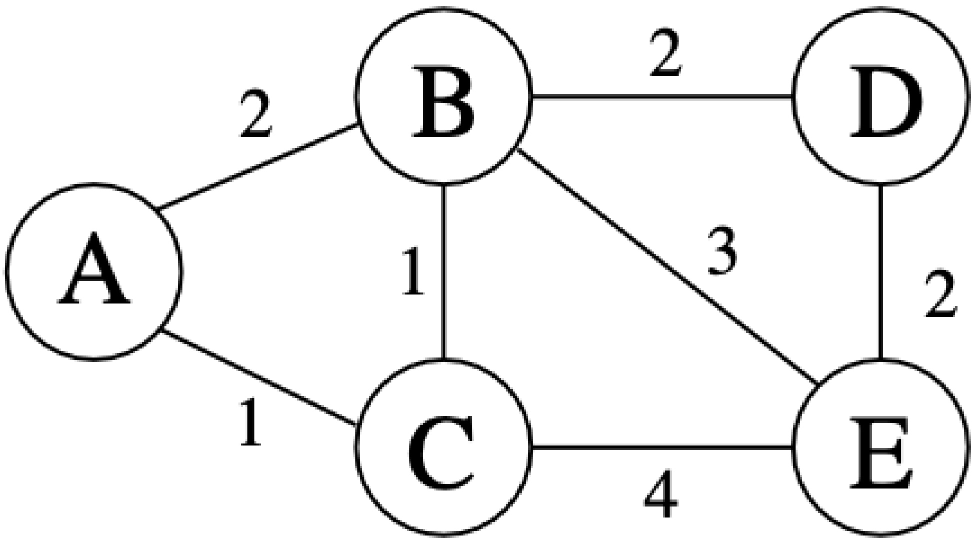
Paths in Graphs

Where rather than just finding paths, we want shortest paths

Figure 4.3 Breadth-first search.

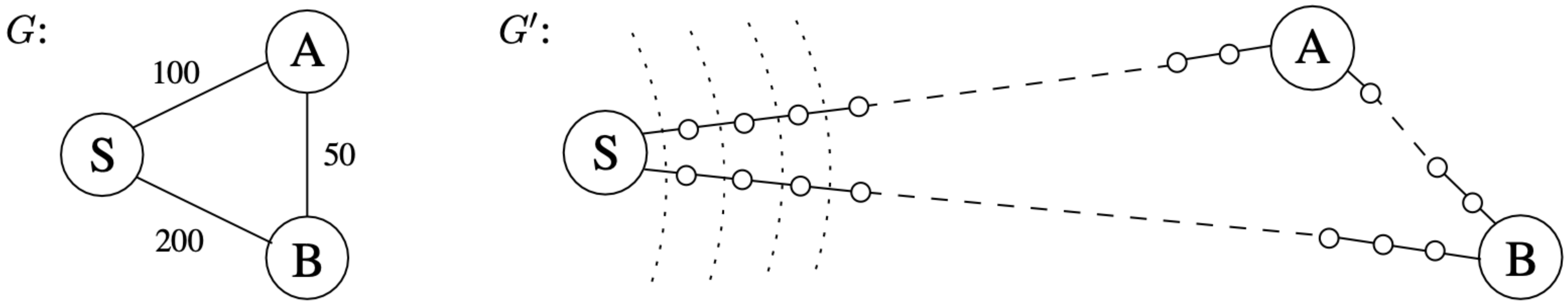
```
procedure bfs(G, s)
Input:    Graph G = (V, E), directed or undirected; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞

dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
    u = eject(Q)
    for all edges (u, v) ∈ E:
        if dist(v) = ∞:
            inject(Q, v)
            dist(v) = dist(u) + 1
```

