Optimal Route Through 65 University of Washington Buildings

TJ Armstrong, Bryan Clark, David Moon

[armstts@u.washington.edu, brclark17@verizon.net, davidaugustmoon@gmail.com]

University of Washington, Math 480 B, professor's Craig Citro and William Stein

[craigcitro@gmail.com, wstein@gmail.com]

June 2, 2010

Abstract:

For Math 480: computer programming for the working mathematician, it was proposed that a solution to a modified traveling salesman problem (tsp) could be programmed for the buildings on the University of Washington campus. The GPS coordinates for the buildings on campus were collected via Google maps. Several functions were proposed starting with an in order tour, and then refined. After simulating the path using the different functions it was determined that the opt_edge_tour yields the shortest distance. The total distance was approximately 4.60266 miles to visit all 65 buildings on the main portion of the UW campus. Maps were made connecting the buildings in the order which they occurred for the different functions.

Problem Description:

The Traveling Salesman Problem (tsp) is one in which an object leaves the start location and visits all the desired destinations and returns back to the starting location in the minimal amount of time and/or distance. We will be performing a modified version of the traveling salesman problem. In our problem we will leave the starting point and visit all the desired location but will not have to return back to the starting point. *It is our goal to achieve, through a series of algorithms, an optimal route in which we travel to all 65 buildings on the University*

of Washington campus in order to walk the minimum distance. A list of GPS coordinates for the buildings was created which will be used as input for each of the different algorithms.

An efficient solution to a traveling salesman problem on UW campus could be a very useful tool for many different departments at the university. It could help with the best route for delivering mail, or it could shorten the distance walked by library delivery staff returning books to other libraries. It is also one of our goals to make this available to any department on campus who wishes to use it.

Simplifications:

The most significant simplification we made to our original problem was to use straight-line distances, rather than make a separate map of walkable paths. We had planned to create functions in python which would compute straight-line distances, and then we could transfer those paths to a graph of walkable paths. Our justification for this is that, on UW campus, you can walk through most everywhere on our map. It does not make sense to walk around a grass field, or even walk around a building when you can simply walk through them. We determined that our straight-line distances would be a more accurate representation of a path which someone could walk, than a map restricting travel to paved trails, paths, and roads.

Mathematical Model:

Our first step in modeling this problem was to create a function which computed the distance accumulated in traveling to a given set of points. To do this we used the function in_order_tour, which computed the total distance traveled by using the path given in the order in which the points are listed. Also, in order to form a baseline tour distance, we mapped our best guess as to an efficient path. It resulted in an approximately 6.02 mile tour.

We proposed several different strategies to determine the shortest possible route through the 65 buildings on the University of Washington campus. The first, and simplest, was to use python's sort command. We sorted our list of GPS coordinates in python, which arranged the list in order of ascending latitude. This resulted in a path where the most southerly building was chosen to travel to next. We then used in_order_tour to calculate the total path distance.

Our second strategy was to create an algorithm which takes a starting point and chooses the next point based on which building is closest to the current position. We used closest_point_tour, inputting a random start point, to arrange the list of UW coordinates in this fashion. Once the list is rearranged, closest_point_tour uses in_order_tour to compute the total path distance.

Our obvious next strategy was to create a greedy algorithm to compute closest_point_tour using all possible starting points, and return the shortest path. We did this with the shortest_closest_point_tour function. After creating the shortest_closest_point_tour path, we plotted the path on a map of UW campus.

Once we plotted the graph of our shortest_closest_point_tour, it was obvious that some simple modifications could be made to the path to decrease the total path distance. We used an optional edge exchange to switch edges which resulted in the path crossing over itself. This resulted in our most efficient path, opt_edge_tour.

We created a Sage Worksheet to graph the paths resulting from all of our strategies. We also made an @interact which takes as input a starting point, and a list of points to travel through. The interact uses closest_point_tour to determine a path, and graphs the resulting path and the total distance in miles. The street boundaries are also mapped onto the graphs for orientation.

