Single Source Shortest Paths

Jared Saia
University of New Mexico

Today's Outline

"The path that can be trodden is not the enduring and unchanging Path. The name that can be named is not the enduring and unchanging Name." - Tao Te Ching

- BFS and DFS
- Single Source Shortest Paths
- Dijkstra's Algorithm
- Bellman-Ford Algorithm

Generic Traverse

Initially mark every vertex unvisited. For directed graphs, scan outgoing edges.

Traverse(s)

$B \leftarrow$ bag containing (None, s)
while $B \neq \varnothing$ **do**
  Remove a pair (p, v) from B
  if v is unvisited **then**
    Mark v visited; parent[v] $\leftarrow p$
    for each $w \in$ Adj[v] **do**
      Insert (v, w) into B

DFS and BFS

- If we implement the "bag" by using a stack, we have *Depth First Search*
- If we implement the "bag" by using a queue, we have *Breadth First Search*

Analysis

- Note that if we use adjacency lists for the graph, the overhead for the "for" loop is only a constant per edge (no matter how we implement the bag)
- If we implement the bag using either stacks or queues, each operation on the bag takes constant time
- Hence the overall runtime is $O(n + m)$

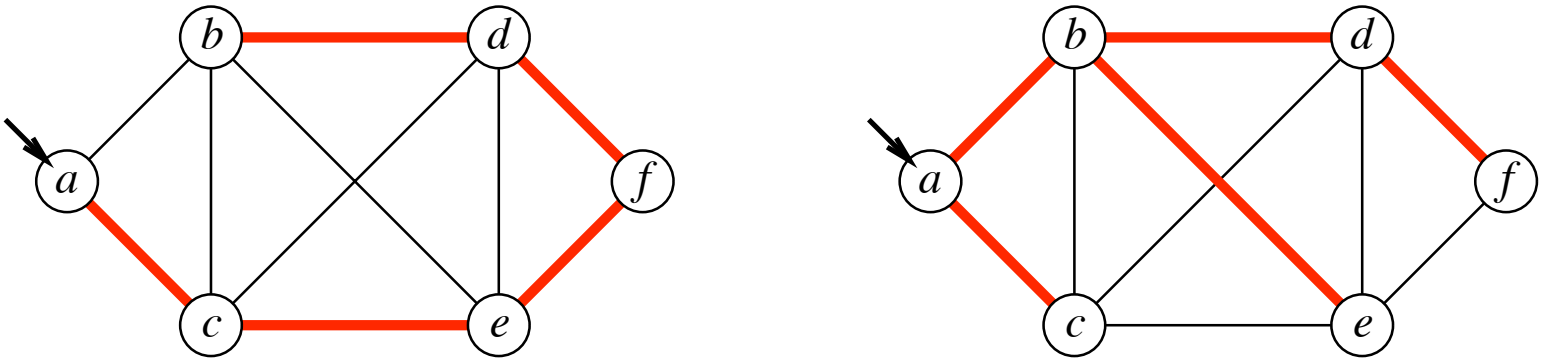
DFS vs BFS

- Note that DFS trees tend to be long and skinny while BFS trees are short and fat
- In addition, the BFS tree contains *shortest paths* from the start vertex s to every vertex reachable from s. (here we define the length of a path to be the number of edges in the path)

Final Note

- Now assume the edges are weighted
- If we implement the "bag" using a *priority queue*, always extracting the minimum weight edge from the bag, then we have a version of Prim's algorithm
- Each extraction from the "bag" now takes $O(\log m)$ time so the total running time is $O(n + m \log m)$

DFS and BFS Trees



DFS (left) and BFS (right), both starting at a.

