putting it all together, it holds that

$$\begin{aligned} &\left[rhs_{a(u)}(v) + h(v); rhs_{a(u)}(v)\right] \\ &\quad \dot{\geqslant} \left[\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(u); \min(g_{b(u)}(u), rhs_{b(u)}(u))\right] \\ &\quad \doteq k_{b(u)}(u). \end{aligned} \tag{A.1}$$

It also holds that

$$\begin{aligned} &\left[g_{a(u)}(v) + h(v); g_{a(u)}(v)\right] \\ &\quad \doteq \left[g_{b(u)}(v) + h(v); g_{b(u)}(v)\right] \\ &\quad \dot{\geqslant} \left[\min(g_{b(u)}(v), rhs_{b(u)}(v)) + h(v); \min(g_{b(u)}(v), rhs_{b(u)}(v))\right] \\ &\quad \doteq k_{b(u)}(v) \\ &\quad \dot{\geqslant} k_{b(u)}(u). \end{aligned} \tag{A.2}$$

Then,

$$\begin{aligned} k_{a(u)}(v) &\doteq \left[\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(v); \min(g_{a(u)}(v), rhs_{a(u)}(v))\right] \\ &\dot{\geqslant} k_{b(u)}(u). \end{aligned}$$

This follows directly from inequality (A.1) if $g_{a(u)}(v) \geqslant rhs_{a(u)}(v)$ and from inequality (A.2) if $g_{a(u)}(v) \leqslant rhs_{a(u)}(v)$.

Case four: The key of vertex v changes, $v \neq u$, and vertex u was locally underconsistent. The g-value of vertex v does not change since $v \neq u$. Thus, $g_{a(u)}(v) = g_{b(u)}(v)$. Since the key of vertex v changes, its rhs-value changes and thus it is a successor of vertex u. However, its rhs-value is guaranteed not to decrease since the g-value of vertex u is set to infinity on line {14} and thus does not decrease. Thus, $rhs_{a(u)}(v) \geqslant rhs_{b(u)}(v)$. Putting it all together,

$$\begin{aligned} k_{a(u)}(v) &\doteq \big[\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(v); \min(g_{a(u)}(v), rhs_{a(u)}(v))\big] \\ &\dot{\geqslant} \big[\min(g_{b(u)}(v), rhs_{b(u)}(v)) + h(v); \min(g_{b(u)}(v), rhs_{b(u)}(v))\big] \\ &\doteq k_{b(u)}(v) \dot{\geqslant} k_{b(u)}(u). \qquad \square \end{aligned}$$

Theorem 1. *The keys of the vertices that* ComputeShortestPath() *selects for expansion on line* {09} *are monotonically nondecreasing over time until* ComputeShortestPath() *terminates.*

Proof. Assume that vertex u is selected for expansion on line {09}. At this point, its key $k_{b(u)}(u)$ is a smallest key of all vertices in the priority queue, that is, of all locally inconsistent vertices according to Lemma A.2. If a locally consistent vertex v becomes locally inconsistent due to the expansion of vertex u, then its new key $k_{a(u)}(v)$ satisfies $k_{a(u)}(v) \dot{>} k_{b(u)}(u)$ according to Lemma A.4. If a locally inconsistent vertex v remains locally inconsistent, then its new key $k_{a(u)}(v)$ satisfies $k_{a(u)}(v) \dot{\geqslant} k_{b(u)}(u)$ according to Lemma A.6. Thus, when the next vertex is selected for expansion on line {09}, its key is at least as large as $k_{b(u)}(u)$. $\square$

Theorem 2. *Let* $k =$ U.TopKey() *during the execution of line* {08}. *If vertex* u *is locally consistent at this point in time with* $k(u) \dot{\leqslant} k$, *then it remains locally consistent until* ComputeShortestPath() *terminates.*

Proof (*By contradiction*). If U is empty, then U.TopKey() returns $[\infty; \infty]$ and thus U.TopKey() $\dot{\geqslant} k(s_{goal})$. Also $rhs(s_{goal}) = g(s_{goal})$ since all vertices are locally consistent. Consequently, the termination condition is satisfied and thus the theorem is trivial. (Similarly, the termination condition is satisfied trivially if the alternative termination condition "while U is not empty" is used.) Thus, we assume that U is not empty.

Assume that vertex u is locally consistent during the execution of line {08}. Let $g(u)$, $rhs(u)$, and $k(u)$ be the g-value, rhs-value, and key of vertex u (respectively) at this point in time. Then, $g(u) = rhs(u)$ since vertex u is locally consistent. Similarly, $k \doteq$ U.TopKey() at this point in time. Assume that $k(u) \dot{\leqslant} k$ and that u becomes locally inconsistent later during the expansion of some vertex v. When v is chosen for expansion, it must be locally inconsistent since only locally inconsistent vertices are expanded. Thus, $v \neq u$. Then, $k_{a(v)}(u) \dot{>} k_{b(v)}(v)$ according to Lemma A.4 and $k_{b(v)}(v) \dot{\geqslant} k$ according to Theorem 1. Consequently,

$$\begin{aligned} &\big[\min(g_{a(v)}(u), rhs_{a(v)}(u)) + h(u); \min(g_{a(v)}(u), rhs_{a(v)}(u))\big] \\ &\quad \doteq k_{a(v)}(u) \dot{>} k_{b(v)}(v) \dot{\geqslant} k \dot{\geqslant} k(u) \\ &\quad \doteq \big[\min(g(u), rhs(u)) + h(u); \min(g(u), rhs(u))\big] \\ &\quad \doteq \big[g(u) + h(u); g(u)\big] \end{aligned}$$

and thus $g_{a(v)}(u) \geqslant \min(g_{a(v)}(u), rhs_{a(v)}(u)) > g(u)$. However, $g_{a(v)}(u) = g(u)$ since vertex u has been locally consistent all the time and thus could not have been assigned a new g-value, which is a contradiction. Consequently, u remains locally consistent until ComputeShortestPath() terminates. $\square$

Theorem 3. *If a locally overconsistent vertex is selected for expansion on line* {09}*, then it is locally consistent the next time line* {08} *is executed and remains locally consistent until* ComputeShortestPath() *terminates.*

