Single-Source Shortest Paths

September 22, 2026

Examples and context

- path
- shortest path
- distance
- eccentricity
- diameter/periphery
- radius/center

Examples and context

Shortest paths on weighted graphs

- What is a weighted graph?
- What is a shortest path (betweeen two vertices)?
- What are negative weights?
- What are negative cycles?
- What are single-source shortest paths?
- What are multiple-source shortest paths?
- What are some algorithms to compute them?

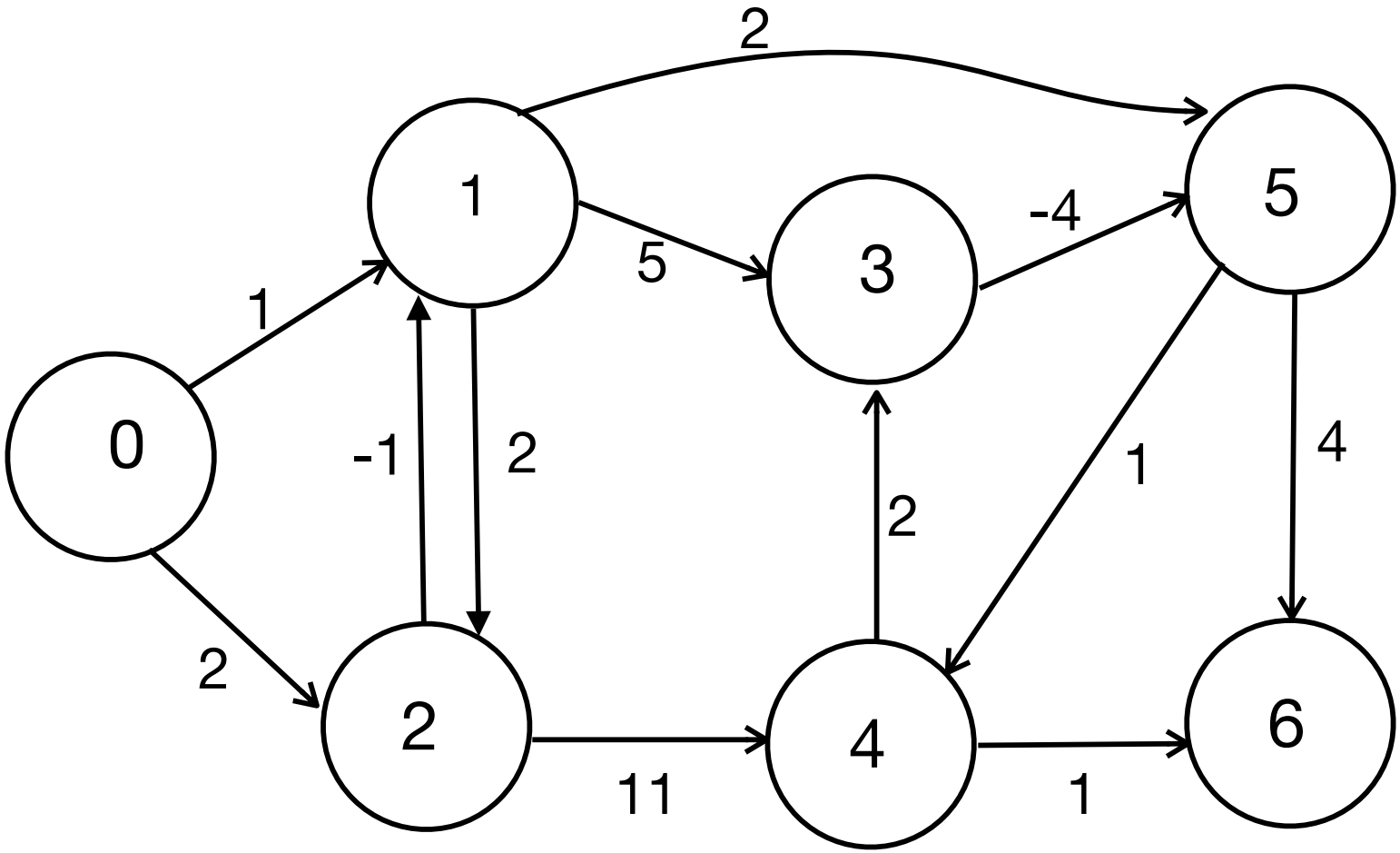
Weighted Graphs

Weighted graphs

Definition (weighted graph)

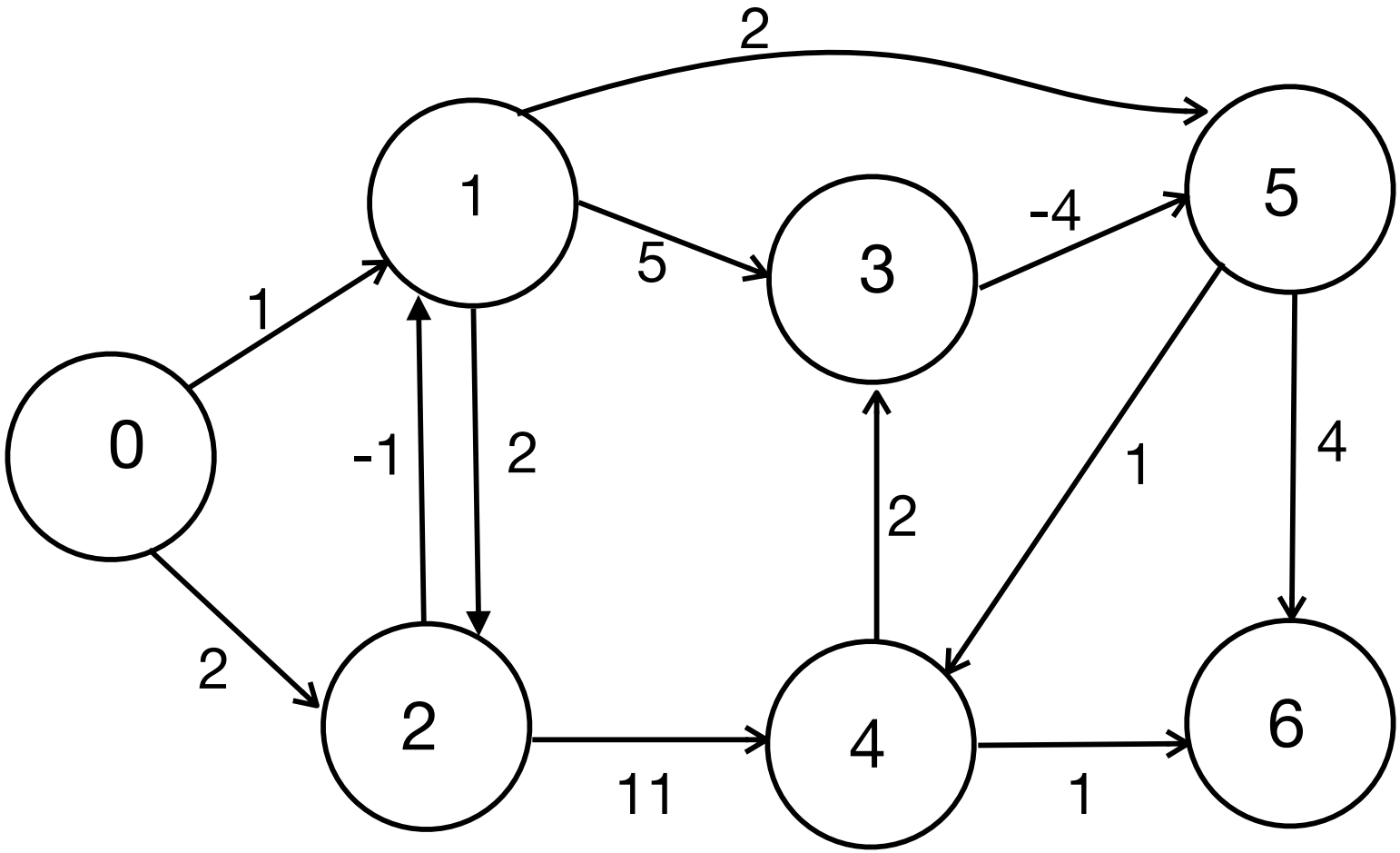
If $G = (V, E)$ is a graph, a *weight* is a function $W : E \to S$. Typically S is $\mathbb{N}$ or $\mathbb{R}$.

Paths in a weighted graph



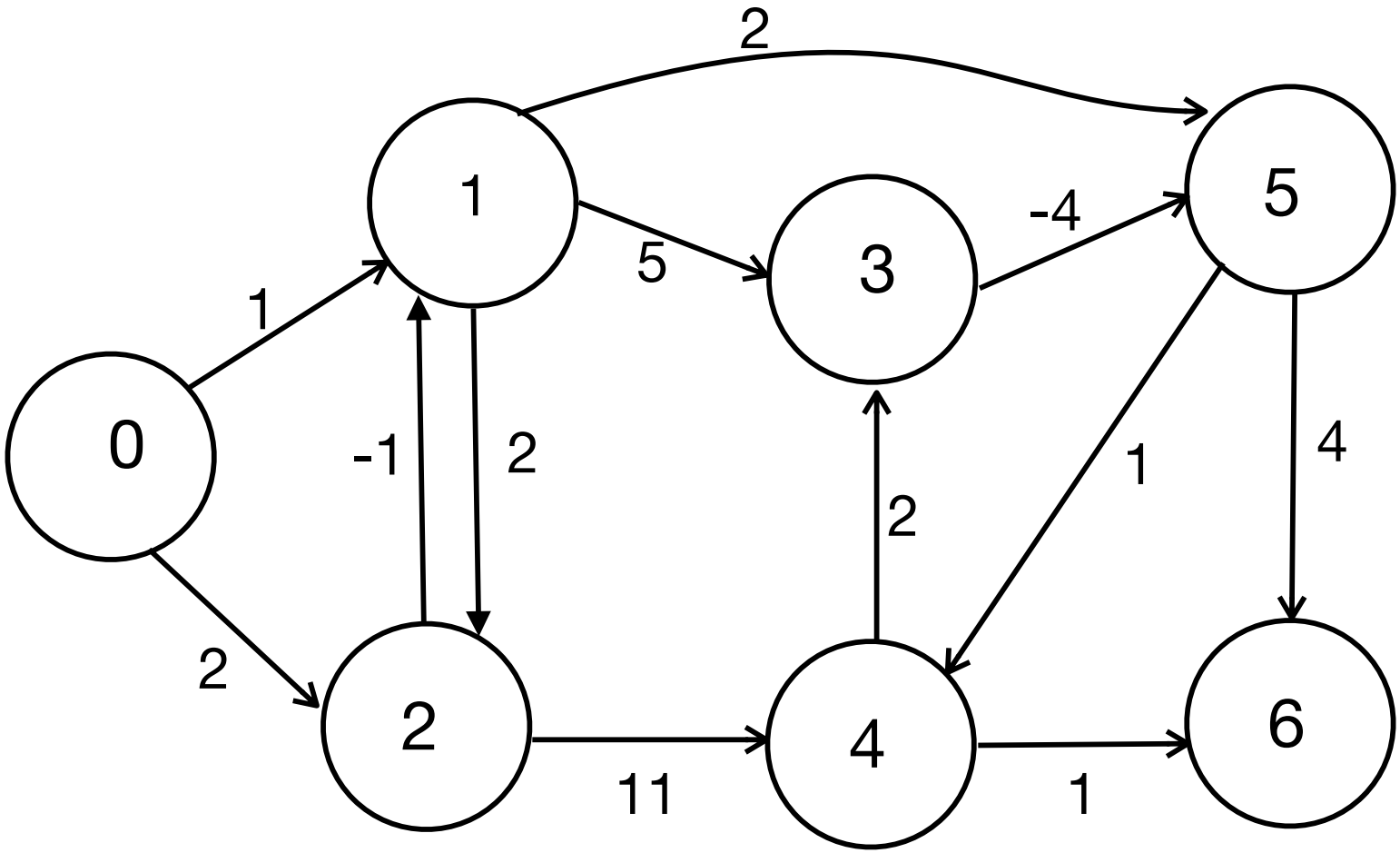
Let $u \xrightarrow{p} v$ denote the path p from u to v.

Paths in a weighted graph



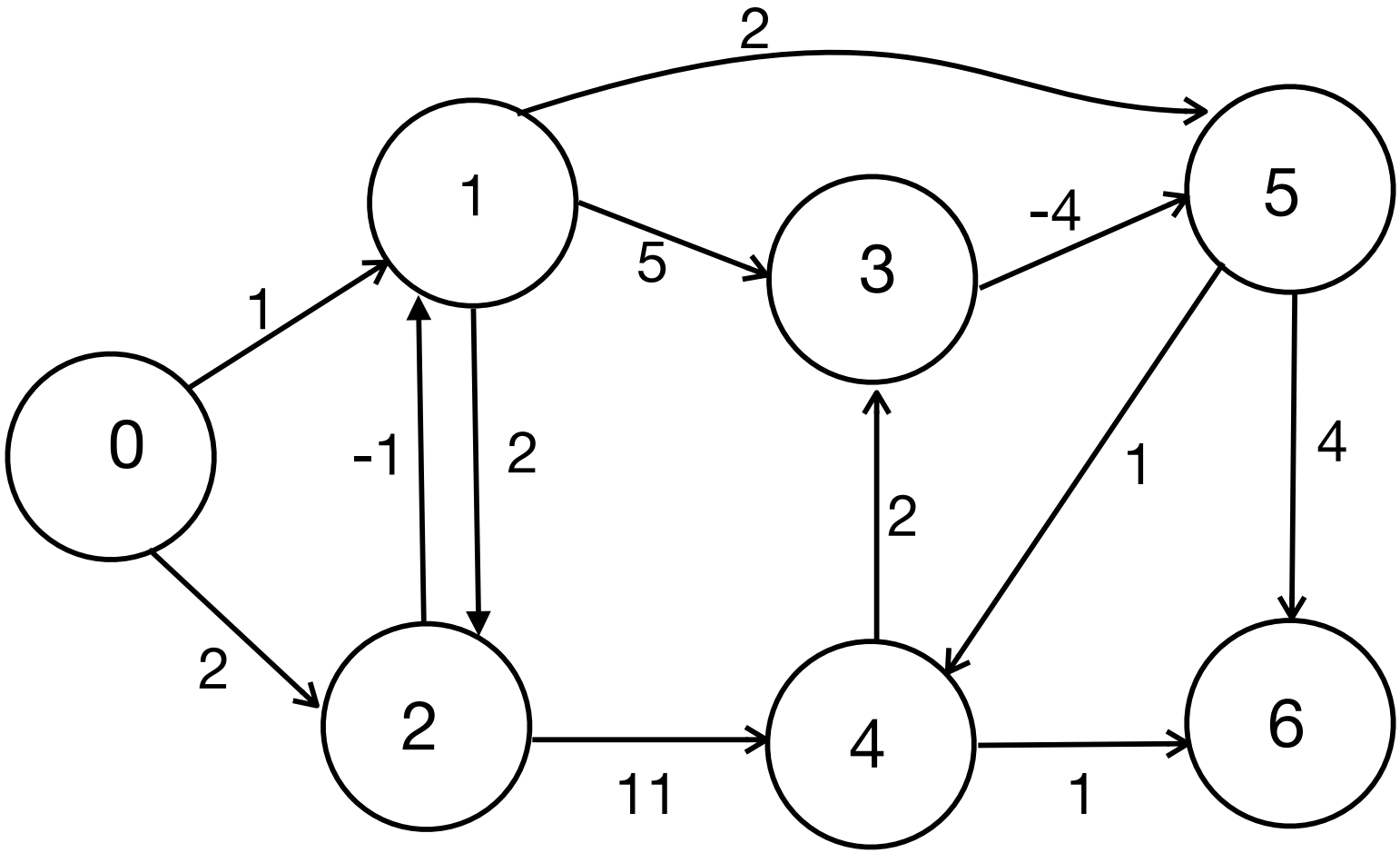
Several paths from u to v, $u \xrightarrow{p_1} v$, $u \xrightarrow{p_2} v$, $\ldots$ $u \xrightarrow{p_m} v$.

Paths in a weighted graph



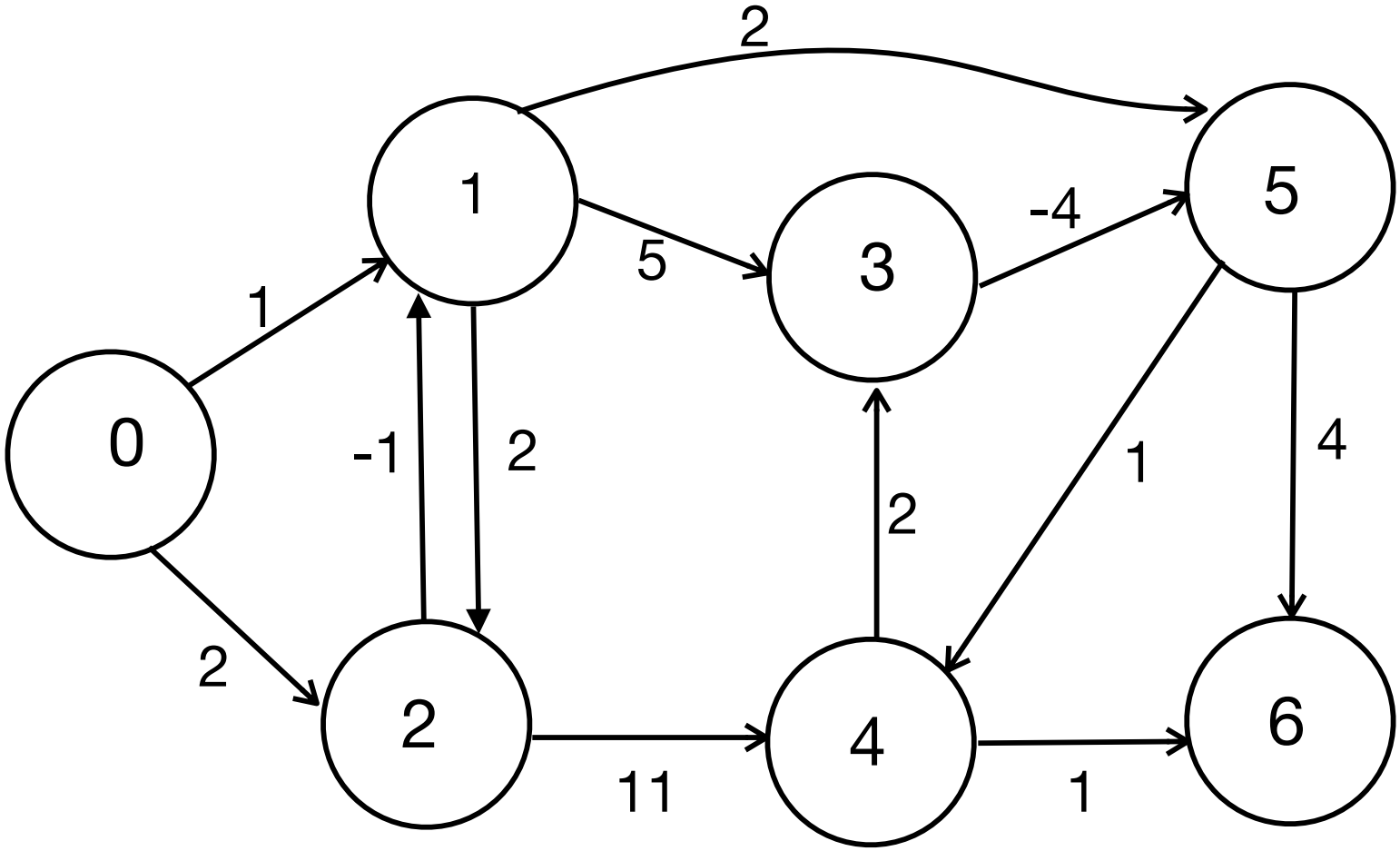
The *weight of a path* is sum of weights along its edges.

