10 *Metric Path Planning*

Chapter objectives:

- Define *Cspace, path relaxation, digitization bias, subgoal obsession, termination condition.*
- Explain the difference between graph and wavefront planners.
- Represent an indoor environment with a *generalized Voronoi graph*, a *regular grid*, or a *quadtree*, and create a graph suitable for path planning.
- Apply the A* search algorithm to a graph to find the optimal path between two locations.
- Apply wavefront propagation to a regular grid.
- Explain the differences between continuous and event-driven replanning.

10.1 Objectives and Overview

QUANTITATIVE NAVIGATION

Metric path planning, or *quantitative navigation*, is the opposite of topological navigation. As with topological methods, the objective is to determine a path to a specified goal. The major philosophical difference is that metric methods generally favor techniques which produce an optimal, according to some measure of best, while qualitative methods seem content to produce a route with identifiable landmarks or gateways. Another difference is that metric paths are usually decompose the path into subgoals consisting of *waypoints*. These waypoints are most often a fixed location or coordinate (x,y). These locations may have a mathematical meaning, but as will be seen with meadow maps, they may not have a sensible correspondence to objects or

WAYPOINTS

landmarks in the world. Topological navigation focused on subgoals which are gateways or locations where the robot could change its primary heading.

The terms "optimal" and "best" have serious ramifications for robotics. In order to say a path is optimal, there is an implied comparison. As will be seen, some metric methods are able to produce an optimal path because they consider all possible paths between points. This can be computationally expensive. Fortunately, some algorithms (especially one named "A*" for reasons that will be discussed later) are more clever about rejecting non-optimal paths sooner than others.

Surprisingly, an optimal path may not appear optimal to the human eye; for example, a mathematically optimal path of a world divided into tiles or grids may be very jagged and irregular rather than straight. The ability to produce and compare all possible paths also assumes that the planning has access to a pre-exisiting (or *a priori*) map of the world. Equally as important, it assumes that the map is accurate and up to date. As such, metric methods are compatible with deliberation, while qualitative methods work well with more reactive systems. As a deliberative function, metric methods tend to be plagued by the same sorts of difficulties that were seen in Hierarchical systems: challenges in world representation, handling dynamic changes and surprises, and computation complexity.

COMPONENTS OF METRIC PATH PLANNERS

Metric path planners have two components: the *representation* (data structure) and the *algorithm*. Path planners first partition the world into a structure amenable for path planning. They use a variety of techniques to represent the world; no one technique is dominant, although regular grids appear to be popular. The intent of any representation is to represent only the salient features, or the relevant configuration of navigationally relevant objects in the space of interest; hence the term configuration space. Path planning algorithms generally work on almost any configuration space representation, although as with any algorithm, some methods work better on certain data structures. The algorithms fall into two broad categories: those which treat path planning as a graph search problem, and those which treat path planning as a graphics coloring problem. Regardless of what algorithm is used, there is always the issue in a Hybrid architecture of when to use it. This is sometimes called the issue of interleaving reaction and planning.

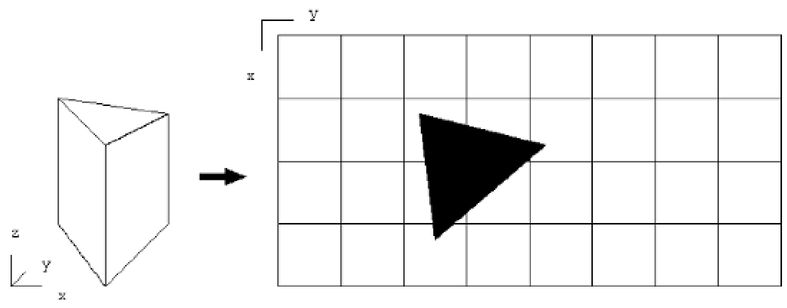


Figure 10.1 Reduction of a 6DOF world space to a 2DOF configuration space.

10.2 Configuration Space

CONFIGURATION SPACE

The physical space robots and obstacles exist in can be thought of as the world space. The *configuration space*, or *Cspace* for short, is a data structure which allows the robot to specify the position (location and orientation) of any objects and the robot.

DEGREES OF FREEDOM

A good Cspace representation reduces the number of dimensions that a planner has to contend with. Consider that it takes six dimensions (also called *degrees of freedom* or DOF) to represent precisely where an object is. A person may specify the location of the object as a (x, y, z) coordinate in some frame of reference. But an object is three-dimensional; it has a front and back, top and bottom. Three more degrees are needed to represent where the front of the chair is facing, whether it is tilted or not, or even upside down. Those are the Euler (pronounced "Oiler") angles, ϕ, θ, γ, also known as pitch, yaw, and roll.

Six degrees of freedom is more than is needed for a mobile ground robot in most cases for planning a path. The z (height) coordinate can be eliminated if every object the robot sits on the floor. However, the z coordinate will be of interest if the robot is an aerial or underwater vehicle. Likewise, the Euler angles may be unnecessary. Who cares which way the robot is facing if all the robot wants to do is to plan a path around it? But the pitch of a planetary rover or slope of an upcoming hill may be critical to a mission over rocky terrain.

Fig. 10.1 shows a transformation of an object into Cspace. In general, metric path planning algorithms for mobile robots have assumed only two DOF, including for the robot. For path planning purposes, the robot can be modeled as round so that the orientation doesn't matter. This implicitly assumes

HOLONOMIC

the robot is *holonomic*, or can turn in place. Many research robots are sufficiently close to holonomic to meet this assumption. However, robots made from radio-controlled cars or golf carts are most certainly not holonomic. Much work is being done in non-holonomic path planning, where the pose of the robot must be considered (i.e., can it actually turn that sharply without colliding?), but there no one algorithm appears to be used in practice. This chapter will restrict itself to holonomic methods.

10.3 Cspace Representations

The number of different Cspace representations is too large to include more than a coarse sampling here. The most representative ones are Voronoi diagrams, regular grids, quadtrees (and their 3D extension, octrees), vertex graphs, and hybrid free space/vertex graphs. The representations offer different ways of partitioning free space. Any open space not occupied by an object (a wall, a chair, an un-navigable hill) is free space, where the robot is free to move without hitting anything modeled. Each partition can be labeled with additional information: "the terrain is rocky," "this is off-limits from 9am to 5am," etc.

10.3.1 Meadow maps

Many early path planning algorithms developed for mobile robots assumed that the robot would have a highly accurate map of the world in advance. This map could be digitized in some manner, and then the robot could apply various algorithms to convert the map to a suitable Cspace representation. An example of a Cspace representation that might be used with an *a priori* map is the *meadow map* or hybrid vertex-graph free-space model.

