Lecture 11

Single-Source Shortest Path

Single Source Shortest Path

In a shortest-paths problem, a weighted, directed graph G = (V, E), with weight function w: E → **R** mapping edges to real-valued weights. The weight w(p) of path p = <v_0, v_1,...., v_k> is the sum of the weights of its constituent edges.

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

We define the shortest-path weight $\delta(u, v)$ from u to v by

$$\delta(u, v) = \begin{cases} min\{w(p) : u \overset{p}{\leadsto} v\}, \text{ if there is a path from u to v.} \\ \infty, \text{ otherwise} \end{cases}$$

A shortest path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$

Single Source Shortest Path Continued

The breadth-first-search algorithm is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge has unit weight. Here, we shall focus on the single-source shortest-paths problem: given a graph G = (V, E), we want to find a shortest path from a given source vertex s ∈ V to each vertex v ∈ V.

Variants of Single Source Shortest Path

The algorithm for the single-source problem can solve many other problems, including the following variants.

1. **Single-Destination Shortest-Paths Problem**: Find a shortest path to a given destination vertex t from each vertex. By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.
2. **Single-Pair Shortest-Path Problem**: Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem also. Moreover, all known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algorithms.

Variants of Single Source Shortest Path Continued

1. **All-pairs Shortest Paths Problem:** Find a shortest path from u to for every pair of vertices u and . Although we can solve this problem by running a singlesource algorithm once from each vertex, we usually can solve it faster. Additionally, its structure is interesting in its own right. Chapter 25 addresses the all-pairs problem in detail.

Optimal Substructure of a Shortest Path

Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it. (The Edmonds-Karp maximum-flow algorithm also relies on this property.) Recall that optimal substructure is one of the key indicators that dynamic programming and the greedy method might apply. Dijkstra’s algorithm, which we shall see is a greedy algorithm, and the Floyd Warshall algorithm, which finds shortest paths between all pairs of vertices is a dynamic-programming algorithm. The following lemma states the optimal-substructure property of shortest paths more precisely.

Optimal Substructure of a Shortest Path Continued

Subpaths of Shortest Paths are Shortest Paths: Given a weighted, directed graph G = (V, E) with weight function $w: E \to \mathbf{R}$, let p = $\langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and, for any i and j such that $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex i to vertex j . Then, p_{ij} is a shortest path from i to j.

Optimal Substructure of a Shortest Path Continued

Proof:

If we decompose path p into, $v_0 \overset{p_{0i}}{\sim} v_i \overset{p_{ij}}{\sim} v_j \overset{p_{jk}}{\sim} v_k$
then we have that, $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$
Now, assume that there is a path
p'_{ij} from v_i to v_j with weight $w(p'_{ij}) < p_{ij}$

Then, $v_0 \overset{p_{0i}}{\sim} v_i \overset{p'_{ij}}{\sim} v_j \overset{p_{jk}}{\sim} v_k$ is a path from v_0 to v_k whose weight,
$w(p_{0i})+w(p'_{ij})+w(p_{jk})$ is less than $w(p)$, which contradicts the assumption that p is a shortest path from v_0 to v_k

Initialization Pseudocode

```
Initialize_Single_Source(G,s):
    for each vertex v $\in$ G.V:
        v.d = $\infty$
        v.$\pi$ = NIL
    s.d = 0
```

After initialization, we have v.π = NIL for all v $\in$ V, s.d = 0 and v.d = ∞ for v $\in$ V - {s}.

Relaxation

```
**Relax**(u,v,w):
    if v.d > u.d + w(u,v):
        v.d = u.d + w(u,v)
        v.$\pi$ = u
```

The process of relaxing an edge (u,v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so updating v.d and v.π. A relaxation step may decrease the value of the shortest path estimate v.d and update v's predecessor attribute v.π.

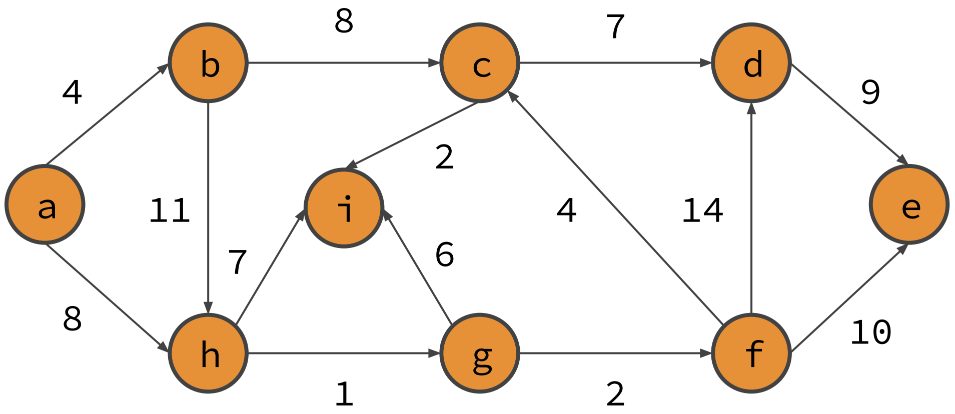
Dijkstra’s Algorithm

Dijkstra’s algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative. We assume that w(u, v) ≥ 0 for each edge (u, v) ∈ E. As we shall see, with a good implementation, the running time of Dijkstra’s algorithm is lower than that of the Bellman-Ford algorithm. Dijkstra’s algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.

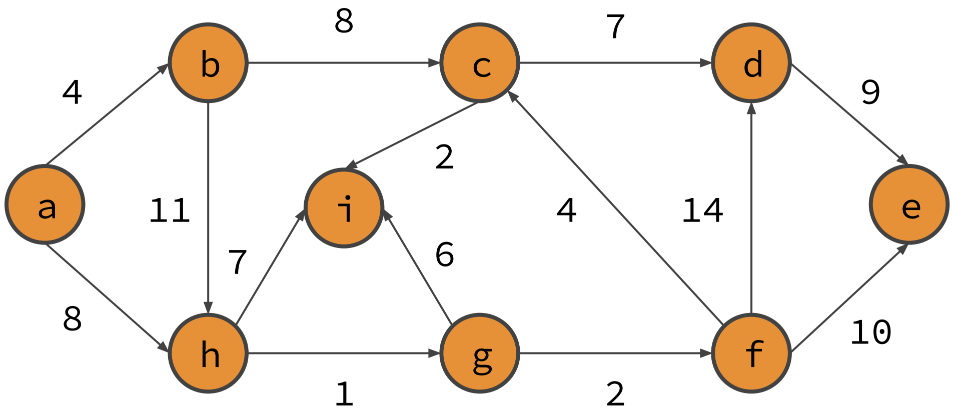
Dijkstra’s Algorithm Pseudocode

```
Dijkstra(G, w, s):
    Initialize_Single_Source(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅:
        u = Extract_Min(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]:
            Relax(u, v, w)
```

