Principles of Robot Motion:
Theory, Algorithms, and Implementation

ERRATA!!!! ¹

Howie Choset, Kevin Lynch, Seth Hutchinson, George Kantor, Wolfram Burgard, Lydia Kavraki, and Sebastian Thrun

August 23, 2010

¹(C) 2007, Choset, Lynch, Hutchinson, Kantor, Burgard, Kavraki, Thrun . Do not copy or distribute without expressed permission from the authors. This is a preliminary draft meant for review.
Chapter 1

Introduction

No bugs to report, yet!
CHAPTER 1. INTRODUCTION
Chapter 2

Bug Algorithms

Even a simple planner can present interesting and difficult issues. The Bug1 and Bug2 algorithms [289] are among the earliest and simplest sensor-based planners with provable guarantees. These algorithms assume the robot is a point operating in the plane with a contact sensor or a zero range sensor to detect obstacles. When the robot has a finite range (non-zero range) sensor, then the Tangent Bug algorithm [208] is a Bug derivative that can use that sensor information to find shorter paths to the goal. The Bug and Bug-like algorithms are straightforward to implement; moreover, a simple analysis shows that their success is guaranteed, when possible. These algorithms require two behaviors: move on a straight line and follow a boundary. To handle boundary-following, we introduce a curve-tracing technique based on the implicit function theorem at the end of this chapter. This technique is general to following any path, but we focus on following a boundary at a fixed distance.

2.1 Bug1 and Bug2

Perhaps the most straightforward path planning approach is to move toward the goal, unless an obstacle is encountered, in which case, circumnavigate the obstacle until motion toward the goal is once again allowable. Essentially, the Bug1 algorithm formalizes the “common sense” idea of moving toward the goal and going around obstacles. The robot is assumed to be a point with perfect positioning (no positioning error) with a contact sensor that can detect an obstacle boundary if the point robot “touches” it. The robot can also measure the distance $d(x, y)$ between any two points $x$ and $y$. Finally, assume that the workspace is bounded. Let $B_r(x)$ denote a ball of radius...
$r$ centered on $x$, i.e., $B_r(x) = \{y \in \mathbb{R}^2 | d(x, y) < r\}$. The fact that the workspace is bounded implies that for all $x \in \mathcal{W}$, there exists an $r$ such that $\mathcal{W} \subset B_r(x)$.

The start and goal are labeled $q_{\text{start}}$ and $q_{\text{goal}}$, respectively. Let $q_{L0} = q_{\text{start}}$ and the $m$-line be the line segment that connects $q_{L0}$ to $q_{\text{goal}}$. Initially, $i = 0$. The Bug1 algorithm exhibits two behaviors: motion-to-goal and boundary-following. During motion-to-goal, the robot moves along the $m$-line toward $q_{\text{goal}}$ until it either encounters the goal or an obstacle. If the robot encounters an obstacle, let $q_{H1}$ be the point where the robot first encounters an obstacle and call this point a hit point. The robot then circumnavigates the obstacle until it returns to $q_{H1}$. Then, the robot determines the closest point to the goal on the perimeter of the obstacle and call this a leave point and labeled $q_{L1}$. From $q_{L1}$, the robot heads straight toward the goal again, i.e., it reinvokes the motion-to-goal behavior. If the line that connects $q_{L1}$ and the goal intersects the current obstacle, then there is no path to the goal; note that this intersection would occur immediately “after” leaving $q_{L1}$. Otherwise, the index $i$ is incremented and this procedure is then repeated for $q_{L1}$ and $q_{H1}$ until the goal is reached or the planner determines that the robot cannot reach the goal (figures 2.1, 2.2). Finally, if the line to the goal “grazes” an obstacle, the robot need not invoke a boundary following behavior, but rather continues onward toward the goal. See algorithm 1 for a description of the Bug1 approach.

Like its Bug1 sibling, the Bug2 algorithm exhibits two behaviors: motion-to-goal and boundary-following. During motion-to-goal, the robot moves toward the goal on the $m$-line; however, in Bug2 the $m$-line connects $q_{\text{start}}$ and $q_{\text{goal}}$, and thus remains fixed. The boundary-following behavior is invoked if the robot encounters an obstacle, but this behavior is different from that of Bug1. For Bug2, the robot circumnavigates the obstacle until it reaches a new point on the $m$-line closer to the goal than the initial point of contact with the obstacle. At this time, the robot proceeds toward the goal, repeating this process if it encounters an object. If the robot re-encounters the original departure point from the $m$-line, then the robot concludes there is no path to the goal (figures 2.3, 2.4).

Let $x \in \mathcal{W}_{\text{free}} \subset \mathbb{R}^2$ be the current position of the robot, $i = 1$, and $q_{L0}$ be the start location. See algorithm 2 for a description of the Bug2 approach.

At first glance, it seems that Bug2 is a more effective algorithm than Bug1 because the robot does not have to entirely circumnavigate the obstacles; however, this is not always the case. This can be seen by comparing the lengths of the paths found by the two algorithms. For Bug1, when
2.1. "BUG1 AND BUG2"

Figure 2.1. The Bug1 algorithm successfully finds the goal.

Figure 2.2. The Bug1 algorithm reports the goal is unreachable.

the ith obstacle is encountered, the robot completely circumnavigates the boundary, and then returns to the leave point. In the worst case, the robot must traverse half the perimeter, $p_i$, of the obstacle to reach this leave point. Moreover, in the worst case, the robot encounters all $n$ obstacles. If there
Algorithm 1 Bug1 Algorithm

Input: A point robot with a tactile sensor
Output: A path to the \( q_{\text{goal}} \) or a conclusion no such path exists

1: while Forever do
2:   repeat
3:     From \( q_{L_{i-1}} \), move toward \( q_{\text{goal}} \).
4:     until \( q_{\text{goal}} \) is reached or an obstacle is encountered at \( q_{H_{i}} \).
5:   if Goal is reached then
6:     Exit.
7:   end if
8:   repeat
9:     Follow the obstacle boundary.
10:    until \( q_{\text{goal}} \) is reached or \( q_{H_{i}} \) is re-encountered.
11:   Determine the point \( q_{L_{i}} \) on the perimeter that has the shortest distance to the goal.
12:   Go to \( q_{L_{i}} \).
13:   if the robot were not able to move toward the goal then
14:     Conclude \( q_{\text{goal}} \) is not reachable and exit.
15:   end if
16: end while

are no obstacles, the robot must traverse a distance of length \( d(q_{\text{start}}, q_{\text{goal}}) \). Thus, we obtain

\[
L_{\text{Bug1}} \leq d(q_{\text{start}}, q_{\text{goal}}) + 1.5 \sum_{i=1}^{n} p_{i}. \tag{2.1}
\]

For Bug2, the path length is a bit more complicated. Suppose that the line through \( q_{\text{start}} \) and \( q_{\text{goal}} \) intersects the \( i \)th obstacle \( n_{i} \) times. Then, there are at most \( n_{i} \) leave points for this obstacle, since the robot may only leave the obstacle when it returns to a point on this line. It is easy to see that half of these intersection points are not valid leave points because they lie on the “wrong side” of the obstacle, i.e., moving toward the goal would cause a collision. In the worst case, the robot will traverse nearly the entire perimeter of the obstacle for each leave point. Thus, we obtain

\[
L_{\text{Bug2}} \leq d(q_{\text{start}}, q_{\text{goal}}) + \frac{1}{2} \sum_{i=1}^{n} n_{i} p_{i}. \tag{2.2}
\]

Naturally, (2.2) is an upper-bound because the summation is over all of the
Algorithm 2 Bug2 Algorithm

**Input:** A point robot with a tactile sensor  
**Output:** A path to $q_{\text{goal}}$ or a conclusion no such path exists

1: \textbf{while} True \textbf{do} 
2: \hspace{1em} \textbf{repeat} 
3: \hspace{2em} From $q_i^L$, move toward $q_{\text{goal}}$ along $m$-line. 
4: \hspace{2em} \textbf{until} \hspace{1em} $q_{\text{goal}}$ is reached \textbf{or} 
5: \hspace{3em} an obstacle is encountered at hit point $q_i^H$. 
6: \hspace{1em} Turn left (or right). 
7: \hspace{1em} \textbf{repeat} 
8: \hspace{2em} Follow boundary 
9: \hspace{2em} \textbf{until} 
10: \hspace{3em} $q_{\text{goal}}$ is reached \textbf{or} 
11: \hspace{4em} $q_i^H$ is re-encountered \textbf{or} 
12: \hspace{5em} $m$-line is re-encountered at a point $m$ such that 
13: \hspace{6em} $m \neq q_i^H$ (robot did not reach the hit point), 
14: \hspace{6em} $d(m, q_{\text{goal}}) < d(m, q_i^H)$ (robot is closer), and 
15: \hspace{6em} if robot moves toward goal, it would not hit the obstacle 
16: \hspace{1em} \textbf{if} Goal is reached \textbf{then} 
17: \hspace{2em} Exit. 
18: \hspace{1em} \textbf{end if} 
19: \hspace{1em} \textbf{if} $q_i^H$ is re-encountered \textbf{then} 
20: \hspace{2em} Conclude goal is unreachable 
21: \hspace{1em} \textbf{end if} 
22: \hspace{1em} Let $q_{i+1}^L = m$ 
23: \hspace{1em} Increment $i$ 
24: \hspace{1em} \textbf{end while}

obstacles as opposed to over the set of obstacles that are encountered by the robot.

A casual examination of (2.1) and (2.2) shows that $L_{\text{Bug2}}$ can be arbitrarily longer than $L_{\text{Bug1}}$. This can be achieved by constructing an obstacle whose boundary has many intersections with the $m$-line. Thus, as the “complexity” of the obstacle increases, it becomes increasingly likely that Bug1 could outperform Bug2 (figure 2.4).

In fact, Bug1 and Bug2 illustrate two basic approaches to search problems. For each obstacle that it encounters, Bug1 performs an exhaustive search to find the optimal leave point. This requires that Bug1 traverse
Tangent Bug

2.2 Tangent Bug

often yields better performance. In contrast, Bug2 uses an opportunistic approach. When Bug2 finds a leave point that is better than any it has seen before, it commits to that leave point. Such an algorithm is also called greedy, since it opts for the first promising option that is found. When the obstacles are simple, the greedy approach of Bug2 gives a quick payoff, but when the obstacles are complex, the more conservative approach of Bug1 often yields better performance.

2.2 Tangent Bug

Tangent Bug [207] serves as an improvement to the Bug2 algorithm in that it
determines a shorter path to the goal using a range sensor with a 360 degree infinite orientation resolution. Sometimes orientation is called \textit{azimuth}. We model this range sensor with the \textit{raw distance function} \( \rho: \mathbb{R}^2 \times S^1 \rightarrow \mathbb{R} \). Consider a point robot situated at \( x \in \mathbb{R}^2 \) with rays radially emanating from it. For each \( \theta \in S^1 \), the value \( \rho(x, \theta) \) is the distance to the closest obstacle along the ray from \( x \) at an angle \( \theta \). More formally,

\[
\rho(x, \theta) = \min_{\lambda \in [0, \infty]} d(x, x + \lambda[\cos \theta, \sin \theta]^T),
\]

such that \( x + \lambda[\cos \theta, \sin \theta]^T \in \bigcup_i \mathcal{W}_i \). \hfill (2.3)

Note that there are infinitely many \( \theta \in S^1 \) and hence the infinite resolution. This assumption is approximated with a finite number of range sensors situated along the circumference of a circular mobile robot which we have modeled as a point.

