CMSC 425: Lecture 19
Motion Panning: Finding Paths

Reading: Today's material comes from various sources, including "AI Game Programming Wisdom 2" by S. Rabin and "Planning Algorithms" by S. M. LaValle (Chapts. 4 and 5).

Finding Paths: Last time we discussed how the problem of planning the motion of a robot amidst a set of obstacles could be reduced to the task of computing a path between two points in the *configuration space* associated with the robot. Unfortunately, computing such a path for even a single point in space is a nontrivial problem. Typically, we are interested not merely in the existence of a path, but rather finding a "good" path, where goodness connotes different things in different contexts (e.g., shortness, low-energy, natural looking, etc.) In this lecture, we will present a number of different approaches for computing such paths.

Our illustrations will be in terms of 2-dimensional space, but these concepts generalize readily to higher dimensions (and configuration spaces). Throughout, let s denote the start point (or start configuration) and let t denote the target point (or target configuration), and let O denote a collection of obstacles. The objective is to compute a short path from s to t while avoiding the obstacles.

Potential-Field Navigation: The analogy to understand this approach is to imagine a smoothly varying terrain with hills and valleys. Suppose you place your target at the bottom of a bowl-shape, and your starting point anywhere on the surface of the bowl. Create obstacles that are like "plateaus" sticking up from the bowl. Now, place a marble at the start point s and let it go. The marble will gently roll downhill towards the target points t, while avoiding the high plateaus.

How can we base a path finding algorithm on this idea? Consider a workspace consisting of a start point s, target t, and obstacles O (see Fig. 1(a)). The idea is to model this as a terrain in 3-dimensional space, so that by simulating a sliding marble placed at s, it will naturally roll towards t while sliding around obstacles (see Fig. 1(d)).

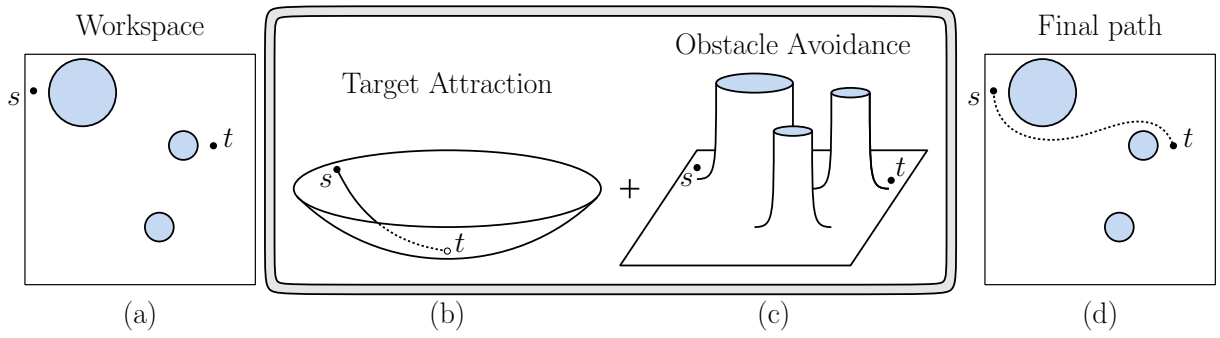


Fig. 1: Computing paths by potential-field navigation.

We do this as follows:

Attraction to Target: Set up an *attractive field* to the target point so that by minimizing this potential we approach the target. For example, this might just be the squared

distance from the current point p to the target, $c \cdot \text{dist}(p,t)^2$, for some constant c. This defines a parabolic "bowl" shape, where t sits at the lowest point of the bowl (see Fig. 1(b)).

Avoidance of Obstacles: We set up a *repulsive field* around obstacles (and generally any forbidden regions of the configuration space). For example, let O denote the set of obstacles, and let p be the current point. For any $o \in O$ define the potential field at p to be the inverse of the squared distance from o to p, that is, $1/\text{dist}(p,o)^2$ (see Fig. 1(c)).

Then, we define the overall potential to be the sum of all these potentials. Finally, we could include weight factors w_t and w_o to control the relative strength of attraction versus repulsion:

$$\Psi(p) \;=\; w_t \cdot \text{dist}(p,t)^2 + w_o \cdot \sum_{o \in O} \frac{1}{\text{dist}(p,o)^2}.$$

Think of $\Psi(p)$ as defining the *height* of the potential terrain at point p. This function achieves its lowest value at t (assuming t is not too close to any obstacle) and its highest value (of ∞) along the boundary of any obstacle. The final potential field Ψ is the sum of these various functions.

