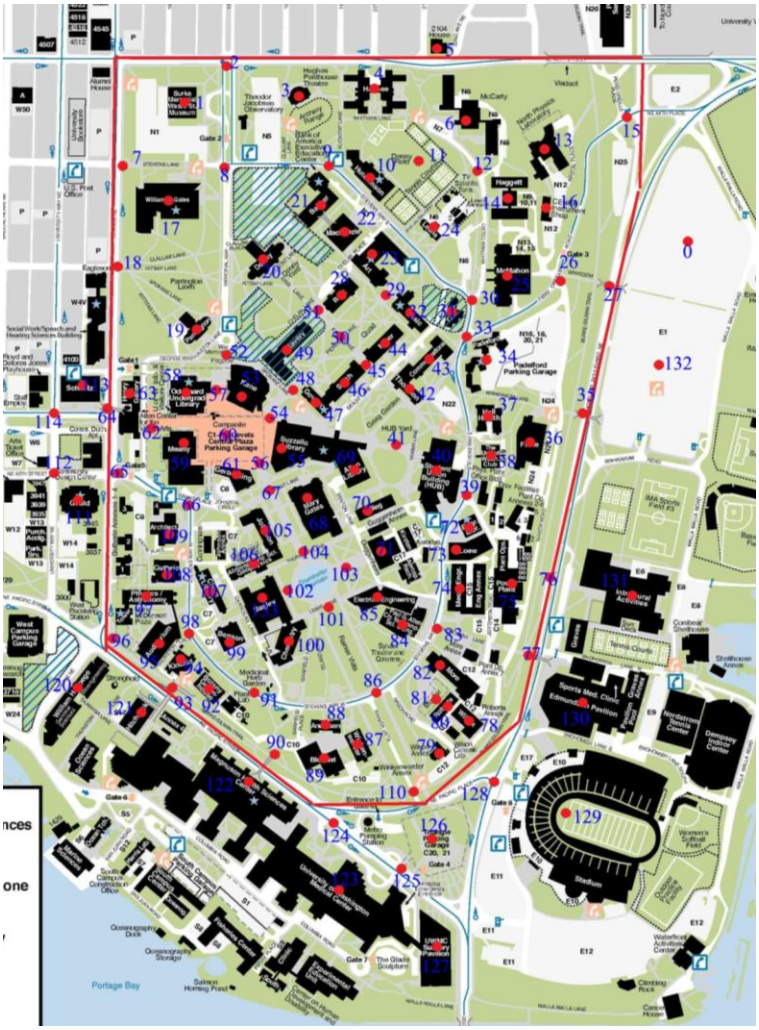
UW Maps - Shortest Path

By: Steven Berry, David Luu-Van, David Ou

The purpose of our project was to create a program that found the shortest walkable path between two buildings on the UW campus. Our first step was to take a map of the University of Washington campus and manually choose points on buildings and vertices of walkways. In total, 134 points were selected. After choosing the points, we assigned each a number that which the program could use to determine the shortest path. Initially the points ran from 1 to 134, but were retroactively zero-indexed to make list calculations in Python more intuitive and less prone to “off by one” errors.



The red line is the boundary for what we considered the "main" campus, and shows the original scope of our project. We decided to include some points off campus, to demonstrate that the algorithm will only select paths that cross into campus in a feasible way (for example, over a footbridge instead of up a wall).

We then created a very simple python module (list_creator) to create a formatted text file with the numbers 0 through 133 (one per line), each followed by a tab. This standardization made it much easier for the data to be later read in by more complicated modules; since all lines contain their own (zero indexed) line number and the entries in each line are tab delineated.

The first use of this template was to create the file "adjacencies.txt" to store information about which vertices are adjacent to one another. For this project we use the term adjacent to mean that you can walk directly from one point to another (and for this specific implementation, vice versa). The text file was formatted according to the aforementioned specifications. The entries in each line (after the first) are the number codes of the vertices that the first entry is adjacent to.

To interpret this text file, the module "adjacency_filler" was created. This module reads in a text file (the passed parameter to the function "fill_adjacencies") and returns a list with 134 entries, each of which is a list with 134 entries. This format will be referred to as a matrix, where the entries in the main list are the rows, and the entries in the sub-lists are referred to as columns. The entries in each of the 17956 cells were either a 0 or 1 (though the Boolean entries True or False would have worked just as well) where a 1 means that the row (vertex number) is unable to walk directly to the column (again, corresponding to a vertex number). A design choice was made to force adjacencies to be non-directed, meaning that if vertex A is adjacent to vertex B, then vertex B is adjacent to vertex A. This was implemented by modifying both the matrix[row][col] and matrix[col][row] as we iterate through the row line of the text file. Since there are no one-way paths on the UW campus, this protects against user-error when creating "adjacencies.txt" if the reverse direction is mistakenly omitted.