Paths in a weighted graph



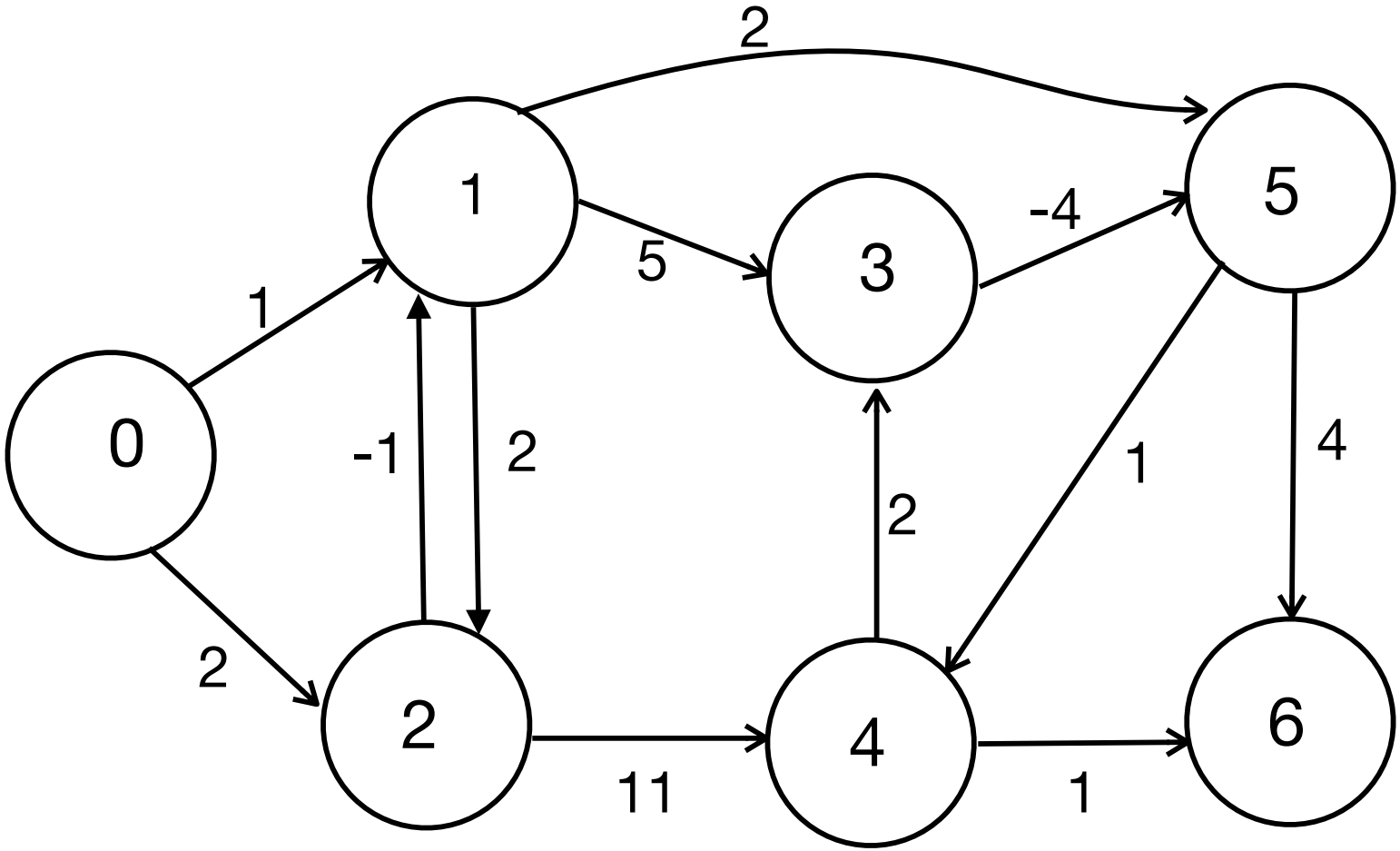
There are only finitely many such paths from u to v

Paths in a weighted graph



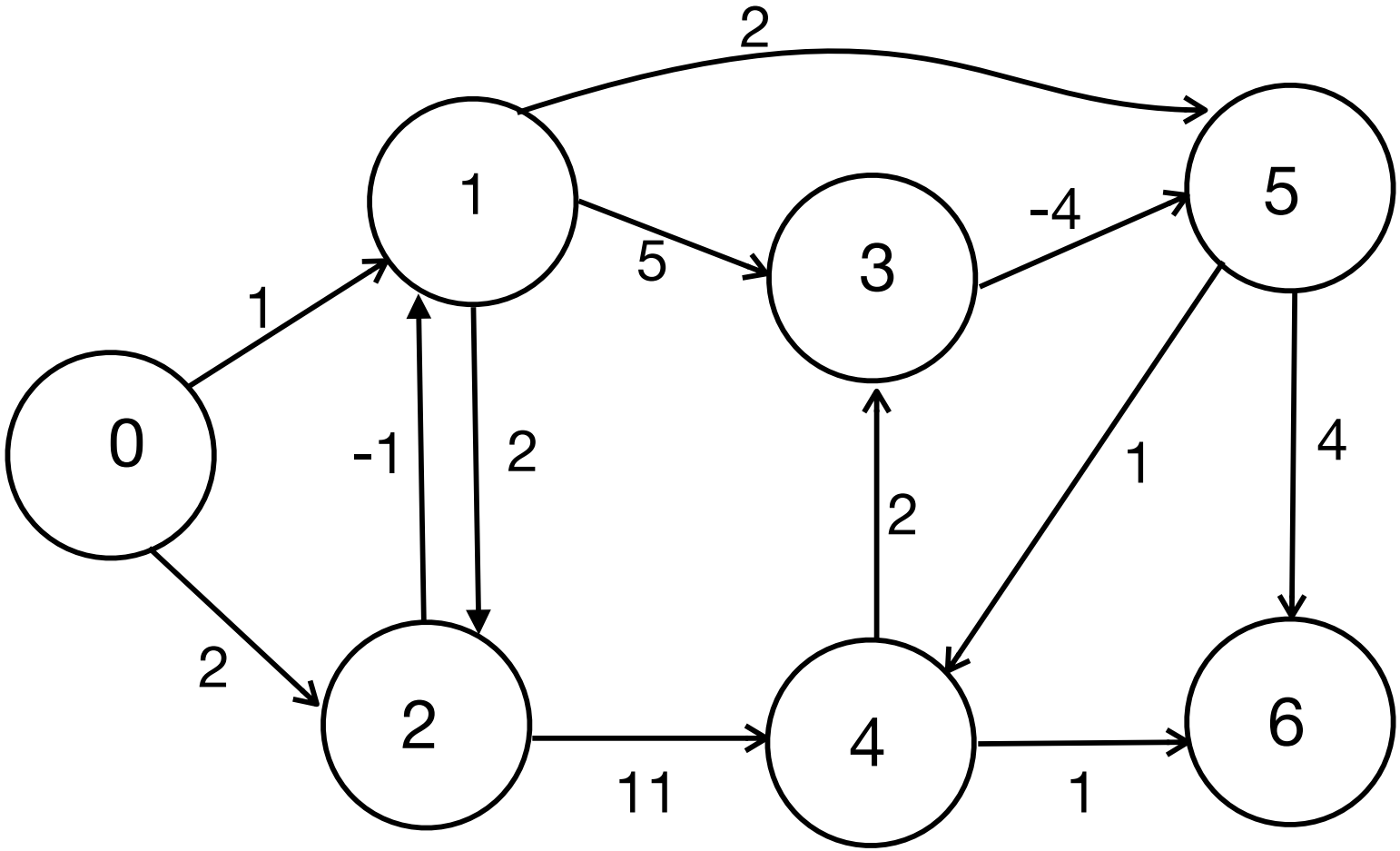
distance between u and v is $\delta(u, v) = \min_p w(u \xrightarrow{p} v)$.

Paths in a weighted graph



If there is no path from u to v, we define $\delta(u, v) = \infty$,

Paths in a weighted graph

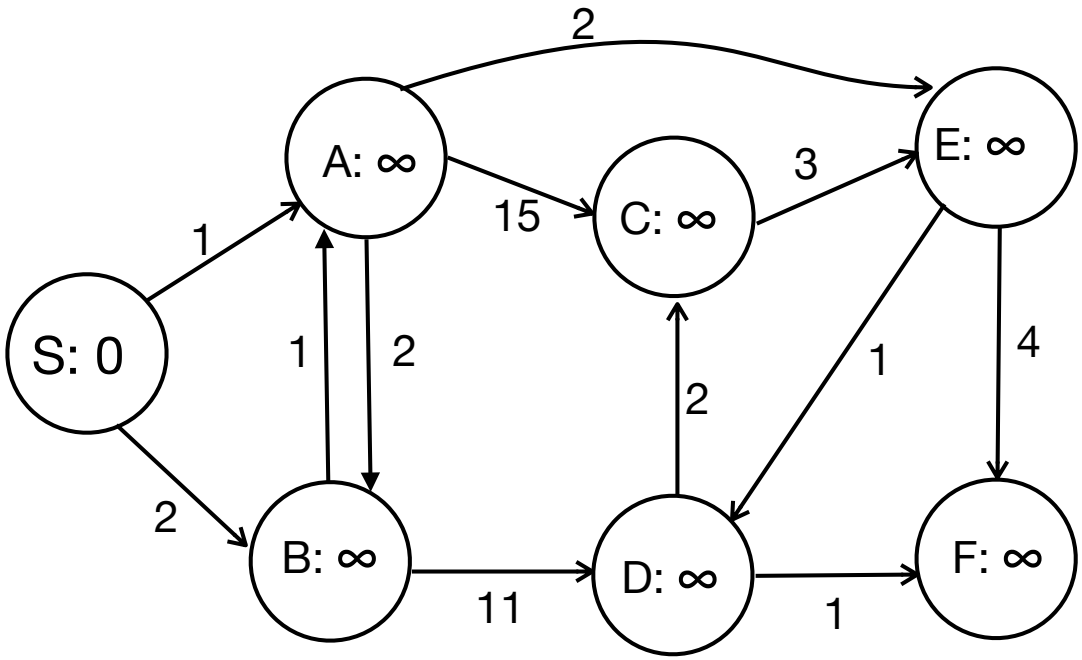


and $\delta(u, u) = 0$.

End of video

Relaxation

Initialization

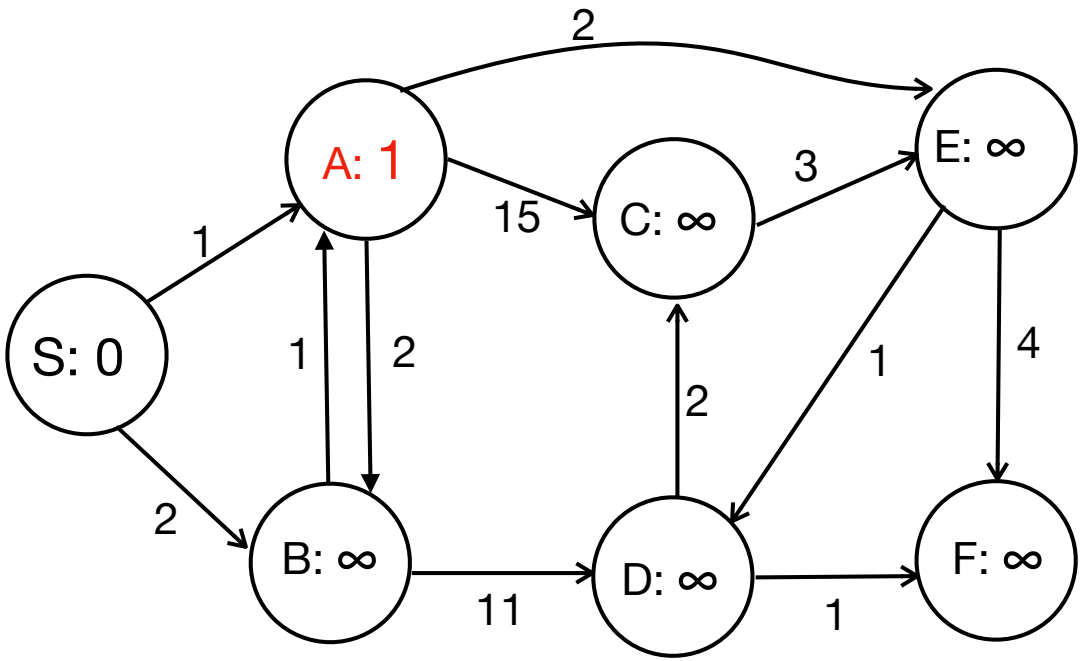


Notation $d_s(v) = d(v) = \delta(s, v)$.

Initial estimate $\delta(s, v)$ inside S: 0 for each $v \in V$.
Initially $d(s) = 0$, and otherwise $d(v) = \infty$.

Relaxation

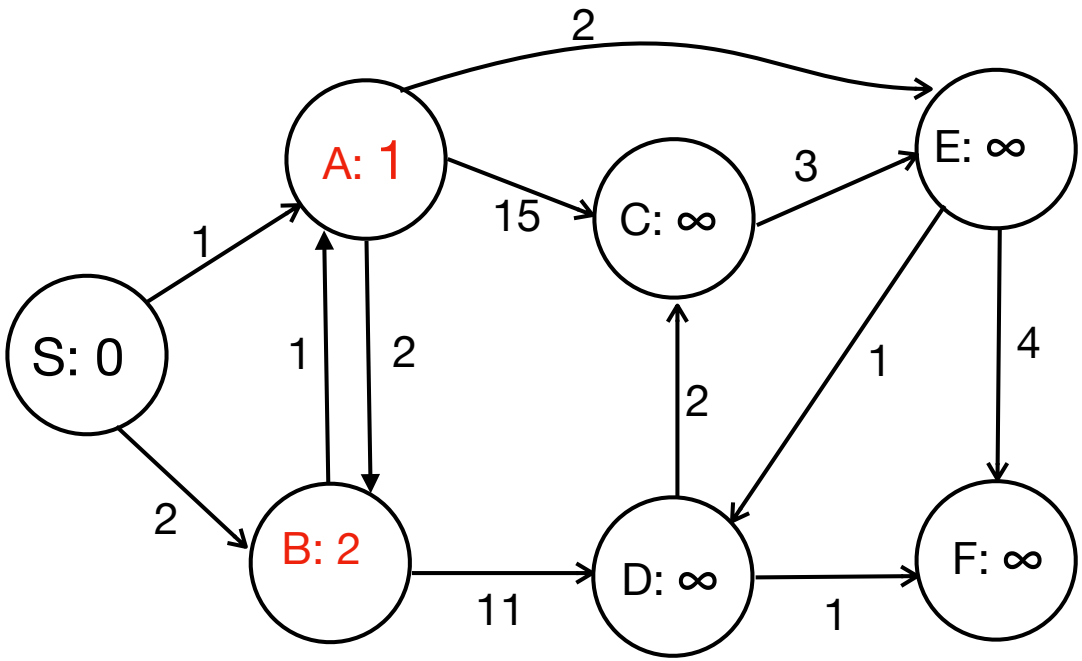
We may improve the estimates of A, B, C, and D.



$$\begin{aligned} d(A) &\leftarrow \min\{d(A),\ d(s) + w(s, A)\} \\ &= \min\{\infty,\ 0 + 1\} \\ &= 1 \end{aligned}$$

Relaxation

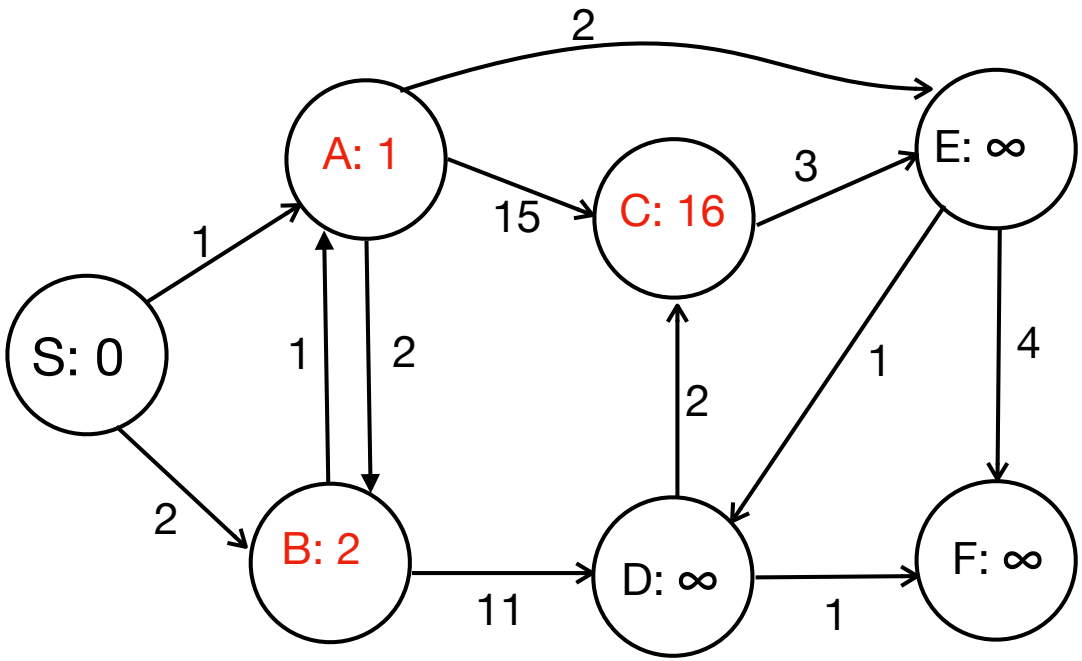
We may improve the estimates of A, B, C, and D.



$$\begin{aligned} d(B) &\leftarrow \min\{d(B),\ d(s) + w(s, B)\} \\ &= \min\{\infty,\ 0 + 2\} \\ &= 2 \end{aligned}$$

Relaxation

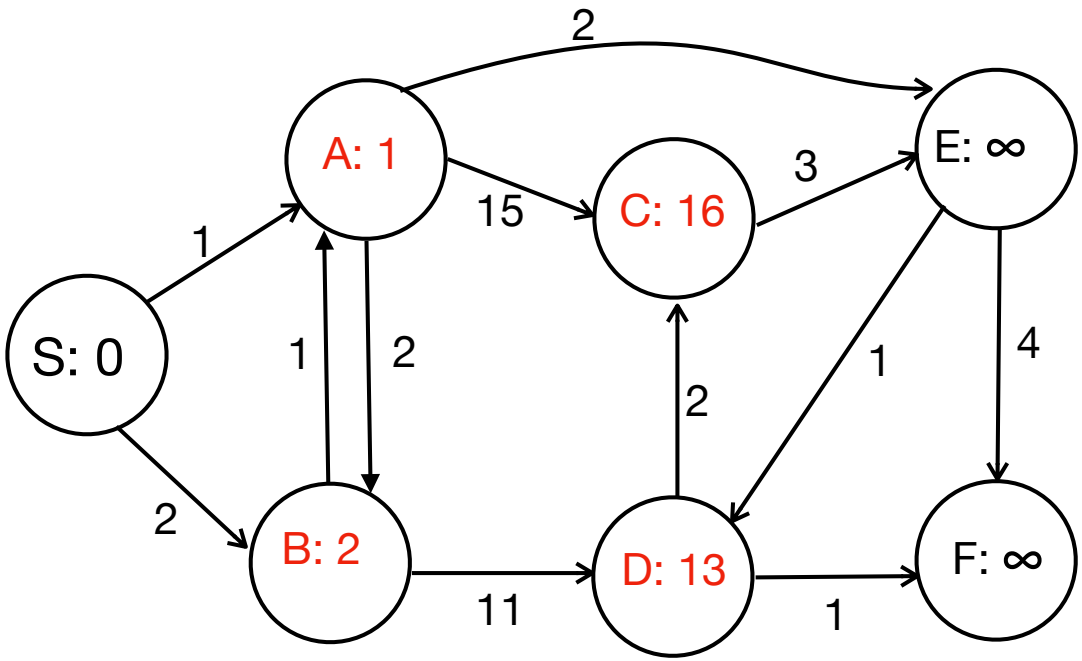
We may improve the estimates of A, B, C, and D.



$$\begin{aligned} d(C) &\leftarrow \min\{d(C),\ d(A) + w(A, C)\} \\ &= \min\{\infty,\ 1 + 15\} \\ &= 16 \end{aligned}$$

Relaxation

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



$$\begin{aligned} d(D) &\leftarrow \min\{d(D),\ d(B) + w(B, D)\} \\ &= \min\{\infty,\ 2 + 11\} \\ &= 13 \end{aligned}$$

Minimize by Relaxation

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

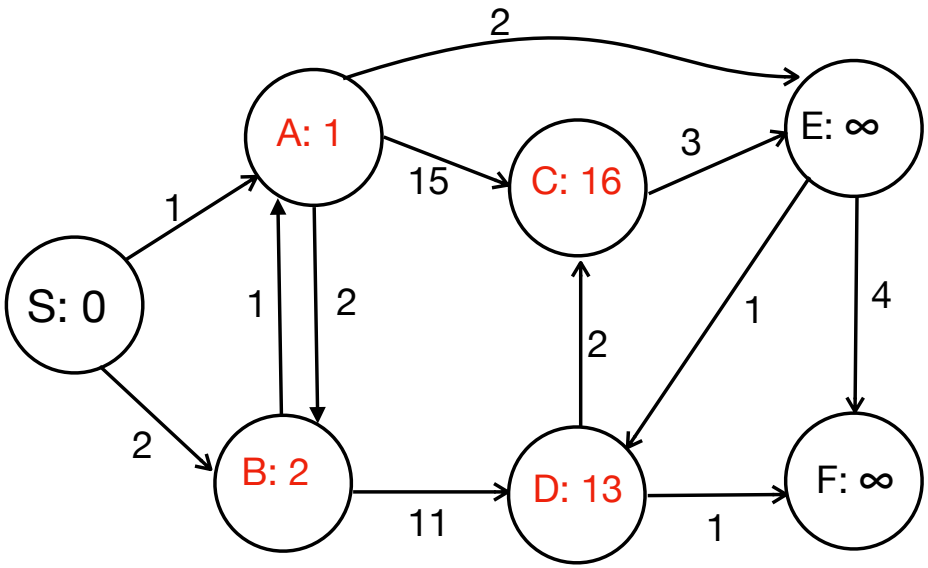
$$d(u) + w(u, v) < d(v)$$

then we can update

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

Minimization by Relaxation

It may not be clear what the best order or how many times to relax the vertices.



Notice, we computed C: 16 by $\min\{\infty, 1 + 15\} = 16$.
However, we now see that we should update C: 15 by $\min\{16, 13 + 2\} = 15$.

Minimization by Relaxation

In the following sections, we examine various algorithms. Each approach relaxes nodes using the same relaxation algorithm. The only difference is the order we visit the vertices to relax.