Path Finding via Gradient Descent: Given our potential field, we can apply a physics simulator to let our robotic marble flow "downhill" from s to t (and hope it eventually arrives!)

How do we compute this path? A natural approach is to compute a path of *steepest descent*. Given a point $p = (x,y)$, let $\Psi(x,y)$ denote the value of the potential field at any point direction (x,y), then the direction of steepest ascent it given by the *gradient vector*, which can be computed from the partial derivatives of Ψ. More formally, the gradient is

$$\nabla\Psi \;=\; \left(\frac{\partial\Psi}{\partial x}, \frac{\partial\Psi}{\partial y}\right)$$

(see Fig. 2). You might wonder why the partial derivative is used here. Observe, for example that if the function grows very rapidly with x, but is almost flat with respect to y, then the gradient will have a very high x-component and the y-component will be very close to zero. It takes a bit of calculus to show that among all directions, the gradient provides the direction of most rapid change. By the way, one reason for using squared distances in the above potential function, rather than standard Euclidean distance. It is much easier to compute derivatives of polynomial functions than functions involving square roots.

Given any starting point p, we can compute the next point along the direction of steepest descent as

$$p' \;\leftarrow\; p - \delta \cdot \nabla\Psi(p),$$

for a suitably small *step size* δ. By repeatedly recomputing the gradient and taking another step, we will eventually walk to a *local minimum* of the potential function.

There is some art in how the step size is determined. If the step size is too big, we may shoot past the minimum, and if the step size is too small, we may require many iterations before converging.

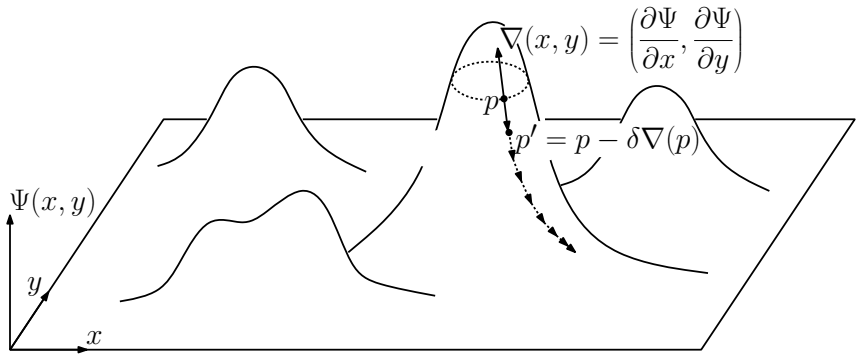


Fig. 2: Finding the path via steepest descent.

Advantages: The potential-field method is very easy to implement. Because the movement point naturally follows a smooth energy-minimizing path, when it converges, it tends to result in smooth, natural-looking motion. However, it is best used for simple motions, where the desired path doesn't involve many twists and turns.

Disadvantages: In addition to the difficulty of determining a good step size, the most significant disadvantage of the potential-based method for path planning is that it can get trapped in *local minima*. If t is not at the bottom of the local minimum then the algorithm simply gets stuck.

Discretizing Configuration Space: Because continuous spaces are difficult to search, it is common to find paths by a two-step process:

Discretize: Reduce the problem to one of searching a discrete structure (either a graph or a subdivision of space).

Search: Apply a path-finding algorithm (such as Breadth-First Search, Dijkstra's algorithm, or A*) to compute the path in the discrete structures. (We will discuss these algorithms in future lectures.)

In the remainder of this lecture, we will discuss a number of approaches for computing the aforementioned discrete structure.

Waypoints and Road Maps: Perhaps the simplest approach for generating a navigation graph, is to scatter a large number of points throughout free space, sometimes called *waypoints*, and then connect nearby waypoints to one another if the segment between them does not intersect any obstacles. (Since this is generally happening in configuration space, the points are in configuration free space and the segments should not intersect any C-obstacles.)

The edges of this graph can be labeled with the distance[1] between the associated points. The resulting graph is called a *road map*. Given the location of the start point s and the destination point t, we join these with edges to nearby waypoints. Finally, we can invoke a shortest path algorithm to compute the final path. If the graph is connected, then this is

[1]In the case of translational motion, distance is an easily defined notion. When the configuration space include rotations, we need to define distances in terms of both rotations and translations.

guaranteed to yield a valid path in configuration space, which is then translated back to a motion plan in the robot's workspace.