Simulation of Dijkstra’s Algorithm

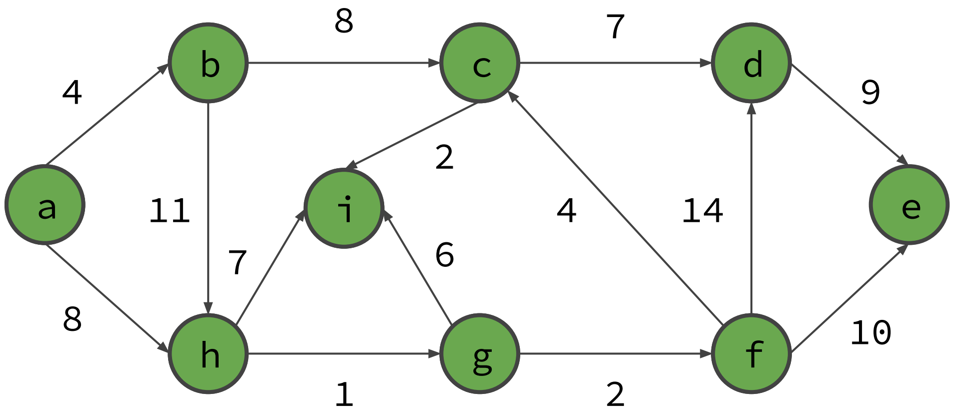


Simulation of Dijkstra’s Algorithm Continued



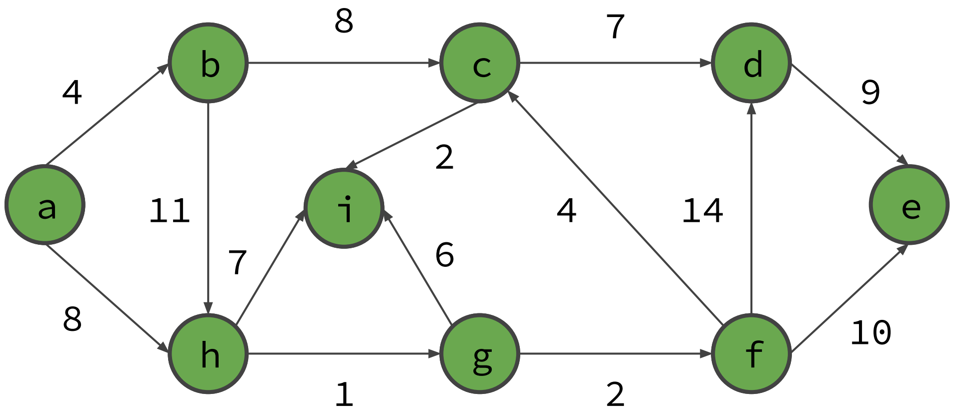
Vertex	Cost	Parent	Visited
a	–	–	–
b	–	–	–
c	–	–	–
d	–	–	–
e	–	–	–
f	–	–	–
g	–	–	–
h	–	–	–
i	–	–	–

Simulation of Dijkstra's Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	-
b	∞	NIL	-
c	∞	NIL	-
d	∞	NIL	-
e	∞	NIL	-
f	∞	NIL	-
g	∞	NIL	-
h	∞	NIL	-
i	∞	NIL	-

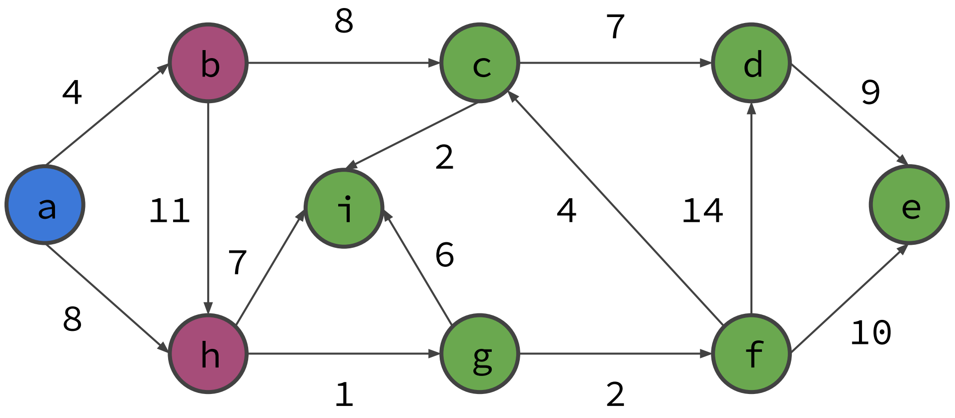
Simulation of Dijkstra’s Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	–
b	∞	NIL	–
c	∞	NIL	–
d	∞	NIL	–
e	∞	NIL	–
f	∞	NIL	–
g	∞	NIL	–
h	∞	NIL	–
i	∞	NIL	–

Min Priority Queue: a b c d e f g h i

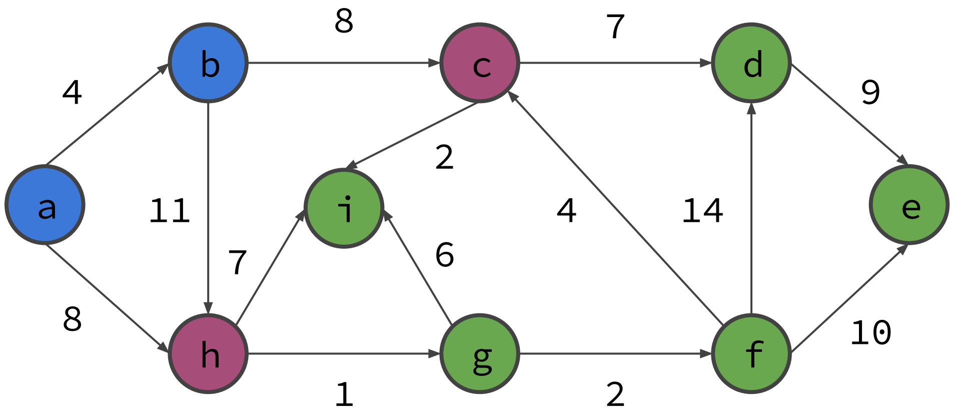
Simulation of Dijkstra's Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	True
c	∞	NIL	-
d	∞	NIL	-
e	∞	NIL	-
f	∞	NIL	-
g	∞	NIL	-
h	8	a	True
i	∞	NIL	-

```
Min Priority Queue: b c d e f g h i
```

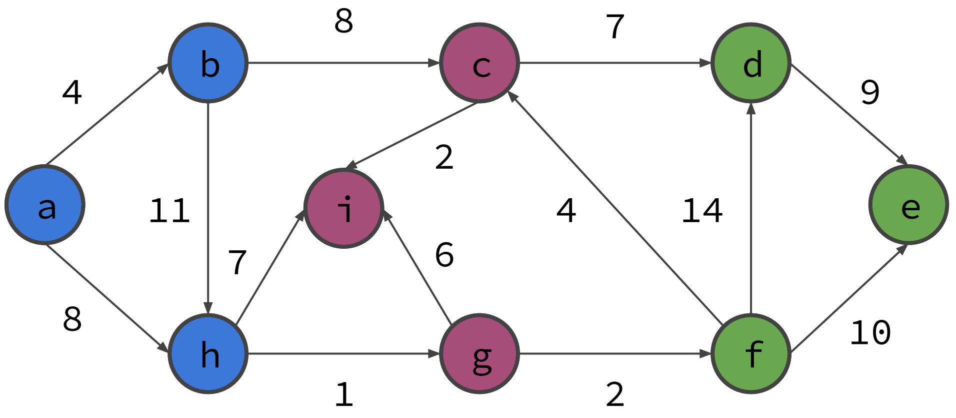
Simulation of Dijkstra’s Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	12	b	True
d	∞	NIL	-
e	∞	NIL	-
f	∞	NIL	-
g	∞	NIL	-
h	8	a	True
i	∞	NIL	-