The next step was to manually recorded the GPS coordinates for each point into "coordinates.txt" for later use. This information was then read in by "distance_filler" which returns a 134x2 matrix. The

268 entries contain the GPS coordinates (latitude and longitude) for the corresponding row index (vertex number). The motivation behind separating the data from the code was to keep the program as modular as possible, both for debugging and modification as well as future applications and expansions. It would be a relatively simple task to point the modules towards another set of text files to create the necessary adjacency and coordinate matrices. This proved immensely helpful when testing out a variety of paths, as it meant that we could simply edit the file "adjacencies.txt" and then re-run the program if we found that there was a mistakenly added or omitted adjacency. This allowed us to fix paths that were "optimal" in the shortest distance sense, but required the walker to proceed through buildings that do not actually have exits on the appropriate sides.

The module "distance_filler" combined the adjacency information and GPS coordinate information to create a 134x134 matrix with the distance between adjacent vertices in each of the 17956 cells. The distance formula

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

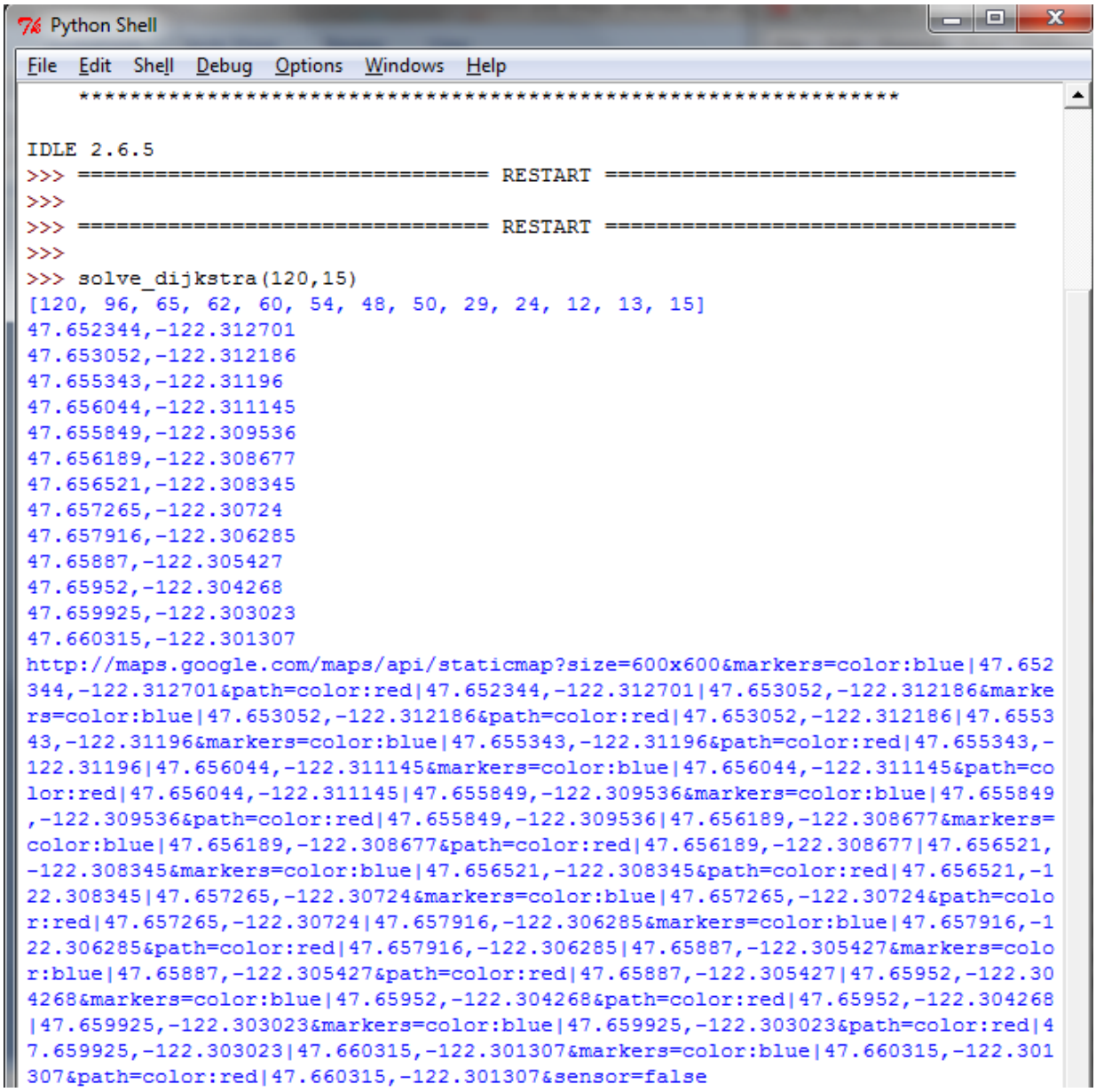
was used because the adjacencies were chosen in such a way that adjacent vertices have a straight path between them. For vertices that were not adjacent, the value float("infinity") was selected to reduce the amount of work that the dijkstra_solver module would need to do.

The main algorithm for our shortest path project was in the dijkstra_solver module. This module uses Dijkstra's method for finding the shortest path between two points. It was selected because it is fairly easy to implement. One drawback to Dijkstra's method is that it does not support negative distances. The Bellman-Ford algorithm does allows for negative distances (not negative distance cycles, though), at the expense of a longer running time. Since we do not need to worry about negative distances, Dijkstra's was selected.

