Lattice Dstar Lite Project Summary

Alvin Shek and Qin Lin

Summary of D* Lite Algorithm

A* is a common algorithm used for finding the shortest or least-cost path from one point to another. However, A* only works well if the robot's known map is the true map. If there are any updates to the map, A* will have to be rerun from scratch since its original path may run into an obstacle. D* Lite comes in by allowing updates to the original path without completely replanning from scratch. This is because not all states' cost values will change from the updates to the map, so there is no need to update those. Not all of the states with changed cost values need to updated either to find a new optimal path: Dstar Lite greedily updates states that are more promising first and may find the best path without needing to update every changed state.

Details of D* Lite Algorithm

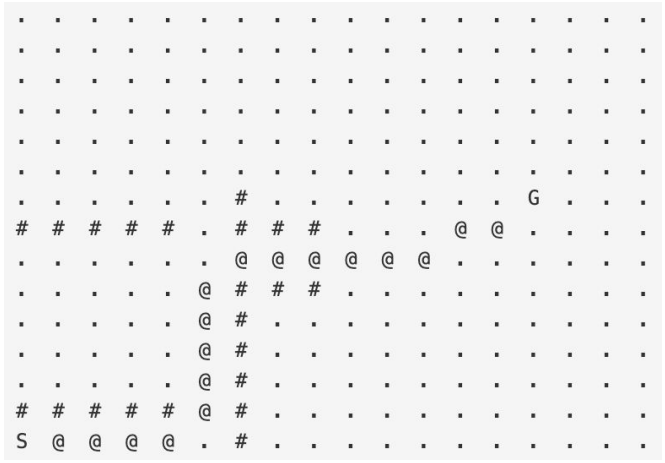
Visual Comparison of A* and D* Lite

Let's walk through an example to understand how D* can reuse its previous plan.
Below shows the scenario where # denotes impassable walls, S is start, and G is goal.

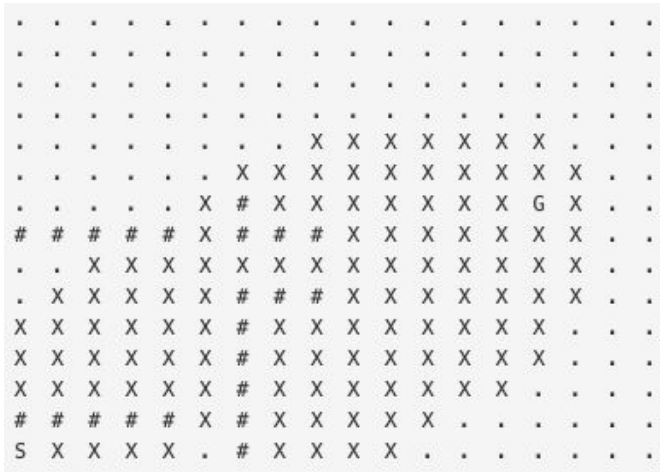
```
. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . .
. . . . . . # . . . . . . . G . . .
# # # # # . # # # . . . . . . . . .
. . . . . . . . . . . . . . . . .
. . . . . . # # # . . . . . . . . .
. . . . . . # . . . . . . . . . . .
. . . . . . # . . . . . . . . . . .
. . . . . . # . . . . . . . . . . .
# # # # # . # . . . . . . . . . . .
S . . . . . # . . . . . . . . . . .
```

D* Lite on the first iteration should run identically to A* since both have to plan from scratch.

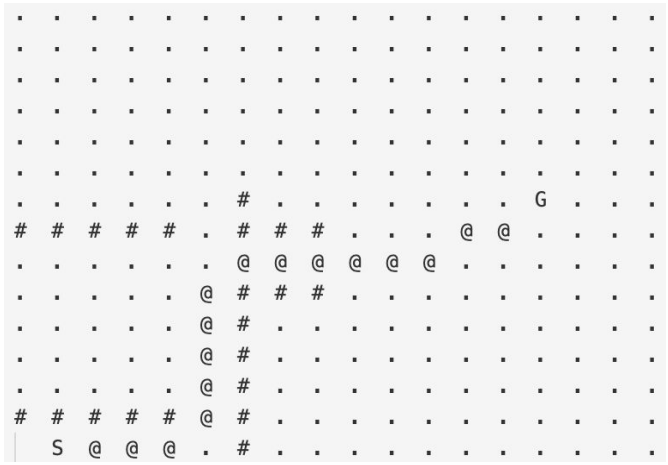
The shortest path would the the following:



The states that were expanded (denoted by X) are equivalent to those expanded by A*.

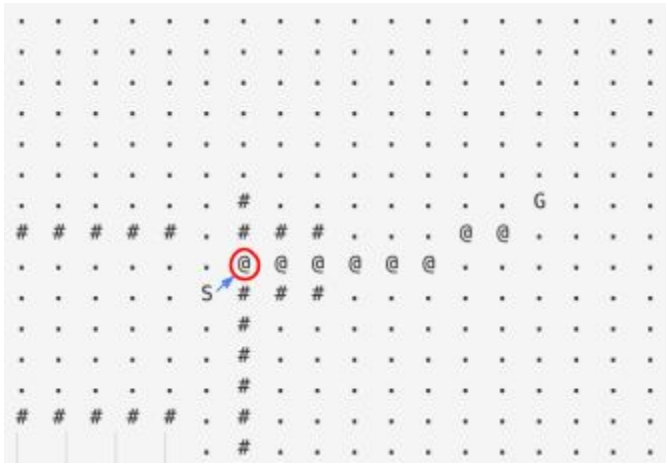


Now as the robot moves along, it doesn't detect any changes to the map and can just use its current plan.

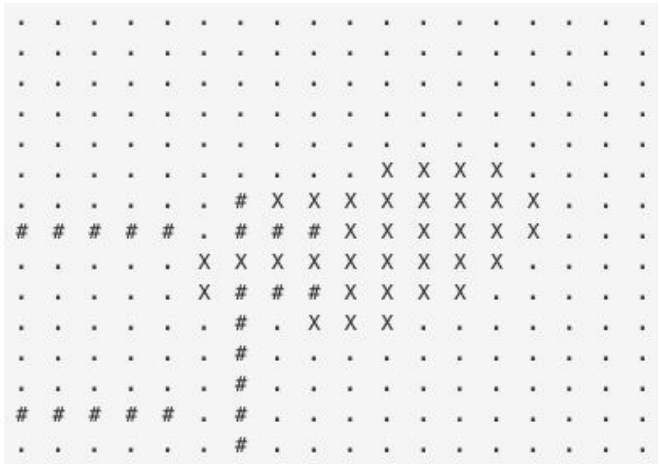


No extra states need to be expanded since there was no change to the plan.

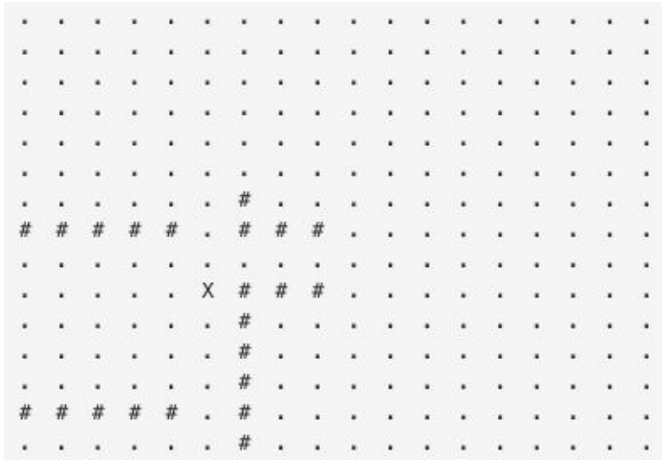
Now, the robot will continue this process of moving and scanning its environment and simply reuse its existing plan since nothing has changed. This is identical between A* and D* Lite. Finally, however, the robot detects a blockage at the entrance of the narrow passageway, circled in red. The robot had planned to move towards that spot given by the direction arrow.