Proof. If a locally overconsistent vertex u is selected for expansion on line {09}, then it becomes locally consistent according to Lemma A.5. Let $k =$ U.TopKey() during the execution of line {08} before u is selected for expansion on line {09}, and $k' =$ U.TopKey() during the execution of line {08} after u is selected for expansion on line {09}. Then, $k_{a(u)}(u) \doteq k_{b(u)}(u)$ according to Lemma A.5, $k_{b(u)}(u) \doteq k$ since u was selected for expansion, $k \dot{\leqslant} k'$ according to Theorem 1 if the priority queue is not empty during the execution of line {08} after u is selected for expansion on line {09}, and $k \dot{\leqslant} k'$ if the priority queue is empty since $k' \doteq [\infty; \infty]$. Putting everything together, it holds that $k_{a(u)}(u) \dot{\leqslant} k'$. To summarize, vertex u is locally consistent during the next execution of line {08} after u is selected for expansion on line {09} with $k_{a(u)}(u) \dot{\leqslant} k'$. Consequently, it remains locally consistent until ComputeShortestPath() terminates, according to Theorem 2. □

Lemma A.7. *If line* {08} *is changed to "while U is not empty", then* ComputeShortestPath() *expands each vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent. The g-values of all vertices after termination equal their respective start distances.*

Proof. Assume that line {08} is changed to "while U is not empty". Then, ComputeShortestPath() terminates when all vertices are locally consistent. When a locally overconsistent vertex is selected for expansion, it becomes locally consistent and remains locally consistent according to Theorem 3. Thus, every vertex is expanded at most once when it is locally overconsistent. Similarly, when a locally underconsistent vertex is selected for expansion, its g-value is set to infinity and the vertex can thus only be either locally consistent or overconsistent before it is expanded again. (It cannot be locally underconsistent because its g-value is infinity and cannot be changed before its next expansion.) Thus, if the vertex is expanded again, it must be locally overconsistent. (Locally consistent vertices are not expanded.) As already shown, it then becomes locally consistent and remains locally consistent. To summarize, every vertex is expanded at most twice before all vertices are locally consistent, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and ComputeShortestPath() thus terminates.

When all vertices are locally consistent, then $g(s) = rhs(s) = 0$ if $s = s_{start}$ and $g(s) = rhs(s) = \min_{s' \in pred(s)}(g(s') + c(s', s))$ otherwise. Thus, the g-values satisfy Eq. (1) and thus are equal to the start distances. □

Lemma A.8. *Let $k =$* U.TopKey() *during the execution of line* {08}*. If vertex u is locally consistent at this point in time with $k(u) \dot{\leqslant} k$, then the g-value of state u equals its start distance and one can trace back a shortest path from s_{start} to u by always moving from the current vertex s, starting at u, to any predecessor s' that minimizes $g(s') + c(s', s)$ until s_{start} is reached* (*ties can be broken arbitrarily*)*.*

Proof. If U is empty, then the theorem follows from Lemma A.7. Thus, we assume that U is not empty.

Assume that vertex u is locally consistent during the execution of line {08} with $k(u) \dot{\leqslant} k$. Let $g(s)$, $rhs(s)$, and $k(s)$ be the g-value, rhs-value, and key of any vertex s (respectively) at this point in time. Then, $g(u) = rhs(u)$ since state u is locally consistent, and $k(u) \dot{\leqslant} k$.

We first show by contradiction that $g(u) < \infty$. Assume that $g(u) = \infty$. Then, $g(u) = rhs(u) = \infty$ since u is locally consistent. Thus, $k(u) \doteq [\min(g(u), rhs(u)) + h(u); \min(g(u), rhs(u))] \doteq [\infty; \infty]$. Consequently, $k \doteq [\infty; \infty]$ since $k(u) \dot{\leqslant} k$. Let v be a locally inconsistent vertex with key k. Such a vertex exists since we assume that U is not empty. Then, $g(v) = rhs(v) = \infty$. Thus, vertex v must be locally consistent, which is a contradiction. Consequently, it holds that $g(u) < \infty$.

If $u = s_{start}$ then $g(u) = rhs(u) = 0$ since vertex u is locally consistent and $rhs(u) = 0$ per definition. Thus, $g(u) = g^*(u)$. Furthermore, one can trivially trace back a shortest path from s_{start} to u by always moving from the current vertex s, starting at u, to any predecessor s' that minimizes $g(s') + c(s', s)$ until s_{start} is reached (ties can be broken arbitrarily). Thus, we assume in the following that $u \neq s_{start}$.

Let w be any predecessor of vertex u that minimizes $g(w) + c(w, u)$. We now show that vertex w is locally consistent during the execution of line {08} with $k(w) \dot{\leqslant} k$. It holds that $g(u) = rhs(u) = \min_{s' \in pred(u)}(g(s') + c(s', u)) = g(w) + c(w, u)$. Thus, $g(w) < g(u)$ since $g(u) < \infty$ and $c(w, u) > 0$. Furthermore, $g(w) + h(w) \leqslant g(u) - c(w, u) + c(w, u) + h(u) = g(u) + h(u)$ since the heuristics are consistent and thus $h(w) \leqslant c(w, u) + h(u)$. Consequently,

$$\begin{aligned} k(w) &\doteq \big[\min(g(w), rhs(w)) + h(w); \min(g(w), rhs(w))\big] \\ &\dot{\leqslant} \big[g(w) + h(w); g(w)\big] \\ &\dot{<} \big[g(u) + h(u); g(u)\big] \\ &\doteq \big[\min(g(u), rhs(u)) + h(u); \min(g(u), rhs(u))\big] \\ &\doteq k(u) \dot{\leqslant} k. \end{aligned}$$

Thus, $k(w) \dot{<} k$. This shows that vertex w is locally consistent during the execution of line {08} with $k(w) \dot{\leqslant} k$ since k is the smallest key of any locally inconsistent vertex.

