DD* Lite: Efficient Incremental Search with State Dominance

G. Ayorkor Mills-Tettey Anthony Stentz
M. Bernardine Dias

CMU-RI-TR-07-12

May 2007

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

This technical report presents DD* Lite, an efficient incremental search algorithm for problems that can capitalize on state dominance. Dominance relationships between nodes are used to prune graphs in search algorithms. Thus, exploiting state dominance relationships can considerably speed up search problems in large state spaces, such as mobile robot path planning considering uncertainty, time, or energy constraints. Incremental search techniques are useful when changes can occur in the search graph, such as when re-planning paths for mobile robots in partially known environments. While algorithms such as D* and D* Lite are very efficient incremental search algorithms, they cannot be applied as formulated to search problems in which state dominance is used to prune the graph. DD* Lite extends D* Lite to seamlessly support reasoning about state dominance. It maintains the algorithmic simplicity and incremental search capability of D* Lite, while resulting in orders of magnitude increase in search efficiency in large state spaces with dominance. We illustrate the efficiency of DD* Lite with simulation results from applying the algorithm to a path planning problem with time and energy constraints. We also prove that DD* Lite is sound, complete, optimal, and efficient.

Contents

1 Introduction

There are many search problems for which state space reduction can be achieved through simple comparison of states without compromising the optimality or completeness of the search [7, 6, 12, 3]. For example, consider the problem of planning a path for a battery powered mobile robot navigating through some terrain. Suppose the state of the robot is parameterized by its position and battery level. Because the available battery power is a finite resource, we can say that a state, A, is always better than another state, B, at the same position if it has more battery power available. We say that state A $dominates$ state B. Dominance relationships, when they exist, are very important in search problems because by exploiting these relationships and disregarding dominated states, we can prune the search space considerably. This enables a much more efficient solution to the search problem. Dominance relations have been successfully exploited in problems as varied as generating tests for combinatorial circuits [3] and solving the manufacturer's pallet loading problem [1].

Formally, given two states in a search algorithm, s_i and s_j, a dominance relation D is defined as a binary relation such that $s_i D s_j$, that is, s_i dominates s_j, implies that s_j cannot lead to a solution better than the best obtainable from s_i [6]. Dominated states may be deleted without expansion in the search, thus eliminating entire branches of the search tree.

Heuristic search techniques such as A* [5] have been successfully used in planning and many other problems. Dynamic or incremental search algorithms [2] involve efficiently computing the paths in a changing graph such as in a communications network where individual links may go up or down, in a transportation network where roads can be detoured, or in a robot's navigation map where its sensors may discover new information about its environment. Combining heuristic search with incremental search capability, in which solutions are repaired locally when changes occur, leads to algorithms that support rapid re-planning and as such are very effective for robot navigation in unknown or partially known environments. Incremental heuristic search techniques such as D* [10] and D* Lite [8] have typically been employed in two-dimensional planning scenarios such as on a regular grid overlaid on the terrain. However, there are many scenarios, such as our example of the battery powered robot, in which the planning and hence re-planning problem involves a search in a larger state space. For example, a path planning problem may require reasoning about time, energy [11] or even uncertainty [4]. Increasing the dimensionality, and hence the size of the state space, greatly limits the set of problems that can be solved efficiently with such techniques. In practice, however, only a small fraction of the entire state space is relevant to search. As such, techniques are needed to eliminate regions of the space which do not need to be explored. Focussing the search using relevant heuristics is one way; pruning the space by exploiting state dominance is another.

While dominance relations have been used extensively for pruning in static search problems [7, 6, 12, 3], it is more complicated to exploit dominance in dynamic or incremental search problems because a previously dominated branch of the search tree may be needed at a later time when costs in the graph change. To exploit state dominance in a dynamic search problem, the TEMPEST planner [11] explicitly resurrects dominated regions of the space when changes in the graph occur. However, keeping track of what

regions of the space need to be resurrected can be complicated.

While the exact definition of dominance relationships depends on the particular problem domain, the manner in which dominated states should be handled during the search is a general problem and can be built into the algorithm. In an incremental search, states may switch between being dominated and not dominated as changes occur in the search graph, and this must be handled seamlessly by the algorithm. In this paper, we introduce the DD* Lite algorithm, an extension to D* Lite, that supports reasoning about state dominance, while preserving soundness, completeness, optimality, and efficiency through incremental search. We apply DD* Lite to a path planning problem with time and energy constraints, and present results illustrating that orders of magnitude gains in performance can result from exploiting state dominance relationships in incremental search.

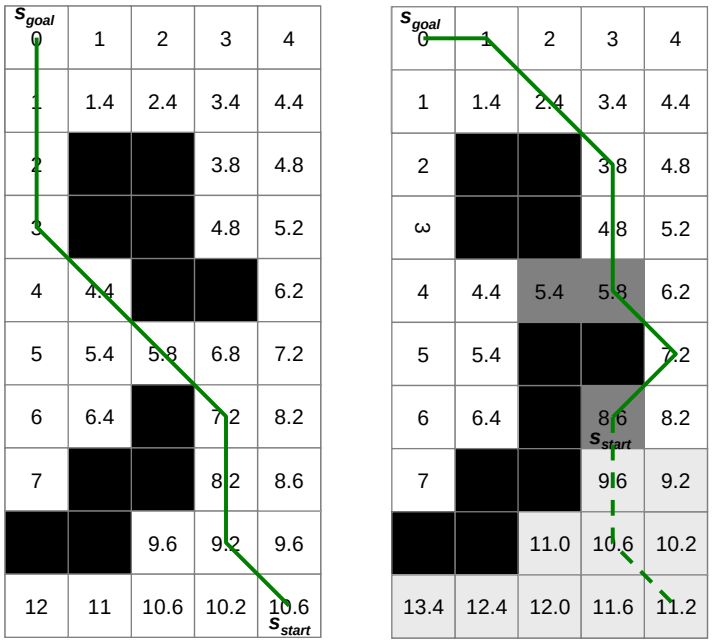
2 Incremental Search

The DD* Lite algorithm extends D* Lite, an incremental search algorithm that enables efficient repair of solutions when changes occur in a search graph.

As an example incremental search problem, consider a robot navigating through partially known terrain. The search graph is an 8-connected graph obtained by overlaying a regular grid on the terrain. Given the initial knowledge of the terrain, the algorithm finds an optimal path from the start to the goal by computing objective function values (minimum costs to the goal) for nodes in the graph, as illustrated in Figure 1(a). Blocked cells are shaded black, while free cells are clear. In the general case, the traversal cost of a cell may lie on a continuum between blocked and free. The robot begins to follow the computed path, but at some point discovers changes in the graph: some cells originally thought to be blocked are actually free, and other cells thought to be free are blocked, as shown in Figure 1(b). The portion of the original path traversed by the robot is indicated with a dashed line. When the discrepancies are discovered, the algorithm re-plans by searching for an optimal path from the robot's current position in the new graph. This path from the new start location to the goal is indicated by a solid line. Cells whose objective function values have changed are shaded gray. Of these, only a few (shaded dark gray) are relevant to finding the new solution: D* Lite's efficiency lies in the fact that it recomputes only these values.

The D* Lite algorithm, like others in the D* family of algorithms, searches from the goal to the start. In the robot navigation problem for which it was designed, changes in the search graph are likely to occur close to the robot's current position, as its sensors discover discrepancies in the environment. Reversing the order of the search in this way, so that areas close to the robot are near the leaves of the search tree, enables very efficient replanning.

During the search, D* Lite maintains two estimates of the objective function of a state s: the g-value and the rhs-value. The g value is the current estimate of the objective function of the state, while the rhs value is a one step lookahead estimate of the objective function of the state based on the g values of its successors (s') in the



(a) Initial solution (b) Modified solution

Figure 1: Incremental Search Example

graph:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{s' \in Succ(s)}(c(s, s') + g(s')) & \text{otherwise} \end{cases} \quad (1)$$

In equation 1, $c(s, s')$ represents the cost of the directed edge from s to s'.

In D* Lite, a state is defined as *consistent* if its g-value equals its rhs-value, and *inconsistent* otherwise. Specifically, it is defined as *overconsistent* if $g > rhs$ and *underconsistent* if $g < rhs$. As the search progresses, inconsistent states are inserted into the priority queue for processing.

At the beginning of the algorithm, the g and rhs values of all states are initialized to ∞, except the goal state, s_{goal}, whose rhs-value is initialized to 0. States can also be initialized when they are first encountered during the search, to avoid instantiating all states beforehand in large state spaces. To start with, the goal is the only inconsistent state and is inserted into the priority queue for processing. The main loop of the algorithm repeatedly processes states from the priority queue. When an overconsistent state is removed from the priority queue, its g value is set equal to its rhs value, thus making it consistent. When an underconsistent state is removed from the priority queue, its g value is set equal to ∞, thus making it overconsistent. In addition, in either case, the state's g value is used to update the rhs values of its predecessors in the graph according to equation 1, and the predecessors are in turn inserted into the priority queue if

they become inconsistent. The main loop of the algorithm terminates when the start state, s_{start}, has been processed and is consistent. At this point, an optimal path from s_{start} to s_{goal} has been computed.

When changes in the graph occur, D* Lite computes new rhs values for the affected states and inserts them into the priority queue if they are inconsistent. The main loop of the algorithm is then executed again until a new optimal path has been computed.

D* Lite is algorithmically simple yet efficient, making it an ideal starting point for an extended algorithm that supports exploiting state dominance.

3 Incremental Search with State Dominance

The basic idea underlying search with state dominance is to identify and dominated states before they are expanded and prune the search trees rooted at these states In incremental search, edge costs in the search graph may change and so states that were once dominated may no longer be dominated, and vice-versa. It is important to keep track of these changes, restoring previously pruned regions of the space as needed.

For a state u to dominate another v in our algorithm, DD* Lite, it must first dominate according to the domain definition of dominance, and secondly, it must have a lower or equal objective function value. This ensures that a dominated state cannot lead to a solution better than the best obtainable from the dominating state. In addition, dominance relations must obey the following properties:

- A state cannot dominate itself
- If a state u dominates another state v, then v *does not* dominate u.
- Transitive property: if a state u dominates another state v, and v in turn dominates w, then u dominates w.

To keep track of which states are dominated as we search through the state space, we label states as *dominated* or *not dominated*. A state is labeled as *dominated* if there is at least one other state in the space which dominates it, and is labeled *not dominated* otherwise. In the search, we do not expand dominated states, effectively pruning the subtree rooted at the dominated state.

We extend the D* Lite algorithm to support this concept. One extension is that, in addition to keeping track of current and one-step lookahead estimates of the objective function value of a state, as described in the previous section, we also keep track of current and one-step lookahead estimates of whether or not a state is dominated. Thus, the g and rhs values are defined as tuples with two components: an objective function component and a dominance component. The objective function component represents the cost of the path from the state to the goal, and can assume values ranging from 0 to ∞ inclusive. The dominance component represents whether or not the state is dominated and can take on one of two discrete values: NOT_DOMINATED or DOMINATED, where NOT_DOMINATED$<$DOMINATED.

$$g(s) = [g_{objf}(s), g_{dom}(s)] \quad (2)$$

$$rhs(s) = [rhs_{objf}(s), rhs_{dom}(s)] \quad (3)$$

The g-value is the current estimate of the objective function and dominance value of a state, while the rhs-value is the one-step lookahead estimate of the objective function and dominance value of a state based on the g-values of its successors in the graph.

