智能软件与工程学院
School of Intelligent Software and Engineering

单源最短路径
Single-Source Shortest Path

钮鑫涛
Nanjing University
2023 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!

The Shortest Path Problem

- Given a map, what's the **shortest path** from s to t?

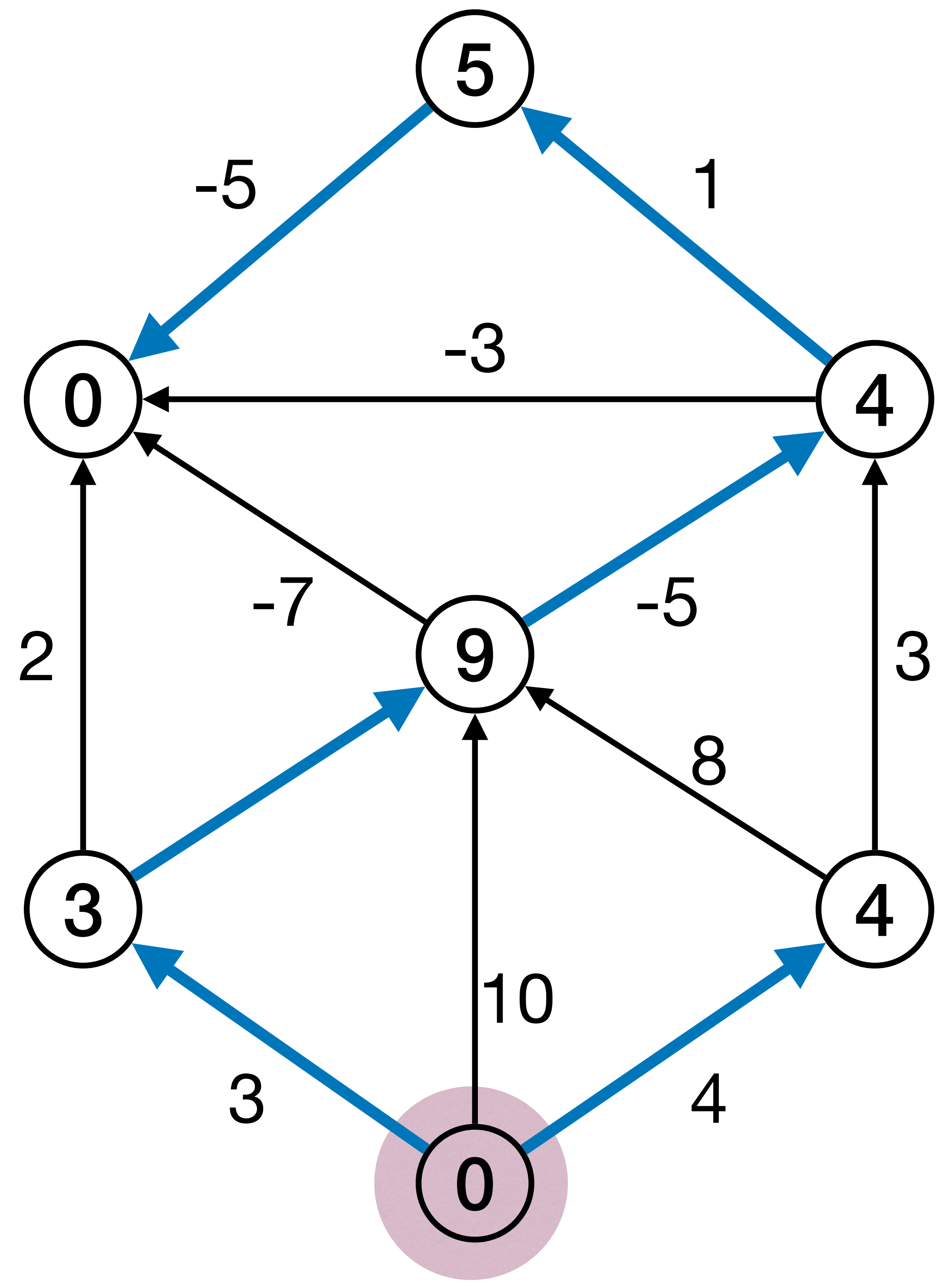
- Consider a graph $G = (V, E)$ and a weight function $w$that associates a real-valued weight $w(u, v)$ to each edge (u, v). Given s and t in V, what's the **min weight path** from s to t?

The Shortest Path Problem

- **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.

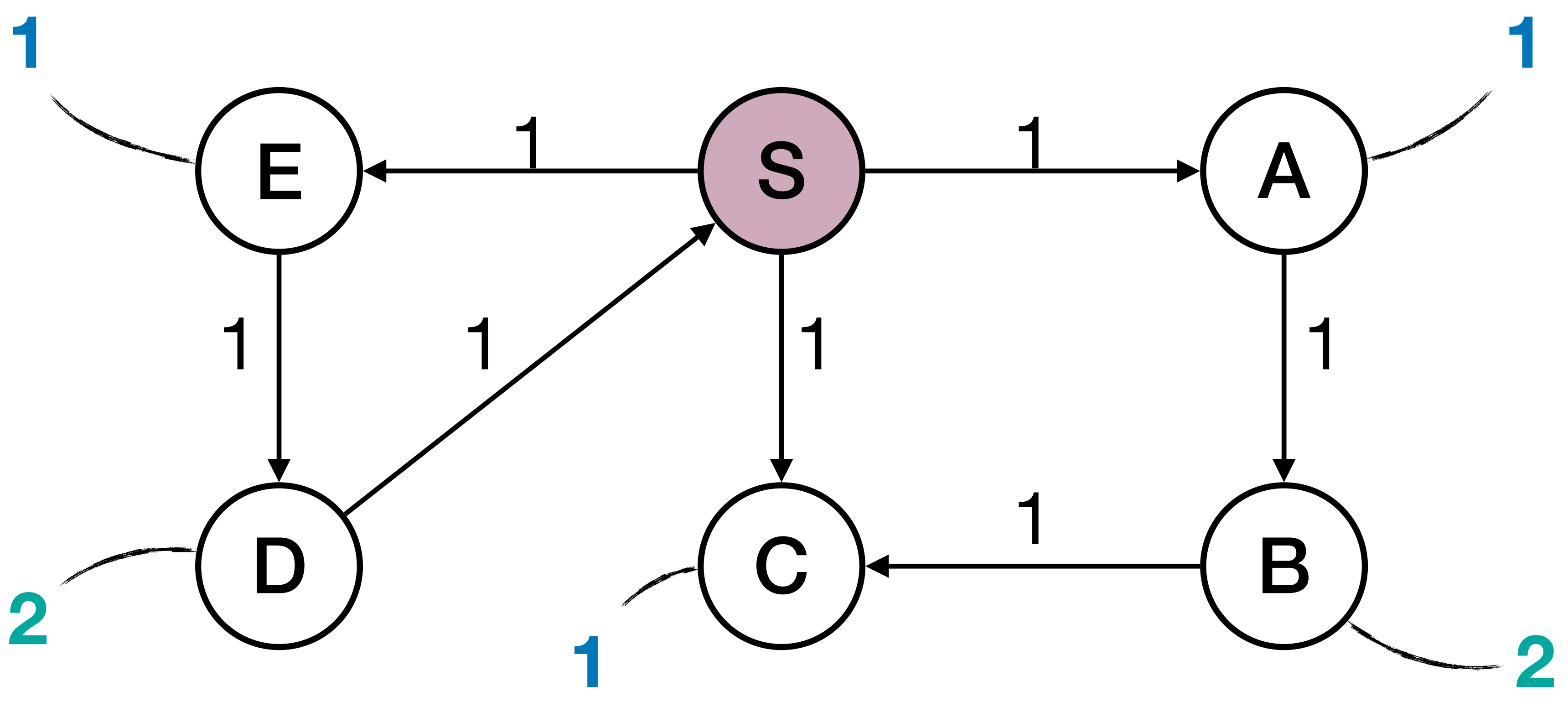
Single-Source Shortest Path (SSSP)

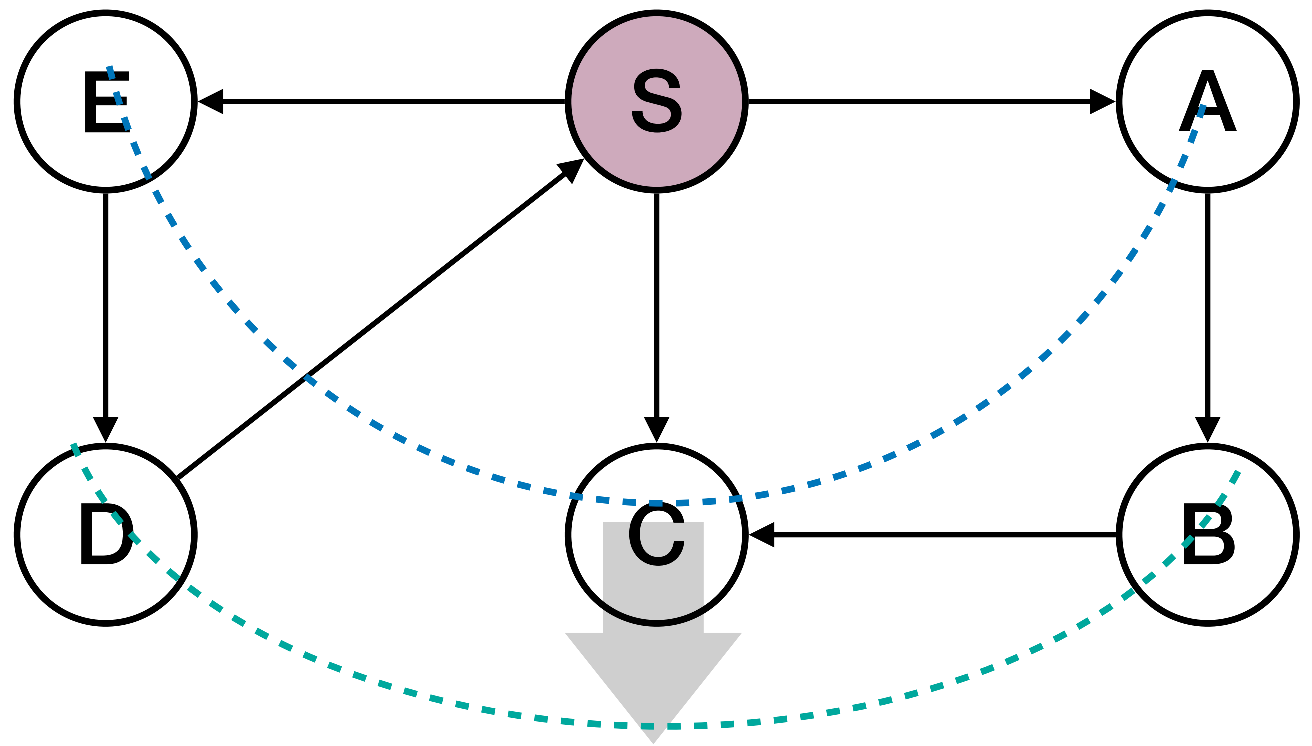
- **The SSSP Problem:** Given a graph $G = (V, E)$ and a weight function w, given a source node s, find a shortest path from s to every node $u \in V$.
- Consider *directed* graphs *without* negative cycle.
  - **Case 1**: Unit weight.
  - **Case 2**: Arbitrary positive weight.
  - **Case 3**: Arbitrary 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 **1**.
- “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

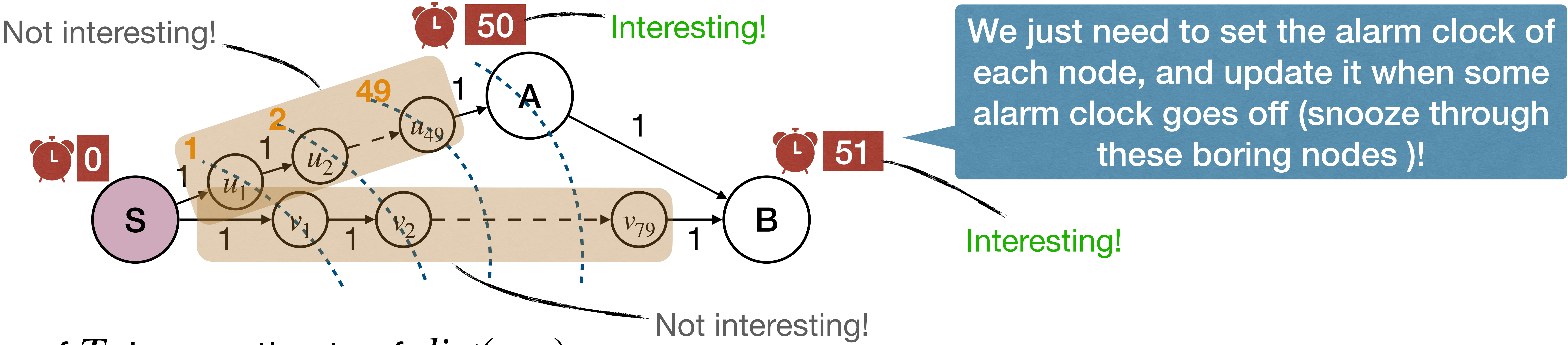
- Solve SSSP in a graph with *arbitrary positive weights*?
- Extension of unit graph SSSP algorithm:
  - Add dummy nodes on edges so graph becomes unit weight graph.
  - Run BFS on the resulting graph.