Now, A* would need to replan completely from scratch and expand all these states:



D* Lite, however, only expands these states:

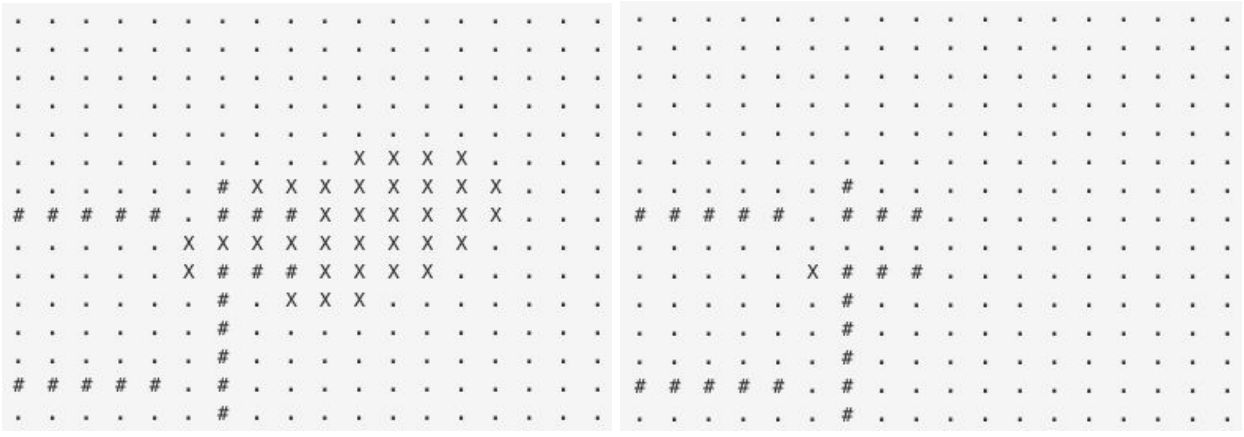


Why only that state? Let's compare G-values before and after we observed the blockage: Before(Top) and After(Bottom). I've circled all the changed values in red. The reason for those values is explained here.

```
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  6.8  5.8  4.8  3.8  2.8  2.4  2.0  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  8.4  7.4  6.4  5.4  4.4  3.4  2.4  1.4  1.0  1.4  Inf  Inf
Inf  Inf  Inf  Inf  Inf  9.8  ###  7.0  6.0  5.0  4.0  3.0  2.0  1.0  0.0  1.0  Inf  Inf
###  ###  ###  ###  ###  10.2 ###  ###  ###  5.4  4.4  3.4  2.4  1.4  1.0  1.4  Inf  Inf
Inf  Inf  12.8 11.8 10.8 9.8  8.8  7.8  6.8  5.8  4.8  3.8  2.8  2.4  2.0  2.4  Inf  Inf
Inf  14.2 13.2 12.2 11.2 10.2 ###  ###  ###  6.2  5.2  4.2  3.8  3.4  3.0  3.4  Inf  Inf
15.7 14.7 13.7 12.7 11.7 11.2 ###  8.7  7.7  6.7  5.7  5.2  4.8  4.4  4.0  Inf  Inf  Inf
16.1 15.1 14.1 13.1 12.7 12.2 ###  9.1  8.1  7.1  6.7  6.2  5.8  5.4  5.0  Inf  Inf  Inf
16.5 15.5 14.5 14.1 13.7 13.2 ###  9.5  8.5  8.1  7.7  7.2  6.8  6.4  Inf  Inf  Inf  Inf
###  ###  ###  ###  ###  14.2 ###  9.9  9.5  9.1  8.7  8.2  Inf  Inf  Inf  Inf  Inf  Inf
19.7 18.7 17.7 16.7 15.7 Inf  ###  10.9 10.5 10.1 9.7  Inf  Inf  Inf  Inf  Inf  Inf  Inf
```

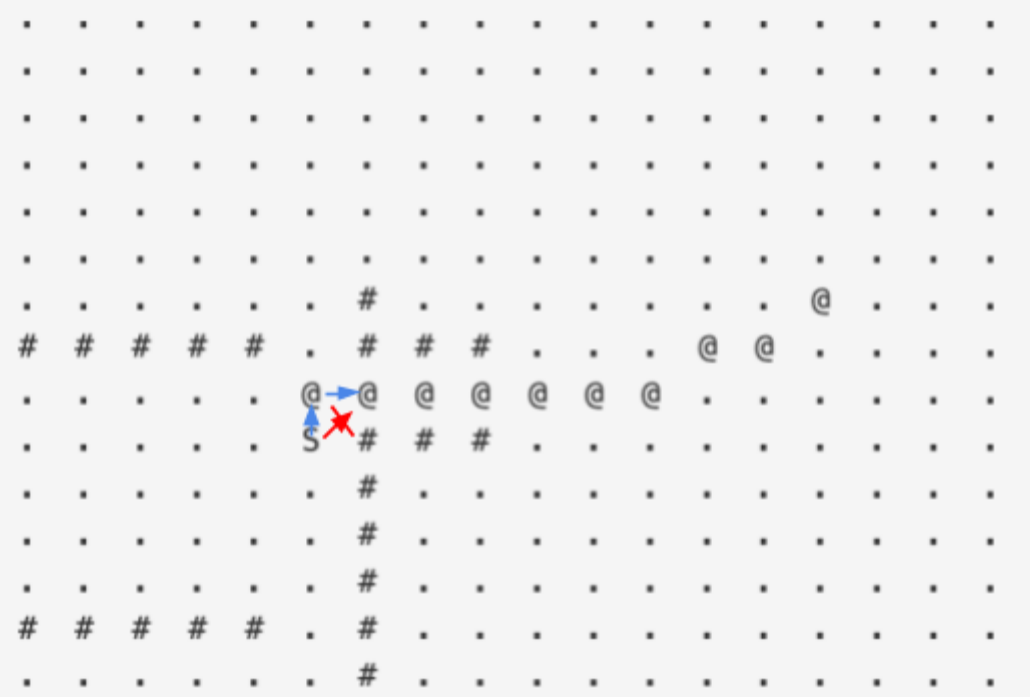
```
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  Inf  Inf  6.8  5.8  4.8  3.8  2.8  2.4  2.0  Inf  Inf  Inf
Inf  Inf  Inf  Inf  Inf  Inf  8.4  7.4  6.4  5.4  4.4  3.4  2.4  1.4  1.0  1.4  Inf  Inf
Inf  Inf  Inf  Inf  Inf  9.8  ###  7.0  6.0  5.0  4.0  3.0  2.0  1.0  0.0  1.0  Inf  Inf
###  ###  ###  ###  ###  10.2 ###  ###  ###  5.4  4.4  3.4  2.4  1.4  1.0  1.4  Inf  Inf
Inf  Inf  12.8 11.8 10.8 9.8  8.8  7.8  6.8  5.8  4.8  3.8  2.8  2.4  2.0  2.4  Inf  Inf
Inf  14.2 13.2 12.2 11.2 10.8 ###  ###  ###  6.2  5.2  4.2  3.8  3.4  3.0  3.4  Inf  Inf
15.7 14.7 13.7 12.7 12.2 11.8 ###  8.7  7.7  6.7  5.7  5.2  4.8  4.4  4.0  Inf  Inf  Inf
16.1 15.1 14.1 13.7 13.2 12.8 ###  9.1  8.1  7.1  6.7  6.2  5.8  5.4  5.0  Inf  Inf  Inf
16.5 15.5 15.1 14.7 14.2 13.8 ###  9.5  8.5  8.1  7.7  7.2  6.8  6.4  Inf  Inf  Inf  Inf
###  ###  ###  ###  ###  14.8 ###  9.9  9.5  9.1  8.7  8.2  Inf  Inf  Inf  Inf  Inf  Inf
20.2 19.2 18.2 17.2 16.2 Inf  ###  10.9 10.5 10.1 9.7  Inf  Inf  Inf  Inf  Inf  Inf  Inf
```

For reference, what A*(left) and D* Lite(right) expanded:

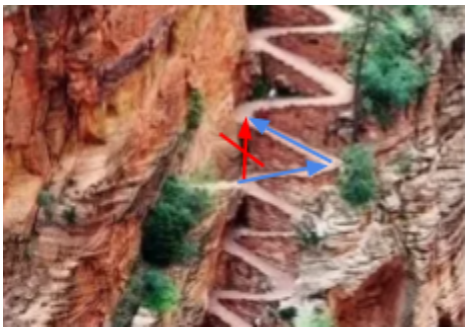


First of all, A* expanded a bunch of states to near the goal that weren't even affected by the blockage. Second, we can see there were a lot of changed G-values below and to the left of the blockage. However, these aren't even worth expanding again since they don't lie on the optimal

path. D* Lite greedily re-expands states, and once it finds a new optimal path, it's finished. In this case, It only had to expand one state to find a new optimal path:



To us, it's **obvious that the blocked cell is a wall**, but to the robot, only the red transition is invalid, but not necessarily that cell itself. It thinks other transitions to that cell are still valid.



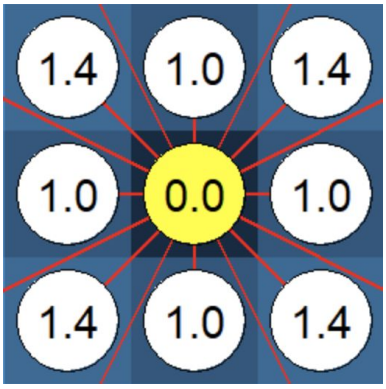
Now, on a 2D grid with 1 or 0 for obstacles, that behavior might sound suboptimal, but consider a real-life example of hiking up a switchback. Moving straight up to a point may be extremely hard, if not impossible, but getting to the same location(state) through a different path/transition is more feasible.

