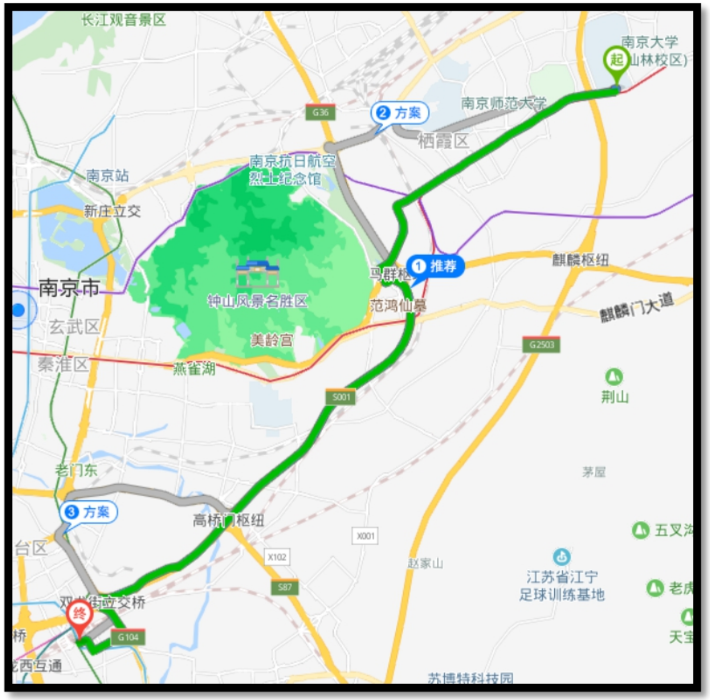
Single-Source Shortest Path

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

The Shortest Path Problem

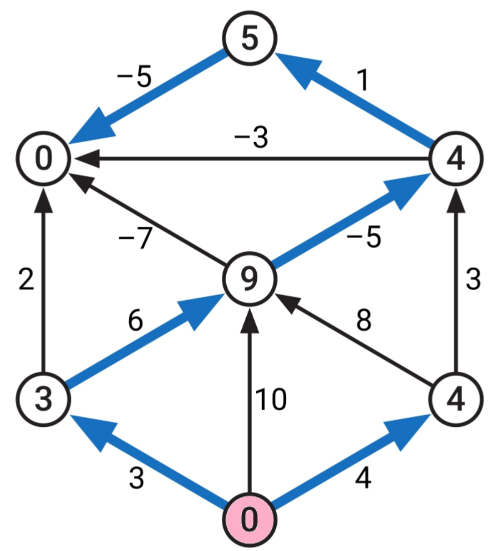
- Given a map, what's the **shortest path** from s to t?
- Consider a graph $G = (V, E)$ and a weight function w that associates a real-valued weight $w(u, v)$ to each edge (u, v). Given s and t in V, what's the **min weight path** from s to t?
- Weights are not always lengths.
    - E.g., time/cost to walk the edge.
- The graph can be directed.
    - Thus $w(u, v) \neq w(v, u)$ possible.
- Negative edge weight allowed.
- Negative cycle *not* allowed.
    - Problem not well-defined then.



Single-Source Shortest Path (SSSP)

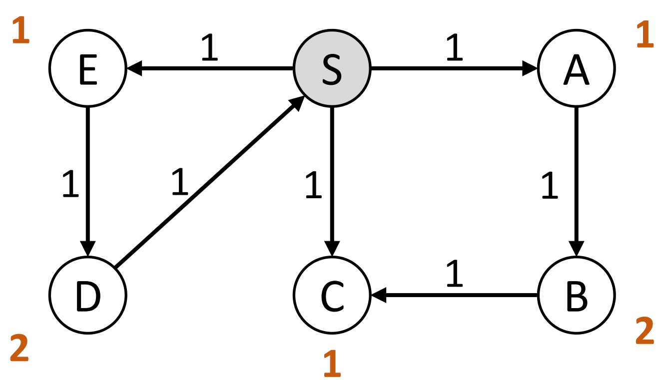
- **The SSSP Problem:** Given a graph $G = (V, E)$ and a weight function w, given a source node s, find a shortest path from s to every node $u \in V$.

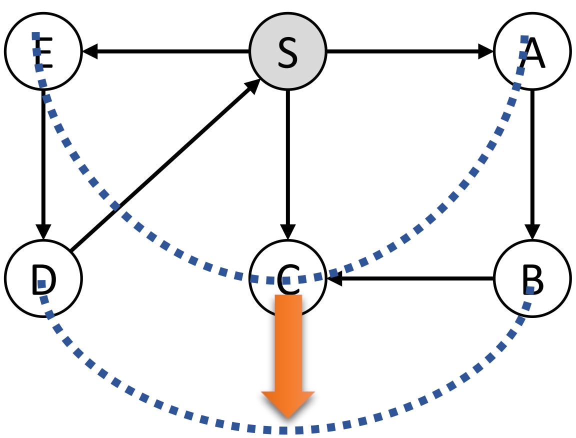
- Consider *directed* graphs *without* negative cycle.
- **Case 1**: Unit weight.
- **Case 2**: Arbitrary positive weight.
- **Case 3**: Arbitrary weight without cycle.
- **Case 4**: Arbitrary weight.



