# Shortest Path

References:

1. Algorithms, Jeff Erickson, Chapter 8
2. Algorithm Design Manual, Skiener, Chapter 6

Problem: Given a directed weighted graph $G = (V, E, w)$ with two special nodes $s, t$, what is the shortest path from $s$ to $t$? (the length of a path is the sum of the all the weights of the edges on the path).

# Shortest Path Trees



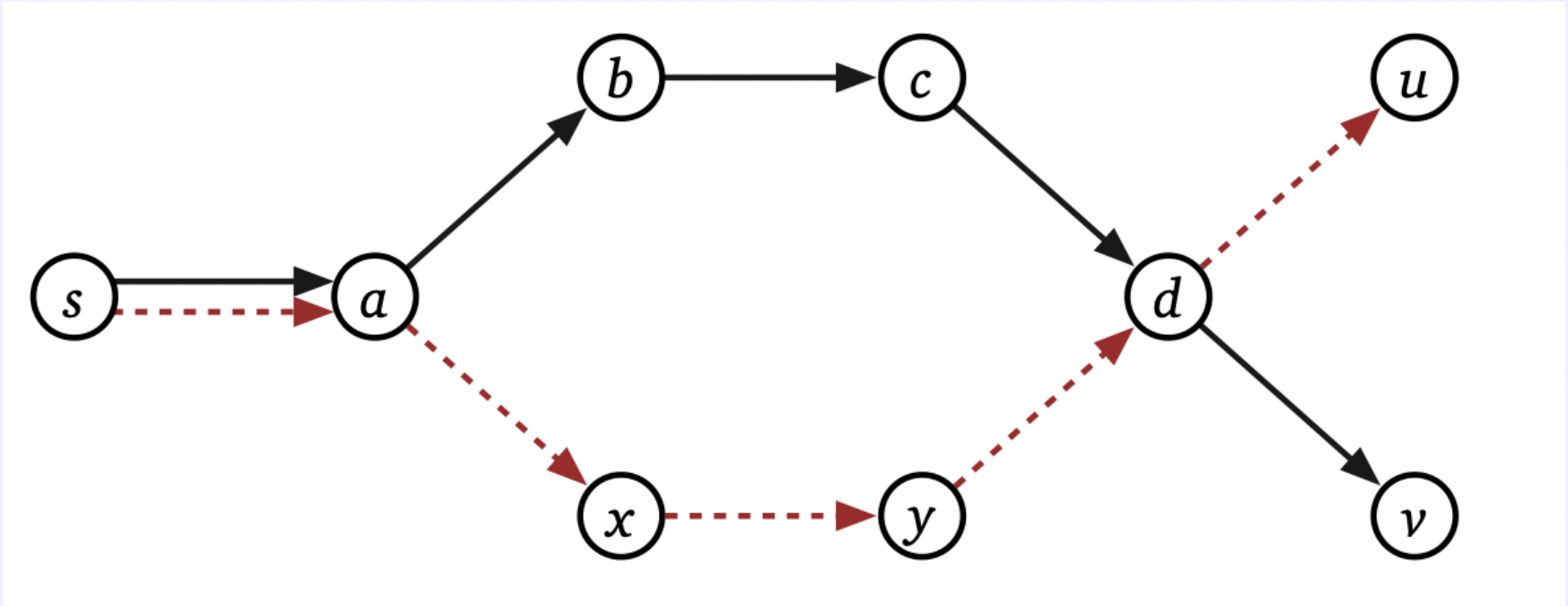
To determine the shortest path from $s$ to $t$, almost all algorithms actually computes more general **single source shortest path (SSSP)** problem:

SSSP: find the shortest path from $s$ to every other nodes.

This problem is solved by finding a **shortest path tree**, rooted at $s$, which contains all desired shortest paths.

Why do all shortest paths constitute a tree?

