

A Generalized Framework for Lifelong Planning A* Search

Maxim Likhachev

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
maxim+@cs.cmu.edu

Sven Koenig

University of Southern California
Computer Science Department
Los Angeles, CA 90089
skoenig@usc.edu

Abstract

Recently, it has been suggested that Lifelong Planning A* (LPA*), an incremental version of A*, might be a good heuristic search-based replanning method for HSP-type planners. LPA* uses consistent heuristics and breaks ties among states with the same f-values in favor of states with smaller g-values. However, HSP-type planners use inconsistent heuristics to trade off plan-execution cost and planning time. In this paper, we therefore develop a general framework that allows one to develop more capable versions of LPA* and its nondeterministic version Minimax LPA*, including versions of LPA* and Minimax LPA* that use inconsistent heuristics and break ties among states with the same f-values in favor of states with larger g-values. We then show experimentally that the new versions of LPA* indeed speed it up on grids and thus promise to provide a good foundation for building heuristic search-based replanners.

Introduction

Heuristic search-based planning is a recent planning paradigm on which very powerful symbolic planners are based, as first demonstrated by HSP and its successor HSP 2.0 (Bonet & Geffner 2001). Heuristic search-based planners typically assume that planning is a one-shot process. In reality, however, planning is often a repetitive process where one needs to solve series of similar planning tasks, for instance, because the situation changes over time. An important example is the aeromedical evacuation of injured people in crisis situations where aftershocks can destroy additional air fields (Kott, Saks, & Mercer 1999). Researchers have therefore developed replanning and plan reuse techniques that reuse information from previous planning episodes to solve series of similar planning tasks much faster than is possible by solving each planning task from scratch. Examples include case-based planning, planning by analogy, plan adaptation, transformational planning, planning by solution replay, repair-based planning, and learning search-control knowledge. Recently, the SHERPA replanner (Koenig, Furcy, & Bauer 2002) demonstrated that Lifelong Planning A* (LPA*) (Koenig & Likhachev 2002b) promises to be a good heuristic search-based replanning method for HSP-type planners. LPA* is an incremental version of A* that uses consistent heuristics and breaks ties among states with the same f-values in favor of states with smaller g-values. Since LPA* uses consistent heuristics, it finds minimum-cost plans. However, planners

that find minimum-cost plans do not scale up to large domains. Heuristic search-based planners therefore use inconsistent heuristics to trade off plan-execution cost and planning time. Unfortunately, LPA* with inconsistent heuristics is neither efficient nor correct. In this paper, we therefore develop Generalized LPA* (GLPA*), a framework that generalizes LPA* and allows one to develop more capable versions of LPA* and its nondeterministic version Minimax LPA* (Likhachev & Koenig 2003), including versions of LPA* and Minimax LPA* that use inconsistent heuristics and break ties among states with the same f-values in favor of states with larger g-values. We prove that the new versions of LPA* expand every state at most twice and terminate, just like the original version. We also derive bounds on the plan-execution cost of the resulting plans. We then show experimentally that the new versions of LPA* indeed speed it up on grids and thus can speed up A* searches.

Lifelong Planning A* (LPA*)

We first describe Lifelong Planning A* (LPA*) (Koenig & Likhachev 2002b), an incremental version of A* that uses consistent heuristics and breaks ties among states with the same f-values in favor of states with smaller g-values. LPA* repeatedly determines a minimum-cost plan from a given start state to a given goal state in a deterministic domain while some of the action costs change. All variants of LPA* and A* described in this paper search from the goal state to the start state (backward search), with the exception of Figure 12. A more detailed description of LPA* than the one given below can be found in (Koenig, Likhachev, & Furcy 2004), that also contains proofs that LPA* finds minimum-cost plans (in form of paths) efficiently, for example, that the first search of LPA* is identical to an A* search with the same heuristics and tie-breaking strategy and that LPA* expands every state at most twice per search episode.

Notation In the following, S denotes the finite set of states of the domain. $A(s)$ denotes the finite set of actions that can be executed in state $s \in S$. $\text{succ}(s, a) \in S$ denotes the successor that results from executing action $a \in A(s)$ in state $s \in S$. Similarly, $\text{Pred}(s) := \{s' \in S \mid s = \text{succ}(s', a) \text{ for some } a \in A(s')\}$ denotes the set of predecessors of state $s \in S$. $0 < c(s, a) \leq \infty$ denotes the cost of executing action $a \in A(s)$ in state $s \in S$. LPA* always determines a minimum-cost plan from a given start state $s_{start} \in S$ to a given goal state $s_{goal} \in S$, knowing both the domain and the current action costs, where a minimum-cost plan is defined to minimize the sum of the costs of the executed actions

The pseudo code uses the following functions to manage the priority queue: $U.TopKey()$ returns the smallest priority of all states in priority queue U . (If U is empty, then $U.TopKey()$ returns the largest possible priority, that is $[\infty; \infty]$.) $U.Pop()$ deletes the state with the smallest priority in priority queue U and returns the state. $U.Insert(s, k)$ inserts state s into priority queue U with priority k . $U.Remove(s)$ removes state s from priority queue U . Finally, $U.Update(s, k)$ sets the priority of state s in the priority queue to k .

```

procedure Initialize()
{01}  $U = \emptyset$ ;
{02} for all  $s \in S$   $rhs(s) = g(s) = \infty$ ;
{03}  $rhs(s_{goal}) = 0$ ;
{04}  $UpdateState(s_{goal})$ ;
procedure UpdateState( $u$ )
{05} if ( $u \neq s_{goal}$ )  $rhs(u) = \min_{a \in A(u)} (c(u, a) + g(succ(u, a)))$ ;
{06} if ( $u \in U$  and  $g(u) \neq rhs(u)$ )  $U.Update(u, K(u))$ ;
{07} else if ( $u \in U$  and  $g(u) = rhs(u)$ )  $U.Remove(u)$ ;
{08} else if ( $u \notin U$  and  $g(u) \neq rhs(u)$ )  $U.Insert(u, K(u))$ ;
procedure ComputePlan()
{09} while ( $U.TopKey() < K(s_{start})$  or  $rhs(s_{start}) \neq g(s_{start})$ )
{10}    $u = U.Pop()$ ;
{11}   if ( $g(u) > rhs(u)$ )
{12}      $g(u) = rhs(u)$ ;
{13}     for all  $s \in Pred(u)$   $UpdateState(s)$ ;
{14}   else
{15}      $g(u) = \infty$ ;
{16}     for all  $s \in Pred(u) \cup \{u\}$   $UpdateState(s)$ ;
procedure Main()
{17}  $Initialize()$ ;
{18} forever
{19}    $ComputePlan()$ ;
{20}   Wait for changes in action costs;
{21}   for all actions with changed action costs  $c(u, a)$ 
{22}     Update the action cost  $c(u, a)$ ;
{23}    $UpdateState(u)$ ;

```

Figure 1: Backward Version of Lifelong Planning A*.

(plan-execution cost). Since the domain is deterministic, the plan is a sequence of actions.

Local Consistency LPA* maintains two kinds of estimates of the goal distance $gd(s)$ of each state s : a g-value $g(s)$ and an rhs-value $rhs(s)$. The rhs-value of a state is based on the g-values of its successors and thus potentially better informed than the g-values. It always satisfies the following relationship for all states s (**Invariant 1**):

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{a \in A(s)} (c(s, a) + g(succ(s, a))) & \text{otherwise.} \end{cases}$$

A state s is called locally consistent if $g(s) = rhs(s)$, otherwise it is called locally inconsistent. If all states are locally consistent then all of their g-values are equal to their respective goal distances, which allows one to find minimum-cost plans from any state to the goal state greedily. However, LPA* does not maintain the local consistency of every state after each search episode. Instead, it uses heuristics $h(s)$ to focus the search and computes only the g-values that are relevant for computing a minimum-cost plan from the start state to the goal state. $h(s)$ approximates the cost of a minimum-cost plan between the start state and state s . The heuristics need to be nonnegative and (backward) consistent, that is, obey the triangle inequality $h(s_{start}) = 0$ and $h(s) \leq h(s') + c(s', a)$ for all states $s \in S$, states $s' \in Pred(s)$ and actions $a \in A(s')$ with $s = succ(s', a)$.