Visiting Every Vertex

One traversal visits exactly the vertices reachable from its start vertex. To visit the entire graph, keep the marks between calls:

TraverseAll(G)

Mark every vertex unvisited
for each $v \in V$ **do**
if v is unvisited **then**
Traverse(v)

DFS and BFS

- Note that we can do DFS and BFS equally well on undirected and directed graphs
- If the graph is undirected, there are two types of edges in G: edges that are in the DFS or BFS tree and edges that are not in this tree
- If the graph is directed, there are several types of edges

DFS in Directed Graphs

- *Tree edges* are edges that are in the tree itself
- *Back edges* are those edges (u, v) connecting a vertex u to an ancestor v in the DFS tree
- *Forward edges* are nontree edges (u, v) that connect a vertex u to a descendant in a DFS tree
- *Cross edges* are all other edges. They go between two vertices where neither vertex is a descendant of the other

Acyclic graphs

- Useful Fact: A directed graph G is acyclic if and only if a DFS of the entire graph G yields no back edges
- Challenge: Try to prove this fact.

Take Away

- BFS and DFS are two useful algorithms for exploring graphs
- Each of these algorithms is an instantiation of the Traverse algorithm. BFS uses a queue to hold the edges and DFS uses a stack
- Each of these algorithms constructs a spanning tree of all the nodes which are reachable from the start node s

Shortest Paths Problem

- Another interesting problem for graphs is that of finding shortest paths
- Assume we are given a weighted *directed* graph $G = (V, E)$ with two special vertices, a source s and a target t
- We want to find the shortest directed path from s to t
- In other words, we want to find the path p starting at s and ending at t minimizing the function

$$w(p) = \sum_{e \in p} w(e)$$

Example

- Imagine we want to find the fastest way to drive from Albuquerque,NM to Seattle,WA
- We might use a graph whose vertices are cities, edges are roads, weights are driving times, s is Albuquerque and t is Seattle
- The graph is directed since driving times along the same road might be different in different directions (e.g. because of construction, speed traps, etc)

Single Source Shortest Paths

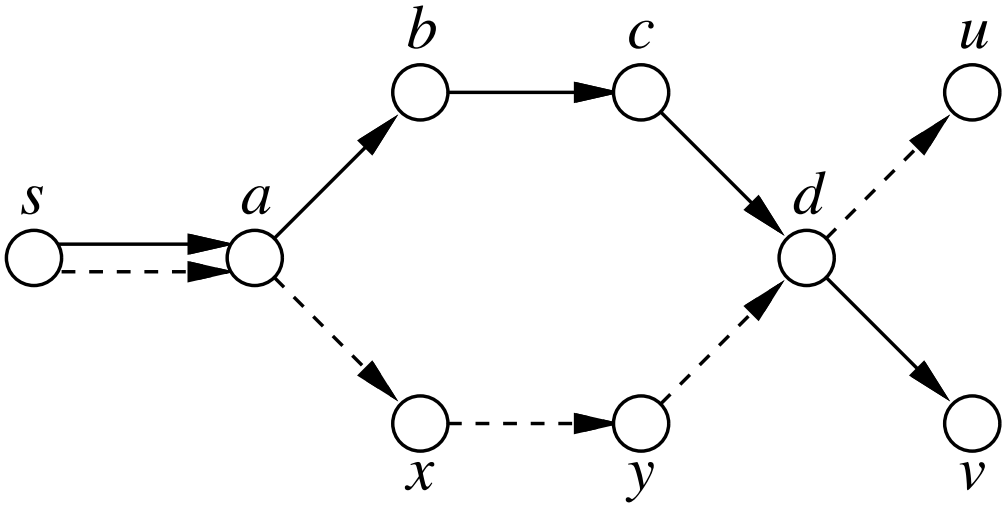
- We will solve the more general *single source shortest paths* (SSSP) problem.
- Find the shortest-path distance from a source s to every vertex reachable from s; unreachable vertices have distance ∞.
- When no negative cycle is reachable from s, these paths can be represented by a *shortest path tree* rooted at s.

Shortest Path Tree

Assume no negative cycle is reachable from s.

- A shortest path tree contains exactly the vertices reachable from s, and its path from s to each vertex is shortest.
- Subpaths of shortest paths are themselves shortest paths.
- If shortest paths are unique, they form this tree.
- With ties, choose paths consistently: whenever paths share a vertex, use the same prefix from s. Such a tree always exists.

