Data Structure and Algorithm (Graph)

Dr. Sambit Bakshi

Graph

- A graph is a collection of nodes (or vertices) and edges (or arcs)
  - Each node contains an element
  - Each edge connects two nodes together (or possibly the same node to itself) and may contain an edge attribute

- A directed graph is one in which the edges have a direction
- An undirected graph (*graph*) is one in which the edges do not have a direction

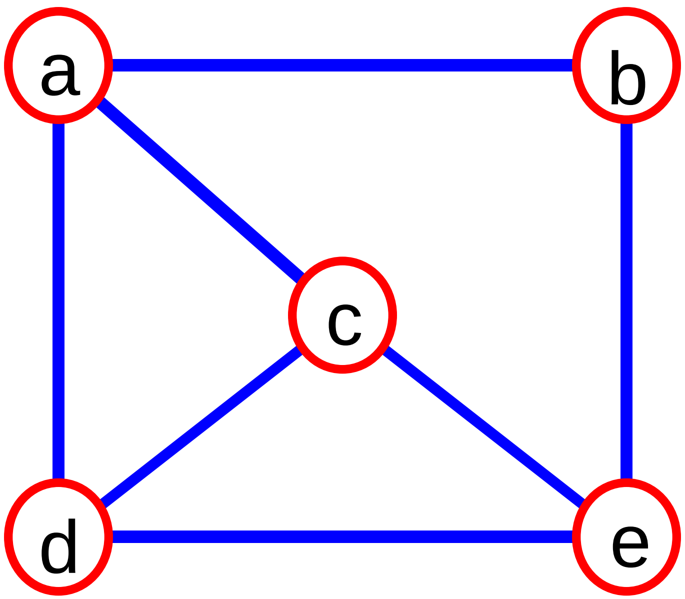
What is a Graph?

- A graph $G = (V,E)$ is composed of:

  V: set of vertices

  E: set of edges connecting the vertices in V

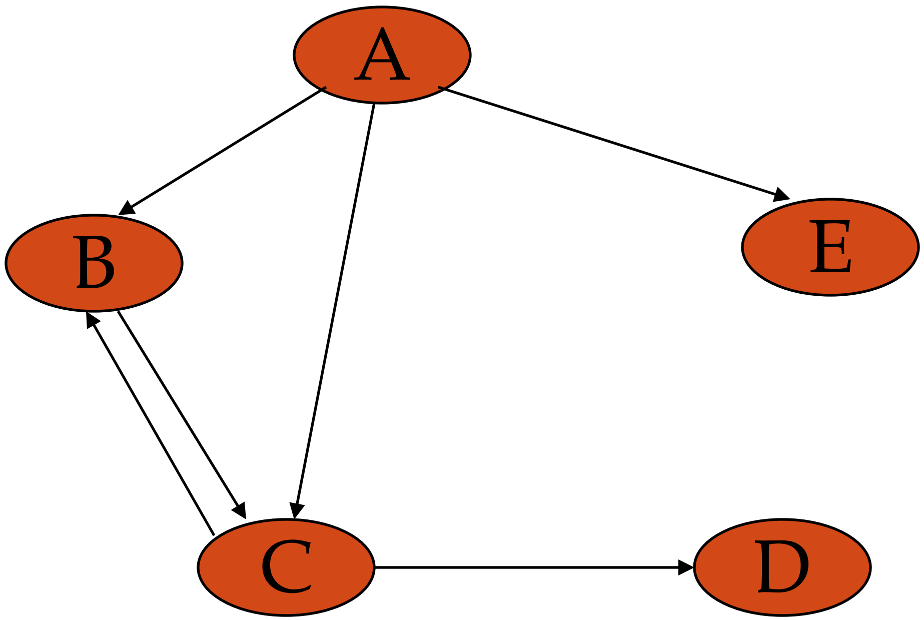
- An edge $e = (u,v)$ is a pair of vertices
- Example:



$V= \{a,b,c,d,e\}$

$E= \{(a,b),(a,c),(a,d),$
$(b,e),(c,d),(c,e),(d,e)\}$

a digraph (Oriented or Directed graph)



$V = \{A, B, C, D, E\}$

$E = \{\langle A,B\rangle, \langle B,C\rangle, \langle C,B\rangle, \langle A,C\rangle, \langle A,E\rangle, \langle C,D\rangle\}$

Directed vs Undirected graph

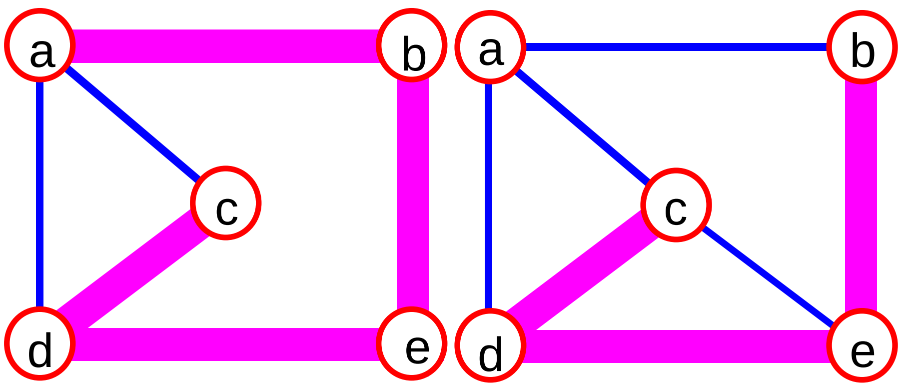
- An undirected graph is one in which the pair of vertices in a edge is unordered, (v0, v1) = (v1,v0)
- A directed graph is one in which each edge is a directed pair of vertices, <v0, v1> != <v1,v0>

Graph terminologies

- The size of a graph is the number of *nodes* in it
- The empty graph has size zero (no nodes)
- If two nodes are connected by an edge, they are neighbors (and the nodes are adjacent to each other)
- The degree of a node is the number of edges it has
- For directed graphs,
  - If a directed edge goes from node S to node D, we call S the source and D the destination of the edge
    - The edge is an out-edge of S and an in-edge of D
    - S is a predecessor of D, and D is a successor of S
  - The in-degree of a node is the number of in-edges it has
  - The out-degree of a node is the number of out-edges it has

Graph terminologies

- path: sequence of vertices $v_1, v_2, v_3. . . v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.

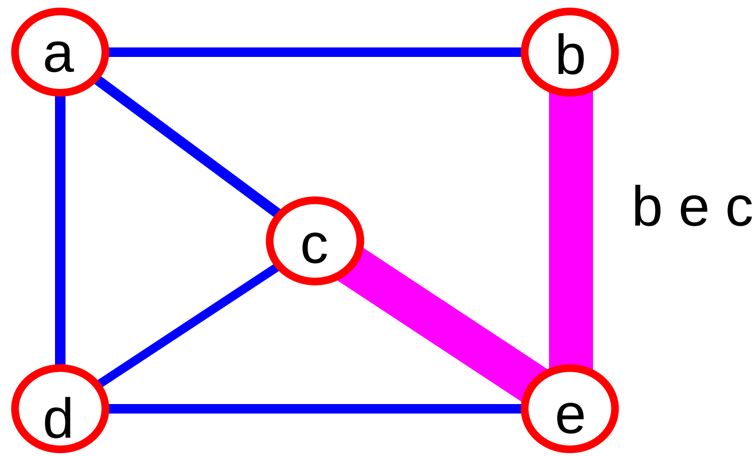


a b e d c

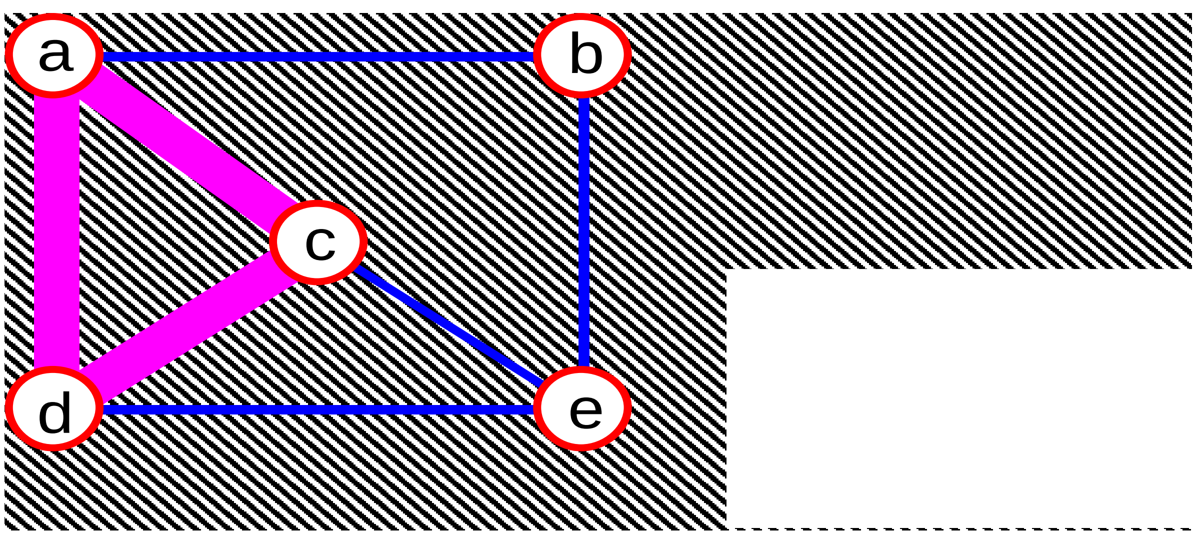
b e d c

More Terminology

- simple path: no repeated vertices

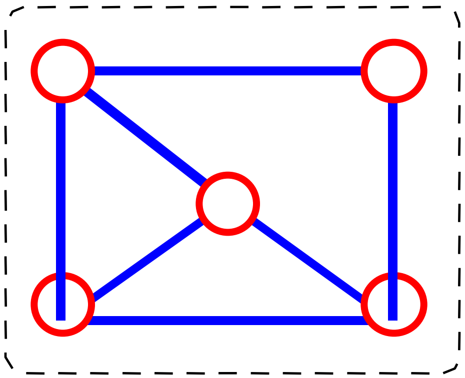


- cycle: simple path, except that the last vertex is the same as the first vertex