So the robot tries another route through that same spot by moving up and left, designated by the blue arrow.

When overall compared to A*, D* Lite not only avoids expanding states that haven't changed, but also greedily picks which of the changed states

Transition Costs for Basic 8-connected grid

Here, we assume an 8-connected grid with euclidean distance as the transition costs:



Source: https://mgimond.github.io/Spatial/raster-operations-in-r.html
In the above figure, moving left, right, up, and down incur cost of 1 as expected. Diagonal transitions incur cost of sqrt(2) by euclidean distance.

Also as a note, A* normally plans from start to goal, so you might be used to seeing the G-values increase from start to goal, not the opposite. But here, I compare backwards A* to D* Lite since D* Lite searches backwards, so G-values are defined with respect to goal and not start.

In-Depth Explanation of Algorithm

```
procedure Main()
{21'} s_last = s_start;
{22'} Initialize();
{23'} ComputeShortestPath();
{24'} while (s_start ≠ s_goal)
{25'}   /* if (g(s_start) = ∞) then there is no known path */
{26'}   s_start = arg min_{s' ∈ Succ(s_start)} (c(s_start, s') + g(s'));
{27'}   Move to s_start;
{28'}   Scan graph for changed edge costs;
{29'}   if any edge costs changed
{30'}     k_m = k_m + h(s_last, s_start);
{31'}     s_last = s_start;
{32'}     for all directed edges (u, v) with changed edge costs
{33'}       Update the edge cost c(u, v);
{34'}       UpdateVertex(u);
{35'}     ComputeShortestPath();
```

Source: Sven Koenig, Maxim Likhachev, "D*lite," Eighteenth national conference on Artificial intelligence,Menlo Park, CA, USA, pp. 476–483, 2002

So first-off, how do we update that current robot state/vertex, u?

```
procedure UpdateVertex(u)
{07'} if (u ≠ s_goal) rhs(u) = min_{s' ∈ Succ(u)} (c(u, s') + g(s'));
{08'} if (u ∈ U) U.Remove(u);
{09'} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
```

The underlined statement reads: Find the successor, *s'* of current state *u* that has the least sum of transition cost *c(u, s')* and g-value, *g(s')*.

The old value of vertex u is 10.2 = 8.8 + 1.4, where g(s') = 8.8 and c(u,s') = 1.4(diagonal):

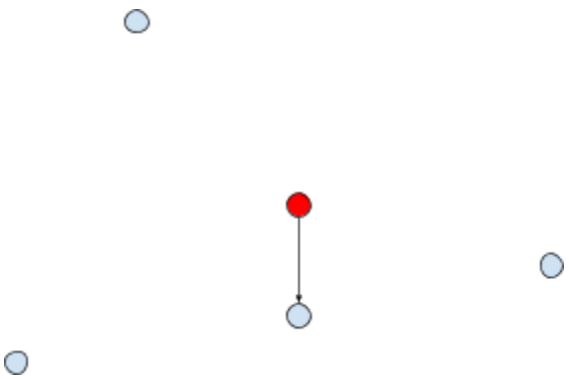
Now, however, since the robot observed that transition as "blocked", the true transition cost there is infinite. So when we perform the operation underlined in red, the new rhs value of current

position u would be 9.8(above state) + 1.0(move up) = 10.8. Now, the optimal path is to move upwards as shown from a previous figure:

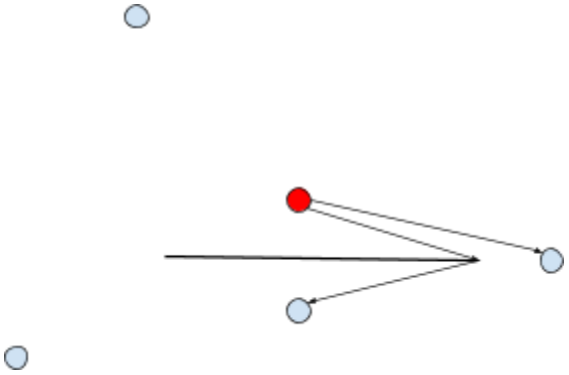
In the next step, D* Lite then computes the shortest path with these updated transition costs:

procedure ComputeShortestPath()
{10'} while (U.TopKey() $\dot{<}$ CalculateKey(s_{start}) OR $rhs(s_{start}) \neq g(s_{start})$)
{11'} $k_{old} = $ U.TopKey();
{12'} $u = $ U.Pop();
{13'} if ($k_{old} \dot{<}$ CalculateKey(u)) **1**
{14'} U.Insert(u, CalculateKey(u));
{15'} else if ($g(u) > rhs(u)$)
{16'} $g(u) = rhs(u)$;
{17'} for all $s \in Pred(u)$ UpdateVertex(s); **2**
{18'} else
{19'} $g(u) = \infty$;
{20'} for all $s \in Pred(u) \cup \{u\}$ UpdateVertex(s); **3**

Step 1 of the function purges the open set of states with any outdated cost values. The open set contains all states that have yet to be expanded/explored to see if it should be in the path. It prioritizes which states should be explored by expanding in the order of the least cost states (a common storage is min heap). This important because these states may not necessarily be good to expand anymore given our updated map info. Consider the picture below:



Suppose we are the red dot and want to move to the closest nearby blue dot. The blue dot directly below is the closest one. However, suppose we detect a wall between the two:



Clearly, the dot to the right is now the closest one since getting to the bottom dot now takes longer by moving around the wall. We'll associate "closest" with "least cost" in this example.

procedure ComputeShortestPath()

{10'} while (U.TopKey() $\dot{<}$ CalculateKey(s_{start}) OR $rhs(s_{start}) \neq g(s_{start})$)
{11'} $k_{old} = $ U.TopKey();
{12'} $u = $ U.Pop();
{13'} if ($k_{old} \dot{<}$ CalculateKey(u)) 1
{14'} U.Insert(u, CalculateKey(u));
{15'} else if ($g(u) > rhs(u)$)
{16'} $g(u) = rhs(u)$;
{17'} for all $s \in Pred(u)$ UpdateVertex(s); 2
{18'} else
{19'} $g(u) = \infty$;
{20'} for all $s \in Pred(u) \cup \{u\}$ UpdateVertex(s); 3

Back to the function, that clean-up step updates the cost associated with each state to reflect changes in the map. What makes up the "cost" is described here: ___link to outline___.

Now for step 2 of the function, we need to explain the difference between “g” and “rhs”. Both represent the same value of cost-incurred, or distance traveled thus far. Having this value encourages the planner to take the shortest path and travel the least distance to reach a target.

“rhs” or “V” is different from G in that it is an early prediction of the true G-value, meaning that we update it first, and only later update the G-value. Below describes what it means to compare the two values:

- consistent: G = V (true cost is known)
- over-consistent: G > V (overestimated true cost, so need to update path)
- under-consistent: G < V (underestimated true cost, so need to update path)

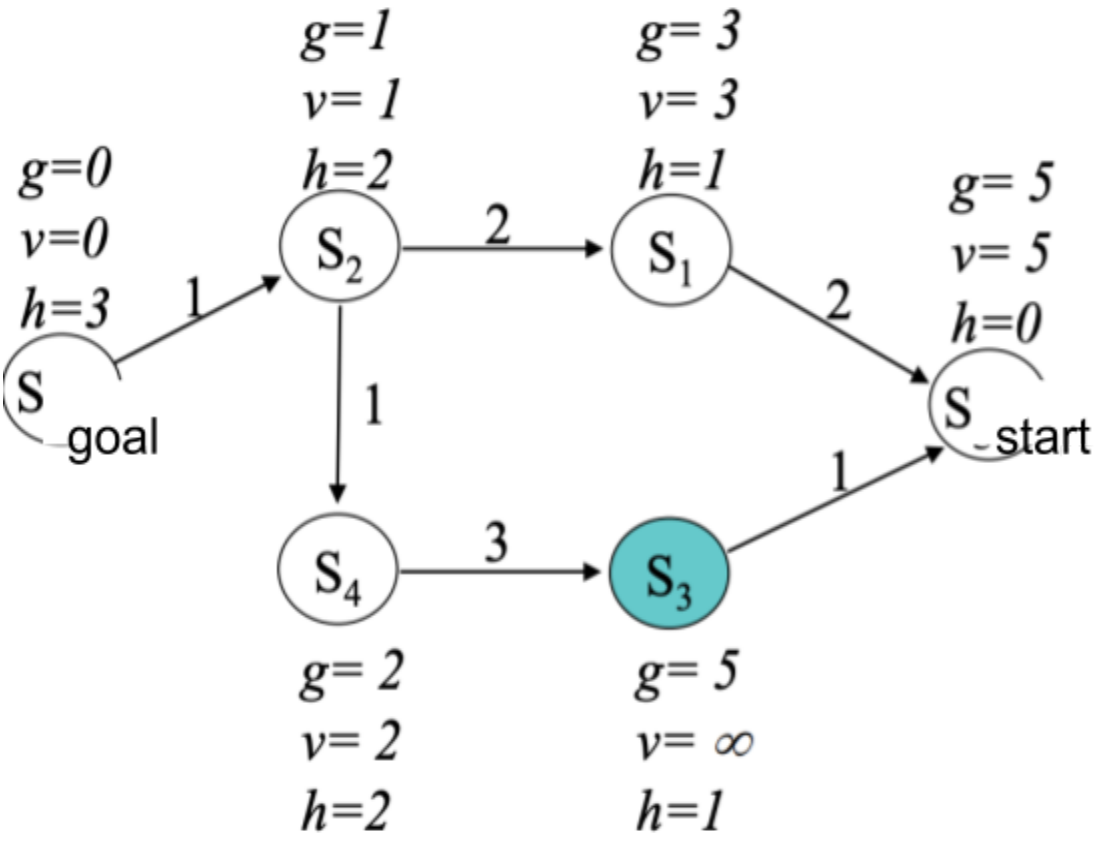
When consistent, we’ve found the true g-value of this state. When we expand the target state(start since backward search) and it is consistent, the search will terminate since we know we’ve found the optimal value for the target state and thus have found an optimal path to it.

