MECEE 4602: Introduction to Robotics, Fall 2019

Motion Planning and Navigation

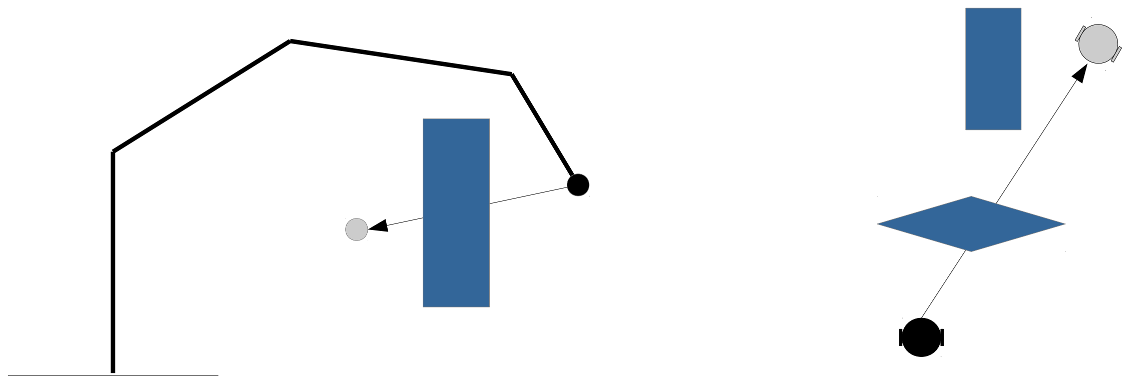
Consider a kinematic chain with a start pose and a goal pose for the end-effector. We've already seen one way to get the end-effector to the goal pose: Cartesian control. At its core, Cartesian control attempts to take a straight line motion towards the goal.

So far, we have ignored problems like joint limits, or collisions. Cartesian control can be augmented to handle these, but its main characteristic remains: it attempts to move in a straight line towards the goal. It falls into the larger category of "gradient descent" (or ascent) algorithms, which take a "local" view of the problem, and generally try to take what appears to the the shortest path to the goal from the current location.

In general, algorithms attempting to find a path from a start point to a goal point are very common in robotics. In fact, many of them are shared between robot arms and mobile robots (we'll see how these problems are similar in a second). For robot arms, they are often referred to as "motion planning" algorithms. For mobile robots, they are generally called "navigation" algorithms. However, that is not a strong distinction, and one can encounter hybrid names such as "arm navigation".

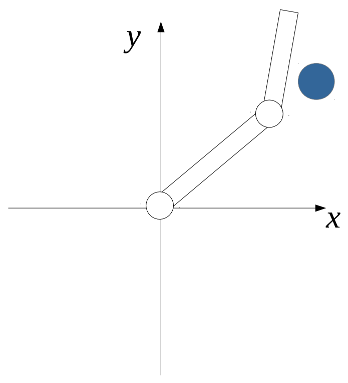
1 Arm Configuration Space

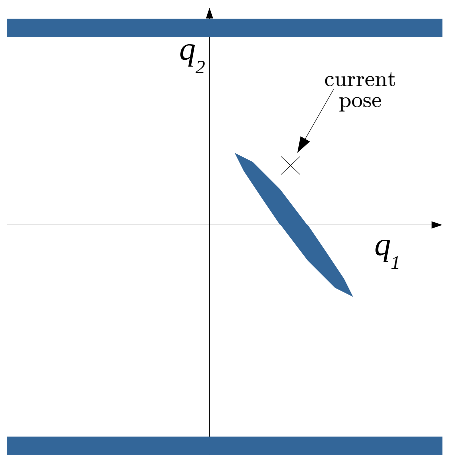
How is the problem of a mobile robot navigating around a room similar to that of a robot arm? At first glance, they appear quite different:



To see the similarities, we must think of the arm movement problem in joint space. We recall once again that this is the space of all values for the joints. A point in joint space is denoted by $\boldsymbol{q} = [q_0, .., q_{n-1}]^T \in \mathcal{R}^n$, where n is the number of joints of the robot. Note that, in this context, the joint space of a robot is often referred to as the **configuration space**.