We can define comparison operators on the domain of objective function and dominance value tuples as follows: first compare the objective function values, and then, in the case of a tie, compare the dominance values. The $<, >, \leq, \geq, =$, and $\neq$ operators can be defined in this way, as can the min() and max() functions. For example, we say that $g(s) < g(s')$ if and only if $(g_{objf}(s) < g_{objf}(s')) \vee (g_{objf}(s) = g_{objf}(s') \wedge g_{dom}(s) < g_{dom}(s'))$. Similarly, $g(s) < rhs(s)$ if and only if $(g_{objf}(s) < rhs_{objf}(s)) \vee (g_{objf}(s) = rhs_{objf}(s) \wedge g_{dom}(s) < rhs_{dom}(s))$.

Given a directed graph in which the cost of traversing an edge from a state s to a state s' is represented by $c(s, s')$, the purpose of the DD* Lite algorithm is to find the optimal (least-cost) path from a start state represented by s_{start} to a goal state represented by s_{goal}. Since the search proceeds backwards from the goal to the start, s_{goal} has the lowest objective function value of all states in the space and is, by definition, not dominated. In addition, it is assumed that s_{start} is not dominated. This can be ensured by the domain definition of dominance. Alternatively, the start state may be only partially specified by defining a *start set* of dominance neighbors, S_{start}, with the idea that we want our true start state, s_{start}, to be the least-cost non-dominated state in S_{start}. For example, in a path-planning problem, a start set might be the set of all states at $[x, y]$ position [4,5]: this set includes many states with different time or energy values. In the algorithm listings in this technical report, a specific start state, s_{start}, is used. It is, however, trivial to change the implementation to use a start set, S_{start}, and this is what is used in practice.

The *goal distance* of a state s is the cost of the least-cost path from s to s_{goal}. Given the definition of dominance stated previously, it is obvious that there should be no dominated states on the least-cost path between a non-dominated state s and s_{goal}.

To focus the search, we define a heuristic function, $h(s, s')$, that yields an estimate of the cost from a state s to another state s'. The heuristic function must be *admissible*, meaning that it must yield a value that is less than or equal to the true cost of the optimal path between s and s'. Furthermore, heuristics must obey the triangle inequality, meaning that $h(s, s'') \leq h(s, s') + h(s', s'')$ for states s, s', and s'' in the graph. If s' and s'' are connected by an edge in the graph, then another expression of the triangle inequality is: $h(s, s'') \leq h(s, s') + c(s', s'')$. Since the search proceeds from the goal state to the start state, the heuristic function, represented by $h(s_{start}, s)$, is an estimate of the cost from the start state, s_{start}, to a given state s. In implementations in which a start set, S_{start} is used, then the heuristic $h(s_{start}, s)$ represents the minimum estimate of the cost of the path from any state in the start set to s, that is, $h(s_{start}, s) = \min_{s' \in S_{start}}(h(s', s))$. We use $g(s) + h(s_{start}, s)$ as a shortcut to represent the tuple, $[g_{objf}(s) + h(s_{start}, s), g_{dom}(s)]$. Similarly, $rhs(s) + h(s_{start}, s)$ represents $[rhs_{objf}(s) + h(s_{start}, s), rhs_{dom}(s)]$.

As in the D* Lite algorithm, a state, s, is described as *consistent* when $g(s) = rhs(s)$, and *inconsistent* otherwise. A state is *overconsistent* if $g(s) > rhs(s)$ and *underconsistent* if $g(s) < rhs(s)$, where the =, $<$, and $>$ operators are as described above. Inconsistent states are inserted into the priority queue for processing.

Another extension we make to the D* Lite algorithm is in the definition of the

neighbors of a state, which are used in computing the state's rhs value, or whose rhs values are affected by changes to the state's g value. In addition to a state's predecessors and successors in the graph, we define a third class of neighbors: dominance neighbors. The set of dominance neighbors of a state, s, is the set of states that can potentially dominate or be dominated by s. Like the predecessors and successors in the graph, the set of dominance neighbors is problem-dependent.

With this new definition of the neighbors of a node, we can specify that the rhs-value of a node, defined in equation 8, always satisfies the following relationship. In the following equations, $F(s)$ is used to refer to the set of non-dominated successors of a state s. $D(s)$ is used to refer to the set of states which dominate state s. As in D* Lite, $c(s, s')$ represents the cost of the directed edge from s to s'.

$$rhs_{objf}(s) = \begin{cases} 0 & \text{if } s = s_{goal} \\ \min_{s' \in F}(c(s, s') + g_{objf}(s')) & \text{otherwise} \end{cases} \tag{4}$$

$$rhs_{dom}(s) = \begin{cases} \text{NOT\_DOMINATED} & \text{if } D(s) = \emptyset \\ \text{DOMINATED} & \text{otherwise} \end{cases} \tag{5}$$

where:

$$F(s) = \{s' : s' \in Succ(s) \wedge g_{dom}(s') = \text{NOT\_DOMINATED}\} \tag{6}$$

$$\begin{aligned} D(s) = \{s' : s' \in & \, DominanceNeighbors(s) \\ & \wedge Dominate(s', s) \wedge (g_{objf}(s') \leq rhs_{objf}(s)) \\ & \wedge (g_{objf}(s') + h(s_{start}, s') \\ & \quad \leq rhs_{objf}(s) + h(s_{start}, s))\} \end{aligned} \tag{7}$$

According to the equations above, the objective function value of a state s is affected by the objective function and dominance values of its successors in the graph. Similarly, the dominance value of s is influenced by the objective function values of its dominance neighbors.

The composition of the set $D(s)$ deserves some explanation. The fact that the exact definition of dominance is problem-dependent is handled by the use of a function $Dominate(s', s)$ which returns TRUE if the state s' dominates the state s according to the domain definition of dominance. The formal definition of dominance requires that a dominated state does not lead to a solution better than the best solution that can be obtained from the dominating state. That is, the dominated state can be ignored without loss of optimality in the solution. Hence, the DD* Lite algorithm also requires that for a state to be labeled DOMINATED, its objective function value must be greater than or equal to the objective function value of the dominating state, as captured by the third term of equation 7. This guarantees that states labeled DOMINATED cannot lead to better solutions than those obtained from the dominating state. Furthermore, a state is labeled DOMINATED only when the dominating state has already been processed off the open list, a condition captured by the final term in equation 7. This makes it possible to bound the number of times a node is processed off the open list, as discussed in the section on "Theoretical Properties".

4 DD* Lite Algorithm

The basic DD* Lite algorithm is shown in Figure 2. Differences from the basic D* Lite algorithm are indicated with line numbers emphasized, such as 1.

The Main() function of the algorithm calls Initialize(), which initializes the g and rhs values of all states in the space. Initially, s_{goal} is inserted into the priority queue as the only inconsistent state. It is worth noting that in practice, due to the potentially large size of the state space, only the goal state is initialized at this time; the other states are dynamically created and hence initialized only as they are encountered during the search. Similarly, states are deleted when they are deemed unreachable (i.e., both the g_{objf} and rhs_{objf} values are ∞): this can occur to predecessors of dominated states, or to states that are unreachable due to obstacles in the state space.

Main() then executes ComputeShortestPath(), which contains the principal loop of the algorithm. Like in D* Lite, ComputeShortestPath() repeatedly removes the state with the smallest key from the priority queue. The key of a state, $k(s)$, has two components, $[k_1(s), k_2(s)]$, where $k_1(s)$ and $k_2(s)$ are each an objective function and dominance value pair, defined as follows: $k_1(s) = min(g(s), rhs(s)) + h(s_{start}, s)$ and $k_2(s) = min(g(s), rhs(s))$. We compare two keys, say $k(s)$ and $k'(s)$, by comparing the first components and, in the case of a tie, comparing the second components. Hence, we say that $k(s) < k'(s)$ if and only if $(k_1(s) < k'_1(s)) \vee (k_1(s) = k'_1(s) \wedge k_2(s) < k'_2(s))$.

An overconsistent state (line 18) is processed by being made consistent on line 19. Cost changes are then propagated to predecessors and dominance neighbors on lines 20-21, by calling UpdateVertex() on these states. Changes to the g_{objf} or the g_{dom} values of a state may affect the rhs_{objf} value of its predecessors as indicated in equation 9. Changes to the g_{objf} value of a state may affect the rhs_{dom} value of its dominance neighbors as indicated in equation 10. UpdateVertex() computes the updated rhs value of a state. The state is then inserted into the priority queue if it is inconsistent. Note that if the state has no non-dominated successors, its rhs_{objf} value is ∞, which eventually results in the state being pruned from the space.

An underconsistent state (line 22) is processed by being made overconsistent on line 23. UpdateVertex() is then called on its predecessors and dominance neighbors as well as the node itself, to allow inconsistent states to be inserted back into the priority queue. ComputeShortestPath() terminates once the start state is consistent and all states that could dominate it have been processed from the priority queue, a condition captured by the expression on line 16.

4.1 Discussion

A couple of ideas underlying the DD* Lite algorithm merit some comment. First, the DD* Lite algorithm conceptually implements a tuple-based objective function where the first element is the solution cost and the second is the dominance relation. However, it maintains sufficient information and performs the checks necessary to incrementally repair the solution when either the objective function or the dominance relation changes, which would not be possible with the straight substitution of a tuple-based objective function.

The priority queue, U, has the following functions: U.Insert($node, key$) inserts a node into the priority queue with the given key, U.Pop() removes the node with the minimum key from the priority queue, U.TopKey() returns the minimum key of all nodes in the priority queue, and U.Remove($node$) removes a node from the priority queue.

procedure CalculateKey()
1 return $[min(g(s), rhs(s)) + h(s_{start}, s); min(g(s), rhs(s))]$;

procedure Initialize()
2 $U \longleftarrow \emptyset$;
3 **for all** $s \in S$ $rhs(s), g(s) \longleftarrow [\infty,$ NOT_DOMINATED];
4 $rhs(s_{goal}) \longleftarrow [0,$ NOT_DOMINATED];
5 U.Insert(s_{goal}, CalculateKey(s_{goal}));

procedure UpdateVertex(s)
6 **if** $s \neq s_{goal}$ ComputeRHS(s);
7 **if** $s \in U$ U.Remove(s);
8 **if** $g(s) \neq rhs(s)$ U.Insert(s, CalculateKey(s));

procedure ComputeRHS(s)
9 $F \longleftarrow \{s' : s' \in Succ(s)$ **and** $g_{dom}(s') =$NOT_DOMINATED $\}$;
10 $temp_{objf} \longleftarrow \min_{s' \in F}(g_{objf}(s') + c(s, s'))$;
11 $temp_{dom} \longleftarrow$ NOT_DOMINATED;
12 **for all** $s' \in DominanceNeighbors(s)$
13 **if** Dominate(s', s) **and** $g_{objf}(s') \leq temp_{objf}$ **and** $g_{objf}(s') + h(s_{start}, s') \leq temp_{objf} + h(s_{start}, s)$
14 $temp_{dom} \longleftarrow$ DOMINATED ;
break;
15 $rhs(s) \longleftarrow [temp_{objf}, temp_{dom}]$;

procedure ComputeShortestPath()
16 **while** U.TopKey() $\leq$ CalculateKey(s_{start}) **or** $rhs(s_{start}) \neq g(s_{start})$
17 $s \longleftarrow U$.Pop();
18 **if** $g(s) > rhs(s)$
19 $g(s) \longleftarrow rhs(s)$;
20 **for all** $s' \in DominanceNeighbors(s) \cup Pred(s)$
21 UpdateVertex(s');
22 **else**
23 $g(s) \longleftarrow [\infty,$ NOT_DOMINATED];
24 **for all** $s' \in DominanceNeighbors(s) \cup Pred(s) \cup \{s\}$
25 UpdateVertex(s');

procedure Main()
26 Initialize() ;
27 **repeat forever**
28 ComputeShortestPath() ;
29 Wait for changes in edge costs ;
30 **for all** *directed edges* (u, v) *with changed edge costs*
31 Update the edge cost $c(u, v)$;
32 UpdateVertex(u);

Figure 2: DD* Lite

Secondly, while it is typically easy to determine the predecessors and successors of the state from the search graph, retrieving the "dominance neighbors" of a node may not always be easy or efficient. Although the correctness of the algorithm is not dependent on identifying all potentially dominated states, the gain in efficiency due to pruning obviously increases with the number of instances of dominance that are identified. In general, domain knowledge about potential dominance relations will need to be exploited to determine how to store states so that retrieving dominance neighbors is efficient. For example, in the robot exploration domain involving a battery-powered rover, we specify dominance neighbors to be all states that share the same spatial dimension, and we store these states in a data structure that enables dominance neighbors to be accessed efficiently. In addition, implementation strategies can allow for efficiently stepping through the set of dominance neighbors. For example, in the exploration domain, we instantiate states only when they are first encountered in the search so that the list of dominance neighbors of a state is initially small, but grows as the search progresses. Furthermore, we stop stepping through dominance neighbors when we encounter one dominating state, so we often do not have to go through the entire set.

