Lecture Notes on Single-Source Shortest Paths

Jim Newton

September 22, 2026

1 Shortest paths on weighted graphs

- What is a weighted graph?
- What is a shortest path?
- What are negative weights?
- What are negative cycles?
- What are multiple source shortest paths?

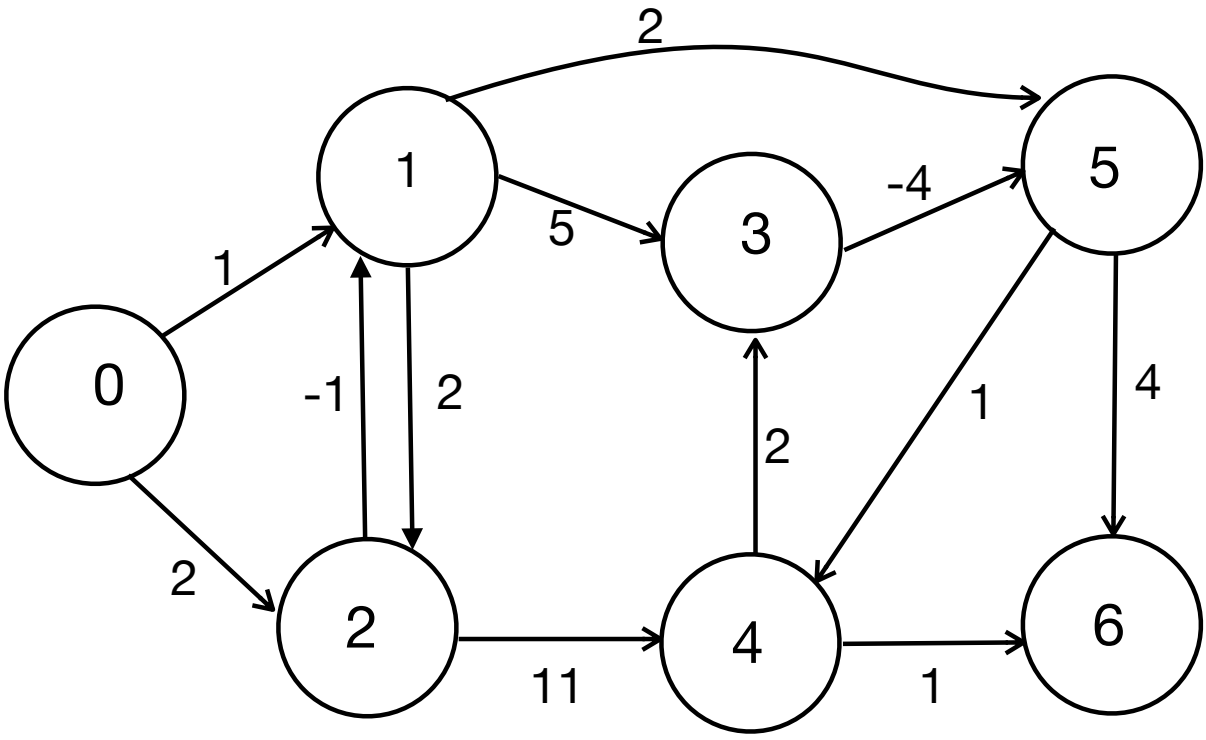
For the moment we are interested in weights we can *add* and compare with $<$.

2 Weighted graphs

Definition 2.1. If $G = (V, E)$ is a graph, a *weight* is a function $W : E \to S$. Typically S is $\mathbb{N}$ or $\mathbb{R}$.

Let $u \xrightarrow{p} v$ denote the path p from u to v. There may be several paths from u to v, $u \xrightarrow{p_1} v$, $u \xrightarrow{p_2} v$, $\ldots u \xrightarrow{p_m} v$. We can find the *weight of a path* by adding the individual weights of the edges. And since there are only finitely many such paths from u to v, it makes sense to talk about the distance between u and v as being $\delta(u, v) = \min_p w(u \xrightarrow{p} v)$. If there is no path from u to v, we define $\delta(u, v) = \infty$, and $\delta(u, u) = 0$.

Consider the example:


0
1
2
3
4
5
6
1
2
-1
2
5
-4
2
1
4
2
11
1

We write our initial estimate of $\delta(s, v)$ inside  for each $v \in V$. Using the notation,

$$d_s(v) = d(v) = \delta(s, v)\,,$$

we note that the initial estimates are that $d(s) = 0$, and otherwise $d(v) = \infty$.

3 Relaxation

We may improve the estimates of A, B, C, and D.

$$\begin{aligned}
d(A) &= \min\{d(A),\ d(s) + w(s, A)\} \\
&= \min\{\infty,\ 0 + 1\} \\
&= 1 \\
d(B) &= \min\{d(B),\ d(s) + w(s, B)\} \\
&= \min\{\infty,\ 0 + 2\} \\
&= 2 \\
d(C) &= \min\{d(C),\ d(A) + w(A, C)\} \\
&= \min\{\infty,\ 1 + 5\} \\
&= 6 \\
d(D) &= \min\{d(D),\ d(B) + w(B, D)\} \\
&= \min\{\infty,\ 2 + 11\} \\
&= 13
\end{aligned}$$

If the current estimate $d(v)$ is known and u is a predecessor of v (i.e., there is an edge (u, v)). Then we can compare $d(v)$ with $d(u) + w(u, v)$, and if

$$d(v) > d(u) + w(u, v)$$

then we can update

$$d(v) \leftarrow d(u) + w(u, v)\,.$$

This relaxation step is possible, because of the triangle inequality, and that δ is a distance function.

$$\delta(v_1, v_2) \leq \delta(v_1, v_3) + \delta(v_3, v_2)$$

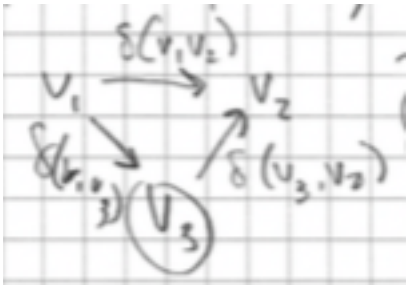
In the example above we calculated $d(D) = 13$, which is effectively the length of the path $S \to B \to D$. But there are several paths from S to D, each with potentially a different length.

$$\begin{aligned} d(S \to B \to D) &= 13 \\ d(S \to B \to A \to E \to D) &= 6 \\ d(S \to B \to A \to C \to E \to D) &= 12 \\ d(S \to B \to A \to B \to D) &= 17 \\ d(S \to B \to A \to B \to A \to B \to A \to B \to D) &= 25 \end{aligned}$$

We may continue, ad hoc updating the estimates. At each iteration we may find another path between two nodes which is shorter than the previous iteration. Since each time we update a value of $d(v)$ the value decreases, and no weights in the graph shown above are negative, the iteration will eventually terminate. The fact that it will terminate with the same values independent of the particular order we relax the vertices is less obvious, but in fact the order does not matter, provided there are no negative cycles.

It may not be clear what the best order to relax the vertices is. In the following sections, we examine various algorithms. They all relax nodes using the same algorithm, the only difference is the order we visit the vertices to relax.

