Navigation on Density-Unbalanced Terrain

Qiang Han, Weiya Yue
University of Cincinnati
School of Electronic & Computing System
Cincinnati, OH 45220, USA
hanqg@ucmail.uc.edu, weiyayue@hotmail.com

Abstract

Navigation algorithms have shown to be important in many practical applications. In an unknown or constantly changing environment, D* Lite, a classic dynamic algorithm, replans a shortest path in an efficient manner. However, if there are multiple shortest paths, the D* Lite algorithm arbitrarily selects one. When it comes to a density-unbalanced terrain, the shortest paths with the same cost may have different meaning. The algorithm performance largely depends on how crowded are the areas through which the selected shortest path traverses. In this paper, we propose the *density-aware* D* Lite algorithm, DAD* Lite, which is capable to take into account crowded areas and avoid them to find a (best) shortest path even when this has more detours than other shortest paths. Experiments show DAD* Lite improves D* Lite by giving better results on *successful runs* and *moving distance*.

Introduction

Navigation algorithms, largely used to develop autonomous vehicles, intelligent agents, etc, are an important area of study in artificial intelligence. Under a dynamic environment, knowledge of the terrain – initially partially known, or unknown – is updated as the agent (for example, an exploring planet rover, and vehicle parking) moves. Replanning in dynamic circumstances is a very practical problem and it is a key part of a navigation algorithm. D* Lite (Koenig & Likhachev 2002) (Koenig & Likhachev 2005) has been proven and largely used as an efficient dynamic navigation algorithm.

The D* Lite algorithm seeks to find a minimum cost path from the start point to the goal point in the dynamic environment. Terrain information is modeled as an undirected graph $G(V, E)$ with a start vertex v_s and a goal vertex v_g. An edge cost function, denoted by $c(v, u)$, is associated to each directed edge (v, u), and the cost of a path is the sum of all the edge costs along that path. Determining the global minimum cost path from v_s to v_g is by no means trivial since the known environment information held by the agent may change unpredictably.

When the environment is static, the well known A* algorithm (Hart, Nilsson, & Raphael 1968) uses a vertex evaluation function $f(v)$ to determine the order in which the algorithm chooses vertices in the search tree to build the path from v_s to v_g. The evaluation function, $f(v)$ has two additive components, that is, $f(v) = g(v) + h(v)$, where $g(v)$ is the actual cost from v_s to the current vertex v, and $h(v)$ is a cost estimating heuristic from v to v_g. Another algorithm, *Lifelong Planning* A* (LPA*) (Koenig, Likhachev, & Furcy 2004) (Koenig & Likhachev 2001), introduces an additional component, $rhs(v)$, which is calculated using the updated *g-value* of v's predecessors and thus potentially better informed and more updated than v's own *g-value*. However, LPA* only recalculates the lowest cost path when the agent is at v_s and it does not replan as the agent moves and finds changes.

D* Lite can be treated as a dynamic version of LPA*. However, unlike LPA*, which detects environment changes globally, D* Lite employs a parameter called *sensor-radius*. Only vertices that are within *sensor-radius* from the current vertex are exactly in sight and the agent has accurate information on these vertices. Therefore, the agent knows only part of the terrain precisely, beyond which it holds old information, which might be out of date.

As the D* algorithm (Stentz 1995) (Stentz 1997), D* Lite performs a backward search from v_g to v_s. For D* and all its descendant algorithms, the backward search is the key point to their success, because the *g-value* of every node v in G is exactly the path cost from v_g to v and can be reused after the agent moves. When edge cost changes are detected, and the *g-values* of the vertices affected by these changes need to be updated, D* Lite propagates the computation from the affected vertices to v_s. In most cases, the backward propagation expands much less vertices than a forward search.

D* Lite also uses the "more informed" rhs function to make better vertex updates during expansion. The rhs function in D* Lite is defined by

$$rhs(v) = \begin{cases} \min_{v' \in succ(v)} g(v') + c(v, v') & v \neq v_g \\ 0 & \text{otherwise.} \end{cases}$$

Three states are defined for a vertex v: the vertes is *locally consistent* if $rhs(v) = g(v)$, *locally overconsistent* if $rhs(v) < g(v)$, and *locally underconsistent* if $rhs(v) > g(v)$. To generate the shortest path, D* Lite maintains

the vertices in a priority queue with in ascending order of $key - values$ defined as $min(g(v), rhs(v)) + h(v_s, v)$.

