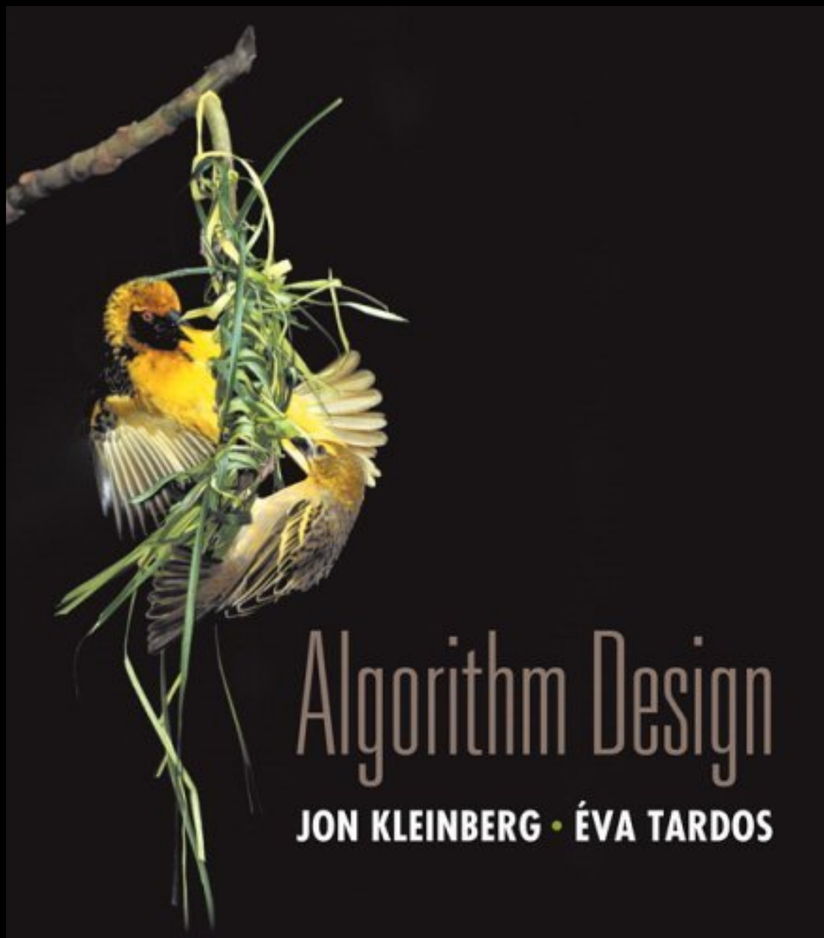
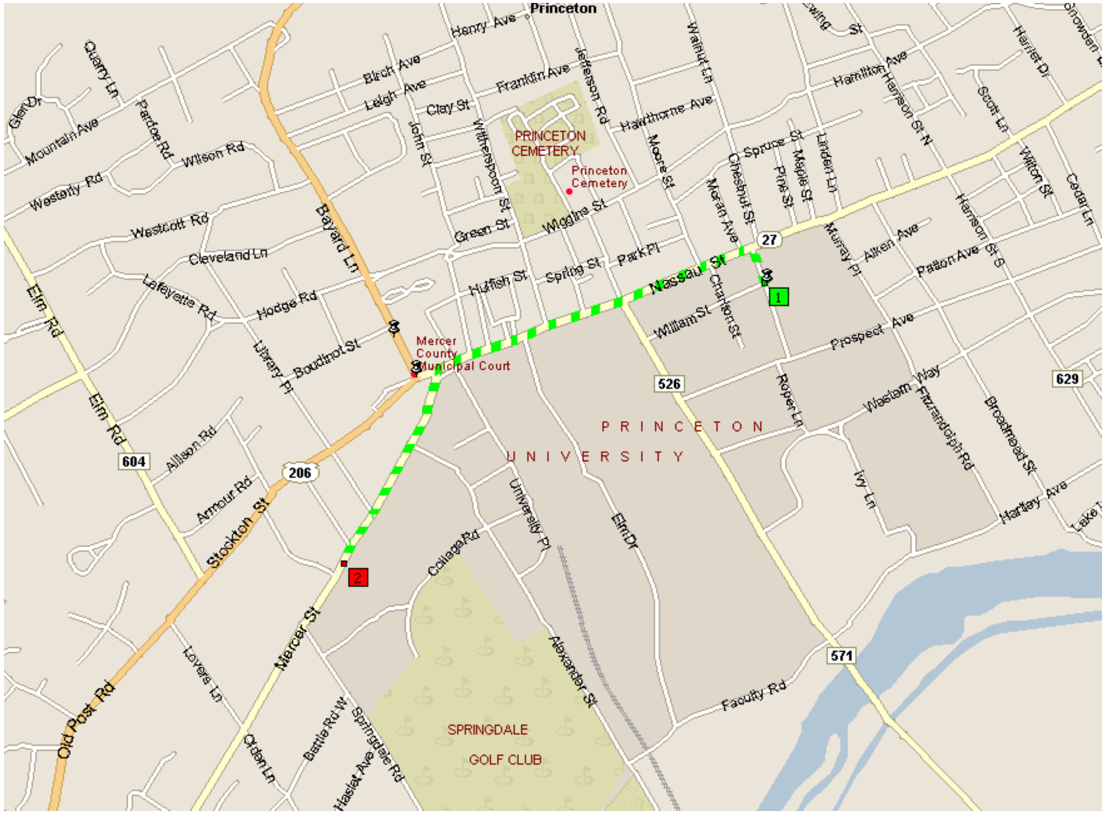


Chapter 4

Greedy Algorithms

Slides by Kevin Wayne.

4.4 Shortest Paths in a Graph



shortest path from Princeton CS department to Einstein's house

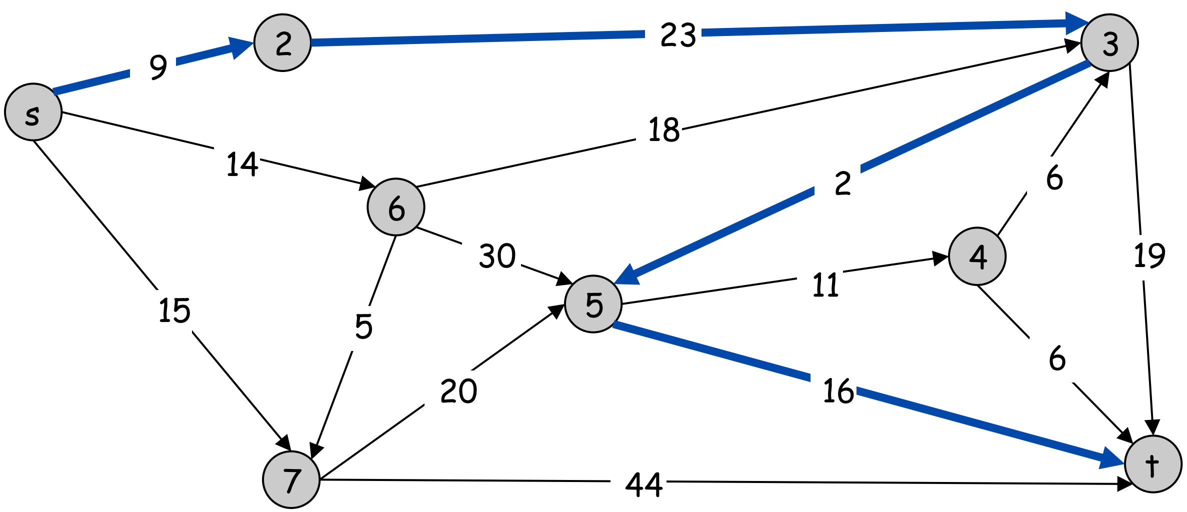
Shortest Path Problem

Shortest path network.

- Directed graph $G = (V, E)$.
- Source s, destination t.
- Length ℓ_e = length of edge e.

Shortest path problem: find shortest directed path from s to t.

cost of path = sum of edge costs in path



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

Dijkstra's Algorithm

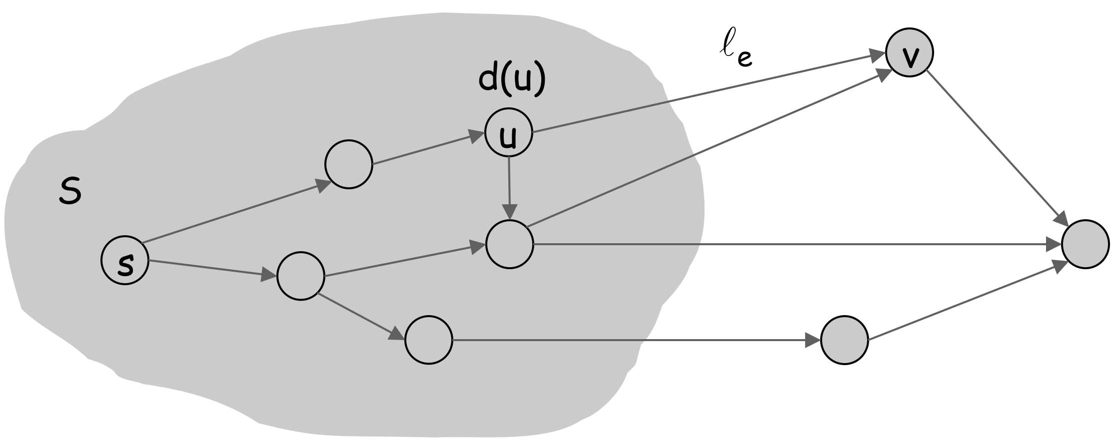
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

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

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



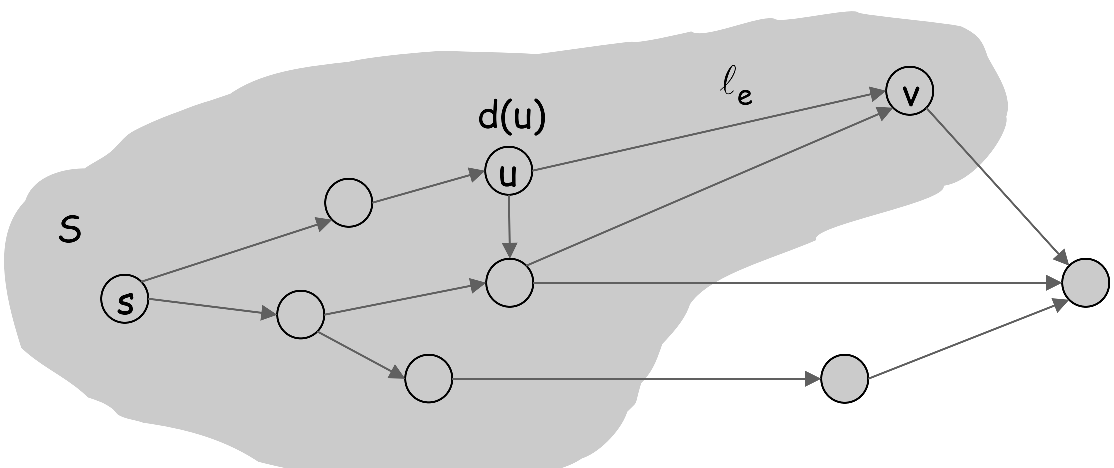
Dijkstra's algorithm.

- Maintain a set of explored nodes S for which we have determined the shortest path distance d(u) from s to u.
- Initialize S = { s }, d(s) = 0.
- Repeatedly choose unexplored node v which minimizes

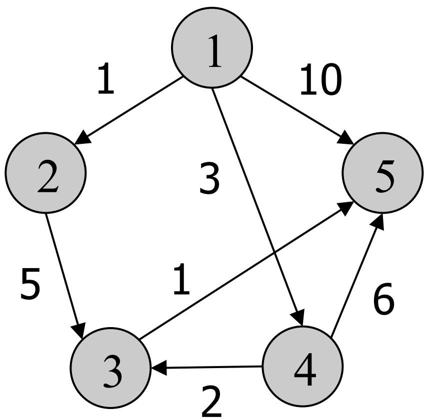
$$\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e ,$$

shortest path to some u in explored part, followed by a single edge (u, v)

add v to S, and set d(v) = π(v).



