Announcements

- **Weekly Reading Assignments: Chapters 23 & 24 (CLRS)**

Definitions for Shortest Paths

- Think of vertices as cities and the edge weights as the distance from one city to another. Define the *length* of a path to be the sum of edge weights along the path. Define the *distance* between two vertices, u and v, $\delta(u,v)$ to be the length of the minimum length path from u to v.

- 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 Paths

- Given a directed graph $G = (V, E)$ with edge weights and a distinguished ***source vertex***, $s \in V$, determine the distance from the source vertex to every vertex in the graph.
- BFS finds short-paths from a single source vertex to all other vertices in $O(n+e)$ time, assuming the graph has no edge weights.
- Edge weights can be negative; but in order for the problem to be well-defined there must be no cycle whose total cost is negative.

Variants

- ***Single-destination shortest-paths problem*: Find a shortest path to a given destination vertex t from every vertex $v \in V$.**
- ***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 shortest paths problem, we also solve this problem.**
- ***All-pairs shortest-paths problem*: Find a shortest path from u to v for every pair of vertices u and v. This can be solved by the single-source problem run for each vertex.**

Relaxation

- Maintain an estimate of the shortest path for each vertex v, call it $d[v]$.
- Initially $d[v]$ will be the length of the shortest path that the algorithm of knows from s to v. This value will always be greater than or equal to the true shortest path distance from s to v.
- Initially, we know of no paths, so all $d[v]=\infty$ & $d[s]=0$.
- As the algorithm goes on and sees more vertices, it tries to update $d[v]$ for each vertex in the graph, until all $d[v]$ values converge to true shortest distances.

Find Shortest Path by Relaxation

- **If the solution is not yet an optimal value, then push a little closer to the optimum. If we find a path from s to v shorter than $d[v]$, then update $d[v]$.**
- **Consider an edge from a vertex u to v whose weight is $w(u,v)$. Suppose that we have already computed current estimates on $d[u]$ and $d[v]$. We know that there is a path from s to u of weight $d[u]$. By taking this path and following it with the edge (u,v) we get a path to v of length $d[u]+w(u,v)$. If this path is better than the existing path of length $d[v]$ to v, we should take it.**

Relax (u, v)

1. if $d[u] + w(u,v) < d[v]$ // is the path thru u shorter?
2. then $d[v] \leftarrow d[u] + w(u,v)$; // yes, then take it.
3. $\pi[v] \leftarrow u$;

// the shortest way back to the source is thru

// u by updating the predecessor pointer

NOTE: If we perform Relax (u, v) repeatedly over all edges of the graph, all the $d[v]$ values will eventually converge to the true final distance values from s. How to do this most efficiently?

