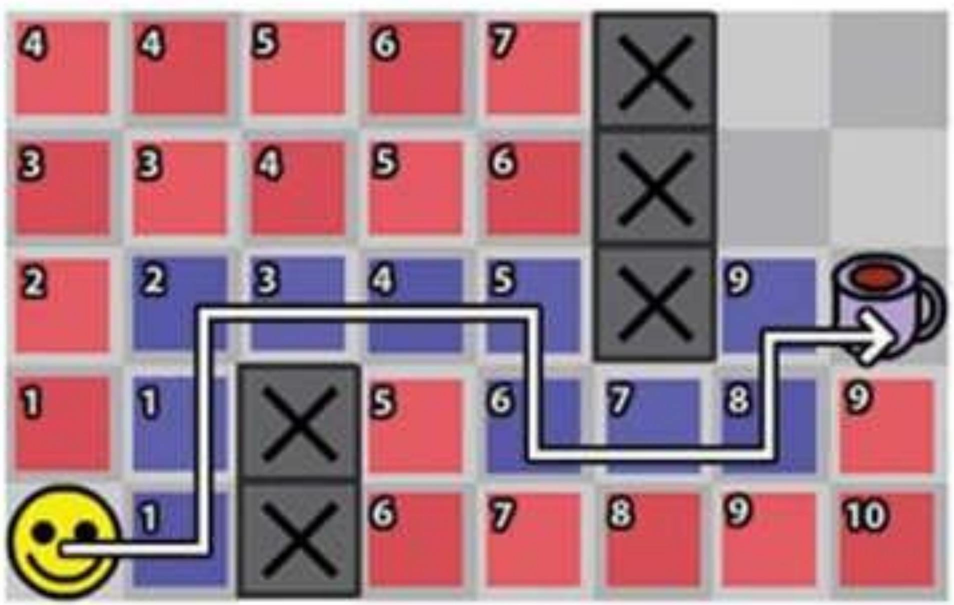
DATA STRUCTURES AND ALGORITHMS
Spring 2025

Graph Part II

Lecturer: Do Thuy Duong

Pathfinding in a graph

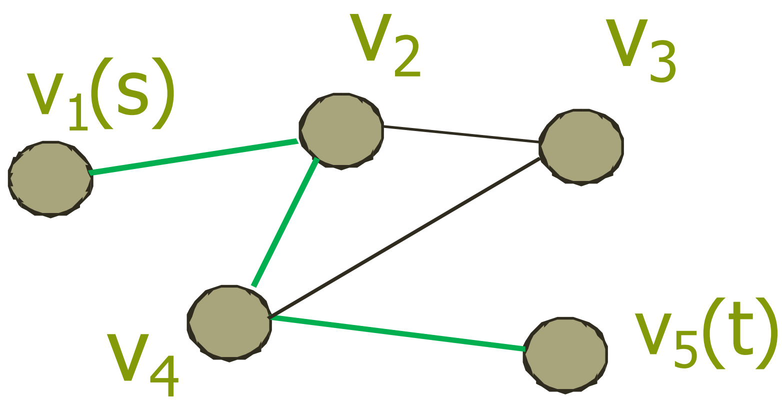


Content

- Pathfinding in a graph
- Shortest path problem
  - Dijkstra algorithm
  - Bellman-Ford algorithm
  - Floyd-Warshall algorithm

Pathfinding in a graph

- **Pathfinding problem**
  - In a graph G=(V,E), find a path from vertex s to vertex t.
    - Find a path
    - Find all possible paths
  - Graph traversal algorithms can be used
    - **BFS**
    - **DFS**



Pathfinding using BFS [1]

Code

Algorithm BFS(V,E,s,t): Boolean

Input: A graph with V & E are the set of vertices and edges, s is the source vertex, t is the destination vertex.

Output: A path from s to t if exist, otherwise return false.

```
For each v in V do
    Color[v]←Black
    Path[v] ← null

Color[s] ← Yellow
Q ← new empty queue
Q.Enqueue(s)
…
```

Pathfinding using BFS [2]

Code

```
while Q is not empty do
   u ← Q.Dequeue()

   If u equals to t then
       showPath(s, t)
       return  true

   For each v adjacent to u do
      If Color[v] is Black then
          Color[v] ← Yellow
          Q.Enqueue(v)
          path[v] ←u
endwhile
return false
```

Pathfinding using BFS [3]

Code

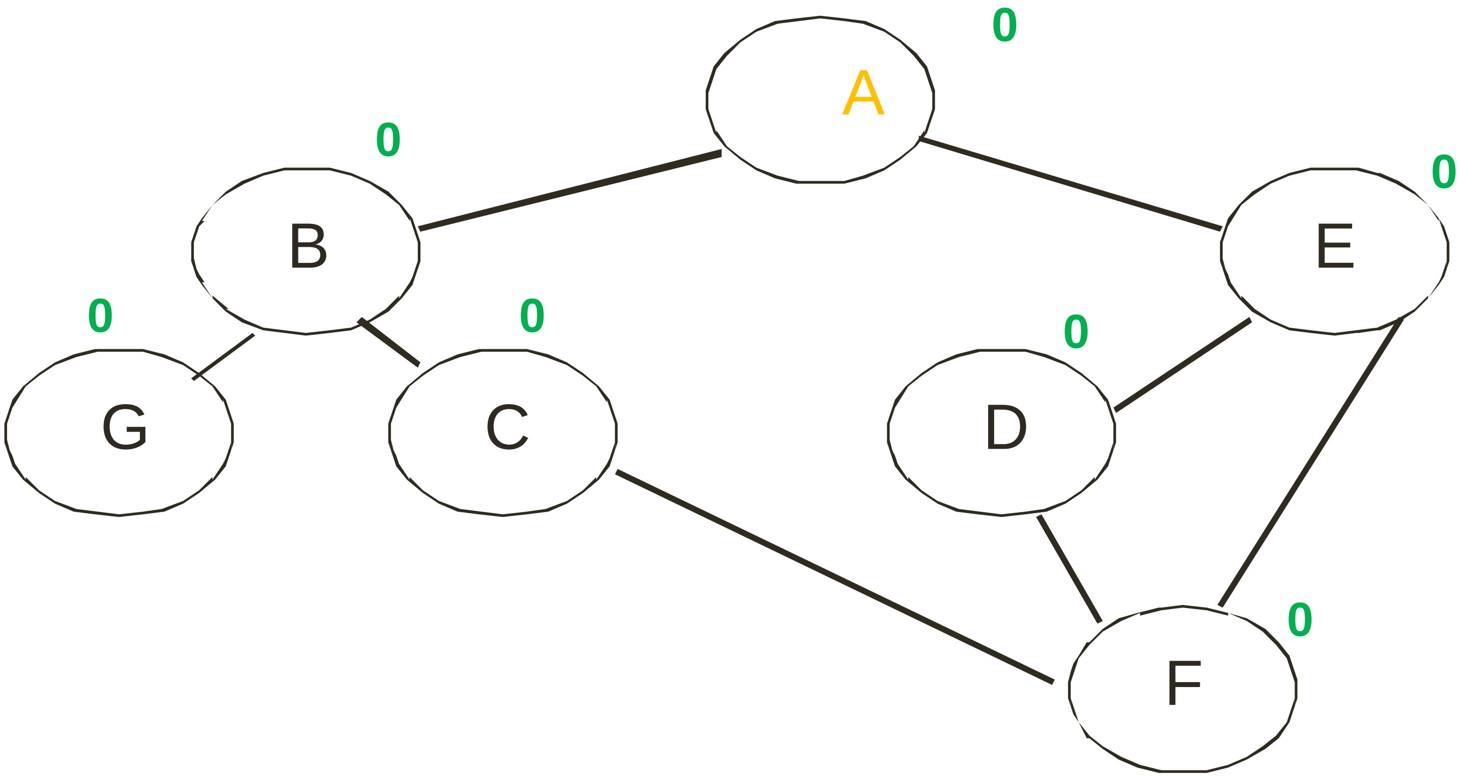
```
Algorithm  showPath(s, t)
    Input: Source vertex s, Destination
    vertex  t.
    Output: print the path from s to t if  exist.

    u ← t
    while u !=s do
        Print u
        u ← path[u]
```

Pathfinding using BFS [4]

Find path from A to D

Start at A. Mark A yellow. Put A to the queue

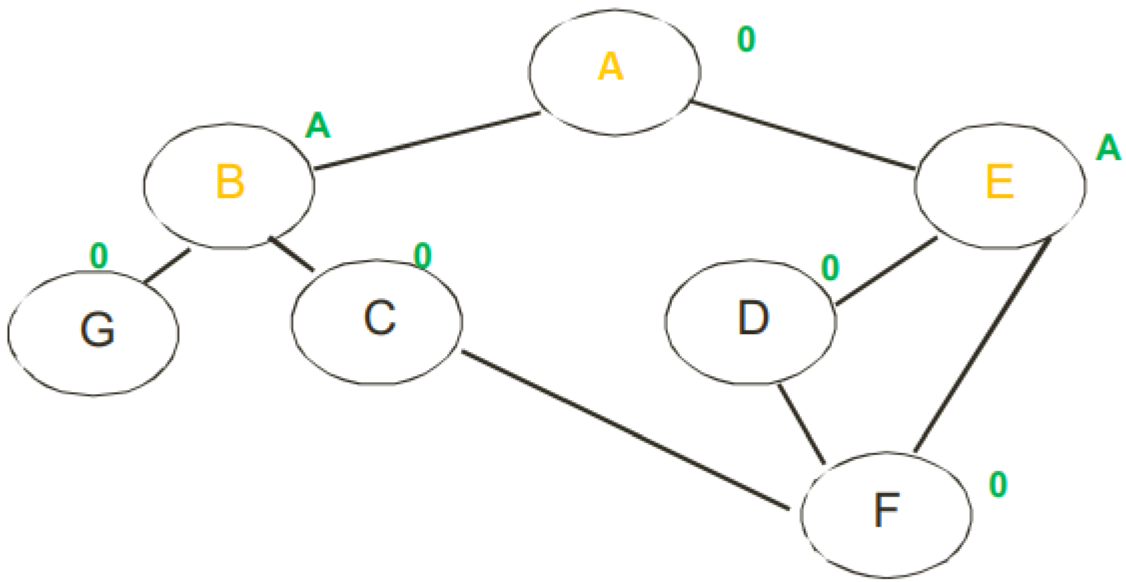
Queue: A



Pathfinding using BFS [5]