Dijkstra's Algorithm

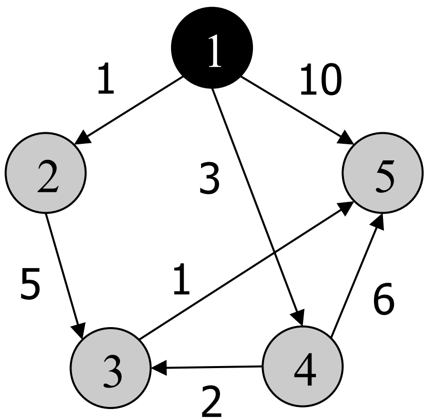


$S = \emptyset$

D =

1	2	3	4	5
	∞	∞	∞	∞

Dijkstra's Algorithm

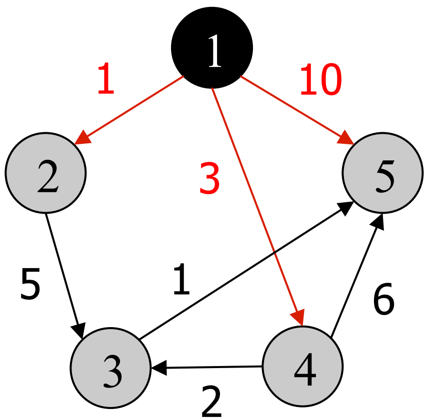


S = {1}

D =

1	2	3	4	5
	∞	∞	∞	∞

Dijkstra's Algorithm

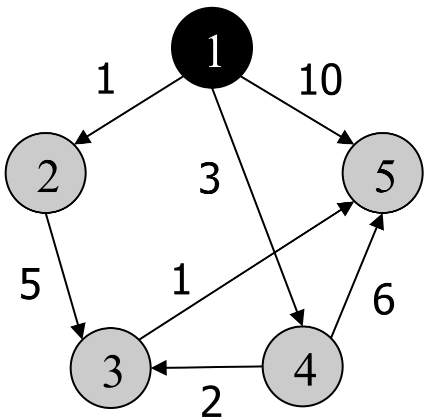


$S = \{1\}$

$D =$

	1	∞	3	10
1	2	3	4	5

Dijkstra's Algorithm

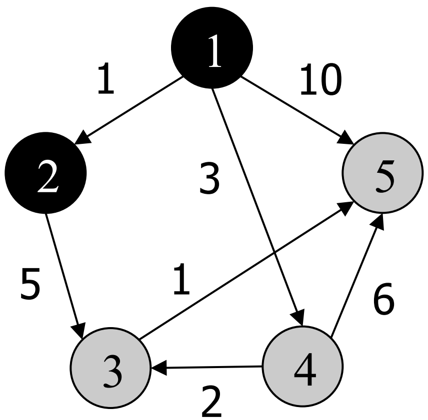


S = {1}

D =

1	2	3	4	5
	1	∞	3	10

Dijkstra's Algorithm

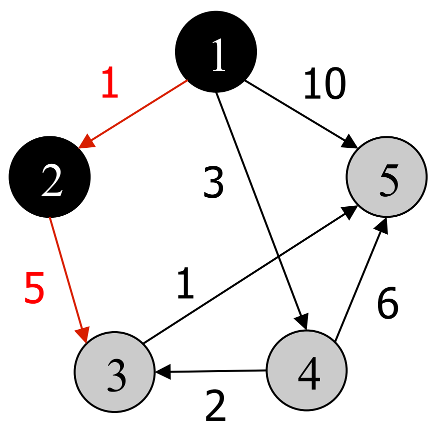


S = {1, 2}

D =

	1	∞	3	10
1	2	3	4	5

Dijkstra's Algorithm

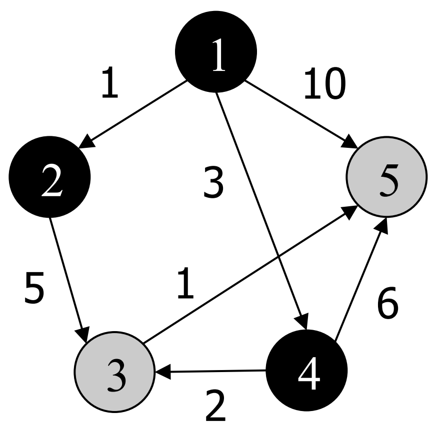


S = {1, 2}

D =

1	2	3	4	5
	1	6	3	10

Dijkstra's Algorithm

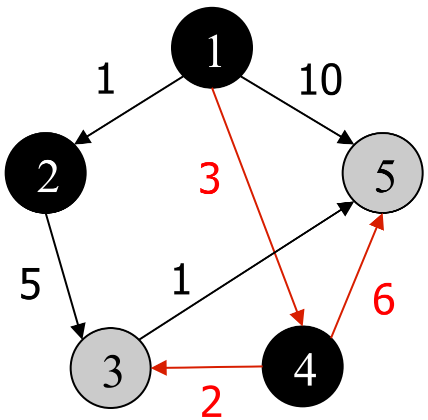


$S = \{1, 2, 4\}$

D =

1	2	3	4	5
	1	6	3	10

Dijkstra's Algorithm

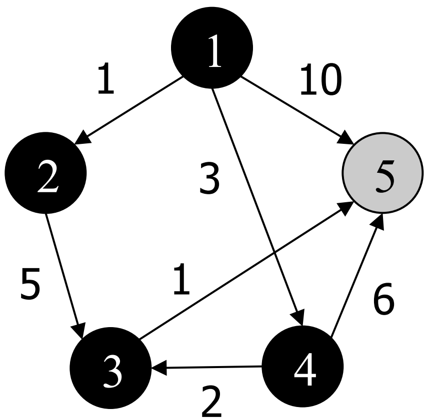


$S = \{1, 2, 4\}$

D =

1	2	3	4	5
	1	5	3	9

Dijkstra's Algorithm

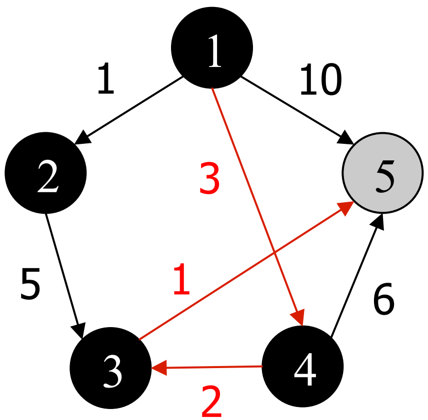


S = {1, 2, 4, 3}

D =

1	2	3	4	5
	1	5	3	9

Dijkstra's Algorithm

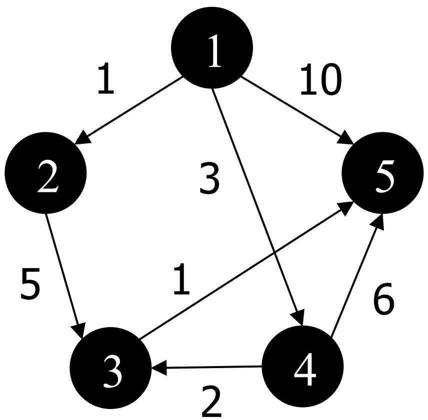


S = {1, 2, 4, 3}

D =

1	2	3	4	5
	1	5	3	6

Dijkstra's Algorithm



S = {1, 2, 4, 3, 5}

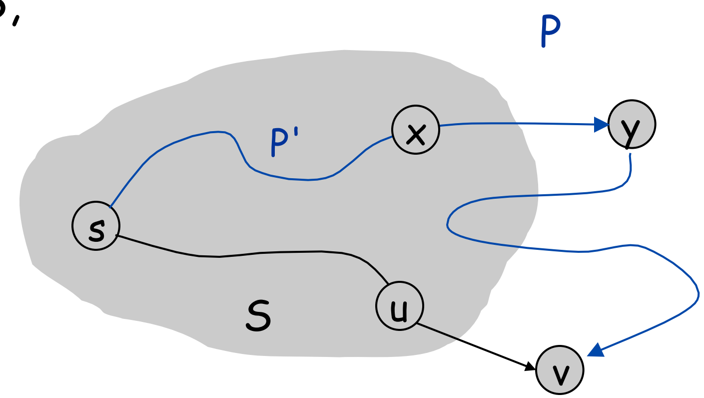
D =

	1	5	3	6
1	2	3	4	5

Dijkstra's Algorithm: Proof of Correctness

Invariant. For each node $u \in S$, $d(u)$ is the length of the shortest s-u path.
Pf. (by induction on $|S|$)
Base case: $|S| = 1$ is trivial.
Inductive hypothesis: Assume true for $|S| = k \geq 1$.

- Let v be next node added to S, and let u-v be the chosen edge.
- The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
- Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
- Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
- P is already too long as soon as it leaves S.



$$\ell(P) \geq \ell(P') + \ell(x,y) \geq d(x) + \ell(x,y) \geq \pi(y) \geq \pi(v)$$

- $\ell(P) \geq \ell(P') + \ell(x,y)$: nonnegative weights
- $\ell(P') + \ell(x,y) \geq d(x) + \ell(x,y)$: inductive hypothesis
- $d(x) + \ell(x,y) \geq \pi(y)$: defn of $\pi(y)$
- $\pi(y) \geq \pi(v)$: Dijkstra chose v instead of y

Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e$.

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}.$$

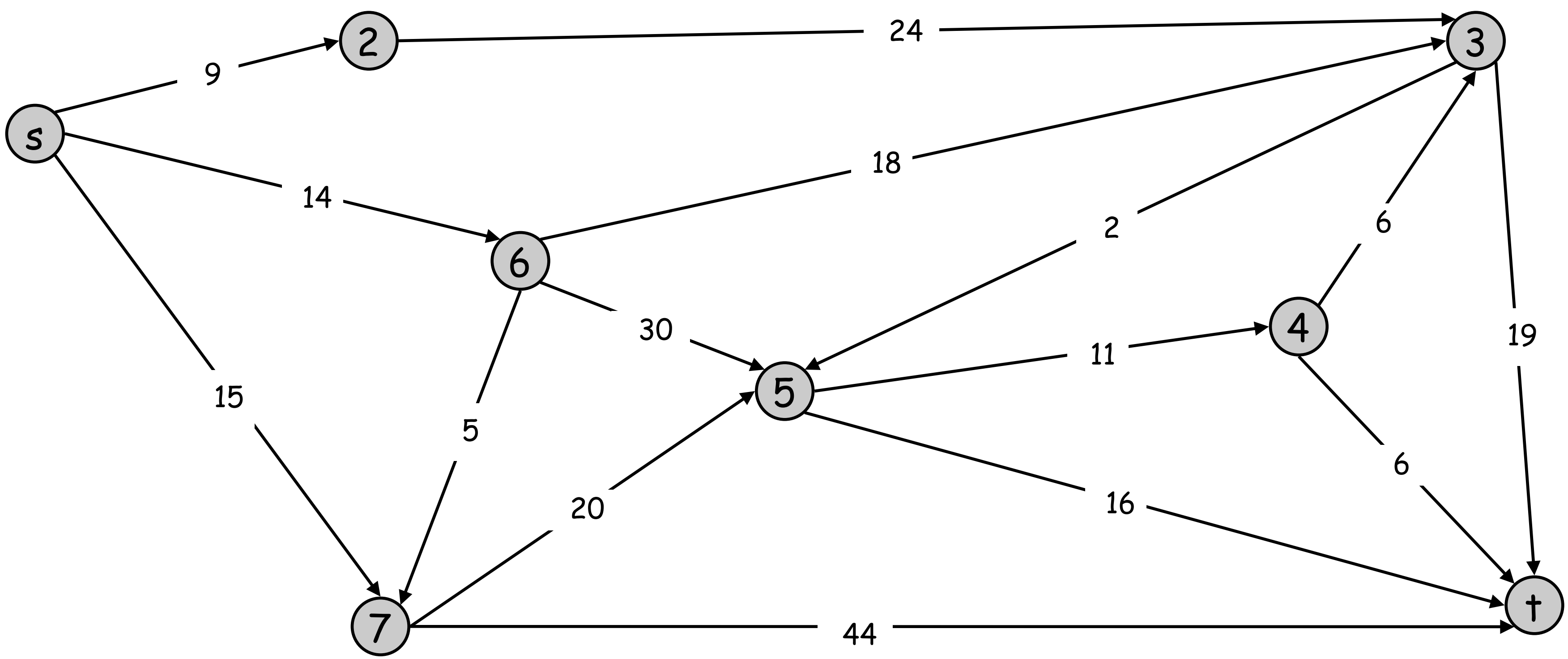
Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