Since real sensors have limited range, we define the \textit{saturated raw distance function}, denoted \( \rho_R: \mathbb{R}^2 \times S^1 \rightarrow \mathbb{R} \), which takes on the same values as \( \rho \) when the obstacle is within sensing range, and has a value of infinity when
the ray lengths are greater than the sensing range, \( R \), meaning that the obstacles are outside the sensing range. More formally,

\[
\rho_R(x, \theta) = \begin{cases} 
\rho(x, \theta), & \text{if } \rho(x, \theta) < R \\
\infty, & \text{otherwise.}
\end{cases}
\]

The set of points within sensing range of the robot can be denoted by

\[ V_R(x) = \{ y \in Q_{\text{free}} | d(x, y) < R \text{ and } x + (1 - \lambda)y \in W_{\text{free}} \text{ for all } \lambda \in [0, 1] \} \]

The Tangent Bug planner assumes that the robot can detect discontinuities in \( \rho_R \) as depicted in figure ???. For a fixed \( x \in \mathbb{R}^2 \), an interval of continuity is defined to be a connected set of points on \( \partial V(x) \) where \( \rho_R \) varies continuously and is finite. These are the points \( x + \rho(x, \theta) [\cos \theta, \sin \theta]^T \) on the boundary of the free space where \( \rho_R(x, \theta) \) is finite and continuous with respect to \( \theta \).

The endpoints of these intervals occur where \( \rho_R(x, \theta) \) loses continuity, either as a result of one obstacle blocking another or the sensor reaching its range limit. The endpoints are denoted \( O_i \). Figure 2.5 contains an example where \( \rho_R \) loses continuity. The points \( O_1, O_2, O_3, O_5, O_6, O_7, \) and \( O_8 \) correspond to losses of continuity associated with obstacles blocking other portions of \( W_{\text{free}} \); note the rays are tangent to the obstacles here. The point \( O_4 \) is a discontinuity because the obstacle boundary falls out of range of the sensor. The sets of points on the boundary of the free space between \( O_1 \) and \( O_2 \), \( O_3 \) and \( O_4 \), \( O_5 \) and \( O_6 \), \( O_7 \) and \( O_8 \) are the intervals of continuity.

Just like the other Bugs, Tangent Bug (algorithm 3) iterates between two behaviors: motion-to-goal and boundary-following. However, these behaviors are different than in the Bug1 and Bug2 approaches. Although motion-to-goal directs the robot to the goal, this behavior may have a phase where the robot follows the boundary. Likewise, the boundary-following behavior may have a phase where the robot does not follow the boundary.

The robot initially invokes the motion-to-goal behavior, which itself has two parts. First, the robot attempts to move in a straight line toward the goal until it senses an obstacle \( R \) units away and directly between it and the goal. This means that a line segment connecting the robot and goal must intersect an interval of continuity. For example, in figure 2.6, \( WO_2 \) is within sensing range, but does not block the goal, but \( WO_1 \) does. When the robot initially senses an obstacle, the circle of radius \( R \) becomes tangent to the obstacle. Immediately after, this tangent point splits into two \( O_i \)'s, which are the endpoints of the interval. If the obstacle is in front of the robot, then this interval intersects the segment connecting the robot and the goal.
2.2. TANGENT BUG

Consider the $O_i$ where $d(O_i, q_{\text{goal}}) < d(x, q_{\text{goal}})$. The robot then moves toward one of these $O_i$ that maximally decreases a heuristic distance to the goal. An example of a heuristic distance is the sum $d(x, O_i) + d(O_i, q_{\text{goal}})$. (The heuristic distance can be more complicated when factoring in available information with regard to the obstacles.) In figure 2.7 (left), the robot sees $WO_1$ and drives to $O_2$ because $i = 2$ minimizes $d(x, O_i) + d(O_i, q_{\text{goal}})$. When the robot is located at $x$, it cannot know that $WO_2$ blocks the path from $O_2$ to the goal. In figure 2.7(right), when the robot is located at $x$ but the goal is different, it has enough sensor information to conclude that $WO_2$ indeed blocks a path from $O_2$ to the goal, and therefore drives toward $O_4$. So, even though driving toward $O_2$ may initially minimize $d(x, O_i) + d(O_i, q_{\text{goal}})$ more than driving toward $O_4$, the planner effectively assigns an infinite cost to $d(O_2, q_{\text{goal}})$ because it has enough information to conclude that any path through $O_2$ will be suboptimal.
CHAPTER 2. BUG ALGORITHMS

Figure 2.6. The vertical represents the path of the robot and the dotted circle its sensing range. Currently, the robot is located at the “top” of the line segment. The points $O_i$ represent the points of discontinuity of the saturated raw distance function. Note that the robot passes by $WO_2$.

Figure 2.7. (Left) The planner selects $O_2$ as a subgoal for the robot. (Right) The planner selects $O_4$ as a subgoal for the robot. Note the line segment between $O_4$ and $q_{goal}$ cuts through the obstacle.

The set $\{O_i\}$ is continuously updated as the robot moves toward a particular $O_i$, which can be seen in figure 2.8. At $t = 1$, the robot has not
sensed the obstacle, hence the robot moves toward the goal. At $t = 2$, the robot initially senses the obstacle, depicted by a thick solid curve. The robot continues to move toward the goal, but off to the side of the obstacle heading toward the discontinuity in $\rho$. For $t = 3$ and $t = 4$, the robot senses more of the obstacle and continues to decrease distance toward the goal while hugging the boundary.

The robot undergoes motion-to-goal until it can no longer decrease the heuristic distance to the goal following the rules of motion-to-goal. Put differently, it finds a point that is like a local minimum of $d(\cdot, O_i) + d(O_i, q_{\text{goal}})$ restricted to the path that motion-to-goal dictates.

When the robot switches to boundary-following, it determines the point $M$ on the currently sensed portion of the blocking obstacle that has the shortest distance to the goal. The robot then moves in the same direction as if it were in the motion-to-goal behavior. It continuously moves toward the $O_i$ on the blocking obstacle in the chosen direction (figure 2.9). While undergoing this motion, the planner also updates two values: $d_{\text{min}}$ and $d_{\text{leave}}$. The value $d_{\text{min}}$ is the shortest distance between the goal and any point on the blocking obstacle boundary that has been sensed thus far. The value $d_{\text{leave}}$ is the shortest distance between the goal and any point in the currently sensed environment at that robot location, i.e., $\partial V_R(x)$. Let $d_{\text{leave}}(x) = \min_{y \in V(x)} d(x, y)$. When $d_{\text{leave}}(x) < d_{\text{min}}$, the robot terminates the boundary-following behavior.
Note that in many cases, when the robot terminates the boundary-following behavior, it will drive toward the goal. In such a case, there is no blocking obstacle; define $T$ to be the point where a circle, centered at $x$ of radius $R$, intersects the segment that connects $x$ and $q_{\text{goal}}$. This is the point on the periphery of the sensing range that is closest to the goal when the robot is located at $x$. When there is no blocking obstacle, $d_{\text{leave}} = d(T, q_{\text{goal}})$. Otherwise, $T$ is undefined and $d_{\text{leave}}$ is constantly updated to be the shortest distance between the goal and any point on an obstacle boundary that is viewable within the robot’s current sensor range.

Figure 2.10 contains a path for a robot with zero sensor range. Here the robot invokes a motion-to-goal behavior until it encounters the first obstacle at hit point $H_1$. Unlike Bug1 and Bug2, encountering a hit point does not change the behavior mode for the robot. The robot continues with the motion-to-goal behavior by turning right and following the boundary of the first obstacle. The robot turned right because that direction minimized its heuristic distance to the goal. The robot departs this boundary at a depart point $D_1$. The robot continues with the motion-to-goal behavior, maneuvering around a second obstacle, until it encounters the third obstacle at $H_3$. The robot turns left and continues to invoke the motion-to-goal behavior until it reaches $M_3$, a minimum point. Now, the planner invokes
Algorithm 3 Tangent Bug Algorithm

**Input:** A point robot with a range sensor

**Output:** A path to the q_{goal} or a conclusion no such path exists

1: while True do
2:   repeat
3:     Continuously move toward the point $n \in \{T, O_i\}$ which minimizes $d(x, n) + d(n, q_{goal})$ where $d(n, q_{goal}) < d(x, q_{goal})$
4:   until
5:     
6:     • the goal is encountered or
7:     • The direction that minimizes $d(x, n) + d(n, q_{goal})$ begins to increase $d(x, q_{goal})$, i.e., the robot detects a “local minimum” of $d(\cdot, q_{goal})$.
8: Choose a boundary following direction which continues in the same direction as the most recent motion-to-goal direction.
9: repeat
10:   Continuously update $d_{\text{leave}}$, $d_{\text{min}}$, and $\{O_i\}$.
11:   Continuously moves toward $n \in \{O_i\}$ that is in the chosen boundary direction.
12: until
13:     
14:     • The goal is reached.
15:     • The robot completes a cycle around the obstacle in which case the goal cannot be achieved.
16:     • $d_{\text{leave}} < d_{\text{min}}$
17: end while

the boundary-following behavior until the robot reaches $L_3$. Note that since we have zero sensing range, $d_{\text{leave}}$ is the distance between the robot and the goal. The procedure continues until the robot reaches the goal. Only at $M_i$ and $L_i$ does the robot switch between behaviors.

Figures 2.11 and 2.12 contain examples where the robot has finite and infinite sensing ranges, respectively. Note that in these examples, since the robot has a non-zero sensor range, it does not reach an $M_i$ but rather reaches an $sw_i$ where it detects its corresponding $M_i$. The $sw_i$ are sometimes called switch points.

Figures 2.13 demonstrates a situation in which the robot invokes bound-
Figure 2.10. The path generated by Tangent Bug with zero sensor range. The dashed lines correspond to the motion-to-goal behavior and the dotted lines correspond to boundary-following.

Figure 2.11. Path generated by Tangent Bug with finite sensor range. The dashed lines correspond to the motion-to-goal behavior and the dotted lines correspond to boundary-following. The dashed-dotted circles correspond to the sensor range of the robot.

boundary following at a minimum point $M_1$ but must update the value for $d_{\text{min}}$ when it encounters another minimum $M_2$. The boundary following behavior in this case continues until the robot reaches $L_2$. 
2.3. IMPLEMENTATION

Essentially, the bug algorithms have two behaviors: drive toward a point and follow an obstacle. The first behavior is simply a form of gradient descent of \( d(\cdot, n) \) where \( n \) is either \( q_{\text{goal}} \) or an \( O_i \). The second behavior, boundary-following, presents a challenge because the obstacle boundary is not known \textit{a priori}. Therefore, the robot planner must rely on sensor information.
to determine the path. However, we must concede that the full path will not be determined from one sensor reading: the sensing range of the robot may be limited and the robot may not be able to “see” the entire world from one vantage point. So, the robot planner has to be incremental. We must determine first what information the robot requires and then where the robot should move to acquire more information. This is indeed the challenge of sensor-based planning. Ideally, we would like this approach to be reactive with sensory information feeding into a simple algorithm that outputs translational and rotational velocity for the robot.

There are three questions: What information does the robot require to circumnavigate the obstacle? How does the robot infer this information from its sensor data? How does the robot use this information to determine (locally) a path?

