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
2024 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?

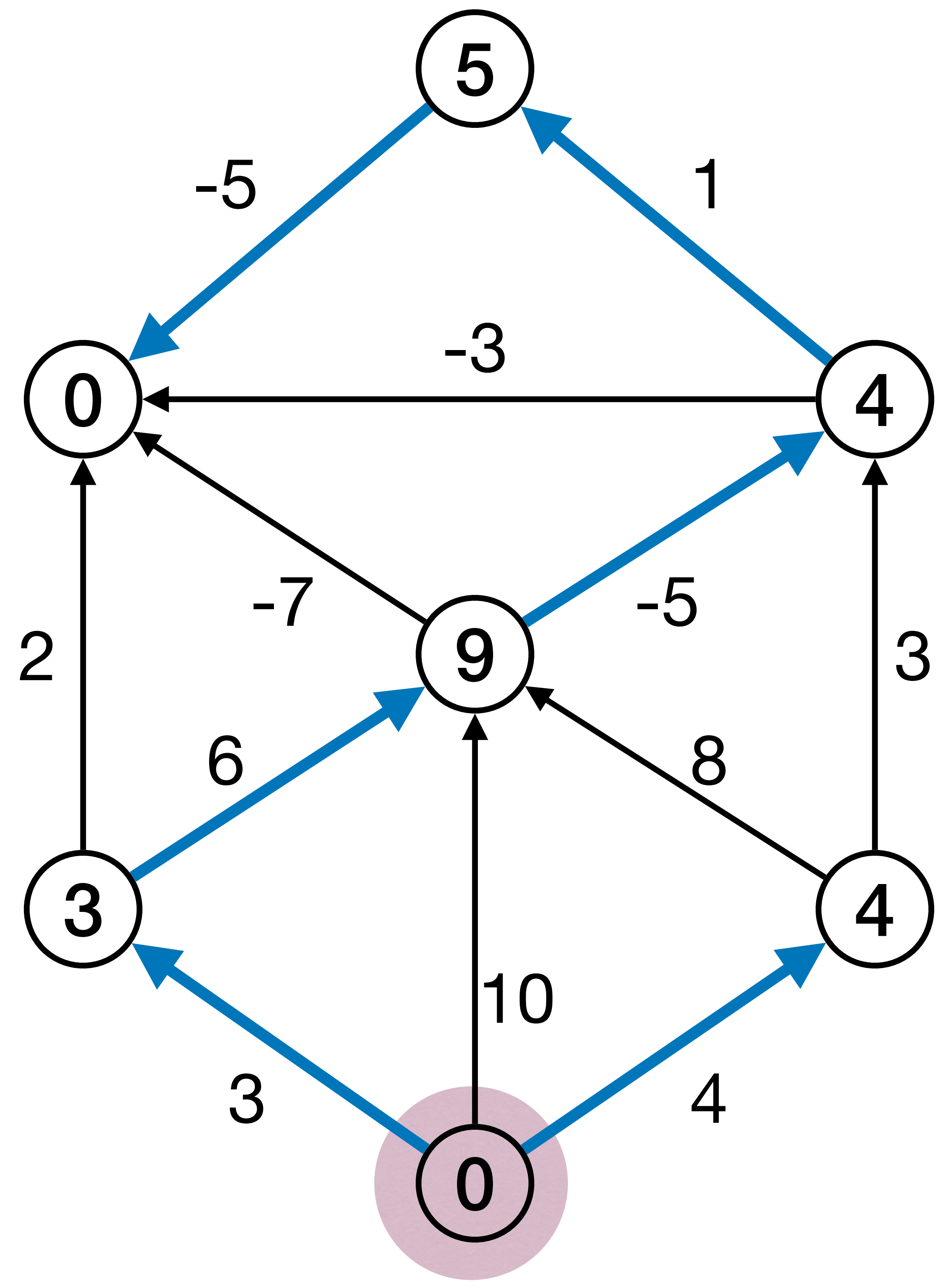
The Shortest Path Problem

- **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)

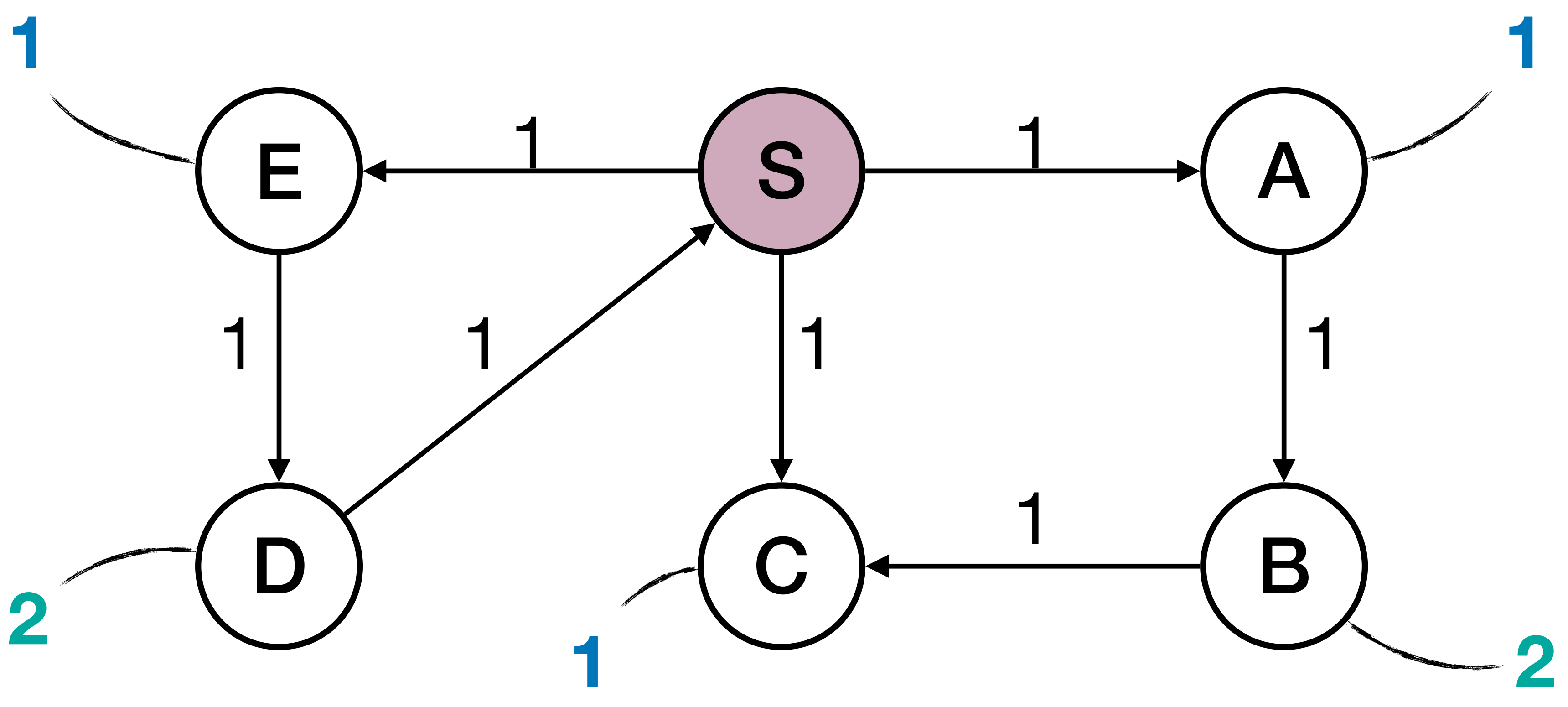
- **The SSSP Problem:** Given a graph $G = (V, E)$ and a weight function w, given a source node s, find a shortest path from s to every node $u \in V$.

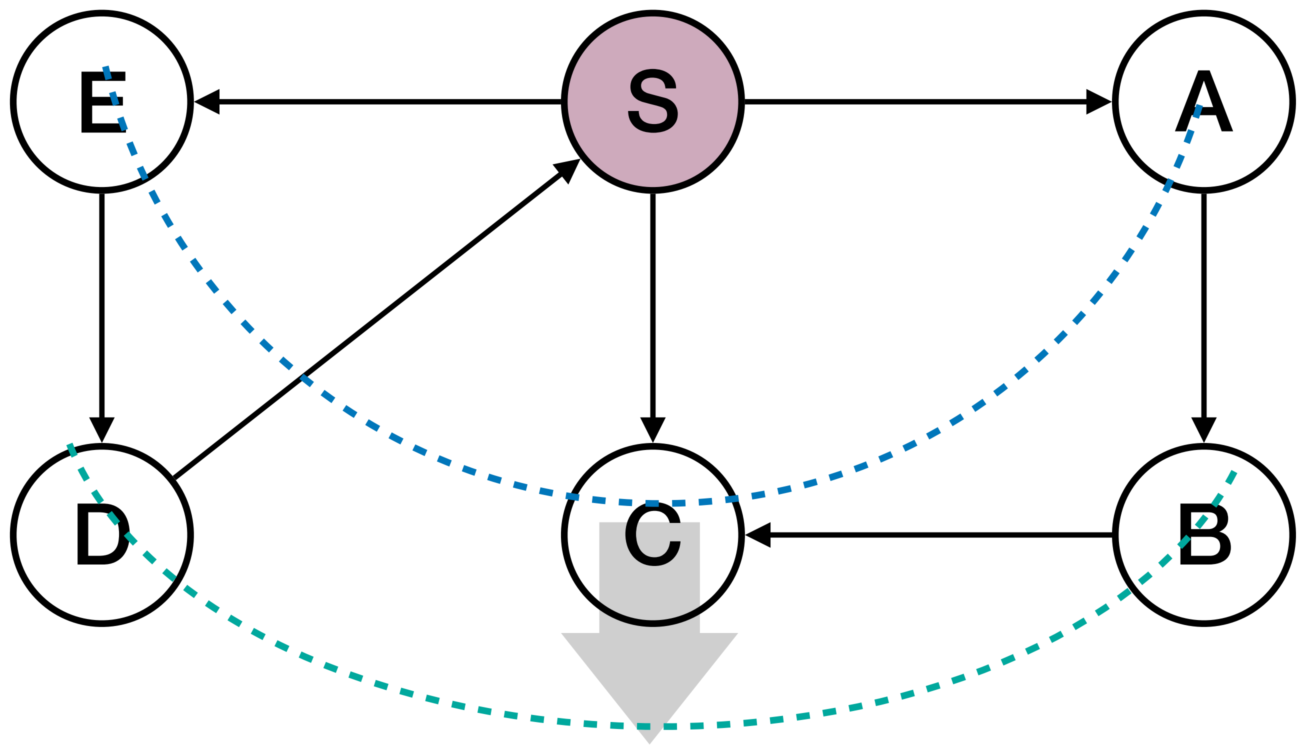
- Consider *directed* graphs *without* **negative cycle**.
  - **Case 1**: Unit weight.
  - **Case 2**: Arbitrary positive weight.
  - **Case 3**: Arbitrary weight without cycle.
  - **Case 4**: Arbitrary weight.



