Dynamic Route Re-planning

1. PROBLEM

Search algorithms can be used to determine the optimal path from one location to another. These algorithms are especially useful when applied to automobile route planning. However, automobile route planning can be significantly influenced by unknown and dynamic information about the environment. The cost of taking a given road may change based on the time of day, the weather, or construction work. Drivers may deviate from the current route for a variety of other reasons. Route planning algorithms running on embedded devices in automobiles must react to changes in the environment in real time so that drivers can continue to their destinations. Traditional informed search algorithms do not provide an efficient way of recalculating routes based on changes in the environment because these algorithms were designed to operate on static information.

2. RELATED WORK

A* is an informed search algorithm widely used for path-finding [Russell and Norvig 2009]. However, A* is not optimized for path finding cases in dynamic environments. A* can be adapted to dynamic environments by restarting once a change in the environment is encountered, however the algorithm will then perform more work than may be necessary. Lifelong Planning A* (LPA*) is an improvement to A* that efficiently updates only the changed paths in a dynamic environment [Koenig et al. 2005]. D* Lite is heavily based on LPA* [Koenig and Likhachev 2002]. Other related work in this area includes many A* variants such as Anytime Repairing A* (ARA*) [Likhachev et al. 2007], Fringe Saving A* (FSA*) [Sun and Koenig 2007], and Generalized Adaptive A* (GAA*) [X. Sun and Yeoh. 2008]. The class text book, Artificial Intelligence: A Modern Approach, also contains A* information [Russell and Norvig 2009].

3. APPROACH

This paper discusses the implementation of D*Lite, which is an algorithmically different improvement of D* that is based on the popular LPA* algorithm [Koenig and Likhachev 2002]. D* (Dynamic A*) is an informed searching algorithm that brings incremental search to A* [Stentz 1994]. When calculating the shortest path, D* assumes that unknown states in the environment are not obstructed [Stentz 1994]. Once an obstructed state is encountered, only the paths that are affected by the obstruction are recalculated and a new path may be selected.

The D* Lite algorithm is shown in Figure 1. This algorithm does not make assumptions about how the edge costs change, only whether they change in the world because the robot revised its initial estimates. D* Lite can be used to solve the problem of unexpected route changes due to road conditions or goal-directed navigation in unknown terrain.

```
procedure CalculateKey(s)
{01"} return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))];
procedure Initialize()
{02"} U = ∅;
{03"} k_m = 0;
{04"} for all s ∈ S rhs(s) = g(s) = ∞;
{05"} rhs(s_goal) = 0;
{06"} U.Insert(s_goal, [h(s_start, s_goal); 0]);
procedure UpdateVertex(u)
{07"} if(g(u) ≠ rhs(u) AND u ∈ U) U.Update(u, CalculateKey(u));
{08"} else if(g(u) ≠ rhs(u) AND u ∉ U) U.Insert(u, CalculateKey(u));
{09"} else if(g(u) = rhs(u) AND u ∈ U) U.Remove(u);
procedure ComputeShortestPath()
{10"} while (U.TopKey() < CalculateKey(s_start) OR rhs(s_start) > g(s_start))
{11"}   u = U.Top();
{12"}   k_old = U.TopKey();
{13"}   k_new = CalculateKey(u));
{14"}   if(k_old < k_new)
{15"}     U.Update(u, k_new);
{16"}   else if (g(u) > rhs(u))
{17"}     g(u) = rhs(u);
{18"}     U.Remove(u);
{19"}     for all s ∈ Pred(u)
{20"}       if (s ≠ s_goal) rhs(s) = min(rhs(s), c(s, u) + g(u));
{21"}       UpdateVertex(s);
{22"}   else
{23"}     g_old = g(u);
{24"}     g(u) = ∞;
{25"}     for all s ∈ Pred(u) ∪ {u}
{26"}       if (rhs(s) = c(s, u) + g_old)
{27"}         if (s ≠ s_goal) rhs(s) = min_{s'∈Succ(s)}(c(s, s') + g(s'));
{28"}       UpdateVertex(s);
procedure Main()
{29"} s_last = s_start;
{30"} Initialize();
{31"} ComputeShortestPath();
{32"} while (s_start ≠ s_goal)
{33"}   /* if (rhs(s_start) = ∞) then there is no known path */
{34"}   s_start = arg min_{s'∈Succ(s_start)}(c(s_start, s') + g(s'));
{35"}   Move to s_start;
{36"}   Scan graph for changed edge costs;
{37"}   if any edge costs changed
{38"}     k_m = k_m + h(s_last, s_start);
{39"}     s_last = s_start;
{40"}     for all directed edges (u, v) with changed edge costs
{41"}       c_old = c(u, v);
{42"}       Update the edge cost c(u, v);
{43"}       if (c_old > c(u, v))
{44"}         if (u ≠ s_goal) rhs(u) = min(rhs(u), c(u, v) + g(v));
{45"}       else if (rhs(u) = c_old + g(v))
{46"}         if (u ≠ s_goal) rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'));
{47"}       UpdateVertex(u);
{48"}     ComputeShortestPath();
```