- Take A from Queue, A ≠ D
- Found 2 black vertices B and E, that are adjacent to A
- Mark B & E yellow. Put B & E to queue, update path[B] & path[E]

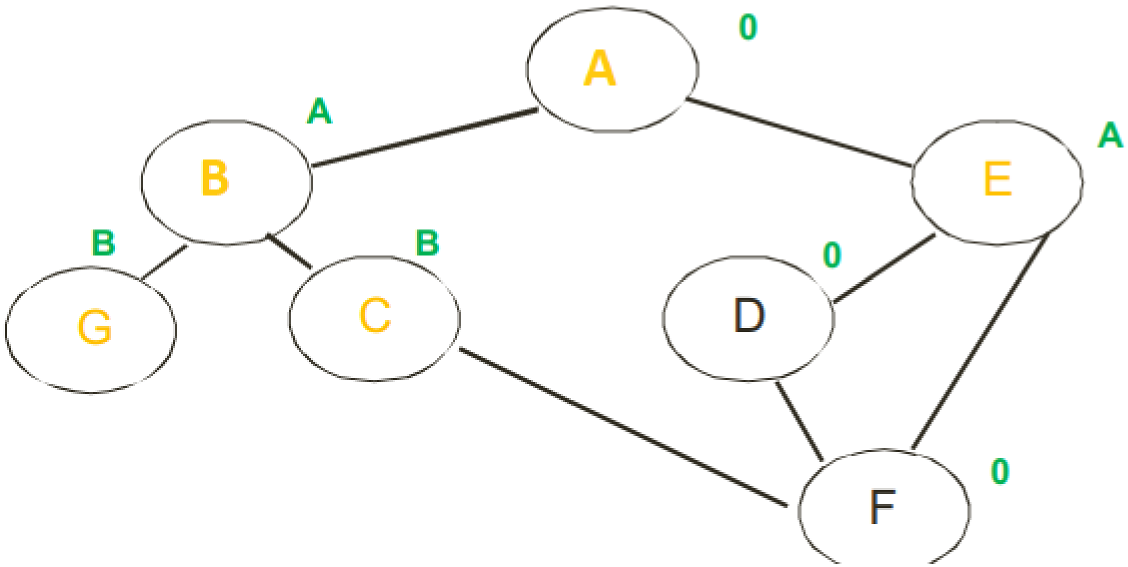
Queue: ~~A~~ B E



Pathfinding using BFS [6]

- Take B from Queue, B ≠ D
- Found 2 black vertices G and C, that are adjacent to B
- Mark G & C yellow. Put G & C to queue, update path[G] & path[C]

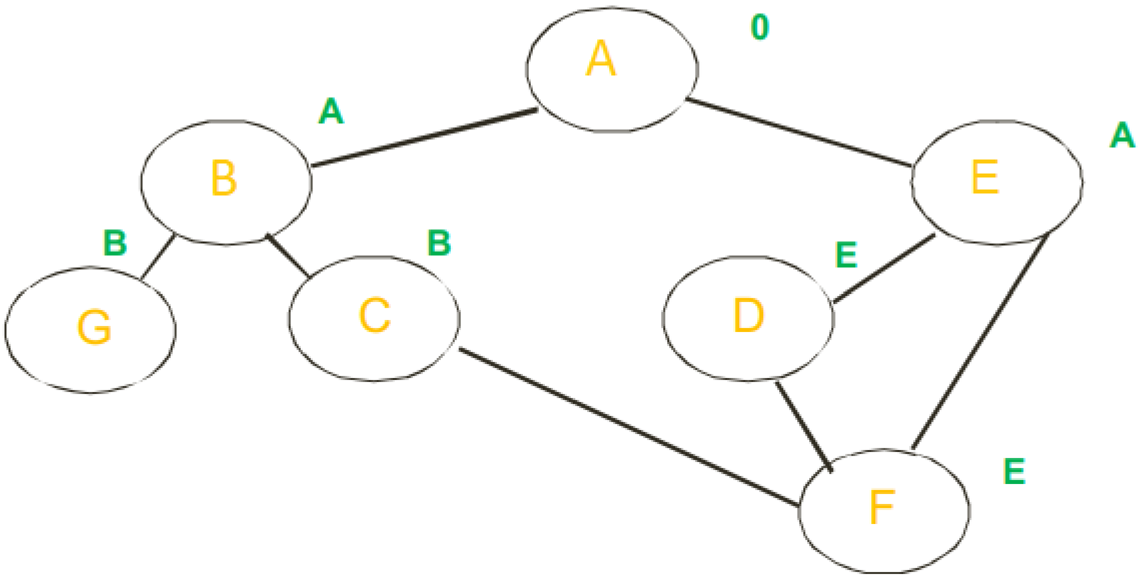
Queue: ~~A B~~ E G C



Pathfinding using BFS [7]

- Take E from Queue, E ≠ D
- Found 2 black vertices D and F, that are adjacent to E
- Mark D & F yellow. Put D & F to queue, update path[D] & path[F]

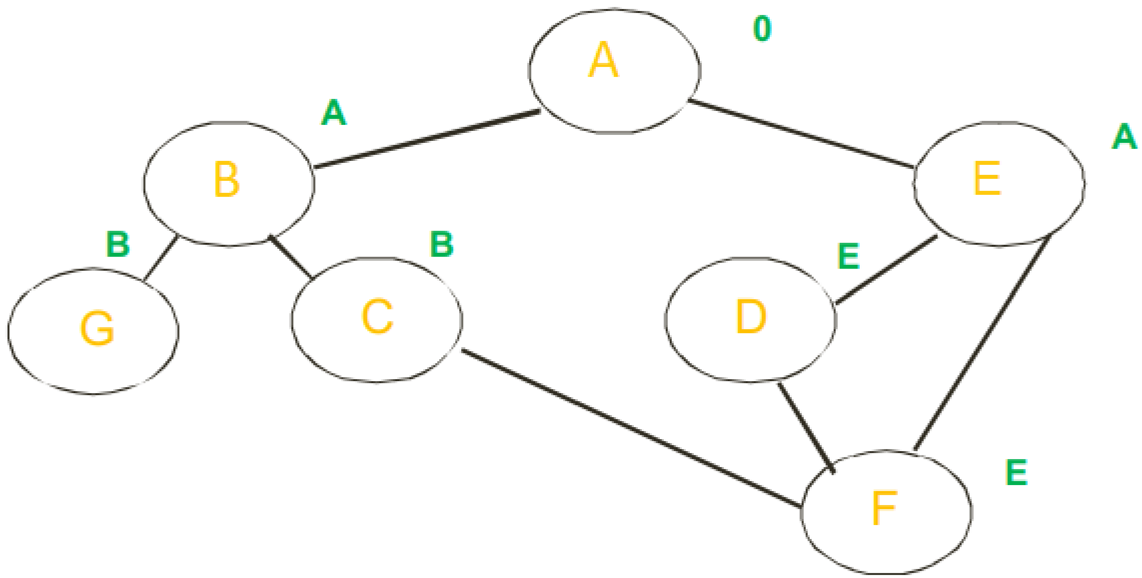
Queue: ~~A B E~~ G C D F



Pathfinding using BFS [8]

- Take G from Queue, G ≠ D
- Nothing put to the queue

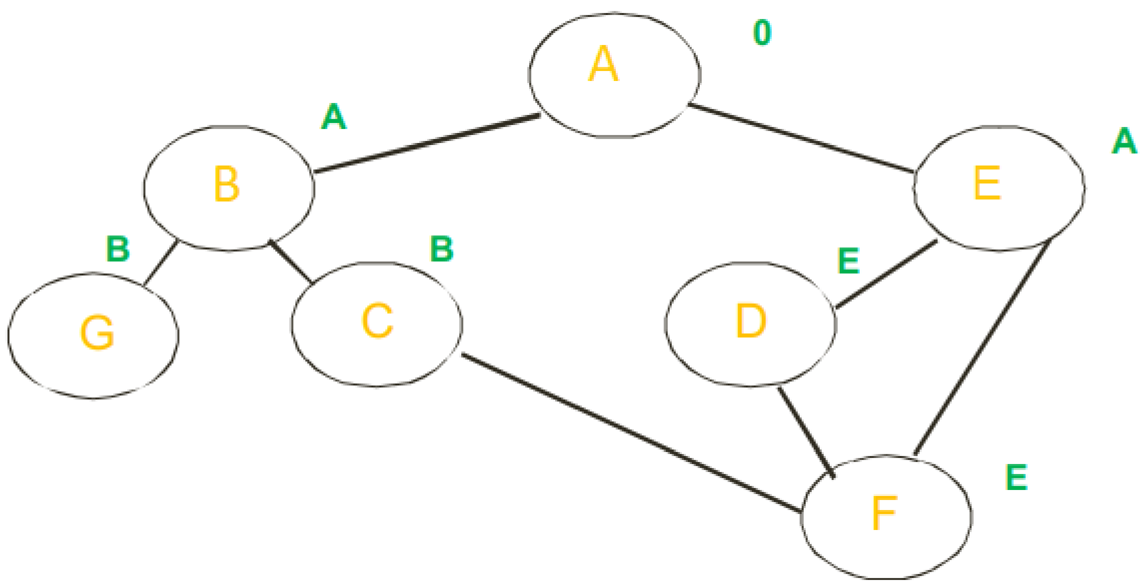
Queue: ~~A B E G~~ C D F



Pathfinding using BFS [9]

- Take C from Queue, C ≠ D
- No black vertices adjacent to C. Nothing put to the queue