We now show that $g(u) = g^*(u)$ and $g(w) = g^*(w)$ during the execution of line {08}. Both vertices are locally consistent and their keys are less than or equal to the smallest key of any locally inconsistent vertex. Thus, they remain locally consistent and thus their g-values are not updated until ComputeShortestPath() terminates even if line {08} is changed to "while U is not empty", according to Theorem 2. Furthermore, the g-values of vertices u and w equal their respective start distances after termination if line {08} is changed to "while U is not empty", according to Lemma A.7. Thus, $g(u) = g^*(u)$ and $g(w) = g^*(w)$ during the execution of line {08}. These relationships must also hold for the termination condition actually used by LPA* since the values that LPA* assigns to the g-values of vertices do not depend on the termination condition.

We now show that the edge from u to w is the last edge of a shortest path from s_{start} to u. This is indeed the case since $g^*(u) = g(u) = g(w) + c(w, u) = g^*(w) + c(w, u)$. Finally,

we can repeatedly apply this property to show that one can trace back a shortest path from s_{start} to u by always moving from the current vertex s, starting at u, to any predecessor s' that minimizes $g(s') + c(s', s)$ until s_{start} is reached (ties can be broken arbitrarily) since vertex w is again locally consistent with $k(w) \dot{\leqslant} k$. □

Theorems 4 and 5. ComputeShortestPath() *expands a vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, and thus terminates. After* ComputeShortestPath() *terminates, one can trace back a shortest path from s_{start} to s_{goal} by always moving from the current vertex u, starting at s_{goal}, to any predecessor u' that minimizes $g(u') + c(u', u)$ until s_{start} is reached* (*ties can be broken arbitrarily*).

Proof. ComputeShortestPath() terminates after it has expanded every vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent according to Lemma A.7 if line {08} is changed to "while U is not empty". It continues to terminate at least when U is empty even if line {08} is not changed because U.TopKey() then returns $[\infty; \infty]$ and thus U.TopKey() $\dot{\geqslant} k(s_{goal})$ and because $rhs(s_{goal}) = g(s_{goal})$ since all vertices are locally consistent. Thus, the termination condition is satisfied. Because the termination condition does not affect which vertices are expanded and in which order they are expanded, ComputeShortestPath() will terminate after it has expanded every vertex at most twice, namely at most once when it is locally underconsistent and at most once when it is locally overconsistent, if it does not already terminate earlier.

$k \dot{\geqslant} k(s_{goal})$ and $rhs(s_{goal}) = g(s_{goal})$ after termination according to the termination condition, where $k =$ U.TopKey() during the execution of line {08}. Consequently, s_{goal} satisfies the conditions of Lemma A.8 after termination. The theorem then follows directly from Lemma A.8. □

The following theorems show some additional properties of LPA*, including its similarity to a version of A* that always breaks ties among vertices with the same f-values in favor of vertices s that minimize the start distance. (We have also developed a version of LPA* that is similar to a version of A* that always breaks ties among vertices with the same f-values in favor of vertices that maximize the start distance.) These theorems only hold for the termination condition on line {08}. We assume in the proofs that A* terminates when its priority queue is empty, it expands s_{goal}, or it is about to expand a vertex with an infinite f-value. We make use of the following properties (for consistent h-values): First, A* expands every vertex at most once. Second, it expands s_{goal} if its f-value is finite, it expands all vertices u with both $[f(u); g^*(u)] \dot{<} [f(s_{goal}); g^*(s_{goal})]$ and $f(u) < \infty$, and it possibly expands some or all vertices u with both $[f(u); g^*(u)] \doteq [f(s_{goal}); g^*(s_{goal})]$ and $f(u) < \infty$. Third, it expands vertices u in monotonically nondecreasing order of $[f(u); g^*(u)]$. Fourth, it can expand vertices u with the same $[f(u); g^*(u)]$ in any order. Fifth, the g-value and f-value of any vertex u expanded by an A* search are $g(u) = g^*(u)$ and $f(u) = g(u) + h(u) = g^*(u) + h(u)$. In the following, we thus refer to the f-value $f(u)$ of any vertex u as a shorthand for $g^*(u) + h(u)$. The above properties simply follow from the following known properties of A*: The g-values of all expanded vertices equal

their start distances. The f-values of all vertices on the same branch of the search tree of A* are monotonically nondecreasing and their g-values are strictly increasing. Consequently, whenever A* expands a vertex u, its successors on the search tree have f-values that are equal to or larger than the f-value of u and their start distances are larger than the start distance of u. Vertices u with the same $[f(u); g^*(u)]$ are on different branches of the search tree and thus can be expanded by A* in any order desired.

Theorem 6. *Whenever* ComputeShortestPath() *selects a locally overconsistent vertex u for expansion on line* {09}*, then* $k_{b(u)}(u) \doteq [f(u); g^*(u)]$.

Proof. Whenever ComputeShortestPath() selects a locally overconsistent vertex u for expansion, then it becomes locally consistent according to Lemma A.5 and thus $g_{a(u)}(u) = rhs_{a(u)}(u)$. It holds that $k_{b(u)}(u) \doteq k_{a(u)}(u)$ according to Lemma A.5. Furthermore, vertex u remains locally consistent until ComputeShortestPath() terminates according to Theorem 3 and thus its g-value is not updated. The g-value of vertex u equals its start distance after termination if line {08} is changed to "while U is not empty", according to Lemma A.7. Thus, $g_{a(u)}(u) = g^*(u)$. This relationship must also hold for the termination condition actually used by LPA* since the values that LPA* assigns to the g-values of vertices do not depend on the termination condition. Put together,

$$\begin{aligned} k_{b(u)}(u) &\doteq k_{a(u)}(u) \\ &\doteq \left[\min(g_{a(u)}(u), rhs_{a(u)}(u)) + h(u); \min(g_{a(u)}(u), rhs_{a(u)}(u))\right] \\ &\doteq \left[g_{a(u)}(u) + h(u); g_{a(u)}(u)\right] \\ &\doteq \left[g^*(u) + h(u); g^*(u)\right] \\ &\doteq \left[f(u); g^*(u)\right]. \qquad \square \end{aligned}$$

Theorem 10. ComputeShortestPath() *does not expand any vertices whose g-values were equal to their respective start distances before* ComputeShortestPath() *was called.*

Proof (*By contradiction*). We prove the theorem under the assumption that line {08} is changed to "while U is not empty". If line {08} is not changed, then ComputeShortestPath() can only terminate earlier and expands no more vertices than if line {08} is changed. Thus, the theorem continues to hold even if line {08} remains unchanged.

