Georgia Tech

CSE 6140/ CX 4140

Computational Science and Engineering

ALGORITHMS

Greedy Algorithms - 4

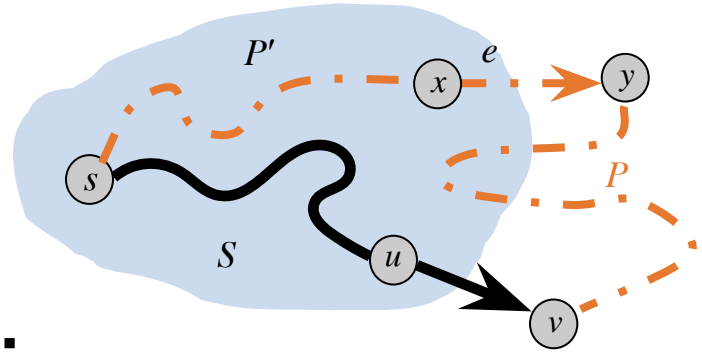
Instructor: Xiuwei Zhang

School of Computational Science and Engineering

Based on slides by Prof. Ümit V. Çatalyürek

Dijkstra's algorithm: proof of correctness (greedy stays ahead)

- Invariant. For each node $u \in S$: $d[u]$ = length of a shortest $s \rightsquigarrow u$ path.
- Pf. [by induction on $|S|$]
- Base case: $|S| = 1$ is easy since $S = \{ s \}$ and $d[s] = 0$.
- Inductive hypothesis: Assume true for $|S| \geq 1$.
  - Let v be next node added to S, and let (u, v) be the final edge.
  - A shortest $s \rightsquigarrow u$ path plus (u, v) is an $s \rightsquigarrow v$ path of length $\pi(v)$.
  - Consider any other $s \rightsquigarrow v$ path P. We show that it is no shorter than $\pi(v)$.
  - Let $e = (x, y)$ be the first edge in P that leaves S, and let P' be the subpath from s to x.
  - The length of P is already $\geq \pi(v)$ as soon as it reaches y:



$$\ell(P) \geq \ell(P') + \ell_e \geq d[x] + \ell_e \geq \pi(y) \geq \pi(v) \quad \blacksquare$$

- $\ell(P) \geq \ell(P') + \ell_e$: non-negative lengths
- $\ell(P') + \ell_e \geq d[x] + \ell_e$: inductive hypothesis
- $d[x] + \ell_e \geq \pi(y)$: definition of $\pi(y)$
- $\pi(y) \geq \pi(v)$: Dijkstra chose v instead of y

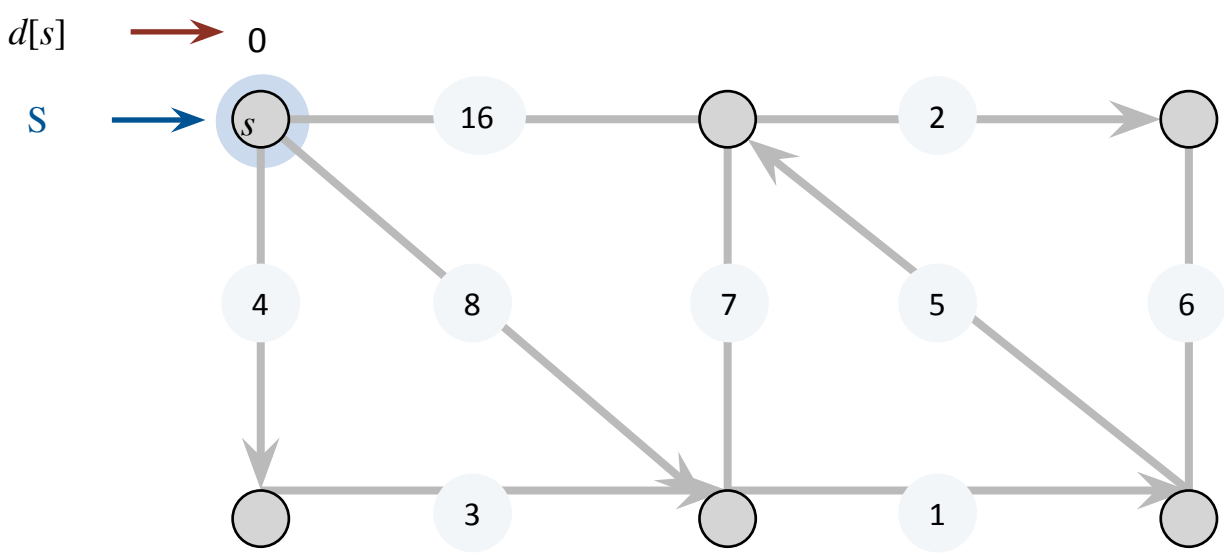
Dijkstra's algorithm demo

- Initialize $S \leftarrow \{ s \}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v) \,:\, u \in S} d[u] + \ell_e$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$

add v to S; set $d[v] \leftarrow \pi(v)$ and $pred[v] \leftarrow$ argmin.



Redundant calculation?

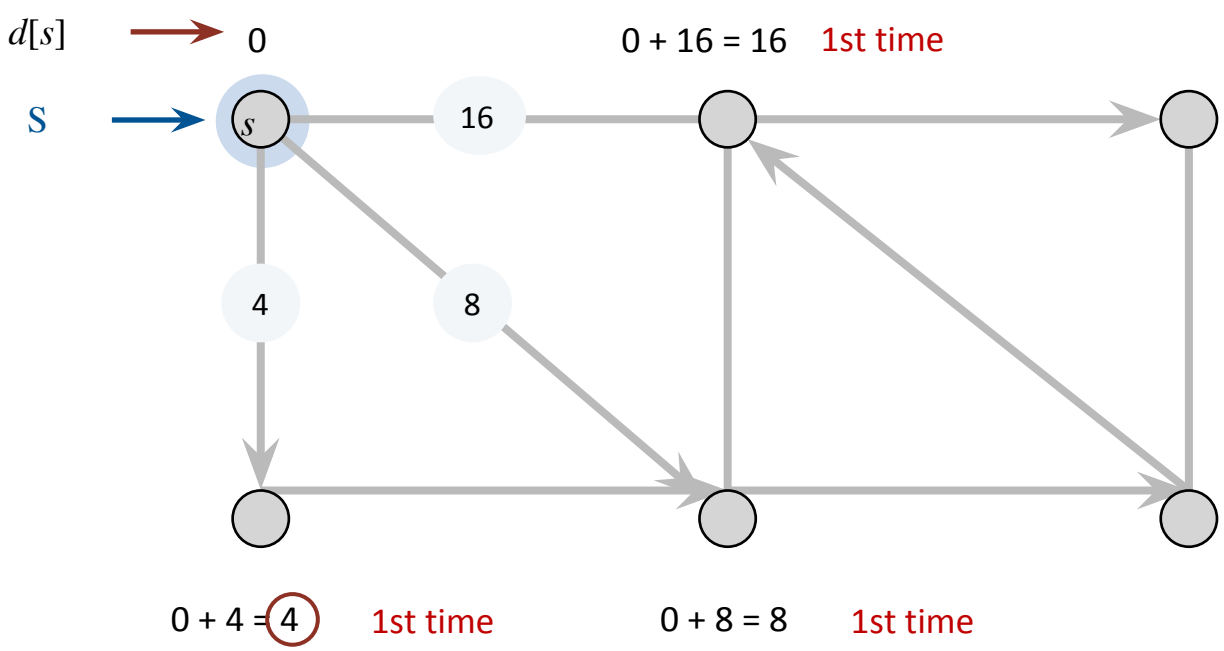
Dijkstra's algorithm demo

- Initialize $S \leftarrow \{ s \}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v) \,:\, u \in S} d[u] + \ell_e$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$

add v to S; set $d[v] \leftarrow \pi(v)$ and $pred[v] \leftarrow$ argmin.



Redundant calculation?

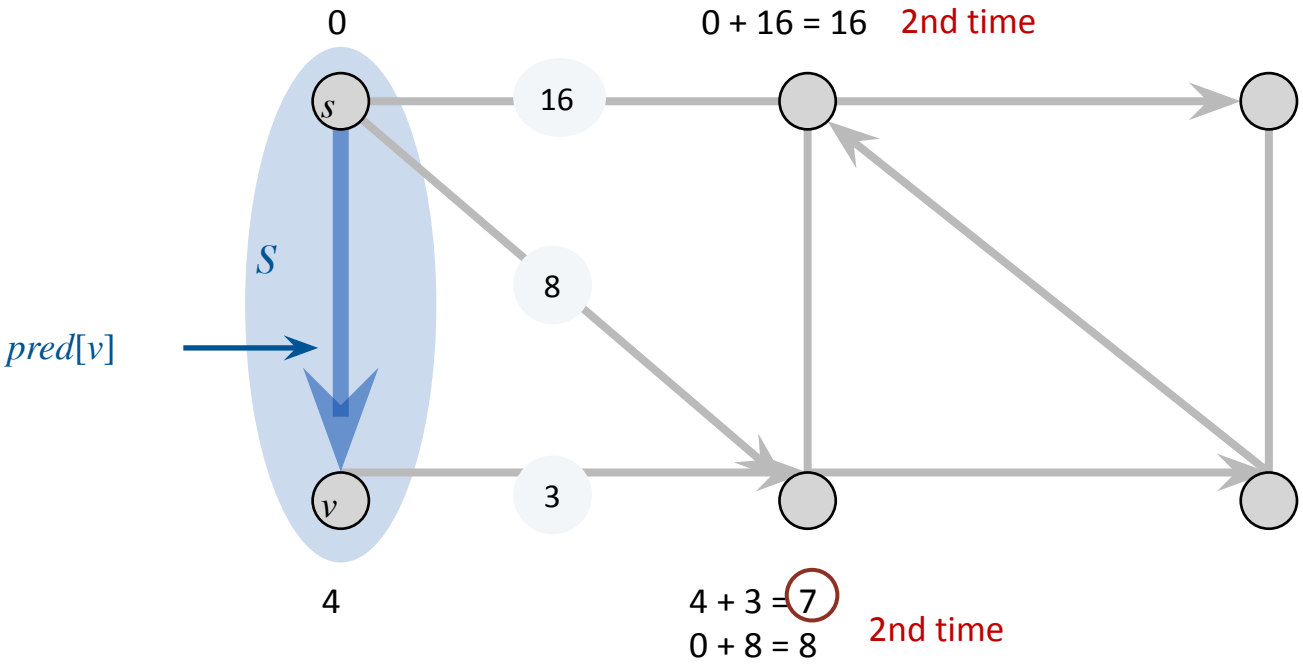
Dijkstra's algorithm demo

- Initialize $S \leftarrow \{ s \}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v) \,:\, u \in S} d[u] + \ell_e$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$

add v to S; set $d[v] \leftarrow \pi(v)$ and $pred[v] \leftarrow$ argmin.



Redundant calculation?

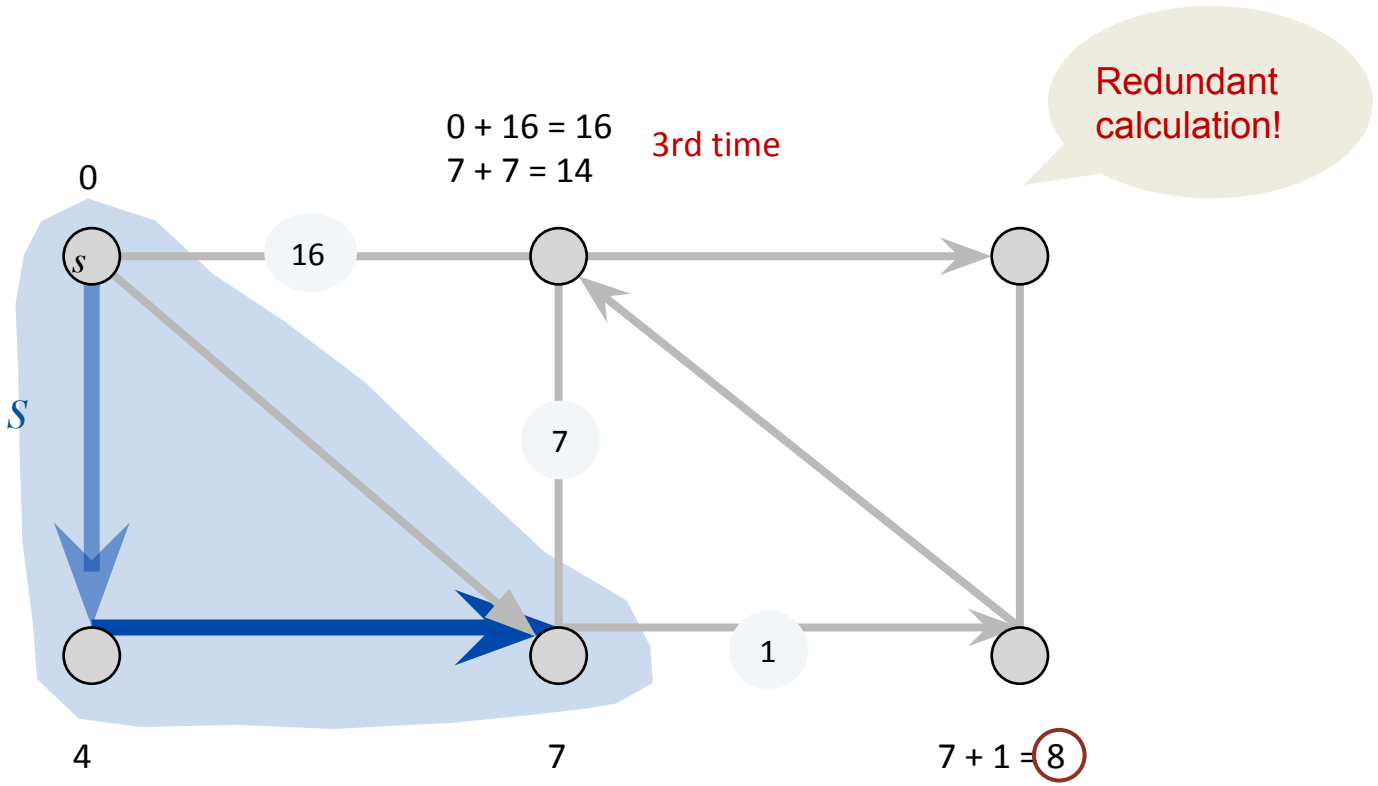
Dijkstra's algorithm demo

- Initialize $S \leftarrow \{ s \}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v) \,:\, u \in S} d[u] + \ell_e$$

the length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$

add v to S; set $d[v] \leftarrow \pi(v)$ and $pred[v] \leftarrow$ argmin.