The basic form of the `RELAX` algorithm is the following.



```
1 Function Relax(u,v,w,d,pred)
      Input  : u a vertex
      Input  : v a vertex
      Input  : w weight function
      Input  : d current estimate array
      Input  : pred current predecessor
               array
2     if d[v] > d[u] + w(u,v) then
3         d[v] ← d[u] + w(u,v)
4         pred[v] ← u
5         Maybe another action
           depending on caller.
```

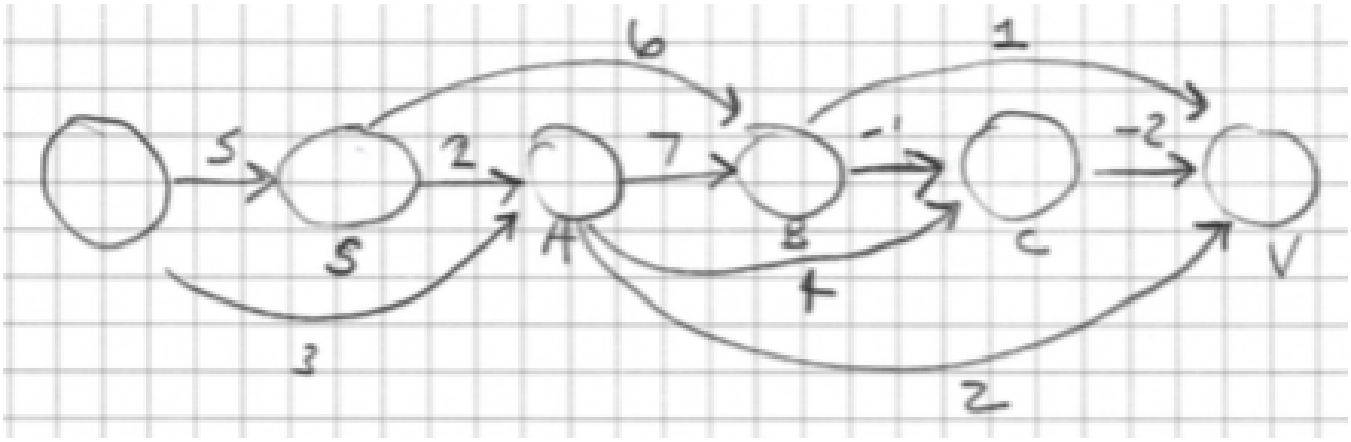
Note that the *if* $(d[v] > d[u] + w(u, v))$ *then* may sometimes by written with fewer lines of code.

```
d[v] = min( d[v], d[u] + w(u,v))
```

The problem with this optimization is that you need to update $pred[u]$ in the case that you are changing the value of $d[v]$, and in some cases such as Dijkstra (Section 8) you need to also take other action when $d[v]$ changes.

4 Shortest path in a DAG

Recall that a *DAG* is a directed acyclic graph. Any DAG can be topologically ordered from left to right such that all arrows point from left to right.



The best order to relax the vertices of a DAG is in topological order. This means a single pass though the graph suffices to find the minimum paths from any starting vertex to all other vertices. This works even if there are negative weights. There are no negative cycles, because a DAG has no cycles (negative or otherwise).

1. Given a DAG for which u precedes v, topologically sort the DAG.

2. Visit the nodes starting at u ending at v in topologically sorted order.

3. Relax the nodes on each outgoing edge, visiting the outgoing edges of a node in any order.

The complexity of this algorithm is $O(V + E)$, given the topological order, because the algorithm visits each vertex at most once (zero times for nodes to the left of u or nodes to the right of v), and visits each edge at most once.

Example, we wish to find the shortest path from S to V in the DAG shown above. What is the topological order, starting at S? We can use the Tarjan or the Kahn algorithm. Here we use the Kahn algorithm.

Constraints	S	A	B	C	V
$S \to A$	~~$S \to A$~~				
$S \to B$	~~$S \to B$~~				
$A \to B$	$A \to B$	~~$A \to B$~~			
$A \to C$	$A \to C$	~~$A \to C$~~			
$B \to V$	$B \to V$	$B \to V$	~~$B \to V$~~		
$B \to C$	$B \to C$	$B \to C$	~~$B \to C$~~		
$C \to V$	$C \to V$	$C \to V$	$C \to V$	~~$C \to V$~~	

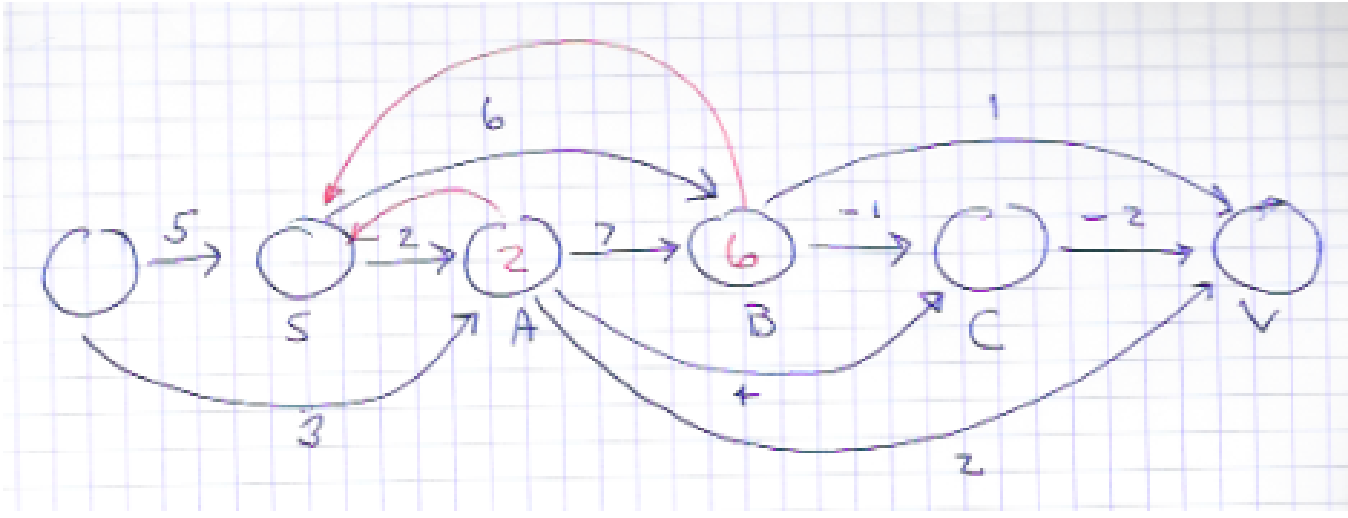
A topological order (the topological order in this case) is

$$S \to A \to B \to C \to V\,.$$

1. $S \to A \to B \to C \to V$

   First we examine the two outgoing edges of node S: $S \to A$, and $S \to B$. We set the predecessor $pred[]$ each time the estimate $d[]$ changes.

