A Travel-Time Optimizing Edge Weighting Scheme for Dynamic Re-Planning

Andrew Feit, Lenrik Toval, Raffi Hovagimian, Rachel Greenstadt

Deparment of Computer Science, Drexel Univsity
3141 Chestnut Street
Philadelphia, PA 19104
af359@drexel.edu, let37@drexel.edu, rch28@drexel.edu, greenie@cs.drexel.edu

Abstract

The success of autonomous vehicles has made path planning in real, physically grounded environments an increasingly important problem. In environments where speed matters and vehicles must maneuver around obstructions, such as autonomous car navigation in hostile environments, the speed with which real vehicles can traverse a path is often dependent on the sharpness of the corners on the path as well as the length of path edges. We present an algorithm that incorporates the use of the turn angle through path nodes as a limiting factor for vehicle speed. Vehicle speed is then used in a time-weighting calculation for each edge. This allows the path planning algorithm to choose potentially longer paths, with less turns in order to minimize path traversal time. Results simulated in the Breve environment show that travel time can be reduced over the solution obtained using the Anytime D* Algorithm by approximately 10% for a vehicle that is speed limited based on turn rate.

Introduction

The speed with which real vehicles can traverse a path is often dependent on the sharpness of the corners on the path as well as the path length. For environments where speed matters and vehicles must maneuver around obstructions, such as autonomous car navigation in hostile environments, UAV close range maneuvering, or high speed robot arm planning, we investigate an algorithm that incorporates turn angles into path planning and show that it produces faster travel times in simulation than algorithms that only consider edge traversal time.

The traditional approach to autonomous vehicle path planning uses algorithms based on A*, D*, or Anytime D*. Typically, the search algorithm is designed to find an optimal path, consisting of connected edges in a pre-determined graph of a space. The optimal solution path minimizes the cost incurred by travelling the chosen route. Costs can be assigned or calculated for each edge; either based on the length of the edge, an estimate of the time for the vehicle to travel that particular edge, or some combination that relates to fuel or energy usage.

Using pre-determined estimates for the time required to traverse an edge of a graph is sufficient for many applications. In cases where edges are long with respect to the size of the vehicle and to the total length of the path, the transitions between edges may not make up a large part of the traversal time and can be ignored. In many cases, robots following a path may have a limited range of speeds they can travel at, so that sharp corners do not hinder their progress significantly. However, in applications that require short edge lengths in order to maneuver around close obstructions, and involving vehicles with wide speed ranges, time optimization of a path can depend on the combination of edges chosen in addition to the sum of the time required to traverse the edges. The maximum turn rate of any type of high speed land or air vehicle is dependent on how fast it is going, and therefore traversal time is dependent on the turning angles formed by the combination of edges that make up a solution path. For example, a car is limited by either the lateral traction the tires can hold, or by the maximum lateral acceleration that will not cause the vehicle to roll. In either case, if the path consists of short, straight edges, the angles between the edges will play a significant role in determining the speed at which they can be travelled.

This paper presents an edge-weighting scheme that takes into account the angles between edges. We incorporate this weighting scheme into the Anytime D* algorithm and show that it produces improvements in travel time when studied in simulation, particularly in highly obstructed environments.

Background and Related Work

There have been many efforts to make path planning algorithms more practical for real vehicle use.

D* [Stentz, A. 1994] improves upon A* by searching backwards from the goal to the starting point. This allows for dynamic re-planning – because the path is searched backwards, changes to graph weights require much less re-planning and many of the existing back-pointers can still be used. Since 1994, others have simplified D* [Koenig, S., and Likhachev, M. 2002] and made additional improvements. Anytime D* [Likhachev, M., et al. 2005] recognizes that a real vehicle may have limited time to plan a path, so it computes sub-optimal paths first, and improves the path as the vehicle travels. This may allow a vehicle to continue moving when it would otherwise have to wait for the optimal route to be calculated.

Subsequent improvements have been to incorporate moving objects into the map. In [Van Den Berg, J., et al. 2006], Anytime D* is implemented with a time assigned to each map position. This allows the search to avoid moving obstacles, and to find a path that gets to the destination in the shortest time, with the assumption that the vehicle can travel at a roughly constant speed.