Fig. 1. D*Lite Algorithm

D* Lite operates similarly to LPA* and is derived from LPA*, but the start and goal vertices are exchanged and all of the edges are reversed in the pseudo code [Koenig and Likhachev 2002]. D* Lite searches from the goal location to the start location while estimating the goal distances. LPA* searches from the start location to the goal location, estimating the start distances.

D* Lite performs a search by first executing the $ComputeShortestPathFunction()$. This function operates on a priority queue that initially contains only the goal node. Over time, the predecessors of each node in the priority queue are added until the algorithm reaches the start node. This function operates on two values, g and rhs. The g value represents the distance of a node to the goal. For example, $g(s)$ indicates the distance from the node, s, to the goal. rhs values are a one-step look that is based on the g values.

$$rhs(s) = min_{s' \epsilon Succ(s)}(c(s, s') + g(s')) \quad (1)$$

The rhs value of a node, s, with respect to its predecessor, s, is defined as the cost of moving from s to s plus the distance from s to the goal. Once the rhs values have been defined and are stable, the shortest path from the start node to the goal node can be found by moving from the start node towards the successor with the lowest rhs value until the goal node is reached.

Additionally, D* Lite employs tie-breaking criterion that have been simplified and priority maintenance techniques are used to remove many complex multi-line nested conditional statements found in regular D* [Koenig and Likhachev 2002]. These improvements make D*Lite substantially more simple than D* and improve the analysis of the program flow.

The implementation of A* in this paper initially comes from the example code in the AIMA [Russell and Norvig 2009]text book, where we have heavily modified and instrumented it to suite our tests. The pseudocode of D*Lite listed in Figure 1 was translated into python code and instrumented for the tests in this paper.

4. EVALUATION

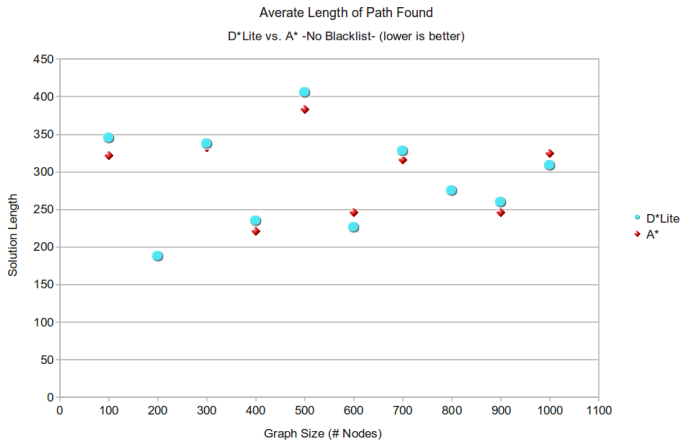

Fig. 2. Distance Traveled Comparision of A* vs D*Lite with no Blacklist

Our D* Lite testing is set in the context of route planning on roadways. In our tests, we compare D*Lite to A* in nodes traveled, distance traveled, and run time, while varying the graph size and the number of obstructed nodes (unexpected roadblocks).