Summary of functions used:

dist(P,Q): Takes as input a point P, and a point Q and computes the straight line distance between the two.

in_order_tour(L): Takes as input a list of points L, and computes the distance between each entry, then sums the distances to return the total distance travelled.

closest_point(P,L): Takes as input a point P, and a list of points L. Determines the closest point to P, from the points in L.

closest_point_tour(P,L): Takes as input a starting point P, and a list of points L. Creates a duplicate of the original list L. Determines the closest point to your current position and

appends that point to a new list tour_points, and removes it from the original list. in_order_tour is then used to compute the total distance of the created list. tour_points can be added to the return line to give the total distance, and the list of points in the order travelled.

shortest_closest_point_tour(L): Takes as input a list of points L. This is a greedy algorithm used to compare the closest_point_tour using every possible starting point in the list L. It returns the shortest distance found, and the starting point used.

sub_buildings(L): Takes as input a list of points L, and returns a list of the building names, as strings, associated with those GPS coordinates.

sub_coords(L): Takes as input a list of building names, as strings, and returns a list of GPS coordinates associated with those building names.

path_edges(L): Takes as input a list of points in a path order, and returns a list of edges in the same path order.

switch_coords(L): Takes as input a list of points, and returns a list with the elements of each point switched. This was used to orient the graphs in Sage according to the actual layout of UW campus.

graph_path(L): Takes as input a list of points in path order, and displays a graph of vertices, edges, and street boundaries.

Results:

In order to gauge how efficient our paths were, we mapped a tour based on our best guess at the shortest route. Our list of 65 UW building coordinates was first sorted using Python's sort command, and put in to the in_order_tour function where it gave a total distance of 14.74 miles, a 244% increase in distance from our best_guess_tour. Second, we put the list of coordinates into the function closest_point_tour using a random start point, which yielded a total distance of 5.55 miles, an 8% decrease in distance. Next we used the shortest_closest_point_tour to determine which closest_point_tour resulted in the shortest distance. It gave the shortest closest_point_tour distance using 'Forest Lab' as a starting point

at about 4.85 miles, a nearly 20% decrease in distance from the best_guess_tour. When the tour was placed on a map we saw that there were four places where the path crossed over itself. We uncrossed these in a process called a 2 opt exchange. After uncrossing the edges, the path distance was calculated and determined to be the best route. The total distance was approximately 4.60 miles, and a 24% decrease in distance from our best_guess_tour.

The results of the edge exchange were mapped, and the tour distance and route seemed like an accurate representation of the route that you should take to visit all 65 buildings on campus. After the routes were calculated a sage worksheet was created with graphic representations of all of the paths created by our functions. We also created an @interact which allows a person to pick the starting point of their tour and it will return the closest_point_tour using a given list and your chosen start point, in our case the coordinates of the buildings. The interact also returns the total distance in degrees and in miles.

Improvements:

One improvement, which could possibly give the best route, would be to create a tree of all of the possible routes you can make from the 65 buildings. Once you have the tree you then could implement a depth first search algorithm. The depth first search algorithm works in the following way; it follows a path to a leaf in the tree and records the distance, it then goes back up to the first node with unvisited paths and goes to a leaf, this repeats for all possible paths on the tree and only records the shortest at any given point in time. Another strategy that may help improve our solution to the traveling salesman problem would be to create a branch and bound algorithm. This algorithm keeps track of partial sums in the search tree and compares them with the best solution at any point in the search. A breadth first search algorithm would give a good solution. Breadth first search algorithms start with the root node and checks all the neighboring nodes. For each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal. Without implementing a greedy algorithm, which compares the distances of every possible path and returns the shortest path, it is very difficult to know whether you have the most optimal solution.

Conclusions:

After choosing our own path manually, by our best judgement, and by designing a few effective algorithms we believe we have found a very efficient route to tour the campus. The idea of walking between 65 buildings on the enormous University of Washington campus in about 4.6 miles seems to be very efficient. Also, we managed to decrease our travel distance from our best_guess_tour by nearly one quarter. We may be able to slightly improve our route; however, for the scope of this course, this was the best solution we found. If our algorithms were used on a larger list of GPS coordinates, be it random houses in a city or businesses across the country, it may be easier to spot a more efficient route. After spotting a more efficient route it may lead to the creation of a more efficient algorithm as well.