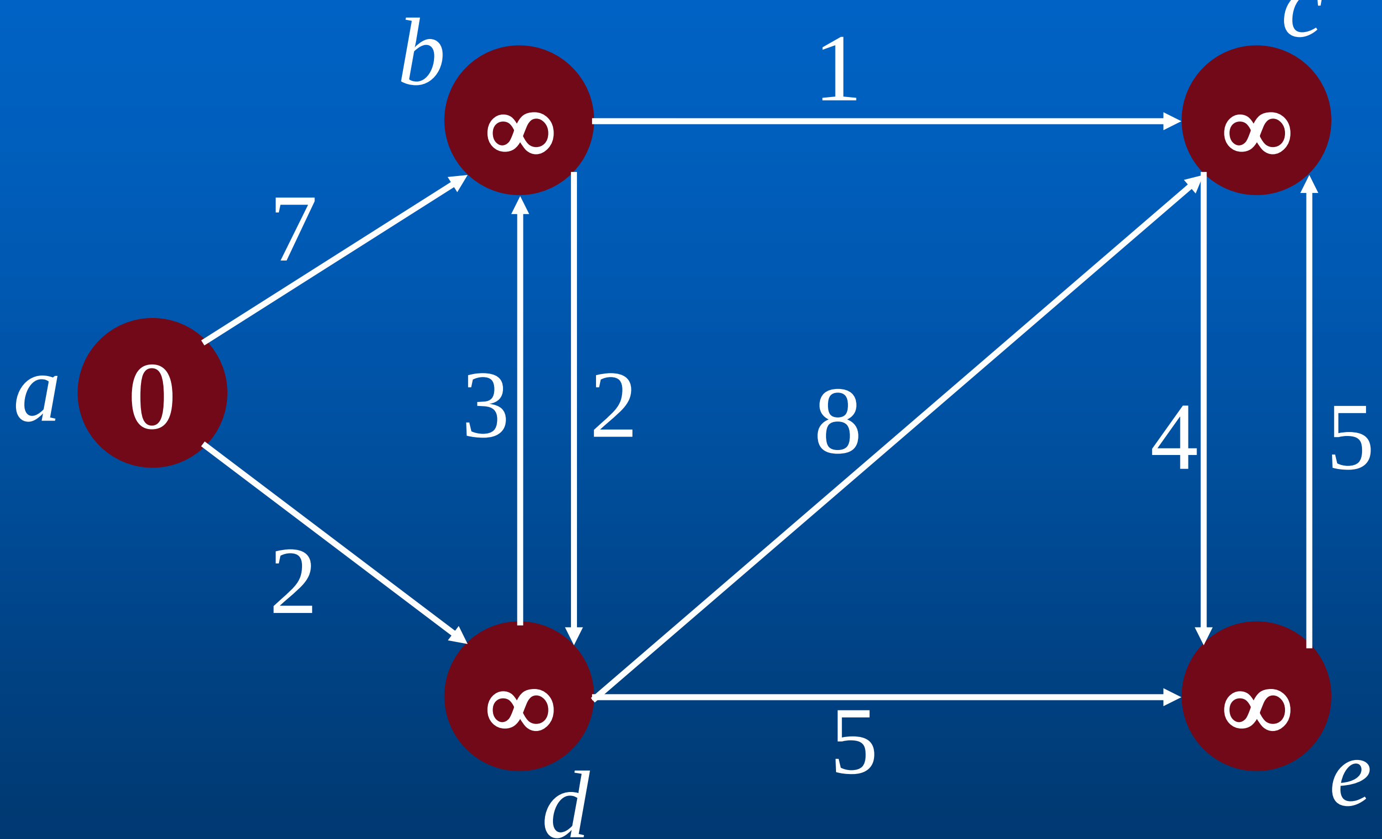
Dijkstra's Algorithm

- Maintains a subset of vertices, $S \subseteq V$, for which we know their true distance $d[u]=\delta(s,u)$. Initially $S = \emptyset$ and we set $d[s]=0$ and all others to ∞. One by one we select vertices from $V - S$ to add to S.
- For each vertex $u \in V\text{-}S$, we have computed a distance estimate $d[u]$. The greedy approach is to take the vertex for which $d[u]$ is minimum, i.e. take unprocessed vertex that is closest to s.
- We store the vertices of $V - S$ in a priority queue (heap), where the key value of each vertex u is $d[u]$. All operations can be done in $O(\lg n)$ time.

Dijkstra(G, w, s)

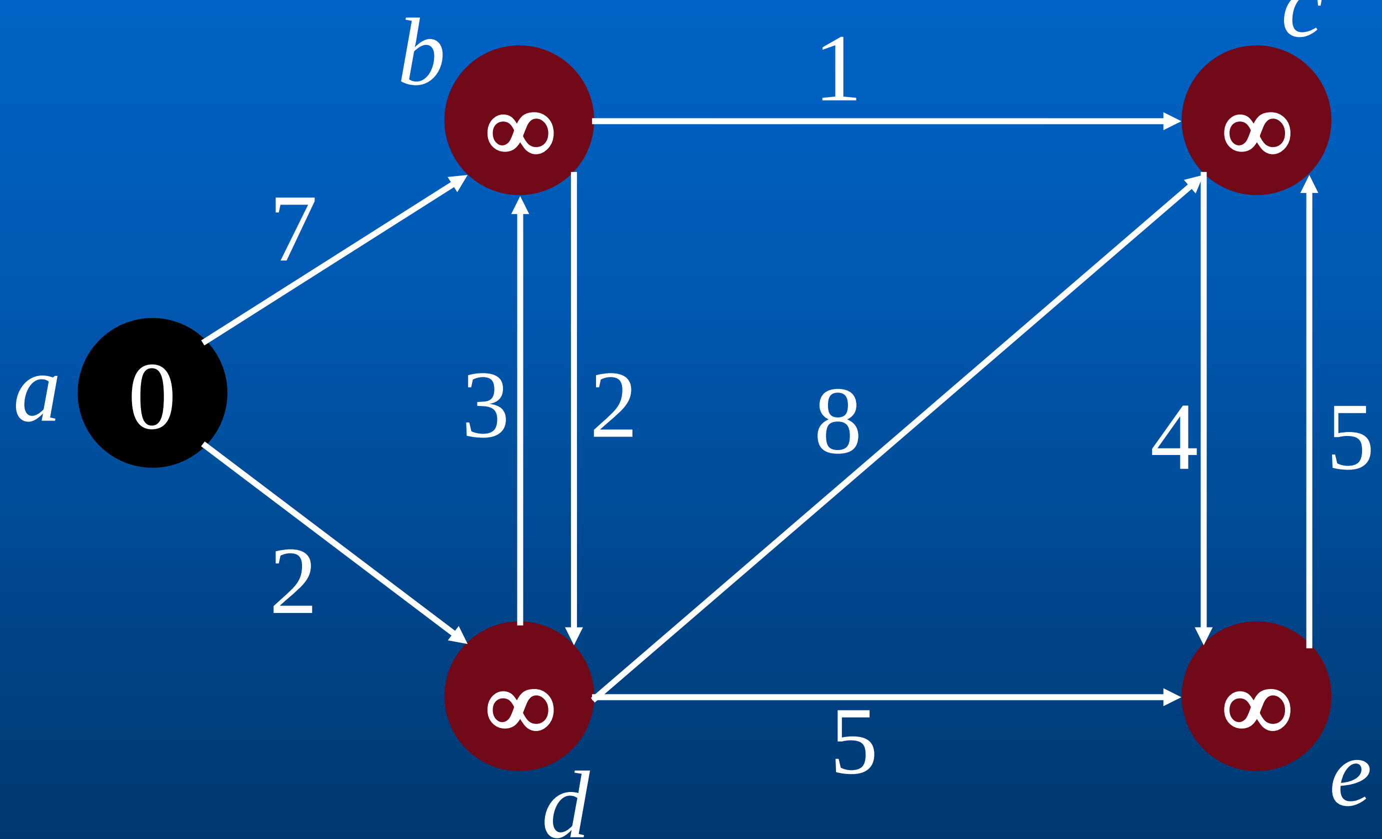
```
1.  Q ← V[G] and S ← ∅
2.  for each vertex u ∈ Q                          // initialization: O(V) time
3.        do d[u] ← ∞ and π[u] ← NIL
4.  d[s] ← 0                                       // start at the source
5.  π[s] ← NIL                                     // set parent of s to be NIL
6.  while Q ≠ ∅                                    // till all vertices processed
7.        do u ← Extract-Min(Q)                    // select closest to s
             S ← S ∪ {u}
8.           for each v ∈ adj[u]
9.                 do if v ∈ Q and (d[u] + w(u,v) < d[v])
10.                       then π[v] ← u
11.                            d[v] ← d[u] + w(u,v)   //  Relax (u,v)
12.                            decrease_Key(Q, v, d[v])
```

