CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2018
Webpage: `http://www.cs.clemson.edu/~bcdean/` — TTh 12:30-1:45
Handout 19: Homework #5 — Earle 100

1 Geometric Shortest Paths

In this assignment we will find the shortest path between two designated points on the Clemson campus, avoiding polygonal obstacles (buildings), and avoiding any route that can be seen by TA Corey, whose location is also specified as part of the input.

Figure 1: Corey.

As usual, all of the material related to this assignment is found in the following directory:

`/group/course/cpsc212/f12/hw05/`

The main input file, `campus_places.txt`, describes the geometry of the buildings on campus. An excerpt from this file describing the Rhodes Engineering Center is:

```
Rhodes_Engineering_Center 6
   1260.976517 637.8549177 0
   1286.720936 634.6215876 0
   1276.498723 579.3660873 1
   1250.754305 582.5994174 0
   1253.63208 598.1771796 0
   1259.298741 628.7882601 0
```

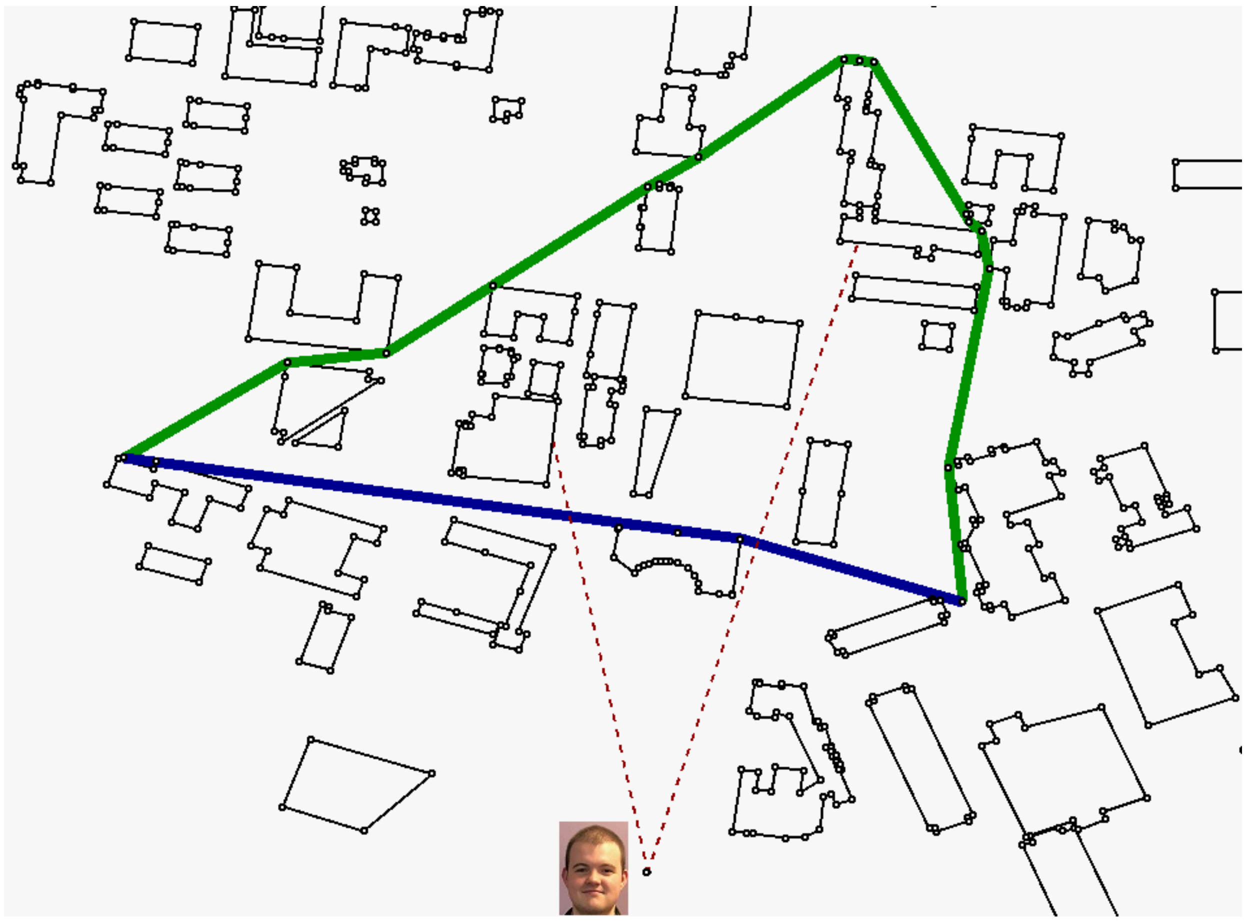

Figure 2: The shortest path from the starting point to the ending point is shown in blue, but this crosses several lines of sight from Corey (shown in red). The shortest path that avoids being seen by Corey is in green.