MEADOW MAP

Meadow maps transform free space into convex polygons. Convex polygons have an important property: if the robot starts on the perimeter and goes in a straight line to any other point on the perimeter, it will not go out of the polygon. The polygon represents a safe region for the robot to traverse. The path planning problem becomes a matter of picking the best series of polygons to transit through. Meadow maps are not that common in robotics, but serve to illustrate the principles of effecting a configuration space and then planning a path over it.

The programming steps are straightforward. First, the planner begins with a metric layout of the world space. Most planners will increase the size of every object by the size of the robot as shown in Fig. 10.2; this allows the

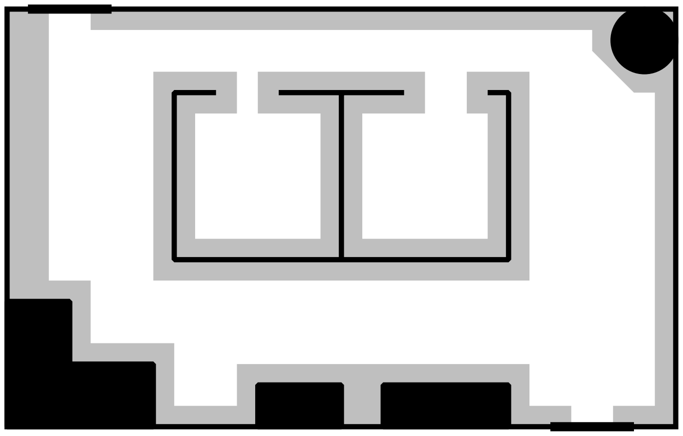


Figure 10.2 A *a priori* map, with object boundaries "grown" to the width of the robot (shown in gray).

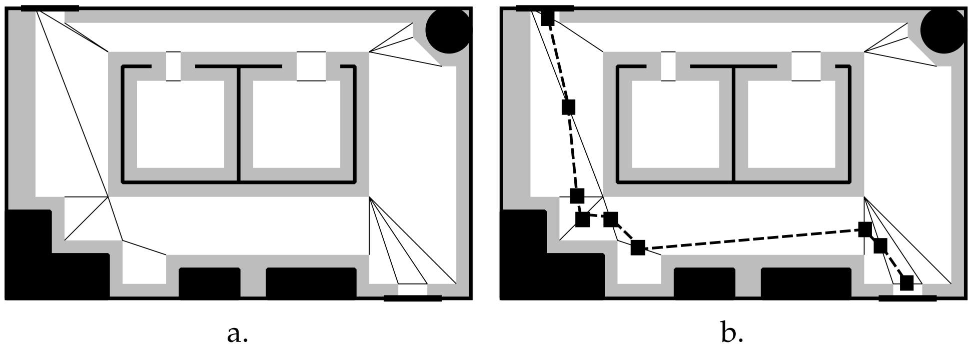


Figure 10.3 Meadow maps: a.) partitioning of free space into convex polygons, and b.) generating a graph using the midpoints.

planner to treat the robot as if it were a point, not a 2D object. Notice how from the very first step, path planners assume holonomic vehicles.

The next step in the algorithm is to construct convex polygons by considering the line segments between pairs of some interesting feature. In the case of indoor maps, these are usually corners, doorways, boundaries of objects, etc. The algorithm can then determine which combination of line segments partitions the free space into convex polygons.

The meadow map is now technically complete, but it is not in a format that supports path planning. Each convex polygon represents a safe passage for the robot. But there is still some work to be done. Some of the line segments

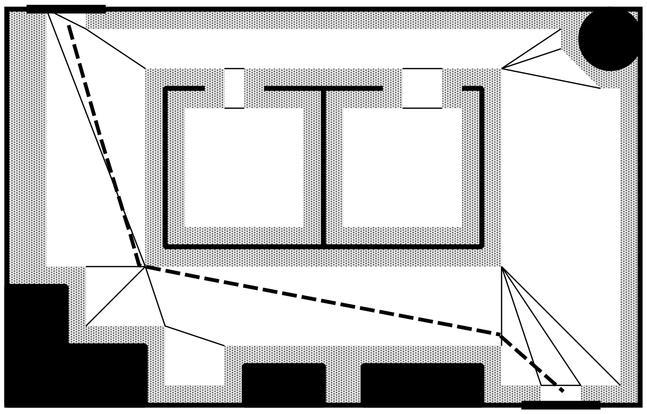


Figure 10.4 String tightening as a relaxation of an initial path.

forming the perimeter aren't connected to another a polygon (i.e., they are part of a wall), so they should be off limits to the planning algorithm. Also, as can be seen by the above figure, some of the line segments are quite long. It would make a difference to the overall path length where the robot cuts across the polygon. It is hard for the planner to discretize a continuous line segment. So the issue becomes how to specify candidate points on the polygon. One solution is to find the middle of each line segment which borders another polygon. Note that each of these midpoints becomes a node, and if edges are drawn between them, an undirected graph emerges. A path planning algorithm would determine the best path to take.

One disadvantage of a meadow map, indeed of any Cspace representation, is evident on inspection of Fig. 10.3: any path which is chosen will be somewhat jagged. Each of the inflection points is essentially a waypoint. One outcome of the partitioning process is that the free space is divided up in a way that makes sense geometrically, but not necessarily for a robot to actually travel. Why go halfway down the hall, then angle off? This may be mathematically optimal on paper, but in practice, it seems downright silly. Chuck Thorpe at CMU devised a solution to paths generated from any kind of discretization of free space.[138] Imagine that the path is a string. Now imagine pulling on both ends to tighten the string (the technical name for this is *path relaxation*) This would remove most of the kinks from the path without violating the safe zone property of convex polygons.

PATH RELAXATION

Meadow maps have three problems which limit their usefulness. One problem is that the technique to generate the polygons is computationally

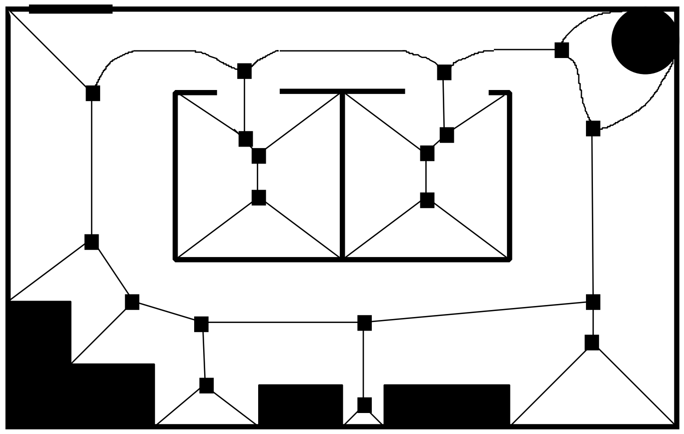


Figure 10.5 Graph from a generalized Voronoi graph (GVG). (Graph courtesy of Howie Choset.)

complex. But more importantly, it uses artifacts of the map to determine the polygon boundaries, rather than things which can be sensed. Unless the robot has accurate localization, how does it know it is halfway down a long, featureless hall and should turn 30°? The third major disadvantage is that it is unclear how to update or repair the diagrams as the robot discovers discrepancies between the *a priori* map and the real world.