Example: Dijkstra's Algorithm



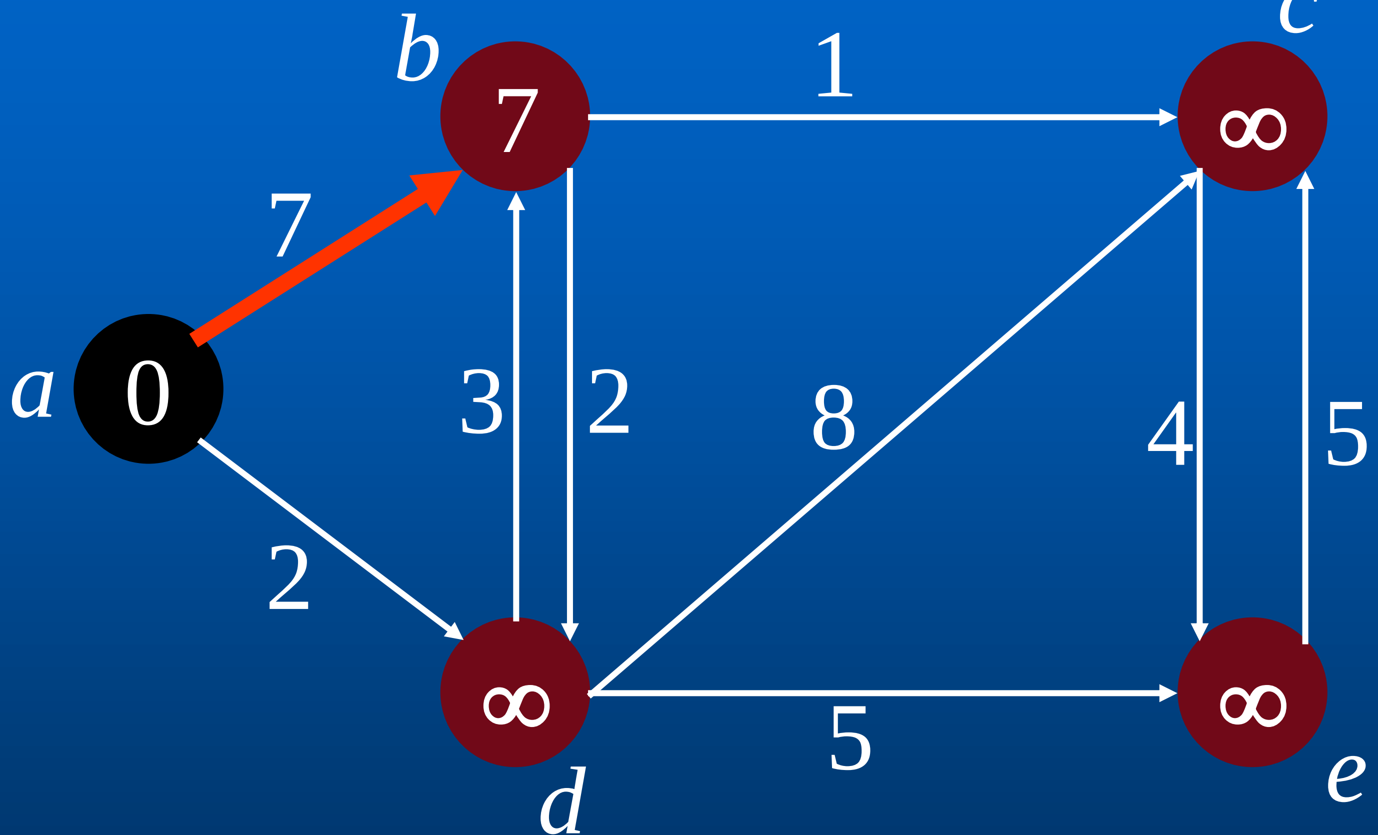
Black: in S

Example: Dijkstra's Algorithm



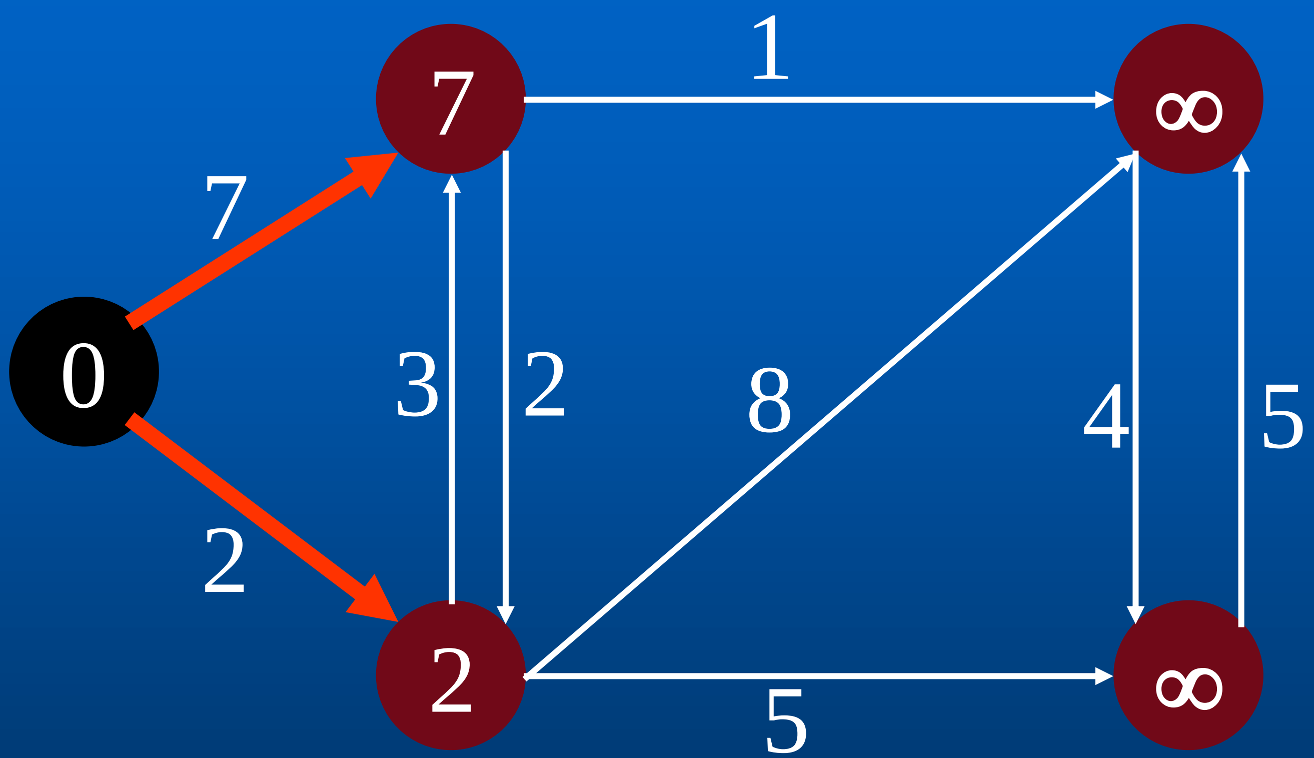
Black: in S

Example: Dijkstra's Algorithm



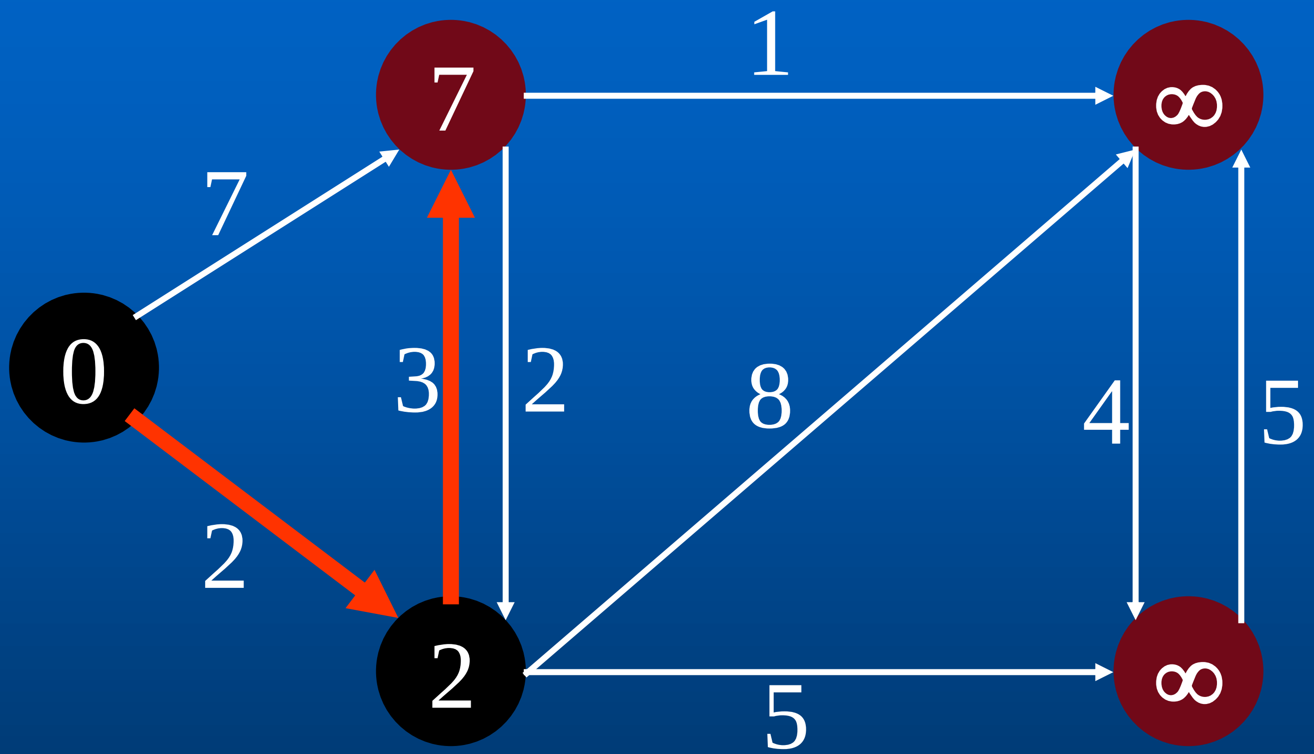
Black: in *S*

Example: Dijkstra's Algorithm



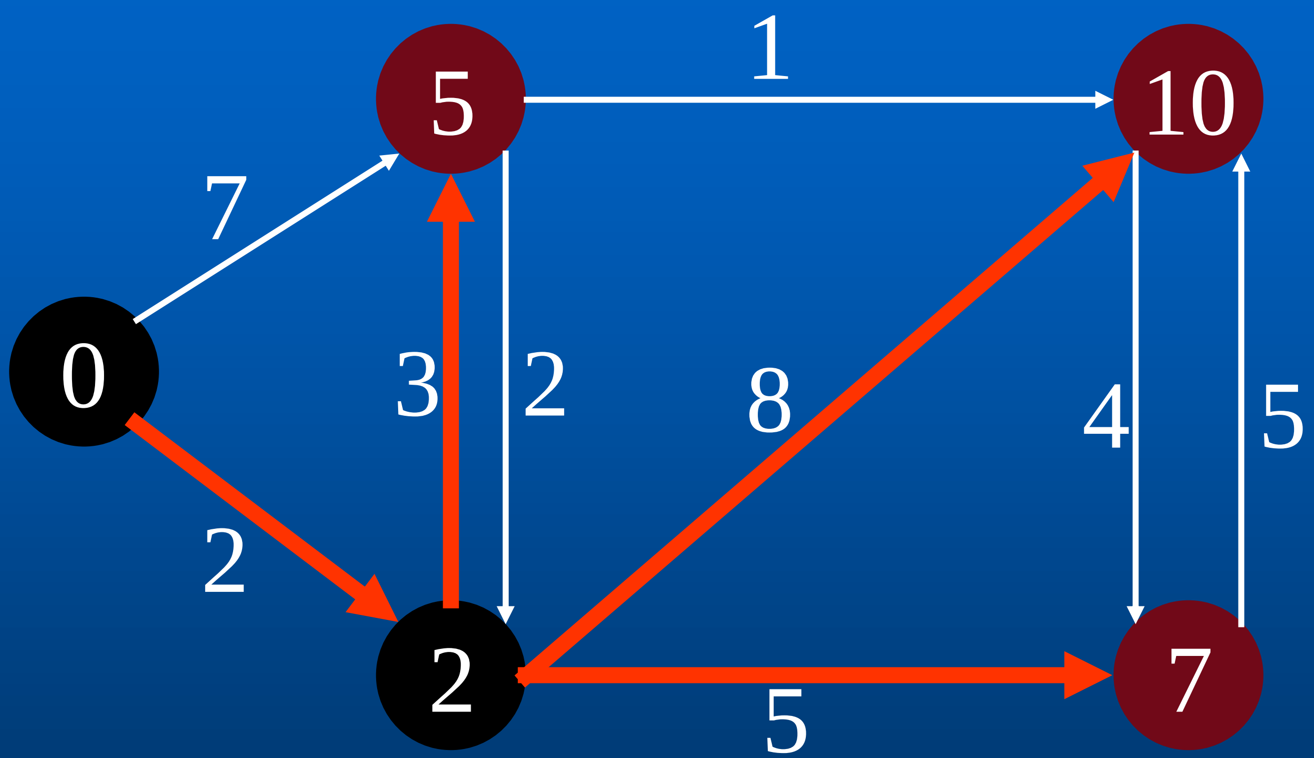
Black: in *S*

Example: Dijkstra's Algorithm



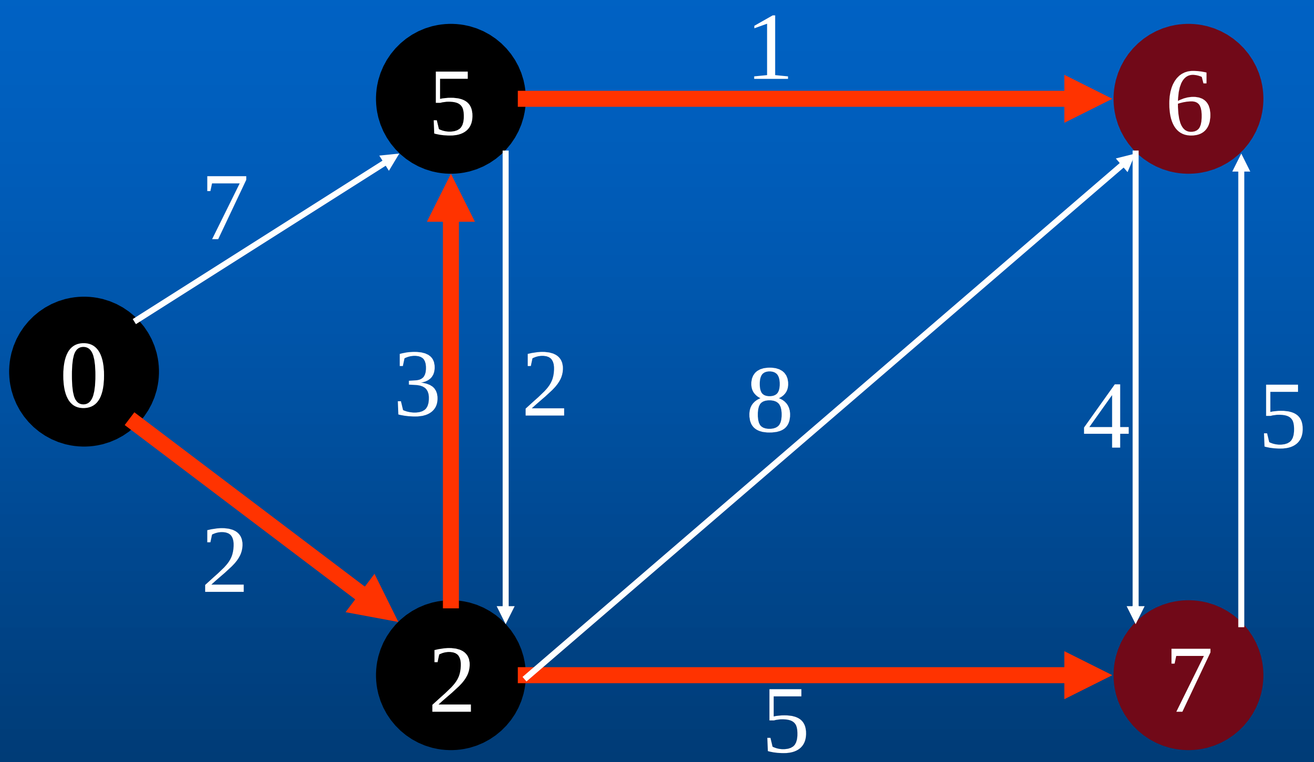
Black: in *S*

Example: Dijkstra’s Algorithm