SSSP in unit weight graphs

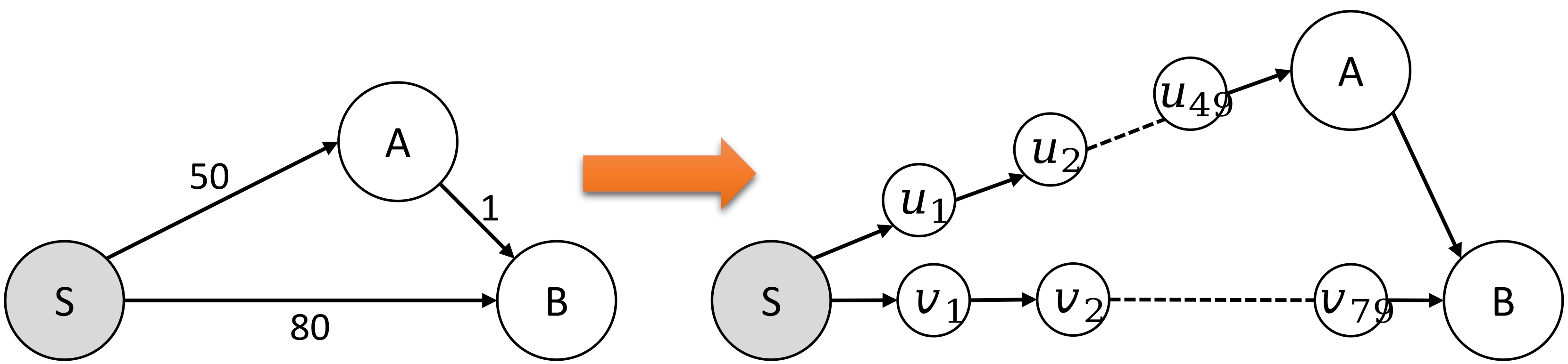
- How to solve SSSP in an unit weight graph?
  - That is, a graph in which each edge is of weight one.
- How to "traverse by layer" in an unweighted graph?
  - Visit all distance d nods before visiting any distance $d + 1$ node.
- **Simple, just use BFS!**





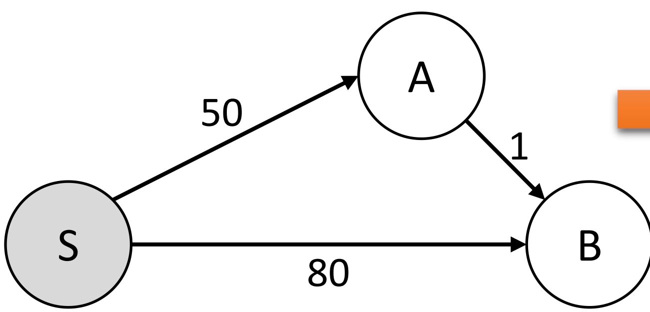
SSSP in positive weight graphs

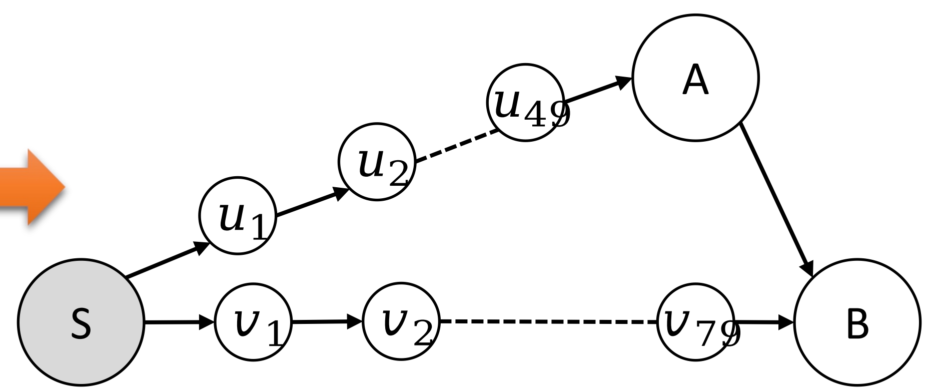
- Solve SSSP in a graph with *arbitrary positive weights*?
- Extension of unit graph SSSP algorithm:
  - Add dummy nodes on edges so graph becomes unit weight graph.
  - Run BFS on the resulting graph.
- Problem with this approach?
- Too slow when edge weights differs a lot!



SSSP in positive weight graphs

- Simple BFS extension for SSSP in positive weight graphs:
  - Add dummy nodes on edges so graph becomes unit weight graph.
  - Run BFS on the resulting graph.
- The algorithm is too slow when edge weights differ a lot!
- To save time, bypass the events that process dummy nodes!
  - Imagine we have an alarm clock T_u for each node u
  - Alarm for source node s goes off at time 0
  - If T_u goes off, for each edge (u, v), update $T_v = \min\{T_v, T_u + w(u, v)\}$