Relaxation Algorithm

Algorithm 1: Relax(u,v,w,d,pred)

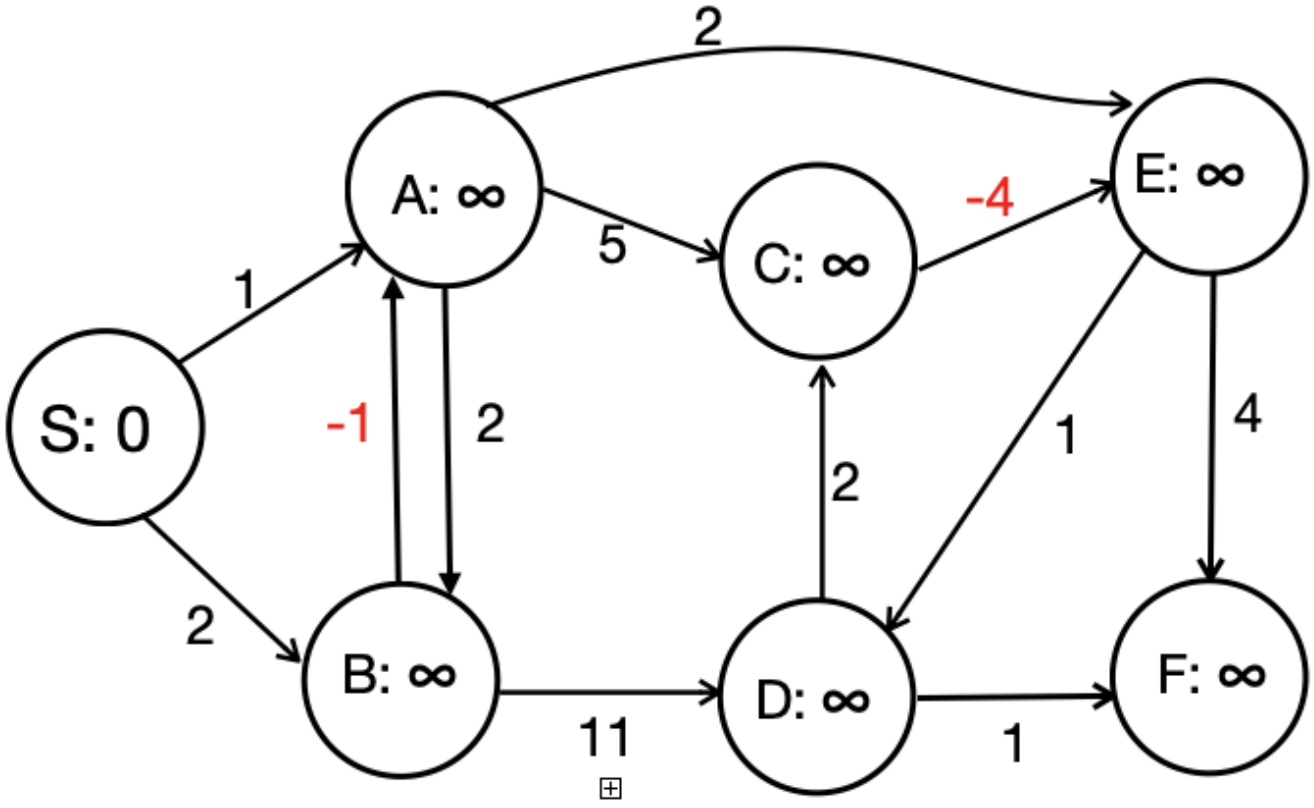
Input : u, v vertices
Input : w weight function
Input : d current estimate array
Input : $pred$ current predecessor array

1 **if** $d[v] > d[u] + w(u, v)$ **then**
2 $\quad d[v] \leftarrow d[u] + w(u, v)$
3 $\quad pred[v] \leftarrow u$
4 $\quad$ Maybe another action depending on caller.

Note: the code:
if $(d[v] > d[u] + w(u, v))$ *then* $d[v] \leftarrow d[u] + w(u, v)$ is sometimes written with fewer lines.

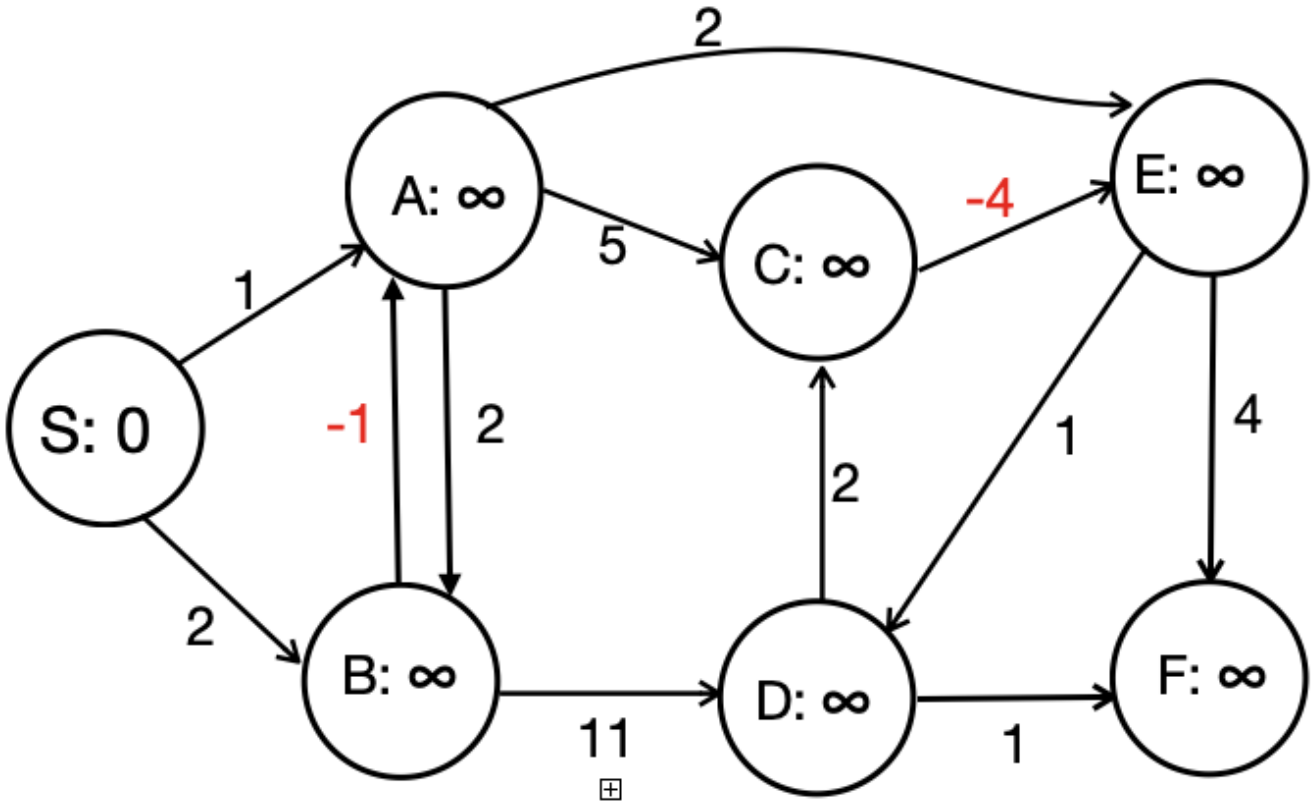
```
d[v] = min( d[v], d[u] + w(u,v))
```

Negative Cycles



What is $d(F)$?

Negative Cycles



What is $d(F)$?
We can make it as small as we want by including multiple loops around $C \to E \to D \to C$.

End of video

Bellman-Ford

Bellman-Ford

Algorithm 2: 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

```
1 Initialize
2 repeat |V| − 1 times
3 |   for (u, v) ∈ E do
4 |   |   Relax(u,v,w,d,pred)
  /* Now check for negative weight cycles       */
5 for (u, v) ∈ E do
6 |   if  d[v] > d[u] + w(u, v) then
7 |   |   report negative weight cycle
```

Optimizations of Bellman-Ford

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.

Complexity of Bellman-Ford

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

End of video

Dijkstra

Dijkstra

- 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.
- Dijkstra assumes positive weights.

Dijkstra

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, $\mathit{priority}(s) = 0$, 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.

Dijkstra

There are two popular types of heaps: `binary-heap` and `fibonacci-heap`.

- If you are writing it yourself, `binary-heap` is easier to implement.

Dijkstra

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.

Dijkstra

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,

Dijkstra

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.

Algorithm 3: Dijkstra(s,V,Adj,w)

Input : s a starting vertex, V set of verticies
Input : Adj adjacency list, w a weight function

```
1  pred[s] ← None
2  d[s] ← 0
3  for v ∈ V \ {s} do
4  |   d[v] ← ∞
5  S ← ∅
6  Q ← INITIALIZE-HEAP(V)
7  while Q ≠ ∅ do
8  |   u ← EXTRACT-MIN(Q)     // removing u from Q
9  |   S ← S ∪ {u}
10 |   for v ∈ Adj[u] \ S do
11 |   |   Relax(u, v, w, d, pred, Q)
12 return d
```

Algorithm 4: 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

```
1 if d[v] > d[u] + w(u, v) then
2 |   d[v] ← d[u] + w(u, v)
3 |   REDUCE-MIN(Q, v, d[v])
4 |   pred[v] ← u
```

Complexity of Dijkstra

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

line 6 — is $O(|V|)$.

line 7 — the `while` loops $O(|V|)$ times.

line 8 — is $O(|V| \times \log(|V|))$.

line 10 — loops $O(|E|)$ total, don't be confused about the `while` loop.

line 11 — is executed $O(|E|)$ with amortized complexity $O(1)$, thus its complexity is $O(|E|)$.

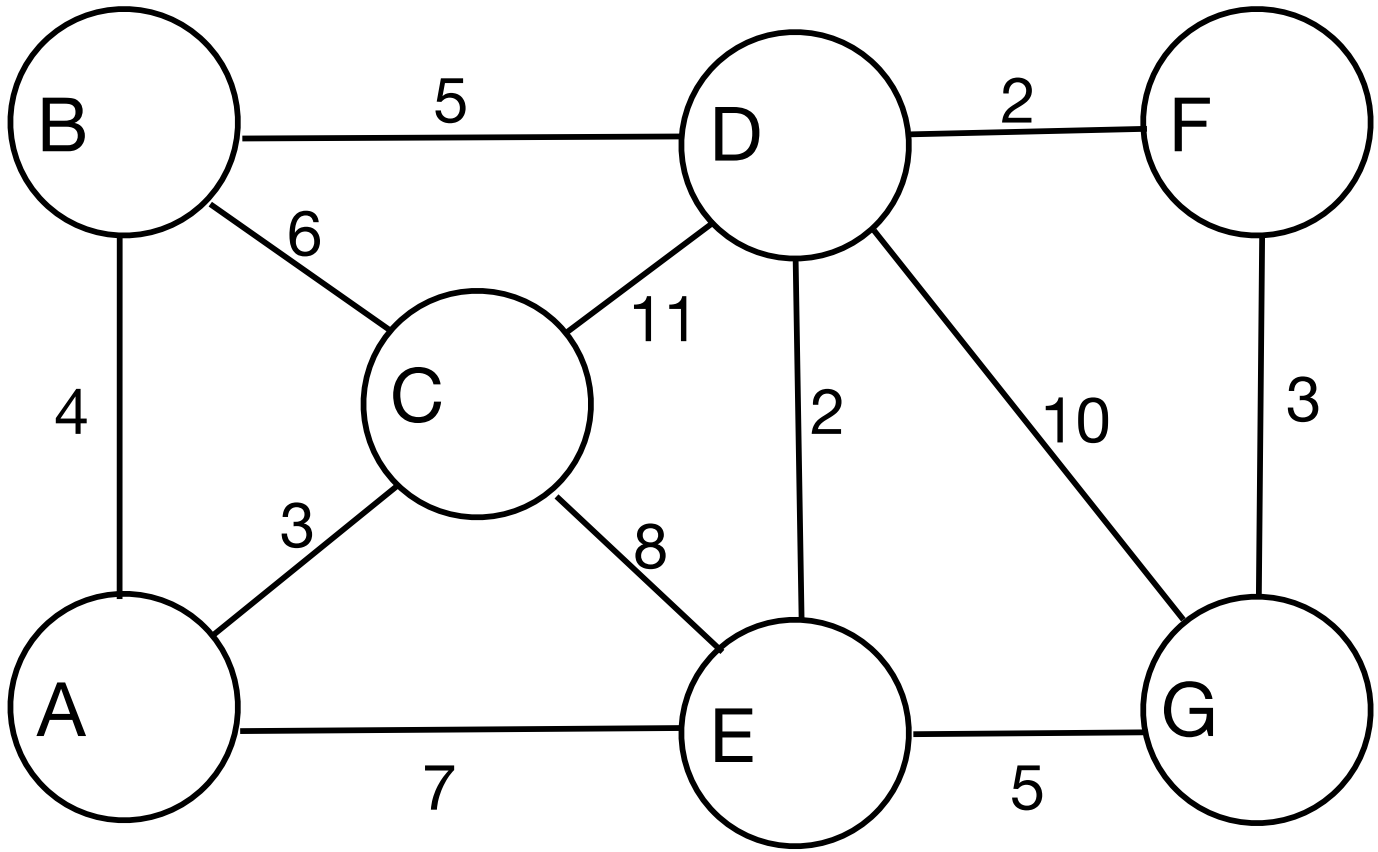
End of video

Dijkstra Example

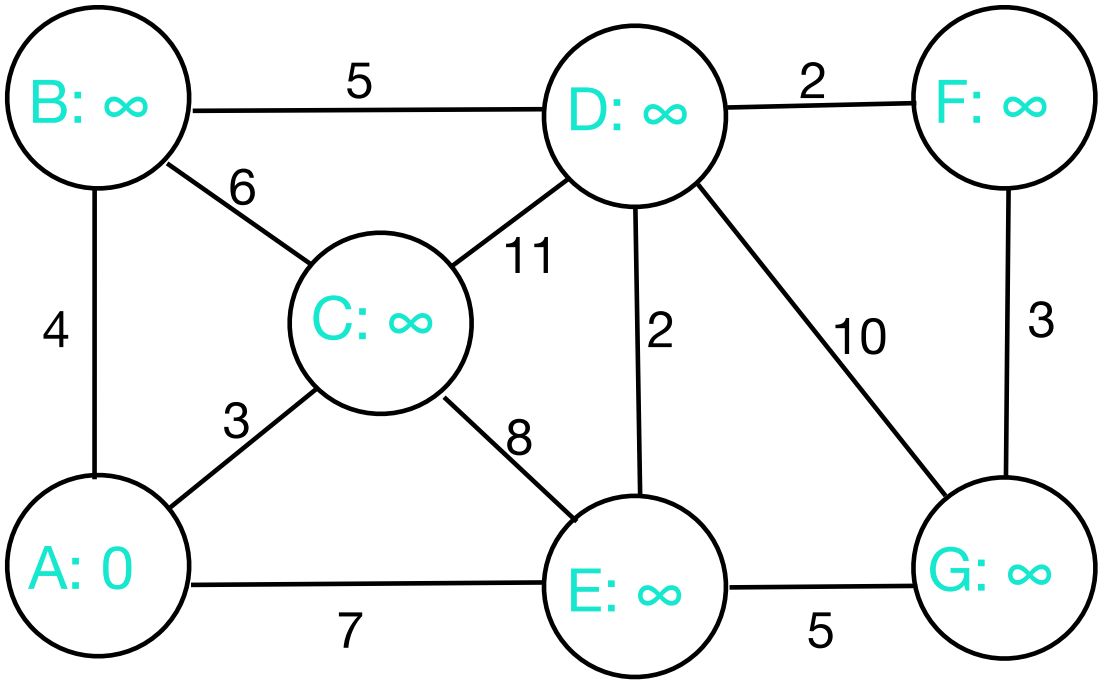
Dijkstra example $A \rightarrow F$

In this example, we will look at the shortest path between two vertices.

We want to find the shortest path from A to F.



Dijkstra example $A \rightarrow F$



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

Dijkstra example $A \rightarrow F$

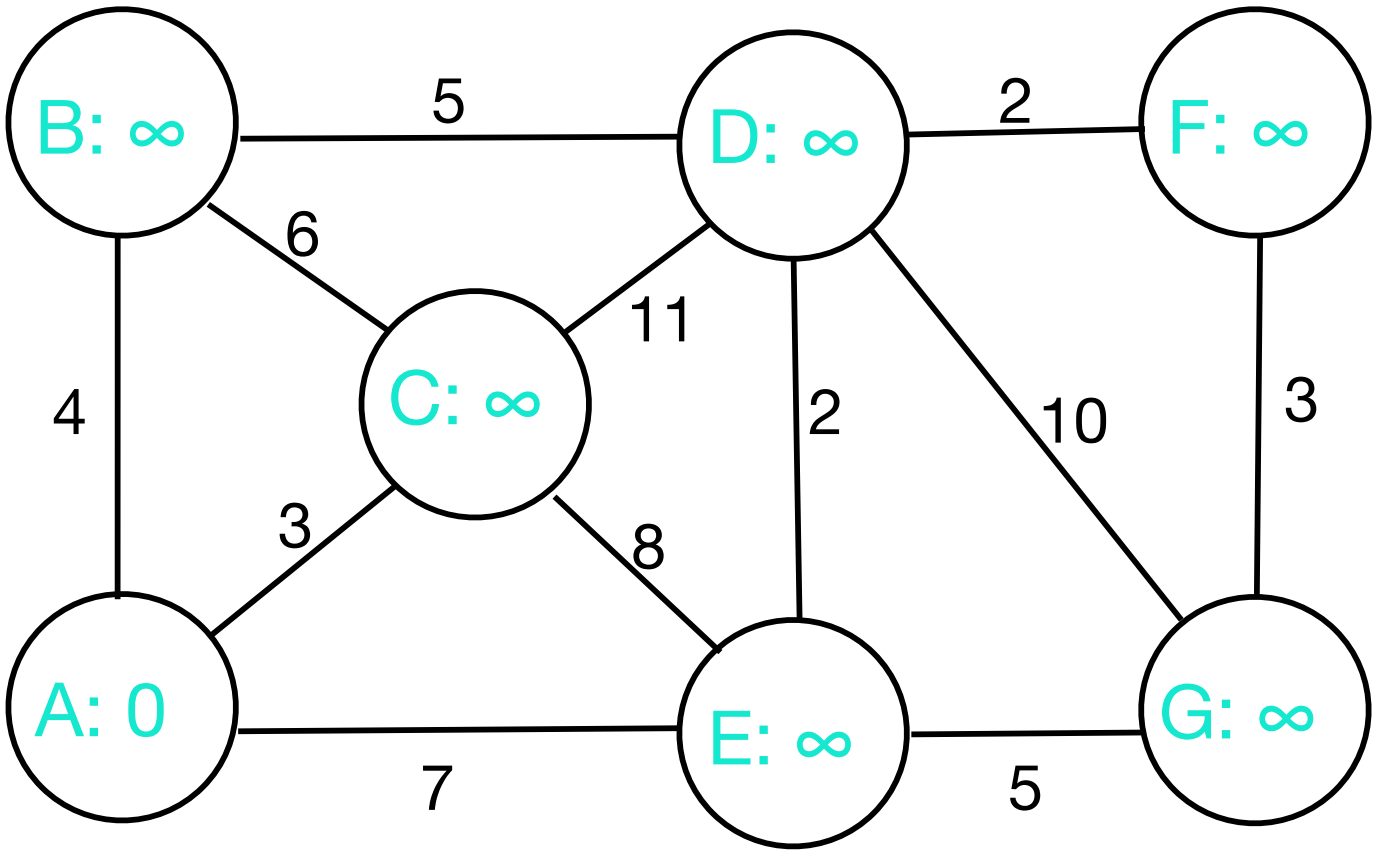
Dealing with ∞.

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. Else you must always know (or be able to check) whether a given vertex is found in the priority queue before relaxing it.

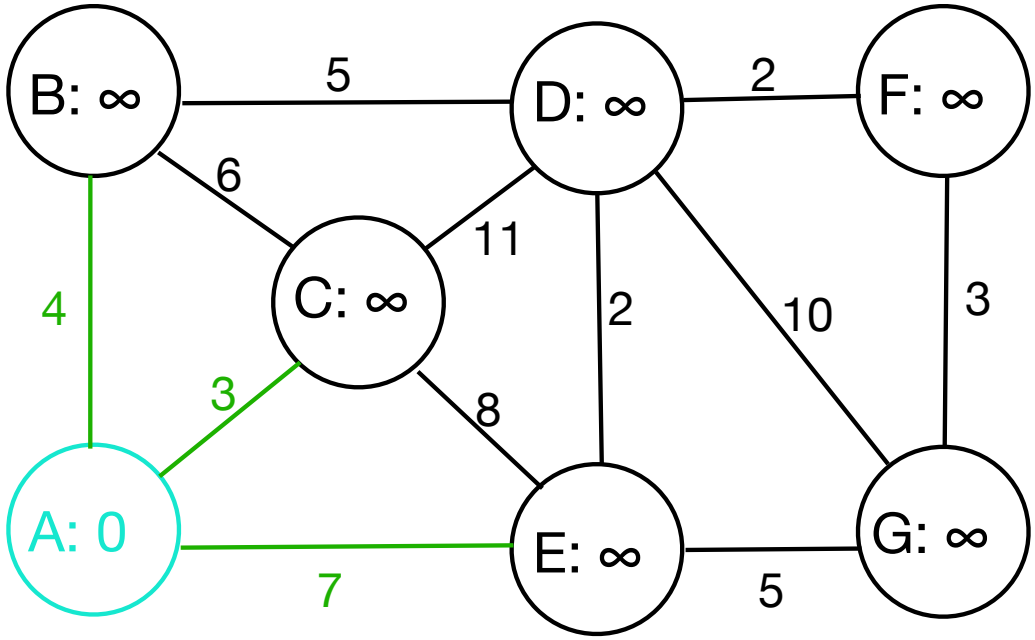
So in the latter case, the relax step will contain one case for "not yet registered", and a second case for "already registered".

Dijkstra example $A \rightarrow F$

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

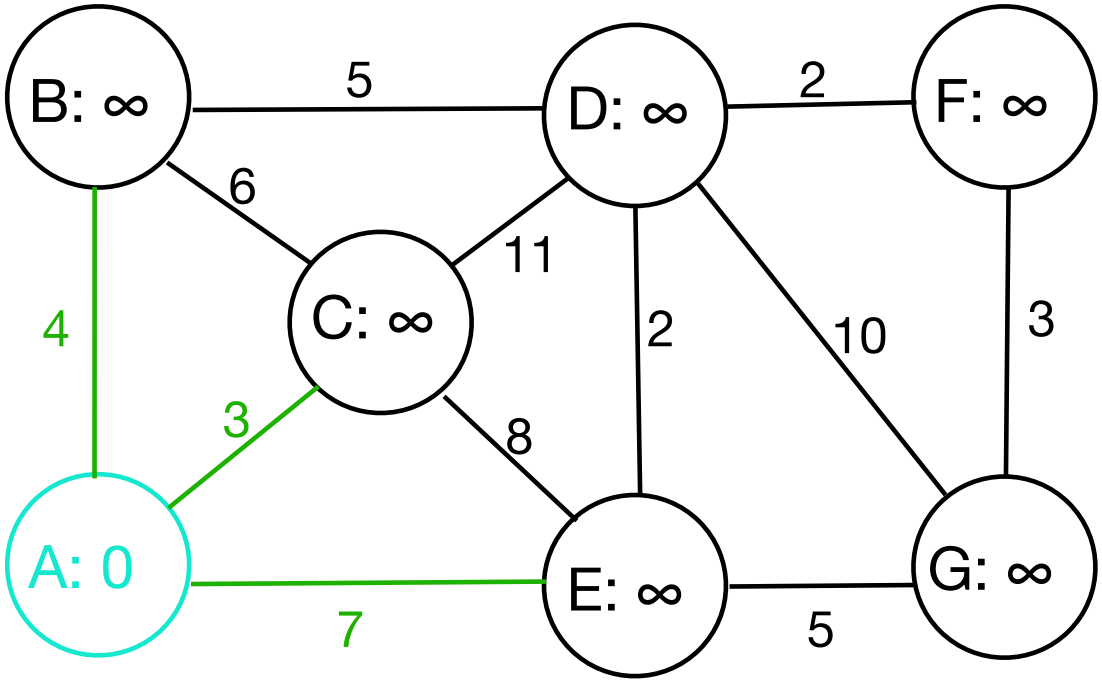


Dijkstra example $A \to F$



- Select the vertex with minimum current $d[]$ value.
- EXTRACT-MIN $\to A$, because $d[A] = 0$.
- Examine all *outgoing* edges of A, as no vertex has yet been visited.
- We may examine the outgoing edges in any order.
- In the image, edges have weights 4, 3, and 7.