Every time an inconsistent vertex is updated by making $g(v) = rhs(v)$, the algorithm needs to check whether its neighbor becomes inconsistent. If this happens, this neighbor is added to the queue. D* Lite propagates calculations until all vertices on the path, including v_s, are locally consistent. When selecting the shortest path, the agent moves to its neighbor vertex with the minimum of *g-value* plus the edge cost, which is the maximum-g-decrease-value. If the agent detects any changes that have been made since the last round, this makes these changed vertices inconsistent, and D* Lite adds them to the priority queue for possible update. Because D* Lite only updates partially inconsistent vertices, making (g, rhs) consistent instead of all vertices, it can perform much more efficient than the other navigation algorithms.

D* Lite has been improved by avoiding unneccessary calculations in the case where the previous planned shortest path is still available and considering the new changes detected there is no better solution to replace this path (Yue & Franco 2009) (Yue & Franco 2010). Additionally, a variant of D*, ID* Lite, seeks to reduce calculation by first expanding the vertices which are more likely to contribute a shortest path (Yue *et al.* 2011). A threshold is used in ID* Lite to control updates. Only the vertices with *key-value* smaller than the threshold can be updated. The threshold value is increased gradually, until the shortest path is found. Due to the way of performing undates, ID* Lite avoids unnecessary update significantly and it shows much better results than D* Lite.

There are many other variants from D* Lite, which deal with different constraints or requirements. For example, DD* Lite (Mills-Tettey, Stentz, & Dias 2006) combines D* Lite with a technique of detecting dominance relationships to solve navigation problems with global constraints. By using the dominance relationship it prunes the search tree obtaining a fast planning. The anytime algorithm family, such as AWA* (Hansen & Zhou 2007), ARA* (Likhachev, Gordon, & Thrun 2003), AD* (Likhachev *et al.* 2005), and IAD* (Yue *et al.* 2012), share the so called *inflated heuristics* strategy, according to which the evaluation function $f(v)$ is replaced by $f'(v) = g(v) + \epsilon \cdot h(v)$, where $\epsilon >= 1$ denotes the *inflation factor*. Its effect is to increase the weight of *h-value* in $f(v)$, which causes fewer vertices to be updated, and a sub-optimal path is returned. Thus, anytime algorithms work best in time-limited and suboptimal solution acceptable environments.

A new algorithm, *Density-Aware* D* Lite (DAD* Lite), which replans a minimum cost path in a density-unbalanced environment is introduced in this study. Section Planning on Density-Unbalanced Terrain describes DAD* Lite and motivation behind it. The experimental setup, results, and analysis are presented in Section Experiments. Finally, Section Conclusion and Future Work concludes the current study and presents future work.

Planning on Density-Unbalanced Terrain

Motivation for DAD* Lite

D* Lite and all its variants conduct experiments on random terrains. For example, each node on the grid world is selected as a block with the same probability and all the unblocked edges have equal cost. However, the real world environments are much more complex for which the assumption of an even-distribution terrain does not hold. Moreover, these algorithms do not consider the information caused by the unbalance, when, in fact, such information can help develop a more efficient navigation algorithm. Fig. 1 is a Mars panorama taken by Mars Exploration Rover (Nasa 2012). In the middle of the picture, there is an area crowded with rocks, while in other regions the terrain is clear of rocks and spacious.