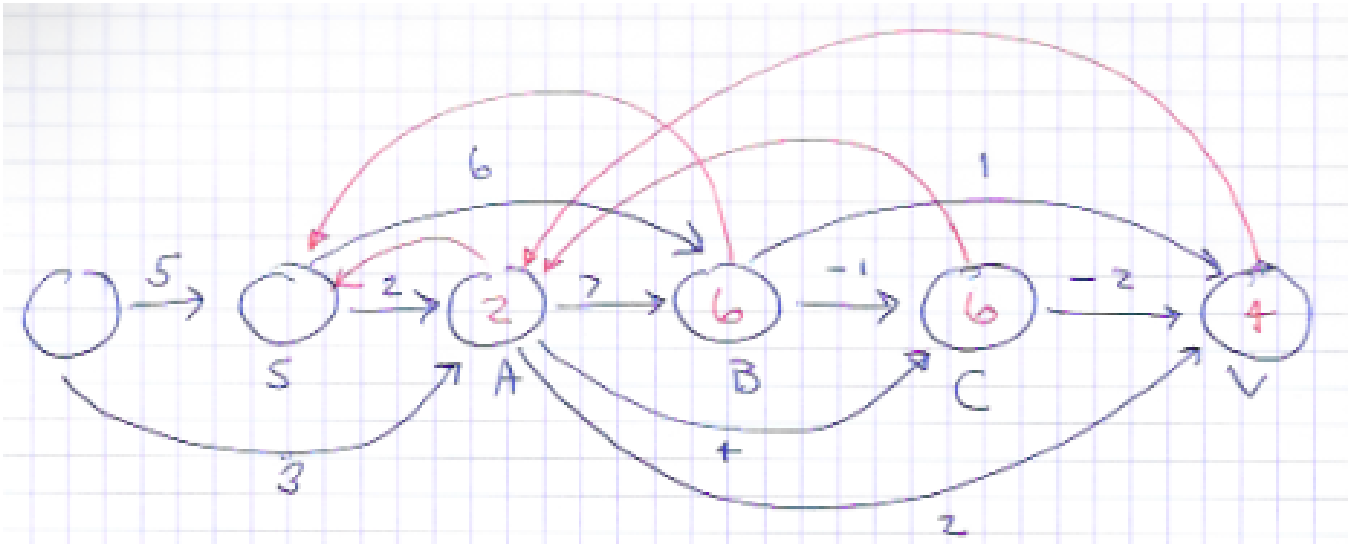
$$d(A) \leftarrow \min\{\infty, 0+2\} = 2 \qquad pred[A] \leftarrow S$$
$$d(B) \leftarrow \min\{\infty, 0+6\} = 6 \qquad pred[B] \leftarrow S$$



2. $S \to A \to B \to C \to V$

   Next, we examine the outgoing edges of A.

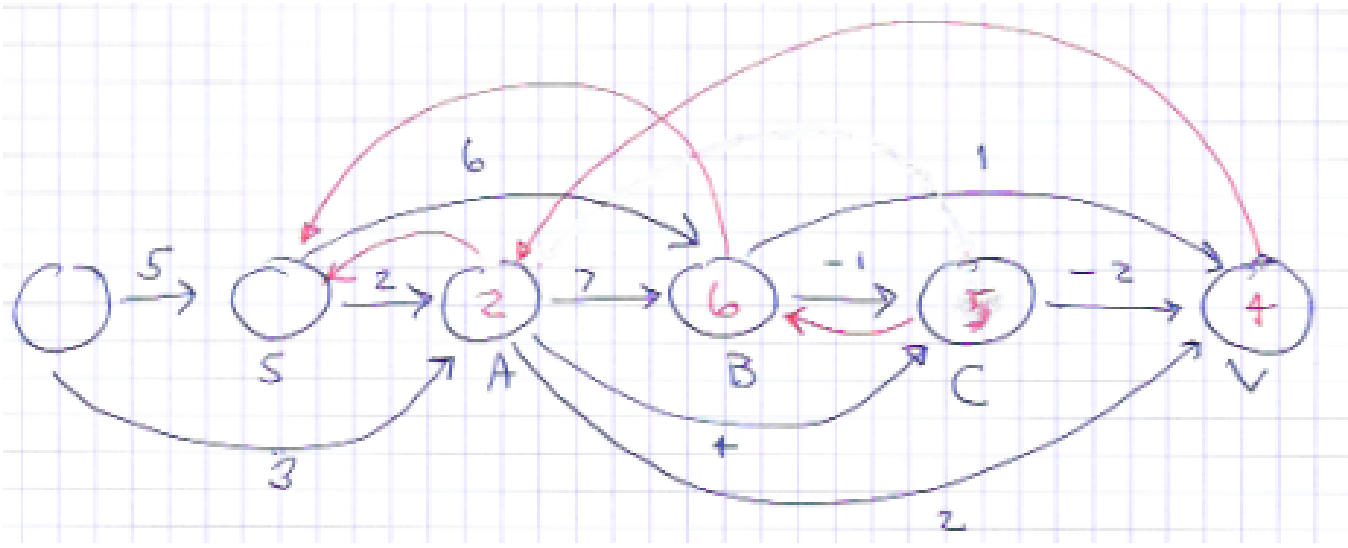
$$d(B) \leftarrow \min\{6, 2+7\} = 6 \qquad pred[B] \text{ remains the same}$$
$$d(C) \leftarrow \min\{\infty, 2+4\} = 6 \qquad pred[C] \leftarrow A$$
$$d(V) \leftarrow \min\{\infty, 2+2\} = 4 \qquad pred[V] \leftarrow A$$



3. $S \to A \to B \to C \to V$

   Next, we examine the outgoing edges of B.

$$d(C) \leftarrow \min\{6, 6+-1\} = 5 \qquad pred[C] \leftarrow B$$
$$d(V) \leftarrow \min\{4, 6+1\} = 4 \qquad pred[V] \text{ remains the same}$$

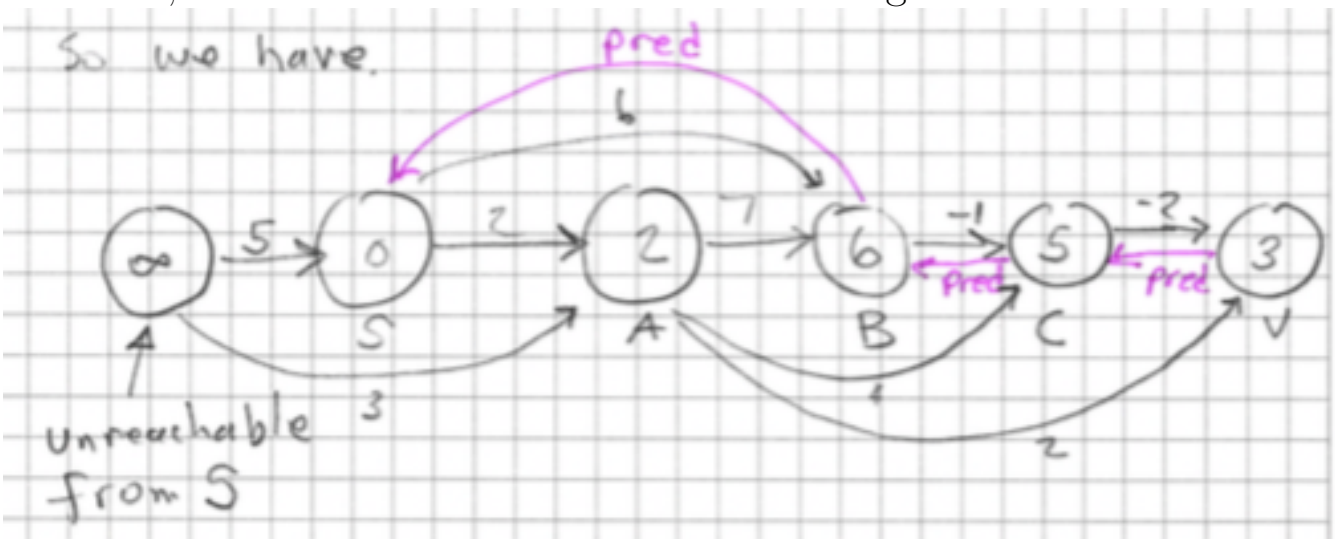


4. $S \to A \to B \to C \to V$

   Finally, we examine the outgoing edges (there is only one) of C.

$$d(C) \leftarrow \min\{4, 5 + -2\} = 3 \qquad pred[V] \leftarrow C$$

So we are left with the following. And we can trace back the arrows from V to A; which is $V \to C \to B \to S$ whose length is $-2+-1+6 = 4$.