When a state is over-consistent, we’ve found a better cost, so we can simply set the g-value equal to the v-value and update predecessor states(explained later). This is step 2 of the above function.

When a state is under-consistent, we need to set G to infinity to reflect the fact that this state’s true cost has changed. We then propagate this infinite value to neighbor** states since their cost value needs to be changed as well. This is step 3 of the function.

In the case of step 2 of the above function, we update predecessor** states. For the above 8-connected grid example where we can move to any adjacent state, predecessor and successor aren’t relevant, but for other use-cases like nonholonomic vehicles, it’s important to distinguish predecessor from successor. Let’s take a car as an example. A car has heading/direction(Θ) it faces as well as either positive or negative velocity. Successors states would be in front of the car since the current position leads to those successors. Predecessor states would be behind the car since those predecessors could lead to the current state.

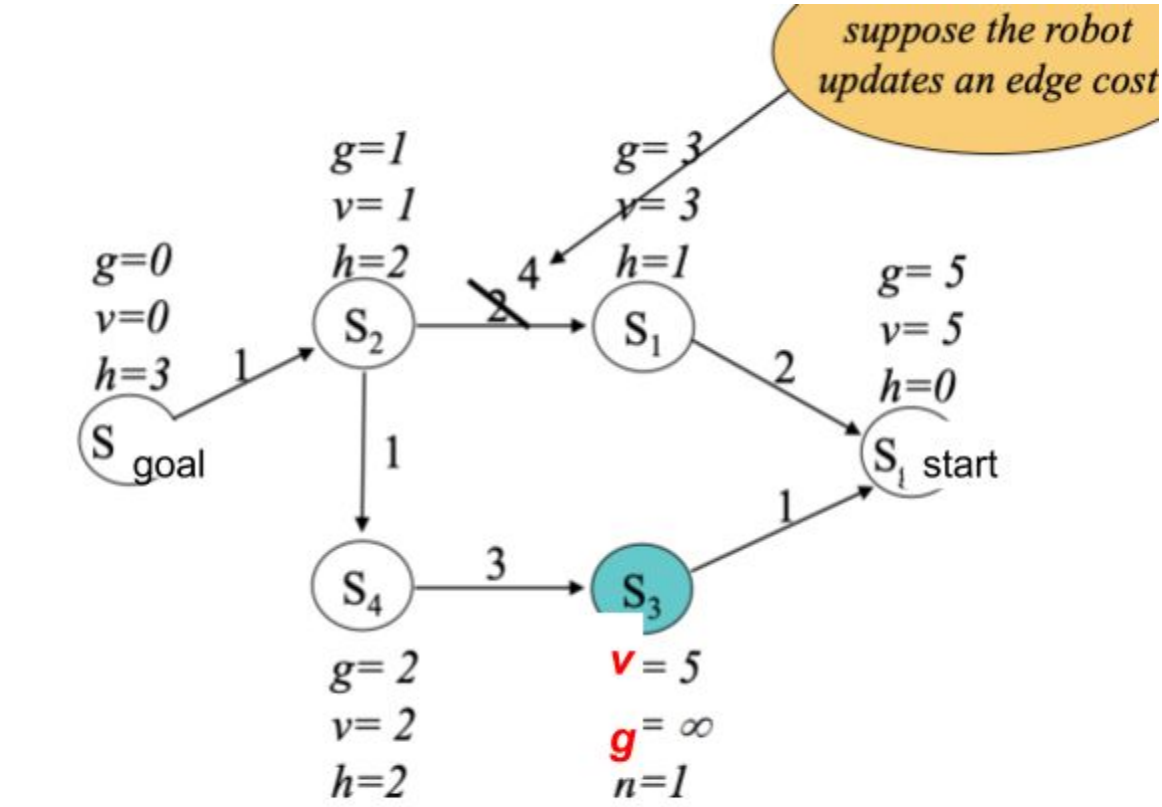
To make it more clear how propagating these values works, let’s look at some slides the motion planning course taught by Dr. Maxim Likhachev:



Source: http://www.cs.cmu.edu/~maxim/classes/robotplanning/lectures/incrementalsearch_16350_sp20.pdf

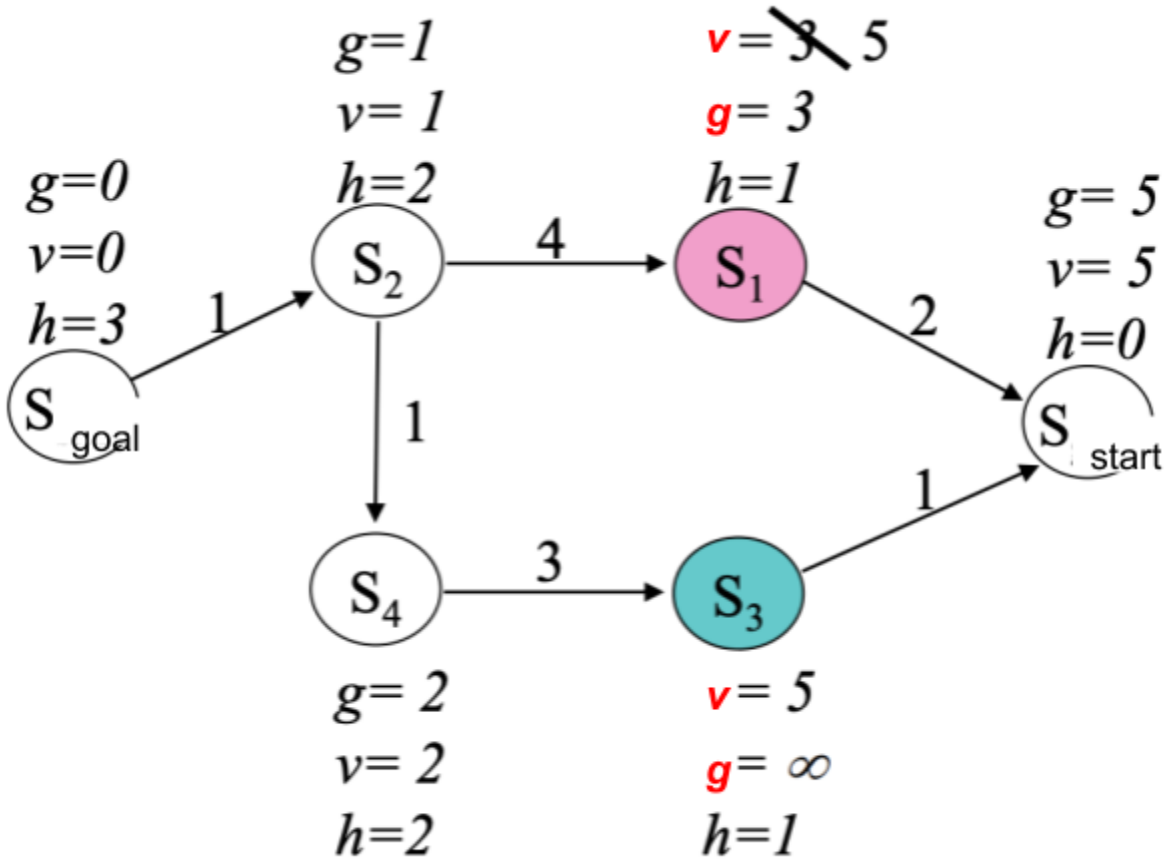
Looking above, we can see that all states are consistent (v = g) and their values make sense. The graph points from goal to start since in D* Lite, we search backwards. For example for state S_1, g = 3 = $\min_{successors}(c(s,s') + g(s'))$ = (2 + 1), where the best successor is S_2. NOTE: I've modified that slide since it shows ARA*(forward search) and not D* Lite(backward search). S_2 is the successor of S_1 even though it looks like S_2 leads to S_1 since we are doing backward search.