Priority Queue LPA* maintains a priority queue that always contains exactly the locally inconsistent states (**Invariant 2**). These are the states whose g-values LPA* potentially needs to change to make them locally consistent. The priority $K(s)$ of state s in the priority queue is always a vector with two components: $K(s) = [K_1(s); K_2(s)]$, where $K_1(s) = \min(g(s), rhs(s)) + h(s)$ and $K_2(s) = \min(g(s), rhs(s))$ (**Invariant 3**). Thus, if its g-value or rhs-value changes, then its priority needs to get re-computed. Priorities are compared according to a lexicographic ordering. For example, a priority $K = [K_1; K_2]$ is smaller than or equal to a priority $K' = [K'_1; K'_2]$ iff either $K_1 < K'_1$ or ($K_1 = K'_1$ and $K_2 \leq K'_2$).

Pseudo Code The main function $Main()$ in the pseudo code of LPA* in Figure 1 first calls $Initialize()$ to initialize the variables {17}. (Numbers in curly brackets refer to line numbers in the pseudo code.) $Initialize()$ sets the initial g-values of all states to infinity and sets their rhs-values to satisfy Invariant 1 {02-03}. Thus, the goal state is initially the only locally inconsistent state and is inserted into the otherwise empty priority queue {04}. In an actual implementation, $Initialize()$ only needs to initialize a state when it encounters it during the search and thus does not need to initialize all states up front. $Main()$ then calls $ComputePlan()$ to find a minimum-cost plan. $ComputePlan()$ repeatedly removes the state with the smallest priority from the priority queue and recalculates its g-value (“expands the state”) {10-16}. It thus expands the locally inconsistent states in nondecreasing order of their priorities. A locally inconsistent state s is called locally over-consistent iff $g(s) > rhs(s)$. When $ComputePlan()$ expands a locally over-consistent state s {12-13}, then it holds that $rhs(s) = gd(s)$, which implies that $K(s) = [f(s); gd(s)]$, where $f(s) = gd(s) + h(s)$. During the expansion of the state, $ComputePlan()$ sets the g-value of the state to its rhs-value and thus its goal distance {12}, which is the desired value and also makes the state locally consistent. Its g-value then no longer changes until $ComputePlan()$ terminates. A locally inconsistent state s is called locally under-consistent iff $g(s) < rhs(s)$. When $ComputePlan()$ expands a locally under-consistent state {15-16}, then it simply sets the g-value of the state to infinity {15}. This makes the state either locally consistent or over-consistent. If the expanded state was locally over-consistent, then the change of its g-value can affect the local consistency of its predecessors {13}. Similarly, if the expanded state was locally under-consistent, then it and its predecessors can be affected {16}. $ComputePlan()$ therefore calls $UpdateState()$ for each potentially affected state to ensure that Invariants 1-3 to continue to hold. $UpdateState()$ updates the rhs-value of the state to ensure that Invariant 1 holds {05}, adds it to or removes it from the priority queue (if needed) to ensure that Invariant 2 holds {07-08}, and updates its priority (if the state remains in the priority queue) to ensure that Invariant 3 holds {06}. LPA* expands states until the start state is locally consistent and the priority of the state to expand next is no smaller than the priority of the start state. If $g(s_{start}) = \infty$ after the search, then there is no plan from the start state to the goal state with finite plan-execution cost. Otherwise, one can find a minimum-cost plan by starting in s_{start} and always executing the action $\arg \min_{a \in A(s)} (c(s, a) + g(succ(s, a)))$ in the

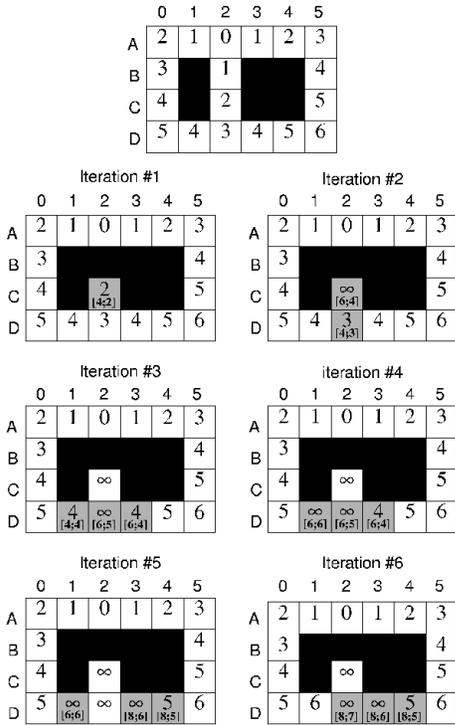
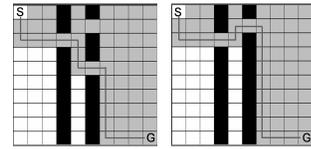


Figure 2: An Example (Koenig, Furcy, & Bauer 2002).

current state s until $s = s_{goal}$ (ties can be broken arbitrarily). Main() then waits for changes in action costs {20}. To maintain Invariants 1-3 if some action costs have changed, it calls UpdateState() {23} for the states potentially affected by the changed action costs, and finally recalculates a minimum-cost plan by calling ComputePlan() again {19}, and iterates.

Example We use the four-connected grid in Figure 2 to demonstrate how LPA* operates. The task is to find a path from the start cell D1 to the goal cell A2 once (top figure) and then a second time after cell B2 becomes blocked (remaining figures). All edge costs are either one (if the edges connect two unblocked cells) or infinity. We use the sum of the absolute difference of the x coordinates of two cells and the absolute difference of their y coordinates (Manhattan distance) as heuristic approximation of their distance, resulting in (forward and backward) consistent heuristics. Assume that the g-values were computed as shown in the top figure during initial planning, making all cells locally consistent. (A cell is locally consistent iff its g-value equals its rhs-value, which is computed based on the g-values of the neighboring cells.) Consequently, the g-values of all cells are equal to their respective goal distances. We now block cell B2, which makes cell C2 locally underconsistent and thus locally inconsistent. Locally inconsistent cells are shown in grey in the figure together with their priorities. During each iteration, LPA* now expands one locally inconsistent state with the smallest priority. It sets the g-value of the state to infinity if the state is locally underconsistent and to its rhs-value if it is locally overconsistent. In the second case, it turns out that the new g-value of the state is equal to its goal distance. Thus, LPA*



(a) 1st search $exp=56, max=1$ (b) 2nd search $exp=53, max=1$

Figure 3: (a,b): A* with consistent heuristics (Manhattan Distance) and bad tie-breaking (A*-TB1). exp is the number of state expansions, and max is the maximum number of expansions of the same state.

expands cell C2 during its first iteration and sets its g-value to infinity, which makes cell D2 locally underconsistent since its rhs-value depends on the g-value of cell C2. This way, LPA* propagates local inconsistencies from cell to cell. The propagation stops at cells D0 and D5 since their rhs-values do not depend on the g-values of cells that LPA* has changed. LPA* does not expand all cells but a sufficient number of them to guarantee that one is able to trace a shortest path from the start cell to the goal cell by starting at the start cell and always greedily decreasing the g-value of the current cell. This example demonstrates how LPA* expands each cell at most twice (namely once as locally underconsistent and once as locally overconsistent), a property that is only guaranteed to hold if the heuristics are (backward) consistent.

