Getting Around: Optimizing Paths Taken Across Campus With Respect to Distance and Weather

21-393 Final Project, Fall 2014

Haozhe Gao
David Wu
Benjamin Zhang

Contents

1 Introduction

1.1 Problem Description

Consider the shortest path between two locations. If the world was just a plane, and you can travel uninhibited through walls and empty space, the shortest path between two points would be obviously a line, from which you cannot get more optimal. However, in practice, the problem of deducing the shortest path between two points usually involves taking a roundabout path, since there is rarely a direct line between where you are and where you would like to go. Thus, to find this shortest path, we have to do more than connect two dots in a three dimensional plane. Furthermore, the above model does not account for other factors besides distance traveled. To this end, and to be able to express a realistic shortest-path model, we turn to graphs. In the formal sense, a graph $G = (V, E)$, where V is the set of vertices, and $E \subseteq V \times V$, as well as a weight function $w : E \to \mathbb{R}^+$. For Carnegie Mellons campus, what are the optimal paths to take between buildings to get to classes to minimize time spent travelling, as well as avoiding unpleasant paths, such as those that demands climbing a lot of stairs or spend a lot of time outside? To tackle this problem, we utilized a graph representation of places on campus, and leveraged algorithms that solved the shortest-path family of problems in a weighted undirected graphs.

1.2 Methodology

The first step in using these graph algorithm was to construct a graph, and to do that, we collected data about actual walking times on campus. We decided to use main campus locations as the nodes of our graph, and their respective possible paths as the edges of our graph. To proxy for actual distance, we used time traveled to measure the distance between locations. To collect data for our graph, we walked around the various well traversed campus locations that we selected, recording our times of travel. In doing so we made a number of assumptions about the data, such as assuming that we maintained a consistent pace. We made repeated routes, and took the average of the travel times, in order to keep our sample from being biased by a fast walker or a slow stair climber.

The edge data was then plugged into a computer, with a custom python script [1] that utilized a 3rd party library to produce a visualization of the results.[2].

1.3 Assumptions

There are many assumptions that we make in the performance and analysis of our project. First, we assume that travel time between two points is constant, regardless of which one is the source and which one is the destination. For the majority of segments of travel, this is approximately true, but for some places, like stairs, it is not. We have made this assumption in order to cut down on the number of nodes on the graph; Instead of representing the entrance and exit to the stairwells as nodes on the graph, this abstraction trades complexity in the nodes for a slight decrease in accuracy.

Secondly, we also assume that congestion is never an issue. In wide open spaces, such as the path from the UC to the Randy Pausch Bridge, this is not a problem, and a good assumption to make; in narrow spaces, such as most staircases, or in places where multiple people share the same resource, such as the elevators, this assumption starts to deviate from the true result. This can be accounted for in a variety to ways. We can take measurements over the course of a day, in order to figure out the expected time that an edge will take,

[1] see Appendix-Code

[2] Installation direction, API, and sample code can be found at https://code.google.com/p/python-graph/

or we can adjust the model dynamically throughout the day, in accordance to the average travel duration for that specific temporal segment. This issue also is apparent, upon further directions outlined in the Further Development section. Steps to alleviate this discrepancy, however, is included in the details of each subsection under Further Development.

2 Data and Analysis

2.1 Results

We collected raw results [3], and then used the a variant of code [4] to generate the visual in [5]. Due to the graphing visualization library used, the length of the edges is, in no way, associated with the cartesian distance between two points. Rather, the time it took to walk between two points, as well as if the segment was inside or outside, are displayed on the labels for each edge. One can see, upon inspection of the graph, that the does not exist any nodes that connects two points far apart; for instance, travel time between Hamerschlag and Doherty is split up into two parts, one from Hamerschlag to Wean, and then from Wean to Doherty. This is due to the fact that there are some source and destination pairs in campus, such that all paths has to go through some other location of interest. For example, travel between Baker Hall and Posner Hall's Hunt exit, will naturally pass by Hunt Library. Thus, these routes are represented not by a single edge in the map, but a path. Calculations can then be done on the graph, using Dijkstra's shortest path algorithm [6], in order to find the shortest path between a pair of nodes on the graph.

