Delayed D*: The Proofs

Dave Ferguson Anthony Stentz

CMU-RI-TR-04-51

September 2004

Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
SEP 2004		**00-00-2004 to 00-00-2004**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Delayed D*: The Proofs	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University ,Robotics Institute,Pittsburgh,PA,15213**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**10**	

1 Analytical Results

In this paper, we prove a number of properties of the Delayed D* algorithm introduced in [1], including its termination and correctness. In what follows, we deal with the fixed initial state version of Delayed D* (shown in Figures 1 and 2), but these results can easily be extended, following similar lines as in [2], to the navigation version.

1.1 Definitions and Heuristics

We first define terms used in our proofs and introduce some heuristic properties.

Let $g^*(s)$ denote the cost of a shortest path from $s \in S$ to the goal. Let $c^*(s, s')$ denote the cost of a shortest path from $s \in S$ to $s' \in S$. Let $h(s, s')$ denote the heuristic cost from state $s \in S$ to state $s' \in S$. For simplicity, let $h(s)$ denote the heuristic cost from the start state, s_{start}, to state $s \in S$.

We call a heuristic function h admissible if and only if it does not overestimate the shortest path cost between *any* two states, i.e., if and only if $h(s, s') \leq c^*(s, s')$ for all $s, s' \in S$. We call a heuristic function h backward consistent if and only if the following holds: $h(s_{start}) = 0$, and $h(s) \leq h(s') + c(s', s)$, for all states $s \in S$ and $s' \in Pred(s)$. Note that backward consistent heuristics are also admissible.

One property of backward consistent heuristics that will be useful to us later is that, for any states $s \in S$, $s' \in Pred(s)$, we have $h(s) \leq h(s') + c^*(s', s)$. We prove this below.

Theorem 1. *If h is a backward consistent heuristic, then for any states $s, s' \in S$, $h(s) \leq h(s') + c^*(s', s)$.*

Proof. By contradiction. Assume there exist some states $s, s' \in S$ such that $h(s) > h(s') + c^*(s', s)$.

Find the state $s^* \in S$ closest to s' along a shortest path from s' to s for which $h(s^*) > h(s') + c^*(s', s^*)$. Let $s_p \in S$ be the predecessor state to s^* along this shortest path. Then,

$h(s^*) \leq h(s_p) + c(s_p, s^*)$	h is backward consistent
$\leq (h(s') + c^*(s', s_p)) + c(s_p, s^*)$	s_p has $h(s_p) \leq h(s') + c^*(s', s_p)$
$= h(s') + (c^*(s', s_p) + c(s_p, s^*))$	rearranging
$= h(s') + c^*(s', s^*)$	since s_p is predecessor of s^* along path.

But we chose s^* such that $h(s^*) > h(s') + c^*(s', s^*)$. Contradiction. Thus, for all states $s, s' \in S$, $h(s) \leq h(s') + c^*(s', s)$. □

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

Initialize()
02. $U = \emptyset$;
03. for all $s \in S$
04. $\quad rhs(s_s) = g(s_g) = \infty$;
05. $rhs(s_{goal}) = 0$;
06. Insert$(U, s_{goal}, [h(s_{start}, s_{goal}), 0])$;

UpdateVertex(s)
07. if $(g(u) \neq rhs(u))$
08. $\quad$ Insert$(U, s,$ CalculateKey$(s))$;
09. else if $(g(s) = rhs(s))$ and $(s \in U)$
10. $\quad$ Remove(U, s);

UpdateVertexLower(s)
11. if $(g(u) > rhs(u))$
12. $\quad$ Insert$(U, s,$ CalculateKey$(s))$;
13. else if $(g(s) = rhs(s))$ and $(s \in U)$
14. $\quad$ Remove(U, s);

ComputeShortestPathDelayed()
15. while (U.MinKey() $\dot{<}$ CalculateKey(s_{start}) OR $g(s_{start}) \neq rhs(s_{start})$)
16. $\quad s = U$.Top();
17. $\quad$ if $(g(s) > rhs(s))$
18. $\quad\quad g(s) = rhs(s)$
19. $\quad\quad$ Remove(U, s);
20. $\quad\quad$ for all $x \in Pred(s)$
21. $\quad\quad\quad rhs(x) = min(rhs(x), c(x, s) + g(s))$;
22. $\quad\quad\quad$ UpdateVertexLower(x);
23. $\quad$ else
24. $\quad\quad g_{old} = g(s)$;
25. $\quad\quad g_s = \infty$;
26. $\quad\quad$ for all $x \in Pred(s) \cup s$
27. $\quad\quad\quad$ if $(rhs(x) = c(x, s) + g_{old}$ OR $x = u)$
28. $\quad\quad\quad\quad$ if $(x \neq s_{goal})$ $rhs(x) = min_{x' \in Succ(x)}(c(x, x') + g(x'))$;
29. $\quad\quad\quad$ UpdateVertex(x);

