H *Graph Representation and Basic Search*

H.1 Graphs

A graph is a collection of *nodes* and *edges,* i.e., $G = (V, E)$. See figure H.1. Sometimes, another term for a node is *vertex,* and this chapter uses the two terms interchangeably. We use G for graph, V for vertex (or node), and E for edge. Typically in motion planning, a node represents a salient location, and an edge connects two nodes that correspond to locations that have an important relationship. This relationship could be that the nodes are mutually accessible from each other, two nodes are within line of sight of each other, two pixels are next to each other in a grid, etc. This relationship does not have to be mutual: if the robot can traverse from nodes V_1 to V_2, but not from V_2 to V_1, we say that the edge E_{12} connecting V_1 and V_2 is directed. Such a collection of nodes and edges is called a *directed graph*. If the robot can travel from V_1 to V_2 and vice versa, then we connect V_1 and V_2 with two directed edges E_{12} and E_{21}. If for each vertex V_i that is connected to V_j, both E_{ij} and E_{ji} exist, then instead of connecting V_i and V_j with two directed edges, we connect them with a single undirected edge. Such a graph is called an *undirected graph.* Sometimes, edges are annotated with a non-negative numerical value reflective of the costs of traversing this edge. Such values are called *weights*.

A *path* or *walk* in a graph is a sequence of nodes $\{V_i\}$ such that for adjacent nodes V_i and V_{i+1}, $E_{i\ i+1}$ exists (and thus connects V_i and V_{i+1}). A graph is *connected* if for all nodes V_i and V_j in the graph, there exists a path connecting V_i and V_j. A *cycle* is a

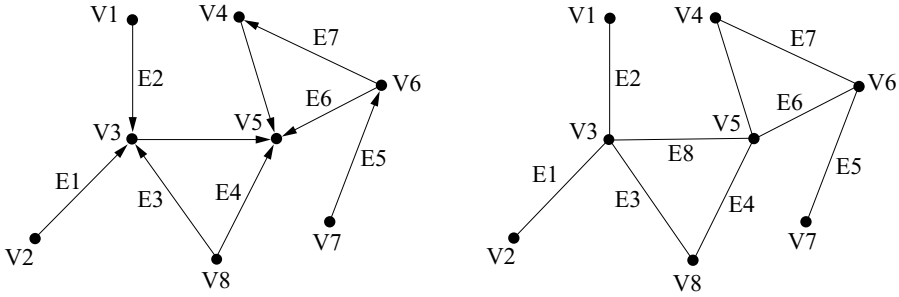


Figure H.1 A graph is a collection of nodes and edges. Edges are either directed (left) or undirected (right).

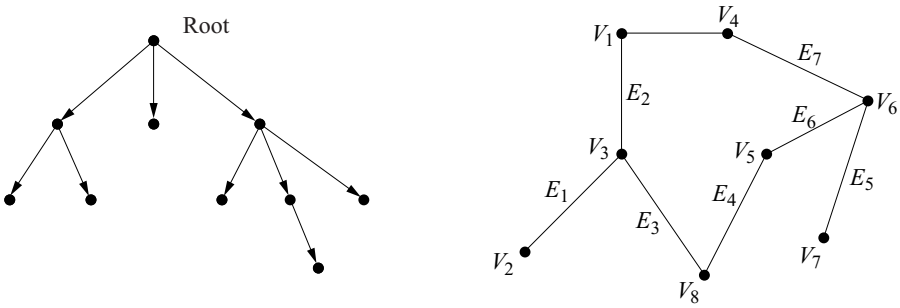


Figure H.2 A tree is a type of directed acyclic graph with a special node called the *root*. A cycle in a graph is a path through the graph that starts and ends at the same node.

path of n vertices such that first and last nodes are the same, i.e., $V_1 = V_n$ (figure H.2). Note that the "direction" of the cycle is ambiguous for undirected graphs, which in many situations is sufficient. For example, a graph embedded in the plane can have an undirected cycle which could be both clockwise and counterclockwise, whereas a directed cycle can have one orientation.

A *tree* is a connected directed graph without any cycles (figure H.2). The tree has a special node called the *root*, which is the only node that possesses no incoming arc. Using a parent-child analogy, a parent node has nodes below it called children; the root is a parent node but cannot be a child node. A node with no children is called a *leaf*. The removal of any nonleaf node breaks the connectivity of the tree.

Typically, one searches a tree for a node with some desired properties such as the goal location for the robot. A *depth-first search* starts at the root, chooses a child,

then that node's child, and so on until finding either the desired node or a leaf. If the search encounters a leaf, the search then backs up a level and then searches through an unvisited child until finding the desired node or a leaf, repeating this process until the desired node is found or all nodes are visited in the graph (figure H.3).

Breadth-first search is the opposite; the search starts at the root and then visits all of the children of the root first. Next, the search then visits all of the grandchildren, and so forth. The belief here is that the target node is near the root, so this search would require less time (figure H.3).

A grid induces a graph where each node corresponds to a pixel and an edge connects nodes of pixels that neighbor each other. Four-point connectivity will only have edges to the north, south, east, and west, whereas eight-point connectivity will have edges to all pixels surrounding the current pixel. See figure H.4.

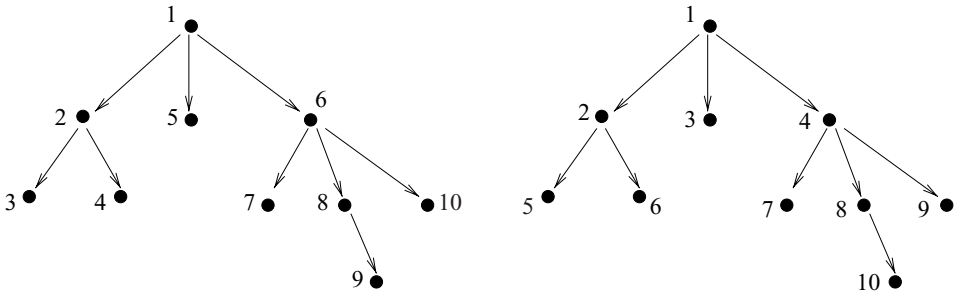


Figure H.3 Depth-first search vs. breadth-first search. The numbers on each node reflect the order in which nodes are expanded in the search.

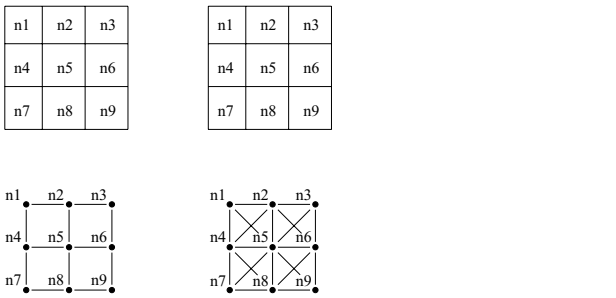


Figure H.4 Four-point connectivity assumes only four neighbors, whereas eight-point connectivity has eight.

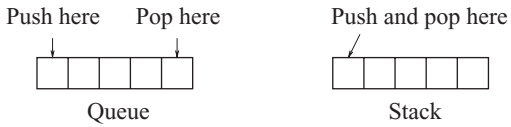


Figure H.5 Queue vs. stack.