Several integrated path planning systems have been implemented that make use of either D* or Anytime D* as part of a system for finding a real steering solution for a vehicle [Stentz, A., and Herbert, M. 1994] [Podsedkowski, L., et al. 2001] [Seder, M. et al. 2005]. One approach is to use D* to find the general path through an area, then use a separate module to compute a smooth path through it. The Anytime D* algorithm was used in a vehicle that successfully completed the DARPA Offroad Challenge [Strum, S., et al. 2006]. Their method took into account turn rates when computing its graph by verifying that each potential successor edge was within the capabilities of the vehicle. The maximum safe speed was then calculated based on the resulting path curvature and other parameters. This method keeps the solution path near the given road, and ensures that the vehicle is capable of traversing it within its physical capabilities. It does not optimize the path curvature for most efficient travel, or take into account the combination of edges chosen within the path-planning algorithm itself. Our contribution is unique in that the path curvature is used as the weight, so that the path curvature can be optimized within the graph search algorithm.

Our algorithm uses concepts from all of these D*-based path planners in an attempt to find faster paths by taking into account the speed at which a real-world vehicle can traverse the path, recognizing that a longer path may reduce travel time in some instances. Our approach is most closely modeled after the Anytime D* implementation in dynamic environments [Van Den Berg, J., et al. 2006]. However, we have not incorporated Anytime processing or re-planning into our evaluation, since those components are not required to show the improvements we are making.

Approach

Algorithm

The goal is to base the cost of a path on both the costs of the individual edges, as well as the costs of combinations of adjacent edges along the path. In the model used here, the maximum speed at which a vehicle can traverse any node in a path will depend on the turning angle formed by the two edges joined by the node. Vehicle speed varies between 0.1 and 1.0. The vehicle can travel at 1.0 if the turning angle is zero, and the maximum speed linearly decreases toward zero for turning angles from 0 to 90 degrees. The maximum allowed speed for a node has a minimum value of 0.1 for turn angles close to 90 degrees. It will be assumed that as the vehicle travels between two nodes, the vehicle speed varies linearly between the maximum speeds at each node.

In this way, the time the vehicle takes to travel on a given edge will depend on the maximum speeds allowed at the edge end nodes, and also on the angles formed by edges at the nodes. This requires a modification in the way edge costs are calculated in the Anytime D* algorithm. Normally, the cost of an edge depends on the positions of the two nodes at either end of that edge. In the method presented here, the travel time of a given predecessor edge also depends on the angle formed with the next edge after that, which will not yet be known at the time a predecessor edge is chosen.

The solution resulting from the D* path planning algorithm consists of a back-pointer assigned to each node that determines the next node that should be travelled on the solution path. In other words, each position on the map is encoded with information about where the vehicle should go next. As a result, each node does not know where the path came from to get to that point, because the algorithm actually worked in the opposite direction - goal toward start - to arrive at the solution. This means that each node does not know anything about the combination of edges that it joins, and cannot determine the turning angle through itself. Each node can, however, know the relationship between two other edges; the edge from itself to its back-pointer, and the edge from its back-pointer to its back-pointer's back-pointer. In Figure 1, point A is closest to the goal, and point C is closest to the start. The speed at point B is a function of the turning angle ABC. When the cost from A to B is being calculated in the search algorithm, the position of C, and therefore the angle ABC, is not yet known.

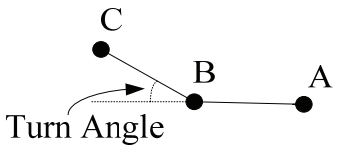

Figure 1- Turning angle on path from C to A.