Here is an approximate example of the joint space of the 2-link robot marking an obstacle as well as joint limits for the second joint:





If a point in joint space is marked as "impossible", it means that that combination of joint angles should not be used. Reasons can include:

- robot is in collision with the environment;
- robot is in collision with itself;
- joint limits are violated;
- etc.

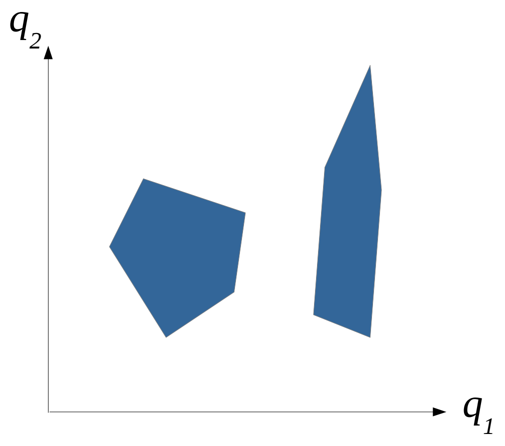
Once we are operating in configuration space, we see that the motion planning problem starts resembling the case for mobile robot: we have a start location, a goal location, and a number of obstacles along the way. The current pose of the robot gives us the start location in configuration space. Performing IK on the goal end-effector pose gives us a goal location (or multiple goal locations) in configuration space. The robot must then "navigate" its way around obstacles to reach the goal.

A great applet for visualizing the configuration space of a 2-link manipulator can be found at `http://robotics.cs.unc.edu/education/c-space/`

2 Building a configuration space map

2.1 Polygonal vs. grid-based maps

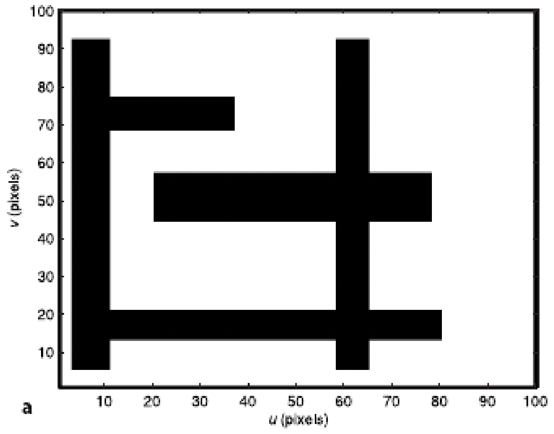
A polygonal map is just a list of obstacles, each expressed as a polygon.



Obstacle 1:
- (a_1, b_1)
- (a_2, b_2)
- …
- (a_n, b_n)

Obstacle 2:
- (a_1, b_1)
- (a_2, b_2)
- …
- (a_m, b_m)

In contrast, a grid-based map is a dense grid, sampling the entire space. The simplest version is a binary grid: each cell contains either a 0 (indicating free space) or a 1 (indicating an obstacle). More complex grids can also hold other types of information in their cells. (Example below from Peter Corke, "Robotics, Vision and Control").



The pros and cons of polygonal maps vs. grids include:

- polygonal maps are much more compact in memory;
- grid-based maps allow for fast testing if a point is inside an obstacle or not;
- grid-based maps are much easier to build for many real-world problems.

Our algorithms and examples will focus almost exclusively on grid-based maps, as they are used in practice much more often.