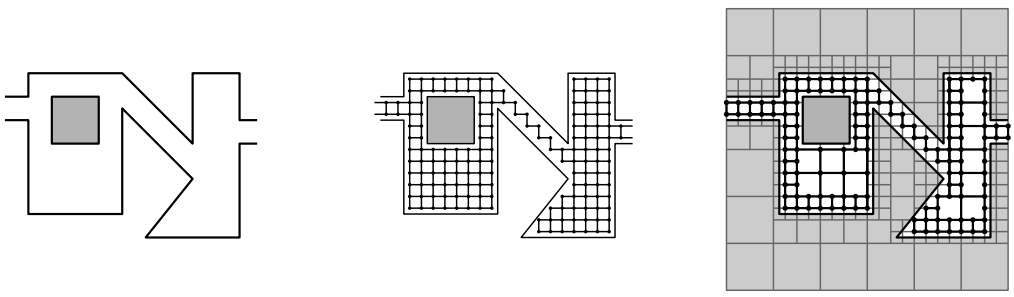
Selecting Waypoints: There are a number of methods for computing waypoints. Here are a few:

Placed by the game designer: The game designer has a notion of where it is natural for the game agents to move, and so places waypoints along routes that he/she deems to be important. For example, this would include points near the entrances and exits of rooms in an indoor environment or along the streets or crosswalks in an urban setting. This gives the designer a lot of control of the motion of the game agents, and a lot of flexibility to add more points where motion is highly constrained and fewer where motion is unconstrained.

In a large environment, however, there may be too many waypoints for a single designer to place. Thus, we would like something more automated.

Grid: The simplest way to cover a large area is to overlay a square grid of sufficiently high density and generate waypoints along the vertices of this grid that lie in free space (see Fig. 3(b)). Each grid point has up to four possible neighbors (assuming 2-dimensional space—in d-dimensional space each has up to $2d$ neighbors).

This has the advantage of simplicity, but there are some notable drawbacks. First, it can result in the generation of a very large number of grid points. The grid resolution has to be set small enough that the narrowest corridor has sufficiently many points, but then wide open areas will have many more points than are needed (see Fig. 3(b)). A second drawback is the path segments generated are all parallel to the coordinate axes. Hence, such a path will zig-zag excessively if the path travels diagonally. This issued can be ameliorated by a postprocessing pass that smooths out the path, but unless care is taken, the smoothing process might introduce shortcuts that pass through obstacles, which is not acceptable.



(a) (b) (c)

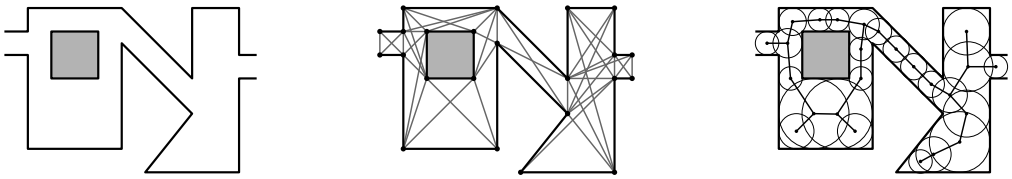
Fig. 3: A road map for a set of obstacles (a) based on waypoints generated on: (b) a grid and (c) a quadtree.

Adaptive Grid (Quadtree): One way to deal with the grid's lack of adaptivity is to apply a hierarchical point placement system that adapts the density of point placement to the distance to the closest obstacle. A natural generalization of the grid approach is to place waypoints on the vertices of a quadtree decomposition. Recall that a quadtree

decomposes space into square *cells* (or generally hypercubes in higher dimensional space). A quadtree cell is said to be *stabbed* if the boundary of some obstacle cuts through it. Assuming that we can detect when a quadtree cell is stabbed, we repeatedly refine any stabbed cells until the cells are deemed to be sufficiently small. The waypoints are then placed at the vertices of the quadtree that lie in free space (see Fig. 3(c)).

While this adds adaptivity, it still does not resolve the issue of path segments that are parallel to the coordinate axes.

Boundary Vertices and the Visibility Graph: In order to deal with the problem of path segments that are aligned with the coordinate axes, we would like a method of generating waypoints that is independent of the coordinate system, and relies solely on the geometry of free space. Let us assume for now that we are working in 2-dimensional space, and the configuration space is bounded by line segments. If one is interested in shortest paths (assuming the Euclidean distance) it is easy to prove that such a path will only make turns at the vertices of the obstacle vertices. We say that two vertices of the boundary of free space are *visible* if the line segment between them does not intersect any obstacles. The *visibility graph* is a graph whose vertices are the vertices of the boundary of free space and whose edges are the visible pairs (see Fig. 3(b)). It follows from the above remarks that the shortest path between any two points is a path in the visibility graph.