Consistent Shortest Paths

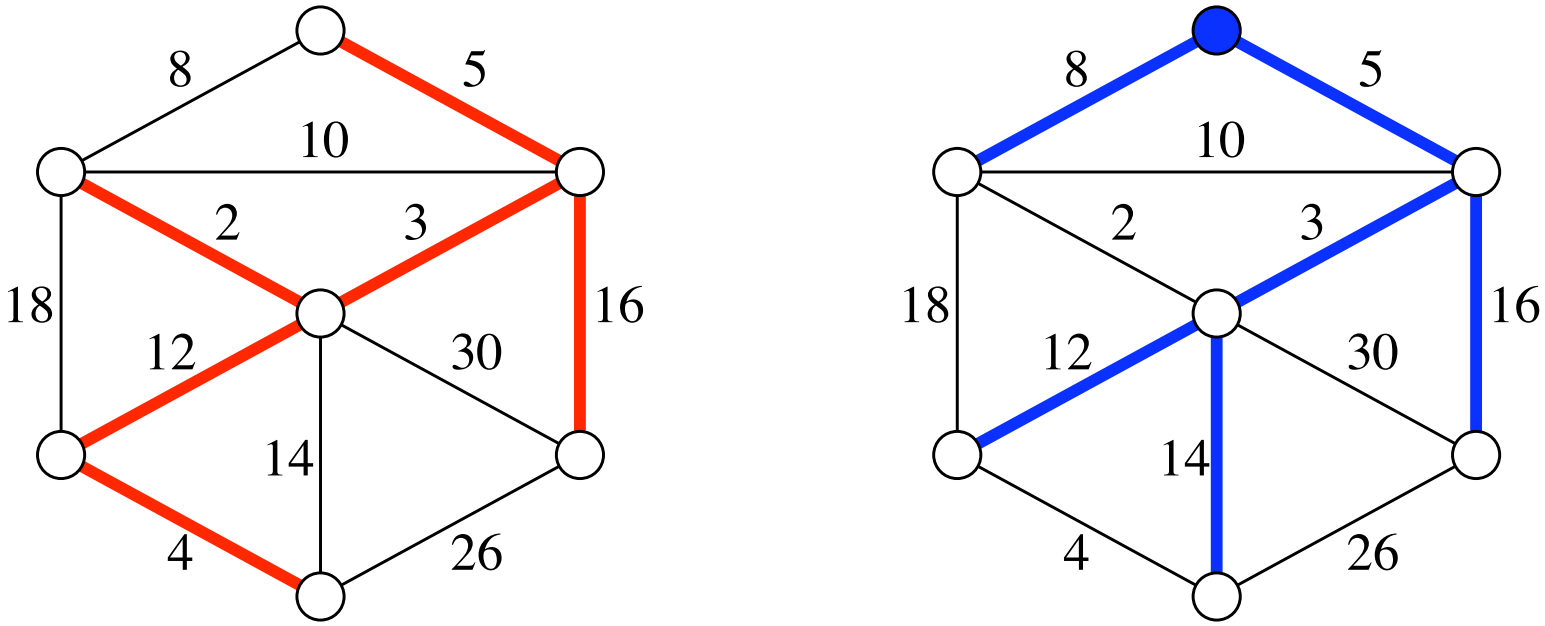


If both routes to d are shortest, either prefix can be used for both u and v.

MST vs SPT

- A **minimum spanning tree** minimizes the total weight of its edges.
- A **shortest path tree** preserves shortest-path distances from a specified source.
- These are different objectives, so the trees can differ.
- An SPT depends on the source and may not be unique when paths tie.

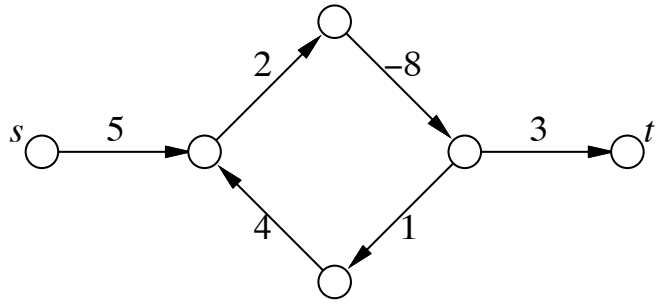
MST and SPT



MST (left); SPT rooted at the top vertex (right).

