Campus Pilot Proposal

Prepared for

Dr. Morse
Dr. Barrett
Professors
Brigham Young University

Prepared by

Brennen Barney
Eric Sorensen
Matt Crowder

March 17, 2017

Abstract
In this paper we will discuss the problem that Campus Pilot attempts to solve, the motivations behind the project, and the algorithms and technologies that we plan to employ. We will also discuss the technologies currently available to solve this problem and how ours plans to improve upon these technologies.

1. Problem

Currently, there are many apps and technologies available that allow people to get walking directions from point A to point B. However, all of these services fall short in two major aspects. First, they do not usually find the actual shortest path from point A to point B. They tend to come up with the shortest path only using sidewalks, roads, and well-known paths. However, there are almost always shorter paths when one takes into account walking through parking lots, grassy areas, and buildings. Second, these services do not usually allow users to customize their route based on personal preference. For example, some people might not want to walk up stairs or steep inclines. We plan to solve these two primary problems. Campus Pilot attempts to calculate the true shortest path from point A to point B and will allow users to adjust settings to find the path that best suits their personal needs.

2. Significance

We hope that this project will greatly impact the lives of busy students and people everywhere. It will allow people to save time and get to their destinations as quickly as possible. More importantly, we believe that Campus Pilot will greatly benefit people with handicaps and disabilities. For example, many people have disabilities that force them to avoid stairs, steep elevations, and large distances. Campus Pilot will allow these people to better navigate their surroundings.

3. Technologies currently available

As stated in Section 1, there are currently many technologies that calculate walking directions from source to destination. Here is a list of a few of the current technologies available: Google Maps, Apple Maps, MapQuest, CoPilot, Waze, and inRoute. However, as stated before, none of these truly find the shortest distance between point A and point B. They fail to take into account buildings, parking lots, and grass. They also fail to allow users to properly customize their route according to specific needs.

4. Algorithms and Technologies

This section details the algorithms and technologies we plan on using to create this web application.

4.1 Forming the Graph

We are using the Google Maps API to download images of the area that the user wishes to traverse. We then place nodes (an object with a latitude and longitude coordinate) every three pixels on the image. This constitutes our initial graph. We will create various cost matrices for each of the different user preference categories (distance, elevation, grass traversal, building traversal, and stairs). The cost matrices will have the cost of going from every point in the graph to every other point in the graph. It is important to note that grass traversal, building traversal are binary matrices. For example, if the path from node A to node B goes through a building then it

has a value of "true" in the matrix. If it does not go through a building then it has a value of "false". This system effectively creates different channels that we can turn on and off depending on user preference. The most basic cost matrix is the distance matrix. It is the cost with all channels turned off. As users turn channels on, we begin to sum the matrices to get the total cost. All of this will be done when a campus area is designated, and stored in our database, to be recalled when a user requests a route. As a result, users should have a fluid and quick experience.

4.1.1 Connected Components Algorithm

In order to actually calculate the cost between two nodes, we loop over the pixels in between the the nodes and check for the RGB values that we can associate with certain objects. For example a green pixel indicates a field. Therefore, if we find a green pixel in our loop we set that space to true in the cost matrix. In order to have this work properly we need to fill in the objects on a map with a single color. For example, we decided to fill all the buildings with red. The Google API allows you to customize colors but it does not work for everything. It only colored about 80% of buildings on BYU's campus. Therefore, we had to use connected components to color the objects that Google missed. Below is pseudocode for the basic connected components algorithm.

4.1.2 Pseudocode

```
function ccFill(image, fillcolor, sw, ne, points):
   for each p in points:
       let visited = 2-D array (init to 0)
       let rgb = color at p
       while rgb is background color:
           increment p (x and y)
           let rgb = color at p
       let seed = mode filter at p
       let next = stack
       let dist = 5 //arbitrary
       next.push(p)
       while stack is not empty:
         let curr = next.pop()
         visited[curr] = true
         for each cardinal direction:
                let neighbor = pixel adjacent to curr
            check dist # of pixels in direction
            if for any, color is not background:
                if not visited[neighbor]
                          set neighbor's color = fillcolor
                  next.push(neighbor)
```

4.2 Dijkstra's Algorithm

When we receive the user's start and end points, we will take our to total cost matrix and run Dijkstra's algorithm on it. This will produce the shortest path according to our cost matrix. Below is pseudocode for our implementation of Dijkstra's algorithm.

4.2.1 Pseudocode

```
function dijkstra (source, graph):
    dist  = [] //holds new distances
    prev = [] //holds pointers back
    for each vertex in graph: //initialize dist and prev
      dist[v] = inf
      prev[v] = null
    dist[source] = 0 //dist from source to source is 0
    prev[source] = -1 //lets us know we have reached source
    Q = list of nodes in graph //queue of nodes in graph
    while Q is not empty:
      u = Q.deletemin() //pops the node with lowest dist
      for all nodes n in graph: //loops over its neighbors
          if dist[n] > dist[u] + dist u to n: //check new dist
                  newDist = dist[u] + dist u to n
```

```
dist[n] = newDist //set new dist
prev[n] = u //set the path for that dist
Q.decreaseKey //update the queue
```

4.3 Database

We are using Firebase for our database. Firebase is a database stored in the cloud by Google. Firebase is nice because we are relying on Google for our security and stability of the server. Another advantage that Google provides is a web interface for Firebase, so we can easily see the data in real time. As of right now, there are two collections in the database: buildings and entrances. All buildings have a list of entrances, title, latitude, longitude, opening time, and closing times. All entrances have a reference back to the building that owns them, latitude, and longitude coordinates. We plan to include a third collection of data that includes pre processed information of areas with a lot of buildings marked.

4.4 Front End

For the front end we have chosen to use Google's web component library, Polymer. Polymer fits our needs best because it supports ES6 and TypeScript, which, when coupled together, make asynchronous development very easy. Polymer also has the behaviour of a single page application, which makes annotating the map very slick because no page reloads occur.

4.5 Labeling

Our idea relies heavily on allowing users to cut through buildings to reach their destination quicker. As a result we need information regarding exits, entrances, and hours of operations for the buildings which users want to use. In order to do this, we plan on having users annotate buildings that they use frequently. In order for our app to work with BYU, we are personally creating all the annotations for the campus. Users will be able to drop pins at building entrances and exits and lable hours of operations through our website.

5. Anticipated Results

We anticipate that users will be able to use our app to find the shortest and most comfortable route possible. Users will be able to find routes that cut across parking lots, fields and buildings to reach their destination as soon as possible. We also anticipate that users will be able to customize their routes to their personal preferences. Our application will work with the BYU campus but we hope others will annotate their surrounding areas so that users outside of BYU can benefit from this application.