Figure 1: **The Delayed D* Algorithm: Fixed Initial State (Part 1).**

FindRaiseStatesOnPath()

30. $s = s_{start}, raise = false, loop = false, ctr = 0$;
31. while ($s \neq s_{goal}$ AND $loop = false$ AND $ctr < maxsteps$)
32. $x = argmin_{s' \in succ(s)}(c(s, s') + g(s'))$;
33. $rhs(s) = c(s, x) + g(x)$;
34. if ($g(s) \neq rhs(s)$)
35. UpdateVertex(x);
36. $raise = true$;
37. if ($x = s$)
38. $loop = true$;
39. else
40. $s = x$;
41. $ctr = ctr + 1$;
42. return $raise$;

Main()

43. Initialize();
44. ComputeShortestPathDelayed();
45. forever
46. Wait for changes in edge costs;
47. for all directed edges (u, v) with changed edge costs
48. $c_{old} = c(u, v)$;
49. Update the edge cost $c(u, v)$;
50. if ($c_{old} > c(u, v)$)
51. $rhs(u) = min(rhs(u), c(u, v) + g(v))$;
52. else if ($rhs(u) = c_{old} + g(v)$)
53. if ($u \neq s_{goal}$) $rhs(u) = min_{u' \in Succ(u)}(c(u, u') + g(u'))$;
54. UpdateVertexLower(u);
55. ComputeShortestPathDelayed();
56. $raise$ = FindRaiseStatesOnPath();
57. while ($raise$)
58. ComputeShortestPathDelayed();
59. $raise$ = FindRaiseStatesOnPath();

Figure 2: **The Delayed D* Algorithm: Fixed Initial State (Part 2).**

1.2 Delayed D* Proofs

We now prove a number of properties of the Delayed D* algorithm, beginning with its termination. We assume the heuristic function used is nonnegative and backward consistent, and that our state space is finite.

Theorem 2. *ComputeShortestPathDelayed() (CSPD) of the Delayed D* algorithm always terminates.*

Proof. By contradiction. Assume that CSPD never terminates. We show this results in a contradiction.

At any point in time, we can partition the state space into two sets:

S_1: those states that will be expanded again in a finite amount of time

S_2: those states that will never be expanded again.

Further, because of our assumption, we know that set S_1 will always be nonempty. Let's choose our point in time, t, to be sufficiently large so that set S_2 is maximized, i.e., after time t *all* states that are only expanded a finite number of times will not be expanded again.

We know from our construction of set S_1 that all members of this set will be expanded again in a finite amount of time after time t. Let t' be the first point in time after t at which all states in S_1 have been expanded at least once since time t. Finally, select t^* to be the first point in time after t' at which there is an overconsistent state at the top of the queue.

Such a t^* must exist. To see this, select a time $t'' > t'$ at which all states in S_1 have been expanded at least once since time t'. Now, each state can only be expanded once as an underconsistent state before it must be expanded as an overconsistent state (since underconsistent states have their g values set to ∞, so the next time they are made inconsistent they must have their rhs values less than their g values). Thus, either some state in S_1 was expanded as an overconsistent state between t' and t'', or the next state expanded must be an overconsistent state. In either case, we have a $t^* \leq t''$ at which there is an overconsistent state at the top of the queue.

Denote the overconsistent state at the top of the queue s. Since s is overconsistent, we know that $g(s) > rhs(s)$. Now, $rhs(s) = c(s, s') + g(s')$, where s' is some successor of s. We examine this state s'. Either s' is inconsistent at time t^*, or it is not.