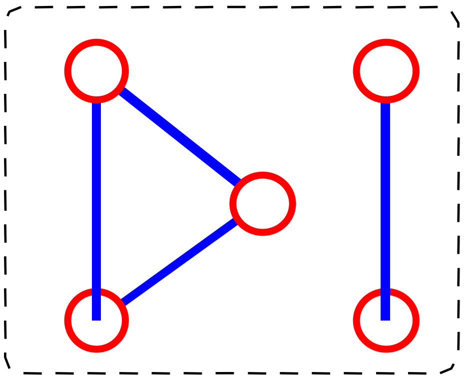


Even More Terminology

- connected graph: any two vertices are connected by some path



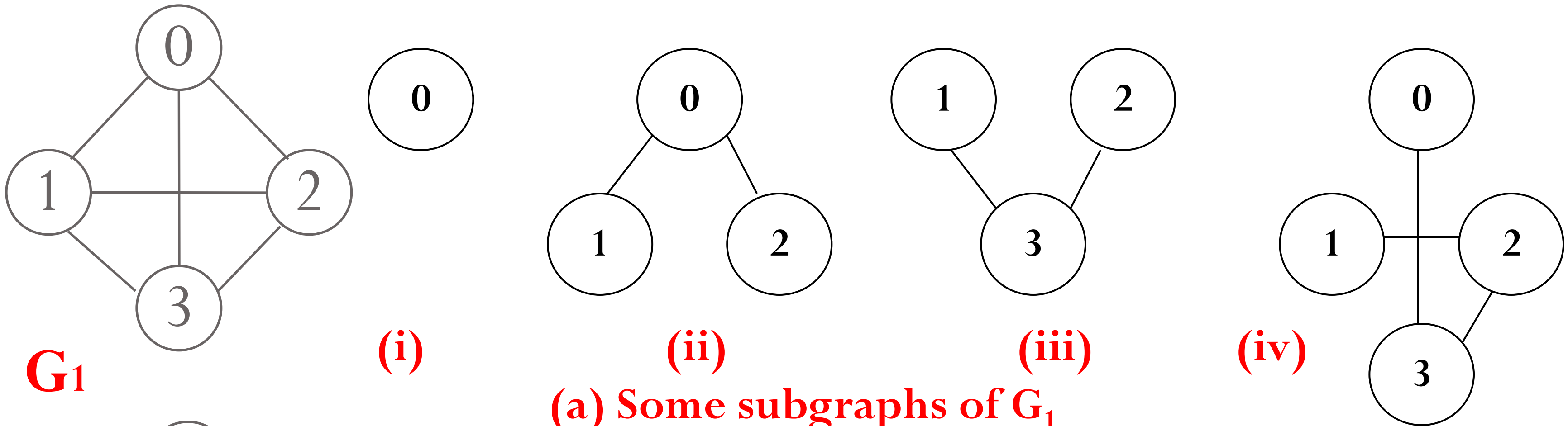
connected



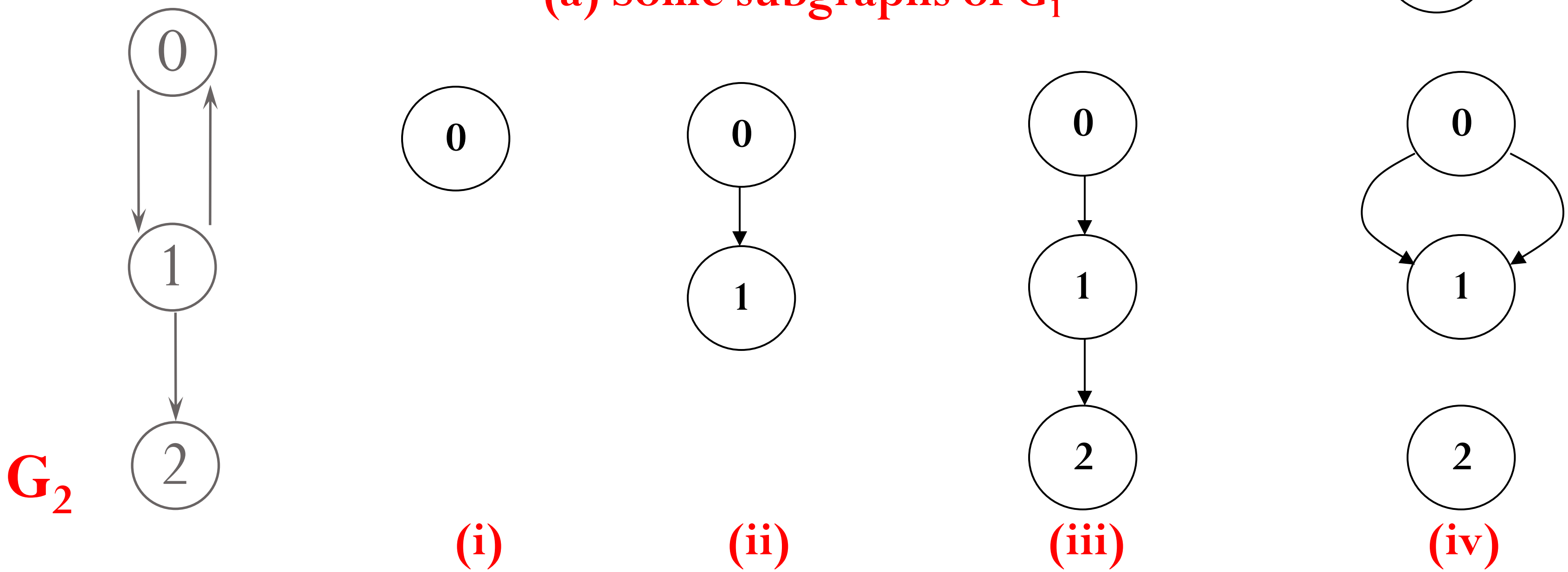
not connected

- subgraph: subset of vertices and edges forming a graph

Subgraph Examples



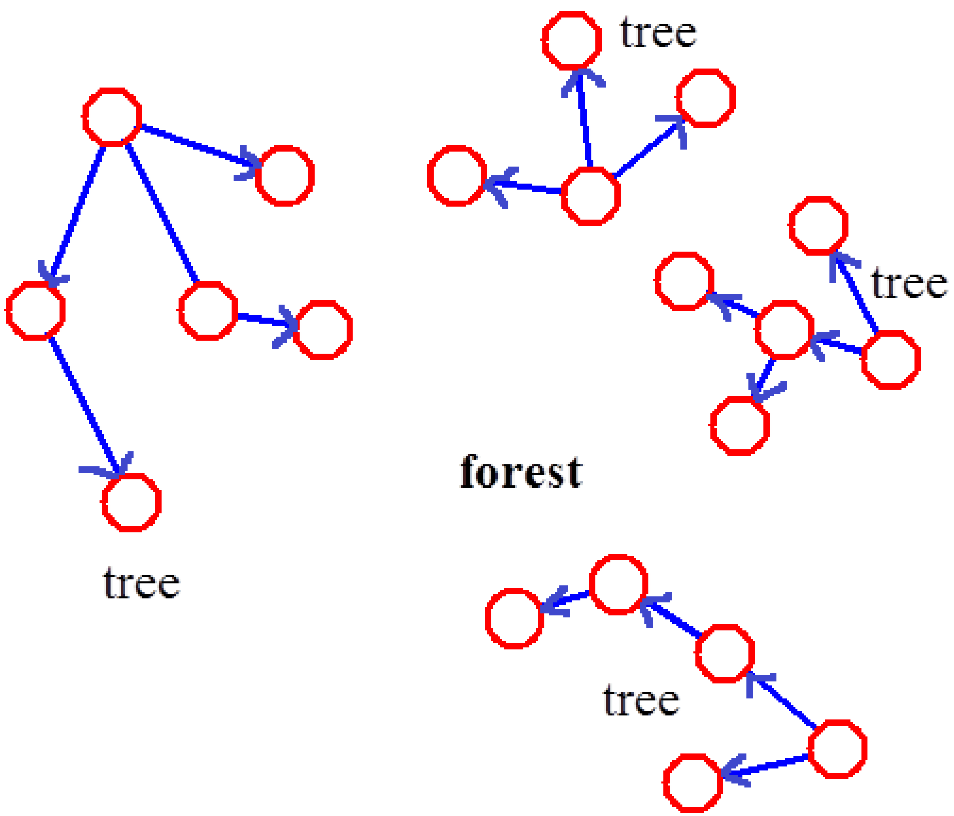
(a) Some subgraphs of G_1



(b) Some subgraphs of G_2

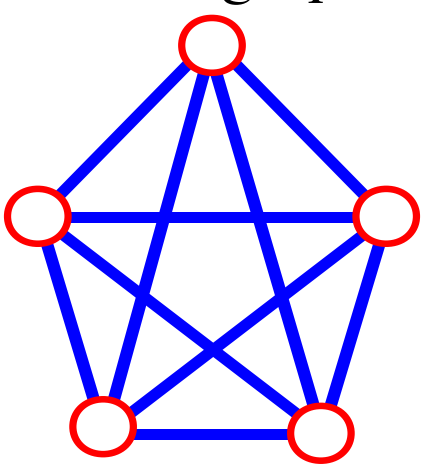
More…

- tree - connected digraph without cycles
- forest - collection of trees



Connectivity

- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice! Therefore, intuitively, m = **n**(**n** - 1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



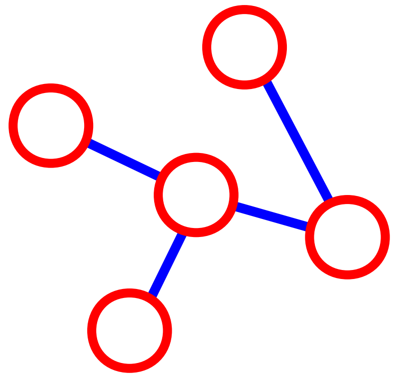
n = 5
m = (5 ∗ 4)/2 = 1(

More on Connectivity

n = #vertices

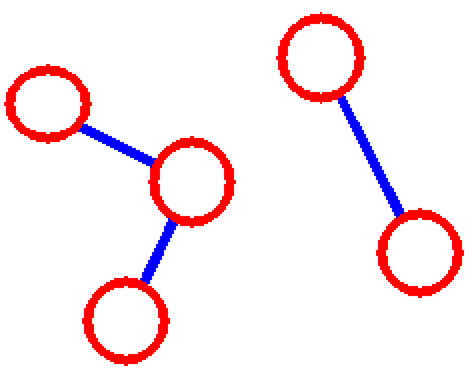
m = #edges

- For a tree **m** = **n** - 1

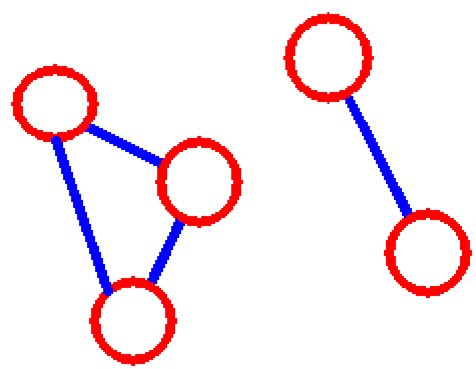


n = 5
m = 4

If **m** < **n** - 1, G is not connected, but the reverse is not necessarily true, i.e. m !< n-1 does not necessary confirm that G is connected.



n = 5
m = 3



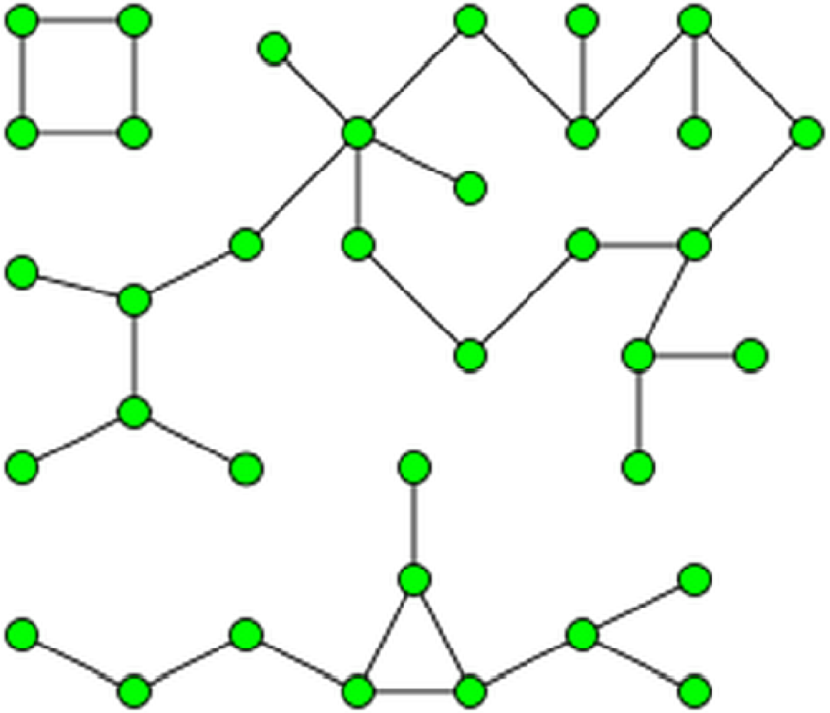
n = 5
m = 4

Graph Terminologies

- An undirected graph is **connected** if there is a path from every node to every other node
- A *directed graph* is **strongly connected** if there is a path from every node to every other node
- A directed graph is **weakly connected** if it is not strongly connected but the underlying undirected graph is connected
- Node X is **reachable** from node Y if there is a path from Y to X

Graph Terminologies

- **Connected component (or Component)**: of a graph is a subgraph which is connected



Graph with 3 connected components

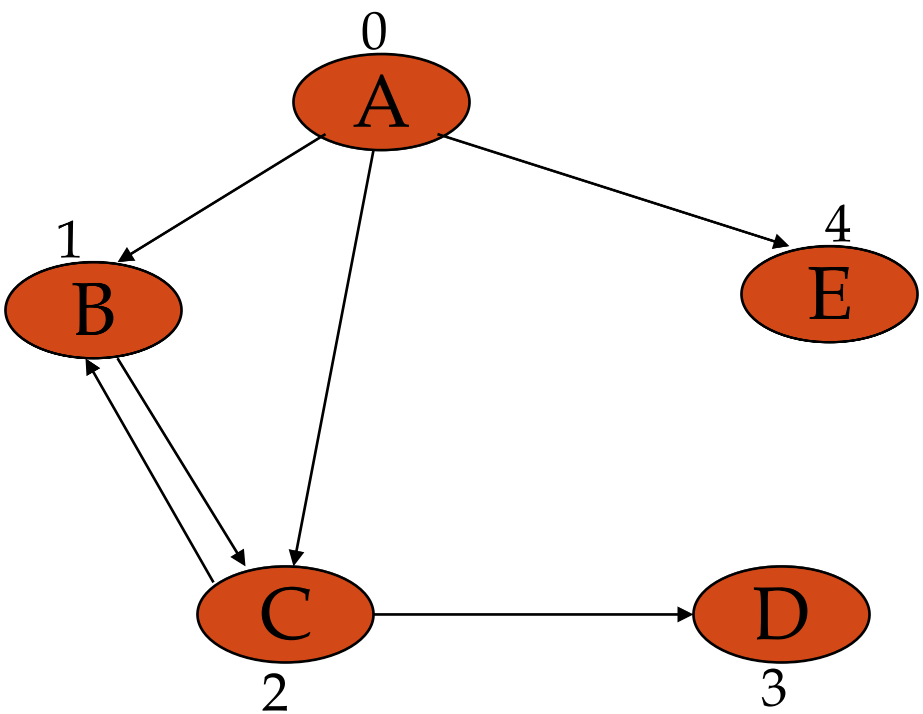
graph data structures

- storing the vertices
  - each vertex has a unique identifier and, maybe, other information
  - for efficiency, associate each vertex with a number that can be used as an index
- storing the edges
  - adjacency matrix – represent all possible edges
  - adjacency lists – represent only the existing edges