When the search algorithm reaches point B, it obtains a list of predecessors to be evaluated next, one of which is C. The difficulty is that the turning angle through a node B depends on the specific predecessor C that the vehicle is traveling through on its way to B, which is not known until the entire path search is complete. To overcome this, the relationship (in this case, angle) between edges at B is stored in node C. Edge cost due to travel time between B and A can then be added to the edge cost of CB. This means that the travel time cost for one edge is getting stored with the previous edge. This method is valid because the algorithms decision of which predecessor to choose determines what this cost will be, regardless of where the cost is actually encountered while travelling the course. The path planner still has an opportunity to choose from each combination of edges based on the total cost of that combination. Figure 2 shows how the 'nextspeed' of each node really contains the maximum speed of the back-pointer of that node, i.e. nextspeed(s) is the maximum speed at *s'* if the path goes through node *s*. The time to travel between nodes *s'* and *s''* is different for each node s that could be chosen. Each node s computes this travel time and adds it to its own edge cost from *s* to *s'*.

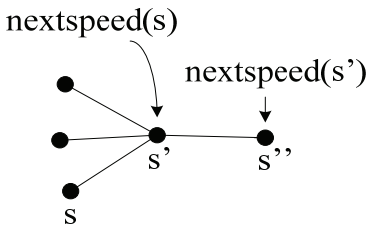

Figure 2- Maximum speed at s' is determined by s.

The Anytime Dynamic planning algorithm computes the cost of an edge with (1).

$$c(s,s') = w_t * (t' - t) + w_c * c_r(s,s') \qquad (1)$$

Where w_t and w_c are weights, and c_r is the cost of the edge from points *s* to *s'*, which are reached at times *t* and *t'* respectively. The predecessor function in D* computes a list of possible points that could have been visited before the current node as the vehicle travels. For each predecessor, our modified algorithm will compute the additional turn-angle based time-weighting that would be incurred if that predecessor ends up being travelled. During the search, the cost that the turning angle incurs on travel time for each predecessor will be added to the cost of that edge when determining the total cost as in (2).

$$c(s,s') = w_t * t_r(s',s'' \mid s) + w_c * c_r(s,s') \qquad (2)$$

$$t_r(s',s'' \mid s) = \frac{2 * dist}{v_{final} + v_{init}} = \frac{2 * |s'' - s'|}{v_{next}(s) + v_{next}(s')} \qquad (3)$$

Equation (3) represents the edge travel time between *s'* and *s''* given that predecessor *s* is chosen, which is then included in the cost calculated for edge *s* to *s'*. The travel time is based on a linearly varying speed between vinit and vfinal. When the path is actually being travelled, the vehicle will look at the node it is currently passing to get the maximum allowable speed for the next node, and it will adjust its speed accordingly.

Modification to D*

In addition to the modified edge cost function, the predecessor function used in this simulation is based on the 'velocity vector' of a node, which is the same as the back-pointer of a node, and determines the direction that the car will be travelling when it arrives at a node. The predecessor set consists of a node continuing in the same direction, and nodes that would be reached by turning at a selection of cardinal angles away from the velocity vector.

As a result of the structure of the predecessor function, this method does not use a pre-determined set of grid points, which has advantages and drawbacks. It allows for more natural curved paths that give a little more flexibility in navigating an area, but when the algorithm runs into an obstruction, backtracking is difficult because it generates a large number of nodes with slightly different positions in a very small area, and this increases the processing time required to find a path. When each predecessor location is computed, the algorithm searches for any existing node instances within a small radius so that predecessors are 'snapped' to existing predecessors in that area. This is implemented in the simulation with a hash function that links node coordinates with the actual node data structure. Snapping the nodes also assures that only one node object is created for each map position. In addition to path cost, the AD* algorithm for dynamic environments [Van Den Berg, J., et al. 2006] uses a heuristic based on time and distance to the start. In this implementation, time values are not assigned to each node, so the heuristic is based entirely on distance to the start.

Implementation

To evaluate the performance of the modified cost function against the normal D* implementation, a simulation was created using the Breve environment [www.spiderland.org]. Breve is a fairly powerful 3-d simulation tool that can model interactions between physical objects using a programming language similar to Python. Several objects were set up to model parts of the evaluation environment:

Ground. The area for the course is a flat, 20 x 20 grid. A "start" marker is setup at {0,0} and the "goal" marker is set up at {20,20}, making the objective to cross diagonally across the square course.