1. If all shortest paths are unique, then the union of shortest paths is a tree (recall: unique paths towards a root node $\longrightarrow$ tree)
2. If there are multiple shortest paths to $t$, we can pick and choose to make the union a tree. For example:




If s->u (solid) and s->v (dashed) are two shortest paths, the paths a->d through solid edges and dashed edges must both be shortest, and we can simply include the solid edges in our tree.

**Differences between Minimium Spanning Tree (MST) and Shortest Path Tree (SPT)**:

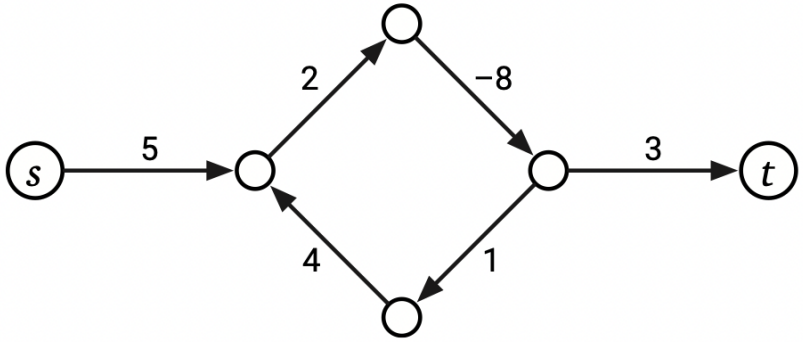
1. MST for undirected graph; SPT for directed graph (can be adapted to undirected too with some caveats)
2. MST does not have a root; SPT does.
3. MST can be unique; SPT are distinct for different root.

# Negative Edges



Negative edges is a pain for SSSP.

If a cycle is negative, then shortest path may not be well-defined!




For example, what is the shortest path from $s$ to $t$?

Negative edges also make our algorithms for **undirected** graph complicated. The union of all shortest paths need not be a tree.

We don't tree **undirected graph with negative edges** here.

# The Only SSSP Algorithm



Just like graph traversal and MST algorithms, there is a "meta" SSSP algorithm:

Each node $v$ maintains two values which describes a **tentative shortest path** from source $s$ to $v$:

- $\text{dist}(v)$: the length of the tentative shortest path; $\infty$ if no path exist.
- $\text{pred}(v)$: the predecessor of $v$ in the tentative shortest path. NULL if no path exist.

The $\text{pred}(v)$ defines a tree rooted at source $s$, like the parent(v) in WhateverFirstSearch().

**The SSSP algorithm:**

- Initialize $\text{dist}(v) \leftarrow \infty$ and $\text{pred}(v) \leftarrow \text{NULL}$ for $v \neq s$.

  Initialize $\text{dist}(s) \leftarrow 0, \text{pred}(v) \leftarrow \text{NULL}$.

- During the execution of algorithm, an edge $u \rightarrow v$ is **tense**, if

$$\text{dist}(u) + w(u \rightarrow v) < \text{dist}(v)$$

  For tense edge $u \rightarrow v$, the tentative shortest path stored in v is clearly incorrect (overestimtation): we already find a better route through $u \rightarrow v$.

- We can correct the overestimate by **relaxing** the edges:

$$\begin{aligned}&\text{Relax}(u \rightarrow v):\\ &\quad \text{dist}(v) \leftarrow \text{dist}(u) + w(u \rightarrow v)\\ &\quad \text{pred}(v) \leftarrow u\end{aligned}$$

Now we have all the pieces, the SSSP algorithm can be described:

**Repeatedly relax tense edges, until no more tense edges.**

```
FordSSSP(s):
    InitSSSP(s)
    while there is at least one tense edge
        Relax any tense edge
```

After the algorithm terminates (no tense edges exist), all $\text{dist}(v)$ will be correct, and $\text{pred}(v)$ will define a SPT.

Some subtleties:

- If $\text{dist}(v) = \infty$, then $v$ is not reachable from source $s$.
- If any negative cycle is reachable from $s$, then the algorithm will never terminate.