storing the vertices

- when a vertex is added to the graph, assign it a number
  - vertices are numbered between 0 and n-1
- graph operations start by looking up the number associated with a vertex
- many data structures to use
  - for small graphs a vector can be used
    - search will be O(n)

the vertex vector



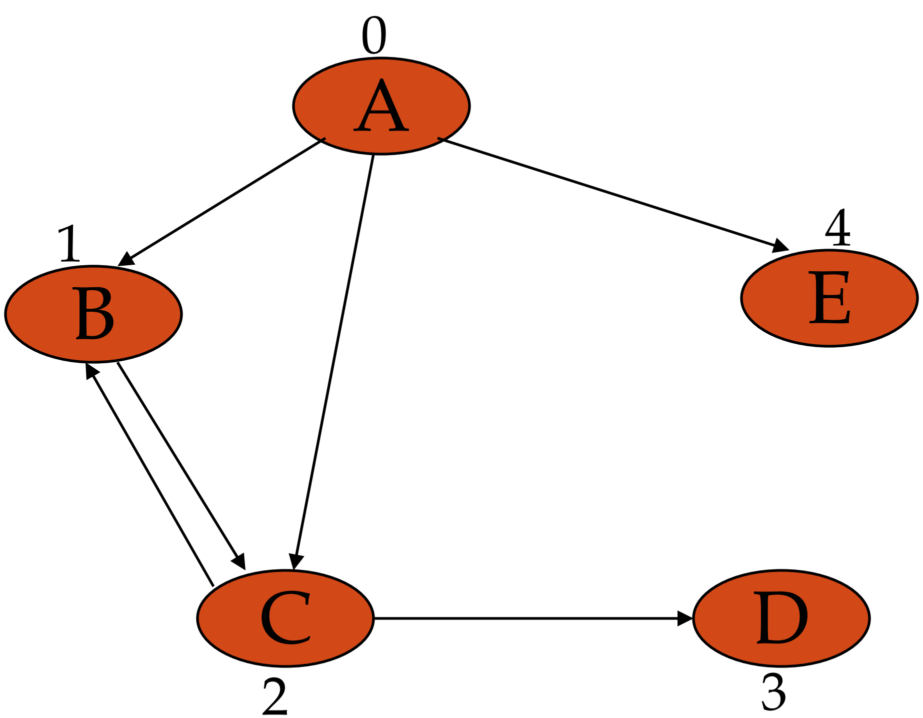
0	A
1	B
2	C
3	D
4	E

adjacency matrix

A_{nxn}, where n:# of vertices

A[i][j]=1 if (i,j) is an edge

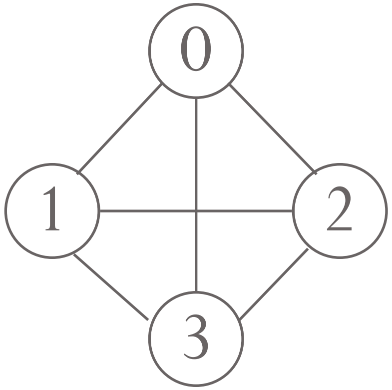
=0 otherwise



0	A
1	B
2	C
3	D
4	E

	0	1	2	3	4
0	0	1	1	0	1
1	0	0	1	0	0
2	0	1	0	1	0
3	0	0	0	0	0
4	0	0	0	0	0

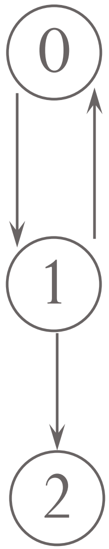
Examples for Adjacency Matrix



G_1

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

symmetric

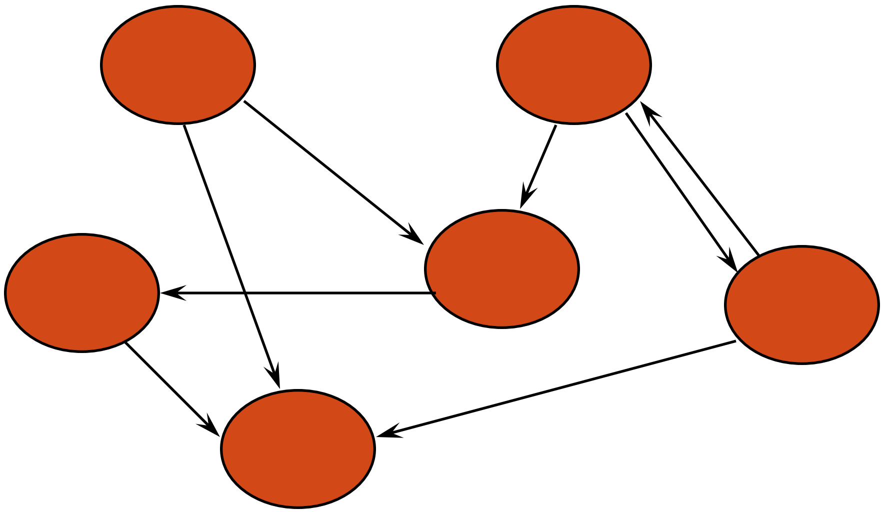


G_2

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Asymmetric

Space required

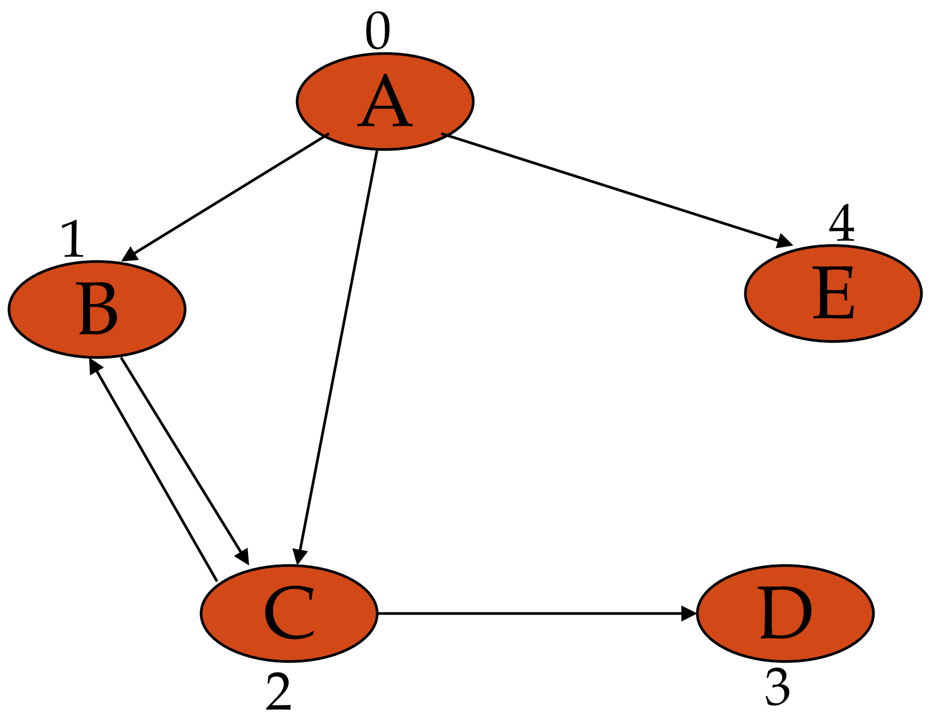


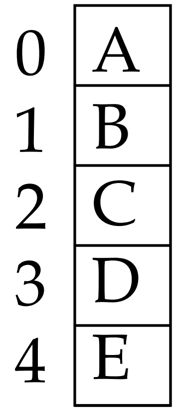
a n^2 matrix is needed for a graph with n vertices

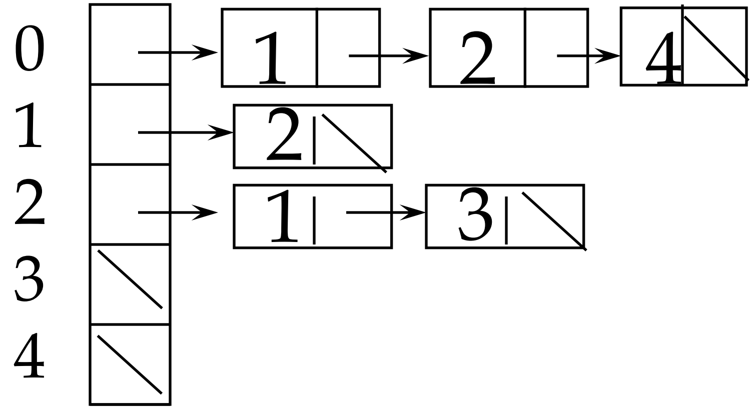
many graphs are “sparse”

- degree of “sparseness” is key factor in choosing a data structure for edges
  - adjacency matrix requires space for **all possible** edges
  - adjacency list requires space for existing edges only
- affects amount of memory space needed
- affects efficiency of graph operations

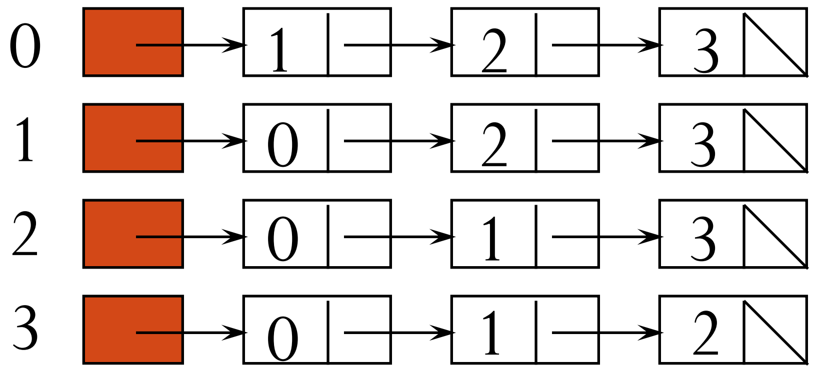
adjacency lists

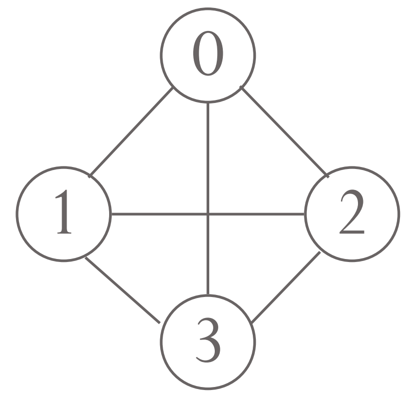




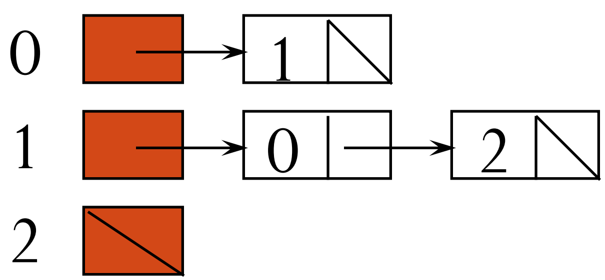


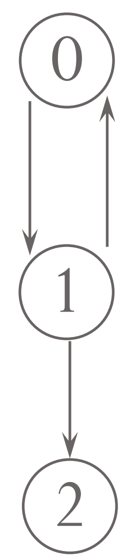
adjacency lists





G1





G2

Shortest-Path (Least-Cost) Algorithms

Dijkstra's Algorithm

Find shortest paths from a source s to all other nodes.