2.2 Mobile robots

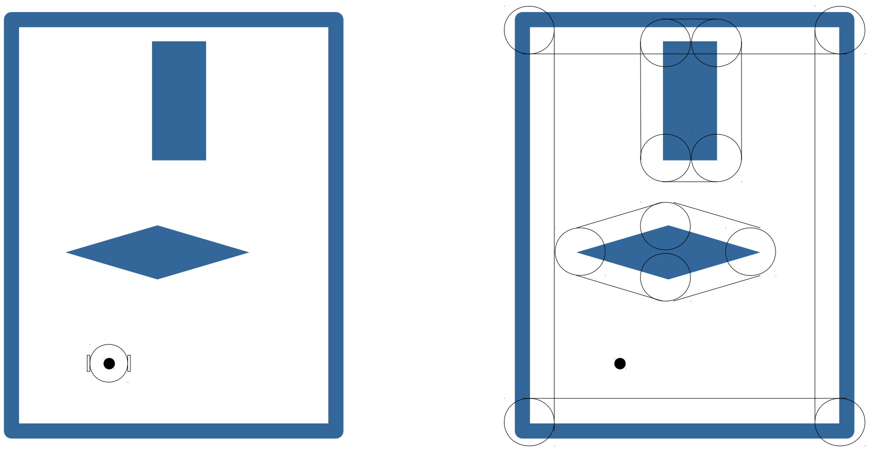
For mobile robots, the configuration space is closely related to the Cartesian space. For a robot navigating in 2 dimensions, a configuration consists of an $[x, y, \theta]$ triple showing the coordinates of a reference point chosen somewhere on the robot, as well as the orientation of the base relative to the world. In general, the configuration space of a robot capable of 2D navigation is thus 3-dimensional.

If the robot has a round footprint, its orientation does not matter for the purpose of collision avoidance. The configuration space is thus considered to be 2-dimensional, containing just $[x, y]$, the coordinates of the reference point.

Assume we have a Cartesian map showing the locations of obstacles in the world. Obviously, if a point $[a, b]$ is inside an obstacle in this map, it should also be marked as infeasible in the configuration space - the robot reference point can not be inside an obstacle. However, the configuration space map has to contain more than that: the robot can collide with an obstacle even if its reference point is not inside the obstacle, but merely too close to it.

The common solution to that problem is to "grow" or "inflate" the obstacles by the footprint of the robot when building the configuration space map. This allows us to think of the robot as a single point in configuration space; as long as that point is in feasible territory, the entire robot is safe.

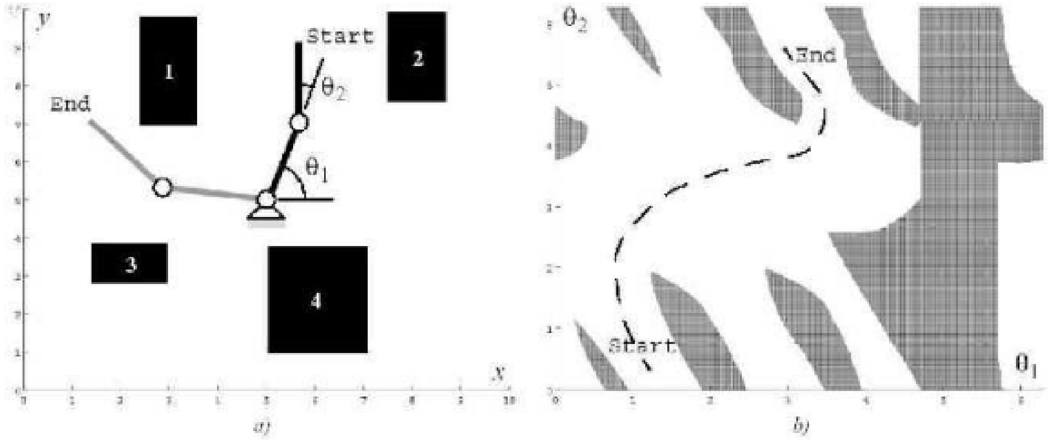
For non-round robots, we can still "inflate" all obstacles by a radius large enough to accomodate the robot in any orientation. This then allows us to again have a 2-dimensional configuration space, where we only care about x and y. The flip side is that we inflate the obstacles more than is strictly needed, we might close off passageways that actually can be traversed.



