

Motion planning

In this assignment, you will implement a simple motion planner using an approximate cell decomposition approach with a quadtree representation. There will also be a few written questions regarding your implementation and related issues.

You may do the programming component in any language you like, but I will only be supporting C++ (Visual C++ and g++). I suggest you use C++ for the following reasons:

- The CGAL library provides support for geometric representation and computation, and it is only available for C++. The portions of this library we will make use of are not so complicated that you couldn't implement them yourself, but this would probably be a few hundred lines of code you'd have to write and debug.
- If, as I am planning, the four assignments for this class build upon each other and will culminate with running your code on the robots in my lab, then you will have to interface to the robot support code in C/C++.

If you want to do some work to interface to C/C++ code, you should be able to work in any programming language. When I say that I am supporting C++ (in UNIX with g++ and in Windows with VisualC++), I mean that I will provide some sample programs that demonstrate using the support code (in this case, the CGAL library and some graphics library).

A. Implementing a simple motion planner

Here are the basic assumptions we will make for this motion planner:

- The robot cannot rotate, so its configuration may be specified by a two dimensional vector.
- The world boundary will be rectangular.
- The robot will be modeled as a convex polygon.
- All obstacles will be modeled as convex polygons, and they may overlap.

The requirements for your motion planner are the following:

1. Read the problem description from files as specified in following section.
2. Create the configuration space representation of the world.
3. Create a quadtree representation of the configuration space.
4. Create an adjacency graph from the quadtree.
5. Search the graph to find the shortest collision-free path from the start to the goal.
6. Be able to graphically display the results from any of the above steps.

By Thursday January 25, you should turn in a program that accomplishes steps 1 and 2 (and graphical display of the results from these steps). A complete implementation is due on February 1. You don't need to have any sort of fancy user interface — a simple console text interface is fine. You should turn in an executable program and also commented source

code that I can compile. You can feel free to share code for parts 1 and 6 of this assignment only.

Please see the course web page for more information on support code and where you can get the CGAL and graphics libraries.

A.1 Read the problem description file

The description of the problem is contained in a file, and you should be able to specify this filename somehow — either on the command line or by prompting the user to enter the filename. The problem description file will contain names of other files to specify the world description and the robot shape; it will also contain the start and goal positions of the robot. All files should be able to have comments using the '#' sign as the first character of a line to preface a comment.

I will provide some sample code to read a basic input file which you can modify. It is acceptable for your program to crash if it is given invalid input, i.e. I'd rather you spent time on stuff related to motion planning than on making your input routines robust for stupid users. Of course, your program should not crash if it is given valid input.

You should feel free to share code for this part of the assignment. I will provide a number of sample input files.

A.1.1 Problem description file

Here's a sample problem description file:

```
# The world file contains the boundary and obstacles
world= world1.txt

# The robot file contains the shape of the robot
robot= robot1.txt

# The start and goal positions for this problem. (These coordinates
# specify the position of the robot reference frame)
start= 1.4 2.8
goal= 10 11.4
```

Assume no whitespace between keywords and the equals sign. Assume there is white space between the equals sign and the parameter (and between subsequent parameters).

A.1.2 World description file

The world description file contains a list of convex polygons. Each polygon is given a name (which I suggest you store as it may be useful). The first polygon is assumed to be the boundary, and subsequent polygons are assumed to be obstacles, irrespective of what their names might be!

Each polygon is represented as a list of points, one per line, which consist of two whitespace separated floating point numbers. The points must appear in counter-clockwise order. The list of points for a polygon is delimited by braces, with the last brace being the first character on a new line. Assume that you could have a degenerate polygon (a two sided polygon).

Here's an example:

```
Boundary {
  -5 -2.3
  8.7 -2.3
  8.7 10
  -5 10
}
```

```
Obstacle1 {
  4 4.3
  6.2 4.2
  6.4 4.5
  6.2 5.7
  3.7 6.1
}
```

A.1.3 Robot description file

The robot description file contains a single convex polygon in the same format as above which describes the shape of the robot. You can assume the polygon name is the name of the robot!

Assume that the this polygon is specified with respect to the reference frame of the robot. Note that the origin need not be inside this polygon!

A.2 Create the configuration space representation

Since both the robot and the obstacles are convex, we can use a convex hull procedure to generate the configuration space obstacle. If we assume that the reference point on the robot is the origin of the robot frame, then we can compute the configuration space obstacle by the following algorithm:

- Let \mathcal{P} be a list of points, initially empty
- For each vertex of the obstacle, \vec{O} with respect to the world frame:
 - For each vertex of the robot, \vec{R} with respect to the robot frame:
 - * Add the point $(\vec{O} - \vec{R})$ to \mathcal{P}
- Compute the convex hull of the points in \mathcal{P}

The CGAL library provides support for representing points and polygons and for computing the convex hull (among other things).