N: Set of Nodes

s: source node

M: Set of processed nodes

d_{ij}: Link cost from node i to node j;

- $d_{ii}=0$
- $d_{ij}=\infty$, if edge (i,j) does not exist
- $d_{ij} \geq 0$, if edge (i,j) exists

D_n: Current least cost from node s to node n that is known to the algorithm

Dijkstra's Shortest Path Algorithm

1. [Initialize]

   $M=\{s\}$ [set of processed nodes is source node only]

   $D_n=d_{sn}$ for all $n \neq s$ [initial path costs to the nodes are simply link costs

   $P_k=s \rightarrow k$ for neighobor nodes **k** or **s** [shortest path to neighbor nodes]

2. Find node $w \notin M$ such that $D_w=\min\{D_j\}$ for $j \notin M$

3. Update D_k for all **k** adjacent to **w**

   If $(D_w+d_{wk} < D_k)$ then do

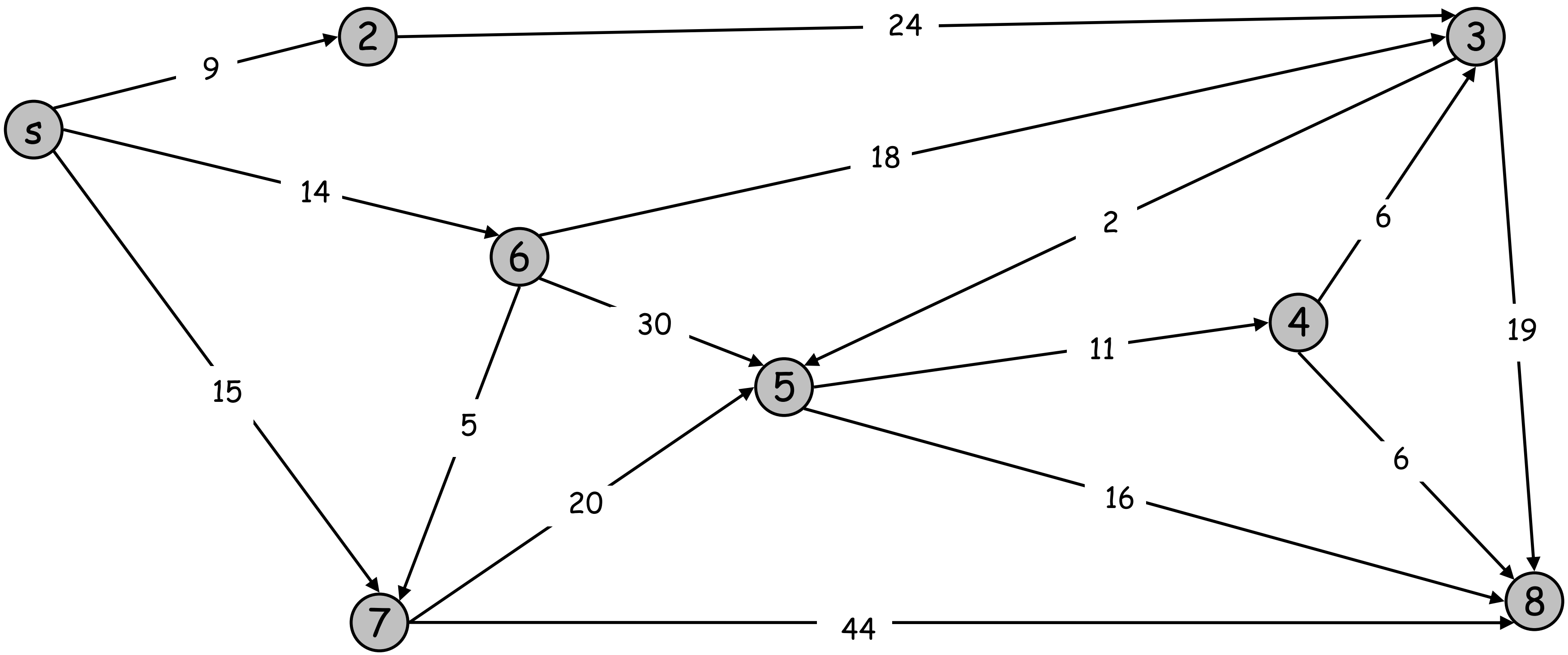
   $D_k = D_w+d_{wk}$

   $P_k= P_w \rightarrow k$

   end if

Dijkstra's Shortest Path Algorithm

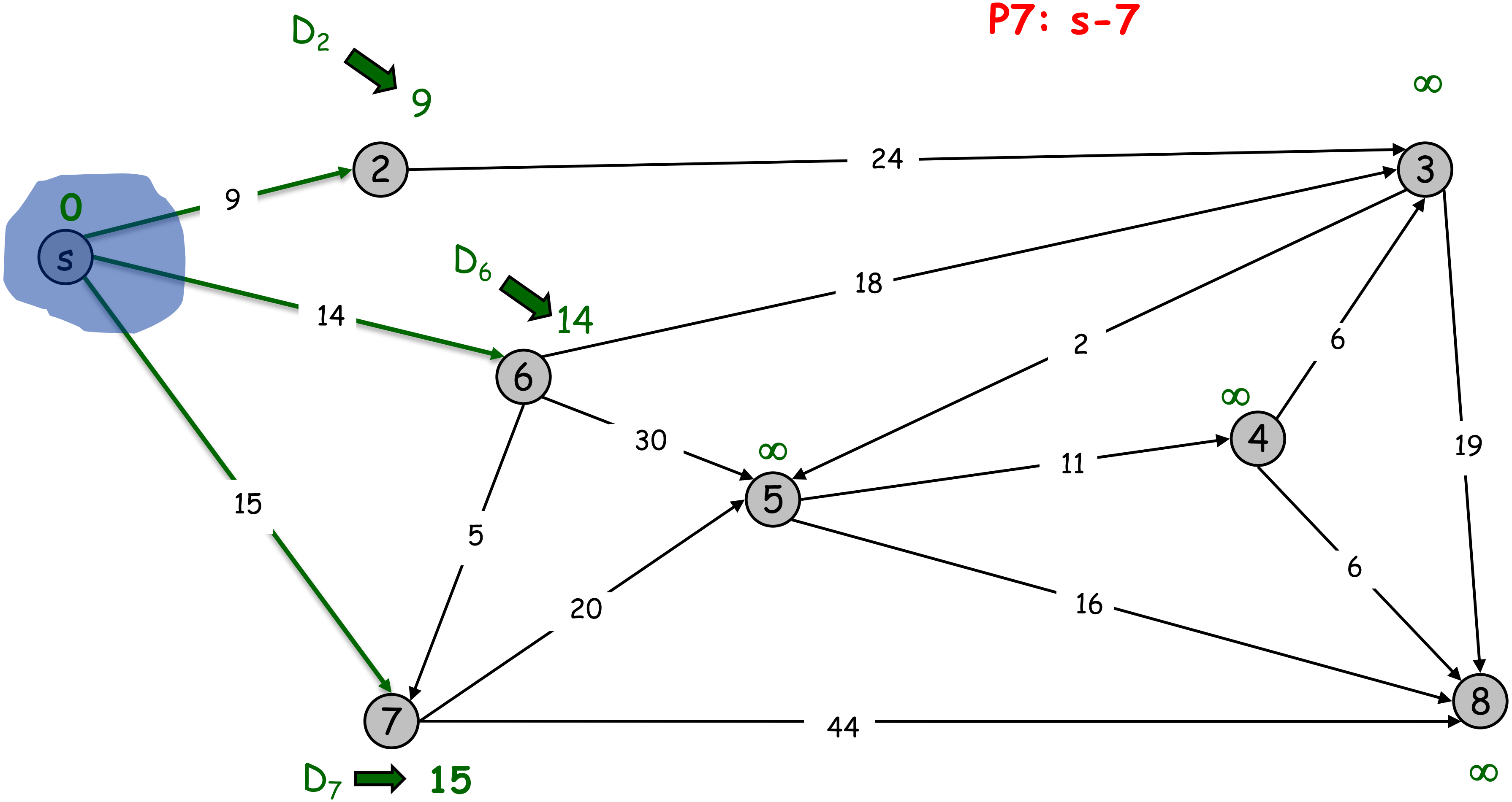
Find shortest path from s to all other nodes.