Dijkstra's algorithm: efficient implementation

- Critical optimization 1. For each node $v \notin S$: explicitly maintain $\pi[v]$ instead of computing directly from definition

$$\pi(v) = \min_{e=(u,v) \,:\, u \in S} d[u] + \ell_e$$

  - For each $v \notin S$: $\pi(v)$ can only decrease (because set S increases).

  - More specifically, suppose u is added to S and there is an edge $e = (u, v)$ leaving u. Then, it suffices to update:

$$\pi[v] \leftarrow \min \{ \pi[v], \pi[u] + \ell_e \}$$

recall: for each $u \in S$,
$\pi[u] = d[u]$ = length of shortest $s \rightsquigarrow u$ path

- Critical optimization 2. Use a min-oriented priority queue (PQ) to choose an unexplored node that minimizes $\pi[v]$.

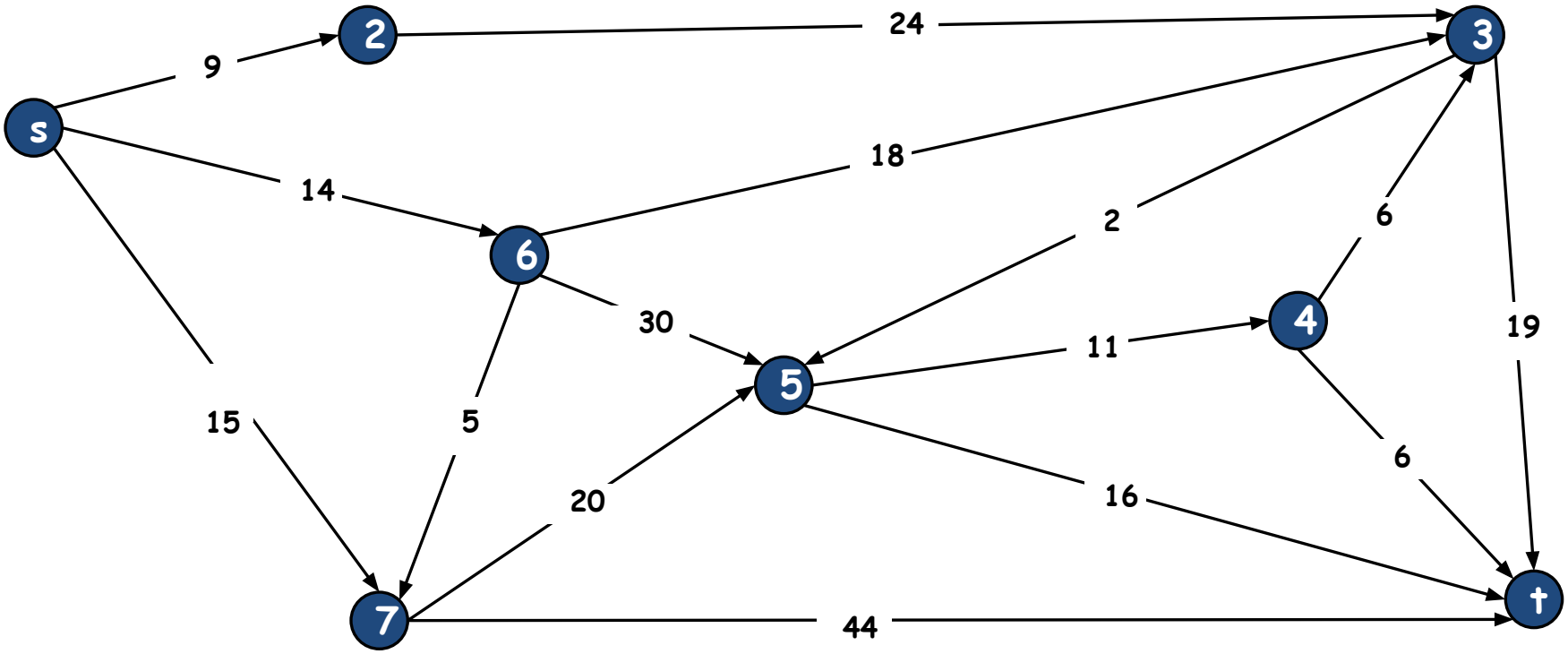
Dijkstra's algorithm: efficient implementation

- Implementation.
  - Algorithm maintains $\pi[v]$ for each node v.
  - Priority queue stores unexplored nodes, using $\pi[\,\cdot\,]$ as priorities.
  - Once u is deleted from the PQ, $\pi[u]$ = length of a shortest $s \rightsquigarrow u$ path.

```
DIJKSTRA (V, E, ℓ, s)

FOREACH v ≠ s : π[v] ← ∞, pred[v] ← null; π[s] ← 0.
Create an empty priority queue PQ.
FOREACH v ∈ V : INSERT(PQ, v, π[v]).
WHILE (IS-NOT-EMPTY(PQ))
    u ← DEL-MIN(PQ).
    FOREACH edge e = (u, v) ∈ E leaving u:
        IF (π[v] > π[u] + ℓ_e)
            π[v] ← π[u] + ℓ_e ; pred[v] ← e.
            DECREASE-KEY(PQ, v, π[v]).
```

Dijkstra's Shortest Path Algorithm

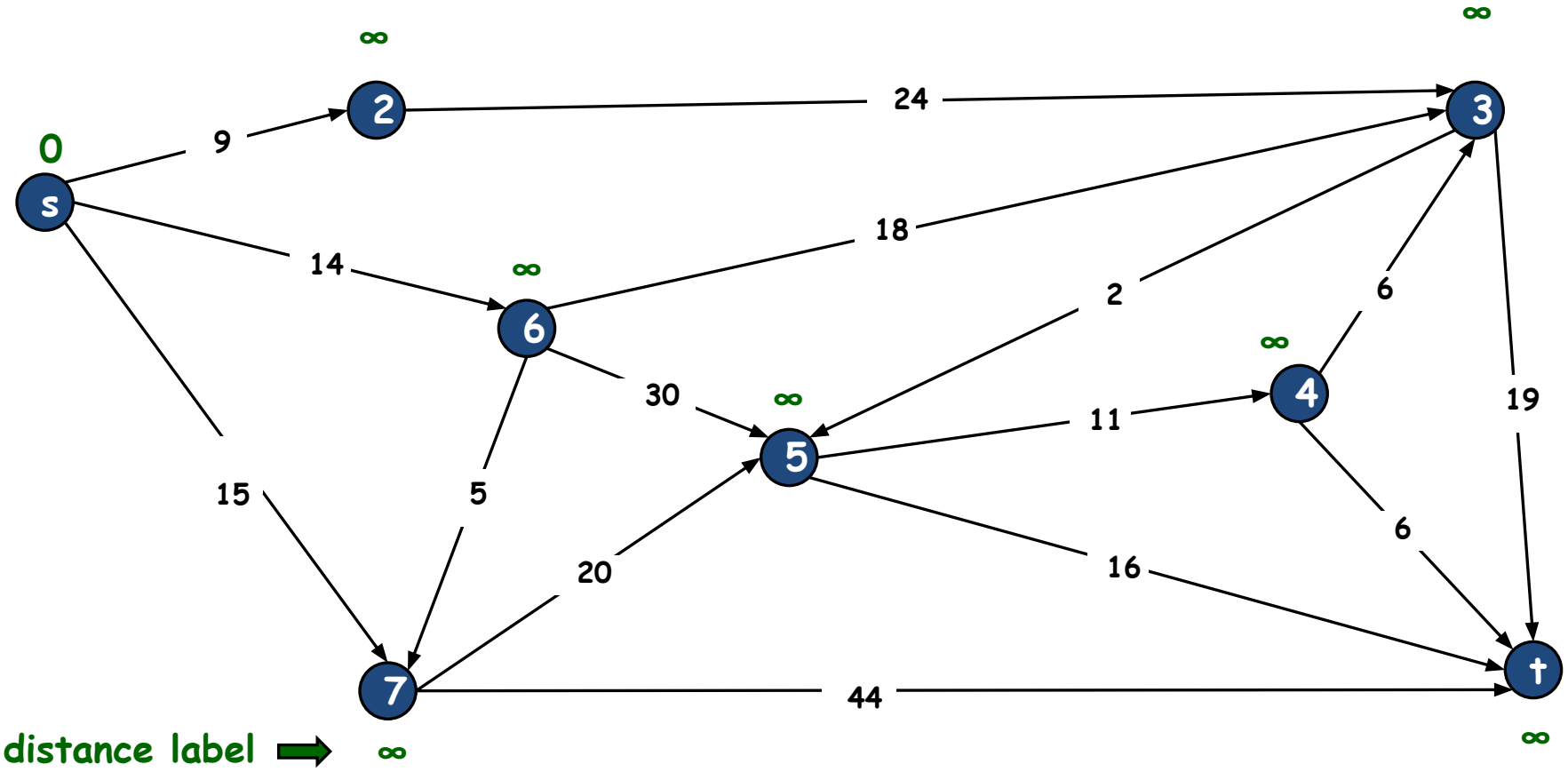
Find shortest path from s to t.