SSSP in unit weight graphs

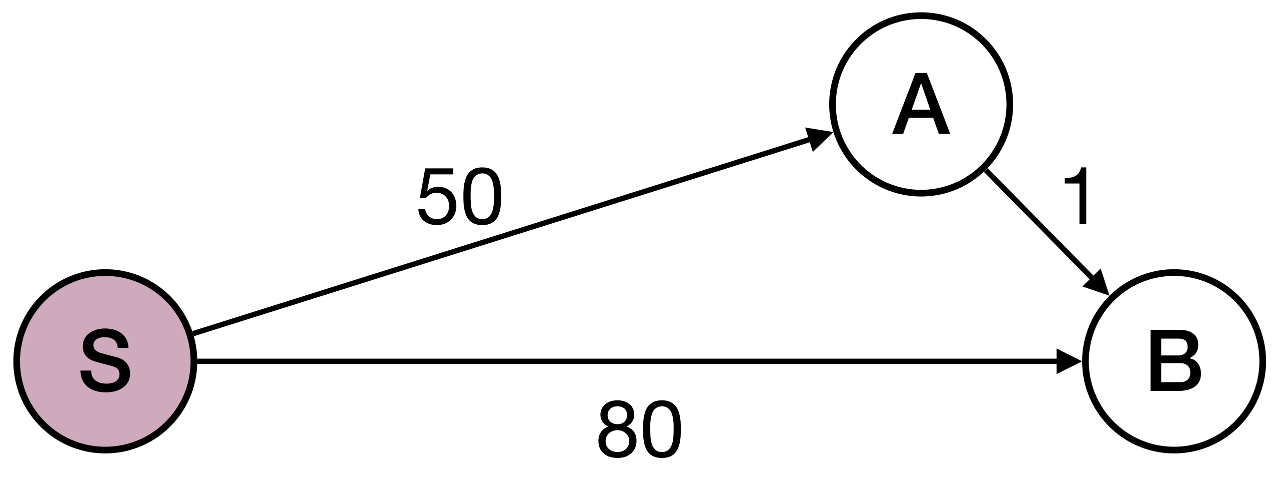
- How to solve SSSP in an unit weight graph?
  - That is, a graph in which each edge is of weight **1**.
- “Traverse by layer” in an unweighted graph!
  - Visit all distance d nods before visiting any distance $d+1$ node.
  - Simple, just use BFS!

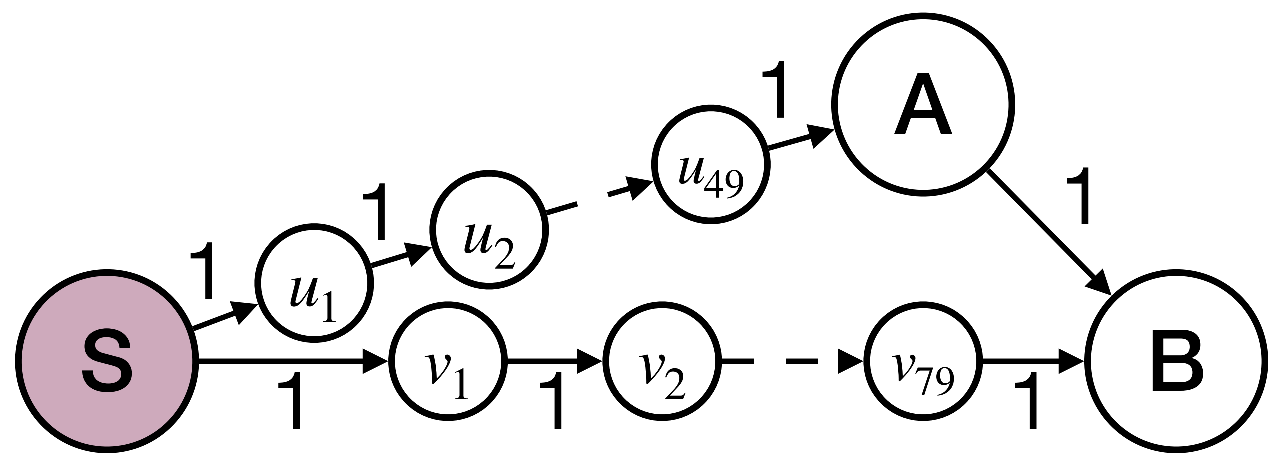




SSSP in positive weight graphs

- 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.

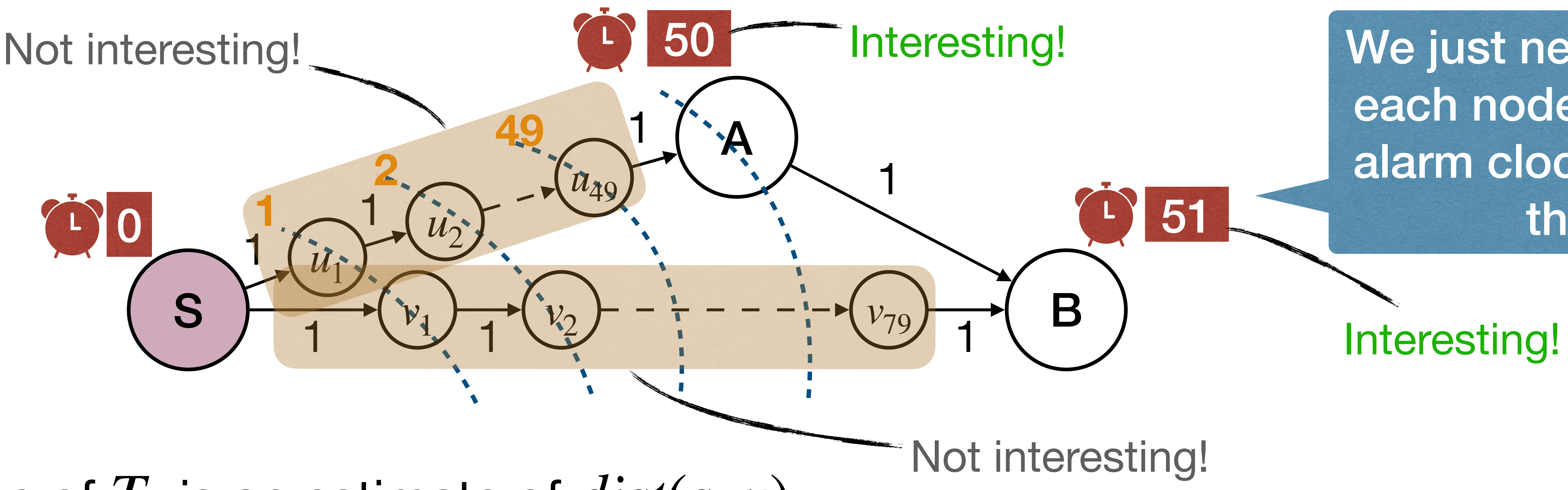




The problem is that it is too slow when edge weights are large!

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 $T_v = \min\{T_v, T_u + w(u, v)\}$

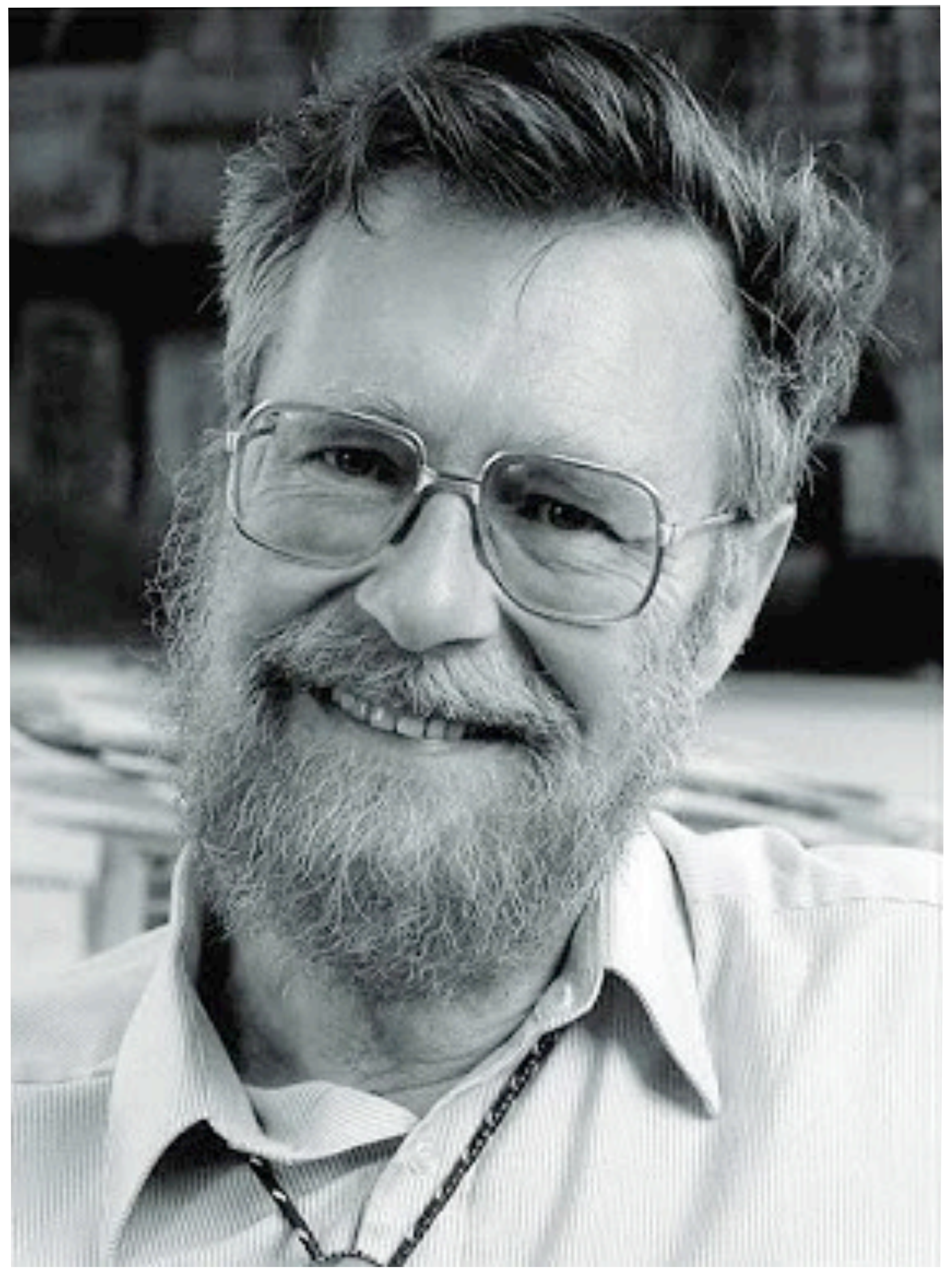


We just need to set the alarm clock of each node, and update it when some alarm clock goes off (snooze through these boring nodes)!