2.3.1 What Information: The Tangent Line

If the obstacle were flat, such as a long wall in a corridor, then following the obstacle is trivial: simply move parallel to the obstacle. This is readily implemented using a sensing system that can determine the obstacle’s surface normal \( n(x) \), and hence a direction parallel to its surface. However, the world is not necessarily populated with flat obstacles; many have non-zero curvature. However, the robot can follow a path that is consistently orthogonal to the surface normal; this direction can be written as \( n(x) \perp \) and the resulting path satisfies \( \dot{c}(t) = v \) where \( v \) is a basis vector in \( (n(c(t))) \perp \). The sign of \( v \) is based on the “previous” direction of \( \dot{c} \).

Consistently determining the surface normal can be quite challenging and therefore for implementation, we can assume that obstacles are “locally flat.” This means the sensing system determines the surface normal, the robot moves orthogonal to this normal for a short distance, and then the process repeats. In a sense, the robot determines the sequence of short straight-line segments to follow based on sensor information.

This flat line, loosely speaking, is the tangent (figure 2.14). It is a linear approximation of the curve at the point where the tangent intersects the curve. The tangent can also be viewed as a first-order approximation to the function that describes the curve. Let \( c: [0, 1] \to Q_{\text{free}} \) be the function that defines a path. Let \( x = c(s_0) \) for a \( s_0 \in [0, 1] \). The tangent at \( x \) is \( \frac{dc}{ds} \big|_{s = s_0} \). The tangent space can be viewed as a line whose basis vector is \( \frac{dc}{ds} \big|_{s = s_0} \), i.e.,

\[
\left\{ \alpha \frac{dc}{ds} \big|_{s = s_0} \mid \alpha \in \mathbb{R} \right\}.
\]
2.3. IMPLEMENTATION

2.3.2 How to Infer Information with Sensors: Distance and Gradient

The next step is to infer the tangent from sensor data. Instead of thinking of the robot as a point in the plane, let’s think of it as a circular base which has a fine array of tactile sensors radially distributed along its circumference (figure 2.15). When the robot contacts an obstacle, the direction from the contacted sensor to the robot’s center approximates the surface normal. With this information, the robot can determine a sequence of tangents to follow the obstacle.

Unfortunately, using a tactile sensor to prescribe a path requires the robot to collide with obstacles, which endangers the obstacles and the robot. Instead, the robot should follow a path at a safe distance \( W^* \in \mathbb{R} \) from the nearest obstacle. Such a path is called an offset curve \([360]\). Let \( D(x) \) be the distance from \( x \) to the closest obstacle, i.e.,

\[
D(x) = \min_{c \in \bigcup_{i} W_{Oi}} d(x, c).
\]  

To measure this distance with a mobile robot equipped with an onboard range sensing ring, we use the raw distance function again. However, instead of looking for discontinuities, we look for the global minimum. In other words, \( D(x) = \min_s \rho(x, s) \) (figure 2.16).

We will need to use the gradient of distance. In general, the gradient is a vector that points in the direction that maximally increases the value of
CHAPTER 2. BUG ALGORITHMS

Figure 2.15. A fine-resolution tactile sensor.

a function. See appendix ?? for more details. Typically, the \( i \)th component of the gradient vector is the partial derivative of the function with respect to its \( i \)th coordinate. In the plane, \( \nabla D(x) = \begin{bmatrix} \frac{\partial D(x)}{\partial x_1} \\ \frac{\partial D(x)}{\partial x_2} \end{bmatrix} \) which points in the direction that increases distance the most. Finally, the gradient is the unit direction associated with the smallest value of the raw distance function. Since the raw distance function seemingly approximates a sensing system with individual range sensing elements radially distributed around the perimeter of the robot, an algorithm defined in terms of \( D \) can often be implemented using realistic sensors.

There are many choices for range sensors; here, we investigate the use of ultrasonic sensors (figure 2.17), which are commonly found on mobile robots. Conventional ultrasonic sensors measure distance using time of flight. When the speed of sound in air is constant, the time that the ultrasound requires to leave the transducer, strike an object, and return is proportional to the distance to the point of reflection on the object \([106]\). This object, however, can be located anywhere along the angular spread of the sonar sensor’s beam pattern (figure 2.18). Therefore, the distance information that sonars
2.3. IMPLEMENTATION

Figure 2.16. The global minimum of the rays determines the distance to the closest obstacle; the gradient points in a direction away from the obstacle along the ray.

provide is fairly accurate in depth, but not in azimuth. The beam pattern can be approximated with a cone (figure 2.19). For the commonly used Polaroid transducer, the arcbase is 22.5 degrees. When the reading of the sensor is $d$, the point of reflection can be anywhere along the arc base of length $\frac{2\pi d \times 22.5}{360}$.

Initially, assume that the echo originates from the center of the sonar cone. We acknowledge that this is a naive model, hence we term this the centerline model (figure 2.19). The ultrasonic sensor with the smallest reading approximates the global minimum of the raw distance function, and hence $D(x)$. The direction that this sensor is facing approximates the negated gradient $-\nabla D(x)$ because this sensor faces the closest obstacle. The tangent is then the line orthogonal to the direction associated with the smallest sensor reading.

2.3.3 How to Process Sensor Information: Continuation Methods

The tangent to the offset curve is $(\nabla D(x))^\perp$, the line orthogonal to $\nabla D(x)$ (figure 2.14). The vector $\nabla D(x)$ points in the direction that maximally increases distance; likewise, the vector $-\nabla D(x)$ points in the direction that
maximally decreases distance; they both point along the same line, but in opposite directions. Therefore, the vector $(\nabla D(x))^\perp$ points in the direction that locally maintains distance; it is perpendicular to both $\nabla D(x)$ and $-\nabla D(x)$. This would be the tangent of the offset curve which maintains distance to the nearby obstacle.

Another way to see why $(\nabla D(x))^\perp$ is the tangent is to look at the def-
2.3. IMPLEMENTATION

Obstacle

Beam Pattern

Sensor Measurement Axis

d

Point

Sensor

Robot

Figure 2.19. Centerline model.

inition of the offset curve. For a safety distance \( W^* \), we can define the offset curve implicitly as the set of points where \( G(x) = D(x) - W^* \) maps to zero. The set of nonzero points (or vectors) that map to zero is called the null space of a map. For a curve implicitly defined by \( G \), the tangent space at a point \( x \) is the null space of \( DG(x) \), the Jacobian of \( G \) \([388]\). In general, the \( i, j \)th component of the Jacobian matrix is the partial derivative of the \( i \)th component function with respect to the \( j \)th coordinate and thus the Jacobian is a mapping between tangent spaces. Since in this case, \( G \) is a real-valued function (\( i = 1 \)), the Jacobian is just a row vector \( DD(x) \). Here, we are reusing the symbol \( D \). The reader is forced to use context to determine if \( D \) means distance or differential.

In Euclidean spaces, the \( i \)th component of a single-row Jacobian equals the \( i \)th component of the gradient and thus \( \nabla D(x) = (DD(x))^T \). Therefore, since the tangent space is the null space of \( DD(x) \), the tangent for boundary-following in the plane is the line orthogonal to \( \nabla D(x) \), i.e., \( (\nabla D(x))^\perp \), and can be derived from sensor information.

Using distance information, the robot can determine the tangent direction to the offset curve. If the obstacles are flat, then the offset curve is also flat, and simply following the tangent is sufficient to follow the boundary of an unknown obstacle. Consider, instead, an obstacle with curvature. We can, however, assume that the obstacle is locally flat. The robot can then move along the tangent for a short distance, but since the obstacle has curvature, the robot will not follow the offset curve, i.e., it will “fall off” of the offset curve. To reaccess the offset curve, the robot moves either toward or away from the obstacle until it reaches the safety distance \( W^* \). In doing so,
the robot is moving along a line defined by $\nabla D(x)$, which can be derived from sensor information.

Essentially, the robot is performing a numerical procedure of prediction and correction. The robot uses the tangent to locally predict the shape of the offset curve and then invokes a correction procedure once the tangent approximation is not valid. Note that the robot does not explicitly trace the path but instead “hovers” around it, resulting in a sampling of the path, not the path itself (figure 2.20).

A numerical tracing procedure can be posed as one which traces the roots of the expression $G(x) = 0$, where in this case $G(x) = D(x) - W^*$. Numerical curve-tracing techniques rest on the implicit function theorem \[6, 222, 294\] which locally defines a curve that is implicitly defined by a map $G: Y \times \mathbb{R} \rightarrow Y$. Specifically, the roots of $G$ locally define a curve parameterized by $\lambda \in \mathbb{R}$. See appendix ?? for a formal definition.

For boundary following at a safety distance $W^*$, the function $G(y, \lambda) = D(y, \lambda) - W^*$ implicitly defines the offset curve. Note that the $\lambda$-coordinate corresponds to a tangent direction and the $y$-coordinates to the line or hyperplane orthogonal to the tangent. Let $Y$ denote this hyperplane and $D_Y G$ be the matrix formed by taking the derivative of $G(x) = D(x) - W^* = 0$ with respect to the $y$-coordinates. It takes the form $D_Y G(x) = D_Y D(x)$ where $D_Y$ denotes the gradient with respect to the $y$-coordinates. If $D_Y G(y, \lambda)$ is surjective at $x = (\lambda, y)^T$, then the implicit function theorem states that the roots of $G(y, \lambda)$ locally define a curve that follows the boundary at a distance $W^*$ as $\lambda$ is varied, i.e., $y(\lambda)$.

By numerically tracing the roots of $G$, we can locally construct a path. While there are a number of curve tracing techniques \[222\], let us consider an adaptation of a common predictor-corrector scheme. Assume that the
robot is located at a point \( x \) which is a fixed distance \( \mathcal{W}^* \) away from the boundary. The robot takes a “small” step, \( \Delta \lambda \), in the \( \lambda \)-direction (i.e., the tangent to the local path). In general, this prediction step takes the robot off the offset path. Next, a correction method is used to bring the robot back onto the offset path. If \( \Delta \lambda \) is small, then the local path will intersect a correcting plane, which is a plane orthogonal to the \( \lambda \)-direction at a distance \( \Delta \lambda \) away from the origin.

The correction step finds the location where the offset path intersects the correcting plane and is an application of the Newton convergence theorem \([222]\). See appendix ?? for a more formal definition of this theorem. The Newton convergence theorem also requires that \( D_Y G(y, \lambda) \) be full rank at every \( (y, \lambda) \) in a neighborhood of the offset path. This is true because for \( G(x) = D(x) - \mathcal{W}^* \), \( [0 \quad D_Y G(y, \lambda)]^T = DG(y, \lambda) \). Since \( DG(y, \lambda) \) is full rank, so must be \( D_Y G(y, \lambda) \) on the offset curve. Since the set of nonsingular matrices is an open set, we know there is a neighborhood around each \( (y, \lambda) \) in the offset path where \( DG(y, \lambda) \) is full rank and hence we can use the iterative Newton method to implement the corrector step. If \( y^h \) and \( \lambda^h \) are the \( h \)th estimates of \( y \) and \( \lambda \), the \( h+1 \)st iteration is defined as

\[
y^{h+1} = y^h - (D_Y G)^{-1} G(y^h, \lambda^h),
\]

where \( D_Y G \) is evaluated at \( (y^h, \lambda^h) \). Note that since we are working in a Euclidean space, we can determine \( D_Y G \) solely from distance gradient, and hence, sensor information.

Problems

1. Prove that \( D(x) \) is the global minimum of \( \rho(x, s) \) with respect to \( s \).

2. What are the tradeoffs between the Bug1 and Bug2 algorithms?

3. Extend the Bug1 and Bug2 algorithms to a two-link manipulator.

4. What is the difference between the Tangent Bug algorithm with zero range detector and Bug2? Draw examples.

5. What are the differences between the path in figure 2.10 and the paths that Bug1 and Bug2 would have generated?

6. The Bug algorithms also assume the planner knows the location of the goal and the robot has perfect positioning. Redesign one of the Bug algorithms to relax the assumption of perfect positioning. Feel free
to introduce a new type of “reasonable” sensor (not a high-resolution
Global Positioning System).