## Why is FordSSSP() algorithm correct?

1. At any moment, for every node v, the dist(v) is either $\infty$ or the length of **a walk** from s to v.
2. If the graph has no negative cycles, then the dist(v) is either $\infty$ or the length of **a simple path** from s to v. This implies that, without negative cycles, the algorithm terminates in finite steps.
3. If no edge is tense, then dist(v) is the length of path $s \to \cdots \to \text{pred}(\text{pred}(v)) \to \text{pred}(v) \to v$.

   Futhermore, If $v$ violates this condition, but $\text{pred}(v)$ does not, then $\text{pred}(v) \to v$ is tense.
4. If no edge is tense, then $s \to \cdots \to \text{pred}(\text{pred}(v)) \to \text{pred}(v) \to v$ is indeed the shortest path to v.

   Furthermore, If $v$ violates this condition, but $\text{pred}(v)$ does not, then $\text{pred}(v) \to v$ is tense.

Exactly **how** to find the tense edges, and **which** edges to relax in what order is not being said here.

There are serveral variants, whose efficiency and correctness depends on the structure of the input graph.

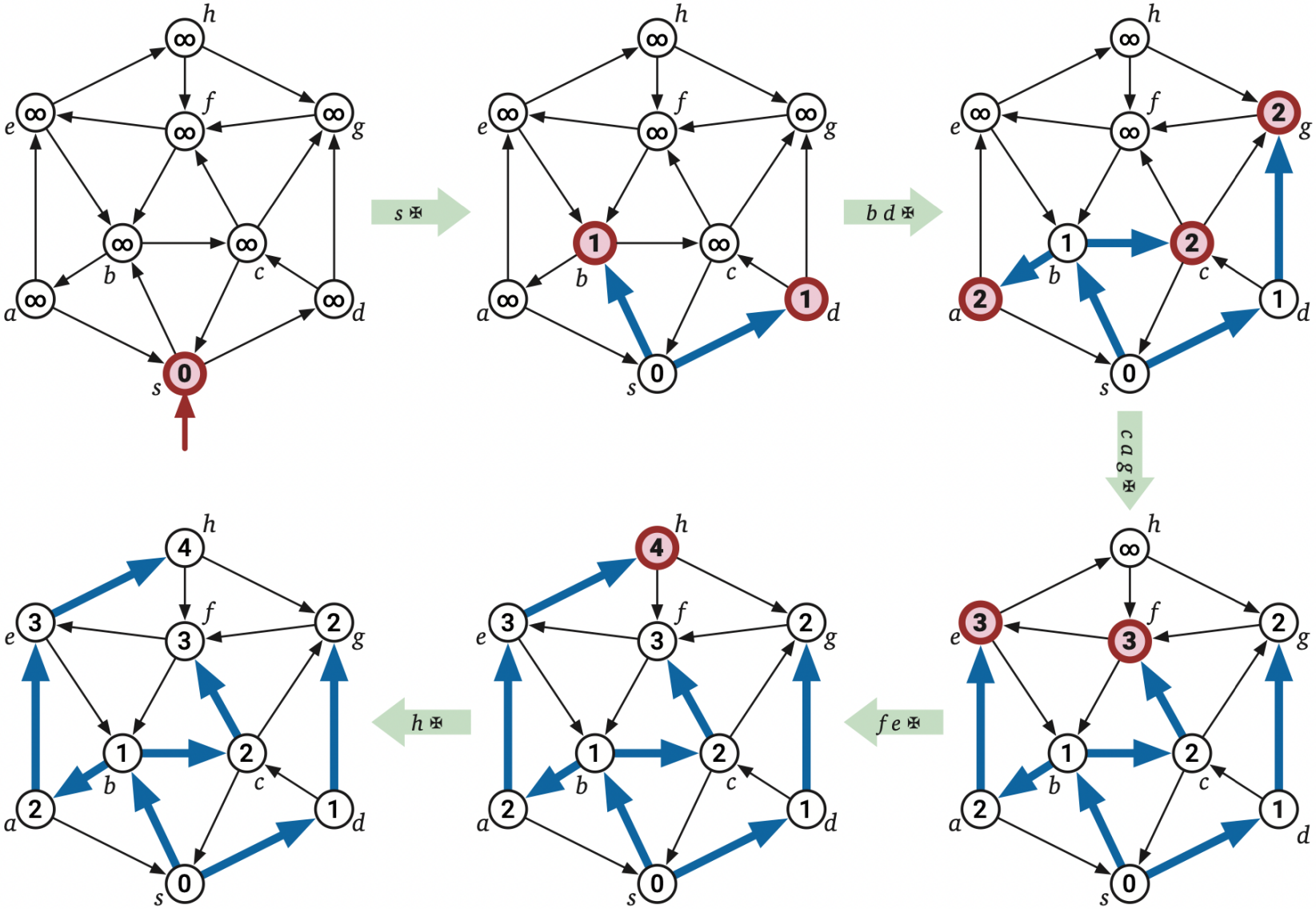
We are going to look at 4 instantiations of Ford's algorithm. Although we can use the general FordSSSP() proof of correctness to argue the instantiation correctness, we may prove in their own context for concreteness.