As can be seen, the graph that represents the grid is not a tree. However, the breadth-first and depth-first search techniques still apply. Let the *link length* be the number of edges in a path of a graph. Sometimes, this is referred to as edge depth. Link length differs from path length in that the weights of the edges are ignored; only the number of edges count. For a general graph, breadth-first search considers each of the nodes that are the same link length from the start node before going onto child nodes. In contrast, depth-first search considers a child first and then continues through the children successively considering nodes of increasing link length away from the start node until it reaches a childless or already visited node (i.e., a cycle). In other words, termination of one iteration of the depth-first search occurs when a node has no unvisited children.

The wave-front planner (chapter 4, section 4.5) is an implementation of a breadth-first search. Breadth-first search, in general, is implemented with a list where the children of the current node are placed into the list in a first-in, first-out (FIFO) manner. This construction is commonly called a *queue* and forces all nodes of the same linklength from the start to be visited first (figure H.5). The breadth-first search starts with placing the start node in the queue. This node is then *expanded* by it being popped off (i.e., removed from the front) the queue and all of its children being placed onto it. This procedure is then repeated until the goal node is found or until there are no more nodes to expand, at which time the queue is empty. Here, we expand all nodes of the same level (i.e., link length from the start) first before expanding more deeply into the graph.

Figure H.6 displays the resulting path of breadth-first search. Note that all paths produced by breadth-first search in a grid with eight-point connectivity are optimal with respect to the "eight-point connectivity metric." Figure H.7 displays the link lengths for all shortest paths between each pixel and the start pixel in the free space in Figure H.6. A path can then be determined using this information via a gradient descent of link length from the goal pixel to the start through the graph as similarly done with the wavefront algorithm.

Depth-first search contrasts breadth-first search in that nodes are placed in a list in a last-in, first-out (LIFO) manner. This construction is commonly called a *stack* and forces nodes that are of greater and greater link length from the start node to be

Figure H.6 White pixels denote the path that was determined with breadth-first search.

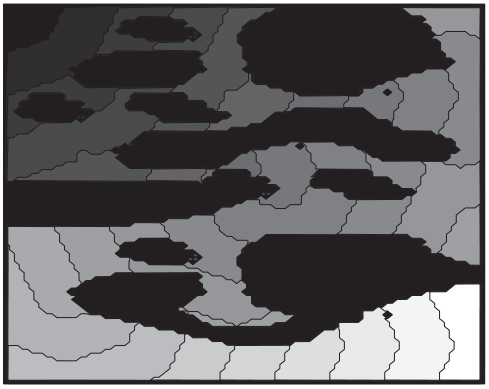


Figure H.7 A plot of linklength values from the start (upper left corner) node where colored pixels correspond to link length (where the lighter the pixel the greater the linklength in the graph) and black pixels correspond to obstacles.

visited first. Now the expansion procedure sounds the same but is a little bit different; here, we pop the stack and push all of its children onto the stack, except popping and pushing occur on the same side of the list (figure H.5). Again, this procedure is repeated until the goal node is found or there are no more nodes to expand. Here, we expand nodes in a path as deep as possible before going onto a different path.

Figure H.8 displays the resulting path of depth-first search. In this example, depth-first search did not return an optimal path but it afforded a more efficient search

Figure H.8 White pixels denote the path that was determined with depth-first search.

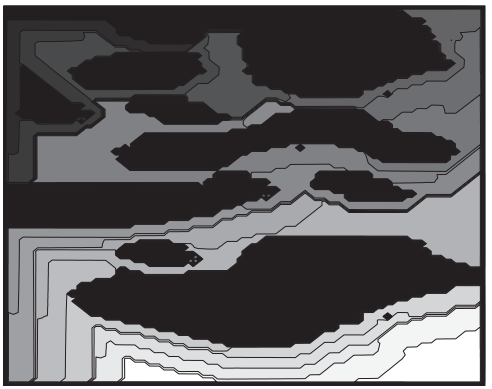


Figure H.9 A plot of linklength values from the start (upper left corner) node where colored pixels correspond to link lengths of paths defined by the depth-first search. The lighter the pixel the greater the linklengths in the graph; black pixels correspond to obstacles.

in that the goal was found more quickly than breadth-first search. Figure H.9 is similar to figure H.7, except the link lengths here do *not* correspond to the shortest path to the start; instead, the link lengths correspond to the paths derived by the depth-first search. Again, we can use a depth-first search algorithm to fill up such a map and then determine a path via gradient descent from the goal pixel to the start.

Another common search is called a *greedy search* which expands nodes that are closest to the goal. Here, the data structure is called a *priority queue* in that nodes are

placed into a sorted list based on a priority value. This priority value is a heuristic that measures distance to the goal node.

H.2 A^* Algorithm

Breadth-first search produces the shortest path to the start node in terms of link lengths. Since the wave-front planner is a breadth-first search, a four-point connectivity wave-front algorithm produces the shortest path with respect to the Manhattan distance function. This is because it implicitly has an underlying graph where each node corresponds to a pixel and neighboring pixels have an edge length of one. However, shortest-path length is not the only metric we may want to optimize. We can tune our graph search to find optimal paths with respect to other metrics such as energy, time, traversability, safety, etc., as well as combinations of them.

When speaking of graph search, there is another opportunity for optimization: minimize the number of nodes that have to be visited to locate the goal node subject to our path-optimality criteria. To distinguish between these forms of optimality, let us reserve the term *optimality* to measure the path and *efficiency* to measure the search, i.e., the number of nodes visited to determine the path. There is no reason to expect depth-first and breadth-first search to be efficient, even though breadth-first search can produce an optimal path.

Depth-first and breadth-first search in a sense are uninformed, in that the search just moves through the graph without any preference for or influence on where the goal node is located. For example, if the coordinates of the goal node are known, then a graph search can use this information to help decide which nodes in the graph to visit (i.e., expand) to locate the goal node.

Alas, although we may have some information about the goal node, the best we can do is define a *heuristic* which hypothesizes an expected, but not necessarily actual, cost to the goal node. For example, a graph search may choose as its next node to explore one that has the shortest Euclidean distance to the goal because such a node has highest possibility, based on local information, of getting closest to the goal. However, there is no guarantee that this node will lead to the (globally) shortest path in the graph to the goal. This is just a good guess. However, these good guesses are based on the best information available to the search.

The A^* algorithm searches a graph efficiently, with respect to a chosen heuristic. If the heuristic is "good," then the search is efficient; if the heuristic is "bad," although a path will be found, its search will take more time than probably required and possibly return a suboptimal path. A^* will produce an optimal path if its heuristic is *optimistic*.

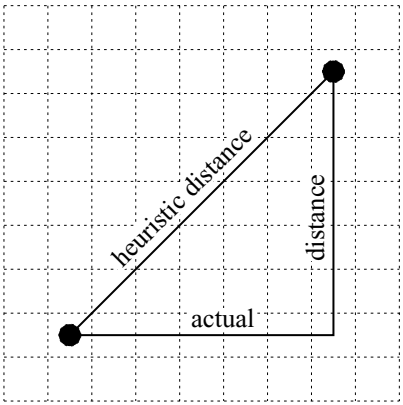


Figure H.10 The heuristic between two nodes is the Euclidean distance, which is less than the actual path length in the grid, making this heuristic optimistic.