2.2 Single Source Shortest Path

We also were interested in building a shortest-path tree (SPT), rooted at a given node. This an useful representation of information, since the tree can represent efficiently the shortest path from the root node to any node - just follow the path in the SPT from the root to the node you want, and then nodes in the path will be nodes that you pass on the best path. In this situation, we can actually improve the runtime of our algorithm by switching to another shortest-path algorithm with a different idea, called the Bellman-Ford algorithm [7]. This algorithm is based on a Dynamic-Programming-esque relaxation principle, unlike Dijkstra's. It runs in $O(|V||E|)$, as compared to Dijsktra's $O(|V|(|E| + |V| \log |V|))$, which is especially great for sparse graphs, such as the one we have in our campus graph, where $|E|inO(|V|)$. In dense graphs, where $|E| \in O(|V|^2)$, both algorithms run in $O(|V||E|)$. The results of such a computation is found in the appendix [8]

2.3 Different Edge Weights

The last analysis we did with the data was to change up the weight function associated with each edge. As mentioned before, the weight function can be retooled to take into account various characteristics of the path. In our methodology, we took a basic view on this, and used a simple weight function to demonstrate how the optimal routes can change with the new factors. With $w : E \to \mathbb{R}^+$ as the old function, we used the function

$$w'(e) = \begin{cases} w(e) & : \quad \text{e is inside} \\ 2w(e) & : \quad \text{e is outside} \end{cases}$$

[3] See Appendix-Table 1
[4] See Appendix-Code
[5] See Appendix-Graph 1
[6] http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[7] http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
[8] see Appendix-Table 2 and Appendix-Graph 2

We then redid the above calculation for the SPT using these new weights, which yielded different results. [9]

3 Further Development

3.1 Web/Phone app

This kind of shortest-path exploration could be useful as an informative mobile application. This approach gives rise to interesting challenges and abilities that a research-then-release methodology does not have. First and foremost, the question of how the application responds to multiple users, in the hundreds or thousands, all querying it at the same time, would respond. Since the number of students on campus does not really match the throughput seen by even one server by a popular web application, and assuming that people not affiliated with the university does not use the application, we should be safe in this regard.

Another possible challenge is the notion that if our application directs everyone the same way, we can be creating traffic jams in places that shouldn't have any. This can slow down travel time, which could make other routes more desirable. To combat this, our application can incorporate a randomized factor in deciding how to position the multiple paths, in order to maximize each user's utility, or a social utility. In this aspect, we can leverage the idea of a correlated equilibrium, which is something that we can calculate in poly-time with an LP formulation, in order to ensure that everyone's travel time is minimized, and that no one would have anything to gain by walking along another path.

However, one exciting opportunity comes with the location tracking feature in most cellphones. By checking a user's location periodically, the application can get a sense of how fast the user is moving down a hallway, and with a multitude of data from the same hallway, get a sense of how crowded it is. It can then dynamically reweigh the graph, so that further calculations takes this delay into account. This way, as long as the first couple of people has already been slowed down by a sudden change in travel time along a path (for instance if a forklift is coming out of a hallway in wean), the application can dynamically adjust to this detour, by getting a sense of how much it is slowing down walking times through the congested path.

3.2 Integration with Google Maps

There are still other applications that we can look towards in studying optimal paths around campus. Google Maps is one of the most widely used mapping services on the web, and it is constantly monitored for updates and improvements. This program has many features; it has the capability to provide satellite and street views, and also include a route planner that tells you the available paths to your destination based on your method of transportation. It also is a great locator for businesses and organizations, especially in decently sized cities. Google Maps is also on the mobile store. We can use the tools Google Maps provides us to give users a more dynamic experience.

One of the nice features of Google Maps is real time position tracking. In other words, the program can calculate the path to a destination based on where you currently are. So, as you move towards or away from your destination, the optimal path is constantly updated. So this kind of feature is very useful for moving around campus, especially if you have forgotten the times between buildings or just find yourself in an unfamiliar part of campus. In the case that there are many users on the app at once, we can provide the real time optimal path which is also based on how crowded the paths between buildings are at the current time.

