IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Peter J. Schmid

January 31, 2017

1 SHORTEST PATH IN GRAPHS (DIJKSTRA ALGORITHM)

A graph is a mathematical construct consisting of a set of vertices that are connected by a set of edges. The edges can contain directional information, i.e. pointing from vertex to vertex, or just simply connecting two vertices. In former case, we refer to the graph as a directed graph; in the latter case, the graph is undirected. Often, the edges in an undirected or directed graph carry a weight which, in applications, is related to distance, capacity or cost of travel.

Graphs appear in many applications, such as traffic flow through cities, telephone or power networks, or work flow charts in manufacturing, to name but a few.

In complex graphs we are often interested in the path through the graph that (i) connects two specified vertices and (ii) has a minimum total weight. For example, in planning a route through a city from point A to point B using a GPS device, we are interested in a set of connected edges that start at A and end at B such that the total sum of the edge weights is as small as possible. The edge weight in this case could be the length of the edge (yielding the shortest path from A to B) or the travel time along the edge (resulting in the fastest path from A to B). The algorithm that efficiently determines the minimal-cost path in a graph is Dijkstra's algorithm. It was conceived and formulated in 1956 by Edsger Dijkstra, a Dutch computer scientist. In its original form it can compute the shortest path from a point A in the graph to a point B, or, more generally, it can calculate the shortest path from a point A to any other vertex in the graph, thus generating a shortest-graph tree structure.

1.1 ALGORITHM

For computing the shortest distance through a graph, we first have to decide on a starting place, i.e., an initial or source node. Dijkstra's algorithm can then be stated in the following three steps.

1. We set the distance to the initial node to zero, and the distance to all other nodes to infinity. Furthermore, we set the initial node to the `current` node and label all other nodes as `unvisited`.

2. From the `current` node, determine all its `unvisited` neighbors. Then compute the distance to all these `unvisited` neighbors by adding the distance to the `current` node to the distances to all `unvisited` neighbors. Only update the distances to the `unvisited` neighbors, if they are smaller than their present values (otherwise, keep the present value).

3. Select from the `unvisited` nodes the one with the smallest distance, record its parent node as the `current` node. Then remove it from the `unvisited` list and select it as the new `current` node. Go to step 2.

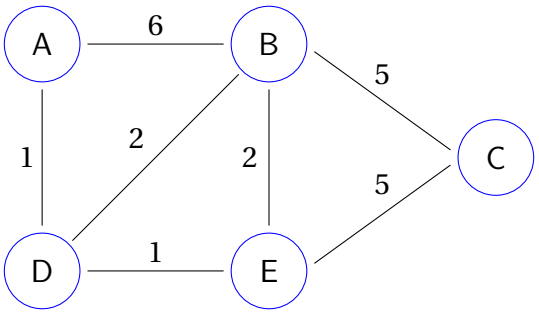

Figure 1.1: Example of a simple graph.

This algorithm terminates when no `unvisited` nodes can be found. The above algorithm is best explained with the help of an example (see figure 1.1). It is based on a simple graph with five nodes and seven edges. Based on the connectivity of the graph, we can formulate a **connectivity** or **adjacency matrix** as follows

$$\mathsf{W} = \begin{pmatrix} 0 & 6 & 0 & 1 & 0 \\ 6 & 0 & 5 & 2 & 2 \\ 0 & 5 & 0 & 0 & 5 \\ 1 & 2 & 0 & 0 & 1 \\ 0 & 2 & 5 & 1 & 0 \end{pmatrix} \tag{1.1}$$

with $\mathsf{W} = \{w_{ij}\}$, where w_{ij} denotes the weight of the edge between node i and node j. The ordering of the matrix rows and columns is following the sequence ABCDE. Furthermore, we take a value of $w_{ij,i\neq j} = 0$, if the node i is not connected to node j. We wish to determine the shortest path from node A to node C.

The tableau below (see figure 1.2) outlines the steps of Dijkstra's algorithm to compute the shortest path for the graph shown in figure 1.1.

`step 0:` We take A as our `current` node and set $d[\mathsf{A}] = 0$ and $d[\mathsf{B}], d[\mathsf{C}], d[\mathsf{D}], d[\mathsf{E}] = \infty$.

`step 1:` We then determine the `unvisited` neighbors of A : the two nodes B and D. The distance to these nodes is the distance to the `current` node A, i.e., $d[\mathsf{A}] = 0$, plus the respective weights of the edges from A to B and D. We update the distance of these two nodes as $d[\mathsf{B}] = 6$ and $d[\mathsf{D}] = 1$. As we update the distance to these two nodes, we also store the parent-node information for the two nodes, marked by the subscript $_{\mathsf{A}}$, which indicates that the path to B and D comes from A. We proceed by choosing the `unvisited` node with the minimum distance, in our case, node D (see the teal square), as the new `current` node and remove it from the `unvisited` list.