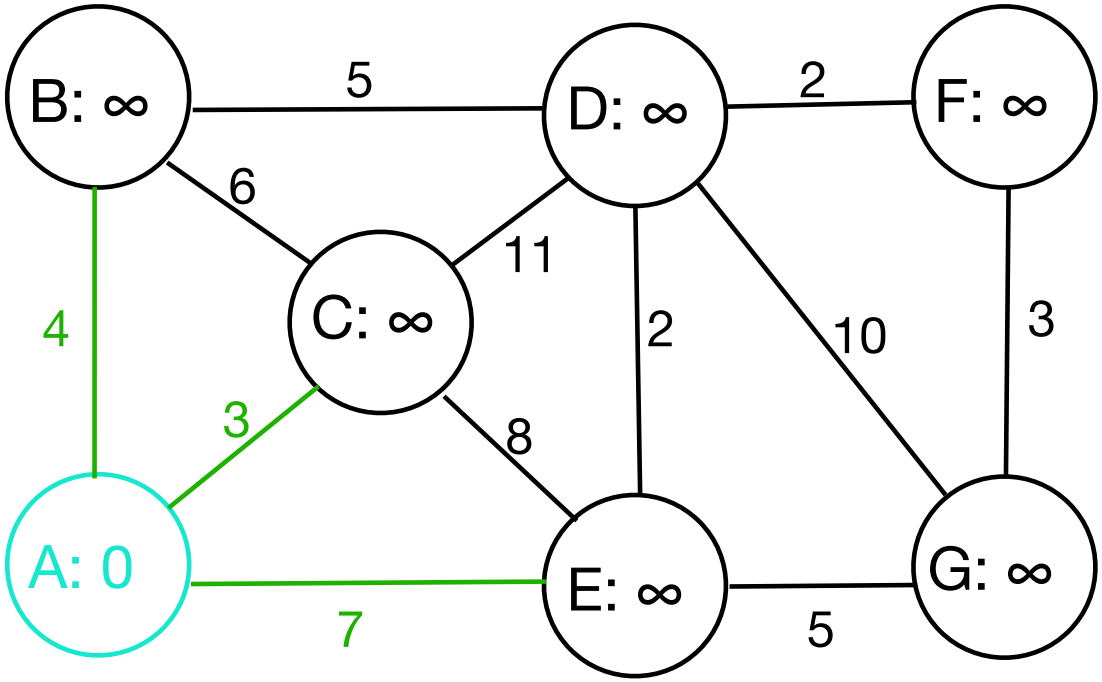
Dijkstra example $A \to F$



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

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

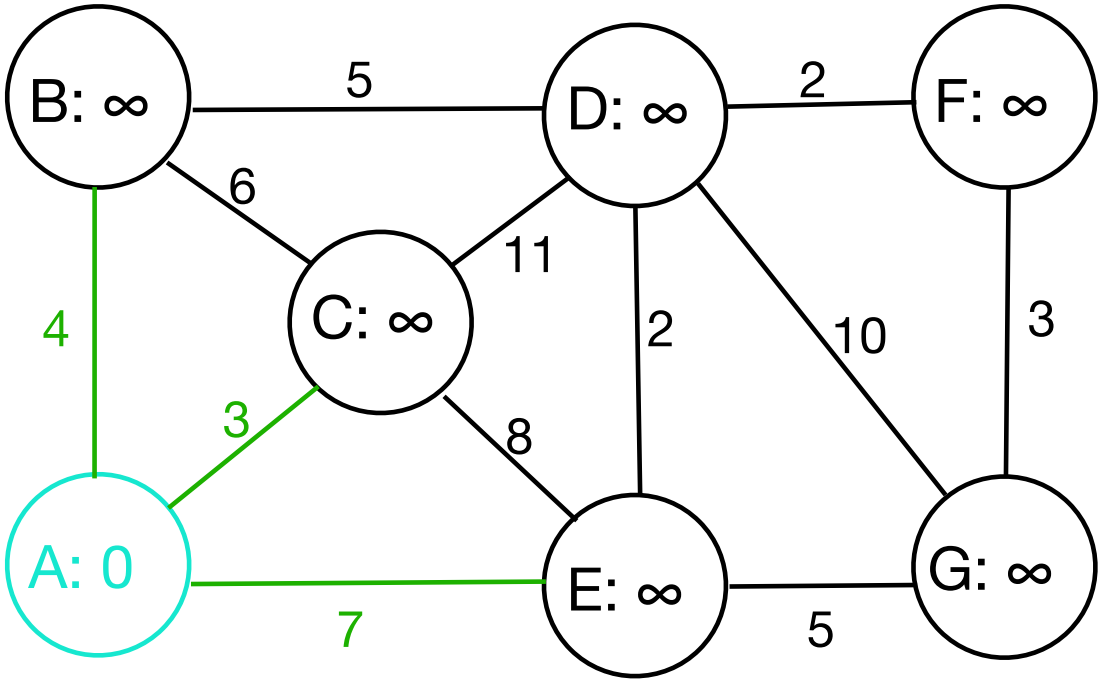
Dijkstra example $A \to F$



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

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

Dijkstra example $A \to F$

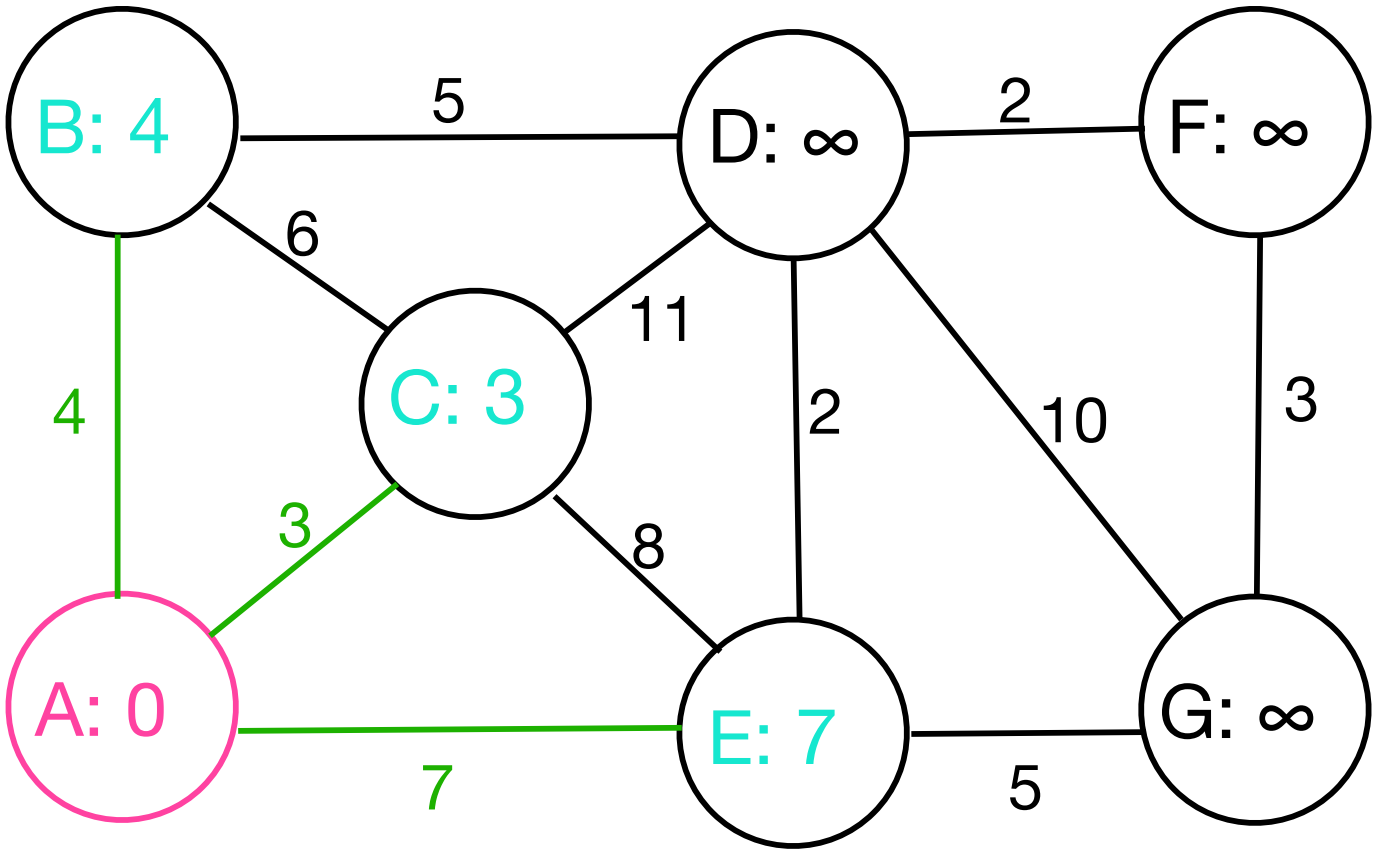


We relax vertices B, C, and E and mark A as visited.

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

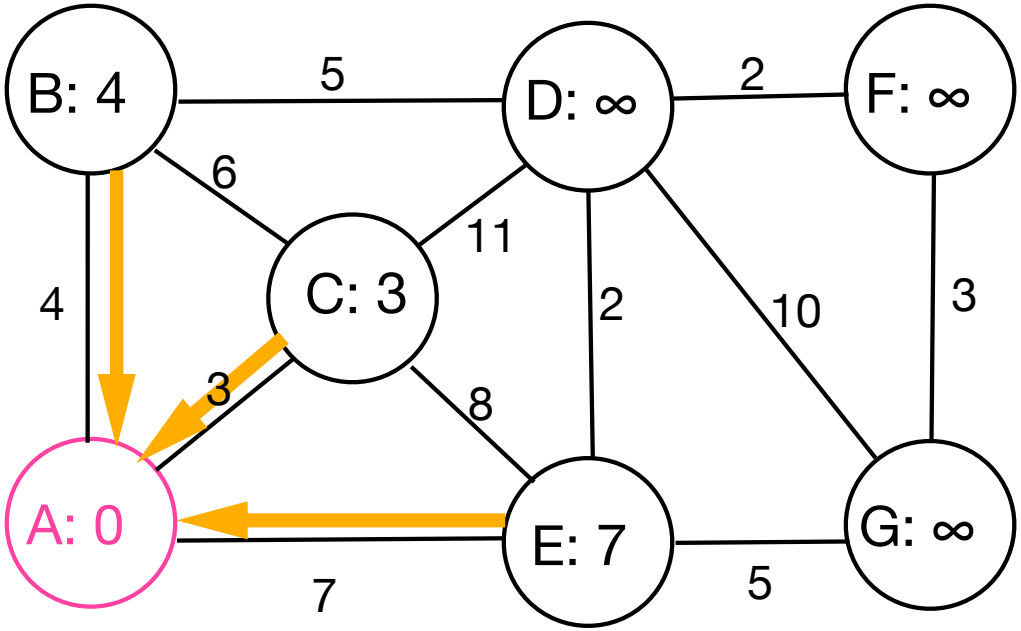
Dijkstra example $A \rightarrow F$

We indicate A as visited as:  .



Dijkstra example $A \rightarrow F$

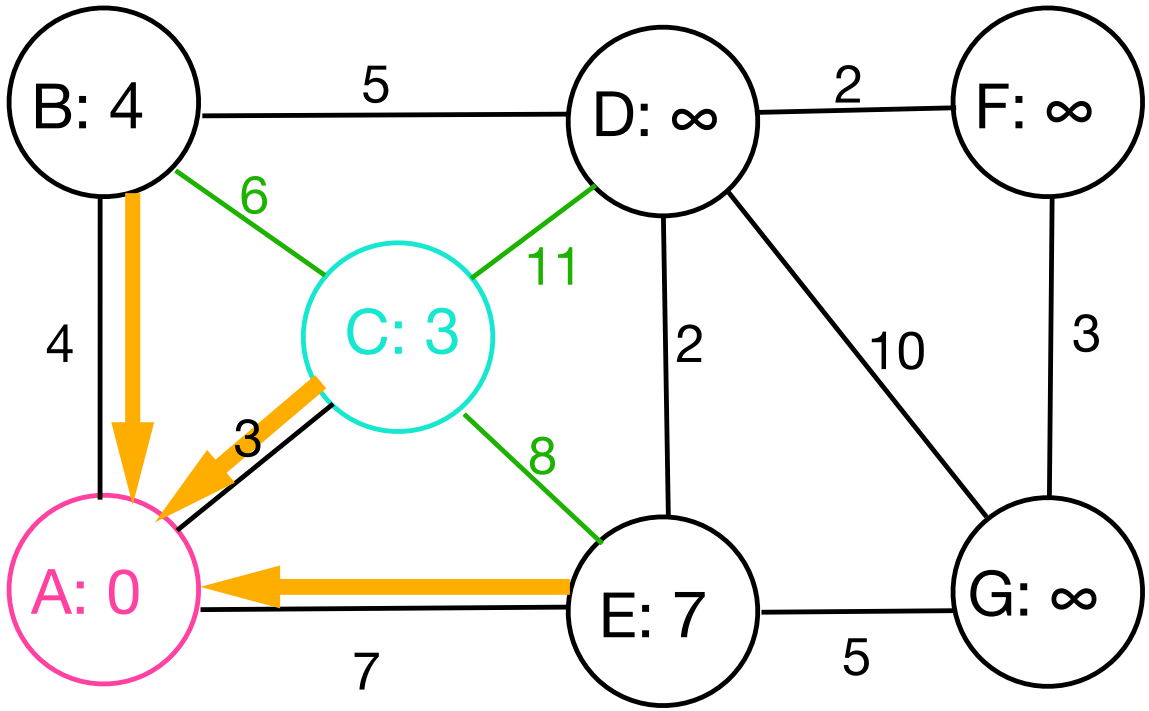
Update `pred` whenever $d[]$ changes. We examined B, C, and E, so $d[B]$, $d[C]$, and $d[E]$ change.



Update three predecessors.

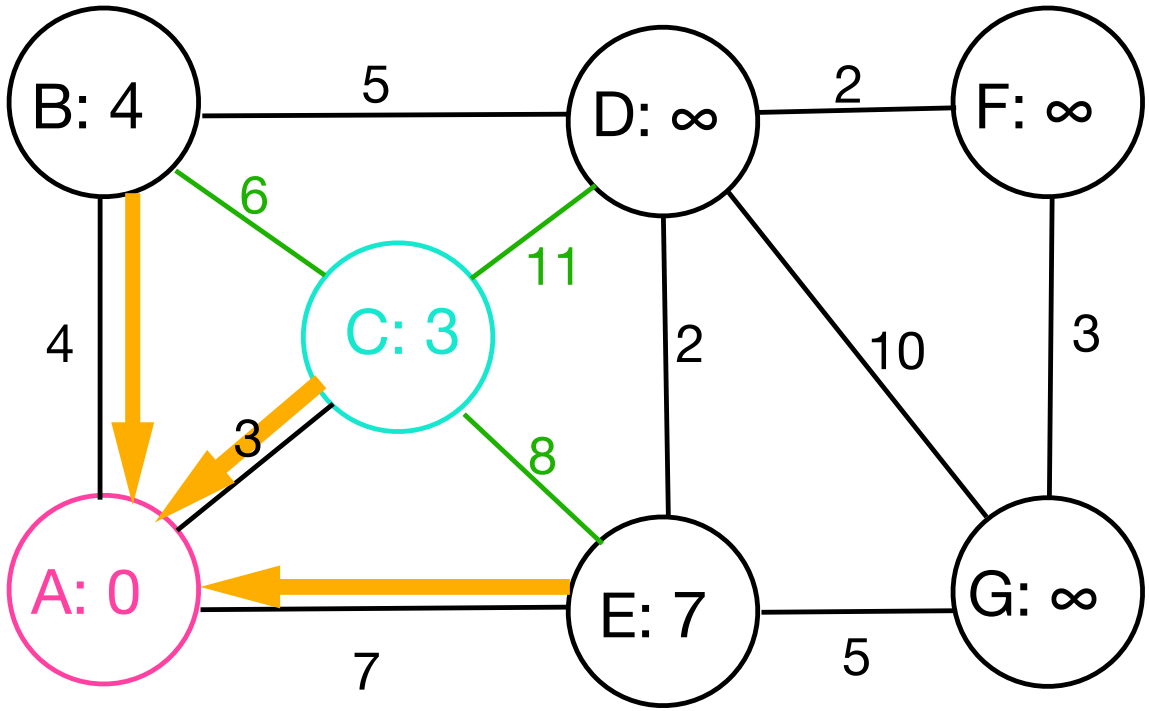
$$pred[B] \leftarrow A \qquad B \xrightarrow{pred} A$$
$$pred[C] \leftarrow A \qquad C \xrightarrow{pred} A$$
$$pred[E] \leftarrow A \qquad C \xrightarrow{pred} A$$

Dijkstra example $A \rightarrow F$



- Extract the minimum from $d[]$; ignore visited vertices.
- EXTRACT-MIN $\rightarrow C$, because $d[C] = 3$.
- Examine all *outgoing* edges of C
- Ignoring edges leading to node A. A is already visited.
- Edges leading to B, D, and E; weights: 6, 11, and 8.

Dijkstra example $A \to F$

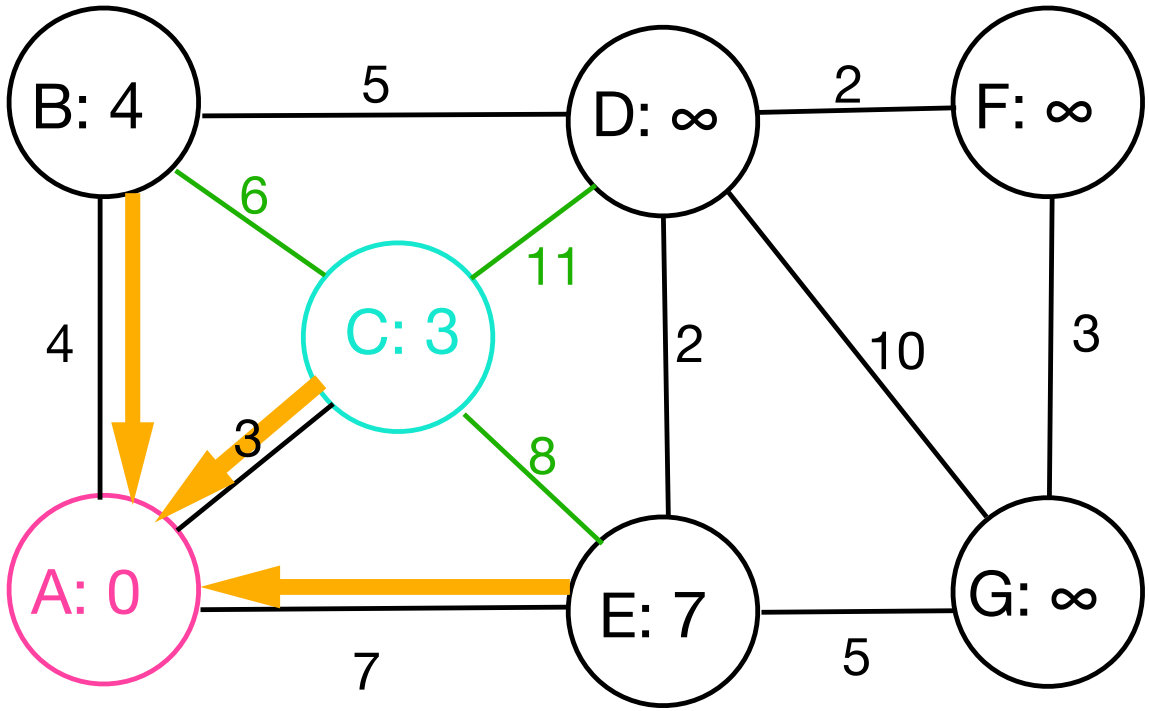


We now relax $d[B]$, $d[D]$, and $d[E]$:

$$\begin{aligned} d[B] &= \min\{d(B), d(C) + w(C, B)\} \\ &= \min\{4, 3 + 6\} \\ &= 4 \end{aligned}$$