We note that if we start with a polygonal map of Cartesian space (that is, a list of polygons making up the obstacles), this method will result in a polygonal map of configuration space as well.

2.3 Robot arms

For a robot arm, we could theoretically build a polygonal configuration space by starting from a Cartesian map of the environment, then performing Inverse Kinematics to map obstacles to the configuration space. However, this is extremely impractical. Most often, arm configuration space maps are grid based, and built simply by sampling the grid and determining if the point is feasible or not through forward kinematics.

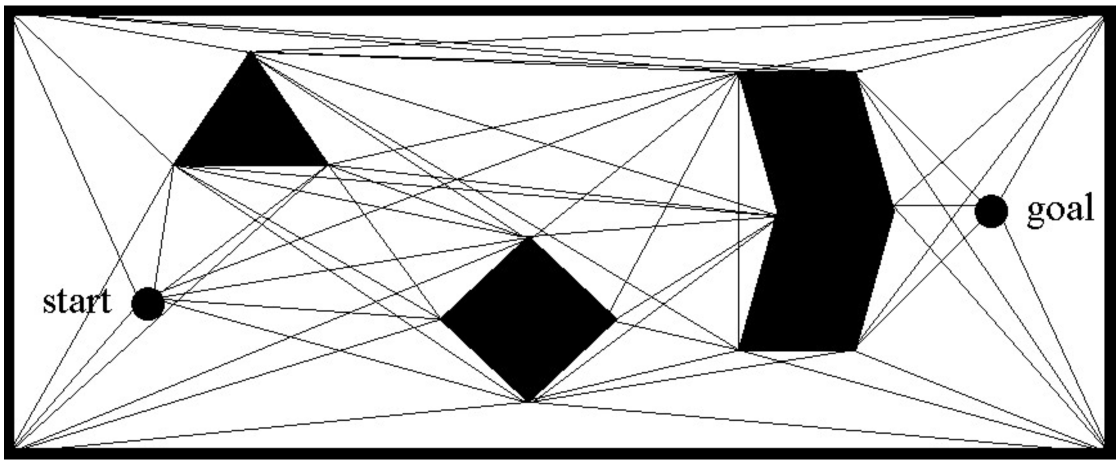


Note that some algorithms require the entire configuration space to be mapped in advance. Others only require us to sample the space as needed to find out if a given point is acceptable or not.

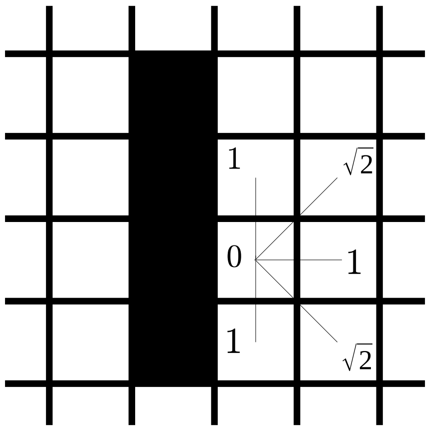
3 Planning in configuration space

Many algorithms for planning in configuration space fall in the broad category of **graph search algorithms**. A graph is a colection of nodes and edges, where each edge can have an associated cost of traversing.

Polygonal configuration space maps can be turned into graphs by computing the visibility graph. The cost of traversing each edge is proportional to the length of the edge (example from http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/16311/www/f05/lecture/lec8.html):



Grid-based maps are also instances of graphs: each cell is a node, connected to all its neighbors. Edges to neighbors that share a cell side are all equal cost; diagonal edges' cost are multiplied by $\sqrt{2}$:



We will now look at a few algorithms with similar requirements: given a graph (that is, a set of nodes and edges connecting them), as well as a start and goal node, find a path that connects them. Since we will be using grid-based examples here, we will generally use "cells" instead of "nodes".