Case 1: s' is inconsistent

If s' is inconsistent, it is either overconsistent or underconsistent. If it is overconsistent, i.e., $g(s') > rhs(s')$, then it must be on the queue, since any time a state is made overconsistent it is immediately added (lines {07 - 08; 11 - 12}). But if it is overconsistent, then its key is:

$$
\begin{aligned}
key(s') &\doteq [min(g(s'), rhs(s')) + h(s'), min(g(s'), rhs(s'))] & \\
&\doteq [rhs(s') + h(s'), rhs(s')] && g(s') > rhs(s') \\
&\dot{\leq} [rhs(s') + h(s) + c^*(s, s'), rhs(s')] && \text{heuristic backward consistent} \\
&\dot{<} [g(s') + h(s) + c^*(s, s'), g(s')] && g(s') > rhs(s') \\
&\doteq [(c^*(s, s') + g(s')) + h(s), g(s')] && \text{rearranging} \\
&\dot{<} [rhs(s) + h(s), rhs(s)] && rhs(s) \text{ computed from } g(s') \\
&\doteq [min(g(s), rhs(s)) + h(s), min(g(s), rhs(s))] && g(s) > rhs(s) \\
&\doteq key(s). && \text{definition of key value.}
\end{aligned}
$$

If this is the case, then we have $key(s') \dot{<} key(s)$ when both s and s' are on the queue, so s would not have been at the top of the queue. Contradiction.

If s' is underconsistent, i.e., $g(s') < rhs(s')$, then it may or may not be on the queue. If it *is* on the queue, then its key is:

$$
\begin{aligned}
key(s') &\doteq [min(g(s'), rhs(s')) + h(s'), min(g(s'), rhs(s'))] & \\
&\doteq [g(s') + h(s'), g(s')] & g(s') < rhs(s') \\
&\dot{\leq} [g(s') + h(s) + c^*(s, s'), g(s')] & \text{heuristic backward consistent} \\
&\doteq [(c^*(s, s') + g(s')) + h(s), g(s')] & \text{rearranging} \\
&\dot{<} [rhs(s) + h(s), rhs(s)] & rhs(s) \text{ computed from } g(s') \\
&\doteq [min(g(s), rhs(s)) + h(s), min(g(s), rhs(s))] & g(s) > rhs(s) \\
&\doteq key(s). & \text{definition of key value.}
\end{aligned}
$$

As above, $key(s') \dot{<} key(s)$ when both s and s' are on the queue, so s would not have been at the top of the queue. Contradiction.

If s' is *not* on the queue, then the only way we could have $g(s') < rhs(s')$ is if s' was underconsistent with the values $\{g(s'), rhs(s')\}$ when CSPD was called. This is because, during the course of the algorithm, any time a state is made underconsistent it is immediately added to the queue (lines {07 - 08}). But we know that at time t^*, all states in S_1 have been expanded at least once. Thus, state s' could not possibly still have these initial values without being on the queue. Contradiction.

Case 2: s' is consistent

If s' is consistent at time t^*, then let us consider how it could ever become inconsistent again. There are two possibilities. The rhs value of state s' could increase above its g value, or it could decrease below this value.

For the rhs value of state s' to increase, its current best successor state must have its g value increase. In other words, state s'' for which $rhs(s') = c(s', s'') + g(s'')$ must have its g value increase. Now, either state s'' is underconsistent at time t^* or it is not. If it is not, then by the same argument as in the previous lines, *its* current best successor must have its g value increase. We can repeat this line of reasoning to conclude that there must be *some* underconsistent state s^* at time t^* that is a *direct descendant* of state s'[1]. This state s^* must be a direct descendant or else its underconsistency would have no effect on state s'. We pick the first such underconsistent direct descendant state, s^*, i.e., the one with highest g value.

Because any underconsistent states at time t^* must be on the queue (see argument at end of Case 1 above), this state s^* must be on the queue at time t^*. It's key value is:

$$
\begin{aligned}
key(s^*) &\doteq [min(g(s^*), rhs(s^*)) + h(s^*), min(g(s^*), rhs(s^*))] & \\
&\doteq [g(s^*) + h(s^*), g(s^*)] & g(s^*) < rhs(s^*) \\
&\dot{\leq} [g(s^*) + h(s') + c^*(s', s^*), g(s^*)] & \text{heuristic backward consistent} \\
&\doteq [(c^*(s', s^*) + g(s^*)) + h(s'), g(s^*)] & \text{rearranging} \\
&\dot{<} [g(s') + h(s'), g(s')] & s^* \text{ is direct descendant of } s' \\
&\dot{\leq} [g(s') + h(s) + c^*(s, s'), g(s')] & \text{heuristic backward consistent} \\
&\doteq [(c^*(s, s') + g(s')) + h(s), g(s')] & \text{rearranging} \\
&\dot{<} [rhs(s) + h(s), rhs(s)] & rhs(s) \text{ computed from } g(s') \\
&\doteq [min(g(s), rhs(s)) + h(s), min(g(s), rhs(s))] & g(s) > rhs(s) \\
&\doteq key(s). & \text{definition of key value.}
\end{aligned}
$$

[1] In other words, there exists a state s^* that can be reached from state s' by always moving from the current vertex x, starting at s', to some successor y that minimizes $c(x, y) + g(y)$.