$d[B]$ did not change

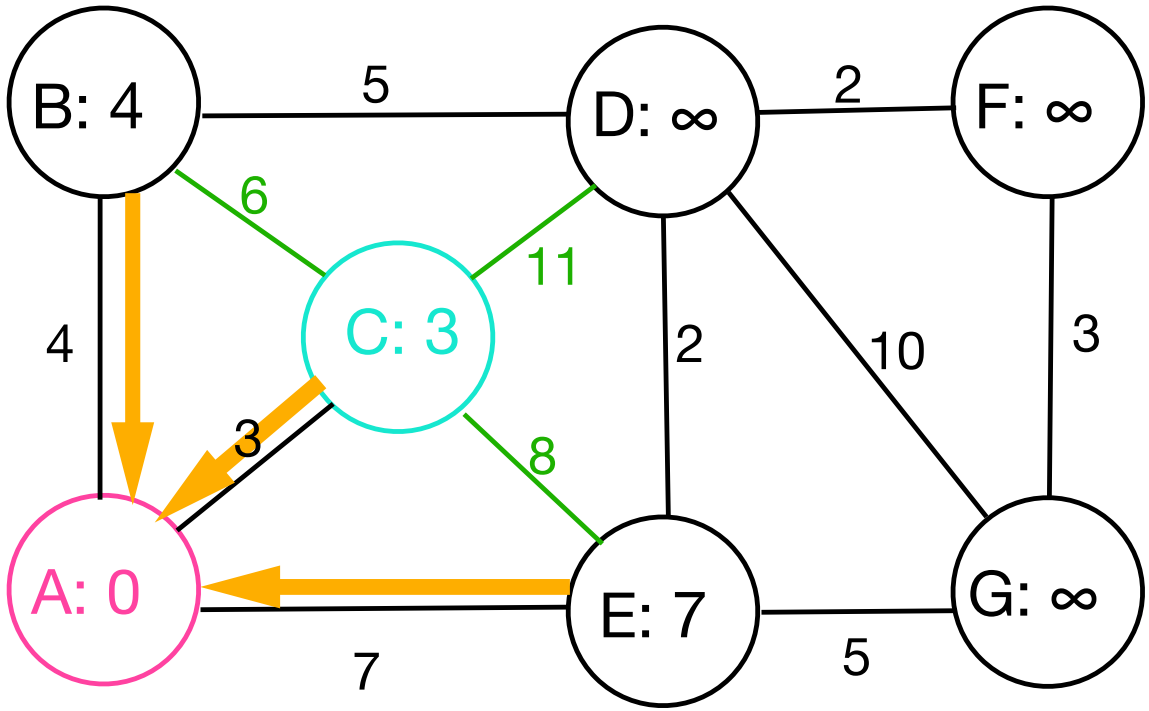
Dijkstra example $A \to F$



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

$$\begin{aligned} d[D] &= \min\{d(D), d(C) + w(C, D)\} \\ &= \min\{\infty, 3 + 11\} \\ &= 14 \qquad d[D] \text{ changed so } pred[D] \leftarrow C \end{aligned}$$

Dijkstra example $A \to F$

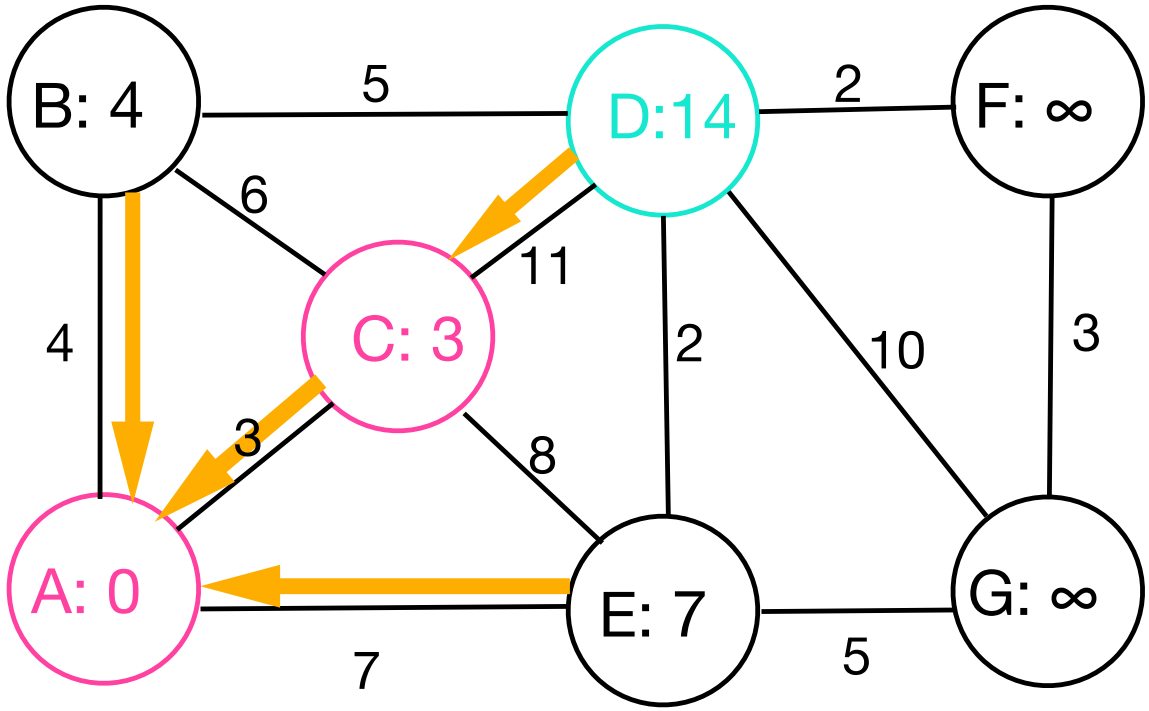


We now relax $d[B]$, $d[D]$, and $d[E]$:

$$\begin{aligned} d[E] &= \min\{d(E), d(C) + w(C, E)\} \\ &= \min\{7, 3 + 8\} \\ &= 7 \end{aligned}$$

$d[E]$ did not change

Dijkstra example $A \rightarrow F$

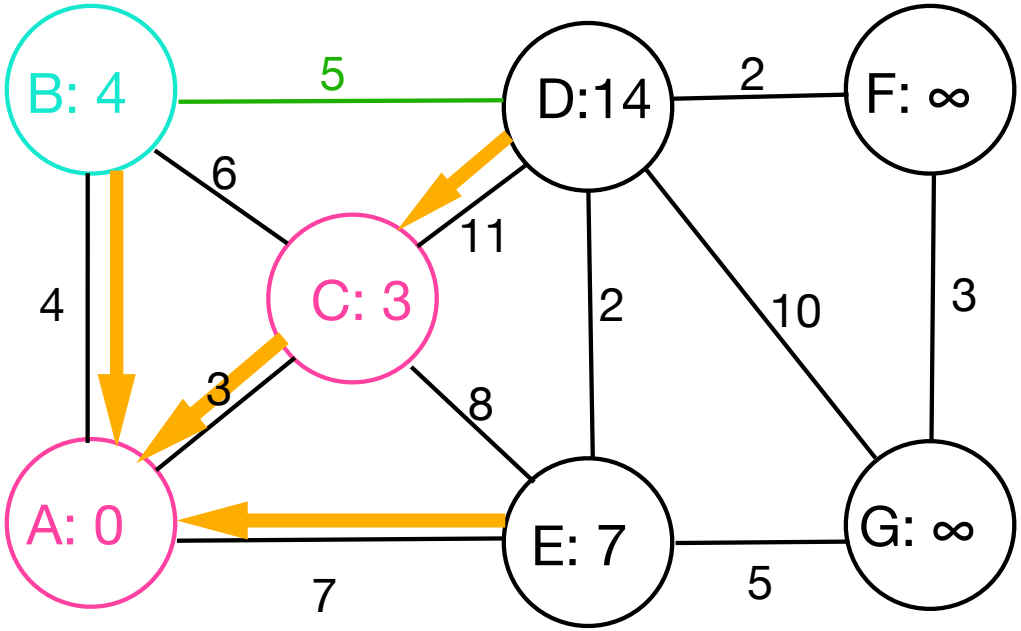


Update `pred[D]` because `d[D]` changed.

$$pred[d] \leftarrow C \qquad D \xrightarrow{pred} C$$

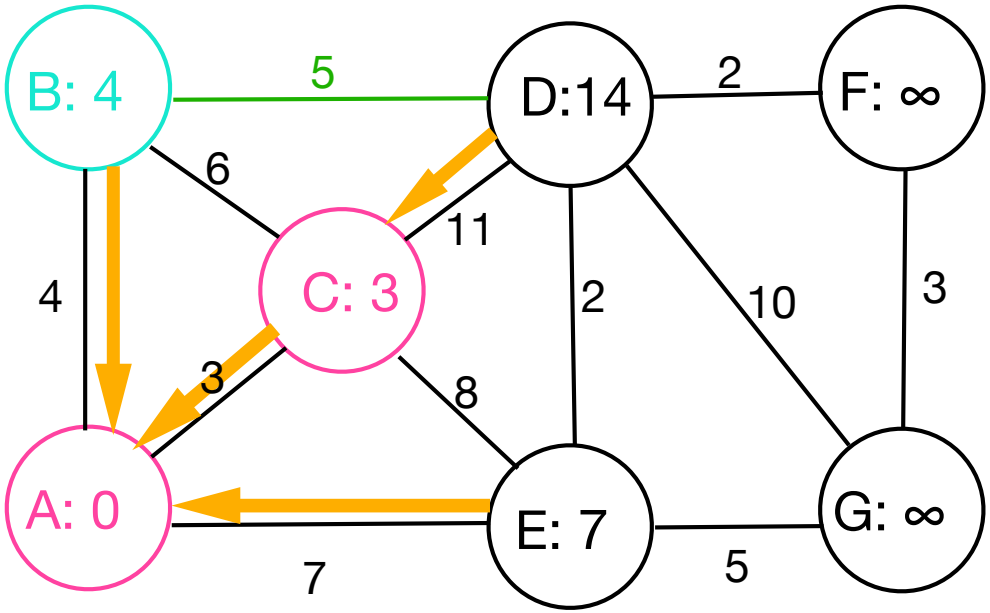
The vertex  goes into the visited list.

Dijkstra example $A \to F$



- Extract minimum, ignore already visited vertices.
- EXTRACT-MIN $\to B$, because $d[B] = 4$ is minimum.
- We now examine all *outgoing* edges, only one in this case,
- because A and C are already visited.

Dijkstra example $A \to F$



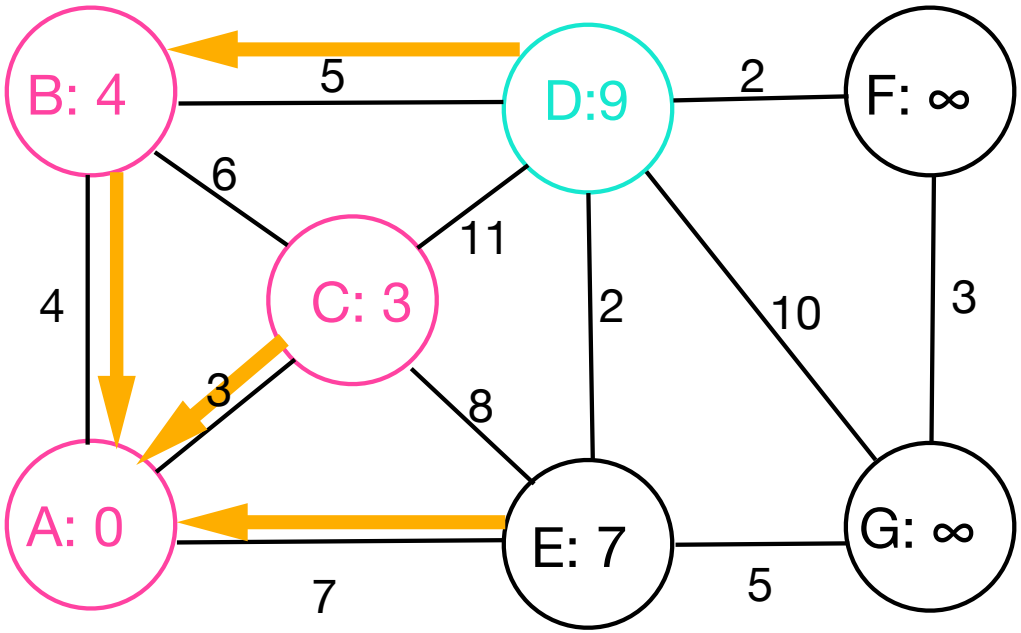
We relax D.

$$\begin{aligned} d[D] &= \min\{d(D), d(B) + w(B, D)\} \\ &= \min\{14, 4 + 5\} \\ &= 9 \end{aligned}$$

`d[D]` changed, so $pred[D] \leftarrow B$.

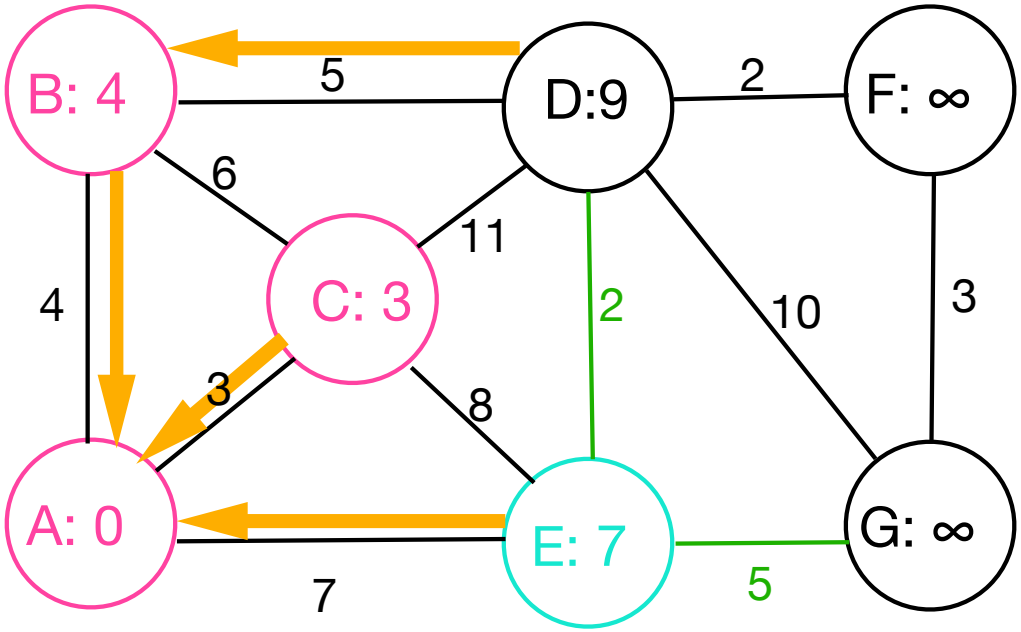
We are finished with B, putting it into the visited list.

Dijkstra example $A \to F$



- EXTRACT-MIN $\to E$, because $d[E] = 7$ is the minimum.
- Now visit edges from E to unvisited vertices: D and G
- Ignore edges to  and  .

Dijkstra example $A \to F$

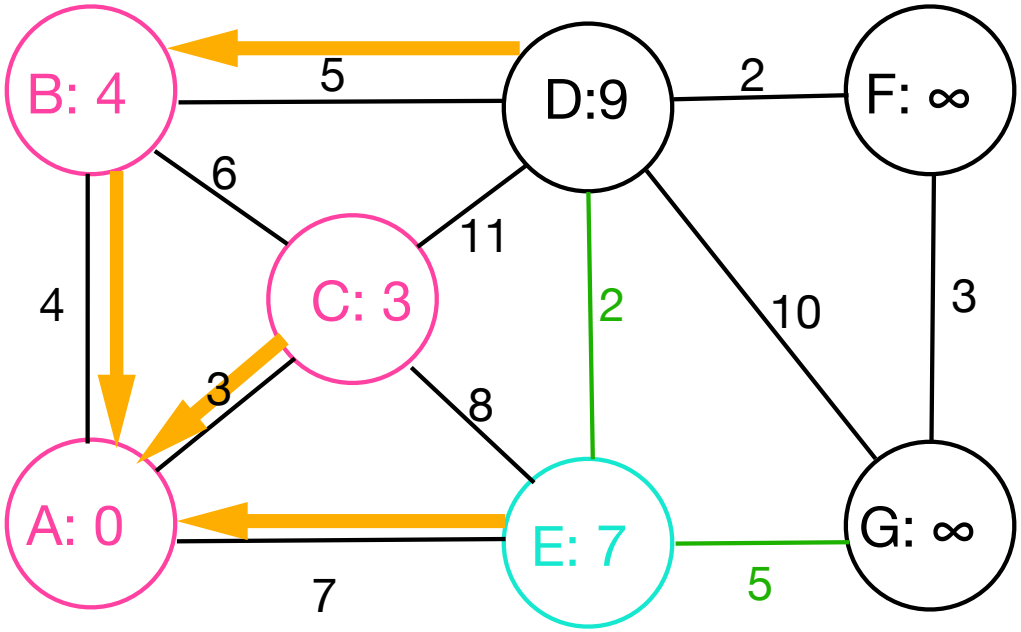


Relaxing D and G:

$$\begin{aligned} d[D] &= \min\{d(D), d(E) + w(E, D)\} \\ &= \min\{9, 7 + 2\} \\ &= 9 \end{aligned}$$

$d[D]$ did not change

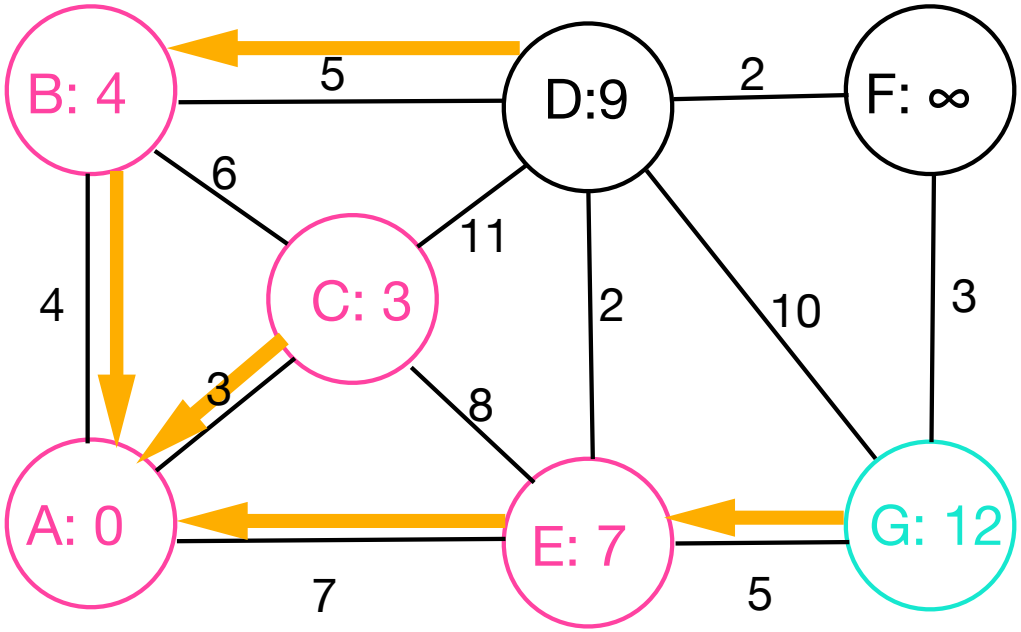
Dijkstra example $A \rightarrow F$



Relaxing D and G:

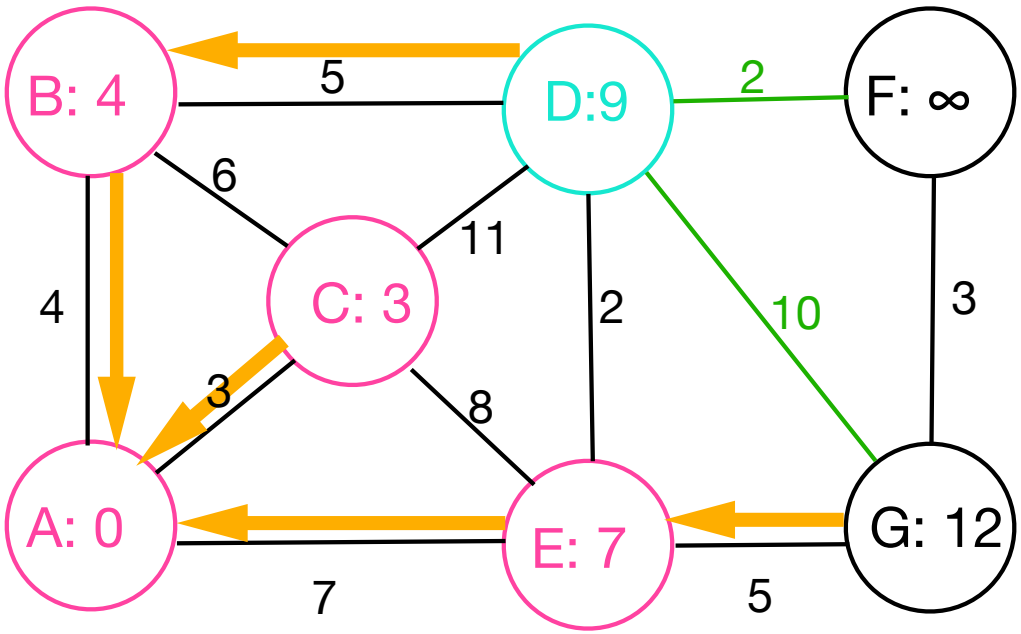
$$\begin{aligned} d[G] &= \min\{d(G), d(E) + w(E, G)\} \\ &= \min\{\infty, 7 + 5\} \\ &= 12 \qquad\qquad d[G] \text{ changed so } pred[G] \leftarrow E \end{aligned}$$

Dijkstra example $A \rightarrow F$



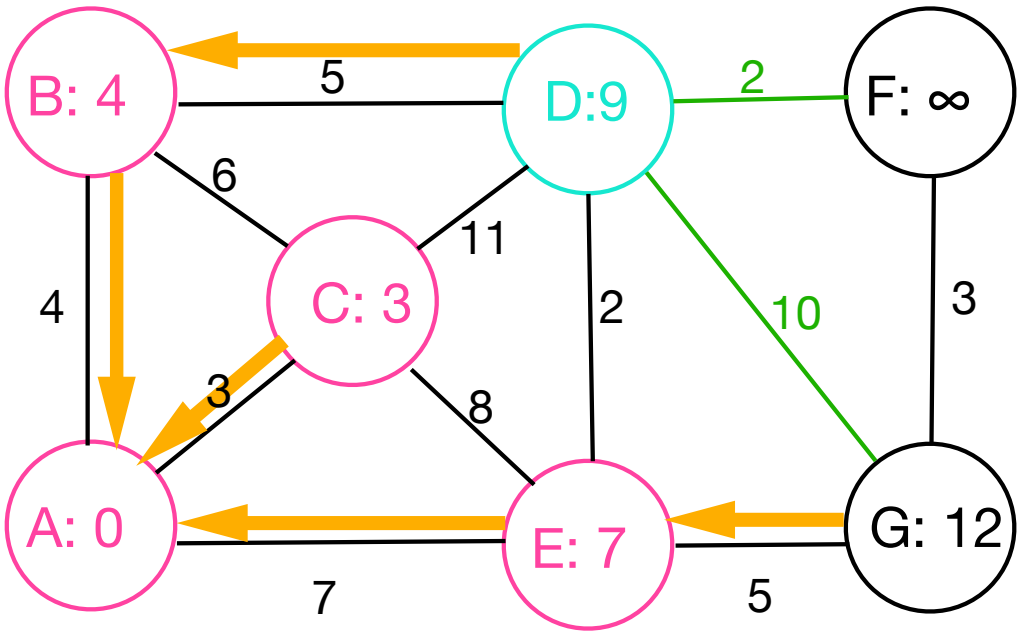
- Finished with E. The visited list is now $\{A, B, C, E\}$.
- EXTRACT-MIN $\rightarrow D$, because $d[D] = 9$ is the minimum.