- 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):

```
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)
```

Shortest-path Tree (Similar to BFS tree.) — annotation pointing to *u.parent*

Dijkstra's algorithm

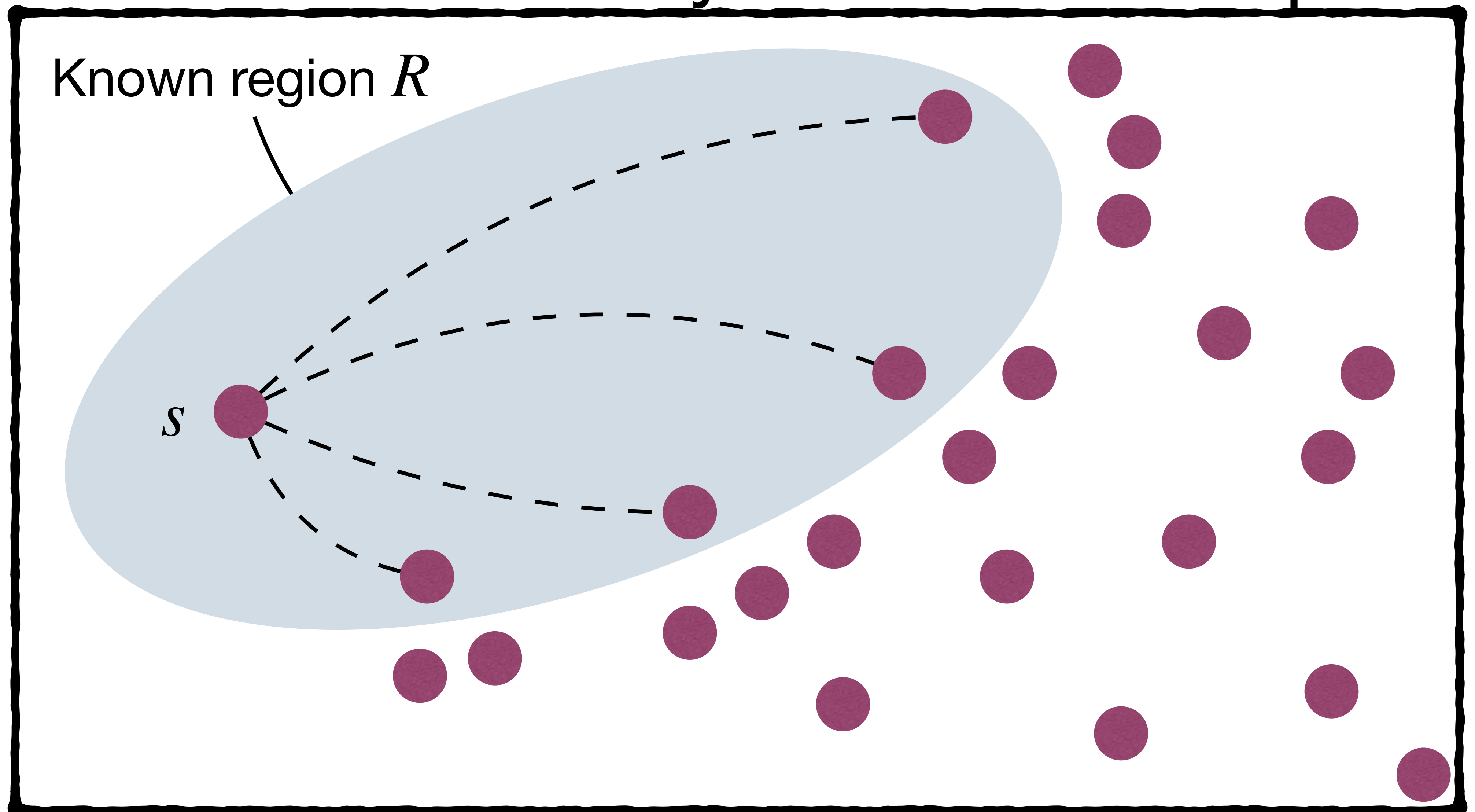
- Correctness of Dijkstra's algorithm?
  - Similar to the correctness proof of BFS.
- Efficiency of Dijkstra's algorithm?
  - $O((n+m)\cdot\log n)$ when using a binary heap.

DijkstraSSSP(G, s):

