Dijkastra Algorithm

```
SP-Dijkstra( )
    n = number of nodes in the graph;
    for i = 1 to n
      cost[v_i] = w(v_1, v_i) ;
      S = { v_1 } ;
    for j = 2 to n {
      find the smallest cost[v_i] s.t. v_i is not in S;
      include v_i to S;
      for (all nodes v_j not in S) {
       if (cost[v_j] > cost[v_i] + w(v_i, v_j))
              cost[v_j] = cost[v_i] + w(v_i, v_j);
      }
    }
```

In the following code, dist(u) refers to the current alarm clock setting for node u. A value of ∞ means the alarm hasn't so far been set. There is also a special array, prevprev, that holds one crucial piece of information for each node u: the identity of the node immediately before it on the shortest path from s to u. By following these back-pointers, we can easily reconstruct shortest paths, and so this array is a compact summary of all the paths found. A full example of the algorithm's operation, along with the final shortest-path tree, is shown below.

**Input:** Graph $G = (V, E)$, directed or undirected; positive edge lengths $\{l_e : e \in E\}$; vertex $s \in V$

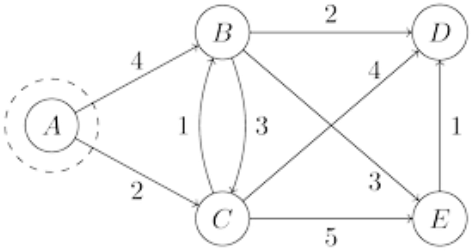
**Output:** For all vertices $u$ reachable from $s$, $dist(u)$ is set to the distance from $s$ to $u$.

**procedure** DIJKSTRA($G$, $l$, $s$)

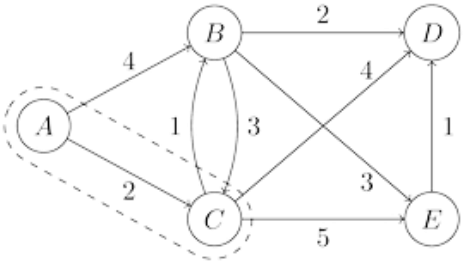
```
for all u ∈ V do
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = MAKEQUEUE(V)                          ▷ using dist-values as keys
while H is not empty do
    u = DELETEMIN(H)
    for all edges (u, v) ∈ E do
        if dist(v) > dist(u) + l(u, v) then
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            DECREASEKEY(H, v)
```

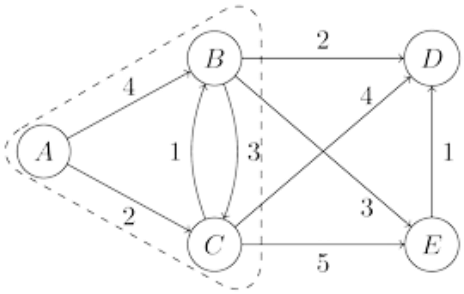
A full example of the algorithm's operation, along with the final shortest-path tree, is shown below.