Naive Attempts at Extending LPA*

LPA* uses consistent heuristics and breaks ties among states with the same f-values in favor of states with smaller g-values. It is known, however, that A* tends to expand fewer states if it uses inconsistent heuristics and breaks ties among states with the same f-values in favor of states with larger g-values. We try to apply the same ideas to LPA* since the first search of LPA* is the same as that of A* with the same heuristics and tie-breaking strategy, and both search algorithms thus expand the same number of states. As example domain, we use the four-connected grid from Figure 3. The task is to find a path from the start cell S in the upper-left corner to the goal cell G in the lower-right corner once and then a second time after a cell on the previous cost-minimal path is blocked. All edge costs are again either one or infinity. Our baseline algorithm is A* that uses consistent heuristics and breaks ties among states with the same f-values in favor of states with smaller g-values, like LPA*. We refer to this tie-breaking strategy as “bad tie-breaking” even though, in many cases, it does not matter how ties are broken and all tie-breaking strategies are then equally effective. We refer to A* with this tie-breaking strategy as A*-TB1. A*-TB1 with consistent heuristics is guaranteed to find a minimum-cost plan. In the example domain, remaining ties are broken in the order W, N, E and S. We use again the Manhattan distance as heuristic approximation of the distances. Figure 3(a) shows (in grey) the cells expanded by A*-TB1 with the Manhattan distance, and Figure 3(b) shows the cells expanded by the same search algorithm after we blocked one of the cells on the initial path and ran it again. Remem-

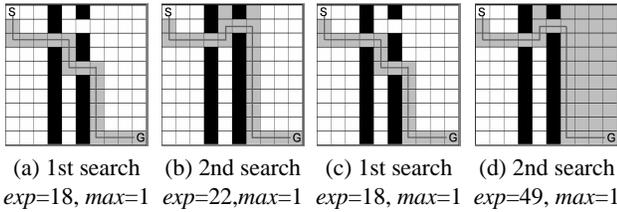


Figure 4: (a,b): A* with inconsistent heuristics (three times the Manhattan Distance) and bad tie-breaking (A*-TB1). (c,d): A* with consistent heuristics (Manhattan Distance) and good tie-breaking (A*-TB2).

ber that all variants of LPA* and A* described in this paper search from the goal state to the start state, with the exception of Figure 12.

Inconsistent Heuristics

A* can be used with inconsistent heuristics without any changes to the search algorithm. For heuristic search problems, these inconsistent heuristics are often derived from consistent heuristics by multiplying all heuristics with the same constant $\epsilon > 1$ to get them closer to the values they are supposed to approximate (resulting in weighted heuristics). A* then trades off plan-execution cost and planning time and thus tends to expand fewer states than A* with consistent heuristics, a property used by HSP-type planners (Bonet & Geffner 2001). While A* is no longer guaranteed to find a minimum-cost plan, it still finds a plan whose plan-execution cost is at most a factor of ϵ larger than minimal (Pearl 1985). Figure 4(a) shows the cells expanded by A*-TB1 with the Manhattan distance multiplied by three, and Figure 4(b) shows the cells expanded by the same search algorithm after we blocked the cell on the path and ran it again. It expands many fewer states than A*-TB1 with the Manhattan distance, demonstrating the advantage of inconsistent heuristics.

One could be tempted to use LPA* with inconsistent heuristics without any code changes. Figure 5(a) shows the cells expanded by LPA* with the Manhattan distance multiplied by three, and Figure 5(b) shows the cells expanded by the same search algorithm after we blocked the cell on the path and ran it again. During the second search, it expands many cells more than twice (shown in dark grey) and some states even ten times, different from LPA* with consistent heuristics that is guaranteed to expand each state at most twice. It expands almost five times more states than A*-TB1 with the same heuristics and almost two times more states than A*-TB1 with the Manhattan distance. Thus, LPA* with inconsistent heuristics is less efficient than search from scratch. Even worse, it turns out that LPA* with inconsistent heuristics can return a plan of infinite plan-execution cost even if a plan of finite plan-execution cost exists, different from A* with inconsistent heuristics that is guaranteed to return a plan of finite plan-execution cost if possible. Thus, LPA* with inconsistent heuristics is not even correct. LPA* thus cannot be used with inconsistent heuristics without any changes to the search algorithm.

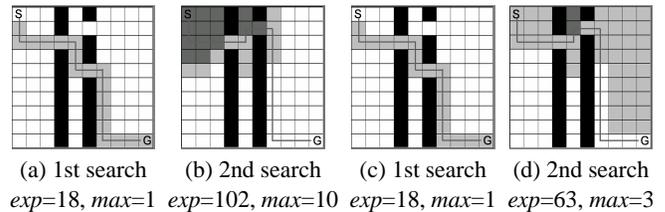


Figure 5: (a,b): A naive version of LPA* with inconsistent heuristics (three times the Manhattan Distance) and bad tie-breaking. (c,d): A naive version of LPA* with consistent heuristics (Manhattan Distance) and good tie-breaking.

Good Tie-Breaking

A* that breaks ties among states with the same f-values in favor of larger g-values tends to expand fewer states than A* that breaks ties in the opposite direction, for example when there are several minimum-cost plans from the start state to the goal state and a large number of states on these plans have the same f-value as the goal state. This case occurs frequently on grids. We refer to this tie-breaking strategy as “good tie-breaking” and to A* with this tie-breaking strategy as A*-TB2. A*-TB2 with consistent heuristics is guaranteed to find a minimum-cost plan. Figure 4(c) shows the cells expanded by A*-TB2 with the Manhattan distance, and Figure 4(d) shows the cells expanded by the same search algorithm after we blocked the cell on the path and ran it again. It expands fewer states than A*-TB1 with the Manhattan distance, demonstrating the advantage of good tie-breaking.

One could be tempted to create a version of LPA* with good tie-breaking as follows: LPA* always expands the state with the smallest priority in its priority queue. The priority $K(s)$ of state s is a two element vector $K(s) = [K_1(s); K_2(s)]$, where $K_1(s) = \min(g(s), rhs(s)) + h(s)$ and $K_2(s) = \min(g(s), rhs(s))$. Thus, the first element of the priority corresponds to the f-value of state s and the second one corresponds to the g-value of state s . A naive attempt at creating a version of LPA* with good tie-breaking is to change the priority of state s to $K(s) = [K_1(s); -K_2(s)]$. Figure 5(c) shows the cells expanded by our naive version of LPA* with the Manhattan distance. However, it did not replan at all after we blocked the cell on the path and ran it again because no inconsistent state had a smaller priority than the start state. Consequently, it returned a plan of infinite plan-execution cost even though a plan of finite plan-execution cost exists. It is thus incorrect. We fixed this problem by forcing it to expand states until it found a plan with finite plan-execution cost from the start state to the goal state. Figure 5(d) shows the cells expanded by this version of LPA* with the Manhattan distance after we blocked the cell on the path and ran it again. It expands some states more than twice, different from LPA* with consistent heuristics that is guaranteed to expand each state at most twice. It expands more states than A*-TB2 with the same heuristics and more states than A*-TB1 with the same heuristics and therefore is less efficient than search from scratch. This behavior can be explained as follows: LPA* with good tie-breaking can repair some part A

of its search tree and only then detect that it also needs to repair a part of its search tree closer to its root. After it has repaired this part of its search tree, the best way of repairing part A might have changed and it then has to repair part A again, possibly resulting in a large number of expansions of the states in part A. Thus, our naive versions of LPA* with good tie-breaking are either incorrect or inefficient. The easiest way of avoiding this problem is to repair the search tree from the root to the leaves, which is exactly what LPA* with bad tie-breaking does.

Generalized Lifelong Planning A* (GLPA*)

We now develop Generalized LPA* (GLPA*), a framework that generalizes LPA* and its nondeterministic version Minimax LPA* (Likhachev & Koenig 2003) and allows one to develop more capable versions of LPA* and Minimax LPA*, including versions of LPA* and Minimax LPA* that use inconsistent heuristics and good tie-breaking.