`eject` removes from front of queue, `inject` adds to back of queue



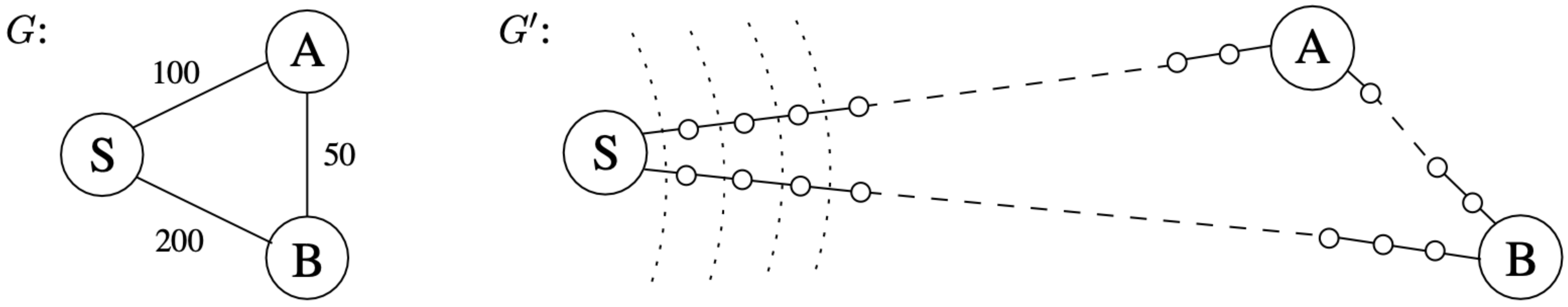
Can we use DFS on this? Why or why not?

Figure 4.7 BFS on G' is mostly uneventful. The dotted lines show some early "wavefronts."



Alarm clocks?

Figure 4.7 BFS on G' is mostly uneventful. The dotted lines show some early "wavefronts."



The following "alarm clock algorithm" faithfully simulates the execution of BFS on G'.

- Set an alarm clock for node s at time 0.
- Repeat until there are no more alarms:

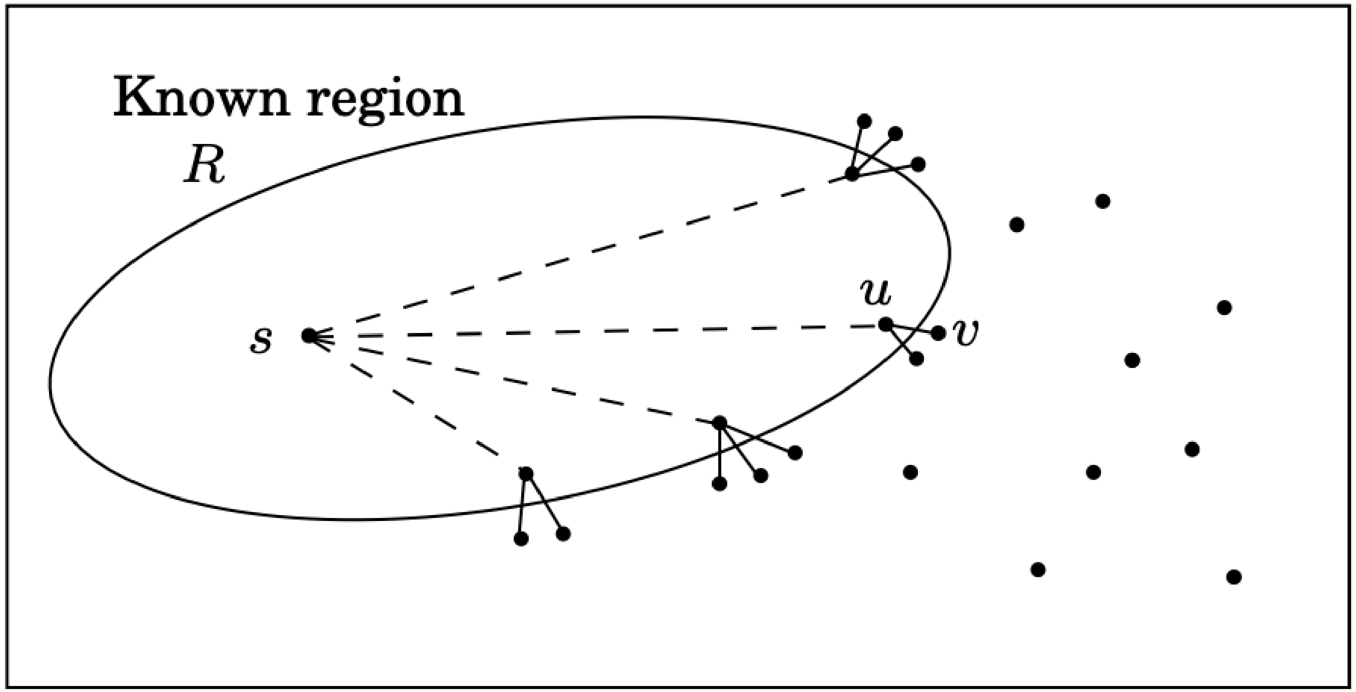
  Say the next alarm goes off at time T, for node u. Then:

  - The distance from s to u is T.
  - For each neighbor v of u in G:
    - If there is no alarm yet for v, set one for time $T + l(u, v)$.
    - If v's alarm is set for later than $T + l(u, v)$, then reset it to this earlier time.

Figure 4.8 Dijkstra's shortest-path algorithm.

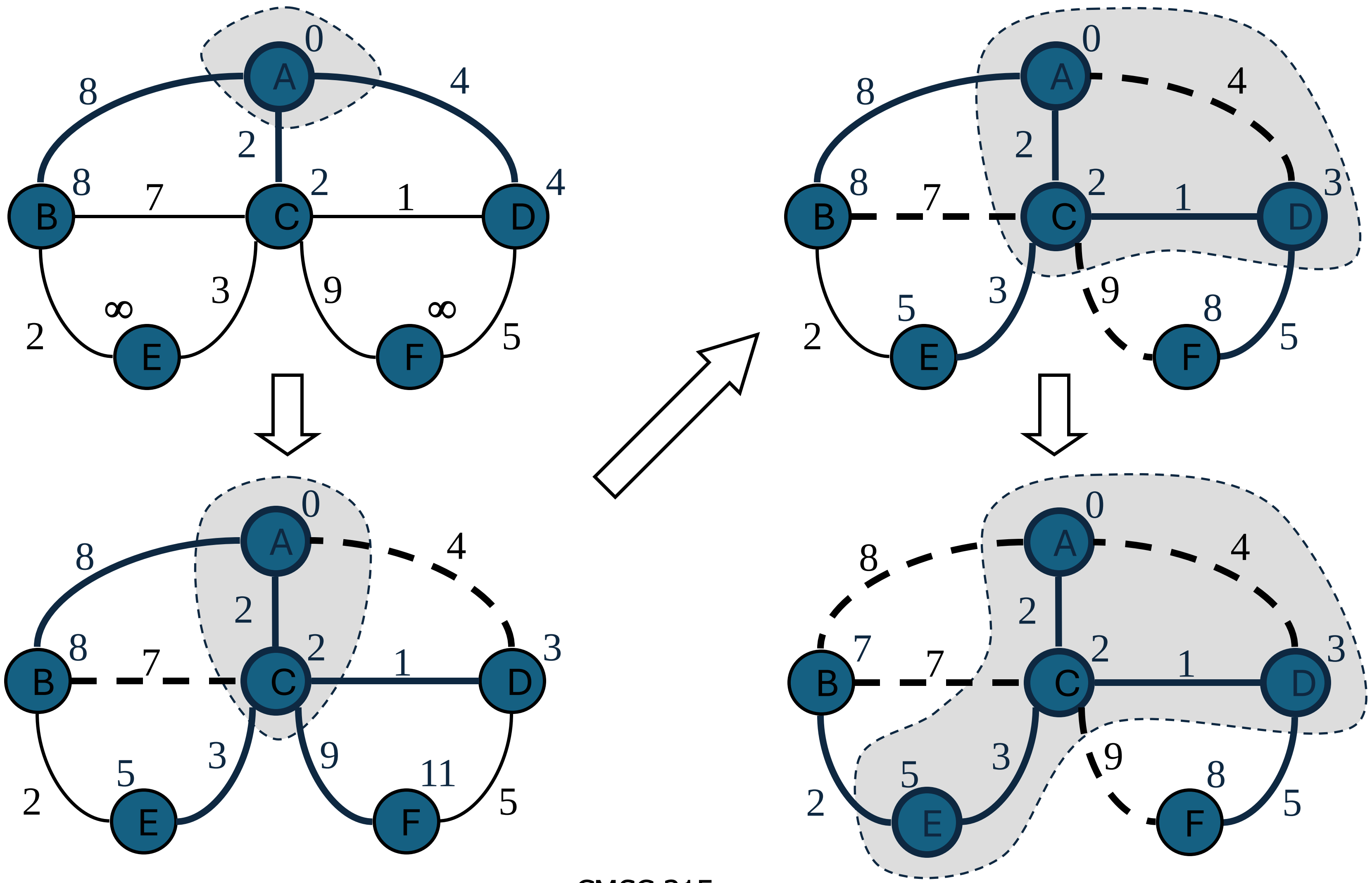
```
procedure dijkstra(G,l,s)
Input:    Graph G = (V,E), directed or undirected;
          positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

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

H = makequeue(V)  (using dist-values as keys)
while H is not empty:
   u = deletemin(H)
   for all edges (u,v) ∈ E:
      if dist(v) > dist(u) + l(u,v):
         dist(v) = dist(u) + l(u,v)
         prev(v) = u
         decreasekey(H,v)
```