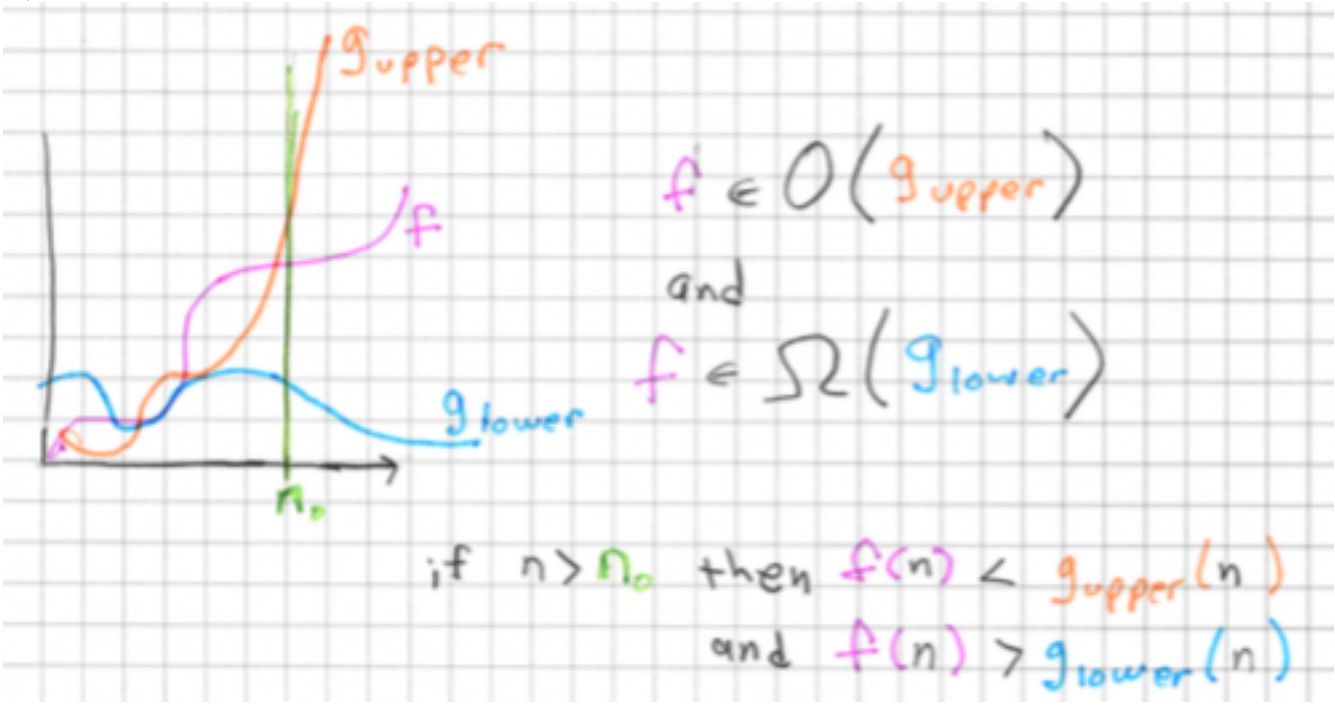
5 Summary of DAG

The shortest path using topological sort works for a DAG, even if there are negative weights. In particular no cycles and no negative weight cycles exist.

If we have cycles but no negative weight edges at all we may use the Dijkstra algorithm (Section 8). If we have negative weight cycles we may use the Bellman-Ford algorithm.

6 Big-O and Big-Ω Notation

Recall that if $f \in O(g)$, then $f(n)$ is eventually bounded above by $cg(n)$ for $n >> 0$, and if $f \in \Omega(g)$, then $f(n)$ is eventually bounded below by $cg(n)$ for $n >> 0$.



7 The Bellman-Ford algorithm

```
1 Function Bellman-Ford(s, V,E,w)
  |  Input  : s a starting vertex
  |  Input  : V set of vertices
  |  Input  : E set of edges
  |  Input  : w a weight function
2 |  Initialize
3 |  repeat |V| − 1 times
4 |  |  for (u, v) ∈ E do
5 |  |  |  Relax(u,v,w,d,pred)
  |  /* Now check for negative weight cycles        */
6 |  for (u, v) ∈ E do
7 |  |  if  d[v] > d[u] + w(u, v) then
8 |  |  |  report negative weight cycle
```

```
1 Function Relax(u,v,w,d,pred)
  |  Input  : u a vertex
  |  Input  : v a vertex
  |  Input  : w array of precedents
  |  Input  : d current estimate array
  |  Input  : pred current predecessor array
2 |  if  d[v] > d[u] + w(u, v) then
3 |  |  d[v] ← d[u] + w(u, v)
4 |  |  pred[v] ← u
```

The complexity is $O(|V||E|)$, but since $E = O(|V|^2)$ worst case, Bellman-Ford may have $O(|V|^3)$ complexity. Note the following:

1. If an iteration of the `repeat` loop fails to modify $d[]$ value, then we can abort the iteration early.

2. If the final iteration did not change $d[]$, then the final checking step can be omitted—there are no negative cycles.

8 The Dijkstra algorithm

The Dijkstra algorithm makes use of a heap or priority queue, which has three important operations.

- `INITIALIZE-HEAP` — $O(n)$ — Copy each vertex into the heap, $priority(s) = 0$, otherwise initializing all other priorities to ∞
- `EXTRACT-MIN` — $O(\log(n))$ — remove item with minimum priority
- `REDUCE-MIN` — amortized $O(1)$ — reduce the priority of some item which is not necessarily already the minimum priority.

There are two popular types of heaps: `binary-heap` and `fibonacci-heap`. If you are writing it yourself, `binary-heap` is easier to implement. But if you are using a standard library which provides `fibonacci-heap`, then that's probably a better choice. Why? Because the `fibonacci-heap` offers a `REDUCE-MIN` operation with amortized $O(1)$ time complexity, whereas for the `binary-heap`, `REDUCE-MIN` has $O(\log(n))$ complexity.

```
1  Function Dijkstra(s,V,Adj,w)
       Input  : s a starting vertex
       Input  : V set of vertices
       Input  : Adj adjacency list as mapping V → List[V]
       Input  : w a weight function
2      pred[s] ← None
3      d[s] ← 0
4      for v ∈ V \ {s} do
5          d[v] ← ∞
6      S ← ∅
7      Q ← INITIALIZE-HEAP(V)     // initialize priority
        queue/heap
8      while Q ≠ ∅ do
9          u ← EXTRACT-MIN(Q)       // removing u from Q
10         S ← S ∪ {u}
11         for v ∈ Adj[u] \ S do
12             Relax(u, v, w, d, pred, Q)
```

```
1  Function Relax(u,v,w,d,pred,Q)
       Input  : u, v vertices
       Input  : w array of weights
       Input  : d current estimate array
       Input  : pred current predecessor array
       Input  : Q the priority queue/heap
2      if d[v] > d[u] + w(u, v) then
3          d[v] ← d[u] + w(u, v)
4          REDUCE-MIN(Q, v, d[v])
5          pred[v] ← u
```

The complexity of the Dijkstra algorithm is $O(|E| + |V| \log |V|)$, because

line 7 — is $O(|V|)$.

line 8 — the `while` loops $O(|V|)$ times.

line 9 — is $O(|V| \times \log(|V|))$.

line 11 — loops $O(|E|)$ total, don't be confused about the `while` loop.