	$d[\mathsf{A}]$	$d[\mathsf{B}]$	$d[\mathsf{C}]$	$d[\mathsf{D}]$	$d[\mathsf{E}]$
step 0	0	∞	∞	∞	∞
step 1	0	6_{A}	∞	1_{A}	∞
step 2	0	3_{D}	∞	1_{A}	2_{D}
step 3	0	3_{D}	7_{E}	1_{A}	2_{D}
step 4	0	3_{D}	7_{E}	1_{A}	2_{D}

Figure 1.2: Dijkstra tableau

`step 2:` We repeat the same process with node D as the `current` node. Its `unvisited` neighbors are nodes B and E. The distances to these nodes are computed as 3 and 2, respectively, simply by adding the associated edge values 2 and 1 to the distance of the `current` node D. The smaller of the two distances corresponds to node E, which will become our next `current` node (teal square).

`step 3:` With E as the new `current` node, its `unvisited` neighbors consist only of node C. The distance to C is the sum of the `current` node's distance ($d[\mathsf{E}] = 2$) and the edge weight from E to C, which is 5. The distance to C is thus 7. We denote by subscript $_{\mathsf{E}}$ the parent node E. With E the only `unvisited` neighboring node, we set the `current` node to E.

`step 4:` A final application of a Dijkstra step does not produce any `unvisited` neighboring nodes, and the algorithm terminates.

The final line in figure 1.2 contains the information about shortest-distance paths from node A to all other nodes in the graph: node B can be reached at a cost of 3, node C at a cost of 7, node D at a cost of 1, and node E can be reached with a cost of 2. The stored subscripts allow us to reconstruct the shortest path. Starting at C, the cost 7_{E} identifies node E as the parent node. The parent node of E is node D whose parent node is A, our

starting node. The shortest path from A to C thus follows: A → D → E → C. See the red path in figure 1.2 and the critical path (in red) in figure 1.3.

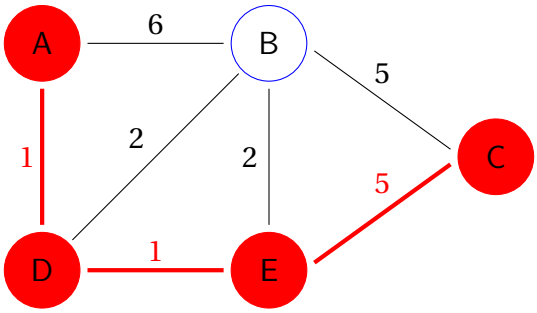

Figure 1.3: graph2

1.2 Implementation

The following python function implements the Dijkstra algorithm. The graph is defined by the weight matrix `wei` which contains its adjacency structure (given by the non-zero elements) and the cost associated with travelling along edges. The `Dijkst`-function specifies the starting (`ist`) and ending (`isp`) node in the network and produces the shortest path through the graph. The function uses an `UnVisited`-array which is 1 for `unvisited` nodes and 0 for visited ones. Line 26 produces a list of neighboring nodes, extracted from the `current` row of the weight matrix; by eliminating the zero elements from this list, only the `unvisited` neighbors remain (see line 27).

```
1  import numpy as np
2  import scipy as sp
3  import math as ma
4  import sys
5  import time
6
7  def Dijkst(ist,isp,wei):
8      # Dijkstra algorithm for shortest path in a graph
9      #    ist: index of starting node
10     #    isp: index of stopping node
11     #    wei: weight matrix
12
13     # exception handling (start = stop)
14     if (ist == isp):
15         shpath = [ist]
16         return shpath
17
18     # initialization
```

```
19      N         =  len(wei)
20      Inf       =  sys.maxint
21      UnVisited =  np.ones(N,int)
22      cost      =  np.ones(N)*1.e6
23      par       = -np.ones(N,int)*Inf
24
25      # set the source point and get its (unvisited) neighbors
26      jj            = ist
27      cost[jj]      = 0
28      UnVisited[jj] = 0
29      tmp           = UnVisited*wei[jj,:]
30      ineigh        = np.array(tmp.nonzero()).flatten()
31      L             = np.array(UnVisited.nonzero()).flatten().size
32
33      # start Dijkstra algorithm
34      while (L != 0):
35          # step 1: update cost of unvisited neighbors,
36          #         compare and (maybe) update
37          for k in ineigh:
38              newcost = cost[jj] + wei[jj,k]
39              if ( newcost < cost[k] ):
40                  cost[k] = newcost
41                  par[k]  = jj
42
43          # step 2: determine minimum-cost point among UnVisited
44          #         vertices and make this point the new point
45          icnsdr     = np.array(UnVisited.nonzero()).flatten()
46          cmin,icmin = cost[icnsdr].min(0),cost[icnsdr].argmin(0)
47          jj         = icnsdr[icmin]
48
49          # step 3: update "visited"-status and determine neighbors of new
50          UnVisited[jj] = 0
51          tmp           = UnVisited*wei[jj,:]
52          ineigh        = np.array(tmp.nonzero()).flatten()
53          L             = np.array(UnVisited.nonzero()).flatten().size
54
55      # determine the shortest path
56      shpath = [isp]
57      while par[isp] != ist:
58          shpath.append(par[isp])
59          isp = par[isp]
60      shpath.append(ist)
61
62      return shpath[::-1]
```