10.3.2 Generalized Voronoi graphs

GENERALIZED VORONOI GRAPHS

Generalized Voronoi Graphs, or *GVGs*, are a popular mechanism for representing Cspace and generating a graph. Unlike a meadow map, a GVG can be constructed as the robot enters a new environment, thereby creating a topological map as shown by Howie Choset at CMU.[36]

VORONOI EDGE

The basic idea of a GVG is to generate a line, called a *Voronoi edge*, equidistant from all points. As seen in Fig. 10.5, this line goes down the middle of hallways and openings. The point where many Voronoi edges meet is known as a *Voronoi vertex*. Notice the vertices often have a physical correspondence to configurations that can be sensed in the environment. This makes it much easier for a robot to follow a path generated from a GVG, since there is an implicit local control strategy of staying equidistant from all obstacles.

If the robot follows the Voronoi edge, it will not collide with any modeled obstacles, because it is staying "in the middle." This obviates the need to grow the obstacle boundaries. Edges serve as freeways or major thorough-

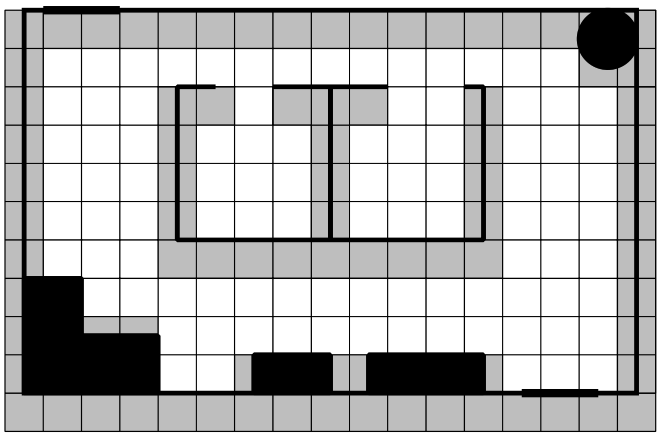


Figure 10.6 Regular grid.

fares. It should also be noted that the curved edges in a GVG do not matter to graph theory or graph algorithms. It is only the length, not the physical reality, of the edges that make any difference.

10.3.3 Regular grids

Another method of partitioning the world space is a regular grid. The regular grid method superimposes a 2D Cartesian grid on the world space, as shown in Fig. 10.6. If there is any object in the area contained by a grid element, that element is marked occupied. Hence, regular grids are often referred to as occupancy grids. Occupancy grids will be detailed in Ch. 11.

Regular grids are straightforward to apply. The center of each element in the grid can become a node, leading to a highly connected graph. Grids are either considered *4-connected* or *8-connected*, depending on whether they permit an arc to be drawn diagonally between nodes or not.

4-CONNECTED NEIGHBORS
8-CONNECTED NEIGHBORS
DIGITIZATION BIAS

Unfortunately, regular grids are not without problems. First, they introduce *digitization bias*, which means that if an object falls into even the smallest portion of a grid element, the whole element is marked occupied. This leads to wasted space and leads to very jagged objects. To reduce the wasted space, regular grids for an indoor room are often finely grained, on the order of 4- to 6-inches square. This fine granularity means a high storage cost, and a high number of nodes for a path planning algorithm to consider.

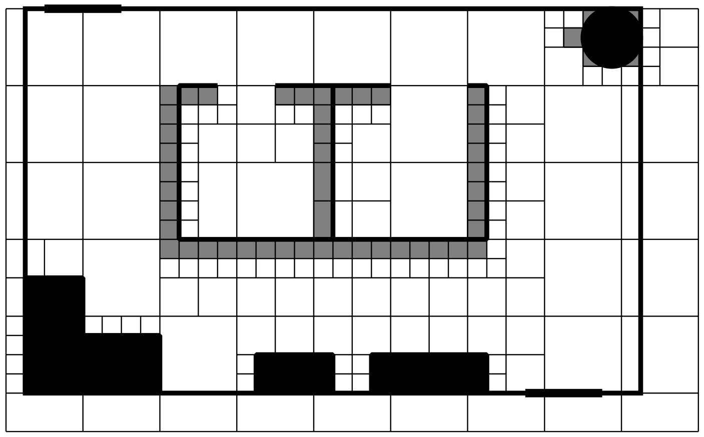


Figure 10.7 Quadtree Cspace representation.

10.3.4 Quadtrees

QUADTREES

A variant on regular grids that attempts to avoid wasted space is *quadtrees*. Quadtrees are a recursive grid. The representation starts out with grid elements representing a large area, perhaps 64-inches square (8 by 8 inches). If an object falls into part of the grid, but not all of it, the Cspace algorithm divides the element into four (hence the name "quad") smaller grids, each 16-inches square. If the object doesn't fill a particular sub-element, the algorithm does another recursive division of that element into four more sub-elements, represented a 4-inches square region. A three dimensional quadtree is called an octree.

10.4 Graph Based Planners

INITIAL NODE

GOAL NODE

As seen in the previous section, most Cspace representations can be converted to graphs. This means that the path between the *initial node* and the *goal node* can be computed using graph search algorithms. Graph search algorithms appear in networks and routing problems, so they form a class of algorithms well understood by computer science. However, many of those algorithms require the program to visit each node on the graph to determine the shortest path between the initial and goal nodes. Visiting every node may be computationally tractable for a sparsely connected graph such as derived from a Voronoi diagram, but rapidly becomes computationally expensive for a highly connected graph such as from a regular grid. Therefore, there has been a great deal of interest in path planners which do a "branch and bound" style of search; that is, ones which prune off paths which aren't optimal. Of course, the trick is knowing when to prune!

The A* search algorithm is the classic method for computing optimal paths for holonomic robots. It is derived from the A search method. In order to explain how A* works, A search will be first presented with an example, then A*. Both assume a metric map, where the location of each node is known in absolute coordinates, and the graph edges represent whether it is possible to traverse between those nodes.