An optimistic, or admissible, heuristic always returns a value less than or equal to the cost of the shortest path from the current node to the goal node within the graph. For example, if a graph represented a grid, an optimistic heuristic could be the Euclidean distance to the goal because the L^2 distance is always less than or equal to the L^1 distance in the plane (figure H.10).

First, we will explain the A^* search via example and then formally introduce the algorithm. See figure H.11 for a sample graph. The A^* search has a *priority queue* which contains a list of nodes sorted by priority, which is determined by the sum of the distance traveled in the graph thus far from the start node, and the heuristic.

The first node to be put into the priority queue is naturally the start node. Next, we *expand* the start node by popping the start node and putting all adjacent nodes to the start node into the priority queue sorted by their corresponding priorities. Since node B has the greatest priority, it is expanded next, i.e., it is popped from the queue and its neighbors are added (figure H.12). Note that only unvisited nodes are added to the priority queue, i.e., do not re-add the start node.

Now, we expand node H because it has the highest priority. It is popped off of the queue and all of its neighbors are added. However, H has no neighbors, so nothing is added to the queue. Since no new nodes are added, no more action or expansion will be associated with node H (figure H.12). Next, we pop off the node with greatest priority, i.e., node A, and expand it, adding all of its adjacent neighbors to the priority queue (figure H.12).

Next, node E is expanded which gives us a path to the goal of cost 5. Note that this cost is the real cost, i.e., the sum of the edge costs to the goal. At this point, there are

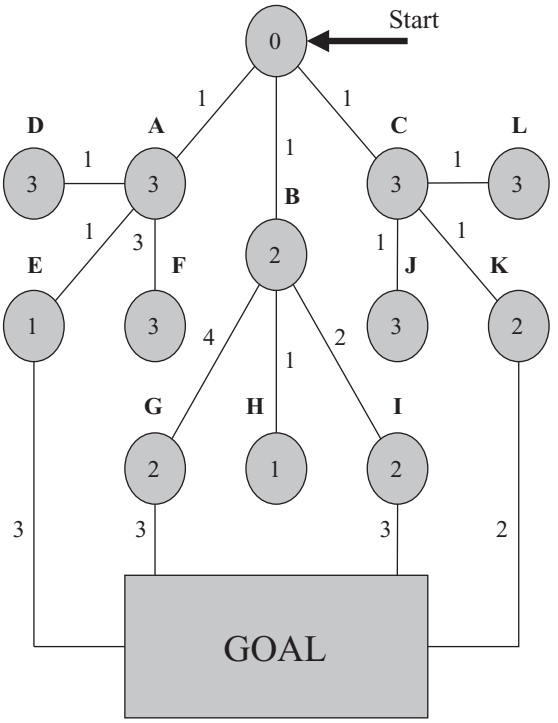


Figure H.11 Sample graph where each node is labeled by a letter and has an associated heuristic value which is contained inside the node icon. Edge costs are represented by numbers adjacent to the edges and the start and goal nodes are labeled. We label the start node with a zero to emphasize that it has the highest priority at first.

nodes in the priority queue which have a priority value greater than the cost to the goal. Since these priority values are lower bounds on path cost to the goal, all paths through these nodes will have a higher cost than the cost of the path already found. Therefore, these nodes can be discarded (figure H.12).

The explicit path through the graph is represented by a series of *back pointers.* A back pointer represents the immediate history of the expansion process. So, the back pointers from nodes A, B, and C all point to the start. Likewise, the back pointers to D, E, and F point to A. Finally, the back pointer of goal points to E. Therefore, the path defined with the back pointers is start, A, E, and goal. The arrows in figure H.12 point in the reverse direction of the back pointers.

Even though a path to the goal has been determined, A^* is not finished because there could be a better path. A^* knows this is possible because the priority queue

Figure H.12 (Left) Priority queue after the start is expanded. (Middle) Priority queue after the second node, B, is expanded. (Right) Three iterations of the priority queue are displayed. Each arrow points from the expanded node to the nodes that were added in each step. Since node H had no unvisited adjacent cells, its arrow points to nothing. The middle queue corresponds to two actions. Node E points to the goal which provides the first candidate path to the goal. Note that nodes D, I, F, and G are shaded out because they were discarded.

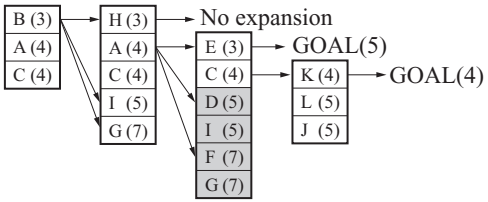


Figure H.13 Four displayed iterations of the priority queue with arrows representing the history of individual expansions. Here, the path to the goal is start, C, K, goal.

still contains nodes whose value is smaller than that of the goal state. The priority queue at this point just contains node C and is then expanded adding nodes J, K, and L to the priority queue. We can immediately remove J and L because their priority values are greater than or equal the cost of the shortest path found thus far. Node K is then expanded finding the goal with a path cost shorter than the previously found path through node E. This path becomes the current best path. Since at this point the priority queue does not possess any elements whose value is smaller than that of the goal node, this path results in the best path (figure H.13).

H.2.1 Basic Notation and Assumptions

Now, we can more formally define the A^* algorithm. The input for A^* is the graph itself. These nodes can naturally be embedded into the robot's free space and thus can have coordinates. Edges correspond to adjacent nodes and have values corresponding to the cost required to traverse between the adjacent nodes. The output of the A^*

algorithm is a back-pointer path, which is a sequence of nodes starting from the goal and going back to the start.

We will use two additional data structures, an open set O and a closed set C. The open set O is the priority queue and the closed set C contains all processed nodes. Other notation includes

- Star(n) represents the set of nodes which are adjacent to n.
- $c(n_1, n_2)$ is the length of edge connecting n_1 and n_2.
- $g(n)$ is the total length of a backpointer path from n to q_{start}.
- $h(n)$ is the heuristic cost function, which returns the estimated cost of shortest path from n to q_{goal}.
- $f(n) = g(n) + h(n)$ is the estimated cost of shortest path from q_{start} to q_{goal} via n.

The algorithm can be found in algorithm 24.

H.2.2 Discussion: Completeness, Efficiency, and Optimality

Here is an informal proof of completeness for A^*. A^* generates a search tree, which by definition, has no cycles. Furthermore, there are a finite number of acyclic paths in the tree, assuming a bounded world. Since A^* uses a tree, it only considers acyclic paths. Since the number of acyclic paths is finite, the most work that can be done,

Algorithm 24 A^* Algorithm

Input: A graph

Output: A path between start and goal nodes

1: **repeat**
2: Pick n_{best} from O such that $f(n_{best}) \leq f(n), \forall n \in O$.
3: Remove n_{best} from O and add to C.
4: If $n_{best} = q_{goal}$, EXIT.
5: Expand n_{best}: for all $x \in$ Star(n_{best}) that are not in C.
6: **if** $x \notin O$ **then**
7: add x to O.
8: **else if** $g(n_{best}) + c(n_{best}, x) < g(x)$ **then**
9: update x's backpointer to point to n_{best}
10: **end if**
11: **until** O is empty

searching all acyclic paths, is also finite. Therefore A^* will always terminate, ensuring completeness.