4.2 Theoretical Properties

As captured in the following theorems, detailed proofs of which appear in the appendix, DD* Lite retains the soundness, completeness and optimality properties of D* Lite. Additionally, we can prove similar properties concerning its efficiency.

Theorem. ComputeShortestPath() expands a non-dominated state in the space at most twice; namely once when it is locally underconsistent and once when it is locally overconsistent (refer to Theorem 12 in Appendix).

Theorem. ComputeShortestPath() expands a dominated state in the space at most four times; namely at most once when it is underconsistent and not dominated, once when it is overconsistent and not dominated, once when it is underconsistent and dominated, and once when it is overconsistent and dominated (refer to Theorem 12 in Appendix).

Theorem. After termination of ComputeShortestPath(), one can follow an optimal path from s_{start} to s_{goal} by always moving from the current state s, starting at s_{start}, to any non-dominated successor s' that minimizes $c(s, s') + g_{objf}(s')$ until s_{goal} is reached (breaking ties arbitrarily) (refer to Theorem 14 in Appendix).

An informal proof of the first two theorems is based on two observations. First is the observation that the keys of the states selected for expansion on line 17 of the algorithm are monotonically nondecreasing over time until ComputeShortestPath() terminates. This implies that once a state s is made consistent on line 19, its rhs_{objf} value does not change until ComputeShortestPath() terminates. This is because no state processed after s has a lower key, and hence a lower objective function value, than s does, implying that a better path to the goal from s cannot be found.

The second observation is that once a state becomes dominated, it stays dominated until ComputeShortestPath() terminates. Because states are processed in order of increasing keys, when a dominated state s is processed from the priority queue, the dominating state s' is already consistent. s can become NOT_DOMINATED again only

if the g_{obj} value of s' increases, which only occurs if s' is processed from the priority queue as an underconsistent state, which in turn does not occur because s' is consistent.

Combining these two observations with the fact that the main loop of the algorithm processes an overconsistent state by making it consistent and an underconsistent state by making it overconsistent, shows that a state is processed from the priority queue at most once in each of the four different cases outlined in the second theorem. If the state is eventually dominated, it may go through all four scenarios. If it is eventually not dominated, it is processed in at most two of the scenarios.

The third theorem follows from the fact that the ComputeShortestPath() terminates only when the start state s_{start} and all states with a lower or equal objective function value are consistent. At this point, the g_{objf} and rhs_{objf} values of all states on the path to the goal satisfy equation 9, and from the equation, none of the rhs_{objf} values are based on dominated states.

Formal proofs of these theorems appear in the appendix. The theorems capture the property that the algorithm correctly finds the optimal path between the start and the goal, and that dominated states are not included on this path. They also describe the efficiency of the algorithm: if no states in the space are dominated, the algorithm does as much work as D* Lite, processing each node at most twice. Dominated states are processed at most four times. Although DD* Lite potentially does more work per node than D* Lite, we show in the next section that the performance gains from exploiting state dominance far outweigh the extra processing required.

5 Simulation Results

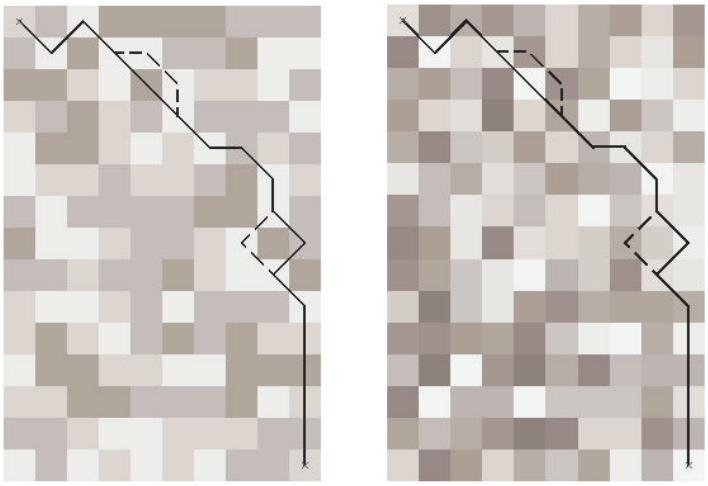
We applied DD* Lite to the problem of planning a path for a solar-powered mobile robot navigating from a start to a goal location in partially known terrain. The robot's solar panel charges a battery which in turn powers the wheels. The robot has a finite battery capacity, MAX_BATTERY, and attempts to reach the goal in the shortest amount of time. This is a path planning problem in three dimensions: each state is parameterized by three variables (x, y, e), where x and y are the two spatial dimensions, and e represents the energy required to reach the goal.

The two spatial dimensions, x and y, are represented as a regular grid. Each grid cell has an associated time and energy cost, c_t and c_e, representing the time and energy respectively required to cross the cell. Time costs are always positive, but energy costs may be positive or negative to account for solar charging as well as energy consumption for locomotion. When the robot transitions from one cell (x_1, y_1) to a neighboring cell (x_2, y_2), the resulting value of the energy variable e_2 depends on the starting energy e_1 and the energy costs of the two cells.

The problem is to plan a path from the start to the goal while optimizing traversal time and satisfying energy constraints. In this domain, we assert that it is always better to require less energy to reach the goal. This results in the following definition of dominace: Two states $s_1 = (x_1, y_1, e_1)$ and $s_2 = (x_2, y_2, e_2)$ are dominance neighbors if they are at the same spatial location, that is, $x_1 = x_2 \wedge y_1 = y_2$. Furthermore, s_1 dominates s_2 if $e_1 < e_2$. That is, for a state to dominate another, it must have a lower energy requirement. In addition, we use dominance to eliminate states that

are too similar, i.e., that are at the same spatial location and have very close energy values. We call this type of dominance "resolution equivalency". This was done to keep the size of the state space manageable. We used the cost of the 8-connected path assuming minimum time costs as the focussing heuristic in this domain. This heuristic is admissible and obeys the triangle inequality.

Figure 3 illustrates a path found by the DD* Lite algorithm. The path is shown superimposed on the time and energy cost maps. In the time map, darker shading represents larger time costs. In the energy cost map, clear cells indicate areas where solar charging more than compensates for the energy requirements of locomotion. Darker cells indicate areas where this is not the case. The selected path (solid line) optimizes time while satisfying energy constraints. The path that would have been selected (dashed line), had there been no energy constraints, is also shown.

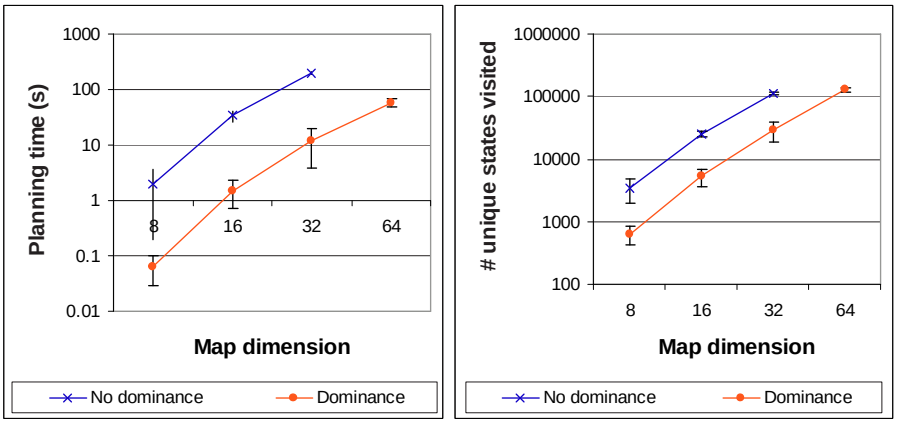


(a) Time cost map (b) Energy cost map

Figure 3: Example DD* lite plan superimposed on time and energy cost maps.

To characterize the performance of DD* Lite, we planned paths through several maps of different sizes with random time and energy costs. For each map size, we planned paths for 10 different random costs fields with the start and goal states at opposite corners of the map. We compared the planning time with dominance turned on to that with dominance turned off (except for resolution equivalency). Figure 4(a) plots the average planning time in seconds on the vertical axis against the size of the map on the horizontal axis. The experiments were run on a Pentium M 770 2.13 GHz processor. All maps were square, e.g. 64x64. The figure illustrates that as expected, exploiting dominance resulted in large improvements in planning time. Figure 4(b) illustrates a similar comparison for an alternative measure of planning efficiency, that is, the number of unique states visited in the search.