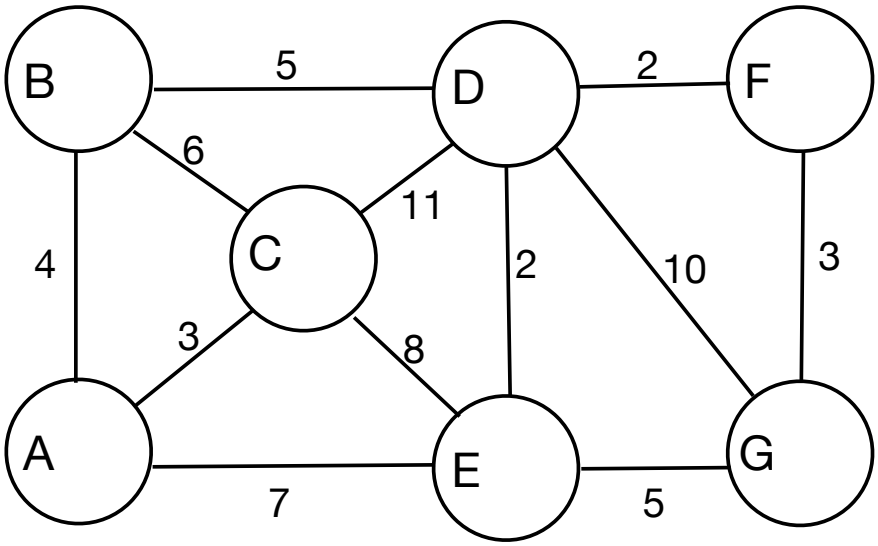
line 12 — is executed $O(|E|)$ with amortized complexity $O(1)$, thus its complexity is $O(|E|)$.

9 Dijkstra example

This example is taken from a YouTube video posted by Nataniel Fan, entitled *Dijkstra's Algorithm*.

The Dijkstra algorithm is capable of finding either the shortest path from one vertex to another, or of finding the shortest path from a given vertex to all other vertices. The only difference is do you stop when finding the shortest path to a particular destination, or do you continue until all the verticies are resolved. In this example, we will look at the sortest path between two vertices.

We want to find the shortest path from A to F in the following graph.



First set $d[A] \leftarrow 0$, because the distance form A to A is 0, and the initial estimate for all the remaining distances is ∞. This operation can be accomplished in different ways.

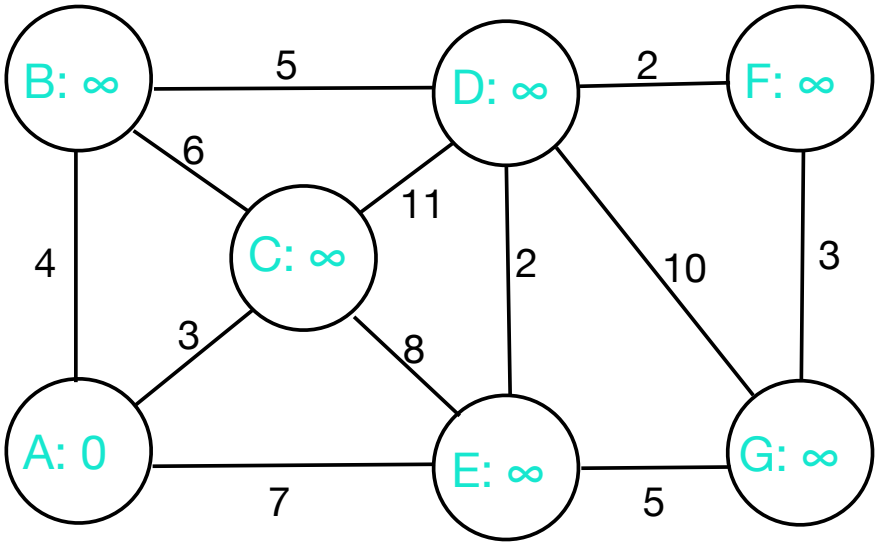
1. Use a large number for ∞, e.g., `MAX_INT` or 1×10^8.

2. If the programming language can represent ∞, then you can enter all the vertices into the priority queue with $priority = \infty$. In this case $\min\{x, \infty\}$ should return x for any x.

3. If your programming language does not support ∞, then you must always know (or be able to check) whether a given vertex is found in the priority before relaxing it.

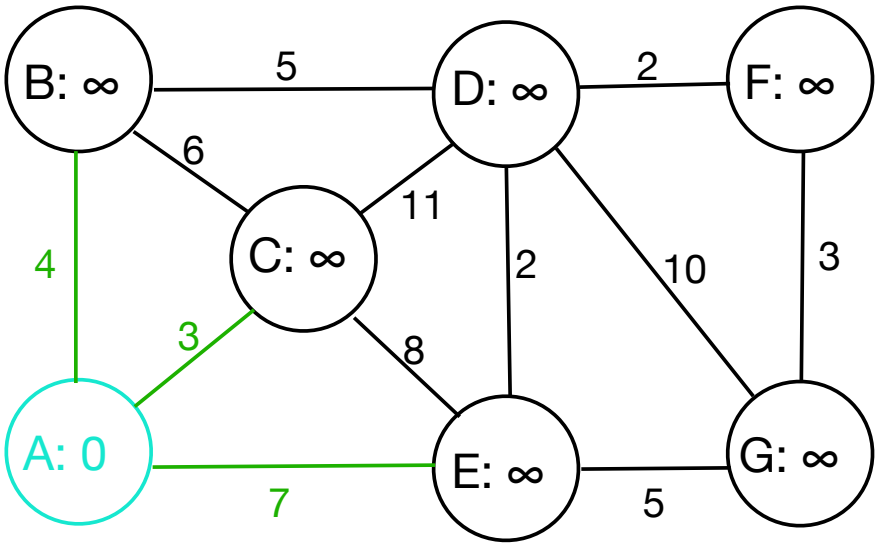
So in the latter case, the relax step will contain one case for "not yet registered", and a second case for "already registered".

For this example, let's suppose that the language supports ∞. The graph looks like the following after initialization.



Select the vertex with minimum current $d[]$ value. EXTRACT-MIN $\rightarrow A$, because $d[A] = 0$. We now examine all *outgoing* edges of vertex A, because no vertices have yet been visited.