† Individual ops are amortized bounds

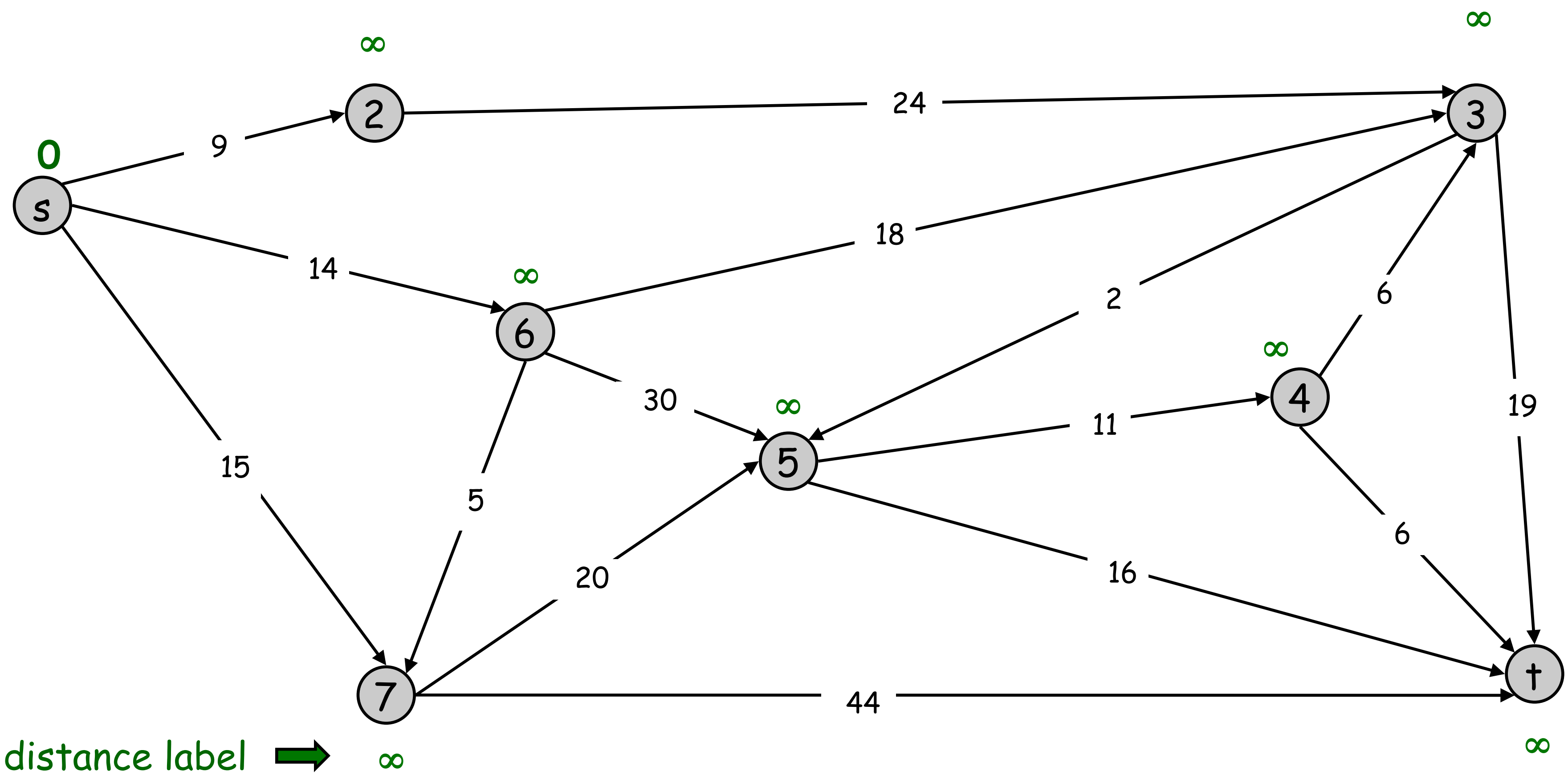
Dijkstra's Shortest Path Algorithm

Find shortest path from s to t.