Since DD* Lite is an incremental search algorithm, the real test is of replanning efficiency. In our example, as the solar powered robot navigates through some terrain,

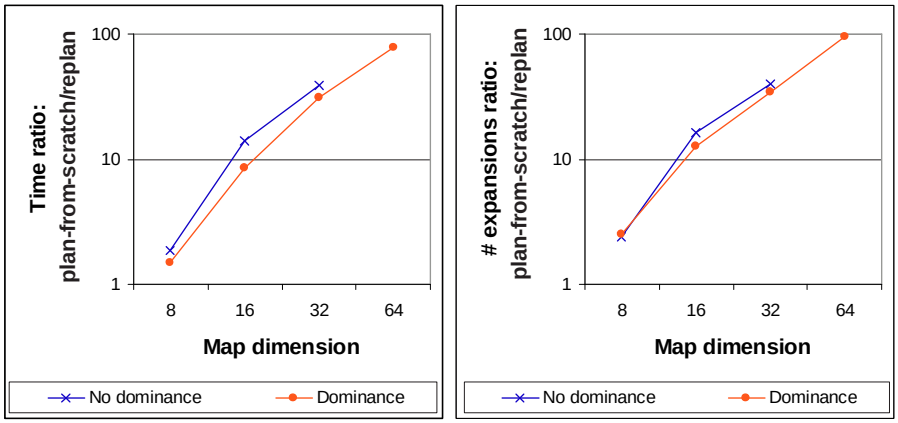


(a) Planning time (b) # unique states encountered

Figure 4: Comparison of planning efficiency with and without dominance

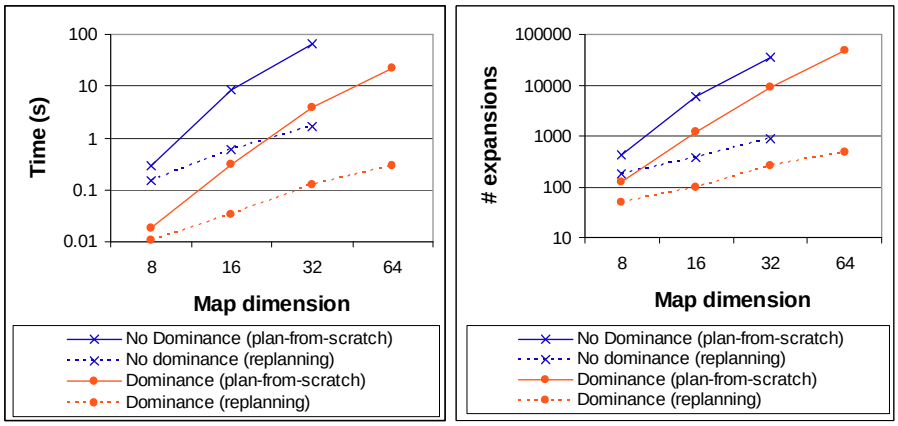
its sensors will discover discrepancies between its environment and its prior model of the world. For example, the terrain in a given cell could be rougher than previously estimated, resulting in higher time and/or energy costs, or there could be a greater exposure to sunlight than previously expected, resulting in lower energy costs. These observed changes cause the robot to modify its map and replan a new path to its current location. We compared the efficiency of replanning versus planning from scratch for this scenario, again using maps of varying sizes with random time and energy costs. The goal was placed at the corner of the grid, while the start for each run was placed at a random location within the grid. We planned an initial path to the start location, made some random changes in the cost field in a 3x3 region at the start location, and replanned a path. The replanning time and the number of states expanded in the search were compared to the planning time and number of states expanded when planning a path from scratch in the new cost field. Figure 5(a) shows the ratio of the total plan-from-scratch time to the total replanning time for 20 runs with random start locations. It shows that replanning is generally more efficient than planning from scratch and that, when expressed as a proportion of plan-from-scratch time, the replanning efficiency when dominance is exploited is comparable to that when dominance is not exploited. Figure 5(b) shows similar results for the ratio of the number of states expanded when planning from scratch to the number of states expanded when replanning. These results illustrate that DD* Lite maintains the incremental search efficiency of D* Lite.

Although the ratio of plan-from-scratch time to replanning time when dominance is exploited is comparable to that when dominance is not exploited, exploiting dominance results in performance gains in absolute terms for re-planning as well as planning. Figure 6(a) compares the average plan-from-scratch time to the average re-planning time for 20 runs with random start locations. Figure 6(b) compares the number of states expanded in the search for the same scenario. Both figures illustrate that exploiting dominance results in increased efficiency in re-planning and planning.



(a) Time (b) # states expanded

Figure 5: Ratio of performance cost of planning from scratch versus replanning, with and without dominance



(a) Planning/replanning time (b) # states expanded

Figure 6: Comparison of efficiency of planning from scratch versus re-planning, with and without dominance

6 Conclusions

We present DD* Lite, an incremental search algorithm that reasons about state dominance. DD* Lite extends D* Lite to support reasoning about state dominance in a domain-independent manner. It maintains the algorithmic simplicity and incremental search capability of D* Lite, whilst enabling orders of magnitude improvements in search efficiency in large state spaces with dominance. In addition, DD* Lite is sound,

complete, optimal, and efficient.

An important contribution of the DD* Lite algorithm is that it enables D*Lite-like incremental search algorithms to be extended into larger state spaces in a manner that is simple, easy to understand, and efficient. Several interesting classes of problems, such as the problem of energy and time constrained path planning that motivated this work, benefit from increasing the feasibility of incremental search in these spaces.

7 Appendix - Proofs of the Basic Version of DD* Lite

As stated in the following theorems, DD* Lite is sound, complete, optimal and efficient. Since DD* Lite extends D* Lite, many of the theorems below are similar to theorems developed by Koenig and Likhachev for D* Lite and in these cases, the proofs borrow heavily from the D* Lite proofs [9]. The table below can serve as a quick reference relating the DD* Lite theorems to the D* Lite ones.

DD* Lite Theorem	Corresponding D* Lite Theorem (if any)	Comments
Theorem 1	Theorem 5	The theorems are different because the DD* Lite version considers dominance.
Theorem 2	Theorem 6	The theorems are the same, but the proof of the DD* Lite version considers dominance.
Theorem 3	Theorem 7	The theorems are the same and the proofs are very similar.
Theorem 4	Theorem 8	The DD* Lite version considers dominance, resulting in a slightly different theorem.
Theorem 5	Theorem 9	The theorems are the same, but the proof of the DD* Lite version considers dominance.
Theorem 6	Theorem 10	The theorems are the same, but the proof of the DD* Lite version considers dominance.
Theorem 7	Theorem 11	The theorem and proof is almost exactly the same in both versions.
Theorem 8	Theorem 12	The theorems are different because the DD* Lite version considers dominance.
Theorem 9	–	No correspondence
Theorem 10	Theorem 12	The DD* Lite Theorems 8 and 10 together roughly correspond to Theorem 12 of D* Lite.
Theorem 11	–	No correspondence
Theorem 12	Theorem 14	The theorems are different because the DD* Lite version considers dominance.
Theorem 13	Theorem 15	The DD* Lite version is very similar to the D* Lite version but considers dominance.
Theorem 14	Theorems 16 & 17	The DD* Lite version is very similar to the D* Lite version but considers dominance.

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

$$rhs(u) = [rhs_{objf}(u),\ rhs_{dom}(u)] \tag{8}$$

$$rhs_{objf}(u) = \begin{cases} 0 & \textit{if } u = s_{goal} \\ \min_{s' \in F}(c(u, s') + g_{objf}(s')) & \textit{otherwise} \end{cases} \tag{9}$$

$$rhs_{dom}(u) = \begin{cases} \text{NOT\_DOMINATED} & \textit{if } D(u) = \emptyset \\ \text{DOMINATED} & \textit{otherwise} \end{cases} \tag{10}$$

where:

$$F(u) = \{s' : s' \in Succ(u) \wedge g_{dom}(s') = \text{NOT\_DOMINATED}\} \tag{11}$$

$$D(u) = \{s' : s' \in DominanceNeighbors(u) \wedge Dominate(s', u) \wedge (g_{objf}(s') < \infty \wedge (g_{objf}(s') \leq rhs_{objf}(u)) \wedge (g_{objf}(s') + h(s_{start}, s') \leq rhs_{objf}(u) + h(s_{start}, u)\} \tag{12}$$

Proof. The rhs-values set by Initialize() satisfy this relationship. The rhs_{objf} and rhs_{dom} values of the equations can change for a vertex when the cost of an outgoing arc changes, or when the g values of its forward neighbors (successors) or dominance neighbors change. This can happen on lines 19, 23, and 31. In all these cases, UpdateVertex() is called on the potentially affected nodes, ensuring that the rhs values continue to satisfy the relationship. □

Theorem 2. *The priority queue contains exactly the locally inconsistent vertices every time line 16 is executed.*

Proof. In Initialize(), the priority queue is initialized to contain exactly the locally inconsistent vertices. Thereafter, the local consistency of a vertex can change when its g-value or its rhs value changes. This can happen on lines 19, 23, or 15.

When a vertex is being expanded, line 17 removes the vertex from the open list. If the node is overconsistent, line 19 then makes the node consistent, and the node correctly remains off the open list.

In all other cases where the g or rhs values can change, UpdateVertex() removes the vertex from the open list if it is on the open list (line 7). It then inserts the vertex into the open list only if it is inconsistent (line 8).

Thus, the priority queue contains exactly the locally inconsistent vertices every time line 16 is executed. □

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

Proof. Whenever a vertex is inserted into the open list, its priority is set equal to its key. The key can change when either its g-value or its rhs-value changes. This can occur on lines 19, 23, and 15.

On line 19, the g value is changed, but the vertex is not on the open list. On line 23, the g value is changed, but UpdateVertex() is called immediately afterwards. UpdateVertex() removes the vertex from the priority queue and reinserts it if it is inconsistent, using a priority equal to its recalculated key. On line 15, the rhs value may change. This occurs in the ComputeRHS() function which is called from UpdateVertex(). After calling this function, UpdateVertex() removes the vertex from the priority queue and reinserts it if it is inconsistent, using a priority equal to its recalculated key. As such, the theorem continues to hold. □

Theorem 4. *Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line 17. If vertex v is locally consistent at this point in time but locally inconsistent the next time line 16 is executed, then the new key $k_{a(u)}(v)$ of vertex v satisfies $k_{a(u)}(v) > k_{b(u)}(u)$ the next time line 17 is executed if v is a backward neighbor (predecessor) of u and $k_{a(u)}(v) \geq k_{b(u)}(u)$ otherwise (i.e., v is only a dominance neighbor of u).*

Proof. Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line 17. Vertex v is locally consistent at this point in time but locally inconsistent the next time line 16 is executed.

The local consistency of a vertex changes when its g or rhs values change. The g value of vertex v does not change in ComputeShortestPath(), because the only node whose g value changes is u, and $v \neq u$. The rhs value of a vertex can change when the cost of an outgoing edge changes, or the g value of one of its forward or dominance neighbors changes. However, the costs of outgoing edges do not change in ComputeShortestPath(). As such, if the local consistency of vertex v changes, it means that its rhs value changed and hence v is a dominance neighbor of u, a backward neighbor (predecessor) of u, or both. We will consider these cases separately. Each case has two sub-cases, corresponding to u being overconsistent and u being underconsistent.

Case 1: v is a dominance neighbor of u, but not a backward neighbor.

Sub-Case 1a: u was overconsistent. This means that $g_{b(u)}(u) > rhs_{b(u)}(u)$, which is equivalent to saying that $g_{objf\ b(u)}(u) > rhs_{objf\ b(u)}(u)$ **or** $g_{objf\ b(u)}(u) = rhs_{objf\ b(u)}(u) \wedge g_{dom\ b(u)}(u) > rhs_{dom\ b(u)}(u)$. The vertex is made consistent on line 19 and subsequently, we have $g_{a(u)}(u) = rhs_{a(u)}(u) = rhs_{b(u)}(u)$. Since v is only a dominance neighbor of u and is not a predecessor, its rhs_{objf} is unaffected by changes to u (per Equation 9). As such, v can only become inconsistent as a result of changes to its rhs_{dom} value due to a change in $g_{objf}(u)$. Specifically, since u was overconsistent, $g_{objf}(u)$ has reduced, and v may now be dominated by u. Suppose that v was previously not dominated ($rhs_{dom}(v) =$NOT_DOMINATED) and that u now dominates v. This implies that u was also not dominated since any state that dominates u would also dominate v. Further, $rhs_{dom}(v)$ now changes from NOT_DOMINATED to DOMINATED. If this happens, v now becomes inconsistent and is put onto the open list with a key equal to [min($g_{a(u)}(v)$, $rhs_{a(u)}(v)$) + $h(s_{start}, v)$; min($g_{a(u)}(v)$, $rhs_{a(u)}(v)$)]. The key can also be written as [$g_{a(u)}(v)$ + $h(s_{start}, v)$; $g_{a(u)}(v)$] because $g_{objf\ a(u)}(v) = rhs_{objf\ a(u)}$ (since v was previously consistent and its rhs_{objf} value did not change) and $g_{dom\ a(u)}(v) < rhs_{dom\ a(u)}$ (since $rhs_{dom}(v)$ increased from NOT_DOMINATED to DOMINATED). Now, since u dominates v, we know from the definition of dominance (Equations 10 and 12) that $g_{objf\ a(u)}(u) \leq rhs_{objf\ a(u)}(v) = g_{objf\ a(u)}(v)$ **and** $g_{objf\ a(u)}(u) + h(s_{start}, u) \leq rhs_{objf\ a(u)}(v) + h(s_{start}, v) = g_{a(u)}(v) + h(s_{start}, v)$. Thus, $k_{a(u)}(v) \geq k_{b(u)}(u)$, and the theorem holds for this sub-case.