[9] see Appendix-Table 3, Appendix-Graph 3

3.3 Desired stops

To put a little more variety into the problem of optimal paths, we can add certain desirable stops along the route to the destination. For example, if a student has a good amount of time between classes, perhaps she/he can stop for a cup of coffee or grab some food beforehand. So, this adds a new perspective on the problem and new variables. Different students have different budgets or tastes, and so their stops will all be different. One of the advantages that CMU has over other schools that makes this problem more interesting is that there is no general dining hall; instead, there are many small locations for food or drink scattered across campus. Furthermore, at different times of the day these locations can have large lines. For example, there is always a long line at the food places in Gates Hillman Center and Wean Hall around noon to early afternoon.

4 Conclusion

Dynamic programming algorithms are a very good way in finding the optimal paths across campus. For our needs we can use Dijkstras algorithm and Bellman-Ford's algorithm after collecting simple data such as times and distances between buildings. To make our problem more interesting we took into account how much being outside during the route effects optimality. Since every person has different needs, the amount of stress placed on being outside or not can be represented using weights on the paths in our graph. Lastly, there are plenty of directions to further our project in, such as the development of a standalone travel app, integration with Google Maps, or adding mandatory stop, that can bring additional functionality, as well as challenges, to our project.

5 Appendix

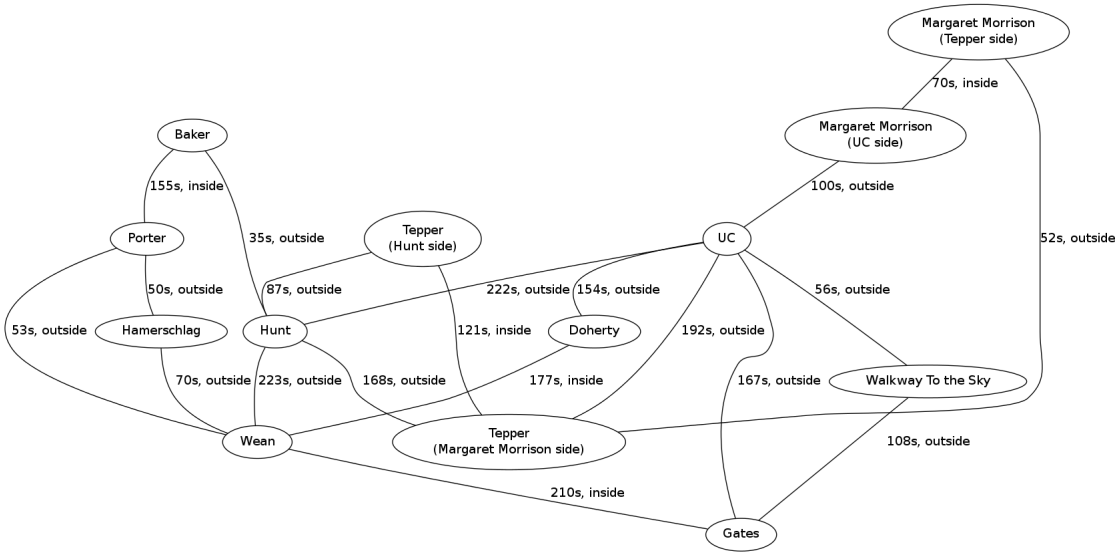
5.1 Code

```
import sys
sys.path.append('..')
sys.path.append('/usr/lib/graphviz/python/')
sys.path.append('/usr/lib64/graphviz/python/')

from pygraph.classes.graph import graph
from pygraph.classes.digraph import digraph
from pygraph.algorithms.searching import breadth_first_search
from pygraph.readwrite.dot import write
from pygraph.algorithms.minmax import shortest_path_bellman_ford

# Graph creation
gr = graph()

OUTSIDE = "outside"
INSIDE = "inside"

# Add nodes and edges

def addEdge(gr, v1,v2, minutes,seconds,label):
    gr.add_edge((v1,v2),
                wt=minutes*60+seconds,
                label ="%ss, %s"%(minutes*60+seconds, label))
```