Now assume that ComputeShortestPath() expands vertex u even though its g-value $g_{init}(u)$ before the call to ComputeShortestPath() equals its start distance. Thus, $g_{init}(u) = g^*(u)$.

Consider the first time ComputeShortestPath() expands vertex u. The indices $b(u)$ and $a(u)$ refer to this expansion. Then, $g_{b(u)}(u) = g_{init}(u)$. Since vertex u is locally inconsistent when ComputeShortestPath() selects it for expansion, it holds that $g_{b(u)}(u) \neq rhs_{b(u)}(u)$. It cannot be the case that vertex u is locally overconsistent ($g_{b(u)}(u) > rhs_{b(u)}(u)$) because otherwise $k_{b(u)}(u) \doteq [f(u); g^*(u)]$ according to Theorem 6 and thus $rhs_{b(u)}(u) = \min(g_{b(u)}(u), rhs_{b(u)}(u)) = g^*(u) = g_{init}(u) = g_{b(u)}(u)$, which is a contradiction. Thus, it must be the case that vertex u is locally underconsistent ($g_{b(u)}(u) < rhs_{b(u)}(u)$), which also implies $g^*(u) = g_{init}(u) = g_{b(u)}(u) < rhs_{b(u)}(u) \leqslant \infty$ and thus $g^*(u) < \infty$.

When expanding a locally underconsistent vertex, ComputeShortestPath sets its g-value to infinity. Thus, $g_{a(u)}(u) = \infty > g^*(u)$. Thus, ComputeShortestPath() needs to expand vertex u again at a later time because the g-value of vertex u after termination equals its start distance according to Lemma A.7.

Now consider the second time ComputeShortestPath() expands vertex u. The indices $b'(u)$ and $a'(u)$ refer to this expansion. Vertex u is locally overconsistent when ComputeShortestPath() selects it again for expansion according to Lemma A.7, implying that $g_{b'(u)}(u) > rhs_{b'(u)}(u)$. Also, according to Theorem 6, it holds that $rhs_{b'(u)}(u) = g^*(u)$. Thus,

$$\begin{aligned}
k_{b'(u)}(u) &\doteq \left[\min(g_{b'(u)}(u), rhs_{b'(u)}(u)) + h(u); \min(g_{b'(u)}(u), rhs_{b'(u)}(u))\right] \\
&\doteq \left[rhs_{b'(u)}(u) + h(u); rhs_{b'(u)}(u)\right] \\
&\doteq \left[g^*(u) + h(u); g^*(u)\right] \\
&\doteq \left[g_{b(u)}(u) + h(u); g_{b(u)}(u)\right] \\
&\doteq \left[\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(u); \min(g_{b(u)}(u), rhs_{b(u)}(u))\right] \\
&\doteq k_{b(u)}(u).
\end{aligned}$$

Note that $rhs_{b(u)}(u) > g_{b(u)}(u) = g^*(u) = rhs_{b'(u)}(u)$. Thus, the rhs-value of vertex u decreased between its expansions. This must be due to the g-value of some vertex v that decreased between the expansions of vertex u with $rhs_{b'(u)}(u) = g_{b'(u)}(v) + c(v,u)$. Consequently, $g_{b(u)}(v) > g_{b'(u)}(v)$ and ComputeShortestPath() expands vertex v at least once between the expansions of vertex u since the g-values of vertices change only when they are expanded and $v \neq u$ since $g_{b'(u)}(u) = \infty$ ($g_{b'(u)}(u)$ is infinite) but $g_{b'(u)}(v) < g_{b'(u)}(v) \leqslant \infty$ ($g_{b'(v)}(u)$ is finite).

Now consider the last time ComputeShortestPath() expands vertex v before it expands vertex u the second time. Thus, $g_{a(v)}(v) = g_{b'(u)}(v)$. Since the keys of the vertices that are selected for expansion on line {09} are monotonically nondecreasing over time according to Theorem 1, it must be that $k_{b(u)}(u) \dot{\leqslant} k_{b(v)}(v) \dot{\leqslant} k_{b'(u)}(u)$. Since $k_{b(u)}(u) \doteq k_{b'(u)}(u)$, it must be that $k_{b(u)}(u) \doteq k_{b(v)}(v) \doteq k_{b'(u)}(u)$. However, we now show that this is impossible.

It holds that $g_{b'(u)}(v) < rhs_{b'(u)}(u)$ since $g_{b'(u)}(v) + c(v,u) = rhs_{b'(u)}(u) = g^*(u) < \infty$ and $c(v,u) > 0$. When expanding a locally underconsistent vertex, ComputeShortestPath() sets its g-value to infinity but $g_{a(v)}(v) = g_{b'(u)}(v) < rhs_{b'(u)}(u) < \infty$ and the g-value is thus set to a finite value. Thus, vertex v is locally overconsistent when ComputeShortestPath() selects it for expansion, implying that $g_{b(v)}(v) > rhs_{b(v)}(v)$. When expanding a locally overconsistent vertex, ComputeShortestPath() sets its g-value to its rhs-value. Thus, $g_{a(v)}(v) = rhs_{b(v)}(v)$. Put together,

$$\begin{aligned}
k_{b(v)}(v) &\doteq \left[\min(g_{b(v)}(v), rhs_{b(v)}(v)) + h(v); \min(g_{b(v)}(v), rhs_{b(v)}(v))\right] \\
&\doteq \left[rhs_{b(v)}(v) + h(v); rhs_{b(v)}(v)\right] \\
&\doteq \left[g_{a(v)}(v) + h(v); g_{a(v)}(v)\right] \\
&\dot{\neq} \left[rhs_{b'(u)}(u) + h(u); rhs_{b'(u)}(u)\right] \\
&\doteq \left[\min(g_{b'(u)}(u), rhs_{b'(u)}(u)) + h(u); \min(g_{b'(u)}(u), rhs_{b'(u)}(u))\right] \\
&\doteq k_{b'(u)}(u),
\end{aligned}$$

where we use the fact that $g_{a(v)}(v) \neq rhs_{b'(u)}(u)$. This is a contradiction with $k_{b(v)}(v) \doteq k_{b'(u)}(u)$. Consequently, the theorem holds. □

Lemma A.9. *Whenever* ComputeShortestPath() *selects a vertex u for expansion on line* {09}, *then* $k_{b(u)}(u) \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})] \doteq [g^*(s_{goal}); g^*(s_{goal})]$.