More concisely,

$$
\begin{aligned}
k_{a(u)}(v) &= [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\
&= [g_{a(u)}(v) + h(s_{start}, v); g_{a(u)}(v)] \\
&= [[g_{objf\ a(u)}(v), \text{NOT\_DOMINATED}] + h(s_{start}, v); [g_{a(u)}(v), \text{NOT\_DOMINATED}]] \\
&= [[rhs_{objf\ a(u)}(v), \text{NOT\_DOMINATED}] + h(s_{start}, v); [rhs_{a(u)}(v), \text{NOT\_DOMINATED}]] \\
&\geq [[g_{objf\ a(u)}(u), \text{NOT\_DOMINATED}] + h(s_{start}, u); [g_{objf\ a(u)}(u), \text{NOT\_DOMINATED}]]
\end{aligned}
$$

$$
\begin{aligned}
&= [[rhs_{objf\ a(u)}(u), \text{NOT\_DOMINATED}] + h(s_{start}, u); [rhs_{objf\ a(u)}(u), \text{NOT\_DOMINATED}]] \\
&= [[rhs_{objf\ b(u)}(u), \text{NOT\_DOMINATED}] + h(s_{start}, u); [rhs_{objf\ b(u)}(u), \text{NOT\_DOMINATED}]] \\
&= [rhs_{b(u)}(u) + h(s_{start}, u); rhs_{b(u)}(u)] \\
&= k_{b(u)}(u)
\end{aligned}
$$

Sub-case 1b: u was underconsistent. This means that $g_{b(u)}(u) < rhs_{b(u)}(u)$, which is equivalent to saying that $g_{objf\ b(u)}(u) < rhs_{objf\ b(u)}(u)$ **or** $g_{objf\ b(u)}(u) = rhs_{objf\ b(u)}(u) \wedge g_{dom\ b(u)}(u) < rhs_{dom\ b(u)}(u)$. After line 23, we have $g(u) = [\infty,$ DOMINATED]. Since v is only a dominance neighbor of u, its rhs_{objf} is unaffected by changes to u. As such, v can only become inconsistent as a result of changes to its rhs_{dom} value due to a change in $g_{objf}(u)$. Specifically, if v was previously dominated by only u, $rhs_{dom}(v)$ may change from DOMINATED to NOT_DOMINATED, since $g_{objf}(u)$ has increased. v now becomes inconsistent and is put onto the open list with a key equal to [min($g_{a(u)}(v)$, $rhs_{a(u)}(v)$) + $h(s_{start}, v)$; min($g_{a(u)}(v)$, $rhs_{a(u)}(v)$)]. The key can also be written as [$rhs_{a(u)}(v) + h(s_{start}, v)$; $rhs_{a(u)}(v)$] because $g_{objf\ a(u)}(v) = rhs_{objf\ a(u)}$ (since v was previously consistent and its rhs_{objf} value did not change) and $g_{dom\ a(u)}(v) > rhs_{dom\ a(u)}$ (since $rhs_{dom}(v)$ decreased from DOMINATED to NOT_DOMINATED). Now, since u previously dominated v, we know that $g_{objf\ b(u)}(u) \leq rhs_{objf\ b(u)}(v)$. And since v is no longer dominated, we know that u was the only node that dominated v, which means that it must have been itself not dominated: $g_{dom}(u)$ =NOT_DOMINATED. Furthermore, since u was underconsistent, $k_{b(u)}(u) = [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] = [g_{b(u)}(u) + h(s_{start}, v); g_{b(u)}(u)]$, and so $k_{a(u)}(v) \geq k_{b(u)}(u)$, and the theorem holds for this sub-case.

More concisely,

$$
\begin{aligned}
k_{a(u)}(v) &= [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\
&= [rhs_{a(u)}(v) + h(s_{start}, v); rhs_{a(u)}(v)] \\
&= [[rhs_{objf\ a(u)}(v), \text{NOT\_DOMINATED}] + h(s_{start}, v); [rhs_{a(u)}(v), \text{NOT\_DOMINATED}]] \\
&= [[rhs_{objf\ b(u)}(v), \text{NOT\_DOMINATED}] + h(s_{start}, v); [rhs_{b(u)}(v), \text{NOT\_DOMINATED}]] \\
&\geq [[g_{objf\ b(u)}(u), \text{NOT\_DOMINATED}] + h(s_{start}, u); [g_{objf\ b(u)}(u), \text{NOT\_DOMINATED}]] \\
&= [g_{b(u)}(u) + h(s_{start}, u); g_{b(u)}(u)] \\
&= k_{b(u)}(u)
\end{aligned}
$$

Case 2: v is a backward neighbor (predecessor) of u (and may also be a dominance neighbor of u)

Sub-Case 2a: u was overconsistent. This means that $g_{b(u)}(u) > rhs_{b(u)}(u)$, which is equivalent to saying that $g_{objf\ b(u)}(u) > rhs_{objf\ b(u)}(u)$ **or** $g_{objf\ b(u)}(u) = rhs_{objf\ b(u)}(u) \wedge g_{dom\ b(u)}(u) > rhs_{dom\ b(u)}(u)$. In addition, $k_{b(u)}(u)$ = [min($g_{b(u)}(u)$, $rhs_{b(u)}(u)$)+$h(s_{start}, u)$; min($g_{b(u)}(u), rhs_{b(u)}(u)$)] = [$rhs_{b(u)}(u)+h(s_{start}, u)$; $rhs_{b(u)}(u)$]. After line 19, we have $g_{a(u)}(u) = rhs_{a(u)}(u) = rhs_{b(u)}(u)$. Since v is a backward neighbor (predecessor) of u, $rhs_{objf}(v)$ may be affected by the reduction in $g(u)$, but only if $g_{dom\ a(u)}(u)$=NOT_DOMINATED and $g_{objf\ a(u)}(u) + c(v, u) < rhs_{objf\ b(u)}(v)$. If this is the case, then $rhs_{objf\ a(u)}(v) = g_{objf\ a(u)}(u) + c(v, u)$. Note that $rhs_{dom}(v)$

may now decrease from DOMINATED to NOT_DOMINATED if v was previously dominated by another state x with $rhs_{objf\ a(u)}(v) < g_{objf}(x) < rhs_{objf\ b(u)}(v)$. Furthermore, if u and v are also dominance neighbors, it is possible for u to dominate v, which would make $rhs_{dom\ a(u)}(v)$ =DOMINATED. Whether or not rhs_{dom} changes, $rhs_{a(u)}(v) < g_{a(u)}(v)$ and v goes back onto the open list with a key of $k_{a(u)}(v)$ = $[[rhs_{obfj\ a(u)}(v), rhs_{dom}(v)] + h(s_{start}, v); [rhs_{obfj\ a(u)}(v), rhs_{dom}(v)]]$ = $[[g_{objf\ a(u)}(u)+c(v,u), rhs_{dom}(v)]+h(s_{start}, v); [g_{objf\ a(u)}(u)+c(v,u), rhs_{dom}(v)]]$. Since $c(v,u) > 0$ and $c(v,u)+h(s_{start}, v) \geq h(s_{start}, u)$ (by the triangle inequality), then $k_{a(u)}(v) > k_{b(u)}(u)$ and the theorem holds for this sub-case.

More concisely,

$$\begin{aligned}
k_{a(u)}(v) &= [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\
&= [rhs_{a(u)}(v) + h(s_{start}, v); rhs_{a(u)}(v)] \\
&= [[g_{objf\ a(u)}(u) + c(v,u) + h(s_{start}, v), rhs_{dom\ a(u)}(v)]; [g_{objf\ a(u)}(u) + c(v,u), rhs_{dom\ a(u)}(v)]] \\
&> [[g_{objf\ a(u)}(u) + h(s_{start}, u), \text{NOT_DOMINATED}]; [g_{objf\ a(u)}(u), \text{NOT_DOMINATED}]] \\
&= [g_{a(u)}(u) + h(s_{start}, u); g_{a(u)}(u)] \\
&= [rhs_{b(u)}(u) + h(s_{start}, u); rhs_{b(u)}(u)] \\
&= k_{b(u)}(u)
\end{aligned}$$

Sub-Case 2b: u was underconsistent. This means that $g_{b(u)}(u) < rhs_{b(u)}(u)$, which is equivalent to saying that $g_{objf\ b(u)}(u) < rhs_{objf\ b(u)}(u)$ **or** $g_{objf\ b(u)}(u) = rhs_{objf\ b(u)}(u) \wedge g_{dom\ b(u)}(u) < rhs_{dom\ b(u)}(u)$. Thus, $k_{b(u)}(u) = [g_{objf\ b(u)}(u) + h(s_{start}, u); g_{objf\ b(u)}(u)]$. After line 23, we have $g(u) = [\infty$, DOMINATED]. Since v is a backward neighbor of u, $rhs_{objf}(v)$ may be affected by the increase in $g(u)$, but only if $rhs_{objf}(v)$ was previously computed from $g_{objf}(u)$, that is, only if $rhs_{objf\ b(u)}(v) = g_{objf\ b(u)}(u)+c(v,u) = \min_{s' \in F(u)}(c(u,s')+g_{obfj}(s'))$ and $g_{dom\ b(u)}(u)$=NOT_DOMINATED. In this case, $rhs_{objf}(v)$ increases. Note that $rhs_{dom}(v)$ may now increase from NOT_DOMINATED to DOMINATED if v is now dominated by another state x with $rhs_{objf\ b(u)}(v) < g_{objf}(x) < rhs_{objf\ a(u)}(v)$. On the other hand, $rhs_{dom}(v)$ may also decrease from DOMINATED to NOT_DOMINATED if it was previously dominated by only u. Whether or not rhs_{dom} changes, $g_{a(u)}(v) < rhs_{a(u)}(v)$ and v is put on the open list with a key of $[g_{a(u)}(v) + h(s_{start}, v); g_{a(u)}(v)]$. Since $rhs_{objf\ b(u)}(v)$ was computed from $g_{objf\ b(u)}(u)$, and v was consistent, $g_{objf\ a(u)}(v) = g_{objf\ b(u)}(v) = rhs_{objf\ b(u)}(v) > g_{objf\ b(u)}(u)$. Thus, when v is put back on the open list, its key is strictly greater than $k_{b(u)}(u)$ and the theorem holds for this sub-case.

