Taking Beauty into Account when Finding the Shortest Path

By : Maha Alkhairy

Abstract::

In this project, we want to find the most beautiful route [3] given a map. Although beauty is an abstract concept, it can still be quantified according to each user's perception of beauty and be treated algorithmically. Thus, the most beautiful path can be determined by the user of the algorithm, who inputs "beauty" as a dictionary of values assigned to the nodes. We then find the most beautiful path, as defined above, by modifying Dijkstra's algorithm to consider beauty as an important factor. Finally, we test the modified algorithm on real world data obtained from openstreetmap.org [5].

a. Introduction ::

Many people use GPS systems to help them find the shortest ("best") route from a starting point to the destination. Is speed the only thing we need when going from place to place? Many people will agree that a slightly longer but more beautiful path is better than the shortest "fastest" path. In this project we will be using Python to modify Dijkstra's algorithm to consider beauty as an important factor. In a similar project, researchers have approached it experimentally (for example, having a group of people rate which path is more beautiful and having huge crowds help validate paths ([2],[6]). Instead, in this project, we shall be approaching it algorithmically.

b. Description / Methods:

We considered two ways of representing beauty: in the first approach, we treat beauty as weights on the edges. In the graph, each edge will represent one meter, and the weight of the edge will be its beauty, with a smaller number representing a higher beauty. To implement this approach, we have to modify the graph (whose edge weights originally represents physical distances) and create more nodes so as to make the distances equal to one meter.

In the second approach, we consider beauty of places (nodes). The edge weights will continue to represent physical distances, and the node weights will represent the beauty of the places, again, with a smaller number representing a higher beauty. We then add the node weights to the outcoming edges of the node, hence creating a weighted directed graph on which we can apply Dijkstra's algorithm.

The first approach works and finds the most beautiful path, but is not very practical because we need to gather information on the beauty of every single meter along a path, in addition to the fact that most people rate the beauty of landmarks, not paths. The second approach is more reliable and practical, but a slight complication will be the balance between beauty of places and the distance of a path -- we have to choose the "beauty weights" carefully.

We decided to implement the second approach. This approach can deal with large data sets much more efficiently than the first approach. To verify our algorithm, we applied it first to smaller manually constructed graphs, then to real world data obtained from [5]. We shall discuss the output of the algorithm in section (d).

c. Code with Instructions ::

Please open `dijkstra_inputSt.py` located in `code.zip` to run

The main code is an enhanced code of [4]::
(https://github.com/nvictus/priority-queue-dictionary/blob/master/examples/dijkstra.py)

I have four real world data text files, obtained from openstreetmap.org [5]:

- "`src1.txt`" :: has the most nodes.
- "`src3.txt`" :: has less nodes.
- "`src2.txt`" :: has the least nodes.
- "`ex.txt`" :: is the map on which we ran the algorithm for the analysis

After you run "`dijkstra_inputSt.py`", you will be prompted to enter ::

1. Filename: one of the three above or any txt file you get from openstreetmap
2. Beauty of the nodes, a dictionary of with the id of the nodes as keys and the beauty as the values
3. The id of the Start node
4. The id of the End node

Output (print to console)::

1. The physical length of the path
2. The list of street names the path goes through

:::: Code ::::