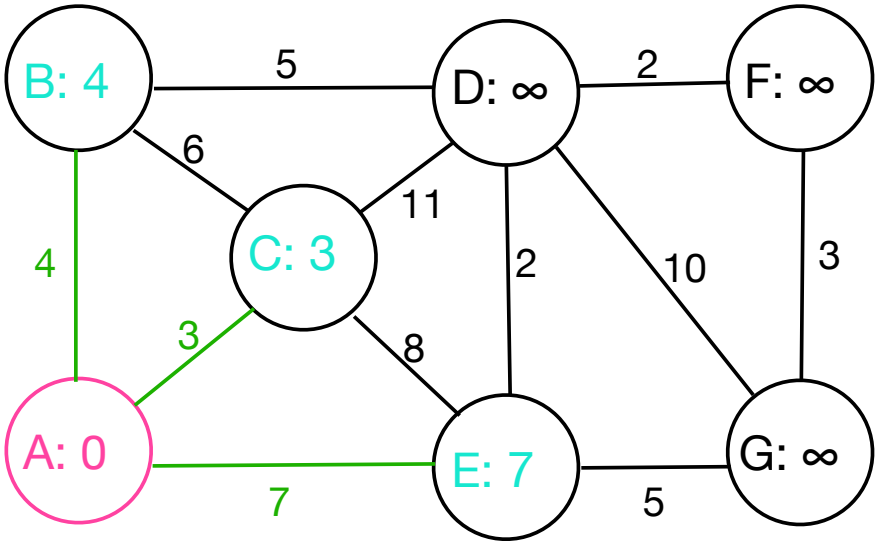
We may now, as always, examine the outgoing edges in any order. The following image shows these outgoing edges with weights 4, 3, and 7.



We relax vertices B, C, and E and mark A as visited. We indicate A as visited as:  .

Relaxation involves the following operations:

$$\begin{aligned} d(B) &= \min\{d(B), d(A) + w(A, B)\} \\ &= \min\{\infty, 0 + 4\} \\ &= 4 \\ d(C) &= \min\{d(C), d(A) + w(A, C)\} \\ &= \min\{\infty, 0 + 3\} \\ &= 3 \\ d(E) &= \min\{d(E), d(A) + w(A, E)\} \\ &= \min\{\infty, 0 + 7\} \\ &= 7 \end{aligned}$$

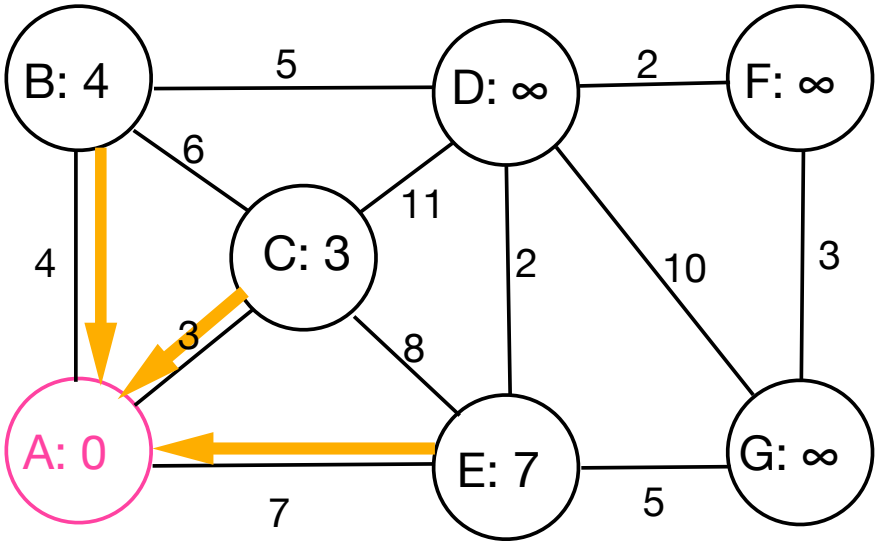


We also update the predecessor every time $d[]$ changes. Since when we examined B, C, and E the values $d[B]$, $d[C]$, and $d[E]$ all changed, we update three predecessors.

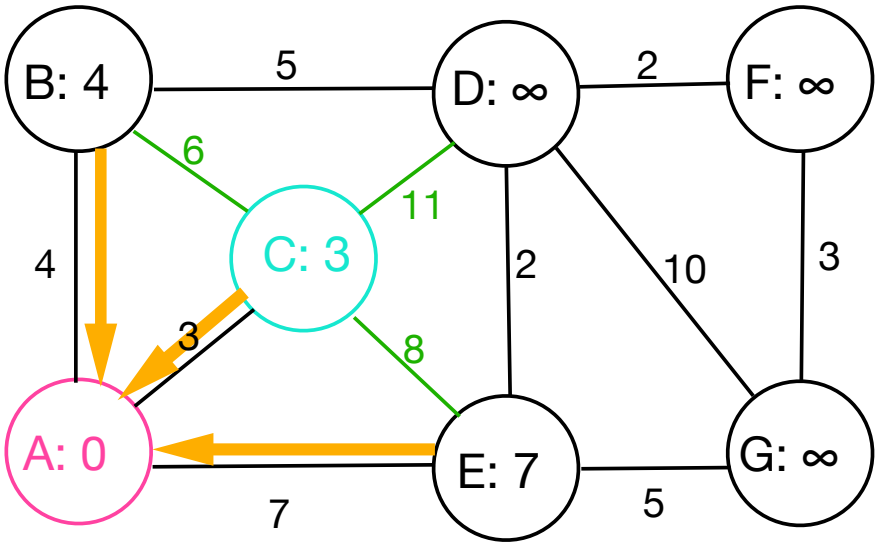
$$pred[B] \leftarrow A \qquad B \xrightarrow{pred} A$$

$$pred[C] \leftarrow A \qquad C \xrightarrow{pred} A$$

$$pred[E] \leftarrow A \qquad C \xrightarrow{pred} A$$



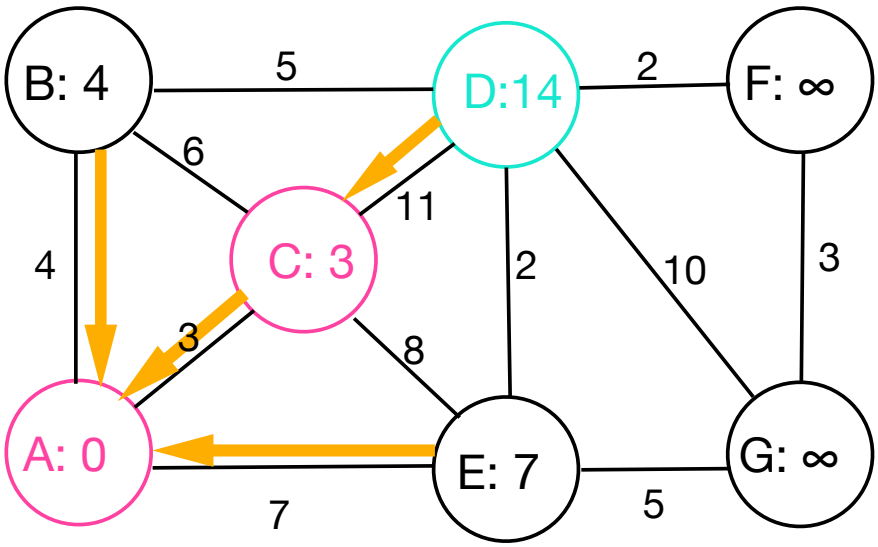
Now we again find the minimum value in $d[]$, considering only the as of yet unvisited vertices. EXTRACT-MIN $\rightarrow C$, because $d[C] = 3$. We now examine all *outgoing* edges of vertex C ignoring edges leading to node A, because A has already been visited. We must examine the edges leading to B, D, and E, with weights 6, 11, and 8, as shown in the image.