Pretend that there is an update to the transition cost between S_1 and S_2:



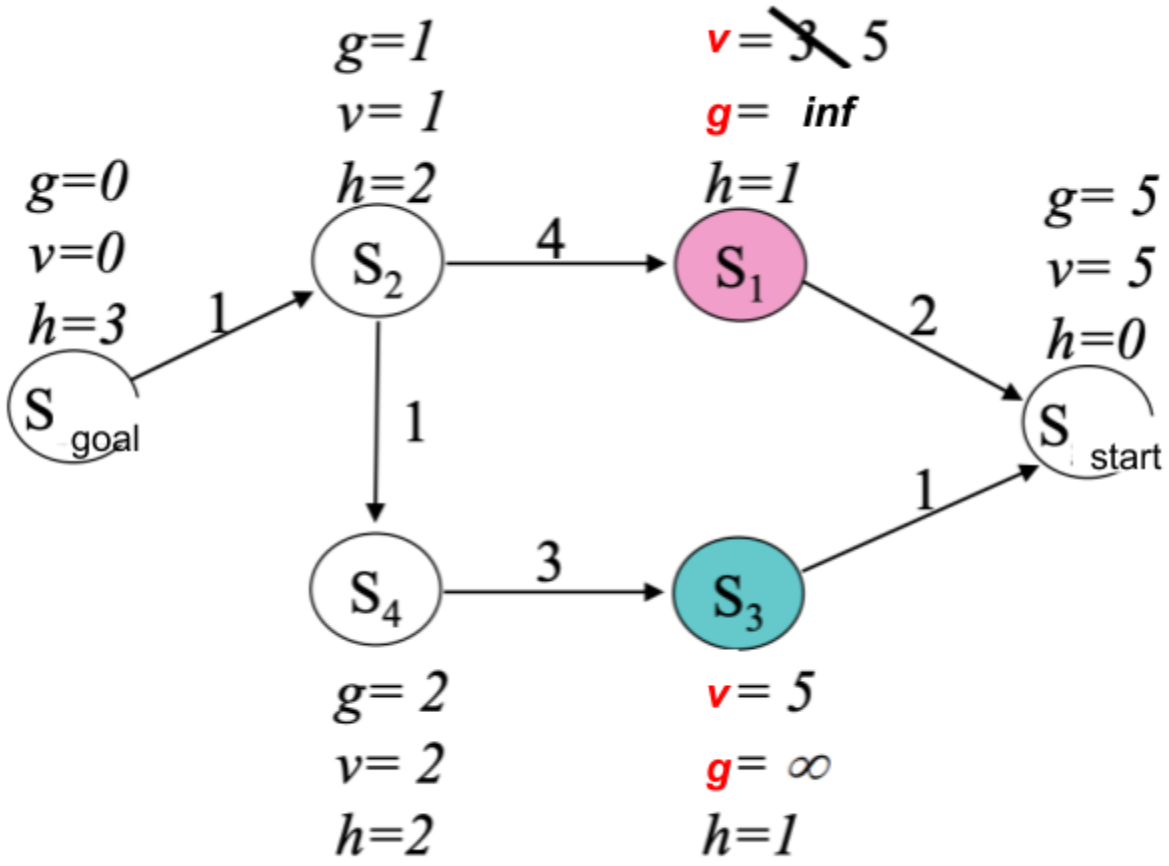
Carnegie Mellon University

We need to update S_1's v-value accordingly:



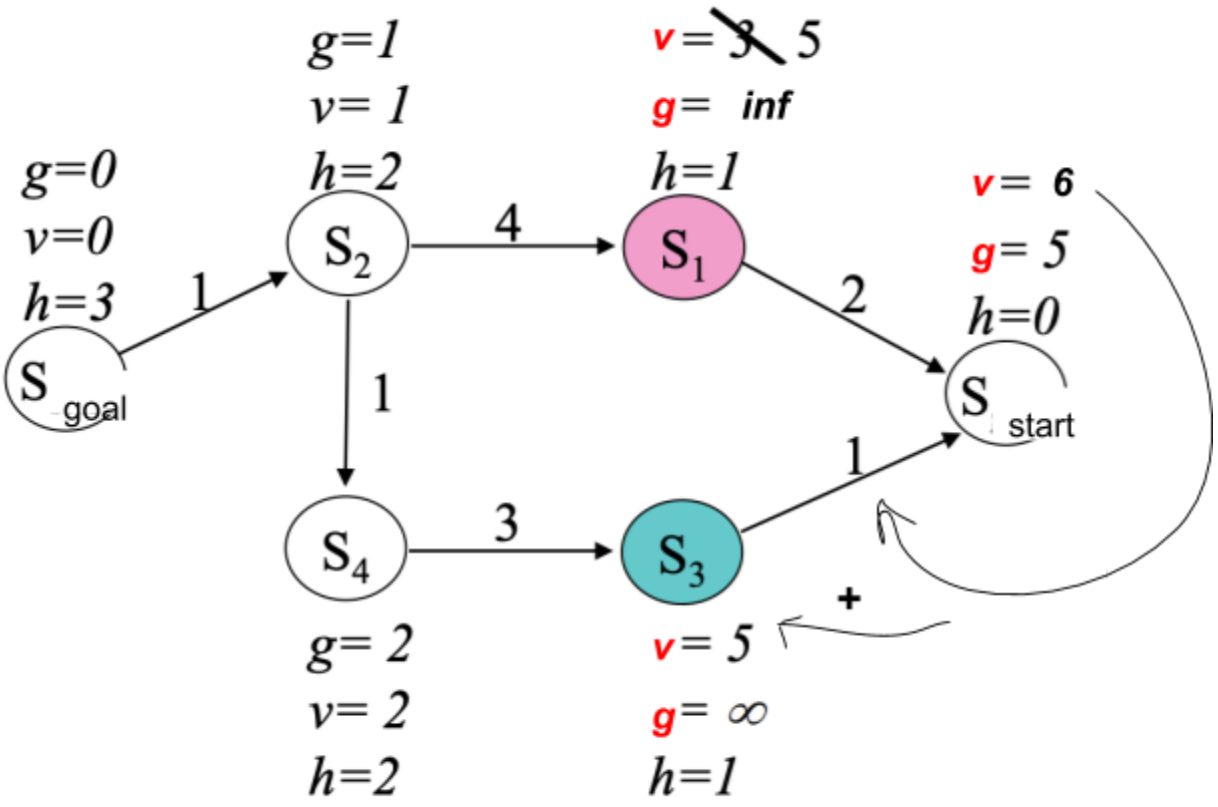
Carnegie Mellon University

Now that G < V, we are under-consistent and need to set G = infinity:



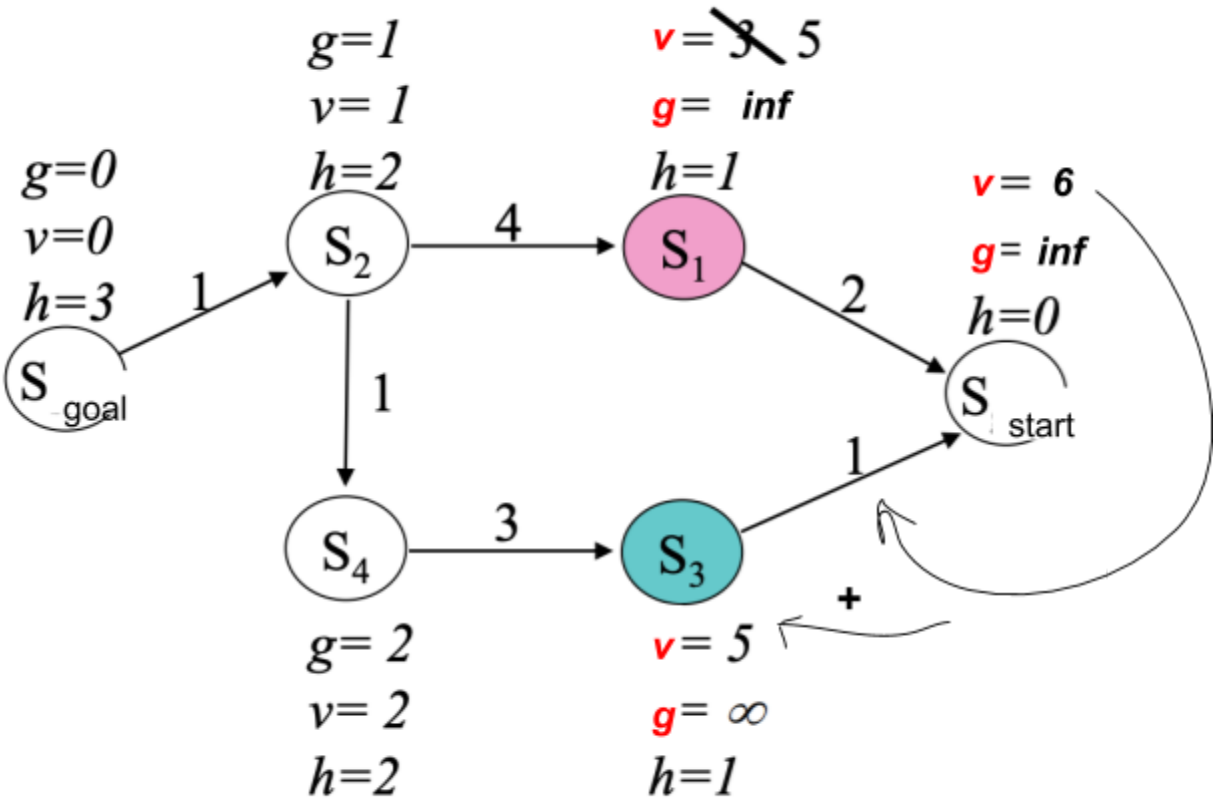
Carnegie Mellon University

We then propagate these changes down further to the start:



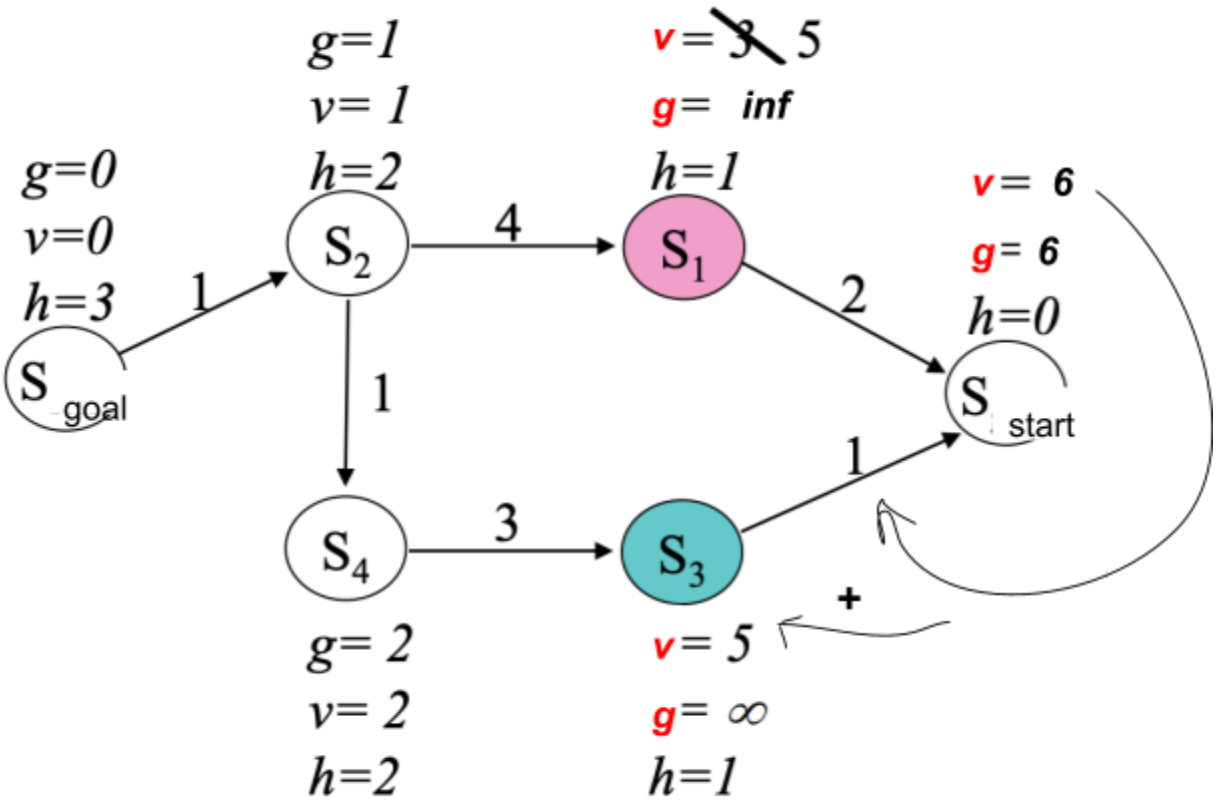
Carnegie Mellon University

Above, we update v-value with the minimum cost successor's value + transition cost.
G again is inconsistent, so we need to update it to infinity.



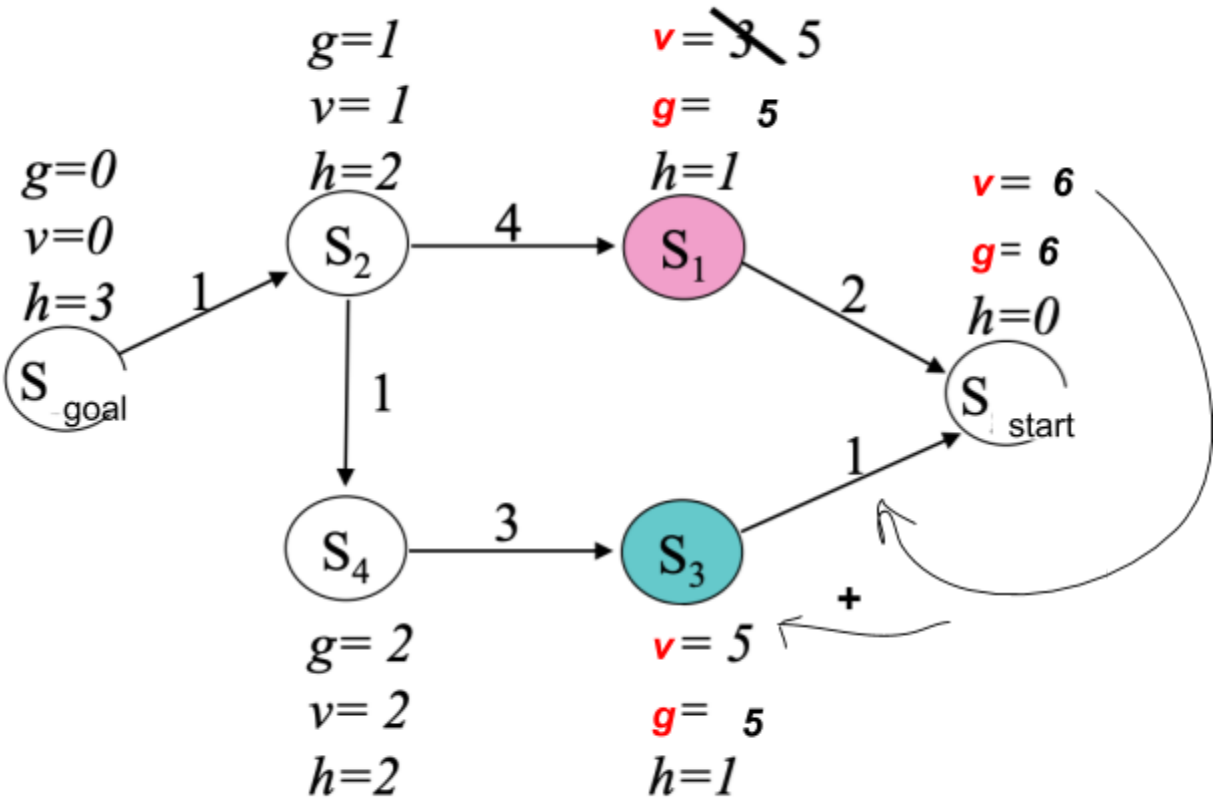
Carnegie Mellon University

Now, both S_1 and start are in the open-set.We expand start first it will have the lowest cost(g + h, explained here: ___). I've modified the algorithm to take the lowest v-value, so now S_{start} is deliberating between (5 + 2) and (5+ 1), so it picks (5 + 1) since this is lower.



Carnegie Mellon University

Now the either S_1 or S_2 is expanded, depending on which has a lower heuristic cost since both have the same v-value. Either way, they both get updated:



Carnegie Mellon University

As referenced in the summary of D* Lite Algorithm section, we will now describe how we greedily expand states using a "cost". This cost for usual A* is defined as

$$f(cost) = g + eps * h$$

where $g = min(V, G)$ and $h = heuristic$, and eps = greedy expansion factor.

Heuristic is a value that guides the search process to expand more promising states that may be more likely to contribute to a solution.