More concisely,

$$\begin{aligned}
k_{a(u)}(v) &= [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\
&= [g_{a(u)}(v) + h(s_{start}, v); g_{a(u)}(v)] \\
&= [g_{b(u)}(v) + h(s_{start}, v); g_{b(u)}(v)] \\
&= [rhs_{b(u)}(v) + h(s_{start}, v); g_{rhs(u)}(v)] \\
&> [g_{b(u)}(u) + h(s_{start}, u); g_{b(u)}(u)] \\
&= k_{b(u)}(u)
\end{aligned}$$

□

Theorem 5. *If a locally overconsistent vertex u with key $k_{b(u)}(u)$ is selected for expansion on line 17, then it is locally consistent the next time line 16 is executed and its new key $k_{a(u)}(u)$ satisfies $k_{a(u)}(u) = k_{b(u)}(u)$.*

Proof. Assume the vertex u selected for expansion on line 17 is overconsistent. This means that $g_{b(u)}(u) > rhs_{b(u)}(u)$, which is equivalent to saying that $g_{objf\ b(u)}(u) > rhs_{objf\ b(u)}(u)$ **or** $g_{objf\ b(u)}(u) = rhs_{objf\ b(u)}(u) \wedge g_{dom\ b(u)}(u) > rhs_{dom\ b(u)}(u)$. On line 19, $g(u)$ is set equal to $rhs(u)$ and it becomes consistent. It remains consistent unless $rhs(u)$ is changed by a call to UpdateVertex() if u is a dominance neighbor or backward neighbor of itself. However, even a call to UpdateVertex() does not change its rhs-value, and u remains consistent. First, consider the rhs_{dom} value. By our definition of dominance, a state cannot dominate itself, therefore the set $D(s)$ of dominating states identified in the definition of the rhs value is the same before and after u is expanded and so $rhs_{dom}(u)$ does not change. Next, consider the rhs_{objf} value. $rhs_{objf}(u) = c(u, w) + g_{objf_{b(u)}}(w)$ for some state $w \neq u$. (Otherwise, $rhs_{objf}(u) = c(u, u) + g_{objf_{b(u)}}(u) > g_{objf_{b(u)}}(u)$ which would be a contradiction to the fact that u is overconsistent). Now, $rhs_{objf}(u)$ will only change if its value can be reduced by using $g_{objf_{a(u)}}(u)$ instead of $g_{objf_{a(u)}}(w)$ (and only if $g_{dom_{a(u)}}(u)$ =NOT_DOMINATED). However, $g_{objf_{a(u)}}(u) = rhs_{objf_{b(u)}}(u)$ so $c(u, u) + g_{objf_{a(u)}}(u) = c(u, u) + rhs_{objf_{b(u)}}(u) > rhs_{objf_{b(u)}}(u)$ and so $rhs_{objf}(u)$ cannot be reduced and does not change. Thus, the new key of u is as follows:

$$
\begin{aligned}
k_{a(u)}(u) &= [\min(g_{a(u)}(u), rhs_{a(u)}(u)) + h(s_{start}, u); \min(g_{a(u)}(u), rhs_{a(u)}(u))] \\
&= [rhs_{a(u)}(u) + h(s_{start}, u); rhs_{a(u)}(u)] \\
&= [rhs_{b(u)}(u) + h(s_{start}, u); rhs_{b(u)}(u)] \\
&= [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] \\
&= k_{b(u)}(u)
\end{aligned}
$$

□

Theorem 6. *Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line 17. If vertex v is locally inconsistent at this point in time and remains locally inconsistent the next time line 16 is executed, then the new key $k_{a(u)}(v)$ of vertex v satisfies $k_{a(u)}(v) \geq k_{b(u)}(u)$ the next time line 16 is executed.*

Proof. Assume that vertex u has key $k_{b(u)}(u)$ and is selected for expansion on line 17. Vertex v is locally inconsistent at this point in time and remains locally inconsistent the next time line 16 is executed. Since vertex u is expanded instead of v, it holds that $k_{b(u)}(v) \geq k_{b(u)}(u)$. We consider four cases:

Case 1: v's key does not change. Thus, $k_{a(u)}(v) = k_{b(u)}(v) \geq k_{b(u)}(u)$ and the theorem holds for this case.

Case 2: v's key changes, and $v = u$. Since vertex $v = u$ remains locally inconsistent, it could not have been overconsistent (according to Theorem 5). As such,

vertex $v = u$ was locally underconsistent ($rhs_{b(u)}(u) > g_{b(u)}(u)$) and furthermore, $g_{b(u)}(u) \leq [\infty,$ NOT_DOMINATED] because it cannot be underconsistent otherwise. $g(u)$ is set to [∞, DOMINATED] on line 23, and as such $g_{a(u)} \geq g_{b(u)}$. When UpdateVertex() is called, $rhs_{objf}(u)$ can change if u is a backward neighbor of itself, but it is guaranteed not to decrease since $g_{objf}(u)$ does not decrease. Similarly, $rhs_{dom}(u)$ is also guaranteed not to decrease because $rhs_{objf}(u)$ does not decrease. As such,

$$\begin{aligned} k_{a(u)}(v) &= k_{a(u)}(u) \\ &= [\min(g_{a(u)}(u), rhs_{a(u)}(u)) + h(s_{start}, u); \min(g_{a(u)}(u), rhs_{a(u)}(u))] \\ &\geq [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] \\ &= k_{b(u)}(u) \end{aligned}$$

Case 3: v's key changes, $v \neq u$, and vertex u was locally overconsistent.

Because u was locally overconsistent, $g_{b(u)}(u) > rhs_{b(u)}(u) = g_{a(u)}(u)$. Furthermore, since $v \neq u$, v's g-value does not change and $g_{a(u)}(v) = g_{b(u)}(v)$. As such, for v's key to change, its rhs-value must change, which will only happen if v is a backward neighbor (predecessor) and/or dominance neighbor of u. The decrease in $g(u)$ on line 19 can affect $rhs(v)$ in one of two situations: (i), if v is a backward neighbor of u, $g_{dom\ a(u)}(u) =$NOT_DOMINATED and $rhs_{objf\ a(u)}(v) = c(v,u) + g_{objf\ a(u)}(u) = c(v,u) + rhs_{objf\ a(u)}(u) = c(v,u) + rhs_{objf\ b(u)}(u)$ and (ii), if situation (i) does not hold but v is a dominance neighbor of u, $rhs_{dom\ b(u)}(v) =$NOT_DOMINATED and u now dominates v, causing $rhs_{dom\ a(u)}(v)$ to be DOMINATED.

For situation (i), we have $rhs_{objf\ a(u)}(v) \geq rhs_{objf\ b(u)}(u)$ and $rhs_{dom\ b(u)}(u) =$NOT_DOMINATED. As such, $rhs_{a(u)}(v) \geq rhs_{b(u)}(u) = min(g_{b(u)}(u), rhs_{b(u)}(u))$. Furthermore, $rhs_{objf\ a(u)}(v) + h(s_{start}, v) = c(v,u) + rhs_{objf\ b(u)}(u) + h(s_{start}, v) \geq rhs_{objf\ b(u)}(u) + h(s_{start}, u)$ (using the fact that $h(s_{start}, v) + c(v,u) \geq h(s_{start}, u)$ since the heuristics obey the triangle inequality). As such $rhs_{a(u)}(v) + h(s_{start}, v) \geq rhs_{b(u)}(u) + h(s_{start}, u) = min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u)$.

Thus:

$$\begin{aligned} &[rhs_{a(u)}(v) + h(s_{start}, v); rhs_{a(u)}(v)] \\ &\quad \geq [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] \\ &\quad = k_{b(u)}(u) \end{aligned} \tag{13}$$

Furthermore,

$$\begin{aligned} &[g_{a(u)}(v) + h(s_{start}, v); g_{a(u)}(v)] \\ &\quad = [g_{b(u)}(v) + h(s_{start}, v); g_{b(u)}(v)] \\ &\quad \geq [\min(g_{b(u)}(v), rhs_{b(u)}(v)) + h(s_{start}, v); \min(g_{b(u)}(v), rhs_{b(u)}(v))] \\ &\quad = k_{b(u)}(v) \\ &\quad \geq k_{b(u)}(u) \end{aligned} \tag{14}$$

Following from inequalities 13 and 14, we have:

$$\begin{aligned} k_{a(u)}(v) &= [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\ &\geq k_{b(u)}(u) \end{aligned}$$

For situation (ii), we have $rhs_{dom\ a(u)}(v) \geq rhs_{dom\ b(u)}(v)$ and $rhs_{objf\ a(u)}(v) = rhs_{objf\ b(u)}(v)$ and so $rhs_{a(u)}(v) \geq rhs_{b(u)}(v)$. Combining this with the fact that $g_{a(u)}(v) = g_{b(u)}(v)$, we have:

$$\begin{aligned} k_{a(u)}(v) &= [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\ &\geq [\min(g_{b(u)}(v), rhs_{b(u)}(v)) + h(s_{start}, v); \min(g_{b(u)}(v), rhs_{b(u)}(v))] \\ &= k_{b(u)}(v) \\ &\geq k_{b(u)}(u) \end{aligned}$$

Case 4: The key of vertex v changes, $v \neq u$, and vertex u was locally underconsistent.

Because u was locally underconsistent, $g_{b(u)}(u) < rhs_{b(u)}(u)$, which is equivalent to saying that $g_{objf\ b(u)}(u) < rhs_{objf\ b(u)}(u)$ **or** $g_{objf\ b(u)}(u) = rhs_{objf\ b(u)}(u) \wedge g_{dom\ b(u)}(u) < rhs_{dom\ b(u)}(u)$. Because $v \neq u$, we know that the g-value of v does not change and $g_{b(u)}(v) = g_{a(u)}(v)$. As such, the change in the key of v must be due to a change in its rhs-value as a result of the change in $g(u)$ on line 23 where $g_{a(u)}(u) = [\infty,$ DOMINATED]. This change can affect the rhs-value of v in one of two situations: (i) if v is a backward neighbor of u and $g_{dom\ b(u)}(u) =$NOT_DOMINATED, and $rhs_{objf\ b(u)}(v) = c(v, u) + g_{objf\ b(u)}(u)$, or (ii) v is a dominance neighbor of u and v was previously dominated by only u ($rhs_{dom\ b(u)}(v) =$DOMINATED) and v is not longer dominated by u because of the increase in $g_{objf}(u)$.

In situation (i), $rhs_{objf}(v)$ is guaranteed not to decrease, since $g_{objf}(v)$ is increased to ∞. Thus $rhs_{objf\ a(u)}(v) \geq rhs_{objf\ b(u)}(v)$. If $rhs_{objf\ a(u)}(v) > rhs_{objf\ b(u)}(v)$, then $rhs_{a(u)}(v) \geq rhs_{b(u)}(v)$ regardless of the values of $rhs_{dom\ a(u)}(v)$ and $rhs_{dom\ b(u)}(v)$. This also holds if $rhs_{objf\ a(u)}(v) = rhs_{objf\ b(u)}(v)$ and ($rhs_{dom\ a(u)} =$DOMINATED or $rhs_{dom\ b(u)}(v) =$NOT_DOMINATED). If, however, $rhs_{objf\ a(u)}(v) = rhs_{objf\ b(u)}(v)$ and($rhs_{dom\ a(u)} =$NOT_DOMINATED and $rhs_{dom\ b(u)}(v) =$DOMINATED) this implies that v is also a dominance neighbor of u, was dominated by only u, and is no longer dominated by u due to the increase in $g_{objf}(u)$. In this case, we need to consider situation (ii). Temporarily ignoring situation (ii), we have for situation (i):

$$\begin{aligned} k_{a(u)}(v) &= [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\ &\geq [\min(g_{b(u)}(v), rhs_{b(u)}(v)) + h(s_{start}, v); \min(g_{b(u)}(v), rhs_{b(u)}(v))] \\ &= k_{b(u)}(v) \\ &\geq k_{b(u)}(u) \end{aligned}$$

In situation (ii), $rhs_{objf\ a(u)}(v) = rhs_{objf\ b(u)}(v)$ and $rhs_{dom\ a(u)}(v) =$NOT_DOMINATED $< rhs_{dom\ b(u)} =$ DOMINATED, implying that $rhs_{a(u)}(v) < rhs_{b(u)}(v)$. However, since v was previously dominated by only u, we know that $rhs_{objf\ b(u)}(v) = rhs_{objf\ a(u)}(v) \geq g_{obfj\ b(u)}(u)$ and $g_{dom\ b(u)}(u) =$NOT_DOMINATED. As such, $rhs_{a(u)}(v) \geq g_{b(u)}(u)$.