Vehicle. The vehicle that traverses the path is displayed as a rectangular box. Vehicle motion is constrained to remain on the ground plane, and the velocity of the car is constrained to be in the direction it is pointed, similar to many wheeled vehicles. The vehicle contains a controller that steers it toward the next node on the solution path, and automatically sequences to the back-pointer of the node as it reaches each one. It also adjusts its speed linearly between the maximum speeds for the nodes before and after its location on the path. This matches the assumption used in the planning algorithm for calculating the travel time for path edges. The vehicle object calls the functions required to perform the D* search to obtain the starting node position and the subsequent chain of nodes that forms the solution path.

Obstructions. All obstructions are represented as blocks of uniform height with random width and length, sitting on the ground plane. They are positioned randomly within the square bounded by {5,5}, {5,15}, {15,5}, and {15,15} to prevent them from being too close to the start or goal point. The positions of the obstructions are also represented in a 200x200 matrix of the map area (10 matrix locations per grid 'unit'). The path planner uses this matrix to determine if a path section is passable or not due to an obstruction.

Path Planner. The path planner class contains all of the state data of the path search algorithm, along with the search functions. It is initiated by a function call which returns a start node that is linked via back-pointers to a chain of nodes leading to the goal.

Path State. Each path node is modeled as an instance of the path state class. This class contains data such as the back-pointer of the node, the various values used in the D* algorithm such as g(s) and rhs(s), as well as the methods to get the predecessor and successor lists of a node.

Evaluation

The performance of travel-time weighting will be evaluated using three metrics: travel time, path distance, and average speed. Travel time is calculated by simply subtracting the initial simulation time from the time at which the car reaches the goal. The path distance is obtained by summing the path edge lengths along the route. Average speed is calculated from path distance and travel time as an additional way to visualize the results.

Cases were run on maps with varying numbers of obstructions, and using various weighting factors. The weight variables wc and wt determine the weighting of edge length and edge travel time when searching the graph. This is similar to the AD* algorithm [Van Den Berg, J. 2006], where increasing the wt value causes the search to rely more on the travel time of the vehicle across nodes based on the turning angle calculation. If wc is set to 1.0 and wt is set to 0.0, then the path will be based entirely upon path distance, as in D*. If wt is set to 1.0 and wc is set to 0.0, then the path will attempt to optimize travel time, but may use a slightly longer path. A combination may be used if a balance between the two methods is desired, such as if fuel consumption is a concern.

The number of obstructions on the map was varied between 5 and 11 to see the relationship between map complexity and the benefit obtained from time-weighting. Maps with too few obstructions sometimes allowed a straight path to the goal, while maps with too many obstructions sometimes formed a single large obstruction in the middle. Figure 3 and Figure 4 show examples of two maps, which have 5 and 11 obstructions respectively. The smaller grey box in each image is the vehicle, and the branched lines represent back-pointers that form the solution path.

The full anytime function with increasing levels of optimization was not implemented here, but the path planner can be set for a constant optimization factor. The optimal factor (eta) of 1.0 often resulted in prohibitively large path processing times in our implementation. Maps were tested with a range optimization factors in order to understand the impact of this on the expected results of using time-weighting. In a real anytime implementation, the optimal path would not be immediately calculated, so testing with a constant inflated weighting factor is not unrealistic. Figure 5 shows the average speeds for paths through one map vs. time-weighting, for eta values of 1.5, 1.2, and 1.1. This shows that the benefit from out method grows as the path becomes closer to optimal.

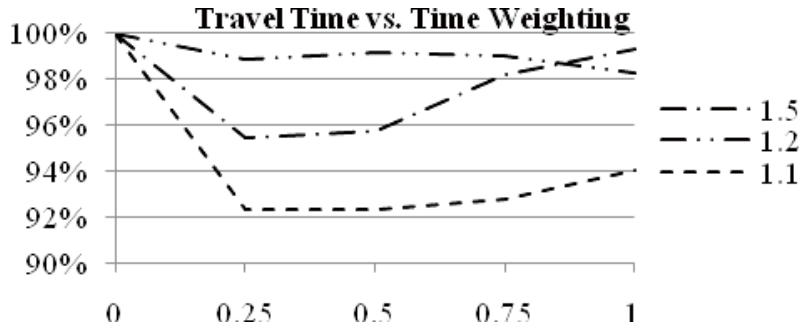

Figure 5 – Travel time reduction vs. optimality.