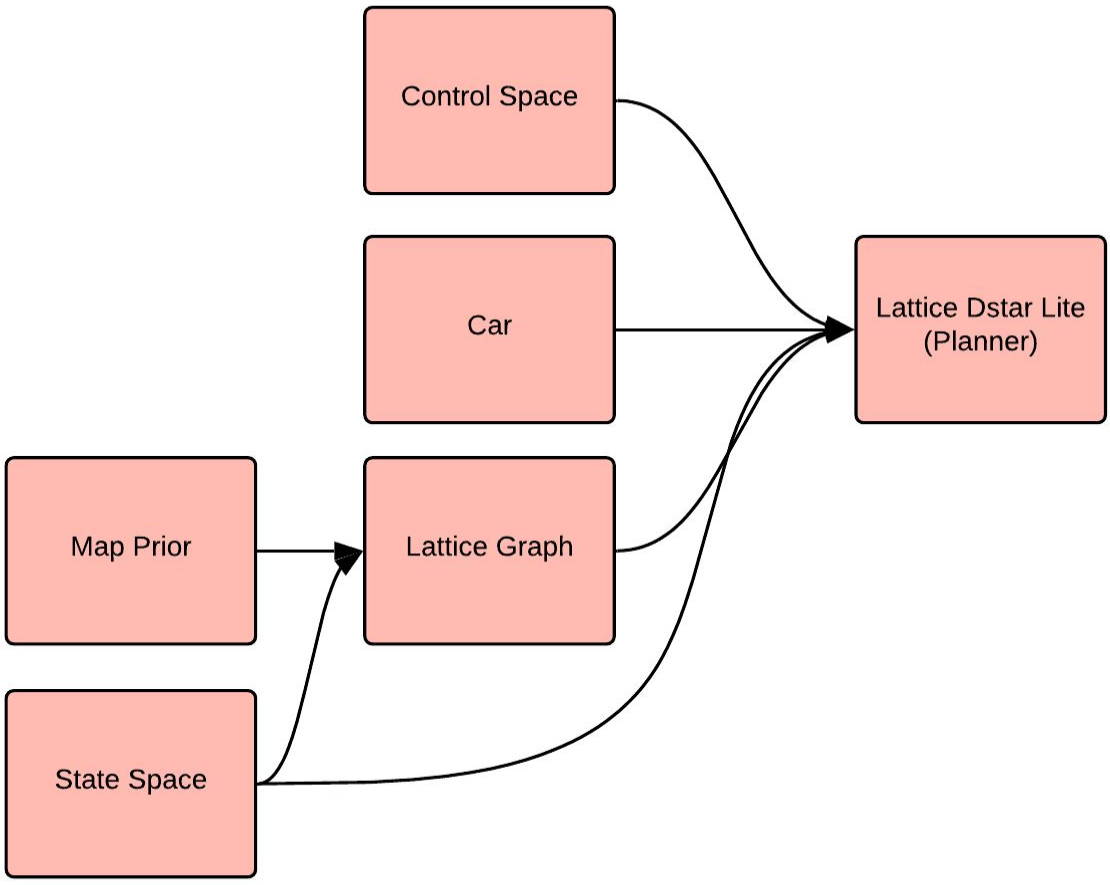
For D* Lite, we have two separate cost values instead of one:
$k1 = f$ and $k2 = g$. This distinction is for tie-breaking if the f-value of two states happen to be the same.

Overview of Lattice Dstar Lite

Here's a video showing the final, yet initial result:
https://drive.google.com/file/d/1YRAnAuptVL6-ZXNkCBL2XQ1yB0jDodmW/view?usp=sharing

Now it's time to discuss the important aspects of my Lattice Dstar Implementation, which has some key, high-level features:

Architecture: Data structures and Computation Pipeline



We'll first define the purpose of each component of the planning/simulation stack.

Map Prior

The initial 3D terrain map stored by the robot. This map will get updated as the robot moves observes new map data. The map is simply a 2D numpy array mapping a 2D coordinate to a height value: {X} x {Y} —> {Z}.

State Space

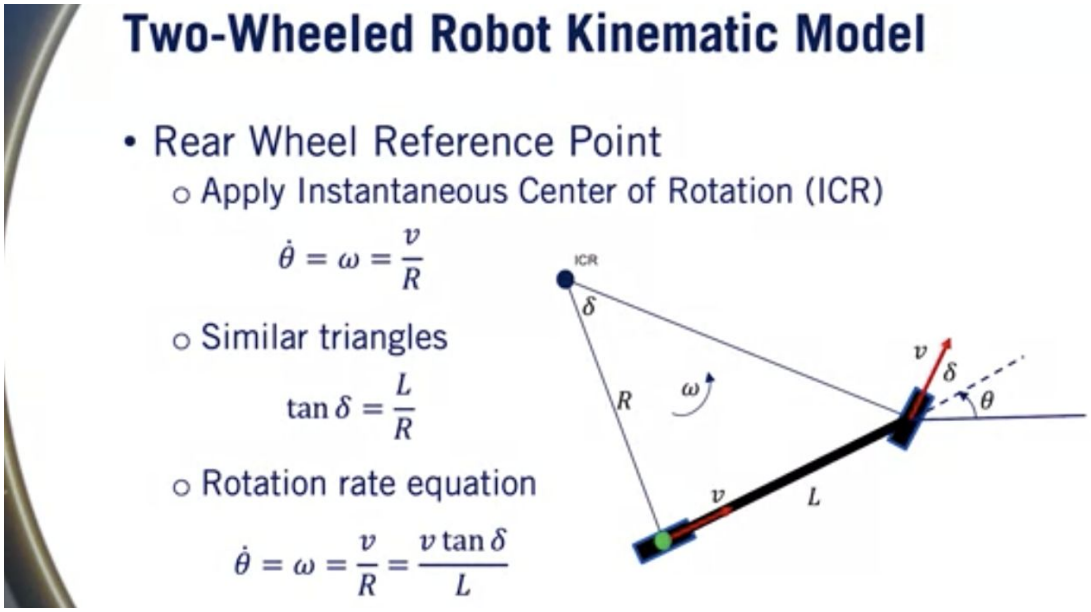
State has 5 dimensions: [x, y, theta, v, t] with discretization [dx, dy, dtheta, dv, dt]. Continuous state space is discretized into hashable tuples (xi, yi, thetai, vi, ti) so they can be stored in sets and dictionaries.

Control Space

Defined by steering angle ϕ, velocity v, time discretization dt, and time horizon T. Assumes instantaneous change in steering and velocity. Velocity is only positive (so no 3-point turns).

Car

Defined by bicycle model:



Generates a rollout given an Action(steer, velocity), current state, and time horizon. Contains adjustable parameter for length L of car.

Lattice Graph

Provides an interface between the planner and its map. Stores the current known map as well as an obstacle map(generated from 3D terrain map). Has an interface to update internal map and

return status of whether any changes were made. Also handles generation of all valid successor states given current state and calculates cost of every successor. Guarantees that returned successors are valid (don't collide with obstacles).
NOTE: It may seem strange that this class should deal with rollouts, but this is because the graph has all the map data and needs to cost the rollouts and remove any invalid successors that collide with obstacles.

Lattice Dstar Lite

Main planner. At initialization, takes in all of the components in the diagram as well as planner-specific hyperparameters such as epsilon (greediness factor in search). Interfaces with simulation by taking in a start and goal and returning a state path and policy of actions and action durations used to generate it. In the same call to generate this plan, the planner also takes in a square window of newly-observed map data, which may or may not match the internally known data. The planner will adjust and replan if its Lattice Graph detects changes.

Typical Planning Cycle

1. Initialize all components
    a. Lattice Dstar Lite will precompute a set of rollouts for every discrete theta bin
    b. It then stores this in the lattice graph
2. While start != goal:
    a. simulator generates a small observation window of ground-truth map data
    b. calls the planner with start, goal, window and gets path and policy
    c. simulator "steps" and executes policy as best as it can. sets new start and restarts loop if observe new map data
        i. **NOTE: could do a more intelligent map update instead of just updating whenever new data is observed?**

Important Details

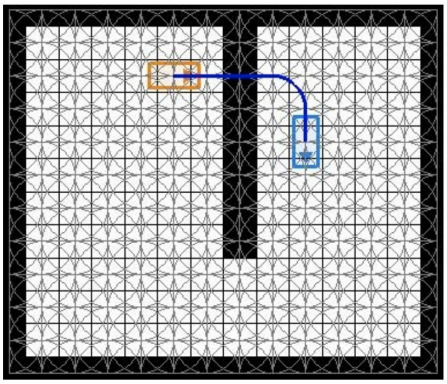
Looking back, we defined the "cost" of every state as f = g + h, where g represents cost incurred (ex: distance traveled) and h represents cost-to-go or heuristic (distance left to the goal). How do we define these?