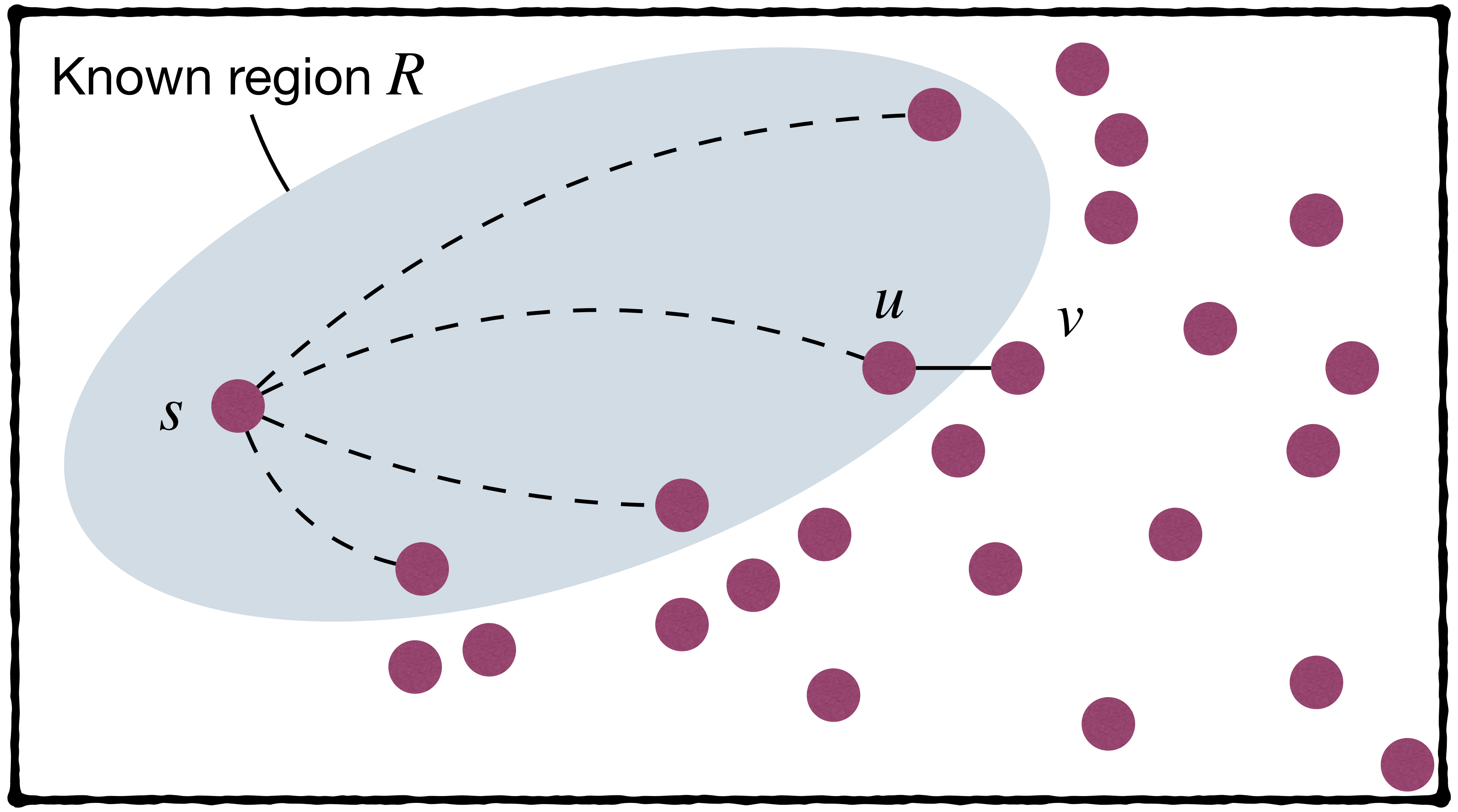
```
for each u in V
    u.dist := INF, u.parent := NIL          O(n)
s.dist := 0
Build priority queue Q based on dist        O(n)
while !Q.empty()
    u := Q.ExtractMin()                     O(n log n)
    for each edge (u,v) in E
        if v.dist > u.dist + w(u, v)
            v.dist := u.dist + w(u,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

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?

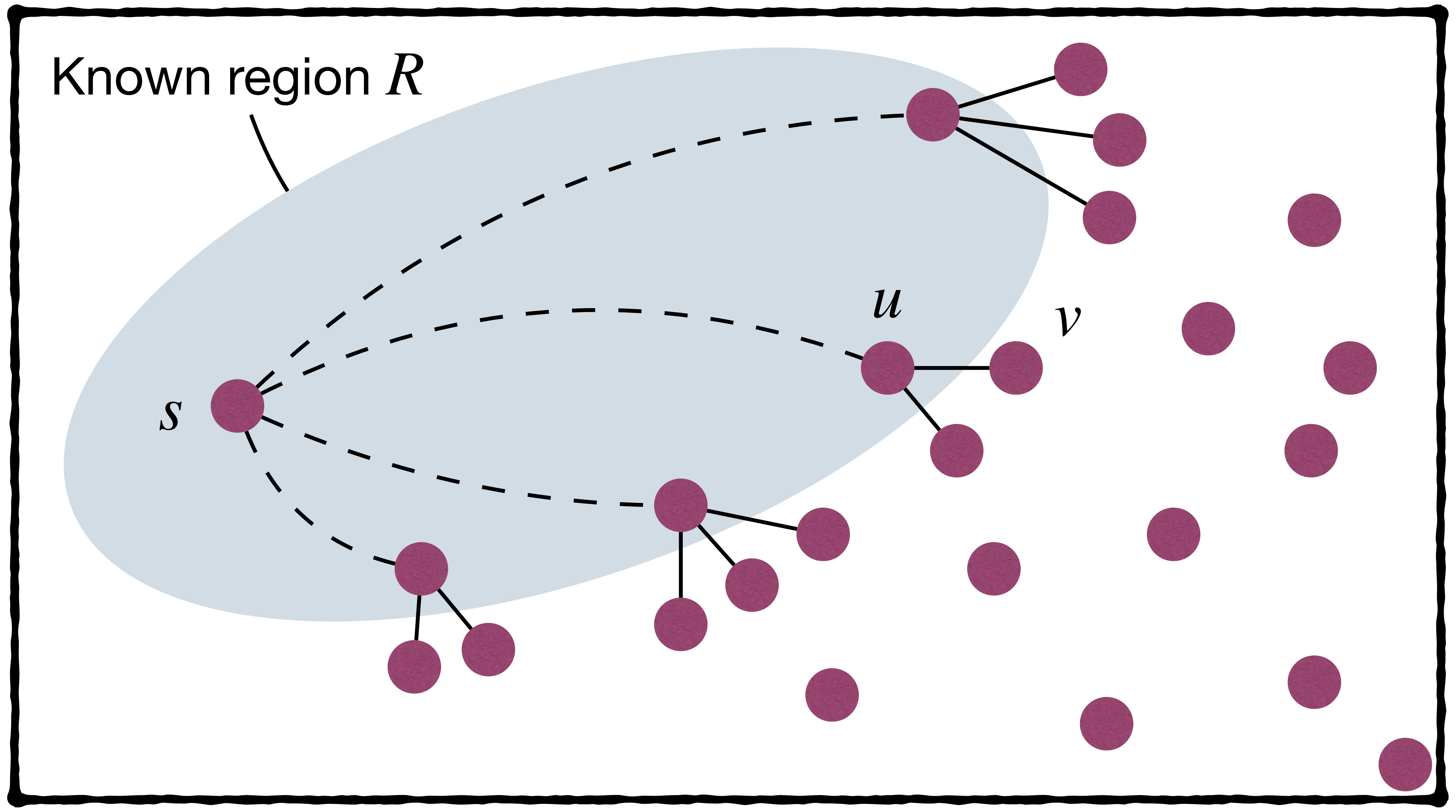


Alternative derivation of Dijkstra's algorithm



- 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



optimal substructure property

- 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):

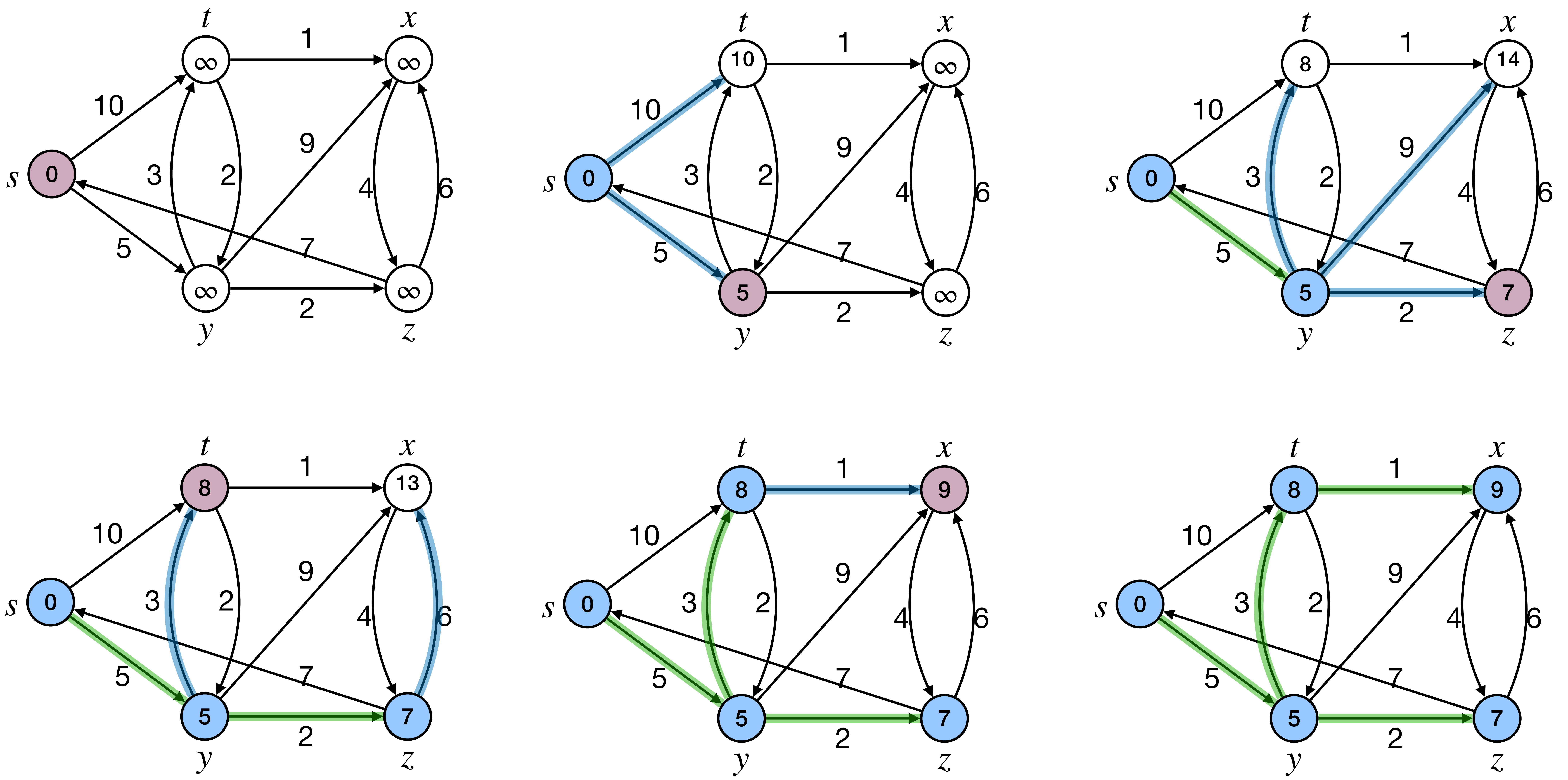
```
for each u in V
    u.dist := INF
s.dist := 0
R := ∅
while R != V
    Find node v in V - R with min v.dist
    Add v to R
    for each edge (v, z) in E
        if z.dist > v.dist + w(v, z)
            z.dist := v.dist + w(v, z)
```

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):

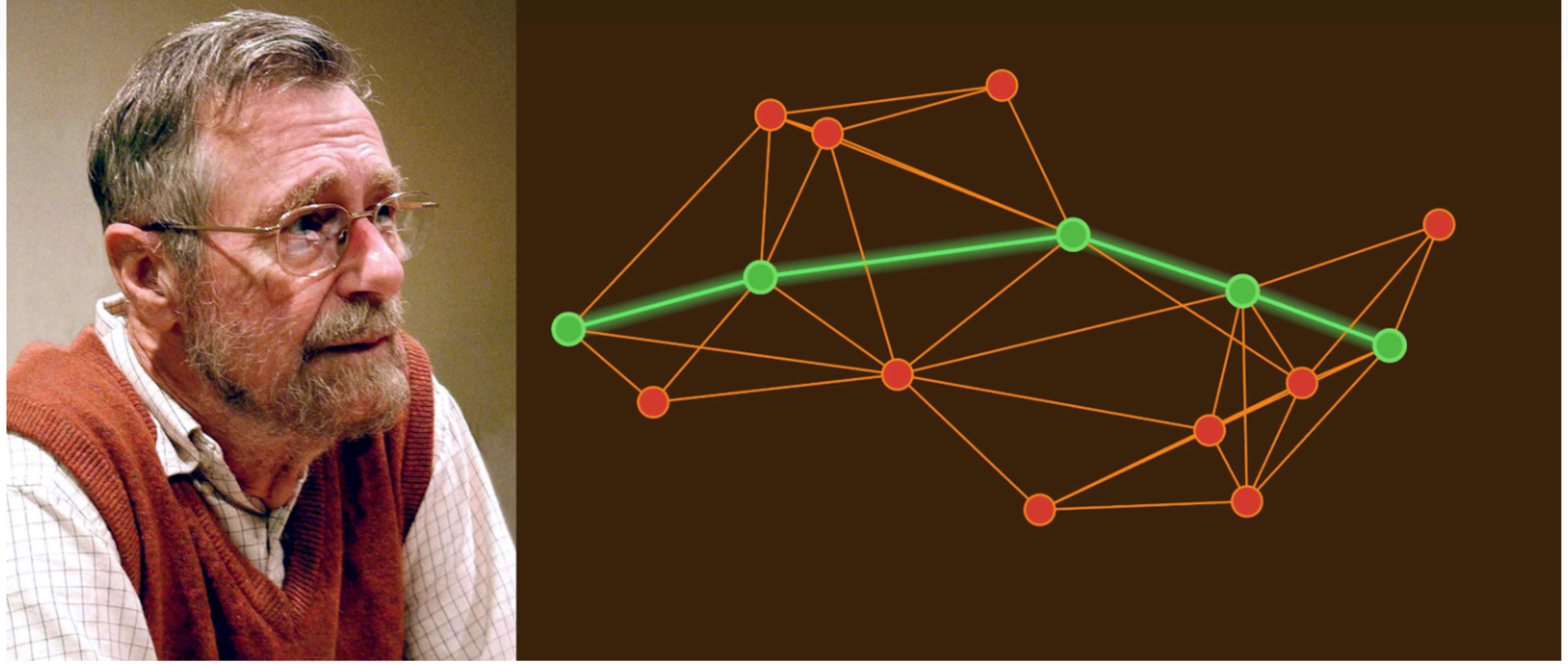
```
PriorityQueue Q
Q.add(x)
while !Q.empty()
    u := Q.remove()
    if !u.visited
        u.visited := True
        for each edge (u, v) in E
            if !v.visited and …
                Q.update(v, …)
```

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)
```