The problem is that it is too slow when edge weights are large!

智能软件与工程学院
School of Intelligent Software and Engineering

Extension of the BFS algorithm

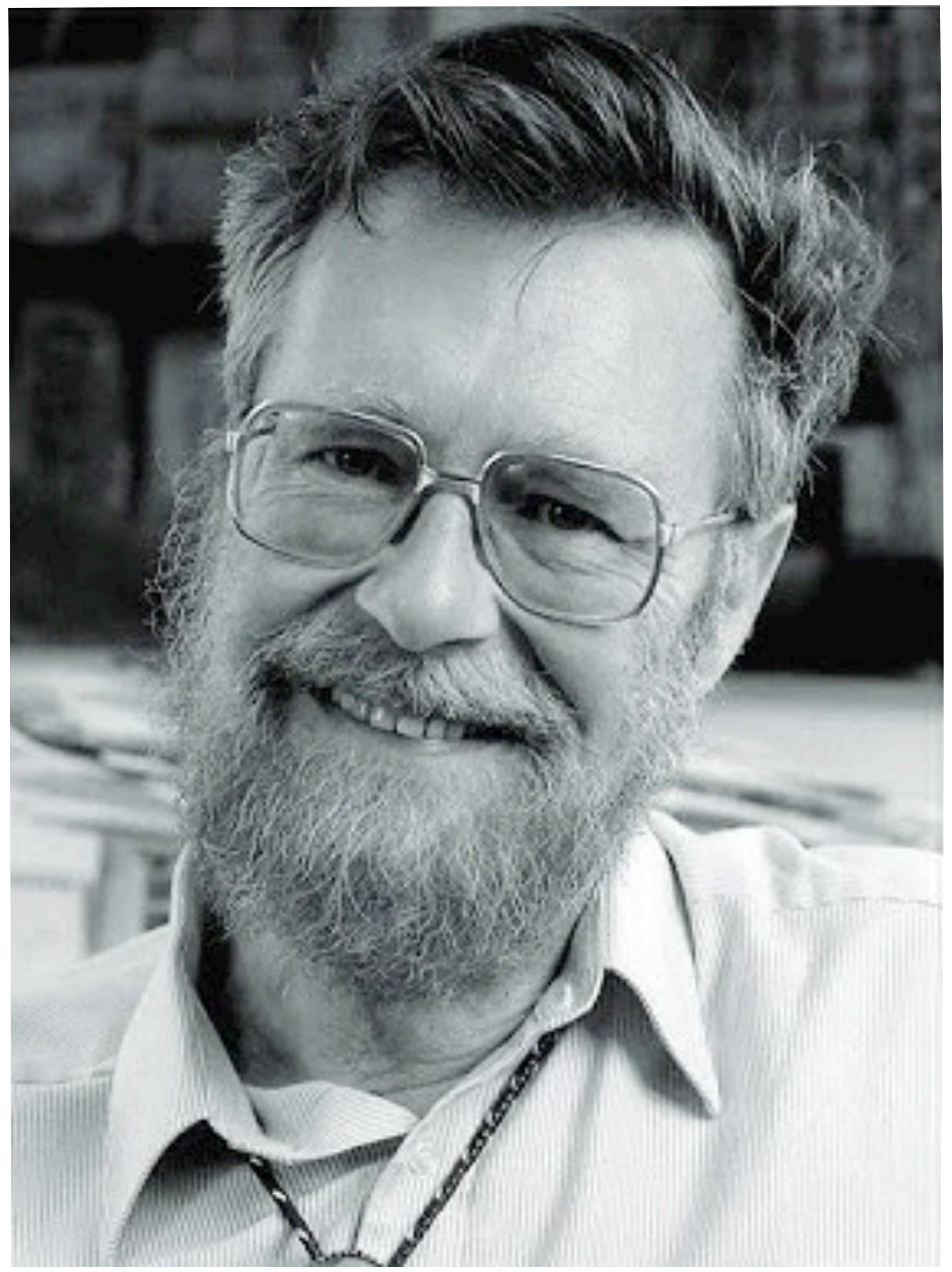
- To save time, bypass the events that process dummy nodes!
  - Imagine we have an alarm clock T_u for each node u.
  - Alarm for source node s goes off at time 0.
  - If T_u goes off, for each edge (u, v), update $T_v = \min\{T_v, T_u + w(u, v)\}$



- 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.

Dijkstra's algorithm

- How to implement the “alarm clock”?
  - Use priority queue (such as binary heap).



Edsger W. Dijkstra

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 $v.dist > u.dist + w(u, v)$
            $v.dist := u.dist + w(u, v)$
            $v.parent := u$
            $Q.UpdateKey(v)$

Dijkstra's algorithm

- Correctness of Dijkstra's algorithm?
  - Similar to the correctness proof of BFS.
- Efficiency of Dijkstra's algorithm?
  - $O((n+m)\cdot\log n)$ when using a binary heap.

DijkstraSSSP(G, s):

for **each** u **in** V — $O(n)$

    $u.dist := INF, u.parent := NIL$

$s.dist := 0$

Build priority queue Q based on dist — $O(n)$

while $!Q.empty()$

    $u := Q.ExtractMin()$ — $O(n\log n)$

    for **each** edge (u,v) **in** E — $O(m\log n)$

        if $v.dist > u.dist + w(u,v)$

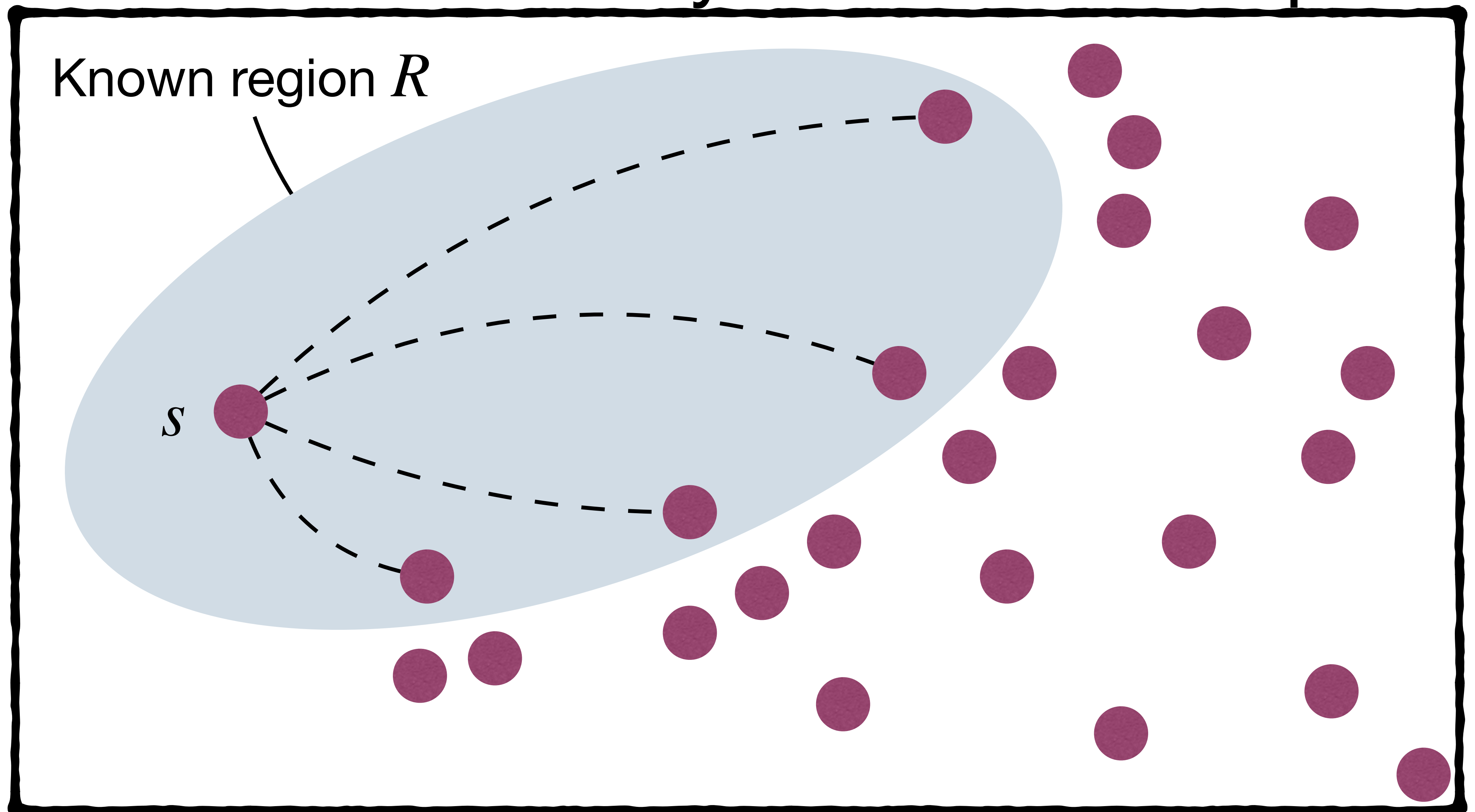
            $v.dist := u.dist + w(u,v)$

            $v.parent := u$

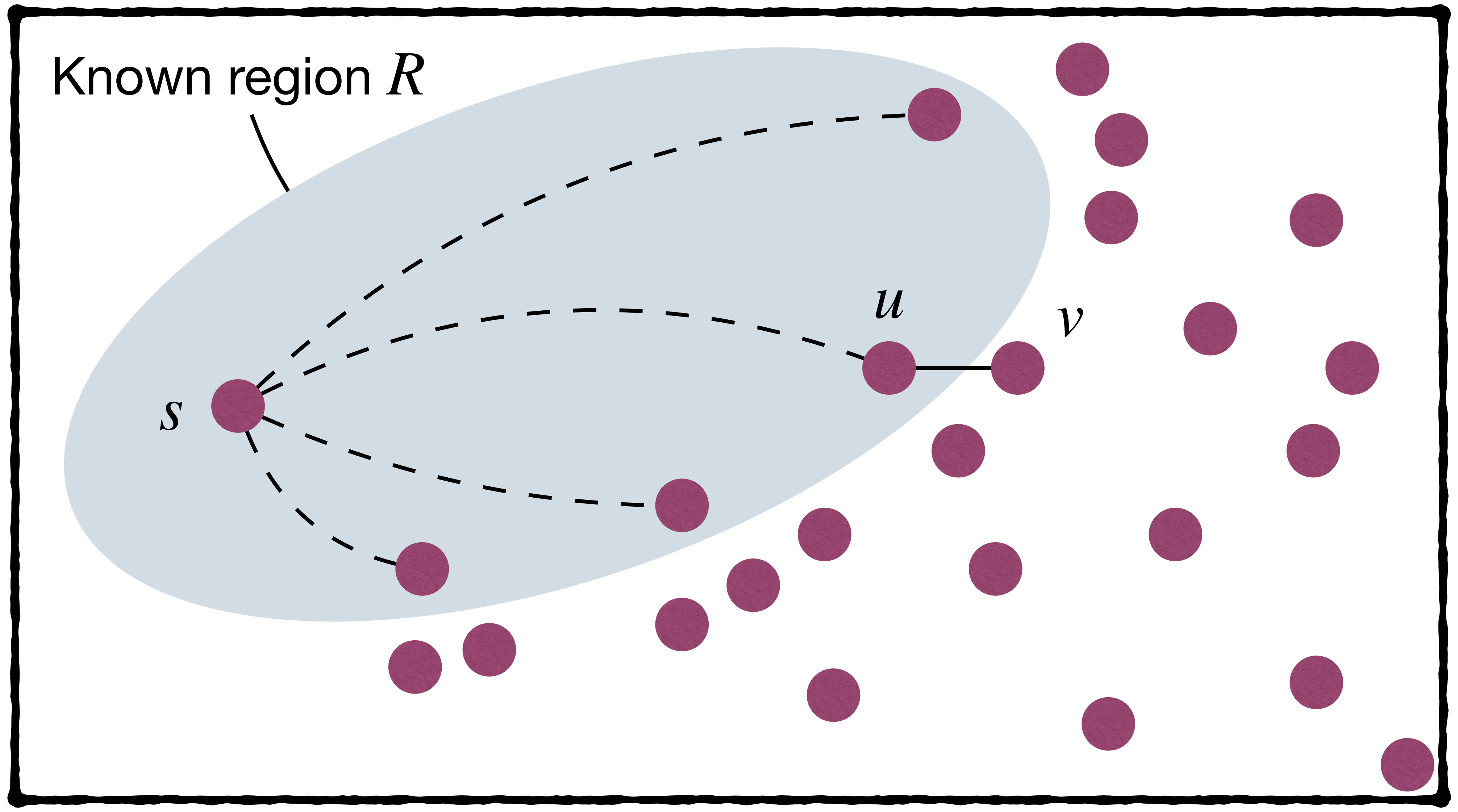
            $Q.UpdateKey(v)$

Alternative derivation of Dijkstra's algorithm

- 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.
- Given "known region R",
  - how to identify the node to expand to?