Thus, $key(s^*) \dot{<} key(s)$ when both s and s^* are on the queue, so s would not have been at the top of the queue. Contradiction.

For the rhs value of state s' to decrease, some successor of s' would need to lower *its* cost value (i.e., state s'' such that $g(s') > g_{new}(s') = c(s', s'') + g_{new}(s'')$). We can repeat this reasoning to create a sequence of states, s', s'', s''', etc. Eventually, one state in this sequence has to be on the queue in order to bring about the changes to those preceding it.

Further, during the course of the algorithm, states can only be made overconsistent when an overconsistent state is expanded (lines {20 - 22})[2]. Thus, some state in this sequence must be on the queue at time t^*, with its rhs value holding its future g value (which we have denoted g_{new}).

So this overconsistent state, call it s^*, must exist on the queue when s is expanded. Its key value is:

$$
\begin{aligned}
key(s^*) &\doteq [min(g(s^*), rhs(s^*)) + h(s^*), min(g(s^*), rhs(s^*))] & \\
&\doteq [rhs(s^*) + h(s^*), rhs(s^*)] & g(s^*) > rhs(s^*) \\
&\doteq [g_{new}(s^*) + h(s^*), g_{new}(s^*)] & rhs(s^*) = g_{new}(s^*) \\
&\dot{\leq} [g_{new}(s^*) + h(s') + c^*(s', s^*), g_{new}(s^*)] & \text{heuristic backward consistent} \\
&\doteq [(c^*(s', s^*) + g_{new}(s^*)) + h(s'), g_{new}(s^*)] & \text{rearranging} \\
&\dot{<} [g_{new}(s') + h(s'), g_{new}(s')] & g_{new}(s') \text{ caused by } g_{new}(s^*) \\
&\dot{\leq} [g_{new}(s') + h(s) + c^*(s, s'), g_{new}(s')] & \text{heuristic backward consistent} \\
&\dot{\leq} [(c^*(s, s') + g_{new}(s')) + h(s), g_{new}(s')] & \text{rearranging} \\
&\dot{<} [rhs(s) + h(s), rhs(s)] & rhs(s) \text{ lowered by } g_{new}(s') \\
&\doteq [min(g(s), rhs(s)) + h(s), min(g(s), rhs(s))] & g(s) > rhs(s) \\
&\doteq key(s). & \text{definition of key value.}
\end{aligned}
$$

Thus, $key(s^*) \dot{<} key(s)$ when both s and s^* are on the queue, so s would not have been at the top of the queue. Contradiction.

Conclusion:

Our assumption that set S_1 is nonempty has led to a contradiction. Therefore, set S_1 must be empty and so CSPD must terminate in finite time.

□

Theorem 3. *After ComputeShortestPathDelayed() terminates, for all $s \in S$ with $key(s) \dot{<} key(s_{start})$ we have $g(s) \leq rhs(s)$.*

Proof. Suppose s is a state with $g(s) > rhs(s)$ after ComputeShortestPathDelayed() has terminated. We wish to show that $key(s) \dot{\geq} key(s_{start})$.

There are only three possible ways in Delayed D* that a state can ever find itself with its g value greater than its rhs value. Firstly, it could have had its rhs value lowered to become less than its g value by a changed arc cost, at lines {50 - 51}. If this occurred, the state would have been added to the queue at line {54}.

Secondly, it could have had its rhs value lowered to become less than its g value by the expansion of an overconsistent successor state, at lines {20 - 21}. Again, if this occurred, the state would have been added to the queue at line {22}.

[2]when an underconsistent state x is expanded, the predecessor states which use x for their rhs values are updated, but there is no way this could result in a decrease of their rhs values: if the cost of using some other state is less than the cost of using x given its previous $g(x)$ value, then that other state has had its g value decrease since the last time the rhs value for the current state was computed. But if its g value has decreased, then when it decreased the rhs value for the current state would have been updated. Thus, the rhs value of the current state can only increase.

Finally, it could have been expanded as an underconsistent state and had its g value set to ∞, at line $\{25\}$. If this occurred, the state would have had its rhs value updated at lines $\{26 - 29\}$. If its new rhs value was less than ∞, the state would have been put back on the queue at line $\{08\}$.