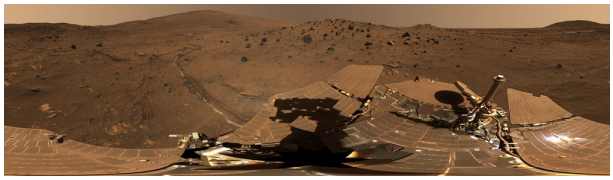


Figure 1: Mars Panorama taken by Mars Exploration Rover

Density-unbalanced terrain can be illustrated in a grid world as shown in Fig. 2, where $v_{i,j}$ denotes the i^{th} row and j^{th} column vertex. Fig. 2 shows three shortest paths from $v_s = v_{3,3}$ to $v_g = v_{0,0}$. When the agent starts to move from v_s, D* Lite picks up one of v_s's on-the-shortest-path successors, $v_{3,2}$ or $v_{2,3}$ arbitrarily. Based on the perception of human beings, going to $v_{2,3}$ is better, because this area is spacious while the bottom left side of the grid is more crowded. $Path_1$ may be blocked more in the future, and it does not have a detour. However, D* Lite makes decisions on the next move only depending on $c(v_s, v) + g(v)$, $v \in succ(v_s)$, thus the agent has the same probability to go to the crowded area or the spacious area.

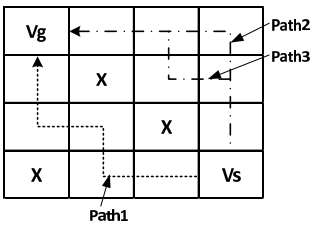


Figure 2: A 4×4 4-directional grid example

Heuristics in DAD* Lite

With the above observations, designing an algorithm able to select a shortest path in the most spacious region, thus avoiding crowded areas is an important problem. To solve this problem, two issues must be considered.

The first issue concerns the pool of candidate paths, in particular, how to obtain them. In DAD* Lite, the candidate pool contains all the shortest paths from v_s to v_g. One could argue that a *slightly-longer-than-shortest path* that goes through a much more spacious and easier route each of the shortest paths, will potentially avoid more interruptions and finally get to v_g with the minimum overall cost. It may help to overcome the shortcomings of the D* Lite family algorithms due to the unpredictable future changes (dynamic environment) and the inaccurate information (limitation of the sensor radius): D* Lite only computes the shortest path based on *local and temporary information* which turns out not to be a globally optimal solution almost certainly. However we treat this non-trivial problem as our future work and in this paper, the proposed DAD* Lite algorithm considers only all the shortest paths as the candidate pool.

As a descendant of D* Lite, DAD* Lite uses the same procedure to calculate shortest paths. D* Lite finds all the shortest paths after calling *ComputerShortestPath*() shown in the pseudocode of (Koenig & Likhachev 2005). The reason is that when one child of a vertex has been updated to be consistent, then all its other children will be updated to be consistent. So if one shortest path has been found, supposing an other arbitrary path p from v_s to v_g is also shortest, then the child of v_c on p must be updated and consistent too. Iteratively, all the vertices on p are updated and consistent, and thus all alternative shortest paths can be found, as shown in (Yue & Franco 2009) and (Yue & Franco 2010). The correctness of DAD* Lite follows from that of D* Lite, and thus DAD* Lite is guaranteed to find all the shortest paths.

The second issue concerns the design of heuristics to select a path along with less crowded areas. DAD* Lite chooses two factors to represent the how crowded a path is:

- Number of blocked path neighbors:
- Number of detours to other shortest paths:

The heuristic value of a path is an accumulation of each vertex's evaluation on this path, and DAD* Lite calculates the evaluation value of each vertex along the shortest paths. The evaluation value reflects the information of both the number of blocked neighbors from this vertex to the goal and the number of detours to the other shortest paths. The detailed computation of the evaluation value is shown next.

DAD* Lite Implementation

Fig. 3 and Fig. 4 illustrate the DAD* Lite pseudocode. Since DAD* Lite is built on D* Lite only the new and changed subroutines are shown; the omitted D* Lite subroutines can be found in (Koenig & Likhachev 2005). In the main function in Fig. 4, when *ComputerShortestPath*() in Line 03 and Line 16 finishes, all the shortest paths are found. Then DAD* Lite calls *GetDAPath*() which aims to find the shortest path avoiding crowded areas. Another difference from D* Lite in the main function, is that when the agent moves, it just moves to $next(v_s)$. $next(v)$ is the next vertex from v based on the density-aware path.