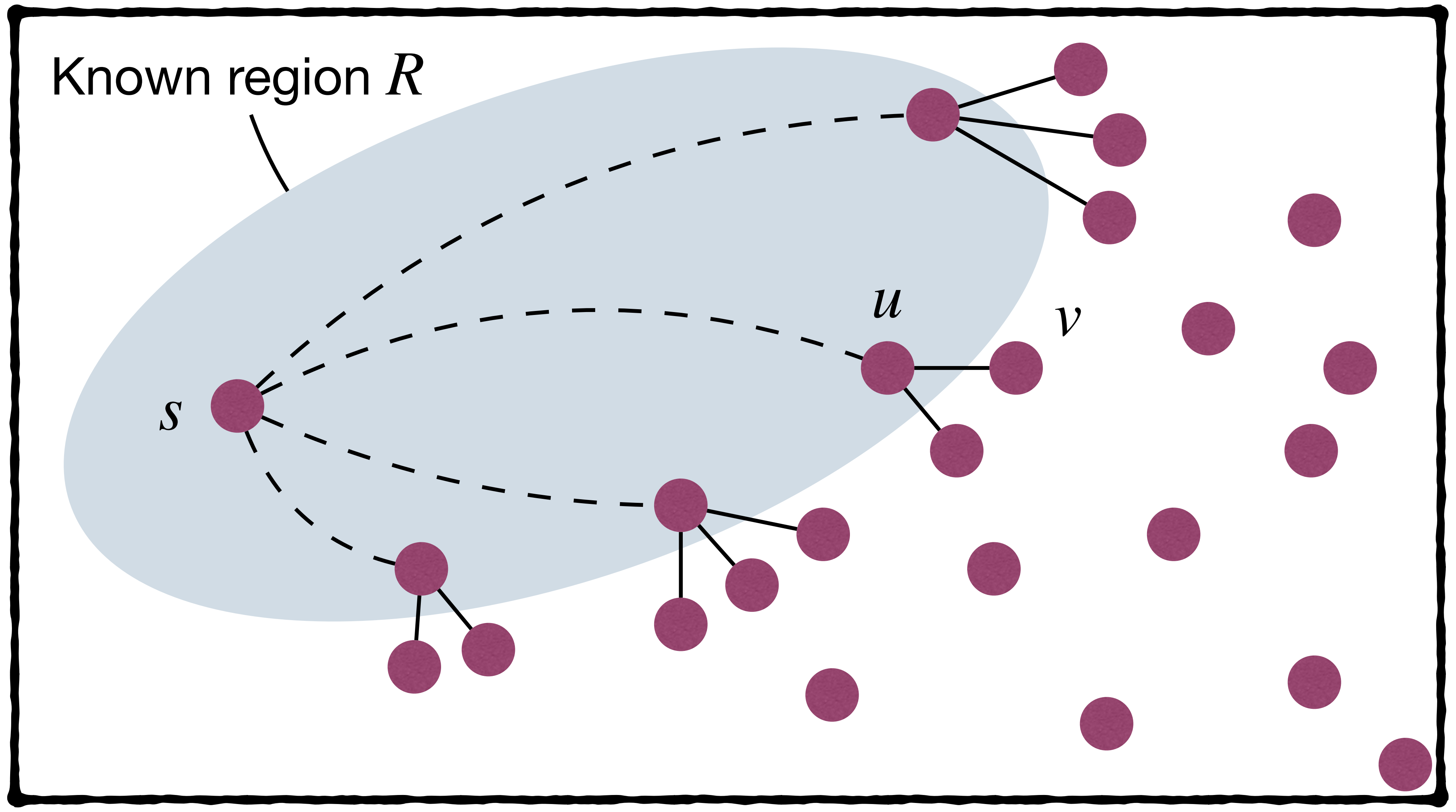


Alternative derivation of Dijkstra's algorithm



- Given "known region R", assume v is such node to expand to (that is, the next closet node to s), let the shortest path from s to v is $s \rightsquigarrow v$.
  - It must be $dist(s, v) \geq dist(s, v')$, for any $v' \in R$. (Otherwise it is already $v \in R$)
  - Let the last node of the path $s \rightsquigarrow v$ before v be u, then it must be $u \in R$. (Otherwise v is not the next closet node to s)

Alternative derivation of Dijkstra's algorithm



- Given "known region R",
  - Find $\min_{u' \in R, v' \in V-R} \{dist(s, u') + w(u', v')\}$,
  - Any satisfied node v is the next node to expand to (the next closet node to s)

Alternative derivation of Dijkstra's algorithm

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

DijkstraSSSPAbs(G, s):

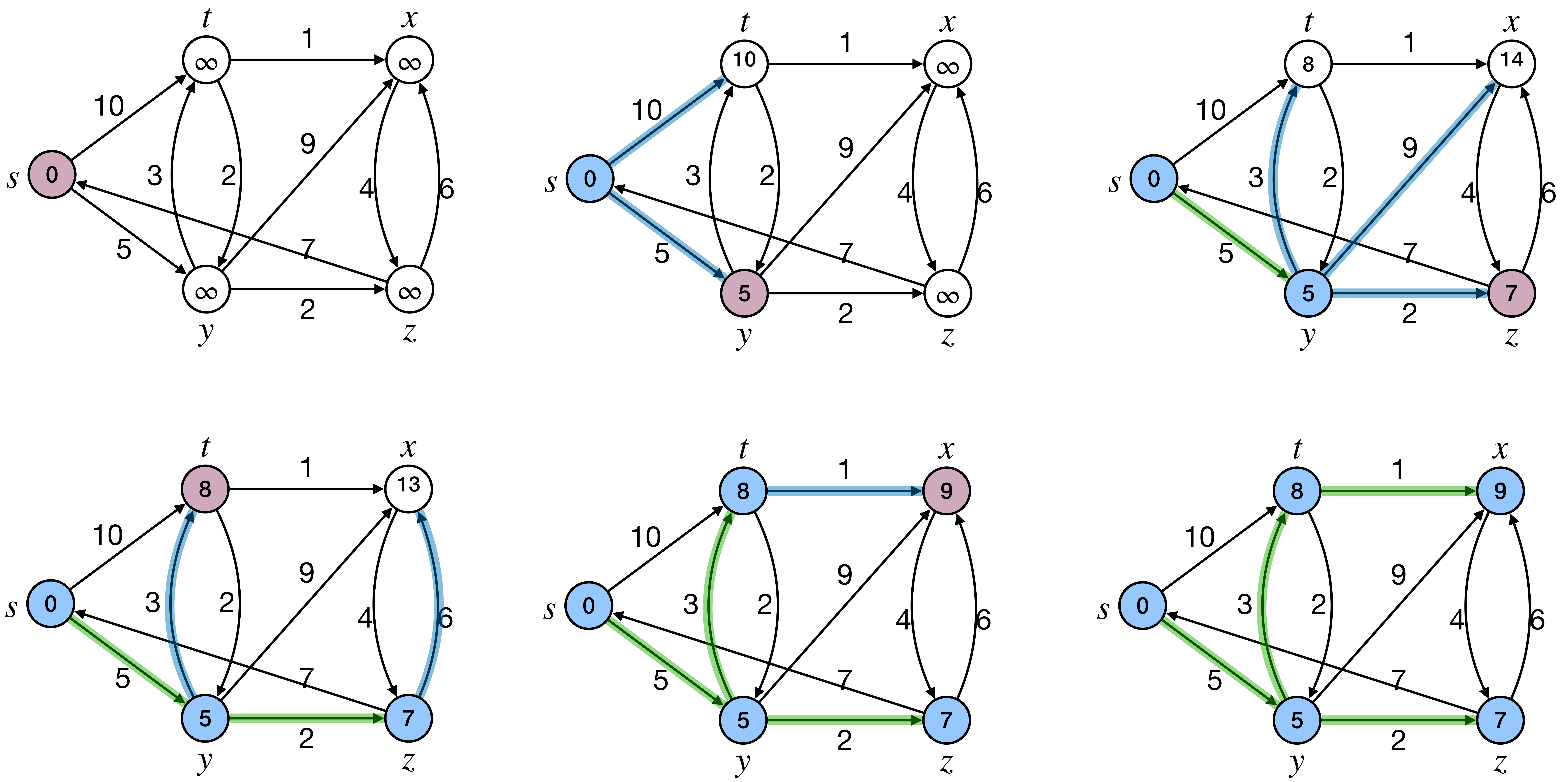
for each u **in** V
 $u.dist := INF$
$s.dist := 0$
$R := \emptyset$
while $R \,!= V$
 Find node v in V - R with min v.dist
 Add v to R
 for each edge (v, z) in E
  if $z.dist > v.dist + w(v, z)$
   $z.dist := v.dist + w(v, z)$

Priority queue implementation

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 $v.dist > u.dist + w(u, v)$
   $v.dist := u.dist + w(u, v)$
   $v.parent := u$
   $Q.UpdateKey(v)$