The code for this project heavily modifies the python example code from the AIMA text book [Russell and Norvig 2009]. The A* implementation was instrumented to restart from the current node when encountering a path cost change. Our D*Lite algorithm was implemented by following the pseudocode listed in Figure 1.

To simulate road driving and route planning, a large map of nodes is randomly created with varying path costs. A blacklist of nodes is also randomly created. This blacklist is used to demarcate where obstructions will occur (via setting the node cost to infinity), but changes only when the search agent (car driver) is about to go along that path. A* and D* Lite are tested with the same graph and blacklist for each run.

A simulation of 100 runs while varying the graph size and blacklist is used to test the relative performance between D*Lite and A*. The A* algorithm requires a restart at all roadblocks while D* Lite incrementally recalculates the changing route.

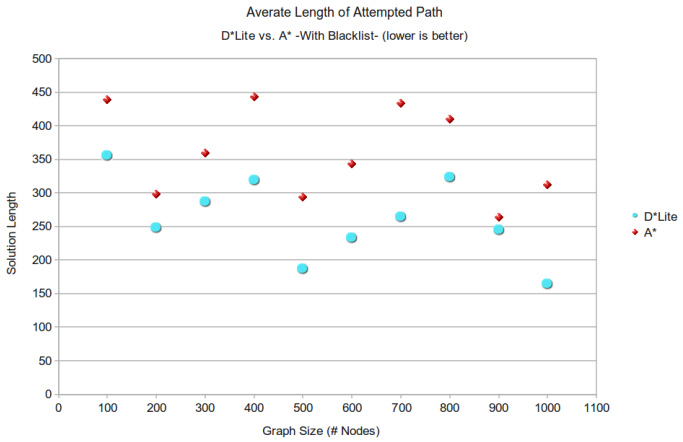

Fig. 3. Distance Traveled Comparison of A* vs D*Lite with Blacklist Nodes than change edge costs suddenly during planning

Figures 2 and 3 show the average solution length (total cost of the final path) compared to the number of nodes in the randomly computed graph. The X axis represents the graph size ranging from 100 to 1000 nodes. The Y axis shows the total cost of the solution.

As seen in Figure 3, the results from our research indicate that D* Lite, on average, finds a route with a lower cost than A* in terms of distance when obstructions are encountered. When there are no obstructions, D* Lite finds a slightly longer path than A*, as shown in Figure 2.

Figure 4 shows the calculation time (in ms) in comparison to the number of obstructions encountered. The X axis represents the number of obstructions that were encountered and the Y axis represents the calculation time.

The results in Figure 4 show that, in most cases, D* Lite takes a shorter amount of time to calculate its path than A*. In particular, when 1 to 4 obstructions are encountered, D* Lite is 2 to 3 times more efficient than A*.

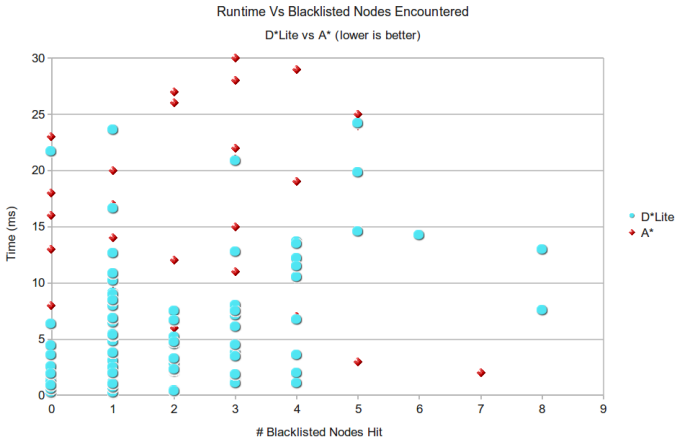

Fig. 4. Runtime Comparison of A* vs D*Lite with Blacklist Nodes than change edge costs suddenly during planning