本科经典算法Dijkstra，被证明是普遍最优了：最坏情况性能也最优！

量子位
2020 年度新知答主

> 金磊 发自 凹非寺
> 量子位 | 公众号 QbitAI

时隔近**70年**，那个用来解决**最短路径问题**的经典算法——**Dijkstra**，现在有了新突破：

> 被证明具有普遍最优性（Universal Optimality）。

什么意思？

这就意味着不论它面对多复杂的图结构，即便在**最坏情况下都能达到理论上的最优性能！**

而且这还是学术界**首次**将这一概念应用于任何序列算法。

*On the universal optimality of Dijkstra's algorithm

- What does "optimal" mean, exactly? The problem space:
  - Let G denote a graph of n nodes and m edges
  - Let $\mathscr{G}_{n,m}$ denote the set of all such G.
  - Let $\mathscr{W}_G$ denote all possible weight functions for a given G.
  - Let $\mathscr{A}$ denote all correct SSSP algorithms (when edge weights are positive)

*On the universal optimality of Dijkstra's algorithm

- Algorithm $A \in \mathscr{A}$ is **existentially optimal** if: The usual definition
  - $\forall n, m: \max_{G \in \mathscr{G}_{n,m}, w \in \mathscr{W}_G} A(G, w) \leq O(1) \cdot \min_{A^*_{n,m} \in \mathscr{A}} \left(\max_{G \in \mathscr{G}_{n,m}, w \in \mathscr{W}_G} A^*_{n,m}(G, w) \right)$
- Algorithm $A \in \mathscr{A}$ is **instance optimal** if: Extremely hard to achieve
  - $\forall n, m, \forall G \in \mathscr{G}_{n,m}, \forall w \in \mathscr{W}_G : A(G, w) \leq O(1) \cdot \min_{A^*_{n,m} \in \mathscr{A}} A^*_{n,m}(G, w)$
- Algorithm $A \in \mathscr{A}$ is **universally optimal** if: Something in between
  - $\forall n, m, \forall G \in \mathscr{G}_{n,m} : \max_{w \in \mathscr{W}_G} A(G, w) \leq O(1) \cdot \min_{A^*_{n,m} \in \mathscr{A}} \left(\max_{w \in \mathscr{W}_G} A^*_{n,m}(G, w) \right)$

Can we design an universally optimal algorithm for SSSP ?

Best paper of FOCS 2024

Universal Optimality of Dijkstra via Beyond-Worst-Case Heaps*

Bernhard Haeupler
INSAIT, Sofia University
"St. Kliment Ohridski"
& ETH Zurich

Richard Hladík
INSAIT, Sofia University
"St. Kliment Ohridski"
& ETH Zurich

Václav Rozhoň
INSAIT, Sofia University
"St. Kliment Ohridski"

Robert E. Tarjan
Princeton University

Jakub Tětek
INSAIT, Sofia University
"St. Kliment Ohridski"

Abstract

This paper proves that Dijkstra's shortest-path algorithm is universally optimal in both its running time and number of comparisons when combined with a sufficiently efficient heap data structure.

Universal optimality is a powerful beyond-worst-case performance guarantee for graph algorithms that informally states that a single algorithm performs as well as possible for every single graph topology. We give the first application of this notion to any sequential algorithm.

We design a new heap data structure with a working-set property guaranteeing that the heap takes advantage of locality in heap operations. Our heap matches the optimal (worst-case) bounds of Fibonacci heaps but also provides the beyond-worst-case guarantee that the cost of extracting the minimum element is merely logarithmic in the number of elements inserted after it instead of logarithmic in the number of all elements in the heap. This makes the extraction of recently added elements cheaper.

We prove that our working-set property guarantees universal optimality for the problem of ordering vertices by their distance from the source vertex: The sequence of heap operations generated by any run of Dijkstra's algorithm on a fixed graph possesses enough locality that one can couple the number of comparisons performed by any heap with our working-set bound to the minimum number of comparisons required to solve the distance ordering problem on this graph for a worst-case choice of arc lengths.

*Universal optimality of Dijkstra's Algorithm

- **Distance Ordering Problem(DOP)**: Given a graph G and a source node $s \in V(G)$, output an ordering of $V(G)$ in increasing order of their distances from s.
  - Difficulty of SSSP $\geq$ Difficulty of DOP

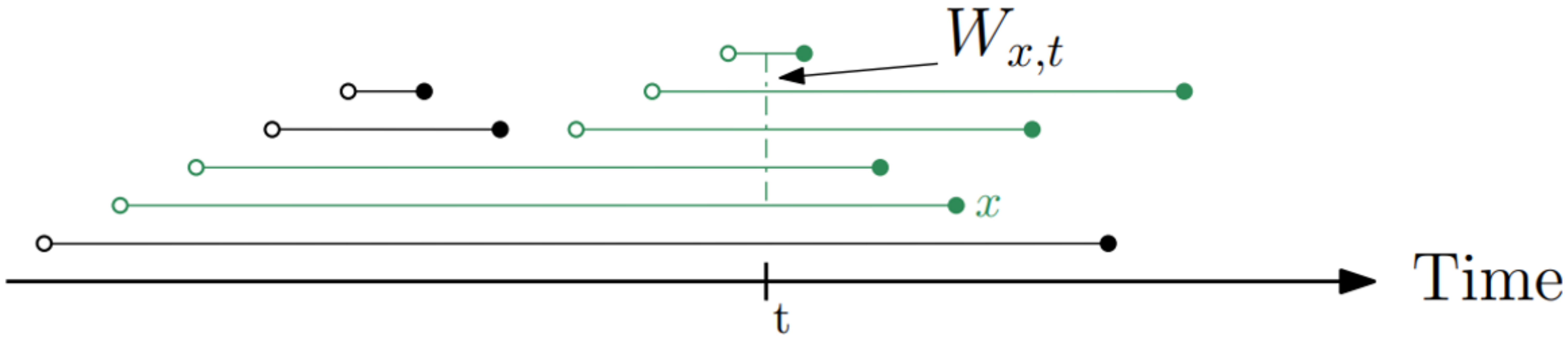
Lower bound: Dijkstra's algorithm implemented with any priority queue with the **working set property** is a **universally optimal algorithm for DO** in comparison-addition model, in terms of running time.

Corollary Dijkstra's algorithm implemented with any priority queue with the **working set property** is a **universally optimal algorithm for SSSP** in comparison-addition model, in terms of running time

Upper bound: There are priority queue implementations with working set property.

*Working set

- **Working set**: Consider any priority queue Q supporting `Insert` and `ExtractMin`. For any $x \in Q$, define its working set W_x in the following way:
  - For any time t between the insertion and extraction of x, define $W_{x,t}$ as the set of elements inserted after x but are still in Q at time t.
  - Let t_0 be an arbitrary time that maximize $|W_{x,t}|$, then $W_x = W_{x,t_0}$



*Working set

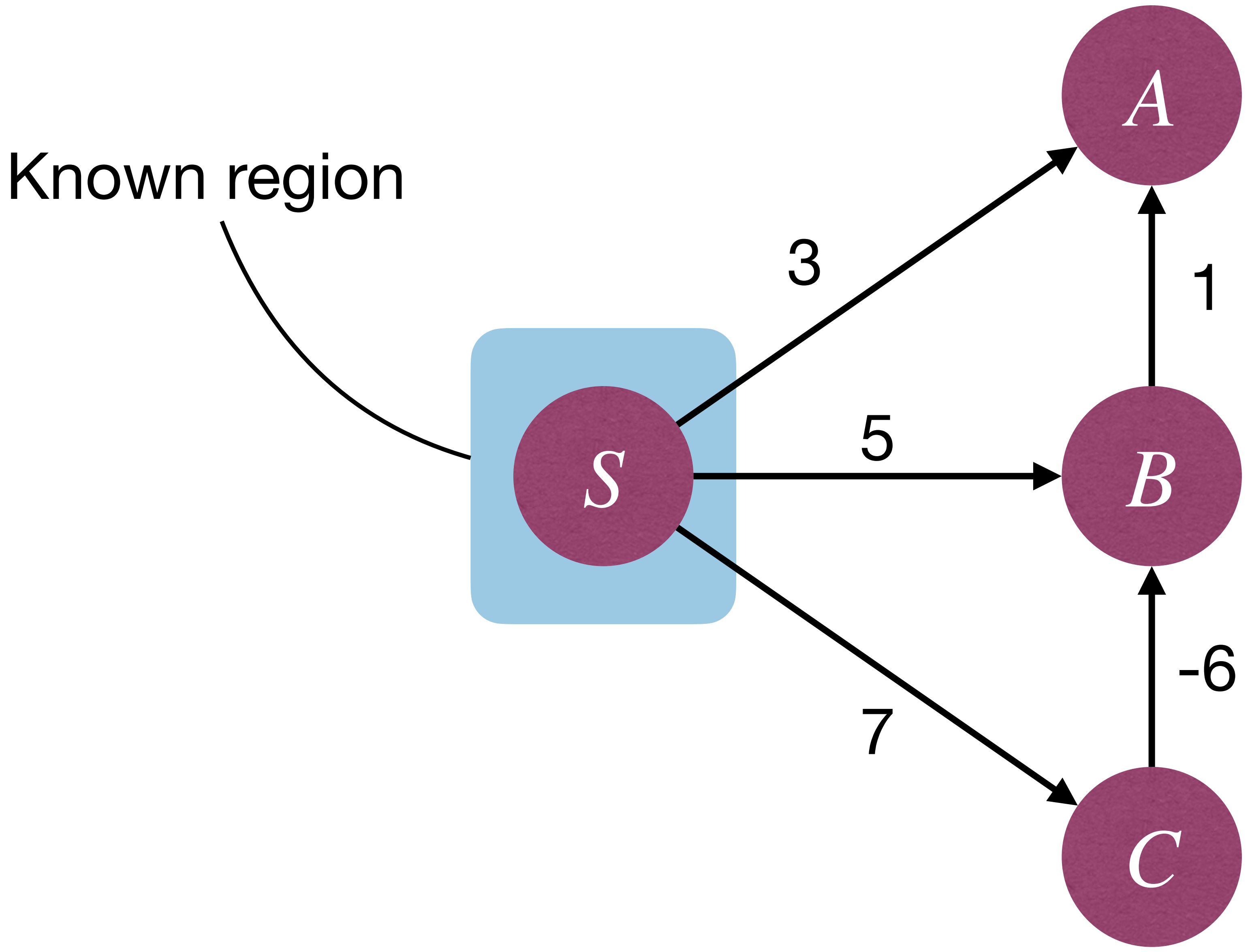
- **Priority Queue with Working Set Property:** A data structure is a priority queue with the working set property if the amortized runtime of its supported operations are:
  - $O(1)$ for `Insert`, `ExtractMin` and `DecreaseKey`.
  - $O(1 + |\log W_x|)$ for (W_x is the working set of the extracted element)

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!