Queue: ~~A B E G C~~ D F

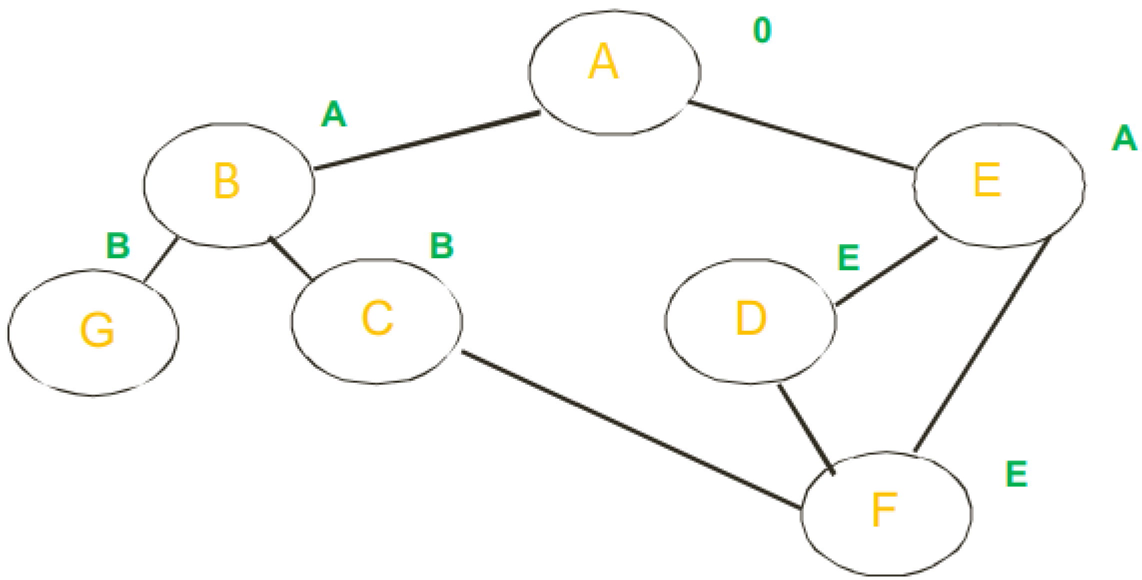


Pathfinding using BFS [10]

Take D from the queue. D is the destination vertex. Show path and return true.

Queue: ~~A B E G C D~~ F

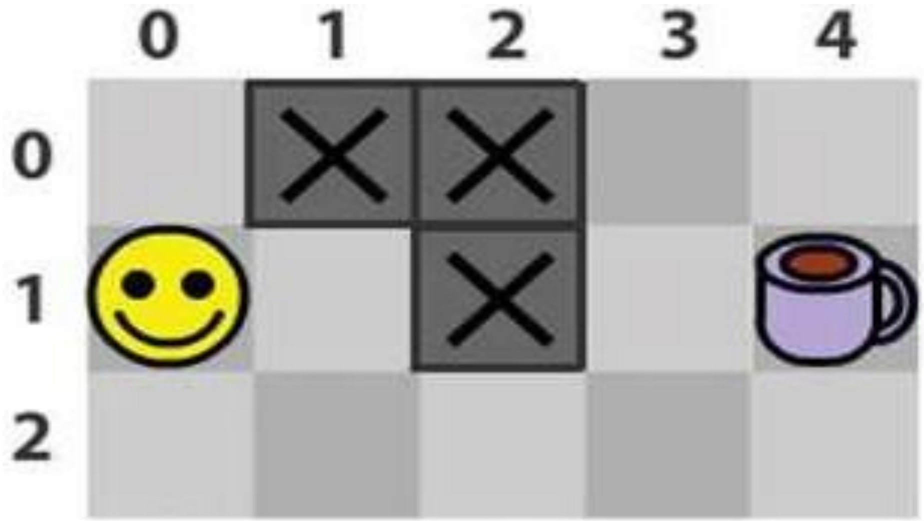
→ **Path: A E D**

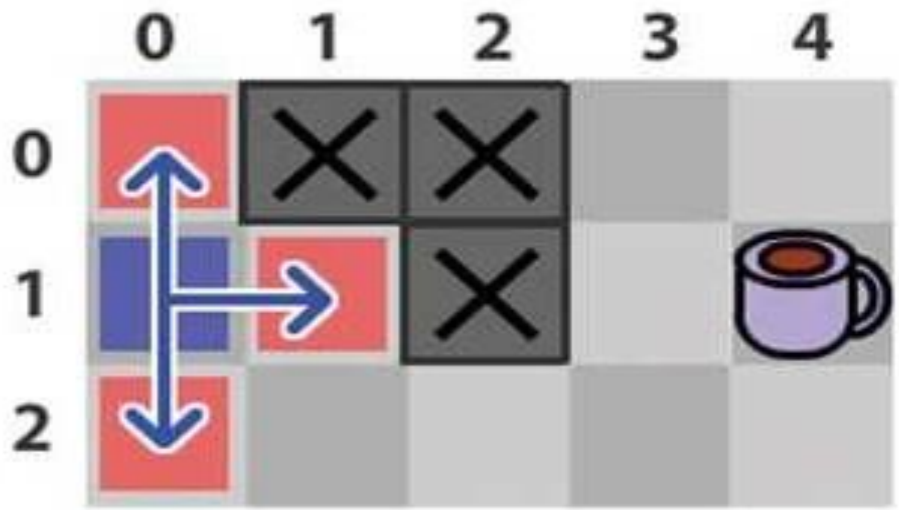


Find my coffee game [1]

Introduction

- Start from a position in a map
- Find the cup of coffee

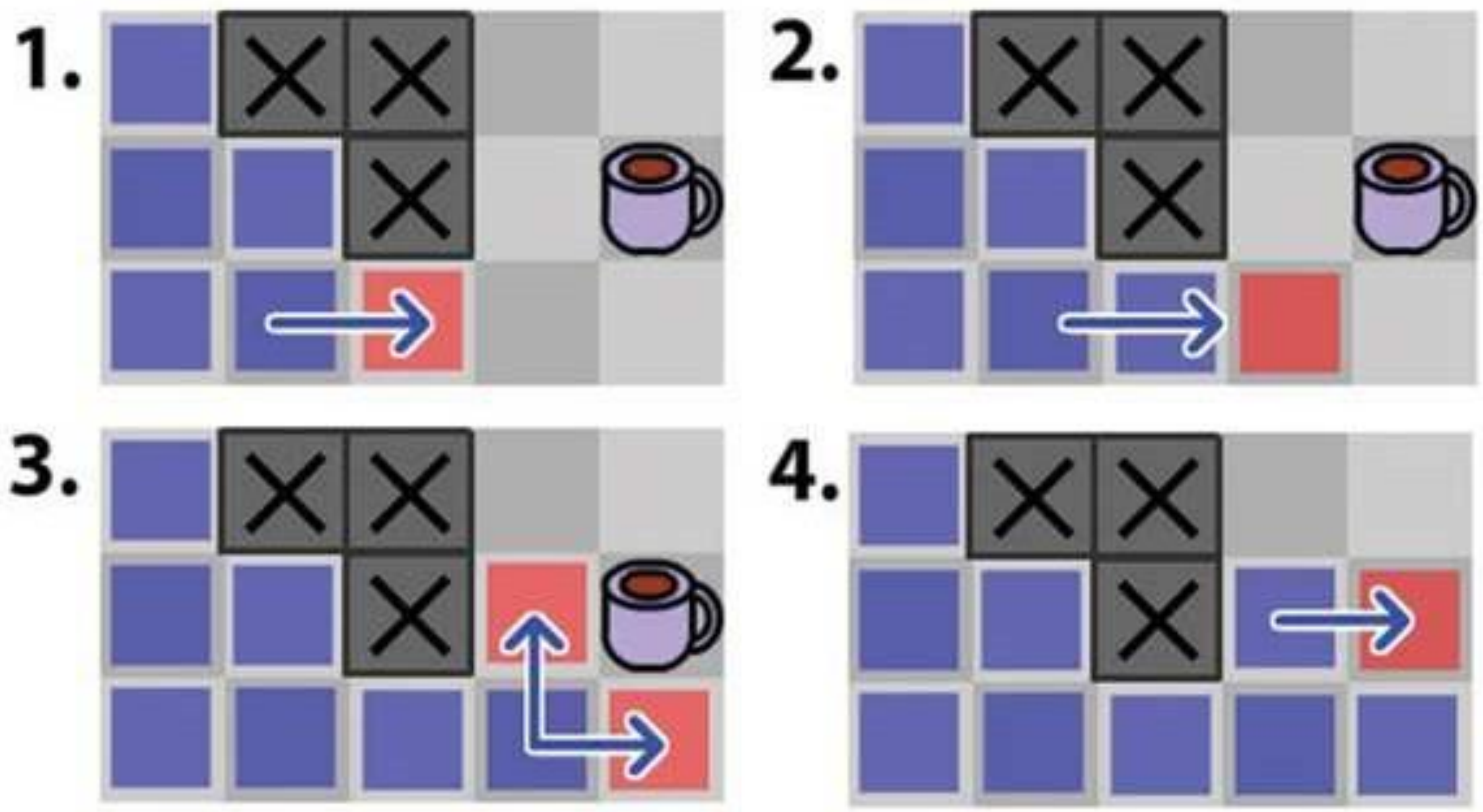




Find my coffee game [2]

Introduction

- BFS is a solution



Find my coffee game [3]

Data representation

- The map
- Using a matrix MxN (M rows, N columns)

	0	1	2	3	4
0	G	X	X	G	G
1	Y	G	X	G	C
2	G	G	G	G	G

Find my coffee game [4]

Data representation

- The graph
- Each cell in the map is a node of the graph. Nodes are indexed from 0 to M*N – 1.
- With node v, the corresponding row and column are: row = v / N; column=v %N;

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14

Find my coffee game [5]

Data representation

- Each node has a list of adjacent nodes.
- Node 0:
  - {5}
- Node 5:
  - {0, 6, 10}
- Node 9:
  - {8, 4, 14}

	0	1	2	3	4
0	G	X	X	G	G
1	Y	G	X	G	C
2	G	G	G	G	G

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14

Find my coffee game [6]

