Dijkstra's Algorithm

Given a graph $\mathcal{G} = (V, E)$, assume $\mathcal{G}$ has the following properties:

1. for each edge $e \in E$ connecting vertices u and v, we assign an associated positive number $d(u, v)$ referred to as the *length* of edge e - but may represent any measurable quantity.

2. there is an *initial vertex* v_0.

3. the length of a path from a vertex v to the initial vertex v_0 is the sum of the lengths of the path's constituent edges.

4. to each vertex $v \in V$ an associated number $\lambda(v)$ is assigned.
   It is initially defined such that $\lambda(v_0) = 0$ and for all other vertices $\lambda(v) = \infty$

1 Description of Algorithm

The goal of Dijkstra's algorithm is to construct for each vertex v a shortest path from v to v_0. Dijkstra's algorithm is a recursive algorithm which at each stage constructs a set S of *visited vertices*. A visited vertex $v \in S$ has the property that among all paths from v to v_0 containing only vertices in S.

1. there is a shortest such path whose length is then recorded as $\lambda(v)$.

2. The path itself is recorded as a sequence of vertices.

The set $U = V \setminus S$ is referred to as the set of *unvisited vertices*. The algorithm recursively adds points from U to S.

At each iteration of the algorithm a current vertex $v_c \in U$ is chosen. Initially the current vertex is $v_c = v_0$. At each iteration three things happen.

1. For each vertex $v \in U$ that is adjacent to v_c, the length $\lambda(v)$ of the shortest path back to v_0 is newly calculated and recorded. The path itself is recorded as a sequence of vertices.

   Note that : (1) $\lambda(v)$ may have already been calculated in some previous iteration and (2) $\lambda(v_c)$ will have been calculated in the previous iteration as in step 2 below. We then now calculate

   $\lambda(v) = \min\{\lambda(v), \lambda(v_c) + d(v_c, v)\}$

2. A new current vertex $w \in U$ is chosen so that for every $u \in U$,
   $\lambda(w) \leq \lambda(u)$.
   Note that : for those unvisited vertices u that have not been adjacent to some current vertex, we have $\lambda(v) = \infty$ - so we may restrict attention to vertices u for which $\lambda(u)$ has already been calculated.

3. Preparing for the next iteration:

   (a) move v_c in set of visited vertices - renaming S as $S \cup v_c$

   (b) set $v_c = w$.

The algorithm terminates when $S = V$.
Alternatively, if the shortest path to a specified vertex is desired, t he algorithm may terminate when this vertex becomes *visited*.

2 Complexity

Assume $\forall v \in V,\ deg(v) \leq k$ for some constant k.

Assume $V = \{v_0, v_1, \cdots, v_n\}$ has $n + 1$ elements.

Each of the vertices will in some iteration become the *current* vertex v_c. For such an iteration with *current vertex* v_c, let $\tau(v_c)$ be the time to calculate $\lambda(v)$ for each vertex v adjacent to v_c. Choose a constant r such that for each current vertex v_c, $r \geq \tau(v_c)$.

The time to complete this stage of the algorithm for each current vertex is thus less than r times the number of vertices - namely rn.

Next, in an arbitrary iteration there is a search among all unvisited vertices for the shortest path back to v_0. That is, a search for

$min\{\lambda(v) : v \in U\}$ - most often this is search can be restricted to those vertices $v \in U$ that are adjacent to v_c

If at an arbitrary iteration, U has m elements, we know this search is of complexity $O(m)$. Thus there is a constant s such that the time to find the shortest path is is less than sm.

Thus the time to complete this iteration is less than $kr + sm$.

Then for all iterations the time is less than

$(rk + sn) + (rk + s(n-1) + \cdots + (rk + s) = nrk + s(n + (n-1) + \cdots + 1) =$

$nrk + s\frac{n(n+1)}{2} = \frac{s}{2}n^2 - \frac{s}{2}n + nrk =$

$\frac{s}{2}n^2 + n(rk - \frac{s}{2}) \leq n^2(\frac{s}{2} + (rk - \frac{s}{2})) = rkn^2.$

This shows that Djikstra's algorithm has complexity at worst $O(n^2)$. Careful coding allows improvements.

Example: Consider the following diagram. The vertex a is considered to be the initial vertex and with $\lambda(a) = 0$. To apply the algorithm we set up a table with the vertices labelling columns and our progressive choices of current vertices labelling the rows.