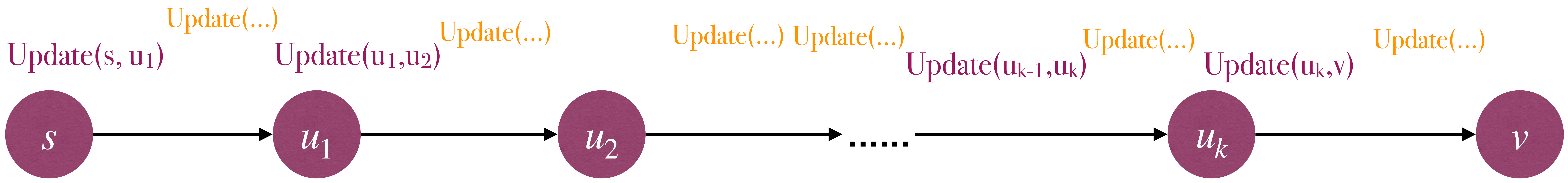
SSSP in graphs with negative weights

- 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.

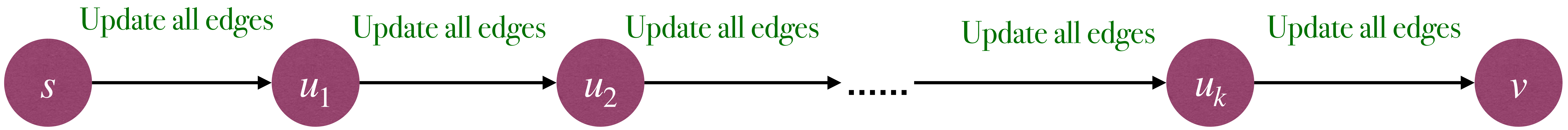
SSSP in graphs with negative weights

- `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 $v.dist$.

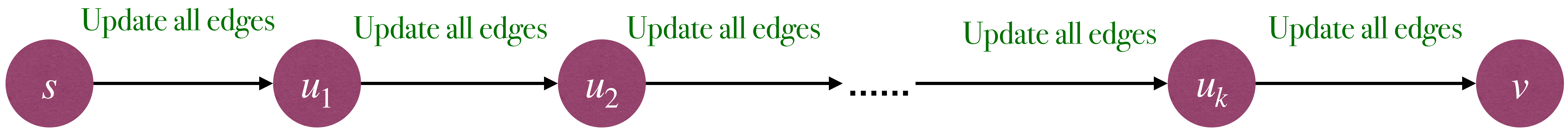
SSSP in graphs with negative weights



- 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` all 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.

