

DD* Lite: Efficient Incremental Search with State Dominance

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

Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213
{gertrude, axs, mbdias}@ri.cmu.edu

Abstract

This paper 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.

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 (Ibaraki 1977; Horowitz & Sahni 1978; Yu & Wah 1988; Fujino & Fujiwara 1994). 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 relationships have been successfully exploited in problems as varied

as generating tests for combinatorial circuits (Fujino & Fujiwara 1994) and solving the manufacturer's pallet loading problem (Bhattacharya & Bhattacharya 1998).

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 (Horowitz & Sahni 1978). Dominated states may be deleted without expansion in the search, thus eliminating entire branches of the search tree.

Heuristic search techniques such as A* (Hart, Nilsson, & Raphael 1968) have been successfully used in planning and many other problems. Dynamic or incremental search algorithms (Frigioni, Marchetti-Spaccamela, & Nanni 2000) 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* (Stentz 1994) and D* Lite (Koenig & Likhachev 2002) 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 (Tompkins 2005) or even uncertainty (Gonzalez & Stentz 2005). 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 (Ibaraki 1977; Horowitz & Sahni 1978; Yu & Wah 1988; Fujino & Fujiwara 1994), 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 (Tompkins 2005) 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.

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,

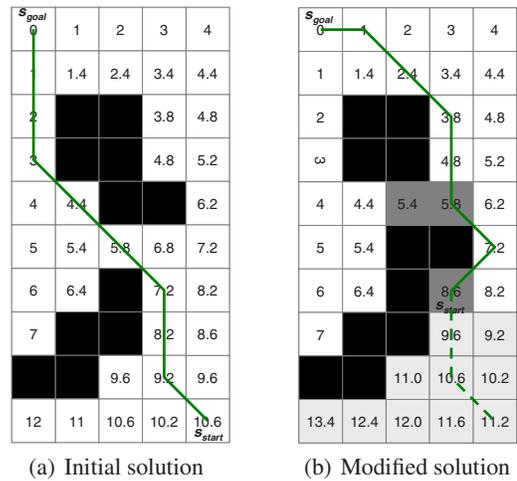


Figure 1: Incremental Search Example

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 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.

Incremental Search with State Dominance

The basic idea underlying search with state dominance is to identify and prune dominated states before they are expanded. 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.

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.

$$\begin{aligned} g(s) &= [g_{objf}(s), g_{dom}(s)] & (2) \\ rhs(s) &= [rhs_{objf}(s), rhs_{dom}(s)] & (3) \end{aligned}$$

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 $(g_{objf}(s) < rhs_{objf}(s)) \vee (g_{objf}(s) = rhs_{objf}(s) \wedge g_{dom}(s) < rhs_{dom}(s))$.

To focus the search, we define a heuristic function. 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 . 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 $[rhs_{objf}(s) + h(s_{start}, s), rhs_{dom}(s)]$. In DD* Lite, the heuristic function must be admissible and consistent. That is, it must represent a lower bound of the cost to the start, and it must satisfy the triangle inequality.

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 3, 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' .

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

$$rhs_{dom}(s) = \begin{cases} \text{NOT_DOMINATED} & \text{if } D(s) = \emptyset \\ \text{DOMINATED} & \text{otherwise} \end{cases} \quad (5)$$

where:

$$\begin{aligned} F(s) &= \{s' : s' \in Succ(s) \wedge g_{dom}(s') = \text{NOT_DOMINATED}\} & (6) \\ D(s) &= \{s' : s' \in DominanceNeighbors(s) \\ &\quad \wedge Dominate(s', s) \wedge (g_{objf}(s') \leq rhs_{objf}(s)) \\ &\quad \wedge (g_{objf}(s') + h(s_{start}, s')) \\ &\quad \leq rhs_{objf}(s) + h(s_{start}, s))\} & (7) \end{aligned}$$

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* 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”.

