Dan Lovell
6.834 Final Project

Purpose

The goal of my project was to develop a Roadmap Path Planner (RPP) to generate a cost map for a Kinodynamical Path Planner (KPP) to be used in receding horizon control. A grid-base all sources, single destination shortest path finding algorithm, D*-lite, was used to generate the cost map. The cost map was generated by repeatedly calling D*-lite on the map until the costs of all locations in the desired cost map were known. In an attempt to save computational effort, each call to D*-lite reused the priority queue of the previous search.

Receding Horizon Controllers

Autonomous vehicles require kinodynamical controllers to navigate through the world because they have finite control output and/or they are non-holonomic. Planning a feasible kinodynamically constrained path is computationally expseive. The aim of Receding horizon control is to reduce the amount of planning required before vehicle can begin to traverse its path while ensuring it won't collide with an object in the environment. The reduction in computation is achieved by planning over only a portion of the overall path while still ensuring that the resulting path will be feasible. To increase the probability that a the shorter path will permit a feasible overall path, a good estimate of the "Cost to Go" (CTG) is required for all locations in reachable over the shortened path to be planned. This CTG is an estimate of the actual cost of the best possible path froma location to the goal. Given unlimited computation time, the CTG of all locations in a given static environment could be determined. This might suggest that a vehicle could precompute and store the CTG over all locations it anticipates being in. However, vehicles must often plan on the fly because of uncertainty in the current environment, dynamic or unknown environments and the because of the prohibitive cost of storing the CTG for all possible robot poses. Therefore an estimate of the CTG is often used.[1]

Many factors can contribute to the actual CTG of a given location, including parameters of the environment and the vehicle. These factors may include the minimum traversable distnace to the goal, the curviness of the path to the goal, the terrain that must be traversed to get to the goal and the current state of the vehicle. For example, given similar environments and vehicles, a path that is twice as long as another is probably twice as costly to traverse, where cost is usually measured in time required to traverse the path. Similarly a very curvy path will likely be more costly than a very straight path of the same length because vehicles must often slow down going into turns. Also very steep or rough terrain is usually more difficult to traverse than very constant or flat terrain. Lastly, given a location next to a wall, the CTG at that location might be low if you're traveling parallel to the wall, while the CTG might be very high or near infinite if you are headed directly towards the wall (due to the control effort required to turn away from the wall or the fact that a collision might then be unavoidable).

I chose to represent the CTG of a location as directly proportional to the distance required to traverse to get to the goal. Several factors contributed to this decision: the typical environment of the vehicle (a fixed wing airplane) for which my roadmap path planner is planning usually flies in fairly open skies, and the radius over which the Roadmap Path Planner plans is on the order of a turning radius of the vehicle so that the cost map produced isn't likely to cause the vehicle to move into a position where collision

is unavoidable.

Given that the cost is directly proportional to the distance of the path, one of the many optimal path planners that minimize path distance can be used. Key factors that effect which algorithm is best for this task is efficiency in finding paths and reparability of the paths found in case unknown obstacles appear. Given unlimited computation time, the specific algorithm used would not important. However, efficient path finding is important because the paths will typically be found online and therefore must not be computationally prohibitive for the onboard computer. Also, the environment is often dynamic, in which case repair must occur online as well.

The reparability requirement suggests that an incremental all sources, single destination algorithm is preferred because obstacles that are present but unknown are usually discovered near the vehicle (as the vehicle approaches the obstacle, it comes into sensor range). An all sources, single destination algorithm only needs to replan the portion of the path between it and the newly discovered obstacle, which will often be shorter than the portion of the path between the obstacle and the goal.

For these reasons, D*-lite was chosen. D*-lite is an all sources, single destination algorithm that is based on LPA*. Like LPA* it is repairable. The algorithm is called D*-lite because, like D*, it is all sources, single destination but it performs better than D*, hence the "lite" suffix.

Background on D*-lite

In D*-lite the world is discretized into uniformly sized grid cells, sometimes called nodes or vertices. Each node, *s*, has associated with it an estimate of the start distance of each node, $g(s)$, a second estimate of the start distance of each node, $rhs(s)$, and an under-estimate of the distance to the goal, $h(s)$. rhs(s) is like a "one step look ahead" and is a way for nodes with changed g-values to notify its neighboring nodes. With an estimate of a node's start distance (the cost of the cheapest path to the node from the start) and an estimate of its goal distance (the cost of the cheapest path to the goal from the node) the total cost of a path from the start to the goal through a node can be estimated as the sum of these terms.