- **Vertex ADT**

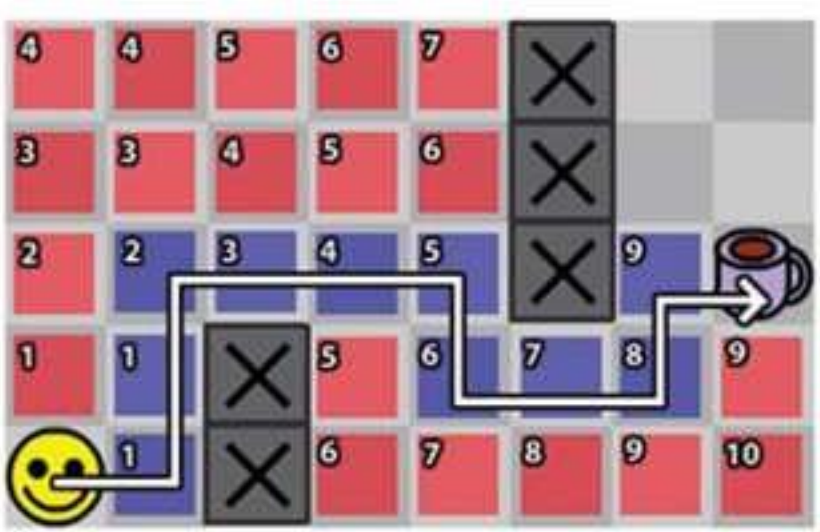
```
GVertex
- index : int //Position in the vertices list
- row, column : int //Which cell in the map
- marked : boolean  //Used in BFS
- numOfAdjVertex : int
- adjVertex : int[ ]  //List of adjacent vertices
________________________________
+ GVertex(index, map, m, n) : void
+ getIndex() : int
+ getRow() : int
+ getColumn() : int
+ ...
```

Find my coffee game [7]

- **Other variables**
  - char[][] map (The map)
  - GVertex [] listVertex (Array contains all M*N vertices)
  - GVertex startVertex, endVertex
    - Your position and position of the coffee cup.
  - ArrayQueue q
    - Vertex queue used in BFS(each queue's item is a vertex)
  - GVertex [] path
    - Path information

Find my coffee game [8]

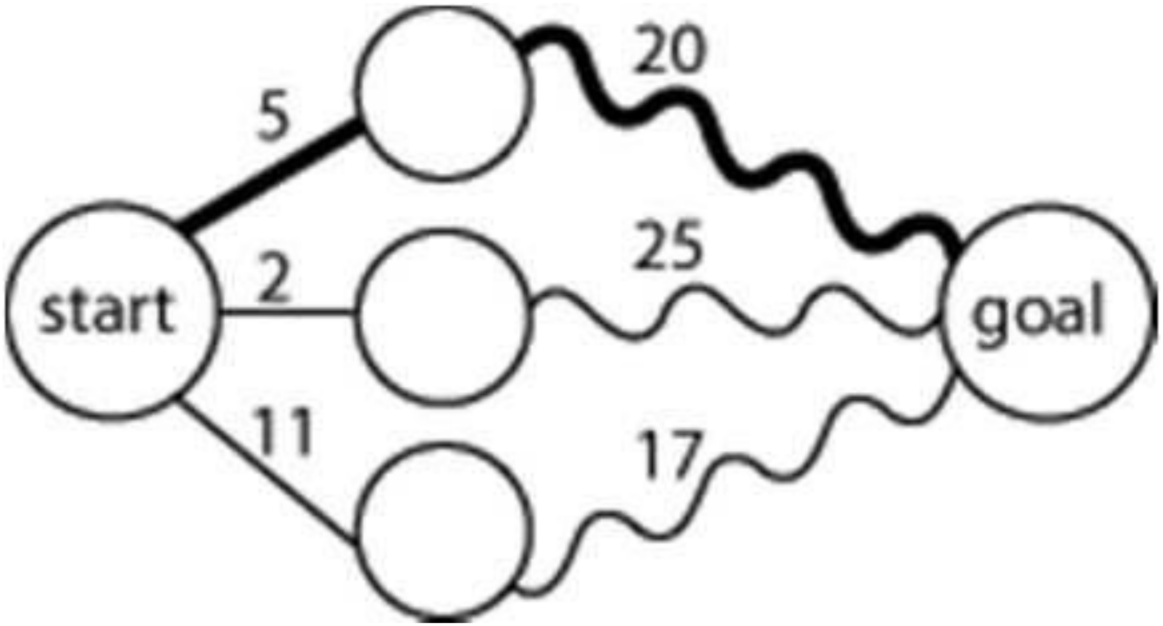
- **Implementation**
  - See Week 12 Example Code for more detail.



	0	1	2	3	4	5	6	7
0	G	G	G	G	G	X	G	G
1	G	G	G	G	G	X	G	G
2	G	G	G	G	G	X	G	C
3	G	G	X	G	G	G	G	G
4	Y	G	X	G	G	G	G	G

(4,0) → (4,1) →(3,1) →(2,1) →(2,2) →(2,3) →(2,4) →(3,4)→(3,5) →(3,6) →(2,6) →(2,7)

Shortest path



Recall a weighted graph [1]

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge.
- Edge weights may represent distances, costs, etc.
- Example:
  - In a train route graph, the weight of an edge represents the distance between the endpoint cities

Recall a weighted graph [2]

	STU	FRA	HAN	COP	BER	LEI
STU	0	210	∞	∞	∞	480
FRA	210	0	350	∞	545	395
HAN	∞	350	0	475	290	∞
COP	∞	∞	475	0	435	∞
BER	∞	545	290	435	0	190
LEI	480	395	∞	∞	190	0

w[i][i]=0
w[i][j]=w[j][i] if G is undirected
w[i][j]=∞ if there is no edge connect vertex i to vertex j

Shortest path problem [1]

- **Definition**
  - In a graph **G=(V,E)**, consider to a **path from s to t**.
    - **P_k={V_1→ V_2→ V_3 → … → V_k}**.
  - Let **W(P_k)** is the **weight of the path** P_k we have:
  - If G is an **un-weighted graph**, then:

    W(P_k)=The number of vertices in P_k=**k**
  - If G is **a weighted graph**, then:

$$W(P_k) = \sum_{i=1}^{k-1} w(V_i, V_{i+1})$$

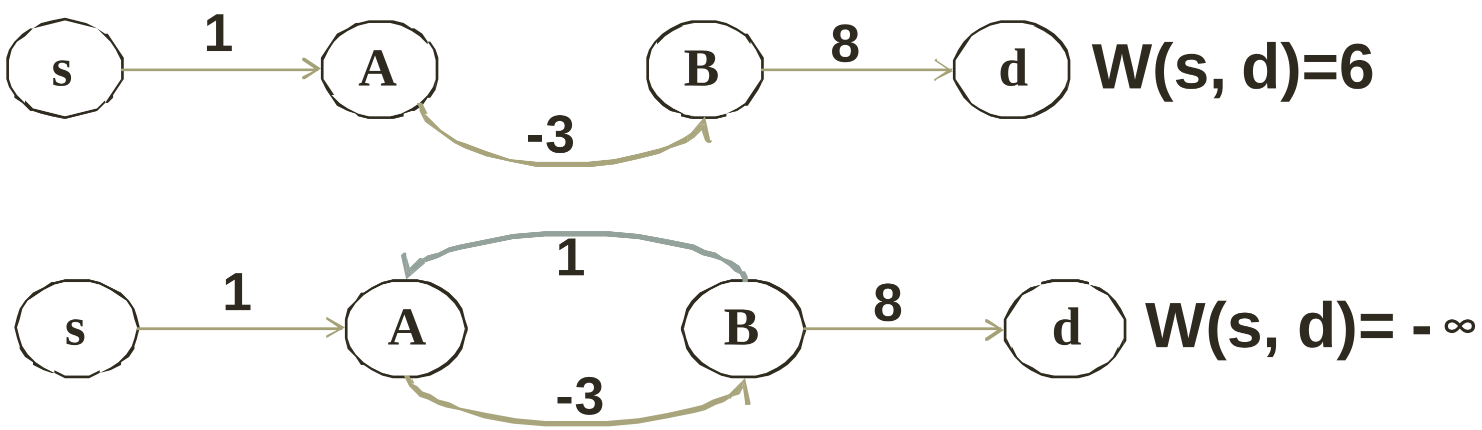
  - Find a P_k so that **W(P_k) is minimum**.

Shortest path problem [2]

- Shortest path properties
  - A **sub-path** of a shortest path is **itself a shortest path**.
  - For example:
    If $P_{sp}=\{V_1 \rightarrow V_3 \rightarrow V_6 \rightarrow V_4 \rightarrow V_5 \rightarrow V_7\}$ is the shortest path from V_1 to V_7, then the path $\{V_3 \rightarrow V_6 \rightarrow V_4\}$ is the shortest path from V_3 to V_4.
  - There is a **tree of shortest path** from a vertex to all other vertices that connect to this vertex.

Shortest path problem [3]

- Shortest path in an un-weighted graph
  - **BFS is a solution**
- Shortest path in a weighted graph
  - Every edge has **positive weight**
  - Edge may have **negative weight**
    - Graph may have **negative weight cycle**



Shortest path problem [4]

- Shortest path algorithm types:
  - **Single-pair shortest path (SPSP)**
    - Find shortest paths from a given vertex to a given vertex.
  - **Single-source shortest path (SSSP)**
    - Find shortest paths from a given vertex to all other vertices.
  - **All-pairs shortest path (APSP)**
    - Find shortest paths for every pair of vertices.