Dijkstra's Shortest Path Algorithm

M = { s }
NT = { 2, 3, 4, 5, 6, 7, 8 }

Shortest Paths
P2: s-2
P3:
P4:
P5:
P6: s-6
P7: s-7



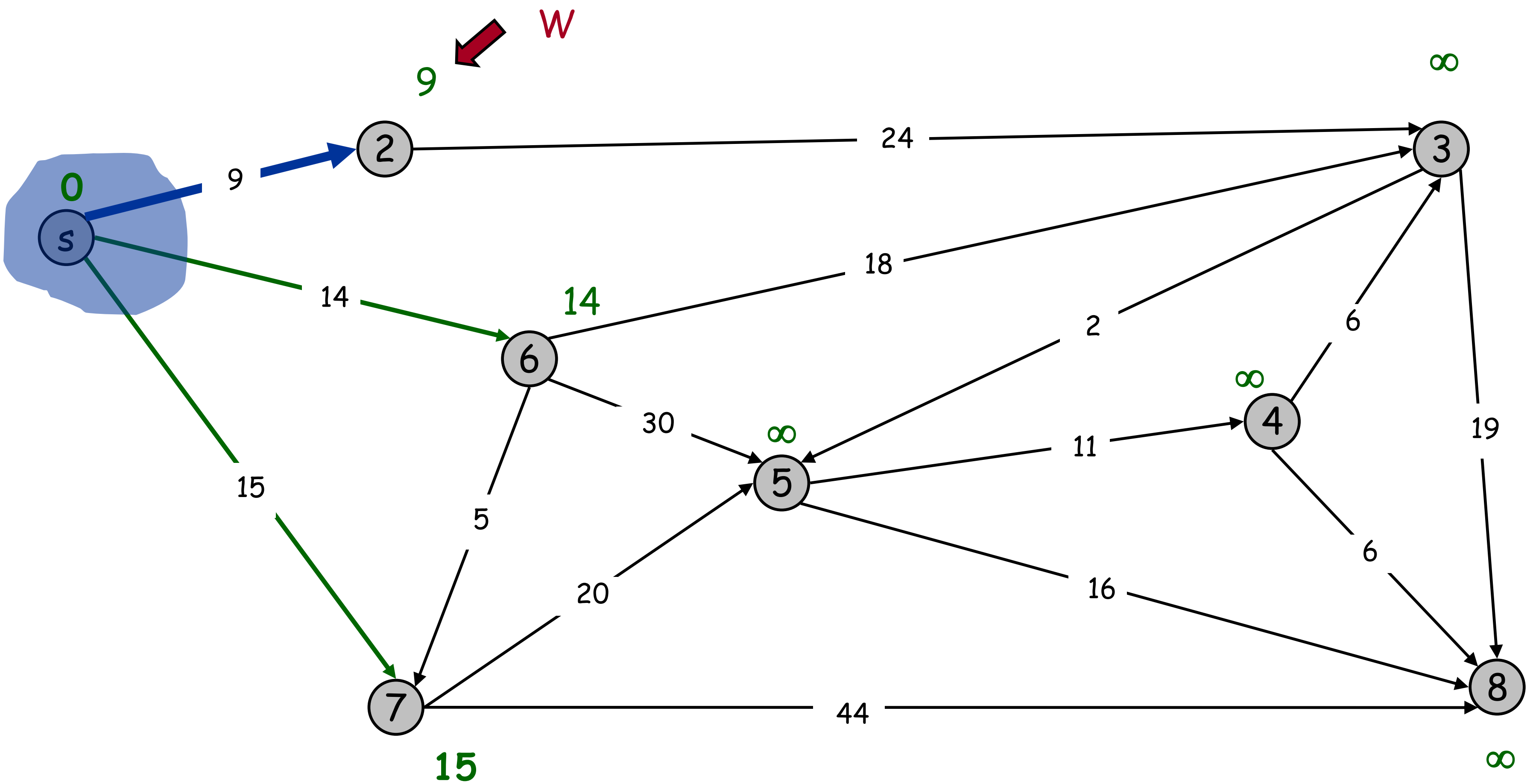
Dijkstra's Shortest Path Algorithm

M = { s }

NT = { 2, 3, 4, 5, 6, 7, 8 }

Shortest Paths

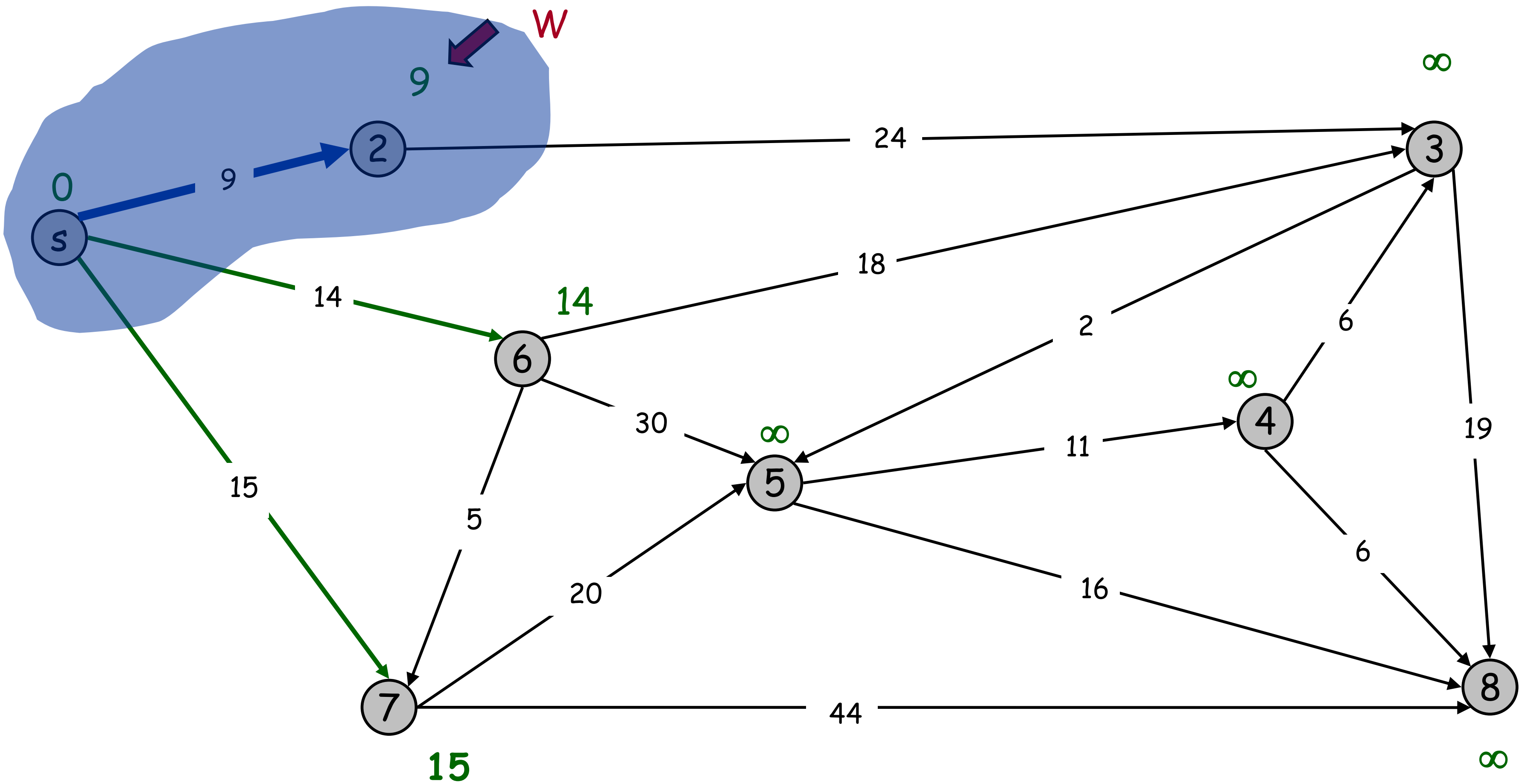
P2: s-2
P3:
P4:
P5:
P6: s-6
P7: s-7



Dijkstra's Shortest Path Algorithm

M = { s, 2 }
NT = { 3, 4, 5, 6, 7, 8 }

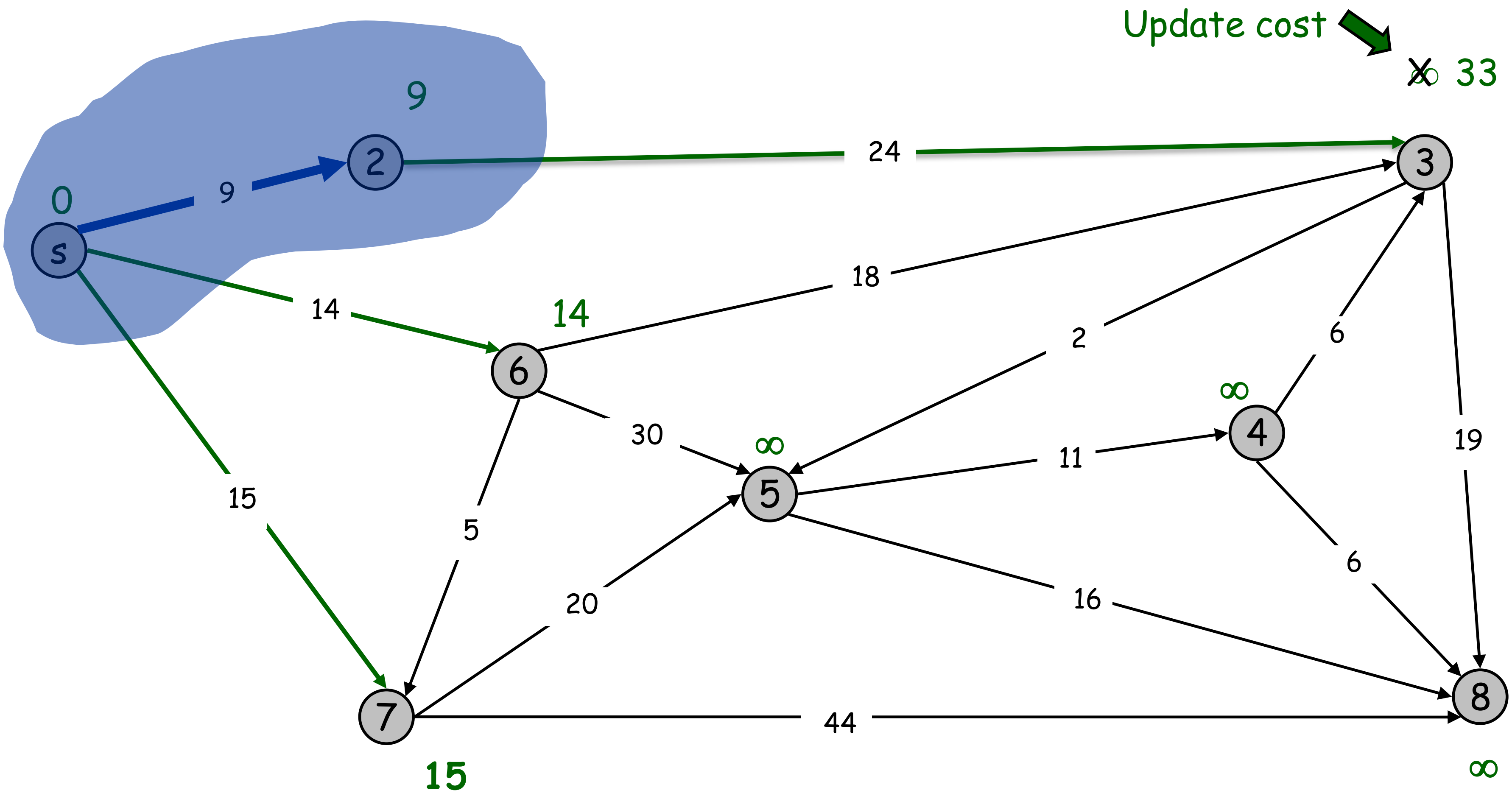
Shortest Paths
P2: s-2
P3:
P4:
P5:
P6: s-6
P7: s-7



Dijkstra's Shortest Path Algorithm

Shortest Paths
P2: s-2
P3: P2-3 ~ s-2-3
P4:
P5:
P6: s-6
P7: s-7

M = { s, 2 }
NT = { 3, 4, 5, 6, 7, 8 }



Dijkstra's Shortest Path Algorithm

M = { s, 2 }

NT = { 3, 4, 5, 6, 7, 8 }

Shortest Paths

P2: s-2

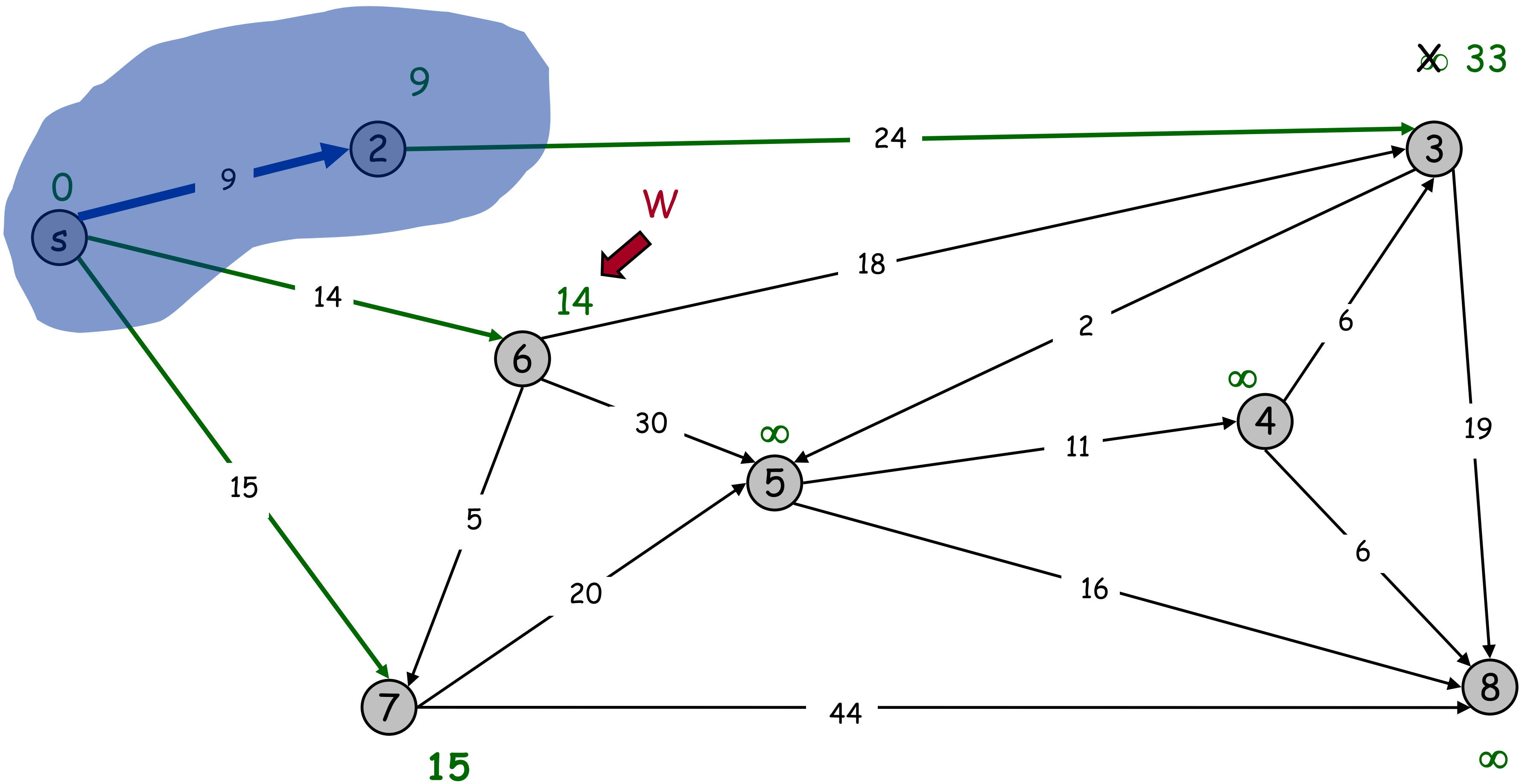
P3: P2-3 ~ s-2-3

P4:

P5:

P6: s-6

P7: s-7

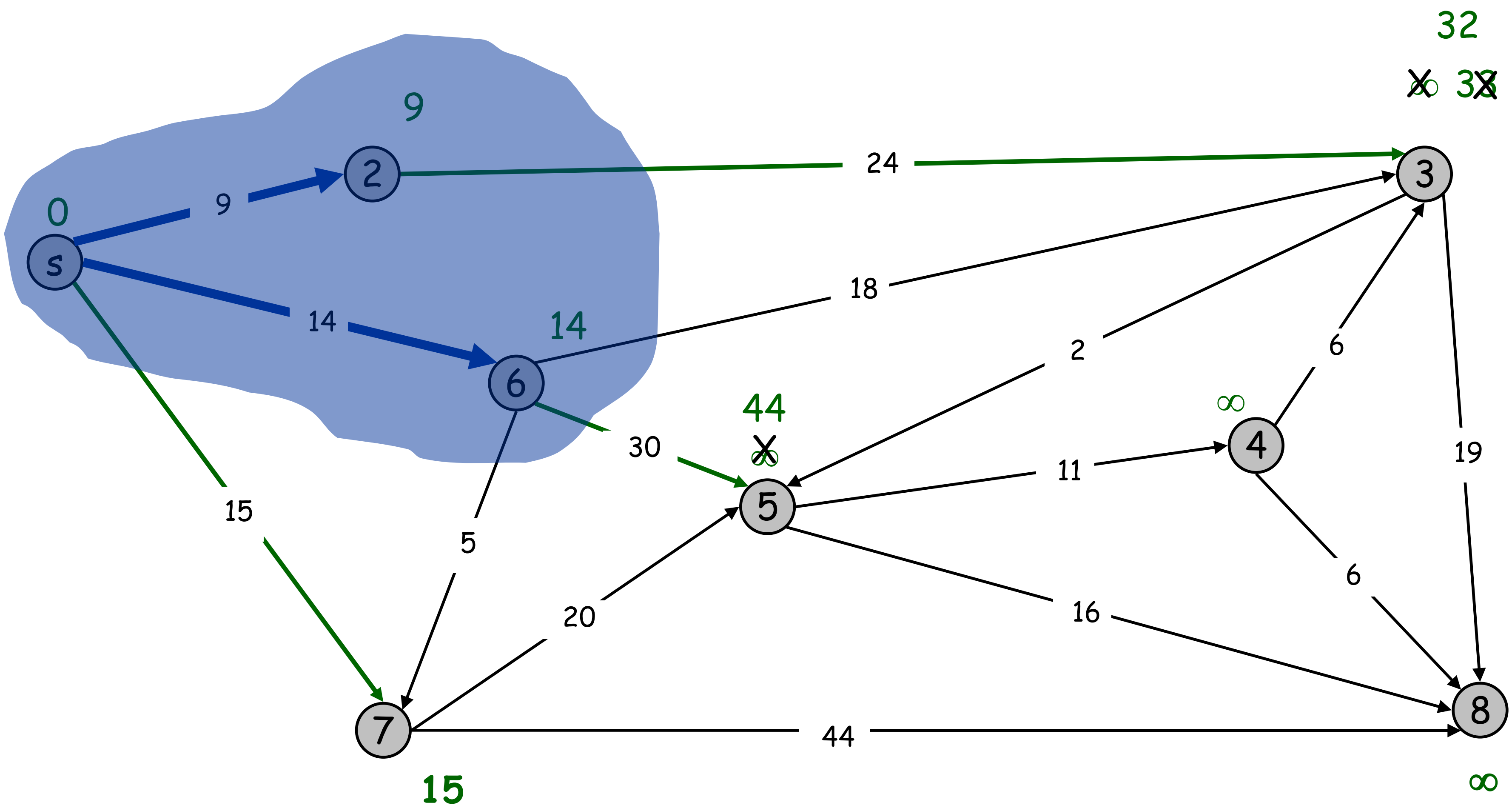


Dijkstra's Shortest Path Algorithm

M = { s, 2, 6 }
NT = { 3, 4, 5, 7, 6 }

Shortest Paths

P2: s-2
P3: P6-3 ~ s-6-3
P4:
P5: P6-5 ~ s-6-5
P6: s-6
P7: s-7

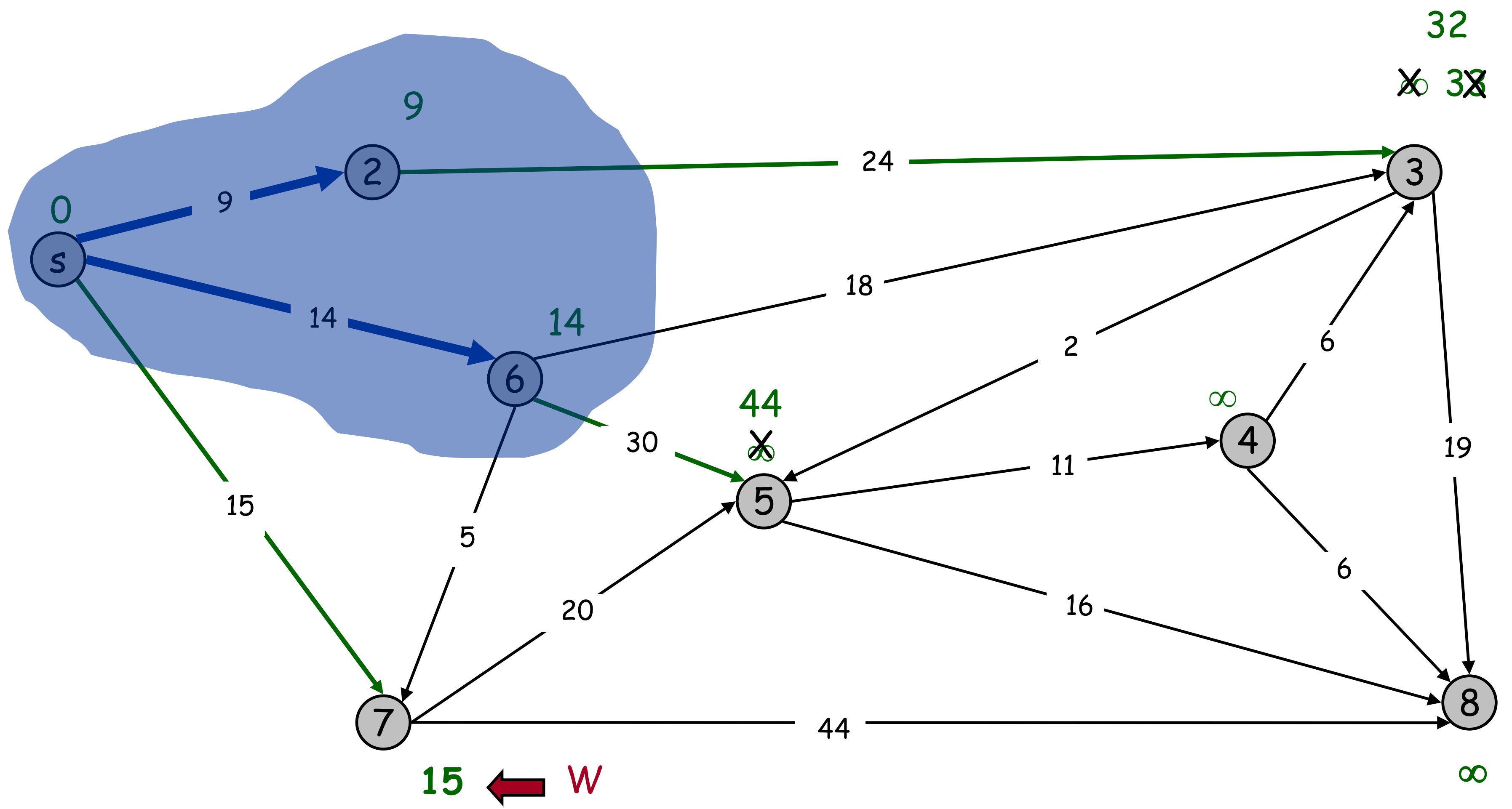


Dijkstra's Shortest Path Algorithm

M = { s, 2, 6 }
NT = { 3, 4, 5, 7, 8 }

Shortest Paths

P2: s-2
P3: P6-3 ~ s-6-3
P4:
P5: P6-5 ~ s-6-5
P6: s-6
P7: s-7

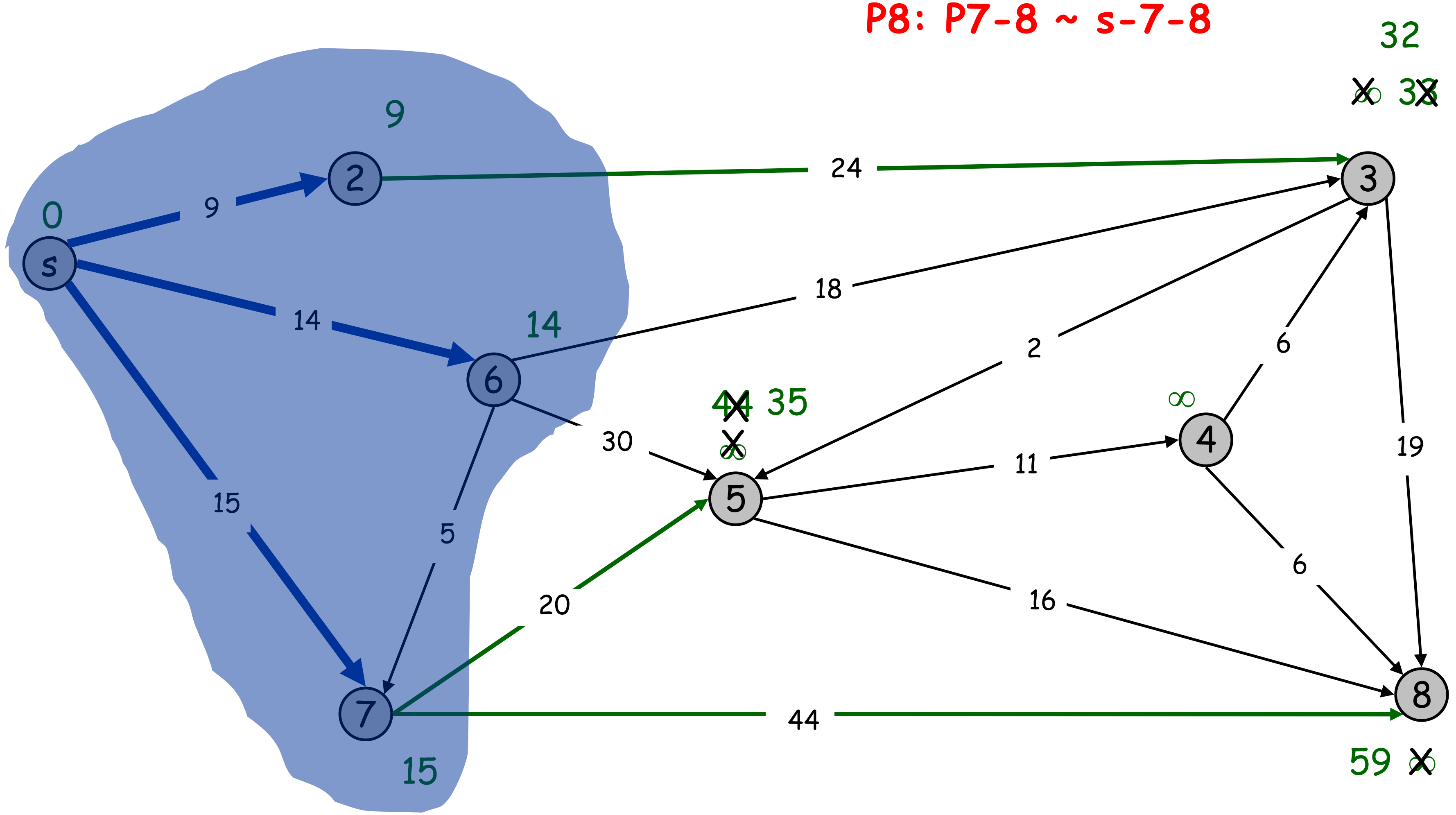


Dijkstra's Shortest Path Algorithm

Shortest Paths

P2: s-2
P3: P6-3 ~ s-6-3
P4:
P5: P7-5 ~ s-7-5
P6: s-6
P7: s-7
P8: P7-8 ~ s-7-8

M = { s, 2, 6, 7 }
NT = { 3, 4, 5, 8 }

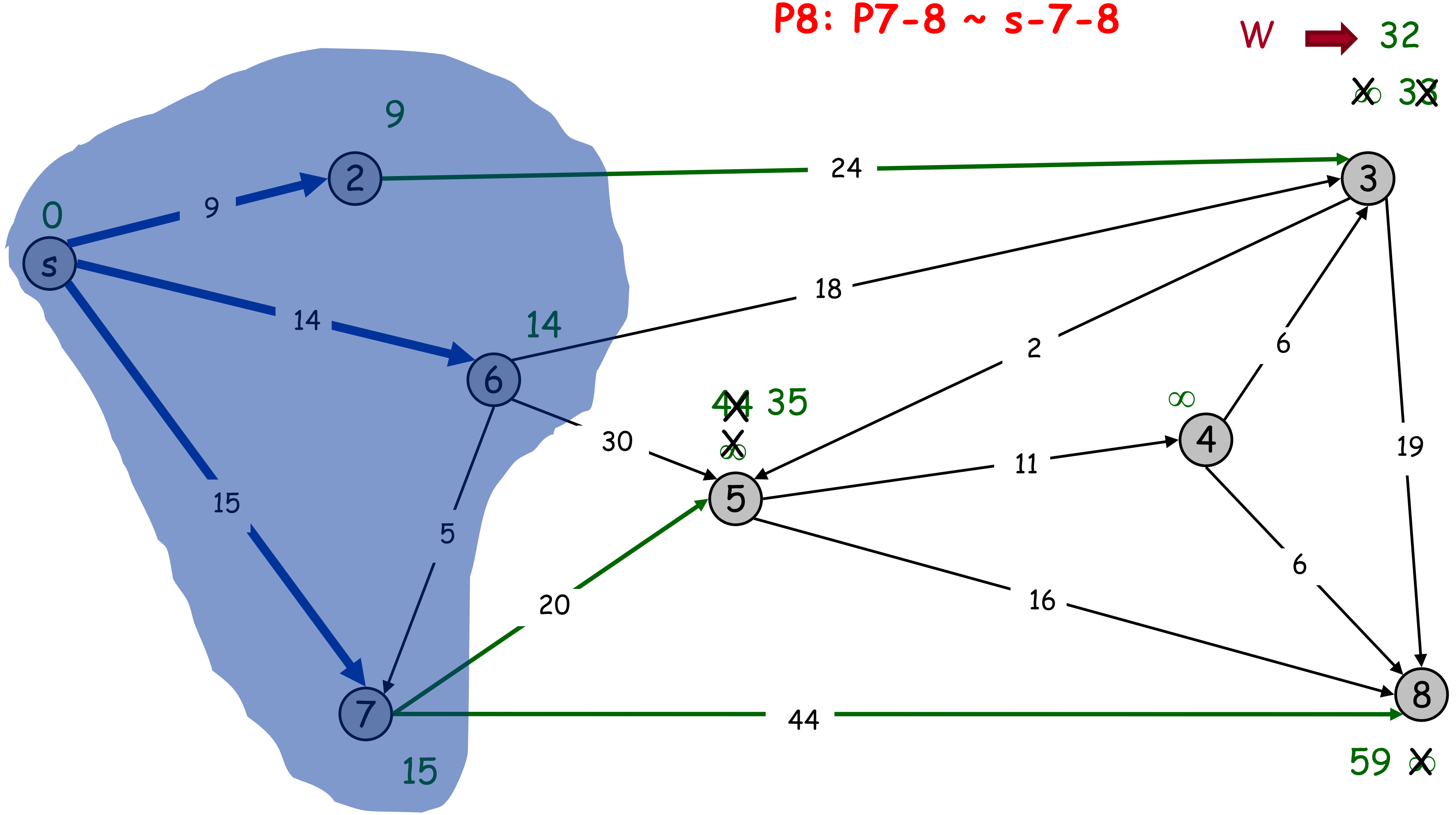


Dijkstra's Shortest Path Algorithm

M = { s, 2, 6, 7 }