Notation We need to generalize some definitions since GLPA* can operate in nondeterministic domains. $Succ(s, a) \subseteq S$ denotes the set of states that can result from executing action $a \in A(s)$ in state s . This set contains only one element in deterministic domains, namely the state $succ(s, a)$. Similarly, $Pred(s) := \{s' \in S \mid s \in Succ(s', a) \text{ for some } a \in A(s')\}$ denotes the set of predecessors of state $s \in S$. $0 < c(s, a, s') \leq \infty$ denotes the cost of executing action $s \in A(s)$ in state $s \in S$ if the execution results in state $s' \in S$. Then, the smallest possible plan-execution cost of any plan from state $s \in S$ to state $s' \in S$ is defined recursively in the following way:

$$c^*(s, s') = \begin{cases} 0 & \text{if } s = s' \\ \min_{a \in A(s)} \min_{s'' \in Succ(s, a)} (c(s, a, s'') + c^*(s'', s')) & \text{otherwise.} \end{cases}$$

GLPA* always determines a best plan (in form of a policy) according to a given optimization criterion from the given start state to the given goal state, knowing both the domain and the current action costs. It gives one a considerable amount of freedom when it comes to specifying the optimization criterion. We borrow some notion from reinforcement learning to explain how this is done: Assume that every state $s \in S$ has a V-value $V(s)$ associated with it. Every state-action pair $s \in S$ and $a \in A(s)$ then has a value $Q_V(s, a) > 0$ associated with it that is calculated from the action costs $c(s, a, s')$ and the values $V(s')$ for all states $s' \in Succ(s, a)$. (We call these values $Q_V(s, a)$ because they are similar to Q-values from reinforcement learning.) Thus, there is a function F such that $Q_V(s, a) = F(c(s, a, \cdot), V(\cdot))$. This function can be chosen arbitrarily subject only to the following three restrictions on the values $Q_V(s, a)$ for all states $s \in S$ and actions $a \in A(s)$ (**Restriction 1**):

- $Q_V(s, a)$ cannot decrease if $V(s')$ for one $s' \in Succ(s, a)$ increases and $V(s'')$ for all $s'' \in Succ(s, a) - \{s'\}$ remains unchanged,
- $\epsilon Q_V(s, a) \geq Q_{\epsilon V}(s, a)$ for all $\epsilon > 0$, and
- $Q_V(s, a) \geq \max_{s' \in Succ(s, a)} (c(s, a, s') + V(s'))$.

We then also define the values $Q_g(s, a) = F(c(s, a, \cdot), g(\cdot))$ and $Q_{gd}(s, a) = F(c(s, a, \cdot), gd(\cdot))$

The pseudocode uses the following functions to manage the priority queue: $U.TopKey()$ returns the smallest priority of all states in priority queue U . (If U is empty, then $U.TopKey()$ returns the largest possible priority.) $U.Pop()$ deletes the state with the smallest priority in priority queue U and returns the state. $U.Insert(s, k)$ inserts state s into priority queue U with priority k . $U.Remove(s)$ removes state s from priority queue U . Finally, $U.Update(s, k)$ sets the priority of state s in the priority queue to k . The predicate $NotYet(s)$ is a shorthand for “state s has not been expanded yet as overconsistent during the current call to $ComputePlan()$.”

```

procedure Initialize()
{01}  $U = \emptyset$ ;
{02} for all  $s \in S$   $rhs(s) = g(s) = \infty$ ;
{03}  $rhs(s_{goal}) = 0$ ;
{04}  $UpdateState(s_{goal})$ ;

procedure UpdateState( $u$ )
{05} if ( $u \neq s_{goal}$ )  $rhs(u) = \min_{a \in A(u)} Q_g(u, a)$ ;
{06} if ( $u \in U$  and  $g(u) \neq rhs(u)$ )  $U.Update(u, K(u))$ ;
{07} else if ( $u \in U$  and  $g(u) = rhs(s)$ )  $U.Remove(u)$ ;
{08} else if ( $u \notin U$  and  $g(u) \neq rhs(u)$  and  $NotYet(u)$ )  $U.Insert(u, K(u))$ ;

procedure ComputePlan()
{09} while ( $U.TopKey() < K(s_{start})$  or  $rhs(s_{start}) \neq g(s_{start})$ )
{10}    $u = U.Pop()$ ;
{11}   if ( $g(u) > rhs(u)$ )
{12}      $g(u) = rhs(u)$ ;
{13}   for all  $s \in Pred(u)$   $UpdateState(s)$ ;
{14}   else
{15}      $g(u) = \infty$ ;
{16}   for all  $s \in Pred(u) \cup \{u\}$   $UpdateState(s)$ ;

procedure Main()
{17}  $Initialize()$ ;
{18} forever
{19}    $ComputePlan()$ ;
{20}   for all inconsistent states  $s \notin U$   $U.Insert(s, K(s))$ ;
{21}   Wait for changes in action costs;
{22}   for all actions with changed action costs  $c(u, a, v)$ 
{23}      $Update$  the action cost  $c(u, a, v)$ ;
{24}      $UpdateState(u)$ ;

```

Figure 6: GLPA*: Generalized Lifelong Planning A*.

for the same function F . The values $Q_g(s, a)$ are maintained by GLPA*, while the values $Q_{gd}(s, a)$ are used to defined the fixpoints $gd(s)$ for all states $s \in S$ as follows:

$$gd(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{a \in A(s)} Q_{gd}(s, a) & \text{otherwise.} \end{cases}$$

Since the domain is potentially nondeterministic, the plan is a policy (a mapping from states to actions). The best plan is defined to result from executing action $\arg \min_{a \in A(s)} Q_{gd}(s, a)$ in state $s \in S$ with $s \neq s_{goal}$. For example, LPA* defines $Q_g(s, a) = c(s, a, s') + g(s')$ for $Succ(s, a) = \{s'\}$.

Local Consistency GLPA* maintains the same two kinds of variables as LPA* for each state s , which again estimate the value $gd(s)$: a g-value $g(s)$ and an rhs-value $rhs(s)$. The rhs-value always satisfies the following relationship (**Invariant 1**):

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{a \in A(s)} Q_g(s, a) & \text{otherwise.} \end{cases}$$

The definitions of local consistency, overconsistency and underconsistency are exactly the same as for LPA*.

Priority Queue GLPA* maintains the same priority queue as LPA* but the priority queue now always contains exactly

the locally inconsistent states that have not yet been expanded as overconsistent during the current call to `ComputePlan()` (**Invariant 2**). GLPA* gives one a considerable amount of freedom when it comes to specifying the priorities of the states, different from LPA*. The priority $K(s)$ of state s is calculated from the values $g(s)$ and $rhs(s)$. The calculation of the priority $K(s)$ is subject only to the restriction that there must exist a constant $1 \leq \epsilon < \infty$ so that for all states $s, s' \in S$ (**Restriction 2**):

- if $g(s') \geq rhs(s')$, $g(s) < rhs(s)$ and $rhs(s') \geq c^*(s', s) + g(s)$, then $K(s') > K(s)$, and
- if $g(s') \geq rhs(s')$, $g(s) > rhs(s)$ and $rhs(s') > \epsilon c^*(s', s) + rhs(s)$, then $K(s') > K(s)$.

We “translate” these restrictions as follows: There are two cases when the priority $K(s)$ of some state s has to be smaller than the priority $K(s')$ of some state s' ($K(s') > K(s)$). In both cases, state s' is either locally consistent or locally overconsistent ($g(s') \geq rhs(s')$). In the first case, state s is locally underconsistent ($g(s) < rhs(s)$) and the rhs-value of state s' might potentially depend on the g-value of s ($rhs(s') \geq c^*(s', s) + g(s)$). In the second case, state s is locally overconsistent and the rhs-value of state s' might potentially overestimate the cost of an optimal plan from state s' to the goal state by more than a factor of ϵ based on the rhs-value of s ($rhs(s') > \epsilon c^*(s', s) + rhs(s)$).