Dijkstra example $A \rightarrow F$



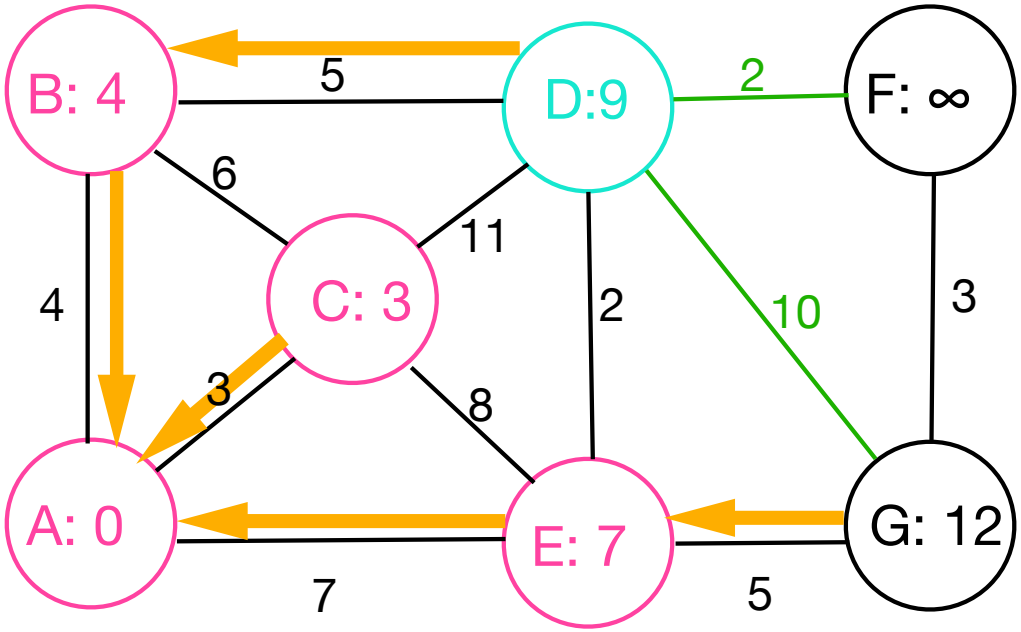
Visit the edges from D to unvisited vertices, F, and G.

Dijkstra example $A \to F$



When we relax G nothing changes.

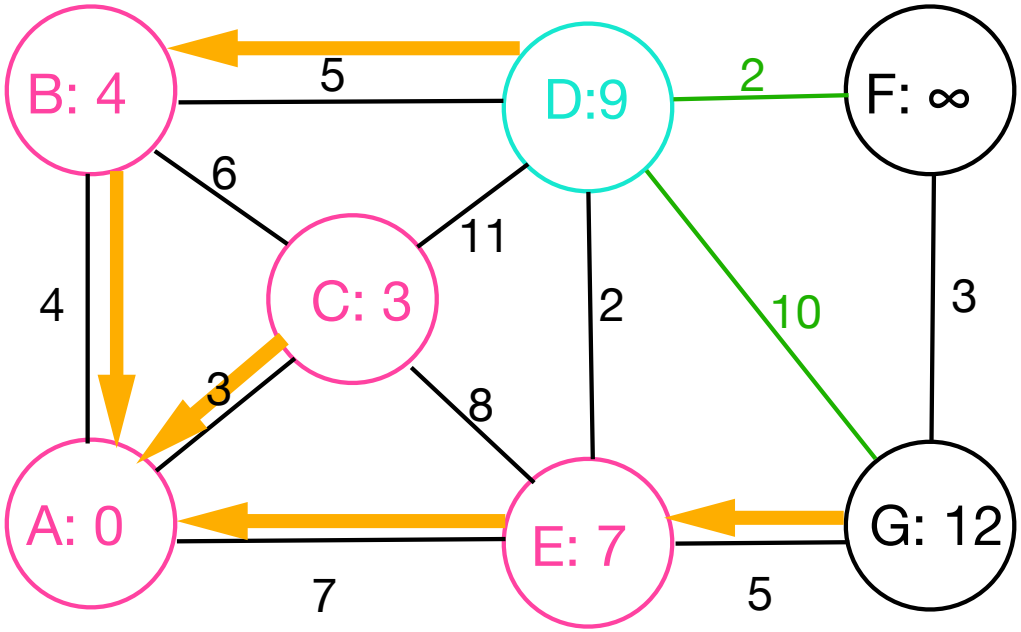
Dijkstra example $A \to F$



$$d[G] = \min\{d(G), d(D) + w(D, G)\}$$
$$= \min\{12, 9 + 10\}$$
$$= 12$$

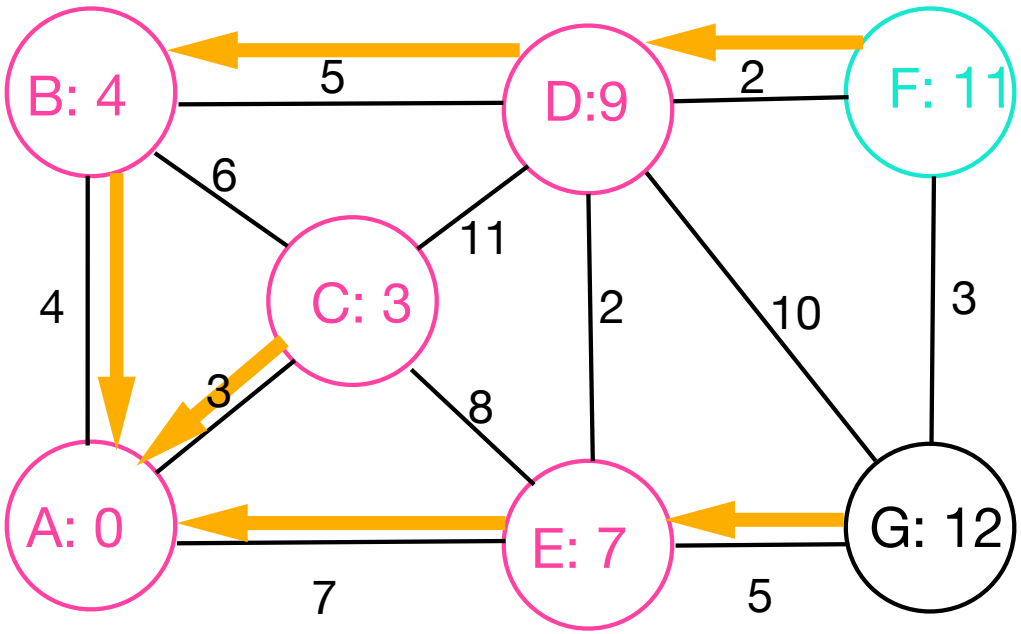
$d[G]$ did not change

Dijkstra example $A \to F$



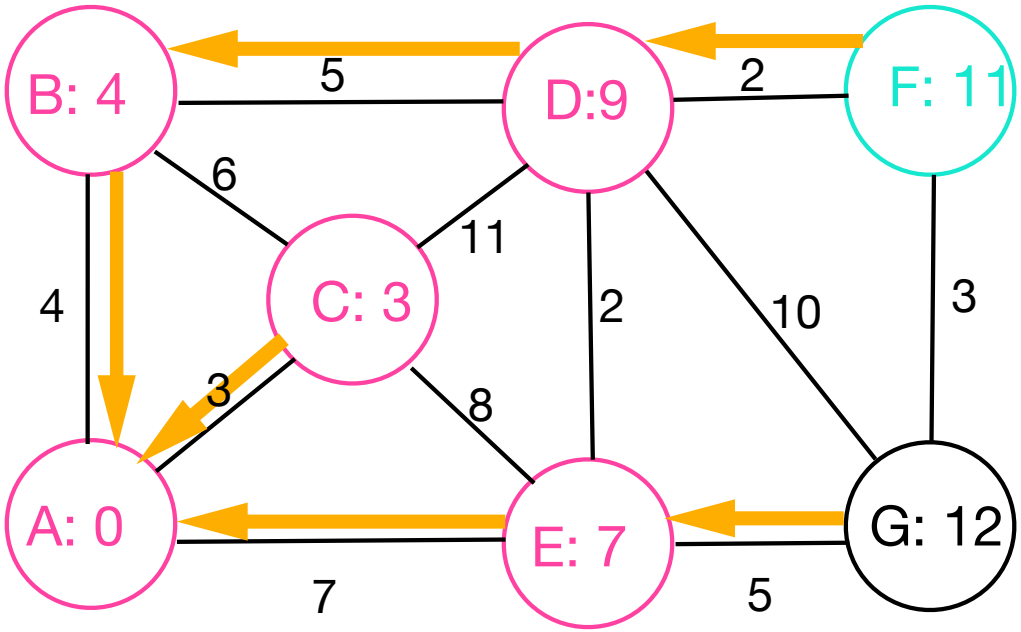
$$
\begin{aligned}
d[F] &= \min\{d(F), d(D) + w(D,F)\} \\
&= \min\{\infty, 9 + 2\} \\
&= 11 \qquad\qquad d[F] \text{ changed so } pred[F] \leftarrow D
\end{aligned}
$$

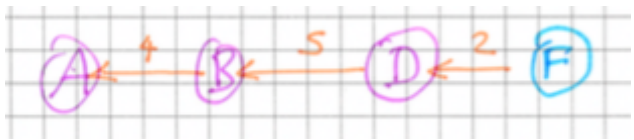
Dijkstra example $A \rightarrow F$



Update visited list to also include D.

Dijkstra example $A \to F$



- This time EXTRACT-MIN $\to F$, because $d[F] = 11$.
- F is the goal vertex. so stop
- We have found the shortest path: $\delta(S, F) = 11$.
- 

End of video

Bellman-Ford vs Dijkstra

- Bellman-Ford
  - Supports negative weights
  - Worst-case: $O(|V||E|) = O(|V|^3)$
  - Simple concentric loops, 3 to 6 lines of code
- Dijkstra
  - No negative weights
  - Worst-case: $O(|E| + |V| \log |V|)$

Worked Example, Job Dependencies Single Thread

Example, Job Dependencies

Given a set of interdependent jobs you wish to run. How can we calculate which order to run the jobs so that all dependencies are met?

- Suppose that jobs $\{J_0, J_1, J_2, J_3, J_4, J_5, J_6\}$ need to run.
- But some are dependent on others.
- We cannot start J_0 until both J_1 and J_3 are finished.
- I.e., J_1 and J_3 depend on J_0.

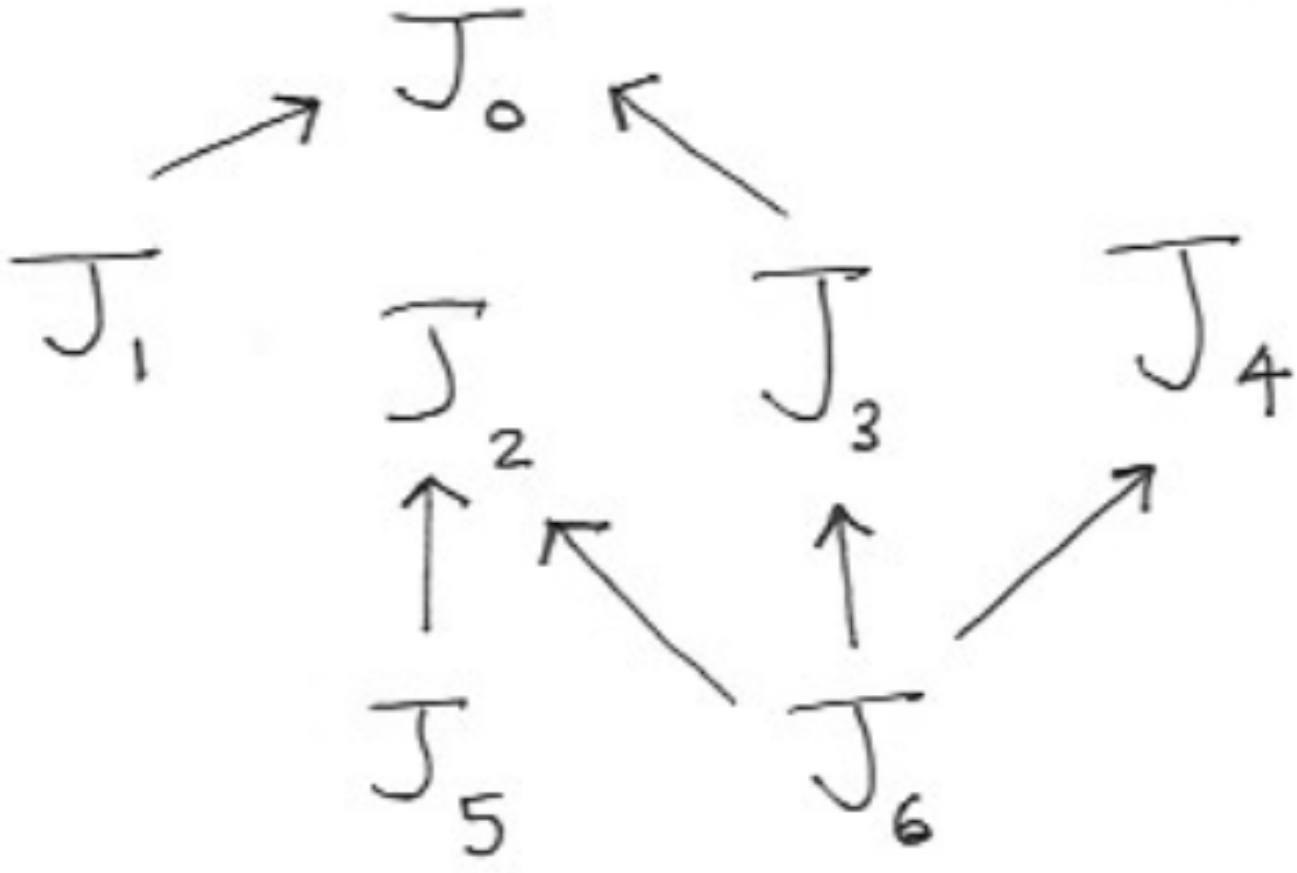
Example, Job Dependencies

Suppose $a \rightarrow \{b_1, b_2, \ldots b_n\}$ means a cannot start before $b_1 \ldots b_n$ all finish, or a depends on b. Consider a set of constraints as follows:

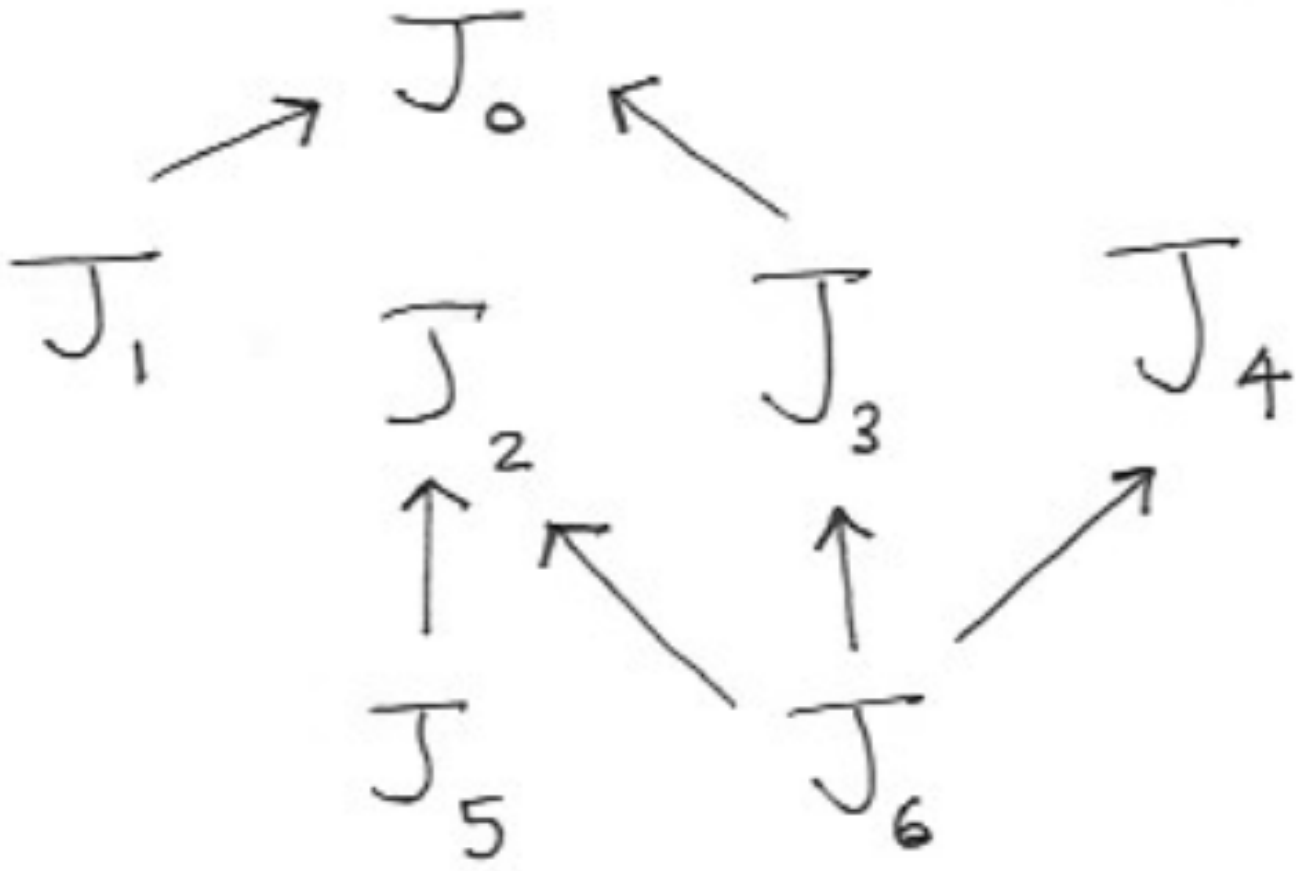
$$
\begin{aligned}
J_1 &\rightarrow \{J_0\} \\
J_3 &\rightarrow \{J_0\} \\
J_6 &\rightarrow \{J_2, J_3, J_4\} \\
J_5 &\rightarrow \{J_2\}
\end{aligned}
$$

Example, Job Dependencies

The graph looks as follows:

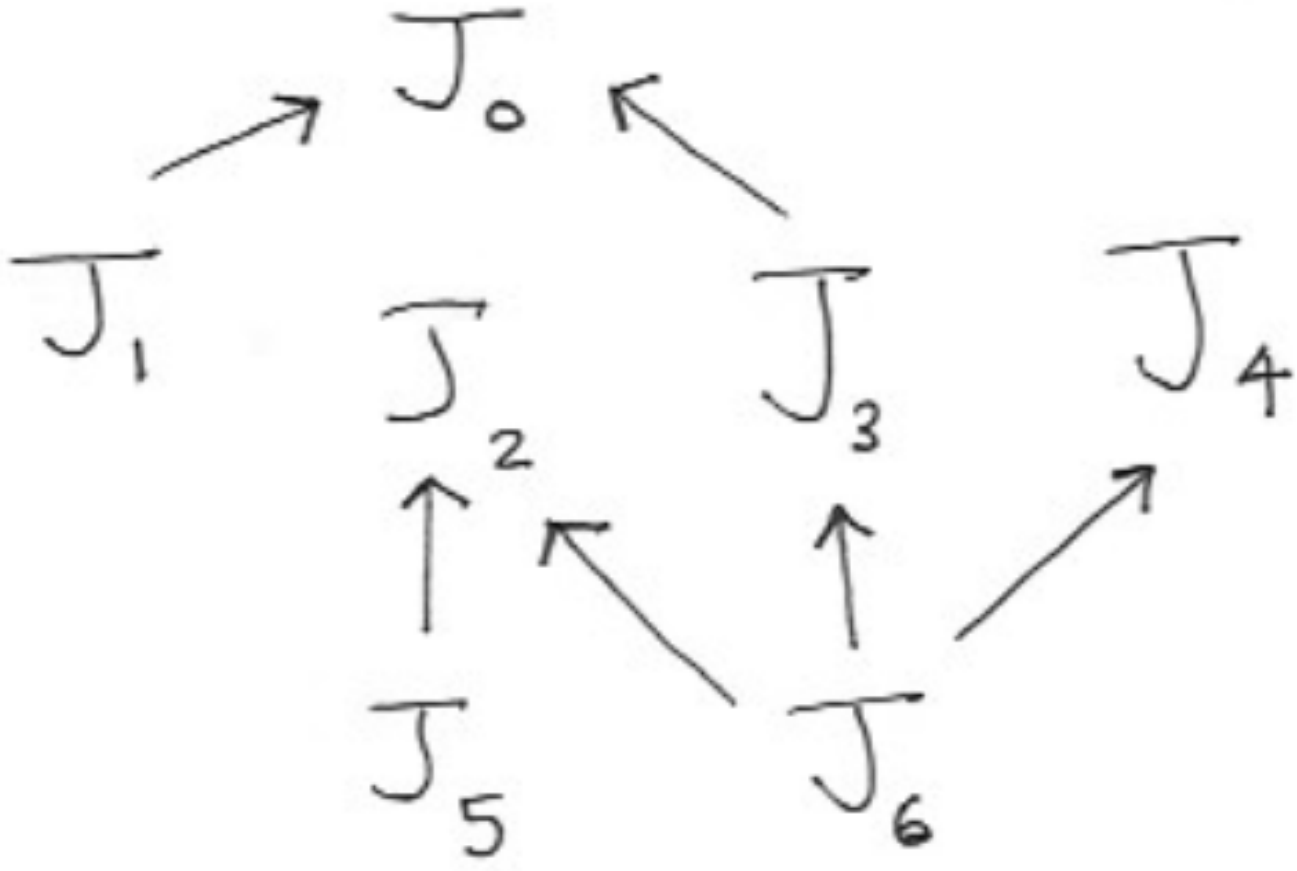


Example, Job Dependencies



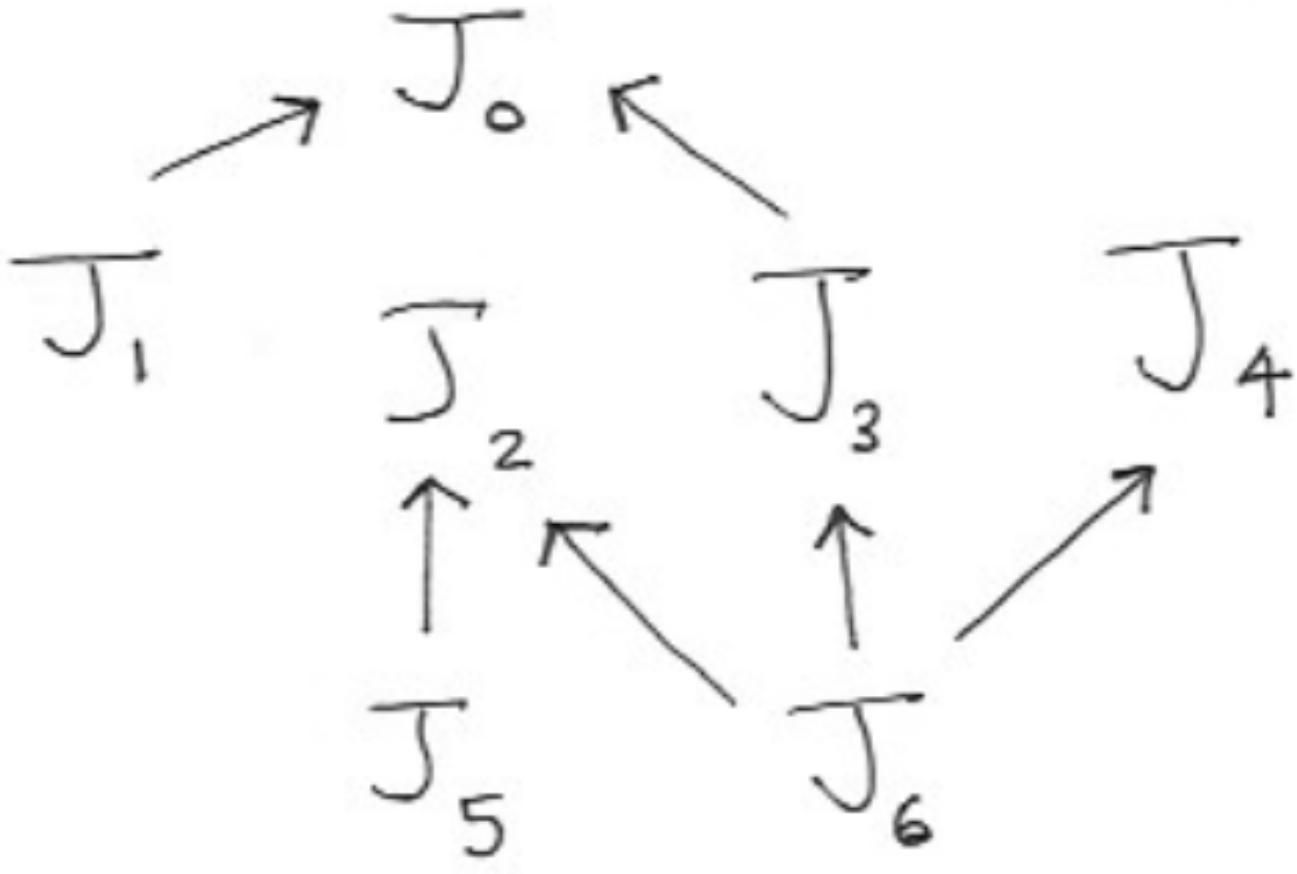
We may use DFS to label the vertices of this graph into a topological order, obeying the constraints.

Example, Job Dependencies



If the jobs must execute sequentially, i.e., we cannot run multiple jobs at once, then any valid order is just as good as any other. So any topologically sorted order will suffice.

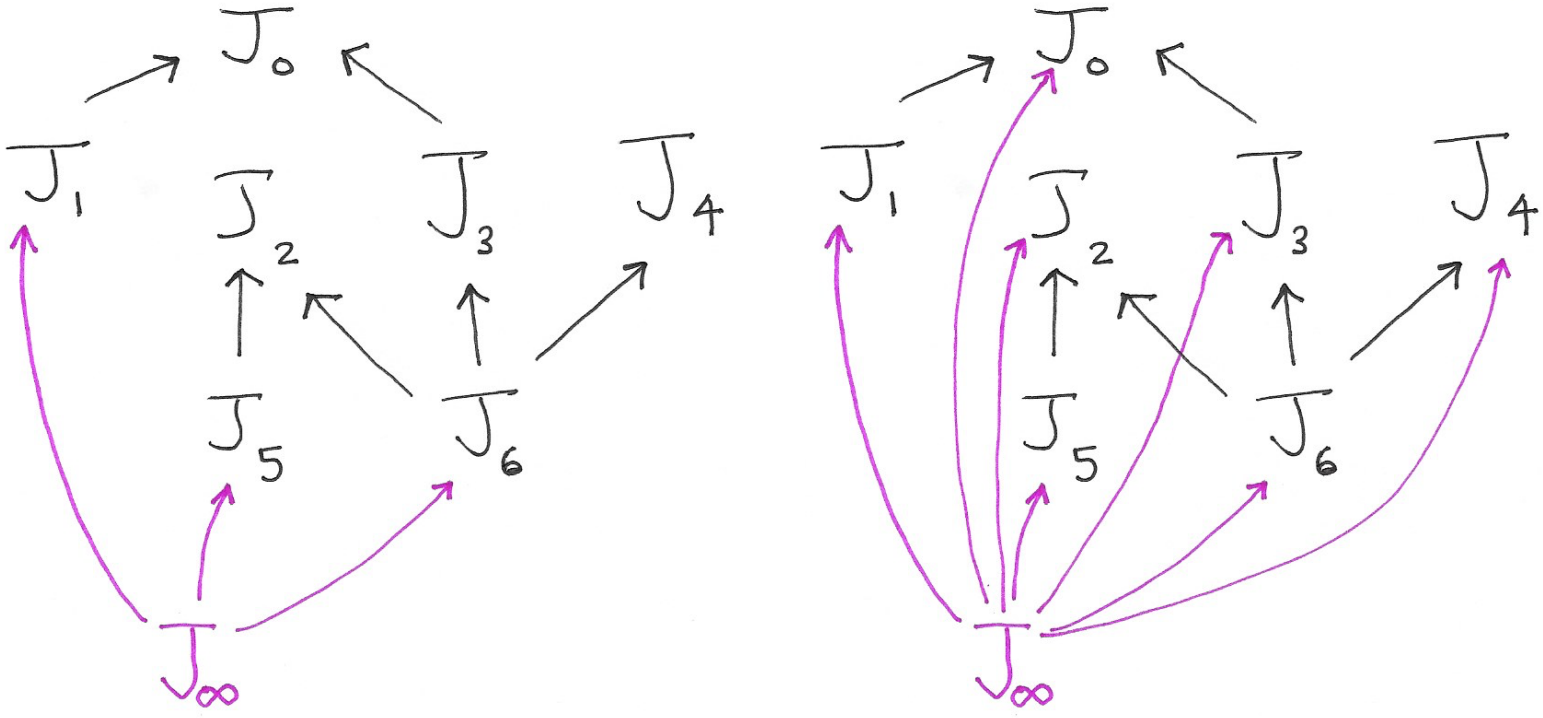
Example, Job Dependencies



Since we have multiple starting (unconstrained) vertices $\{J_1, J_5, J_6\}$ we don't know where to start the DFS.

Example, Job Dependencies

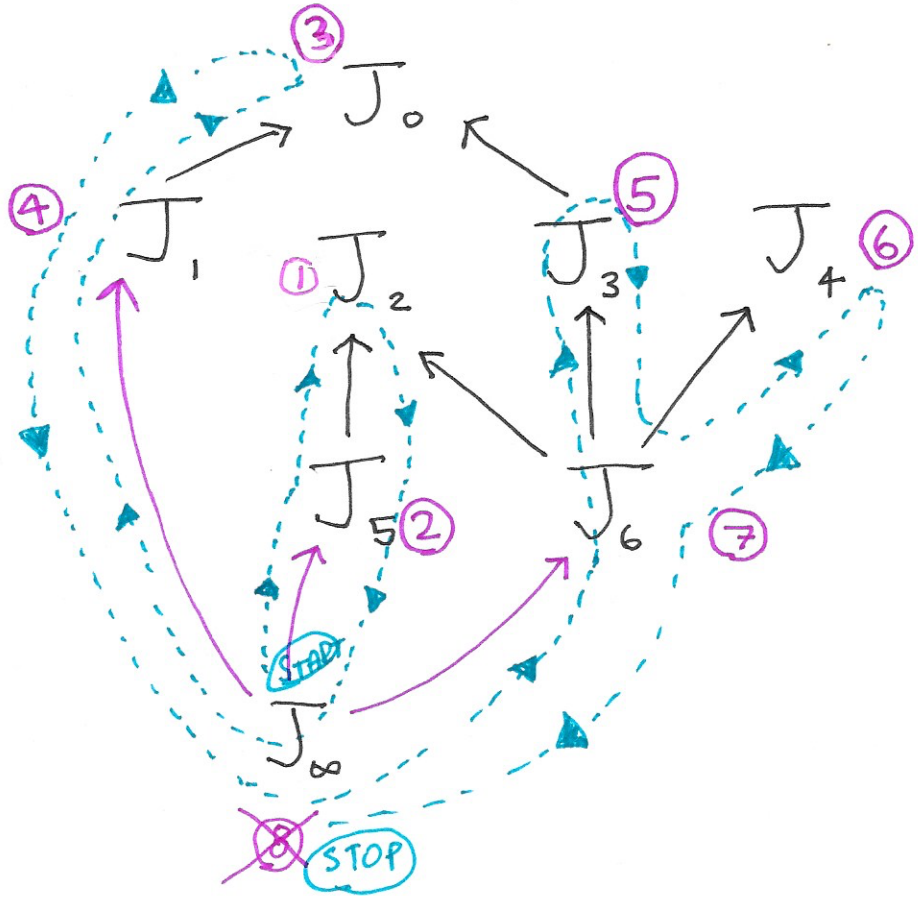
We create an *artificial* vertex, J_∞ connected either to the vertices in $\{J_1, J_5, J_6\}$, or to all the vertices in the graph.



We can then perform the simple DFS starting at vertex J_∞. Labeling each node after all its neighbors have been visited.

Example, Job Dependencies

$$\{J_2, J_5, J_0, J_1, J_3, J_4, J_6\}$$



This may result in the following order, depending on the order neighbors are visited.

Example 2, Job Dependencies Multiple Threads

Job Scheduling with Multiple Threads

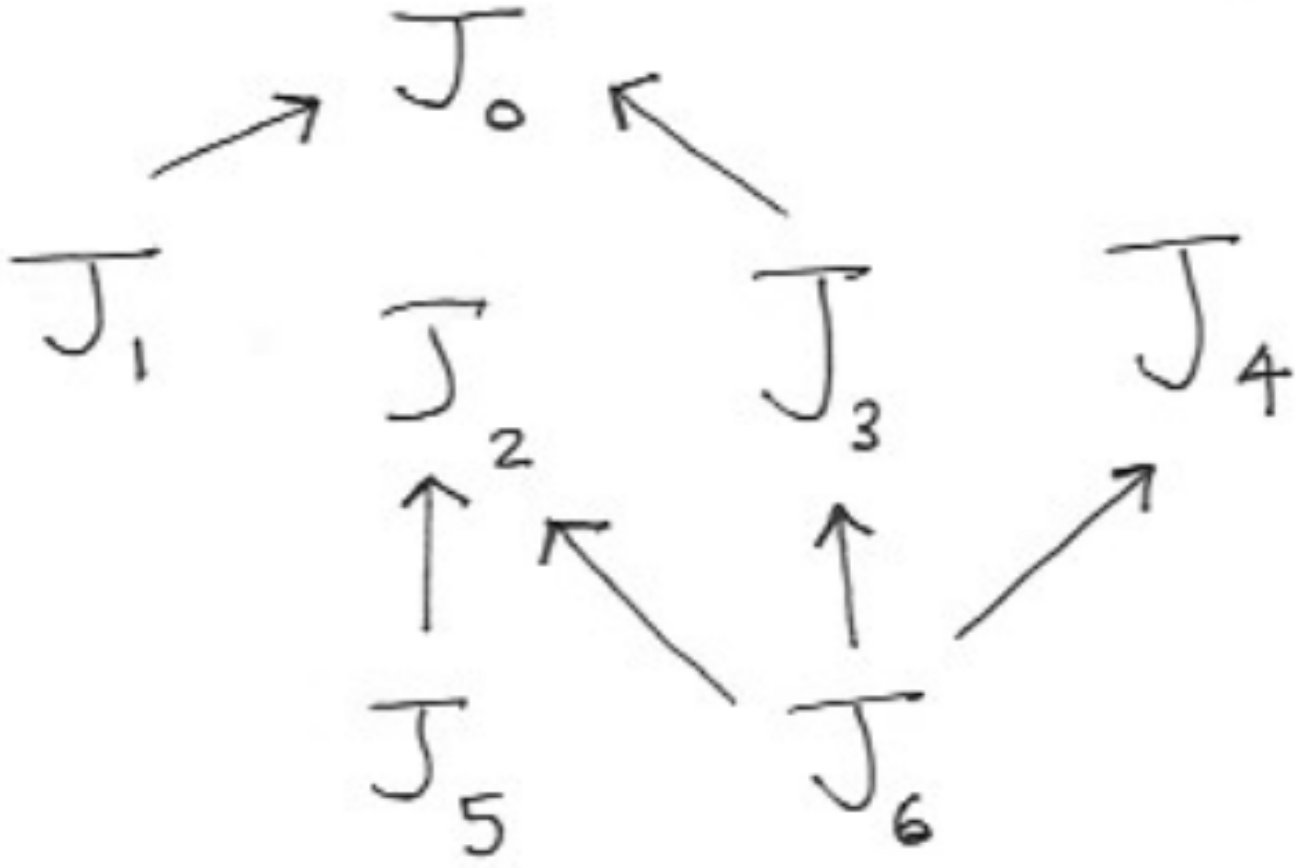
Now we revisit the scheduling problem assuming we have two threads to use

- We may run two jobs simultaneously,
- as long as we never violate the dependencies.
- Assume that we have time estimates for the jobs.

Job Scheduling with Multiple Threads

Job No.	**Estimated Time (sec)**
0	$\Delta t = 3$
1	$\Delta t = 4$
2	$\Delta t = 6$
3	$\Delta t = 2$
4	$\Delta t = 5$
5	$\Delta t = 3$
6	$\Delta t = 10$

Job Scheduling with Multiple Threads

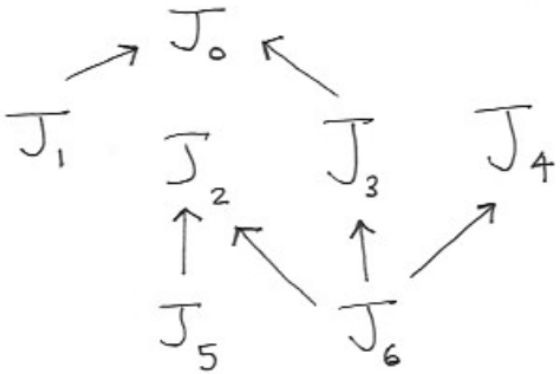


For this exercise we swap the semantics of the arrows.

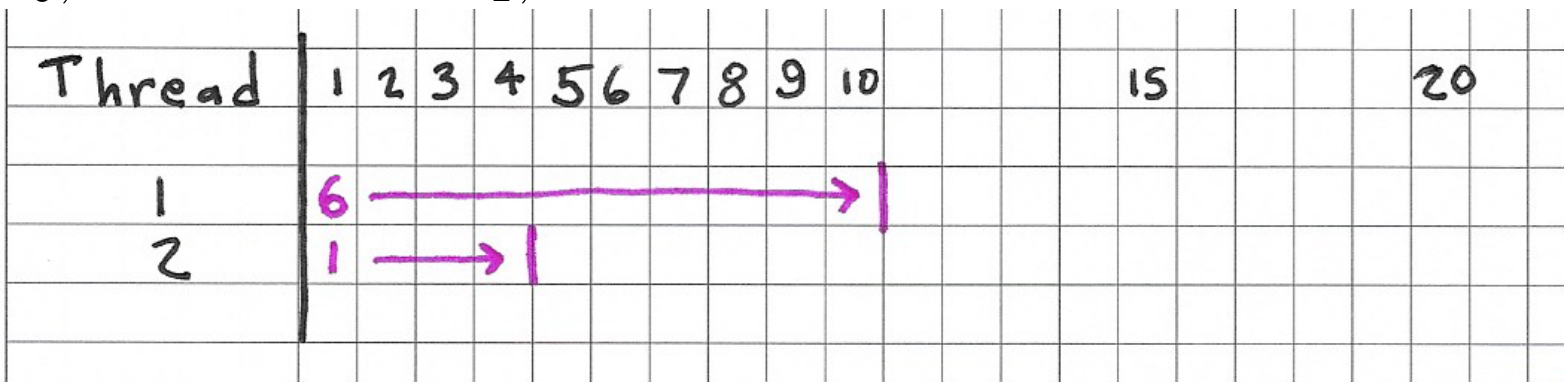
Now $J_a \to J_b$ means: J_a must finish before J_b can start.

Job Scheduling with Multiple Threads

Whenever we have the choice of 2 or more jobs to start, we will start the one projected to run slowest.

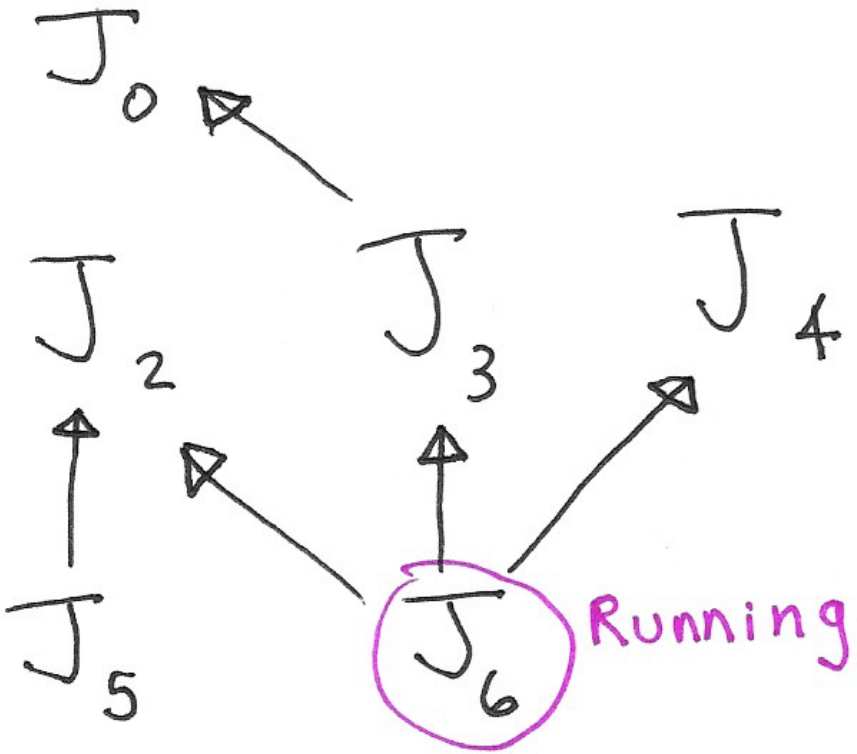


We choose among J_1, J_5, and J_6. The slowest two are $J_6, \Delta t = 10$ and $J_1, \Delta t = 4$.



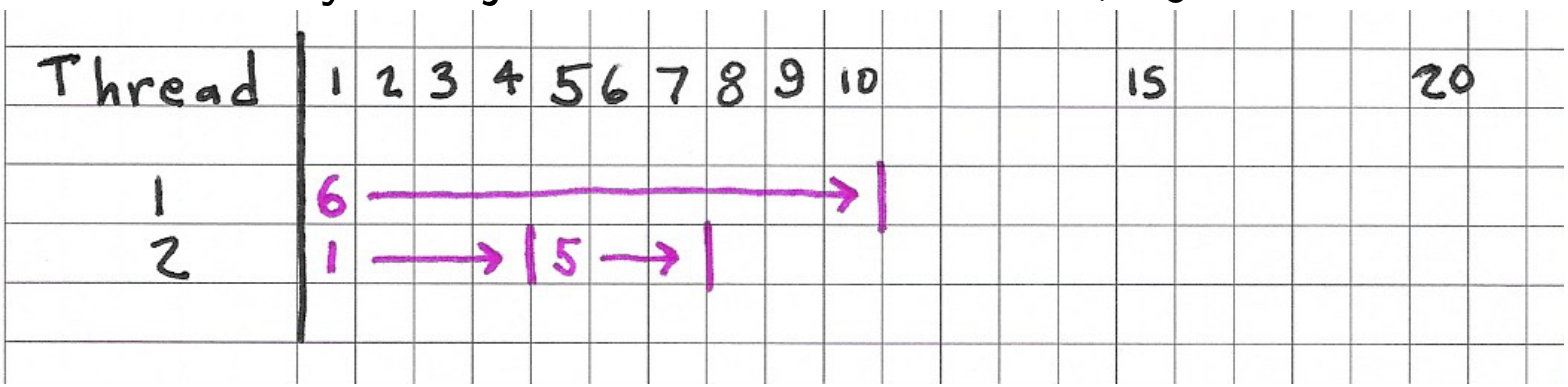
Job Scheduling with Multiple Threads

J_1 finishes at $t = 4$, leaving J_6 running and the graph.



