

CIS 391 Homework #2, Part 2

October 1, 2007

(based on 3.15 from the book)

Imagine you are a robot, though a very unusual one: you're one-dimensional (a point) and you live in a two dimensional world. You are bounded in by a fence that is a square of side length ten centered on the origin. The world also contains many obstacles which are convex polygons. (see illustration on p. 92) Your job, given two points, is to get between them by the shortest path possible. You can go along the sides of the polygons, but you cannot go where two polygons meet (that is, if two polygons share a vertex, you may not go "through" that vertex to go between them).

1. Suppose the state space for this problem consists of all positions (x,y) in the plane. How many states are there? How many paths between any two points?
2. Explain briefly why the shortest path from one polygon vertex to any other in the scene must consist of straight line segments joining some of the vertices in the polygons. Define a good state space now. How large is this state space?
3. Describe, in general terms, how to determine the successors of a state.
4. Write a program to simulate the robot: that is, you will be given a set of obstacles and two points, and you must determine the path between them. Implement using three search algorithms: both breadth-first and depth-first search (to find *any* path) and A* search to find the *shortest* path. The path found should be output as a tuple of (x,y) pairs (themselves tuples).

`robotnav.py` contains some helper methods you may find useful. In particular you are **required** to use the `read` function there to read in input files. `pathIntersectsAnyLines` will also be useful - it takes a start point of a line segment, its ending point, and a list of line segments (each in the form $((0,0), (1,1))$) and determines if the line intersects any of the line segments in the list.

`guihelp.py` contains helper functions for drawing lines and obstacles which may assist you in you visualization. You don't have to use it if you don't want to. It requires Tkinter.

A few examples files are provided: `square_in_the_middle`, `triangle_in_the_middle`, `tall_triangle_in_the_middle`, and `big_triangles`.

To turn in: The answers to 1, 2, and 3 by e-mail by Friday, 5 October, and to part 4 as well-commented Python code which I may run (also by e-mail).