Dijkstra's Shortest Path Algorithm

S = { }

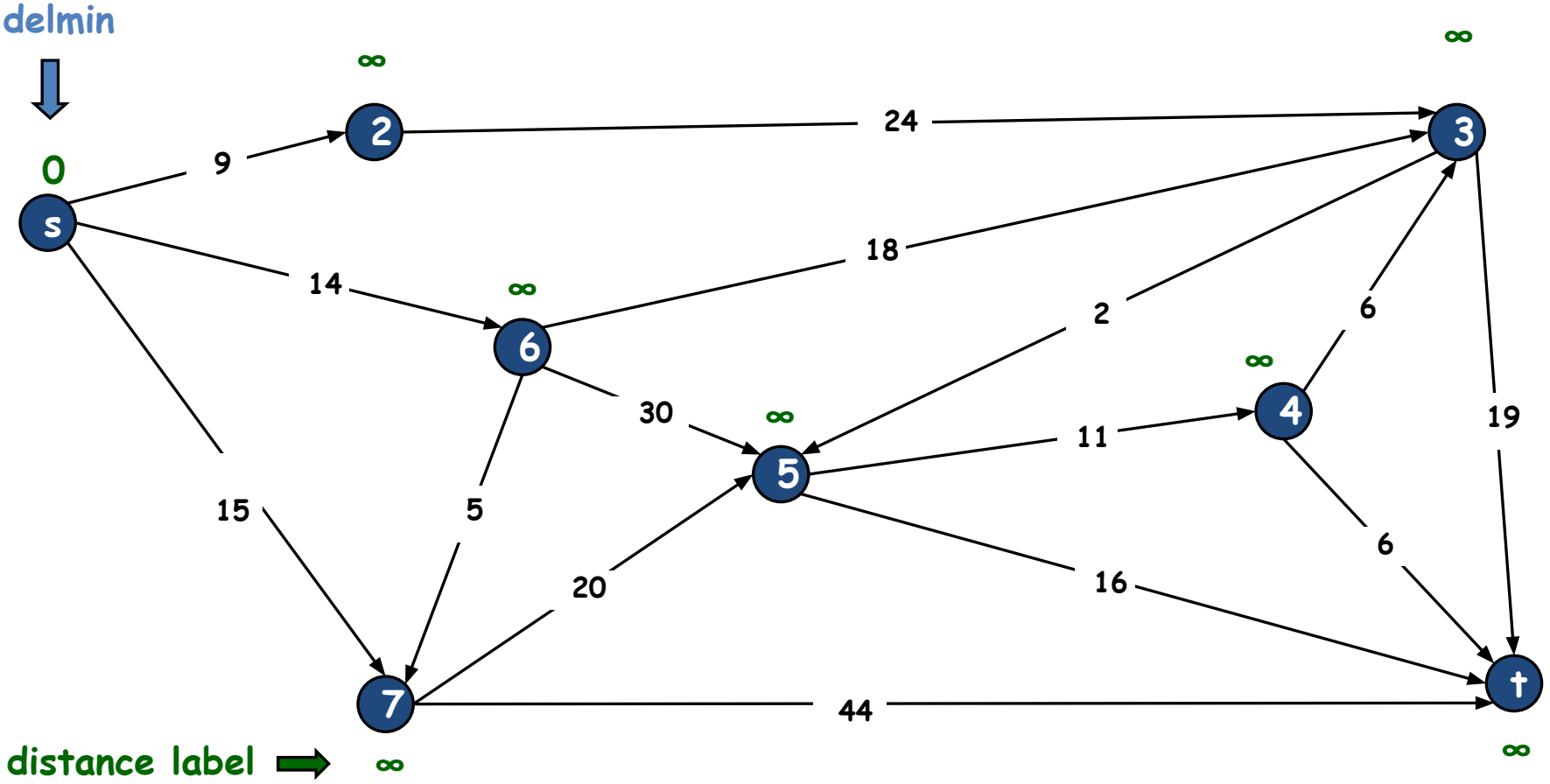
PQ = { s, 2, 3, 4, 5, 6, 7, t }



Dijkstra's Shortest Path Algorithm

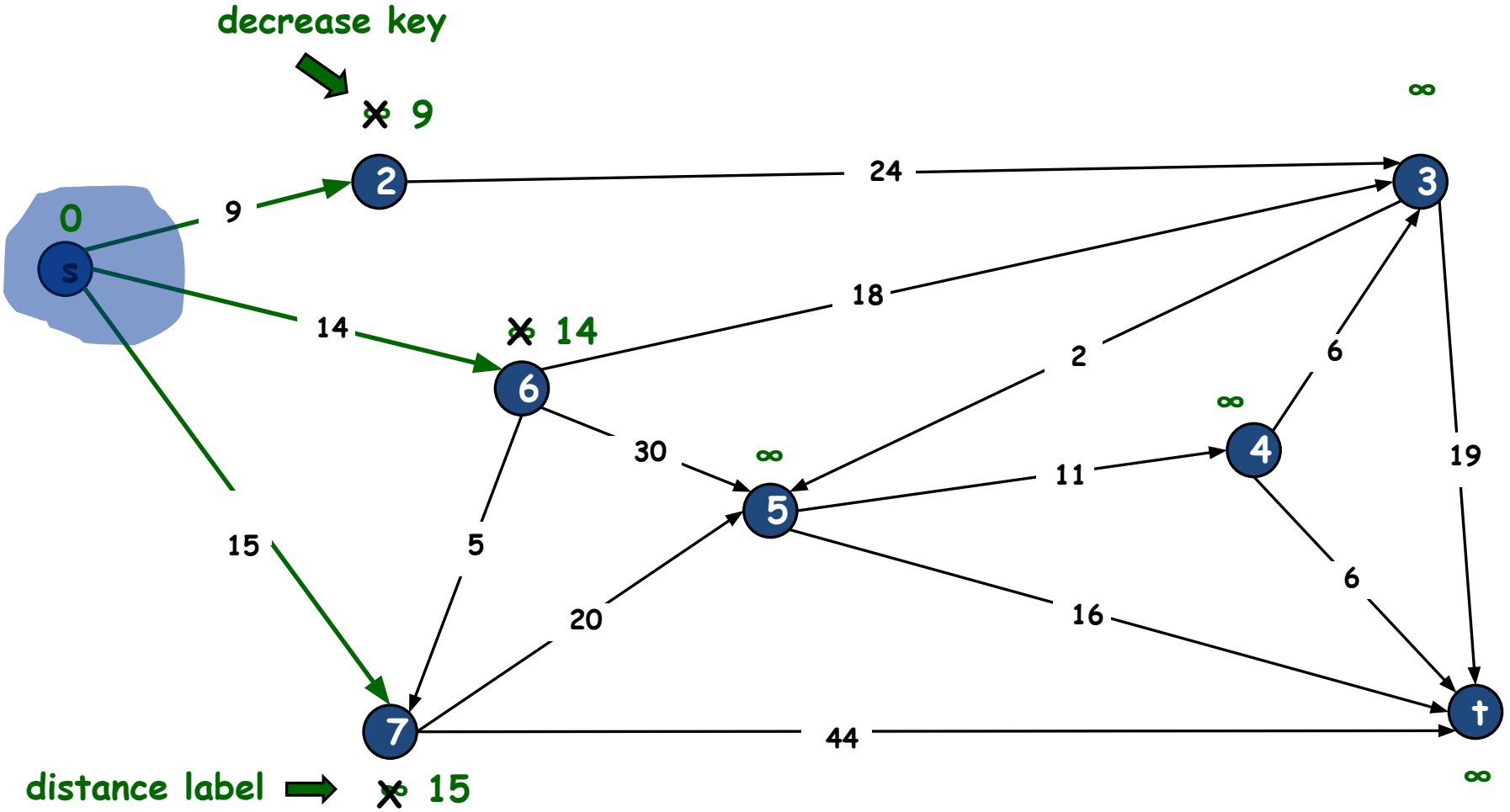
S = { }

PQ = { s, 2, 3, 4, 5, 6, 7, t }



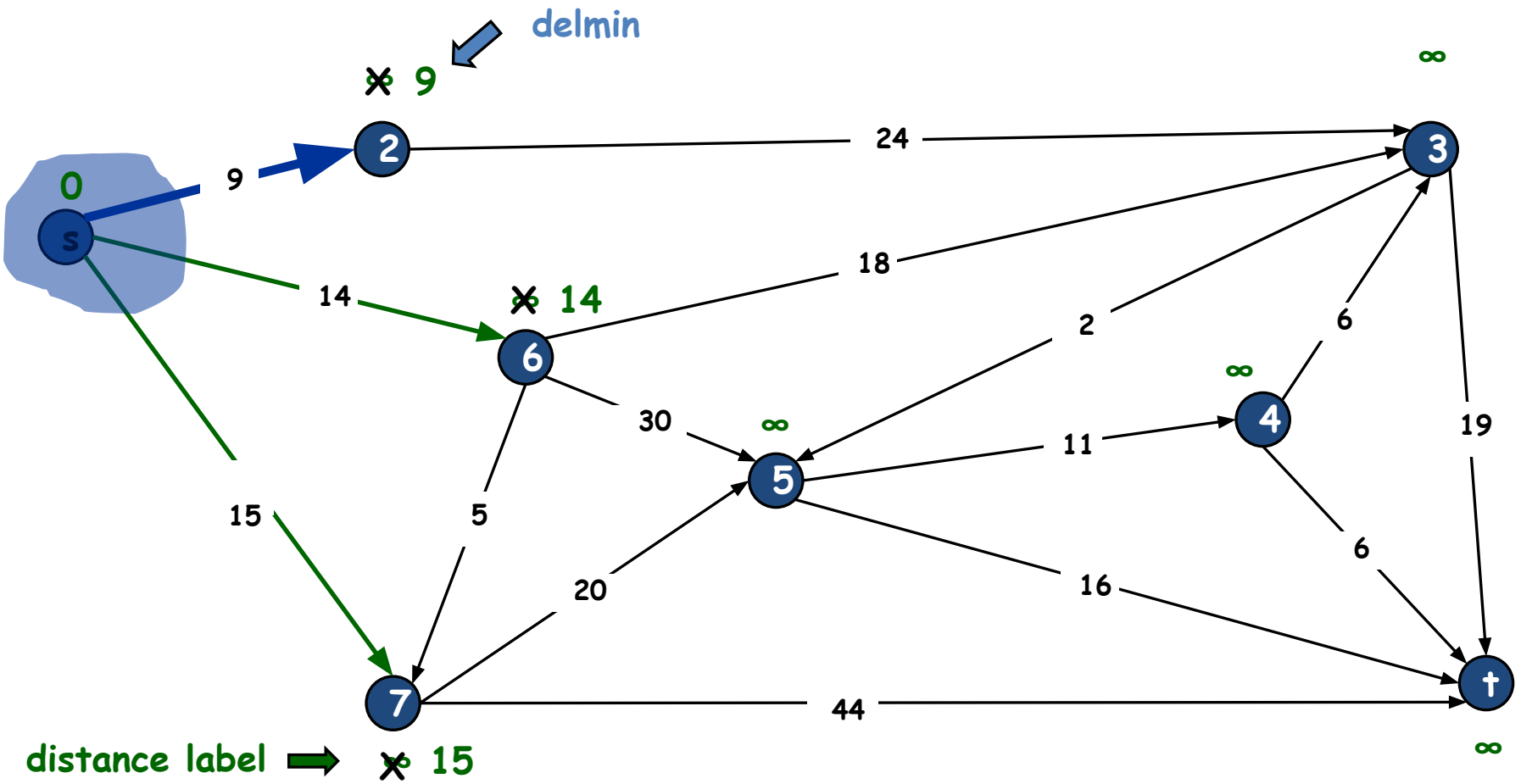
Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



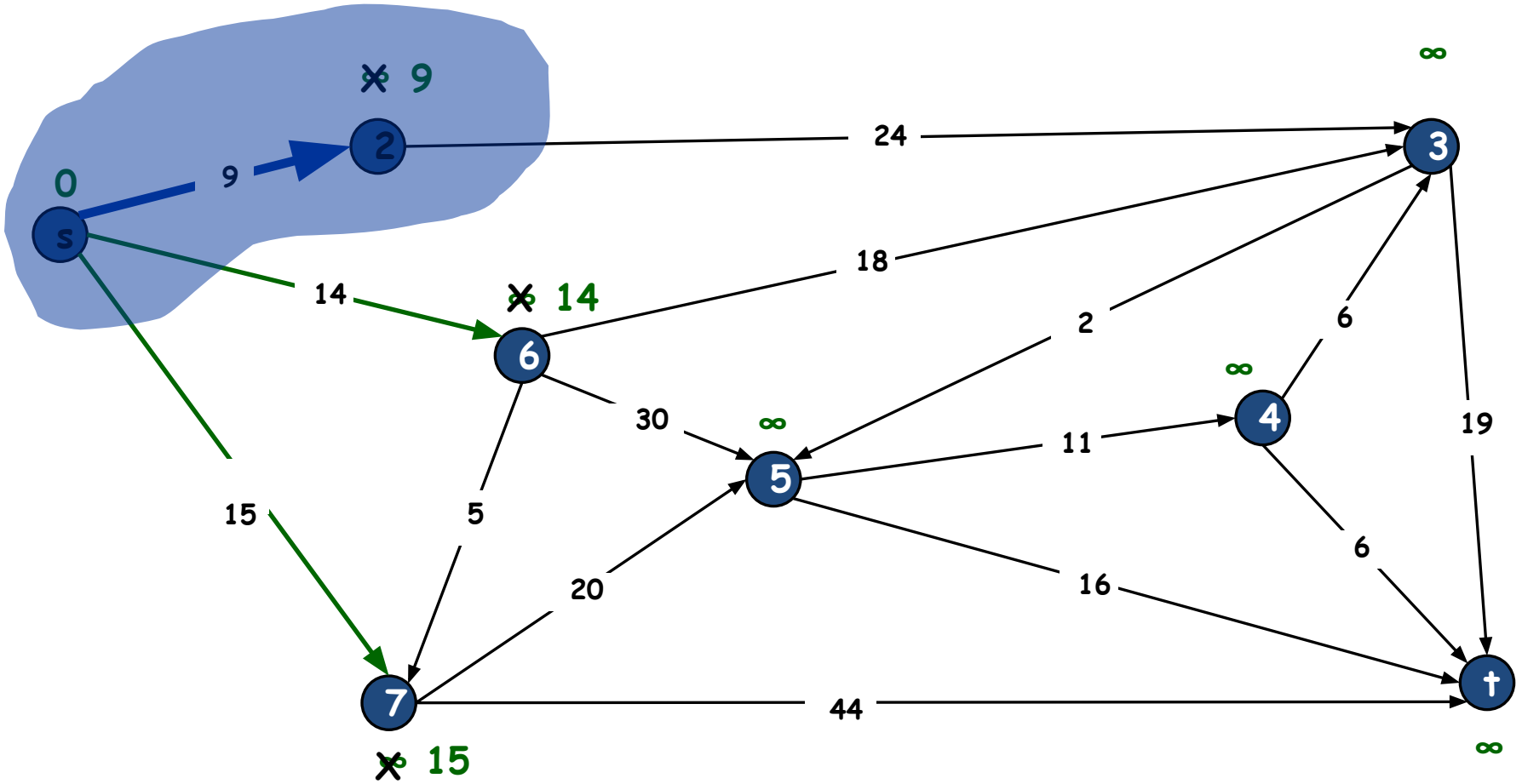
Dijkstra's Shortest Path Algorithm

S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



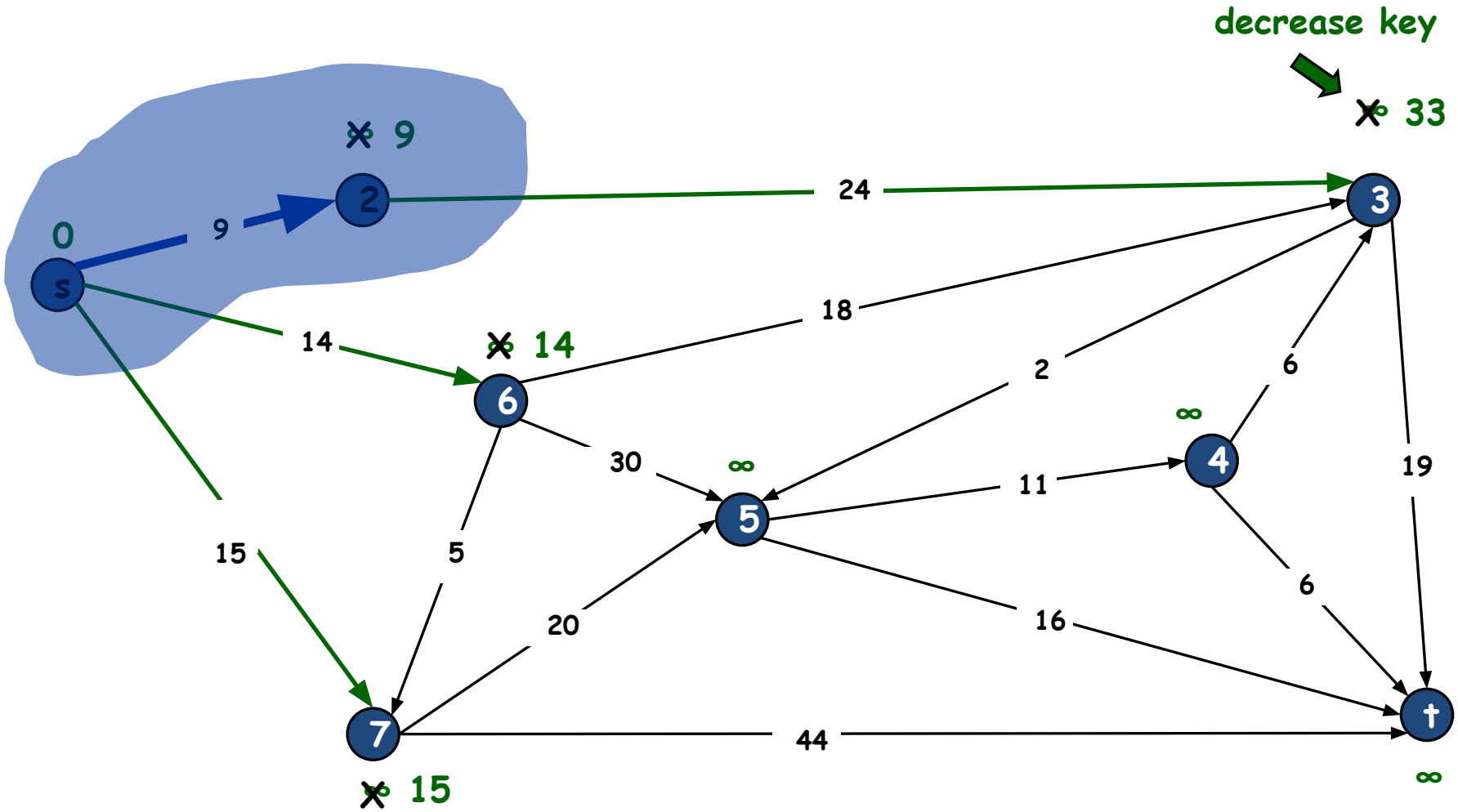
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