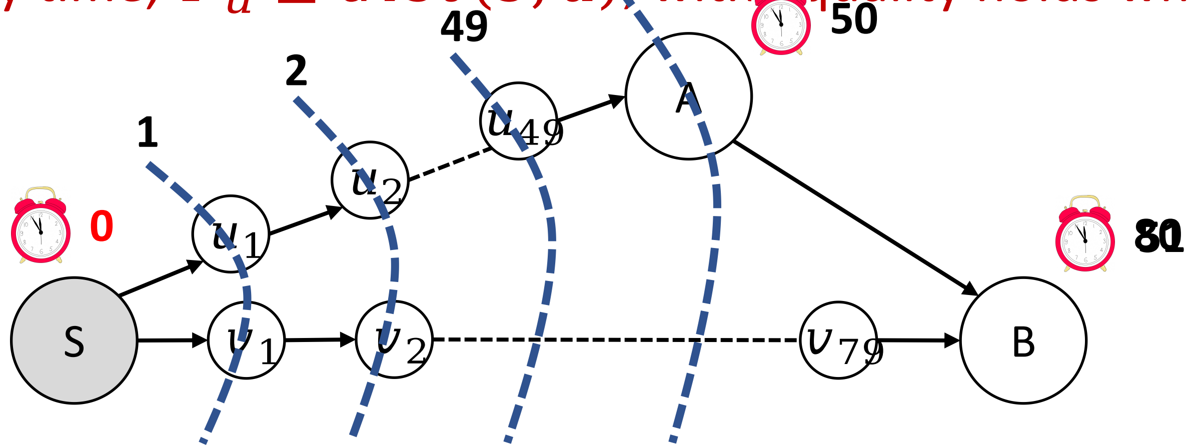




SSSP in positive weight graphs

Extension of the BFS algorithm

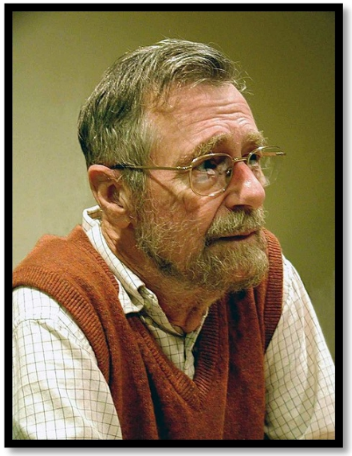
- Extension of BFS for SSSP in positive weight graphs:
  - Imagine we have an alarm clock T_u for each node u
  - Alarm for source node s goes off at time 0
  - If T_u goes off, for each edge (u, v), update $T_v = \min\{T_v, T_u + w(u, v)\}$
- This process is just mimicking the BFS process!
- At any time, value of T_u is an estimate of $dist(s, u)$.
- At any time, $T_u \geq dist(s, u)$, with equality holds when T_u goes off.



SSSP in positive weight graphs via extension of BFS

Dijkstra's algorithm

- Extension of BFS for SSSP in positive weight graphs:
  - Imagine we have an alarm clock T_u for each node u
  - Alarm for source node s goes off at time 0
  - If T_u goes off, for each edge (u, v), update $T_v = \min\{T_v, T_u + w(u, v)\}$
- How to implement the "alarm clock"?
- Use priority queue (such as binary heap).



Edsger W. Dijkstra (1930-2002)
ACM Turing Award Recipient

SSSP in positive weight graphs via extension of BFS

Dijkstra's algorithm

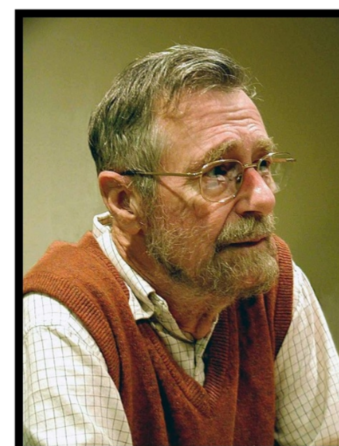
- Extension of BFS for SSSP in positive weight graphs:
  - Imagine we have an alarm clock T_u for each node u
  - Alarm for source node s goes off at time 0
  - If T_u goes off, for each edge (u, v), update $T_v = \min\{T_v, T_u + w(u, v)\}$
- How to implement the "alarm clock"?

p).

DijkstraSSSP(G,s):

Shortest-path Tree
(Similar to BFS tree.)

```
for (each u in V)
  u.dist=INF, u.parent=NIL
s.dist = 0
Build priority queue Q based on dist
while (!Q.empty())
  u = Q.ExtractMin()
  for (each edge (u,v) in E)
    if (v.dist > u.dist + w(u,v))
      v.dist = u.dist + w(u,v)
      v.parent = u
      Q.DecreaseKey(v)
```



Edsger W. Dijkstra (1930-2002)
ACM Turing Award Recipient

SSSP in positive weight graphs via extension of BFS

Dijkstra's algorithm

- Correctness of Dijkstra's algorithm?
- Similar to the correctness proof of BFS.

- Efficiency of Dijkstra's algorithm?
- $O((n+m)\log n)$ when using a binary heap.

DijkstraSSSP(G,s):

```
for (each u in V)
  u.dist=INF, u.parent=NIL
s.dist = 0
Build priority queue Q based on dist
while (!Q.empty())
  u = Q.ExtractMin()
  for (each edge (u,v) in E)
    if (v.dist > u.dist + w(u,v))
      v.dist = u.dist + w(u,v)
      v.parent = u
      Q.DecreaseKey(v)
```

$O(n)$ in total

$O(n)$ in total

$O(n\log n)$ in total

$O(m\log n)$ in total

Alternative derivation of Dijkstra’s alg.

- What’s BFS doing: *expand* outward from s, growing the *region* to which distances and shortest paths are known.
- Growth should be *orderly*: **closest nodes first**.
- **Q:** But how to identify the node to expend to?

- Consider a *shortest path* from source s to v via u.

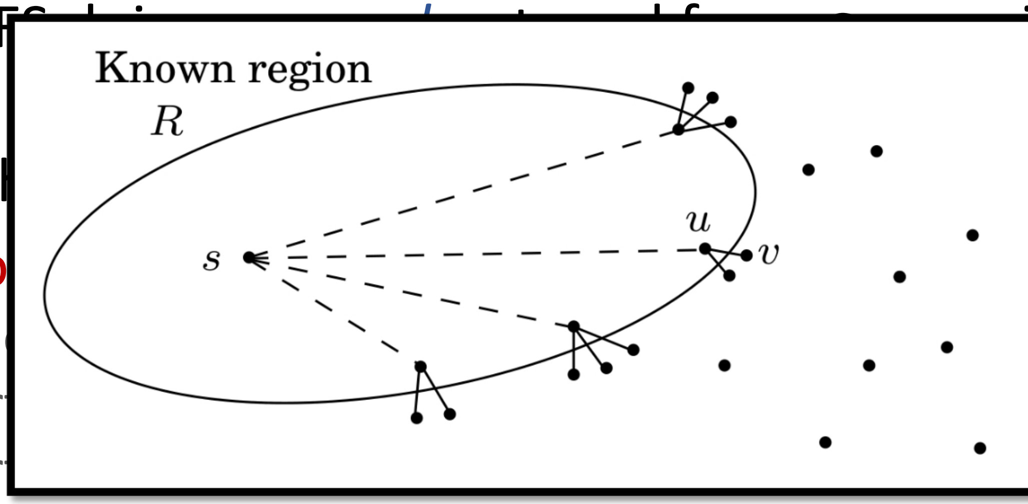
- It must be $dist(s, v) = dist(s, u) + w(u, v)$.
  - Thus shortest path exhibits **optimal substructure** property.
- It must be $dist(s, v) > dist(s, u)$.
  - Since we are considering positive edge weight graphs.