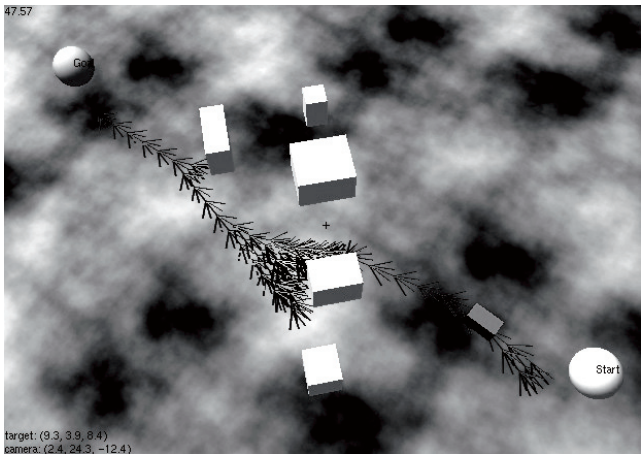

Figure 3 - 5 obstructions.

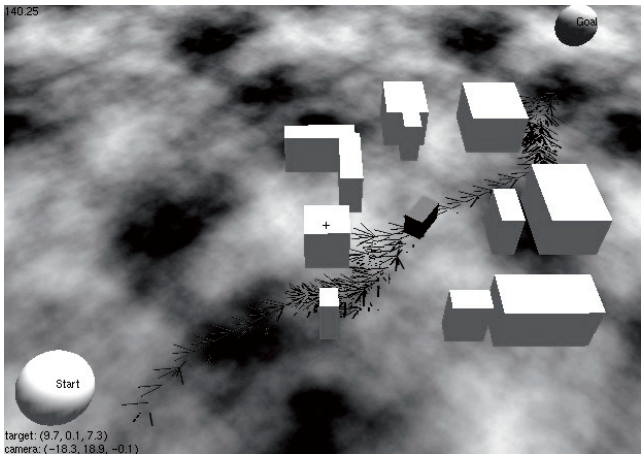

Figure 4 - 11 obstructions.

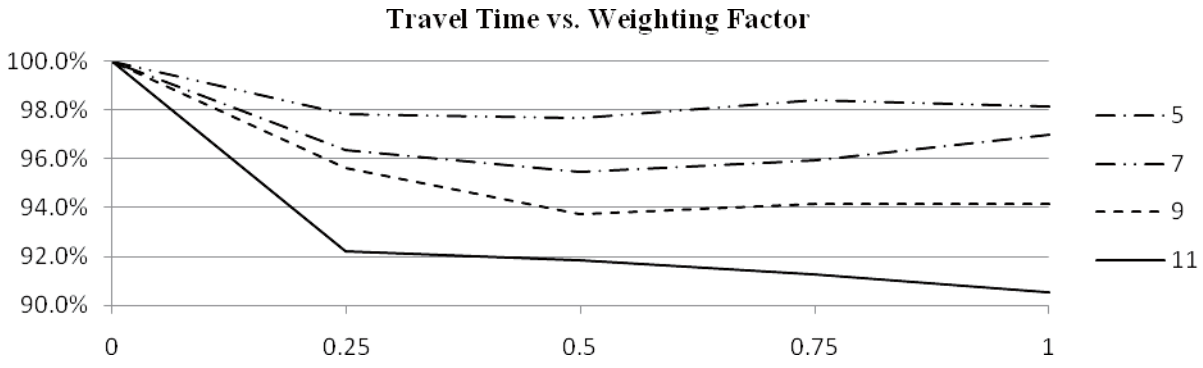

Figure 6 – Travel time at 1.15 optimization factor.