If s has $g(s) > rhs(s)$ after CSPD terminates, then one of these three events had to have occurred most recently. In other words, the last time state s had its g or rhs value change, it had to have been put back on the queue. But this means that it was on the queue when the function terminated. Since the function doesn't terminate until the minimum key value on the queue is at least as large as the key of the start state, we must conclude that $key(s) \dot{\geq} key(s_{start})$.

Thus, for all $s \in S$ with $key(s) \dot{<} key(s_{start})$, $g(s) \leq rhs(s)$ when ComputeShortestPathDelayed() terminates. □

Theorem 4. *After ComputeShortestPathDelayed() terminates, for all $s \in S$ with $key(s) \dot{<} key(s_{start})$ we have $g(s) \leq g^*(s)$.*

Proof. By contradiction. Assume there is some nonempty set of states R containing all $s \in S$ such that $key(s) \dot{<} key(s_{start})$ and $g(s) > g^*(s)$ after CSPD terminates. We know that we can also define a nonempty set of states O containing all $s \in S$ such that $key(s) \dot{<} key(s_{start})$ and $g(s) \leq g^*(s)$. This set O is guaranteed to be nonempty because, at the very least, the goal will be a member.

Suppose s is an element in R for which an *optimal successor* is in set O[3]. Such an s must exist, as the optimal successors of each state in R can be followed to the goal, which is in set O, so there must be some state in R for which an optimal successor is in set O[4].

Now, the optimal successor of this state s, call this s', has $g(s') \leq g^*(s')$ as it is in set O. But when s' had its g value set to this value, it would have updated the rhs values of all possible predecessor states (lines $\{20 - 22\}$). Each of these states would use $g(s')$ to update its rhs value if this would provide a lower value than its current rhs value. Thus,

$$\begin{aligned} rhs(s) &\leq c(s, s') + g(s') && rhs(s) \text{ updated to be at least as low as } c(s, s') + g(s') \\ &\leq c(s, s') + g^*(s') && g(s') \leq g^*(s') \\ &= g^*(s) && s' \text{ is optimal successor of } s. \end{aligned}$$

But this means that s is *not* in R. Contradiction. Thus, our assumption that set R is nonempty is incorrect. □

Theorem 5. *After FindRaiseStatesOnPath() (FRSOP), if no underconsistent states have been added to the queue, then an optimal solution path can be followed from s_{start} to s_{goal} by moving from the current state s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g(s')$.*

Proof. If, after FRSOP, no states have been added, then we have arrived at the goal by starting at s_{start} and repeatedly moving from the current state s to any successor s' that minimizes $c(s, s') + g(s')$ (line $\{32\}$). Further, we have not encountered any states s along this path with $g(s) < rhs(s)$ (line $\{34\}$). Now, all states along this path must also have key values less than $key(s_{start})$, since they each contribute to the current g value of state s_{start}. Thus, according to Theorem 4, each of these states s has $g(s) \leq g^*(s)$. But if we were able to traverse the entire path without encountering *any* state s with $g(s) < rhs(s)$, then every state on this path is consistent, and the g value of each

[3] In other words, if all states were consistent and had their optimal g values, then $g^*(s) = c(s, s') + g^*(s')$, for some successor s'. We call this state s' an *optimal successor* of s.

[4] The only exception is if *no* element in set R has a path to the goal, in which case all elements in O have g^* values of ∞, so clearly cannot have their g values greater than their g^* values - a contradiction.

state must in fact equal its actual cost when following this path. Since the actual cost from any state s cannot possibly be less than $g^*(s)$, we must have $g(s) = g*(s)$ for each state along this path, including the state s_{start}. Since these costs were derived from following the traversed path, this path must be an optimal path. □

Theorem 6. *The Delayed D* algorithm always terminates and when it does, an optimal solution path can be followed from s_{start} to s_{goal} by moving from the current state s, starting at s_{start}, to any successor s' that minimizes $c(s, s') + g(s')$.*

Proof. The only nontrivial portion of the Delayed D* algorithm is the loop at lines {57 - 59}. We have already shown, in Theorem 2, that CSPD will always terminate. Further, Theorem 4 proved that when it does terminate, we have $g(s) \leq g^*(s)$, for all $s \in S$ such that $key(s) < key(s_{start})$. Now, when FRSOP terminates (which it must, since it has a counter that expires after $maxsteps$), either some underconsistent states have been found along the current path from s_{start} to s_{goal}, or an optimal solution path exists (by Theorem 5). In the latter case, the $raise$ flag is set to $false$ and the Delayed D* update phase (the loop at lines {57 - 59}) terminates. In the former case, the underconsistent states are added to the queue and CSPD is called. We can show that this loop will only be performed a finite number of times as follows.