Negative Weights

- We'll actually allow negative weights on edges
- The presence of a negative cycle might mean that there is no shortest path
- A finite shortest-path distance from s to t exists exactly when t is reachable from s and no negative cycle reachable from s can also reach t
- In the following example, there is no shortest path from s to t



SSSP Algorithms

- We study shortest paths in weighted directed graphs.
- For undirected graphs with nonnegative weights, replace each edge by two oppositely directed edges of the same weight.
- A reachable undirected negative edge permits arbitrarily negative walks by repeated backtracking.
- The algorithms differ in how they choose edges to relax.

Distances and Predecessors

Each vertex v stores:

- $dist(v)$: a tentative distance from s, initially ∞ except at the source.
- $pred(v)$: the vertex used in the most recent improvement to $dist(v)$, initially None.
- When no tense edges remain, predecessor paths give the final distances to all reachable vertices.

Initialize Distances

InitSSSP(G, s)

  for each $v \in V$ **do**

    $dist(v) \leftarrow \infty$

    $pred(v) \leftarrow$ None

  $dist(s) \leftarrow 0$

Relaxation

An edge (u, v) is *tense* if $dist(u) + w(u, v) < dist(v)$. Relaxation uses it to improve the tentative distance to v.

Relax(u, v)

  if $dist(u) + w(u, v) < dist(v)$ **then**

    $dist(v) \leftarrow dist(u) + w(u, v)$

    $pred(v) \leftarrow u$

    return True

  return False

Our generic algorithm keeps relaxing tense edges until none remain.

Generic SSSP

GenericSSSP(G, s)

InitSSSP(G, s)

$B \leftarrow$ bag containing s

while $B \neq \varnothing$ **do**

Remove a vertex u from B

for each outgoing edge (u, v) **do**

if Relax(u, v) **then**

Insert v into B if absent

return $dist, pred$

Correctness

- The correctness of the relaxation algorithm follows directly from three simple claims
- The run time of the algorithm will depend on the way that we make choices about which edges to relax

Claim 1

Assume no negative cycle is reachable from s. For each vertex v with finite $dist(v)$, its predecessors form a path P_v from s to v, and

$$dist(v) \geq w(P_v).$$

Proof left as an exercise: an easy induction on the number of successful edge relaxations.

Claim 2

- CLAIM 2: If the algorithm halts, then $dist(v) \leq w(s \rightsquigarrow v)$ for *any* path $s \rightsquigarrow v$.
- This is easy to prove by induction on the number of edges in the path $s \rightsquigarrow v$. (left as an exercise)

Claim 3

Claim: Repeatedly relaxing tense edges halts exactly when no negative cycle is reachable from s.

- **No reachable negative cycle:** each finite $dist(v)$ value is the weight of some simple path from s to v; completing a nonnegative cycle cannot give a strict improvement. There are finitely many simple paths, and each successful relaxation strictly decreases an estimate. Thus only finitely many relaxations are possible.
- **Reachable negative cycle C:** at termination, all its vertices would have finite estimates and no edge would be tense. Summing $dist(v) \leq dist(u) + w(u, v)$ around C gives $0 \leq w(C) < 0$, a contradiction.

Generic SSSP

- Generic SSSP uses a bag to choose which vertex to scan next
- The choice of bag determines the order of relaxations
- We will maintain a "bag" of vertices initially containing just the source vertex s
- Whenever we take a vertex u out of the bag, we scan all of its outgoing edges, looking for something to relax
- Whenever we successfully relax an edge (u, v), we put v in the bag if it is not already there

Generic SSSP

- Just as with graph traversal, using different data structures for the bag gives us different algorithms
- Some obvious choices are: a stack, a queue and a heap
- With a stack, the number of relaxation steps can be exponential
- The other possibilities are more efficient

Dijkstra's Algorithm

Assume all edge weights are nonnegative.

- Initialize $dist(s) = 0$ and all other distances to ∞. Put every vertex in a min-heap keyed by $dist$.
- Keep one heap entry per unprocessed vertex. After a distance improves, update its entry with DecreaseKey.
- Extract a minimum-key vertex u and relax its outgoing edges to vertices still in the heap.

