

On the Science Behind Computing (Part I)

Samir Khuller *

1 Shortest Path Routing in Graphs

Many applications need to compute routes to go between nodes in a graph. This occurs when we use online maps to get directions. When we use online maps, we enter a starting and ending address, and ask for a route to go from the start to the destination. The starting and ending points are really nodes in a very large graph. When we get directions, we really get a path in the graph that connects the two nodes. A *path* is a sequence of nodes connected by edges. Consider the graph shown in Fig. 1. For example, the following is a path – r, a, d, e , since we have a sequence of edges $(r, a), (a, d), (d, e)$. Note that there are many many possible paths between a pair of nodes in a graph. Another path from r to e is r, b, f, e . Most map based software packages try to optimize for some quantity. For example, one could try to find the *shortest* possible route, or perhaps a route that avoids highways, or a route that minimizes travel time (this may be different from taking the route that is shortest in distance due to a variety of factors – traffic lights, different speed limits etc.).

In data networks such as the Internet, data is transferred in packets – each time your web browser downloads a video, or any image, the data is transferred over the network from the server to your computer. Each computer on the Internet has an address. An IP address looks something like 70.108.35.210. (You can use the URL `www.ip-address.com` to find out the IP address of your computer.) This data may travel large distances over the Internet and may go through a sequence of nodes (called routers) before it gets to your computer. The packets do not travel in an uncontrolled way, otherwise there would be complete chaos! The routers “know” where to forward each packet simply based on the destination address. In this case nodes maintain “lookup tables”. For example node A may receive a packet on some link that has the destination address X . Where do we send this packet? In the lookup table T_A (the table for node A), we will have an array `addr` such that if `addr[j] = X` then `forward[j]=B` which means that a packet meant for node X should be forwarded on the link to neighbor B . Thus when we get a packet meant for node A , we first check to see if is our address. If so, we do not forward the packet. Otherwise we look up the table to find the index j such that `addr[j]=X` (recall binary search!). We now forward the packet to the neighbor specified in `forward[j]`.

The following is an example of what may be stored in the table for a node. Ofcourse in reality we would not have letters like A, B etc. but IP addresses of nodes.

To simplify the problem a little, let us focus on a special case. Let us focus our attention on a particular node, say r (called the root). Suppose now that each node v wishes to compute the address of the neighbor to forward the packet to if the destination address is r . In fact, for this case assume that all packets are going to r – each node simply needs to know where to forward packets meant for node r . If we can do this computation for *every* possible node r , then we can

*Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail : `samir@cs.umd.edu`.

addr	forward
A	-
B	B
C	C
D	C
X	B
Y	B
Z	C

build the desired tables. So when we get a packet for any destination node, our table would tell us what to do with the packet. Moreover, this routing scheme will guarantee that at each step when we forward the packet, we are getting closer to r .

We will study an algorithm proposed by Edsger Dijkstra in 1959. The main idea behind the algorithm is to maintain an array called **Visited**, which is initially set to **false** for every node i . In other words we set $\text{Visited}[i] = \text{false}$ for each node i . When the algorithm terminates, then $\text{Visited}[i]$ is **true** for all nodes.

In addition, the algorithm maintains two other arrays – the first array $\text{forward}[i]$ tells node i which neighbor to send packets to, that are meant for the root r . The second array $\mathbf{d}[i]$ records the length of a known path to the root r .

For nodes i that have $\text{Visited}[i] = \text{true}$, the value of $\mathbf{d}[i]$ is *exactly* the length of a shortest path to the root r . For the remaining nodes it is clearly an upper bound on the length of a shortest path to r , since we know of a path of length $\mathbf{d}[i]$ but are not confident that this is indeed the shortest path to r .

When the algorithm starts, we initially set $\mathbf{d}[r]$ to be 0. For the remaining nodes, we can set the $\mathbf{d}[i]$ value as ∞ . Among all the nodes with $\text{Visited}[i]$ being false, clearly the node with the minimum $\mathbf{d}[i]$ value is r (see Fig. 1). If we scan the edges adjacent to r , then we can see that we have paths to a and b of lengths 1 and 5 respectively. Note that these are currently estimates of the shortest path lengths. In other words, we know that there is a path of this length to the node, but we are not confident that these actually denote the shortest path length. In fact, for node b , the shortest path has length 4 by going through a .

However, the algorithm proposed by Dijkstra observes that if we consider the node p with minimum $\mathbf{d}[j]$ among all nodes that have $\text{Visited}[j] = \text{false}$, for these nodes we can **confirm** that the \mathbf{d} value is indeed correct, or it is the shortest path distance from root r . If so, we can set $\text{Visited}[p] = \text{true}$. We repeat this until all nodes have $\text{Visited}[i]$ set to **true**. In the previous example, this would allow us to claim that for node a the distance value is indeed correct and no shorter path to a can exist.