Consider the first underconsistent state encountered along the path traversed in FRSOP the first time the loop is entered. Call this state s. Since this state was reached from the start state, s_{start}, by always moving from the current vertex x to some successor y that minimizes $c(x, y) + g(y)$, (line {32}) it is a direct descendant of s_{start}. Since this state is the ***first*** underconsistent direct descendant of s_{start} along this path, we must have $g(s_{start}) \geq c^*(s_{start}, s) + g(s)$[5]. Then,

$$
\begin{aligned}
key(s) &\doteq [min(g(s), rhs(s)) + h(s), min(g(s), rhs(s))] & \\
&\doteq [g(s) + h(s), g(s)] & g(s) < rhs(s) \\
&\dot{\leq} [g(s) + h(s_{start}) + c^*(s_{start}, s), g(s)] & h \text{ is backward consistent} \\
&\doteq [(g(s) + c^*(s_{start}, s)) + h(s_{start}), g(s)] & \text{rearranging} \\
&\dot{<} [g(s_{start}) + h(s_{start}), g(s_{start})] & g(s_{start}) \geq c^*(s_{start}, s) + g(s) \\
&\doteq key(s_{start}) & \text{definition of key value}
\end{aligned}
$$

where $key(s_{start})$ is the key of the start state before CSPD is called. Since the only difference in CSPD between the upcoming call and its last call is that there have been some underconsistent states added to the queue, there is no way that the key value of the start state can *decrease* during this call to CSPD. Thus, since $key(s) \dot{<} key(s_{start})$ when CSPD is called, and $key(s)$ cannot decrease until $g(s)$ decreases (which will not happen until s is expanded), we know that s will be expanded during this call of CSPD. This means that s is made overconsistent at some point during this call of CSPD.

Now, s may be made underconsistent again, during this call of CSPD or some subsequent call. But, during this update phase (i.e., while the loop at lines {57 - 59} is still being performed), it will never again be the first underconsistent state encountered along the path traversed in FRSOP. This is because, if s is made underconsistent again, then it is automatically put back on the queue, at lines {29; 07 - 08}. If s is then later encountered as the first underconsistent state encountered along the path traversed in FRSOP, then from before we know that its key value is:

[5]Otherwise, if $g(s_{start}) < c^*(s_{start}, s) + g(s)$, then there is some other state s' between s_{start} and s along this path with $g(s') < rhs(s')$. Contradiction.

$$
\begin{aligned}
key(s) &\doteq [min(g(s), rhs(s)) + h(s), min(g(s), rhs(s))] & \\
&\doteq [g(s) + h(s), g(s)] & g(s) < rhs(s) \\
&\dot{\leq} [g(s) + h(s_{start}) + c^*(s_{start}, s), g(s)] & h \text{ is backward consistent} \\
&\doteq [(g(s) + c^*(s_{start}, s)) + h(s_{start}), g(s)] & \text{rearranging} \\
&\dot{<} [g(s_{start}) + h(s_{start}), g(s_{start})] & g(s_{start}) \geq c^*(s_{start}, s) + g(s) \\
&\doteq key(s_{start}) & \text{definition of key value.}
\end{aligned}
$$

But we know that s is on the queue, so if its key value is less than the key value of the start state, then it would have been expanded the previous call to CSPD. Contradiction. Thus, once a state has been the first underconsistent state encountered along the path traversed in FRSOP, it will never again be in this position. Since our state space is finite, this means that FRSOP can only return $true$ a finite number of times. Thus, eventually FRSOP will return $false$ and the entire update phase will terminate, leaving an optimal solution path from the start to the goal (by Theorem 5). □

2 Acknowledgement

This work was sponsored by the U.S. Army Research Laboratory, under contract "Robotics Collaborative Technology Alliance". The views contained in this document are those of the authors and do not represent the official policies or endorsements of the U.S. Government.

References

[1] D. Ferguson and A. Stentz, "The Delayed D* Algorithm for Efficient Path Replanning," submitted to *IEEE International Conference on Robotics and Automation (ICRA), 2005*.

[2] M. Likhachev and S. Koenig, "Lifelong Planning A* and D* Lite: The Proofs," College of Computing, Georgia Institute of Technology, Tech. Rep., 2001.