Dijkstra: Proof Setup

- Assume all edge weights are nonnegative.
- Let $\delta(v)$ be the true shortest-path distance from s to v.
- Estimates never underestimate: $dist(v) \geq \delta(v)$.
- To *scan* a vertex means to extract it from the heap and process it in the body of the while loop: relax its outgoing edges to vertices still in the heap.
- Stop if the minimum key is ∞. All remaining vertices are unreachable; otherwise a path from s would give its first unprocessed vertex a finite key.

Dijkstra: Correctness

Theorem 1: Each scanned v has $dist(v) = \delta(v)$.

Proof. Induct on k, the number of scanned vertices.

Base case ($k = 1$): Scan s first, with $dist(s) = 0 = \delta(s)$.

Inductive step: Assume the first $k-1$ vertices had correct distances after scanning; let v be next. Let y be the first unscanned vertex on a shortest path from s to v, and let x be the vertex just before y.

By the IH, after scanning x, $dist(x) = \delta(x)$. That scan relaxed (x, y), giving $dist(y) \leq \delta(x) + w(x, y) = \delta(y)$.

The heap's ExtractMin property gives $dist(v) \leq dist(y)$; nonnegative weights give $\delta(y) \leq \delta(v)$. Hence,

$$\delta(v) \leq dist(v) \leq dist(y) \leq \delta(y) \leq \delta(v).$$

Thus $dist(v) = \delta(v)$. □

Dijkstra's Runtime

Assume nonnegative weights and one heap entry per vertex.

- Scan adjacency lists in $O(n + m)$ total time.
- At most n insertions, n extractions, and m key decreases.
- **Binary heap:** $O(\log n)$ per heap operation, giving

$$O((n + m) \log n).$$

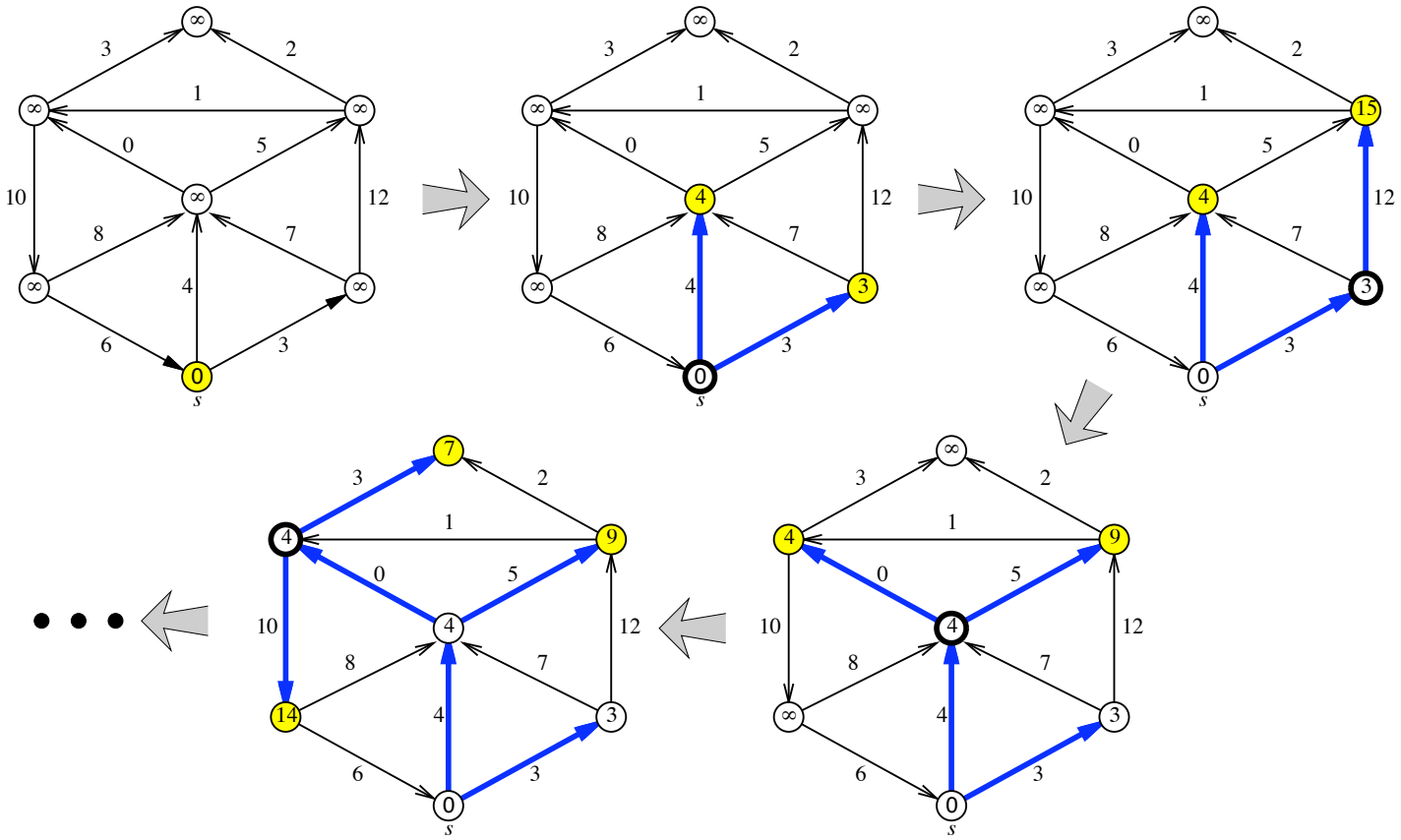
- **Fibonacci heap:** amortized $O(1)$ key decreases and $O(\log n)$ extractions, giving