dijkstra_inputSt.py (main code)

```
import pqdict
import parse

def dijkstra(graph, source, target=None):
    """
    Computes the shortests paths from a source vertex to every other vertex in
    a graph

    """
# The entire main loop is O( (m+n) log n ), where n is the number of
# vertices and m is the number of edges. If the graph is connected
# (i.e. the graph is in one piece), m normally dominates over n, making the
# algorithm O(m log n) overall.

    dist = {}
    pred = {}

# Store distance scores in a priority queue dictionary
    pq = pqdict.PQDict()
    for node in graph:
        if node == source:
            pq[node] = 0
        else:
            pq[node] = float('inf')

    # Remove the head node of the "frontier" edge from pqdict: O(log n).
    for node, min_dist in pq.iteritems():
        # Each node in the graph gets processed just once.
        # Overall this is O(n log n).
        dist[node] = min_dist
        if node == target:
            break

        # Updating the score of any edge's node is O(log n) using pqdict.
        # There is _at most_ one score update for each _edge_ in the graph.
        # Overall this is O(m log n).
        for neighbor in graph[node]:
            if neighbor in pq:
                new_score = dist[node] + graph[node][neighbor]
                if new_score < pq[neighbor]:
                    pq[neighbor] = new_score
                    pred[neighbor] = node

    return dist, pred

def shortest_path(graph, source, target):
    dist, pred = dijkstra(graph, source, target)
    end = target
    path = [end]
    while end != source:
        end = pred[end]
        path.append(end)
```

```
    path.reverse()
    return path

ggggg = parse.getGraph()
Graph = ggggg[0]
SN = ggggg[1]

GraphB = input('Enter Graph - with beauty "example: {"a":1.00(very beautiful),
"b":x(not beautiful) x < 100.00}" : ')

StartNode = input('Enter "StartNode": ')

EndNode = input('Enter "EndNode": ')

## increments the edge weights by the beauty on the node
newGraph={}
for key in Graph.keys():
    newGraph[key]={};
    for keykey in Graph[key]:
        if  isinstance(GraphB.get(key),float):
            newGraph[key][keykey]= float(GraphB.get(key)) + Graph[key][keykey]
        else:
            newGraph[key][keykey] =  Graph[key][keykey] + 100.00

## get the physical length of the path
sspp = shortest_path(newGraph, StartNode, EndNode)
dddd = 0
for iiii in range(len(sspp) - 1):
    dddd += Graph[sspp[iiii]][sspp[iiii+1]]
nnmm = []
for iiii in range(len(sspp) - 1):
    nnmm.append(SN[sspp[iiii]][sspp[iiii+1]])

if __name__ == '__main__':
    # A simple edge-labeled graph using a dict of dicts
    graph = newGraph

    dist, path = dijkstra(graph, source = StartNode)

    print "physical world distance: " + str(dddd) + " meters"
    print "street names: " + str(nnmm)
```

d. Results::

We were able to extend Dijkstra's algorithm [4] to find a reasonably short path with beautiful nodes (places) along its way. As an application for this algorithm, we can use it on a real map and weight the beauty of the nodes with respect to historical monuments and tourist attractions.

To demonstrate the algorithm, we obtained real world data from openstreetmap.org [5] to get a map of Boston, considering T-stations as "beautiful." All other nodes are considered not beautiful, with a score of 100 (a lower score corresponds to higher beauty).

We chose the starting point to be "Kendall Square"`("61318452")` and the ending point to be the "Aquarium"(`"515465236"`), and we ran the algorithm on the data. We got the following result and mapped it to a map.

Dictionary of "Beauty"