In addition, after we are sure that we have the shortest path to p , we can consider the edges adjacent to p in turn. If we have an edge (p, q) with $\text{Visited}[q] = \text{false}$ then we check to see if we have discovered a shorter path from q to r , through p by checking if $\mathbf{d}[p] + w(p, q) < \mathbf{d}[q]$. If this condition is true then we have indeed found a shorter path to r and we also update $\text{forward}[q]$ to be p , and set $\mathbf{d}[q]$ to $\mathbf{d}[p] + w(p, q)$.

The argument as to why this algorithm works is not immediately obvious. The main property that this proof relies on that may not be immediately obvious is that the $\mathbf{d}[i]$ is actually the shortest path from r to i , that is *restricted* to going through nodes that have their **Visited** value

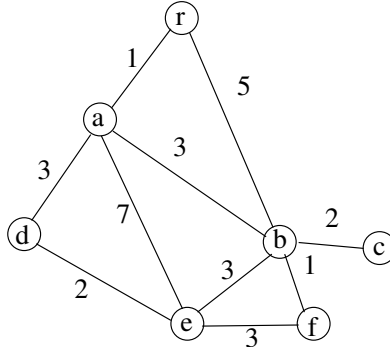


Figure 1: Figure showing a small graph.

Node	r	a	b	c	d	e	f
	0	∞	∞	∞	∞	∞	∞
r	-	$1(r)$	$5(r)$	∞	∞	∞	∞
a	-	-	$4(a)$	∞	$4(a)$	$8(a)$	∞
b	-	-	-	$6(b)$	$4(a)$	$7(b)$	$5(b)$
d	-	-	-	$6(b)$	-	$6(d)$	$5(b)$
f	-	-	-	$6(b)$	-	$6(d)$	-
c	-	-	-	-	-	$6(d)$	-
e	-	-	-	-	-	-	-

Figure 2: The first column shows the nodes chosen at each iteration. For each node we have a column that displays the **d (forward)** values, until they stop changing.

as **true**. This is a point worth pondering, since it really explains why the algorithm works. For example, when all nodes have **Visited** set to **false**, then there is only a valid path to r . After we process r and set **Visited**[r] to **true**, then the **d** values for both a and b are correct since they are set to 1 and 5 respectively, since we are not allowed to go through any other vertices to get to either a or b . After we set **Visited**[a] as **true** we are allowed to cut through node a , and thus we set **d** values for both d and e , and also update the **d** value of b .

We illustrate the algorithm by an example using r as the root. In the first column we show the node chosen in each iteration, and the **d** values of all nodes that have **Visited** set to **false** after processing the chosen node. In () we show the neighbor to which packets should be forwarded if the destination is r .

We now discuss *why* the algorithm works. Among all the nodes that have their **Visited** value as **false** let x be the node with the minimum **d** value. We would like to observe that any path that comes to x from r must have length at least **d**[x]. Lets say $P(r, x)$ is a shortest path to x in the graph. If the path cuts through only nodes that have **Visited** set to **true** then we have already “captured” this path since the **d**[x] value denotes the length of the shortest path cutting through nodes that have **Visited** set to **true**. Suppose the path $P(x, r)$ cuts through nodes that have **Visited** set to **false**. Let y be the first such node as we traverse the path from r to x . Since **d**[y] \geq **d**[x], it must be that any path $P(r, x)$ that goes through y is at least as long as **d**[x].

r=Root

```

N=Numnodes
G=Array.new

N.times{|i| G[i] = Array.new}
# Create graph here
# We also create a weight array called W, so that W[i][j] is the weight of the
# edge connecting node i to node G[i][j]
W=Array.new

def minDvalue(g,d,v)
  min=nil
  minvalue= \mbox{\$\\infty\$}
  g.length.times{|i|
    if not (v[i]) then
      if d[i]< minvalue then
        minvalue=d[i]; min=i
      end
    end
  }
end

end

Visited =Array.new
G.length.times{|i| Visited[i] = false}
D=Array.new
# Let M be a large number, greater than any possible shortest path length
G.length.times{|i| D[i]= M}

D[r]=0
numVisited =0
forward[r]=nil

while (numVisited < N)
  p = minDvalue(G, D, Visited)
  Visited[p] = true
  numVisited = numVisited+1
  G[p].length.times{|j|
    q = G[p][j]
    If (Visited[q]==false) and (D[q] > D[p]+W[p,j]) then
      D[q] = D[p]+ W(p,j)
      forward[q]=p
    end
  }
end
end

```