CSCI–4190 Introduction to Robotic Algorithms, Spring 2006
Assignment 1: out Monday January 23, due Thursday February 9

Comparison of Random versus Quasirandom Sampling for PRM Motion Planners

For this assignment, you will write a program that plans a path for a mobile robot moving in the plane in the presence of obstacles. The motion planning technique you will implement is a Probabilistic Roadmap (PRM) planner. I am asking you to implement a basic PRM planner and a simple variation and compare the performance of these two approaches.

A few notes on this assignment:

- This assignment is to be done in C++. I will provide a program that will read in the description of the world from a file and draw it on the screen. I will also provide support code for intersecting two line segments which you will need. *Except for the provided support code, you are to implement all aspects of this assignment yourself.*
- I expect to support both Windows and UNIX (definitely Linux, probably FreeBSD, not sure about MacOS) environments. However, your code must compile and run under a Linux environment. As long as you stick to standard C++ libraries, you should be fine. The support code will depend on the OpenGL and glut libraries, so you may need to install these.
- Support code and further details about the assignment will be on the "assignment 1 information page" off the course home page.
- We will use the WebCT discussion boards for general questions and clarifications.

Some assumptions that we will make:

- The robot is circular. (Its radius will be a parameter of the program.)
- The world will consist of a rectangular boundary and convex polygonal obstacles. These obstacles may overlap.
- The path your program will plan will consist of a sequence of straight line segments.

PRM motion planners

Here is a basic PRM motion planning algorithm. There are a number a variations and refinements that we'll discuss when we cover this in class later in the semester, but this will do for now.

A PRM motion planner constructs a graph from a set of points sampled from the robot's configuration space. The robot's start and goal configurations are connected to this graph, and then the graph is searched for a path form the start to the goal nodes. The algorithm is as follows:

1. Sample N collision-free robot configuration points. Let this set of points be V. (These will be the vertices in the graph.)
2. For each point $p \in V$:
   (a) Find the k closest (other) points in V. Let this set of closest points be Q.
   (b) For each point $q \in Q$, if the straight-line path from p to q is collision free, then add an edge in the graph from p to q.
3. Add the start and goal configurations as vertices in the graph and attempt to connect them to their k closest vertices, as above.
4. Search (using A*) to find a path from the start vertex to the goal vertex.

Random sampling

For "random sampling", you will simply pick a random point in the rectangular world boundary (in step 1). Since this is a cartesian space, you can simply pick each coordinate independently. You should generate a random number as follows:

```
#include <stdlib.h>

double r = drand48();
```

This generates a pseudorandom number between 0 (inclusive) and 1 (exclusive).

Note that you can "seed" the random number generator using the `srand48` procedure. The `drand48` procedure will return the same sequence of pseudorandom numbers for a given seed value.

Quasirandom sampling

The name "quasirandom" is somewhat misleading here, as the points selected under quasirandom sampling really aren't random at all. For this assignment, we will be using a "Halton sequence" to generate two dimensional points.

To generate the i^{th} point (starting from $i = 0$), we need two relatively prime numbers b_1 and b_2. These are generally taken to be the two smallest primes, so $b_1 = 2$ and $b_2 = 3$. The i^{th} point is given by:

$$\begin{bmatrix} r_{b_1}(i) & r_{b_2}(i) \end{bmatrix}$$

In order to compute the $r_b(i)$ function, we first need to take the number i and express it in base b:

$$i = a_0 + a_1 b + a_2 b^2 + \cdots$$

where each a coefficient is an integer between 0 and $b-1$, inclusive. The $r_b(i)$ function can then be expressed in terms of the base-b digits $a_0, a_1, a_2, \ldots$

$$r_b(i) = \frac{a_0}{b} + \frac{a_1}{b^2} + \frac{a_2}{b^3} + \cdots$$

In essence, this just takes the base-b representation of i and "mirrors" it as a base-b decimal number. For example:

i	i in base 2	$r_2(i)$ in base 2	$r_2(i)$	i in base 3	$r_3(i)$ in base 3	$r_3(i)$
0	0	0.0	0	0	0.0	0
1	1	0.1	1/2	1	0.1	1/3
2	10	0.01	1/4	2	0.2	2/3
3	11	0.11	3/4	10	0.01	1/9
4	100	0.001	1/8	11	0.11	4/9
5	101	0.101	5/8	12	0.21	7/9
6	110	0.011	3/8	20	0.02	2/9
7	111	0.111	7/8	21	0.12	5/9
8	1000	0.0001	1/16	22	0.22	8/9
9	1001	0.1001	9/16	100	0.001	1/27

Note that this produces points sampled in the unit square, so you will scale these points to select robot configurations in the rectangular world boundary.

Geometric tests

There are two geometric tests that are required for constructing a PRM for this assignment:

1. whether the robot, at a given configuration, intersects with an obstacle
2. whether the robot, as it travels along a straight-line path from one configuration to another, collides with an obstacle

These tests can both be solved by determining whether two convex polygons intersect.

For the first test, we will bound the circular robot with a regular polygon. I suggest using an octagon, though the number of sides could be a parameter to your program. You can then test whether this polygon intersects with any obstacle. This results in a slightly conservative test, but that's fine. (Determining whether a circle intersects with a polygon is not too complicated, but it would require writing another intersection test.)