The main code

```
if __name__ == '__main__':

    # starting and stopping node
    ist = 4
    isp = 3

    # adjacency matrix
    wei = np.array([[ 0, 20,  0, 80, 0,  0, 90,  0],
                    [ 0,  0,  0,  0, 0, 10,  0,  0],
                    [ 0,  0,  0, 10, 0, 50,  0, 20],
                    [ 0,  0, 10,  0, 0,  0, 20,  0],
                    [ 0, 50,  0,  0, 0,  0, 30,  0],
                    [ 0,  0, 10, 40, 0,  0,  0,  0],
                    [20,  0,  0,  0, 0,  0,  0,  0],
                    [ 0,  0,  0,  0, 0,  0,  0,  0]])

    shpath = Dijkst(ist,isp,wei)
    print ist,' -> ',isp,' is ',shpath
```

1.3 Example

As a more realistic example, we wish to find the shortest and fastest path through a network of streets in a large city. This is a task that is routinely solved by a navigational GPS system in your car (although with a slightly different algorithm).
Figure 1.4 shows a city map of Rome, with some of the city's attractions marked. In addition, we have "discretized" some of the main streets by introducing vertices and connectig them by edges; a total of 58 nodes have been added. The edges between the nodes carry a weight that includes their distance as well as the speed limit: in this manner, we can compute the shortest path (using distance as a weight) or the fastest path (using the time it takes to traverse an edge as a weight) through the city. We select St. Peter's Square (node 13 in the Northwest of the city) as our starting point and the Coliseum (node 52 in the Southeast of the city) as our destination. Most streets (edges) can be traversed in both directions, although not necessarily at the same speed; some streets, however, are one-way streets and can be traversed only in one direction (e.g. the two main roads along the river are one-way streets with $3-5-8-\ldots-53$ on the Western bank going mostly North-South and $54-49-47-\ldots-2$ on the Eastern bank going mostly South-North).
Dijkstra's algorithm can be readily applied to the network, once the adjacency matrix has been established from the coordinate and speed-limit information. An additional function, producing the weight matrix from this information, is necessary (see below).