Proof (*By contradiction*). The theorem is trivial if $g^*(s_{goal}) = \infty$ since then $f(s_{goal}) = \infty$ and thus $k_{b(u)}(u) \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})] = [\infty; \infty]$ for all vertices u. Thus, we assume in the following that $g^*(s_{goal}) < \infty$. Assume that ComputeShortestPath() expands a vertex u with $k_{b(u)}(u) \dot{>} [f(s_{goal}); g^*(s_{goal})]$.

Let $k =$ U.TopKey() during the execution of line {08} before u is selected for expansion on line {09}. Thus, $k \doteq k_{b(u)}(u)$. We distinguish two cases:

Case one: It holds that $k_{b(u)}(s_{goal}) \dot{<} k_{b(u)}(u) \doteq k$. In this case, s_{goal} must be locally consistent according to Lemma A.2. ComputeShortestPath() terminates if s_{goal} is locally consistent with $k_{b(u)}(s_{goal}) \dot{\leqslant} k$, which is a contradiction.

Case two: It holds that $k_{b(u)}(s_{goal}) \dot{\geqslant} k_{b(u)}(u)$. In this case, it holds that

$$\begin{aligned}
&\big[g_{b(u)}(s_{goal}); g_{b(u)}(s_{goal})\big] \\
&\quad \doteq \big[g_{b(u)}(s_{goal}) + h(s_{goal}); g_{b(u)}(s_{goal})\big] \\
&\quad \dot{\geqslant} \big[\min(g_{b(u)}(s_{goal}), rhs_{b(u)}(s_{goal})) + h(s_{goal}); \min(g_{b(u)}(s_{goal}), rhs_{b(u)}(s_{goal}))\big] \\
&\quad \doteq k_{b(u)}(s_{goal}) \\
&\quad \dot{\geqslant} k_{b(u)}(u) \\
&\quad \dot{>} \big[f(s_{goal}); g^*(s_{goal})\big] \\
&\quad \doteq \big[g^*(s_{goal}) + h(s_{goal}); g^*(s_{goal})\big] \\
&\quad \doteq \big[g^*(s_{goal}); g^*(s_{goal})\big].
\end{aligned}$$

Thus, $g_{b(u)}(s_{goal}) > g^*(s_{goal})$. Since the g-value of s_{goal} after termination equals $g^*(s_{goal})$ according to Lemma A.8 and its g-value can only change when it is expanded, there exists an expansion of s_{goal} during (if $s_{goal} = u$) or after the expansion of u where the g-value of s_{goal} is set to $g^*(s_{goal})$ and thus $g_{a(s_{goal})}(s_{goal}) = g^*(s_{goal}) < \infty$. If s_{goal} was locally underconsistent directly before this expansion, its g-value would be set to infinity. Thus, s_{goal} is locally overconsistent directly before this expansion. Then, $k_{a(s_{goal})}(s_{goal}) \doteq k_{b(s_{goal})}(s_{goal})$ and $g_{a(s_{goal})}(s_{goal}) = rhs_{a(s_{goal})}(s_{goal})$, both according to Lemma A.5. Thus,

$$\begin{aligned}
k_{b(s_{goal})}(s_{goal}) &\doteq k_{a(s_{goal})}(s_{goal}) \\
&\doteq \big[\min(g_{a(s_{goal})}(s_{goal}), rhs_{a(s_{goal})}(s_{goal})) + h(s_{goal}); \\
&\qquad \min(g_{a(s_{goal})}(s_{goal}), rhs_{a(s_{goal})}(s_{goal}))\big] \\
&\doteq \big[g_{a(s_{goal})}(s_{goal}) + h(s_{goal}); g_{a(s_{goal})}(s_{goal})\big] \\
&\doteq \big[g^*(s_{goal}) + h(s_{goal}); g^*(s_{goal})\big] \\
&\doteq \big[f(s_{goal}); g^*(s_{goal})\big] \\
&\dot{<} k_{b(u)}(u).
\end{aligned}$$

Since $k_{b(s_{goal})}(s_{goal}) \dot{<} k_{b(u)}(u)$, the expansion of s_{goal} cannot coincide with the expansion of u. On the other hand, the expansion of s_{goal} after the expansion of u contradicts Theorem 1. Thus, ComputeShortestPath() expands at most those vertices u with $k_{b(u)}(u) \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$. □

Theorem 8. ComputeShortestPath() *expands at most those locally overconsistent vertices* u *with* $[f(u); g^*(u)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$.

Proof. According to Theorem 6 whenever ComputeShortestPath() selects a locally overconsistent vertex u for expansion, then $k_{b(u)}(u) \doteq [f(u); g^*(u)]$. On the other hand, Lemma A.9 states that $k_{b(u)}(u) \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$. It, thus, follows that $[f(u); g^*(u)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$. □

Theorem 11. ComputeShortestPath() *expands at most those vertices* u *with* $[f(u); g^*(u)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$ *or* $[f_{old}(u); g_{old}(u)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$, *where* $g_{old}(u)$ *is the g-value and* $f_{old}(u) = g_{old}(u) + h(u)$ *is the f-value of vertex* u *directly before the call to* ComputeShortestPath().

Proof. When ComputeShortestPath() selects a vertex u for expansion on line {09}, the vertex is locally inconsistent according to Lemma A.2. We distinguish two cases:

Case one: It holds that $g_{b(u)}(u) > rhs_{b(u)}(u)$, that is, vertex u is locally overconsistent. Then, $[f(u); g^*(u)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$ according to Theorem 8, which proves the theorem.