Dijkstra algorithm [1]

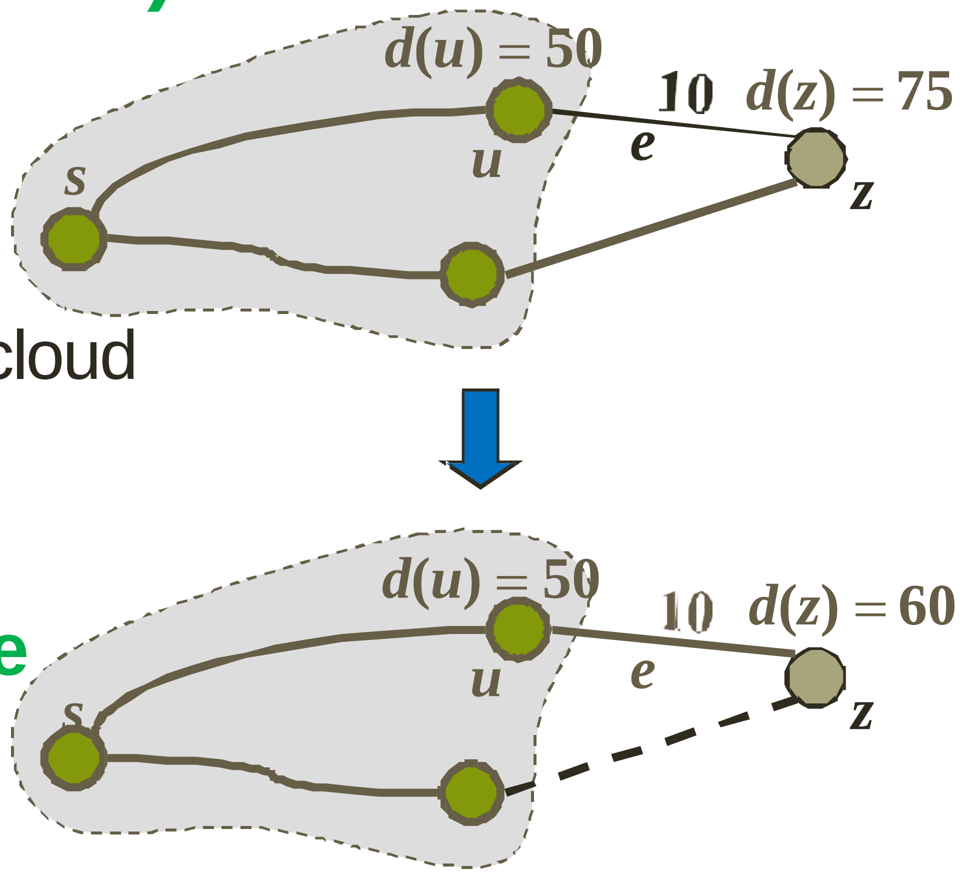
- A **SSSP algorithm**
  - Find shortest paths from **vertex s** to all other vertices.
  - **Assumption**
    - The graph is connected.
    - The edge **weights are non-negative**.
  - Definition of the **distance** from **s to v**
    - D[v] is the total weight of the shortest path from s to v.
    - With a given vertex s, Dijkstra algorithm will compute the D[v] for all v in V.

Dijkstra algorithm [2]

- **Dijkstra algorithm idea**
  - We grow a **“cloud” of vertices**, beginning with s and eventually covering all the vertices.
  - We store with each vertex **v** a label **d(v)** representing **the distance of v from s in the sub-graph** consisting of the cloud and its adjacent vertices
  - At each step:
    - We **add** to the cloud the vertex **u** outside the cloud with the smallest distance label, **d(u)**
    - We **update** the labels of the vertices adjacent to **u**

Dijkstra algorithm[3]

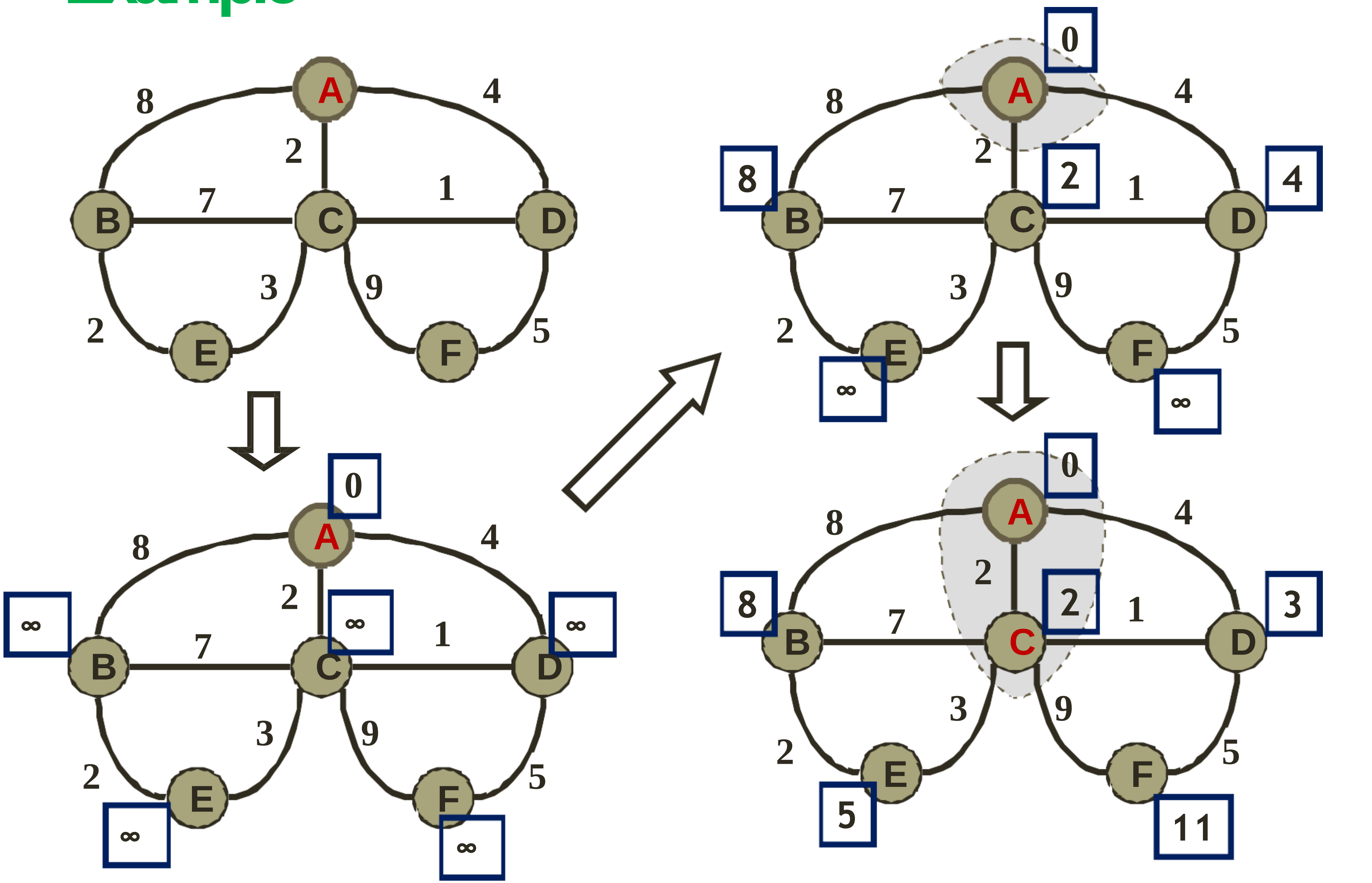
- **Update (Edge relaxation)**
- Consider an edge **e = (u,z)** such that:
  - **u** is the vertex most recently added to the cloud
  - **z** is not in the cloud



- The relaxation of edge **e** updates distance **d(z)** as follows:
  d(z) $\leftarrow$ min{**d(z),d(u)** + **weight(u,z)**}

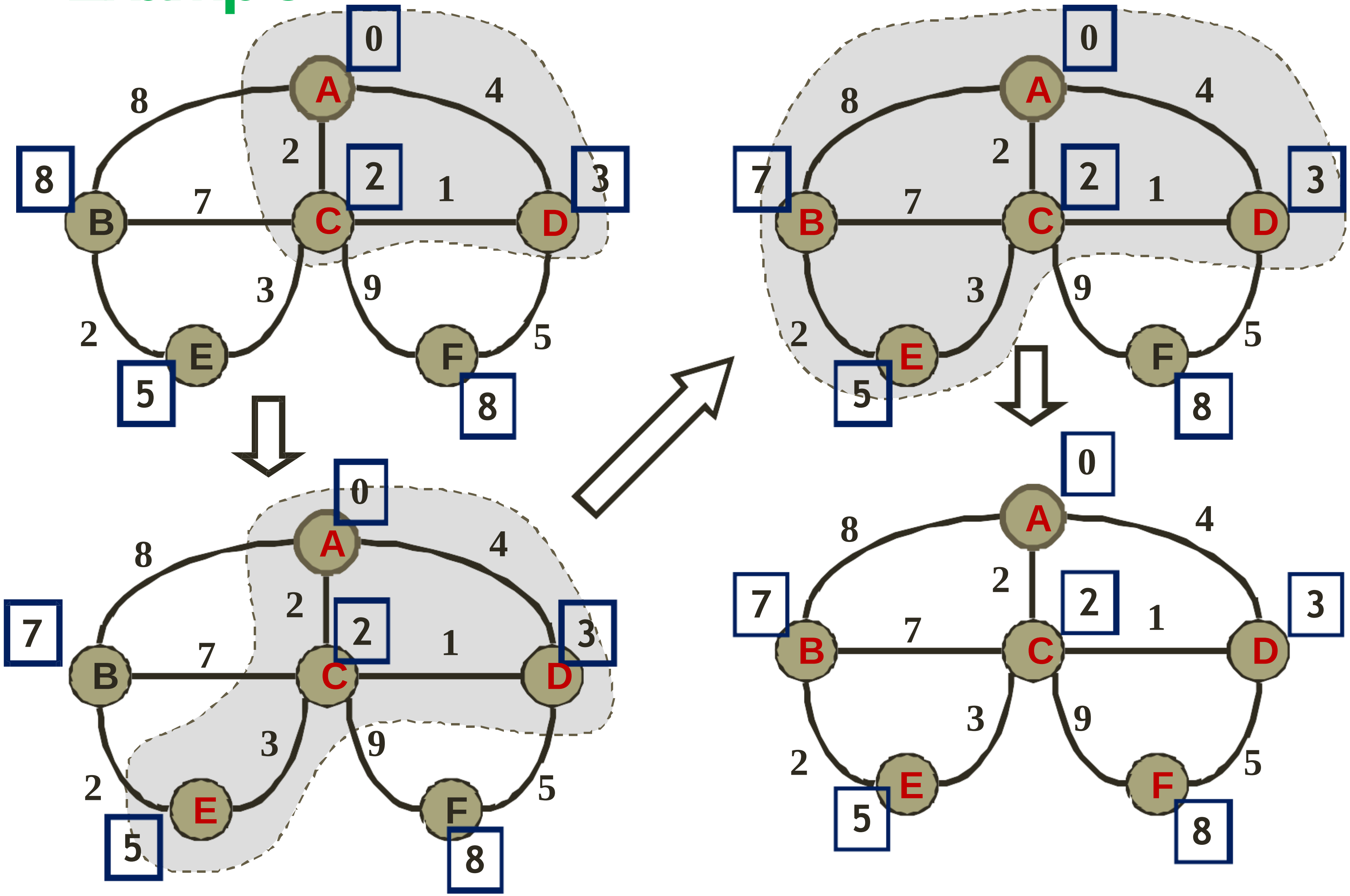
Dijkstra algorithm[4]