3.1 Dijkstra's algorithm

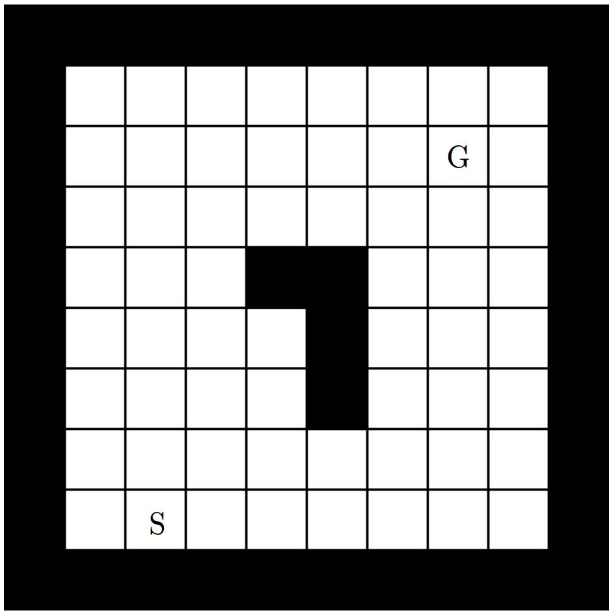
Dijkstra's algorithm is a well known approach that is guaranteed to find the shortest path between two nodes in a graph (or cells on a grid). It works by keeping track, for each node it visits, of the shortest path from the start to this node.

The grid version of Dijkstra's algorithm is below. $g(c)$ is the current estimate of the shortest path from the start to cell c, and $d(c, n)$ is the length of the path from cell c to cell n.

- label all cells as "unvisited"
- mark starting cell s as having path length $g(s) = 0$
- while unvisited cells remain:
  - choose unvisited cell c with lowest path length $g(c)$
  - mark it as "visited"

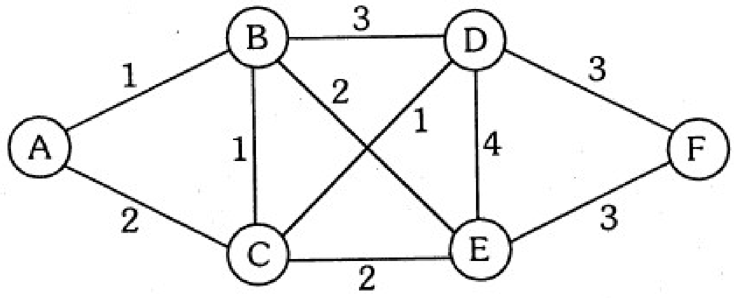
```
    – for each neighbor $n$:
        ∗ update $g(n)$ to min$[g(n), g(c) + d(c, n)]$
```

We can use the grid below to practice executing Dijkstra's algorithm:



Alternatively, the algorithm can be stopped when the goal cell has been marked as "visited". However, from an algorithmic complexity perspective, it makes no difference if we stop there, or keep going until we've processed all the cells. In the worst case, we expect the running time of the algorithm to be on the order of n^2, where n is the number of cells in our grid.

Here is a very simple example of Dijkstra's algorithm run on a general graph (as opposed to a grid).



We will use the table below to keep track of how the algorithm progresses. For each step, we will write the following:

- the node v currently being visited
- the length $g(v)$ of the shortest path from the start node to the node currently being visited.
- all the values of $g(n)$ computed so far for unvisited nodes n.

step	node v being visited	$g(v)$	$g(n)$ computed so far for unvisited nodes n
0	A	$g(A) = 0$	$g(B) = 1$, $g(C) = 2$
1			
2			
3			
4			

3.2 Distance transform

As we mentioned, Dijkstra's algorithm can compute the shortest path from the start to every cell on the grid, including the goal. That obviously comes in handy if we need to compute paths from the start to multiple goals.