Alternative derivation of Dijkstra's alg.

- What's BFS doing: *expand* outward from s, growing the *region* to which distances and shortest paths are known.
- Growth should be *orderly*: **closest nodes first**.
- **Q:** But how to identify the node to expend to?
  - Consider a shortest path from source s to v via u.
  - Property 1: It must be $dist(s, v) = dist(s, u) + w(u, v)$.
  - Property 2: It must be $dist(s, v) > dist(s, u)$.
- **A:** Given "known region R",
  find $\min_{u' \in R, v' \in V-R} \{dist(s, u') + w(u', v')\}$.
  - Assume v is the node to expend to. (A shortest path is $s \to u \to v$.)
  - Property 2 ensures $u \in R$.
  - Property 1 then ensures we correctly identify v to expend to.

Alternative derivation of Dijkstra's alg.

- What's BF ing the *region* to known.
- Growth s
- **Q:** But ho
  - Consid
  - Proper (u, v).
  - Proper

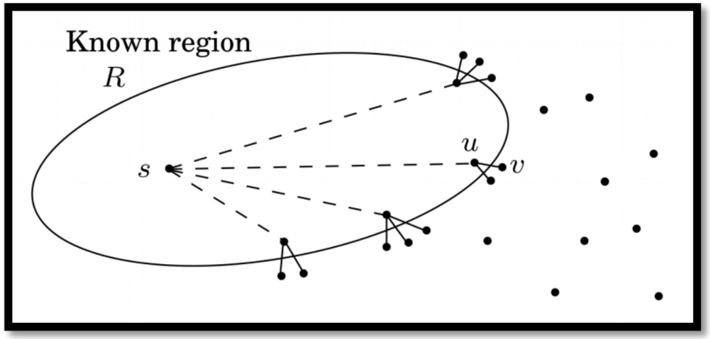


- **A:** Given "known region R", find $\min_{u' \in R, v' \in V-R} \{dist(s, u') + w(u', v')\}$.
  - Assume v is the node to expend to. (A shortest path is $s \to u \to v$.)
  - Property 2 ensures $u \in R$.
  - Property 1 then ensures we correctly identify v to expend to.

- What's BFS doing: *expand* outward from s, growing the *region* to which distances and shortest paths are known.
- How to expend: Given "known region R", expend to node with $\min_{u' \in R, v' \in V-R} \{dist(s,u') + w(u',v')\}$.

DijkstraSSSP(G,s):

```
for (each u in V)
  u.dist=INF, u.parent=NIL
s.dist = 0
Build priority queue Q based on dist
while (!Q.empty())
  u = Q.ExtractMin()
  for (each edge (u,v) in E)
    if (v.dist > u.dist + w(u,v))
      v.dist = u.dist + w(u,v)
      v.parent = u
      Q.DecreaseKey(v)
```