(a) (b) (c)

Fig. 4: A road map for a set of obstacles (a) based on waypoints generated on: (b) the visibility graph of the obstacle vertices, and (c) the medial axis.

The visibility graph is an intrinsic structure, meaning that it depends only on the object's geometry, not on the placement of the coordinate system. While it has a number of nice theoretical properties, it also has a number of drawbacks that make it unsuitable as a general purpose solution. First, the number of edges in the visibility graph can be as high as $O(n^2)$, where n is the number of vertices. If n is very large, this quadratic size may be unacceptable. This problem can be ameliorated by pruning the graph, say by keeping only the shorter of two edges that share a common vertex and have a very similar slope.

A second problem of the visibility graph is that the paths it generates, while of minimal length, have the undesirable property that they point scrapes right only the boundary. This doesn't generate very natural looking motion. This issue also can be ameliorated by first constructing an artificial boundary that is slightly offset from the actual boundary, and then constructing the visibility graph of the offset boundary.

A third problem with the visibility graph is that it guarantees shortest paths only in

2-dimensional space. In 3-dimensional space, it is not longer the case that the shortest path between points bends at vertices. It may bend in the interior of an boundary edge. The locations of these interior bending points cannot be predicted in advance (since they generally depend on the locations of the starting and ending points).

Medial-Axis Waypoints: The shortcomings of the visibility graph suggest a very different approach to path finding. People who walk down a corridor do not usually "scrape" along the boundary. Rather they usually seek a path near the center of the corridor, that is, they seek a path of *maximum clearance* from the obstacles. How can we compute such a path?

We say that a circular disk D lying entirely in free space is *maximal* if there is no obstacle-free disk of larger radius that contains D. The union of the centers of all maximal disks naturally defines a set of points that runs along the center of the free-space domain. It is a fundamental object in geometry, called the *medial axis* (see Fig. 4(c)).

By sampling waypoints on or near the medial axis, the robot will naturally move along the centers of corridors. (Of course, you can add to this a bit of random variation.) This method of placing waypoints is best for 2-dimensional domains (since it is messier to compute the medial axis of higher dimensional configuration spaces).

Adaptive Randomized Placement and PRMs: Another adaptive approach to placing waypoints that avoids dependencies on the coordinate axes is to select the waypoint placement randomly. Here is the idea. On the ith iteration, we generate a random point p within the domain of interest. We test to see whether p lies within free space. If not we discard it and go to the next iteration. Otherwise, we next attempt to connect p to other nearby previously sampled points. This can be done in various ways. The simplest way to do this is to compute the k nearest neighbors of p. (Using distance alone as a criterion might produce neighbors that lie exclusively to one side of p, which is not good. More sophisticate methods can be used to ensure, if possible, that p's neighbors cover all the directions about p.) Next, we consider a line segment joining p to each of these neighbors. If the line segment lies entirely within free space, we add this segment to our road map. We repeat this process until some stopping condition is met, for example, until some fixed number of successful samples are generated or until the entire structure consists of one connected component. Because of the randomized nature in which points are generated, the resulting structure is called a *probabilistic road map* (or PRM).

PRMs are very popular in the field of robotics. They can be applied in arbitrary dimensions. It can be proved that, if there is a path between two configurations then with high probability the PRM will eventually discover it. Of course if the path travels through a narrow passageway (as seen in the middle part of the obstacle set of Fig. 5) it may take a lot of samples to discover this connection.

Because points are randomly generated throughout the domain, it suffers from some of the same issues as uniform grids. Wide open areas of free space will receive an excessive number of waypoints while narrow corridors may receive too few. Unlike uniform grids, it is possible to detect that a newly added waypoint is redundant and so it can be ignored. PRMs suffer the same problem as all the other waypoint methods we have discussed so far. The paths do not generally travel along natural paths, but rather they zig-zag from one waypoint to the next.