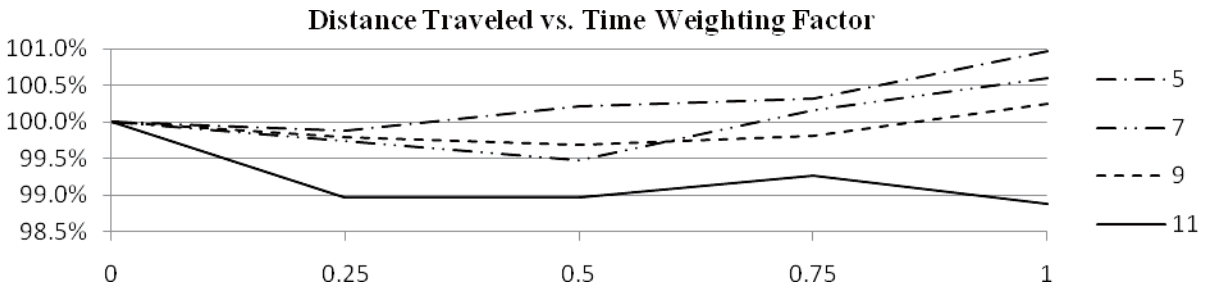

Figure 7 – Distance Travelled at 1.15 optimization factor.

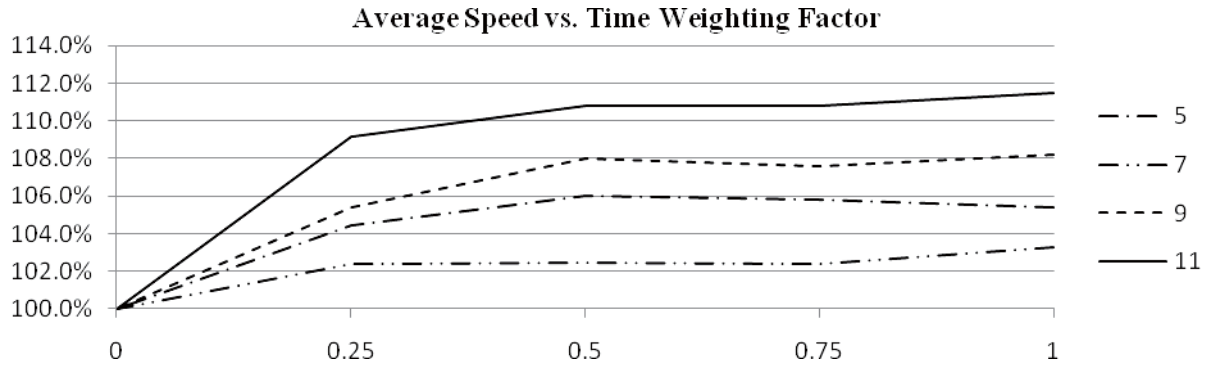

Figure 8 – Average speed at 1.15 optimization factor.

Figures 6-8 above show travel time, path distance, and average vehicle speed while moving from start to goal on the solution path vs the travel-time weighting factor. All values are normalized, so that 100% corresponds to the time, distance, and speed of the path resulting from edge costs based only on length. Each plot shows the results for twelve cases – each line is the average of three maps with the same number of obstructions. This was repeated for maps with 5, 7, 9, and 11 obstructions. As expected, improvements are greater for maps with more obstructions. An optimization factor of 1.15 was used for all cases.

As expected, there is a trend of increasing average speed as the time weighting is increased. Higher values of wt result in less sharp corners in the path, so the vehicle does not slow down as often. The trend in path travel time is also as expected, given the trend in average vehicle speed. For optimal search solutions, we would expect that the shortest path would be found when wt = 0, and that for higher values of wt, the path distance would increase. In Figure 7, some increase is seen, although the pattern is not consistent for 11 obstructions. A few factors could be affecting this; primarily, these are not optimal path solutions (eta > 1). Also, the effect is very dependent on the specific obstacle configuration. Certain configurations will have greater tradeoffs between speed and time.