NT = { 3, 4, 5, 8 }

Shortest Paths

P2: s-2
P3: P6-3 ~ s-6-3
P4:
P5: P7-5 ~ s-7-5
P6: s-6
P7: s-7
P8: P7-8 ~ s-7-8

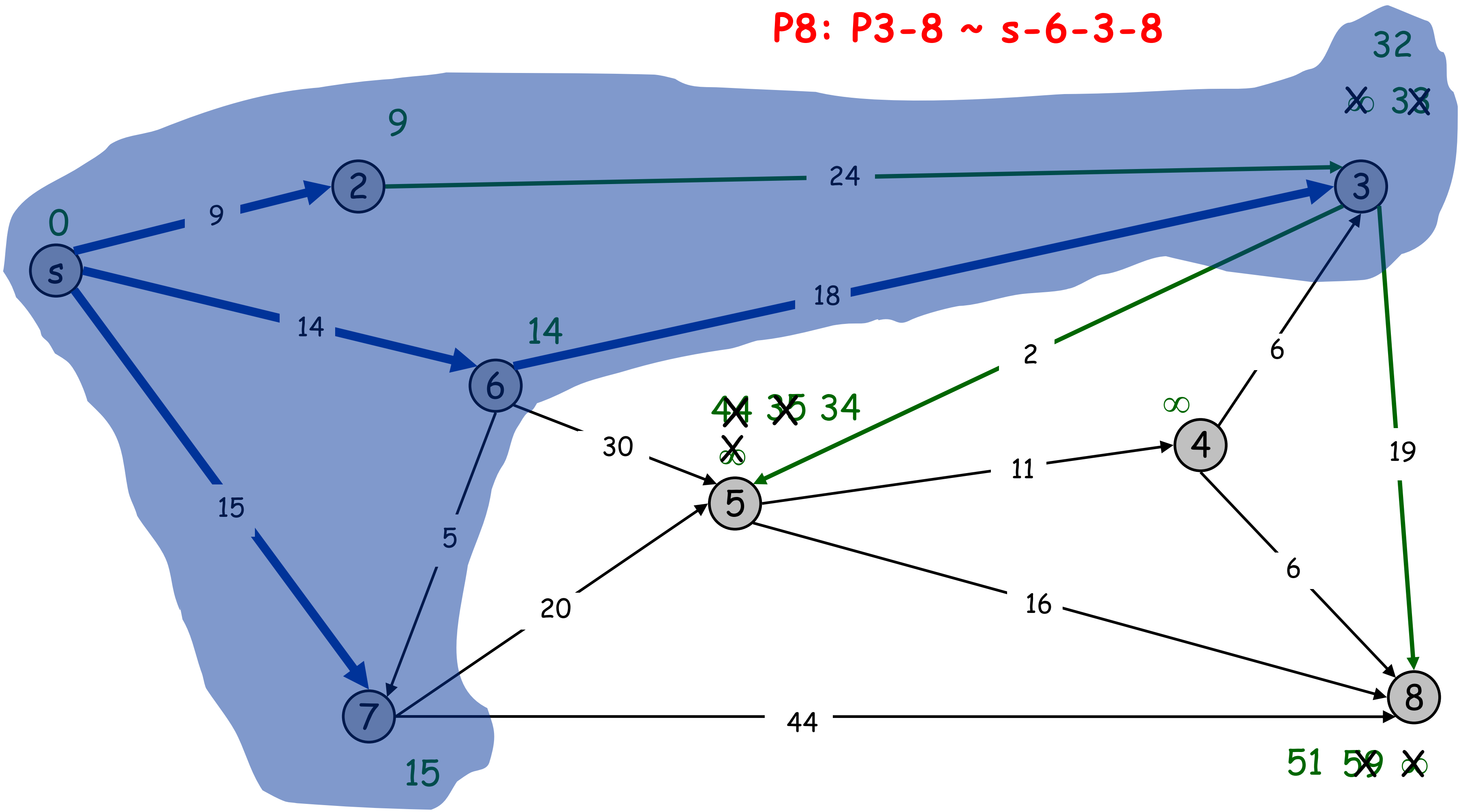


Dijkstra's Shortest Path Algorithm

Shortest Paths

P2: s-2
P3: P6-3 ~ s-6-3
P4:
P5: P3-5 ~ s-6-3-5
P6: s-6
P7: s-7
P8: P3-8 ~ s-6-3-8

M = { s, 2, 3, 6, 7 }
NT = { 4, 5, 8 }

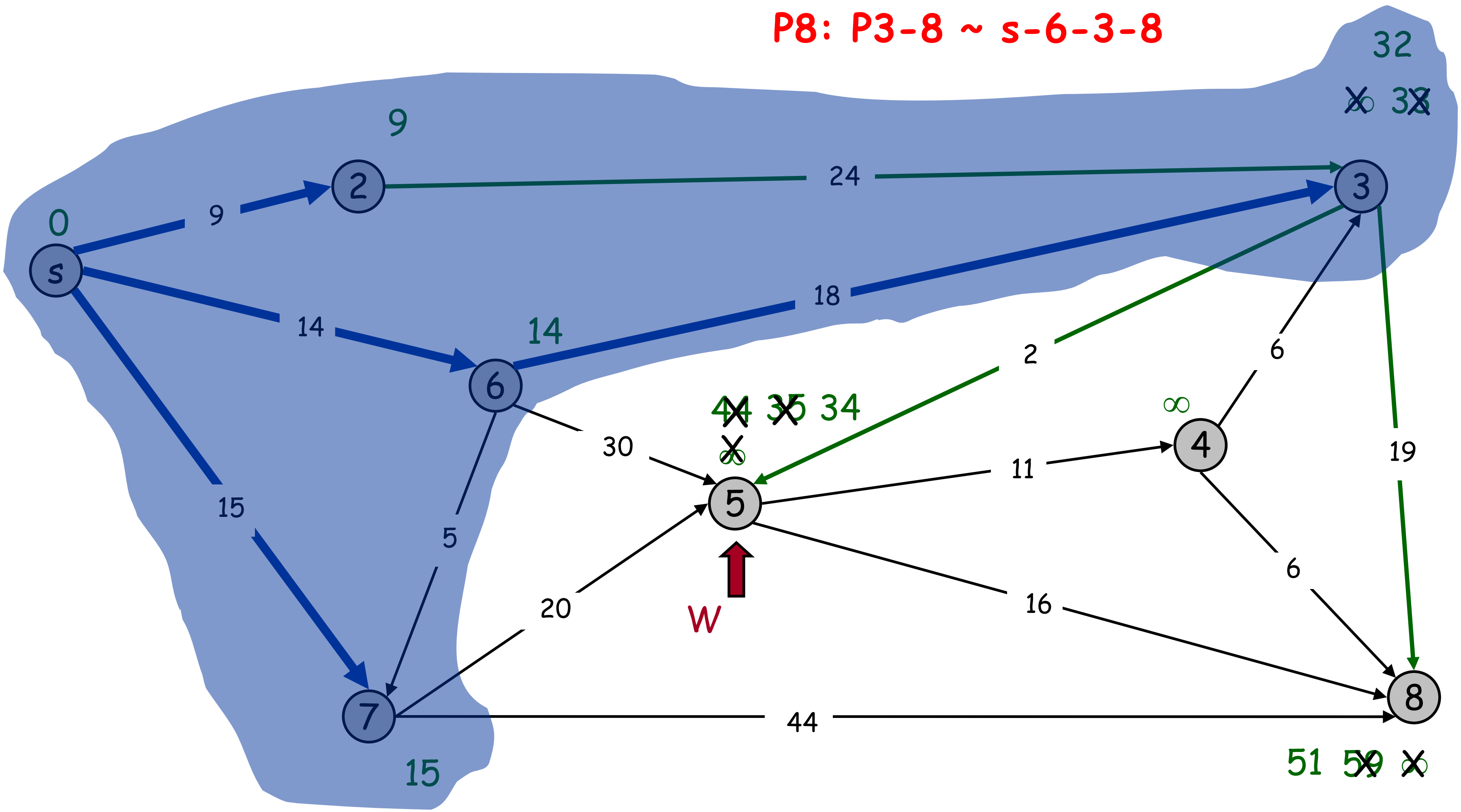


Dijkstra's Shortest Path Algorithm

Shortest Paths

P2: s-2
P3: s-6-3
P4:
P5: P3-5 ~ s-6-3-5
P6: s-6
P7: s-7
P8: P3-8 ~ s-6-3-8

M = { s, 2, 3, 6, 7 }
NT = { 4, 5, 8 }

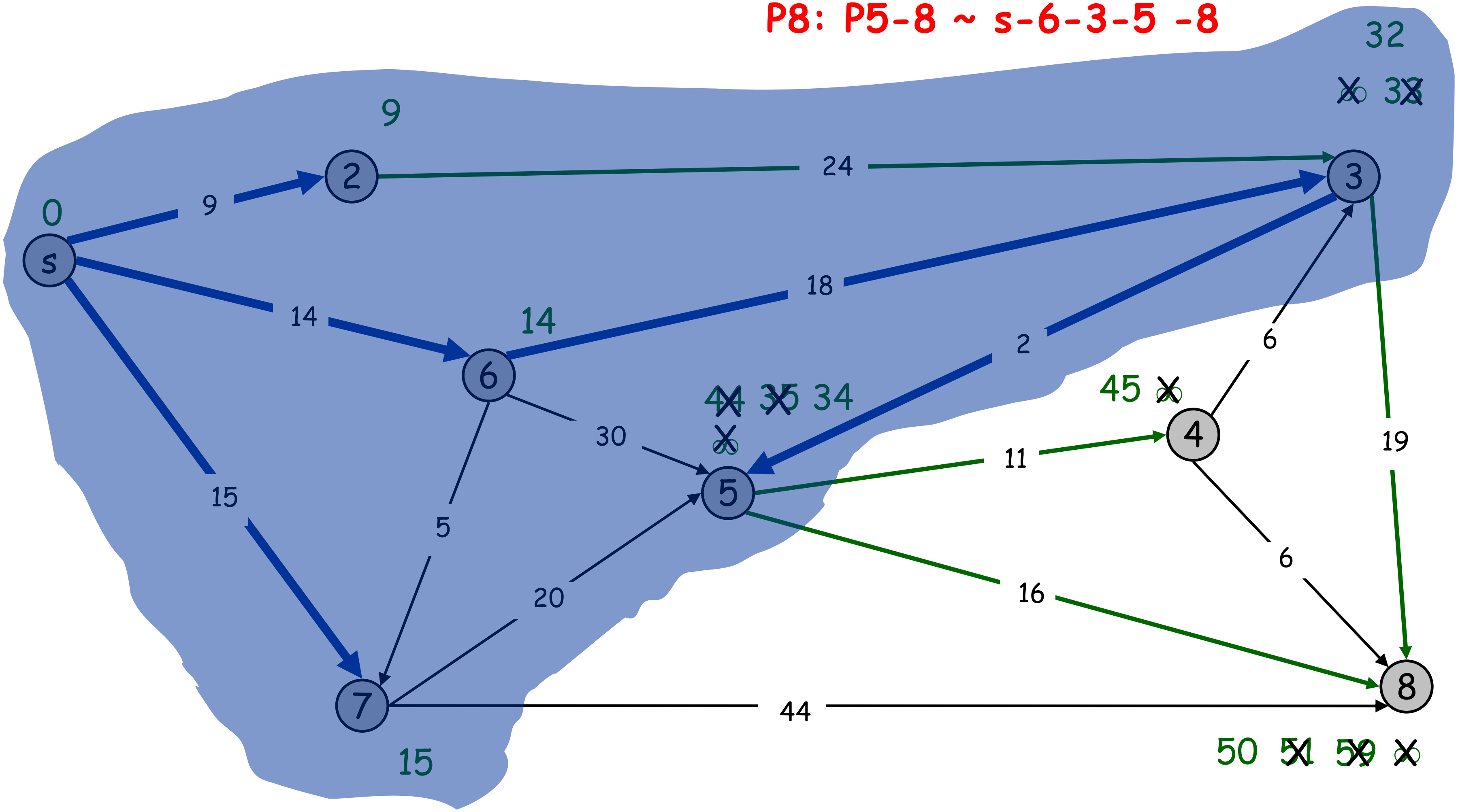


Dijkstra's Shortest Path Algorithm

<u>**Shortest Paths**</u>

P2: s-2
P3: s-6-3
P4: P5-4 ~ s-6-3-5-4
P5: s-6-3-5
P6: s-6
P7: s-7
P8: P5-8 ~ s-6-3-5 -8