5. DISCUSSION

5.1 Analysis

In this paper, we have presented D* Lite, a novel fast re-planning method for vehicle navigation in unknown or changing terrain which implements the same navigation strategies as Dynamic A* (D*). Both algorithms search from the goal location towards the current location of the vehicle, use heuristics to focus the search, and use similar ways to minimize reordering of the priority queue.

The results from our testing, as seen in Figure 3, indicate that D* Lite finds a shorter path than A* when obstructions are encountered. The A* algorithm will find the optimal path between two points in a static graph. With our implementation, obstructions are not encountered until after a path has been decided and the agent is about to move in to the node that is obstructed. From this point A* is run again with the start node now representing the current location. Because of the dynamic environment and frequent restarts, A* may not find the optimal path in this situation.

Our results also show that D* Lite is significantly faster that A* in terms of computation time when roadblocks are present. D* Lite selectively updates information about attributes that have changed in the environment. For example, the rhs and g values will only be recalculated on nodes that could possibly exist along a new path. In contrast, when A* is restarted, information about every node will be recalculated. Because of the selective recalculation, D* Lite performed faster than A* in our testing.

Finally, our results show that, in most cases, the less roadblocks that were encountered, the better A* performs with respect to time as compared to D*Lite. This is due to the fact that an efficient A* implementation was compared to a non-optimized implementation of D*Lite. With a more correct D* Lite implementation, the performance of A* and D* Lite on static environments is expected the be very similar.

5.2 Future Work

In our testing, we found that most of the obstructed nodes occurred near the beginning of the graph. The location of obstructed nodes can have a profound impact on the performance of search algorithms. Obstructions that are encountered near the beginning of a graph search will require more computation to correct than obstructions that are found near the end of the graph search. Obstructions near the start may change the current path significantly. In future research, we would like to ensure that obstructions are equally distributed throughout the graph.

To extend the research conducted for this project, in the future, we would like to implement other dynamic route planning algorithms such as Fringe Saving A* or Lifelong Planning A*. A comparison between LPA* and D* Lite would also be interesting because the two algorithms are very similar. D* Lite is based off of LPA*, but traverses the nodes in reverse order. These algorithms could be compared to determine the advantages and disadvantages of searching from the goal to the start.

D* Lite builds on LPA* which has its roots in A*. D* Lite is easy to understand and extend while being at least as efcient as D*. Our results provide a foundation for further research on path planning in dynamic environments. We believe that D* Lite would provide accurate directions in a short amount of time for a variety of navigation applications.

REFERENCES

KOENIG, S. AND LIKHACHEV, M. 2002. D* lite. *American Association for Articial Intelligence*.

KOENIG, S., LIKHACHEV, M., AND FURCY, D. 2005. Lifelong planning a*. *Elsevier Science*.

LIKHACHEV, M., FERGUSON, D., GORDON, G., STENTZ, A., AND THRUN, S. 2007. Anytime search in dynamic graphs. *Elsevier Science*.

RUSSELL, S. J. AND NORVIG, P. 2009. *Artificial Intelligence: A Modern Approach, Third Edition*. Pearson.

STENTZ, A. 1994. Optimal and efficient path planning for partially-known environments. *In Proceedings IEEE International Conference on Robotics and Automation*.

SUN, X. AND KOENIG, S. 2007. The fringe-saving a* search algorithm - a feasibility study. *In Proceedings of the International Joint Conference on Artificial Intelligence*.

X. SUN, S. K. AND YEOH., W. 2008. Generalized adaptive a*. *In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*.