A priority queue holds all nodes that are inconsistent. An inconsistent node is one that has a smaller rhs-value than g-value representing the fact that a neighboring node has reduced its rhs-value but it has not checked to see if it has neighboring nodes that can give it a better rhs-value. Inconsistent nodes represent the edges of the currently explored map. The order of nodes in the priority queue depends on the node's key, k. k is represented as $[k_1,k_2]$ and $k^a < k^b$ if $k^a_1 < k^b_1$ with k_2 used as a tie breaker ($k^a_1 = k^b_1$). The rules for the tiebreaker can vary and have an effect on how the map is explored. The top node of the priority queue is the next node that will be explored and is the best estimate of the node that will lead to the best path to the start.

During the search process, the map is assumed to be static and the g-values can be guaranteed to be correct. This fact, together with the requirement that h-values be under-estimates (often referred to as an admissible heuristic) of the actual cost of traversing to the goal guarantees that the cost estimate $\min(g(s),rhs(s)) + h(s)$ is less than the actual cost. Therefore when the start node is the top node of the priority queue it is not possible that there is another node in the priority queue that would result in a more optimal path. Therefore the combination of the best first search with an admissible heuristic guarantees that the first path found is the optimal path.

Path repair must occur whenever an obstacle appears or disappears (sometimes

referred to as a change in edge cost). All neighboring nodes must reevaluate their rhs-values and be added to the priority queue if their rhs-value has changed so that this change can be propagated if necessary.[2]

The Effects of $h(s_1,s_2) \neq c(s_1,s_2)$ and the tie breaker rule

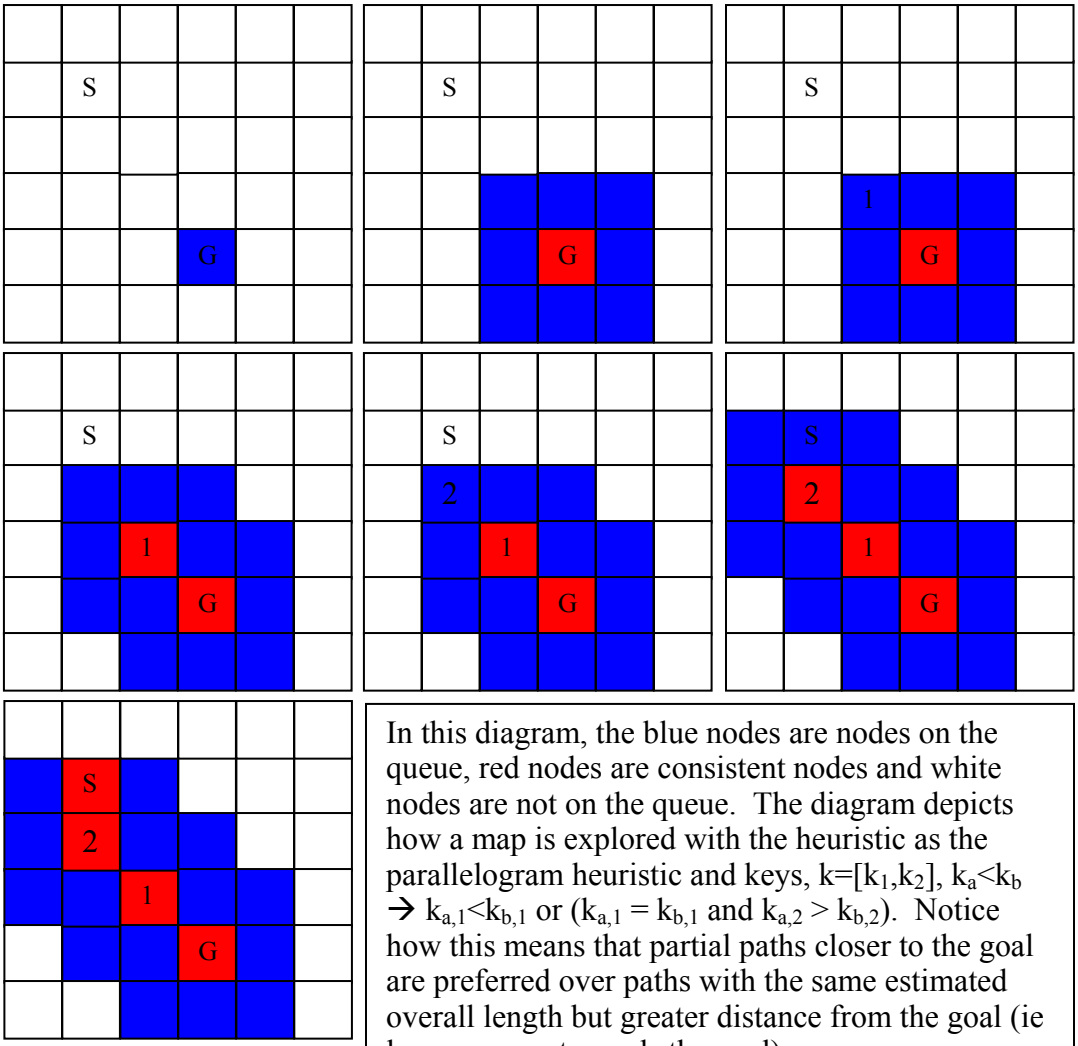
Most grids for shortest path algorithms are either 4-connected or 8-connected. 4-connected grids worlds are worlds where you can only move up, down, left or right and 8-connected grid worlds are worlds where you can move along the diagonals as well. Also the costs of movement can be either the Euclidean distance between the center of the nodes or a constant (where diagonal moves are as cheap as horizontal/vertical moves) or something else. For the autonomous aerial vehicle problem, the 8-connected grid world with cost of movement being the Euclidean distance is more suitable than any of the options since the vehicle can move diagonally between nodes and to go from the center of one square to the center of another the cost is the Euclidean distance.