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 20) is processed by being made consistent on line 21. Cost changes are then propagated to predecessors and dominance neighbors on lines 22-23, 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 4. Changes to the g_{objf} value of a state may affect the rhs_{dom} value of its dominance neighbors as indicated in equation 5. 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 24) is processed by being made overconsistent on line 25. UpdateVertex() is then called on its predecessors and dominance neighbors as well as the node itself, to allow inconsistent states to be inserted

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   $U \leftarrow \emptyset$ ;
3  for all  $s \in S$ 
4      $rhs(s), g(s) \leftarrow [\infty, \text{NOT\_DOMINATED}]$ ;
5      $rhs(s_{goal}) \leftarrow [0, \text{NOT\_DOMINATED}]$ ;
6   $U.Insert(s_{goal}, CalculateKey(s_{goal}))$ ;

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

procedure ComputeRHS(s)
11  $F \leftarrow \{s' : s' \in Succ(s) \text{ and}$ 
    $g_{dom}(s') = \text{NOT\_DOMINATED}\}$ ;
12  $temp_{objf} \leftarrow \min_{s' \in F} (g_{objf}(s') + c(s, s'))$ ;
13  $temp_{dom} \leftarrow \text{NOT\_DOMINATED}$ ;
14 for all  $s' \in DominanceNeighbors(s)$ 
15     if Dominate( $s', s$ ) and  $g_{objf}(s') \leq temp_{objf}$ 
   and
    $g_{objf}(s') + h(s_{start}, s') \leq temp_{objf} + h(s_{start}, s)$ 
16      $temp_{dom} \leftarrow \text{DOMINATED}$ ;
   break;
17  $rhs(s) \leftarrow [temp_{objf}, temp_{dom}]$ ;

procedure ComputeShortestPath()
18 while  $U.TopKey() \leq CalculateKey(s_{start})$  or
    $rhs(s_{start}) \neq g(s_{start})$ 
19      $s \leftarrow U.Pop()$ ;
20     if  $g(s) > rhs(s)$ 
21          $g(s) \leftarrow rhs(s)$ ;
22         for all
23              $s' \in DominanceNeighbors(s) \cup Pred(s)$ 
   UpdateVertex( $s'$ );
24     else
25          $g(s) \leftarrow [\infty, \text{NOT\_DOMINATED}]$ ;
26         for all  $s' \in$ 
    $DominanceNeighbors(s) \cup Pred(s) \cup \{s\}$ 
27             UpdateVertex( $s'$ );