The A search algorithm produces an optimal path by starting at the initial node and then working through the graph to the goal node. It generates the optimal path incrementally; each update, it considers the nodes that could be added to the path and picks the best one. It picks the "right" node to add to the path every time it expands the path (the "right node" is more formally known as the plausible move). The heart of the method is the formula (or evaluation function) for measuring the plausibility of a node:

$$f(n) = g(n) + h(n)$$

where:

- $f(n)$ measures how good the move to node n is
- $g(n)$ measures the cost of getting to node n from the initial node. Since A expands from the initial node outward, this is just the distance of the path generated so far plus the distance of the edge to node n
- $h(n)$ is the cheapest cost of getting from n to goal

Consider how the formula is used in the example below. Assume that a Cspace representation yielded the graph in Fig. 10.8.

The A search algorithm begins at node A, and creates a decision tree-like structure to determine which are the possible nodes it can consider adding to its path. There are only two nodes to choose from: *B* and *C*.

In order to determine which node is the right node to add, the A search algorithm evaluates the plausibility of *B* and *C* by looking at the edges. The plausibility of *B* as the next move is:

$$f(B) = g(B) + h(B) = 1 + 2.24 = 3.24$$

where $g(B)$ is the cost of going from A to B, and $h(B)$ is the cost to get from B to the goal E.

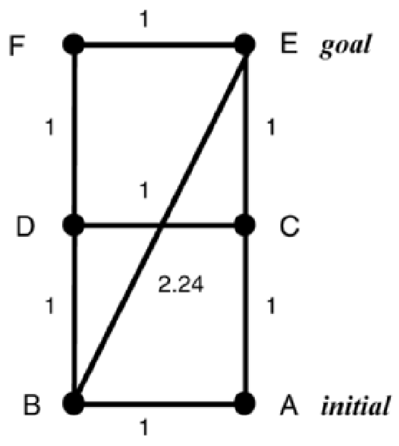


Figure 10.8 Graph for A search algorithm.

The plausibility of C is:

$f(C) = g(C) + h(C) = 1 + 1 = 2.0$

where $g(C)$ is the cost of going from A to C, and $h(C)$ is the cost of getting from C to E. Since $f(C) > f(B)$, the path should go from A to C.

But this assumes that $h(n)$ was known at every node. This meant that the algorithm had to recurse in order to find the correct value of $h(n)$. This leads to visiting all the nodes.

A search is guaranteed to produce the optimal path because it compares all possible paths to each other. A* search takes an interesting approach to reducing the amount of paths that have to be generated and compared: it compares the possible paths to the best possible path, even if there isn't a path in the real world that goes that way. The algorithm estimates h rather than checks to see if there is actually a path segment that can get to the goal in that distance. The estimate can then be used to determine which nodes are the most promising, and which paths have no chance of reaching the goal better than other candidates and should be pruned from the search.

Under A* the evaluation function becomes:

$f^*(n) = g^*(n) + h^*(n)$

where the $*$ means that the functions are estimates of the values that would

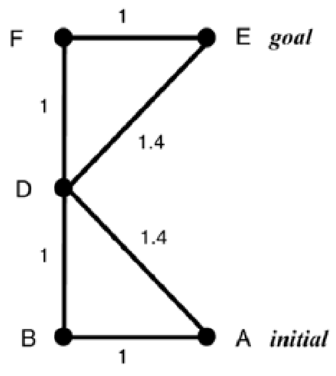


Figure 10.9 An A* example.

have been plugged into the A search evaluation. In path planning, $g^*(n)$ is the same as $g(n)$: the cost of getting from the initial node to n, which is known through the incremental build of the path. $h^*(n)$ is the real difference. So what is a way to estimate the cost of going from n to the goal? Furthermore, how can we be sure that the estimate will be accurate enough that we don't end up choosing a path which isn't truly optimal? This can be done by making sure that $h^*(n)$ will never be smaller than $h(n)$. The restriction that $h^*(n) \leq h(n)$ is called the *admissibility condition*. Since $h^*(n)$ is an estimate, it is also called a *heuristic function*, since it uses a rule of thumb to decide which is the best node to consider.

ADMISSIBILITY CONDITION
HEURISTIC FUNCTION

Fortunately, there is a natural heuristic function for estimating the cost from n to the goal: the Euclidean (straight line) distance. Recall that the locations of each node are known independently of the edges. Therefore, it is straightforward to compute the straight line distance between two nodes. The straight line distance is always the shortest path between two points, barring curvature of the earth, etc. Since the real path can never be shorter than that, the admissibility condition is satisfied.

To see how A* uses this to actually eliminate visiting nodes, consider the example in Fig. 10.9. As with A search, the first step in A* is to consider the choices from the initial node.

The choices are B and D, which can be thought of as a search tree (see Fig. 10.10a) or as a subset of the original graph (see Fig 10.10b). Regardless