7. In the Bug1 algorithm, prove or disprove that the robot does not
encounter any obstacle that does not intersect the disk of radius
d(q_{\text{start}}, q_{\text{goal}}) centered at q_{\text{goal}}.

8. What assumptions do the Bug1, Bug2, and Tangent Bug algorithms
make on robot localization, both in position and orientation?

9. Prove the completeness of the Tangent Bug algorithm.

10. Adapt the Tangent Bug algorithm so that it has a limited field of view
sensor, i.e., it does not have a 360 degree field of view range sensor.

11. Write out $D_Y G$ for boundary following in the planar case.

12. Let $G_1(x) = D(x) + 1$ and let $G_2(x) = D(x) + 2$. Why are their
Jacobians the same?

13. Let $G(x, y) = y^3 + y - x^2$. Write out $y$ as a function of $x$ in an
interval about the origin for the curve defined by $G(x, y) = 0$. 
Chapter 3

Configuration Space

- Page 56, in the paragraph
  As an example, consider the one-dimensional manifold $S^1 = \{ x = (x_1, x_2) \in \mathbb{R}^2 \mid x_1^2 + x_2^2 = 1 \}$. For any point $x \in S^1$ we can define a neighborhood that is diffeomorphic to $\mathbb{R}$. For example, consider the upper portion of the circle, $U_1 = \{ x \in S^1 \mid x_2 > 0 \}$. The chart $\phi_1 : U_1 \rightarrow (0, 1)$ is given by $\phi_1(x) = x_1$, and thus $x_1$ can be used to define a local coordinate system for the upper semicircle. In the other direction, the upper portion of the circle can be parameterized by $z \in (0, 1) \subset \mathbb{R}$, with $\phi_1^{-1}(z) = (z, (1 - z)^{1/2})$, which maps the open unit interval to the upper semicircle. But $S^1$ is not globally diffeomorphic to $\mathbb{R}^1$; we cannot find a single chart whose domain includes all of $S^1$.

- All of the $(0, 1)$'s should be $(-1, 1)$.

- The $\phi_1^{-1}(z) = (z, (1 - z)^{1/2})$ should read $\phi_1^{-1}(z) = (z, (1 - z^2)^{1/2})$

- Page 57, the

\[\begin{align*}
\phi_1^{-1}(z) &= (z, 1 - z^2) \\
\phi_2^{-1}(z) &= (z, z^2 - 1) \\
\phi_3^{-1}(z) &= (1 - z^2, z) \\
\phi_4^{-1}(z) &= (z^2 - 1, z).
\end{align*}\]
should read

\[
\begin{align*}
\phi_1^{-1}(z) &= (z, (1 - z^2)^{\frac{1}{2}}) \\
\phi_2^{-1}(z) &= (z, (z^2 - 1)^{\frac{1}{2}}) \\
\phi_3^{-1}(z) &= ((1 - z^2)^{\frac{1}{2}}, z) \\
\phi_4^{-1}(z) &= ((z^2 - 1)^{\frac{1}{2}}, z).
\end{align*}
\]

• Page 58, the \(1 - z^2\) should read \((1 - z^2)^{\frac{1}{2}}\).

• Page 71, in the Jacobian matrix, element \((2, 3)\) lower right-hand corner, the element should not be \(\frac{d\phi_3}{dq_2}\), but instead it should be \(\frac{d\phi_2}{dq_2}\), so the Jacobian at the end of the chapter (before the problems) which currently reads

\[
J(q) = \frac{\partial \phi}{\partial q} = \begin{bmatrix}
\frac{\partial \phi_1}{\partial q_1} & \frac{\partial \phi_1}{\partial q_2} & \frac{\partial \phi_3}{\partial q_3} \\
\frac{\partial \phi_2}{\partial q_1} & \frac{\partial \phi_2}{\partial q_2} & \frac{\partial \phi_3}{\partial q_3}
\end{bmatrix} = \begin{bmatrix}
1 & 0 & -r_1 \sin q_3 - r_2 \cos q_3 \\
0 & 1 & r_1 \cos q_3 - r_2 \sin q_3
\end{bmatrix}.
\]

should read

\[
J(q) = \frac{\partial \phi}{\partial q} = \begin{bmatrix}
\frac{\partial \phi_1}{\partial q_1} & \frac{\partial \phi_1}{\partial q_2} & \frac{\partial \phi_2}{\partial q_3} \\
\frac{\partial \phi_2}{\partial q_1} & \frac{\partial \phi_2}{\partial q_2} & \frac{\partial \phi_2}{\partial q_3}
\end{bmatrix} = \begin{bmatrix}
1 & 0 & -r_1 \sin q_3 - r_2 \cos q_3 \\
0 & 1 & r_1 \cos q_3 - r_2 \sin q_3
\end{bmatrix}.
\]
Chapter 4

Potential Functions

• On Pg 78, the Hessian matrix has the indices in the lower left corner incorrect. The $\frac{\partial^2 U}{\partial q_i \partial q_n}$ should read $\frac{\partial^2 U}{\partial q_n \partial q_i}$, so the matrix

$$
\begin{bmatrix}
\frac{\partial^2 U}{\partial q_1^2} & \cdots & \frac{\partial^2 U}{\partial q_1 \partial q_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial^2 U}{\partial q_n \partial q_1} & \cdots & \frac{\partial^2 U}{\partial q_n^2}
\end{bmatrix}
$$

should read

$$
\begin{bmatrix}
\frac{\partial^2 U}{\partial q_1^2} & \cdots & \frac{\partial^2 U}{\partial q_1 \partial q_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial^2 U}{\partial q_n \partial q_1} & \cdots & \frac{\partial^2 U}{\partial q_n^2}
\end{bmatrix}
$$

• On page 103 eq. 4.26 (there are two equations) are wrong. The u’s should be upper case and the double summation notation should be consistent.

$$
u(q) = \sum_j u_{att} + \sum_{ij} u_{rep ij}
$$

$$
= \sum_j J_j^T(q) \nabla u_{att} + \sum_i \sum_j J_j^T(q) \nabla u_{rep ij}
\quad (4.1)
$$

should read
\[
U(q) = \sum_j U_{\text{att}j} + \sum_i \sum_j U_{\text{rep}ij} \\
= \sum_j J_j^T(q) \nabla U_{\text{att}j}(q) + \sum_i \sum_j J_j^T(q) \nabla U_{\text{rep}ij} \quad (4.2)
\]
Chapter 5

Roadmaps

- Chap 5, pg 145, an $z$ should be replaced by a $q$, so

$$Q_\lambda = \{ x \in Q | \pi_1(q) = \lambda \}$$

should read

$$Q_\lambda = \{ q \in Q | \pi_1(q) = \lambda \}$$
CHAPTER 5. ROADMAPS
Chapter 6

Cell Decompositions

Bugs coming soon!
Chapter 7

Sampling-Based Algorithms

Bugs coming soon!
Chapter 8

Kalman Filtering

- Apparently, there is an Appendix J with no space but I cannot find it again.
- Page 276, 7 lines from the bottom, the line $H(k+1)^TK(y(k+1) - H(k+1)x(k+1|k))$. should be replaced with $H(k+1)^TK(y(k+1) - H(k+1)x(k+1|k))$.  
  There was missing hat on the $x(k+1|k)$.
- Page 277, the $\Delta x = Kv$ below eq. 8.7 should read $\Delta x = H^TKv$
- Page 285, eq. 8.30 has an extra right parenthesis
- Page 286, second line of eq. 8.34 $\triangleq Gx(k) + w(k)$ should be replaced with $\triangleq Hx(k) + w(k)$
- Page 287, fig 8.5 - bold circle is an ellipse but looks like a circle because the axes were scaled incorrectly.
- Page 291, the phrase
  the process model for this robot nonlinear, i.e.,
  is missing “is”.
- Page 297, eq. 8.48 $y(k)_i$ should be replaced with $y_i(k)$.
- on page 293, ninth line from the top $\chi_{ij}^2 = \nu_{ij} S_{ij}^{-1} \nu_{ij}^T$ should be replaced with $\chi_{ij}^2 = \nu_{ij}^T S_{ij}^{-1} \nu_{ij}$. 

37
Chapter 9

Bayesian Methods

• on page 308, in the first sentence of Section 9.1.1, \( P(x \mid u(0:k-1), y(1:k)) \) should have an \( x(k) \) not an \( x \) so should read
  \[ P(x(k) \mid u(0:k-1), y(1:k)) \]

• on page 319, eq. 9.18
  \[
  x_i = x_i^0 + \begin{bmatrix}
    x_{r_i}^0 + d' \cos(\theta_{r_i}^0 + \alpha') \\
    y_{r_i}^0 + d' \sin(\theta_{r_i}^0 + \alpha') \\
    \theta_{r_i}^0 + \alpha' + \beta'
  \end{bmatrix}
  \]
  should be replaced with
  \[
  x_i = x_i^0 + \begin{bmatrix}
    d' \cos(\theta_{r_i}^0 + \alpha') \\
    d' \sin(\theta_{r_i}^0 + \alpha') \\
    \alpha' + \beta'
  \end{bmatrix}
  \]

• on page 325, the \( h_{ij} \) below eq. (9.23) should read \( h_{i,j} \).
Chapter 10

Robot Dynamics

- page 362, section 10.4.1 Planar Rotation, sentence 3. There is a repeated “to.”
Chapter 11

Trajectory Planning

Bugs coming soon!
Chapter 12

Nonholonomic and Underactuated Systems

Bugs coming soon!
Appendix A

Mathematical Notation

No bugs to report, yet!
Appendix B

Basic Set Definitions

No bugs to report, yet!
Appendix C

Topology and Metric Spaces

No bugs to report, yet!
Appendix D

Mathematical Notation

No bugs to report, yet!
Appendix E

Representations of Orientation

No bugs to report, yet!
Appendix F

Polyhedra Robots in Polyhedra Worlds

No bugs to report, yet!
Appendix G

Analysis of Algorithms and Complexity Classes

No bugs to report, yet!
I believe that there were so many mistakes in the graph search appendix, that we just rewrote the whole thing. This files needs to replace the MIT Press official file.
Appendix H

Graph Representation and Basic Search

H.1 Graphs

Thus far, we have been operating on grids without taking full advantage of
the neighboring relationships among the cells. A grid, as well as other maps,
can be represented by a graph which encodes these neighboring relationships.
A graph is a collection of nodes and edges, i.e., $G = (V, E)$. 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
dge. 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 cells 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 vica 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
dge. Such values are called weights.

A path or walk in a graph is a sequence of nodes $\{V_i\}$ such that for
**APPENDIX H. GRAPH REPRESENTATION AND BASIC SEARCH**

### 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.

An adjacent nodes $V_i$ and $V_{i+1}$, $E_{i+1}$ 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 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
non-leaf node breaks the connectivity of the graph.

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 cell and an edge connects nodes of cells 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 cells surrounding the current cell. See figure H.4.

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 ??, section ??) is 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 for 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 the “eight-point connectivity metric.” Figure H.7 displays the link lengths for all shortest paths between each cell.
Figure H.6. White cells denote the path that was determined with breadth-first search.