Min Priority Queue: c d e f g h i

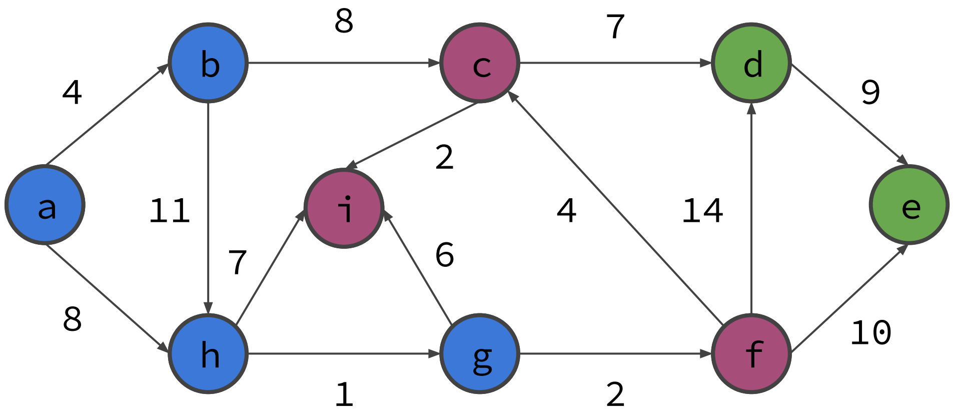
Simulation of Dijkstra’s Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	12	b	True
d	∞	NIL	–
e	∞	NIL	–
f	∞	NIL	–
g	9	h	True
h	8	a	Extracted
i	15	h	True

```
Min Priority Queue: c d e f g i
```

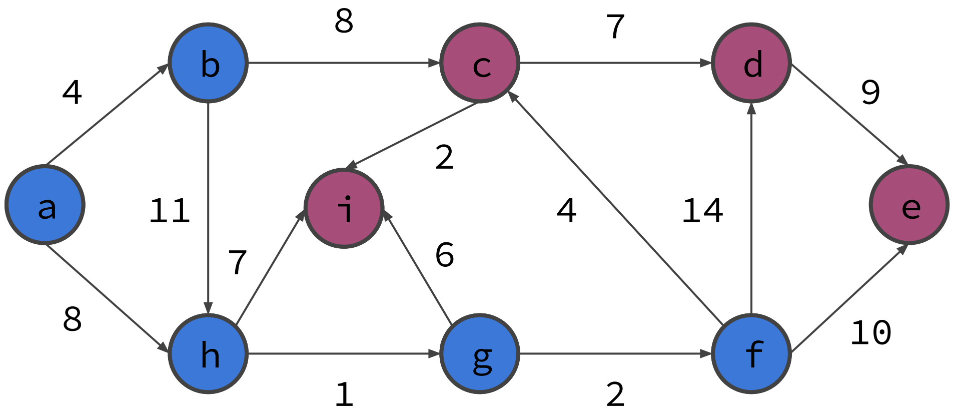
Simulation of Dijkstra’s Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	12	b	True
d	∞	NIL	-
e	∞	NIL	-
f	11	g	True
g	9	h	Extracted
h	8	a	Extracted
i	15	h	True

Min Priority Queue: c d e f i

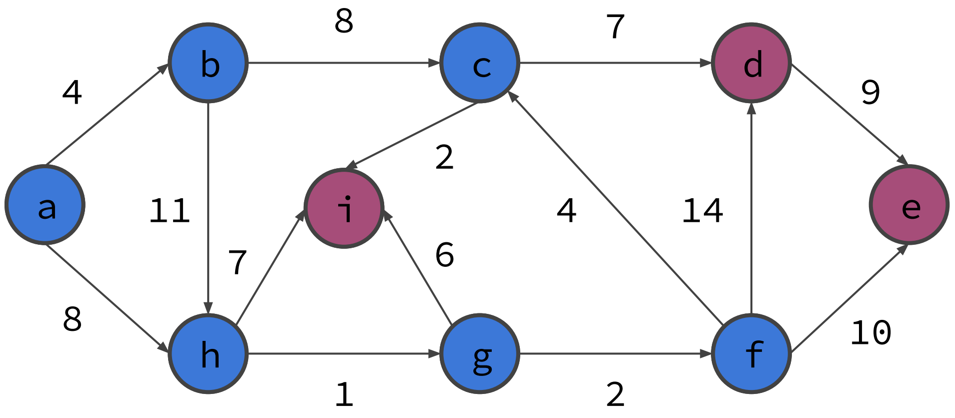
Simulation of Dijkstra’s Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	12	b	True
d	25	f	True
e	21	f	True
f	11	g	Extracted
g	9	h	Extracted
h	8	a	Extracted
i	15	h	True

Min Priority Queue: c d e i

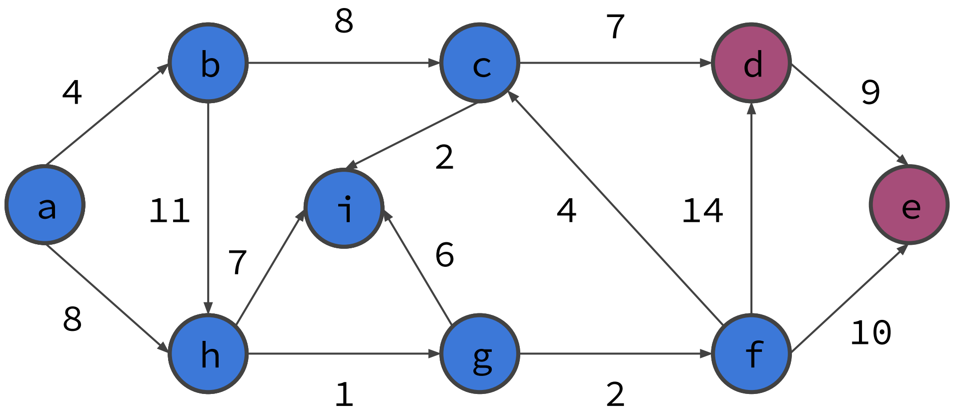
Simulation of Dijkstra’s Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	12	b	Extracted
d	19	c	True
e	21	f	True
f	11	g	Extracted
g	9	h	Extracted
h	8	a	Extracted
i	14	c	True

Min Priority Queue: d e i

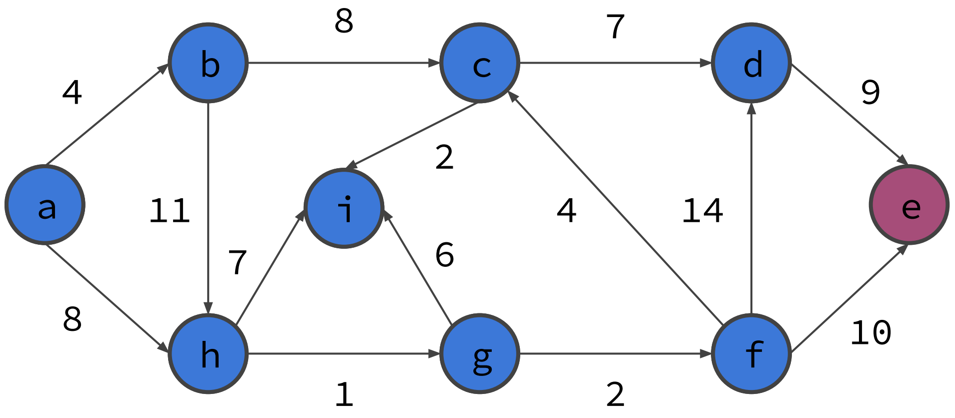
Simulation of Dijkstra's Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	12	b	Extracted
d	19	c	True
e	21	f	True
f	11	g	Extracted
g	9	h	Extracted
h	8	a	Extracted
i	14	c	Extracted

```
Min Priority Queue: d e
```