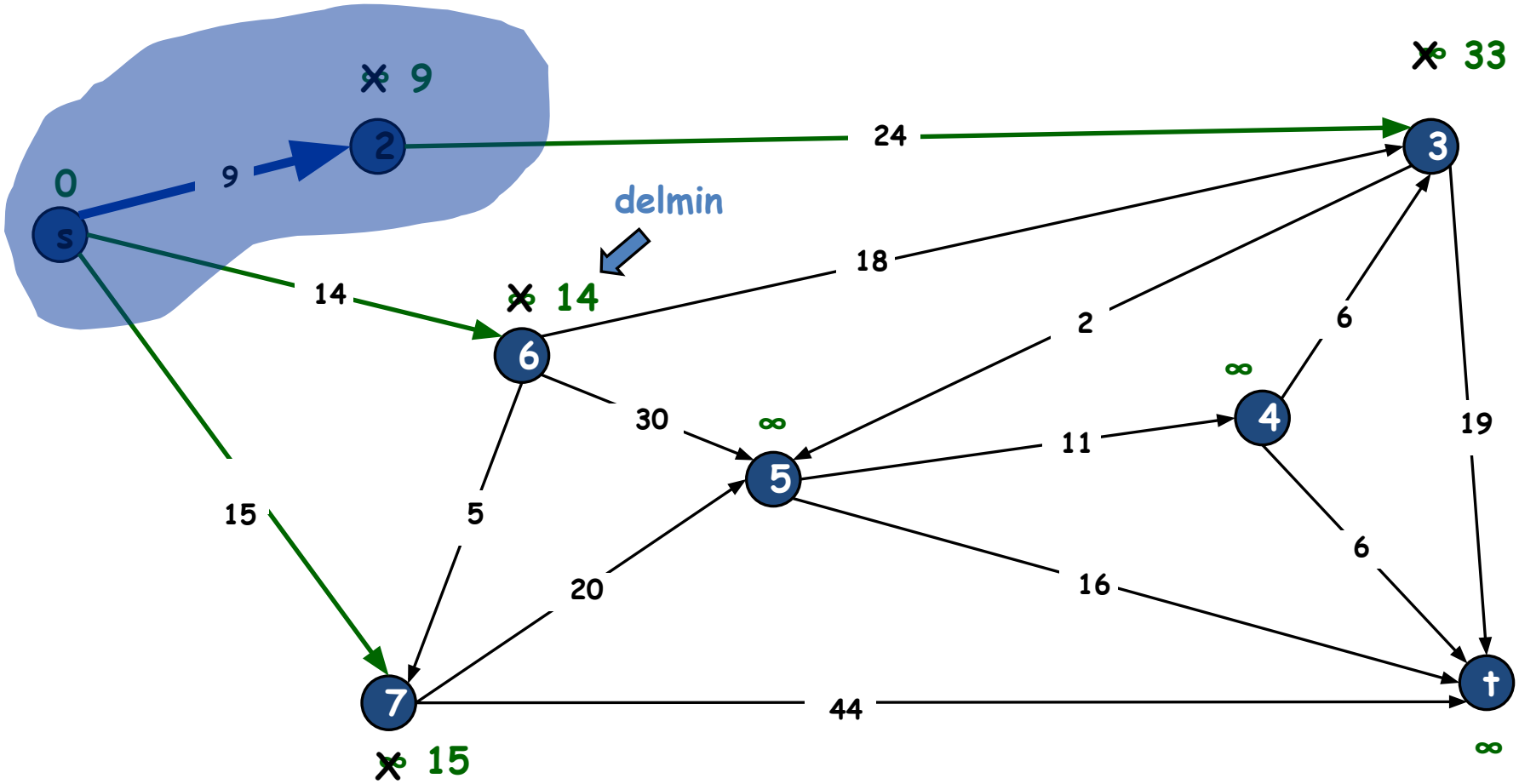
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



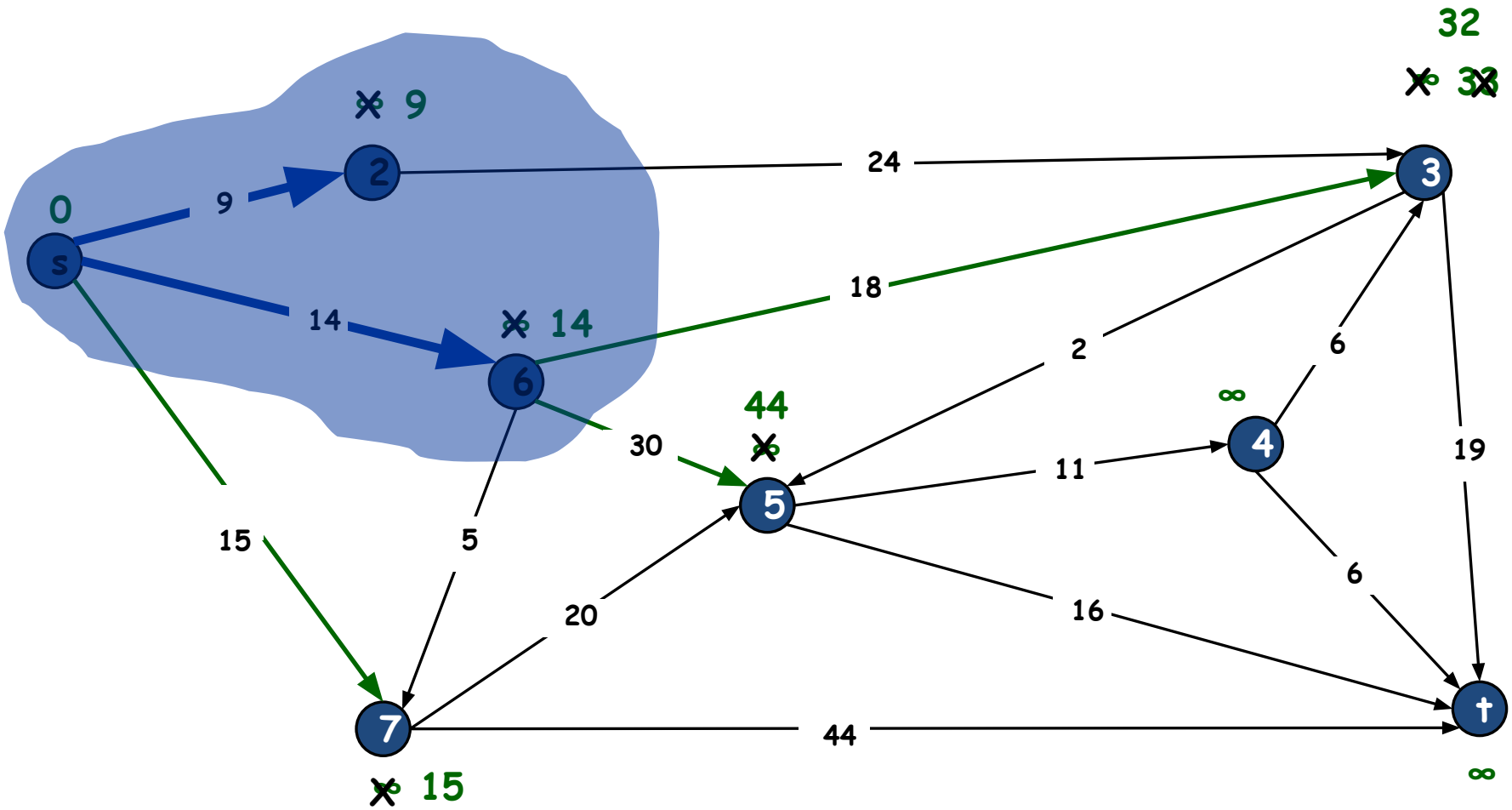
Dijkstra's Shortest Path Algorithm

S = { s, 2 }
PQ = { 3, 4, 5, 6, 7, t }



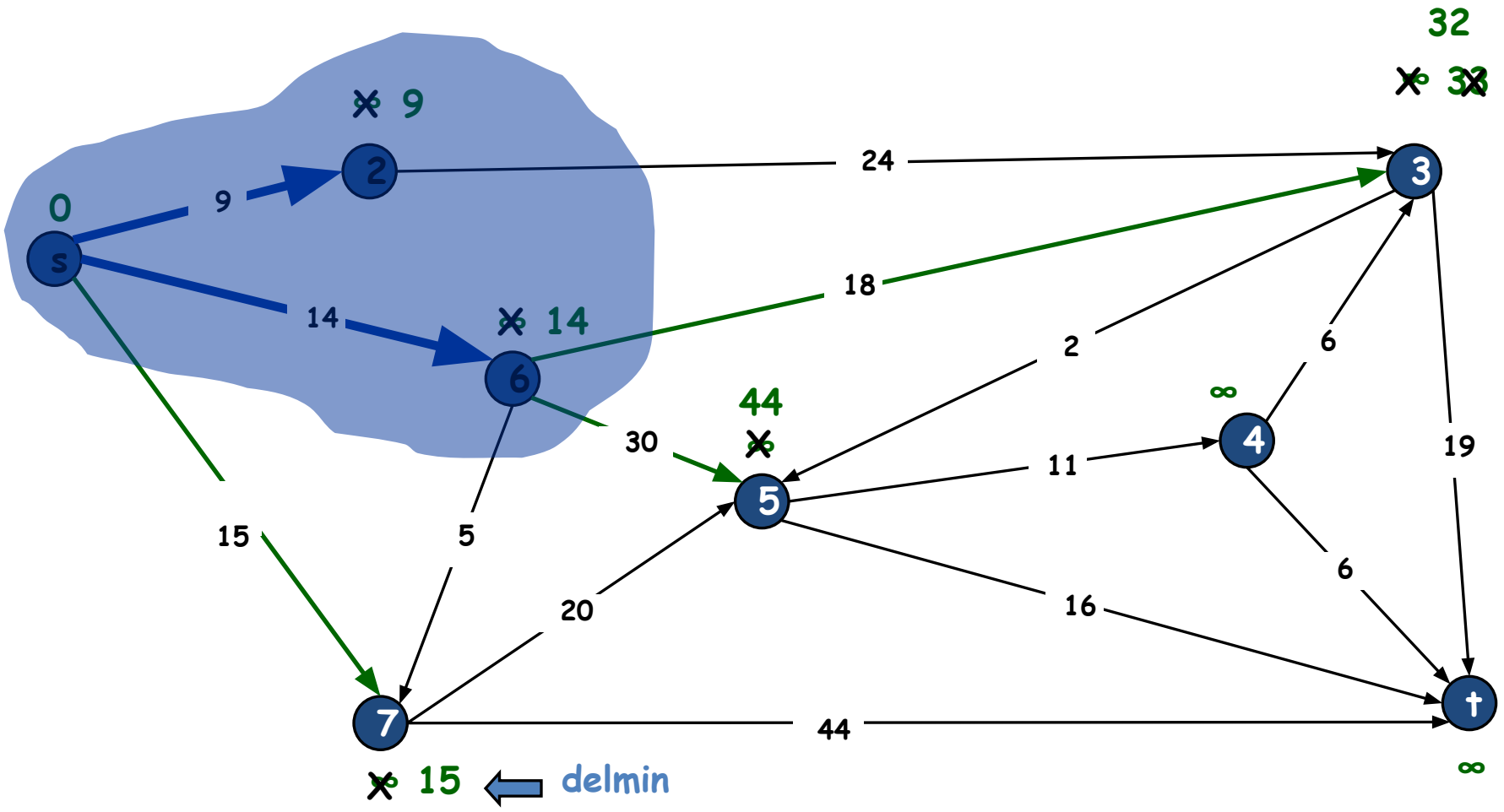
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



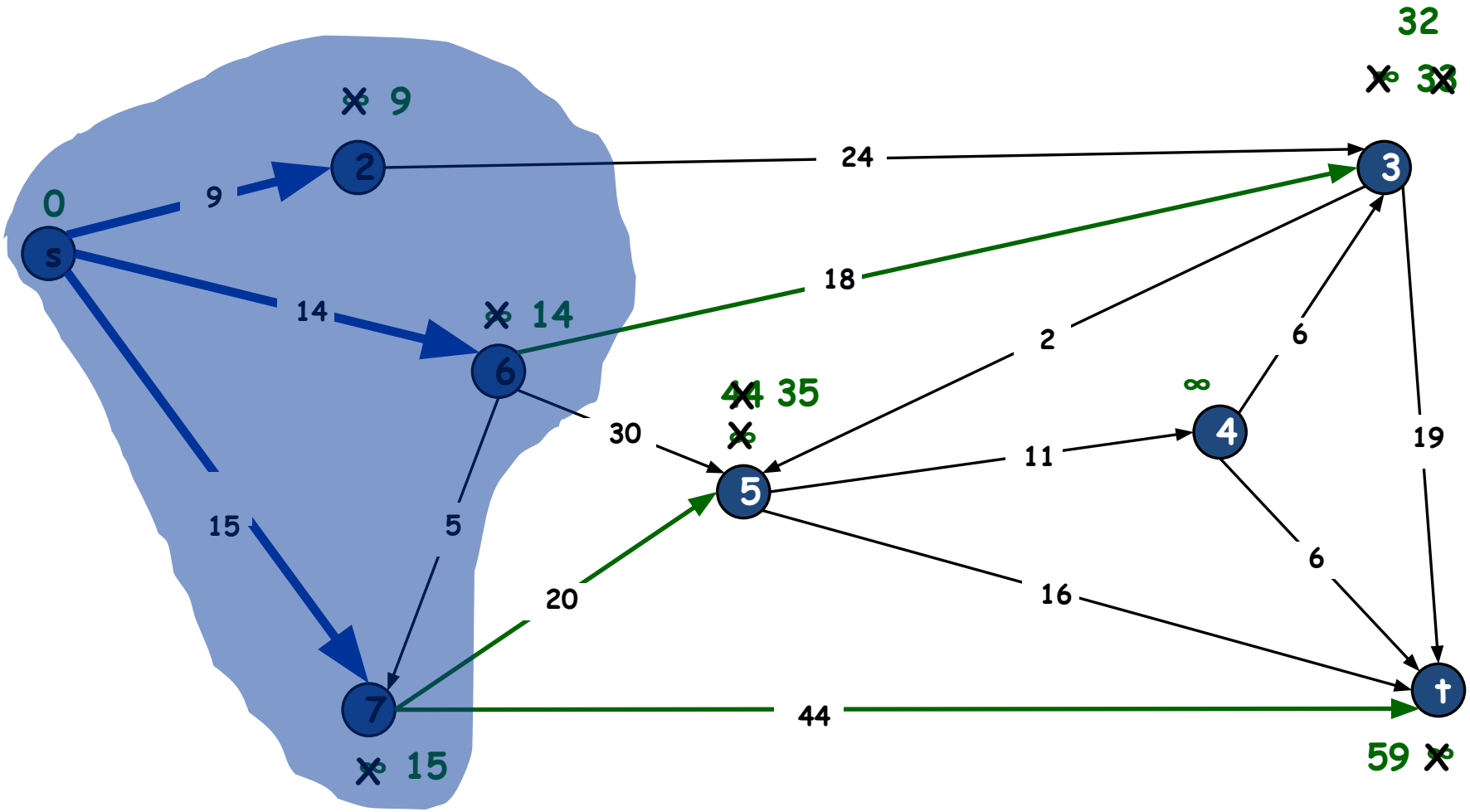
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