For the second test, we will bound the points swept out by the robot as it moves along a straight-line path. For PRMs, we know that the endpoints have already been checked for intersection with obstacles, so we can therefore take the rectangle that bounds the other untested robot points.

Convex polygon intersection

To test if two convex polygons intersect, you can follow the following basic procedure. Note that we will consider a polygon to consist of its boundary and all the points in the interior — therefore one polygon contained completely inside another does intersect with the other polygon, even though no edges of the polygon intersect.

1. If any vertex of one polygon lies inside (or on the boundary) of the other polygon, then they intersect.
2. Otherwise, if any edge of one polygon intersects with an edge of the other polygon, then they intersect.
3. Otherwise, they do not intersect.

I will provide a procedure to do line segment intersection, so this will help you with step 2.

For step 1, you can test if a point lies inside or on the boundary of a polygon by computing the "signed distance" of the point with respect to the line defined by each edge. A positive signed distance means that that point lies on the opposite side of the line from the polygon and therefore is completely outside the polygon. If the signed distance is nonpositive (i.e., less than or equal to zero) for *all* edges, then the point lies inside the polygon or on its boundary.

The signed distance can be computed very easily using the cross product. Choose a vector $\vec{v}$ that lies on the edge of the polygon. This vector defines a directed line — the direction of the vector should be such that the interior of the polygon lies of the left side (when facing in the direction of the vector). Then form a vector $\vec{u}$ from any point on the line to the point in question. The signed distance is given by:

$$\frac{(\vec{u} \times \vec{v})_z}{|\vec{v}|} = \frac{u_x v_y - u_y v_x}{|\vec{v}|}$$

However, since we only care about the sign of the signed distance, you can just test the value $(u_x v_y - u_y v_x)$.

A* search

You should use the A* search to search the graph to find a path from the start to goal node. A* is a heuristic best-first search. Best first searches follow the following basic algorithm, phrased in a "queue formulation:"

- Put the start node on Q
- Repeat
  - If Q is empty, return failure
  - Remove the best node n from Q
  - If n is the goal, then return success
  - Otherwise, put the successors of n on Q

The A* search defines "best" as the lowest total estimated path cost: the actual cost of the path taken so far, plus a heuristic estimate of the cost remaining to get to the goal. The heuristic you should use to estimate the remaining cost is the straight-line distance from the last node on the path to the goal. A* search bears some resemblance to Dijkstra's algorithm, but A* search will generally be much better because of the heuristic information directing it towards the goal.

Here is the actual algorithm that you should use to implement A* search with this heuristic:

- Put the start node on a list OPEN
- Create an empty list CLOSED
- Repeat:
  - If OPEN is empty, return failure
  - Select the node n from OPEN with lowest $f(\cdot)$
    Break ties arbitrarily, but always in favor of a goal.
  - Remove n from OPEN and add it to CLOSED
  - If n is a goal, return success
  - Find the neighbors V of n
  - For each node $v \in$ V:
    - if v is not on OPEN or CLOSED, add to OPEN
    - if v is on OPEN, update $f(v)$ if necessary

A few explanatory notes:

- Note that the OPEN list should really be a priority queue; however, I suggest you implement whatever is easiest for you to code, even if it just does linear search to find the node with lowest $f(\cdot)$ value. Similarly, you may implement the CLOSED list however is simplest for you to code.
- This algorithm assumes that you store the total estimated cost $f(\cdot)$ with each node. You will also need to store a "pointer" from each node to its predecessor in the best path found to that node so that you can recover the path once the search is complete. These can be initialized when you put a node on the OPEN list.
- The total estimated cost for a node n is defined as $f(n) = g(n) + h(n)$ where $g(n)$ is the actual path cost and $h(n)$ is the heuristic estimate of the cost remaining to the goal.
- When you compute $f(v)$ for a child v of n, you will first compute $g(v) = g(n) + c(n, v)$ where $c(n, v)$ is the cost of the edge from n to v. This is the total cost of the actual path by which v was reached from the start node. Note that if the node v was already discovered and put on the OPEN list, this new path to v may or may not be a lower-cost path. If this $f(v)$ is lower, then you need to update the $f(\cdot)$ value as well as change the pointer to its predecessor to n.

A few additional notes

- Your program must display its results to the graphics window. This, of course, will also be useful for debugging your code. Details on what to display will be on the assignment web page.
- The support code will make use of a version of the DOLT library for graphical display. We are currently updating/modifying this library and should have it ready in a day or two. In the meantime, I have put an include file from the original library, `geometry.h` on the assignment web page so you can see what the classes for geometric representation will be. (I don't expect these to change.) Note that although the code makes reference to CGAL (a computational geometry library), we will not be using it.

Questions

- I will ask you to test your code using a variety of worlds, a few of which I will provide, but others you will have to create yourself. You can vary the number of obstacles, the spacing of obstacles, the size of the world, etc.
- I will ask you to answer a number of (written) questions and turn in hardcopy of your answers. Details to be provided on the assignment web page.