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

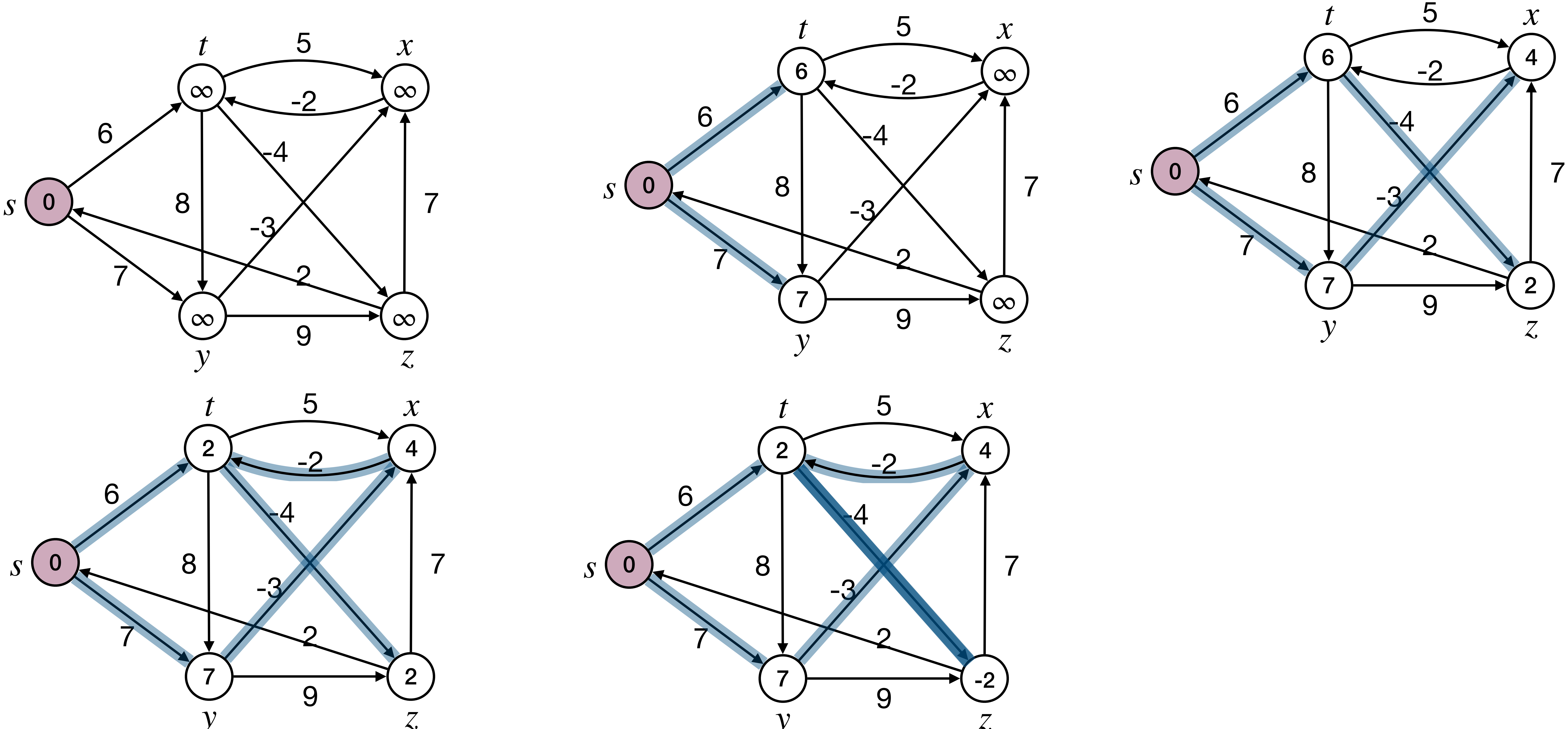
        if $v.dist > u.dist + w(u, v)$

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

            $v.parent := u$

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"
```

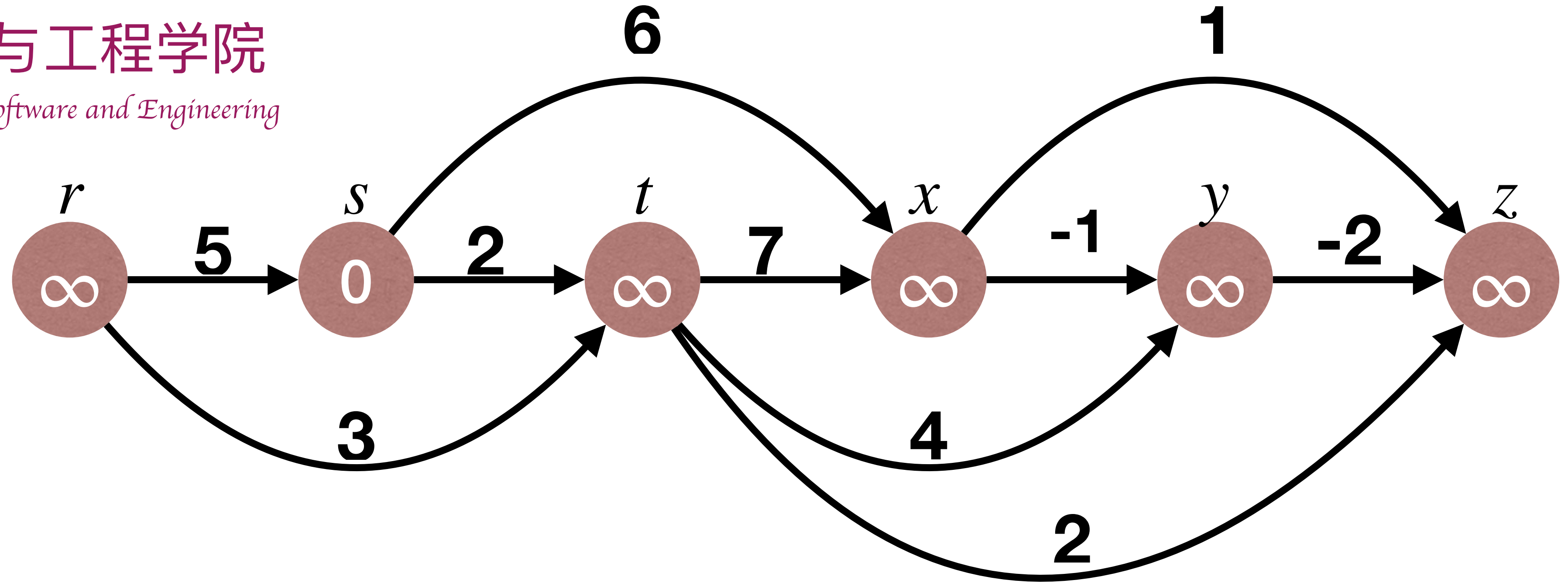
SSSP in DAG (with negative weights)

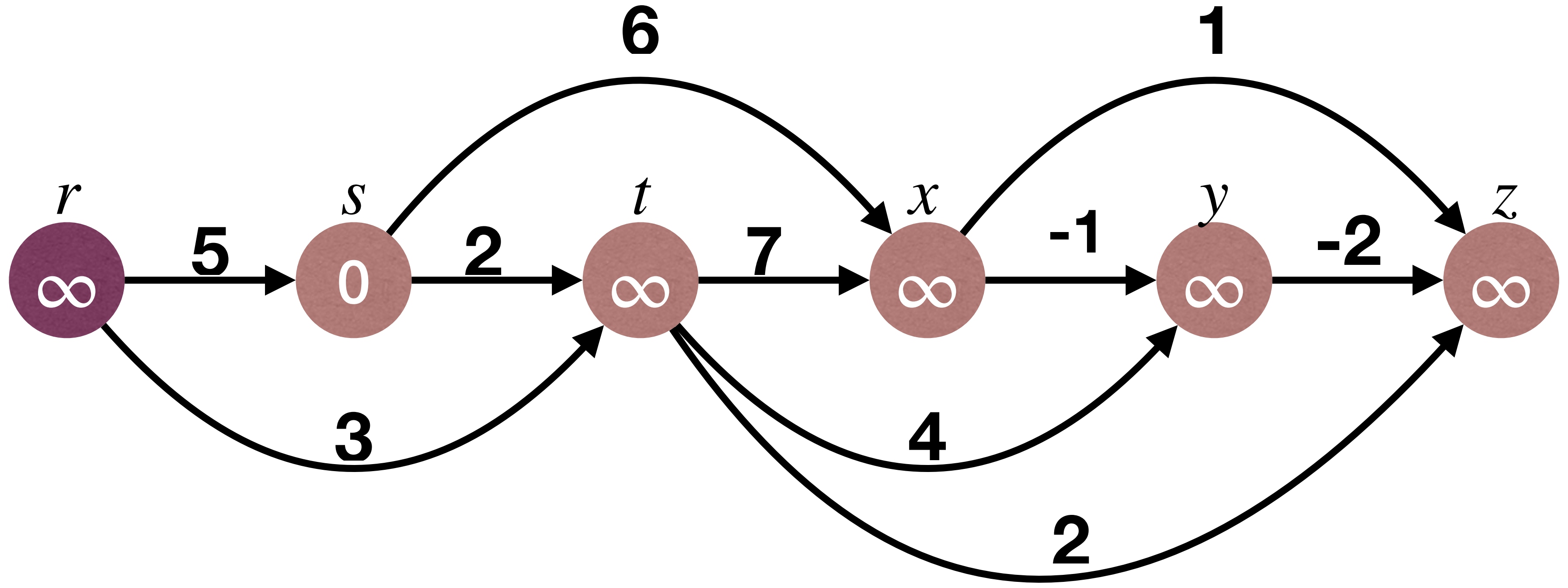
$O(m+n)$ time complexity

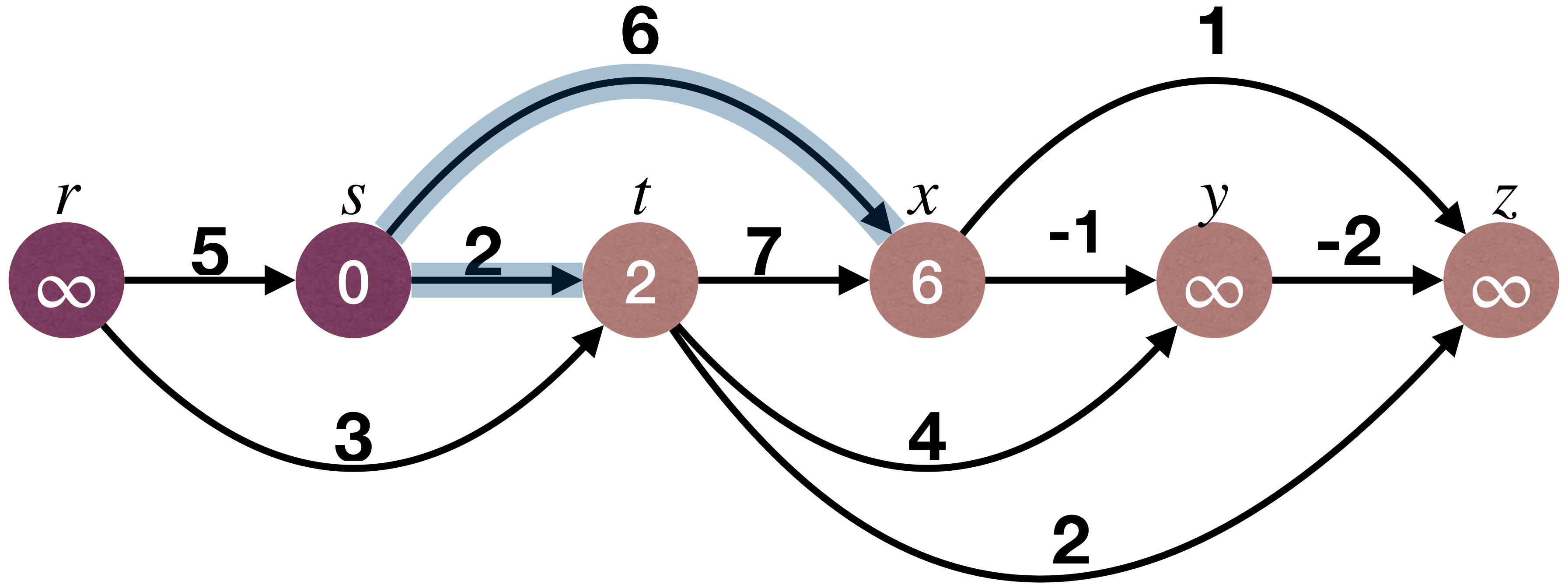
- Bellman-Ford still works, but we can be more efficient!
- Core idea of Bellman-Ford: perform a sequence of `Update` that includes every shortest path as a subsequence.
- Observation: in DAG, every path, thus every shortest path, is a subsequence in the topological order.

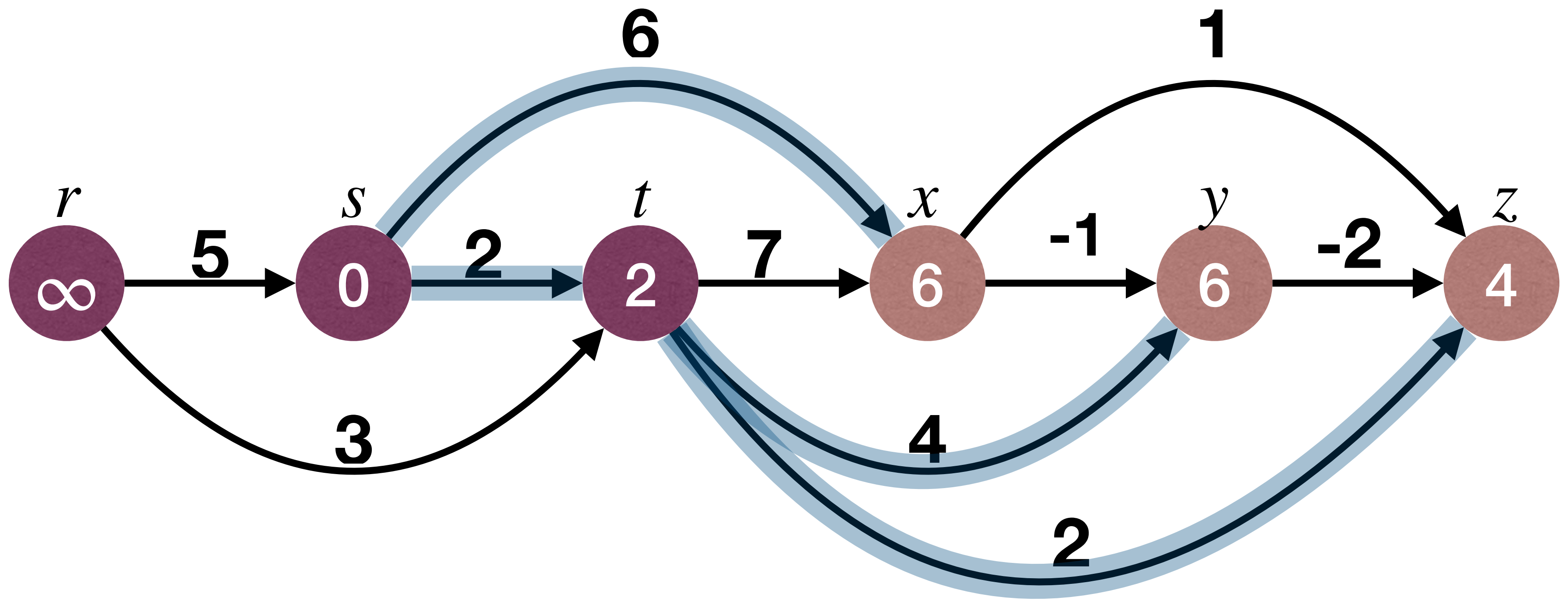
DAGSSSP(G,s):

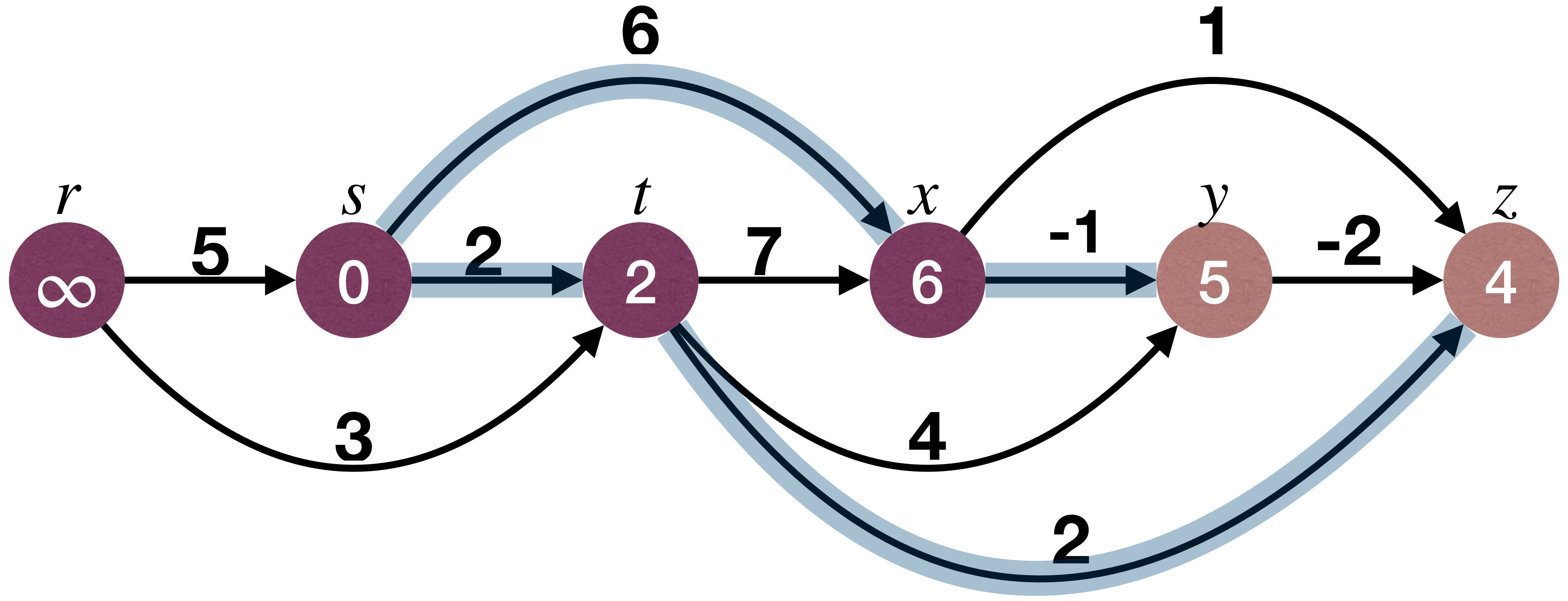
for **each** u **in** V
    $u.dist := INF, u.parent := NIL$
$s.dist := 0$
Run DFS to obtain topological order
for **each** *node* u **in** *topological order*
    for **each** *edge* (u, v) **in** E
        if $v.dist > u.dist + w(u, v)$
            $v.dist := u.dist + w(u, v)$
            $v.parent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

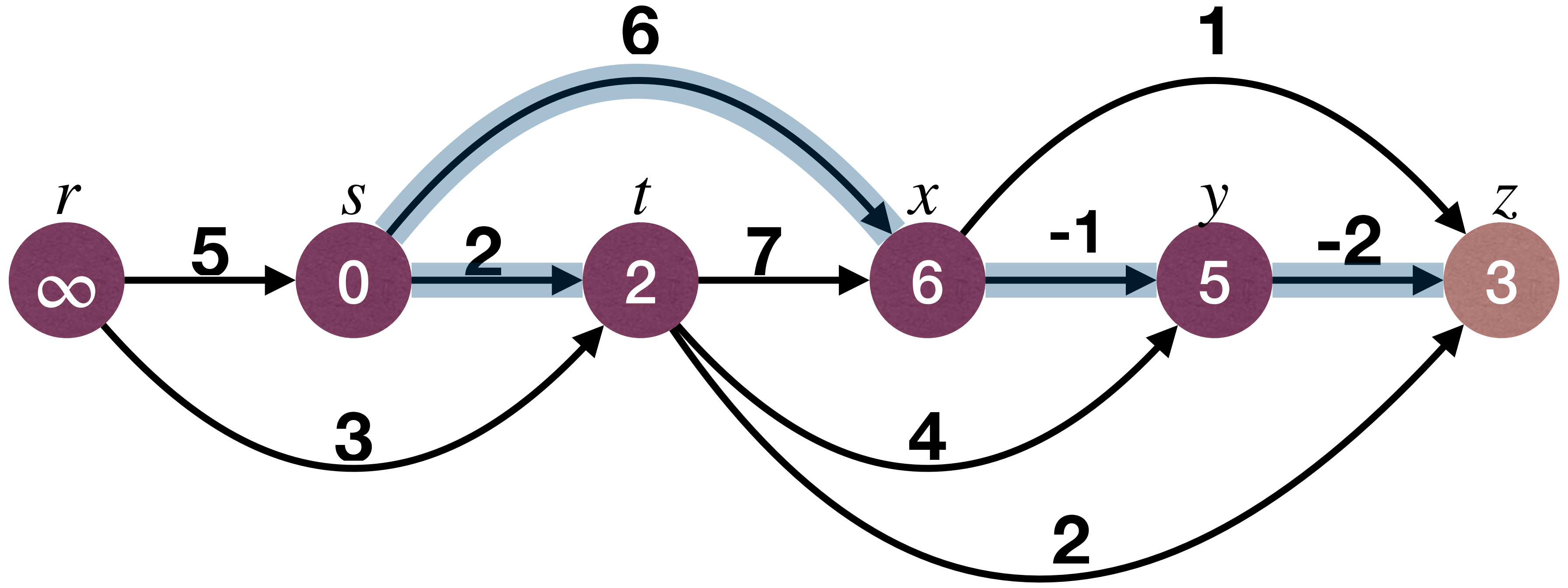

6
1
r s t x y z
5 2 7 -1 -2
∞ 0 ∞ ∞ ∞ ∞
3
4
2

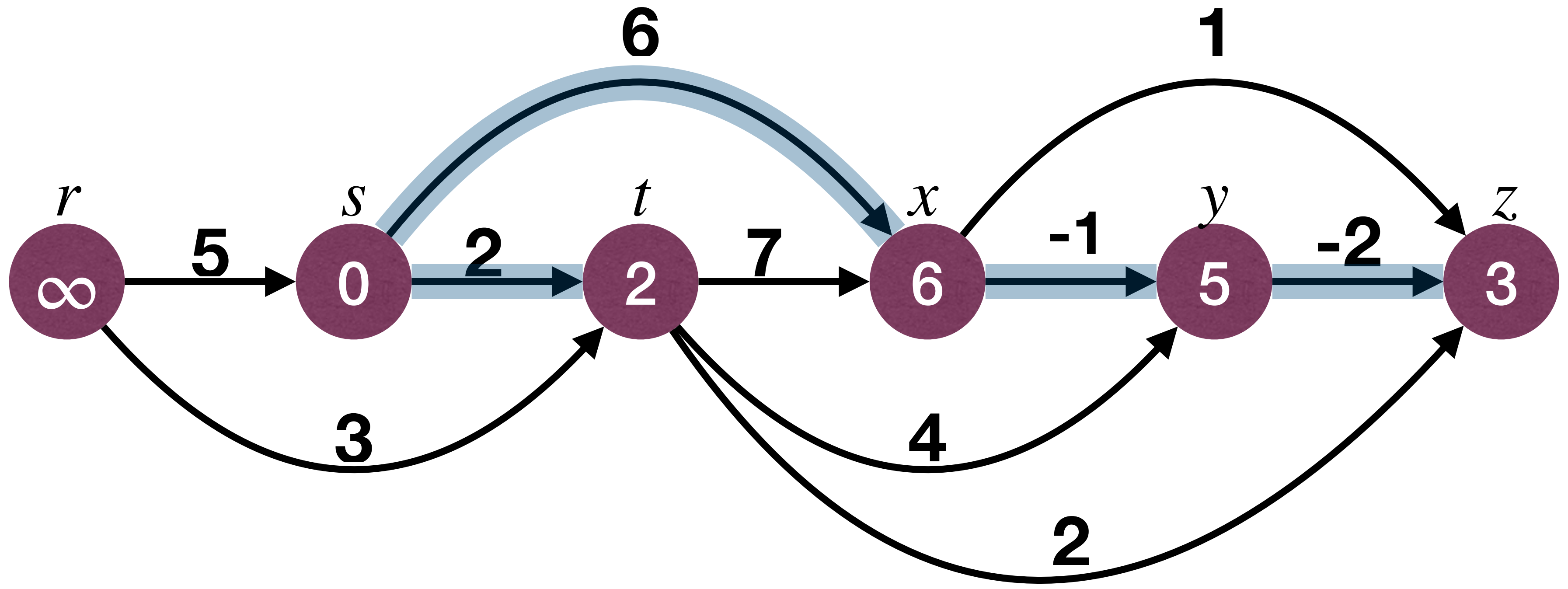

6
1
r s t x y z
5 2 7 -1 -2
∞ 0 ∞ ∞ ∞ ∞
3
4
2


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

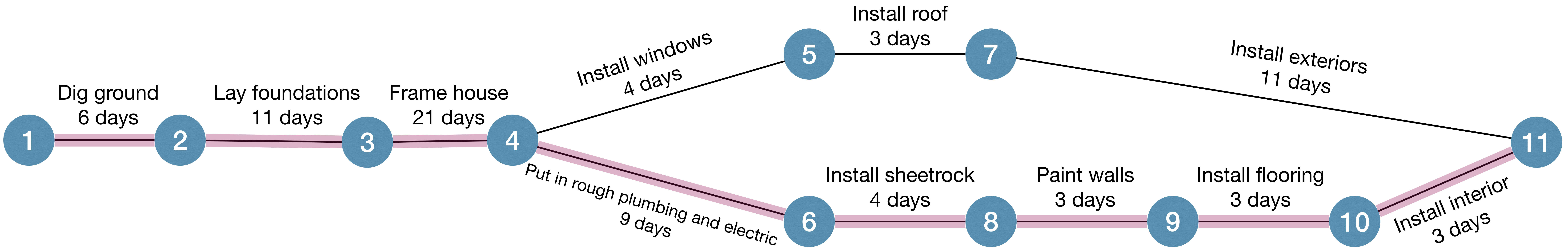

6
1
r s t x y z
5 2 7 -1 -2
∞ 0 2 6 5 4
3
4
2


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

Pathfinding Framework

GreedyBFS(G, s, t):