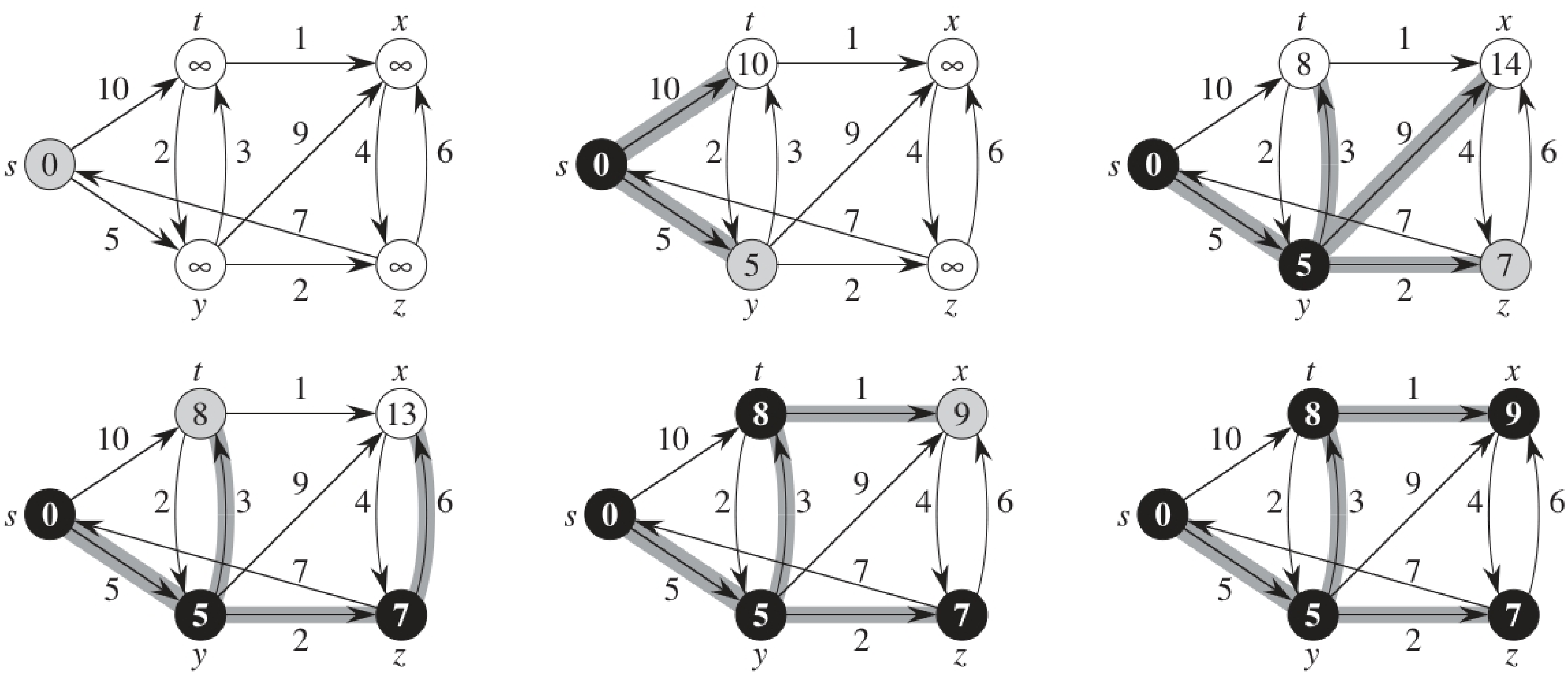
Priority queue implementation

DijkstraSSSPAbs(G,s):

```
for (each u in V)
  u.dist = INF
s.dist = 0
R = ∅
while (R != V)
  Find node v in V-R with min v.dist
  Add v to R
  for (each edge (v,z) in E)
    if (z.dist > v.dist + w(v,z))
      z.dist = v.dist + w(v,z)
```

DijkstraSSSP(G,s):

```
for (each u in V)
  u.dist=INF, u.parent=NIL
s.dist = 0
Build priority queue Q based on dist
while (!Q.empty())
  u = Q.ExtractMin()
  for (each edge (u,v) in E)
    if (v.dist > u.dist + w(u,v))
      v.dist = u.dist + w(u,v)
      v.parent = u
      Q.DecreaseKey(v)
```



DFS, BFS, Prim, Dijkstra, and others...

DFSIterSkeleton(G,s):

```
Stack Q
Q.push(s)
while (!Q.empty())
  u = Q.pop()
  if (!u.visited)
    u.visited = true
    for (each edge (u,v) in E)
      Q.push(v)
```

BFSSkeletonAlt(G,s):

```
FIFOQueue Q
Q.enque(s)
while (!Q.empty())
  u = Q.dequeue()
  if (!u.visited)
    u.visited = true
    for (each edge (u,v) in E)
      Q.enque(v)
```

PrimMSTSkeleton(G,x):

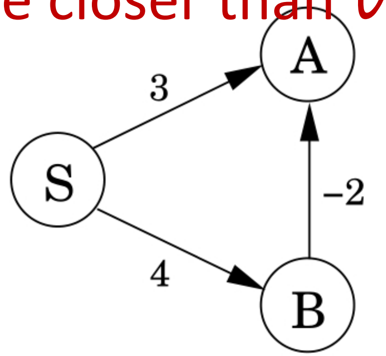
```
PriorityQueue Q
Q.add(x)
while (!Q.empty())
  u = Q.remove()
  if (!u.visited)
    u.visited = true
    for (each edge (u,v) in E)
      if (!v.visited and ...)
        Q.update(v,...)
```

DijkstraSSSPSkeleton(G,x):

```
PriorityQueue Q
Q.add(x)
while (!Q.empty())
  u = Q.remove()
  if (!u.visited)
    u.visited = true
    for (each edge (u,v) in E)
      if (!v.visited and ...)
        Q.update(v,...)
```

SSSP in graphs with negative weights

- **Dijkstra's algorithm no longer works!**
- **Why would this happen?**
- Dijkstra's algorithm for finding next closest node to expend to:
  Given "known region R", find $\min_{u' \in R, v' \in V-R} \{dist(s,u') + w(u',v')\}$.
  - Assume v is the node to expend to. (A shortest path is $s \to u \to v$.)
  - Positive edge weights ensures $u \in R$.
  - Optimal substructure then ensures we correctly identify v to expend to.
- "Shortest path from s to any node v must pass exclusively through nodes that are closer than v" no longer holds!



SSSP in graphs with negative weights

- But how $dist$ values are maintained in Dijkstra is helpful:
  - Each node $u \neq s$ initially set $u.dist = \infty$, and $s.dist = 0$
  - When processing edge (u, v), execute procedure $\mathrm{Update}(u, v)$: $v.dist = \min\{v.dist, u.dist + w(u, v)\}$
- This way two properties are maintained:
  - For any v, at any time, $v.dist$ is either an overestimate, or correct.
  - Assume u is the last node on a shortest path from s to v. If $u.dist$ is correct and we run $\mathrm{Update}(u, v)$, then $v.dist$ becomes correct.
- $\mathrm{Update}(u, v)$ is **safe** and helpful!
  - **[Safe]** Regardless of the sequence of `Update` operations we execute, for any node v, value $v.dist$ is either an overestimate or correct.
  - **[Helpful]** With correct sequence of `Update`, we get correct $v.dist$.

SSSP in graphs with negative weights

- $\text{Update}(u, v)$: $v.dist = \min\{v.dist, u.dist + w(u, v)\}$
- $\text{Update}(u, v)$ is **safe** and helpful!
  - **[Safe]** Regardless of the sequence of `Update` operations we execute, for any node v, value $v.dist$ is either overestimate or correct.
  - **[Helpful]** Assume u is the last node on a shortest path from s to v. If $u.dist$ is correct and we run $\text{Update}(u, v)$, then $v.dist$ becomes correct.
- Consider a shortest path from s to v.