Alternative derivation of Dijkstra's algorithm



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, …)
```

```
GraphExploreSkeleton(G, s):
GenericQueue Q
Q.add(s)
while !Q.empty()
    u := Q.remove()
    if  !u.visited
        u.visited := True
        for each edge (u, v) in E
            Q.add(v)
```

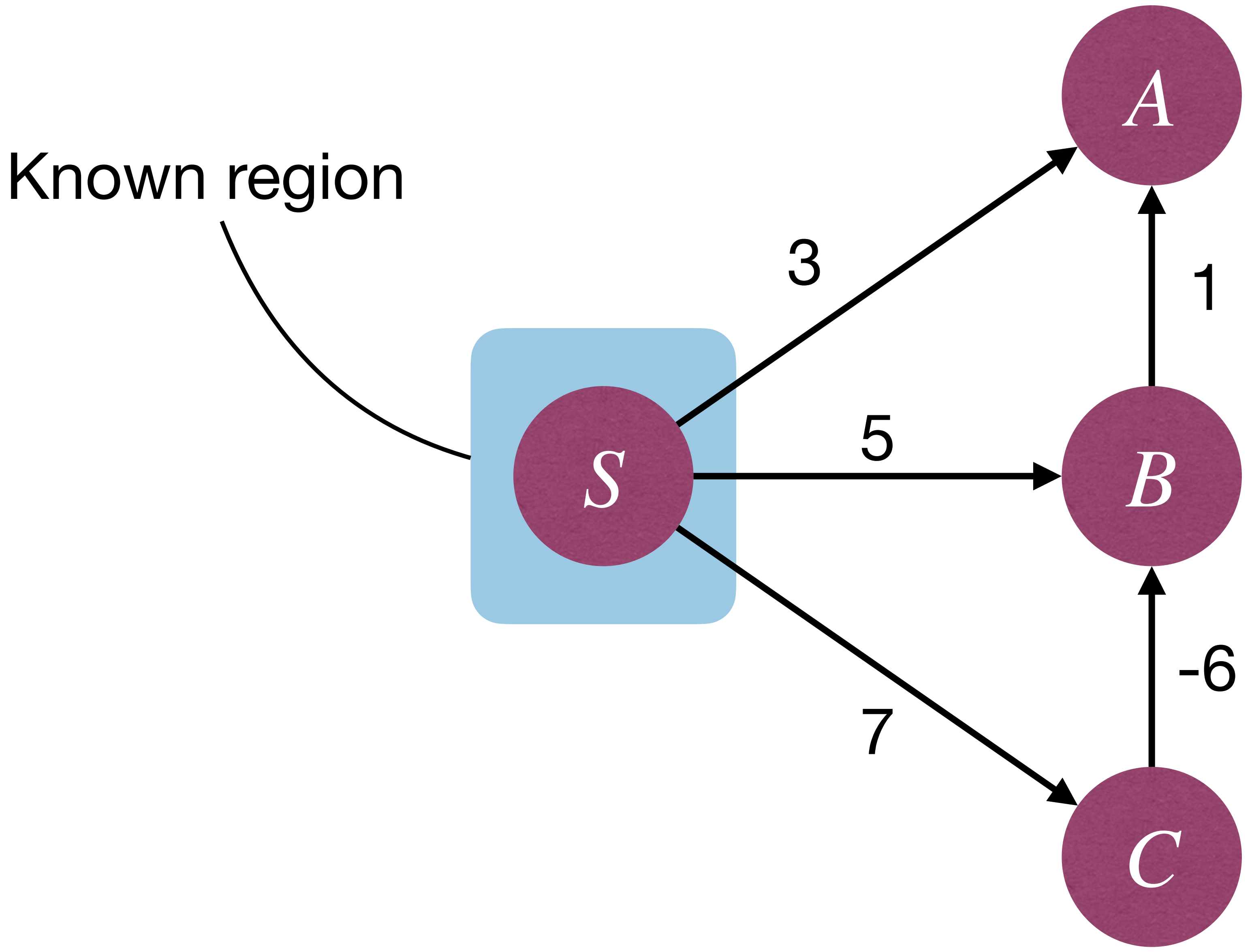
SSSP in graphs with negative weights

- 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 $\min_{u' \in R, v' \in V-R} \{dist(s,u') + w(u',v')\}$.

  - This is because: Let the last node of the path $s \rightsquigarrow v$ before v be u, then it must be $u \in R$. (Otherwise v is not the next closet node to s)

However, negative edge makes this does not hold!

SSSP in graphs with negative weights



Shortest distance from S to node A is 3? No!!!
Try $S \to C \to B \to A$

- "*Shortest path from s to any node v must pass through nodes that are closer than v*" no longer holds!

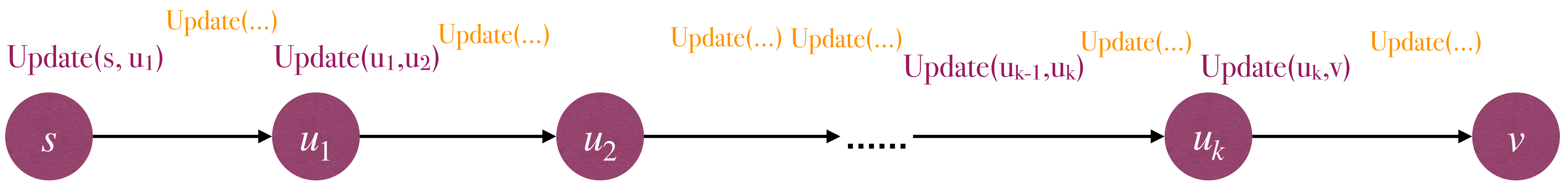
SSSP in graphs with negative weights

- But how $dist$ values are maintained in Dijkstra is helpful:
  - Initially set $s.dist = 0$, and for each node $u \neq s$, set $u.dist = \infty$.
  - When processing edge (u, v), execute procedure `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 `Update(u, v)`, then $v.dist$ becomes correct.

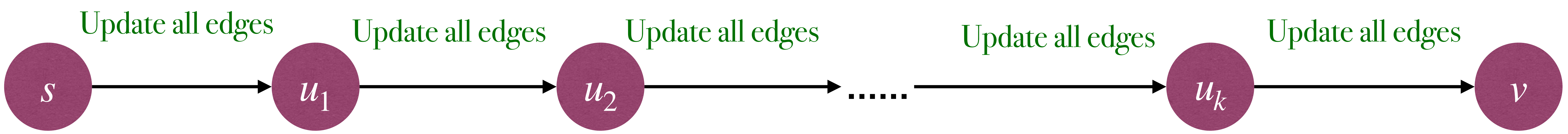
SSSP in graphs with negative weights

- `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 $v.dist$.

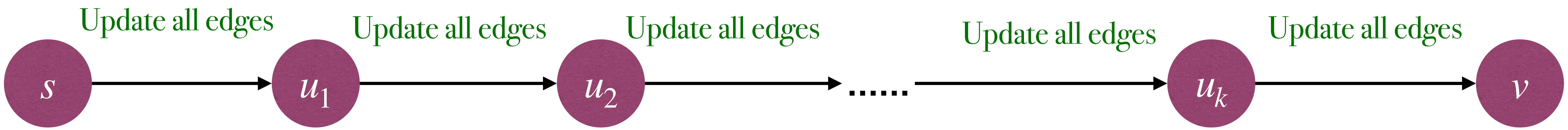
SSSP in graphs with negative weights



- Consider a shortest path from s to v.