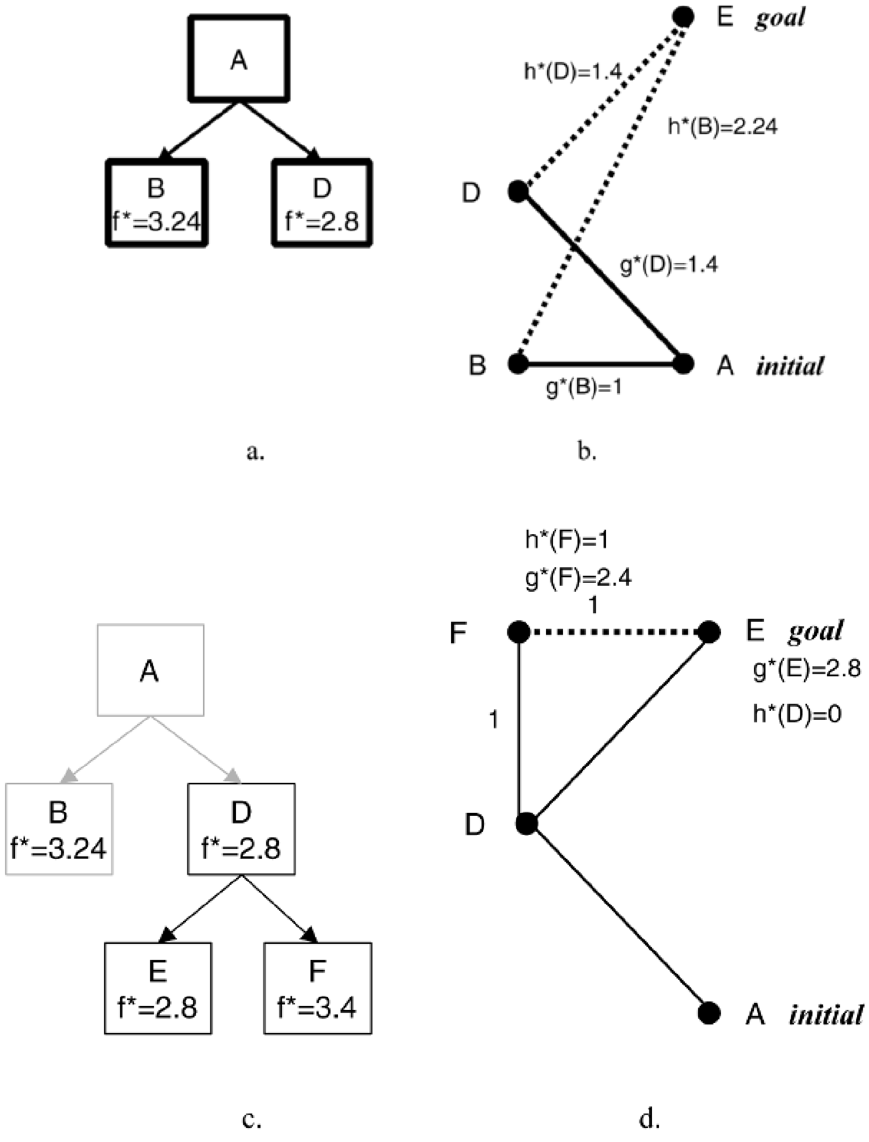


Figure 10.10 a.) What A* "sees" as it considers a path A-?-E, and b.) the original graph. c.) What A* "sees" from considering a path A-D-?-E, and d.) the associated graph.

of how it is visualized, each node is evaluated to determine which is the most plausible move. The above figure shows what the algorithm "sees" at this point in the execution. The choices evaluate to:

$$\begin{aligned} f^*(B) &= g^*(B) + h^*(B) = 1 + 2.24 = 3.24 \\ f^*(D) &= g^*(D) + h^*(D) = 1.4 + 1.4 = 2.8 \end{aligned}$$

A path going from $A - D - ? - E$ has the potential to be shorter than a path going from $A - B - ? - E$. So, D is the most plausible node. Notice that A* can't eliminate a path through B because the algorithm can't "see" a path that actually goes from D to E and determine if it is indeed as short as possible.

At step 2, A* recurses (repeats the evaluation) from D, since D is the most plausible, as shown in Fig. 10.10.

The two options from D are E and F, which are evaluated next:

$$\begin{aligned} f^*(E) &= g^*(E) + h^*(E) = 2.8 + 0 = 2.8 \\ f^*(F) &= g^*(F) + h^*(F) = 2.4 + 1.0 = 3.4 \end{aligned}$$

Now the algorithm sees that E is the best choice among the leaves of the search tree, including the branch through B. (If B was the best, then the algorithm would have changed branches.) It is better than F and B. When it goes to expand E, it notices that E is the goal, so the algorithm is complete. The optimal path is $A - D - E$, and we didn't have to explicitly consider $A - B - F - E$. There are other ways to improve the procedure described so far. $f^*(F)$ didn't need to be computed if the algorithm looks at its choices and sees that one of them is the goal. Any other choice has to be longer because edges aren't allowed to be negative, so $D - F - E$ has to be longer than $D - E$.

Another important insight is that any path between A and E has to go through D, so the B branch of the search tree could have been pruned. Of course, in the above example the algorithm never had an opportunity to notice this because B was never expanded. It's easy to imagine in a larger graph that there might be a case where after several expansions through D, the leaf at B in the search tree came up the most plausible. Then the algorithm would have expanded A and seen the choices were D. Since D already occurred in another branch, with a cheaper $g^*(D)$, the B branch could be safely pruned.

This is particularly useful when A* is applied to a graph created from a regular grid, where the resulting graph is highly connected.

One very attractive feature of the A* path planner is that it can be used with any Cspace representation that can be transformed into a graph. The major impact that Cspace has on the A* planner is how many computations it takes to find the path.

A limitation of A* is that it is very hard to use for path planning where there are factors other than distance to consider in generating the path. For example, the straight line distance may be over rocky terrain or sand that poses a risk to the robot. Likewise, the robot may wish to avoid going over hills in order to conserve energy, but at the same time might wish to go down hills whenever possible for the same reason. In order to factor in the impact of terrain on the path costs, the heuristic function $h^*(n)$ has to be changed. But recall that the new heuristic function must continue to satisfy the admissibility condition: $h^* \leq h$. If the new h^* just takes the worst case energy cost or safety cost, it will be admissible, but not particularly useful in pruning paths. Also, gaining energy going downhill is essentially having an edge in the graph with a negative weight, which A* can't handle. (Negative weights pose an interesting problem in that the robot can get into a loop of rolling down a hill repeatedly because it's an energy-efficient solution rather than actually making forward progress! Bellman-Ford types of algorithms deal with this situation.)

10.5 Wavefront Based Planners

Wavefront propagation styles of planners are well suited for grid types of representations. The basic principle is that a wavefront considers the Cspace to be a conductive material with heat radiating out from the initial node to the goal node. Eventually the heat will spread and reach the goal, if there is a way, as shown in the sequence in Fig. 10.11. Another analogy for wavefront planners is region coloring in graphics, where the color spreads out to neighboring pixels. An interesting aspect of wavefront propagation is that the optimal path from all grid elements to the goal can be computed as a side effect. The result is a map which looks like a potential field. One of the many wavefront types of path planners is the Trulla algorithm developed by Ken Hughes.[106] It exploits the similarities with a potential field to let the path itself represent what the robot should do as if the path were a sensor observation.