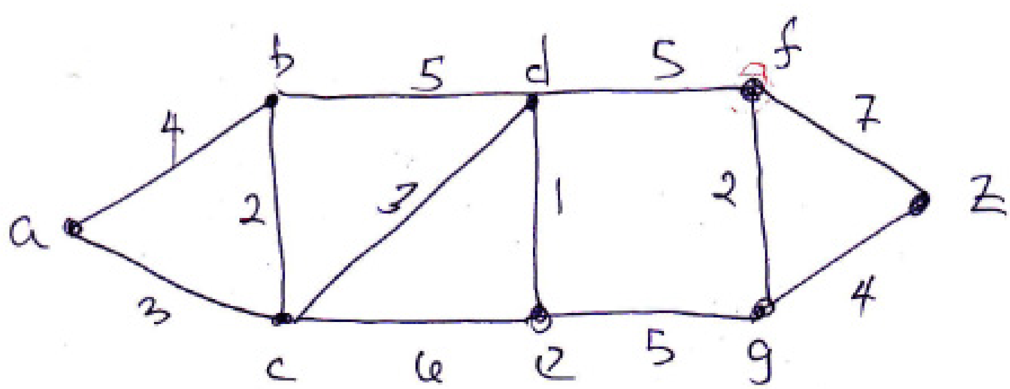
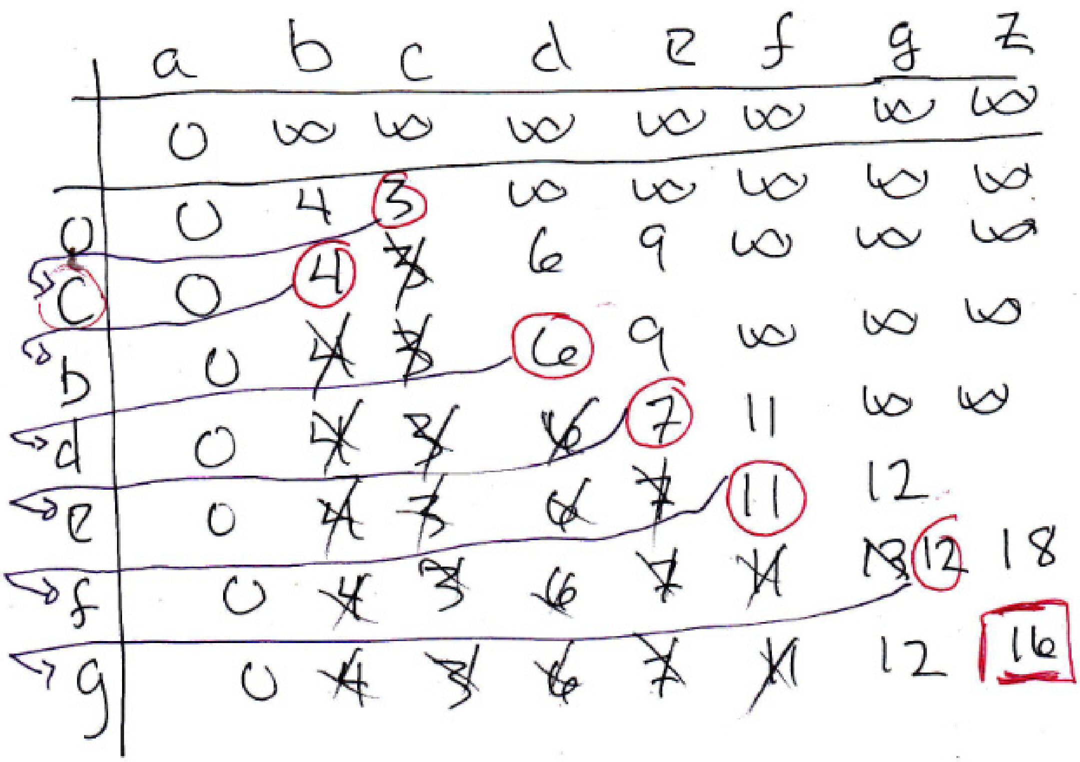


Figure 1: Dijkstra diagram

The first row of the table will indicate the the starting position of the algorithm i n which the various λ values are indicated Note that for each vertex other than a we have a value of ∞. Looking at the first row, we search for the vertex with the least λ. This vertex will become the first *current* vertex. The vertex is of course is a. We place a in the second row in the *current* vertex column. We can now construct the second row of the table by examining the vertices adjacent to a, these are b and c. We record the lengths of the paths back to a, as 4 for b and 3 for c. We now consider a to be visited vertex, setting $S = \{a\}$.

Next we look for the vertex adjacent to a whose distance back to a is least. This vertex is c. The new *current* vertex is thus chosen equal to c. The third row is then labeled at the left with the letter c. We now go to the next iteration looking at those vertices not in S that are adjacent to c. They are b, d and e, and for each we calculate the distance of the shortest path back to a and record the results in the table with 4 in the column labeled b, 6 in the column labeled d and 9 in the column labeled e. We now consider c as visited and add c to S - setting $S = \{a, c\}$. Next we calculate the shortest distance back to a from each of b, d, e and record the results in the fourth row. The vertex c is now *visited* and is placed in the set S - so $S = \{a, c\}$.

In the 3rd row we now look for the vertex with the shortest path back to a - this vertex is b - so b becomes the new *current vertex* and is laced in the first column to label the fourth row. The process now continues in the same way. T he full table is shown below.



	a	b	c	d	e	f	g	z
	0	∞	∞	∞	∞	∞	∞	∞
a	0	4	3	∞	∞	∞	∞	∞
c	0	4	3	6	9	∞	∞	∞
b	0	4	3	6	9	∞	∞	∞
d	0	4	3	6	7	11	∞	∞
e	0	4	3	6	7	11	12	
f	0	4	3	6	7	11	13 12	18
g	0	4	3	6	7	11	12	16

Figure 2: Dijkstra TAble