$$\boxed{O(m + n \log n)}.$$

Negative Edges

- Dijkstra's algorithm requires nonnegative edge weights.
- With negative edges, use Bellman-Ford. It also detects negative cycles reachable from the source.

Dijkstra's Algorithm



First four scans. Yellow: in the heap; bold outline: just scanned.
Blue edges: current predecessors.

Bellman-Ford

- Implement the bag in GenericSSSP as a FIFO queue. When a distance improves, enqueue that vertex if it is absent.
- Start with s followed by a token. A *phase* scans all vertices before the token; phase 0 scans only s.
- On removing the token, stop if the queue is empty; otherwise, reinsert the token at the back to begin the next phase.
- To simplify, replace each queue phase by a pass that relaxes *every* edge. This gives the algorithm on the next slide.
- Negative edge weights are allowed: use $n - 1$ passes, then check for a reachable negative cycle.

Bellman-Ford

Each queue phase becomes a full pass over all edges:

BellmanFord(G, s)

  InitSSSP(G, s)

  repeat $n - 1$ **times**

    for each edge $(u, v) \in E$ **do**

      Relax(u, v)

  for each edge $(u, v) \in E$ **do**

    if $dist(u) + w(u, v) < dist(v)$ **then**

      return "Reachable negative cycle"

  return $dist, pred$

Bellman-Ford: Why It Works

Theorem 2: After pass i, $dist(v) \leq w(P)$ for every s-to-v path P with at most i edges.

Proof. Induct on the number i of completed passes.