A suitable heuristic must be chosen for this 8-connected grid world. The ideal heuristic would be exactly correct, though it only needs to be admissible. The Manhattan distance is not suitable since its not admissible. The Euclidean distance is admissible but its underestimate could cause more nodes to be explored than necessary. The best heuristic takes into account the fact that laterally traversing 1 node and horizontally traversing 2 nodes does not cost sqrt(3) to traverse but actually costs 1+sqrt(2) to traverse. The formula for a heuristic that takes into the account the shortest possible path in an unobstructed grid world between nodes (x_1,y_1) and (x_2,y_2) makes $\min(x_1-x_2,y_1-y_2)$ diagonal moves with cost sqrt(2) and $\text{abs}((x_1-x_2) – (y_1-y_2))$ lateral or horizontal moves with cost 1 for a total cost of $C = \text{sqrt}(2)* \min(x_1-x_2,y_1-y_2) + \text{abs}((x_1-x_2) – (y_1-y_2))$. I will refer to this as the parallelogram heuristic.[3]

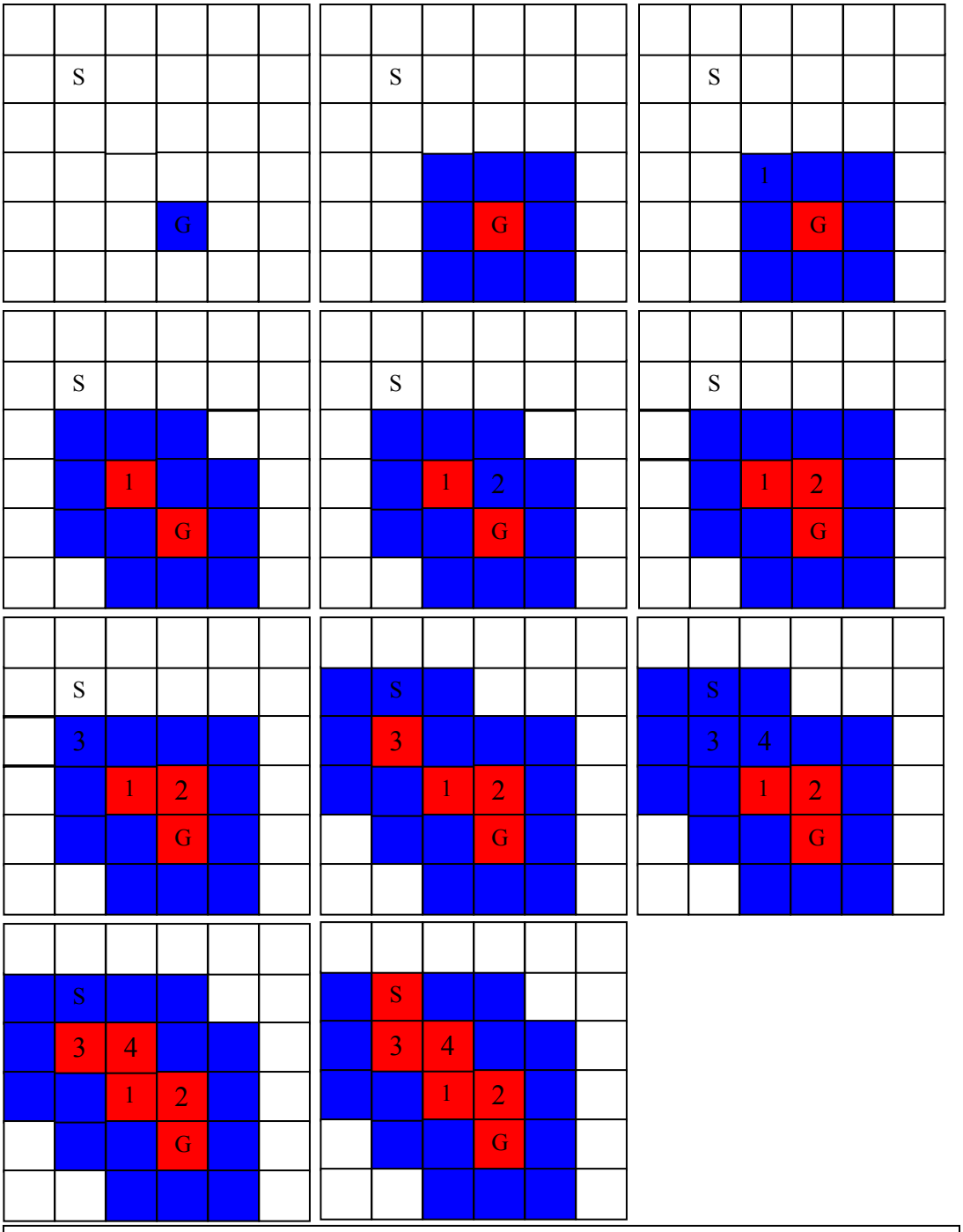
The tiebreaker, discussed previously, can affect how much of the map is explored. If the tiebreaker goes in favor of shorter paths then all possible paths that might reach the start with the same cost will be explored. With the heuristic decided on, this means that all nodes within a parallelogram defined by a diagonal with slope 1 and length $\min(x_1-x_2,y_1-y_2)$ and horizontal sides of length $\text{abs}((x_1-x_2) – (y_1-y_2))$ will be explored. Therefore the tiebreaker should be in favor of longer paths, which will be closer to the goal.

D*-lite-RPP concept