This is not to say A^* will always search all acyclic paths since it can terminate as soon as it explores all paths with greater cost than the minimum goal cost found. Thanks to the priority queue, A^* explores paths likely to reach the goal quickly first. By doing so, it is efficient. If A^* does search every acyclic path and does not find the goal, the algorithm still terminates and simply returns that a path does not exist. Of course, this also makes sense if every possible path is searched.

Now, there is no guarantee that the first path to the goal found is the cheapest/best path. So, in quest for optimality (once again, with respect to the defined metric), all branches must be explored to the extent that a branch's terminating node cost (sum of edge costs) is greater than the lowest goal cost. Effectively, all paths with overall cost lower than the goal must be explored to guarantee that an even shorter one does not exist. Therefore, A^* is also optimal (with respect to the chosen metric).

H.2.3 Greedy-Search and Dijkstra's Algorithm

There are variations or special cases of A^*. When $f(n) = h(n)$, then the search becomes a *greedy* search because the search is only considering what it "believes" is the best path to the goal from the current node. When $f(n) = g(n)$, the planner is not using any heuristic information but rather growing a path that is shortest from the start until it encounters the goal. This is a classic search called *Dijkstra's algorithm.* Figure H.14 contains a graph which demonstrates Dijkstra's Algorithm. In this example, we also show backpointers being updated (which can also occur with A^*). The following lists the open and closed sets for the Dijkstra search in each step.

1. $O = \{S\}$

2. $O = \{2, 4, 1, 5\}$; $C = \{S\}$ (1, 2, 4, 5 all point back to S)

3. $O = \{4, 1, 5\}$; $C = \{S, 2\}$ (there are no adjacent nodes not in C)

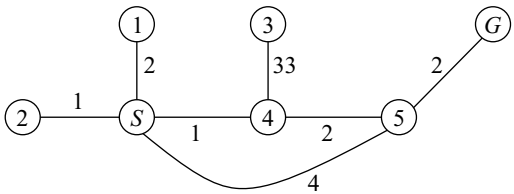


Figure H.14 Dijkstra graph search example.

4. O = {1,5,3}; C = {S, 2, 4} (1, 2, 4 point to *S*; *5 points to 4*)

5. O = {5,3}; C = {S, 2, 4, 1}

6. O = {3, G}; C = {S, 2, 4, 1} (goal points to 5 which points to 4 which points to *S*)

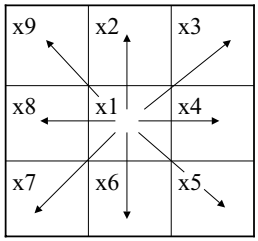
H.2.4 Example of A^* on a Grid

Figure H.15 contains an example of a grid world with a start and a goal identified accordingly. We will assume that the free space uses eight-point connectivity, and thus cell (3, 2) is adjacent to cell (4, 3), i.e., the robot can travel from (3, 2) to (4, 3). Each of the cells also has its heuristic distance to the goal where we use a modified metric which is not the Manhattan or the Euclidean distance. Instead, between free space pixels, a vertical or horizontal step has length 1 and a diagonal has length 1.4 (our approximation of $\sqrt{2}$). The cost of traveling from a free space pixel to an obstacle pixel is made to be arbitrarily high; we chose 10000. So one pixel step from a free space to an obstacle pixel along a vertical or horizontal direction costs 10000 and one pixel step along a diagonal direction costs 10000.4. Here, we are assuming that our graph connects *all* cells in the grid, not just the free space, and the prohibitively high cost of moving into an obstacle will prevent the robot from collision (figure H.16).

Note that this metric, in the free space, does not induce a true Euclidean metric because two cells sideways and one cell up is 2.4, not $\sqrt{5}$. However, this metric is quite representative of path length within the grid. This heuristic is optimistic because the actual cost to current cell to the goal will always be greater than or equal

6	h =6 f = b=()	h =5 f = b=( )	h =4 f = b=()	h =3 f = b=()	h =2 f = b=()	h =1 f = b=()	h =0 f = Goal
5	h =6.4 f = b=()	h =5.4 f = b=()	h =4.4 f = b=()	h =3.4 f = b=()	h =2.4 f = b=()	h =1.4 f = b=()	h =1 f = b=()
4	h =6.8 f = b=()	h =5.8 f = b=()	h =4.8 f = b=()	h =3.8 f = b=()	h =2.8 f = b=()	h =2.4 f = b=()	h =2 f = b=()
3	h =7.2 f = b=()	h =6.2 f = b=()	h =5.2 f = b=()	h =4.2 f = b=()	h =3.8 f = b=()	h =3.4 f = b=()	h =3 f = b=()
2	h =7.6 f = b=()	h =6.6 f = b=()	h =5.6 f = b=()	h =5.2 f = b=()	h =4.8 f = b=()	h =4.4 f = b=()	h =4 f = b=()
1	h =8.0 f = b=()	h =7.0 f = Start	h =6.6 f = b=()	h=6.2 f= b=()	h =5.8 f = b=()	h =5.4 f = b=()	h =5 f = b=()
r/c	1	2	3	4	5	6	7

Figure H.15 Heuristic values are set, but backpointers and priorities have not.



c(x1, x2) = 1
c(x1, x9) = 1.4

c(x1, x8) = 10000, if x8 is in obstacle, x1 is a free cell

c(x1,x9) = 10000.4, if x9 is in obstacle, x1 is a free cell

Figure H.16 Eight-point connectivity and possible cost values.

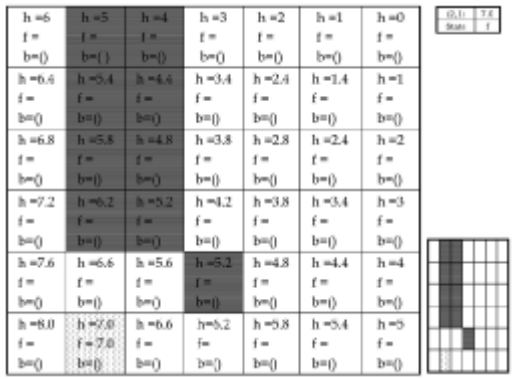


Figure H.17 Start node is put on priority queue, displayed in upper right.

to the heuristic. Thus far, in figure H.15 the back pointers and priorities have not been set.

The start pixel is put on the priority queue with a priority equal to its heuristic. See figure H.17. Next, the start node is expanded and the priority values for each of the start's neighbors are determined. They are all put on the priority queue sorted in ascending order by priority. See figure H.18(left). Cell (3, 2) is expanded next, as depicted in figure H.18(right). Here, cells (4, 1), (4, 2), (4, 3), (3, 3), and (2, 3) are added onto the priority queue because our graph representation of the grid includes both free space and obstacle pixels. However, cells (4, 2), (3, 3), and (2, 3) correspond to obstacles and thus have a high cost. If a path exists in the free space or the longest path in the free space has a traversal cost less than our arbitrarily high number chosen for obstacles (figure H.16), then these pixels will never be expanded. Therefore, in the figures below, we did not display them on the priority queue.