Case two: It holds that $g_{b(u)}(u) < rhs_{b(u)}(u)$, that is, vertex u is locally underconsistent. Since $k_{b(u)}(u) \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$ according to Lemma A.9, it follows that $[g_{b(u)}(u) + h(u); g_{b(u)}(u)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$. Below we show that it must be the case that vertex u is expanded for the first time. Thus, $g_{old}(u) = g_{b(u)}(u)$ and it follows that $[g_{old}(u) + h(u); g_{old}(u)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$, which proves the theorem.

It remains to be shown that, when a locally underconsistent vertex is expanded, it is the first time that it is expanded. If a locally overconsistent vertex is expanded then it becomes locally consistent and remains locally consistent according to Theorem 3 and thus cannot be expanded again, and a vertex can only be expanded once as locally underconsistent according to Theorem 4. This implies that a vertex that has already been expanded one or more times cannot be expanded again as locally underconsistent. □

Theorem 7. ComputeShortestPath() *expands locally overconsistent vertices with finite f-values in the same order as* A* (*possibly except for vertices* u *with the same keys*), *provided that* A* *always breaks ties among vertices with the same f-values in favor of vertices with smaller start distances and, in case of remaining ties, expands* s_{goal} *last.*

Proof. ComputeShortestPath() expands locally overconsistent vertices u in monotonically nondecreasing order of their keys $[f(u); g^*(u)]$ according to Theorems 1 and 6. Furthermore, it expands at most those locally overconsistent vertices u with $[f(u); g^*(u)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$ according to Theorem 8. A* also expands vertices u in monotonically nondecreasing order of $[f(u); g^*(u)]$ and therefore also expands all vertices u

with $[f(u); g^*(u)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$. Thus, if ComputeShortestPath() first expands locally overconsistent vertex u_1 and then locally overconsistent vertex u_2 and both vertices have finite f-values with $[f(u_1); g^*(u_1)] \dot{\neq} [f(u_2); g^*(u_2)]$, then $[f(u_1); g^*(u_1)] \dot{<} [f(u_2); g^*(u_2)] \dot{\leqslant} [f(s_{goal}); g^*(s_{goal})]$. Thus, A* also first expands vertex u_1 and then vertex u_2. □

Theorem 9. LPA* *shares with* A* *the following property for* s_{goal} *and all vertices* u *that* A* *expands* (*possibly except for vertices with* $[f(u); g^*(u)] \doteq [f(s_{goal}); g^*(s_{goal})]$), *provided that* A* *always breaks ties among vertices with the same f-values in favor of vertices with the smallest start distances and its g-values are assumed to be infinity if* A* *has not calculated them*: *The g-values of these vertices* u *equal their respective start distances after termination and one can trace back a shortest path from* s_{start} *to them by always moving from the current vertex* s, *starting at* u, *to any predecessor* s' *that minimizes* $g(s') + c(s', s)$ *until* s_{start} *is reached* (*ties can be broken arbitrarily*).

Proof. The statement is true for A*. In the following, we prove it for LPA*.

If U is empty after termination, then the g-values of all vertices after termination equal their respective start distances according to Lemma A.7 and the second part of the theorem follows immediately. Thus, we assume that U is not empty.

Let $k =$ U.TopKey() when ComputeShortestPath() terminates. Furthermore, let $g(u)$, $rhs(u)$, and $k(u)$ be the g-value, rhs-value, and key of any vertex u (respectively) after termination. We first show that $g(s_{goal}) = rhs(s_{goal}) = g^*(s_{goal})$. It holds that $g(s_{goal}) = rhs(s_{goal})$ since s_{goal} is locally consistent after termination according to the termination criterion. Furthermore, $k(s_{goal}) \dot{\leqslant} k$ according to the termination condition. Thus, $g(s_{goal}) = rhs(s_{goal}) = g^*(s_{goal})$ according to Lemma A.8.

We now show by contradiction that $k \dot{<} [\infty; \infty]$. Assume that this relationship does not hold and consider any vertex $u \in U$. It holds that $k(u) \dot{\geqslant} k \doteq [\infty; \infty]$. However, $k(u) \doteq [\infty; \infty]$ implies that $\min(g(u), rhs(u)) = \infty$, which in turn implies that $g(u) = rhs(u)$ and thus $u \notin U$ according to Lemma A.2. This is a contradiction and thus it holds that $k \dot{<} [\infty; \infty]$.

We now show that $g^*(s_{goal}) < \infty$. This relationship holds because $k(s_{goal}) \dot{\leqslant} k \dot{<} [\infty; \infty]$ implies that $g(s_{goal}) = rhs(s_{goal}) = g^*(s_{goal}) < \infty$.

We now show by contradiction that every vertex u with $[f(u); g^*(u)] \dot{<} [f(s_{goal}); g^*(s_{goal})]$ also satisfies $g(u) = g^*(u)$. Assume that $[f(u); g^*(u)] \dot{<} [f(s_{goal}); g^*(s_{goal})]$ but $g(u) \neq g^*(u)$. If line {08} is changed to "while U is not empty" then there must be some later expansion of u so that $g_{a(u)}(u) = g^*(u)$ according to Lemma A.7. $g_{a(u)}(u)$ is finite since

$$\begin{aligned} \left[g_{a(u)}(u) + h(u); g_{a(u)}(u)\right] &\doteq \left[g^*(u) + h(u); g^*(u)\right] \\ &\doteq \left[f(u); g^*(u)\right] \\ &\dot{<} \left[f(s_{goal}); g^*(s_{goal})\right] \\ &\dot{\leqslant} [\infty; \infty]. \end{aligned}$$

Thus, u could not have been locally underconsistent when it was selected for expansion on line {09} because then its g-value would have been set to infinity and thus $g_{a(u)}(u) = \infty$.