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

```

Figure 2: DD* Lite

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 18.

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.

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 list.

Theoretical Properties

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

Theorem 1 *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.*

Theorem 2 *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.*

Theorem 3 *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 $c(s, s') + g_{obj}(s')$ until s_{goal} is reached (breaking ties arbitrarily).*

An informal proof of Theorems 1 and 2 is based on two observations. First is the observation that the keys of the states selected for expansion on line 19 of the algorithm are monotonically nondecreasing over time until ComputeShortestPath() terminates. This implies that once a state s is made consistent on line 21, its rhs_{obj} 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 Theorem 2. 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.

Theorem 3 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_{obj} and rhs_{obj} values of all states on the path to the goal satisfy equation 4, and from the equation, none of the rhs_{obj} values are based on dominated states.

Formal proofs of these theorems appear in an extended technical report (Mills-Tettey, Stentz, & Dias 2006). 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.

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 dominance: 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 both admissible and consistent.

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.

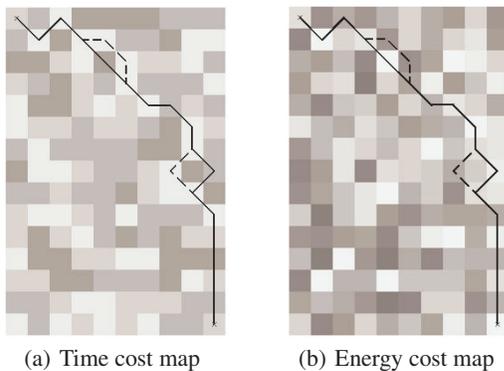
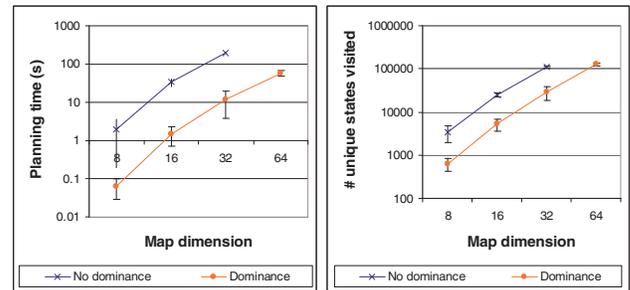


Figure 3: Example DD* lite plan superimposed on time and energy cost maps.

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.



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

Figure 4: Comparison of planning efficiency with and without dominance

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, 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.

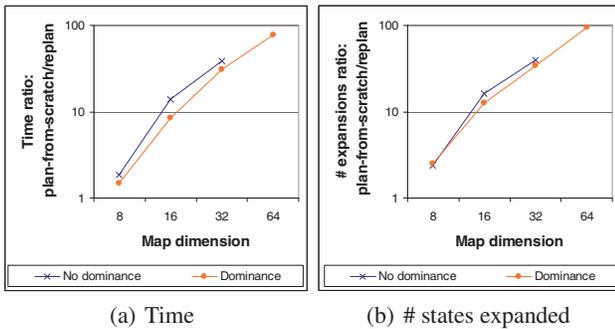


Figure 5: Ratio of performance cost of planning from scratch versus replanning, with and without dominance

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.

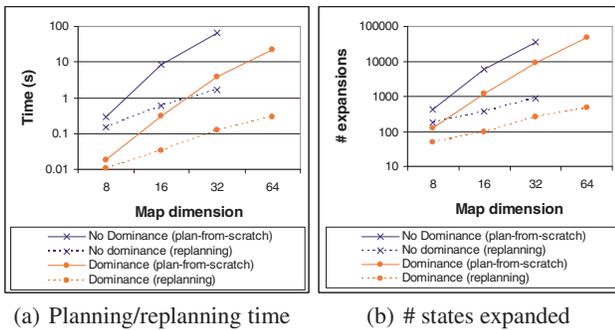


Figure 6: Comparison of efficiency of planning from scratch versus re-planning, with and without dominance

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.

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

- Bhattacharya, R., and Bhattacharya, S. 1998. An exact depth-first algorithm for the pallet loading problem. *European Journal of Operational Research* 110:610–625.
- Frigioni, D.; Marchetti-Spaccamela, A.; and Nanni, U. 2000. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms* 34:351–381.
- Fujino, T., and Fujiwara, H. 1994. A method of search space pruning based on search state dominance. *Systems and Computers in Japan* 25(4):1–12.
- Gonzalez, J. P., and Stentz, A. 2005. Planning with uncertainty in position: An optimal and efficient planner. In *Proceedings of the IEEE International Conference on Intelligent Robots and Systems (IROS '05)*.
- Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics* SSC-4(2):100–107.
- Horowitz, E., and Sahni, S. 1978. *Fundamentals of Computer Algorithms*. Computer Science Press.
- Ibaraki, T. 1977. The power of dominance relations in branch-and-bound algorithms. *J. ACM* 24(2):264–279.
- Koenig, S., and Likhachev, M. 2002. D*lite. In *AAAI/IAAI*, 476–483.
- Mills-Tettey, G. A.; Stentz, A.; and Dias, M. B. 2006. DD* Lite: Efficient incremental search with state dominance. Technical report, Carnegie Mellon University. (Forthcoming).
- Stentz, A. 1994. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)*, volume 4, 3310 – 3317.
- Tompkins, P. 2005. *Mission-Directed Path Planning for Planetary Rover Exploration*. Ph.D. Dissertation, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA.
- Yu, C., and Wah, B. W. 1988. Learning dominance relations in combined search problems. *IEEE Transactions on Software Engineering* 14(8):1155–1175.