In fact, once we've finished running Dijkstra's algorithm, we have what is called a "distance map" from the chosen start point. Let's assume that each cell c has been marked with $g(c)$ as computed by Dijkstra's algorithm. Now, we want to find the shortest path **from the same start cell** to a new goal cell. We run a procedure backwards, starting at the goal and working our way back towards the start. The procedure can be summarized as follows:

- set current node c equal to goal node
- repeat until current node c is the same as start node:
  - find the neighbor n of c that minimizes g(n) + d(n,c)
  - set c = n

The key is that we are not just finding the neighbor n with the lowest g(n), but instead the neighbor n that minimizes g(n) + d(n,c) where c is the current cell. With this approach, a path to each new goal can be computed using running time on the order of n. However, we still incur the n^2 cost once, the first time we run Dijkstra's algorithm.

3.3 The A* algorithm

A* works a lot like Dijkstra's algorithm, keeping track of the shortest path **from the start to each cell** that has been visited. However, it also uses a heuristic to "estimate" how long the path **from a cell to the goal** could be. If a cell has the best current combination of known path from the start plus estimated path to the goal, it will be the next cell visited.

There is an important constraint on the function used to "estimate" the lenth of a path from a cell to the goal: it must not over-estimate the true distance. A commonly used heuristic is the straight-line distance from a cell to the goal, as it obviously can never over-estimate the length of the actual path.

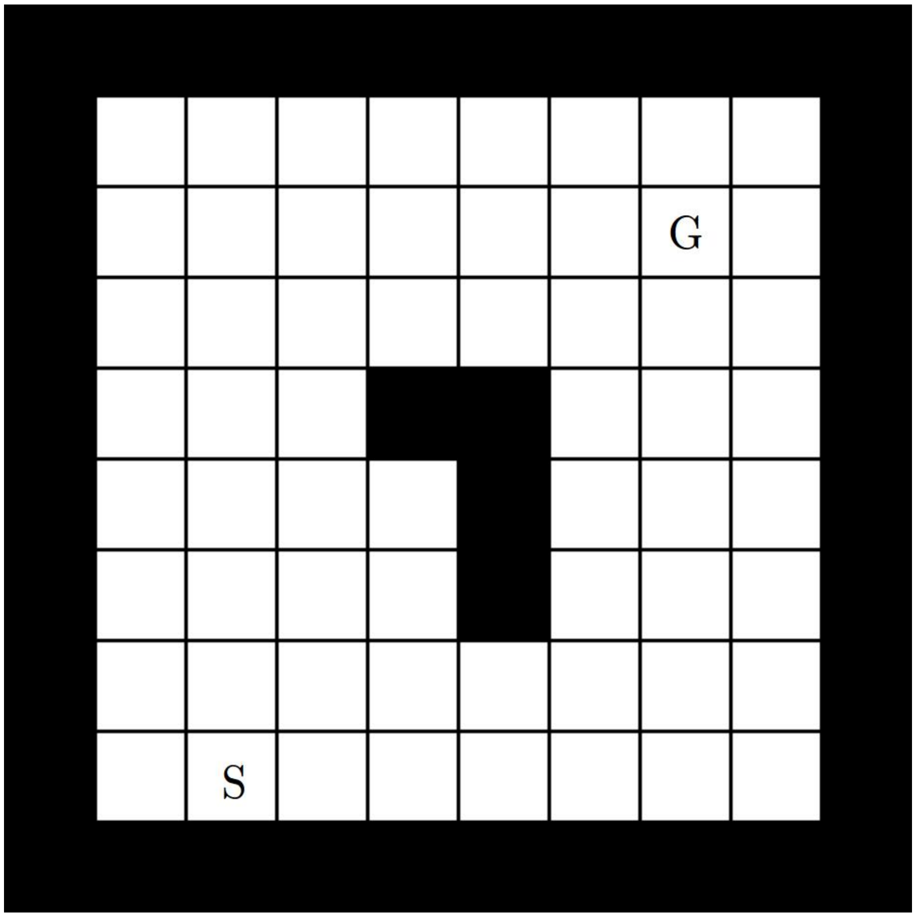
Here is the grid version of the A* algorithm:

- label all cells as "unvisited"
- mark starting cell s as having $g(s) = 0$, $f(s) = g(s) + h(s)$
- while unvisited cells remain:
  - choose unvisited cell c with lowest $f(c)$

```
    – mark it as "visited"
    – for each neighbor cell $n$:
        * update $g(n)$ to $\texttt{min}[g(n), g(c) + d(c, n)]$
        * update $f(n)$ to $g(n) + h(n)$
```

We use $h(c)$ to denote the heuristic function estimating the length of the path from cell c to the goal.

We can use the grid below to practice executing the A* algorithm:



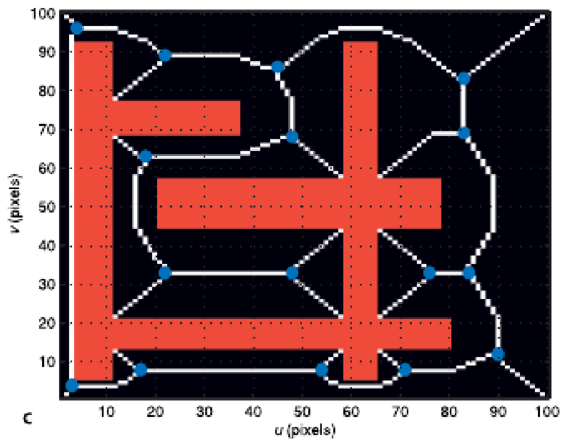
Note how much faster A* gets to the goal compared to Dijkstra's algorithm, and how many fewer cells it visits on the way there. Even though A* has the same worst-case performance as Dijkstra's algorithm, in practice, A* is generally much faster. It is a commonly used algorithm for real-world applications.

3.4 Voronoi Roadmap

Dijkstra's algorithm and A* are good at finding the shortest path to the goal. Short paths are often desirable, as they take least amount of time to traverse, save energy, etc. However, they have a very important practical disadvantage: by nature, they bring the robot very close to obstacles; in fact, as close as possible to some obstacles. This means that a small error in execution or calibration can lead to a collision.

A different approach is to ask the robot to reach the goal while staying as far as possible from obstacles. This is equivalent to traveling on the boundaries of the **Voronoi diagram**.

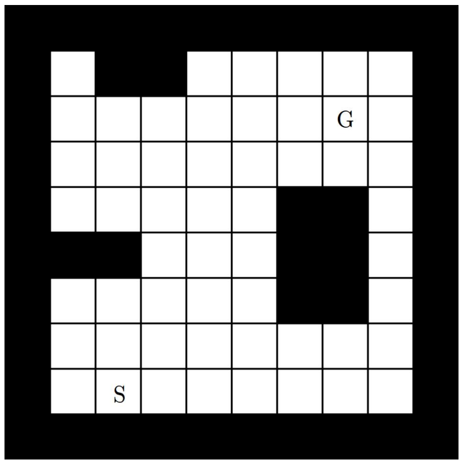
The Voronoi region of an obstacle is the area around it whose points are closer to that obstacle than any other one. The boundaries of Voronoi areas are points that are equidistant from two or more obstacles. (Example below from Peter Corke, "Robotics, Vision and Control".)