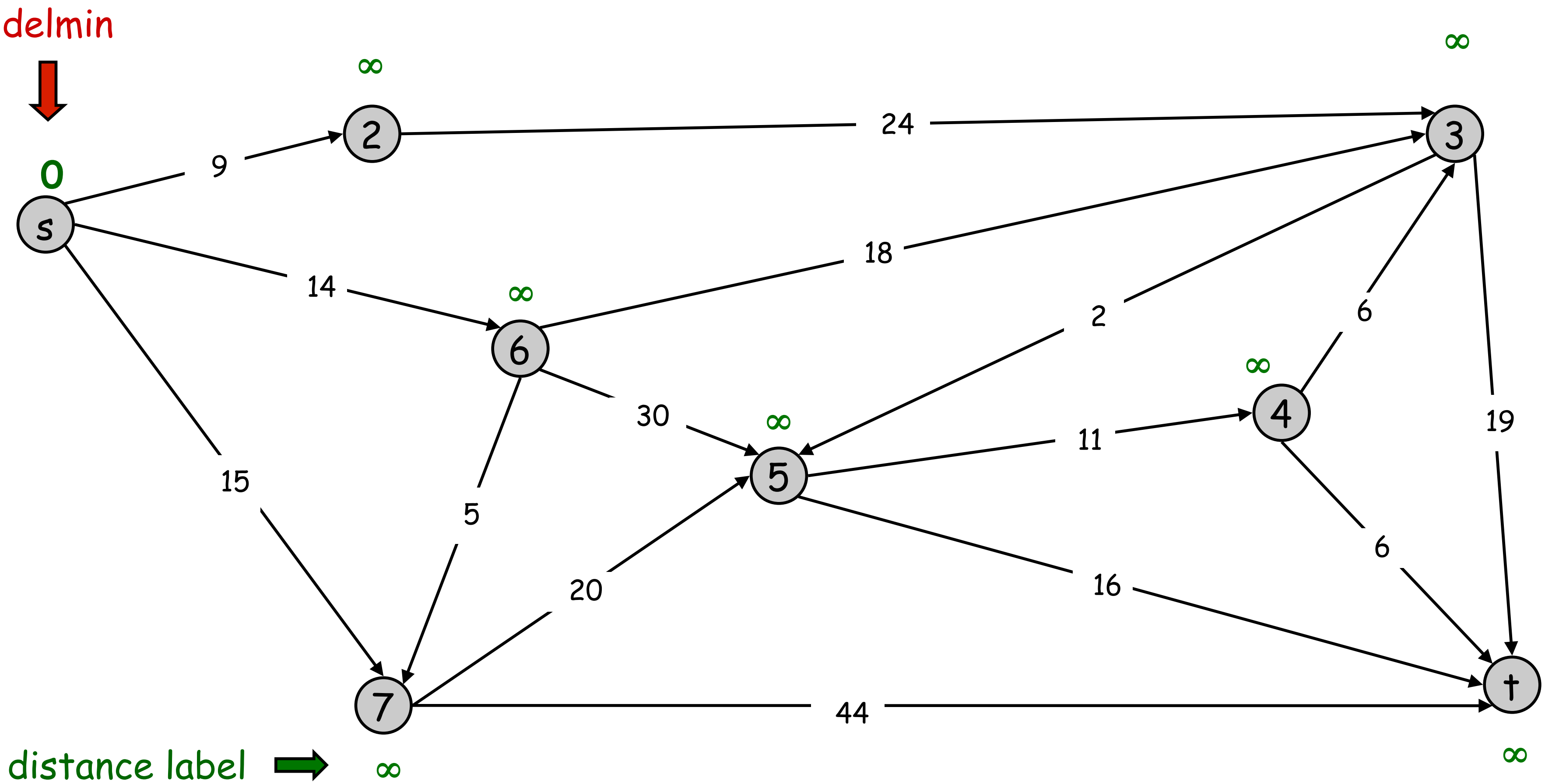
Dijkstra's Shortest Path Algorithm

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



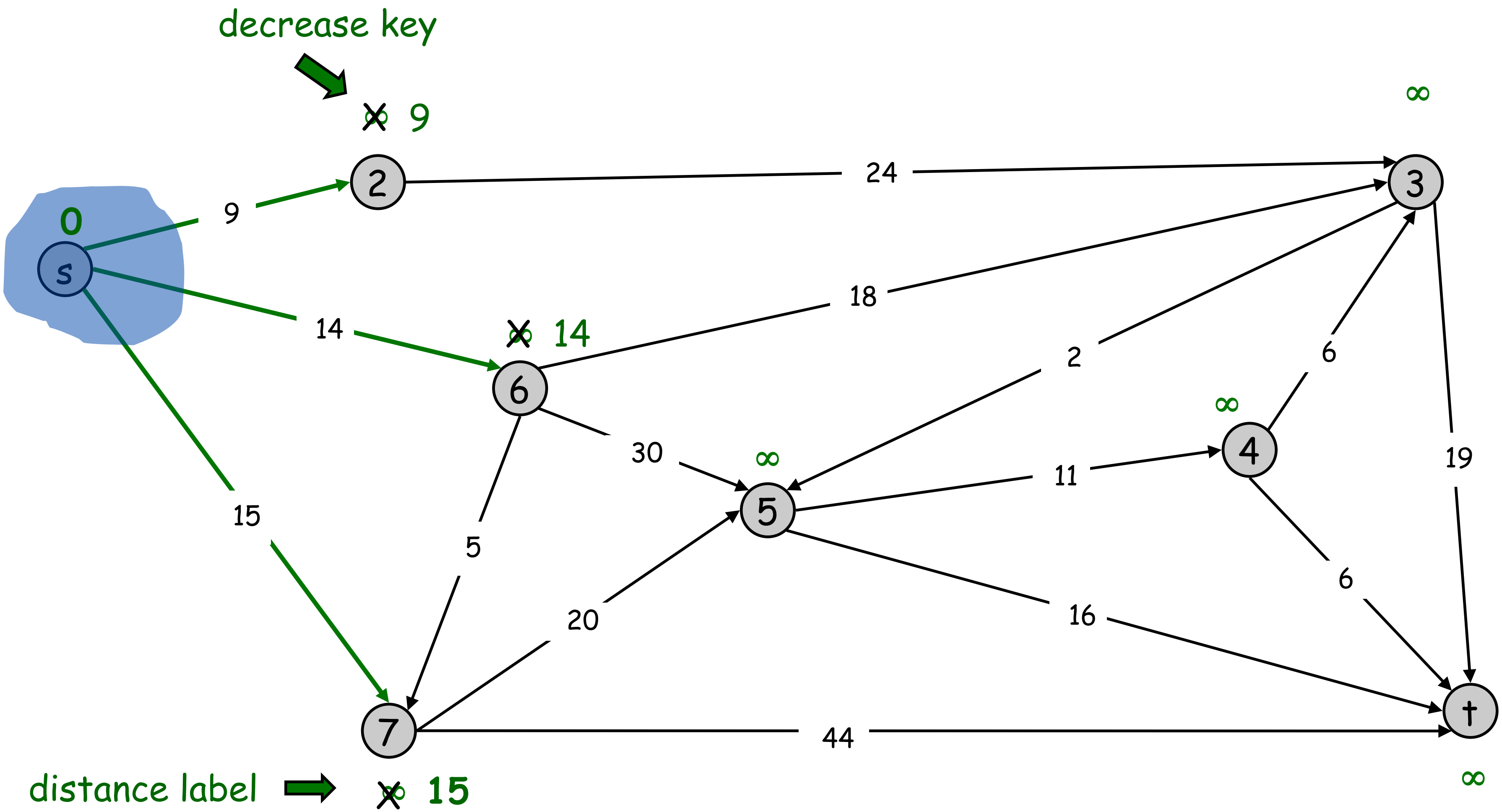
Dijkstra's Shortest Path Algorithm

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



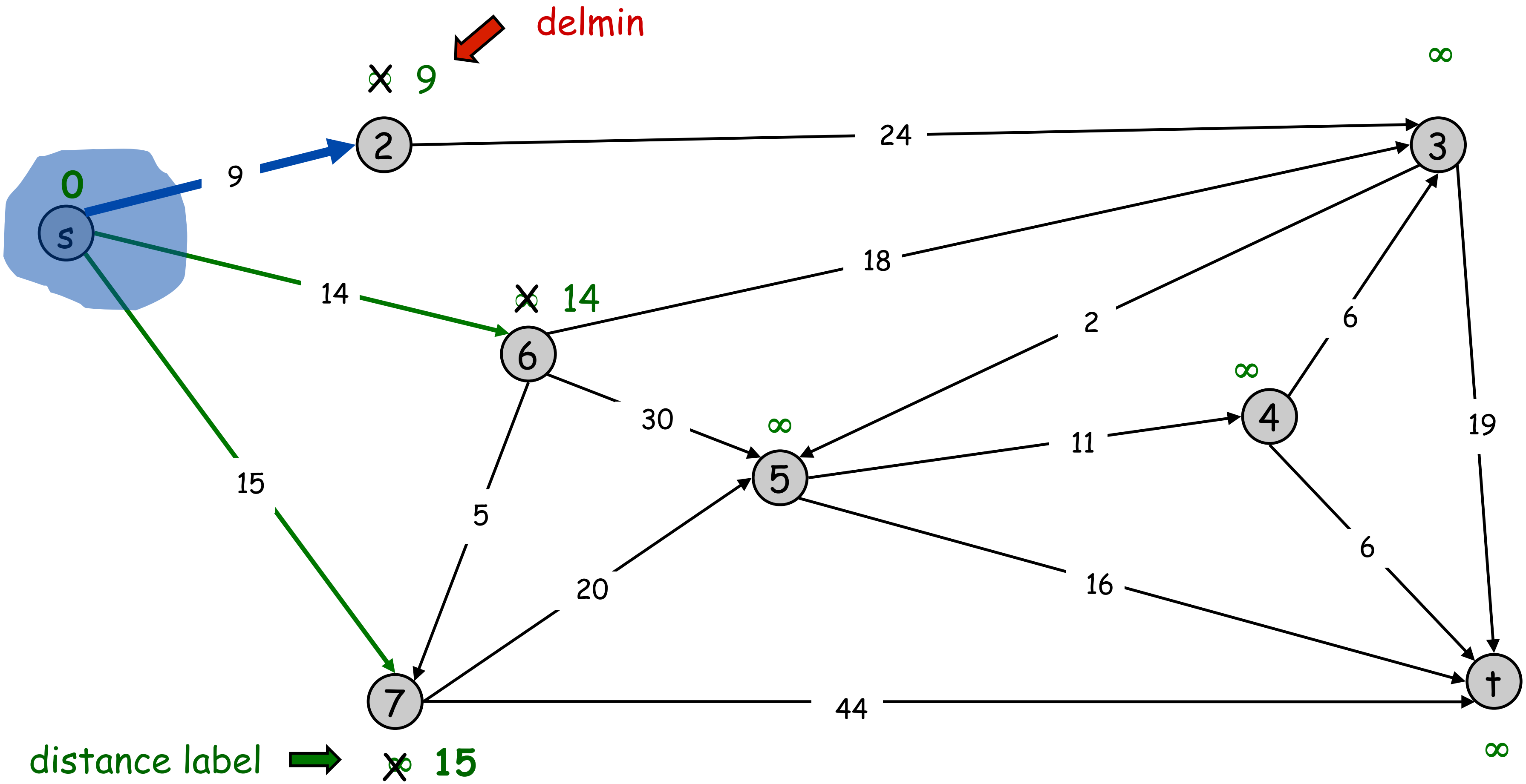
Dijkstra's Shortest Path Algorithm

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



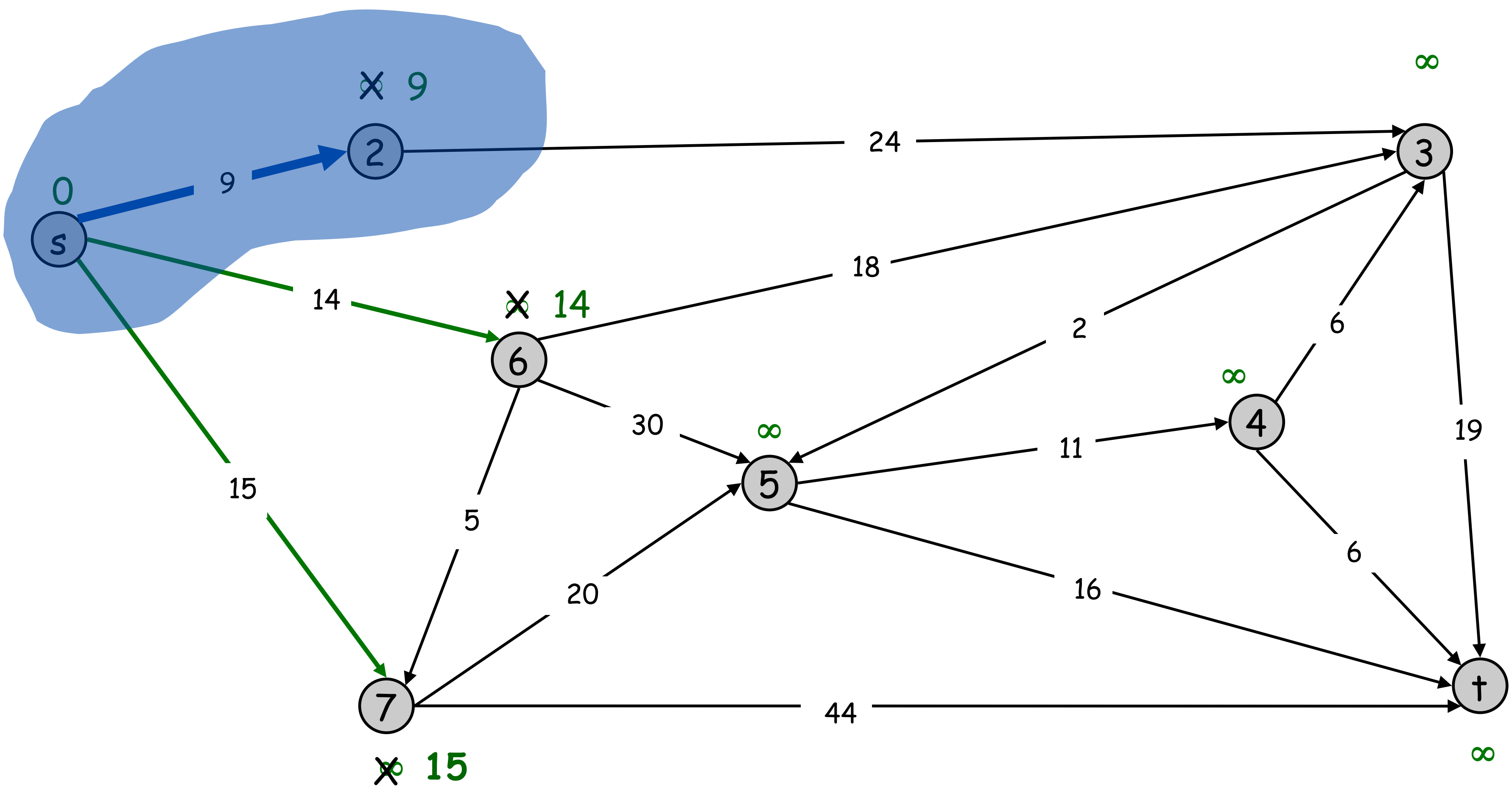
Dijkstra's Shortest Path Algorithm

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



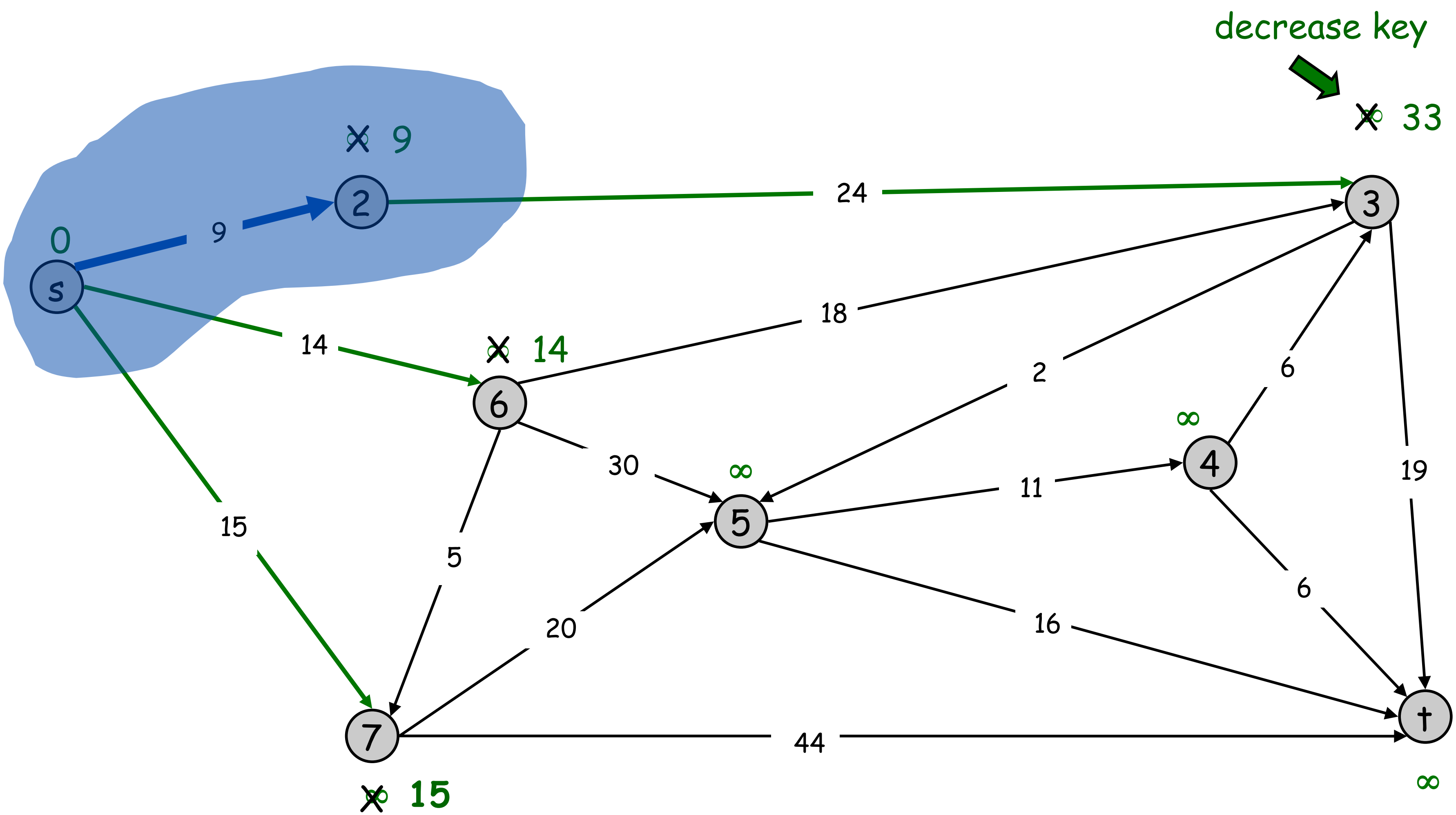
Dijkstra's Shortest Path Algorithm

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



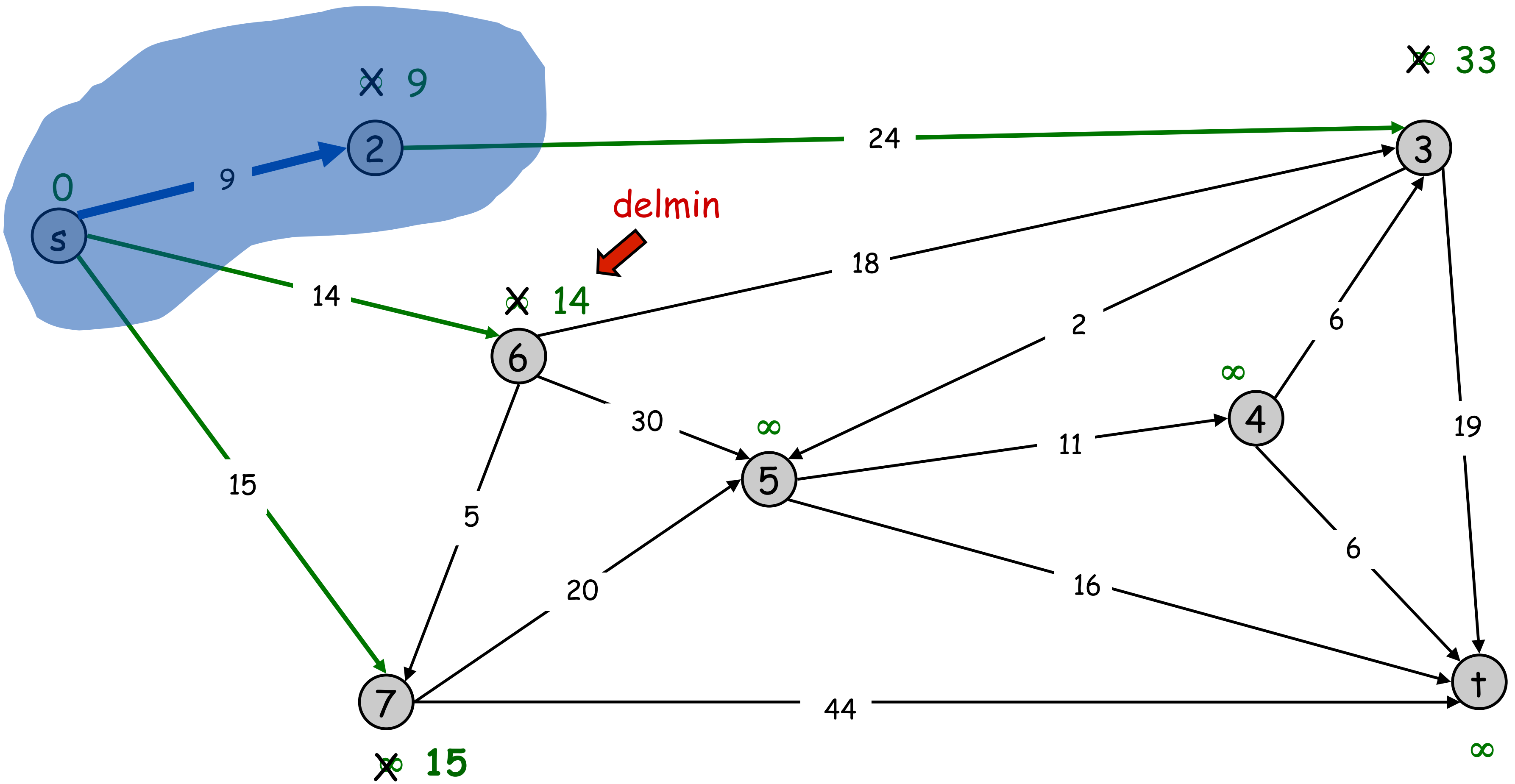
Dijkstra's Shortest Path Algorithm

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



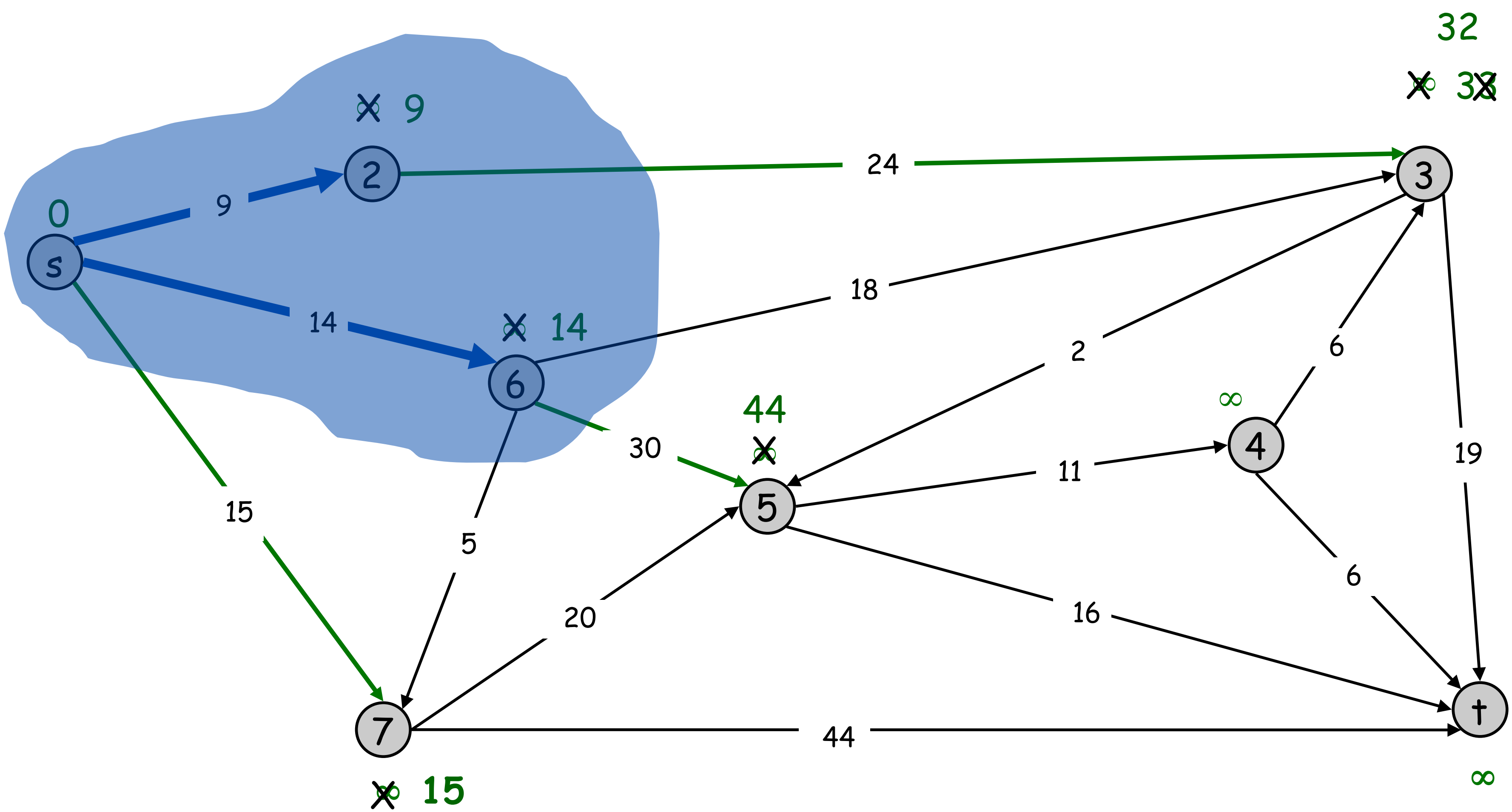
Dijkstra's Shortest Path Algorithm

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



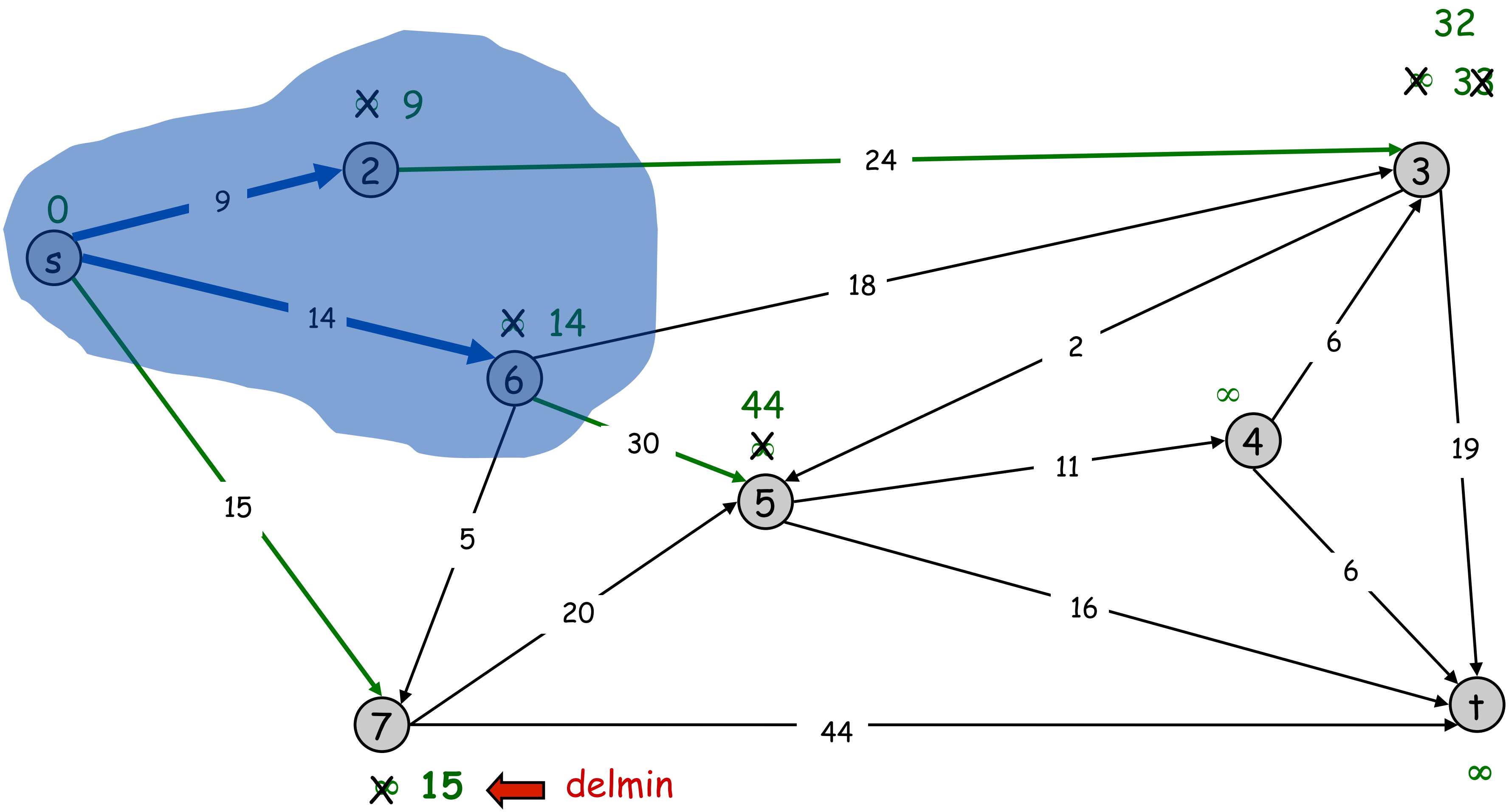
Dijkstra's Shortest Path Algorithm

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



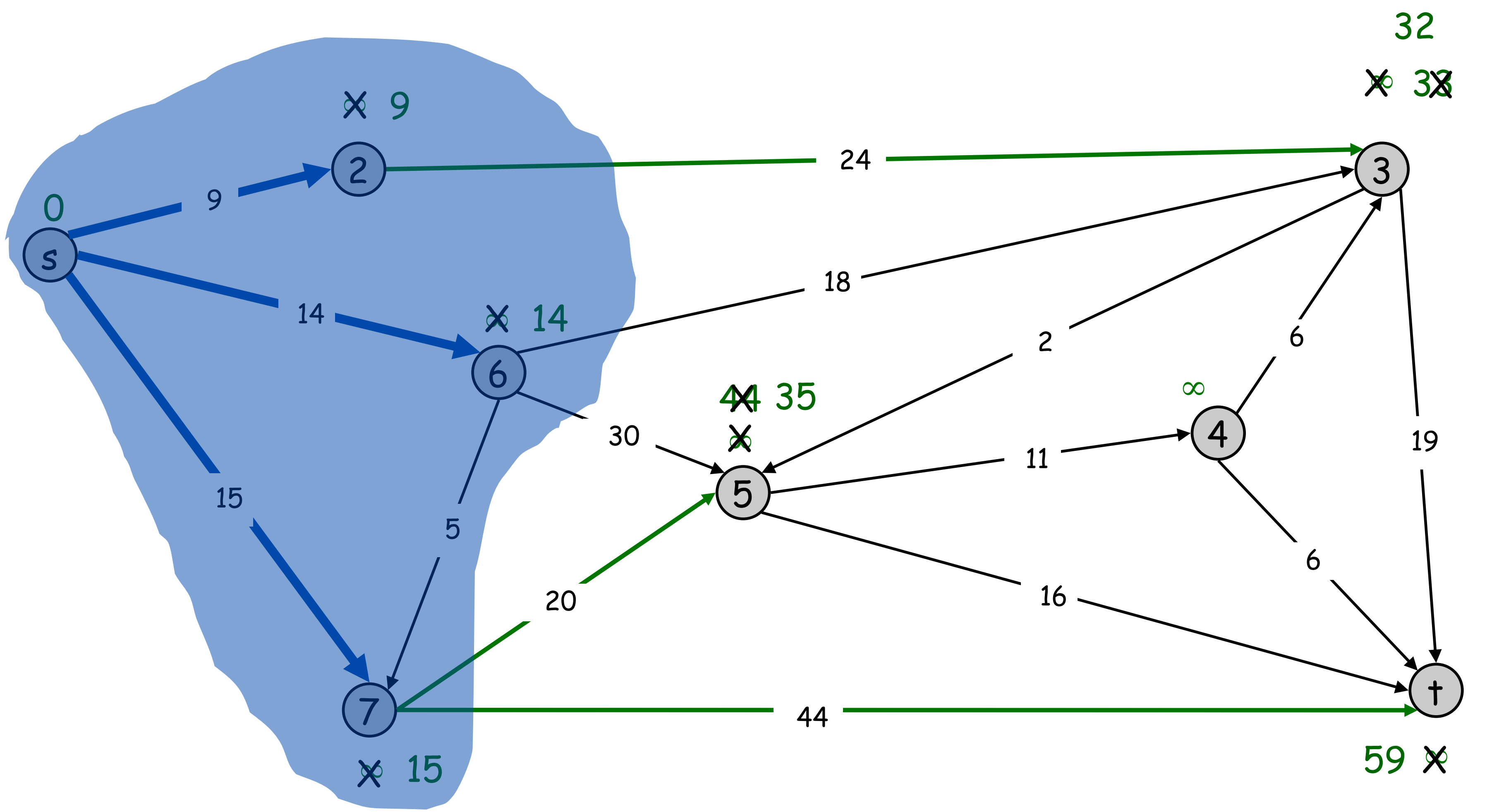
Dijkstra's Shortest Path Algorithm

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



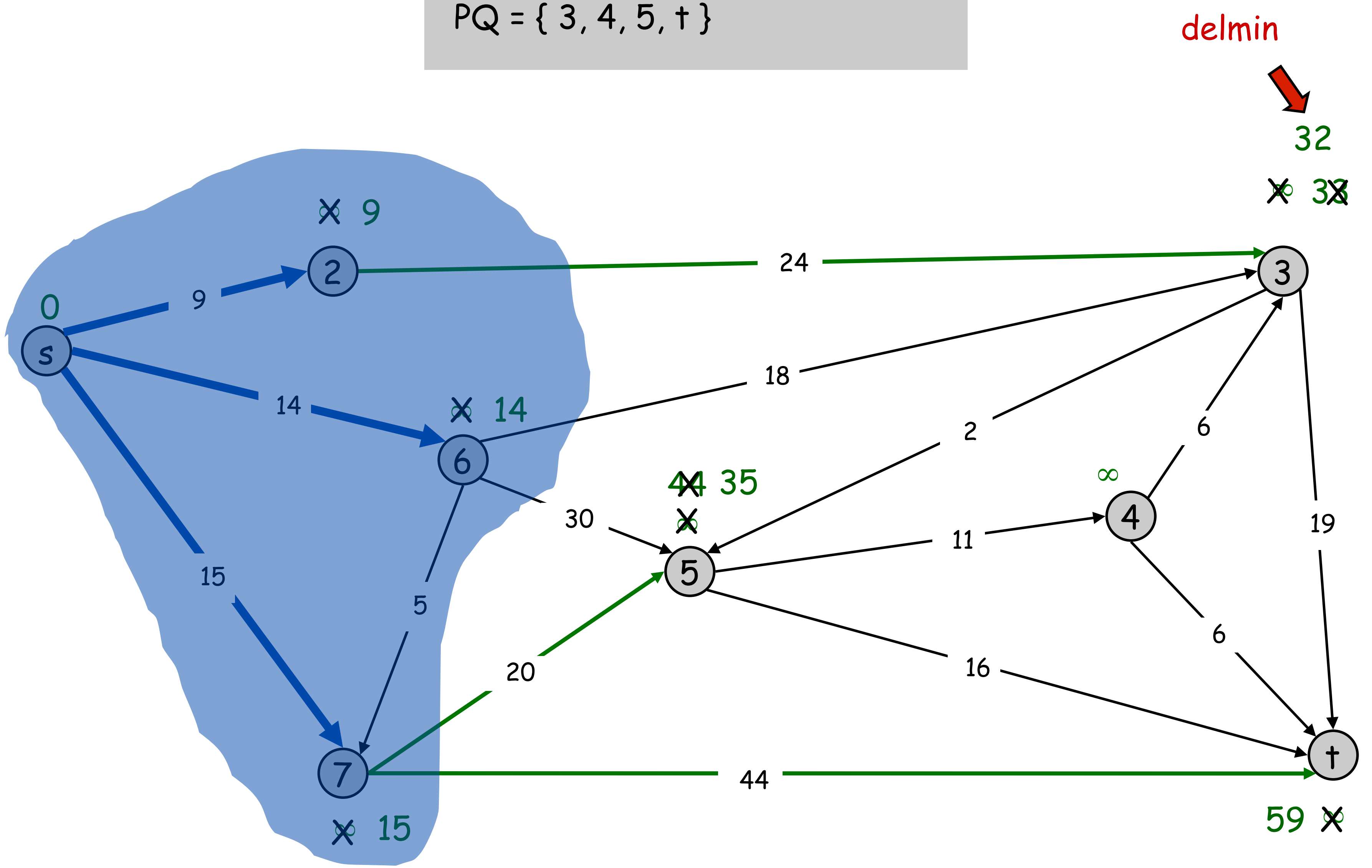
Dijkstra's Shortest Path Algorithm

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