# Algo #1: BFS for unweighted graph



In the simplest special SSSP, the edge is unweighted, or equivalently, all edges have the same weight 1. In this case, BFS finds the SSSP solution in $O(V + E)$ time:

```
BFS(s):
    InitSSSP(s)
    Push(s)
    while the queue is not empty
        u ← Pull( )
        for all edges u→v
            if dist(v) > dist(u) + 1          ⟨⟨if u→v is tense⟩⟩
                dist(v) ← dist(u) + 1
                pred(v) ← u                   ⟨⟨relax u→v⟩⟩
                Push(v)
```


h
f
e
g
b
c
a
d
s
∞
0
1
2
3
4
s ✠
b d ✠
c a g ✠
f e ✠
h ✠

# Algo #2: DFS for DAG



SSSP for Directed Acyclic Graph (DAG) is also easy to compute, even if some edges have negative weights.

Since DAG has no cycle, SSSP is always well defined.

The dist(v) satisfies the obvious recurrence:

$$dist(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{u \to v} (dist(u) + w(u \to v)) & \text{otherwise} \end{cases}$$

In fact this recurrence holds not only for DAG, but for all graphs!

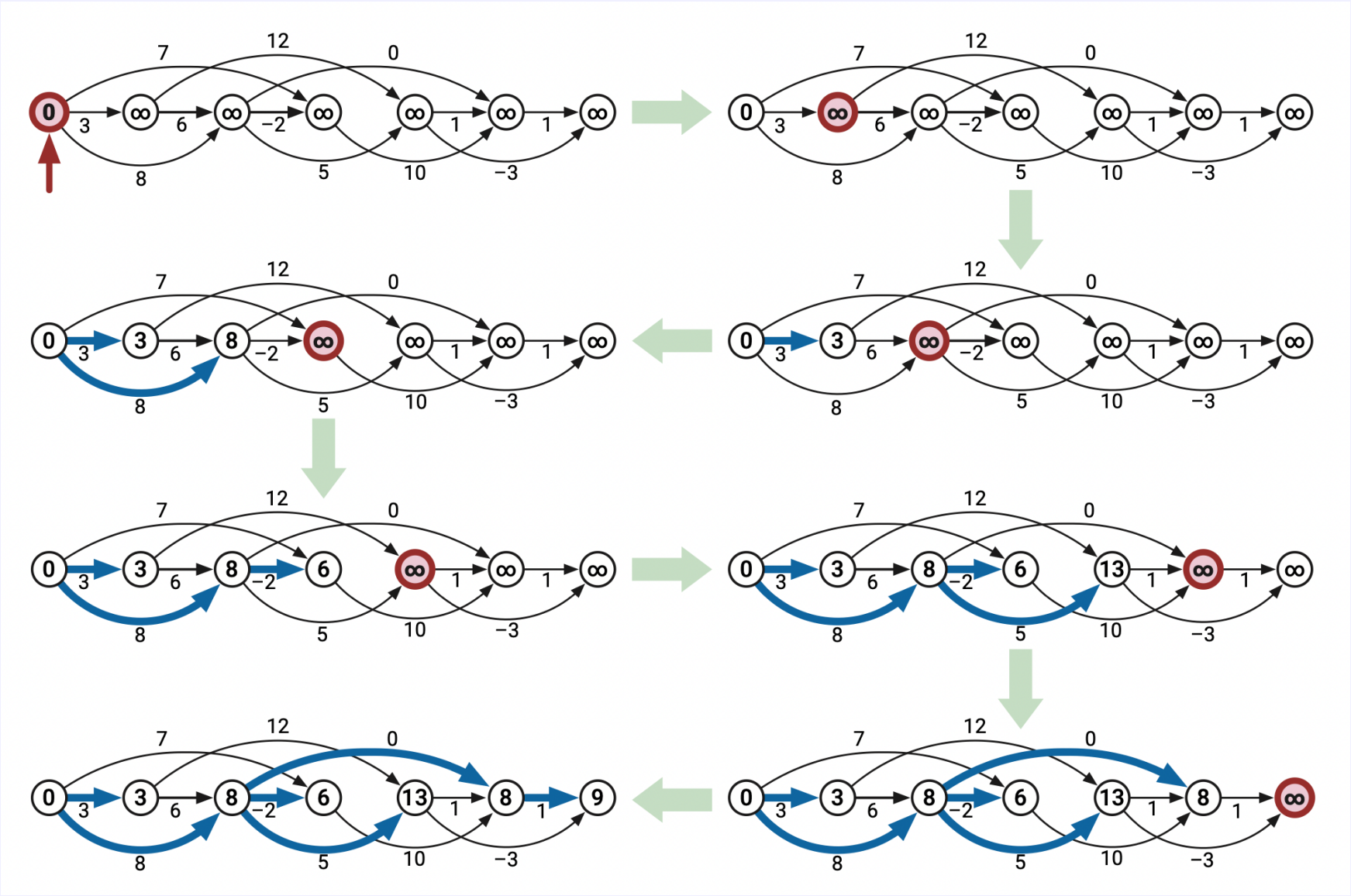
However, only for DAG this recurrence is computable. (Why? Because if there is cycle, then the recursion never terminates).

For DAG, this recurrence actually lead to dynamic programming.

```
DAGSSSP(s):
  for all vertices v in topological order
      if v = s
          dist(v) ← 0
      else
          dist(v) ← ∞
          for all edges u→v
              if dist(v) > dist(u) + w(u→v)        ⟨⟨if u→v is tense⟩⟩
                  dist(v) ← dist(u) + w(u→v)       ⟨⟨relax u→v⟩⟩
```

In fact, this dynamic programming algorithm can be viewed as Ford's algorithm instantiation:

```
DAGSSSP(s):
  INITSSSP(s)
  for all vertices v in topological order
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
```




We can also visit all **outgoing** edges instead of **incoming** edges.

# Algo #3: Best-First: Dijkstra's



If we modify BFS by using Priority-Queue instead of FIFO queue, we arrive at Dijkstra's algorithm:

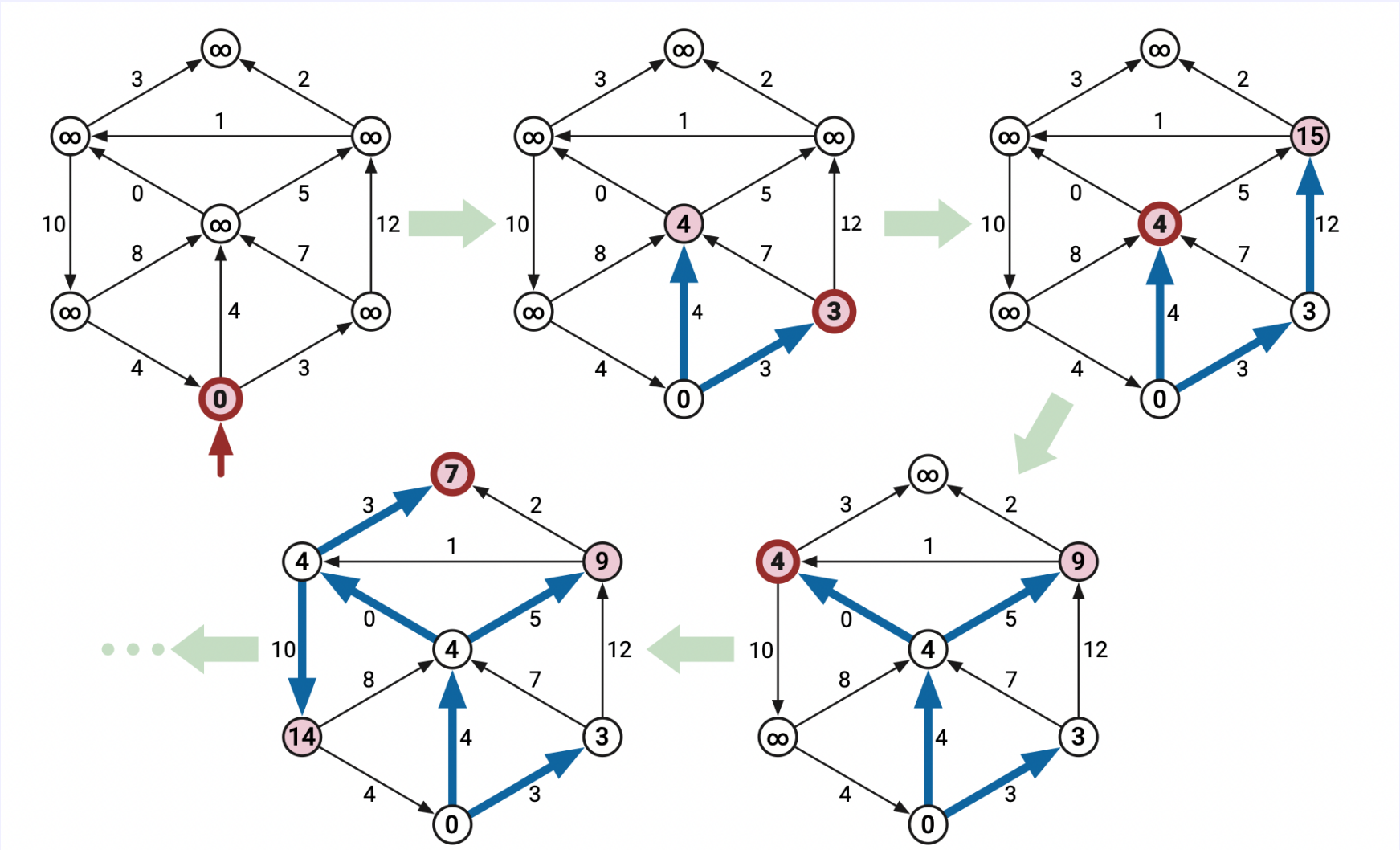
```
DIJKSTRA(s):
    INITSSSP(s)
    INSERT(s, 0)
    while the priority queue is not empty
        u ← EXTRACTMIN( )
        for all edges u→v
            if u→v is tense
                RELAX(u→v)
                if v is in the priority queue
                    DECREASEKEY(v, dist(v))
                else
                    INSERT(v, dist(v))
```

Dijkstra's algorithm is an instantiation of Ford's, so it correctly finds SSSP provided that there's no negative cycle.