Eventually, the goal cell is reached (figure H.19 (left)). Since the priority value of the goal is less than the priorities of all other cells in the priority queue, the resulting

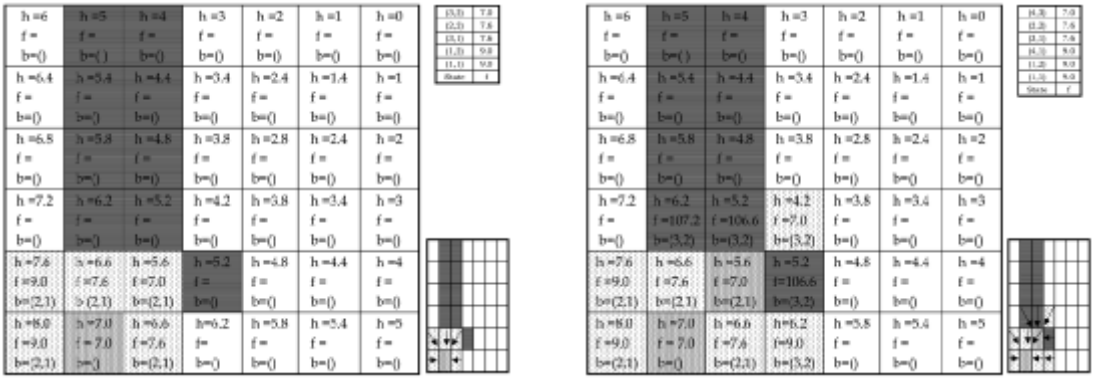


Figure H.18 (Left) The start node is expanded, the priority queue is updated, and the backpointers are set, which are represented by the right bottom icon. $b = (i, j)$ points to cell (i, j). (Right) Cell (3, 2) was expanded. Note that pixels (3, 3), (2, 3), and (4, 2) are not displayed in the priority queue because they correspond to obstacles.

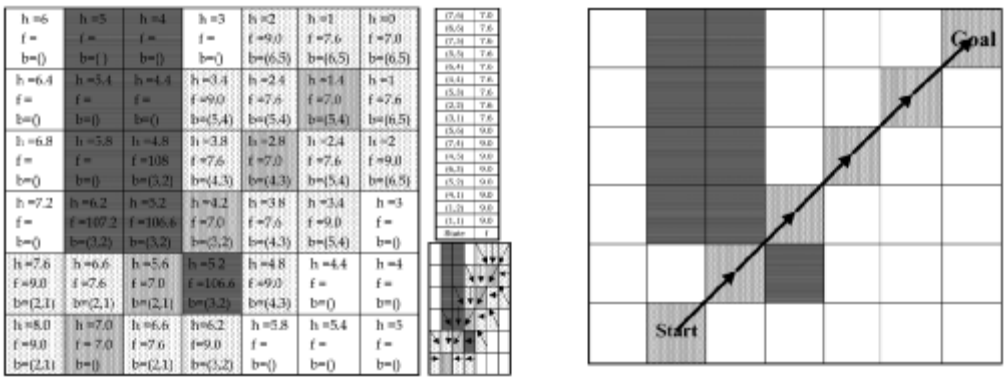


Figure H.19 (Left) The goal state is expanded. (Right) Resulting path.

path is optimal and A^* terminates. A^* traces the backpointers to find the optimal path from start to goal (figure H.19 (right)).

H.2.5 Nonoptimistic Example

Figure H.20 contains an example of a graph whose heuristic values are nonoptimistic and thus force A^* to produce a nonoptimal path. A^* puts node S on the priority queue and then expands it. Next, A^* expands node A because its priority value is 7. The goal

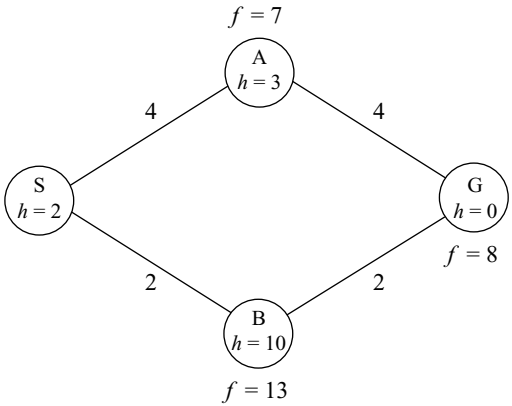


Figure H.20 A nonoptimistic heuristic leads to a nonoptimal path with A^*.

node is then reached with priority value 8, which is still less than node B's priority value of 13. At this point, node B will be eliminated from the priority queue because its value is greater than the goal's priority value. However, the optimal path passes through B, not A. Here, the heuristic is not optimistic because from B to G, $h = 10$ when the actual edge length was 2.

H.3 D^* Algorithm

So far we have only considered *static* environments where only the robot experiences motion. However, we can see that many worlds have moving obstacles, which could be other robots themselves. We term such environments *dynamic*. There are three types of dynamic obstacles: ones that move significantly slower than the robot, those that move at the same speed, and finally obstacles that move much faster than the robot. The superfast obstacle case is easy to ignore because the obstacles will be moving so fast that there probably is no need to plan for them because they will either move too fast for the planner to have time to account for them or they will be in and out of the robot's path so quickly that it does not require any consideration. In this section, we consider dynamic environments where the world changes at a speed much slower than the robot. An example can be a door opening and closing.

Consider the grid environment in figure H.21(left) which is identical to the one in figure H.15, except pixel (4, 3) is a gate which can either be a free-space pixel or an obstacle pixel. Let's assume it starts as a free-space pixel. We can run the A^* or

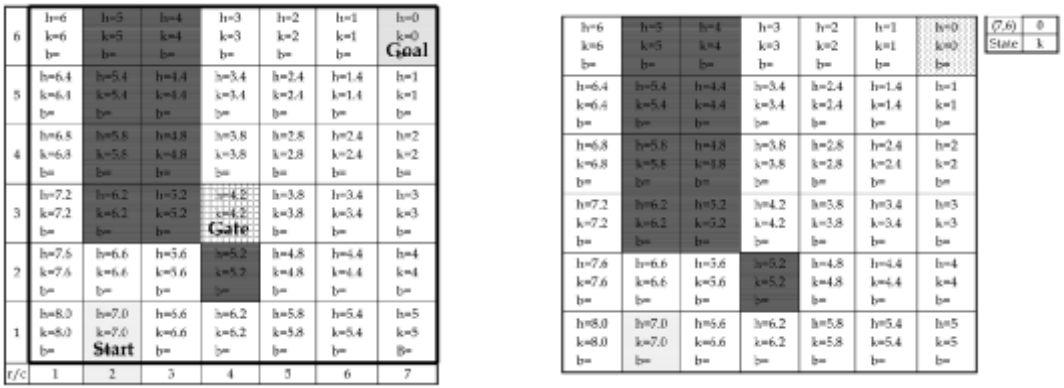


Figure H.21 (Left) A pixel world similar to figure H.15, except it has a gate, heuristic, and minimum heuristic values. (Right) Goal node is expanded.