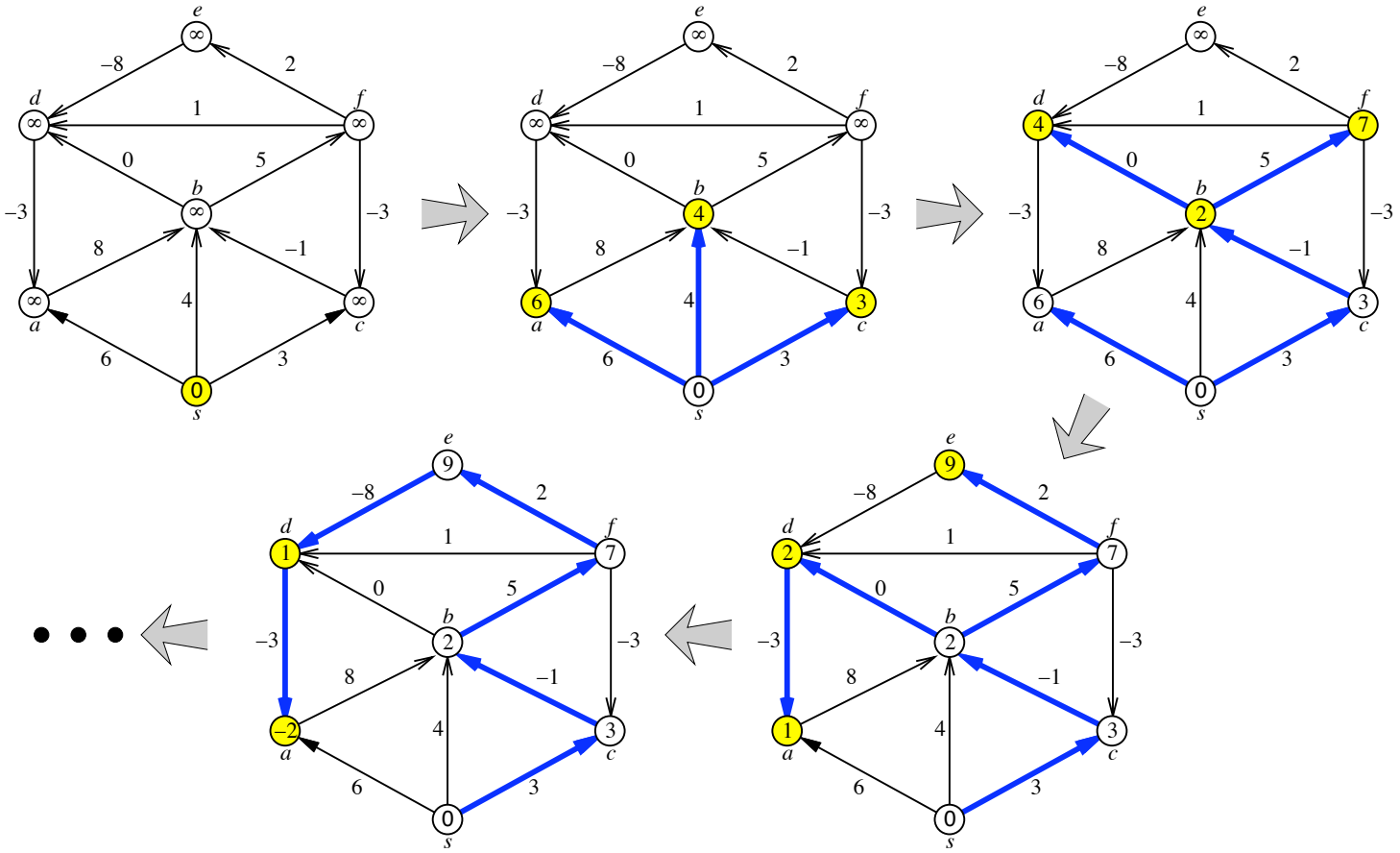
Base case ($i = 0$): The only zero-edge path from s ends at s, and $dist(s) = 0$.

Inductive step: Assume the bound after pass $i - 1$. Take a nonempty path P with at most i edges, ending with (u, v). Its prefix P' uses at most $i - 1$ edges, so by IH, $dist(u) \leq w(P')$ before pass i. When (u, v) is relaxed in that pass,

$$dist(v) \leq dist(u) + w(u, v) \leq w(P') + w(u, v) = w(P).$$

Distances only decrease, so the bound holds at the end of the pass. □

Bellman-Ford: Queue Example



Queue version: initial state, then token-delimited phases 0–3.
Yellow: queued vertices; blue edges: current predecessors.
The simplified algorithm replaces each phase by a full edge pass.

Bellman-Ford's Runtime

- Initialization takes $O(n)$ time.
- Each pass scans m edges in $O(m)$ time.
- There are $n - 1$ relaxation passes and one final check.
- Total: $O(n + nm)$, usually written $O(nm)$ when $m \geq 1$.
- If an entire pass makes no changes, we may stop early.

Algorithm Comparison

Algorithm	**Edge weights**	**Time**
BFS	All equal to 1	$O(n+m)$
Dijkstra	Nonnegative	$O(m+n\log n)$
Bellman-Ford	May be negative	$O(n+nm)$

Dijkstra's bound uses a Fibonacci heap. Bellman-Ford also detects negative cycles reachable from the source.