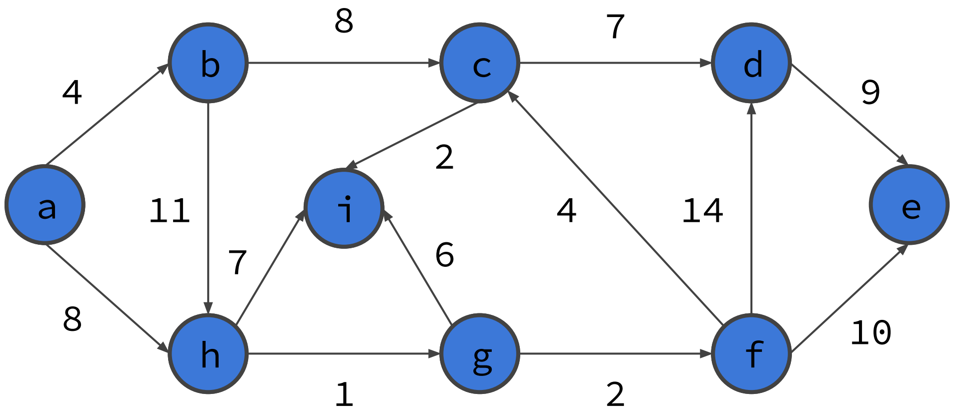
Simulation of Dijkstra’s Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	12	b	Extracted
d	19	c	Extracted
e	21	f	True
f	11	g	Extracted
g	9	h	Extracted
h	8	a	Extracted
i	14	c	Extracted

Min Priority Queue: e

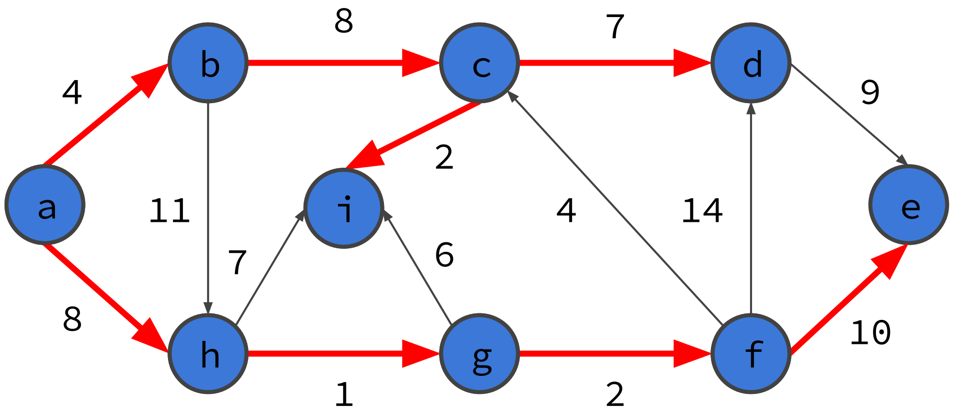
Simulation of Dijkstra’s Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	12	b	Extracted
d	19	c	Extracted
e	21	f	Extracted
f	11	g	Extracted
g	9	h	Extracted
h	8	a	Extracted
i	14	c	Extracted

Min Priority Queue:

Simulation of Dijkstra’s Algorithm Continued



Vertex	Cost	Parent	Visited
a	0	NIL	Extracted
b	4	a	Extracted
c	12	b	Extracted
d	19	c	Extracted
e	21	f	Extracted
f	11	g	Extracted
g	9	h	Extracted
h	8	a	Extracted
i	14	c	Extracted

Min Priority Queue:

Complexity Analysis of Dijkstra’s Algorithm

Dijkstra’s algorithm maintains the min-priority queue Q by calling three priority-queue operations: Insert, Extract_Min, and Decrease_Key. The algorithm calls both Insert and Extract_Min once per vertex. Because each vertex u ∈ V is added to set S exactly once, each edge in the adjacency list Adj[u] is examined in the for loop exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is |E|, this for loop iterates a total of |E| times, and thus the algorithm calls Decrease_Key at most |E| times overall. (Observe once again that we are using aggregate analysis.)

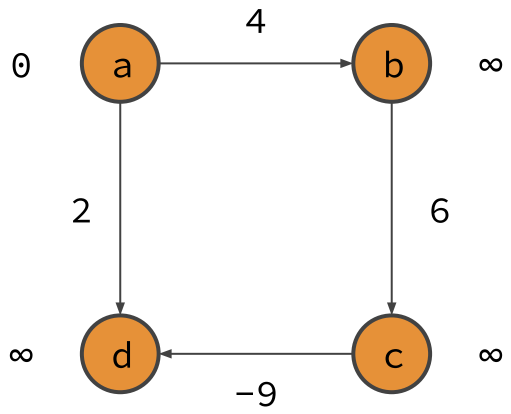
Complexity Analysis of Dijkstra’s Algorithm Continued

The running time of Dijkstra’s algorithm depends on how we implement the min-priority queue. Consider first the case in which we maintain the min-priority queue by taking advantage of the vertices being numbered 1 to |V|. We simply store v.d in the vth entry of an array. Each Insert and Decrease_Key operation takes O(lgV) time, and each Extract_Min operation takes O(lgV) time. The time to build the binary min-heap is O(V). The total running time is therefore O((V + E)lgV), which is O(ElgV) if all vertices are reachable from the source.

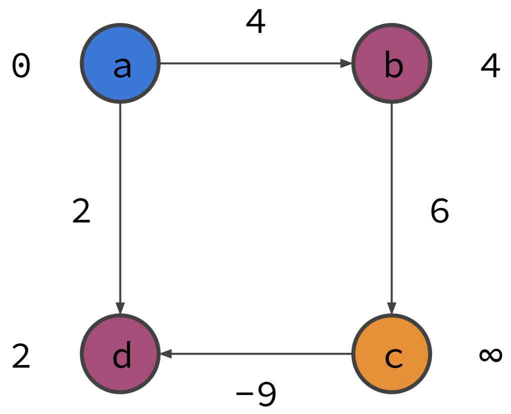
Negative Weight Edges

Some instances of the single-source shortest-paths problem may include edges whose weights are negative. If the graph $G = (V, E)$ contains no negative weight cycles reachable from the source s, then for all $v \in V$, the shortest-path weight $\delta(s, v)$ remains well defined, even if it has a negative value. If the graph contains a negative-weight cycle reachable from s, however, shortest-path weights are not well defined. No path from s to a vertex on the cycle can be a shortest path—we can always find a path with lower weight by following the proposed “shortest” path and then traversing the negative-weight cycle. If there is a negative weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$

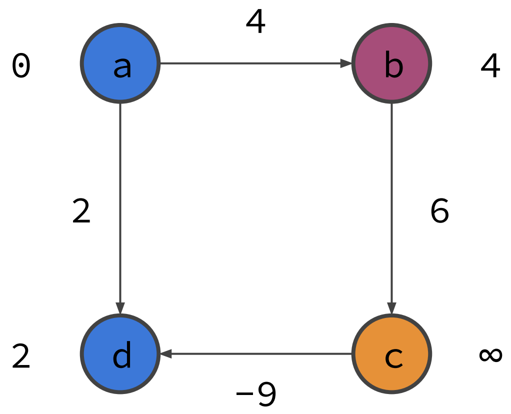
Dijkstra’s Algorithm does not work on Negative Weighted Graph