Dijkstra's algorithm to determine a path from start to goal, and then follow that path until an unexpected change occurs, which in figure H.21(left) happens at (4, 3). When the robot encounters pixel (4, 3) and determines that it changed from a free-space to an obstacle pixel, it can simply reinvoke the A^* algorithm to determine a new path. This, however, can become quite inefficient if many pixels are changing from obstacle to free space and back. The D^* algorithm was devised to "locally repair" the graph allowing for an efficient updated searching in dynamic environments, hence the term D^* [397].

D^* initially determines a path starting with the goal and working back to the start using a slightly modified Dijkstra's search. The modification involves updating a heuristic and a minimum heuristic function. Each cell in figure H.21(left) contains a heuristic cost (h) which for D^* is an *estimate* of path length from the particular cell to the goal, not necessarily the shortest path length to the goal as it was for A^*. In this example, the h values do not respect the presence of obstacles when reflecting distance to the goal node; in other words, computation of h assumes that the robot can pass through obstacles. For example, cell (1, 6) has an h value of 6. These h values will be updated during the initial Dijkstra search to reflect the existence of obstacles. The minimum heuristic values (k) are the *estimate* of the *shortest* path length to the goal. Both the h and the k values will vary as the D^* search runs, but they are equal upon initialization, and were derived from the metric described in figure H.16.

Initially, the goal node is placed on the queue with $h = 0$ and then is expanded (figure H.21, right), adding (6, 6), (6, 5), and (7, 5) onto the queue. Next, pixel (6, 6) is expanded adding cells (5, 6), (5, 5) onto the queue. Note that the k values are used

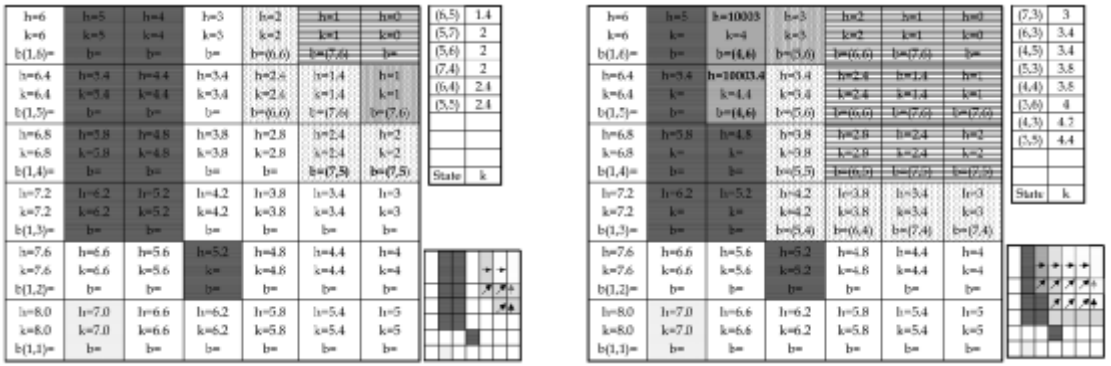


Figure H.22 (Left) First (6, 6) and then (7, 5) is expanded. (Right) The h values in obstacle cells that are put on priority queue are updated.

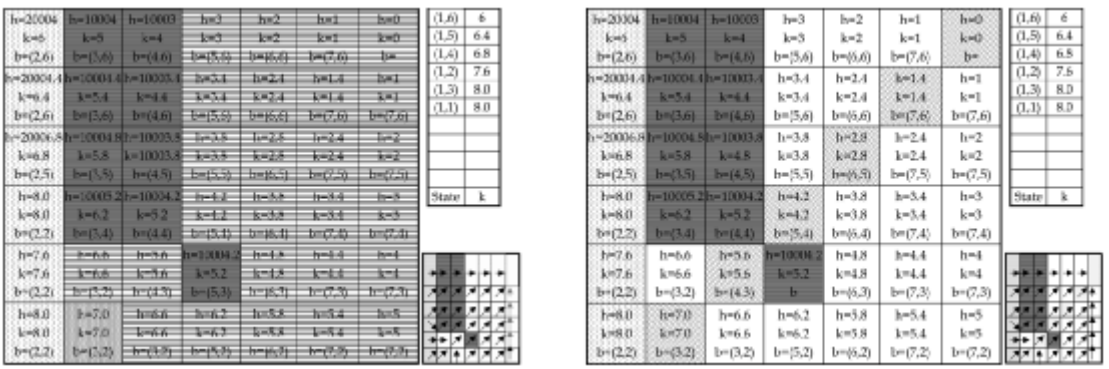


Figure H.23 (Left) Termination of Dijkstra's search phase: start cell is expanded. (Right) Tracing backpointers yields the optimal path.

to determine the priority for the Dijkstra's search (and later on for the D^* search) and that they are equal to the h values for the initial Dijkstra's search.

Next, pixel (7, 5) is expanded adding cells (6, 4) and (7, 4) onto the queue (figure H.22, left). More pixels are expanded until we arrive at pixel (4, 6) (figure H.22, right). When (4, 6) is expanded, pixels (3, 6) and (3, 5), which are obstacle pixels, are placed onto the priority queue. Unlike our A^* example, we display these obstacle pixels in the priority queue in figure H.22(right). Note that the h values of the expanded obstacle pixels are all updated to prohibitively high values which reflects the fact that they lie on obstacles.

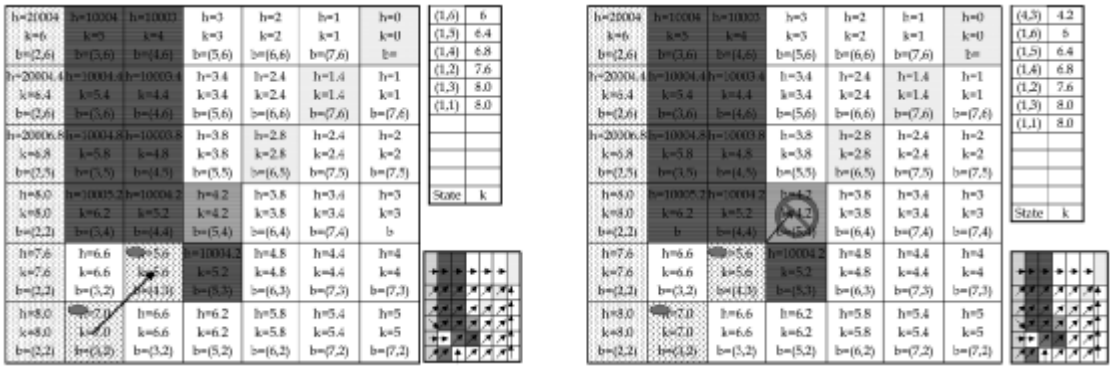


Figure H.24 (Left) The robot physically starts tracing the optimal path. (Right) The robot cannot trace the assumed optimal path: gate (4, 3) is closed.