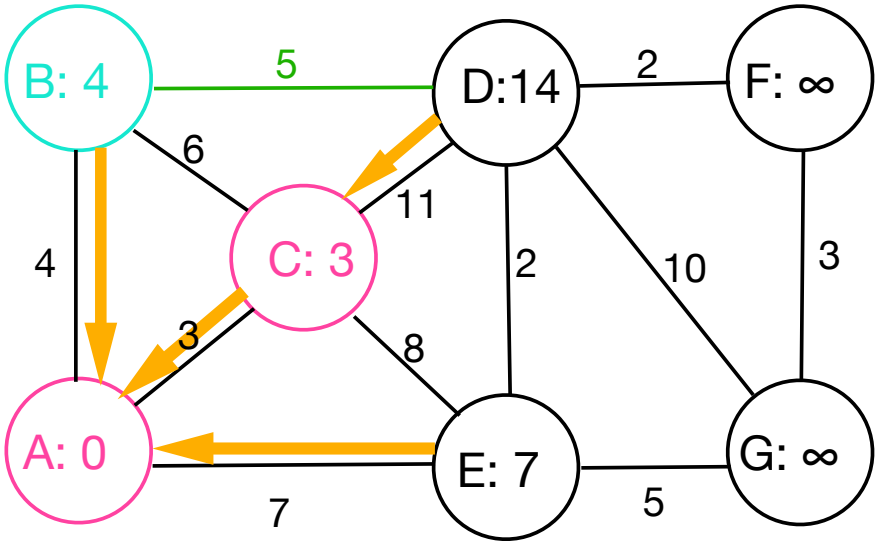
We now relax $d[B]$, $d[D]$, and $d[E]$ as follows:

$$
\begin{aligned}
d[B] &= \min\{d(B), d(C) + w(C, B)\} \\
&= \min\{4, 3 + 6\} \\
&= 4 && d[B] \text{ did not change} \\
d[D] &= \min\{d(D), d(C) + w(C, D)\} \\
&= \min\{\infty, 3 + 11\} \\
&= 14 && d[D] \text{ changed so } pred[D] \leftarrow C \\
d[E] &= \min\{d(E), d(C) + w(C, E)\} \\
&= \min\{7, 3 + 8\} \\
&= 7 && d[E] \text{ did not change}
\end{aligned}
$$

The vertex  goes into the visited list, and the updated graph is as follows:



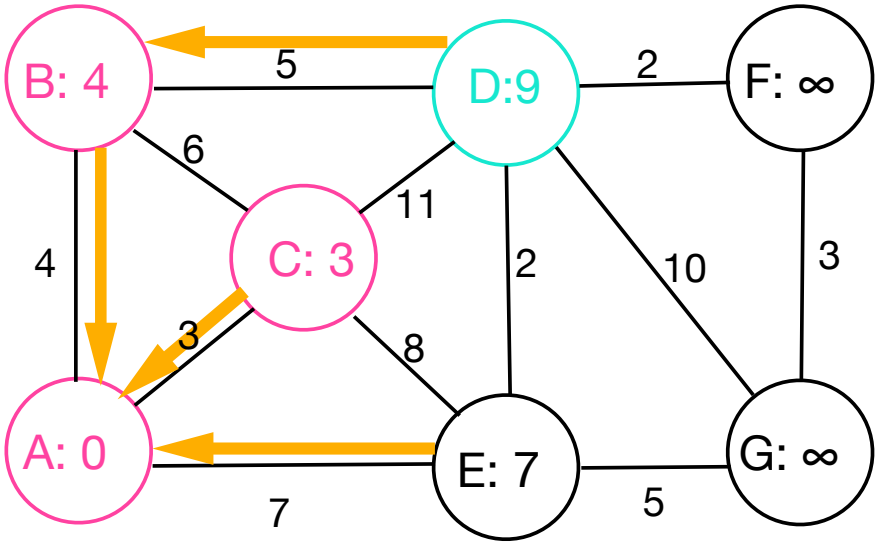
We now find the minimum value ignoring already visited vertices. EXTRACT-MIN $\rightarrow$ B, because $d[B] = 4$ is the minimum. We now examine all *outgoing* edges, only one in this case, because A and C have already been visited.



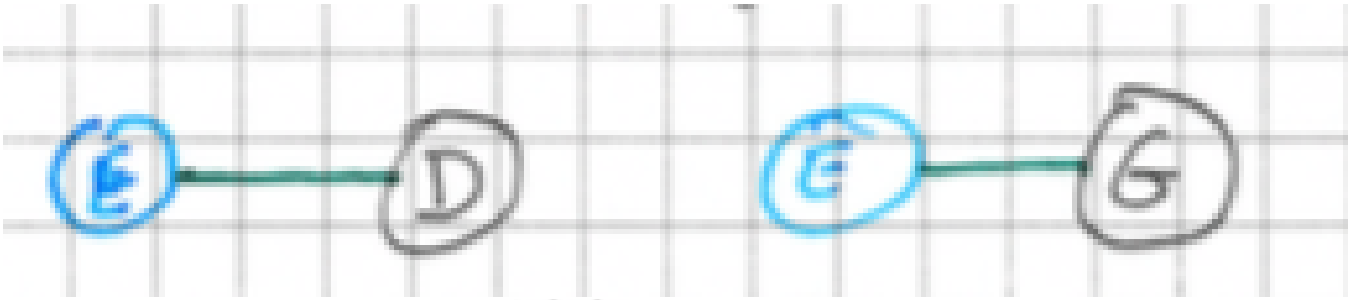
We relax D.

$$\begin{aligned} d[D] &= \min\{d(D), d(B) + w(B, D)\} \\ &= \min\{15, 4 + 5\} \\ &= 9 \qquad\qquad d[D] \text{ changed so } pred[D] \leftarrow B \end{aligned}$$

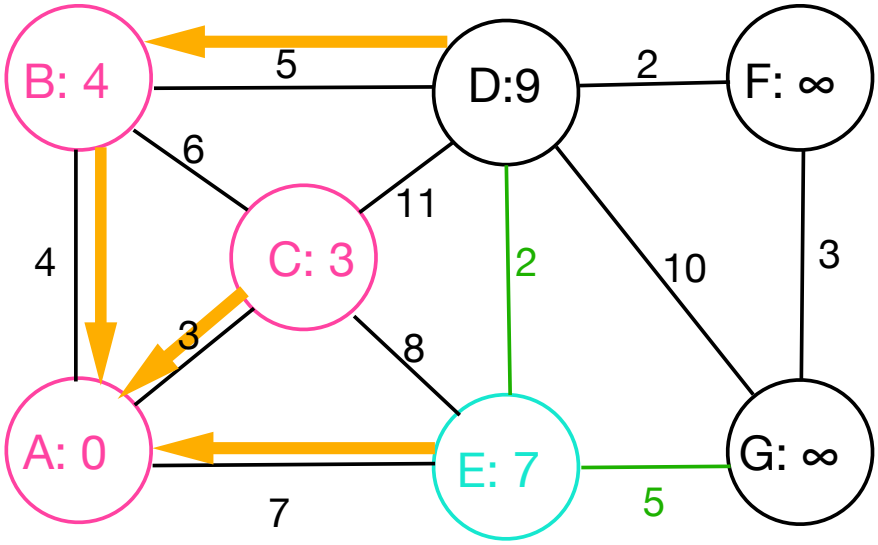
We are finished with B, putting it into the visited list.