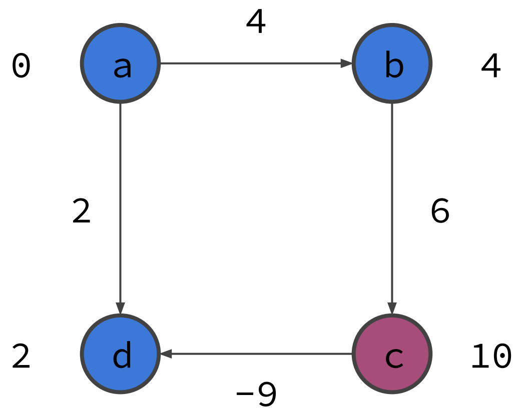
Dijkstra’s Algorithm does not work on Negative Weighted Graph Continued



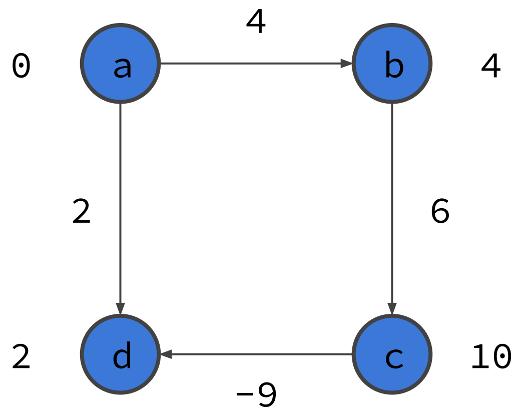
Dijkstra's Algorithm does not work on Negative Weighted Graph Continued



Dijkstra’s Algorithm does not work on Negative Weighted Graph Continued

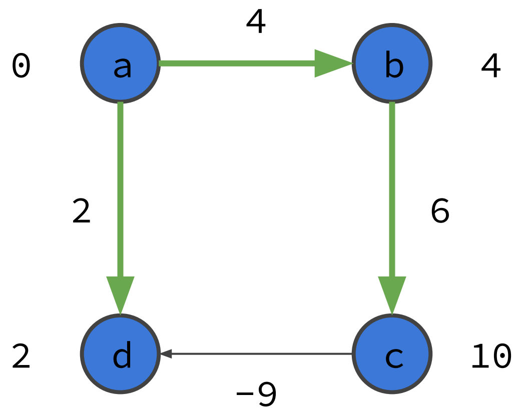


Dijkstra’s Algorithm does not work on Negative Weighted Graph Continued



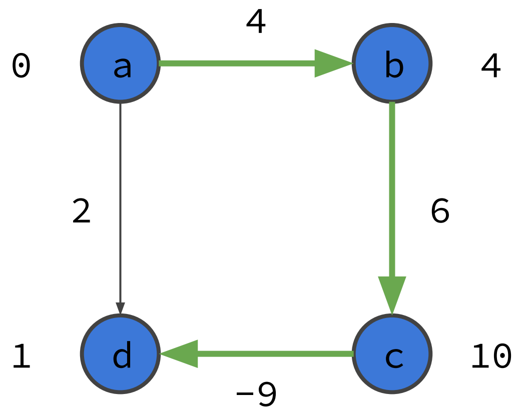
Dijkstra’s Algorithm does not work on Negative Weighted Graph Continued

---



Dijkstra’s Algorithm provides this path. However, this is not the shortest path.

Dijkstra’s Algorithm does not work on Negative Weighted Graph Continued



This is the actual shortest path.

Cycles

Can a shortest path contain a cycle?

As we have just seen, it cannot contain a negative-weight cycle. Nor can it contain a positive-weight cycle, since removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight. That is, if $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a path and $c = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ is a positive-weight cycle on this path (so that $v_i = v_j$ and $w(c) > 0$), then the path $p' = \langle v_0, v_1, \ldots, v_i, v_{j+1}, v_{j+2} \ldots, v_k \rangle$ has weight $w(p') = w(p) - w(c) < w(p)$, and so p cannot be a shortest path from v_0 to v_k. That leaves only 0-weight cycles. We can remove a 0-weight cycle from any path to produce another path whose weight is the same.

Cycles Continued

Thus, if there is a shortest path from a source vertex s to a destination vertex v that contains a 0-weight cycle, then there is another shortest path from s to v without this cycle. As long as a shortest path has 0-weight cycles, we can repeatedly remove these cycles from the path until we have a shortest path that is cycle-free. Therefore, without loss of generality we can assume that when we are finding shortest paths, they have no cycles, i.e., they are simple paths. Since any acyclic path in a graph G = (V, E) contains at most jV j distinct vertices, it also contains at most |V| - 1 edges. Thus, we can restrict our attention to shortest paths of at most |V| - 1 edges.

Bellman-Ford Algorithm

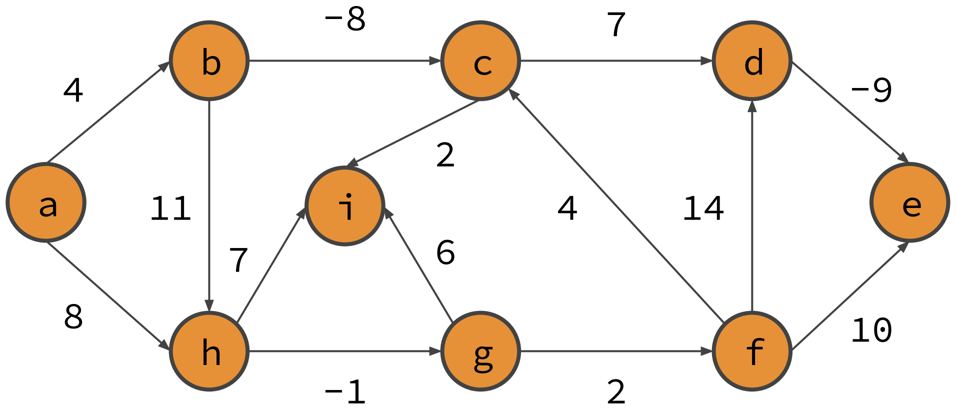
---

1. The Bellman-Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph.
2. The algorithm was first proposed by Alfonso Shimbel (1955), but is instead named after Richard Bellman and Lester Ford Jr., who published it in 1958 and 1956, respectively.
3. Edward F. Moore also published the same algorithm in 1957, and for this reason it is also sometimes called the Bellman-Ford-Moore algorithm.[1]
4. The Bellman-Ford algorithm is a way to find single source shortest paths in a graph with negative edge weights (but no negative cycles).
5.

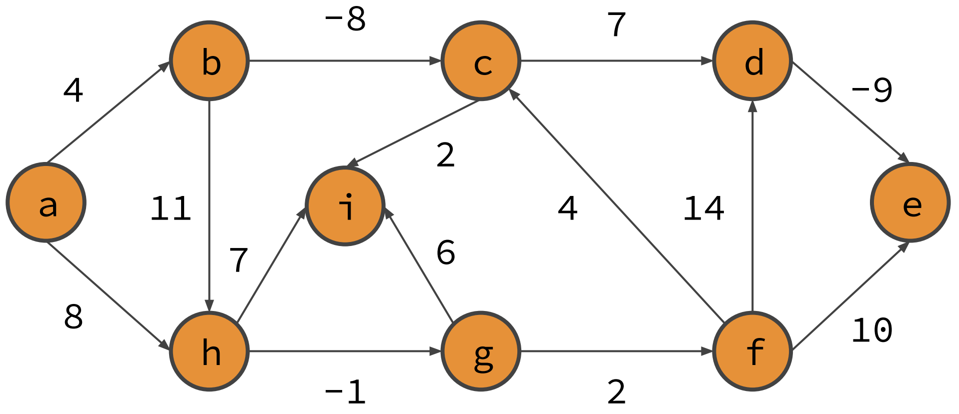
Bellman-Ford Algorithm Pseudocode

```
Bellman_Ford(G,w,s):
    Initialize_Single_Source(G,s);
    for i = 1 to |G.V| - 1:
        for each edge (u,v) ϵ G.E:
            Relax(u,v,w)
    for each edge (u,v) ϵ G.E:
        if v.d > u.d + w(u,v)
            return false
    return true
```