  $\text{Update}(s, u_1)$ $\text{Update}(\ldots)$ $\text{Update}(u_1, u_2)$ $\text{Update}(\ldots)$ $\text{Update}(\ldots)$ $\text{Update}(u_k, v)$ $\text{Update}(\ldots)$

- **Observation 1**: if $\text{Update}(s, u_1)$, $\text{Update}(u_1, u_2)$, …, $\text{Update}(u_{k-1}, u_k)$, $\text{Update}(u_k, v)$ are executed, then we correctly obtain the shortest path.
- **Observation 2**: in above sequence, before and after each `Update`, we can add arbitrary `Update` sequence, and still get shortest path from s to t.
- **Algorithm:** simply `Update` *all* edges, for $k + 1$ times!

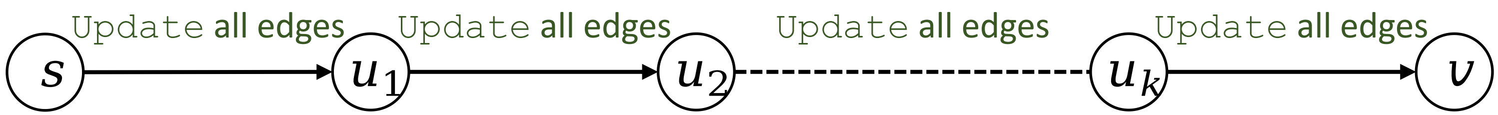
SSSP in graphs with negative weights

- $\text{Update}(u, v)$: $v.dist = \min\{v.dist, u.dist + w(u,v)\}$
- $\text{Update}(u, v)$ is **safe** and helpful!
  - **[Safe]** Regardless of the sequence of `Update` operations we execute, for any node v, value $v.dist$ is either overestimate or correct.
  - **[Helpful]** Assume u is the last node on a shortest path from s to v. If $u.dist$ is correct and we run $\text{Update}(u, v)$, then $v.dist$ becomes correct.
- Consider a shortest path from s to v.

- **Observation 1**: if $\text{Update}(s, u_1)$, $\text{Update}(u_1, u_2)$, …, $\text{Update}(u_{k-1}, u_k)$, $\text{Update}(u_k, v)$ are executed, then we correctly obtain the shortest path.
- **Observation 2**: in above sequence, before and after each `Update`, we can add arbitrary `Update` sequence, and still get shortest path from s to t.
- **Algorithm:** simply `Update` <u>*all*</u> edges, for $k+1$ times!

SSSP in graphs with negative weights

- Consider a shortest path from s to v.



- **Observation 1**: if $\mathrm{Update}(s, u_1)$, $\mathrm{Update}(u_1, u_2)$, …, $\mathrm{Update}(u_{k-1}, u_k)$, $\mathrm{Update}(u_k, v)$ are executed, then we correctly obtain the shortest path.
- **Observation 2**: in above sequence, before and after each `Update`, we can add arbitrary `Update` sequence, and still get shortest path from s to t.
- **Algorithm:** simply `Update` *all* edges, for $k + 1$ times!
- But how large is $k + 1$?
- **Observation 3**: any shortest path cannot contain a cycle. (WHY?)
- **Algorithm:** simply `Update` *all* edges, for $n - 1$ times!

SSSP in directed graphs with negative weights

The Bellman-Ford Algorithm

- **Bellman-Ford Algorithm:**
  - `Update` all edges;
  - Repeat above step for $n-1$ times.
- Time complexity of Bellman-Ford: $\Theta(nm)$

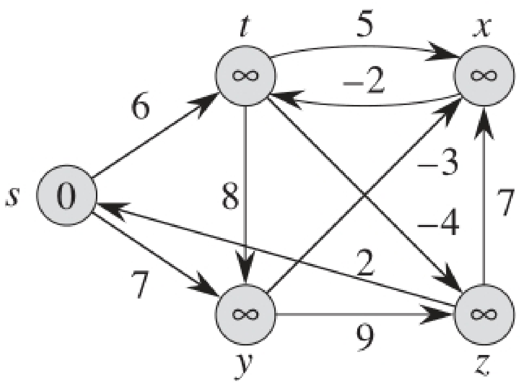
BellmanFordSSSP(G,s):

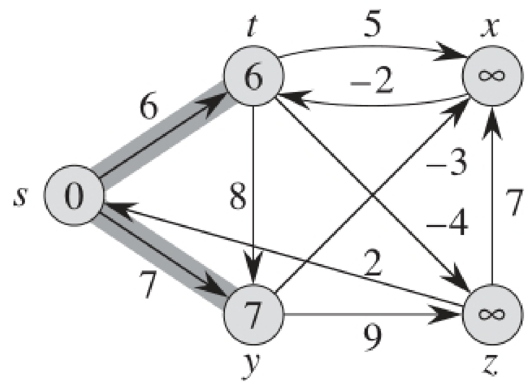
```
for (each u in V)
  u.dist=INF, u.parent=NIL
s.dist = 0
repeat n-1 times:
  for (each edge (u,v) in E)
    if (v.dist > u.dist + w(u,v))
      v.dist = u.dist + w(u,v)
      v.parent = u
```

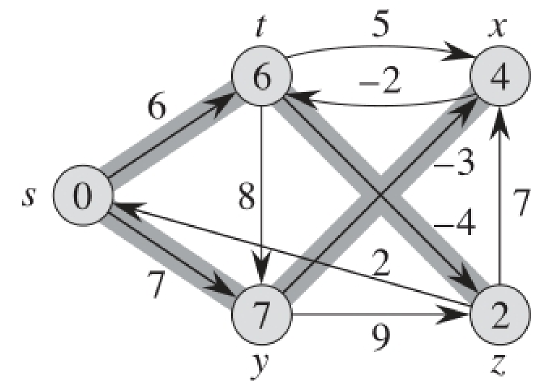
BellmanFordSSSP(G,s):