```
for each node u in V
    u.metric := INFINITY
s.metric:= est_to_goal(s,t)
Build priority queue Q based on metric
while !Q.empty()
    u := Q.ExtractMin()
    for each edge (u,v) in E
        new_metric := est_to_goal(v,t)
        if v ∉ Q or new_metric < v.metric
            v.metric := new_metric
            v.parent := u
            Q.AddorUpdate(v)
```

Dijkstra(G, s, t):

```
for each node u in V
    u.metric := INFINITY
s.metric := est_to_source(s,s) := 0
Build priority queue Q based on metric
while !Q.empty()
    u := Q.ExtractMin()
    for each edge (u,v) in E
        new_metric := update_est_to_source(v, u, s)
                   := min(v.metric, u.metric + dist(u,v))
                   := min(v.metric, dist(s, u)+ dist(u,v))
        if v ∉ Q or new_metric < v.metric
            v.metric := new_metric
            v.parent := u
            Q.AddorUpdate(v)
```

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 ∉ 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)*

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** $v.est\_to\_s > u.est\_to\_s + dist(u, v)$
      $v.est\_to\_s := u.est\_to\_s + dist(u, v)$
      $v.metric := v.est\_to\_s + v.est\_to\_t$
      $v.parent := u$
      $Q.Add(v)$

f : metric, h: estimate to goal

The A* algorithm

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.

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