The Dijkstra's search is complete when the start node (2, 1) is expanded (figure H.23, left). The optimal path from start to goal (assuming that the gate pixel (4, 3) is open) is found by traversing the backpointers starting from the start node to the goal node (figure H.23(right)). The optimal path is (2, 1) $\longrightarrow$ (3, 2) $\longrightarrow$ (4, 3) $\longrightarrow$ (5, 4) $\longrightarrow$ (6, 5) $\longrightarrow$ (7, 6). Note that pixels (1, 1), (1, 2), (1, 3), (1, 4), and (1, 6) are still on the priority queue.

The robot then starts tracing the optimal path from the start pixel to the goal pixel. In figure H.24(left), the robot moves from pixel (2, 1) to (3, 2). When the robot tries to move from pixel (3, 2) to (4, 3), it finds that the gate pixel (4, 3) is closed (figure H.24, left). In the initial search for an optimal path, we had assumed that the gate pixel was open, and hence the current path may not be feasible. At this stage, instead of replanning for an optimal path from the current pixel (3, 2) to goal pixel using A^*, D^* tries to make local changes to the optimal path.

D^* puts the pixel (4, 3) on the priority queue because it corresponds to a discrepancy between the map and the actual environment. Note that this pixel must have the lowest minimum heuristic, i.e., k value, because all other pixels on the priority queue have a k value greater than or equal to the start and all pixels along the previously determined optimal path have a k value less than the start. The idea here is to put the changed pixel onto the priority queue and then expand it again, thereby propagating the possible changes in the heuristic, i.e., the h values, to pixels for which an optimal path to the goal passes through the changed pixel.

In the current example, pixel (4, 3) is expanded, i.e., it is popped off the priority queue and pixels whose optimal paths pass through (4, 3) are placed onto the priority

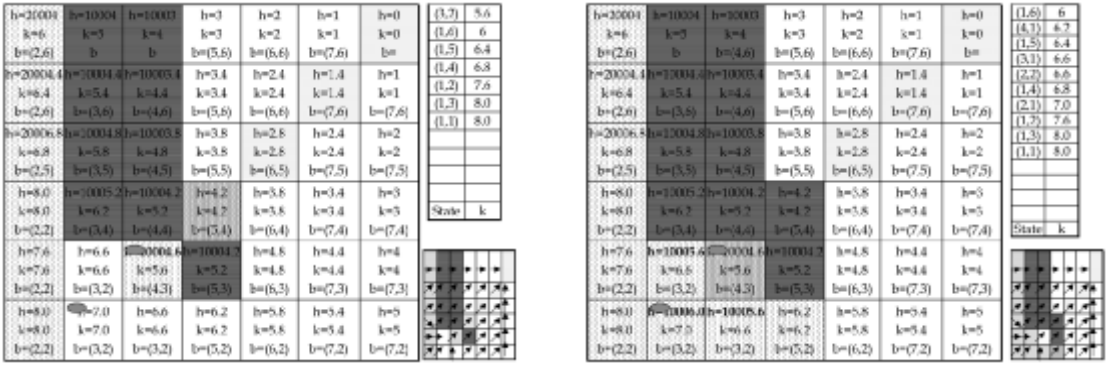


Figure H.25 (Left) The gate pixel (4, 3) is put on priority queue and expanded. The assumed optimal path from (3, 2) to goal passed through (4, 3), so the h value is increased to a high value to reflect that the assumed optimal path may not in fact be optimal. (Right) Pixel (3, 2) is expanded; the h values of (2, 2),(2, 1) and (3, 1) are updated because the assumed optimal path from these cells passed through the expanded cell. (4, 1)) remains unaffected.

queue with updated heuristic values (h values). The new h values are the h values of the changed pixel plus the path cost from the changed pixel to the given pixel. This path cost is a high number which we set to 10000.4 if it passes diagonally through an obstacle pixel. Therefore, pixel (3, 2) has an h value equal to 10004.6 (figure H.25, left).

Next, pixel (3, 2) is expanded because its k value is the smallest. However, its k value is less than its h value and we term such pixels as having a *raised state.* When a pixel is in a raised state, its back pointer may no longer point to an optimal path. Now, pixels (2, 2), (2, 1), (3, 1), and (4, 1) are on the priority queue. The h values of cells (2, 2), (2, 1), and (3, 1) are updated to high values to reflect that the estimated optimal path from these cells to the goal passed through the gate cell, and may not be optimal anymore. However, the optimal path for pixel (4, 1) did not pass through the gate, and hence its h value stays the same (figure H.25, right).

Pixel (1, 6) is expanded next, but it does not affect the h values of its neighbors. Next pixel (4, 1) is expanded and pixels (3, 2), (3, 1), (5, 1), and (5, 2) are put onto the priority queue (figure H.26(left). Now the h values of (5, 1) and (5, 2) remain unaffected, however, for cell (3, 2), the goal can now possibly be reached in $h(4, 1) + 1.4 = 6.2 + 1.4 = 7.6$ because cell (4, 1) is in a *lowered state,* i.e., it is a cell whose h values did not have to be updated because of the gate. Therefore, cell (3, 2) receives an h value of 7.6. The backpointer of (3, 2) is now set pointing toward (4, 1). The initially determined optimal path from (4, 1) to the goal did not pass through the

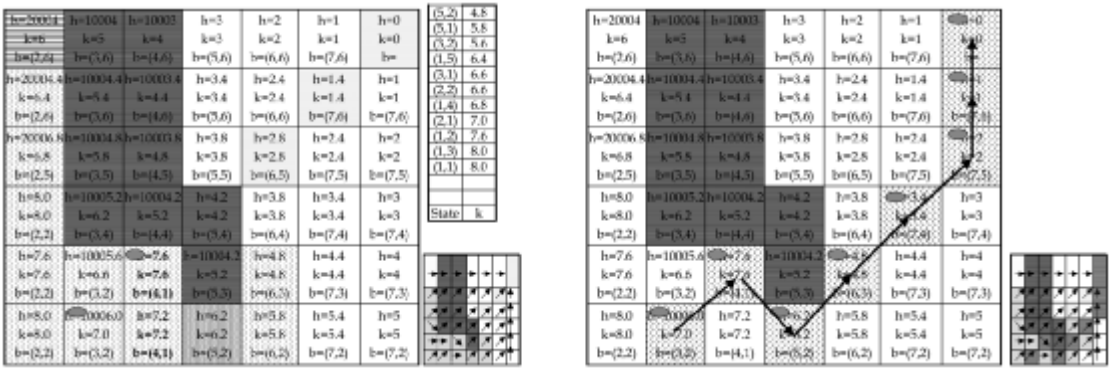


Figure H.26 (Left) (4, 1) is expanded; (5, 1) and (5, 2) remain unaffected, the h values of (3, 2) and (3, 1) are lowered to reflect the lowered heuristic value because of detour, while the minimum-heuristic values of these two cells are increased, and the backpointers of these cells are set pointing to (4, 1). (Right) The robot traces the new locally modified optimal path.

gate pixel, and hence it indeed is optimal even after the change of state of the gate pixel. Then, the path obtained by concatenating the (3, 2) $\rightarrow$ (4, 1) transition and the optimal path from (4, 1) will be optimal for (3, 2). Thus, the estimate for the best path from (3, 2) toward the goal, i.e., $k(3, 2)$, is now 7.6, and the process terminates. The robot then physically traces the new optimal path from (3, 2) to reach the goal (figure H.26(right)).