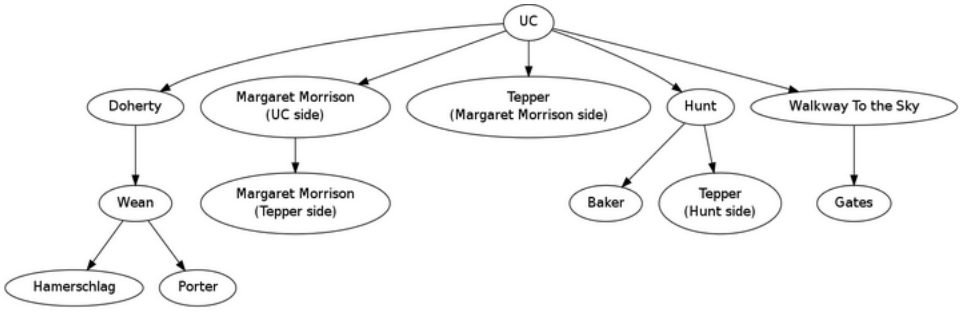
```
edges= [("UC","Hunt",3,42,OUTSIDE)
        ,("Hunt","Wean",3,43, OUTSIDE)
        ,("Wean","Doherty",2,57, INSIDE)
        ,("UC","Doherty",2,34, OUTSIDE)
        ,("UC", "Gates",2,47, OUTSIDE)
        ,("Gates", "Wean",6-3,17+60-47, INSIDE)
        ,("Wean","Hamerschlag", 1,10, OUTSIDE)
        ,("Hamerschlag","Porter", 0,50,OUTSIDE)
        ,("Porter","Baker",2,35,INSIDE)
        ,("Baker","Hunt", 0,35,OUTSIDE)
        ,("Tepper\n(Margaret Morrison side)","Hunt", 2,48,OUTSIDE)
        ,("Tepper\n(Margaret Morrison side)","UC", 3,12,OUTSIDE)
        ,("Tepper\n(Margaret Morrison side)","Margaret Morrison\n(Tepper side)", 0,52,OUTSIDE)
        ,("Margaret Morrison\n(Tepper side)","Margaret Morrison\n(UC side)", 1,10,INSIDE)
        ,("Margaret Morrison\n(UC side)","UC",1,40,OUTSIDE)
        ,("Tepper\n(Hunt side)", "Tepper\n(Margaret Morrison side)",2,01,INSIDE)
,("Tepper\n(Hunt side)","Hunt",1,27,OUTSIDE)
        ,("Porter","Wean",0,53,OUTSIDE)
        ,("Gates","Walkway To the Sky",1,97-49,OUTSIDE)
        ,("UC","Walkway To the Sky",0,56,OUTSIDE)]

def constructRegularGraph():
    gr = graph()
    gr.add_nodes(["UC", "Hunt", "Wean", "Doherty", "Gates",
                  "Hamerschlag", "Porter","Baker",
                  "Margaret Morrison\n(Tepper side)",
                  "Margaret Morrison\n(UC side)",
                  "Tepper\n(Margaret Morrison side)","Tepper\n(Hunt side)",
                  "Walkway To the Sky"])

    for edge in edges:
        addEdge(gr,*edge)
    return gr

def constructInsideGraph():
    gr = graph()
    gr.add_nodes(["UC", "Hunt", "Wean", "Doherty", "Gates",
                  "Hamerschlag", "Porter","Baker",
                  "Margaret Morrison\n(Tepper side)",
                  "Margaret Morrison\n(UC side)",
                  "Tepper\n(Margaret Morrison side)","Tepper\n(Hunt side)",
                  "Walkway To the Sky"])

    for edge in edges:
        v1,v2,minutes,seconds,inOrOut= edge
        if (inOrOut ==OUTSIDE):
            addEdge(gr,v1,v2,minutes*2,seconds*2,inOrOut)
        else:
            addEdge(gr,v1,v2,minutes,seconds,inOrOut)
    return gr


tree, counts = shortest_path_bellman_ford(constructRegularGraph(),"UC")
```

```
def printEdges(edges):
    for edge in edges:
        v1,v2,minutes,seconds,inOrOut= edge
        print " %s &  %s & %d:%02d & %s\\\\"%(v1.replace("\n",""),v2.replace("\n",""), minutes, s

def printAll(counts):
    for dest in counts:
        print "%s & %ds \\\\\\\hline"%(dest.replace("\n",""), counts[dest])
printEdges(edges)

g = constructRegularGraph()
dot = write(g)

f = open('or.dot', 'a')
f.write(dot)
f.close()
```