To analyze performance, we divide into two cases:

**Case 1: No Negative Edges**: Dijkstra works great! $O(E \log V)$.

Dijkstra's algorithm works like BFS in a wavefront way, expanding the front like:




what is the time complexity? It's not obvious how many times a node can be put into the priority queue.

To analyze performance, we define:

- $u_i$ denotes the node returned by the i-th ExtractMin() returns.
- $d_i$ denotes $\text{dist}(u_i)$ immediately after the i-th ExtractMin().

$u_i, i = 1 \ldots$ need not be distinct.

In particular, we have $u_1 = s, d_1 = 0$.

And we would like to claims that **each node is ExtractMin() at most once**, therefore the Dijkstra algorithm runs in $O(E \log V)$ time, if the priority queue is implemented with binary heap.

**Lemma 1.** *If G has no negative-weight edges, then for all i<j, we have* $d_i \leqslant d_j$.

Proof: fix index $i$; we'd like to show $d_i \leqslant d_{i+1}$. Two cases:

1. If $u_i \longrightarrow u_{i+1}$ is an edge, and this edge is tense during the i-th iteration, then $\text{dist}(u_{i+1}) = \text{dist}(u_i) + w(u_i \longrightarrow u_{i+1}) \geqslant \text{dist}(u_i)$

because the non-negative edge weight.

2. Otherwise, $u_{i+1}$ must be already in the queue, and $\text{dist}(u_{i+1}) \geqslant \text{dist}(u_i)$ because we retrieve the min dist() which is $\text{dist}(u_i)$.

□

**Lemma 2.** *If G has no negative-weight edges, then each node of G is Extracted from the priority queue at most once.*

Proof: By contradiction. Suppose $v$ is extracted more than once. Suppose $v$ is extracted at i-th iteration, and re-inserted at j-th iteration, and re-extracted at k-th iteration, for some indices $i < j < k$, then $v = u_i = u_k$.

The $\text{dist}(u_i) = \text{dist}(v)$ must decrease strictly in j-th iteration, just before $v$ is re-inserted. Therefore $\text{dist}(u_k) < \text{dist}(u_i)$, which contradicts $u_k = u_i = v$. □

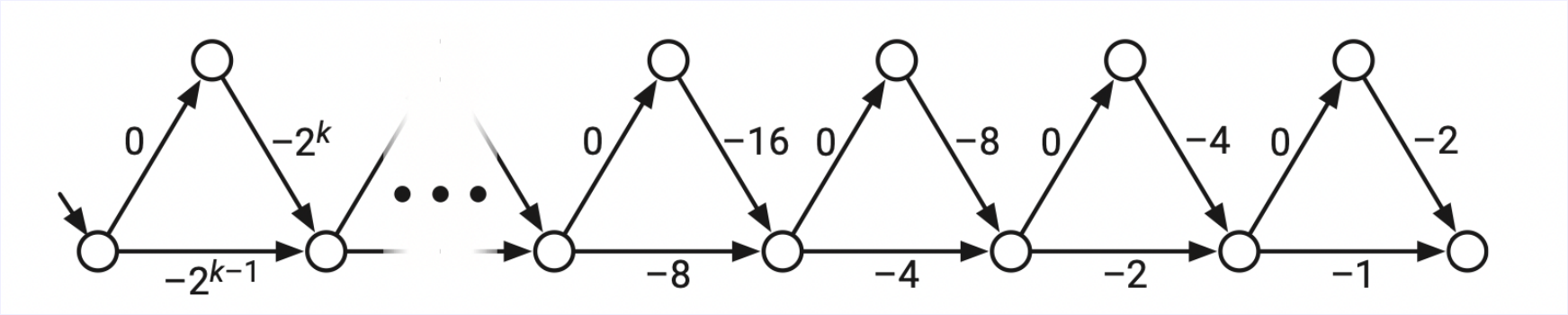
What does Lemma 2 tells us?

1. Each node is extracted at most once, thus the time complexity of the algorithm is $O(E \log V)$.

2. The first time a node is put into the priority queue, its dist(.) is $\infty$. Later it may get DecreaseKey() several times; after it's extracted, its dist(.) never changes again.