In Fig. 3, firstly *GetDAPath*() initializes $eval(v)$ for vertices which are visited in the last round of *GetDAPath*() computation. Then it calls *GetDAPath*(v). *GetDAPath*(v) is a re-

Procedure Initialize():
01. $U = \emptyset$;
02. $k_m = 0$;
03. for all $v \in V$ $rhs(v) = g(v) = \infty; eval(v) = 0$;
04. $rhs(v_g) = 0$;
05. $eval(v_g) = 1$;
06. U.Insert(v_g, **CalcKey**(v_g));

Procedure GetDAPath(u)
01. if($u \neq v_g$)
02. for all $v \in succ(u)$
03. if ($rhs(u) = g(v) + c(u, v)$ && $eval(v) = 0$)
04. $eval(u)+ =$**GetDAPath**(v);
05. $next(u) = arg\max_{v \in succ(u) \&\& rhs(u) = g(v) + c(u,v)} eval(v)$;
06. $cnt =$ # of blocked $v \mid v \in pred(u)$;
07. $eval(u) = eval(u)/2^{cnt}$;
08. return ($eval(u)$);

Procedure GetDAPath()
01. for all visited v in last round except v_g
02. $eval(v) = 0$;
03. **GetDAPath**(v_s);

Figure 3: Subroutines of DAD* Lite

Procedure Main():
01. $v_{last} = v_s$;
02. **Initialize**();
03. **ComputeShortestPath**();
04. **GetDAPath**();
05. while ($v_s \neq v_g$)
06. /* if ($g(v_g) = \inf$) then there is no known path */
07. $v_s = next(v_s)$;
08. Agent moves to v_s;
09. Scan graph for changed edge costs;
10. if any edge cost changes were observed
11. $k_m = k_m + h(v_{last}, v_s)$;
12. $v_{last} = v_s$;
13. for all directed edges (u, v) with changed edge costs
14. Update the edge cost $c(u, v)$;
15. **UpdateVertex**(u);
16. **ComputeShortestPath**();
17. **GetDAPath**();

Figure 4: Main function of DAD* Lite

cursive function, which computes $eval(v)$ for each vertex v on the shortest paths. Lines 02-07 show how the heuristics are calculated. Lines 02-04 sum the $eval(v)$ of u's successors only if its successor v is on the shortest paths. This calculation gives higher value for the vertices with more detours. Line 05 chooses the vertex v which has the highest evaluation value from all u's successors and saves it to $next(u)$. In line 06, cnt counts the number of the blocked v in u's predecessors. The larger cnt, the higher probability that a block will move to u and block the path from u to v_g. Line 07 gives the formula of $eval(u)$, which is is directly proportional to the sum of u's *on-the-shortest-path* successors' evaluation value, and is inversely proportional to two to the power of the number of blocks around u.

An Example

Fig. 5 shows how DAD* Lite works in a grid world example. In Fig. 5(a), (g, rhs) values are calculated in *ComputerShortestPath*(). *GetDAPath*() calculates $eval(v)$ in depth-first-search fashion from the root v_s. For example, in Fig. 5(b) to calculate $eval(v_{1,3})$, its two successors $v_{1,2}$ and $v_{0,3}$ evaluation values are $\frac{1}{32}$ and $\frac{1}{32}$ respectively, and $v_{1,3}$ has one blocked neighbor. So $eval(v_{1,3})$ is calculated as $(\frac{1}{32} + \frac{1}{32})/2^1 = \frac{1}{32}$.

Vg	(1,1)	(2,2)	(3,3)
(1,1)	X	(3,3)	(4,4)
(2,2)	(3,3)	X	(5,5)
X	(4,4)	(5,5)	Vs

(a) (g, rhs) value

Vg	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{32}$
$\frac{1}{4}$	X	$\frac{1}{32}$	$\frac{1}{32}$
$\frac{1}{16}$	$\frac{1}{64}$	X	$\frac{1}{128}$
X	$\frac{1}{256}$	$\frac{1}{1028}$	Vs

(b) $eval(v)$ value

Figure 5: A 4-direction grid world example on $eval(v)$ calculation

In Fig. 5(b), arrows indicate the vertex pointed to by $next(v)$. Then $next(v)$ always goes to the successor with the maximum $eval$. That is, $next(v_s)$ points to $v_{2,3}$, because it has a larger $eval$ (equal to $\frac{1}{128}$) than $v_{3,2}$. It can be seen in Fig. 5(b) that the $eval$ value has the capability to reflect directly how crowded a shortest path is.