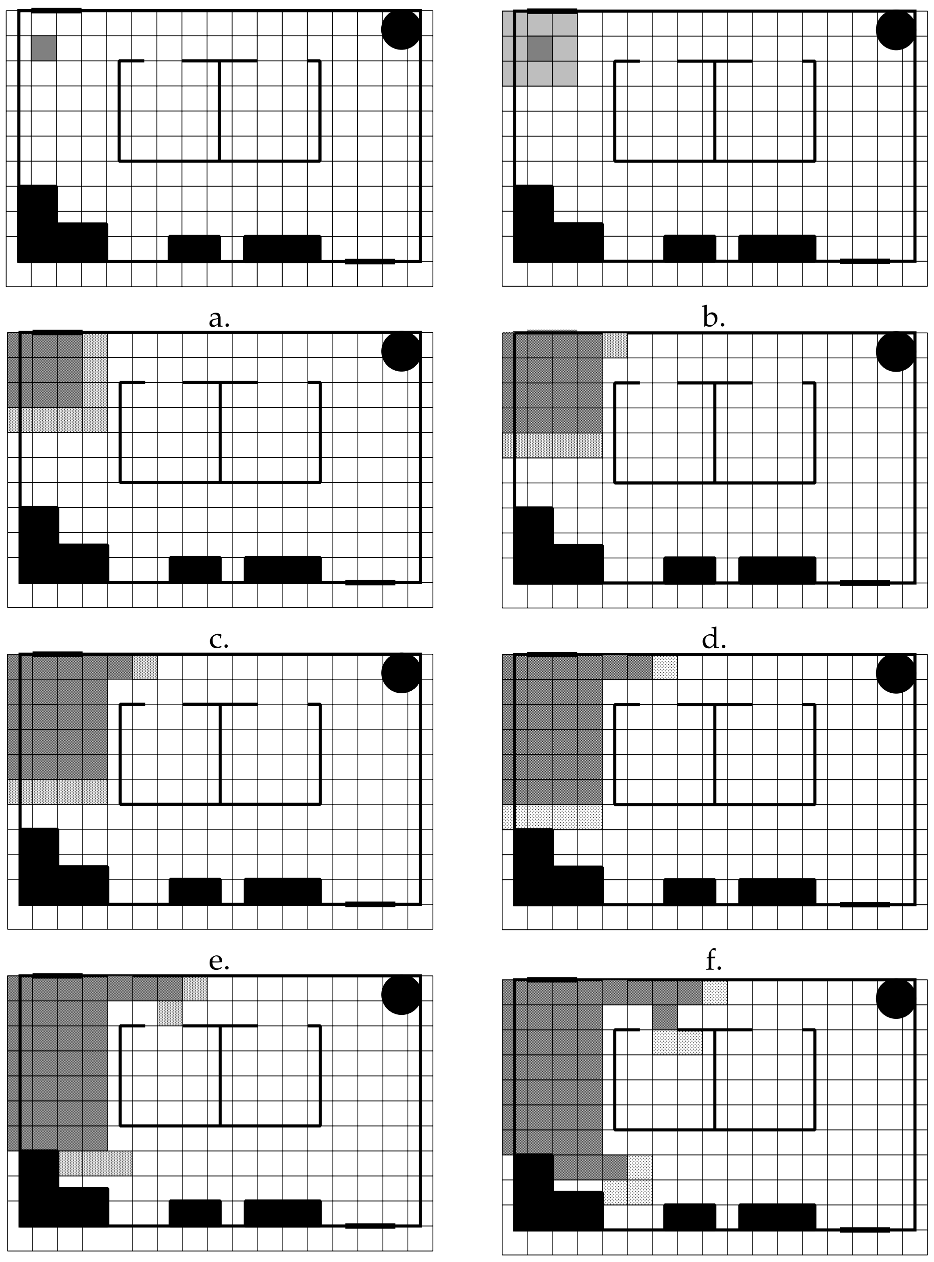


Figure 10.11 A wave propagating through a regular grid. Elements holding the current front are shown in gray, older elements are shown in dark gray.

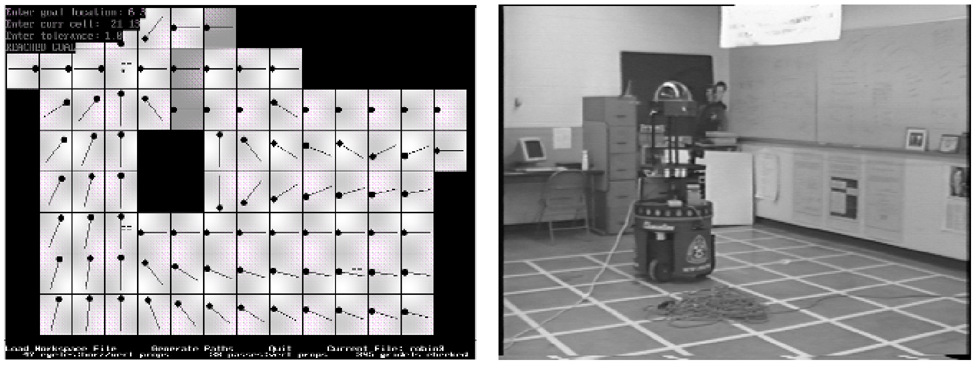
An attractive feature of wavefront propagation is how it treats terrain types. A modeled obstacle can have a conductivity of 0.0 (no heat can be propagated through that grid element), while an open area has infinite conductivity. But undesirable terrains, which are traversable, but not desirable to go across, can be given a low value of conductivity. This means that heat will travel slower through undesirable regions. But it may turn out that this is the best path, even with the loss through the undesirable regions. A wavefront naturally handles the tradeoffs between a longer path through desirable terrain versus taking shortcuts through less desirable terrain.

The examples in Fig. 10.12 use the Trulla planner. Undesirable terrain is shown on the Trulla display in gray, obstacles in black and open areas in white. The intensity of gray reflects the degree of undesirability of the terrain. In Fig. 10.12a, the robot Clementine can move over an extension cord but would prefer not to, as shown by the gray region. The path planner accordingly routes her around the cord. In Fig. 10.12 this isn't possible due to the placement of obstacles. Therefore, she has to cut through.

10.6 Interleaving Path Planning and Reactive Execution

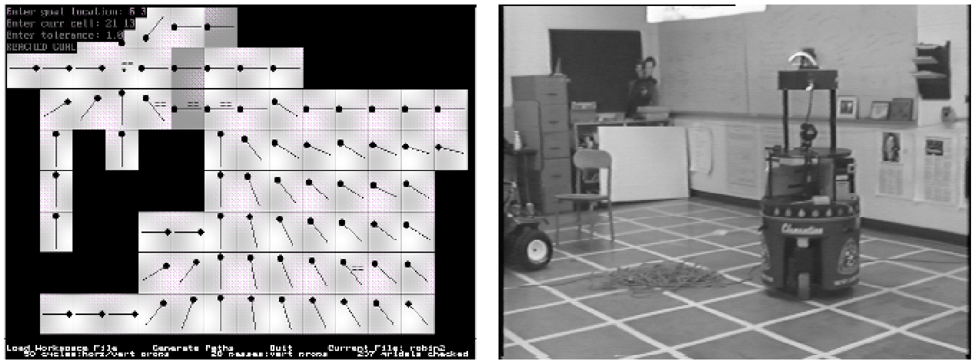
Most path planning algorithms are employed in a strictly plan once, then reactively execute style. Almost all techniques break a path into segments; even a wavefront planner actually produces a goal location (waypoint) for each directional vector. This is well suited for a Hybrid architecture with a Cartographer handing off path segments, or an entire path, for a Sequencer. The Sequencer can then employ a series of move-to-goal behaviors, deactivating and re-instantiating the behavior as a new subgoal is met. Unfortunately there are two problems with reactive execution of a metric path as described above: *subgoal obsession* and the lack of *opportunistic replanning*. Subgoal obsession is when the robot spends too much time and energy trying to reach the exact subgoal position, or more precisely, when the termination conditions are set with an unrealistic tolerance. The problem with termination conditions for subgoals is best defined by an example. Suppose the next waypoint is at location (35, 50). If the robot has shaft encoders or GPS, this should be straightforward to do in theory. In practice, it is very hard for a robot, even a holonomic robot, to reach any location exactly because it is hard to give the robot precise movement. The robot may reach (34.5, 50). The behavior sees the goal is now 0.5 meters ahead, and move again to attempt to reach (35, 50). On that move, it may overshoot and end up at (35.5,50.5).