Simulation of Bellman-Ford Algorithm



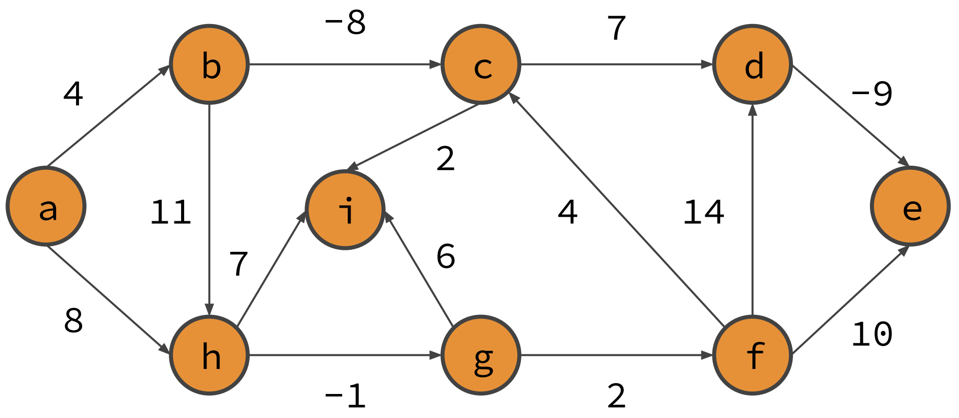
Simulation of Bellman-Ford Algorithm Continued



Parent	–	–	–	–	–	–	–	–	–
vertex	a	b	c	d	e	f	g	h	i
init	–	–	–	–	–	–	–	–	–
1	–	–	–	–	–	–	–	–	–
2	–	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

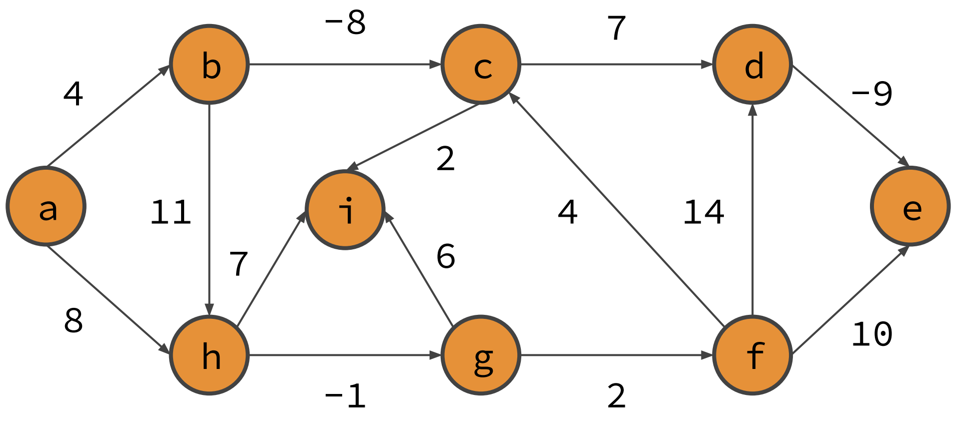
Simulation of Bellman-Ford Algorithm Continued



Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL
vertex	a	b	c	d	e	f	g	h	i
init	0	∞	∞	∞	∞	∞	∞	∞	∞
1	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Simulation of Bellman-Ford Algorithm Continued

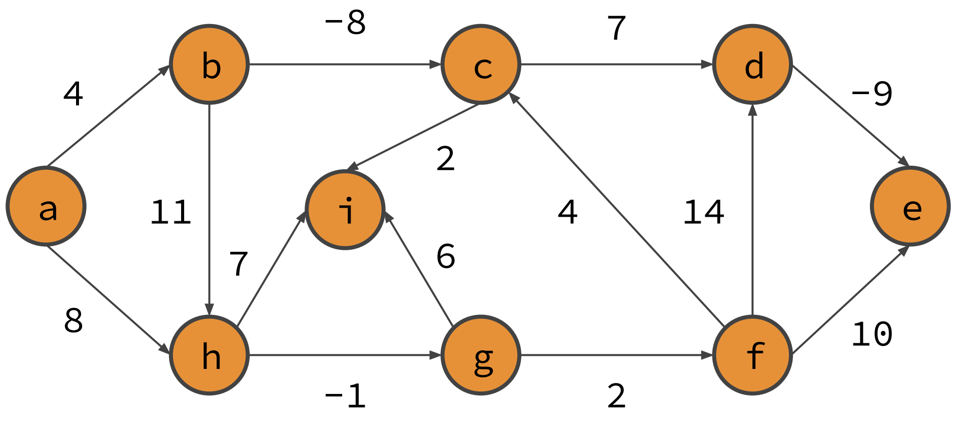


Pass: 1

Parent	NIL	a	b	c	d	NIL	h	a	c
vertex	a	b	c	d	e	f	g	h	i
init	0	∞	∞	∞	∞	∞	∞	∞	∞
1	–	4	-4	3	-6	–	7	8	-2
2	–	–	–	–	–	–	–	–	–
3	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Simulation of Bellman-Ford Algorithm Continued

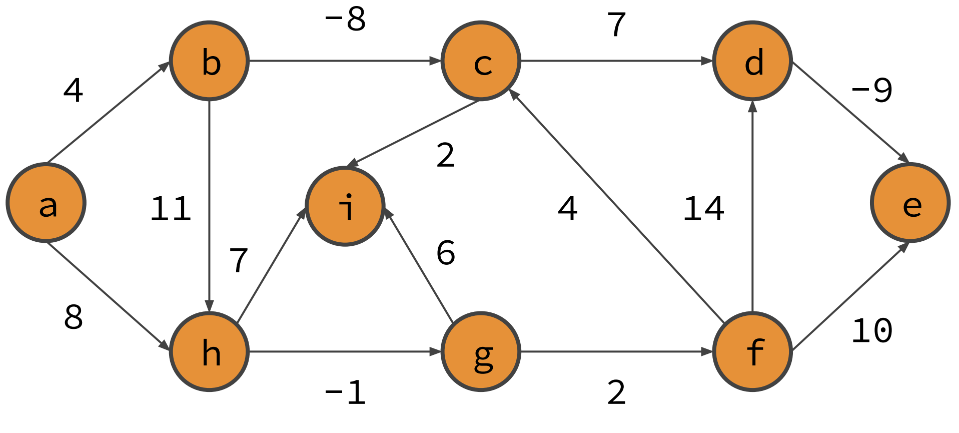


Pass: 2

Parent	NIL	a	b	c	d	g	h	a	c
vertex	a	b	c	d	e	f	g	h	i
init	0	∞	∞	∞	∞	∞	∞	∞	∞
1	–	4	-4	3	-6	–	7	8	-2
2	–	–	–	–	–	9	–	–	–
3	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Simulation of Bellman-Ford Algorithm Continued