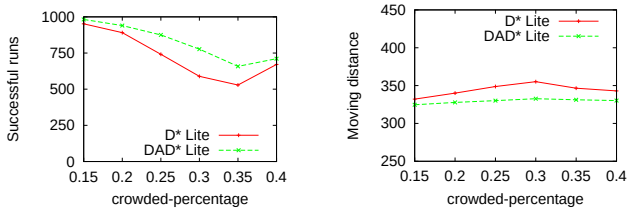
Experiments

Experimental Setup

In this section, DAD* Lite is compared with D* Lite on random grid terrains. The grid world is a 4-direction square terrain with $size \times size$ vertices. All the experiments are carried out on the terrain with $size = 200$. $v_g = v_{(20,20)}$ and $v_s = v_{(180,180)}$. The *sensor-radius* is the observable distance from the agent's current vertex. In every experiment, each vertex in the spacious area is randomly selected as blocked with some probability *spacious-percentage*, which higher for vertices in crowded areas, described by *crowded-percentage*. To simulate the crowded areas, a number of squares, with a random side length from 30 to 50, are injected in the terrains. The total crowded areas sum up to 30% of the whole terrain area; different crowded areas overlap. Before every navigation, the agent has an old map, in which every obstacle is considered wrongly to be at its neighbor position with probability 0.5. As the agent moves, in each step, all the obstacles in the terrain have probability 0.5 to move to their neighbors excluding v_s and v_g.

Results and Analysis

In Fig. 6, the performance of D* Lite and DAD* Lite is compared along two aspects – *successful runs* and *moving distance* – with different *crowded-percentage*. Experiments of D* Lite and DAD* Lite, respectively, run 1000 times. In each experiment, navigation stops when the algorithm cannot find a single path from v_s to v_g and thus v_g can never be reached. *successful runs* represents the number of experiments in which the agent reaches v_g in the end. Among all the successful runs, *moving distance* measures the average distance that the agent moves from v_s to v_g. Compared to the other D* Lite family algorithms experiments, where the emphasis is mostly on running time and heap operation, *successful runs* and *moving distance* reflect a more global point of view.



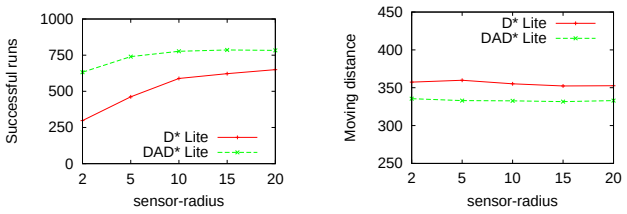
(a) (b)

Figure 6: $size = 200$, $sensor - radius = 10$, $spacious - percentage = 0.1$

In Fig. 6(a), DAD* Lite shows better results in *successful runs* than D* Lite. This benefit reaches the maximum at *crowded-percentage* $= 0.3$. When *crowded-percentage* is small, the terrain is actually rather spacious, with small crowded areas, so the advantage of DAD* Lite is not obvious. As *crowded-percentage* increases, the crowded areas have more obstacles. If a shortest path exists, D* Lite picks up one arbitrarily. If this shortest path goes through crowded areas, the probability that the agent is trapped in one of these increases. Therefore, DAD* Lite outperforms D* Lite in this situation. As *crowded-percentage* increases, fewer and fewer paths can pass through the crowded areas, so the shortest paths through the crowded areas account for a small proportion among all the shortest paths and therefore, it becomes easy for D* Lite to select a path avoiding crowded areas. This explains why DAD* Lite gains less in *successful runs* when *crowded-percentage* $= 0.4$.