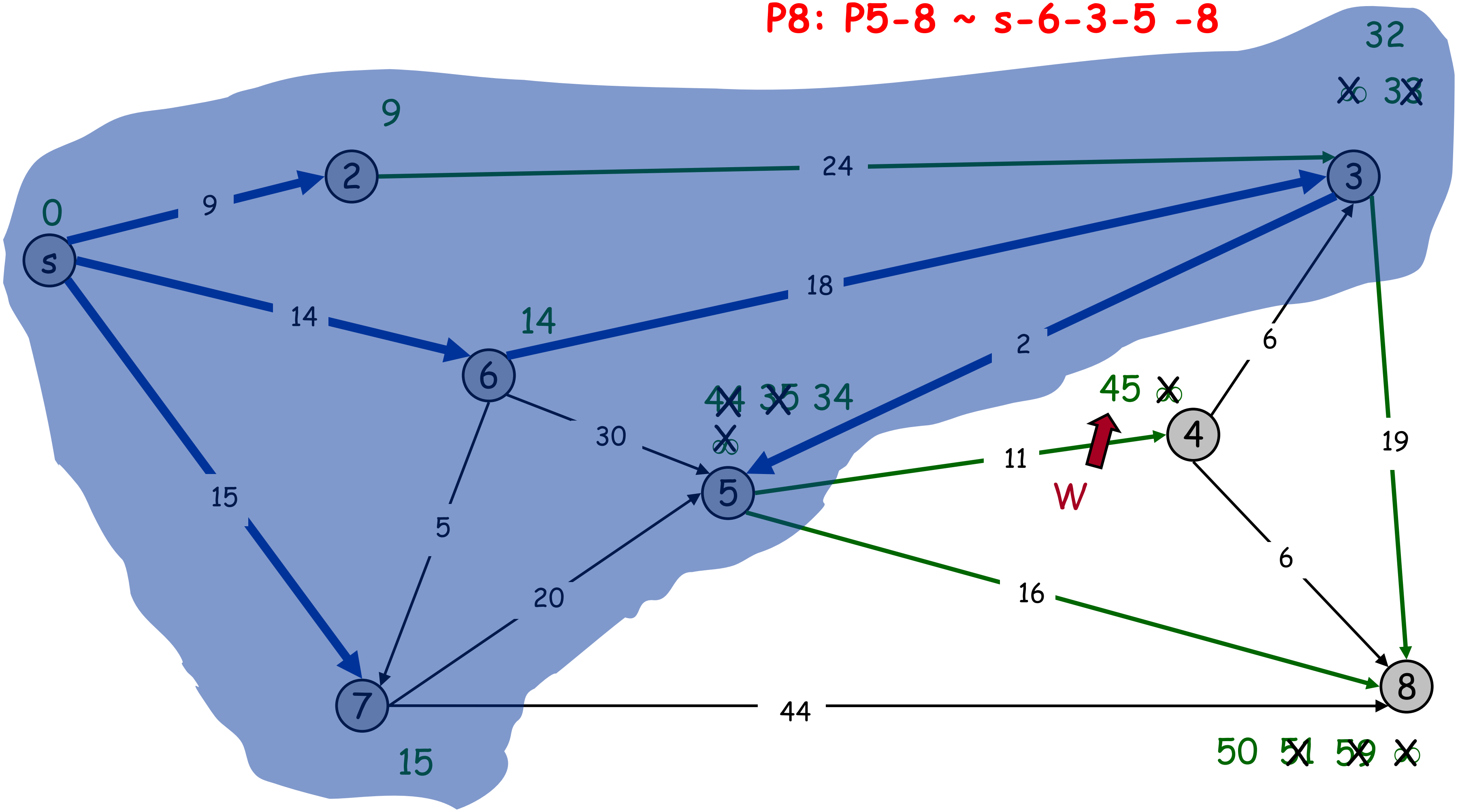
M = { s, 2, 3, 5, 6, 7 }
NT = { 4, 8 }



Dijkstra's Shortest Path Algorithm

M = { s, 2, 3, 5, 6, 7 }
NT = { 4, 8 }

Shortest Paths
P2: s-2
P3: s-6-3
P4: P5-4 ~ s-6-3-5-4
P5: s-6-3-5
P6: s-6
P7: s-7
P8: P5-8 ~ s-6-3-5 -8

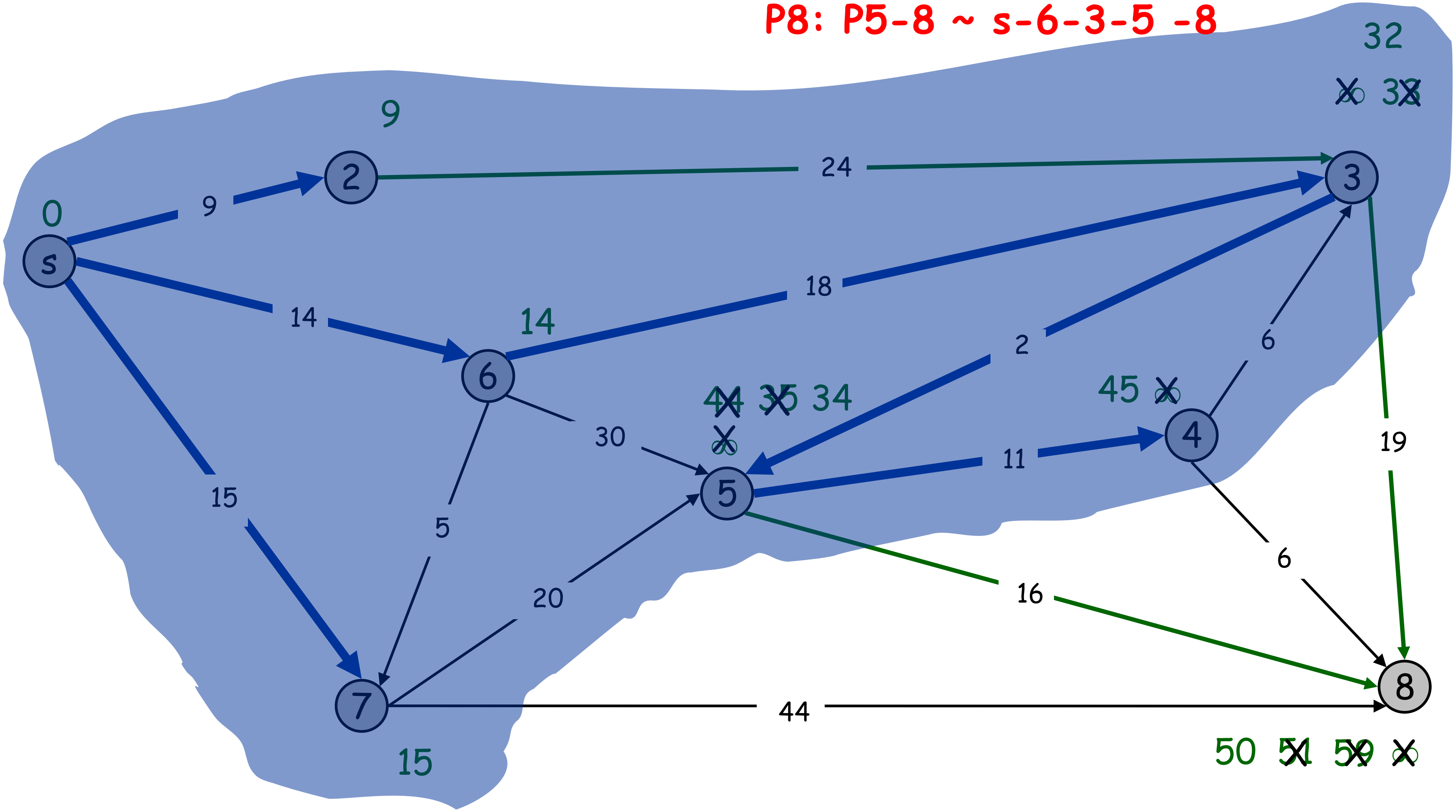


Dijkstra's Shortest Path Algorithm

M = { s, 2, 3, 4, 5, 6, 7 }
NT = { 8 }

Shortest Paths

P2: s-2
P3: s-6-3
P4: s-6-3-5-4
P5: s-6-3-5
P6: s-6
P7: s-7
P8: P5-8 ~ s-6-3-5 -8

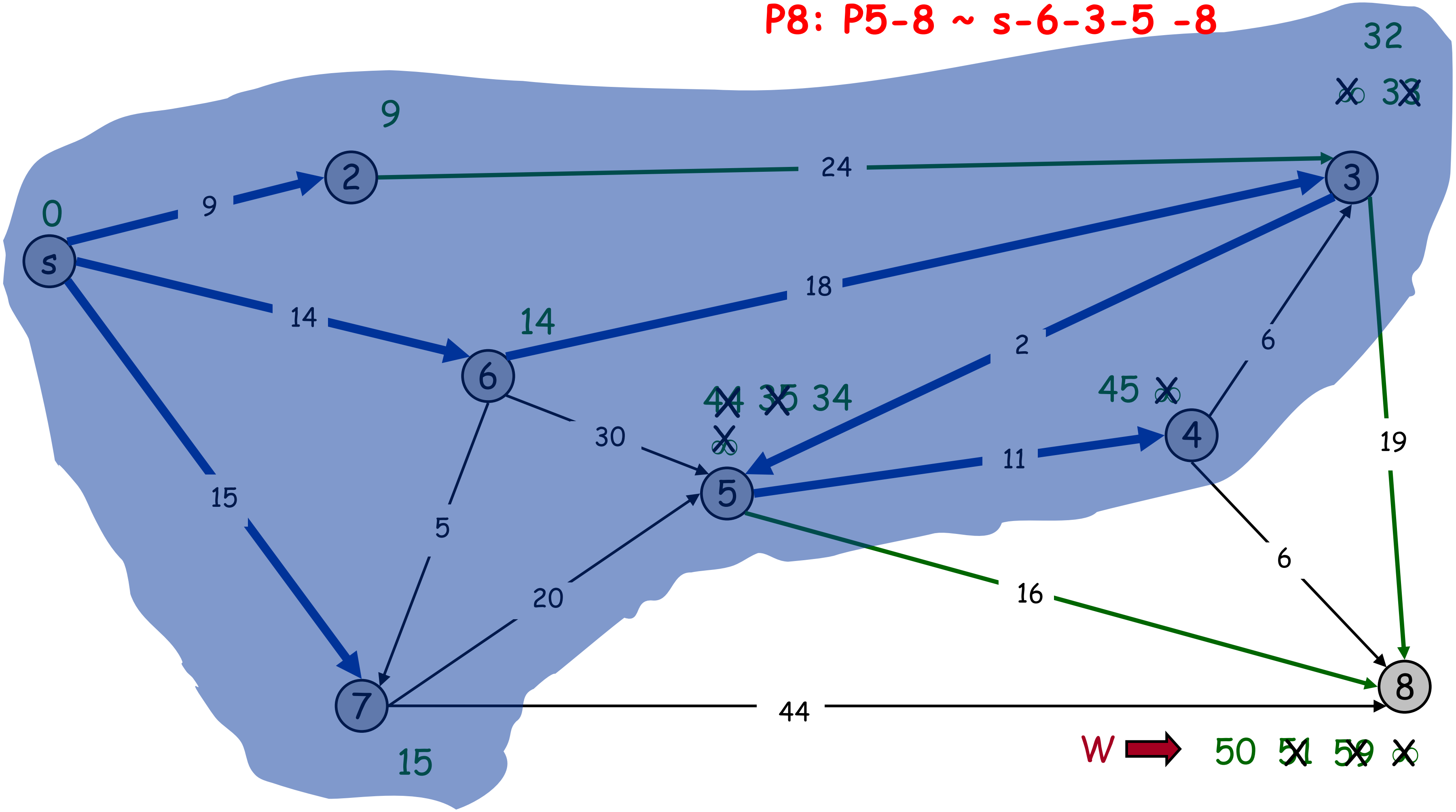


Dijkstra's Shortest Path Algorithm

M = { s, 2, 3, 4, 5, 6, 7 }
NT = { 8 }

Shortest Paths

P2: s-2
P3: s-6-3
P4: s-6-3-5-4
P5: s-6-3-5
P6: s-6
P7: s-7
P8: P5-8 ~ s-6-3-5 -8



Dijkstra's Shortest Path Algorithm

Shortest Paths

P2: s-2
P3: s-6-3
P4: s-6-3-5-4
P5: s-6-3-5
P6: s-6
P7: s-7
P8: s-6-3-5 -8

M = { s, 2, 3, 4, 5, 6, 7, 8 }
NT = { }