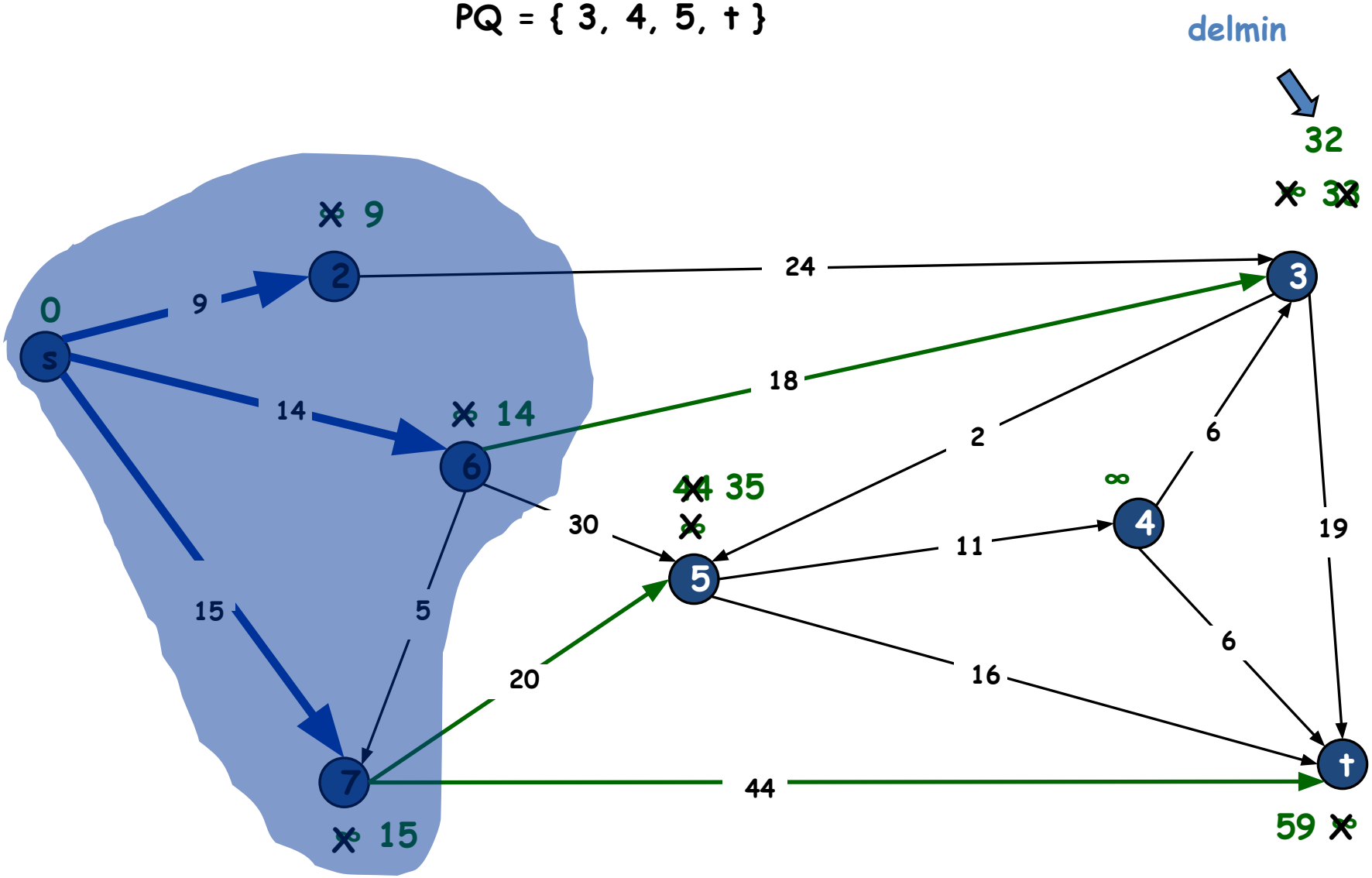
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



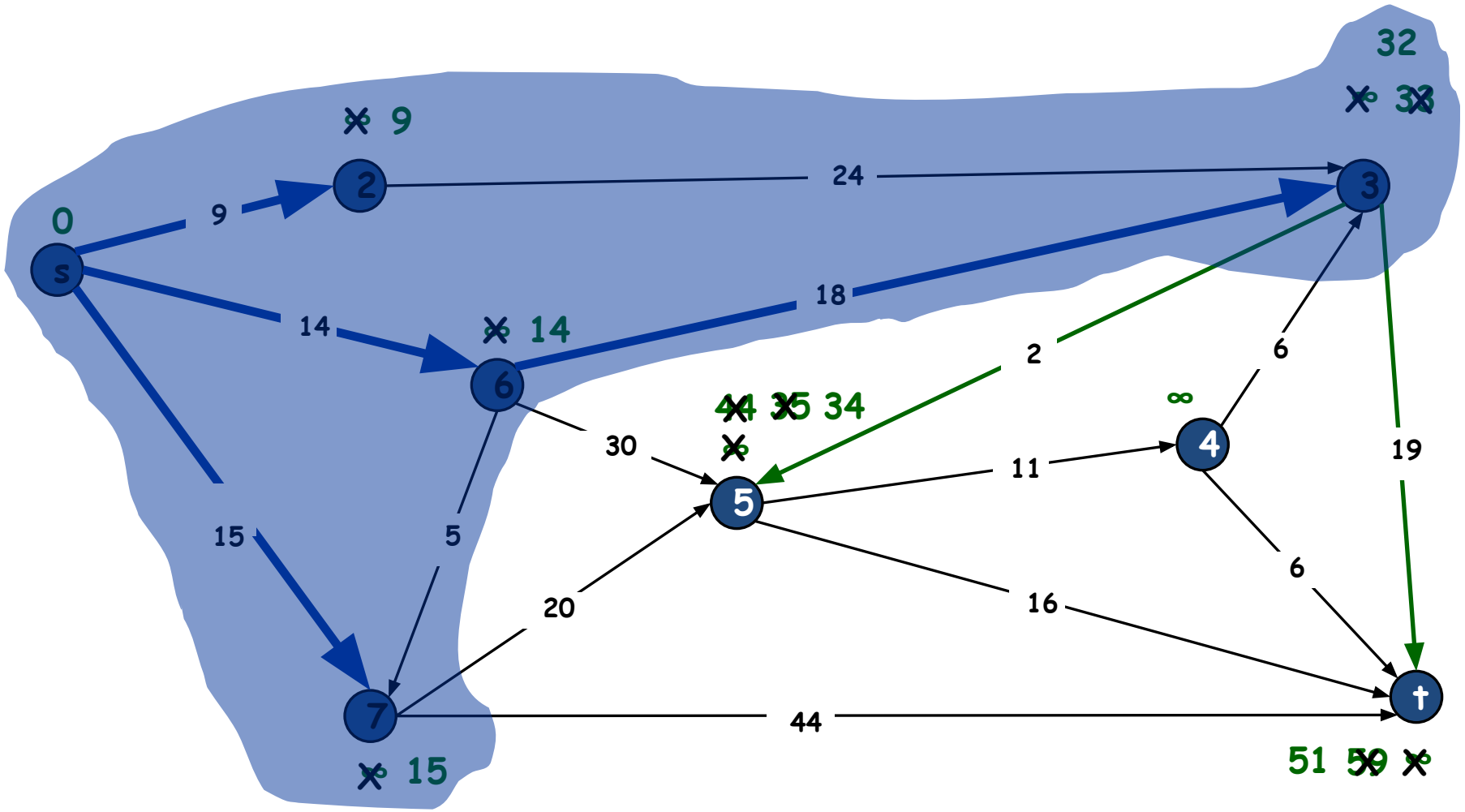
Dijkstra's Shortest Path Algorithm

S = { s, 2, 6, 7 }
PQ = { 3, 4, 5, t }



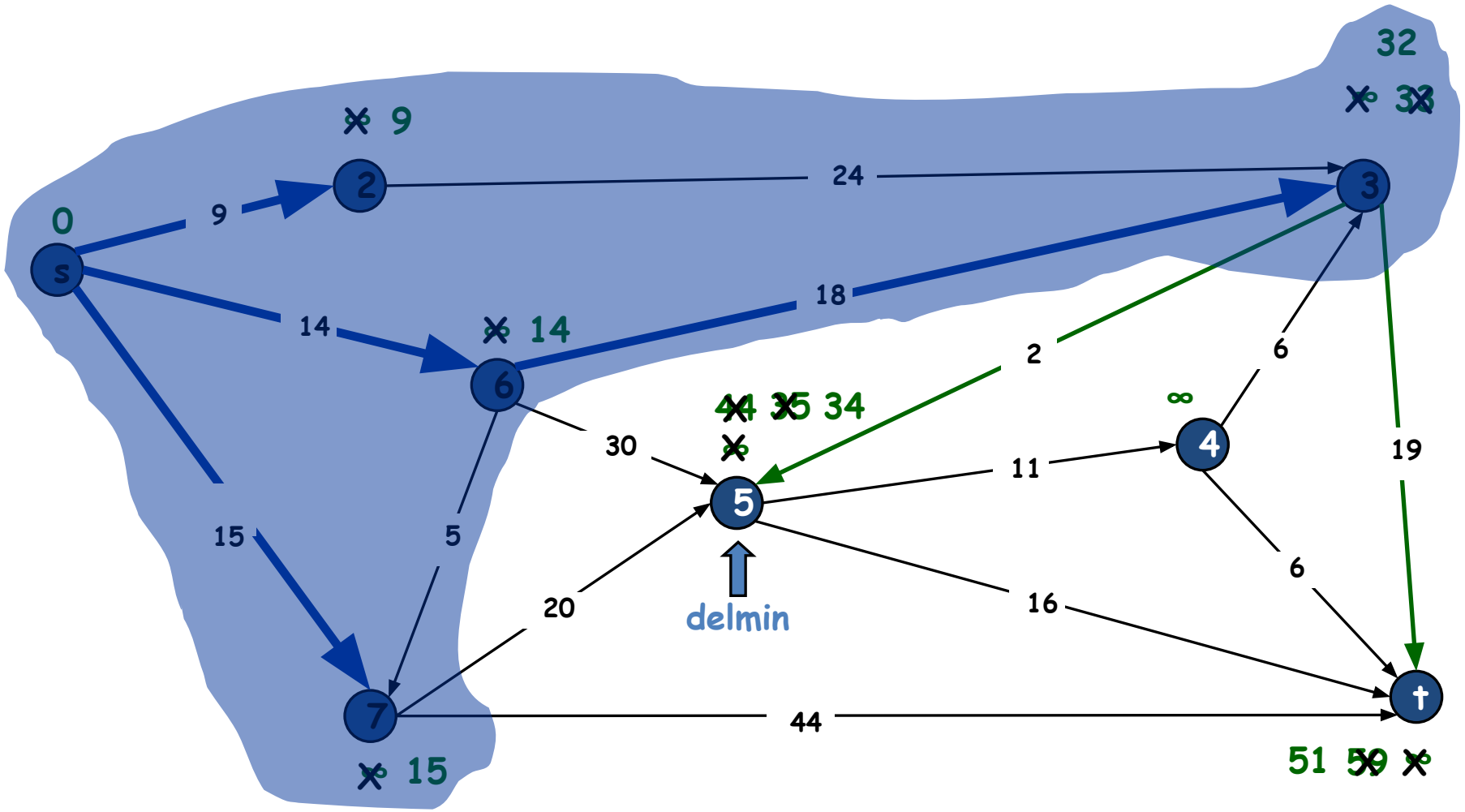
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



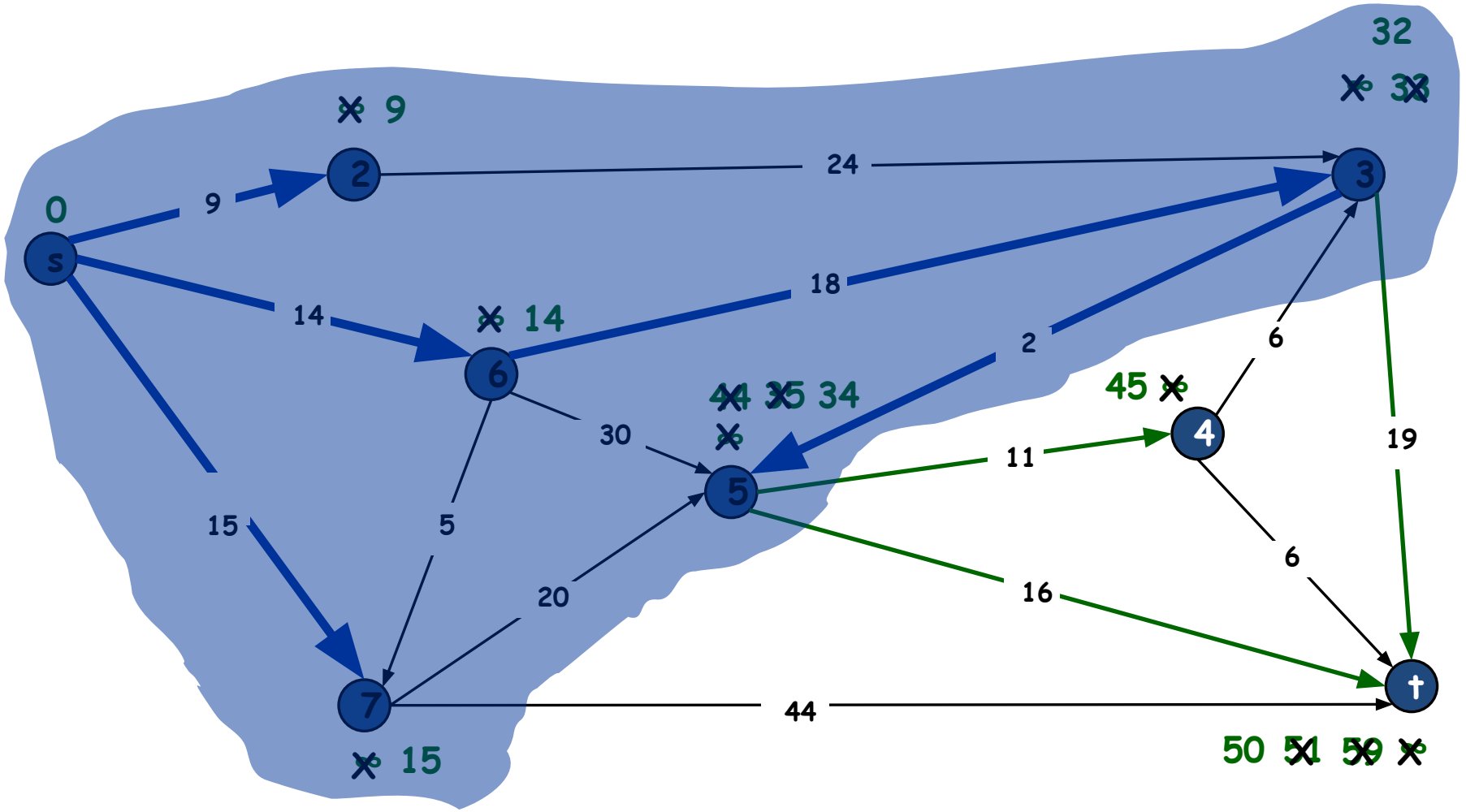
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 6, 7 }
PQ = { 4, 5, t }