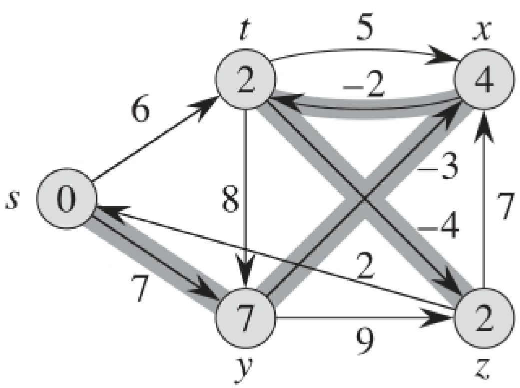
```
for (each u in V)
  u.dist=INF, u.parent=NIL
s.dist = 0
repeat n-1 times:
  for (each edge (u,v) in E)
    if (v.dist > u.dist + w(u,v))
      v.dist = u.dist + w(u,v)
      v.parent = u
```

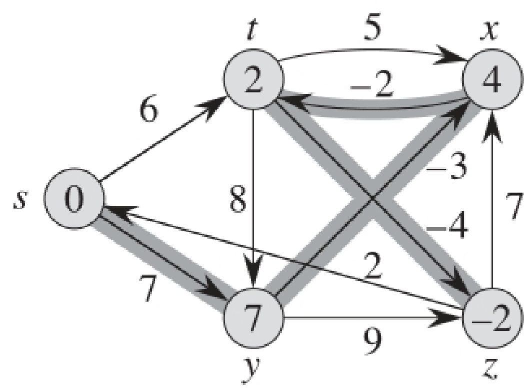
Edge order: $(t, x), (t, y), (t, z),\ (x, t), (y, x), (y, z), (z, x),\ (z, s), (s,$











SSSP in directed graphs with negative weights

The Bellman-Ford Algorithm

- What if the graph contains a negative cycle?
- After $n-1$ repetitions of “Update all edges”, some node v still has $v.dist > u.dist + w(u,v)$.
- Bellman-Ford can also detect negative cycle!

BellmanFordSSSP(G,s):

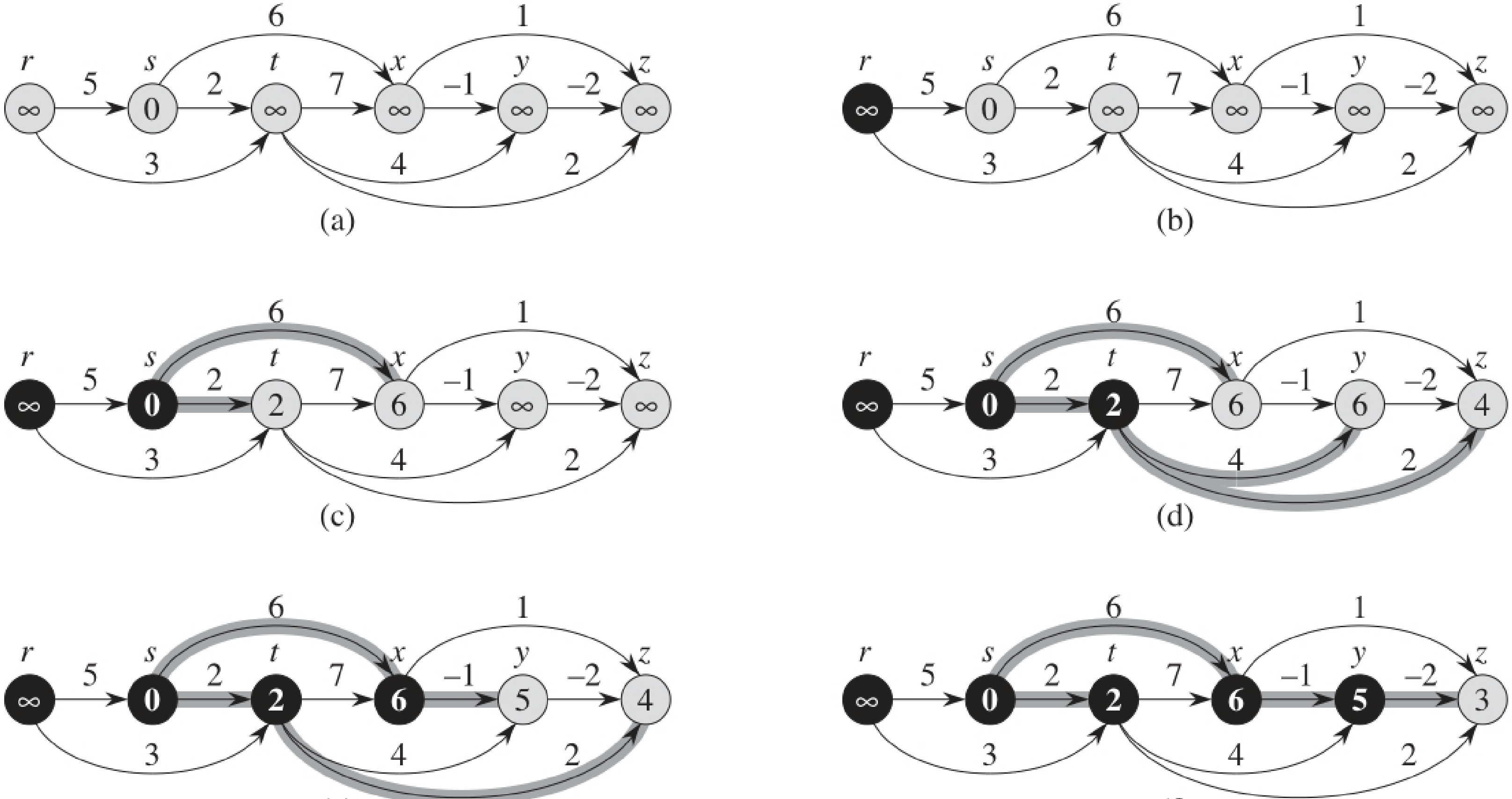
```
for (each u in V)
  u.dist=INF, u.parent=NIL
s.dist = 0
repeat n-1 times:
  for (each edge (u,v) in E)
    if (v.dist > u.dist + w(u,v))
      v.dist = u.dist + w(u,v)
      v.parent = u
for (each edge (u,v) in E)
  if (v.dist > u.dist + w(u,v))
    return "Negative Cycle"
```