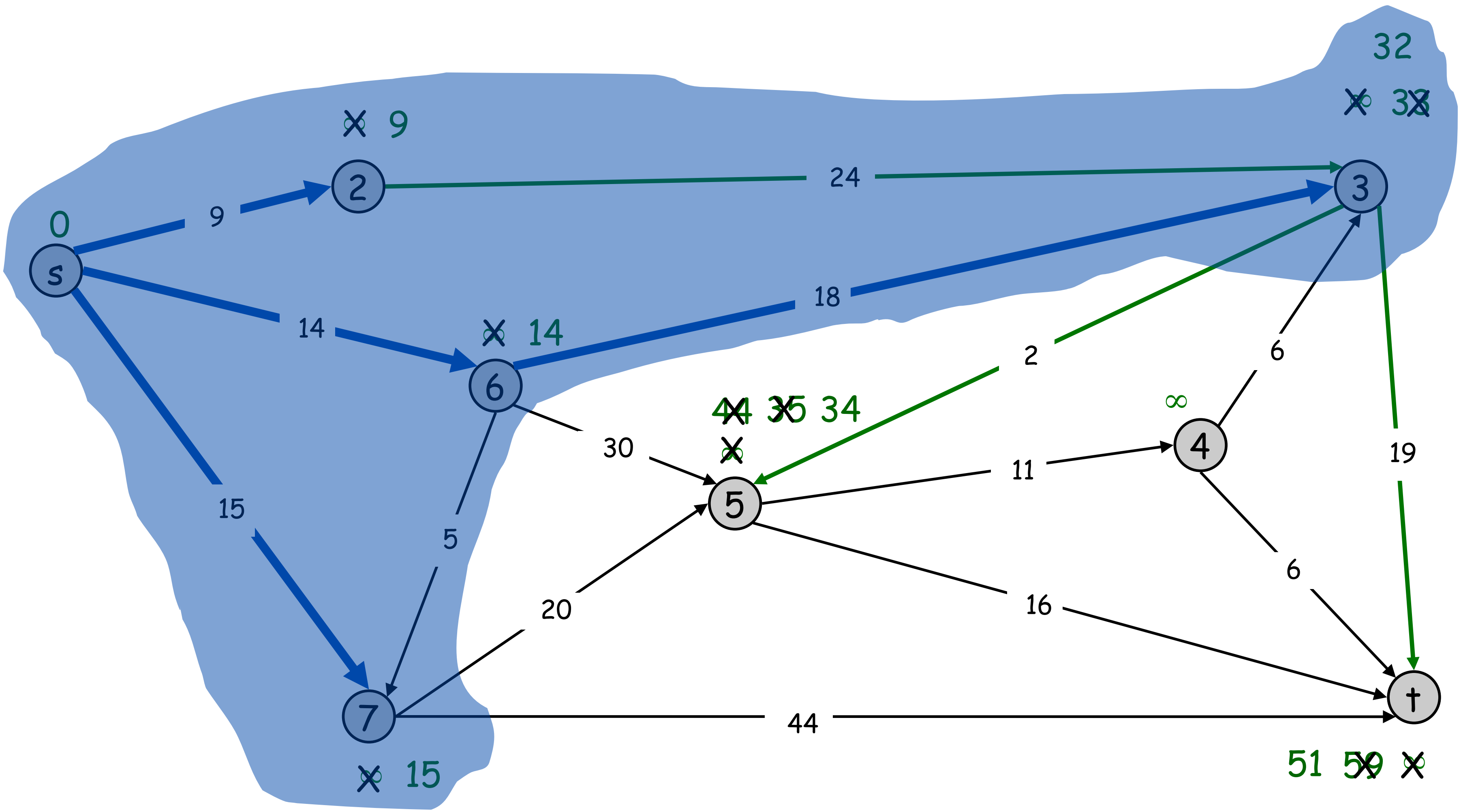
Dijkstra's Shortest Path Algorithm

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



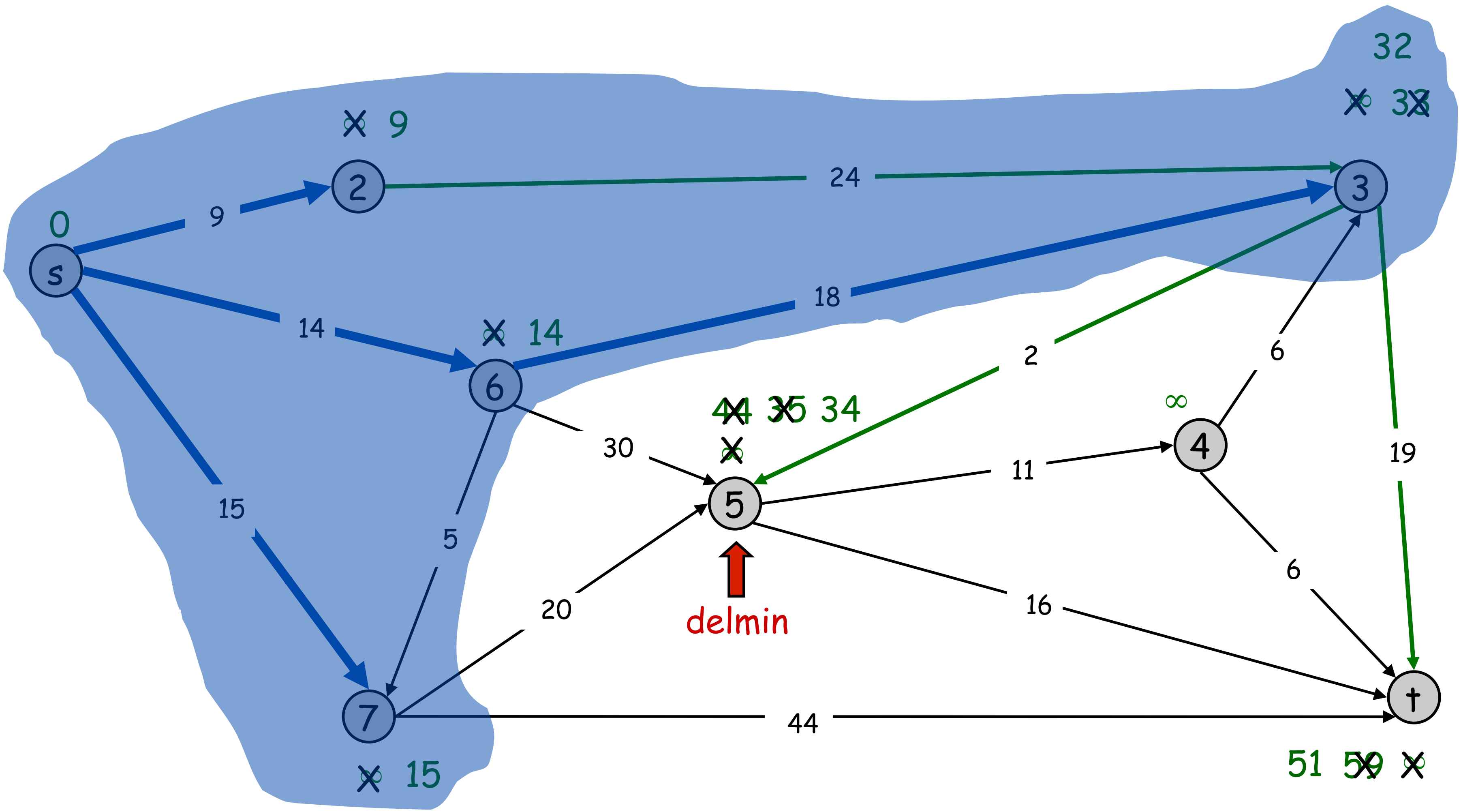
Dijkstra's Shortest Path Algorithm

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



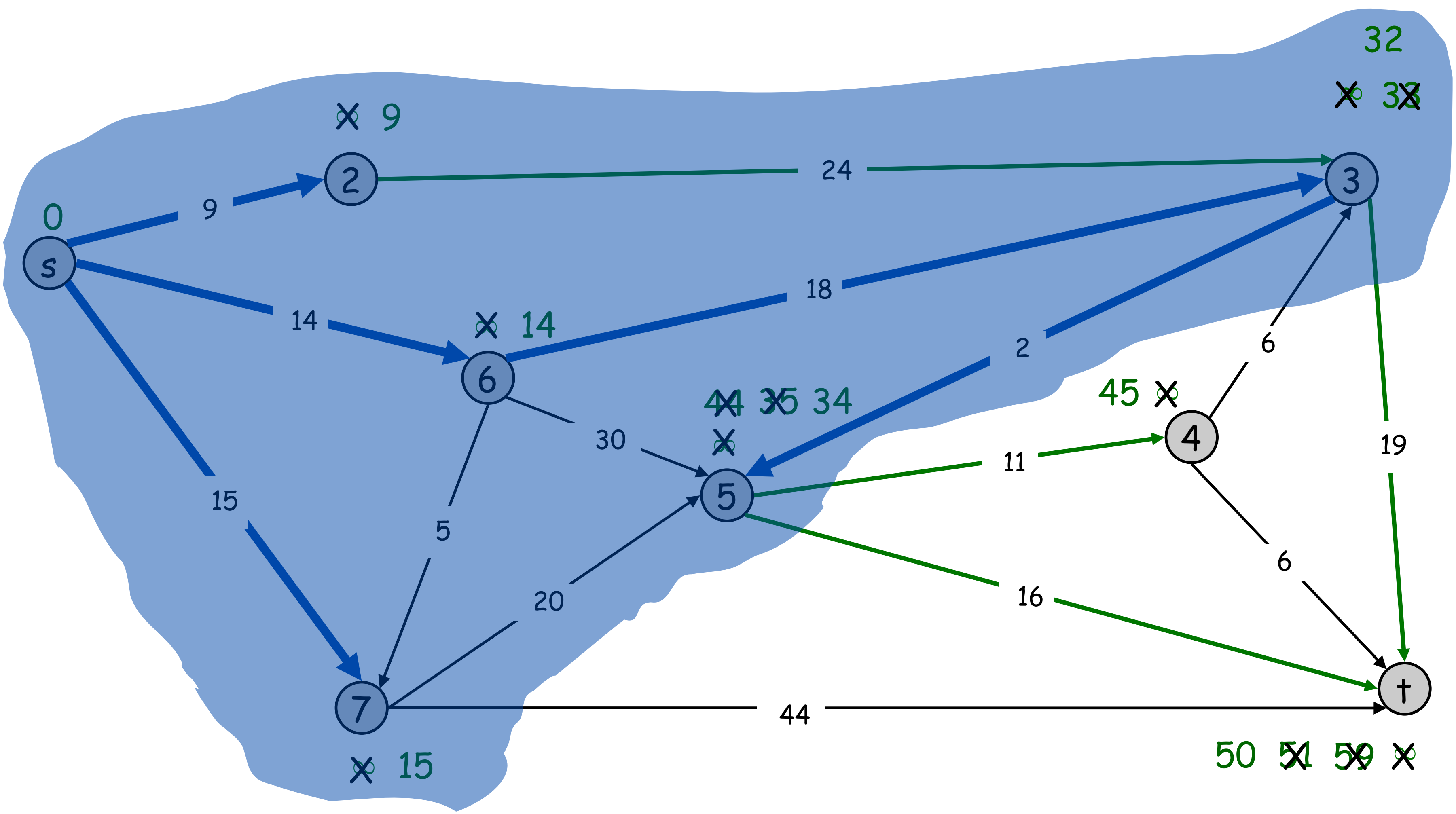
Dijkstra's Shortest Path Algorithm

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



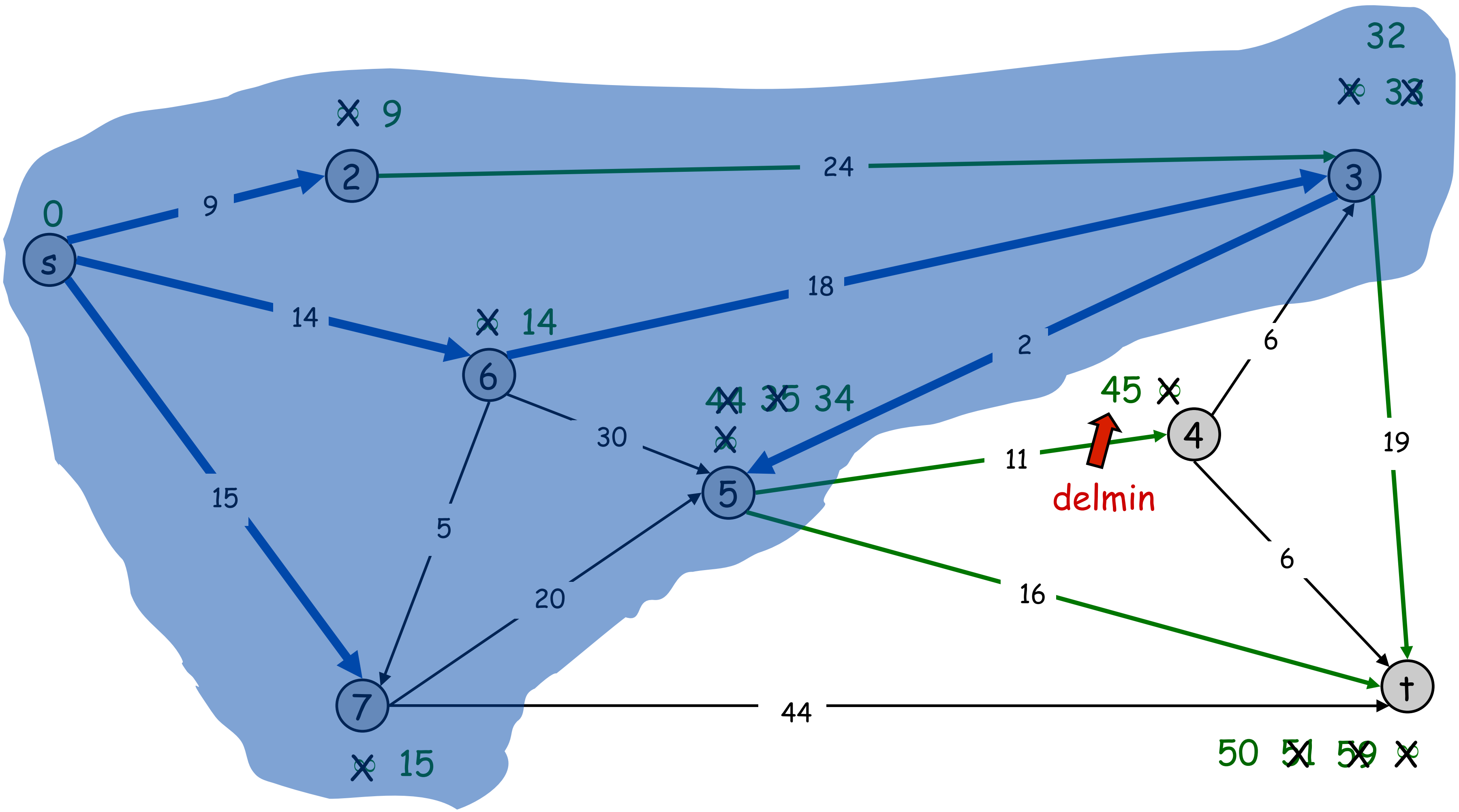
Dijkstra's Shortest Path Algorithm

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



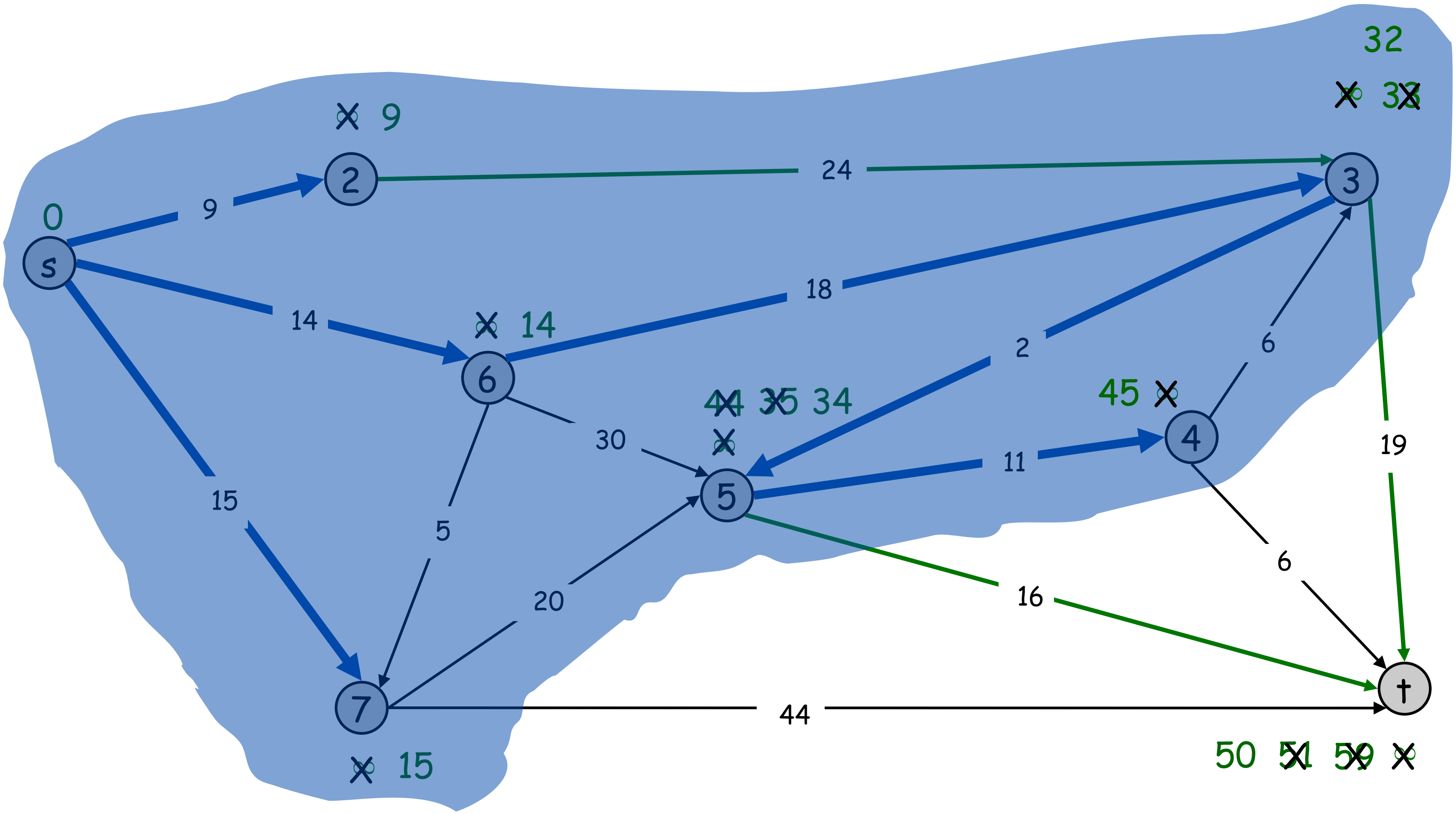
Dijkstra's Shortest Path Algorithm

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



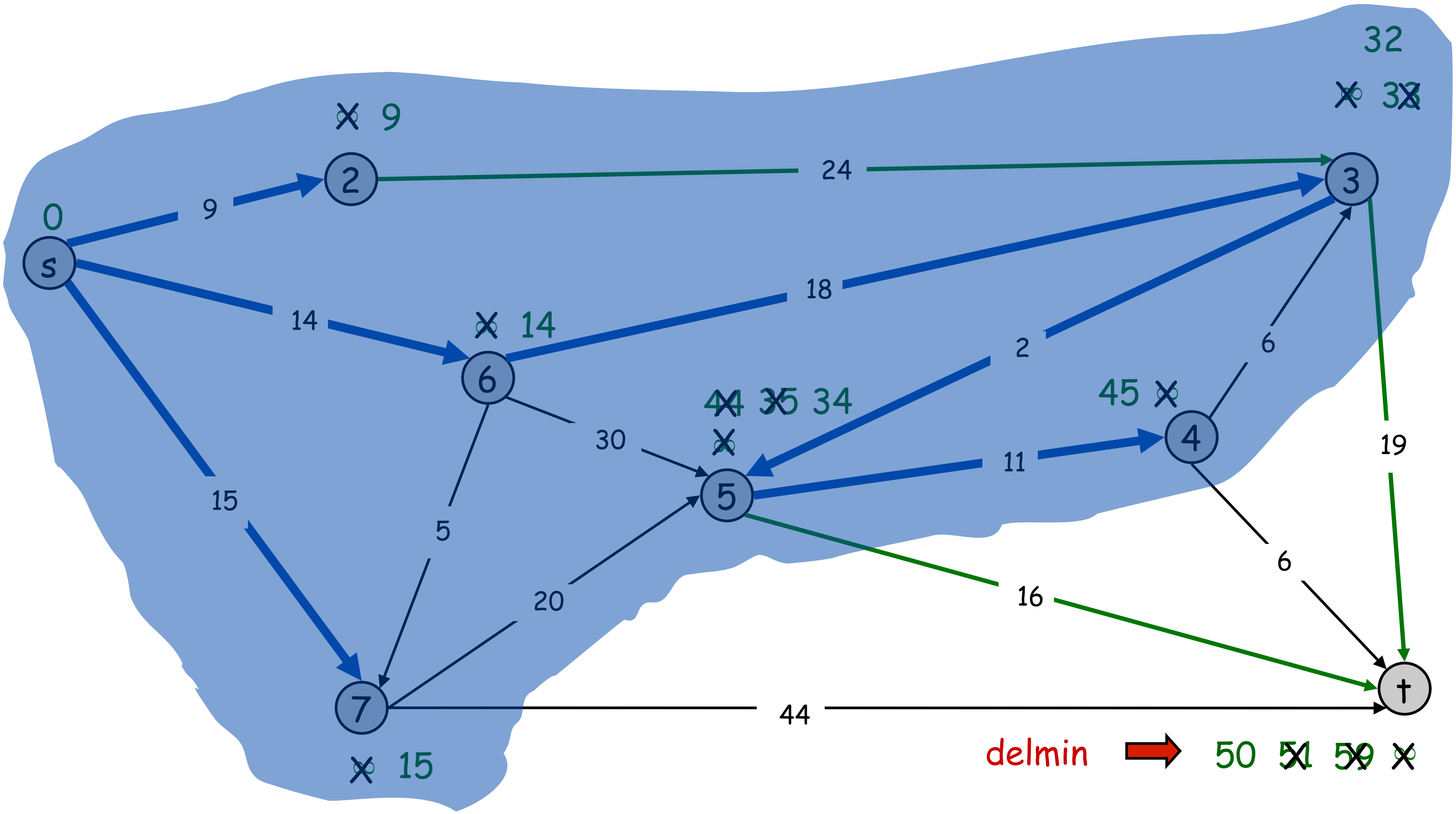
Dijkstra's Shortest Path Algorithm

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



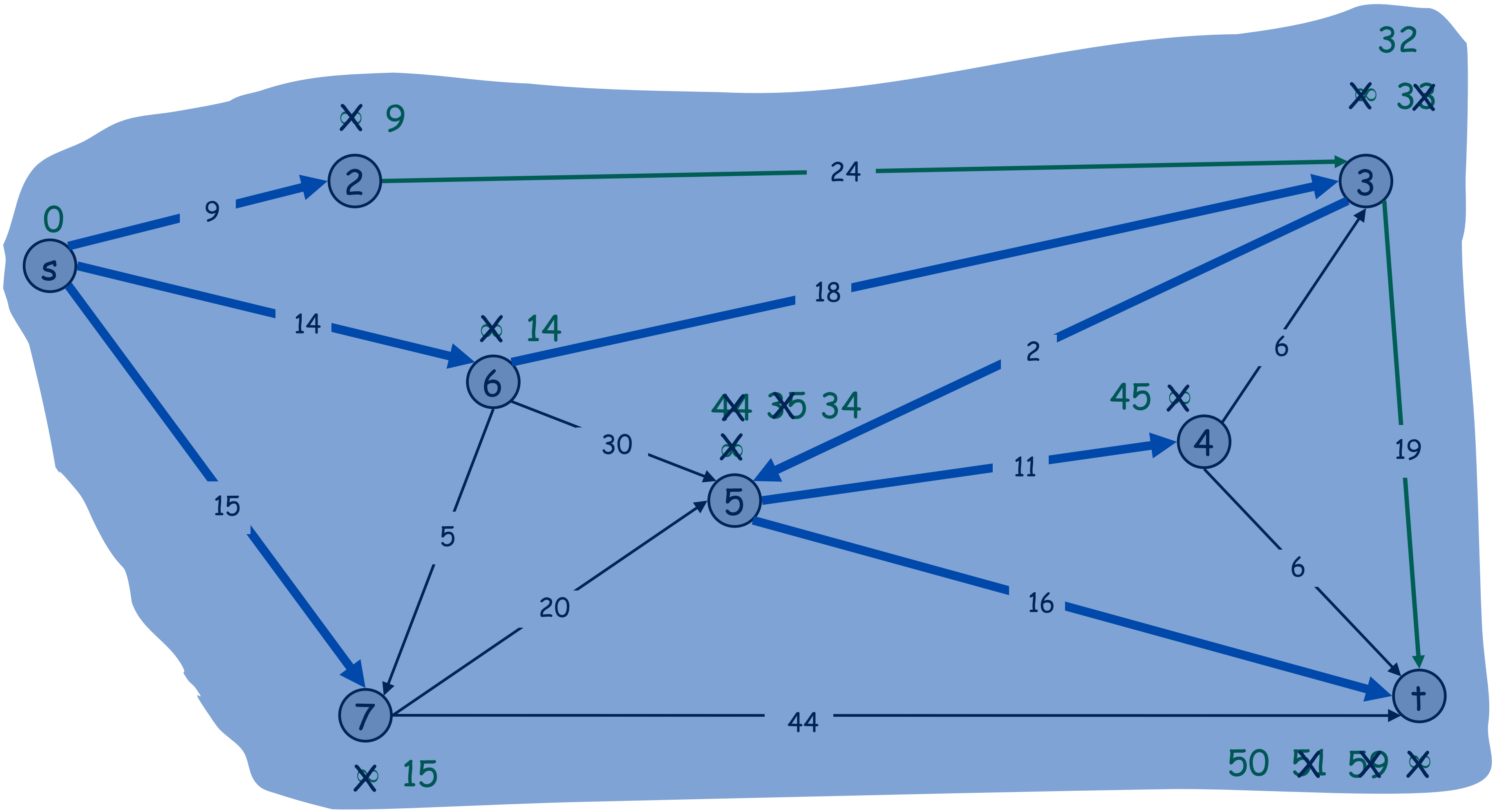
Dijkstra's Shortest Path Algorithm

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



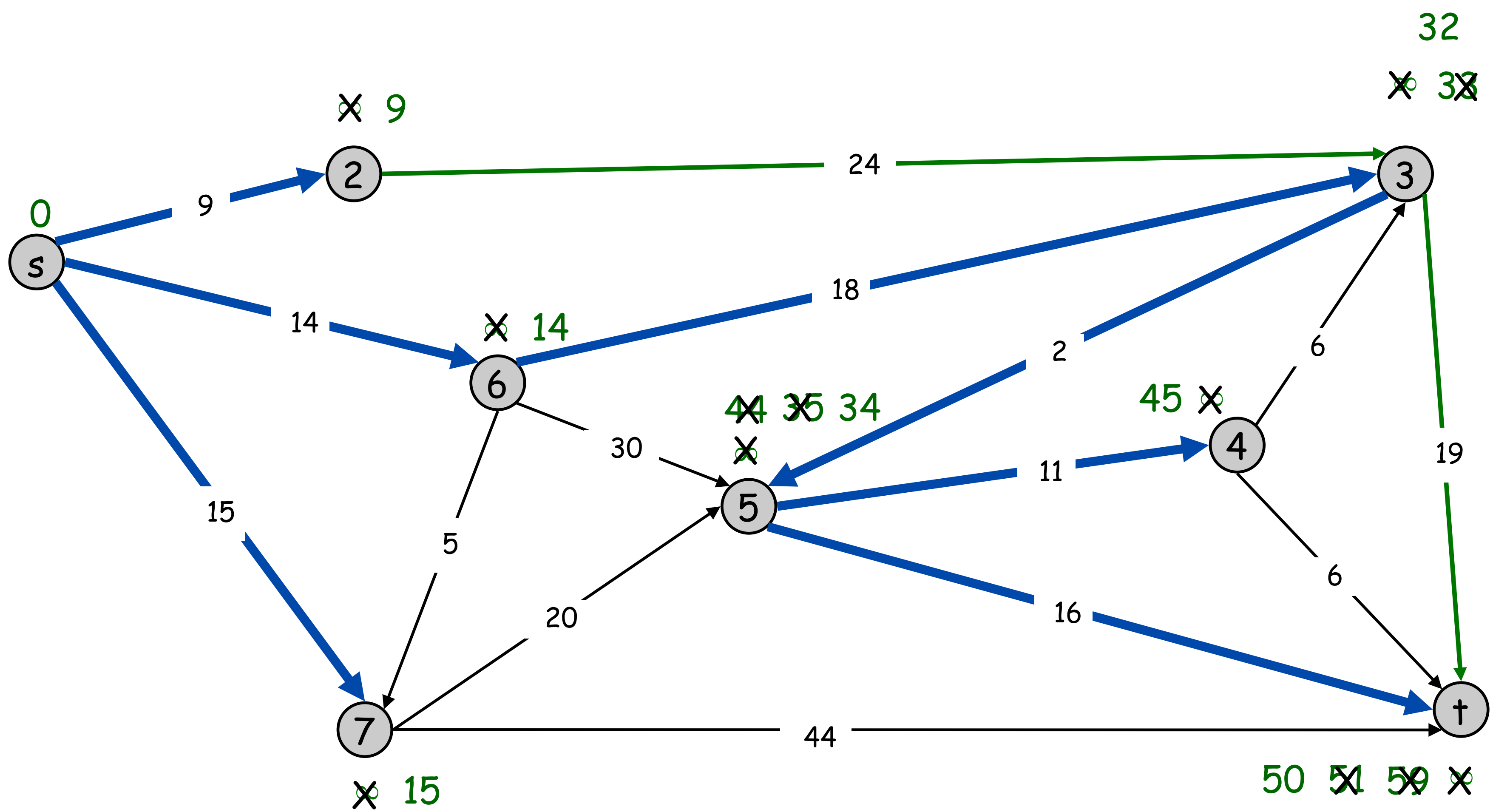
Dijkstra's Shortest Path Algorithm

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: Implementation

```
procedimento Dijkstra(origem: TVertice, var G: TGrafo)
variáveis
   D: vetor[TVertice] de TPeso;
   w: TVertice;
   S, V: conjunto de TVertice;
início
   S := {origem};
   V := {todos os vértices de G};
   D[origem] := 0;