On a discrete grid, we can use the brushfire algorithm, which computes, for each cell c, the minimum distance from c to an obstacle, or $o(c)$:

```
• mark all obstacle cells as having $o(c) = 0$ and place them in open list
• while open list is not empty
    – choose cell $c$ in open list with lowest $o(c)$
    – for each neighbor $n$ that is not in closed list:
        ∗ update $o(n)$ to $\texttt{min}[o(n), o(c) + d(c, n)]$
        ∗ place $n$ in open list
    – place $c$ in closed list
```

We can use the grid below to practice executing the brushfire algorithm:



In this class, we will not go into details on how the Voronoi diagram is computed. However, assuming it is given to use, then we can use it to compute a safe-as-possible path to the goal as follows:

- from the start cell, get to the closest cell that is on a Voronoi ridge. Call this cell the V-start.
- from the goal cell, get to the closest cell that is on a Voronoi ridge. Call this cell the V-goal.
- run Dijkstra's algorithm exclusively on Voronoi ridge cells to find the shortest path between the V-start cell and the V-goal cell.

Voronoi path planning is in a way like using a pre-defined public transit network: find your way from the start to the nearest station, take public transit to the station nearest to the goal, then find the shortest path to the goal.

4 Stochastic or probabilistic approaches

All the methods described so far can make some guarantees about the path they find. Dijkstra's algorithm and A* find the shortest path. The Voronoi roadmap stays as far away from obstacles as possible. The downside however is that the worst case running time is exponential in the number of cells in the grid.

While these methods can in theory be applied in an arbitrary number of dimensions, they become less practical as the dimensionality of the configuration space increases. In practice, they are often used for 2- or 3-dimensional spaces, often for mobile robots. Highly articulated arms, with 6- or 7-dimensional configuration spaces, require a different approach.

Two of the most powerful approaches for dealing with such cases fall in the category of stochastic methods. The configuration space is not searched in an orderly, potentially exhaustive fashion. Rather, these algorithms explore by sampling random points in configuration space.

4.1 Probabilistic Roadmaps (PRM)

PRMs work by choosing and connecting random points in configuration space until there is a "reasonably" well-connected network of points. A path from a start point to a goal point is then found by hopping on and off this pre-computed network, much like in the Voronoi case.

A simple variant of the PRM construction algorithm is:

- while number of points in roadmap lower than threshold:
  - sample random point in configuration space
  - if point is feasible:
    - connect new point to all other points on roadmap via straight lines, as long as lines do not go through obstacles

Once we've done this computation, we can compute the path from a start to a goal via the following algorithm:

- for the start point, find the closest point in the PRM that can be connected via a straight line that does not go through an obstacle. Call this point PRM-start.
- for the goal point, find the closest point in the PRM that can be connected via a straight line that does not go through an obstacle. Call this point PRM-goal.
- run Dijkstra's algorithm exclusively on the PRM to find the shortest path between PRM-start and PRM-goal.

4.2 Rapidly-exploring Random Trees (RRT)

RRTs work by "growing" away from the start point, in random directions, until it is possible to connect to the goal. A simple variant works as follows:

- insert start point in tree.
- while tree can not connect to goal:
  - sample random point r in configuration space.
  - find point p in tree that is closest to r.
  - add branch of predefined length from p in the direction of r. If branch intersects an obstacle, shorten branch until it no longer intersects obstacle.

There are many variations on this theme; production-quality implementations of RRT's have additional subtleties. However, the key idea remains the same as the one shown here.

4.3 Properties of stochastic algorithms

While PRMs and RRTs are some of the most powerful path planning methods in high-dimensional spaces, and they tend to work very well in practice, it is important to keep in mind the following characteristics:

- algorithms belonging to this family can rarely make any guarantees regarding the worst-case running time. Sometimes, they can be shown to be **probabilistically complete**: if a path exists, it will be found in finite time. However, the time to find the path, while finite, can potentially be very very large (even larger than exhaustive search). For good stochastic algorithms though, that very rarely happens.
- similarly, there are no guarantees regarding the quality of the found path. While, these algorithms work well in practice, they are not guaranteed to find the shortest or highest quality path.
- there is often need for post-processing of the resulting path, to eliminate unneeded zigs and zags.