Dijkstra's works by storing extra information about each vertex, notably "found," "distance" and "previous." Found means that the shortest path from the starting vertex to that vertex has been found, distance is the total distance travelled along that path, and previous means the vertex immediately before

it on that path. Initially, the starting vertex has distance of zero, all other vertices have distance of infinity. In each iteration, you select the nearest vertex that has not already had a shortest path found, and add change the "found" value to true. The next step is to check all neighbors of that vertex and compare their current distance to the sum of (the newly added vertex's distance, and the distance between that vertex and the adjacent one). The smaller value is kept, and "previous" is updated (or left alone) accordingly. Dijkstra's ends when either all vertices have been added (all have "found" as true) or the distance to the next nearest vertex is "infinity," meaning that it (and all other not "found" vertices) cannot be reached from the starting vertex.

The output is a list, in the order of the vertices (by their number code) along the shortest path from the starting vertex to the ending vertex. Since the number codes are relatively unhelpful for anyone that does not have the labeled map, we created the file "html_generator" to give more user-friendly output. Dijkstra's sends the output list to "html_generator" after printing it (for those who do have the labeled map). html_generator then converts the vertex number codes back into GPS coordinates by calling the coordinate_filler module. The html_generator module prints the latitude and longitude, in order and on separate lines, and then a very large URL. The URL is formatted according to the specifications to display an image from Google Static Maps. The URL contains the ordered coordinates of each point along the path, as well as the formatting information for the markers and path. The URL can be pasted into any web browser, giving a very user-friendly output, as seen below for the example from vertex 120 to vertex 15. Unfortunately, there is no way to attach names (or even numers) to the markers displayed on the map, because they are limited to a single character by the Google Static Maps specifications.


Python Shell
File Edit Shell Debug Options Windows Help
****
IDLE 2.6.5
>>> ========== RESTART ==========
>>>
>>> ========== RESTART ==========
>>>
>>> solve_dijkstra(120,15)
[120, 96, 65, 62, 60, 54, 48, 50, 29, 24, 12, 13, 15]
47.652344,-122.312701
47.653052,-122.312186
47.655343,-122.31196
47.656044,-122.311145
47.655849,-122.309536
47.656189,-122.308677
47.656521,-122.308345
47.657265,-122.30724
47.657916,-122.306285
47.65887,-122.305427
47.65952,-122.304268
47.659925,-122.303023
47.660315,-122.301307
http://maps.google.com/maps/api/staticmap?size=600x600&markers=color:blue|47.652344,-122.312701&path=color:red|47.652344,-122.312701|47.653052,-122.312186&markers=color:blue|47.653052,-122.312186&path=color:red|47.653052,-122.312186|47.655343,-122.31196&markers=color:blue|47.655343,-122.31196&path=color:red|47.655343,-122.31196|47.656044,-122.311145&markers=color:blue|47.656044,-122.311145&path=color:red|47.656044,-122.311145|47.655849,-122.309536&markers=color:blue|47.655849,-122.309536&path=color:red|47.655849,-122.309536|47.656189,-122.308677&markers=color:blue|47.656189,-122.308677&path=color:red|47.656189,-122.308677|47.656521,-122.308345&markers=color:blue|47.656521,-122.308345&path=color:red|47.656521,-122.308345|47.657265,-122.30724&markers=color:blue|47.657265,-122.30724&path=color:red|47.657265,-122.30724|47.657916,-122.306285&markers=color:blue|47.657916,-122.306285&path=color:red|47.657916,-122.306285|47.65887,-122.305427&markers=color:blue|47.65887,-122.305427&path=color:red|47.65887,-122.305427|47.65952,-122.304268&markers=color:blue|47.65952,-122.304268&path=color:red|47.65952,-122.304268|47.659925,-122.303023&markers=color:blue|47.659925,-122.303023&path=color:red|47.659925,-122.303023|47.660315,-122.301307&markers=color:blue|47.660315,-122.301307&path=color:red|47.660315,-122.301307&sensor=false

One possible extension for this project (that we have been working on) is to create a web page that can be accessed by smartphone, that displays the standard Google Map interactive object (with the path) rather than a static image. To do this we have been primarily using JavaScript, which reads in the block text of ordered coordinates that Dijkstra's prints out when it runs, and then contacts the Google Maps servers to display the object. One downside (that could be fixed by another possible extension) is that this still requires Python to pre-process everything, so you would have to set up the web page from your computer before you could read the map on your smartphone. To solve this we would need to port

the Python code to a browser-supported language (or phone app). This still leaves the problem of inputting vertices as numbers rather than names, though a good naming scheme has proven difficult to work out, since there are several vertices along the same road and distinguishing between them results in names that are very difficult to remember. If we were to develop this program for use in other environments, it would likely be very worthwhile to streamline the input process by allowing the user to simply click a map to add a point (and GPS information). This still leaves the tedious work of establishing adjacencies, for which there seems to be no good automation.