As you can see, each building is described by its name, which is a string with no whitespace, followed by the number of vertices in the polygonal shape of the building. These vertices are then listed in clockwise order around the building. Along with each vertex is a binary flag specifying if the vertex is not usable as a waypoint along a path. One of the vertices in Rhodes is not usable since it is along a wall shared with Rhodes Annex (the two buildings are directly adjacent, so it is impossible to pass between them). The (x, y) coordinates[1] of each vertex have been scaled so that one unit of distance corresponds to one meter.

The file `input.txt` contains three (x, y) coordinates – those of the starting point (between Barre Hall and McAdams Hall), the destination point (just outside Earle Hall), and Corey's location. Using this data, the optimal route across campus is shown in Figure 2.

[1]Note that y coordiantes increase as you move north, as is customary for points in the 2D plane. This may cause the image of campus to appear vertically flipped if you draw it to the screen, since screen coordinates usually have the origin in the upper-left corner.

2 Goal #1: Point-In-Polygon Testing

Neither the start point, end point, or Corey should be inside any building. If any of these three points is inside a building, your program should print an appropriate error message to this effect and exit, without trying to compute a shortest path.

In order to achieve this goal, you will need to write code that tests whether a point is in a non-convex polygon, running it on each building in the scene for each of the start point, end point, and Corey. Recall from our discussion in class how we test if a point is inside a non-convex polygon: we cast a long ray outward from the point in a random direction, and count the number of intersections with the polygon. Each intersection toggles between being inside versus outside, so if this number is odd, the point is inside the polygon. This test will be easy to write once you have written code to test whether two segments intersect, which we also discussed in class (remember this uses tricks based on cross products).

For this part and any other part of your homework solution, feel welcome to use any of the geometry code provided from our geometry lecture. This includes structures for storing points (Vec2D), segments (Segment2D), and rays (Ray2D), and a function that computes the intersection point between a ray and a segment (which could potentially be used for the segment-segment intersection testing above).

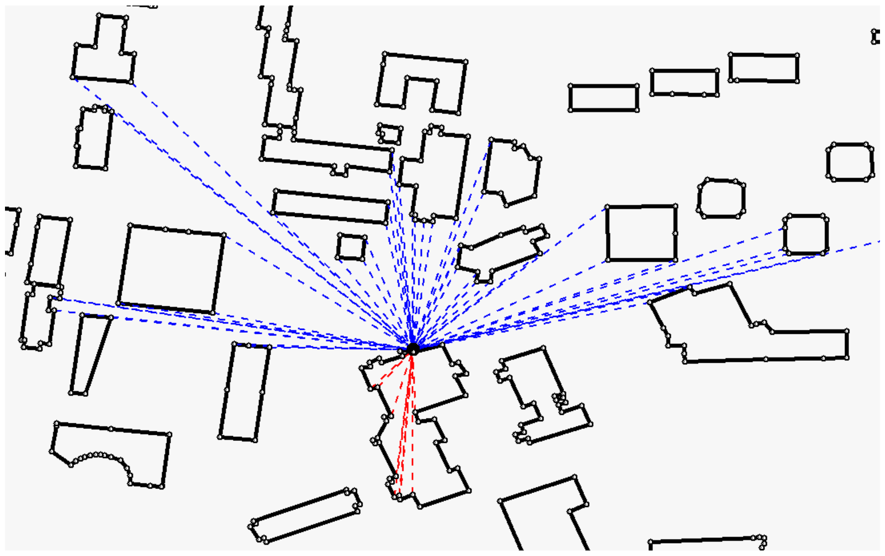

Figure 3: Blue lines indicate edges in the visibility graph from a single vertex on the McAdams Hall polygon. Red lines are not blocked by visibility constraints, but they aren't part of the visibility graph since they are interior to the polygon.

3 Goal #2: Build the Visibility Graph

To compute shortest paths through campus, you will first build a graph, then run Dijkstra's algorithm. The key observation here is that any shortest path in the 2D plane with polygonal obstacles will move from vertex to vertex as intermediate points. That is, we only need to consider vertices of our polygonal obstacles as potential waypoints, stepping between these with straight segments. We therefore build what is called a *visibility graph*. As shown in Figure 3, the nodes are all the vertices in our scene, along with also the start and end points. An edge connects any two vertices p and q that can see each-other, with no walls intersecting the segment from p to q.