**Case #2: negative edges**: Dijkstra may be slow!

If there's negative edges, then the same node may be inserted/extracted multiple times!




For the above graph, Dijkstra's algorithm must do $\Theta(2^{V/2})$ relaxations. The worst case is actually $\Theta(2^V)$.

But in practice, Dijkstra's seems to work fast even with negative edges.

# Algo #4: Bellman-Ford: Relax all edges



The most general and simplest to describe:

BELLMAN-FORD: Relax ***ALL*** the tense edges, then recurse.

```
BellmanFord(s)
    InitSSSP(s)
    while there is at least one tense edge
        for every edge u→v
            if u→v is tense
                Relax(u→v)
```

What's the time complexity?

Define: for node $v$ and integer $i$, let $\mathbf{dist_{\leqslant i}(v)}$ denote the length of the shortest **walk** in G from s to v, **consisting of at most i** edges. In particular, $\text{dist}_{\leqslant 0}(s) = 0$ and $\text{dist}_{\leqslant 0}(v) = \infty$ for $v \neq s$.

**Lemma 3.** *For every node v and $i \geqslant 0$, after i iteration of main loop in BellmanFord(),* $\text{dist}(v) \leqslant \text{dist}_{\leqslant i}(v)$.

Proof: induction on i. Suppose $\text{dist}(u) \leqslant \text{dist}_{\leqslant i-1}(u)$ for all $u$ after $(i-1)$-th iteration. We prove $\text{dist}(v) \leqslant \text{dist}_{\leqslant i}(v)$ for all $v$ after $i$-th iteration.

Suppose W is **a shortest walk** from $s$ to v, with at most i edges. Let $u \to v$ be the last edge in W.

During the i-th iteration in BellmanFord(), when we consider edge $u \to v$, we make sure that $\text{dist}(v) \leqslant \text{dist}_{\leqslant i-1}(u) + w(u \to v) = \text{dist}_{\leqslant i}(v)$. $\square$

What does this lemma tell us?

1. If the graph has no negative cycles, the shortest walk from s to any node is a simple path, with at most $V-1$ edges. Therefore Bellman-Ford() must terminate in at most $V-1$ iterations.

2. Conversely, if any edge is still tense after $V-1$ iterations, then the input has negative cycle! We have a simple **negative cycle detector** built-in Bellman-Ford().

Therefore we can re-write the loop:

```
BellmanFord(s)
    InitSSSP(s)
    repeat V − 1 times
        for every edge u→v
            if u→v is tense
                Relax(u→v)
    for every edge u→v
        if u→v is tense
            return "Negative cycle!"
```

The time complexity is $O(VE)$, regardless of the presence of negative edges, or negative cycles.

For non-negative edge graph, Dijkstra's is faster.

Another way to derive the Bellman-Ford algorithm is through dynamic programming. First we have the recurrence: (we've seen this in DAG DFS solution)

$$dist(v) = \begin{cases} 0 & \text{if } v = s \\ \min\limits_{u \to v} (dist(u) + w(u \to v)) & \text{otherwise} \end{cases}$$

However for non-DAG graph, this recursion might not terminate!

To "make" it terminate, we add restriction to the definition of dist():

$dist_{\leqslant i}(v)$ denote the length of the shortest walk from s to v consisting of at most i edges. Then we have recurrence:

$$dist_{\le i}(v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min \left\{ \begin{array}{c} dist_{\le i-1}(v) \\ \min\limits_{u \to v} \left( dist_{\le i-1}(u) + w(u \to v) \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

```
BellmanFordDP(s)
  dist[0, s] ← 0
  for every vertex v ≠ s
      dist[0, v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i, v] ← dist[i − 1, v]
          for every edge u→v
              if dist[i, v] > dist[i − 1, u] + w(u→v)
                  dist[i, v] ← dist[i − 1, u] + w(u→v)
```