See algorithms 25–31 for a description of the D^* algorithm. This algorithm uses the following notation.

- X represents a state.
- O is the priority queue.
- L is the list of all states.
- G is the goal state.
- S is the start state.
- $t(X)$ is value of state with regards to the priority queue.
  - $t(X) = NEW$, if X has never been in O,
  - $t(X) = OPEN$, if X is currently in O, and
  - $t(X) = CLOSED$, if X was in O but currently is not.

Algorithm 25 D^* Algorithm

Input: List of all states L

Output: The goal state, if it is reachable, and the list of states L are updated so that the backpointer list describes a path from the start to the goal. If the goal state is not reachable, return NULL.

```
1: for each X ∈ L do
2:    t(X) = NEW
3: end for
4: h(G) = 0
5: O = {G}
6: X_c = S
   {The following loop is Dijkstra's search for an initial path}
7: repeat
8:    k_min = PROCESS − STATE(O, L)
9: until (k_min = −1) or (t(X_c) = CLOSED)
10: P = GET − BACKPOINTER − LIST(L, X_c, G) (algorithm 26)
11: if P = NULL then
12:    Return (NULL)
13: end if
14: repeat
15:    for each neighbor Y ∈ L of X_c do
16:       if r(X_c, Y) ≠ c(X_c, Y) then
17:          MODIFY − COST(O, X_c, Y, r(X_c, Y))
18:          repeat
19:             k_min = PROCESS − STATE(O, L)
20:          until (k_min ≥ h(X_c)) or (k_min = −1)
21:          P = GET − BACKPOINTER − LIST(L, X_c, G)
22:          if P = NULL then
23:             Return (NULL)
24:          end if
25:       end if
26:    end for
27:    X_c = the second element of P {Move to the next state in P}.
28:    P = GET − BACKPOINTER − LIST(L, X_c, G)
29: until X_c = G
30: Return (X_c)
```

Algorithm 26 $GET - BACKPOINTER - LIST(L, S, G)$

Input: A list of states L and two states (start and goal)

Output: A list of states from start to goal as described by the backpointers in the list of states L

if path exists **then**
  Return (The list of states)
else
  Return ($NULL$)
end if

Algorithm 27 $INSERT(O, X, h_{new})$

Input: Open list, a state, and an h-value

Output: Open list is modified

if $t(X) = NEW$ **then**
  $k(X) = h_{new}$
else if $t(X) = OPEN$ **then**
  $k(X) = \min(k(X), h_{new})$
else if $t(X) = CLOSED$ **then**
  $k(X) = \min(h(X), h_{new})$
end if
$h(X) = h_{new}$
$t(X) = OPEN$
Sort O based on increasing k values

- $c(X, Y)$ is the estimated path length between adjacent states X and Y.
- $h(X)$ is the estimated cost of a path from X to Goal (heuristic).
- $k(X)$ is the estimated cost of a shortest path from X to Goal (minimum-heuristic $= \min h(X)$ before X is put on O, values $h(X)$ takes after X is put on O).
- $b(X) = Y$ implies that Y is a parent state of X, i.e. the path is like $X \longrightarrow Y \longrightarrow G$.
- $r(X, Y)$ is the measured distance adjacent states X and Y.

Algorithm 28 $MODIFY-COST(O, X, Y, cval)$

Input: The open list, two states and a value

Output: A k-value and the open list gets updated

1: $c(X, Y) = cval$
2: **if** $t(X) = CLOSED$ **then**
3: $INSERT(O, X, h(X))$
4: **end if**
5: Return $GET-KMIN(O)$ (algorithm 30)

Algorithm 29 $MIN-STATE(O)$

Input: The open list O

Output: The state with minimum k value in the list related values

1: **if** $O = \emptyset$ **then**
2: Return (-1)
3: **else**
4: Return $(\text{argmin}_{Y \in O} k(Y))$
5: **end if**

Algorithm 30 $GET-KMIN(O)$

Input: The open list O

Output: Lowest k-value of all states in the open list

1: **if** $O = \emptyset$ **then**
2: Return (-1)
3: **else**
4: Return $(\min_{Y \in O} k(Y))$
5: **end if**

Algorithm 31 PROCESS-STATE

Input: List of all states L and the list of all states that are open O

Output: A k_{min}, an updated list of all states, and an updated open list

1: $X = MIN - STATE(O)$ (algorithm 29)
2: **if** $X = NULL$ **then**
3: Return (-1)
4: **end if**
5: $k_{old} = GET - KMIN(O)$ (algorithm 30)
6: $DELETE(X)$
7: **if** $k_{old} < h(X)$ **then**
8: **for each** neighbor $Y \in L$ of X **do**
9: **if** $h(Y) \leq k_{old}$ and $h(X) > h(Y) + c(Y, X)$ **then**
10: $b(X) = Y$
11: $h(X) = h(Y) + c(Y, X)$;
12: **end if**
13: **end for**
14: **else if** $k_{old} = h(X)$ **then**
15: **for each** neighbor $Y \in L$ of X **do**
16: **if** $(t(Y) = NEW)$ or $(b(Y) = X$ and $h(Y) \neq h(X) + c(X, Y))$ or $(b(Y) \neq X$ and $h(Y) > h(X) + c(X, Y))$ **then**
17: $b(Y) = X$
18: $INSERT(O, Y, h(X) + c(X, Y))$ (algorithm 27)
19: **end if**
20: **end for**
21: **else**
22: **for each** neighbor $Y \in L$ of X **do**
23: **if** $(t(Y) = NEW)$ or $(b(Y) = X$ and $h(Y) \neq h(X) + c(X, Y))$ **then**
24: $b(Y) = X$
25: $INSERT(O, Y, h(X) + c(X, Y))$
26: **else if** $b(Y) \neq X$ and $h(Y) > h(X) + c(X, Y)$ **then**
27: $INSERT(O, X, h(X))$
28: **else if** $(b(Y) \neq X$ and $h(X) > h(Y) + c(X, Y))$ and $(t(Y) = CLOSED)$ and $(h(Y) > k_{old})$ **then**
29: $INSERT(O, Y, h(Y))$
30: **end if**
31: **end for**
32: **end if**
33: Return $GET - KMIN(O)$ (algorithm 30)

H.4 Optimal Plans

There exists a huge number of search algorithms in the literature, with the ones discussed here being just the most basic ones. All of the techniques discussed here result in a path. A path is sufficient if the robot is able to follow it. Sometimes, randomness may push the robot off its path. One possibility is to replan, as we did in D^* (albeit for different reasons: above the environment changed and thereby mandated replanning). Another is to determine the best action for *all* nodes in the graph, not just the ones along the shortest path. A mapping from nodes to actions is called a *universal plan,* or *policy*. Techniques for finding optimal policies are known as universal planners and can be computationally more involved than the shortest path techniques surveyed here. One simple way to attain a universal plan to a goal is to run Dijkstra's algorithm backward (as in D^*): After completion, we know for each node in the graph the length of an optimal path to the goal, along with the appropriate action. Generalizations of this approach are commonly used in stochastic domains, where the outcome of actions is modeled by a probability distribution over nodes in the graph.