  - **Observation 1**: if `Update(s, u₁)`, `Update(u₁,u₂)`, …, `Update(uₖ₋₁,uₖ)`, `Update(uₖ,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 v.
- Algorithm: simply `Update` all edges, for $k+1$ times!



SSSP in graphs with negative weights



- But how large is $k + 1$?
  - **Observation 3**: any shortest path cannot contain a cycle. (WHY?)
- Algorithm: simply `Update` <u>all</u> edges, for $n - 1$ times!
  - The Bellman-Ford Algorithm!

The Bellman-Ford Algorithm

- Bellman-Ford Algorithm:
  - `Update` all edges;
  - Repeat above step for $n-1$ times.
- The complexity is : $\Theta\left(n(m+n)\right)$

Richard E. Bellman

Lester Randolph Ford Jr.

<u>BellmanFordSSSP(G, s):</u>

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

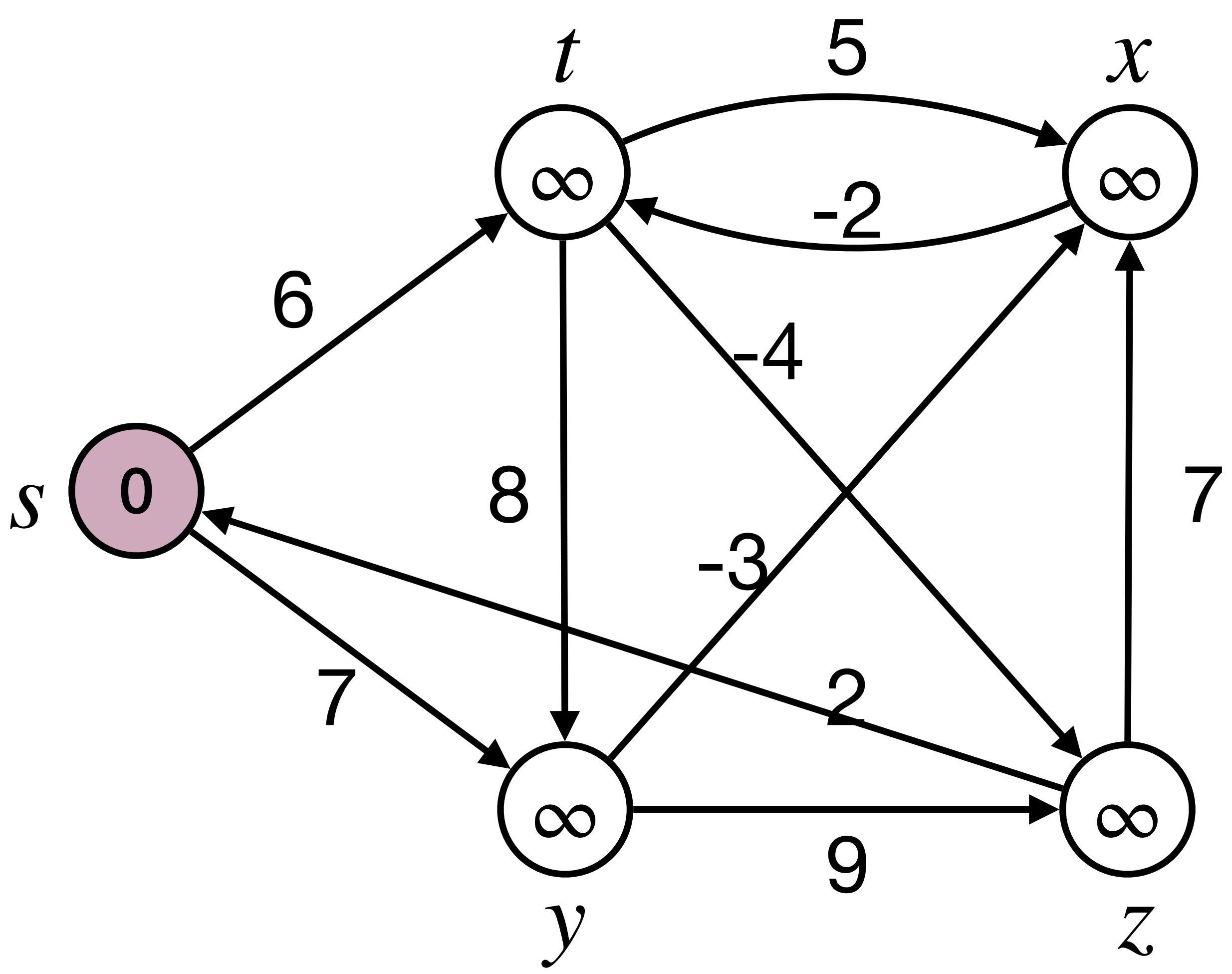
    if $v.dist > u.dist + w(u, v)$

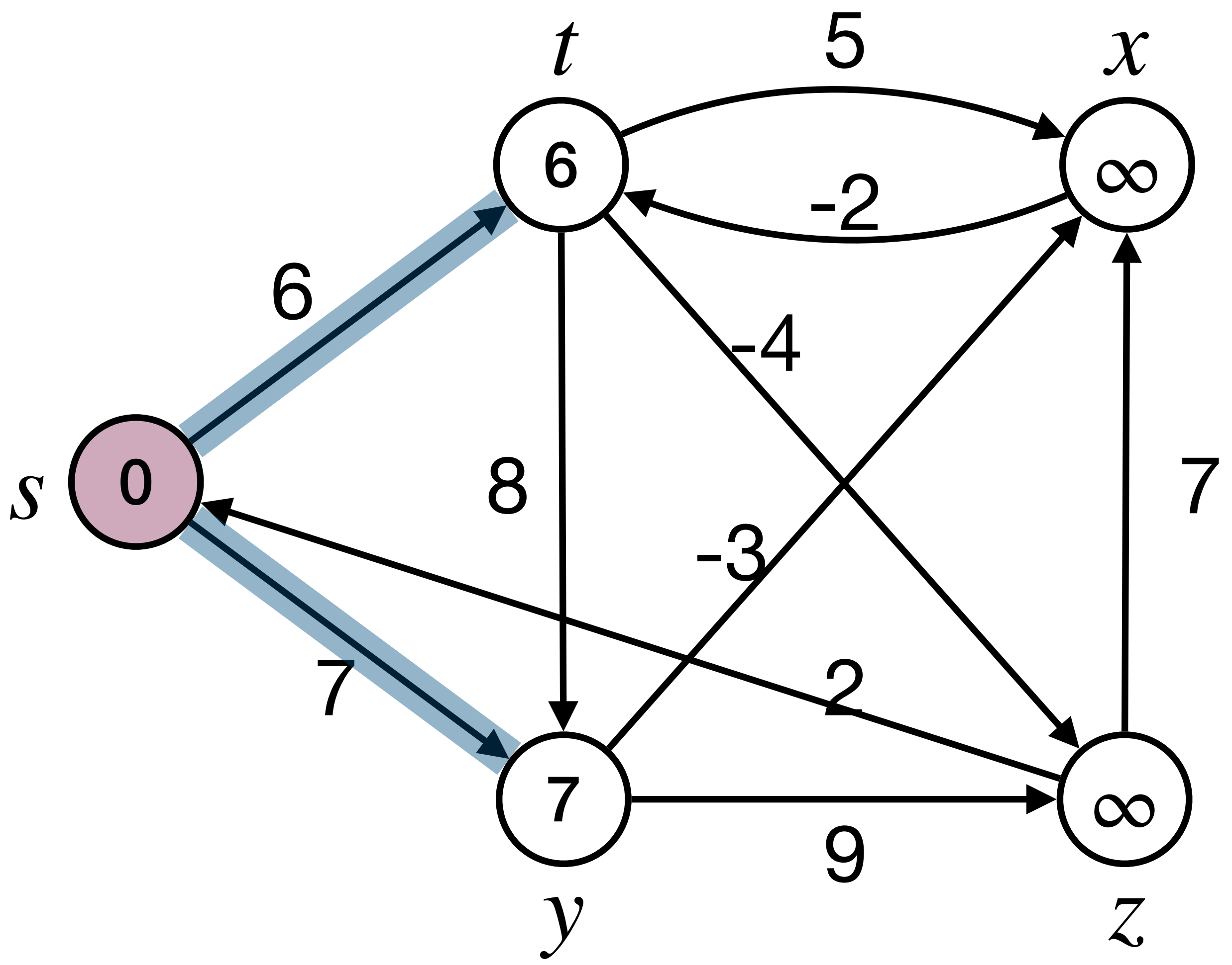
      $v.dist := u.dist + w(u, v)$

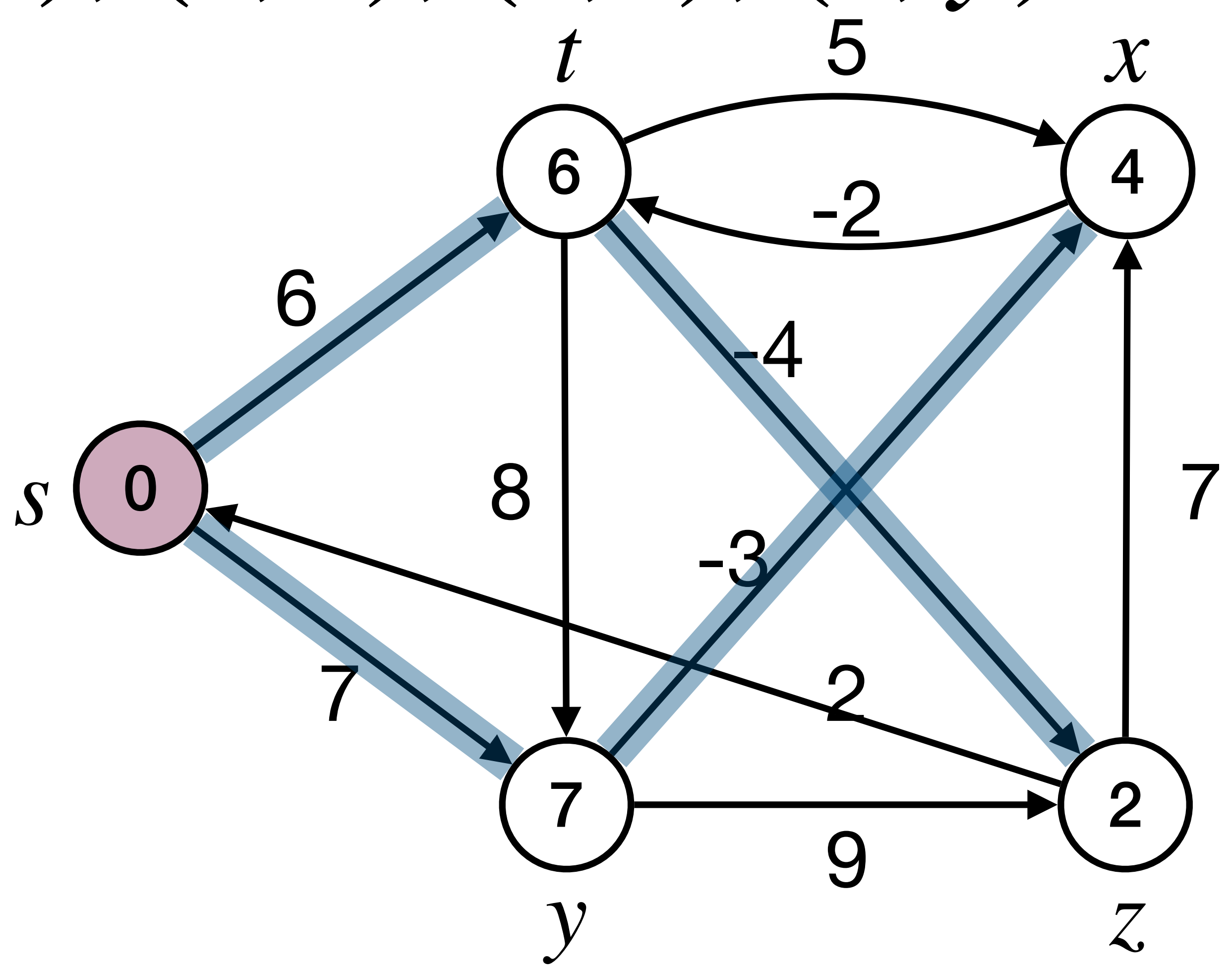
      $v.parent := u$

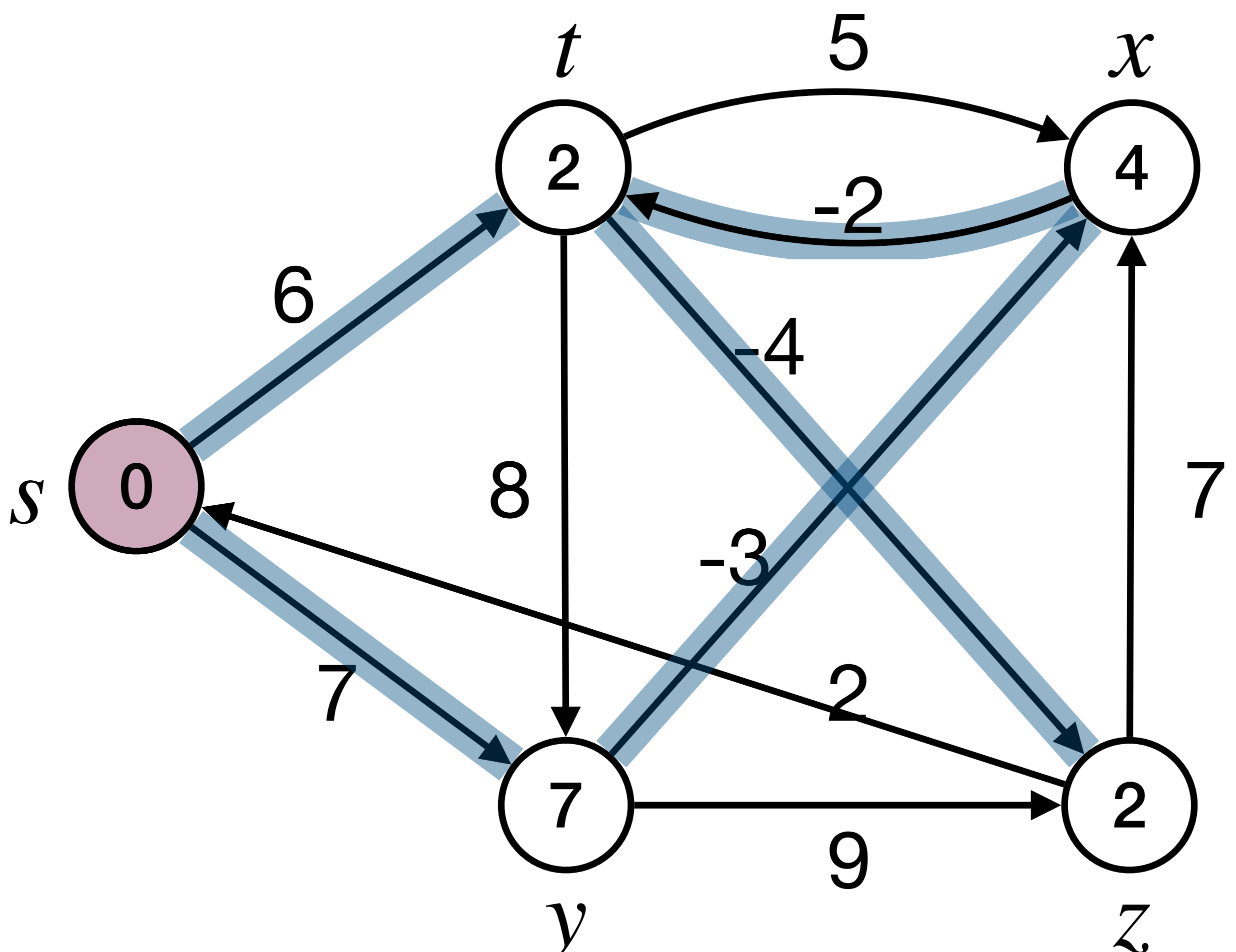
The Bellman-Ford Algorithm

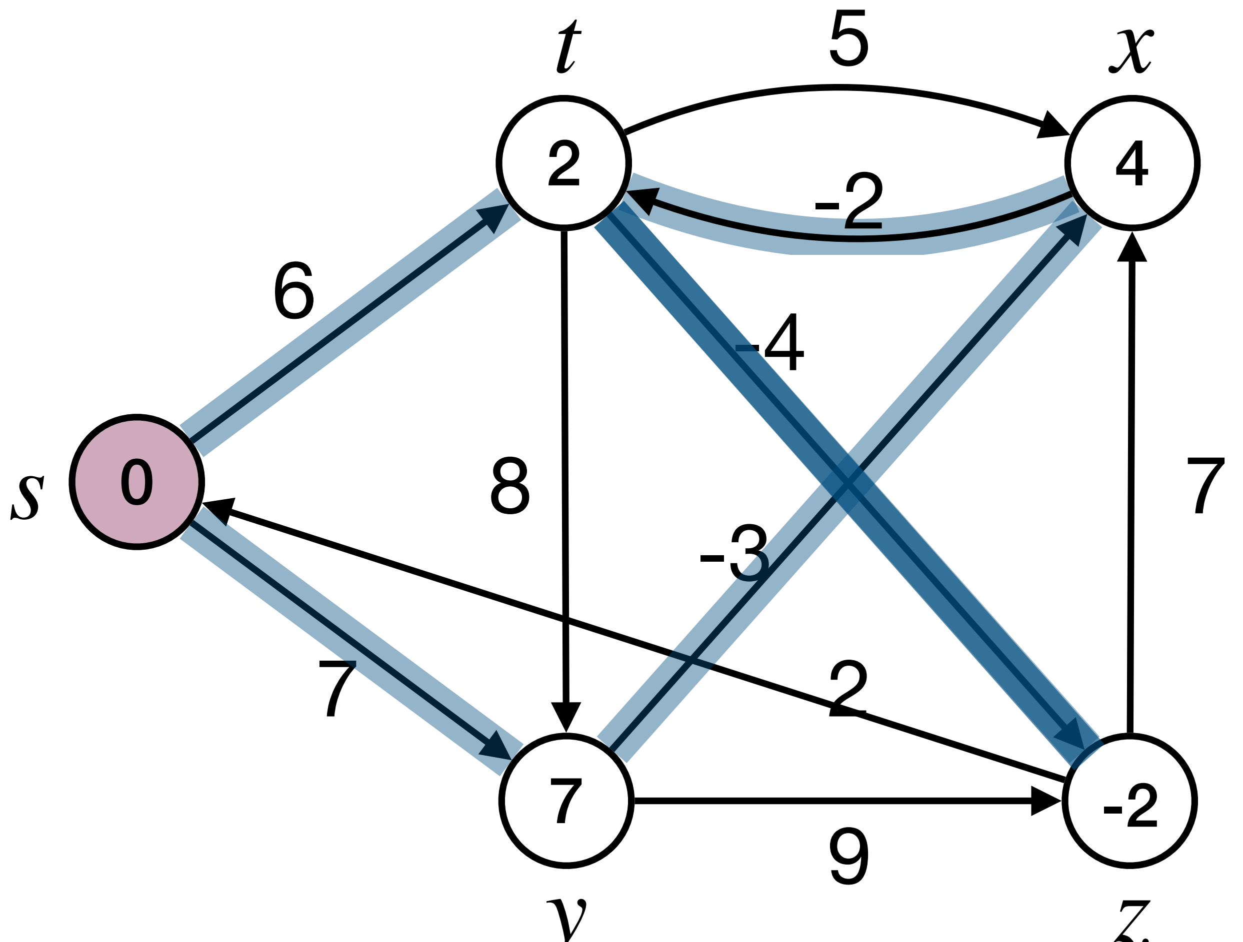
- Edge order: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$











The Bellman-Ford Algorithm

- What if the graph contains a negative cycle?
  - Then the **Observation 3** (any shortest path cannot contain a cycle.) does not hold!
  - It means that 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 circles"

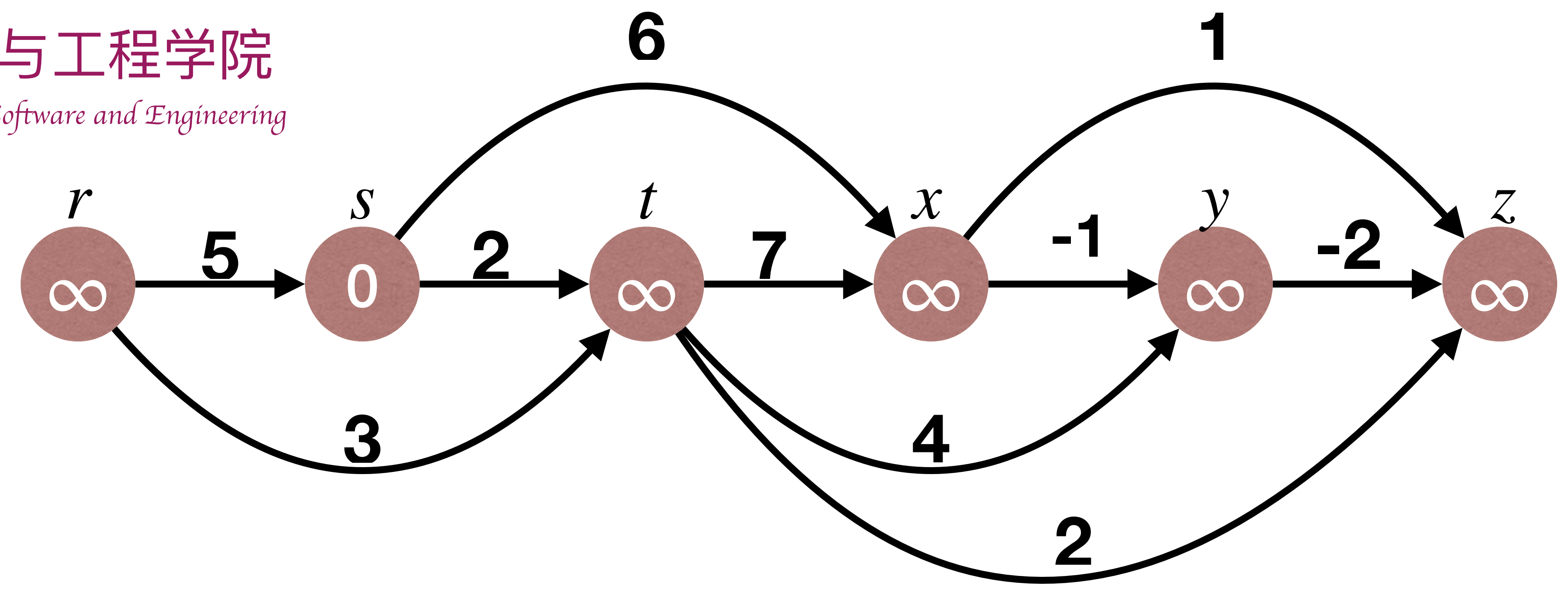
SSSP in DAG (with negative weights)

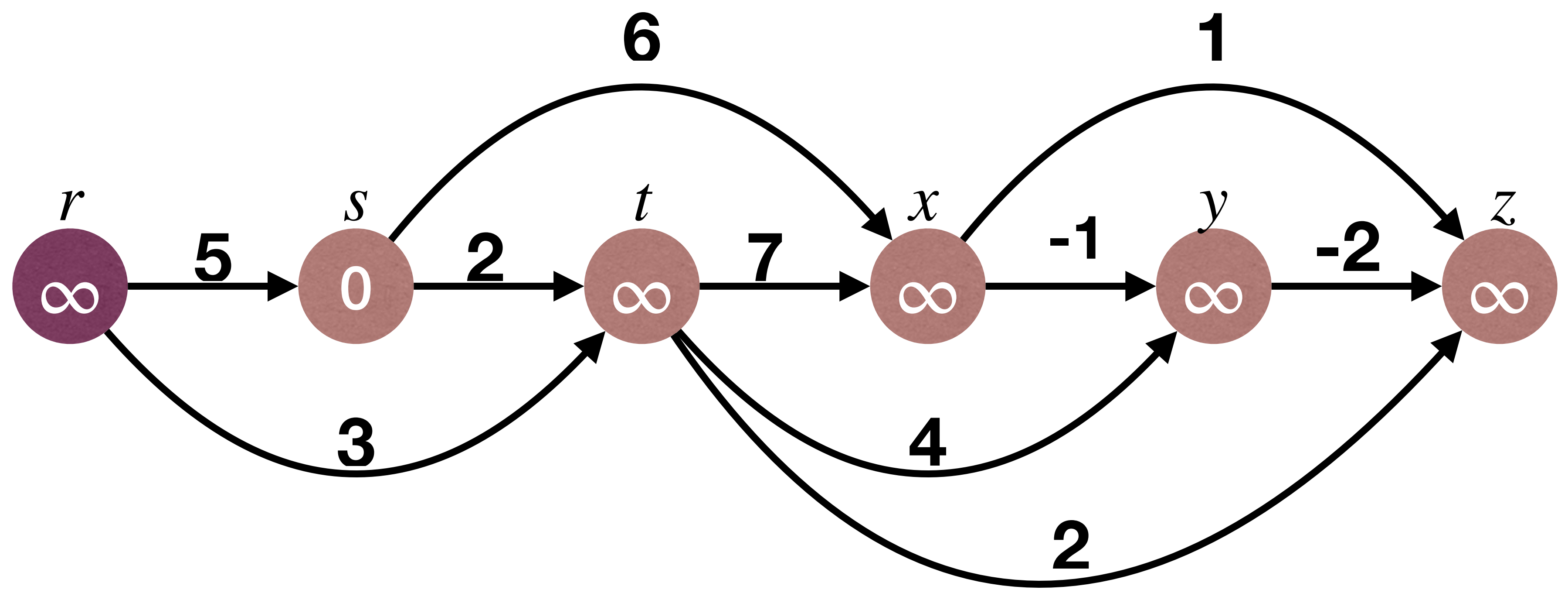
$O(m+n)$ time complexity

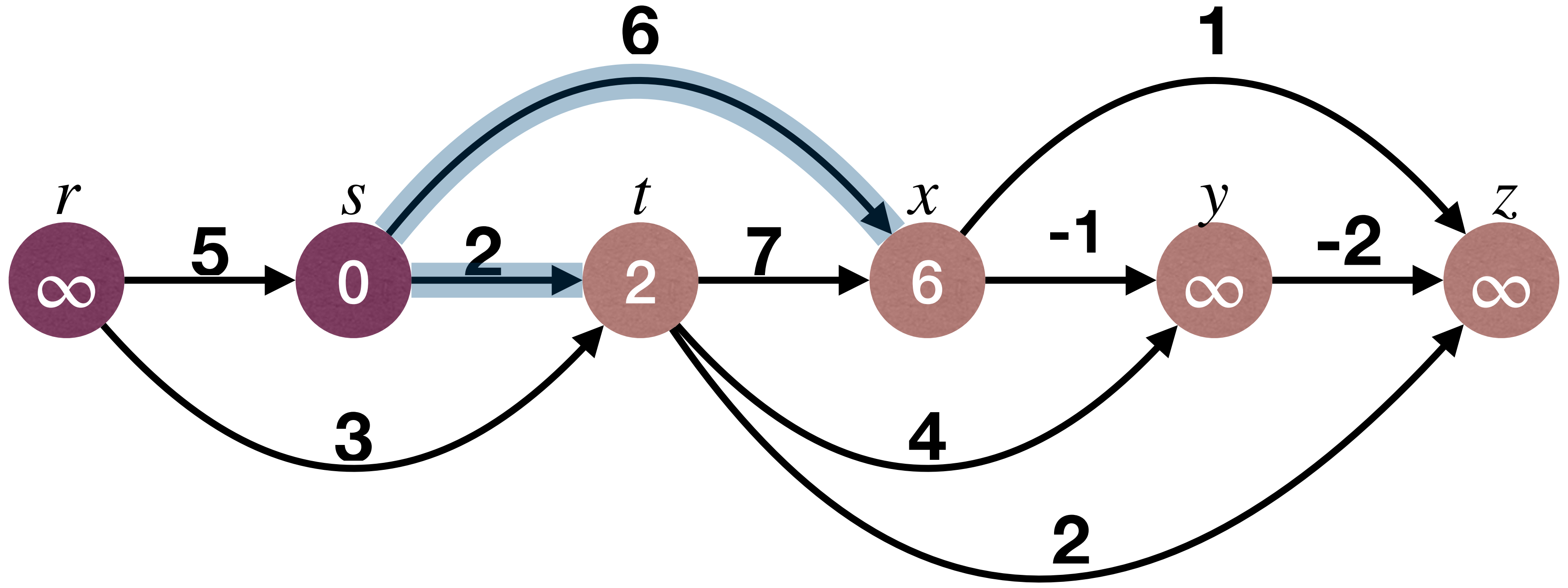
- Bellman-Ford still works, but we can be more efficient!
- Core idea of Bellman-Ford: perform a sequence of `Update` that includes every shortest path as a subsequence.