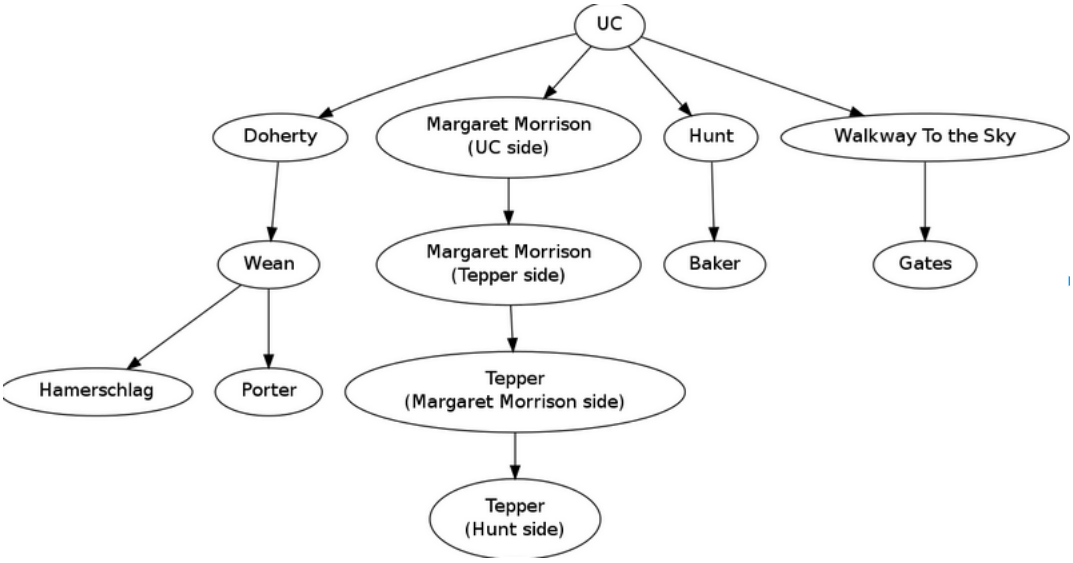
5.2 Graph 1



5.3 Graph 2



5.4 Graph 3



5.5 Table 1

Locale	Locale	Time	Inside/outside
UC	Hunt	3:42	outside
Hunt	Wean	3:43	outside
Wean	Doherty	2:57	inside
UC	Doherty	2:34	outside
UC	Gates	2:47	outside
Gates	Wean	3:30	inside
Wean	Hamerschlag	1:10	outside
Hamerschlag	Porter	0:50	outside
Porter	Baker	2:35	inside
Baker	Hunt	0:35	outside
Tepper(Margaret Morrison side)	Hunt	2:48	outside
Tepper(Margaret Morrison side)	UC	3:12	outside
Tepper(Margaret Morrison side)	Margaret Morrison(Tepper side)	0:52	outside
Margaret Morrison(Tepper side)	Margaret Morrison(UC side)	1:10	inside
Margaret Morrison(UC side)	UC	1:40	outside
Tepper(Hunt side)	Tepper(Margaret Morrison side)	2:01	inside
Tepper(Hunt side)	Hunt	1:27	outside
Porter	Wean	0:53	outside
Gates	Walkway To the Sky	1:48	outside
UC	Walkway To the Sky	0:56	outside

5.6 Table 2

Unweighted times	
Gates	164s
Doherty	154s
Margaret Morrison(UC side)	100s
Tepper(Margaret Morrison side)	192s
Wean	331s
Baker	257s
Hunt	222s
Tepper(Hunt side)	309s
Margaret Morrison(Tepper side)	170s
Porter	384s
Walkway To the Sky	56s
Hamerschlag	401s
UC	0s

5.7 Table 3

Weighed times

Destination	time (s)
Gates	328s
Doherty	308s
Margaret Morrison(UC side)	200s
Tepper(Margaret Morrison side)	374s
Wean	485s
Baker	514s
Hunt	444s
Tepper(Hunt side)	495s
Margaret Morrison(Tepper side)	270s
Porter	591s
Walkway To the Sky	112s
Hamerschlag	625s
UC	0s