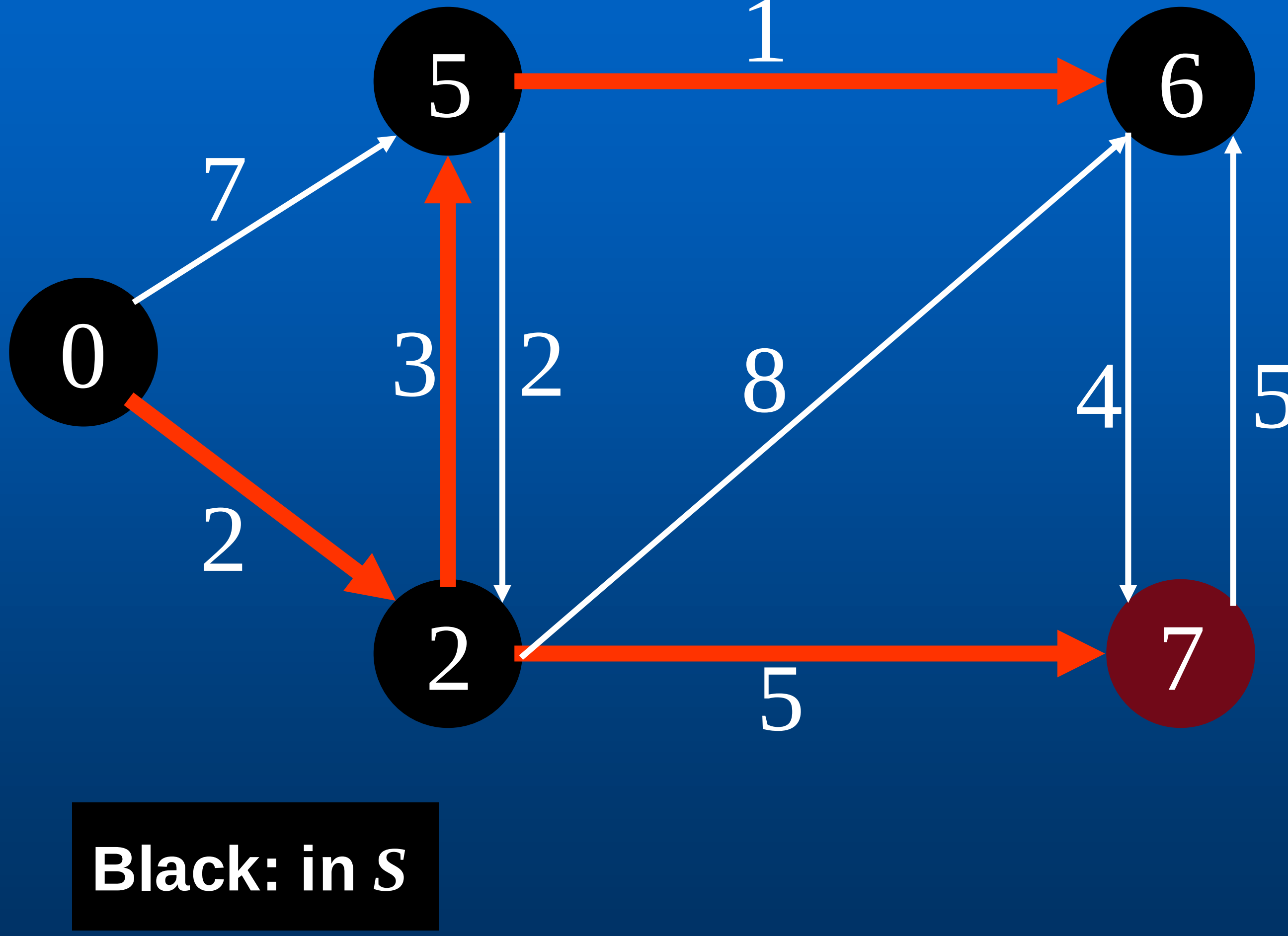
Black: in *S*

Example: Dijkstra's Algorithm

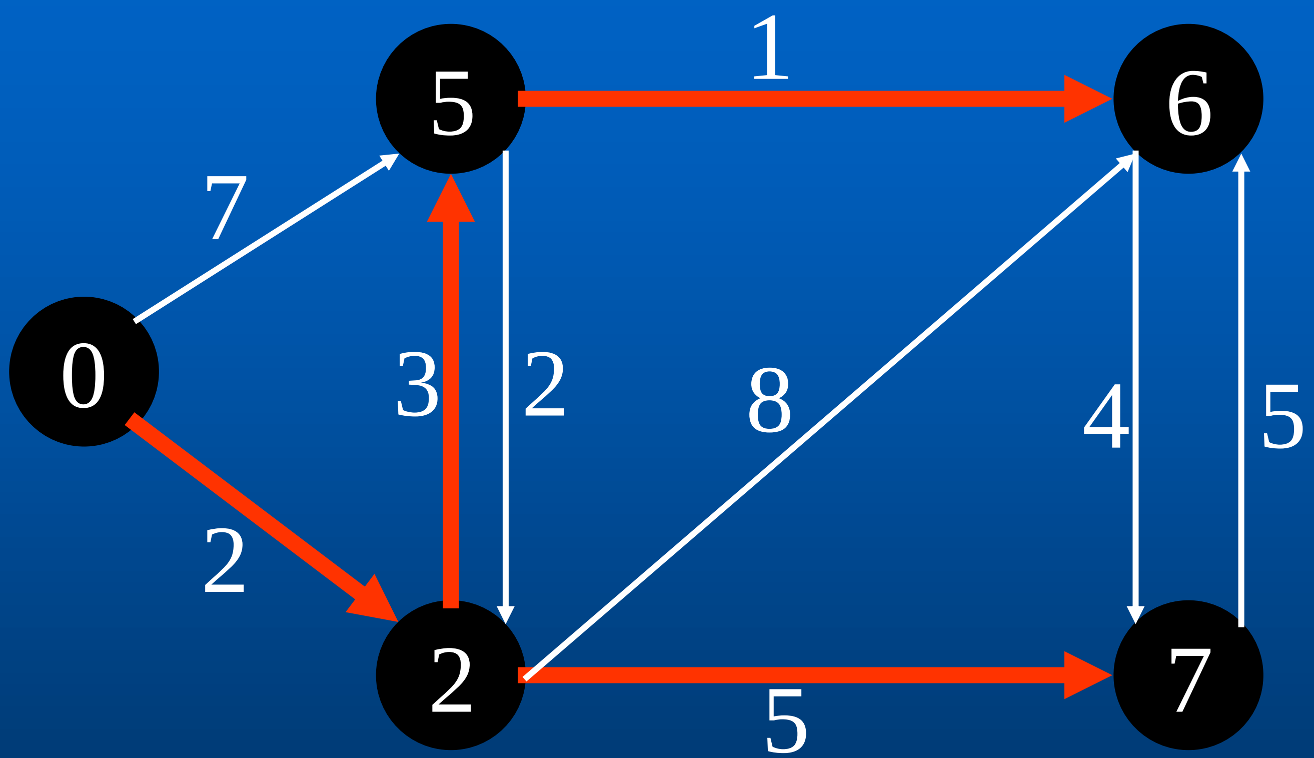


Black: in *S*

Example: Dijkstra's Algorithm

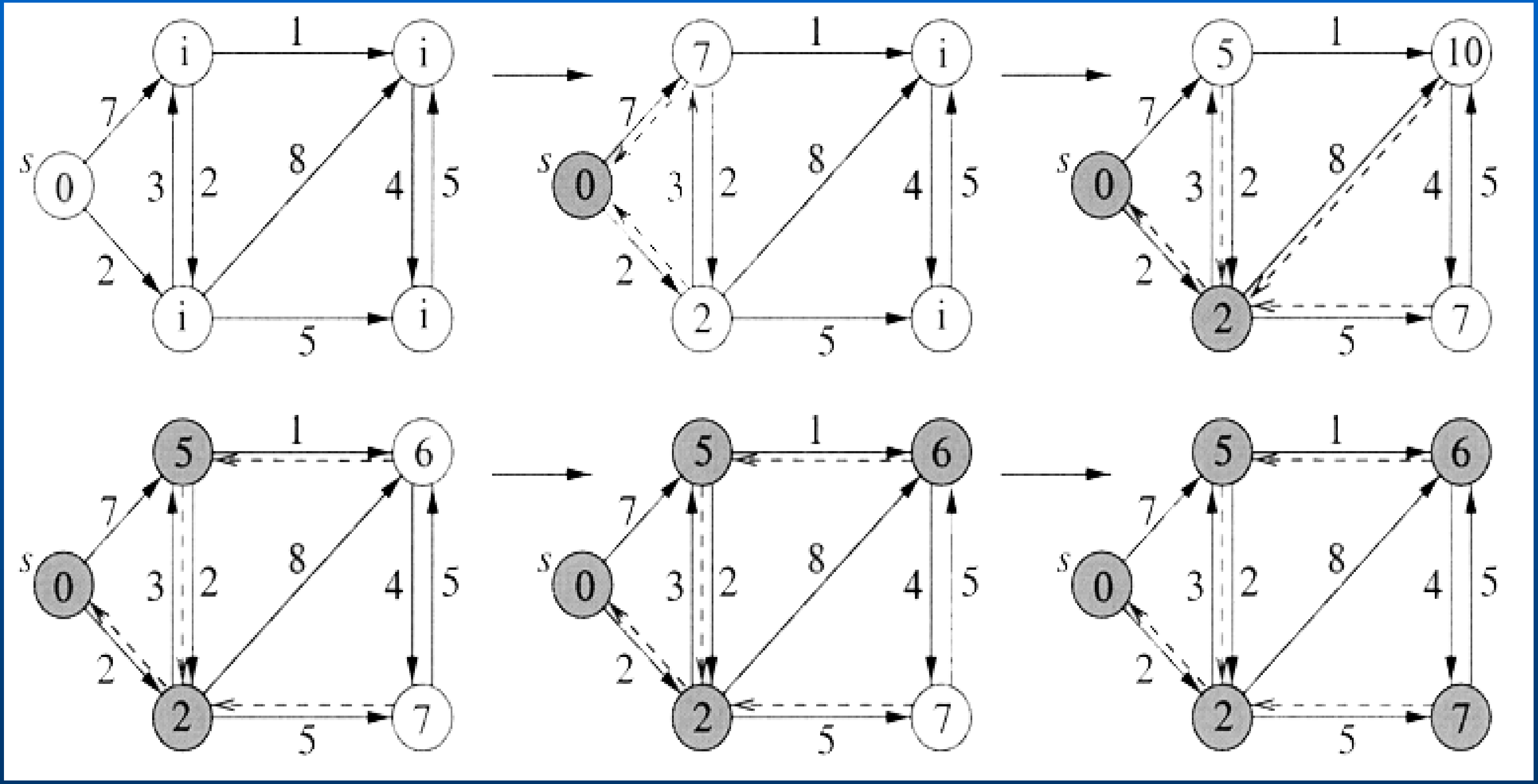


Example: Dijkstra's Algorithm



Black: in *S*

The Sequence of Relaxations



Correctness

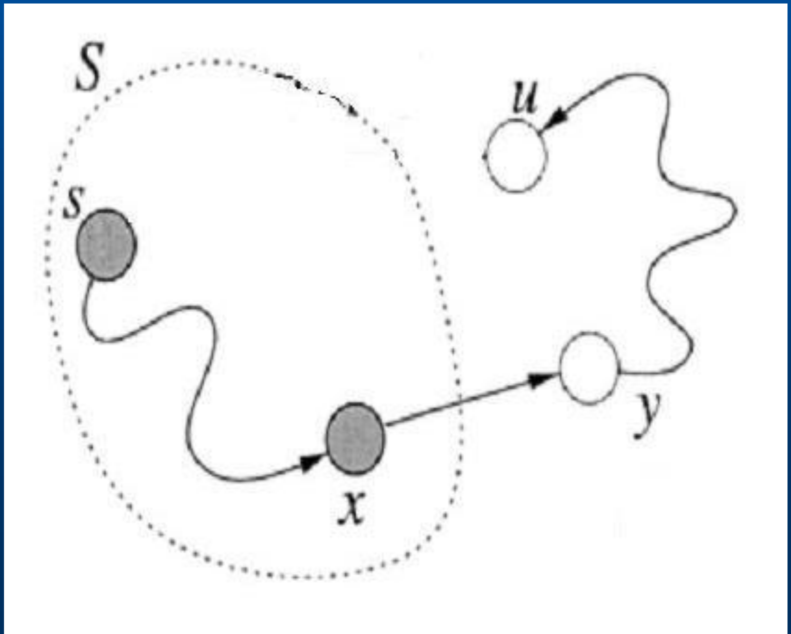
- Let $\delta(s,v)$ be length of true shortest path from s to v. Need to show: $d[v]=\delta(s,v)$ for each v when the algorithm terminates