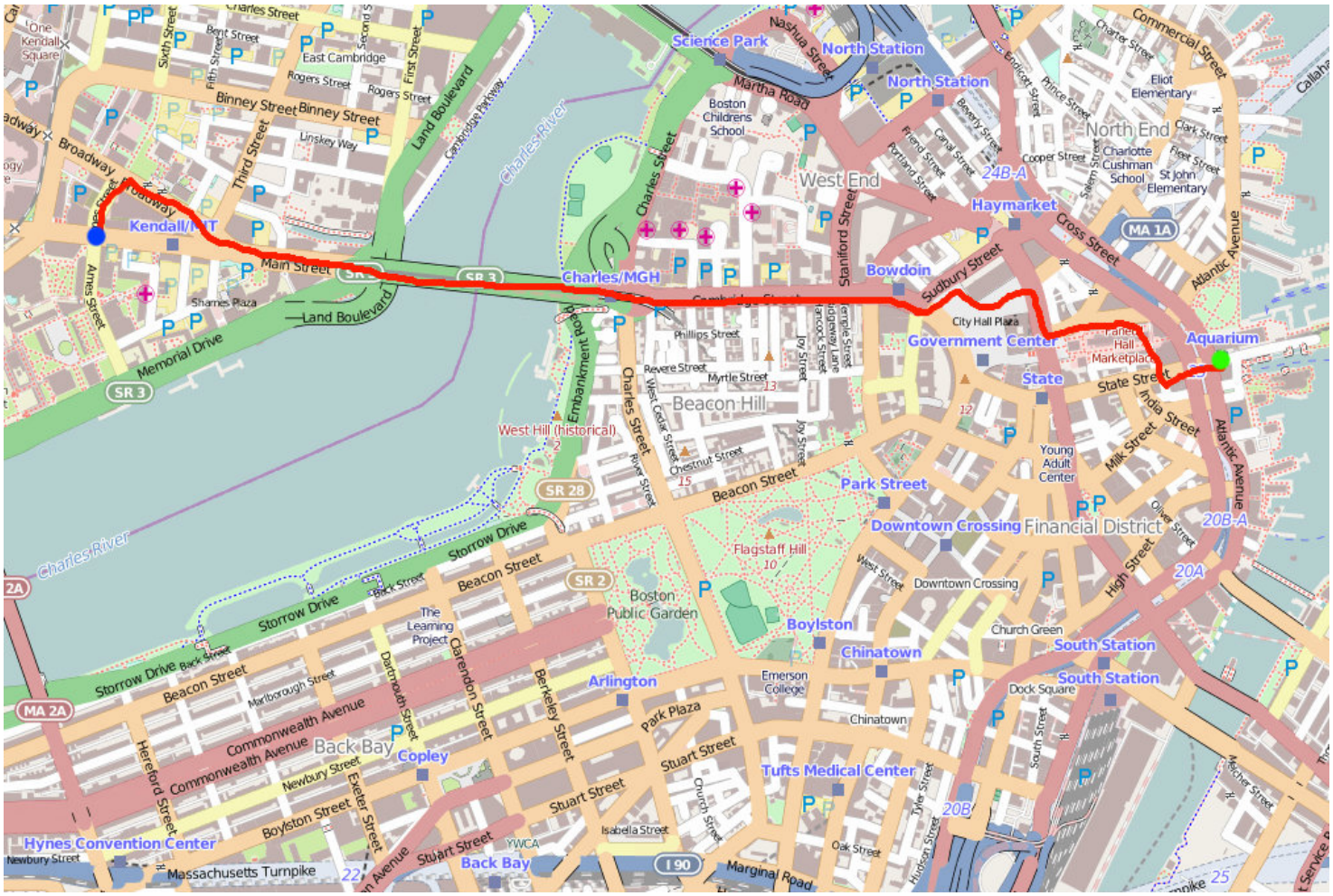
Node ID	(Name)	Beauty Rating
"61318452"	kendall	12.00
"61372757"	charles/mgh	15.00
"1818197688"	hynes convention center	4.00
"1185490502"	back bay	3.00
"61341300"	copley	1.00
"1132394763"	arlington	1.00
"1399799372"	tufts	13.00
"1038884878"	boylston	1.00
"61344172"	china town	40.00
"61341689"	park st	1.00
"61341425"	downtown crossing	25.00
"61356671"	bowdoin	6.00
"61519547"	government center	10.00
"61372620"	science park	12.00
"61473920"	haymarket	15.00
"61364086"	north station	20.00
"515465236"	state	30.00
"61381868"	south station	60.00
"515465236"	aquarium	0.00