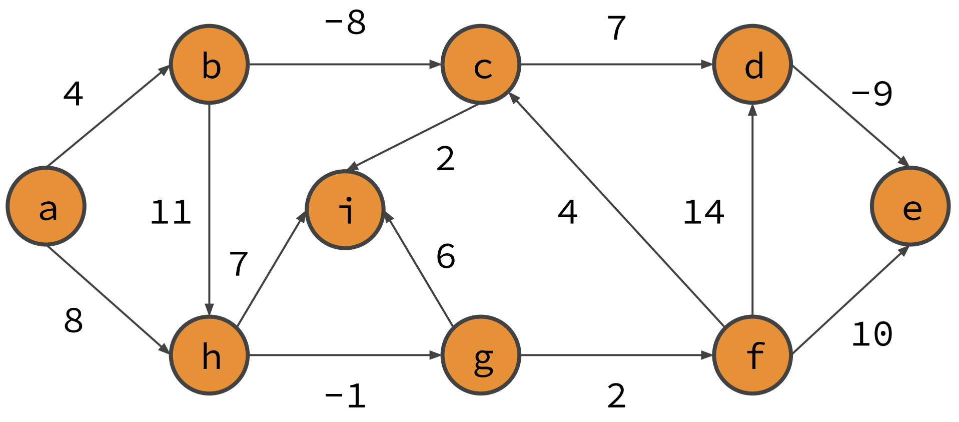


Pass: 3

Parent	NIL	a	b	c	d	g	h	a	c
vertex	a	b	c	d	e	f	g	h	i
init	0	∞	∞	∞	∞	∞	∞	∞	∞
1	–	4	-4	3	-6	–	7	8	-2
2	–	–	–	–	–	9	–	–	–
3	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Simulation of Bellman-Ford Algorithm Continued

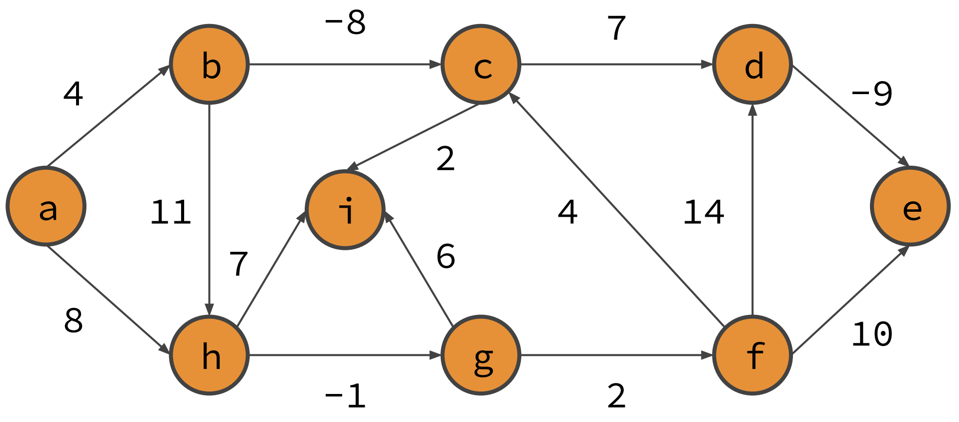


Pass: 4

Parent	NIL	a	b	c	d	g	h	a	c
vertex	a	b	c	d	e	f	g	h	i
init	0	∞	∞	∞	∞	∞	∞	∞	∞
1	-	4	-4	3	-6	-	7	8	-2
2	-	-	-	-	-	9	-	-	-
3	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Simulation of Bellman-Ford Algorithm Continued

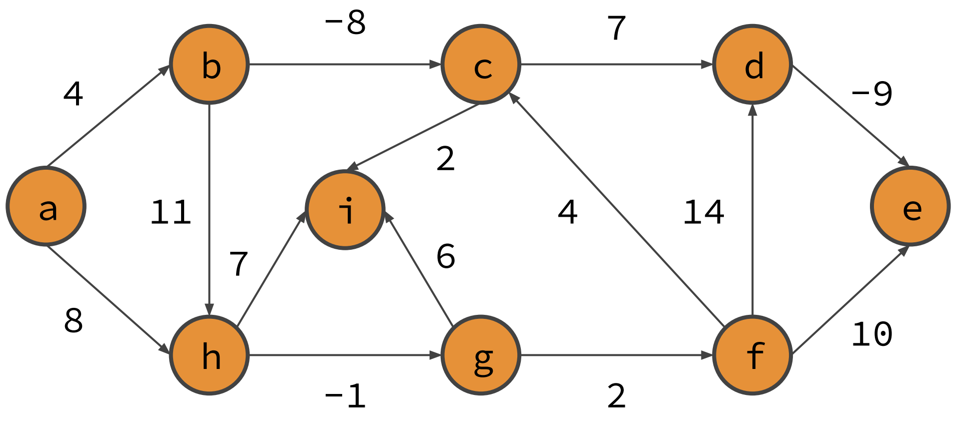


Pass: 5

Parent	NIL	a	b	c	d	g	h	a	c
vertex	a	b	c	d	e	f	g	h	i
init	0	∞	∞	∞	∞	∞	∞	∞	∞
1	–	4	-4	3	-6	–	7	8	-2
2	–	–	–	–	–	9	–	–	–
3	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Simulation of Bellman-Ford Algorithm Continued

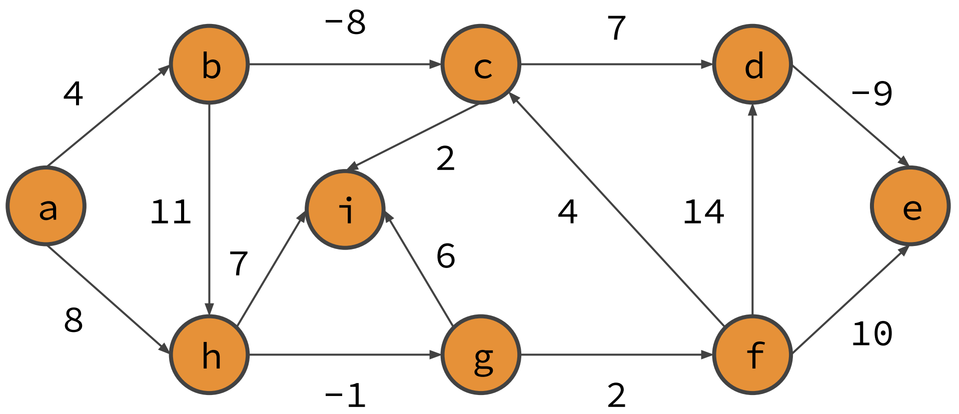


Pass: 6

Parent	NIL	a	b	c	d	g	h	a	c
vertex	a	b	c	d	e	f	g	h	i
init	0	∞	∞	∞	∞	∞	∞	∞	∞
1	-	4	-4	3	-6	-	7	8	-2
2	-	-	-	-	-	9	-	-	-
3	-	-	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-	-	-
7	-	-	-	-	-	-	-	-	-
8	-	-	-	-	-	-	-	-	-

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Simulation of Bellman-Ford Algorithm Continued