- Lemma: When a vertex u is added to S, $d[u]=\delta(s,u)$.

Proof:

- Suppose that at some point Dijkstra's algorithm first attempts to add a vertex u to S for which $d[u] \neq \delta(s,u)$.
  - Note $d[u]$ is never less than $\delta(s,u)$, thus $d[u]>\delta(s,u)$.

Proof (I)

- **Just prior to the insertion of u, consider the true shortest path from s to u.**
  - **Because $s \in S$ and $u \in V - S$, at some point this path must first jump out of S.**
  - **Let (x, y) be the edge where it jumps out, so that $x \in S$ and $y \in V - S$. (It might happen that $x=s$ and/or $y=u$).**



Proof (II): $y \neq u$

- We argue that $y \neq u$ after all
  - Since $x \in S$ we have $d[x]=\delta(s,x)$. (Remember that u was the first vertex added to S that violated this criterion.)
  - Since we applied relaxation to x when it was added, we would have set $d[y] = d[x] + w(x,y) = \delta(s,y)$.
  - Since (x,y) is on a shortest path from s to u, it is on a shortest path from s to y. Thus $d[y]$ is now correct.
  - By hypothesis, $d[u]$ is not correct, so u and y cannot be the same.

Conclusion of Proof

- **Notice:**
  - y appears somewhere along the shortest path from s to u (but not at u)
  - All subsequent edges following y are of weight ≥ 0,
- **Therefore,** $\delta(s,y) \leq \delta(s,u)$**, so** $d[y]=\delta(s,y) \leq \delta(s,u)<d[u]$
- **But,** $d[u] \leq d[y]$ **because** u **is added before** y**, and we add nodes with lower** d **values first.**

Contradiction!

Correctness

- We have just proved that, when a vertex u is added to S, $d[u]=\delta(s,u)$.
- Every vertex is eventually added to S, so the algorithm assigns the correct distances to all vertices.
- This guarantees that the shortest-paths tree is correct because relaxation correctly updates parent pointers (see book for details).