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

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

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 cell and neighboring cells 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
APPENDIX H. GRAPH REPRESENTATION AND BASIC SEARCH

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 cho-
"on 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. 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 in-
troduce 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. This
priority is determined by the sum of the distance from the start node to the
current node and the heuristic at the current node.

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 highest priority, it is ex-
"on 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 highest priority, i.e., node A, and expand it,
H.2. A* ALGORITHM

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 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

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.
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.

The priority queue still contains nodes whose values are 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 values are smaller than that of the goal node, this path results in the best path.
H.2. A* ALGORITHM

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.

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
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:

- $\text{Star}(n)$ 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 $q_{\text{start}}$.
- $h(n)$ is the heuristic cost function, which returns the estimated cost of shortest path from $n$ to $q_{\text{goal}}$.
- $f(n) = g(n) + h(n)$ is the estimated cost of shortest path from $q_{\text{start}}$ to $q_{\text{goal}}$ via $n$.

The algorithm can be found in algorithm 4.

**Algorithm 4 $A^*$ Algorithm**

**Input:** A graph  
**Output:** A path between start and goal nodes

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

The $A^*$ algorithm searches for a path from the start to the goal. In such a case, the $g$ function is sometimes called the cost-to-come or cost-from-start function. If the search were to occur in reverse, from goal to start, then the $g$ function is called the cost-to-go function which measures the path cost to the goal. Likewise, the heuristic then becomes an estimated cost of the shortest path from the current node to the start. The objective function $f$ is still the sum of $g$ and $h$. 
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, 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 the 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.

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

5. \( O = \{3, G\}; C = \{S, 2, 4, 1\} \) (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 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 cells, 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 cell to an obstacle cell is made to be arbitrarily high; we chose 10000. So one cell step from a free space to an obstacle cell along a vertical or horizontal direction costs 10000 and one cell 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 to the heuristic. Thus far, in figure H.15 the back pointers and priorities have not been set.

The start cell 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 pri-
### H.2. A* ALGORITHM

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>h=6</strong></td>
<td>h = 6</td>
<td>h = 3</td>
<td>h = 2</td>
<td>h = 1</td>
<td>h = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>f = b()</strong></td>
<td>f = b()</td>
<td>f = b()</td>
<td>f = b()</td>
<td>f = b()</td>
<td>f = b()</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>h=5.4</strong></td>
<td>h = 5.4</td>
<td>h = 5.8</td>
<td>h = 6.2</td>
<td>h = 6.6</td>
<td>h = 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>f = b()</strong></td>
<td>f = b()</td>
<td>f = b()</td>
<td>f = b()</td>
<td>f = b()</td>
<td>f = b()</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>h=6.8</strong></td>
<td>h = 6.8</td>
<td>h = 7.2</td>
<td>h = 7.6</td>
<td>h = 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>f = b()</strong></td>
<td>f = b()</td>
<td>f = b()</td>
<td>f = b()</td>
<td>f = b()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

### Figure H.16.

Eight-point connectivity and possible cost values.

<table>
<thead>
<tr>
<th>x9</th>
<th>x2</th>
<th>x3</th>
</tr>
</thead>
<tbody>
<tr>
<td>c(x1,x2)=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c(x1,x9)=1.4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x8</th>
<th>x1</th>
<th>x4</th>
</tr>
</thead>
<tbody>
<tr>
<td>c(x1,x8)=10000, if x8 is in obstacle, x1 is a freecell</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x7</th>
<th>x6</th>
<th>x5</th>
</tr>
</thead>
<tbody>
<tr>
<td>c(x1,x9)=10000.4, if x9 is in obstacle, x1 is a freecell</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 cells. 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 cells 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 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 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. We can address dynamic environments by initially invoking the $A^*$ algorithm to determine a path from start to goal, follow that path until an unexpected change occurs (see (4, 3) in figure H.21(left)) and then reinvoke the $A^*$ algorithm to determine a new path. This, however, can become quite inefficient if many cells are changing from obstacle to free space and vice versa. The $D^*$ algorithm was devised to “locally repair” the graph allowing for an efficient updated searching in dynamic environments, hence the term $D^*$ [376].

The $D^*$ algorithm (algorithm 5) uses the notation found in table H.1. Just like $A^*$, $D^*$ has a data structure called the open list $O$. However, $D^*$ may process states after they are removed from the open list, so $t(X)$ is used to classify states as $NEW$, $OPEN$, and $CLOSED$ to mean $X$ has
H.3. D* ALGORITHM

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
<th>f</th>
<th>h</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>9.0</td>
<td>7.0</td>
<td>()</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>9.0</td>
<td>6.2</td>
<td>()</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>7.6</td>
<td>6.6</td>
<td>(2,1)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>7.0</td>
<td>6.8</td>
<td>()</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>7.2</td>
<td>7.2</td>
<td>()</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6.8</td>
<td>6.8</td>
<td>()</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6.4</td>
<td>6.4</td>
<td>()</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>6.0</td>
<td>6.0</td>
<td>()</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5.8</td>
<td>5.8</td>
<td>()</td>
</tr>
</tbody>
</table>

*ALGORITHM*

The function $h(X)$ measures path cost from the goal to $X$; this is some-

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

**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 cells $(3,3)$, $(2,3)$, and $(4,2)$ are not displayed in the priority queue because they correspond to obstacles.

never been in $O$, $X$ is currently in 0, and $X$ was in $O$ but currently is not, respectively.
### APPENDIX H. GRAPH REPRESENTATION AND BASIC SEARCH

<table>
<thead>
<tr>
<th>State</th>
<th>Heuristic $h$</th>
<th>Function Cost $f$</th>
<th>Parent State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>2</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

#### Figure H.20. A nonoptimistic heuristic leads to a non-optimal path with $A^*$.

The $D^*$ algorithm uses the $k$ values to determine the priority of the nodes in the open list. The $k$ value for a particular node $X$ is the smallest $h$ value for that node since it was most recently inserted onto the open list.
H.3. $D^*$ ALGORITHM

<table>
<thead>
<tr>
<th>State</th>
<th>Functions</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>$X$</td>
<td>$c(X,Y)$</td>
</tr>
<tr>
<td>Current State</td>
<td>$X_c$</td>
<td>$r(X,Y)$</td>
</tr>
<tr>
<td>Neighbor State</td>
<td>$X_n$</td>
<td>$h(X)$</td>
</tr>
<tr>
<td>Start</td>
<td>$S$</td>
<td>$k(X)$</td>
</tr>
<tr>
<td>Goal</td>
<td>$G$</td>
<td>$t(X)$</td>
</tr>
<tr>
<td>List of all States</td>
<td>$L$</td>
<td>$b(X)$</td>
</tr>
<tr>
<td>Open List</td>
<td>$O$</td>
<td></td>
</tr>
</tbody>
</table>

Table H.1. Common notation used in $D^*$

This means if $X$ were inserted into the open list, if its $h$ value changes, then the $k$ value is the smallest of the $h$ values while $X$ remains in the open list. However, once $X$ is removed from the open list, if it is re-inserted, $k(X)$ starts off with the new $h$ value.

The significance of $k(X)$ is that it distinguishes raise states, those where $k(X) > h(X)$ from lower states, those where $k(X) = h(X)$. It is possible for obstacle nodes to be placed on the open list, but they will have high $k$ values meaning they will probably never be processed. After the initial search, the $k$-values are identical to the $h$ values.

We once again consider a directed graph where a node corresponds to a robot state and an edge connects the nodes of adjacent states. Note that two states may not physically lie side-by-side, so an edge really corresponds to two nodes for which a robot can traverse from one state $X$ to another state $Y$ for some edge cost $c(Y,X)$. It is possible that $c(X,Y) \neq c(Y,X)$.

After some initialization, $D^*$ performs a modified Dijkstra’s search using $\text{INIT-PLAN}$ (algorithm 6), which calls $\text{PROCESS-STATE}$ (algorithm 14) to expand nodes. The initial search starts by placing the goal node onto the open list and ends when the start node is labeled as closed. Once the search is complete, the robot begins to follow the optimal path to the goal. This happens in $\text{PREPARE-REPAIR}$ (algorithm 7) and $\text{REPAIR-REPLAN}$ (algorithm 8). Effectively, $\text{PREPARE-REPAIR}$ looks for changes in the environment that affects the costs among nodes within sensor range and if such a changes exist, $\text{REPAIR-REPLAN}$ locally repairs the graph by readjusting the backpointers, so a new optimal path can be found. If no change exists, $\text{REPAIR-REPLAN}$ simply directs the robot along the current optimal path.

The actual repairing of the backpointers and directed motion of the
Algorithm 5 $D^*$ 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 $X \in L$ do
2.   $t(X) = \text{NEW}$
3. end for
4. $h(G) = 0$
5. $\text{INSERT}(O, G, h(G))$
6. $X_c = S$
7. $P = \text{INIT} - \text{PLAN}(O, L, X_c, G)$ (algorithm 6)
8. if $P = \text{NULL}$ then
9.   Return (NULL)
10. end if
11. while $X_c \neq G$ do
12.   $\text{PREPARE} - \text{REPAIR}(O, L, X_c)$ (algorithm 7)
13.   $P = \text{REPAIR} - \text{REPLAN}(O, L, X_c, G)$ (algorithm 8)
14.   if $P = \text{NULL}$ then
15.     Return (NULL)
16.   end if
17.   $X_c$ = the second element of $P$ \{Move to the next state in $P$\}.
18. end while
19. Return $(X_c)$

Algorithm 6 $\text{INIT} - \text{PLAN}(O, L, X_c, G)$

**Input:** Open list $O$, List of all states $L$, Current Position $X_c$, Goal $G$

**Output:** A list of states to goal as described by back pointers in the list of states $L$; Open List $O$ is modified

1. repeat
2.   $k_{min} = \text{PROCESS} - \text{STATE}(O, L)$
3. until ($k_{min} = -1$) or ($t(X_c) = \text{CLOSED}$)
4. $P = \text{GET} - \text{BACKPOINTER} - \text{LIST}(L, X_c, G)$ (algorithm 9)
5. Return $(P)$

Robot occurs in $\text{REPAIR} - \text{REPLAN}$. Notice that $\text{INIT} - \text{PLAN}$ and $\text{REPAIR} - \text{REPLAN}$ look quite similar except the “until” terminating condition is different. This is because $\text{INIT} - \text{PLAN}$ only accesses one
part of \textit{PROCESS - STATE} since \( t(X) = \text{NEW} \) whereas \textit{REPAIR - REPLAN} uses all parts of \textit{PROCESS - STATE}. The repairing process terminates when \( k \) value of any node in the open list is greater than or equal to \( h(X_c) \). In other words, the process terminates when the minimum path cost from any node in the open list to the goal is greater than or equal to the the path cost from the current robot position \( X_c \) to the goal. This is the terminating condition in \textit{REPAIR - REPLAN} which is different from the terminating condition in \textit{INIT - PLAN}.

There is one piece of terminology which should be made clear at this point. In actuality, \( D^* \) is not considering actual path costs but rather the perceived path costs which are derived from the robot’s current understanding of the graph being searched. If the edges of the graph are incorrect, say due to a change in the environment, then the \( D^* \) algorithm does not use the updated information until the robot discovers a change in the environment, i.e., until the robot discovers a change in an edge cost. The \( D^* \) literature hence uses the term estimated path cost to reflect this, but such terminology could be confusing because a heuristic also estimates path cost and there are many other ways to estimate path cost. Therefore, this description of \( D^* \) avoids the use of this term but the reader should be aware of it when reading the literature. Later on, the focused \( D^* \) algorithm was developed to include a heuristic function for guiding, or focusing, the repairing process. Typically for \( D^* \), \( g \) is used as the heuristic function switching the \( g \) and \( h \) notation convention from \( A^* \). Often, when people speak of \( D^* \), they really mean focused \( D^* \), but this section focuses, no pun intended, on the original \( D^* \) algorithm.