- Example



Dijkstra algorithm[5]

- Example



Dijkstra algorithm[6]

- **Dijkstra algorithm**

Code

```
Algorithm  Dijkstra(V,E,s):
    Input: A graph G=(V,E), w is the weighted matrix, s is the
           source vertex.
    Output: The shortest paths from s to other vertices.
  For each v in V do
     D[v] ← ∞
     path[v] ← -1
  D[s] ←     0
  Create empty S set      //The “cloud” set
  Do
     Find vertex u in V \ S so that D[u] is minimum
     Move u from V to S
     For each z in V \ S do
        D[z] ←      min{D[z], D[u]+w[u][z]}
        path[z] ← u  //Only update if D[u]+w[u][z]<D[z]
  While could not find any u
```

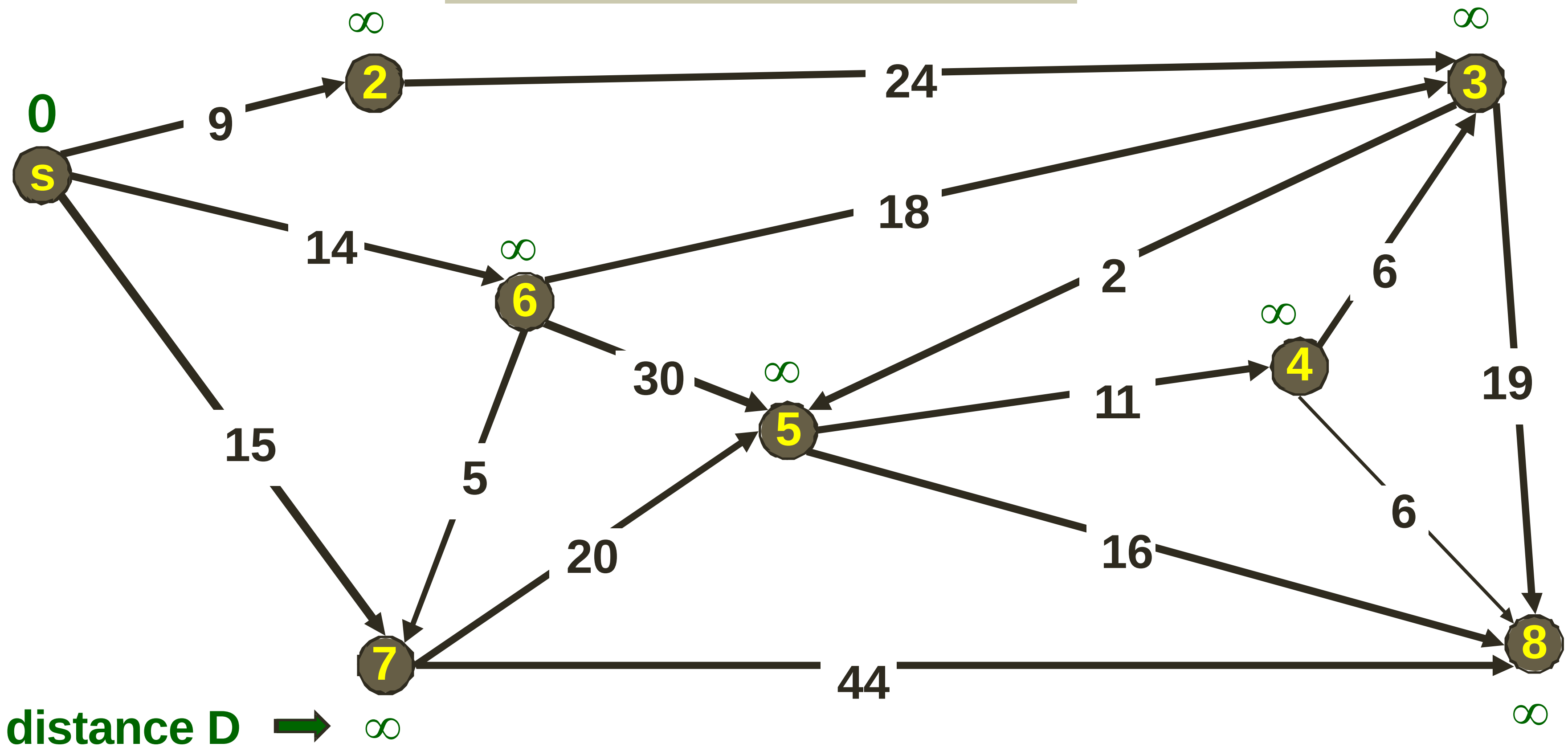
Dijkstra algorithm[7]

- **Dijkstra algorithm demonstration**

S ={ }

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

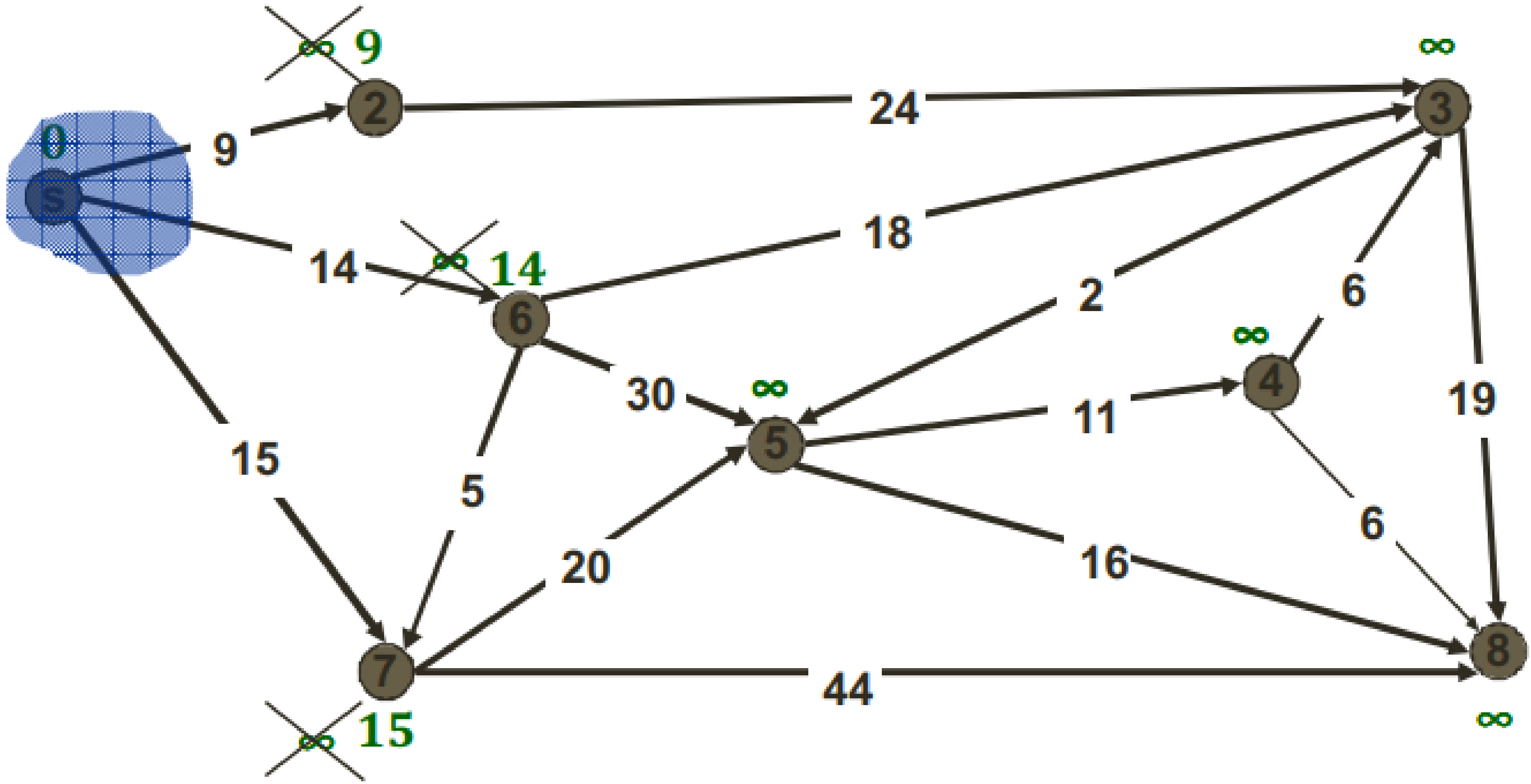
D = {0, ∞, ∞, ∞, ∞, ∞, ∞, ∞}



Dijkstra algorithm[8]