Figure 10 shows the velocity profile of a map for two different wt values, at a constant eta of 1.1. This gives an idea of how the speed of the vehicle varies as it traverses the path, and where the increased value of wt is able to improve travel time. In the cases where wt=0, the speed drops down to around 0.5 in a few places. For for wt = 1.0, the speed stays 1.0 for a large portion of the time and drops to a lower speed less often.

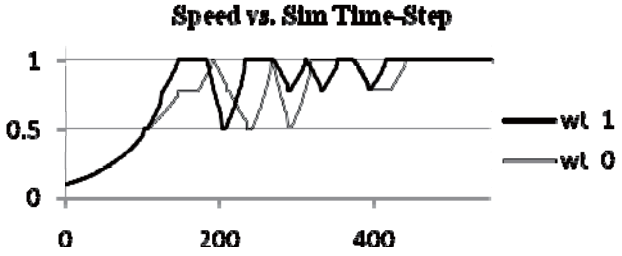

Figure 10 – Speed profile for different weights.

For the cases shown here, the maximum improvement in path travel time is close to 10%, and the trends in Figures 6-8 suggest that further improvements can be realized in

more densely obstrustructed environments. This is significant given that the processing and memory requirements of the algorithm are not considerably increased. This algorithm requires some additional processing when computing the predecessor function for each node, however this is still within a scale factor of processing and memory requirements for the original D* implementation.

Conclusions

This paper has demonstrated a feasible way of computing edge costs in a D* graph planning algorithm based on relationships between edges. This allows path curvature to be taken into account even when the predecessor function uses only straight edges between nodes. The method demonstrated has shown to consistently produce shorter-time paths around obstructions than when only edge length is considered, with a minimal amount of additional processing overhead required.

A key part of this method is to store the maximum traversal speed for a given node in the predecessor of that node. Information is stored in the node on which the information depends rather than the node where it will be applied. This highlights the way that a graph search can evaluate edge costs based on the decisions that effect that cost, which may be applied at a different time from when that cost is actually incurred as the path is being travelled. Several previous papers cover the tradeoff between path distance and speed in search for a path, however many use a traditional start to goal method, and don't apply this to the anytime D* algorithm as we do here.

Many future applications of this modification are available for research. It would be interesting to do a direct comparison between this method and an alternate approach of using curved edges (and therefore zero angles between edges). In addition, there may be cases where the cost of a given series of edges depends on more than two edges on that path. Furthermore, it would be interesting to put together a complete Anytime, Dynamic path planning implementation that deals with moving obstructions using the method presented here to see the effect in conjunction with the physical turning speed limitations of a real vehicle.

References

Van Den Berg, J., Ferguson, D., and Kuffner, J. 2006. Anytime Path Planning and Replanning in Dynamic Environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA*)

Klein, J. 2009. Breve: A 3d Simulation Environment for Multi-Agent Simulations and Artificial Life (Breve 2.7.2). http://www.spiderland.org (jk@spiderland.org)

Koenig, S., and Likhachev, M. 2002. Improved fast replanning for robot navigation in unknown terrain. In *Proceeding of the IEEE International Conference on Robotics and Automation (ICRA)*

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)*

Podsedkowski, L., Nowakowski, J., Idzikowski, M., and Vizvary, I. 2001. A new solution for path planning in partially known or unknown environment for nonholonomic mobile robots. *Robotics and Autonomous Systems* 34(2):145-152

Seder, M. et al. 2005. An integrated approach to real-time mobile robot control in partially known indoor environments. In *Proceeding of the 31st Annual Conference of the IEEE Industrial Electronics Society (IECON)*

Stentz, A. 1994. Optimal and Efficient Path Planning for Partially-Known Environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*

Stentz, A., and Herbert, M. 1994. A Complete Navigation System for Goal Acquisition in Unknown Environments. *Carnegie-Mellon Technical Report CMU-RI-TR-94-7*.

Sud, A., Anderson, E., Curtis, S., Lin, M., and Manocha, D. 2007. Real-time Path Planning for Virtual Agents in Dynamic Environments. In *International Conference on Computer Graphics and Interactive Techniques*

Thrun, S., et al. 2006, The Robot that Won the DARPA Grand Challenge, *Journal of Field Robotics* 23:661-692