Result of running the code::

```
physical world distance:
3320.88777063 meters

street names:
['Ames Street', 'Broadway', 'Broadway', 'Broadway',
'Broadway', 'Broadway', 'Third Street', 'Broadway',
'Broadway', 'Broadway', 'Broadway', 'Broadway',
'Broadway', 'Main Street', 'Main Street', 'NoName',
'NoName', 'NoName', 'Longfellow Bridge', 'Longfellow
Bridge', 'Longfellow Bridge', 'Longfellow Bridge',
'Cambridge Street', 'Cambridge Street', 'Cambridge
Street', 'Cambridge Street', 'Cambridge Street',
'Cambridge Street', 'Cambridge Street', 'Cambridge
Street', 'Cambridge Street', 'Cambridge Street',
'Cambridge Street', 'Cambridge Street', 'Cambridge
Street', 'Cambridge Street', 'Cambridge Street',
'Cambridge Street', 'Cambridge Street', 'Cambridge
Street', 'Cambridge Street', 'Cambridge Street',
'Cambridge Street', 'Cambridge Street', 'Cambridge
Street', 'Cambridge Street', 'Cambridge Street',
'Cambridge Street', 'Cambridge Street', 'Cambridge
Street', 'Cambridge Street', 'Sudbury Street', 'NoName',
'City Hall Plaza', 'City Hall Plaza', 'City Hall Plaza',
'City Hall Plaza', 'City Hall Plaza', 'City Hall Plaza',
'City Hall Plaza', 'City Hall Plaza', 'City Hall Plaza',
'NoName', 'Congress Street', 'Congress Street', 'Congress
Street', 'Congress Street', 'NoName', 'NoName', 'North
Street', 'NoName', 'NoName', 'NoName', 'NoName', 'NoName',
'NoName', 'Commercial Street', 'State Street', 'State
Street']
```

Which corresponds to the following path:



e. Discussion::

The algorithm finds a relatively short beautiful path, as desired (see map above). However, there is a potential problem with the algorithm: we need a thorough data set for "beauty" to make the algorithm output a more accurate result. As we can see from the map, towards the end of the path, we would have expected the algorithm to return a path that contains "Government Center" and "State," given that they are beautiful nodes. The algorithm, however, does the contrary. This is because of the lack of data for other nodes, which defaults the beauty to 100, making the algorithm somewhat dependent on the number of nodes in the path. If we had better data, e.g. "beauty" for all nodes, this problem can be solved.

Another potential problem will be the trade off between walking distance and beauty: as discussed in the previous section, we need to choose the beauty carefully to make it correspond to intuition: how many more meters is the user willing to walk for more beautiful places in the path? This requires experimental research, e.g. by surveys of large number of tourists.

Despite its imperfections, this program explored possible ways to enhance one of the classical problems in algorithms: the shortest path. We have investigated the role of "beauty" in finding a path, and allowed for user customization of the concept of beauty. We firmly believe that with more field research and a better data set, this project can be extended to various real life applications.

References

1) Alkhairy, Maha A. "Tracking Progress - 4800 Project - Maha Alkhairy." Web. 02 Aug. 2014. <https://docs.google.com/a/husky.neu.edu/document/d/1n058JQTgV-Qh0IaSG3BrcFIz1TW0HRjZdrpbN9yk8EA/edit>.

2) "Dquercia@yahoo-inc.com Yahoo Labs Schifane@di.unito.it Alucca@yahoo-inc.com." Web. <http://arxiv.org/pdf/1407.1031v1.pdf>.

3) "Forget the Shortest Route Across a City; New Algorithm Finds the Most Beautiful | MIT Technology Review." *MIT Technology Review*. Web. 02 Aug. 2014. <http://www.technologyreview.com/view/528836/forget-the-shortest-route-across-a-city-new-algorithm-finds-the-most-beautiful>.

4) "Nvictus/priority-queue-dictionary." *GitHub*. Web. 02 Aug. 2014. <https://github.com/nvictus/priority-queue-dictionary/blob/master/examples/dijkstra.py>.

5) "OpenStreetMap." *OpenStreetMap*. Web. 01 Aug. 2014. <https://www.openstreetmap.org/export#map=19/42.38635/-71.07889>.

6) "We Need This: A Maps App That Algorithmically Finds You the Scenic Route | Design | WIRED." *Wired.com*. Conde Nast Digital, 13 July 0014. Web. 02 Aug. 2014. <http://www.wired.com/2014/07/we-need-this-a-maps-app-that-algorithmically-finds-you-the-scenic-route/>.

7) "We Need This: A Maps App That Algorithmically Finds You the Scenic Route | Design | WIRED." *Wired.com*. Conde Nast Digital, 13 July 0014. Web. 02 Aug. 2014. <http://www.wired.com/2014/07/we-need-this-a-maps-app-that-algorithmically-finds-you-the-scenic-route/>.