- Observation: in DAG, every path, thus every shortest path, is a subsequence in the topological order.

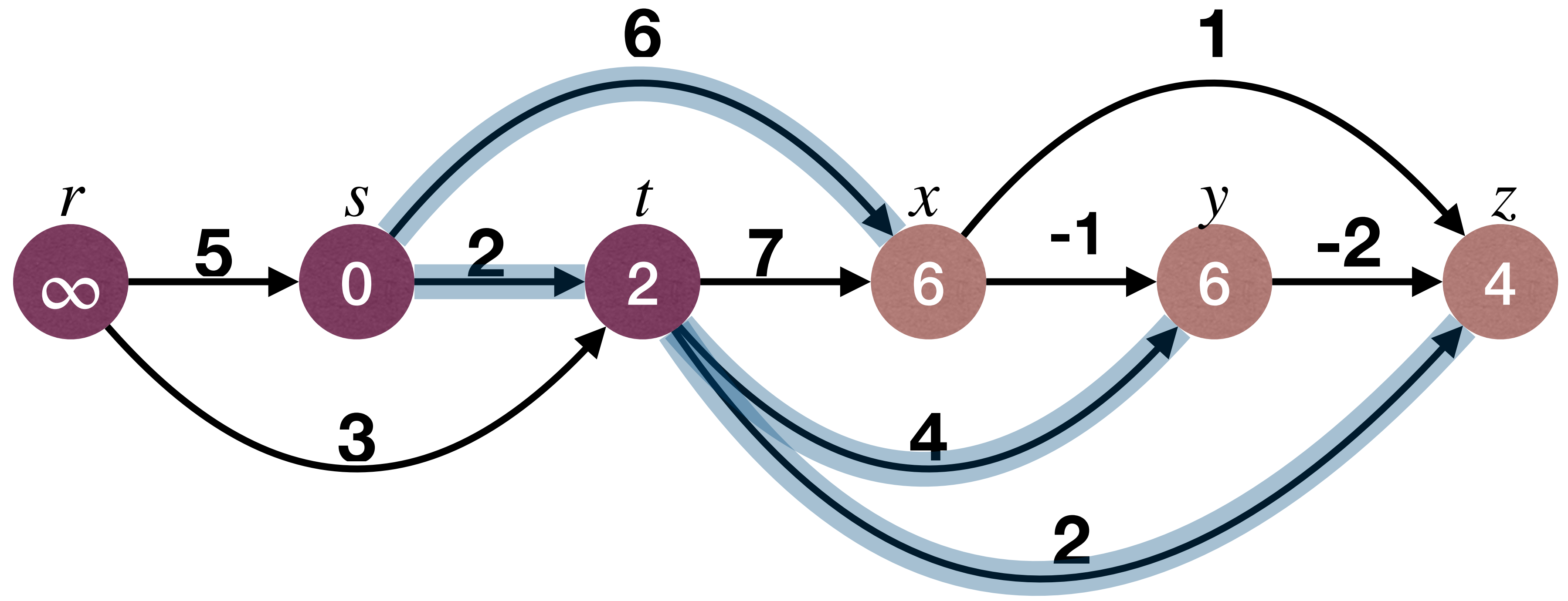
DAGSSSP(G,s):

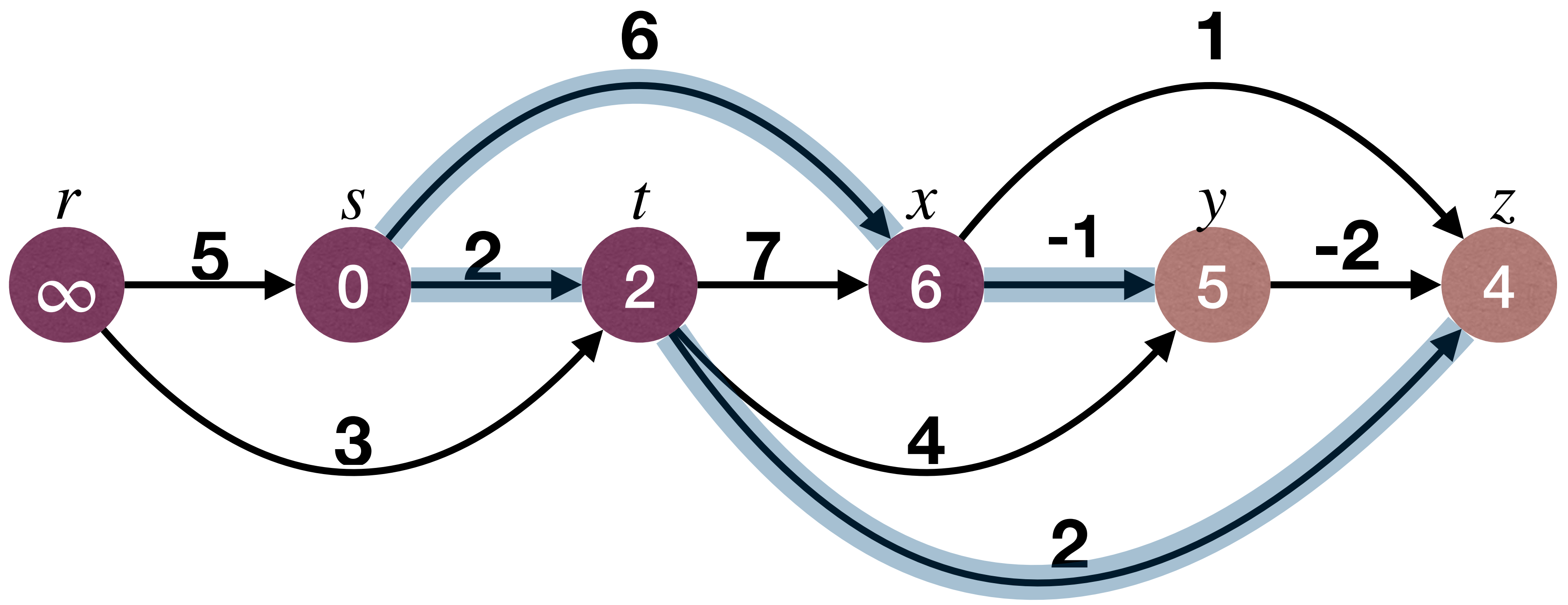
for **each** u **in** V
    $u.dist := INF, u.parent := NIL$
$s.dist := 0$
Run DFS to obtain topological order
for **each** *node* u **in** *topological order*
    for **each** *edge* (u, v) **in** E
        if $v.dist > u.dist + w(u, v)$
            $v.dist := u.dist + w(u, v)$
            $v.parent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

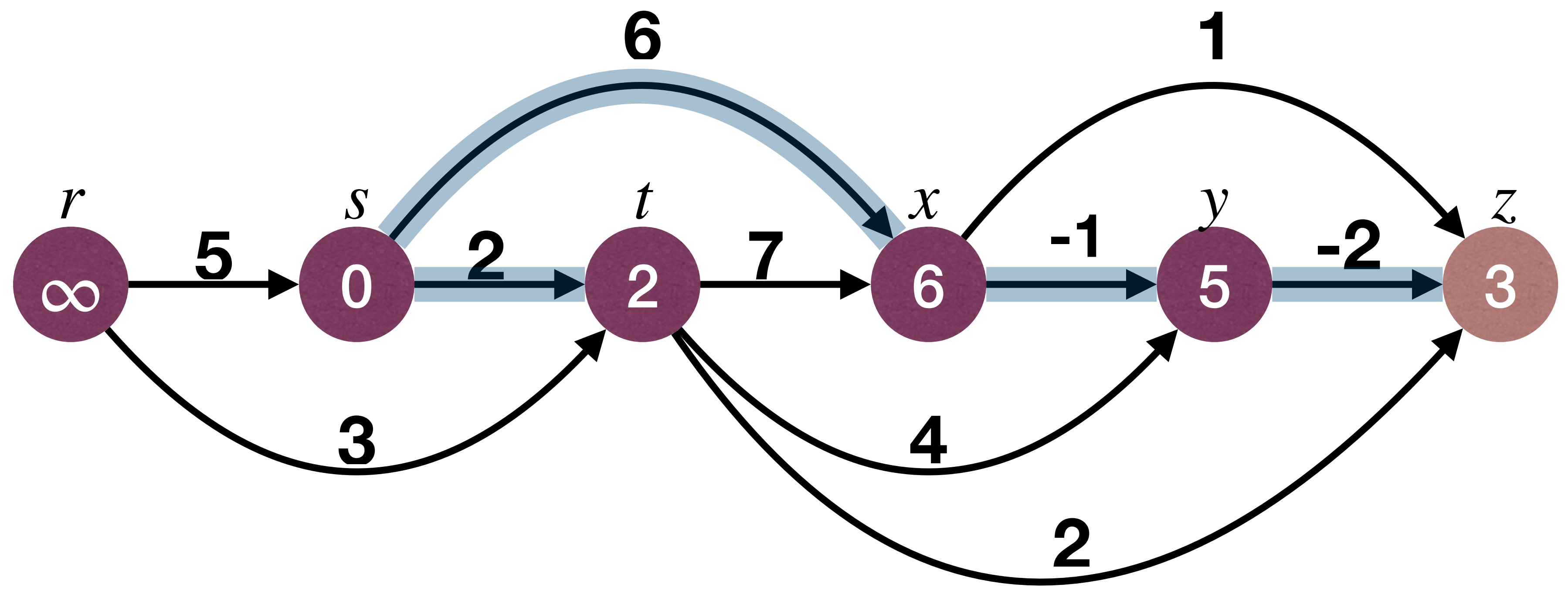

6 1
r s t x y z
5 2 7 -1 -2
∞ 0 ∞ ∞ ∞ ∞
3 4 2

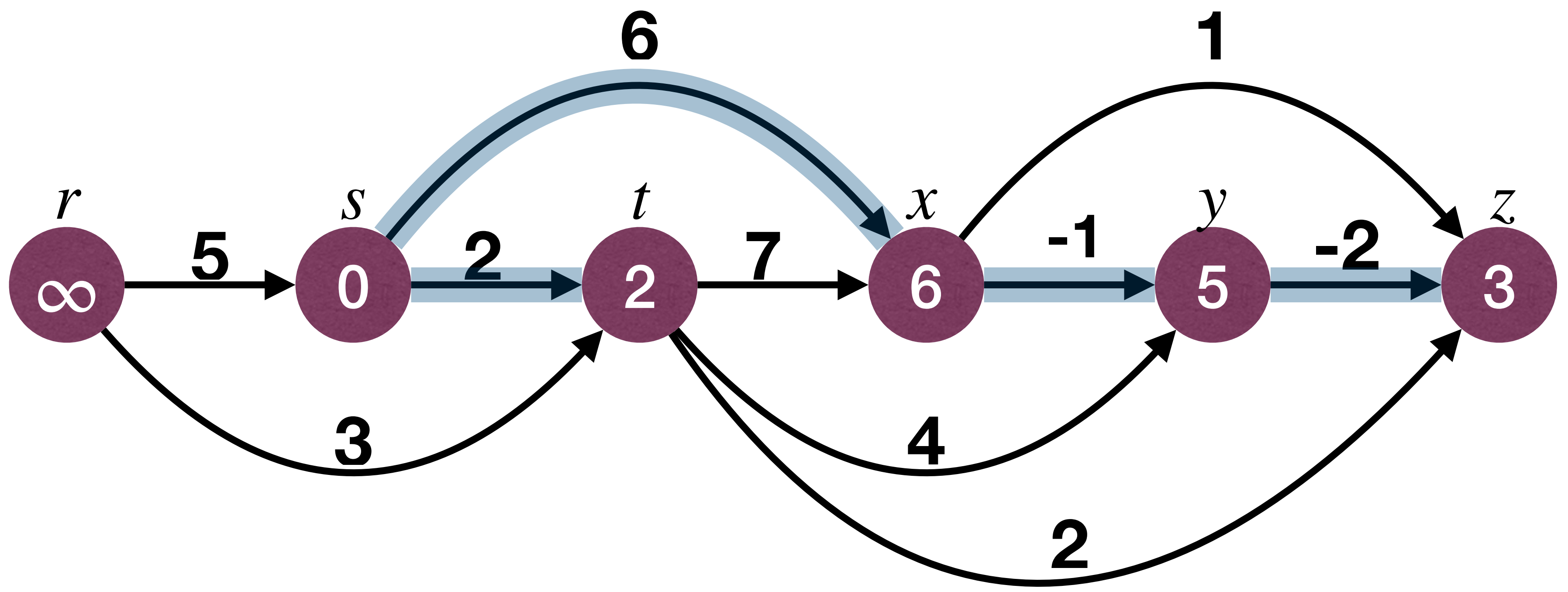

6 1
r s t x y z
5 2 7 -1 -2
∞ 0 ∞ ∞ ∞ ∞
3 4 2


6 1
r s t x y z
5 2 7 -1 -2
∞ 0 2 6 ∞ ∞
3 4 2


6 1
r s t x y z
5 2 7 -1 -2
∞ 0 2 6 6 4
3 4 2

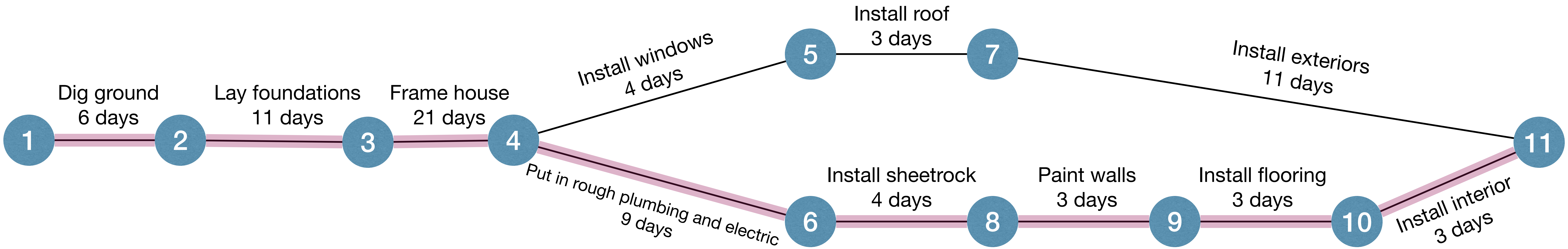

6 1
r s t x y z
5 2 7 -1 -2
∞ 0 2 6 5 4
3 4 2


6 1
r s t x y z
5 2 7 -1 -2
∞ 0 2 6 5 3
3 4 2


6 1
r s t x y z
5 2 7 -1 -2
∞ 0 2 6 5 3
3 4 2

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. $\Theta(n(m+n))$ runtime, can detect negative cycle.

The shortest path problem has optimal substructure property.

`Update` **is a safe and helpful operation.**

Pathfinding*

(Shortest) Pathfinding*

- Given a graph $G = (V, E)$, how to find a (shortest) path from a source s to a destination t, preferably efficiently.

We could use BFS or Dijkstra.

But we could be **MUCH** faster!

Greedy Best-First Search

GreedyBFS(G, s, t):

$s.est\_to\_goal := heuristic(s,t)$

Build priority queue Q based on est_to_goal

while $!Q.empty()$

    $u := Q.ExtractMin()$