The priority of a state s in the priority queue always corresponds to $K(s)$ (**Invariant 3**). GLPA* compares priorities in the same way as LPA*, namely according to a lexicographic ordering, although priorities no longer need to be pairs.

Pseudo Code The pseudo code of GLPA* in Figure 6 is very similar to the one of LPA*. There are only two differences: First, GLPA* generalizes the calculation of the rhs-values and the priorities, which gives one a considerable amount of freedom when it comes to specifying the optimization criterion and the order in which to expand states. Second, the priority queue of GLPA* does not contain all locally inconsistent states but only those locally inconsistent states that have not yet been expanded as overconsistent during the current call to `ComputePlan()`. This can be done by maintaining an unordered list of states that are locally inconsistent but not in the priority queue. The predicate `NotYet(s)` {08} is therefore a shorthand for “state s has not been expanded yet as overconsistent during the current call to `ComputePlan()`.” However, GLPA* updates the priority queue to contain all locally inconsistent states between calls to `ComputePlan()` {20}.

Theoretical Analysis

We now prove the termination, efficiency and correctness of GLPA*. When `ComputePlan()` expands a state as overconsistent, then it removes the state from the priority queue and cannot insert it again since it does not insert states into the priority queue that have already been expanded as overconsistent. Thus, `ComputePlan()` does not expand a state again after it has expanded the state as overconsistent. When `ComputePlan()` expands a state as underconsistent, it sets the g-value of the state to infinity. Thus, if it expands the state

again, it expands the state as overconsistent next and then cannot expand it again. GLPA* thus provides the same guarantee as LPA*, and the next theorem about the termination and efficiency of GLPA* follows:

Theorem 1 *GLPA* expands every state at most once as underconsistent and at most once as overconsistent during each call to `ComputePlan()` and thus terminates.*

The correctness of GLPA* follows from the following lemma:

Lemma 1 *Assume that `ComputePlan()` executes line {09} and consider a state $u \in U$ for which $g(u) \geq rhs(u)$ and $gd(s) \leq rhs(s) \leq g(s) \leq \epsilon gd(s)$ for all states $s \in S$ with $K(s) < K(u)$. Then $gd(u) \leq rhs(u) \leq \epsilon gd(u)$, and the plan-execution cost is no larger than $rhs(u)$ if one starts in u and always executes the action $\arg \min_{a \in A(s)} Q_g(s, a)$ in the current state s until $s = s_{goal}$.*

The reason why GLPA* is correct is similar to the reason why breadth-first search is correct. Whenever breadth-first search expands a state with the smallest priority among all states that have not been expanded yet, then the g-values of all states with smaller priorities are equal to their respective start distances. Consequently, the g-value of the expanded state is also equal to its start distance. Similarly, whenever `ComputePlan()` expands an overconsistent state u with the smallest priority among all inconsistent states that have not been expanded yet, then all states s with $K(s) < K(u)$ satisfy $gd(s) \leq rhs(s) \leq g(s) \leq \epsilon gd(s)$. According to the above lemma, it then also holds that $gd(u) \leq rhs(u) \leq \epsilon gd(u)$. When the overconsistent state u is expanded, its g-value is set to its rhs-value and it then holds that $gd(u) \leq rhs(u) \leq g(u) \leq \epsilon gd(u)$. After `ComputePlan()` terminates, it holds that $U.TopKey() \geq K(s_{start})$ and $g(s_{start}) = rhs(s_{start})$. Thus, none of the states s with $K(s_{start}) > K(s)$ are in the priority queue and consequently they satisfy $gd(s) \leq rhs(s) \leq g(s) \leq \epsilon gd(s)$. The above lemma thus applies, and the next theorem about the correctness of GLPA* follows:

Theorem 2 *After `ComputePlan()` terminates, it holds that $gd(s_{start}) \leq rhs(s_{start}) \leq \epsilon gd(s_{start})$ and the plan-execution cost is no larger than $rhs(s_{start})$ if one starts in s_{start} and always executes the action $\arg \min_{a \in A(s)} Q_g(s, a)$ in the current state s until $s = s_{goal}$.*

The plan-execution cost of GLPA* is thus at most a factor of ϵ larger than minimal.

Possible Instantiations of GLPA*

We define $Q_g(s, a) = \max_{s' \in Succ(s, a)} (c(s, a, s') + g(s'))$ in deterministic and nondeterministic domains for all states $s \in S$ and actions $a \in A(s)$. This definition reduces to $Q_g(s, a) = c(s, a, s') + g(s')$ for $Succ(s, a) = \{s'\}$ in deterministic domains, which is the definition used by LPA*. It is easy to show that this choice of $Q_g(s, a)$ satisfies Restriction 1. We now show how to define the priorities for various scenarios so that the resulting choices of $K(s)$ satisfy Restriction 2. Consequently, the restrictions are not very limiting.

LPA* and Minimax LPA* It is easy to show that GLPA* reduces to LPA* in deterministic domains and to Minimax

LPA* in nondeterministic domains if one calculates $K(s)$ in the following way:

```

if ( $g(s) < rhs(s)$ )
   $K(s) = [g(s) + h(s); g(s)]$ 
else
   $K(s) = [rhs(s) + h(s); rhs(s)]$ 

```

It is easy to show that this choice of $K(s)$ satisfies Restriction 2 with $\epsilon = 1$ if the heuristics are nonnegative and (backward) consistent, which now means that $h(s_{start}) = 0$ and $h(s) \leq h(s') + c(s', a, s)$ for all states $s \in S$, states $s' \in Pred(s)$, and actions $a \in A(s')$ with $s \in Succ(s', a)$. This property implies that $h(s) \leq h(s') + c^*(s', s)$ for all states $s, s' \in S$.

- Assume that $g(s') \geq rhs(s')$, $g(s) < rhs(s)$ and $rhs(s') \geq c^*(s', s) + g(s)$. Then, a) $s \neq s'$ since otherwise we have the contradiction that $g(s) \geq rhs(s)$ and $g(s) < rhs(s)$, b) $rhs(s') \geq c^*(s', s) + g(s) > g(s)$ since $s \neq s'$ and therefore $c^*(s', s) > 0$, and c) $h(s') + c^*(s', s) \geq h(s)$ since the heuristics are (backward) consistent. Put together, it follows that $h(s') + rhs(s') \geq h(s') + c^*(s', s) + g(s) \geq h(s) + g(s)$ and $rhs(s') > g(s)$, which implies $K(s') = [rhs(s') + h(s'); rhs(s')] > [g(s) + h(s); g(s)] = K(s)$.
- Assume that $g(s') \geq rhs(s')$, $g(s) > rhs(s)$ and $rhs(s') > c^*(s', s) + rhs(s)$. Then, a) $rhs(s') > c^*(s', s) + rhs(s) > rhs(s)$ since $c^*(s', s) \geq 0$, and b) $h(s') + c^*(s', s) \geq h(s)$ since the heuristics are (backward) consistent. Put together, it follows that $h(s') + rhs(s') > h(s') + c^*(s', s) + rhs(s) \geq h(s) + rhs(s)$ and $rhs(s') > rhs(s)$, which implies $K(s') = [rhs(s') + h(s'); rhs(s')] > [rhs(s) + h(s); rhs(s)] = K(s)$.

According to Theorem 2, the plan-execution cost (worst-case plan-execution cost) of GLPA* is minimal in deterministic (nondeterministic) domains. It is easy to show that the first search of GLPA* in deterministic domains expands exactly the same states as A*-TB1 with the same consistent heuristics if GLPA* and A*-TB1 break remaining ties in the same way.