Note that the world boundary is different than an obstacle, so you will need to create its configuration space obstacle differently.

A.3 Create the quadtree representation

We use a quadtree representation to efficiently represent space. This representation has increased resolution only where it is necessary. The basic concept behind quadtrees extends to arbitrary dimensions; in three dimensions, they are called octrees, and in m -dimensional space, they are called 2^m -trees.

The basic idea (in two dimensions) is that we start with a single rectangle and iteratively subdivide this rectangle until either we can generate a path for the robot or until the

subdivision has reached some specified resolution. (You may either check for a path after each iteration or simply perform a fixed number of iterations.)

At each iteration, we consider each sub-rectangle. If it is completely free or completely occupied (by a configuration space obstacle), then it is not subdivided in this or any subsequent iteration. If it is partially occupied, then we divide it equally into four subrectangles.

The CGAL library provides functions that will find the intersections of polygons which you can use to determine whether a subrectangle is free, occupied, or partially occupied.

A.4 Create the adjacency graph

Although (as the name implies) quadtrees implicitly construct a hierarchical representation of space that can be represented as a tree, for motion planning, we want a graph in which each subrectangle is a node and there is an edge between the nodes of adjacent subrectangles.

Assume that we do not consider diagonal connectivity. Note that a large subrectangle may have multiple smaller subrectangles adjacent to it on any given side. It will probably be easiest to create the adjacency graph and create the quadtree representation of the configuration space at the same time.

A.5 Search the adjacency graph

Use the A* search algorithm to search the adjacency graph to find the shortest path from the start to the goal configuration. I suggest you use the following formulation:

1. Let OPEN be a list initially containing the start node
2. Let CLOSED be a list, initially empty
3. If OPEN is empty then return fail
4. Remove the node on OPEN with minimum $f()$, let this node be N
5. Add N to the CLOSED list
6. If N is the goal node, the return success
7. For each child (i.e. neighbor) N' of N :
 - a. If N' is on the OPEN list, then update its $f()$ value if necessary
 - b. If N' is on the CLOSED list, then do nothing
 - c. Otherwise, add N' to the OPEN list
8. Go to step 3

Recall that A* is essentially a best-first heuristic search, where “best” means minimum estimated cost to reach the goal. This cost is represented by the $f()$ value of a node:

$$f(n) = g(n) + h(n) \tag{1}$$

where $g(n)$ is the cost to reach node n from the start node and where $h(n)$ is an estimate of the distance remaining to reach the goal from n . The above algorithm assumes that the heuristic function $h()$ is *admissible* and *monotonic*. An admissible heuristic will never overestimate the distance to the goal; most admissible heuristics are monotonic. (If the heuristic is not admissible and monotonic, then step 7b must put N' back on the open list if the new $f()$ value is lower than the old $f()$ value.)

You should use the straight line distance from the node to the goal as your heuristic. This heuristic is admissible because the actual robot path cannot be shorter than the straight line distance. Any heuristic (in particular straight line (or Euclidean) distance) that obeys the triangle inequality is monotonic.

A.6 Graphical display

Not only will a graphical display help me test whether your program is working properly, but it will also help you with debugging your code.

I have found two simple graphical libraries, one for Windows and one for X11/UNIX. You can feel free to use whatever graphical package you want (and in fact, you might know of better ones than those that I found). Your graphics library should have the capability of drawing lines and nonconvex filled polygons and be able to do some sort of simple animation (for future assignments). Interaction with the mouse would probably be nice but isn't necessary. Feel free to share code for this part of the assignment.

The two packages I've found are called VOGLE (for X11/UNIX) and TOOGL (for Windows). I'll have links or local copies of these packages on the course web page.

B. Written questions

1. Describe how you created the configuration space representation of the world boundary. Describe any other important design decisions you had to make.
2. Why do we tend to use the A* search algorithm for searching graphs in motion planning?
3. Characterize the paths that your motion planner produces. Are they the shortest paths from the start to the goal? Are they good paths? How could you improve these paths (without doing motion planning again)?
4. I was originally going to have you write a motion planner for a robot that could rotate. What would you have to change in your program to make it run under this assumption? Describe and sketch solutions to the main problems you would run into.
5. How would you need to change your motion planner if the obstacles were nonconvex? What if the world boundary were not rectangular (but still convex)? What if the world boundary were not convex? Describe what you'd have to do (and in a little detail how you'd do it) to adapt your motion planner for these assumptions.