References:

"Google Maps" <http://maps.google.com/>

"Latitude and Longitude of a Point." *Maps and Utilities for the PC, Mac, iPhone, iPod touch and Smartphones*. N.p., n.d. Web. 19 May 2010. <http://itouchmap.com/latlong.html>.

Mertens, Stephan . "TSP Algorithms in Action Animated Examples of Heuristic Algorithms." *Startseite aller URZ-Webserver*. 1999-05-10, 10 May 1999. Web. 26 May 2010. <http://www-e.uni-magdeburg.de/mertens/TSP/index.html>.

"Traveling Salesman Problem." *Traveling Salesman Problem*. N.p., n.d. Web. 19 May 2010. <http://www.tsp.gatech.edu/>.

"Traveling salesman problem - Wikipedia, the free encyclopedia." *Wikipedia, the free encyclopedia*. N.p., n.d. Web. 25 May 2010. <http://en.wikipedia.org/wiki/Travelling_salesman_problem

Appendix A

(Python Code)

```
def dist(P,Q):
    '''calculate the distance between two points.'''
    d = sqrt((P[1]-Q[1])**2+(P[0]-Q[0])**2)
    return d

def cartesian_matrix(coords):
    '''create a distance matrix for the coords that uses straight line distance'''
    matrix=[]
    for i in L:
        for j in L:
            matrix.append([[i,j],dist(i,j)])
    return matrix

def in_order_tour(L):
    ''' Returns the total distance of a tour to each point traveled in order from first to last.'''
    tour_dist = []
    for n in range(1,len(L)):
        distn = dist(L[n-1],L[n])
        tour_dist.append(distn)
    return sum(tour_dist)
    #print "Tour distance:",sum(tour_dist)*69,"miles"

def closest_point(P,L):
    ''' Returns the closest point to P, from the list of points L.'''
    closest_point = L[0]
    for x in range(1,len(L)):
        if dist(P,L[x])<dist(P,closest_point):
            closest_point = L[x]
    return closest_point

def closest_point_tour(P,L):
    '''Returns a tour using the closest point to your current point.'''
    # Requires a starting point P
    # Also prints the path used to yield the distance returned.
    L2 = L[:]
    L2.append(P)
```

```
    tour_points = [P]
    L2.remove(P)
    while len(L2)>0:
        td = len(tour_points)
        a = tour_points[td-1]
        n = closest_point(a,L2)
        tour_points.append(n)
        L2.remove(n)
    return in_order_tour(tour_points)
    #print "Total tour distance in degrees:",in_order_tour(tour_points)
    #print "Total tour distance in miles:",69*in_order_tour(tour_points)

def shortest_closest_point_tour(L):
    '''Returns the shortest path from closest_point_tour(P,L) by comparing all possible starting points P.'''
    L2 = L[:]
    shortest_path = in_order_tour(L2)
    start_point = L2[0]
    for x in range(len(L2)):
        if closest_point_tour(L2[x],L2)<shortest_path:
            shortest_path = closest_point_tour(L2[x],L2)
            start_point = L2[x]
    print 'Distance in degrees:', shortest_path, start_point
    #print "Shortest path distance:",shortest_path,"degrees"
    #print "Shortest path distance:",69*shortest_path,"miles"

def sub_buildings(L):
    bld = []
    for x in L:
        if x == [47.660643780526165,-122.3104932971437]:
            bld.append('Burke Museum')
        if x == [47.65924370681618,-122.31082589106154]:
            bld.append('William H. Gates')
        …….. Buildings 3 through 63
        if x == [47.6604143516642,-122.3045334287126]:
            bld.append('McCarthy')
        if x == [47.660345703462404,-122.30927557425093]:
            bld.append('Theodor J observ')
    return bld
```

```
def sub_coords(L):
    coord = []
    for x in L:
        if x == 'Burke Museum':
            coord.append([47.660643780526165,-122.3104932971437])
        if x == 'William H. Gates':
            coord.append([47.65924370681618,-122.31082589106154])
         …….. Buildings 3 through 63
        if x == 'McCarthy':
            coord.append([47.6604143516642,-122.3045334287126])
        if x == 'Theodor J observ':
            coord.append([47.660345703462404,-122.30927557425093])
    return coord

def path_edges(path):
    edges = []
    for x in range(1,len(path)):
        a = [path[x-1],path[x]]
        edges.append(a)
    return edges

def switch_coords(L):
    switched_coords = []
    for x in L:
        switched_coords.append((x[1],x[0]))
    return switched_coords
```