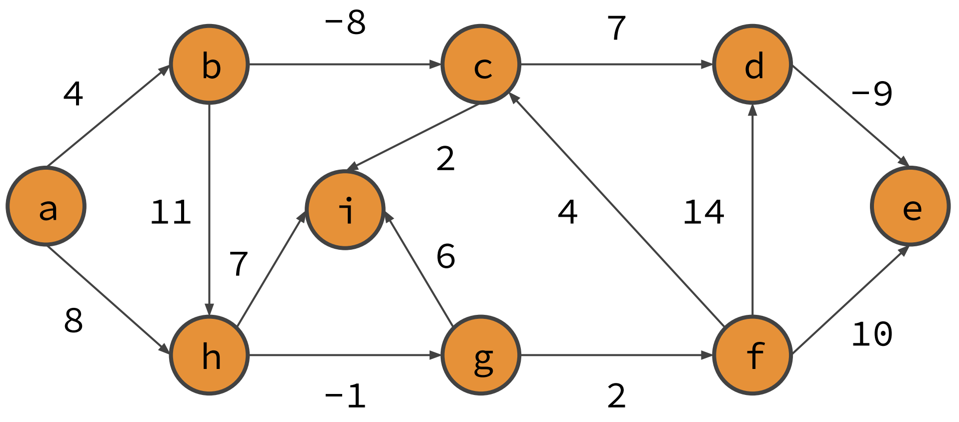


Pass: 7

Parent	NIL	a	b	c	d	g	h	a	c
vertex	a	b	c	d	e	f	g	h	i
init	0	∞	∞	∞	∞	∞	∞	∞	∞
1	–	4	-4	3	-6	–	7	8	-2
2	–	–	–	–	–	9	–	–	–
3	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Simulation of Bellman-Ford Algorithm Continued

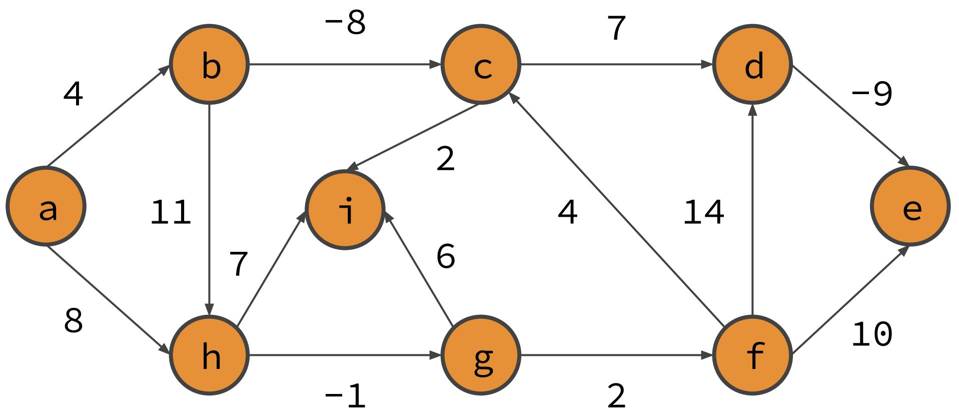


Pass: 8

Parent	NIL	a	b	c	d	g	h	a	c
vertex	a	b	c	d	e	f	g	h	i
init	0	∞	∞	∞	∞	∞	∞	∞	∞
1	–	4	-4	3	-6	–	7	8	-2
2	–	–	–	–	–	9	–	–	–
3	–	–	–	–	–	–	–	–	–
4	–	–	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–	–	–
7	–	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–	–

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Simulation of Bellman-Ford Algorithm Continued



We have to check whether any of the edges can be relaxed any more. In this example no edges can be relaxed after |V| - 1 iterations so, there is no negative cycle.

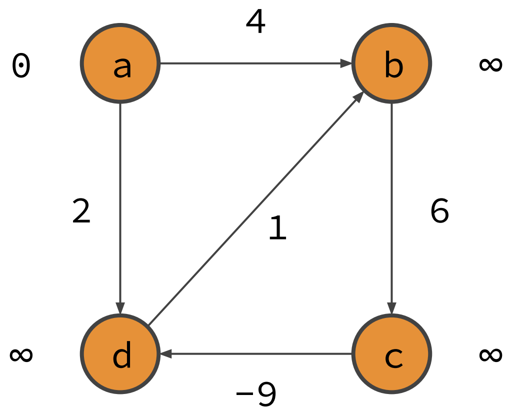
Pass: 8

Edge List: (a, b), (a, h), (b, c), (b, h), (c, d), (c, i), (d, e), (f, c), (f, d), (f, e), (g, f), (g, i), (h, g), (h, i)

Complexity Analysis of Bellman-Ford Algorithm

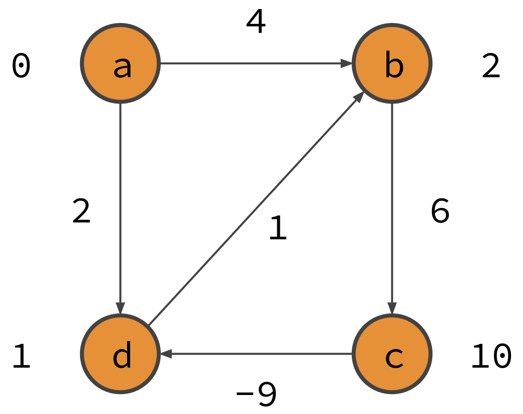
The Bellman-Ford algorithm runs in time O(VE), since the initialization takes Θ(V) time, each of the |V| - 1 passes over the edges takes Θ(E) time, and the last for loop takes O(E) time.

Negative Cycles in Bellman-Ford Algorithm



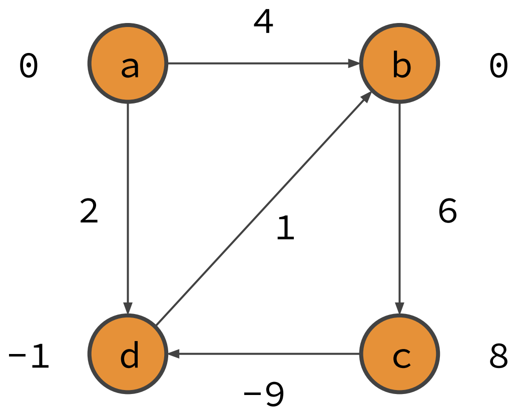
Negative Cycles in Bellman-Ford Algorithm Continued

```
Pass: 1
```



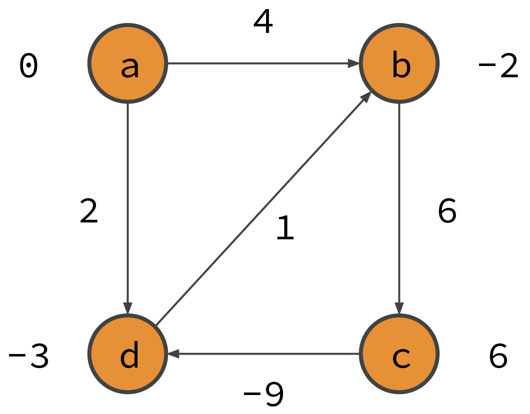
Negative Cycles in Bellman-Ford Algorithm Continued

```
Pass: 2
```



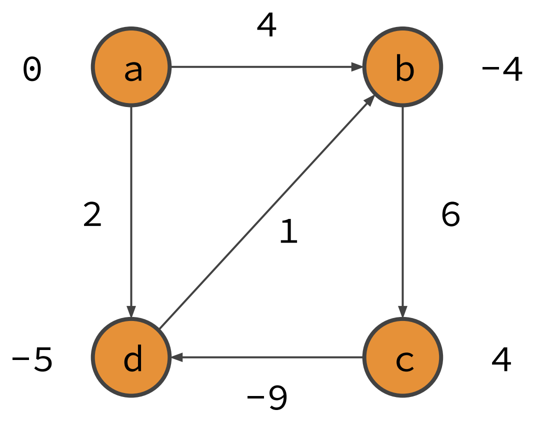
Negative Cycles in Bellman-Ford Algorithm Continued

```
Pass: 3
```



Negative Cycles in Bellman-Ford Algorithm Continued

Pass: 4



The bellman ford algorithm on this graph crossed |V|-1 steps and it still can be relaxed.