Inconsistent Heuristics We have argued that A* is often used with inconsistent heuristics to trade off plan-execution cost and planning time and then tends to expand fewer states than A* with consistent heuristics. Assume that the heuristics are nonnegative and (backward) ϵ -consistent, that is, satisfy $h(s_{start}) = 0$ and $h(s) \leq h(s') + \epsilon * c(s', a, s)$ for all states $s \in S$, states $s' \in Pred(s)$ and actions $a \in A(s')$ with $s \in Succ(s', a)$. One can then use GLPA* if one calculates $K(s)$ in the following way:

```

if ( $g(s) < rhs(s)$ )
   $K(s) = [g(s) + h_{cons}(s); g(s)]$ 
else
   $K(s) = [rhs(s) + h(s); rhs(s)]$ 

```

$h_{cons}(s)$ denotes any (backward) consistent heuristics, for example, the zero heuristics. It is easy to show that this choice of $K(s)$ satisfies Restriction 2 for the given ϵ . According to Theorem 2, the plan-execution cost (worst-case plan-execution cost) of GLPA* is at most a factor of ϵ larger than minimal in deterministic (nondeterministic) domains. It is easy to show that the first search of GLPA* in deterministic domains expands exactly the same states as A*-TB1 with

the same inconsistent heuristics if GLPA* and A*-TB1 break remaining ties in the same way. We now give examples of heuristics that are (backward) ϵ -consistent:

- In search, inconsistent heuristics are often derived from (backward) consistent heuristics $h^o(s)$ by multiplying them with a constant $\epsilon > 1$ (resulting in weighted heuristics). Then, $h^o(s_{start}) = 0$ and $h^o(s) \leq h^o(s') + c(s', a, s)$ for all states $s \in S$, states $s' \in Pred(s)$ and actions $a \in A(s')$ with $s \in Succ(s', a)$. Consequently, $h(s_{start}) = \epsilon h^o(s_{start}) = 0$ and $h(s) = \epsilon h^o(s) \leq \epsilon h^o(s') + \epsilon c(s', a, s) \leq h(s') + \epsilon c(s', a, s)$ for all states $s \in S$, states $s' \in Pred(s)$ and actions $a \in A(s')$ with $s \in Succ(s', a)$. The inconsistent heuristics are thus (backward) ϵ -consistent. One can use $h_{cons}(s) = h^o(s)$ for all $s \in S$.
- In HSP-type planning, inconsistent heuristics are sometimes obtained by adding the values of n (backward) consistent heuristics $h_i^o(s)$ (Bonet & Geffner 2001). Then, $h_i^o(s_{start}) = 0$ and $h_i^o(s) \leq h_i^o(s') + c(s', a, s)$ for all states $s \in S$, states $s' \in Pred(s)$ and actions $a \in A(s')$ with $s \in Succ(s', a)$. Consequently, $h(s_{start}) = \sum_i h_i^o(s_{start}) = 0$ and $h(s) = \sum_i h_i^o(s) \leq \sum_i h_i^o(s') + \sum_i c(s', a, s) \leq h(s') + n c(s', a, s)$ for all states $s \in S$, states $s' \in Pred(s)$ and actions $a \in A(s')$ with $s \in Succ(s', a)$. The inconsistent heuristics are thus (backward) ϵ -consistent for $\epsilon = n$. One can use $h_{cons}(s) = \max_i h_i^o(s)$ for all $s \in S$.
- Sometimes the inconsistent heuristics are not derived from (backward) consistent heuristics but it is known that $\epsilon_l c^*(s_{start}, s) \leq h(s) \leq \epsilon_u c^*(s_{start}, s)$ for all states $s \in S$ and given constants ϵ_l and ϵ_u , where ϵ_l can be zero. Then, $0 = \epsilon_l c^*(s_{start}, s_{start}) \leq h(s_{start}) \leq \epsilon_u c^*(s_{start}, s_{start}) = 0$ and $h(s) \leq \epsilon_u c^*(s_{start}, s) \leq \epsilon_u (c^*(s_{start}, s') + c(s', a, s)) = \epsilon_u c^*(s_{start}, s') + \epsilon_u c(s', a, s) = \epsilon_u c^*(s_{start}, s') + \epsilon_u c(s', a, s) - \epsilon_l c^*(s_{start}, s') + \epsilon_l c^*(s_{start}, s') = (\epsilon_u + (\epsilon_u - \epsilon_l) \frac{c^*(s_{start}, s')}{c(s', a, s)}) c(s', a, s) + \epsilon_l c^*(s_{start}, s') \leq (\epsilon_u + (\epsilon_u - \epsilon_l) \frac{\max_{s \in S} c^*(s_{start}, s)}{c_{min}}) c(s', a, s) + \epsilon_l c^*(s_{start}, s') = \epsilon c(s', a, s) + \epsilon_l c^*(s_{start}, s') \leq \epsilon c(s', a, s) + h(s')$ for all states $s \in S$, states $s' \in Pred(s)$ and actions $a \in A(s')$ with $s \in Succ(s', a)$, where $c_{min} = \min_{s \in S, a \in A(s), s' \in Succ(s, a)} c(s, a, s')$. The inconsistent heuristics are thus (backward) ϵ -consistent for $\epsilon = \epsilon_u + (\epsilon_u - \epsilon_l) \frac{\max_{s \in S} c^*(s_{start}, s)}{c_{min}}$.

Good Tie-Breaking We have argued that A* is often used with good tie-breaking and then tends to expand fewer states than A* with bad tie-breaking. One can use GLPA* in the same way if one calculates $K(s)$ in the following way:

```

if ( $g(s) \leq rhs(s)$ )
   $K(s) = [g(s) + h(s); 0; g(s)]$ 
else
   $K(s) = [rhs(s) + h(s); 1; h(s)]$ 

```

It is easy to show that this choice of $K(s)$ satisfies Restriction 2 with $\epsilon = 1$ if the heuristics are nonnegative and (backward) consistent. According to Theorem 2, the plan-execution cost (worst-case plan-execution cost) of GLPA* is

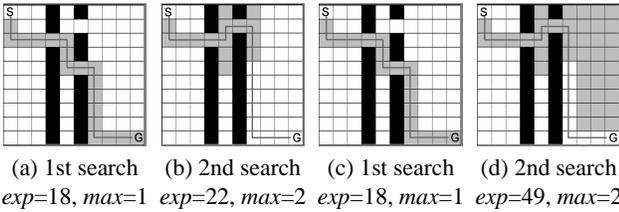


Figure 7: (a,b): LPA* with inconsistent heuristics (three times the Manhattan Distance) and bad tie-breaking (LPA*-TB1). (c,d): LPA* with consistent heuristics (Manhattan Distance) and good tie-breaking (LPA*-TB2). exp is the number of state expansions, and max is the maximum number of expansions of the same state.

minimal in deterministic (nondeterministic) domains. It is easy to show that the first search of GLPA* in deterministic domains expands exactly the same states as A*-TB2 with the same consistent heuristics if GLPA* and A*-TB2 break remaining ties in the same way.

Good Tie-Breaking and Inconsistent Heuristics One can combine good tie-breaking and inconsistent heuristics that are (backward) ϵ -consistent if one calculates $K(s)$ in the following way:

$$\begin{aligned} &\text{if } (g(s) < rhs(s)) \\ &\quad K(s) = [g(s) + h_{cons}(s); 0; g(s)] \\ &\text{else if } (g(s) = rhs(s)) \\ &\quad K(s) = [g(s) + h(s); 0; g(s)] \\ &\text{else} \\ &\quad K(s) = [rhs(s) + h(s); 1; h(s)]; \end{aligned}$$

$h_{cons}(s)$ again denotes any (backward) consistent heuristics, for example, the zero heuristics. It is easy to show that this choice of $K(s)$ satisfies Restriction 2 for the given ϵ if the heuristics are nonnegative and (backward) ϵ -consistent. According to Theorem 2, the plan-execution cost (worst-case plan-execution cost) of GLPA* is at most a factor of ϵ larger than minimal in deterministic (nondeterministic) domains. It is easy to show that the first search of GLPA* in deterministic domains expands exactly the same states as A*-TB2 with the same inconsistent heuristics if GLPA* and A*-TB2 break remaining ties in the same way.