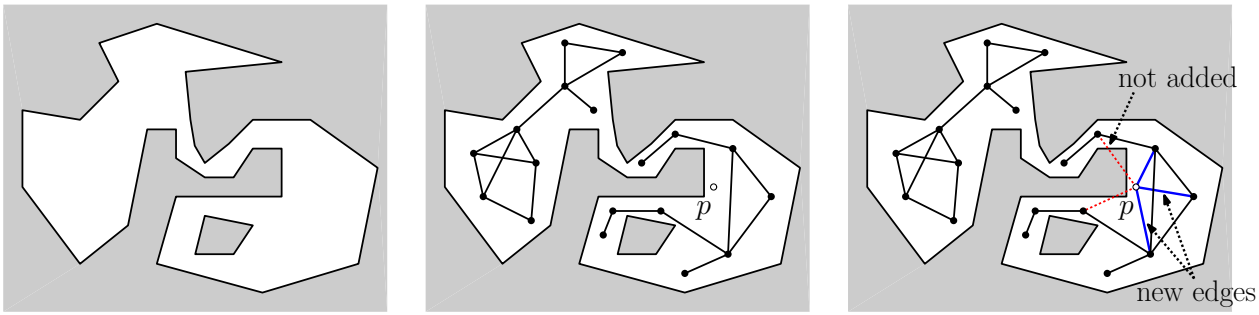


Fig. 5: Generating a probabilistic road map (PRM) for a set of obstacles. (b) The roadmap after iteration $i-1$ and (c) the result of adding the ith point. Edges that intersect obstacles (red) are not added.

Rapidly-expanded Random Trees (RRTs): One of the issues with PRMs is that the structure that is generated is not necessarily connected. Another popular adaptive approach to generating a roadmap through randomization is through that process that guarantees that the structure that is generated is connected, and in fact it is a spanning tree over the set of sample points. Spanning trees are nice for navigation because (due to the fact that they are connected and acyclic) there is a unique path between any two points. While this may not be great for computing shortest paths, it is useful for determining the existence of any valid motion.

As with PRMs, the process begins by random sampling points from the domain. In this case, we will keep every sampled point, even if it does not lie within free space. Let us assume that we have already computed a spanning tree for the existing set of sample points (consider just the line segment p_0p_1 in Fig. 6(a)), and we are considering the addition of a new sample point p. We compute the closest point q on the current spanning tree to p. Note that q does not need to be a sampled point. It is allowed to lie within the interior of an edge of the spanning tree. If so, we add the point q as a new vertex to the spanning tree. We then consider the line segment from p to q. If this line segment lies entirely within free space, we add it to the tree (see points p_2 and q_2 in Fig. 6(a) and p_3 and q_3 in Fig. 6(b)). If not (see q_4 to p_4 in Fig. 6(c)), we trim the segment back to obtain the longest subsegment that lies within free space with one endpoint at q. Let qp' denote this segment (see $q_4p'_4$ in Fig. 6(d)). We add this segment to the tree. The result is called a *rapidly-expanding random tree* or (RRT).

Next to PRMs, RRTs are perhaps the most widely used method for computing connectivity structures in configuration spaces. Notice that both PRMs and RRTs have the advantage that they can be applied in configuration spaces of arbitrary dimensions.

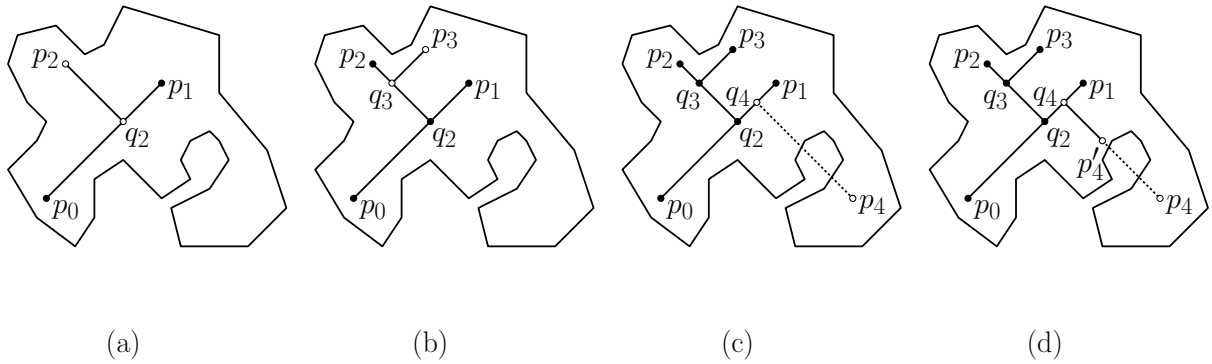


Fig. 6: Generating a roadmap through the use of rapidly-expanding random trees (RRTs).