At this point instead of directly explaining the details of \textit{PROCESS - STATE}, we give an example of the entire \( D^* \) algorithm. Consider the grid environment in figure H.21(left) which is identical to the one in figure H.15, except cell (4, 3) is a gate which can either be a free-space cell or an obstacle cell. Assume it starts as a free-space cell. Finally, a node in the search graph corresponds to a cell, regardless if it is obstacle or free space, and since this graph is directed, a pair of nodes corresponding to adjacent cells have two directed edges.

To achieve the initial Dijkstra-like search from the goal back to the start, the goal node is first placed on the open list (figure H.21(right)) with \( h = 0 \). It is then expanded (figure H.22(left)), adding (6, 6), (6, 5), and (7, 5) onto the queue (algorithm 14, lines 17-19). \( D^* \) increments the \( k \) and \( h \) values according to the metric described in figure H.16. Unless stated otherwise, when a node is expanded, it is automatically put on the closed list, so the goal is expanded and then put on the closed list.
Algorithm 7 PREPARE – REPAIR\((O, L, X_c)\)

**Input:** Open list \(O\), List of all states \(L\), Current Position \(X_c\)

**Output:** Open List \(O\) is modified

1: for each state \(X \in L\) within sensor range of \(X_c\) and \(X_c\) do
2: for each neighbor \(Y\) of \(X\) do
3: if \(r(Y, X) \neq c(Y, X)\) then
4: \(\textit{MODIFY – COST}(O, Y, X, r(Y, X))\)
5: end if
6: end for
7: for each neighbor \(Y\) of \(X\) do
8: if \(r(X, Y) \neq c(X, Y)\) then
9: \(\textit{MODIFY – COST}(O, X, Y, r(X, Y))\)
10: end if
11: end for
12: end for

Algorithm 8 REPAIR – REPLAN\((O, L, X_c, G)\)

**Input:** Open list \(O\), List of all states \(L\), Current Position \(X_c\), Goal \(G\)

**Output:** A list of states to goal as described by back pointers in the list of states \(L\); Open List \(O\) is modified

1: repeat
2: \(k_{\text{min}} = \textit{PROCESS – STATE}(O, L)\)
3: until \((k_{\text{min}} \geq h(X_c))\) or \((k_{\text{min}} = -1)\)
4: \(P = \textit{GET – BACKPOINTER – LIST}(L, X_c, G)\)
5: Return \(P\)

Next, node \((6, 6)\) is expanded adding nodes \((5, 6)\) and \((5, 5)\) onto the open list (figure H.22(right)). The node \((7, 5)\) is then expanded adding nodes \((6, 4)\) and \((7, 4)\) into the open list. (figure H.23(left)). More nodes are expanded until we arrive at node \((4, 6)\) (figure H.23(right)). When \((4, 6)\) is expanded, nodes \((3, 6)\) and \((3, 5)\), which are obstacle nodes, are placed onto the open list, but with high \(k\) and \(h\) values figure H.23(right). Since the \(h\) values of the expanded obstacle nodes are high, they will most likely never be expanded, which makes sense because they are obstacle nodes.

The Dijkstra-like search is complete when the start node \((2, 1)\) is expanded (figure H.24(left)). Note that some cells may not have been considered by the \(D^*\) algorithm. The optimal path from start to goal (assuming that the gate cell \((4, 3)\) is not an obstacle) is found by traversing the back-
H.3. $D^*$ ALGORITHM

Figure H.21. (Left) A cell world similar to figure H.15, except it has a gate, $h$ values and $k$ values. (Right) Put goal node on open list.

Figure H.22. (Left) Expand the Goal Node (Right) Node (6,6) is expanded.

pointers starting from the start node to the goal node (figure H.24(right)). The optimal path is $(2,1) \rightarrow (3,2) \rightarrow (4,3) \rightarrow (5,4) \rightarrow (6,5) \rightarrow (7,6)$. Note that nodes $(1,1), (1,2), (1,3), (2,3), (3,3), (3,4), (3,5), (3,6)$ and $(4,2)$ are still on the open list.

The robot then starts tracing the optimal path from the start node to the goal node. In figure H.25(left), the robot moves from node $(2,1)$ to $(3,2)$. When the robot tries to move from node $(3,2)$ to $(4,3)$, it finds that the
Figure H.23. (Left) Expand (7, 5). (Right) Expand (4, 6).

gate node (4, 3) is an obstacle (figure H.25(left)). In the initial search for an optimal path, we had assumed that the gate node was a free space node, and hence the current path is not feasible. At this stage, instead of re-planning for an optimal path from the current node (3, 2) to goal node, $D^*$ tries to make local repairs to the graph until a new optimal path is found.

There is one subtlety about the sensor range which should be noted. When we assume the robot has a sensor range of one cell, does this mean that the robot can see all neighboring cells, the current cell, and the edge costs among all such cells, or does this mean that it can see all such cells and all of the edges costs associated with these cells. If the latter, then some edges may be considered “out of sensor range” because such edge costs are between a cell within sensor range and a cell outside of sensor range. However, it is reasonable to expect such edge costs to change because the cell within sensor range has changed. In this example, we assume that the robot infers that the peripheral edge costs change as well. Either assumption is fine, as long as the implementation is consistent.

To address the fact that (4, 3) is now an obstacle, $D^*$ increases by a large number the transition cost to and from (4, 3) for all nodes adjacent to (4, 3). Actually, in our example, we simply set the transition cost to a high number, say 10,000. Next, all nodes affected by the increased transition costs (all eight neighbors and (4, 3)) are put on the open list (algorithm 7). Recall that $D^*$ is repairing a directed graph, so MODIFY – COST is called 16 times, once for each neighbor and eight times on (4, 3) but (4, 3) is put onto the open list only once. See figure H.25 (right). Note that some neighbors
of (4,3), and (4,3) itself have lower k values than most elements on the open list already. Therefore, these nodes will be popped first.

The node (5, 4) is now popped because its k value is the smallest. Since its k and h are the same, consider each neighbor of (5, 4). One such neighbor is (4, 3). The node (4, 3) has a back pointer which points to (5, 4) but its original h value is no longer the sum of the h value of (5, 4) and the transition cost, which was just raised due to the obstacle (algorithm 14, line 17). Therefore, (4, 3) is put on the open list but with a high h value (algorithm 14, line 19). Note that since (4, 3) is already on the open list, its k value remains the same and hence the node (4, 3) is a raise state because h > k. See figure H.26 (left). Next, (5, 3) is popped but this will not affect anything because none of the surrounding nodes are new, and the h values of the surrounding nodes are correct. A similar non-action happens for (4, 4). See figure H.26 (right).

Now, the node (4, 3) is popped off the open list and since k < h, the objective is to try to decrease the h value (algorithm 14, line 7). This is akin to finding a better path via (4, 3) to the goal, but this is not possible because (4, 3) is an obstacle. Looking more carefully at the algorithm, consider node (5, 3), which is a neighbor of (4, 3) and has an h value which is less than the k value of (4, 3) but h value of (4, 3) “equals” the sum of the h value of (5, 3) and the transition cost. This means that nothing is improved coming from (4, 3) to (5, 3). This is also true for (5, 4) and (4, 4). See figure H.27 (left). Note that our notion of equality is not precise in that any two “large” numbers are equal; so for example, 10000 = 10001.4.

So, we cannot find a path through any of the neighbors of (4, 3) to reduce h. Therefore, the node (4, 3) is expanded next, which places all nodes whose back pointers point to (4, 3), which in this case is only (3, 2), onto the open list with a high h value (algorithm 14, line 17). Now, (3, 2) is also a raise state. Note that the k value of (3, 2) is set to the minimum of its old and new h values (this setting happens in the insert function). Next, we pop (5, 2) but this will not affect anything because none of the surrounding nodes are new, and the h values of the surrounding nodes are correct. See figure H.27 (left).

Now, node (3, 2) is popped off the open list. Since k < h, \( D^* \) looks for a neighbor whose h value is less than the k value of (3, 2) (algorithm 14, line 9). If such a neighbor exists, then \( D^* \) would redirect the backpointer through this neighbor. However, no such neighbor exists.

Next, \( D^* \) looks for neighboring nodes whose back pointers point to (3, 2) and have an “incorrect” h value, i.e., all neighboring nodes with h values not equal to the h value of (3, 2) plus its associated transition cost. Such
nodes are also placed onto the open list with a high h value, making them raise states (algorithm 14, line 24). These are (3, 1), (2, 1), and (2, 2). Note that the k values of these nodes are set to the minimum of the new h value and the old h value.

Also, D* looks for neighboring nodes whose back pointer does not point to (3, 2), whose h value plus the transition cost is less than the h value of (3, 2), which is on the closed list, and whose h value is greater than the k value of (3, 2) (algorithm 14, line 29). The only such neighbor is (4, 1). This could potentially lead to a lower cost path. So, the neighbor (4, 1) is put on the open list with its current h value because it could potentially reduce the h value of (3, 2). It is called a lower state because h = k. See figure H.27 (right).

Continuing with the lowest k value node, the node (4, 1) is popped off the open list and expanded. Since the h and k values of (4, 1) are the same, D* considers the neighbors whose pack pointers do not point to (4, 1) to see if passing through (4, 1) reduces any of the neighbors h values (algorithm 14, line 17). This redirects the backpointers of (3, 2) and (3, 1) to pass through (4, 1); moreover, these nodes are then put onto the open list. However, since (3, 2) was “closed,” its new k value is the smaller of its old and new h values, making it a lower state (since k = h). Similarly, since (3, 1) was “open” (already on the open list), its new k value is the smaller of its old k value and its new h value. See figure H.28 (left).

Next, the node (3, 1) is popped off the open list. Since its k value 6.6 is less than its h value 7.2, D* looks for a neighbor whose h value is less than the k value of (3, 1) (algorithm 14, line 9). The only such neighbor is (4, 1). This gives us hope that there is a lower cost path through (4, 1). However, since the sum of the transition cost to (4, 1) and the h value of (4, 1) is greater than the h value of (3, 1), no such improved path exist and nothing happens. However, the node (3, 1) can be used to form a reduced cost path for its neighbors, so (3, 1) is put back on the open list but with a k value set to the minimum of its old h value and new h value. Thus, it now also becomes a lower state. See figure H.28 (right).

The node (2, 2) is then popped off the open list and expanded. This increases the h values of the nodes that pass through (2, 2) and puts them back on the open list. When the nodes (1, 1), (1, 2) and (1, 3) are put back onto the open list, their k values are unaffected, hence their position in the open list remains the same, but their h values are increased making them raise states (algorithm 14, line 24). See figure H.29 (left). Next the node (2, 1) is popped off the queue and since k < h and it cannot reduce the cost to any of its neighbors, so this has no effect. See figure H.29 (right).
Now, the node (3,1) is popped off the open list and expanded. This has the effect of redirecting the back pointers of (2,2) and (2,1) through (3,1) and putting them back on the open list with a $k$ value equal to the minimum of the old and new $h$ values (algorithm 14, line 17). Because $k$ equals $h$, they are now lower states. See figure H.30 (left). Now, $k_{\text{min}} = h(X_c)$ which is the $h$-value of the current robot position, the terminating condition of $\text{REPAIR} - \text{REPLAN}$ (algorithm 8). Note that $X_c$ is still on the open list and this should not be a concern because even if $X_c$ were popped off of the open list, no improvement can be made because the current path cost $h$ is already optimal. Finally, the new path is determined via gradient descent of the $h$ values (figure H.30 (right)), and then the robot follows the path to the goal (figure H.31).