In Fig. 6(b), DAD* Lite has shorter *moving distance* than D* Lite. Because the benchmark used is random and the terrain block density is still small, if the planned shortest path is interrupted, it is easier for the algorithm to find an alternative shortest path with the same distance as the old one, and hence, for both DAD* Lite and D* Lite, the *moving distance* is not big. However, it should be noticed that since

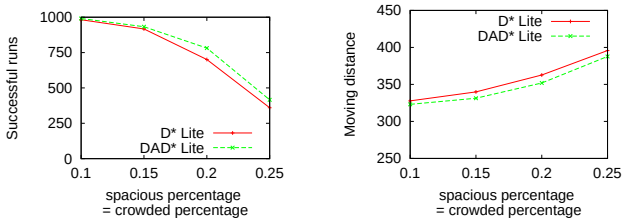
$v_s = v_{(180,180)}$ and $v_g = v_{(20,20)}$, the cost of the possible shortest path is 320. In Fig. 6(b) the values for *moving distance* produced by DAD* Lite are all closed to 320, which shows that the benefit of DAD* Lite with respcet to the *moving distance* is relatively stable.



(a) (b)

Figure 7: $size = 200$, *spacious-percentage* $= 0.1$, *crowded-percentage* $= 0.3$

In Fig. 7, two algorithms are compared with different *sensor-radius*. Fig. 7(a) shows that the number of *successful runs* decreases with *sensor-radius* for both DAD* Lite and D*. However, compared to D* Lite, DAD* Lite is less affected by the change of *sensor-radius*. More importantly, it can be seen that DAD* Lite with *sensor-radius* $= 2$ shows the similar result to D* Lite with *sensor-radius* $= 20$. In other words, in the same environment, the result of using DAD* Lite is equal to that of using D* Lite with a $10\times$ *sensor-radius*. Fig. 7(b) shows the improvement of DAD* Lite with respect to the *moving distance*.

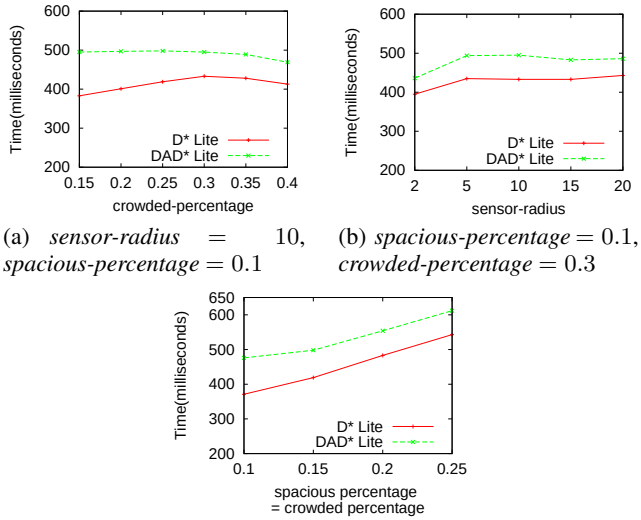


(a) (b)

Figure 8: $size = 200$, *sensor-radius* $= 10$

Fig. 8 depicts the results of two algorithms in the terrains when *spacious percentage* $=$ *crowded percentage*. In this case, all blocks are randomly chosen and evenly distributed among the terrain. DAD* Lite performs better than D* Lite with respect to both *successful runs* and *moving distance*. The advantage is small but stable. This result shows that even for a block-evenly-distributed terrain, there may exist some random crowded areas, of which DAD* Lite can take advantage to a certain extent.

Fig. 9 shows the running time comparisons of the above three experiment configurations, responding to Fig. 6, Fig. 7 and Fig. 8 respectively. Running time is the average time of all the successful runs in each experiment configuration. Because DAD* Lite inserts the procedure *GetDAPath*() and it basically has the same structure as D* Lite, potentially its running time is longer. *GetDAPath*() has to traverse all the vertices on the shortest paths to calculate evaluation values.



(a) *sensor-radius* $= 10$, *spacious-percentage* $= 0.1$ (b) *spacious-percentage* $= 0.1$, *crowded-percentage* $= 0.3$

(c) *sensor-radius* $= 10$

Figure 9: Running time comparisons

Similarly, ID* Lite (Yue *et al.* 2011) also calls a subroutine, *get-alternative*(), to generate the shortest path. Because ID* Lite reduces the number of vertices to expand, it consumes much less time than D* Lite in total. Another big difference is that the subroutine *get-alternative*() in ID* Lite only expands one shortest path, and thus does not have to traverse all the vertices on all the shortest paths as DAD* Lite does.