   para i:=1 até G.NumVertices faça
   início
        se i != origem e existe a aresta (origem, i) então
                D[i] := Peso da aresta (origem, i)
        senão
                D[i] := ∞;
   fim;
   enquanto S ≠ V faça
   início
        encontre um vértice w ∈ V - S tal que D[w] é mínimo;
        S := S ∪ {w};
        para todo v adjacente a w faça
                D[v] := min(D[v],D[w] + Peso da aresta (w,v));
   fim;
fim;
```

Dijkstra's Algorithm: Implementation

```
void dijkstra(tvertice v, tgrafo *grafo) {
    std::priority_queue<taresta> heap;
    tpeso d, peso;
    tvertice w;
    tapontador p;
    int marc[MAXNUMVERTICES];
    int i;

    for (i = 0; i < grafo->num_vertices; i++) {
        grafo->custo[i] = INFINITO;
        grafo->antecessor[i] = NULO;
        marc[i] = BRANCO;
    }
    grafo->custo[v] = 0.0;
    heap.push(cria_aresta(0.0, v));
```

Dijkstra's Algorithm: Implementation

```
void dijkstra(tvertice v, tgrafo *grafo) {
    std::priority_queue<taresta> heap;
    tpeso d, peso;
    tvertice w;
    tapontador p;
    int marc[MAXNUMVERTICES];
    int i;

    for (i = 0; i < grafo->num_vertices; i++) {
        grafo->custo[i] = INFINITO;
        gra
        mar
    }
    grafo->
    heap.pus
```

```
struct taresta{
    tpeso peso;
    tvertice dest;