Our heuristic is made up of two separate heuristics:

1. Non-holonomic shortest path without considering obstacles
2. Holonomic with obstacles

In regards to admissibility(underestimate of true cost) of heuristics, if two heuristics *h1* and *h2* are admissible, then *max(h1, h2)* is also admissible. So our heuristic is the max of those two. The source of that idea is from this paper: http://ppniv12.irccyn.ec-nantes.fr/paper/4Choi.pdf

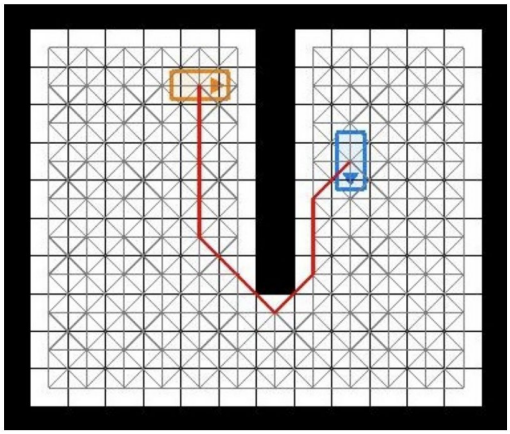
Non-holonomic without obstacle



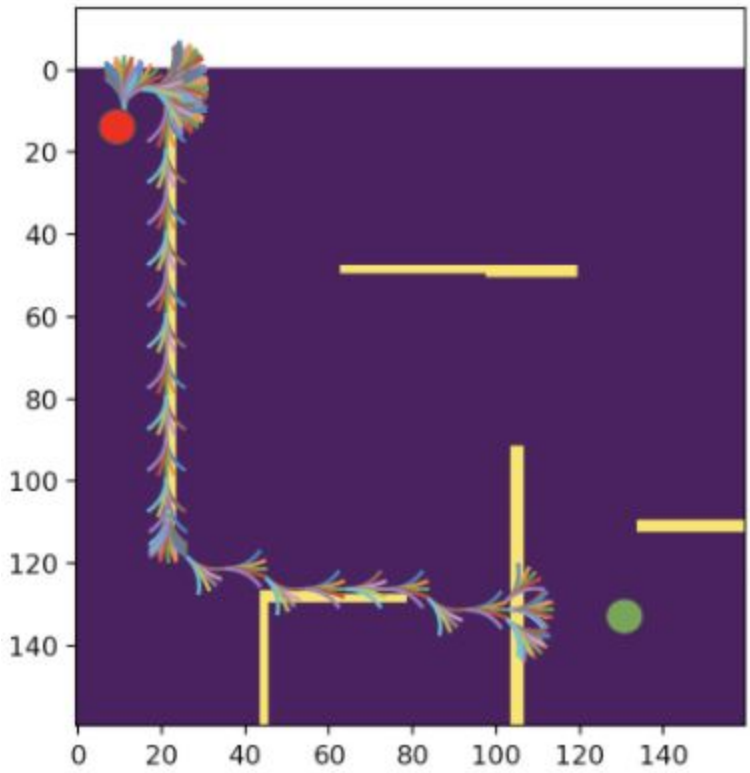
(b) Nonholonomic-without-obstacle for h_{3D}^{-}

Nonholonomic vehicles like cars can't just move in any direction at a given state, but rather has some heading direction that constrains its motion. For this reason, simple Euclidean distance severely underestimates the actual distance required to reach a goal by ignoring the current and goal heading. We use dubin's path to quickly find a path with heading considered and take the length of this path as the first heuristic. This python module was used: https://pypi.org/project/dubins/

Holonomic with obstacle



Notice how pure Euclidean Distance will just drive the search into a local minima and get stuck at the wall rather than movie around the wall:



On top of that, we also have transition costs, which are domain-specific. Since our project focuses on safe planning over rough terrain, we have various cost functions to capture what safe behavior should look like.

- Distance traveled
    - Encourage shortest distance path
- Time taken
    - Encourage fastest path since velocity is part of state space
- terrain roughness
- terrain incline
- others?

Trajectories with collisions are marked as infeasible and thrown out.

Low-level Implementation Details

MinHeap with Lazy Removal

In A*, the open-set can be implemented as a minHeap to allow O(1) removal of the min-cost state for expansion. However, in D* Lite, we need this behavior along with the ability to update any given state quickly, not just the min state. Updating a state involves removing it from the open set and reinserting it. In a minHeap, the only way to remove an item is to iterate through the entire data structure until the item is found. This O(N) operation becomes really expensive when we have many states with a fine level of discretization.

The solution is to have two separate data structures with our desired properties:

- open: minHeap for O(1) removal of min-cost state
- open_set: Set for O(1) removal and lookup of any state

When an item is added to open with some cost value, the state is also added to open_set.
When an item is removed from open (if its cost was changed), the state is only removed from the open_set, not the open. To address this discrepancy between open_set and open, we clean up open whenever we want to pop or peek the min cost item from open. Pseudocode for that is below:

```
def cleanup():
    while len(self.open) > 0:
        min_element = self.open[0]  # peek min element
        if min_element not in self.open_set:
            self.open.pop()         # pop that outdated element
        else:
            break                   # guaranteed that current min element is legitimate
```

Dynamic Constraints on Available Actions

We don’t consider the dynamic constraints fully at this level of planning, but still enforce restrictions on certain impossible actions. Suppose the car is moving at velocity = +20mph with steering angle = 5º. The car can’t instantaneously change its velocity to -20mph, so this is infeasible. While the car also cannot change instantaneously to +10mph, we still allow this since

our level of planning is high and the controller will handle the real constraints. The same idea applies to steering angle.

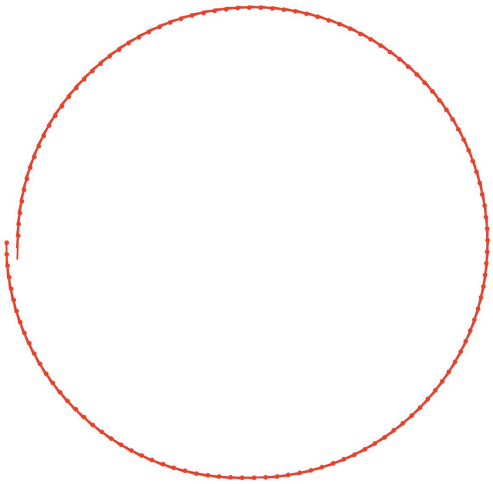
Threshold for Reaching a Target State

What defines “reaching the goal”? We only check that position and heading are within some tolerance of the goal position and heading.

```
is_similar_heading = (abs(theta1 - theta2) % TWO_PI) < heading_threshold
is_spatially_near = heuristic(state1, state2) < spatial_threshold
```

Notice we use the heuristic rather than bare Euclidean distance for spatial distance. If position is within some small tolerance, then the heuristic will return euclidean distance. Otherwise, it will return the usual min(dubins path length, obstacle-safe path length). The tolerance is chosen small enough so there is no chance of an obstacle lying in between the car and the goal.

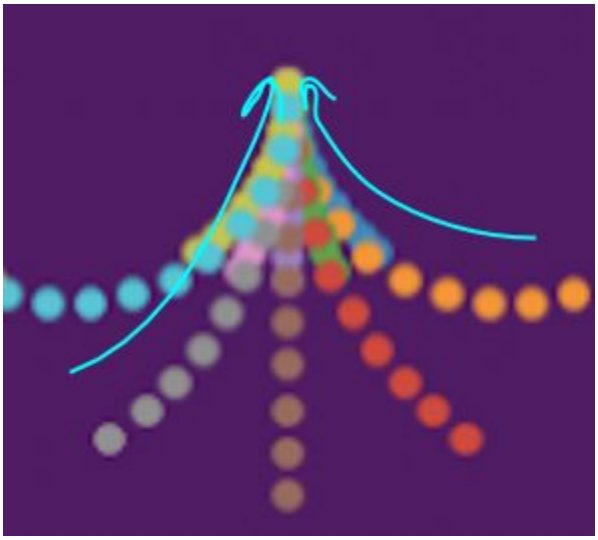
Dubin’s Path also returns extremely large path length for very similar states:



In the above figure, notice how the start and end of the dubins path arc are very close and can be assumed to be identical. Dubins path, however, would mark them as very far away from each other.

Visualization Discrepancies: Graphics Frame v.s Cartesian Frame and Backwards Search

When running the simulator’s visualization of planning, please note that we search backwards from goal to start, so it may look like one state branches out to multiple states, but it’s actually the opposite: multiple states converge to one state:



It looks like goal should face up, but theta is chosen in image-frame as is the y-coordinates, so -90 faces upwards on our screen and +90 faces down… If we choose goal with theta = +90, the first set of expansions we see seem to all face upwards, but actually +90 corresponds to facing down. The reason they "look like" they're all facing up is that we're visualizing predecessors of the goal state, or all the possible trajectories that could lead directly to the goal.