SSSP in DAG (with negative weights)

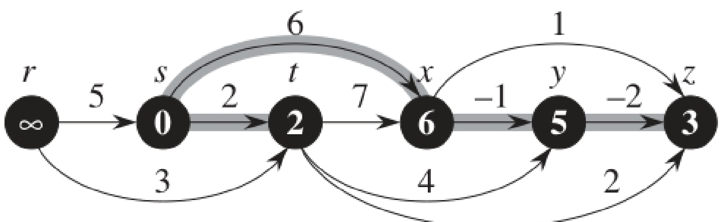
- Bellman-Ford still works, but we can be more efficient!
- **Core idea of Bellman-Ford:** perform a sequence of `Update` that includes every shortest path as a subsequence.
- **Observation**: in DAG, every path, thus every shortest path, is a subsequence in the topological order.

DAGSSSP(G,s): $O(n+m)$ time

```
for (each u in V)
  u.dist=INF, u.parent=NIL
s.dist = 0
Run DFS to obtain topological order
for (each node u in topological order)
  for (each edge (u,v) in E)
    if (v.dist > u.dist + w(u,v))
      v.dist = u.dist + w(u,v)
      v.parent = u
```



(a) (b) (c) (d) (e) (f)



(g)

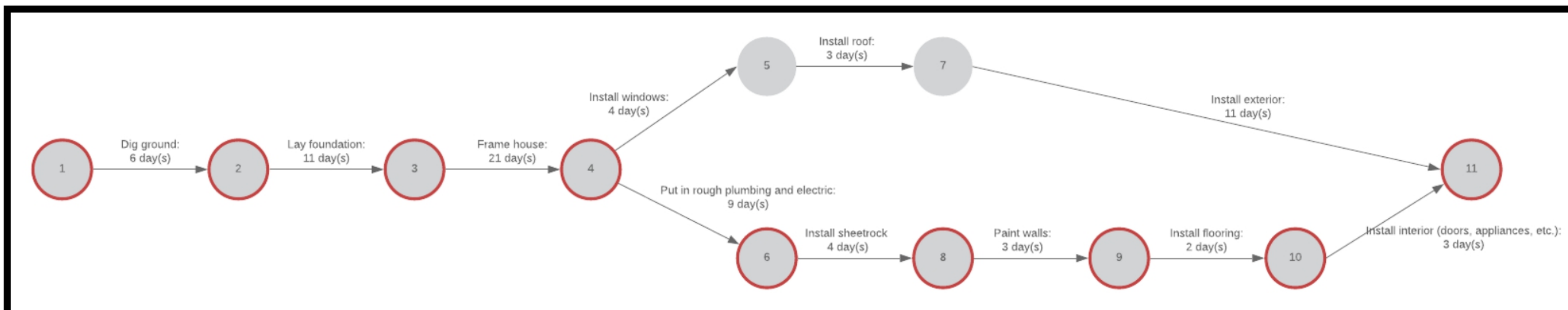
DAGSSSP(G,s):

```
for (each u in V)
  u.dist=INF, u.parent=NIL
s.dist = 0
Run DFS to obtain topological order
for (each node u in topological order)
  for (each edge (u,v) in E)
    if (v.dist > u.dist + w(u,v))
      v.dist = u.dist + w(u,v)
      v.parent = u
```

Application of SSSP in DAG

Computing Critical Path

- Assume you want to finish a task that involves multiple steps.
  Each step takes some time.
  For some step(s), it can only begin after certain steps are done.
- These dependency can be modeled as a DAG. (**PERT Chart**)
- How fast can you finish this task?
- Equivalently, *longest path*, a.k.a. **critical path**, in the DAG?
- Negate edge weights and compute a shortest path.



Summary

- **The SSSP Problem:** Given a graph $G = (V, E)$ and a weight function w, given a source node s, find a shortest path from s to every node $v \in V$.
- **Case 1**: Unit weight graphs (directed or undirected).
  - Simply use BFS. $O(n + m)$ runtime.
- **Case 2**: Arbitrary positive weight graphs (directed or undirected).
  - Dijkstra's algorithm. A greedy algorithm. $O((n + m)\log n)$ runtime.
- **Case 3**: Arbitrary weight without cycle in directed graphs.
  - `Update` in topological order. $O(n + m)$ runtime.
- **Case 4**: Arbitrary weight without negative cycle in directed graphs.
  - Bellman-Ford algorithm. $\Theta(nm)$ runtime, can detect negative cycle.
- The shortest path problem has *optimal substructure* property.
- `Update` is a *safe* and *helpful* operation.

Reading

- [DPV] Ch.4 (More intuitive presentation.)
- [CLRS] Ch.24 (excluding 24.4) (Formal and rigorous.)
- Optional reading: [Erickson v1] Ch.8