Illustration

We use the four-connected grid from Figure 3 to evaluate our new versions of LPA*. Figure 7(a,b) shows the cells expanded by the new version of LPA* with inconsistent heuristics, and Figure 7(c,d) shows the cells expanded by the new version of LPA* with good tie-breaking. The new versions of LPA* indeed expand each state at most twice and expand many fewer states than our earlier naive modifications of LPA* but the grid is not large enough to demonstrate that they expand fewer states than search from scratch. We therefore present a larger and more systematic case study in the following, where we average over 100 much larger grids.

Experimental Evaluation

We use four-connected grids of size 200 by 200 cells, with the exception of Figure 10 where we use eight-connected grids. The start cell is at (20,20), and the goal cell is at (180,180). We created 100 grids by randomly blocking 10 percent of the cells (= 4,000 cells) and then changed each grid 500 times in a row (resulting in one planning episode and 500 replanning episodes) by changing the blockage status of 20 randomly chosen blocked and 20 randomly chosen unblocked cells, with one restriction: Since LPA* is generally more efficient than A* in situations where the changes occur more around the goal states of the search (Koenig, Furcy, & Bauer 2002), we chose 90 percent of the blocked cells and 90 percent of the unblocked cells within a distance of 50 cells from the goal cell of the search. We chose the remaining cells from outside of this region. We use two different heuristics. The strong (weak) heuristics uses the sum (maximum) of the absolute difference of the x coordinates of two cells and the absolute difference of their y coordinates as heuristic approximation of their distance. Both heuristics are (forward and backward) consistent. The strong heuristics (Manhattan distance) dominate the weak ones, which explains our choice of names. We therefore expect all search algorithms to have a smaller planning time with the strong heuristics than the weak heuristics. The priority queues of all search algorithms are implemented as binary heaps. (We could speed up A* by using buckets instead of binary heaps but it is currently unclear how to speed up LPA* in the same way.) Whenever we need a variant of LPA* with consistent heuristics (that is $\epsilon = 1$) and bad tie-breaking, we use the original version of LPA*. Otherwise, we use one of the new versions of LPA*. All versions of LPA* use the optimizations described in (Koenig & Likhachev 2002b). The versions of A* never expand any state more than once per (re-)planning episode and do not replan whenever all edges with changed costs are outside of their previous search trees. As a consequence, their planning time averaged over all (re-)planning episodes can be smaller than their planning time averaged over the first planning episodes only. All planning times are reported in milliseconds on a Pentium 1.8 GHz PC.

Inconsistent Heuristics

We create inconsistent heuristics by multiplying either the strong or the weak heuristics with the same constant ϵ , where ϵ ranges from 1.0 to 2.4. Tables 8, and 9 report for A*-TB1, A*-TB2 and our new LPA* with bad tie-breaking (LPA*-TB1) both the resulting planning time and the resulting plan-execution cost averaged over both the first planning episode and the 500 subsequent replanning episodes. We observe the following trends: All search algorithms trade off plan-execution cost and planning time. The planning time decreases and the plan-execution cost increases as ϵ increases. The decrease in planning time is more pronounced for the inconsistent heuristics based on the weak heuristics than the strong heuristics since the strong heuristics are more informative. Also, the decrease in planning time is more pronounced for A*-TB1 and A*-TB2 than for LPA*-TB1 since LPA*-TB1 is already much faster than A*-TB1 and A*-TB2

Planning Time									
ϵ	1.0	1.2	1.4	1.6	1.8	2.0	2.2	2.4	
A*-TB1	16.60	0.63	0.48	0.42	0.40	0.37	0.37	0.37	
A*-TB2	1.73	0.41	0.38	0.38	0.37	0.37	0.37	0.37	
LPA*-TB1	0.16	0.13	0.13	0.12	0.12	0.12	0.12	0.12	

Plan-Execution Cost									
ϵ	1.0	1.2	1.4	1.6	1.8	2.0	2.2	2.4	
A*-TB1	320.07	330.74	333.93	335.08	335.82	336.19	336.39	336.58	
A*-TB2	320.07	333.44	335.62	336.20	336.51	336.66	336.86	337.17	
LPA*-TB1	320.07	330.35	334.66	336.48	337.33	337.82	338.19	338.40	

Figure 8: Strong heuristics.

Planning Time									
ϵ	1.0	1.2	1.4	1.6	1.8	2.0	2.2	2.4	
A*-TB1	24.07	20.70	16.43	11.02	4.88	0.63	0.39	0.36	
A*-TB2	23.96	20.69	16.34	11.21	4.72	0.47	0.31	0.30	
LPA*-TB1	0.20	0.17	0.14	0.12	0.10	0.09	0.10	0.10	

Plan-Execution Cost									
ϵ	1.0	1.2	1.4	1.6	1.8	2.0	2.2	2.4	
A*-TB1	320.07	320.07	320.07	320.08	320.08	320.12	320.14	320.15	
A*-TB2	320.07	320.07	320.08	320.08	320.11	320.16	320.20	320.23	
LPA*-TB1	320.07	320.07	320.07	320.08	320.08	320.09	320.17	320.19	

Figure 9: Weak heuristics.

	planning time 501 episodes	planning time initial episodes	plan-execution cost
A*-TB1	0.24	0.34	172.66
A*-TB2	0.24	0.35	174.87
LPA*-TB1	0.14	0.43	173.54

Figure 10: Planning time for $\epsilon = 1$ averaged over (a) 501 (re-)planning episodes and (b) the initial planning episode only; and plan-execution cost

for $\epsilon = 1$. Overall, the planning times of LPA*-TB1 are always at least a factor of two smaller than the planning times of A*-TB1 and A*-TB2, yet the resulting plan-execution costs are comparable, demonstrating the advantage of the new version of LPA* with inconsistent heuristics if the planning times need to be small. We also created inconsistent heuristics in a different way, namely by using the strong heuristics on eight-connected (rather than four-connected) grids. The strong heuristics are inconsistent on eight-connected grids but ϵ -consistent for $\epsilon = 2$ since they are the sum of two consistent heuristics, namely the absolute difference of the x coordinates of two cells and the absolute difference of their y coordinates. We used $h_{cons}(s) = 0$ for all $s \in S$. Table 10 reports for A*-TB1, A*-TB2 and LPA*-TB1 both the resulting planning time and the plan-execution cost. For the planning times, we report both an average over the first planning episode and the 500 subsequent replanning episodes and an average over the first planning episode only. For the plan-execution time, we only report an average over the first planning episode and the 500 subsequent replanning episodes. The plan-execution costs of all three search methods are about the same but the planning time of LPA*-TB1 averaged over all planning episodes is somewhat smaller than the ones of A*-TB1 and A*-TB2. Note that the original version of LPA* cannot be used with inconsistent heuristics because it is not guaranteed to find plans of finite plan-execution cost even if they exist.

	strong heuristics 51 episodes	strong heuristics initial episodes	weak heuristics 51 episodes	weak heuristics initial episodes
A*-TB1	16.65	16.15	24.08	23.49
A*-TB2	2.05	1.99	23.91	23.46
LPA*-TB1	0.78	18.83	0.97	27.23
LPA*-TB2	0.52	2.37	0.97	27.35

Figure 11: Planning time for $\epsilon = 1$ averaged over (a) 51 (re-)planning episodes and (b) the initial planning episode only.