    bool operator<(const taresta &a) const {
        return peso>a.peso;
    }
};
```

Dijkstra's Algorithm: Implementation

```
void dijkstra(tvertice v, tgrafo *grafo) {
    std::priority_queue<taresta> heap;
    tpeso d, peso;
    tvertice w;
    tapontador p;
    int marc[MAXNUMVERTICES];
    int i;

    for (i = 0; i < grafo->num_vertices; i++) {
        grafo->custo[i] = INFINITO;
        grafo->antecessor[i] = NULO;
        marc[i] = BRANCO;
    }
    grafo->custo[v] = 0.0;
    heap.push(cria_aresta(0.0, v));
```

Dijkstra's Algorithm: Implementation

```
void dijkstra(tvertice v, tgrafo *grafo) {
    std::priority_queue<taresta> heap;
    tpes
    tver
    tapo
    int
    int

    for
        grafo->custo[i] = INFINITO;
        grafo->antecessor[i] = NULO;
        marc[i] = BRANCO;
    }
    grafo->custo[v] = 0.0;
    heap.push(cria_aresta(0.0, v));
```

```
taresta cria_aresta(tpeso peso, tvertice dest) {
    taresta aresta;

    aresta.peso = peso;
    aresta.dest = dest;
    return aresta;
}
```

Dijkstra's Algorithm: Implementation

```
    while (!heap.empty()) {
        v = heap.top().dest; heap.pop();
        if (marc[v] == PRETO) continue;
        marc[v] = PRETO;
        p = primeiro_adj(v, grafo);
        while (p != NULO) {
            recupera_adj(v, p, &w, &peso, grafo);
            d = grafo->custo[v] + peso;
            if (cmp(d, grafo->custo[w]) < 0) {
                grafo->custo[w] = d;
                heap.push(cria_aresta(d, w));
                grafo->antecessor[w] = v;
            }
            p = proximo_adj(v, p, grafo);
        }
    }
}
```

Dijkstra's Algorithm: Implementation

```
    while (!heap.empty()) {
        v = heap.top().dest; heap.pop();
        if (marc[v] == PRETO) continue;
        marc[v] = PRETO;
        p = primeiro_adj(v, grafo);
        while (p != NULO) {
            recupera_adj(v, p, &w, &peso, grafo);
            d = grafo->custo[v] + peso;
            if (cmp(d, grafo->custo[w]) < 0) {
                grafo->custo[w] = d;
        }
    }
}
```

```
int cmp(double x, double y = 0, double tol = DBL_EPSILON) {
  return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}
```

Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain $\pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e$.

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update

$$\pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}.$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	n	n	log n	$d \log_d n$	1
ExtractMin	n	n	log n	$d \log_d n$	log n
ChangeKey	m	1	log n	$\log_d n$	1
IsEmpty	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

† Individual ops are amortized bounds