Figure 4.10 Single-edge extensions of known shortest paths.

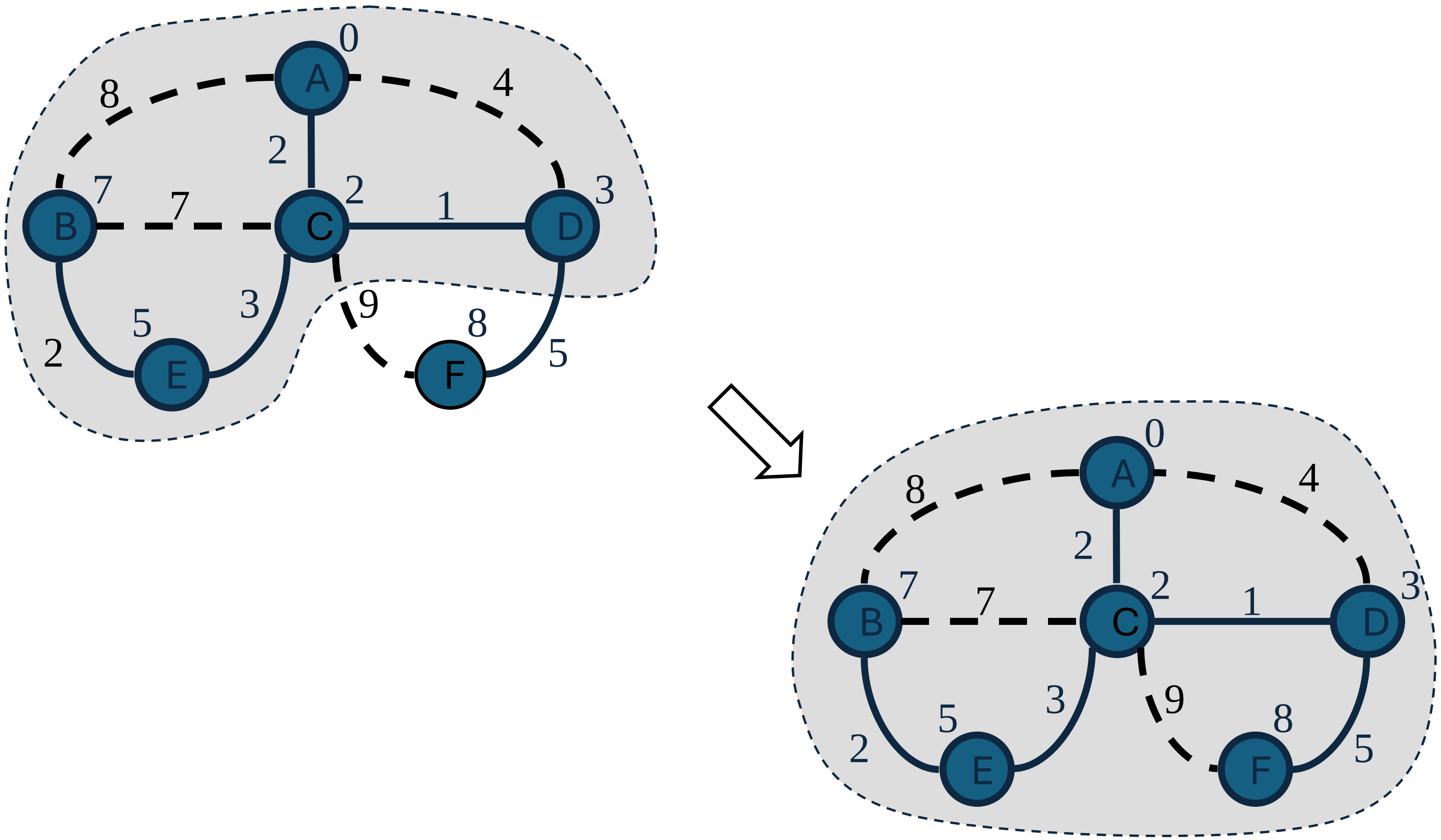


```
Initialize dist(s) to 0, other dist(·) values to ∞
R = { } (the ''known region'')
while R ≠ V:
    Pick the node v ∉ R with smallest dist(·)
    Add v to R
    for all edges (v, z) ∈ E:
        if dist(z) > dist(v) + l(v, z):
            dist(z) = dist(v) + l(v, z)
```

