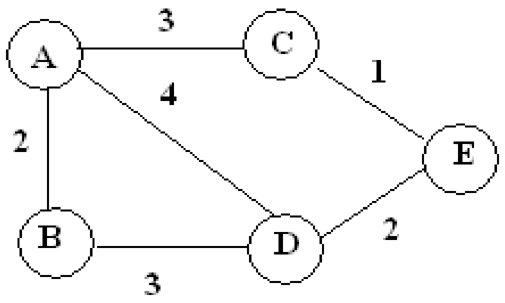
Dijkstar's Shortest Path Algorithm

1. Create a distance list, a previous vertex list, a visited list, and a current vertex.
2. All the values in the distance list are set to infinity except the starting vertex which is set to zero.
3. All values in visited list are set to false.
4. All values in the previous list are set to a special value signifying that they are undefined, such as null.
5. Current vertex is set as the starting vertex.
6. Mark the current vertex as visited.
7. Update distance and previous lists based on those vertices which can be immediately reached from the current vertex.
8. Update the current vertex to the unvisited vertex that can be reached by the shortest path from the starting vertex.
9. repeat (from step 6) until all nodes are visited.



To find shortest path: A→E: A-C-E

1. We begin by marking the node A as permanent.
2. Then we examine in turn each of the nodes adjacent to A, the working node now.(B,C,D) and we re label in bracket with the distance from A to this node.
3. When we re-label a node we also label it with the node from which the probe was made so as to use to re construct the path later.
4. Having seen all the nodes adjacent to A, we examine all the tentatively labeled nodes in the whole graph and make the one with the smallest label permanent, and this node becomes the working node now.
5. Now start with node found in step 4 and examine all the nodes adjacent to it
6. If the sum of the label on B and the distance from B to the node being considered is less than the label on that node, we have a shorter path, so the node is relabeled.
7. Go to step 4 again

8. An Explanation to the Algorithm

We see the graph as a weighted graph. Our aim is to find the shortest path between two nodes say *s* and *t,* which will be input the programe that we build. The input to the function is the *weight[][]* array, such that it defines for *weight[i][j],* is the weight of the edges from node *i* to node *j* . If there is no arc from *i, to j* then *weight[i][j],* is set to an very large value [infinity] to indicate the impossibility of going from *i, to j.*

Here, the array *distance[i]*, keeps the value of the shortest path known thus far from the starting node, *s* to *i.* initially, *distance[s]*=0, and *distance[i]=infinity* for all *i !=s.* Another array *perm* contains all the nodes whose minimal distance from *s*, is known- that is, those nodes hose distance values is permanent and will not change. If a node *i*, is a member of the array *perm[]*, then *distance[i]* , is the minimum distance from *s* to *i.* At the beginning of the programe, the only member of *perm[]*, is *s.* Once *t,* becomes a member of *perm[]*, then *distance[t],* is the shortest distance from *s* to *t.*

Current: the node that has been added to *perm[]* most recently. Initially, *current=s.*

Whenever a node *current* is added to *perm[]*, *distance* must be recalculated for all successors of current. Then for every successor *i,* of *current,* if *distance[current]+weight(current ,i)* is less than *distance[i],* then the distance from *s* to *i* through *current* is smaller than any other distance from *s* to *i,* so far calculated. Hence *distance[i],* must be replaced with this new smaller value.

Once the distance has been recalculated for every successor of *current*, *distance[j]*, for any j, will represent the shortest path from *s* to *j,* that includes only members of *perm[], (of course except j itself).* That is, for the node *k,* which is not in *perm[]*, for which *distance[k],* is the smallest, there is no path from *s*, to *k* whose length is shorter than *distance[k]*. So as now, *distance[k],* is the shortest distance to k that includes only nodes in *perm[]*. And any path to k that includes a node *nd* as its first node not in *perm[]* must be longer, since *distance[nd]* is greater than *distance[k]*. Thus k can be added to *perm[]* current can be reset to k and process is repeated.

Data
Structures Using C and C++ by A Tanenbaum. p. 527.

The Algorithm

/ Input to the programe is the weight matrix weight[MAXNODES][MAXNODES], the starting node s, the ending node, t, and the output is the distance and we have the path from s to t in the array preced[].The weight matrix is to be created, where $W_{ij} = W_{ji}$ */*

```
Define MAX 10000; Define MAXNODES; Define member 1;Define nomember 0;
Int shortestpath(int weight[][], int s, int t)
{
        Int distance[MAXNODES], perm[MAXNODES];
        Int current, I, k, dc;
        Int smalldist, newdist;
/* initialization of the distance and perm arrays*/
        For(i=0; i<i<MAXNODES; i++)
        {
                Perm[i]=nomember;
                Distance[i]=    MAX;
        }
        Perm[s]=member;
        Distance[s]=0;
        Current=s;
        While(current !=t)
        {
                Smalldist=MAX;
                Dc=distance[current]

                For(i=0;i<MAXNODES;i++)
                        If(perm[i]==nomember)
                        {
                                Newdist=dc+weight[current][i];
                                If(newdist<distance[i])
                                { /*distance from s to i through current is smaller than distance[i]*/
                                        Distance[i]=newdist;
                                        Precede[i]=current;
                                }
                                If(distance[i]<smalldist)
                                {
                                        Smalldist=distance[i];
                                        k=I;
                                }
                        }
        Current=k;
        Perm[current]=member;
        }
Return(distance[t]);
}
```