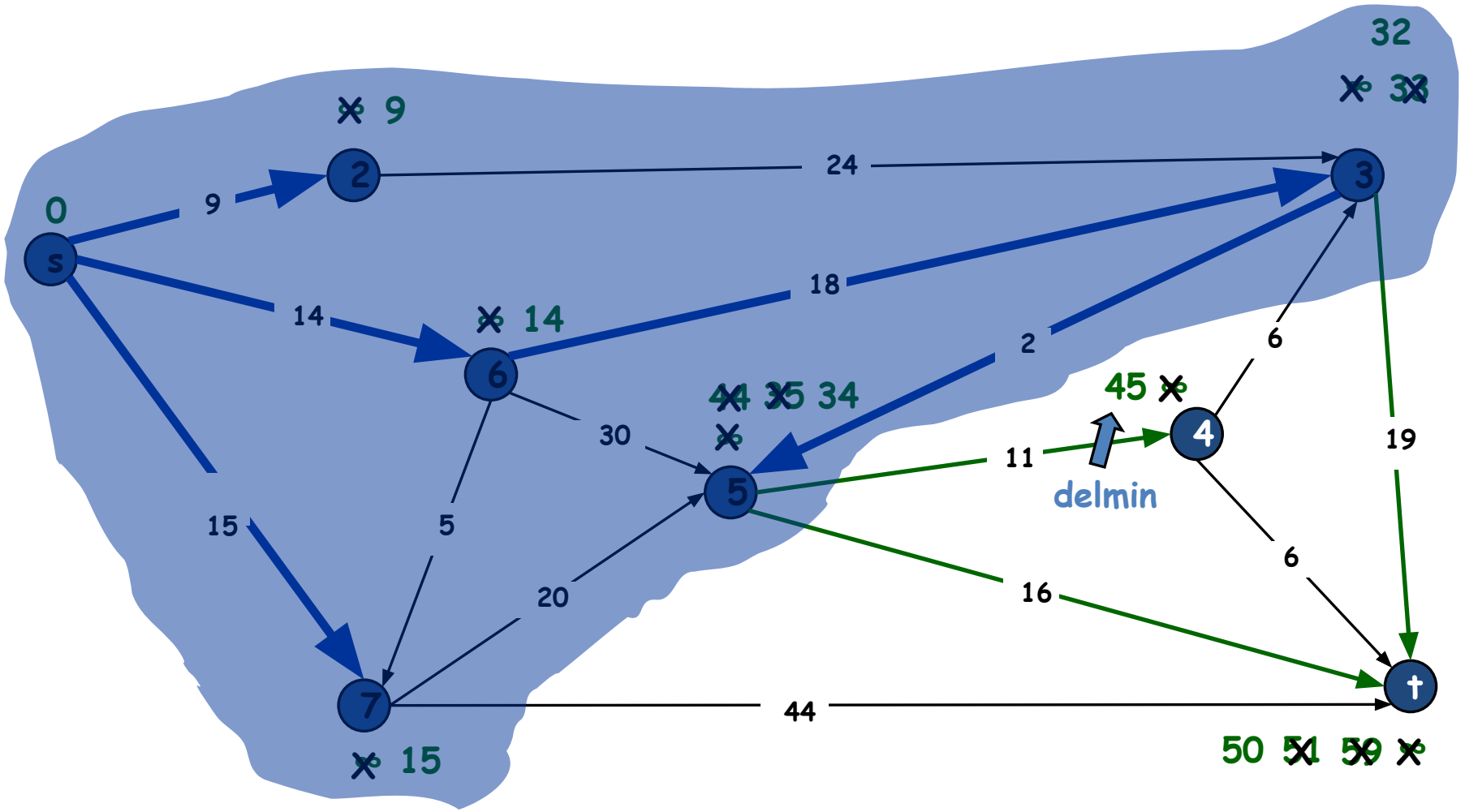
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



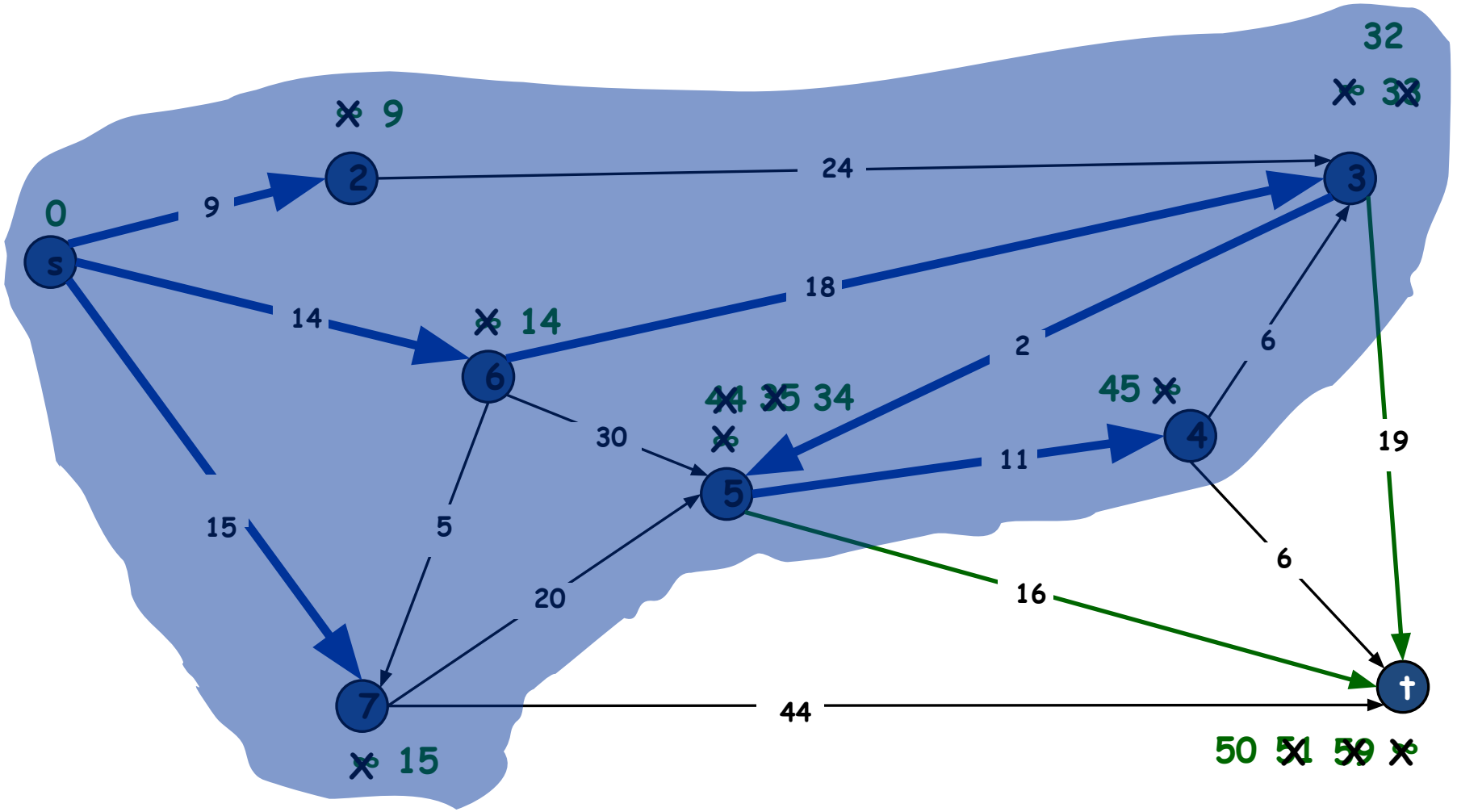
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



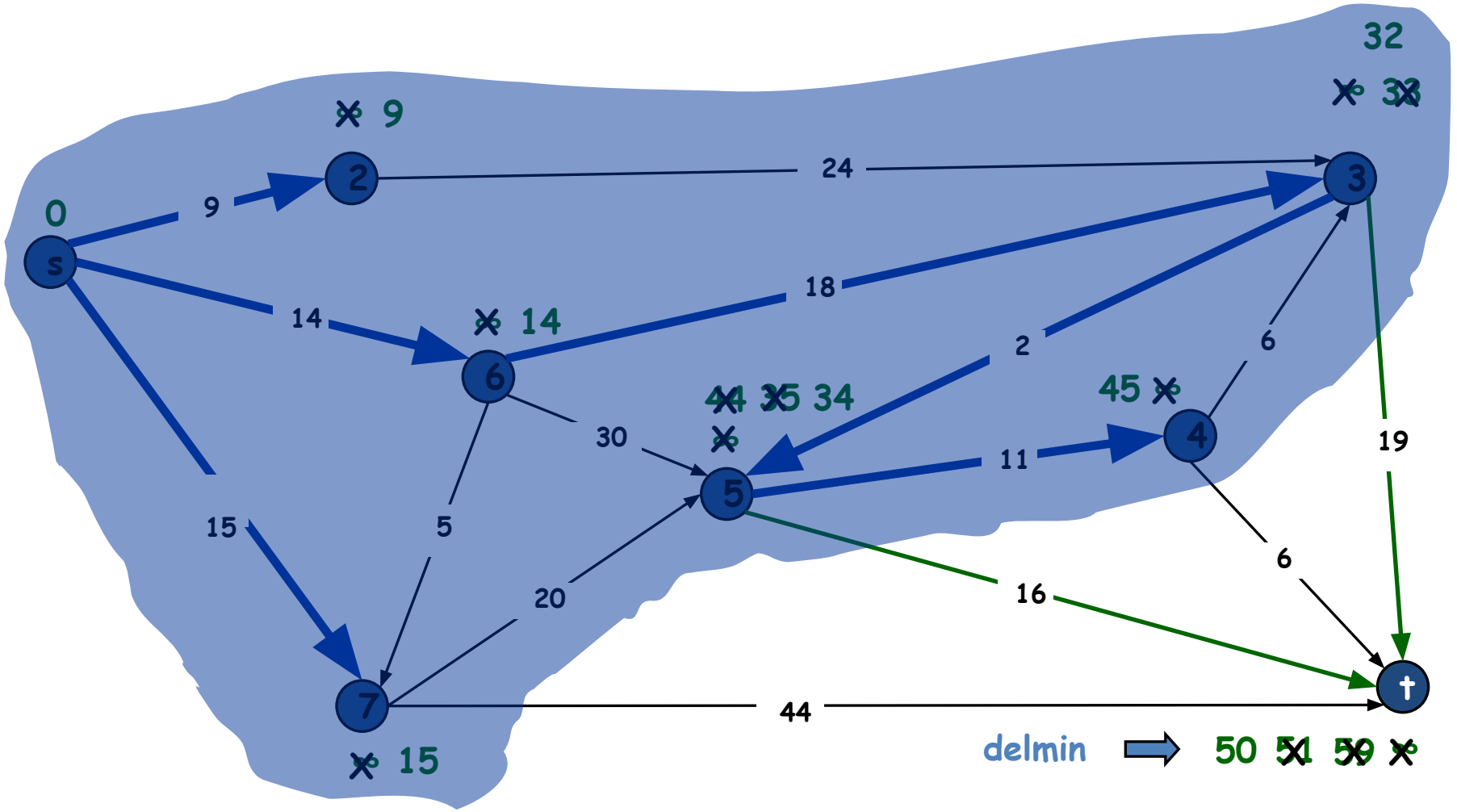
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