Example


0
A
8
4
2
8
2
4
B
7
C
1
D
3
9
∞
∞
2
E
F
5
0
A
8
4
2
8
2
3
B
7
C
1
D
3
9
5
11
2
E
F
5
0
A
8
4
2
8
2
3
B
7
C
1
D
5
3
9
8
2
E
F
5
0
A
8
4
2
7
2
3
B
7
C
1
D
5
3
9
8
2
E
F
5

Example (cont.)



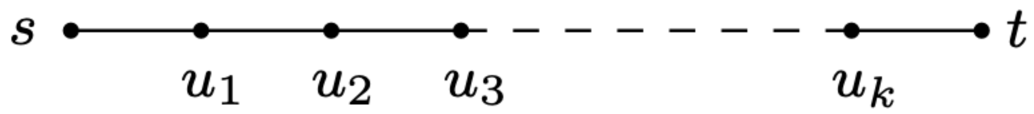
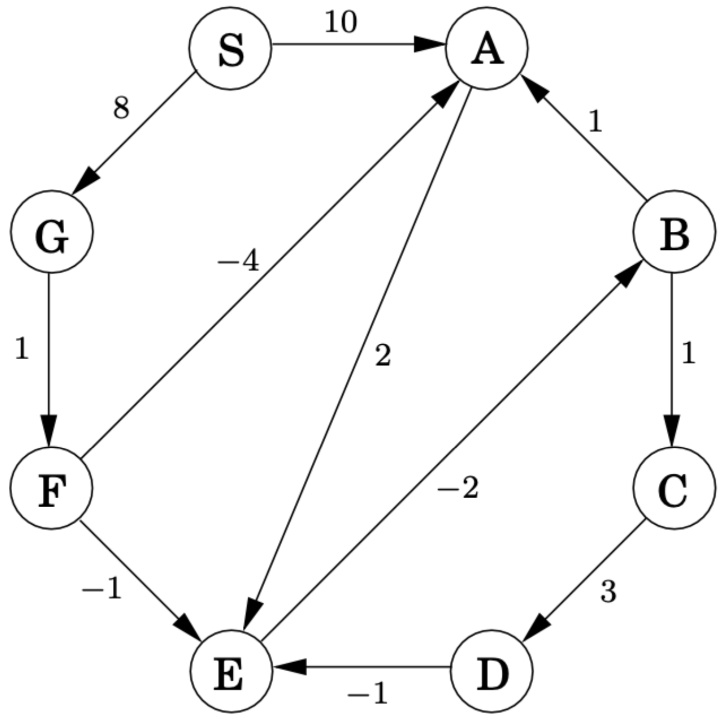

s
u1
u2
u3
uk
t

Figure 4.13 The Bellman-Ford algorithm for single-source shortest paths in general graphs.

```
procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)
```

Figure 4.14 The Bellman-Ford algorithm illustrated on a sample graph.



	Iteration							
Node	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8