Furthermore, we know that because u was underconsistent, $k_{b(u)}(u) = [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, u); \min(g_{b(u)}(u), rhs_{b(u)}(u))] = [g_{b(u)}(u) + h(s_{start}, u); g_{b(u)}(u)]$. We know further that $k_{b(u)}(v) = [\min(g_{b(u)}(v), rhs_{b(u)}(v)) + h(s_{start}, v); \min(g_{b(u)}(v), rhs_{b(u)}(v))] \geq k_{b(u)}(u)$ because u was popped off the open list before v. As such,

$$\begin{aligned} g_{b(u)}(v) + h(s_{start}, v) &= [g_{objf\ b(u)}(v) + h(s_{start}, v); g_{dom\ b(u)}(v)] \\ &\geq g_{b(u)}(u) + h(s_{start}, u) \\ &= [g_{objf\ b(u)}(u) + h(s_{start}, u); \text{NOT\_DOMINATED}] \end{aligned} \quad (15)$$

and

$$\begin{aligned} rhs_{b(u)}(v) + h(s_{start}, v) &= [rhs_{objf\ b(u)}(v) + h(s_{start}, v); \text{DOMINATED}] \\ &\geq g_{b(u)}(u) + h(s_{start}, u) \\ &= [g_{objf\ b(u)}(u) + h(s_{start}, u); \text{NOT\_DOMINATED}] \end{aligned} \quad (16)$$

From inequality 16, we have: $rhs_{objf\ a(u)}(v) + h(s_{start}, v) = rhs_{objf\ b(u)}(v) + h(s_{start}, v) \geq g_{objf\ b(u)}(u) + h(s_{start}, u)$. Putting this all together, we have:

$$\begin{aligned} k_{a(u)}(v) &= [\min(g_{a(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{a(u)}(v), rhs_{a(u)}(v))] \\ &= [\min(g_{b(u)}(v), rhs_{a(u)}(v)) + h(s_{start}, v); \min(g_{b(u)}(v), rhs_{a(u)}(v))] \\ &\geq [g_{b(u)}(u) + h(s_{start}, u); g_{b(u)}(u)] \\ &= [\min(g_{b(u)}(u), rhs_{b(u)}(u)) + h(s_{start}, v); \min(g_{b(u)}(u), rhs_{b(u)}(u))] \\ &= k_{b(u)}(u) \end{aligned}$$

□

Theorem 7. *The keys of the vertices that are selected for expansion on line 17 are monotonically nondecreasing over time until ComputeShortestPath() terminates.*

Proof. Assume a vertex u is selected for expansion on line 17. When it is selected for expansion, its key $k_{b(u)}(u)$ is the smallest of the set of all vertices on the open list, which is the same as the set of all inconsistent vertices according to Theorem 2. If a locally consistent vertex v becomes inconsistent as a result of this expansion, it is put on the open list with a key $k_{a(u)}(v) \geq k_{b(u)}(u)$ according to Theorem 4. If a locally inconsistent vertex v remains inconsistent after this expansion, its key on the open list is at least as large as $k_{b(u)}(u)$ according to Theorem 6. As such, the next vertex w to be expanded will have a key at least as large as $k_{b(u)}(u)$. □

Theorem 8. *Let $k = U.TopKey()$ during the execution of line 16. If vertex u is locally consistent at this point in time with $k(u) \leq k$, then it its rhs_{objf} value does not change until ComputeShortestPath() terminates. Furthermore, if $k(u) < k$, then neither its rhs_{objf} value nor its rhs_{dom} value changes and it will hence remain consistent until ComputeShortestPath() terminates.*

Proof. We will prove by contradiction that the Theorem holds.

Assume that vertex u is consistent, that is, $g(u) = rhs(u)$. Assume also that $k(u) \leq k = U.TopKey()$. Suppose that $rhs_{objf}(u)$ subsequently changes during the expansion of some vertex v. We know that $k_{b(v)}(u) \leq k_{b(v)}(v)$ since v is processed after u. Also, $k_{b(v)}(u) = [\min(g_{b(v)}(u), rhs_{b(v)}(u)) + h(s_{start}, u); \min(g_{b(v)}(u), rhs_{b(v)}(u))] = [g_{b(v)}(u) + h(s_{start}, u); g_{b(v)}(u)]$ since u is locally consistent. By the definitions of rhs-values (equations 8-12), $rhs_{objf}(u)$ can change during the expansion of v only if u is a backward neighbor (predecessor) of v. Whether $rhs_{objf}(u)$ increases or decreases, we have $g_{a(v)}(u) = g_{b(v)}(u)$ and as such, $k_{a(v)}(u) = [\min(g_{a(v)}(u), rhs_{a(v)}(u)) + h(s_{start}, u); \min(g_{a(v)}(u), rhs_{a(v)}(u))] \leq [g_{b(v)}(u) + h(s_{start}, u); g_{b(v)}(u)] = k_{b(v)}(u) < k_{b(v)}(v)$. That is, the new key of u is no greater than its previous key, which is itself no greater than the key of v. However, according to Theorem 4, if u is a backward neighbor of v and becomes inconsistent during the expansion of v, its new key should be strictly greater than that of v, which is a contradiction. As such, the first part of the theorem holds.

Now, to prove the second part of the theorem, assume again that u is consistent, that is, $g(u) = rhs(u)$. Also also that $k(u) < k = U.TopKey()$. Suppose $rhs_{dom}(u)$ subsequently changes during the expansion of some vertex v. Because v is expanded off the open list, we know that $k \leq k(v)$. As such, $k_{b(v)}(u) < k_{b(v)}(v)$. As in the proof for the first part of the theorem, we know that whether $rhs(u)$ increases or decreases, its new key, $k_{a(v)}(u)$ is no greater than its previous key, $k_{b(v)}(u)$ and as such, $k_{a(v)}(u) < k_{b(v)}(v)$. However, by Theorem 4, if u becomes inconsistent during the expansion of v its new key should be at least as large as that of v, which is a contradiction. This proves the second part of the theorem. □

Theorem 9. *Assume that vertex u has key $k_(u)$ and is selected for expansion on line 17. Assume that there exists some vertex v at this point in time with $k(v) \leq k(u)$. If, during the expansion of u, $rhs_{dom}(v)$ becomes* DOMINATED*, then $rhs_{dom}(v)$ remains* DOMINATED *until ComputeShortestPath() terminates.*

Proof. For $rhs_{dom}(v)$ to become DOMINATED during the expansion of u, u would have to dominate v according to the domain definition of dominance, and $g_{objf}(u)$ would have to decrease such that it becomes less than or equal to $rhs_{objf}(v)$. Thus, u must be overconsistent. It is made consistent on line 19, and at this point, $g_{objf\ a(u)}(u) = rhs_{objf\ a(u)}(u) = rhs_{objf\ b(u)}(u)$. Subsequently, a necessary requirement for $rhs_{dom}(v)$ to change from being DOMINATED to NOT_DOMINATEDs would be for $g_{objf}(u)$ to increase or $rhs_{objf}(v)$ to decrease.

In the first case, $g_{objf}(u)$ can increase only if u is underconsistent. u can become underconsistent if its rhs_{objf} value increases and/or its rhs_{dom} value increases. According to Theorem 8, the rhs_{objf} does not change until ComputeShortestPath() terminates. Furthermore, its rhs_{dom} value can only increase by changing from NOT_DOMINATED to DOMINATED. For $rhs_{dom}(u)$ to become DOMINATED, there must be another state, w, that dominates u. However, any vertex that dominates u also dominates v. As such, v would also be dominated by w and $rhs_{dom}(v)$ would remain DOMINATED.

In the second case, $rhs_{objf}(v)$ can only decrease if a state w is later selected for expansion such that v is a backward neighbor of w and $g_{dom\ a(w)}(w) =$NOT_DOMINATED and $c(v,w) + g_{objf\ a(w)}(w) < rhs_{objf\ b(w)}(v)$. However, if such a state exsists, it would be true that $k(w) < k(v) \leq k(u)$ and by Theorem 7, such a state would have been processed for expansion before v and u. As such, such a state cannot be processed after u and $rhs_{dom}(v)$ remains DOMINATED until ComputeShortestPath() terminates. □

Theorem 10. *Let $k = U.TopKey()$ during the execution of line 16. If vertex u is locally consistent at this point in time with $k(u) < k$, it remains locally consistent until ComputeShortestPath() terminates. Also, if a vertex u' is locally consistent and dominated at this point in time with $k(u') \leq k$, then it remains locally consistent and dominated until ComputeShortestPath() terminates.*

Proof. If a state is locally consistent, it can become inconsistent only if its rhs-value changes. According to Theorem 8, if u is locally consistent with $k(u) < k$, then neither its rhs_{objf} value nor its rhs_{dom} value will change until ComputeShortestPath terminates. As such, u remains locally consistent until ComputeShortestPath() terminates, proving the first part of the theorem. Also according to Theorem 8, if u' is locally consistent with $k(u') \leq k$, then its $rhs_{objf}(u)$ will not change until ComputeShortestPath() terminates. Furthermore, according to Theorem 9, if u' is dominated, $rhs_{dom}(u')$ will not change until ComputeShortestPath() terminates. As such, u' remains locally consistent and dominated until ComputeShortestPath() terminates. □

Theorem 11. *Let $k = U.TopKey()$ during the execution of line 16. If vertex u is locally consistent at this point in time with $k(u) \leq k$, but later becomes inconsistent, then it becomes underconsistent with $g_{objf} = rhs_{objf}$, $g_{dom} =$*NOT_DOMINATED *and $rhs_{dom} =$* DOMINATED.

Proof. A consistent vertex can become inconsistent as a result of a change in either its rhs_{objf} or rhs_{dom} values. According to Theorem 8, once a vertex becomes consistent, its rhs_{objf} does not change until ComputeShortestPath() terminates. Thus, if a consistent vertex becomes inconsistent, it is due to a change in its rhs_{dom} value: $g_{objf} = rhs_{objf}$ **and** $g_{dom} \neq rhs_{dom}$. According to Theorem 10, if u is dominated, it stays dominated. Thus, if u becomes inconsistent, $g_{dom} \neq$DOMINATED. Thus, $g_{dom} =$NOT_DOMINATED. Now, when u becomes inconsistent, $g_{dom} \neq rhs_{dom}$ and $g_{dom} =$NOT_DOMINATED. Thus, $rhs_{dom} =$ DOMINATED and the theorem holds.