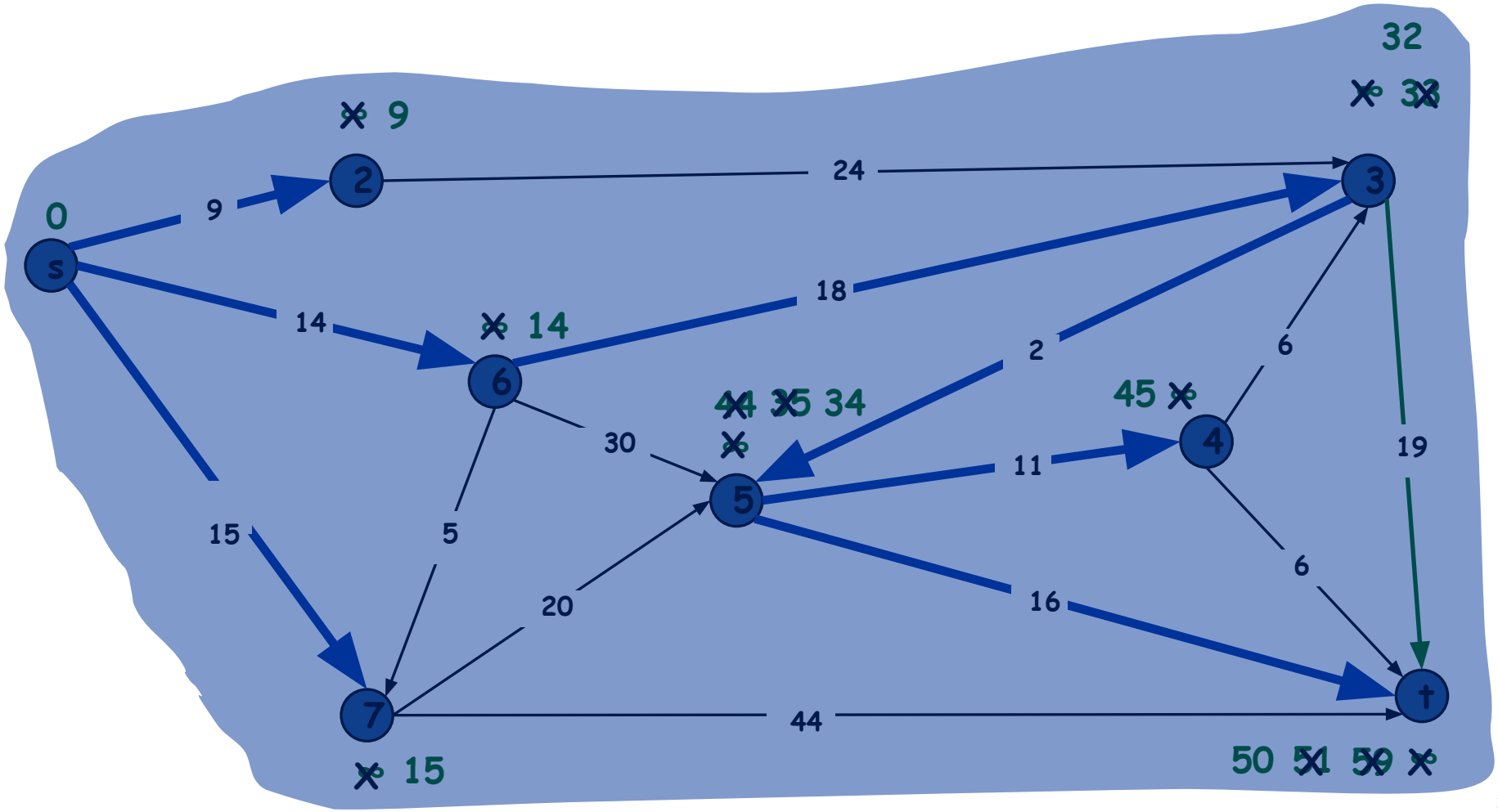
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



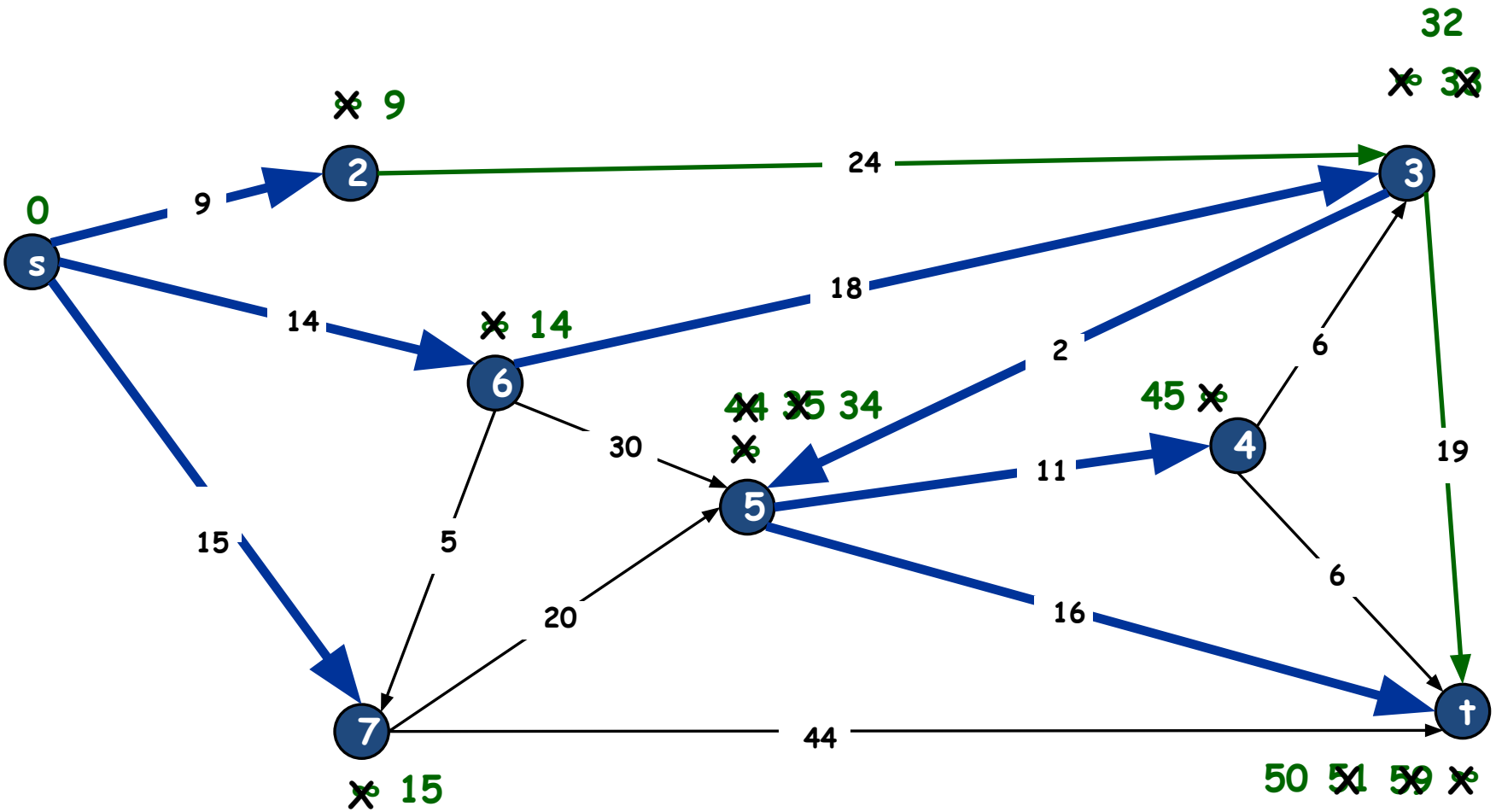
Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Dijkstra's Shortest Path Algorithm

S = { s, 2, 3, 4, 5, 6, 7, t }
PQ = { }



Dijkstra's algorithm: efficient implementation

- Implementation.
  - Algorithm maintains $\pi[v]$ for each node v.
  - Priority queue stores unexplored nodes, using $\pi[\,\cdot\,]$ as priorities.
  - Once u is deleted from the PQ, $\pi[u]$ = length of a shortest $s \rightsquigarrow u$ path.

```
DIJKSTRA (V, E, ℓ, s)

FOREACH v ≠ s : π[v] ← ∞, pred[v] ← null; π[s] ← 0.
Create an empty priority queue PQ.
FOREACH v ∈ V : INSERT(PQ, v, π[v]).
WHILE (IS-NOT-EMPTY(PQ))
    u ← DEL-MIN(PQ).
    FOREACH edge e = (u, v) ∈ E leaving u:
        IF (π[v] > π[u] + ℓ_e)
            π[v] ← π[u] + ℓ_e ; pred[v] ← e.
            DECREASE-KEY(PQ, v, π[v]).
```

Dijkstra's algorithm: which priority queue?

- Performance. Depends on PQ: n INSERT, n DELETE-MIN, $\leq m$ DECREASE-KEY.
  - Array implementation optimal for dense graphs. ← $\Theta(n^2)$ edges
  - Binary heap much faster for sparse graphs. ← $\Theta(n)$ edges
  - 4-way heap worth the trouble in performance-critical situations.

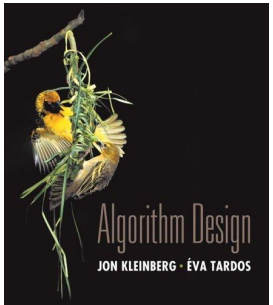
priority queue	INSERT	DELETE-MIN	DECREASE-KEY	total
unordered array	$O(1)$	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(m \log n)$
d-way heap (Johnson 1975)	$O(d \log n)$	$O(d \log_d n)$	$O(\log_d n)$	$O(m \log_{m/n} n)$
Fibonacci heap (Fredman–Tarjan 1984)	$O(1)$	$O(\log n)$ †	$O(1)$ †	$O(m + n \log n)$
integer priority queue (Thorup 2004)	$O(1)$	$O(\log \log n)$	$O(1)$	$O(m + n \log \log n)$

† amortized

MINIMUM SPANNING TREES [KT 4.5]

Adapted from Slides by
Kevin Wayne.

And Bistra Dilkina, Anne Benoit



Trees

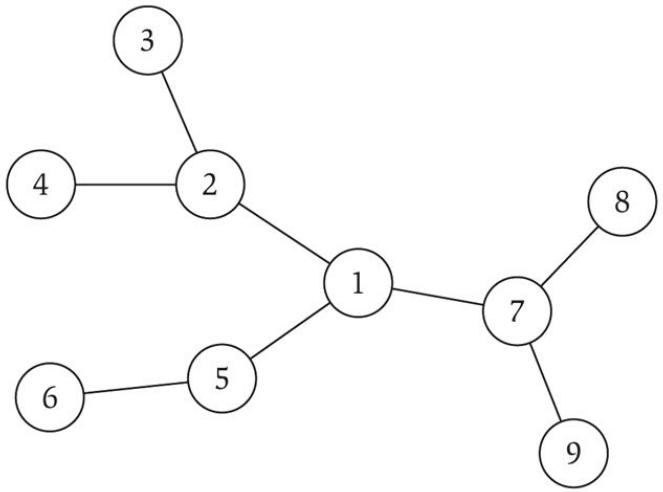
Def. A path is a sequence edges which connects a sequence of nodes.

Def. A cycle is a path with no repeated nodes or edges other than the start and end nodes.

Def. An undirected graph is a tree if it is connected and does not contain a cycle.

Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third.

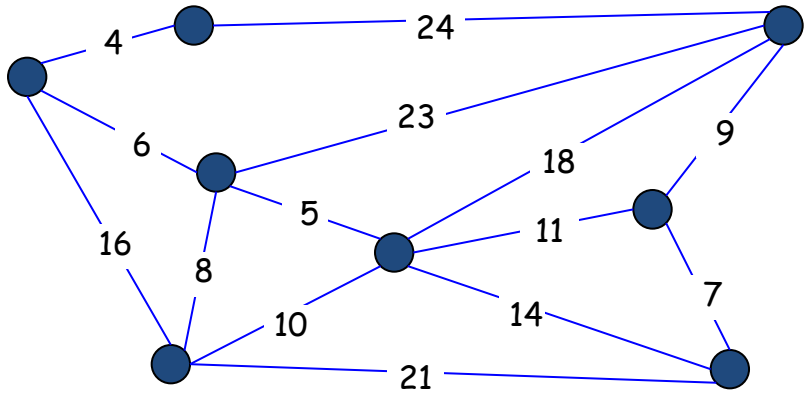
- G is connected.
- G does not contain a cycle.
- G has n-1 edges.



Spanning Tree

Def. An undirected graph is a spanning tree if it is a tree and touches every vertex in G.

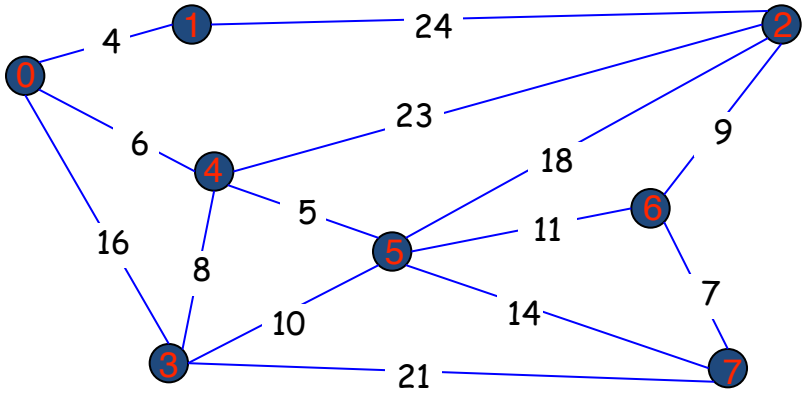
Cayley's Theorem. There are n^{n-2} spanning trees of K_n.



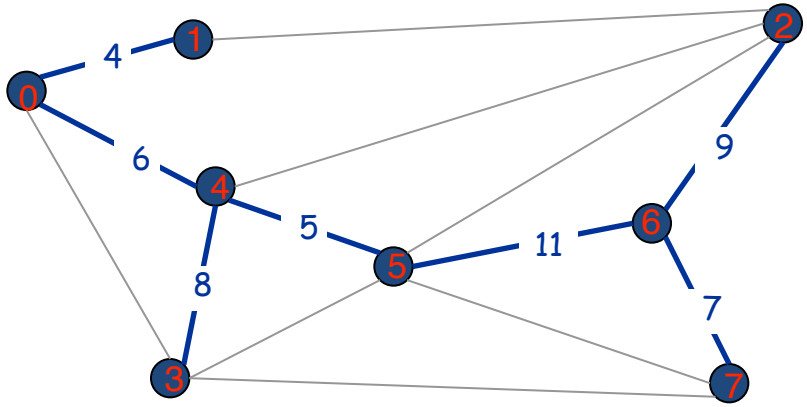
$G = (V, E)$

Minimum Spanning Tree

Minimum spanning tree. Given a connected graph G = (V, E) with real-valued edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a spanning tree whose sum of edge weights is minimized.



$G = (V, E)$



T, $\Sigma_{e \in T} c_e = 50$

Can't solve by brute force

Applications

MST is a fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Computational biology (single cells)
  - Eg. reconstruct trajectory of stem cell differentiation
- Cluster analysis.

Greedy Algorithms

Greedy Algorithms

Prim's algorithm. Start with some root node *s* and greedily grow a tree *T* from *s* outward. At each step, add the cheapest edge *e* to *T* that has exactly one endpoint in *T*.

Greedy Algorithms

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

Kruskal's algorithm. Start with $T = \varnothing$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Greedy Algorithms

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

Kruskal's algorithm. Start with $T = \varnothing$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Greedy Algorithms

Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

Kruskal's algorithm. Start with $T = \varnothing$. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.