- **Dijkstra algorithm demonstration**

S = {s}
V = { -, 2, 3, 4, 5, 6, 7, 8}
D = {0, 9, ∞, ∞, ∞, 14, 15, ∞}



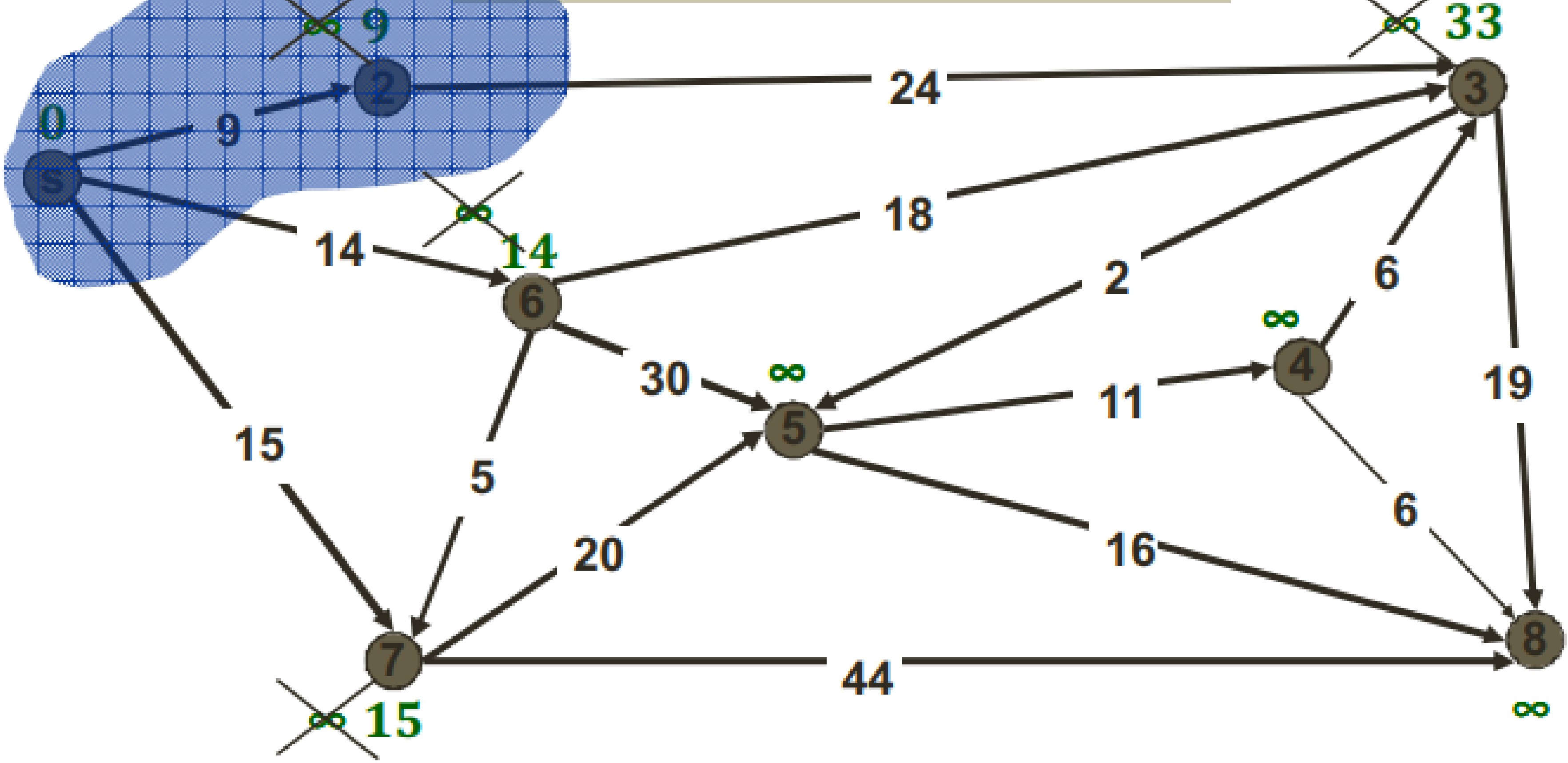
Dijkstra algorithm[9]

- **Dijkstra algorithm demonstration**

S ={s, 2}

V = { -, -, 3, 4, 5, 6, 7, 8}

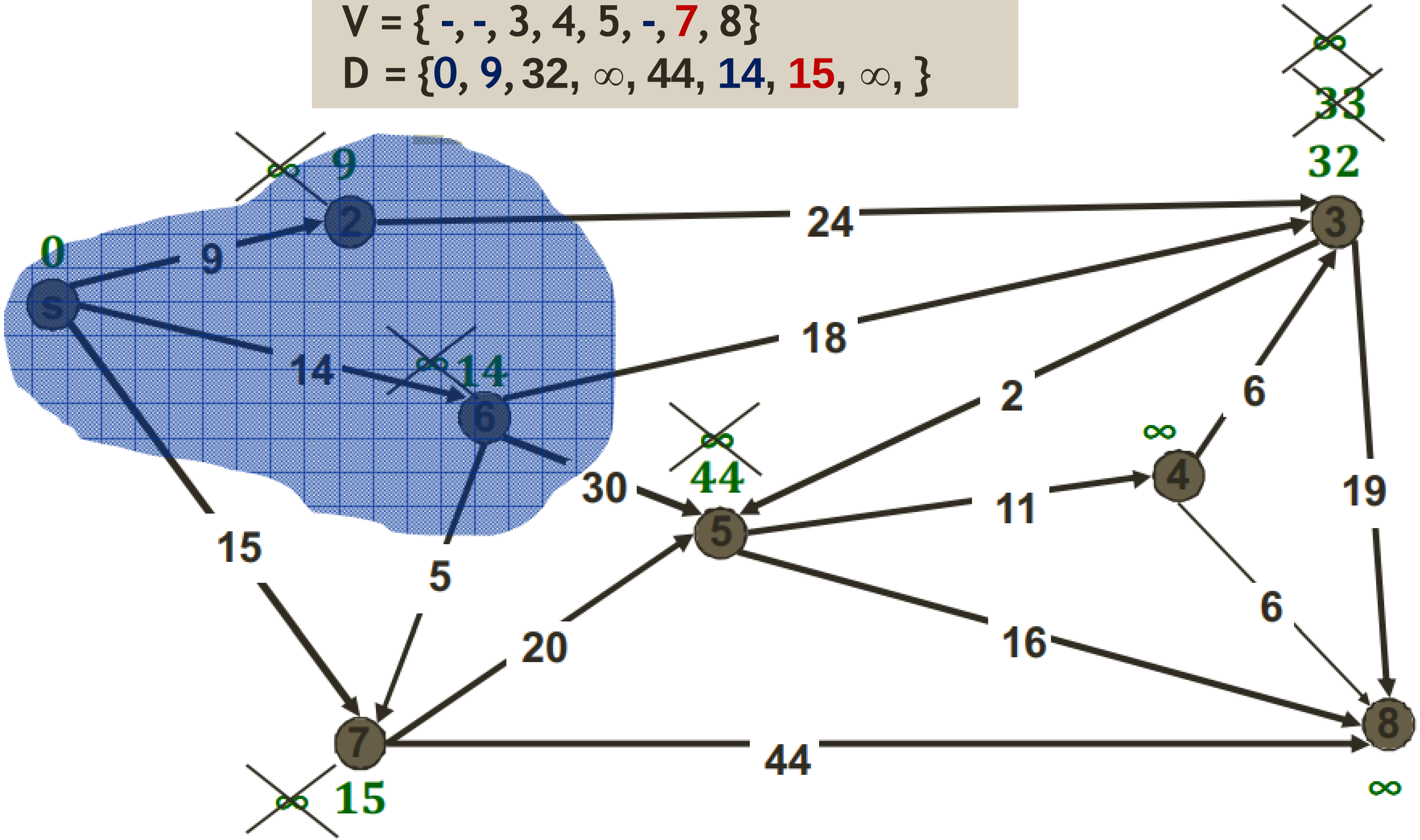
D = {0, 9, 33, ∞, ∞, 14, 15, ∞}



Dijkstra algorithm[10]

- **Dijkstra algorithm demonstration**

S ={s, 2, 6}
V = { -, -, 3, 4, 5, -, 7, 8}
D = {0, 9, 32, ∞, 44, 14, 15, ∞, }



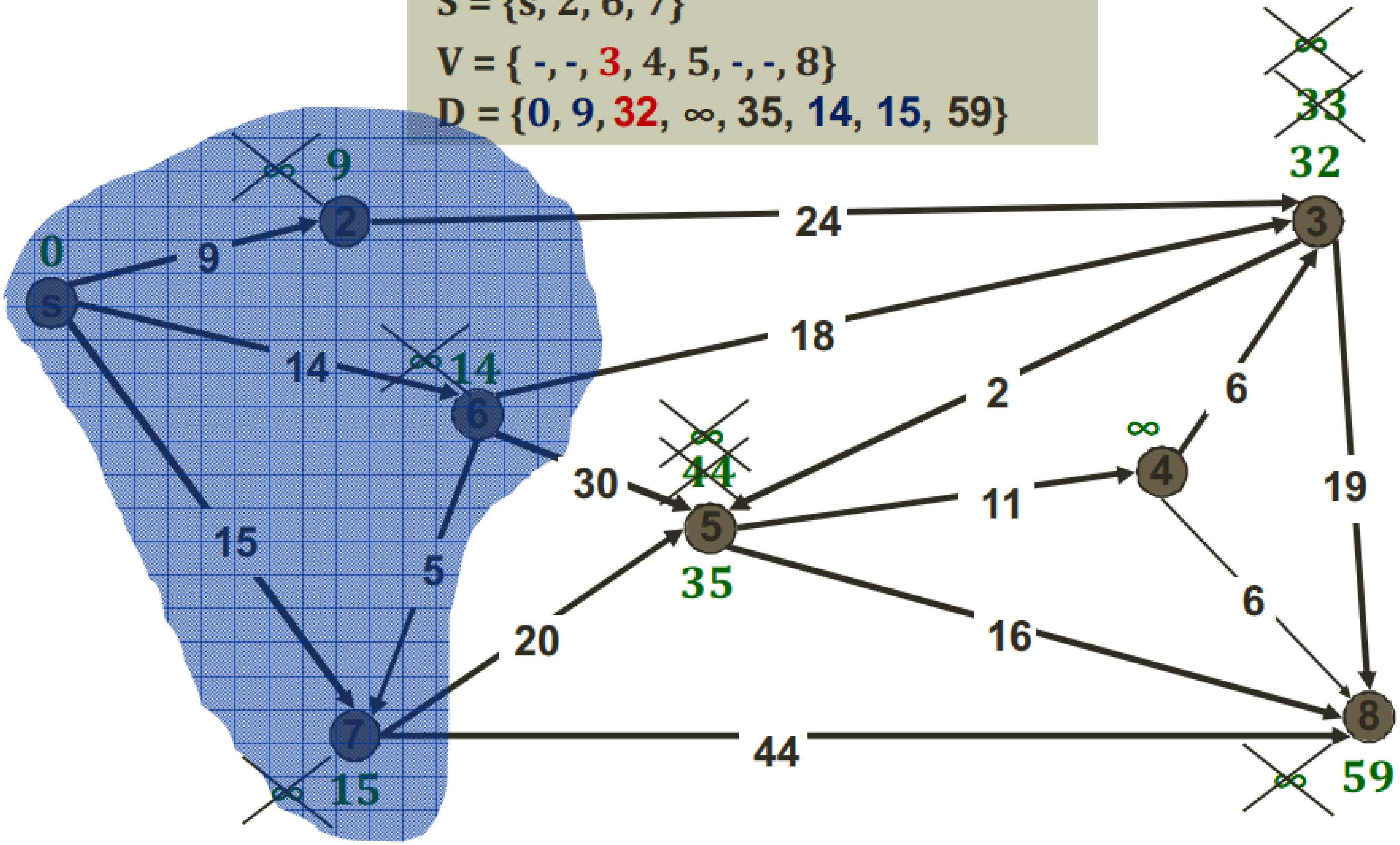
Dijkstra algorithm[11]