In Fig. 9(a), the difference between two algorithms reaches the minimum point at *crowded-percentage* $= 0.3$, because DAD* Lite saves more *moving distance* than D* Lite as shown in Fig. 6(b). In Fig. 9(c), the biggest difference between the two algorithms comes when *block percentage* $= 0.1$. That is, when the terrain becomes more spacious, there are more shortest paths, therefore *GetDAPath*() needs to expand more vertices and thus, it consumes more time.

Conclusion and Future Work

This study proposed DAD* Lite, the *Density-Aware* D* Lite algorithm. Compared to D* Lite selecting a shortest path arbitrarily, DAD* Lite finds a shortest path which may have more detours than other shortest paths and avoid the crowded areas. From a global point of view, by using a heuristic to quantify how crowded an area is, DAD* Lite produces better results than D* Lite. This improvement is reflected experimentally in the values of *successful runs* and *moving distance*.

From the experimental results, DAD* Lite consumes more time than D* Lite, which is determined by the structure of the algorithm. Future work will set a threshold on running time of *GetDAPath*() or a threshold on the number of the shortest paths to search in *GetDAPath*() to prune the search tree. Furthermore, the option of embedding DAD* Lite heuristic with ID* Lite, in order to shorten computation time, will also be explored. However, for a system that does not have a strict time limit, and when crowded areas are an important issue (e.g., the agent may be damaged if traveling

in such areas), DAD* Lite will always have a higher chance to reach v_g successfully on the shorter path (i.e., moving less distance).

As already discussed, D* Lite has its inherent shortcomings because the local and temporary nature of the information it owns. Without global information, it is impossible to find a globally optimal solution. The DAD* Lite algorithm presented here is a good start in the direction of obtaining a globally optimal solution. Improving the candidate pool of the paths and development of better heuristics will likely lead solutions close to global optimality.

References

Hansen, E., and Zhou, R. 2007. Anytime heuristic search. *Journal of Artificial Intelligence Research* 28(1):267–297.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on* 4(2):100–107.

Koenig, S., and Likhachev, M. 2001. Incremental a*. *Advances in neural information processing systems* 14:1539–1546.

Koenig, S., and Likhachev, M. 2002. D*lite. In *Eighteenth national conference on Artificial intelligence 2002*, 476–483.

Koenig, S., and Likhachev, M. 2005. Fast replanning for navigation in unknown terrain. *Robotics, IEEE Transactions on* 21(3):354–363.

Koenig, S.; Likhachev, M.; and Furcy, D. 2004. Lifelong planning a*. *Artificial Intelligence* 155(1):93–146.

Likhachev, M.; Ferguson, D.; Gordon, G.; Stentz, A.; and Thrun, S. 2005. Anytime dynamic a*: An anytime, replanning algorithm. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 262–271.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. Ara*: anytime a* with provable bounds on sub-optimality. *Advances in Neural Information Processing Systems*.

Mills-Tettey, G.; Stentz, A.; and Dias, M. 2006. Ddˆ* lite: Efficient incremental search with state dominance. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21(2), 1032. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

Nasa. 2012. Spirit mars rover in 'mcmurdo' panorama. `http://marsrovers.jpl.nasa.gov/gallery/press/spirit/20121109a.html`.

Stentz, A. 1995. The focussed d* algorithm for real-time replanning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1652–1659.

Stentz, A. 1997. Optimal and efficient path planning for partially-known environments. *The Kluwer International Series in Engineering and Computer Science* 388:203–220.

Yue, W., and Franco, J. 2009. Avoiding unnecessary calculations in robot navigation. In *Proceedings of World Congress on Engineering and Computer Science*, 718–723.

Yue, W., and Franco, J. 2010. A new way to reduce computing in navigation algorithm. *Journal of Engineering Letters* 18(4):EL_18_4_03.

Yue, W.; Franco, J.; Cao, W.; and Yue, H. 2011. Id* lite: improved d* lite algorithm. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, 1364–1369. ACM.

Yue, W.; Franco, J.; Cao, W.; and Han, Q. 2012. A new anytime dynamic navigation algorithm. In *Proceedings of the World Congress on Engineering and Computer Science 2012, WCECS 2012*, volume 1, 17–22.