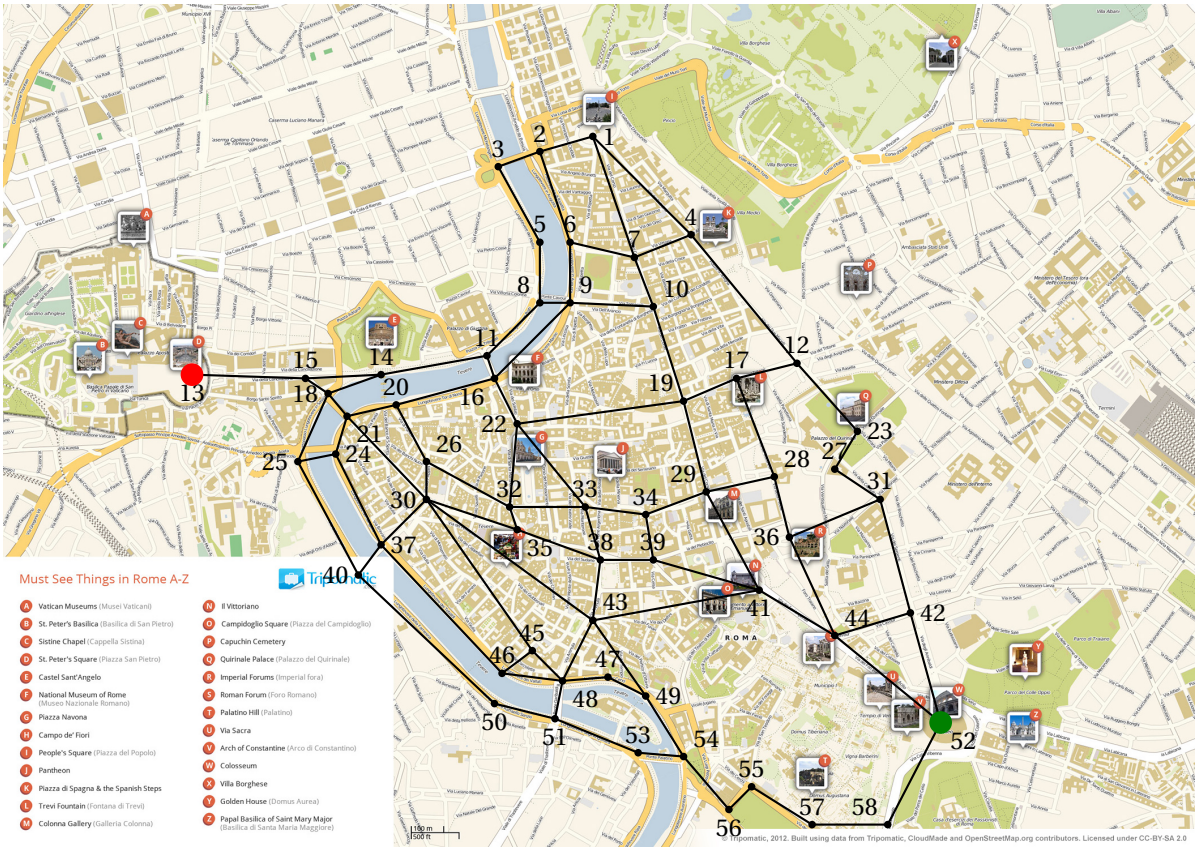

Figure 1.4: Map of Rome with a simplified traffic network consisting of 58 vertices and 156 edges (counting bi-directional edges double). What is the shortest or fastest path from node 13 (red symbol: St. Peter's Square) to node 52 (green symbol: Coliseum)?

```
def calcWei(RX,RY,RA,RB,RV):
    # calculate the weight matrix between the points

    n    = len(RX)
    wei = np.zeros((n,n),dtype=float)
    m    = len(RA)
    for i in range(m):
        xa = RX[RA[i]-1]
        ya = RY[RA[i]-1]
        xb = RX[RB[i]-1]
        yb = RY[RB[i]-1]
        dd = ma.sqrt((xb-xa)**2 + (yb-ya)**2)
        tt = dd/RV[i]
        wei[RA[i]-1,RB[i]-1] = tt
    return wei
```

The main code then reads in the geometric information (coordinate of vertices, `RomeX` and `RomeY`, and connectivity, `RomeA` and `RomeB`) from file; this information would be stored in form of maps on your GPS device. Additional information about the speed on each edge (`RomeV`) would come from established speed limits or interactively from current traffic information (reporting accidents, traffic jams, temporary detours or closures).

```
if __name__ == '__main__':

    # EXAMPLE 2 (path through Rome)
    RomeX = np.empty(0,dtype=float)
    RomeY = np.empty(0,dtype=float)
    with open('RomeVertices','r') as file:
        AAA = csv.reader(file)
        for row in AAA:
            RomeX = np.concatenate((RomeX,[float(row[1])]))
            RomeY = np.concatenate((RomeY,[float(row[2])]))
    file.close()

    RomeA = np.empty(0,dtype=int)
    RomeB = np.empty(0,dtype=int)
    RomeV = np.empty(0,dtype=float)
    with open('RomeEdges','r') as file:
        AAA = csv.reader(file)
        for row in AAA:
            RomeA = np.concatenate((RomeA,[int(row[0])]))
            RomeB = np.concatenate((RomeB,[int(row[1])]))
            RomeV = np.concatenate((RomeV,[float(row[2])]))
```

```
22      file.close()
23
24      wei = calcWei(RomeX,RomeY,RomeA,RomeB,RomeV)
25
26      ist = 12 # St. Peter's Square
27      isp = 51 # Coliseum
28
29      # use the Dijkstra algorithm
30      t0 = time.time()
31      shpath = Dijkst(12,51,wei)
32      t1 = time.time()
33      print 'Dijkstra:     ',ist+1,' -> ',isp+1,' is ',np.array(shpath)+1,t
```

This exercise also demonstrates the efficiency of Dijkstra's algorithm: the above task of finding the optimal path through a simplified traffic network can be solved on a standard laptop computer in about a millisecond.