    for each *edge* (u,v) **in** E

        if $v \notin Q$

            $v.est\_to\_goal := heuristic(v,t)$

            $v.parent := u$

            $Q.Add(v)$

Does greedy BFS always generate correct answer?

- A (not necessarily accurate) *estimate* on the distance from v to t.
  - On 2D grid, we can set $heuristic(v,t) = ManhattanDist(v,t) = |v.x - t.x| + |v.y - t.y|$.

Greedy BFS does not always generate correct answer

PathfindingFramework(G, s, t):

for each *node u* **in** *V*

  $u.metric$:= $INFINITY$

$s.metric$:= $CalcMetric(s,s,t)$

Build priority queue Q based on metric

while $!Q.empty()$

  $u := Q.ExtractMin()$

  for each *edge* (u,v) **in** E

    $new\_metric := UpdateMetric(v, u, s, t)$

    if $v \notin Q$ **or** $new\_metric < v.metric$

      $v.metric := new\_metric$

      $v.parent := u$

      $Q.AddorUpdate(v)$

GreedyBFS: $est\_to\_goal(s, t)$

Dijkstra: $est\_to\_source(s,s) := 0$

GreedyBFS: $est\_to\_goal(v, t)$

Dijkstra:

$update\_est\_to\_source(v,u,s)$

$\min\{v.metric, u.metric + dist(u,v)\}$

a.k.a, $\min\{v.metric, dist(s, u) + dist(u,v)\}$

GreedyBFS is fast, but may be incorrect;
Dijkstra's algorithm is slower, but always correct;
Can we have an algorithm that is both fast and correct?

The A* algorithm

- For each node u:
  - $u.est\_to\_s$ maintains an (**over** or accurate) estimate of $dist(u,s)$, and this value changes during execution;
  - $u.est\_to\_t$ maintains an (**under** or accurate) estimate of $dist(u,t)$, and this value does not change during execution.
  - Use $u.est\_to\_s + u.est\_to\_t$ as the metric to guide the search!
- Usually set to the straight-line distance between u and t.