**Algorithm 9** $\text{GET} - \text{BACKPOINTER} - \text{LIST}(L,S,G)$

- **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 back-pointers in the list of states $L$

1. if path exists then
2. Return (The list of states)
3. else
4. Return (NULL)
5. end if

**Algorithm 10** $\text{INSERT}(O,X,h_{new})$

- **Input:** Open list, a state, and an $h$-value
- **Output:** Open list is modified

1. if $t(X) = \text{NEW}$ then
2. $k(X) = h_{new}$
3. else if $t(X) = \text{OPEN}$ then
4. $k(X) = \min(k(X), h_{new})$
5. else if $t(X) = \text{CLOSED}$ then
6. $k(X) = \min(h(X), h_{new})$
7. end if
8. $h(X) = h_{new}$
9. $t(X) = \text{OPEN}$
10. Sort $O$ based on increasing $k$ values
Algorithm 11 \textit{MODIFY} – \textit{COST}(O, X, Y, cval)
\begin{itemize}
\item \textbf{Input:} The open list, two states and a value
\item \textbf{Output:} A $k$-value and the open list gets updated
\end{itemize}
1: $c(X, Y) = cval$
2: if $t(X) = CLOSED$ then
3: \hspace{1em} INSERT($O, X, h(X)$)
4: end if
5: Return $GET – KMIN(O)$ (algorithm 13)

Algorithm 12 \textit{MIN – STATE}(O)
\begin{itemize}
\item \textbf{Input:} The open list $O$
\item \textbf{Output:} The state with minimum $k$ value in the list related values
\end{itemize}
1: if $O = \emptyset$ then
2: \hspace{1em} Return $(-1)$
3: else
4: \hspace{1em} Return ($\arg \min_{Y \in O} k(Y)$)
5: end if

Algorithm 13 \textit{GET – KMIN}(O)
\begin{itemize}
\item \textbf{Input:} The open list $O$
\item \textbf{Output:} Lowest $k$-value of all states in the open list
\end{itemize}
1: if $O = \emptyset$ then
2: \hspace{1em} Return $(-1)$
3: else
4: \hspace{1em} Return ($\min_{Y \in O} k(Y)$)
5: end if
### Figure H.24

(Left) Termination of initial search phase: start cell is expanded. (Right) Tracing backpointers yields the optimal path, or is it?

![Figure H.24](image)

### Figure H.25

(Left) The robot physically starts tracing the optimal path. (Right) The robot cannot trace the assumed optimal path: gate (4,3) prevents passed. All nodes surrounding (4,3) are put on the open list.

![Figure H.25](image)
Figure H.26. (Left) Pop (5, 4) off of the open list and expand; node (4, 3) becomes a raise state. (Right) Pop (5, 3) off of open list but this has no effect.

Figure H.27. (Left) Pop (4, 3) off of open list, and try to find a better path through it; none exist. Eventually (3, 2) is put on the open list as a raise state. (Right) Pop (3, 2) off of open list.
Figure H.28. (Left) Pop (4, 1) off of open list (Right) Pop (3, 1) off of open list.

Figure H.29. (Left) Pop (2, 2) off the queue and expand it (Right) Pop (2, 1) off of open list.
Figure H.30. (Left) Pop (3,1) off the queue and expand it. (Right) Determine optimal path from the current location to the goal by following gradient of h values.

Figure H.31. Robot moves to goal from its current location.
**Algorithm 14 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 12)
2: if $X = NULL$ then
3: Return (−1)
4: end if
5: $k_{old} = GET – KMIN(O)$ (algorithm 13)
6: DELETE($X$)
7: if $k_{old} < h(X)$ then
8: for each neighbor $Y \in L$ of $X$ do
9: if $t(Y) \neq NEW$ and $h(Y) \leq k_{old}$ 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: end if
15: if $k_{old} = h(X)$ then
16: for each neighbor $Y \in L$ of $X$ do
17: 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
18: $b(Y) = X$
19: INSERT($O, Y, h(X) + c(X,Y)$) (algorithm 10)
20: end if
21: end for
22: else
23: for each neighbor $Y \in L$ of $X$ do
24: if ($t(Y) = NEW$) or ($b(Y) = X$ and $h(Y) \neq h(X) + c(X,Y)$) then
25: $b(Y) = X$
26: INSERT($O, Y, h(X) + c(X,Y)$)
27: end if
28: if ($b(Y) \neq X$ and $h(X) > h(Y) + c(X,Y)$) then
29: INSERT($O, X, h(X)$)
30: end if
31: end if
32: end for
33: end if
34: Return $GET – KMIN(O)$ (algorithm 13)
H.4 D* Lite

The D* Lite algorithm is perhaps an easier-to-understand advancement over the D* approach and is therefore more often used. One need not know the details of D* to understand D* Lite, however we make some comparisons here for the sake of explanation. Just like D*, D* Lite has the effect of locally repairing the graph when a change occurs. However, D* Lite does not have any back pointers to determine a path; instead each D* Lite node contains additional values: an objective function $g$ and a “look ahead” function $rhs$. In general, $g$ is a type of cost-to-goal function.

Nodes are called consistent if their $g$ and $rhs$ values are the same and likewise are inconsistent if their $g$ and $rhs$ functions differ. If $g > rhs$, then a node is over-consistent and if $g < rhs$, the a node is under-consistent. This notion of consistency is analogous to the raise and lower states of D*.

Finally, there is a heuristic function $h$, which has the same meaning as $h$ from A* and is therefore different from the $h$ in D*.

The graph being search is assumed to be a directed graph where $c(u, v)$ is the cost to traverse a directed edge from the source node $u$ to the destination $v$ ($D*$ defined $c(u, v)$ to be the cost from $v$ to $u$). Hence, the $Succ(u)$ and the $Pred(u)$ are the successors and predecessors, respectively, of the node $u$. With these terms in-hand, the $rhs$ function is defined as

$$rhs(u) = \min_{s' \in Succ(u)} (c(u, s') + g(s'))$$

There is an open list (again sometimes called a priority queue) $U$ whose key $k$ is now a two-vector, as opposed to the real $k$ values as before. This key is

$$k(s) = \left[ \min(g(s), rhs(s)) + h(s_{start}, s) \right]$$

Sometimes, the first and second components of the key are called the primary key and secondary key, respectively. If the primary key of $u$ is less than the primary key of $v$, then $k(u) < k(v)$. If the primary keys are equal, then the secondary keys are used as a tie breaker.

The D* Lite algorithm (algorithm 15) takes as input a directed graph, the edge costs (which can be modified), a start state and a goal state. In general, after an initial search, D* Lite determines a path by performing gradient descent on the sum of the edge costs and objective function $g$, i.e., chose the next node whose $g$ value in addition to the cost to get there is smallest. This procedure terminates either when the robot reaches the goal or detects a change in edge cost. At this point, the edge costs are repaired.
and the rhs and g values are updated to reflect these changes. Once all of the changes are computed, D* Lite continues with gradient descent to follow the optimal path. Ultimately, algorithm 15 does not output anything, per say, but the search graph is updated so that the rhs and g values are assigned appropriately.

Now, let us take a closer look at algorithm 15. Assume without loss of generality that the start and goal nodes initially differ, so the objective is to move from the start to the goal. First, algorithm 16 initializes the open list to empty, sets all of the rhs and g values to infinity, assigns the rhs value of the goal to zero, and places the goal on the open list. This makes the goal inconsistent, specifically over-consistent. Therefore, when ComputeShortestPath (algorithm 17) is called, the highest priority element, i.e., the lowest key value, of the open list is the goal. Naturally, the goal’s key value is less than the key value of the start. ComputeShortestPath then makes the goal consistent by setting the g value equal to its rhs value and for all nodes with outgoing edges that terminate at the goal, each node has its rhs value updated. This process repeats until the start node is consistent and the top key on the open list is not less than the key of the start node. At this point, the loop terminates and an optimal path can be determined.

While the current and goal nodes are not the same, the robot moves from the current node toward the goal. At each step, the planner directs the robot to the successor node whose g value summed with cost to traverse the edge to that successor node is minimal over all successor nodes, i.e., from u, chose the next node such that \( c(u, s') + g(s') \) is minimized over all \( s' \in \text{Succ}(u) \).

In the process of following the optimal path to the goal, if there are any changes in the graph, or more specifically if there are any changes in the graph within sensor range of the robot, D* Lite first updates these edge costs, updates the source nodes of the affected edges, updates the keys of the appropriate nodes in the open list, and then calls ComputeShortestPath again to make the appropriate nodes consistent. This last action has the ultimate effect of locally repairing the optimal path by altering the g and rhs values. This entire process continues until the current and goal states are the same.

The ComputeShortestPath (algorithm 17) does nothing unless the start node is inconsistent or the lowest priority node in the open list has a key value less than the start’s key value. If this is the case, the lowest priority state u is popped off the open list. If it is over-consistent, ComputeShortestPath makes u consistent and updates all of the nodes with edges terminating at u. If u is under-consistent, then ComputeShortestPath makes u over-
Algorithm 15 $D^*\text{Lite}(S,s_{start},s_{goal})$

**Input:** A graph of nodes $S$ and two nodes (start and goal)
**Output:** A modified graph of nodes $S$ with their $k$ and $rhs$ values properly set.

1: $s_{current} = s_{start}$
2: $Initialize()$ (algorithm 16)
3: $ComputeShortestPath()$ (algorithm 17)
4: while $s_{current} \neq s_{goal}$ do
5: if $g(s_{current}) = \infty$ then
6: Break (No path exists)
7: end if
8: $s_{current} = \arg\min_{s' \in \text{Succ}(s_{current})} (c(s_{current}, s') + g(s'))$
9: Move to $s_{current}$
10: Scan graph for any changed costs (within sensor limits)
11: if any edge cost changed then
12: for each directed edge $(u, v)$ with changed cost do
13: Update edge cost $c(u, v)$
14: $UpdateVertex(u)$
15: end for
16: for each $s \in U$ do
17: $Update(U, s, CalculateKey(s))$
18: end for
19: $ComputeShortestPath()$
20: end if
21: end while

Algorithm 16 $Initialize()$

**Input:** The start $s_{start}$, $s_{current}$, goal $s_{goal}$, open list $U$ and graph of states $S$ are global variables.
**Output:** A modified open list $U$ and modified graph of states $S$ with updated $rhs$ and $g$ values.

1: $U = \emptyset$
2: for each $s \in S$ do
3: $rhs(s) = g(s) = \infty$
4: end for
5: $rhs(s_{goal}) = 0$
6: $Insert(U, s_{goal}, CalculateKey(s_{goal}))$
Algorithm 17 ComputeShortestPath()

Input: The start $s_{\text{start}}$, $s_{\text{current}}$, goal $s_{\text{goal}}$, open list $U$ and graph of states $S$ are global variables.

Output: A modified open list $U$ and modified graph of states $S$ with updated $rhs$ and $g$ values.