SUBGOAL OBSESSION
OPPORTUNISTIC REPLANNING



```
Enter goal location: 6 3
Enter curr cell: 21 13
Enter tolerance: 1.0
REACHED GOAL
Load Workspace File     Generate Paths     Quit     Current File: robin3
```

a.



```
Enter goal location: 6 3
Enter curr cell: 21 13
Enter tolerance: 1.0
REACHED GOAL
Load Workspace File     Generate Paths     Quit     Current File: robin2
```

b.

Figure 10.12 a.) Trulla output and photo of Clementine navigating around cord. b.) Trulla output and photo of Clementine navigating over the cord.

Now it has to turn and move again. A resulting see-sawing set of motions emerges that can go on for minutes. This wastes time and energy, as well as makes the robot appear to be unintelligent. The problem is exacerbated by non-holonomic vehicles which may have to back up in order to turn to reach a particular location. Backing up almost always introduces more errors in navigation.

To handle subgoal obsession, many roboticists program into their move-to-goal behaviors a tolerance on the terminating condition for reaching a goal. A common heuristic for holonomic robots is to place a tolerance of +/- the width of the robot. So if a cylindrical, holonomic robot with a diameter of 1 meter is given a goal of (35, 50), it will stop when 34.5 $< y <$ 35.5 and 49.5

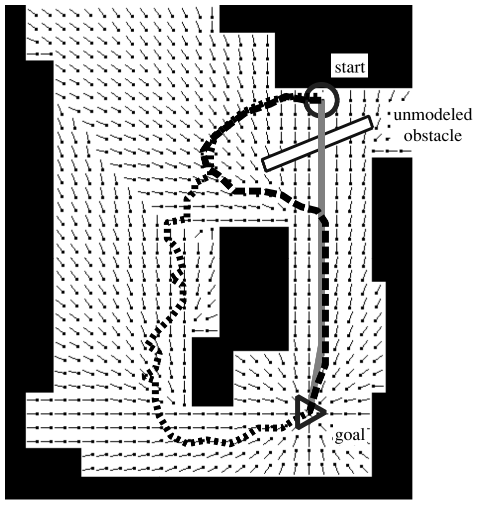
$< y < 50.5$. There is no common heuristic for non-holonomic robots, because the maneuverability of each platform is different. A more vexing aspect of subgoal obsession is when the goal is blocked and the robot can't reach the terminating condition. For example, consider a subgoal at the opposite end of a hall from a robot, but the hall is blocked and there is no way around. Because the robot is executing reactively, it doesn't necessarily realize that it isn't making progress. One solution is for the Sequencer to estimate a maximum allowable time for the robot to reach the goal. This can be implemented either as a parameter on the behavior (terminate with an error code after n seconds), or as an internal state releaser. The advantage of the latter solution is that the code can become part of a monitor, leading to some form of self-awareness.

Related to subgoal obsession is the fact that reactive execution of plans often lacks opportunistic improvements. Suppose that the robot is heading for Subgoal 2, when an unmodeled obstacle diverts the robot from its intended path. Now suppose that from its new position, the robot can perceive Subgoal 3. In a classic implementation, the robot would not be looking for Subgoal 3, so it would continue to move to Subgoal 2 even though it would be more optimal to head straight for Subgoal 3.

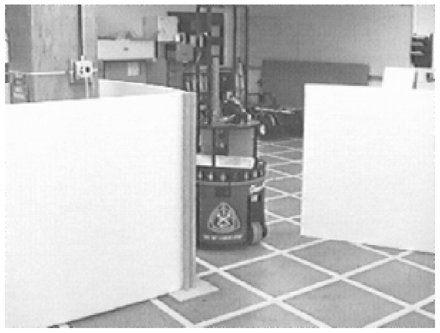
The need to opportunistically replan also arises when an *a priori* map turns out to be incorrect. What happens when the robot discovers it is being sent through a patch of muddy ground? Trying to reactively navigate around the mud patch seems unintelligent because choosing left or right may have serious consequences. Instead the robot should return control to the Cartographer, which will update its map and replan. The issue becomes how does a robot know it has deviated too far from its intended path and needs to replan?

D* ALGORITHM

Two different types of planners address the problems of subgoal obsession and replanning: the *D* algorithm* developed by Tony Stentz[136] which is a variant of the A* replanning algorithm, and an extension to the Trulla algorithm. Both planners begin with an *a priori* map and compute the optimal path from every location to the goal. D* does it by executing an A* search from each possible location to the goal in advance; this converts A* from being a single-source shortest path algorithm into an all-paths algorithm. This is computationally expensive and time-consuming, but that isn't a problem since the paths are computed when the robot starts the mission and is sitting still. Since Trulla is a wavefront type of planner, it generates the optimal path between all pairs of points in Cspace as a side effect of computing the path from the starting location to the goal.



a.



b.

Figure 10.13 Layout showing unmodeled obstacle. a.) Gray line shows expected path, long dashed line the actual path with Trulla, and short dashed line shows purely reactive path. b.) *Clementine* opportunistically turning.

Computing the optimal path from every location to the goal actually helps with reactive execution of the path. It means that if the robot can localize itself on the *a priori* map, it can read the optimal subgoal for `move-to-goal` on each update. If the robot has to swing wide to avoid an unmodeled obstacle in Fig. 10.13, the robot automatically becomes redirected to the optimal path without having to replan. Note how the metric path becomes a virtual sensor, guiding the move-to-goal behavior replacing the direct sensor data. This is a rich mechanism for the deliberative and reactive components of Hybrid architectures to interact.

This approach eliminates subgoal obsession, since the robot can change "optimal" paths reactively and opportunistically move to a closer waypoint. As with most things in life, too much of a good thing is bad. At some point though, the sheer number of unmodeled obstacles might force the robot to get trapped or wander about, changing subgoals but making no real progress. The D* solution to this problem is to continuously update the map and dynamically repair the A* paths affected by the changes in the map. D* represents one extreme on the replanning scale: *continuous replanning*.