A RPP could be implemented using D*-lite simply by executing a new D*-lite search for each start location. However, this would throw away two sources of information you've already obtained: all consistent nodes will remain consistent and the ordering of the priority queue. That all consistent nodes remain consistent is true because being consistent means that the shortest path from the node to the goal is known. Changing the start location will not affect this path. Though the priority queue cannot be reused as is for the next search, it is valuable because the work to enqueue any node that would be required by subsequent searches is saved. Its possible that some of the subsequent searches needed to generate the cost map wont need some of the nodes in the priority queue and the effort spent reorganizing these nodes in the priority queue would not be useful. This is not likely, though, since the desired cost map is (usually) contiguous and subsequent searches will likely go through similar regions.



In this diagram, the blue nodes are nodes on the queue, red nodes are consistent nodes and white nodes are not on the queue. The diagram depicts how a map is explored with the heuristic as the parallelogram heuristic and keys, $k=[k_1,k_2]$, $k_a<k_b \rightarrow k_{a,1}<k_{b,1}$ or ($k_{a,1} = k_{b,1}$ and $k_{a,2} > k_{b,2}$). Notice how this means that partial paths closer to the goal are preferred over paths with the same estimated overall length but greater distance from the goal (ie less progress towards the goal).



In this diagram, the blue nodes are nodes on the queue, red nodes are consistent nodes and white nodes are not on the queue. Here, the parallelogram heuristic is used and $k=[k_1,k_2]$, $k_a<k_b \rightarrow k_{a,1}<k_{b,1}$ or ($k_{a,1} = k_{b,1}$ and $k_{a,2} <= k_{b,2}$). Notice how this means that partial paths closer to the start are preferred over paths with the same estimated overall length but greater g-value. Notice also the parallelogram formed by the consistent vertices.