Appendix B

(Sage Notebook)

Dist, in_order_tour, closest_point, closest_point_tour, and shortest_closest_point_tour all the same as appendix A

```
def graph_path(L):
    points_graph = point(L,color='black',size=25)
    path_graph = line(L,color='red')
    start_point = text('Start',L[0],color='green',fontsize=12,vertical_alignment='top')
    end_point = text('End',L[len(L)-1],color='green',fontsize=12,vertical_alignment='top')
    streets = line(switch_street_boundary,color='blue',thickness=2)
    street1 = text('15th Ave. NE',(-
122.31205701828003,47.65730869671061),color='black',fontsize=12,horizontal_alignment='left')
    street2 = text('NE 45th St.',(-
122.30645656585693,47.661254244799366),color='black',fontsize=12,vertical_alignment='top')
    street3 = text('Montlake Blvd.',(-
122.30203628540039,47.65613798222676),color='black',fontsize=12,horizontal_alignment='right')
    street4 = text('NE Pacific St.',(-
122.31057643890381,47.652004138405346),color='black',fontsize=12,vertical_alignment='bottom')
    street5 = text('NE Pacific Pl.',(-
122.30538368225097,47.650732120623),color='black',fontsize=12,vertical_alignment='bottom')
    # If plotting points other than on UW campus, you must take out streets, street1,...,street5 from the show line.

show(plot(points_graph)+plot(path_graph)+start_point+end_point+streets+street1+street2+street3+street4+stree
t5,axes=false)
    print "Total Distance:",in_order_tour(L)*69.047,"miles"
    print "Total Distance:",in_order_tour(L),"degrees"

@interact
def graph_closest_point_path(start_point=(0..len(switch_uw_paren)), L=input_box(default=switch_uw_paren)):
  try:  # The input list in the interact must use points on UW campus because it maps the street boundaries around
UW.
    chosen_path = closest_point_tour(L[start_point],L)
    chosen_path_graph = line(chosen_path,color='red')
    chosen_path_points = point(chosen_path,color='black',size=20)
    interact_start_point = text('Start',chosen_path[0],color='green',fontsize=12,vertical_alignment='top')
    interact_end_point = text('End',chosen_path[len(L)],color='green',fontsize=12,vertical_alignment='top')
    streets = line(switch_street_boundary,color='blue',thickness=2)
```

```
   street1 = text('15th Ave. NE',(-
122.31205701828003,47.65730869671061),color='black',fontsize=12,horizontal_alignment='left')
   street2 = text('NE 45th St.',(-
122.30645656585693,47.661254244799366),color='black',fontsize=12,vertical_alignment='top')
   street3 = text('Montlake Blvd.',(-
122.30203628540039,47.65613798222676),color='black',fontsize=12,horizontal_alignment='right')
   street4 = text('NE Pacific St.',(-
122.31057643890381,47.652004138405346),color='black',fontsize=12,vertical_alignment='bottom')
   street5 = text('NE Pacific Pl.',(-
122.30538368225097,47.650732120623),color='black',fontsize=12,vertical_alignment='bottom')

show(plot(chosen_path_graph)+plot(chosen_path_points)+interact_start_point+interact_end_point+streets+stree
t1+street2+street3+street4+street5, axes=False)
   print 'Total Distance:',in_order_tour(chosen_path)*69.047,'miles'
   print 'Total Distance:',in_order_tour(chosen_path),'miles'
 except:
   print "..."
```