Thus, u was locally overconsistent when it was selected for expansion on line {09} and thus $g_{b(u)}(u) \dot{>} rhs_{b(u)}(u)$. Consequently, its g-value is set to its rhs-value during its expansion and thus $rhs_{b(u)}(u) = g^*(u)$, which implies that $\min(g_{b(u)}(u), rhs_{b(u)}(u)) = rhs_{b(u)}(u) = g^*(u)$. Thus,

$$\begin{aligned}
k_{b(u)}(u) &\doteq \big[\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(u); \min(g_{b(u)}(u), rhs_{b(u)}(u))\big] \\
&\doteq \big[g^*(u) + h(u); g^*(u)\big] \\
&\doteq \big[f(u); g^*(u)\big] \\
&\dot{<} \big[f(s_{goal}); g^*(s_{goal})\big] \\
&\doteq \big[g^*(s_{goal}) + h(s_{goal}); g^*(s_{goal})\big] \\
&\doteq \big[\min(g(s_{goal}), rhs(s_{goal})) + h(s_{goal}); \min(g(s_{goal}), rhs(s_{goal}))\big] \\
&\doteq k(s_{goal}) \dot{\leqslant} k.
\end{aligned}$$

Since line {08} was changed to "while U is not empty", ComputeShortestPath() will first expand a vertex with priority k and later vertex u with key $k_{b(u)}(u)$. Since $k_{b(u)}(u) \dot{<} k$, the expansion of the vertices cannot coincide. This, however, contradicts Theorem 1. Thus, every vertex with $[f(u); g^*(u)] \dot{<} [f(s_{goal}); g^*(s_{goal})]$ also satisfies $g(u) = g^*(u)$.

We now show that every vertex u with $[f(u); g^*(u)] \dot{<} [f(s_{goal}); g^*(s_{goal})]$ also satisfies $k(u) \dot{<} k(s_{goal})$, as follows:

$$\begin{aligned}
k(u) &\doteq \big[\min(g(u), rhs(u)) + h(u); \min(g(u), rhs(u))\big] \\
&\dot{\leqslant} \big[g(u) + h(u); g(u)\big] \\
&\doteq \big[g^*(u) + h(u); g^*(u)\big] \\
&\doteq \big[f(u); g^*(u)\big] \\
&\dot{<} \big[f(s_{goal}); g^*(s_{goal})\big] \\
&\doteq \big[g^*(s_{goal}) + h(s_{goal}); g^*(s_{goal})\big] \\
&\doteq \big[\min(g(s_{goal}), rhs(s_{goal})) + h(s_{goal}); \min(g(s_{goal}), rhs(s_{goal}))\big] \\
&\doteq k(s_{goal}).
\end{aligned}$$

Finally, every vertex u with $[f(u); g^*(u)] \dot{<} [f(s_{goal}); g^*(s_{goal})]$ also satisfies $k(u) \dot{<} k$ since $k(u) \dot{<} k(s_{goal})$ and $k(s_{goal}) \dot{\leqslant} k$ according to the termination condition. Thus, $k(u) \dot{<} k$ and $g(u) = rhs(u)$ according to Lemma A.2.

If A* breaks ties among vertices with the same f-values in favor of vertices with smaller start distances then it expands all vertices u with $[f(u); g^*(u)] \dot{<} [f(s_{goal}); g^*(s_{goal})]$ and does not expand the vertices u with $[f(u); g^*(u)] \dot{>} [f(s_{goal}); g^*(s_{goal})]$. We have shown that $g(u) = rhs(u)$ and $k(u) \dot{<} k$ if $[f(u); g^*(u)] \dot{<} [f(s_{goal}); g^*(s_{goal})]$. We have also shown that s_{goal} is locally consistent with $k(s_{goal}) \dot{\leqslant} k$. Thus, the theorem follows directly from Lemma A.8. □

References

[1] M. desJardins, E. Durfee, C. Ortiz, M. Wolverton, A survey of research in distributed, continual planning, Artificial Intelligence Magazine 20 (4) (1999) 13–22.
[2] A. Kott, V. Saks, A. Mercer, A new technique enables dynamic replanning and rescheduling of aeromedical evacuation, Artificial Intelligence Magazine 20 (1) (1999) 43–53.
[3] K. Myers, CPEF: A continuous planning and execution framework, Artificial Intelligence Magazine 20 (4) (1999) 63–69.
[4] J. Pemberton, R. Korf, Incremental search algorithms for real-time decision making, in: Proceedings of the International Conference on Artificial Intelligence Planning Systems, Chicago, IL, 1994, pp. 140–145.
[5] G. Ramalingam, T. Reps, An incremental algorithm for a generalization of the shortest-path problem, J. Algorithms 21 (1996) 267–305.
[6] N. Nilsson, Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.
[7] S. Thrun, Lifelong learning algorithms, in: S. Thrun, L. Pratt (Eds.), Learning To Learn, Kluwer Academic, Dordrecht, 1998.
[8] M. Likhachev, S. Koenig, Speeding up the parti-game algorithm, in: S. Becker, S. Thrun, K. Obermayer (Eds.), Advances in Neural Information Processing Systems, vol. 15, MIT Press, Cambridge, MA, 2002.
[9] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, Reading, MA, 1985.
[10] R. Bellman, Dynamic Programming, Princeton University Press, Princeton, NJ, 1957.
[11] B. Nebel, J. Koehler, Plan reuse versus plan generation: A theoretical and empirical analysis, Artificial Intelligence 76 (1–12) (1995) 427–454.
[12] P. Narváez, K. Siu, H. Tzeng, New dynamic algorithms for shortest path tree computation, IEEE/ACM Trans. Networking 8 (6) (2000) 734–746.
[13] L. Buriol, M. Resende, C. Ribeiro, M. Thorup, A memetic algorithm for OSPF routing, in: Proceedings of the INFORMS Telecommunications Conference, 2002, pp. 187–188.
[14] D. McDermott, A heuristic estimator for means-ends analysis in planning, in: Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling, 1996, pp. 142–149.
[15] B. Bonet, G. Loerincs, H. Geffner, A robust and fast action selection mechanism, in: Proc. AAAI-97, Providence, RI, 1997, pp. 714–719.
[16] B. Bonet, H. Geffner, Heuristic search planner 2.0, Artificial Intelligence Magazine 22 (3) (2000) 77–80.
[17] J. Hoffmann, FF: The fast-forward planning systems, Artificial Intelligence Magazine 22 (3) (2000) 57–62.
[18] I. Refanidis, I. Vlahavas, GRT: A domain-independent heuristic for STRIPS worlds based on greedy regression tables, in: Proceedings of the European Conference on Planning, 1999, pp. 346–358.
[19] B. Srivastava, X. Nguyen, S. Kambhampati, M. Do, U. Nambiar, Z. Nie, R. Niganda, T. Zimmerman, AltAlt: Combining graphplan and heuristic state search, Artificial Intelligence Magazine 22 (3) (2000) 88–90.
[20] S. Koenig, D. Furcy, C. Bauer, Heuristic search-based replanning, in: Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling, 2002, pp. 294–301.
[21] S. Hanks, D. Weld, A domain-independent algorithm for plan adaptation, J. Artificial Intelligence Res. 2 (1995) 319–360.
[22] R. Dechter, A. Dechter, Belief maintenance in dynamic constraint networks, in: Proc. AAAI-88, St. Paul. MN, 1988, pp. 37–42.
[23] G. Verfaillie, T. Schiex, Solution reuse in dynamic constraint satisfaction problems, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 307–312.
[24] S. Mittal, B. Falkenhainer, Dynamic constraint satisfaction problems, in: Proc. AAAI-90, Boston, MA, 1990, pp. 25–32.
[25] I. Miguel, Q. Shen, Extending FCSP to support dynamically changing problems, in: Proceedings of IEEE International Fuzzy Systems Conference, 1999, pp. 1615–1620.
[26] K. Hammond, Explaining and repairing plans that fail, Artificial Intelligence 45 (1990) 173–228.
[27] R. Simmons, A theory of debugging plans and interpretations, in: Proc. AAAI-88, St. Paul, MN, 1988, pp. 94–99.
[28] A. Gerevini, I. Serina, Fast plan adaptation through planning graphs: Local and systematic search techniques, in: Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling, 2000, pp. 112–121.