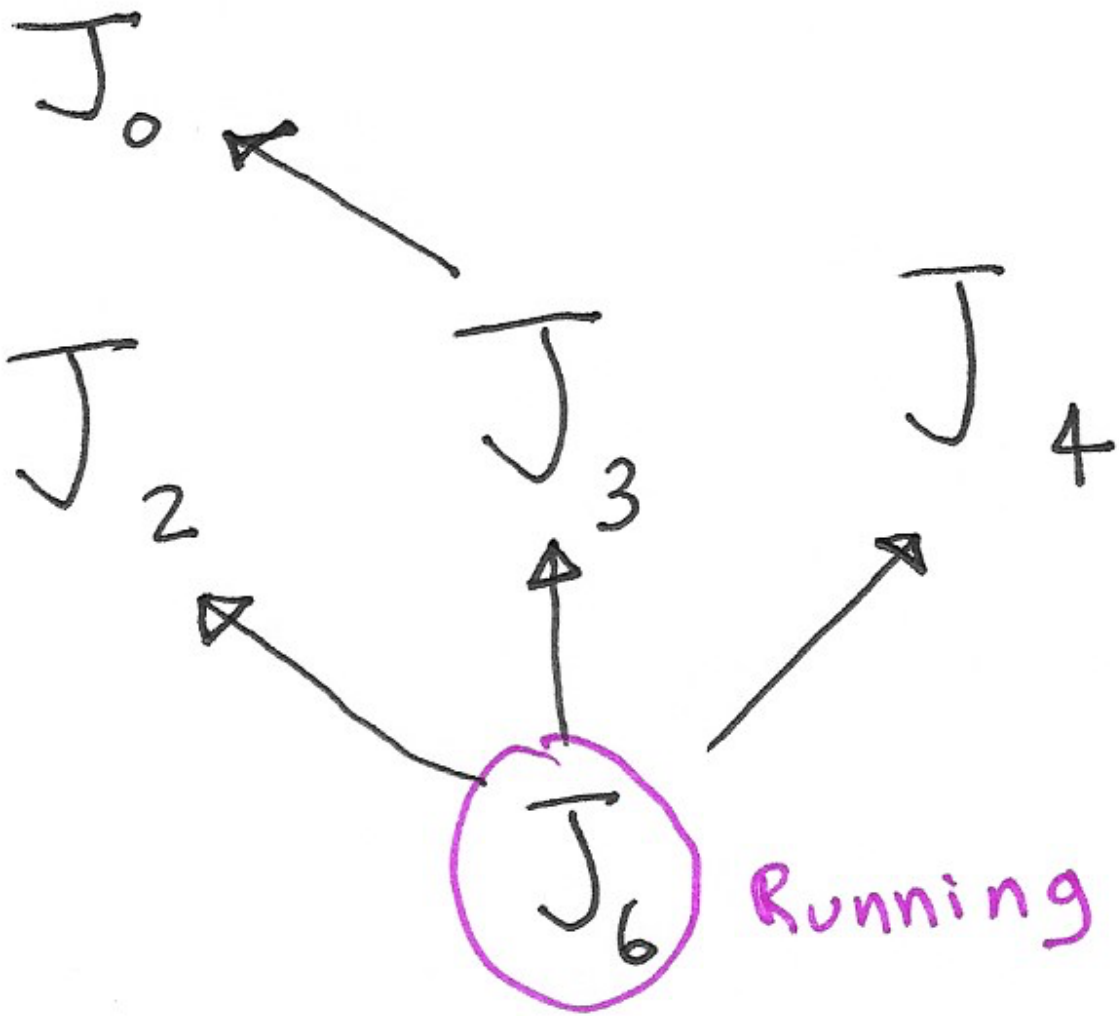
Job Scheduling with Multiple Threads

There is only one job which we can start, J_5.



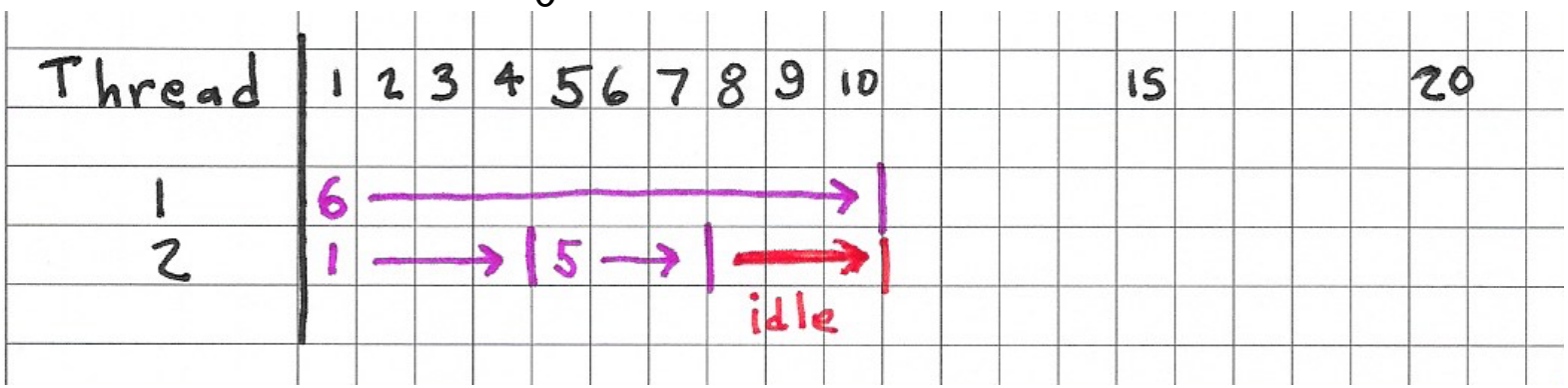
Job Scheduling with Multiple Threads

At after $\Delta t = 3$, at time $t = 7$, job J_5 finishes, but J_6 is still running.



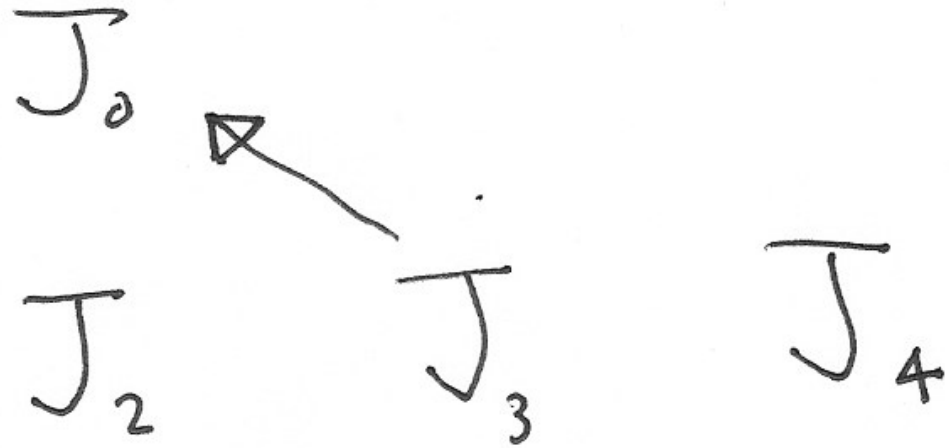
Job Scheduling with Multiple Threads

At this point we cannot start any new job. We must wait with thread 2 idle until J_6 finishes at time $t = 10$.



Job Scheduling with Multiple Threads

After job J_6 finishes at time $t = 10$ the remaining dependency graph looks like the following.



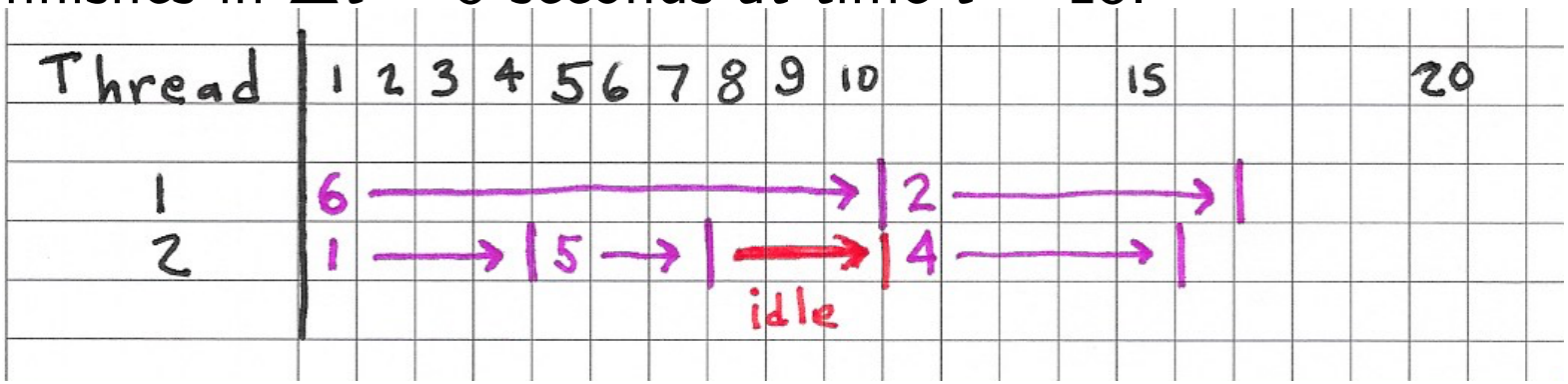
At this point there are two threads free, so we can start two jobs. There are three candidates:

$J_2 \quad \Delta t = 6$
$J_3 \quad \Delta t = 2$
$J_4 \quad \Delta t = 5$

Job Scheduling with Multiple Threads

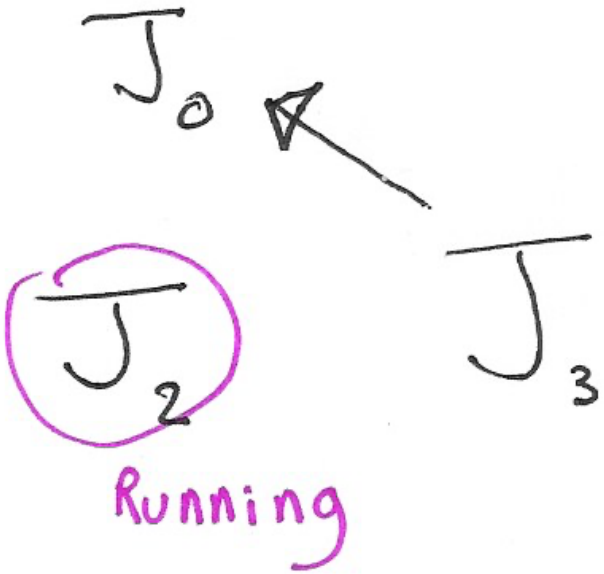
$J_2 \quad \Delta t = 6$
$J_3 \quad \Delta t = 2$
$J_4 \quad \Delta t = 5$

Choosing the slowest two of the three: i.e., J_2 and J_4. Job J_4 finishes in $\Delta t = 5$ seconds at time $t = 15$.



Job Scheduling with Multiple Threads

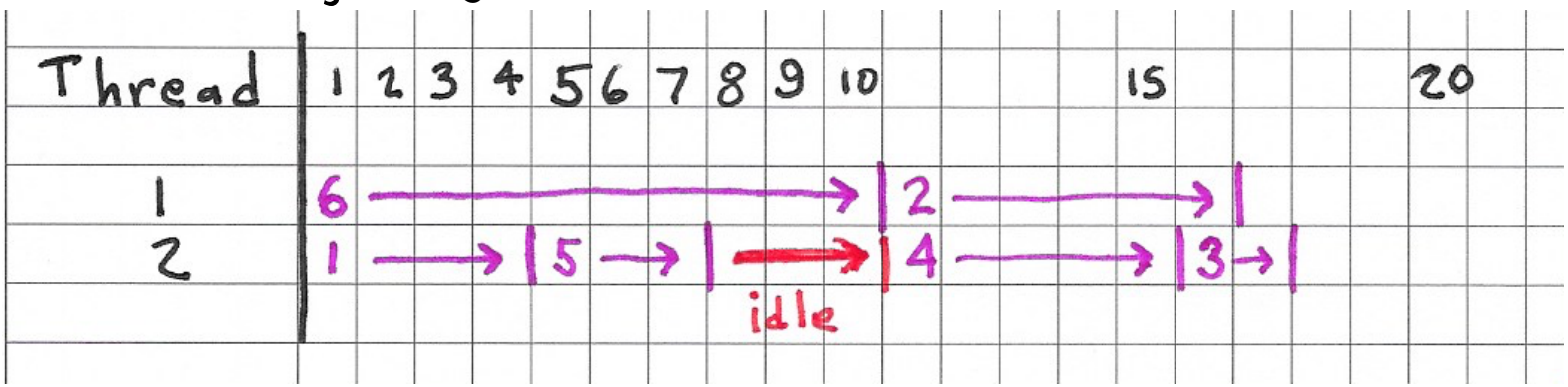
This leaves J_2 running, but J_3 can be launched.



.

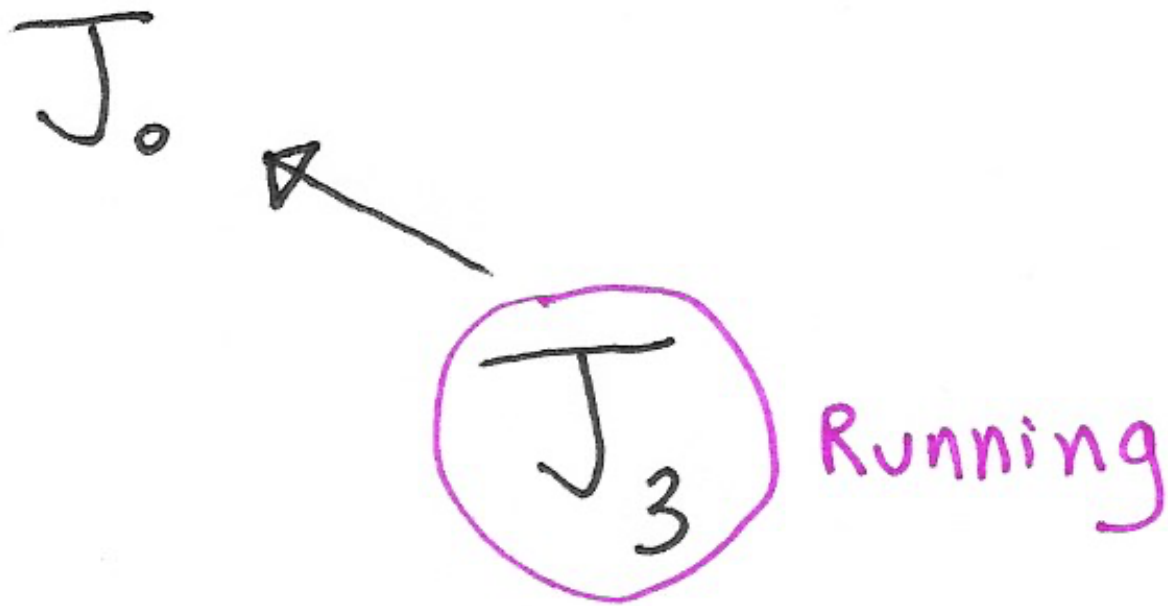
Job Scheduling with Multiple Threads

So we start job J_3 at time $t = 15$.



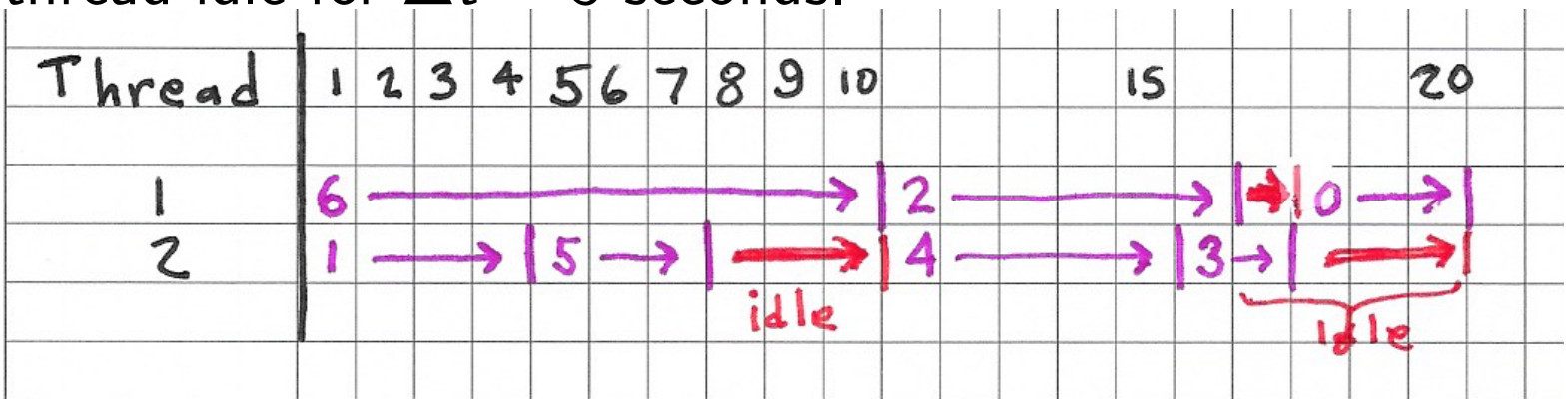
Job Scheduling with Multiple Threads

At time $t = 16$, job J_2 finishes, having run $\Delta t = 6$ seconds. This leaves the following dependency graph.



Job Scheduling with Multiple Threads

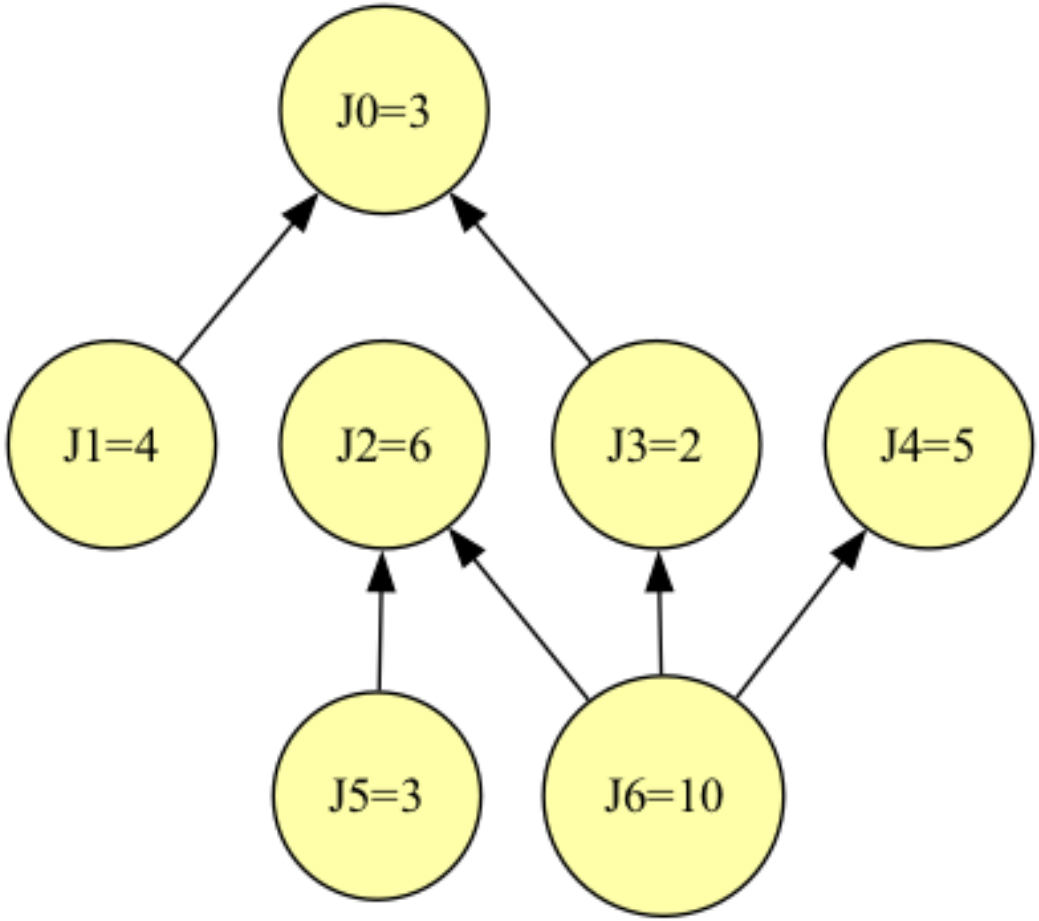
Job J_3 is still running; we cannot launch another job until J_3 finishes, leaving thread 1 idle for one second until time $t = 17$. At time $t = 17$ we may launch the final job J_0, leaving one thread idle for $\Delta t = 3$ seconds.



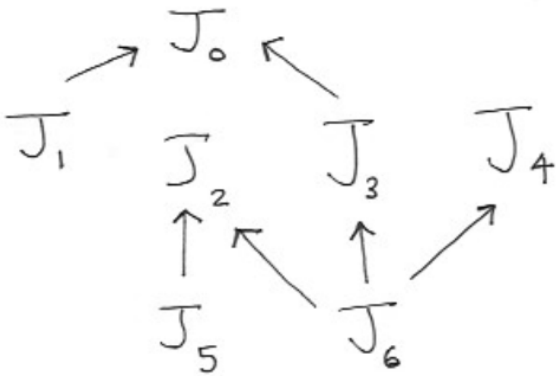
The total wall time of execution is $\Delta t = 20$ seconds, including 7 seconds of idle time.

Exercise for the Student

Rework the job schedule example using fast before slow as a tie-breaker. What is the wall time for everything to finish? Is it better or worse? How much time is wasted as idle time?



Exercises for the Student



What if we consider not the time of the individual job, but the total time of that job and its dependent jobs? For J_6 the time estimate is

$$\max\{J_6 + J_2, J_6 + J_3 + J_0, J_6 + J_4\} = \max\{16, 15, 15\} = 16\,,$$

and then use the tie-breaker where we start the job with the maximum estimated time. What changes?

Exercise for the Student

Rework the example using three threads, with any tie breaker you wish or invent a new one.