Your code that you have already written for segment-segment intersection testing should make it relatively easy to compute the visibility graph. One challenge, however, will be omitting edges (shown in red in the figure) that are between two vertices visible from each-other but that are interior to a polygon. Since our path is not allowed to go through buildings, we cannot include these edges in our graph. Feel welcome to try any strategy you like to get rid of these edges. Here is one suggestion: to test if the edge from p to q goes through the interior of a polygon, move one meter from p in the direction of q and then test if the resulting point is inside a building, using your solution from the previous goal.

Note that the process of building the visibility graph can be a bit computationally heavy. We are testing every pair of vertices against every segment in the scene, so if our input contains n vertices, the time complexity is $\Theta(n^3)$. To allow your program to run quickly, you may want to compile with full optimization turned on, using the `-O3` flag with `g++`.

Once you have built the visibility graph, you should be able to calculate the shortest path distance from the start point to the end point. This doesn't take Corey into consideration yet, but if you print out the length of this path as output, you will receive substantial partial credit for this assignment.

4 Visualizing Things

As you develop and debug your code, it can be helpful to be able to draw pictures of the 2D layout of campus along with the path you are computing. Feel welcome to use the OpenGL-based rendering code from our geometry lecture to do this. However, since this code requires you to use one of the lab machines (and is not easy to use over a remote connection), an alternative library has been provided to help with visualization. The file `vis.cpp` in the homework directory contains this code, which currently just draws the campus map, but you should feel welcome to extend it to draw anything else you like, such as edges in your visibility graph, or the shortest paths you are computing.

The code in `vis.cpp` outputs a text file that can be read by `xfig`, a rudimentary figure-drawing program available on most Unix systems that is often used for technical drawings. This output can be converted to encapsulated postscript with the `fig2dev` command, and then to a PDF file, as follows, assuming `a.out` is your executable that produces the `xfig` output:

```
./a.out | fig2dev -L eps > myfile.eps ; epstopdf myfile.eps
```

You don't need to do any visualization as part of this assignment, but it can certainly be helpful in terms of debugging your code and ensuring your output is correct.

5 Goal #3: Avoiding Corey

For full credit on this assignment, your final challenge involves finding the length of a shortest path that avoids being seen by Corey (or determines that no such path exists). At a high level, this is straightforward: we simply don't add any edge to the visibility graph if it cross a line of sight from Corey. More precisely, we first compute a sufficient set of "line of sight" (LOS) segments from Corey (shown in Figure 4), and test against these whenever we consider adding an edge to the visibility graph.

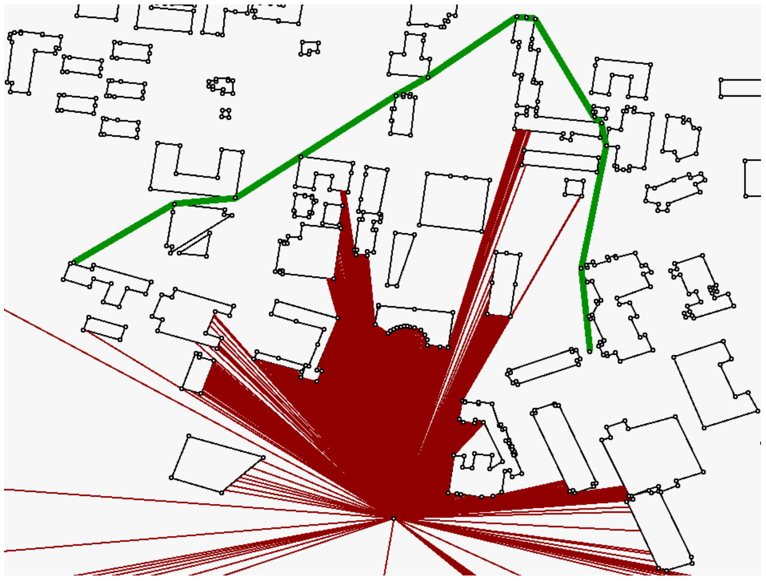

Figure 4: Line-of-sight segments indicating visibility from Corey are shown in red.

To compute all the necessary LOS segments from Corey, we use the approach discussed in the first lecture after the Thanksgiving break: first draw a ray outward from Corey through every vertex in the scene. We sort these rays by their angles from Corey (which we compute using the `atan2` function). We then cast a ray from Corey between each consecutive pair of angles, stopping at the first segment we hit. These give us the LOS segments.

The final output of your program should be the length of a shortest path avoiding Corey.

6 Submission and Grading

To submit your final results, be sure to submit all of the files necessary to run your code (including the `geom.h` and `geom.cpp` files, if used). You may also include a Makefile if compiling your code is not straightforward.

Your assignment will be graded based on correctness, and also on the clarity and organization of your code. Final submissions are due by 11:59pm on the evening of Friday, December 7. No late submissions will be accepted.