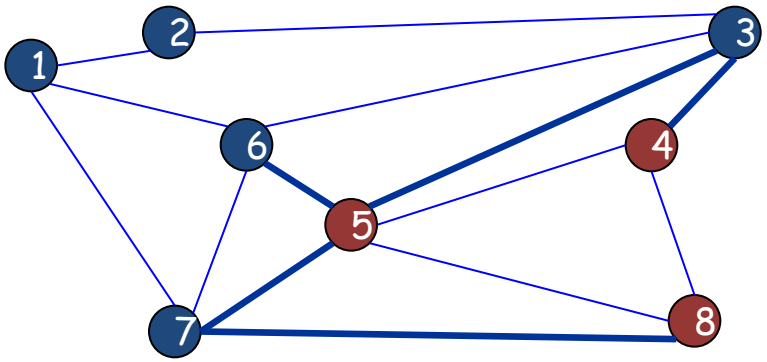
Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Remark. All three algorithms produce an MST.

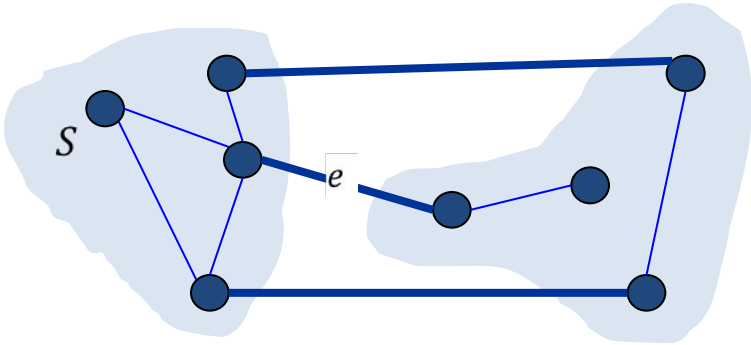
Greedy Algorithms

Def. A cut is a partition of the nodes into two nonempty subsets S and $V - S$.

Def. The cutset of a cut S is the set of edges with exactly one endpoint in S.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8



e is in the MST

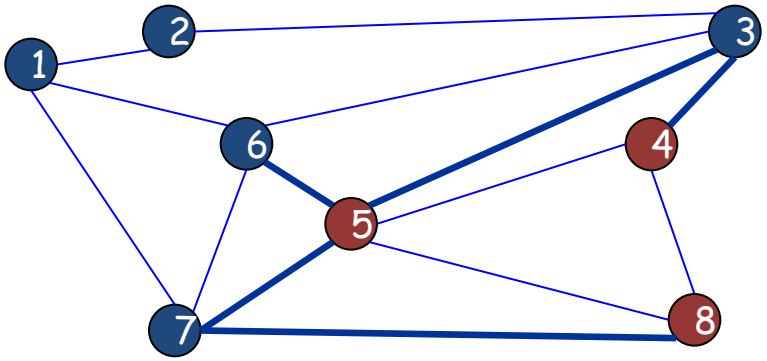
Greedy Algorithms

Def. A cut is a partition of the nodes into two nonempty subsets S and $V - S$.

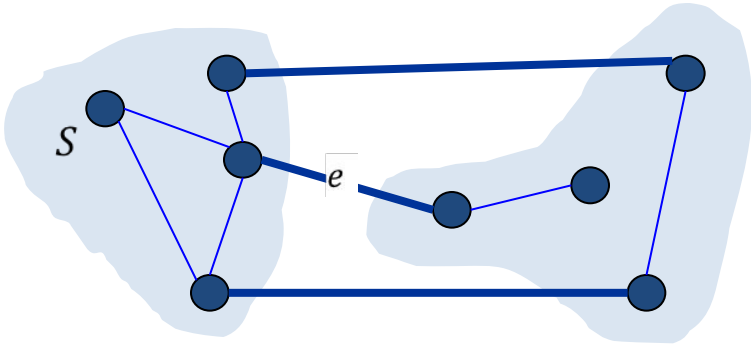
Def. The cutset of a cut S is the set of edges with exactly one endpoint in S.

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then every MST contains e.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8



e is in the MST

Proof of the cut property

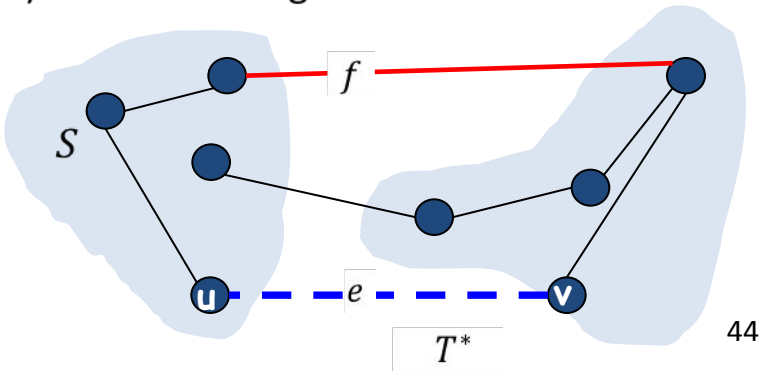
Cut Property

Cut property. Let S be any subset of nodes, and let $e = (u, v)$ be the min cost edge with exactly one endpoint in S. Then every MST T^* contains e.

Simplifying assumption. All edge costs c_e are distinct.

Pf. (exchange argument)

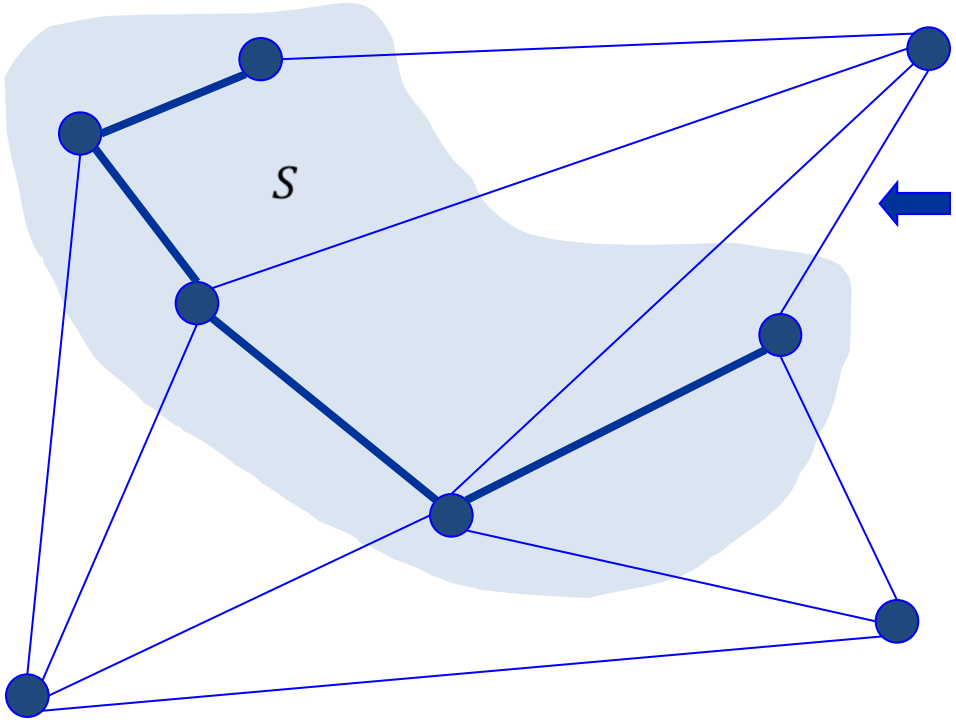
- Given: e is the min cost edge in the cutset for a set S, T^* is an MST
- Suppose e does not belong to T^*, and let's see what happens.
- The endpoints u and v of e must be connected by a path in T^* (spanning)
- => Adding e to T^* creates a cycle C in T^*
- Edge e is both in the cycle C and in the cutset corresponding to $S \Rightarrow$ there exists another edge, say f, that is in both C and cutset of S.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree, because:
  - All n nodes are connected (f, e were part of a cycle) AND $n - 1$ edges
- Since $c_e < c_f$, $cost(T') < cost(T^*)$.
- This is a contradiction since T^* is an optimal spanning tree, our assumption about e not being in T^* must be wrong.



Prim's Algorithm

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialize S = any node.
- (Apply cut property to S.)
- Add to tree the min cost edge (u, v) in cutset corresponding to S, and add one new explored node u to S.



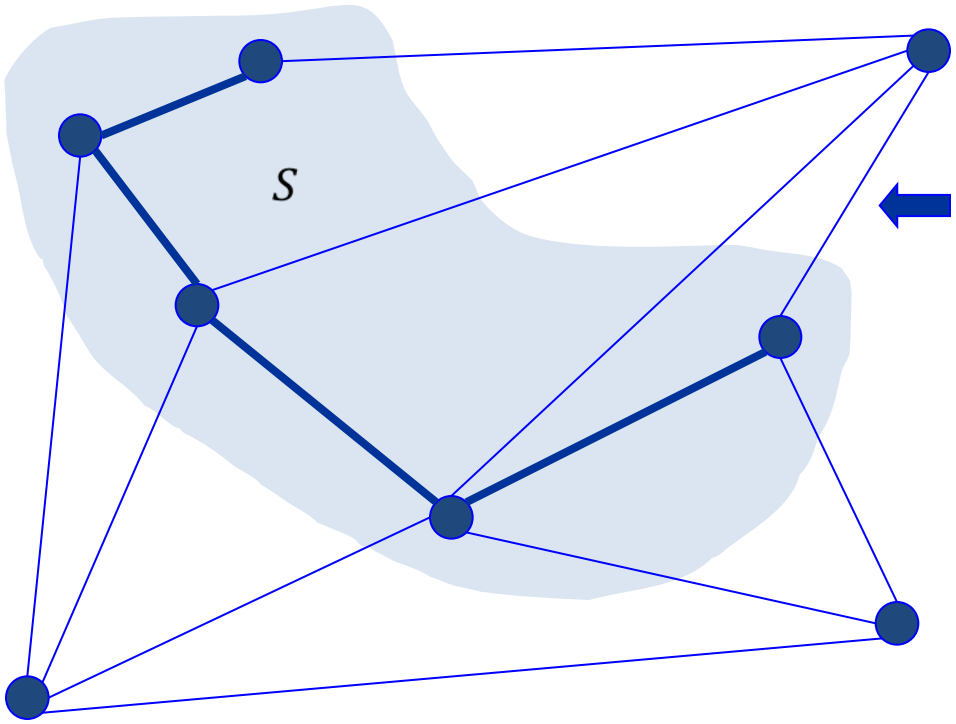
Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialize S = any node.
- (Apply cut property to S.)
- Add to tree the min cost edge (u, v) in cutset corresponding to S, and add one new explored node u to S.

1) Every edge added satisfies the Cut Property: by design

2) It produces a spanning tree: alg stops when $S = V$



Implementation: Prim's Algorithm

Implementation: Prim's Algorithm

Implementation. Use a priority queue (as for Dijkstra).

- Maintain set of explored nodes S.
- For each unexplored node v, maintain attachment cost $a[v]$ = cost of cheapest edge v to a node in S.
- $O(n^2)$ with an array; $O(m \log n)$ with a binary heap.

Cheapest edge between v and a node in explored set S

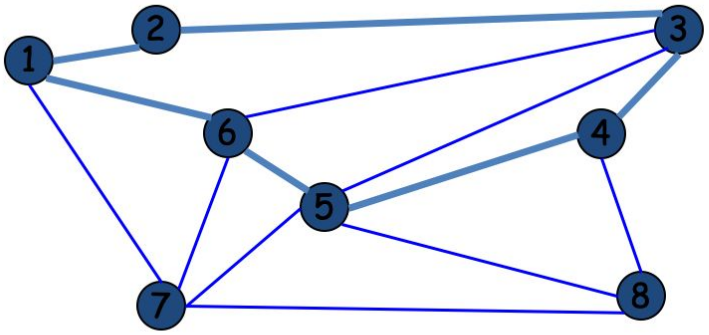
```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞
   Initialize an empty priority queue Q
   foreach (v ∈ V) insert v onto Q
   Initialize set of explored nodes S ← φ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
         if ((v ∉ S) and (c_e < a[v]))
            decrease priority a[v] to c_e
}
```

MST: cycle property

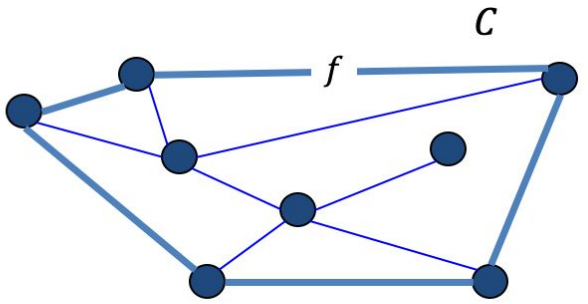
Simplifying assumption: All edge costs c_e are distinct.

Cycle. Set of edges the form a-b, b-c, c-d, …, y-z, z-a.

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then every MST does not contain f.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1



f is not in the MST

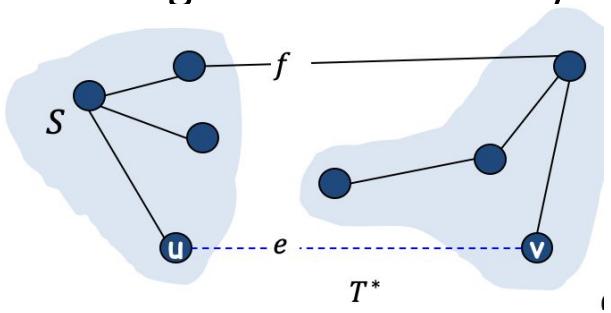
Cycle property: proof

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then every MST T* does not contain f.

Simplifying assumption. All edge costs c_e are distinct.

Pf. (exchange argument)

- Given: f is the max cost edge in a cycle C, T* is an MST
- Suppose f belongs to T*, and let's see what happens.
- Deleting f from T* disconnects T* and creates a cut S in T*.
- Edge f is both in the cycle C and in the cutset D corresponding to S => there exists another edge, say e, that is in both C and D.
- T′=T* U { e } - { f } is also a spanning tree (same argument as before).
- Since $c_e < c_f$, cost(T′) < cost(T*).
- This is a contradiction since T* is an optimal spanning tree, hence our assumption about f is wrong.



Reverse-Delete Algorithm: Proof of Correctness

The algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge e from T unless doing so would disconnect T.

Th. This algorithm produces an MST of G.

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then every MST T^* does not contain f.

Pf.

- Edges removed do not belong to any MST.
- The output is a spanning tree: connected & no cycle.