□

Theorem 12. *If line 16 is changed to "while U is not empty", then ComputestShortestPath() expands each vertex at most four times; namely at most once when it locally underconsistent and not dominated, once when it locally underconsistent and dominated, once when it is locally overconsistent and not dominated, and once when it locally overconsistent and dominated. The g_{objf} values of all vertices after termination equal their respective goal distances.*

Proof. Assume that line 16 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. If it is dominated, it remains locally consistent and dominated according to Theorem 10 and is not expanded again. If it is not dominated, it remains locally consistent unless its rhs_{dom} value becomes DOMINATED, according to Theorem 11, at which point it becomes underconsistent. When a locally underconsistent vertex is selected for expansion, it is made overconsistent by setting its g_{objf} to ∞ and its g_{dom} value to DOMINATED.

Thus, these are the possibilities scenarios for the transitions of a vertex off and on the open list. Transitions off the open list (node expansions on line 17) are indicated by *expanded*$\rightarrow$, and transitions back onto the list (open list inserts on line 8) following updates are indicated by *updated*$\dashrightarrow$.

1. If the vertex is eventually dominated
   (a) (overconsisent, rhs_{dom} =DOMINATED) *expanded*$\rightarrow$ (consistent, rhs_{dom} =DOMINATED)
   (b) (overconsisent, rhs_{dom} =NOT_DOMINATED) *expanded*$\rightarrow$
   (consistent, rhs_{dom} =DOMINATED) *updated*$\dashrightarrow$
   (underconsistent, rhs_{dom} =DOMINATED) *expanded*$\rightarrow$ *updated*$\dashrightarrow$
   (overconsistent, rhs_{dom} =rhs_{dom} =DOMINATED) *expanded*$\rightarrow$ (consistent, rhs_{dom} =DOMINATED)
   (c) (underconsistent, rhs_{dom} =DOMINATED) *expanded*$\rightarrow$ *updated*$\dashrightarrow$
   (overconsistent, rhs_{dom} =DOMINATED) *expanded*$\rightarrow$ (consistent, rhs_{dom} =DOMINATED)
   (d) (underconsistent, rhs_{dom} =NOT_DOMINATED) *expanded*$\rightarrow$ *updated*$\dashrightarrow$
   (overconsisent, rhs_{dom} =NOT_DOMINATED) *expanded*$\rightarrow$
   (consistent, rhs_{dom} =DOMINATED) *updated*$\dashrightarrow$
   (underconsistent, rhs_{dom} =DOMINATED) *expanded*$\rightarrow$ *updated*$\dashrightarrow$
   (overconsistent, rhs_{dom} =DOMINATED) *expanded*$\rightarrow$ (consistent, rhs_{dom} =DOMINATED)
2. If the vertex is not eventually not dominated
   (a) (overconsistent, rhs_{dom} =NOT_DOMINATED) *expanded*$\rightarrow$ (consistent, rhs_{dom} =NOT_DOMINATED)
   (b) (underconsistent, rhs_{dom} =NOT_DOMINATED) *expanded*$\rightarrow$ *updated*$\dashrightarrow$
   (overconsisent, rhs_{dom} =NOT_DOMINATED) *expanded*$\rightarrow$ (consistent, rhs_{dom} =NOT_DOMINATED)

Thus, if the vertex is eventually not dominated, it is expanded at most twice: once when it is underconsistent and once when it is overconsistent. If the vertex is eventually dominated, it is expanded at most 4 times.

When all vertices are locally consistent, then $g(s) = rhs(s) = 0$ if $s = s_{goal}$ and $g(s) = \min_{s' \in Succ(u), g_{dom}(s') = \text{NOT\_DOMINATED}}(c(u, s') + g_{obfj}(s'))$ otherwise. Thus, the g_{objf} values satisfy the definition of the goal distances and thus equal them. □

Theorem 13. *Let $k = U.TopKey()$ during the execution of line 16. If vertex u is locally consistent at this point in time with $k(u) < k$, then the g_{objf} value of state u equals its minimax goal distance and one can follow a shortest path from u to s_{goal} by always moving from the current vertex s, starting at u, to any forward neighbor s' that minimizes $c(s, s') + g_{objf}(s')$ until s_{goal} is reached (ties can be broken arbitrarily).*

Proof. If U is empty, then the theorem follows from Theorem 14. Thus, we assume that U is not empty. Since u is locally consistent during the execution of line 16, $g(u) = rhs(u)$.

We first show by contradiction that $g(u) < [\infty,\text{NOT\_DOMINATED}] \Rightarrow g_{objf}(u) < \infty$. Assume first that $g(u)$ = [∞,DOMINATED] > [∞,NOT_DOMINATED]. Since u is consistent, $g(u) = rhs(u) = [\infty,\text{DOMINATED}]$. However, there is no key k such that $k(u) < k$, which is a contradiction of the assumption that U is not empty and that $k(u) < k = U.TopKey()$. Assume then that $g(u) = [\infty,\text{NOT\_DOMINATED}]$. Then, since u is consistent, $g(u) = rhs(u) = [\infty,\text{NOT\_DOMINATED}]$. Thus, k = [[∞,DOMINATED],[∞,DOMINATED]] since $k(u) < k$. Let v be a locally inconsistent vertex with key k. Such a vertex exists since we assume that U is not empty. Then, $g(v) = rhs(v) = [\infty,\text{DOMINATED}]$. Thus, vertex v must be locally consistent, which is a contradiction. Consequently, it holds that $g(u) < [\infty,\text{NOT\_DOMINATED}]$.

If $u = s_{goal}$, then $g(u) = rhs(u) = 0$ since vertex u is locally consistent and $rhs(u) = 0$ by definition. Thus, $g(u)$ equals its goal distance and one can trivially follow a shortest path from u to s_{goal} by starting at u and moving to any successor s' that minimizes $c(s, s') + g_{objf}(s')$ until s_{goal} is reached. Thus, we assume that $u \neq s_{goal}$.

Let w be a non-dominated forward neighbor (successor) of vertex u that minimizes $c(u, w) + g(w)$. Then, by the definition of rhs-values, $g_{objf}(u) = rhs_{objf}(u) = \min_{s' \in Succ(u), g_{dom}(s')=\text{NOT\_DOMINATED}}(c(u, s') + g_{objfj}(s')) = c(u, w) + g_{objfj}(w)$. Thus $g_{objf}(w) < g_{objf}(u) < \infty$ and, because $g_{dom}(w)$=NOT_DOMINATED, it follows that $g(w) < g(u)$. Furthermore, $h(s_{start}, w) \leq h(s_{start}, u) + c(u, w)$ (because heuristics obey the triangle inequality), implying that $g_{objf}(w) + h(s_{start}, w) \leq g_{objf}(w) + h(s_{start}, u) + c(u, w) = g_{objf}(u) + h(s_{start}, u)$. Again because $g_{dom}(w)$=NOT_DOMINATED, we can say that $g(w) + h(s_{start}, w) \leq g(u) + h(s_{start}, u)$. Thus:

$$\begin{aligned}
k(w) &= [\min(g(w), rhs(w)) + h(s_{start}, w); \min(g(w), rhs(w))] \\
&\leq [g(w) + h(s_{start}, w); g(w)] \\
&< [g(u) + h(s_{start}, u); g(u)] \\
&= [\min(g(u), rhs(u)) + h(s_{start}, u); \min(g(u), rhs(u))] \\
&= k(u) \\
&< k
\end{aligned}$$

As such, $k(w) < k(u) < k$ which means that w must be locally consistent during the execution of 16. Furthermore, because their keys are strictly less than the smallest key of any locally inconsistent vertex, Theorem 10 implies that both u and w remain consistent until the termination of ComputeShortestPath(), even if line 16 is changed to "while U is not empty". Furthermore, according to Theorem 12, $g_{objf}(u)$ and $rhs_{objf}(w)$ equal their respective goal distances after termination if line 16 is

changed to "while U is not empty". Thus, $g_{objf}(u)$ and $rhs_{objf}(w)$ must already equal their goal distances at the execution of line 16 when $k(w) < k(u) < k$.

If the goal distance of u is represented as $gd(u)$, then $gd(u) = c(u, w) + gd(w)$. Thus, moving from vertex u to any non-dominated vertex w that minimizes $c(u, w) + g(w)$ is the beginning of a shortest path from u to s_{goal}. This property can be repeatedly applied to show that one can follow a shortest path from u to s_{goal} by always moving from the current vertex s, starting at u, to any forward neighbor s' that minimizes $c(s, s') + g_{objf}(s')$ until s_{goal} is reached □

Theorem 14. *ComputeShortestPath() expands a vertex at most four times; namely at most once when it is locally underconsistent and not dominated, once when it is locally underconsistent and dominated, once when it is locally overconsistent and not dominated, and once when it is locally overconsistent and dominated. After termination, one can follow a shortest path from s_{start} to s_{goal} by always moving from the current vertex s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g_{objf}(s')$ until s_{goal} is reached (ties can be broken arbitrarily).*

Proof. According to Theorem 12, ComputeShortestPath() terminates after it has expanded every vertex at most four times if line 16 is changed to "while U is not empty". Even if line 16 is not changed, it will still terminate at least when U is empty because in this case U.TopKey() returns [[∞,DOMINATED], [∞,DOMINATED]] $> k(s_{start})$ because $rhs(s_{start}) = g(s_{start}) \leq$[$\infty$,NOT_DOMINATED] since all vertices are locally consistent and the start state by definition is non-dominated. Thus, the termination condition is satisfied. As such, ComputeShortestPath will terminate when U becomes empty, after it has expanded every vertex at most four times, if it does not terminate earlier. Furthermore, s_{start} satisfies the conditions of Theorem 13 after completion, and the Theorem follows directly from Theorem 13. □

8 Acknowledgments

This work was sponsored by the Jet Propulsion Laboratory, under contract "Reliable and Efficient Long-Range Autonomous Rover Navigation" (contract number 1263676, task order number NM0710764). 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 NASA or the U.S. Government. The authors wish to acknowledge Maxim Likhachev's contributions in reviewing this work.

References

[1] BHATTACHARYA, R., AND BHATTACHARYA, S. An exact depth-first algorithm for the pallet loading problem. *European Journal of Operational Research 110* (1998), 610–625.

[2] FRIGIONI, D., MARCHETTI-SPACCAMELA, A., AND NANNI, U. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms 34* (2000), 351–381.

[3] FUJINO, T., AND FUJIWARA, H. A method of search space pruning based on search state dominance. *Systems and Computers in Japan 25*, 4 (April 1994), 1–12.

[4] GONZALEZ, J. P., AND STENTZ, A. Planning with uncertainty in position: An optimal and efficient planner. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS '05)* (August 2005).

[5] HART, P. E., NILSSON, N. J., AND RAPHAEL, B. A formal basis for the determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics SSC-4*, 2 (1968), 100–107.

[6] HOROWITZ, E., AND SAHNI, S. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

[7] IBARAKI, T. The power of dominance relations in branch-and-bound algorithms. *J. ACM 24*, 2 (1977), 264–279.

[8] KOENIG, S., AND LIKHACHEV, M. D*lite. In *AAAI/IAAI* (2002), pp. 476–483.

[9] KOENIG, S., AND LIKHACHEV, M. Improved fast replanning for robot navigation in unknown terrain. Tech. Rep. GIT-COGSCI-2002/3, Georgia Institute of Technology, College of Computing, 2002.

[10] STENTZ, A. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)* (May 1994), vol. 4, pp. 3310 – 3317.

[11] TOMPKINS, P. *Mission-Directed Path Planning for Planetary Rover Exploration*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2005.

[12] YU, C., AND WAH, B. W. Learning dominance relations in combined search problems. *IEEE Transactions on Software Engineering 14*, 8 (August 1988), 1155–1175.