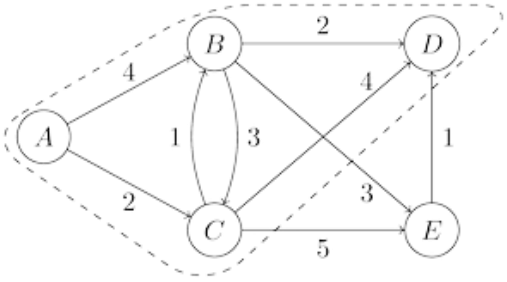

| | |
|---|---|
| $A: 0$ | $D: \infty$ |
| $B: 4$ | $E: \infty$ |
| $C: 2$ | |



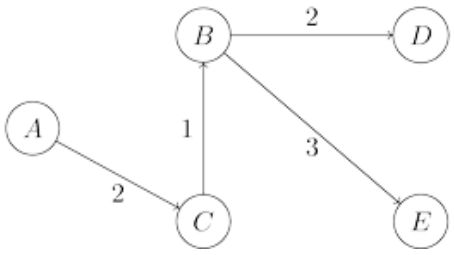

| | |
|---|---|
| $A: 0$ | $D: 6$ |
| $B: 3$ | $E: 7$ |
| $C: 2$ | |




| | |
|---|---|
| $A: 0$ | $D: 5$ |
| $B: 3$ | $E: 6$ |
| $C: 2$ | |

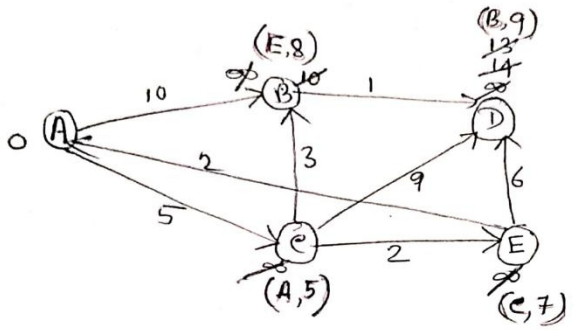



| | |
|---|---|
| $A: 0$ | $D: 5$ |
| $B: 3$ | $E: 6$ |
| $C: 2$ | |




In summary, we can think of Dijkstra's algorithm as just BFS, except it uses a priority queue instead of a regular queue, so as to prioritize nodes in a way that takes edge lengths into account. This viewpoint gives a concrete appreciation of how and why the algorithm works, but there is a more direct, more abstract derivation that doesn't depend upon BFS at all.

Example 2:



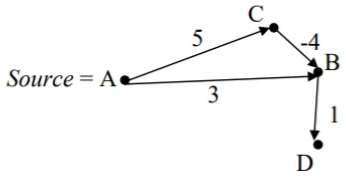

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | [0] | ∞ | ∞ | ∞ | ∞ |
| C | | 10 | [5] | ∞ | ∞ |
| E | | 8 | | 14 | [7] |
| B | | [8] | | 13 | |
| D | | | | [9] | |

Path example : A to E will be A → C → E

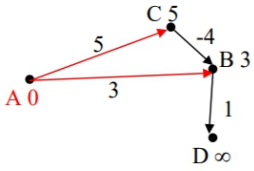
and A to D will be A → C → B → D

Limitation of Dijkstra's algorithm:
This algorithm may not work for negetive edge..

Dijksta's algorithm does not work if the graph has negative weight edges, i.e., it might return incorrect result, as the following example shows:

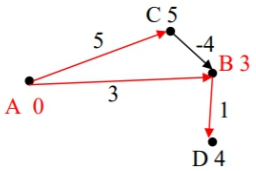



Lets see how the algorithm would run on this graph; first the source A is popped out of the queue and edges (A, C) and (A, B) are relaxed.

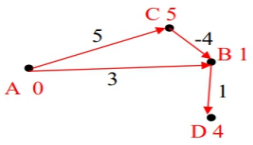



```
Dijkstra(G,s)
for each v ∈ V
    d[v] ← ∞
d[s] ← 0
S ← ∅
Q ← V
while Q ≠ ∅
      u ←ExtractMin(Q)
      S ←S U {u}
      for each v ∈ Adj[u]
          if d[v] > d[u]+w(u,v)
                DecreaseKey(v, d[u]+ w(u,v))
```

Now B is on the top of min-heap queue Q. We now pop B and relax the only edge out of B, i.e., (B, D) giving D distance 3+1 = 4:




Now D is on the top of the min-heap, and we remove D. There are no edges going out of D to relax. The only vertex now left in the queue is C. We remove C and relax the edge (C, B). Since 5 - 4 = 1 < 3, distance to B must be updated to 1:




The algorithm now terminates because the queue is empty. However, the path A→C→B→D has total length 5 + (-4) + 1 = 2 < 4; thus the length of the shortest path from the source A to D was not correctly evaluated.