Good Tie-Breaking

Table 11 reports for A*-TB1, A*-TB2, the original version of LPA* (with bad tie-breaking, LPA*-TB1) and the new version of LPA* (with good tie-breaking, LPA*-TB2) with both the strong and weak heuristics the planning time. For the planning times, we report both an average over the first planning episode and the 50 subsequent replanning episodes and an average over the first planning episode only. The planning time averaged over all planning episodes is likely to be more important if the number of replanning episodes is large, and the planning time averaged over the first planning episode only is likely to be more important if the number of replanning episodes is very small. Since the start and goal cells are placed diagonally from each other and the density of blocked cells is relatively small, there tends to be a large number of minimum-cost paths from the start cell to the goal cell and a large number of cells on these paths have the same f-value as the goal cell. Thus, the tie-breaking strategy can make a large difference. (In contrast, if the start and goal cells are placed horizontally from each other, then there tends to be a much smaller number of minimum-cost paths from the start cell to the goal cell and the tie-breaking strategy makes much less of a difference.) We do not report the plan-execution costs since all search algorithms find cost-minimal paths. We observe the following trends: A*-TB2 has smaller planning times than A*-TB1. Therefore, there is no advantage of using A*-TB1 over A*-TB2 and we thus compare the versions of LPA* against A*-TB2. The original LPA*-TB1 corresponds to the state of the art before our research. Its average planning time is much smaller than that of A*-TB2 if the number of replanning episodes is large, but it is much larger than that of A*-TB2 if the number of replanning episodes is small (and the heuristics are strong). The reason for the latter fact is that LPA*-TB1 and A*-TB1 expand exactly the same states during the first search, and A*-TB1 in turn expands many more states than A*-TB2 in domains where the tie-breaking strategy makes a difference. (Also, LPA*-TB1 has some overhead over A*-TB1 per state expansion.) Our new LPA*-TB2 remedies this problem. Its planning time averaged over a large number of replanning episodes is identical to that of LPA*-TB1, and its planning time for the first planning episode is comparable to that of A*-TB2. The first property implies that there is no advantage of using LPA*-TB1 over LPA*-TB2, demonstrating the advantage of the new version of LPA*. The latter property implies that there is no advantage to the following alternative to LPA*-TB2: To obtain good planning times whether the number of replanning episodes is small or large, one could use A*-TB2 for the first planning episode, pass the priority queue and search tree from A*-TB2 to LPA*-TB1,

```

procedure Initialize()
{01}  $U = \emptyset$ ;
{02} for all  $s \in S$   $rhs(s) = g(s) = \infty$ ;
{03}  $rhs(s_{start}) = 0$ ;
{04} UpdateState( $s_{start}$ );
procedure UpdateState( $u$ )
{05} if ( $u \neq s_{start}$ )  $rhs(u) = \min_{s \in S, a \in A(s): succ(s,a)=u} (g(s) + c(s, a, u))$ ;
{06} if ( $u \in U$  and  $g(u) \neq rhs(u)$ ) U.Update( $u, K(u)$ );
{07} else if ( $u \in U$  and  $g(u) = rhs(s)$ ) U.Remove( $u$ );
{08} else if ( $u \notin U$  and  $g(u) \neq rhs(u)$  and NotYet( $u$ )) U.Insert( $u, K(u)$ );
procedure ComputePlan()
{09} while (U.TopKey() <  $K(s_{goal})$  or  $rhs(s_{goal}) \neq g(s_{goal})$ )
{10}    $u = U.Pop()$ ;
{11}   if ( $g(u) > rhs(u)$ )
{12}      $g(u) = rhs(u)$ ;
{13}   for all  $s \in Succ(u)$  UpdateState( $s$ );
{14}   else
{15}      $g(u) = \infty$ ;
{16}   for all  $s \in Succ(u) \cup \{u\}$  UpdateState( $s$ );
procedure Main()
{17} Initialize();
{18} forever
{19}   ComputePlan();
{20}   for all inconsistent states  $s \notin U$  U.Insert( $s, K(s)$ );
{21}   Wait for changes in action costs;
{22}   for all actions with changed action costs  $c(u, a, v)$ 
{23}     Update the action cost  $c(u, a, v)$ ;
{24}     UpdateState( $v$ );

```

Figure 12: Forward Version of GLPA*.

and then use LPA*-TB1 for all replanning episodes. Table 11 suggests that this algorithm likely achieves planning times similar to those of LPA*-TB2, but it is certainly much more complicated to implement than LPA*-TB2.

GLPA* and HSP-Type Planning

The forward version of LPA* has been used as heuristic search-based replanning method for HSP-type planners (Koenig, Furcy, & Bauer 2002). In the following, we change the search direction of GLPA* to obtain a forward version of it for deterministic domains. The pseudo code of the forward version of GLPA* in Figure 12 is the same as that of GLPA* except that we exchanged the start state and goal state and reversed all of the actions. It is suitable for HSP-type planning because it can handle goal states that are only partially defined and needs to determine only the predecessors of those states that are already in the search tree, which is trivial since their predecessors are already known. It is easier to see the similarity of A* to the forward version of GLPA* than the backward version of GLPA* because their search directions are identical. For example, the values $g(s)$ and $h(s)$ of the forward version of GLPA* correspond to the g-values and h-values of A* for state s , respectively. The heuristics now need to be nonnegative and satisfy $h(s_{goal}) = 0$ and $h(s) \leq c(s, a, s') + h(s')$ for all states $s \in S$, states $s' \in Succ(s)$ and actions $a \in A(s)$ with $s' = succ(s, a)$, which is the same definition of (forward) consistency used in the context of A*. The forward version of GLPA* can handle planning problems with changing action costs and thus also planning problems where actions become feasible or infeasible over time. One can extend it in the same way as one can extend the backward version of LPA* to D* Lite (Koenig & Likhachev 2002a), which allows it to handle planning prob-

lems with changing goal states as well. However, much work remains to be done. For example, planning domains are much larger than grids and might no longer completely fit into memory, in which case memory management becomes a problem. Also, the quality of the heuristics for HSP-type planning can be arbitrarily bad. Thus, it remains future work to build and evaluate a heuristic search-based replanner that makes use of GLPA* with inconsistent heuristics.

Conclusions

In this paper, we developed GLPA*, a framework that generalizes an incremental version of A*, called LPA*, and allows one to develop more capable versions of LPA* and its non-deterministic version Minimax LPA*, including a version of LPA* that uses inconsistent heuristics and a version of LPA* that breaks ties among states with the same f-values in favor of states with larger g-values. We showed experimentally that GLPA* indeed speeds LPA* on grids and thus promises to provide a good foundation for building heuristic search-based replanners.

Acknowledgments

The Intelligent Decision-Making Group is partly supported by NSF awards to Sven Koenig under contracts IIS-0098807 and IIS-0350584. 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 U.S. government.

References

- Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.
- Koenig, S., and Likhachev, M. 2002a. D* Lite. In *Proceedings of the National Conference on Artificial Intelligence*, 476–483.
- Koenig, S., and Likhachev, M. 2002b. Incremental A*. In Dietterich, T.; Becker, S.; and Ghahramani, Z., eds., *Advances in Neural Information Processing Systems 14*. Cambridge, MA: MIT Press.
- Koenig, S.; Furcy, D.; and Bauer, C. 2002. Heuristic search-based replanning. In *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling*, 294–301.
- Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong Planning A*. *Artificial Intelligence* 155(1-2):93–146.
- Kott, A.; Saks, V.; and Mercer, A. 1999. A new technique enables dynamic replanning and rescheduling of aeromedical evacuation. *Artificial Intelligence Magazine* 20(1):43–53.
- Likhachev, M., and Koenig, S. 2003. Speeding up the parti-game algorithm. In Becker, S.; Thrun, S.; and Obermayer, K., eds., *Advances in Neural Information Processing Systems 15*. Cambridge, MA: MIT Press.
- Pearl, J. 1985. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.