[29] J. Koehler, Flexible plan reuse in a formal framework, in: C. Bäckström, E. Sandewall (Eds.), Current Trends in AI Planning, IOS Press, Amsterdam, 1994, pp. 171–184.
[30] M. Veloso, Planning and Learning by Analogical Reasoning, Springer, Berlin, 1994.
[31] R. Alterman, Adaptive planning, Cognitive Sci. 12 (3) (1988) 393–421.
[32] S. Kambhampati, J. Hendler, A validation-structure-based theory of plan modification and reuse, Artificial Intelligence 55 (1992) 193–258.
[33] G. Ramalingam, T. Reps, On the computational complexity of dynamic graph problems, Theoret. Comput. Sci. 158 (1–2) (1996) 233–277.
[34] G. Ausiello, G. Italiano, A. Marchetti-Spaccamela, U. Nanni, Incremental algorithms for minimal length paths, J. Algorithms 12 (4) (1991) 615–638.
[35] S. Even, Y. Shiloach, An on-line edge deletion problem, J. ACM 28 (1) (1981) 1–4.
[36] S. Even, H. Gazit, Updating distances in dynamic graphs, Meth. Oper. Res. 49 (1985) 371–387.
[37] E. Feuerstein, A. Marchetti-Spaccamela, Dynamic algorithms for shortest paths in planar graphs, Theoret. Comput. Sci. 116 (2) (1993) 359–371.
[38] P. Franciosa, D. Frigioni, R. Giaccio, Semi-dynamic breadth-first search in digraphs, Theoret. Comput. Sci. 250 (1–2) (2001) 201–217.
[39] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni, Fully dynamic output bounded single source shortest path problem, in: Proceedings of the Symposium on Discrete Algorithms, Atlanta, GA, 1996, pp. 212–221.
[40] S. Goto, A. Sangiovanni-Vincentelli, A new shortest path updating algorithm, Networks 8 (4) (1978) 341–372.
[41] G. Italiano, Finding paths and deleting edges in directed acyclic graphs, Inform. Process. Lett. 28 (1) (1988) 5–11.
[42] P. Klein, S. Subramanian, Fully dynamic approximation schemes for shortest path problems in planar graphs, in: Proceedings of the International Workshop on Algorithms and Data Structures, 1993, pp. 443–451.
[43] C. Lin, R. Chang, On the dynamic shortest path problem, J. Inform. Process. 13 (4) (1990) 470–476.
[44] H. Rohnert, A dynamization of the all pairs least cost path problem, in: Proceedings of the Symposium on Theoretical Aspects of Computer Science, Saarbrücken, Germany, 1985, pp. 279–286.
[45] P. Spira, A. Pan, On finding and updating spanning trees and shortest paths, SIAM J. Comput. 4 (1975) 375–380.
[46] S. Edelkamp, Updating shortest paths, in: Proceedings of the European Conference on Artificial Intelligence, Brighton, UK, 1998, pp. 655–659.
[47] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni, Semidynamic algorithms for maintaining single source shortest path trees, Algorithmica 22 (3) (1998) 250–274.
[48] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni, Fully dynamic algorithms for maintaining shortest paths trees, J. Algorithms 34 (2) (2000) 251–281.
[49] A. Stentz, The focussed D* algorithm for real-time replanning, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 1652–1659.
[50] S. Koenig, M. Likhachev, Improved fast replanning for robot navigation in unknown terrain, in: Proceedings of the International Conference on Robotics and Automation, Washington, DC, 2002, pp. 968–975.
[51] M. Al-Ansari, Efficient reinforcement learning in continuous environments, PhD Thesis, College of Computer Science, Northeastern University, Boston, MA, 2001.
[52] C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, U. Nanni, Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study, in: Proceedings of the Workshop on Algorithm Engineering, 2000, pp. 218–229.
[53] G. Proietti, Dynamic maintenance versus swapping: an experimental study on shortest paths trees, in: Proceedings of the Workshop on Algorithm Engineering, 2000, pp. 207–217.