1: while $(\text{TopKey}(U) < \text{CalculateKey}(s_{\text{current}}))$ or $(\text{rhs}(s_{\text{current}}) \neq g(s_{\text{current}}))$ do
2: $u = \text{Pop}(U)$
3: if $g(u) > \text{rhs}(u)$ then
4: $g(u) = \text{rhs}(u)$
5: for each $s \in \text{Pred}(u)$ do
6: $\text{UpdateVertex}(s)$
7: end for
8: else
9: $g(u) = \infty$
10: for each $s \in \text{Pred}(u) \cup \{u\}$ do
11: $\text{UpdateVertex}(s)$
12: end for
13: end if
14: end while

consistent and updates $u$, as well as all nodes with edges terminating at $u$. The nodes are updated in algorithm 18 and the keys are calculated in algorithm 19.

Algorithm 18 UpdateVertex($u$)

Input: A node $u$, and the start $s_{\text{start}}$, $s_{\text{current}}$, goal $s_{\text{goal}}$, and open list $U$ are global variables.

Output: A modified node $u$ and a modified open list $U$.

if $u \neq s_{\text{goal}}$ then
$rhs(u) = \min_{s' \in \text{Succ}(u)} (c(u, s') + g(s'))$
end if
if $u \in U$ then
$\text{Remove}(U, u)$
end if
if $g(u) \neq rhs(u)$ then
$\text{Insert}(U, u, \text{CalculateKey}(u))$
end if
Algorithm 19 CalculateKey(s)

**Input:** A node s.

**Output:** A key for s.

Return \((\min(g(s), rhs(s))) + h(s_{current}, s); \min(g(s), rhs(s)))\)

---

**Figure H.32.** Eight connected grids with pairwise directed edges between neighboring cells.

Let’s consider the example where \(D^*\) Lite searches a grid of cells. Each node in the graph corresponds to a cell and each pair of neighboring nodes \(u\) and \(v\) has a pair of directed edges: one from \(u\) to \(v\) and visa versa. The cost to travel from one free cell to a neighboring free cell is 1 if it is an up, down, left or right motion, and is 1.4 if it is a diagonal motion. The cost of travel either from an obstacle cell or to an obstacle cell infinite, as depicted in figure H.32.

Initially, all of the nodes’ \(rhs\) and \(g\) values are set to infinity, except for the goal whose \(rhs\) value is set to zero and its \(g\) value is set to infinity. Since the goal is now inconsistent, it is put on the open list. See figure H.33.

\(D^*\) Lite then calls \(ComputeShortestPath\) which immediately pops the goal off of the open list, and since it is over-consistent, makes it consistent with \(rhs\) and \(g\) values of zero. Now, \(ComputeShortestPath\) expands the popped node by calling \(UpdateVertex\) on all of its predecessors. This computes \(rhs\) values for the predecessors and puts them on the open list, but only if they become inconsistent. Node (1,1) is a predecessor but not put on the open list because it remained consistent. See figure H.34 where the small arrows indicate which node is used to compute the \(rhs\) value, e.g., the
**Figure H.33.** (Left) Goal rhs value set to zero and all other rhs and g values to infinity (Right) Goal is put on open list. Arrows are not back pointers but rather represent gradient directions.

**Figure H.34.** (Left) The goal is popped of the open list. (Right) The goal is expanded and the resulting inconsistent nodes are put the on open list.

rhs value of (0, 1) is computed using the g value of (0, 0) and the transition cost from (1, 0) to (0, 0), i.e., $1 = 0 + 1$.

Continue in ComputeShortestPath by popping (0, 1), which is tied with (1, 0) for the minimum node in the open list. Here, UpdateVertex is called on all of the predecessors of the popped node. When UpdateVertex is called...
on each predecessor node, the \( rhs \) value of each predecessor is updated by examining the \( g \)-values of each of the predecessor’s successors in the graph. Since, \((0, 1)\) is over-consistent, it is made consistent and all predecessors of \((0, 1)\) have their \( rhs \) values updated via \( \text{UpdateVertex} \). Two of its predecessors become inconsistent and are put on the open list. Again, the \( rhs \) values of the predecessors \((0, 0)\) and \((1, 1)\) did not change, and as such, did not become inconsistent and are not put on the open list. The \( rhs \) value of \((1, 0)\) did not also did not change but was already inconsistent and on the open list. See figure H.35. Now, \((1, 0)\) is expanded but no predecessors of \((1, 0)\) are put on the open list because they remained consistent after calling \( \text{UpdateVertex} \).

This Dijkstra-like search continues until the start node is effectively expanded and made consistent. In figure H.36, \((3, 1)\) is popped and expanded, and all of its predecessors, which become inconsistent, are put on the open list. Note that the start was already on the open list. At this point, the start has the lowest key value, so is popped off the open list and made consistent. In this case, none of the predecessors of the start become inconsistent, so are not put on the open list, although some were already on the open list. At this point, since the start node is consistent and the top key on the open list is not less than the key of the start node, an optimal path exists. This allows the \( \text{ComputeShortestPath} \) loop to terminate. Finally, note that some
nodes remain on the open list and for examples with larger graphs, some nodes may not have been considered at all.

The robot then follows the negated gradient of $g$ from the start to the goal until the robot detects an obstacle at $(2, 2)$. See figure H.37. The algorithm dictates that for all directed edges $(u, v)$ with changed edge costs, $UpdateVertex(u)$ is called. Since the edges are directed and in this example all neighboring cells $u$ and $v$ have two edges, one from $u$ to $v$ and visa versa, $(2, 2)$ has 16 affected edges. See figure H.38.

Let’s consider the outgoing and incoming edges to $(2, 2)$ separately. For each of the outgoing edges, $UpdateVertex$ is called on $(2, 2)$. First, the outgoing edge to $(2, 3)$ is called. Since the edge cost is now infinite, the $rhs$ value of $(2, 2)$ is raised to infinity making it inconsistent and hence $(2, 2)$ is put on the open list. Now, when $UpdateVertex$ is called for the rest of the outgoing edges of $(2, 2)$, nothing happens because $(2, 2)$ remains inconsistent.

Now, consider the incoming edges to $(2, 2)$. One of the predecessors of the incoming edge to $(2, 2)$ is $(3, 3)$, so $UpdateVertex$ is called on $(3, 3)$. The minimum possible $rhs$ value of $(3, 3)$ is still 4.8, but this value is based on the $g$ value of $(2, 3)$, not $(2, 2)$. The node $(3, 3)$ is still consistent, so it is not put on the open list. Another incoming edge to $(2, 2)$ comes from $(3, 2)$, so $UpdateVertex$ is called on this node. Since the transition cost to $(2, 2)$ increased, the minimum possible $rhs$ value of $(3, 2)$ is now 5.2, computed
APPENDIX H. GRAPH REPRESENTATION AND BASIC SEARCH

Figure H.37. (Left) Follow optimal path via gradient descent of $g$. (Right) The robot discovers that (2, 2) is an obstacle.

Figure H.38. (Left) Outgoing edges to (2, 2). (Right) Incoming edges to (2, 2).

based on the $g$ value of (2, 3) where $5.2 = 3.8 + 1.4$. See figure H.39.

Another incoming edge to (2, 2) comes from (3, 1). The minimum possible \textit{rhs} value of (3, 1) is now 5.4, computed based on the $g$ value of (3, 2). Again, note that the \textit{rhs} value of a node is always computed using the $g$, not a \textit{rhs}, values of its successors. The remaining five nodes – (1, 1), (1, 2), (1, 3), (2, 3) and (2, 1) – remain consistent and hence are not put on the open list. See figure H.40.
Note that the processing order of (3, 2) versus (3, 1) does not matter because when (3, 2) is processed, its \textit{rhs}-value, not its \textit{g}-value, is updated. Then, when (3, 1) is updated, its \textit{rhs}-value is updated and is based on the \textit{g}-value of (3, 2), which has not changed, and not its \textit{rhs} value. As such, we will get the same effect whether we process (3, 2) before (3, 1) or vice versa.

Now, D* Lite goes back to \textit{ComptShortestPath} until a new optimal path is found. Note that the current robot position is inconsistent and does not have the smallest key value in the open list. This indicates an optimal
path, based on all available information, has not been determined. The node with the minimum key is (2, 2). It is under-consistent, so its $g$ value is set to infinity and $UpdateVertex$ is called on all of its predecessors. This does not make any of the predecessors inconsistent, so none are put on the open list. See figure H.41.

Next, (3, 2) is popped and it is under-consistent, so its $g$ is set to infinity. Next, its predecessors are updated: (4, 2) becomes inconsistent, (3, 1) is updated but remains inconsistent, (4, 1) remains inconsistent but its $rhs$ value does not change and is now computed from the $g$ value of (3, 1). Also, (3, 2) is updated, remains inconsistent and is put back on the open list. See figure H.42.

Still in the $ComputeShortestPath$ procedure, (3, 1) is popped off the open list and since it is under-consistent, its $g$ value is made infinite and its predecessors are updated: (4, 1) is updated and remains inconsistent, while (3, 0) and (4, 0) are updated but are now consistent since both $g$ and $rhs$ are infinite. See figure H.43.

Also, since (3, 1) is under-consistent, $ComputeShortestPath$ path calls $UpdateVertex$ on (3, 1), which results in putting (3, 1) back on the open list since it remains inconsistent. Now, (3, 2) has the smallest key value, so it is popped off of the open list and since it is over-consistent, its $g$ value is set to its $rhs$ value. See figure H.44. When its predecessors are updated, (3, 1) is modified but still remains inconsistent, so it stays on the open list. See figure H.45 (left).
Figure H.42. Expand (3, 2) and update its predecessors and it.

Figure H.43. (Left) Node (3, 1) is popped. (Right) The predecessor nodes of (3, 1) are updated.

Once again, the node corresponding to the robots current position is consistent and the top key on the open list is not less than the key of current position. Therefore, a new optimal path has been found and ComputeShortestPath breaks out of its loop. Once again, the optimal path is determined by following the gradient of $g$. See figure H.45.
Figures H.44. (Left) Node (3, 1) remains on the open list (Right) Node (3, 2) is expanded.

Figures H.45. (Left) Update the predecessors of (3, 2) (Right) A new optimal path has been found.

**H.5 A Comment on Reverse Searching**

The search starts at the goal, and works backward, for good reason. After the initial search, the back pointers form a search tree which is rooted at the goal. Bearing in mind that the robot detects changes in edge costs near its current position, one could easily see that if the root of the search tree were the start node, then the search tree would have be drastically recomputed. With the goal node at the root of the tree, when an edge cost changes,
only a subtree that contains the current robot position is repaired where part of that subtree will be redirected to nodes in a neighboring subtree. In fact, in general, the entire subtree is not repaired; the repairing process terminates when an optimal path from the current robot position to the goal is determined.

A change in the environment is just one reason why a robot may need to replan its path. Another has to do with the stochastic nature of the robot’s motion. For example, error in control or unforeseen slippage may cause the robot to fall off its intended path. The benefit of performing the reverse search is that for small perturbations, an optimal path to the goal for nearby nodes was already computed during the initial Dijkstra-like search. In fact, one can 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.
Appendix I

Statistics Primer

- On Pg 548, the $\land$ should be a $\cap$. So
  \[ Pr(E_1 \cap E_2) = Pr(E_1) \cdot Pr(E_2) \]
  should read
  \[ Pr(E_1 \cap E_2) = Pr(E_1) \cdot Pr(E_2) \]
- On pg 549, cumulative is misspelled, twice.
Appendix J

Linear Systems and Control

- The dot over $x(k + 1)$ should not be there in eq (J.10). So the $\dot{x}(k + 1)$ should read $x(k + 1)$. 
Bibliography


on Research in Computational Molecular Biology, pages 12–21, April 2002.


BIBLIOGRAPHY


[404] S. Wilmarth, N. M. Amato, and P. Stiller. MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space.