Implementation

The RPP was implemented on top of a D*-lite framework. Given the D*-lite psuedo-code, D*-lite-RPP, the Roadmap Path Planner that utilizes D*-lite, can be implemented as follows

The psuedocode requires the following additional function to manage the priority queue: U.Reorder() means reorder the queue taking into account the current s_{start}, which can be interpreted as "for all u in U U.Insert(u,CalculateKey(u))"

procedure RPP()
{31} For some c_1 in C
{32} $s_{start} = c_1$
{33} Initialize()
{34} ComputeShortestPath()
{35} For all c in $C \cap c_1$
{36} U.Reorder()
{37} ComputeShortestPath();
{38}Scan graph for any changed edge costs
{39} if any edge costs changed
{40} for all directed edges {u,v} with changed edge costs
{41} Update the edge cost c(u,v)
{42} UpdateVertex(u)
{43} for all c in C
{44} U.Reorder()
{45} ComputeShortestPath()

Conclusions/Directions to Explore

Unfortunately, I was unable to fully debug my implementation. This means I don't have statistics for computation required for the D*-lite-RPP algorithm to draw conclusions on. I was expecting to see that reusing the priority queue is most useful for situations where: 1) the path is long relative to the size of the cost map. This would mean that the changes in the priority queue would be minimal for the large portion of the path that is far from the start locations. 2) the paths from locations in the cost map to the goal are not separated by an obstacle, so that there is never a great disparity between queue orders as the algorithm searches for paths that switch which way around the obstacle they go around because of the start location. Lastly, I believe the number of searches (and hence queue reorderings) can be reduced by ordering the searches to start locations so that in searching for a path to a location in the cost map, you go through other locations in the cost map. Such a scheme could be as simple as continuing to search in the neighborhood of locations in the cost map that result in other search start locations becoming consistent, otherwise randomly choose a location in the cost map to search to.

D*-lite : Basic Version[3]

The psuedocode uses the following functions to manage the priority queue. U.TopKey() returns the smallest priority of all vertices in priority queue U. (If U is empty, then U.TopKey() returns [∞;∞]). U.Pop() deletes the vertex with the smallest priority k. U.Update(s,k) changes the priority of vertex s in the priority queue U to k. (It does nothing if the current priority of vertex s already equals k.) Finally, U.Remove(s) removes vertex s from the priority queue U.

```
procedure CalculateKey(s)
{01} return [min(g(s),rhs(s) + h(s_start,s); min(g(s),rhs(s))];

procedure Initialize()
{02} U := 0
{03} for all s in S, rhs(s) = g(s) = ∞
{04} rhs(s_goal) = 0
{05} U.insert(s_goal,CalculateKey(s_goal)

procedure UpdateVertex(u)
{06} if(u ≠ s_goal), rhs(u) = min_{t in Succ(u)}(c(u,t)+g(t))
{07} if (u in U), U.Remove(u)
{08} if(g(u) ≠ rhs(u)) U.insert(u,CalculateKey(u))

procedure ComputeShortestPath()
{09} while(U.TopKey() < CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start)
{10}    u = U.Pop();
{11}    if(g(u) > rhs(s))
{12}            g(u) = rhs(u)
{13}            for all s in Pred(u) UpdateVertex(s)
{14}    else
{15}            g(u) = ∞
{16}            for all s in Pred(u) U UpdateVertex(s)

procedure Main()
{17} Initialize()
{18} ComputeShortestPath();
{19} while(s_start ≠ s_goal)
{20} /* if(g(s_start) = ∞) then there is no known path */
{21} s_start = arg min_{t in Succ(s_start)} (c(s_start,t) + g(t))
{22} Move to s_start
{23} Scan graph for changed edge costs
{24} if any edge costs changed
{25}    for all directed edges {u,v} with changed edge costs
{26}            Update the edge cost c(u,v)
{27}            UpdateVertex(u)
{28}    for all s in U
{29}            U.Update(s,CalculateKey(s))
{30}    ComputeShortestPath()
```

References

[1] Bellingham, J., Richards, A., How, J.P., "Receding Horizon Control of Autonomous Aerial Vehicles," ACC, 2002.

2 Koenig, S., Likhachev, M., "Improved Fast Replanning for Robot Navigation in Unknown Terrain," *Proceedings of the International Conference on Robotics and Automation,* 2002

3 http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#S5