CONTINUOUS REPLANNING

Continuous replanning has two disadvantages. First, it may be too computationally expensive to be practical for a robot with an embedded processor and memory limitations, such as a planetary rover. Second, continuous replanning is highly dependent on the sensing quality. If the robot senses an unmodeled obstacle at time T1, it computes a new path and makes a large course correction. If it no longer senses that obstacle at time T2 because the first reading was a phantom from sensor noise, it will recompute another large course correction. The result can be a robot which has a very jerky motion and actually takes longer to reach the goal.

EVENT-DRIVEN REPLANNING

In the cases of path planning with embedded processors and noisy sensors, it would be desirable to have some sort of *event-driven* scheme, where an event noticeable by the reactive system would trigger replanning. Trulla uses the dot-product of the intended path vector and the actual path vector. When the actual path deviates by 90° or more, the dot product of the path vector and the actual vector the robot is following becomes 0 or negative. Therefore the dot product acts as an affordance for triggering replanning: the robot doesn't have to know why it is drifting off-course, only that it has drifted noticeably off-course.

This is very good for situations that would interfere with making progress on the originally computed path, in effect, situations where the real world is less amenable to reaching the intended goal. But it doesn't handle the situation where the real world is actually friendlier. In Fig. 10.14, an obstacle thought to be there really isn't. The robot could achieve a significant savings in navigation by opportunistically going through the gap.

Such opportunism requires the robot to notice that the world is really more favorable than originally modeled. A continuous replanner such as D* has a distinct advantage, since it will automatically notice the change in the world and respond appropriately, whereas Trulla will not notice the favorable change because it won't lead to a path deviation. It is an open research question whether there are affordances for noticing favorable changes in the world that allow the robot to opportunistically optimize it path.

10.7 Summary

Metric path planning converts the world space to a configuration space, or *Cspace*, representation that facilitates path planning. Cspace representations such as generalized Voronoi diagrams exploit interesting geometric properties of the environment. These representations can then be converted to graphs, suitable for an A* search. Since Voronoi diagrams tend to produce sparser graphs, they work particularly well with A*. Regular grids work

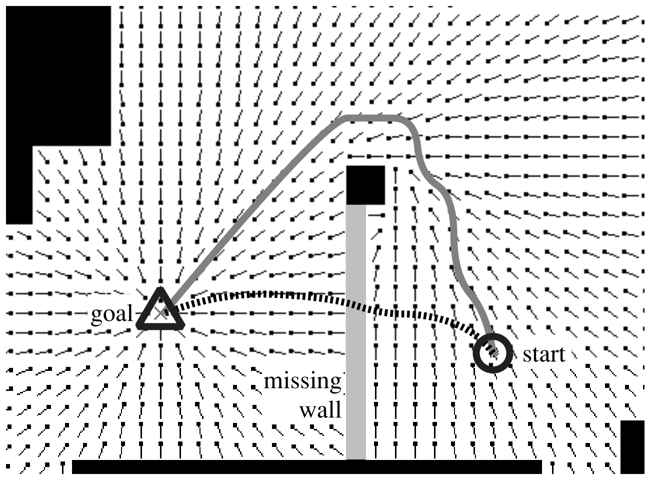


Figure 10.14 Opportunity to improve path. The gray line is the actual path, while the dashed line represents a more desirable path.

well with wavefront planners, which treat path planning as an expanding heat wave from the initial location.

Metric path planning tends to be expensive, both in computation and in storage. However, they can be interleaved with the reactive component of a Hybrid architecture, where the Cartographer gives the Sequencer a set of waypoints. Two problems in interleaving metric path planning with reactive execution are subgoal obsession and when to replan. Optimal path planning techniques for *a priori* fixed maps are well understood, but it is less clear how to update or repair the path(s) without starting over if the robot encounters a significant deviation from the *a priori* map. One solution is to continuously replan if resources permit and sensor reliability is high; another is event-driven replanning which uses affordances to detect when to replan.

Cspace representations and algorithms often do not consider how to represent and reason about terrain types, and special cases such as the robot actually conserving or generating energy going downhill are usually ignored. A possibly even more serious omission is that popular path planners are applicable for only holonomic vehicles.

10.8 Exercises

Exercise 10.1

Represent your indoor environment as a GVG, regular grid, and quadtree.

Exercise 10.2

Represent your indoor environment as a regular grid with a 10cm scale. Write down the rule you use to decide how to mark a grid element empty or occupied when there is a small portion overlapping it.

Exercise 10.3

Define *path relaxation*.

Exercise 10.4

Consider a regular grid of size 20 by 20. How many edges will a graph have if the neighbors are

a. 4-connected?

b. 8-connected?

Exercise 10.5

Convert a meadow map into a graph using:

a. the midpoints of the open boundaries, and

b. the midpoints plus the 2 endpoints.

Draw a path from A to B on both graphs. Describe the differences.

Exercise 10.6

Apply the A* search algorithm by hand to a small graph to find the optimal path between two locations.

Exercise 10.7

What is a heuristic function?

Exercise 10.8

Apply wavefront propagation to a regular grid.

Exercise 10.9

Subgoal obsession has been described as a problem for metric planners. Can hybrid systems which use topological planning exhibit subgoal obsession?

Exercise 10.10

Trulla uses a dot product of 0 or less to trigger replanning, which corresponds to $90°$ from the desired path. What are the advantages or disadvantages of $90°$? What would happen if Trulla used $45°$ or $135°$?

Exercise 10.11

Describe the difference between *continuous* and *event-driven replanning*. Which would be more appropriate for a planetary rover? Justify your answer.

Exercise 10.12 [*Programming*]

Obtain a copy of the Trulla simulator suitable for running under Windows. Model at least three different scenarios and see what the path generated is.

Exercise 10.13 [*Programming*]

Program an A* path planner. Compare the results to the results for the Dijkstra's single source shortest path program from Ch. 9.

10.9 End Notes

For a roboticist's bookshelf.
Robot Motion Planning by Jean-Claude Latombe of Stanford University is the best-known book dealing with configuration space.[82]

Voronoi diagrams.
Voronoi diagrams may be the oldest Cspace representation. *Computational Geometry: Theory and Applications*[44] reports that the principles were first uncovered in 1850, although it wasn't until 1908 that Voronoi wrote about them, thereby getting his name on the diagram.

On heuristic functions for path planning.
Kevin Gifford and George Morgenthaler have explored some possible formulations of a heuristic function for path planning over different terrain types. Gifford also developed an algorithm which can consider the energy savings or capture associated with going downhill.[60]

Trulla.
The Trulla algorithm was developed by Ken Hughes for integration on a VLSI chip. Through grants from the CRA Distributed Mentoring for Women program, Eva Noll and Aliza Marzilli implemented it in software. The algorithm was found to work fast enough for continuous updating even on an Intel 486 processor. Dave Hershberger helped write the display routines used in this book.