EXTRACT-MIN $\rightarrow E$, because $d[E] = 7$ is the minimum. We now visit the edges from E to unvisited vertices.



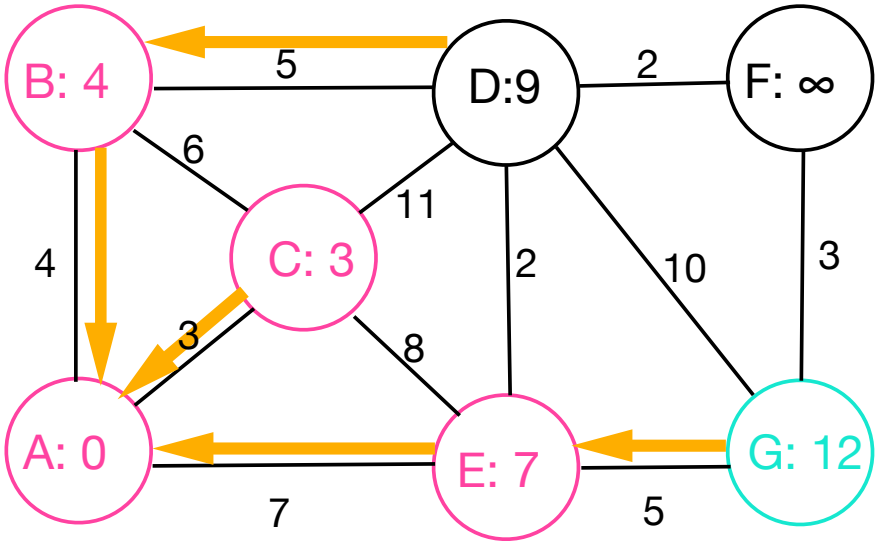
We ignore edges from E to A and C.



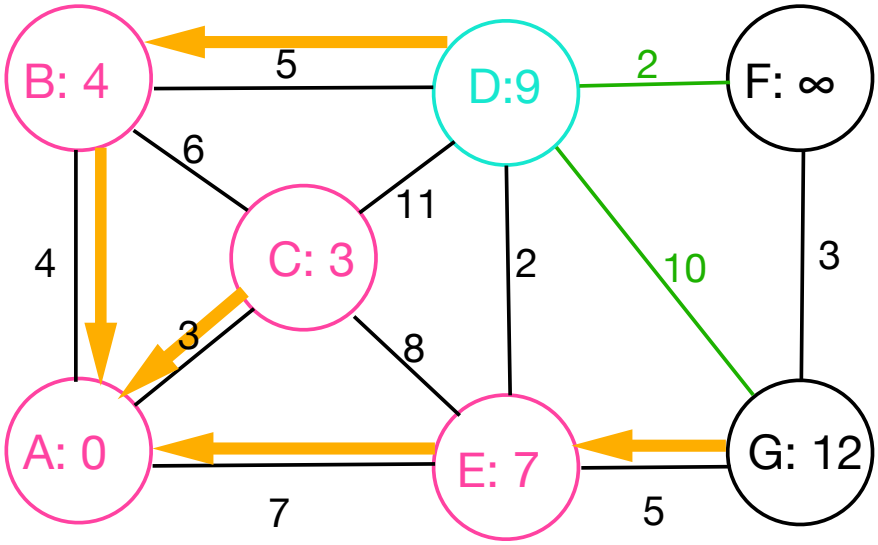
Relaxing D and G:

$$
\begin{aligned}
d[D] &= \min\{d(D), d(E) + w(E, D)\} \\
&= \min\{9, 7 + 2\} \\
&= 9 && d[D] \text{ did not change} \\
d[G] &= \min\{d(G), d(E) + w(E, G)\} \\
&= \min\{\infty, 7 + 5\} \\
&= 12 && d[G] \text{ changed so } pred[G] \leftarrow E
\end{aligned}
$$

We are finished with E, so the visited list is now $\{A, B, C, E\}$.



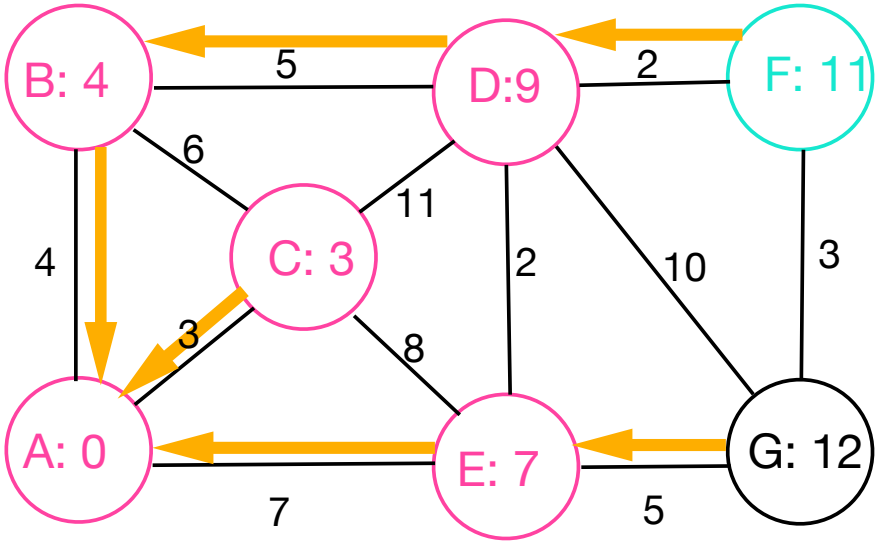
EXTRACT-MIN $\rightarrow D$, because $d[D] = 9$ is the minimum. We now visit the edges from D to unvisited vertices, F, and G. When we relax G nothing changes.



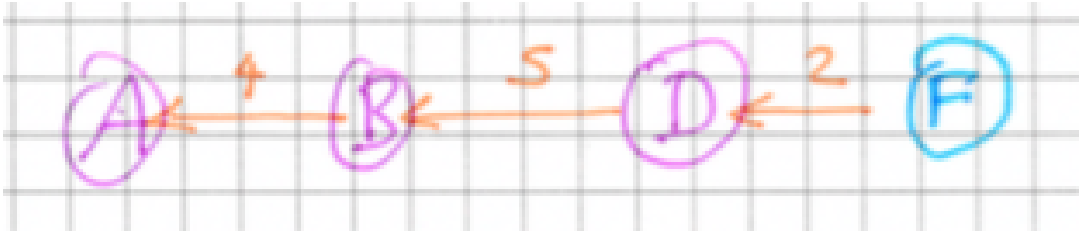
$$
\begin{aligned}
d[F] &= \min\{d(F), d(D) + w(D, F)\} \\
&= \min\{\infty, 9 + 2\} \\
&= 11 && d[F] \text{ changed so } pred[F] \leftarrow D \\
d[G] &= \min\{d(G), d(D) + w(D, G)\} \\
&= \min\{12, 9 + 10\} \\
&= 12 && d[G] \text{ did not change}
\end{aligned}
$$

The visited list is updated to also include D.

This time EXTRACT-MIN $\rightarrow F$, because $d[F] = 11$. Since F is the goal vertex, we are done.



The shortest path from A to F has a weight of 11. The actual path may be collected by following the *pred* pointers (in reverse).



10 Formulas to remember

$$\sum_{v \in V} degree(v) = \Theta(|E|)$$

$$|E| = O|V|^2$$

$$|V| = O|E|$$