AStarPathfinding(G, s, t):

for each *node* u **in** V

  $u.est\_to\_s := INFINITY$

  $u.est\_to\_t := heuristic(u,t)$

  $u.metric := u.est\_to\_s + u.est\_to\_t$

$s.est\_to\_s := 0$, $s.metric := s.est\_to\_s + s.est\_to\_t$

Build priority queue Q based on metric

while $!Q.empty()$

  $u := Q.ExtractMin()$

  for each *edge* (u,v) **in** E

    if $v \notin Q$ **or** $v.est\_to\_s > u.est\_to\_s + dist(u, v)$

      $v.est\_to\_s := u.est\_to\_s + dist(u, v)$

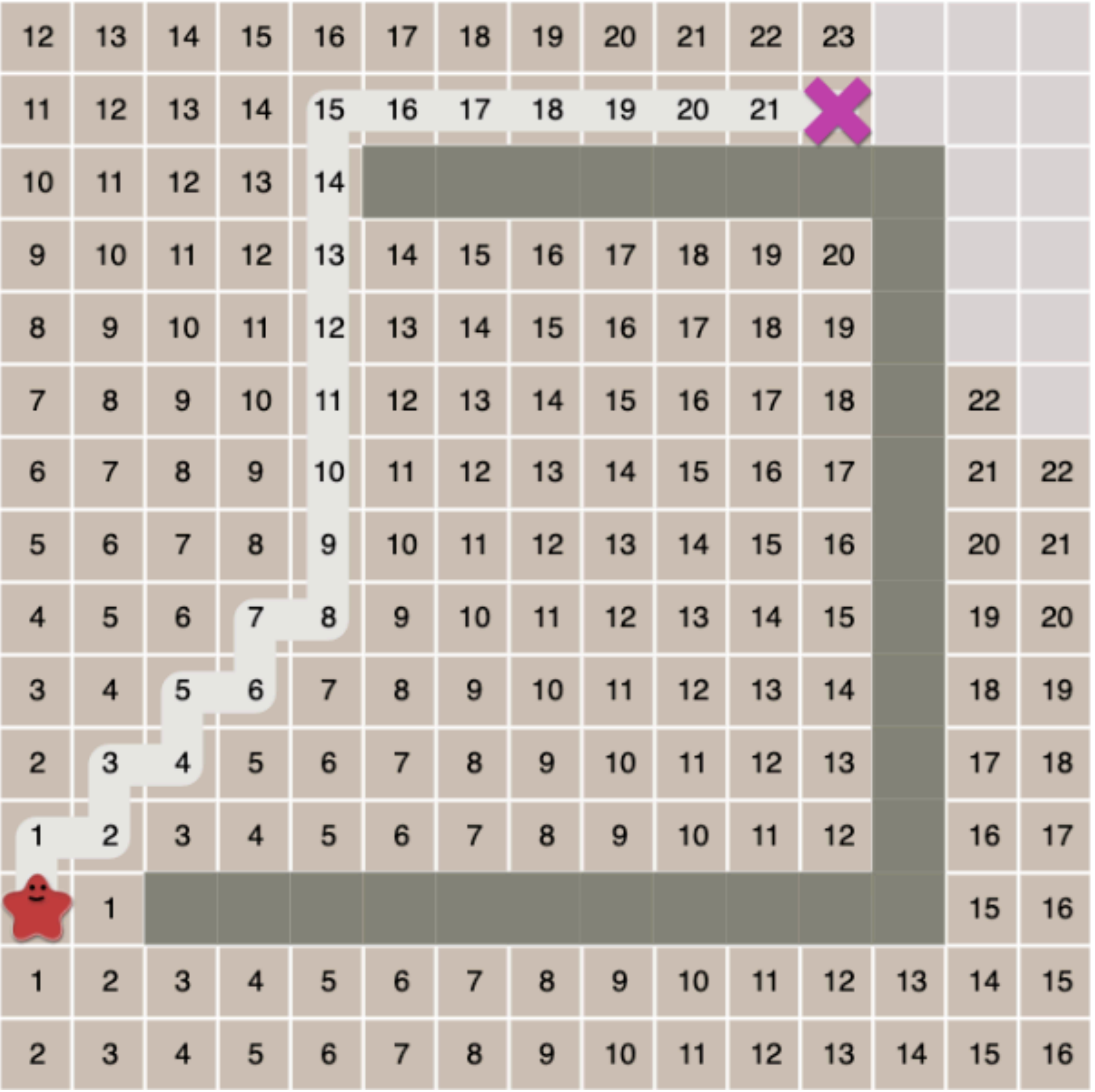
      $v.metric := v.est\_to\_s + v.est\_to\_t$

      $v.parent := u$

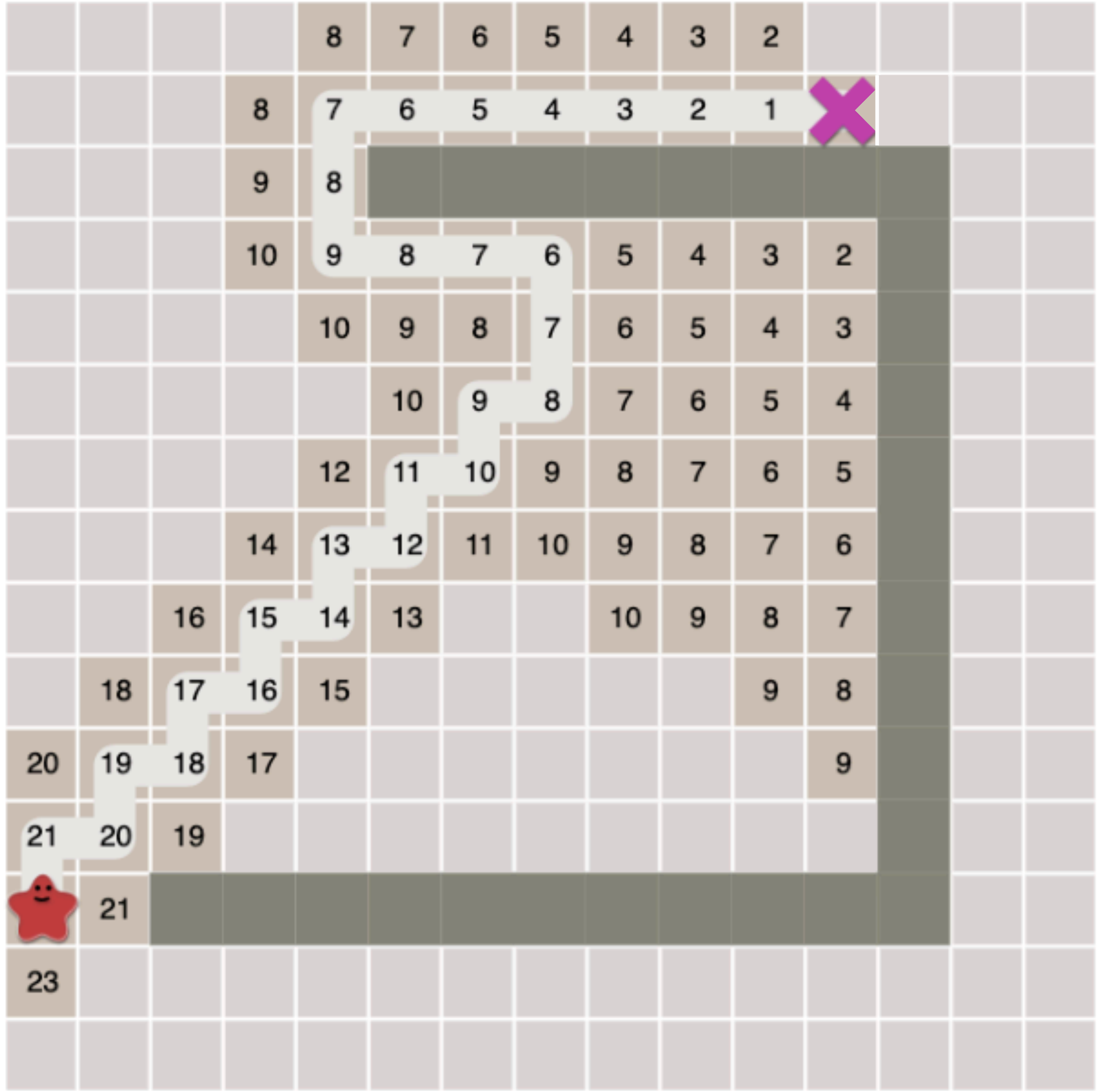
      $Q.Add(v)$

The A* algorithm

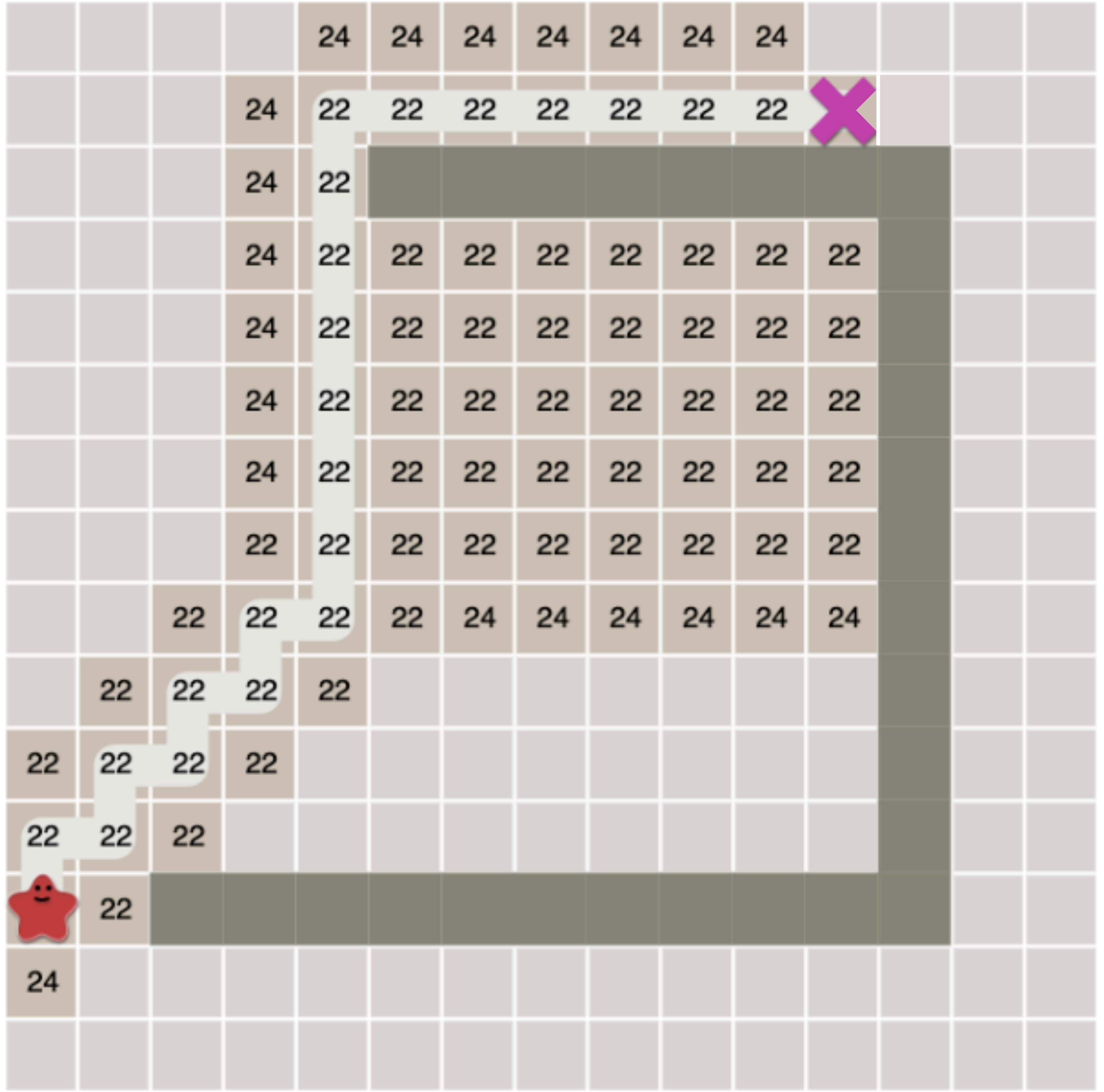
Dijkstra's Algorithm



Greedy Best-First



A* Search



f : metric, h: estimate to goal

The A* algorithm

- Correctness of the A* algorithm?
  - It is correct as long as $u.est\_to\_t \leq dist(u,t)$ always hold.
- Time complexity of the A* algorithm?
  - More complicated as a node may be added to the queue multiple times.
  - In AI community, it is normally considered to be $O(b^d)$, where b is the branching factor (the average number of successors per state), and d is the depth of the solution (the shortest path).
  - The heuristic function has a major effect on the practical performance of A* search, since a good heuristic allows A* to prune away many of the b^d nodes.

智能软件与工程学院
School of Intelligent Software and Engineering

Further reading

- [CLRS] Ch.24 (excluding 24.4)
- [DPV] Ch.4
- [Erickson] Ch.8
- Refer to https://www.redblobgames.com/pathfinding/a-star/introduction.html if you want to know more about A* algorithm