- **Dijkstra algorithm demonstration**

S = {s, 2, 6, 7}

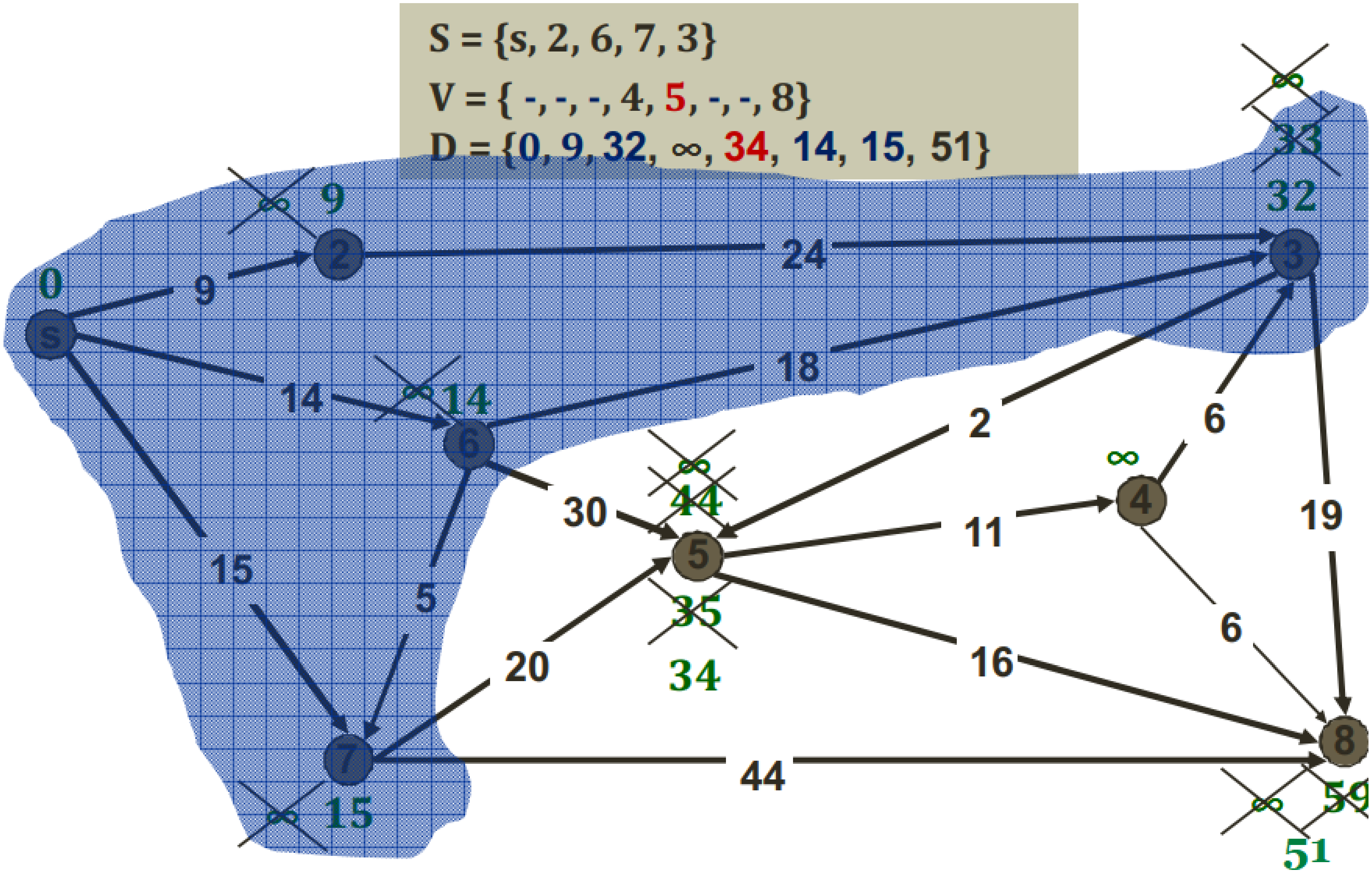
V = { -, -, 3, 4, 5, -, -, 8}

D = {0, 9, 32, ∞, 35, 14, 15, 59}



Dijkstra algorithm[12]

- **Dijkstra algorithm demonstration**



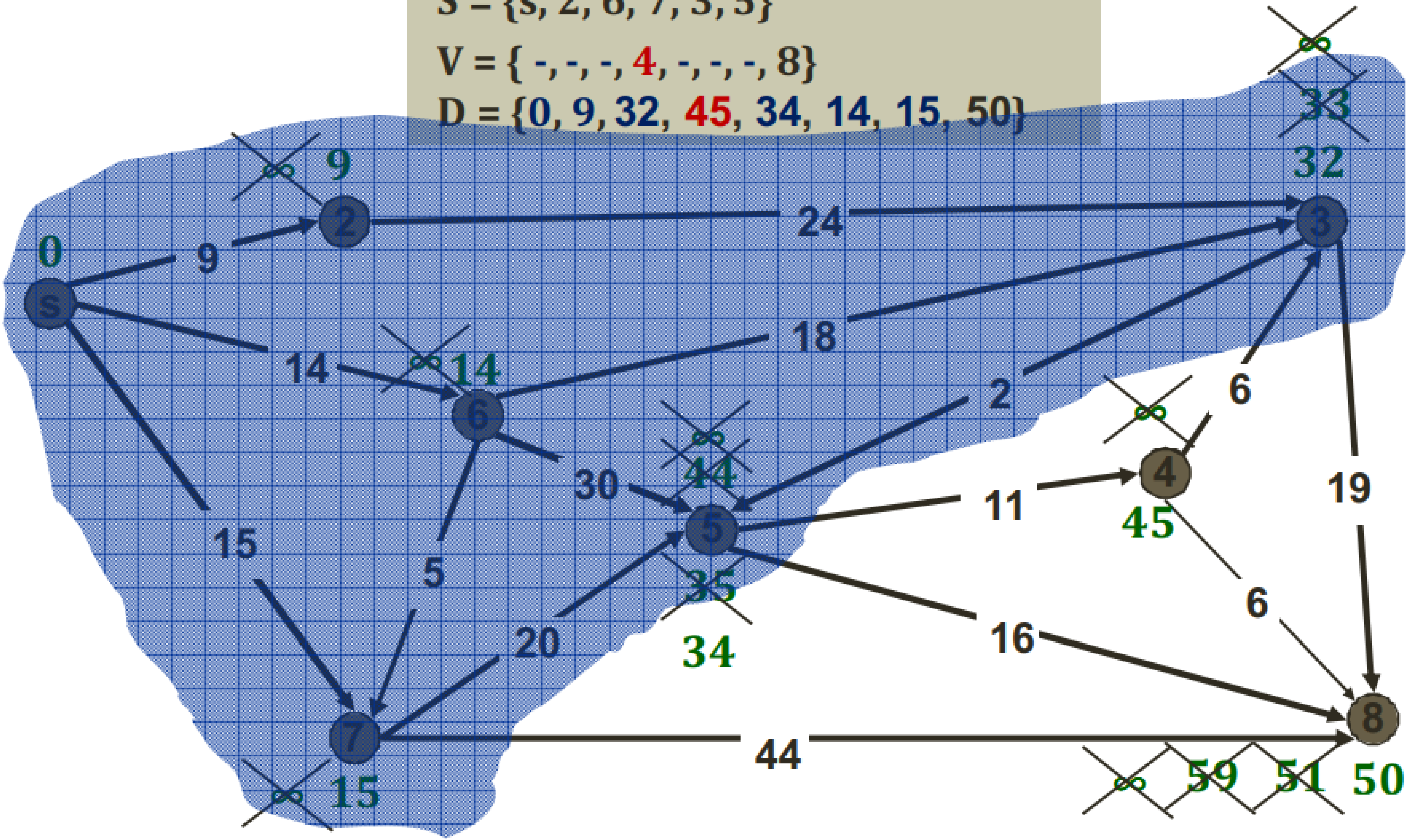
Dijkstra algorithm[13]

- **Dijkstra algorithm demonstration**

S = {s, 2, 6, 7, 3, 5}

V = { -, -, -, 4, -, -, -, 8}

D = {0, 9, 32, 45, 34, 14, 15, 50}



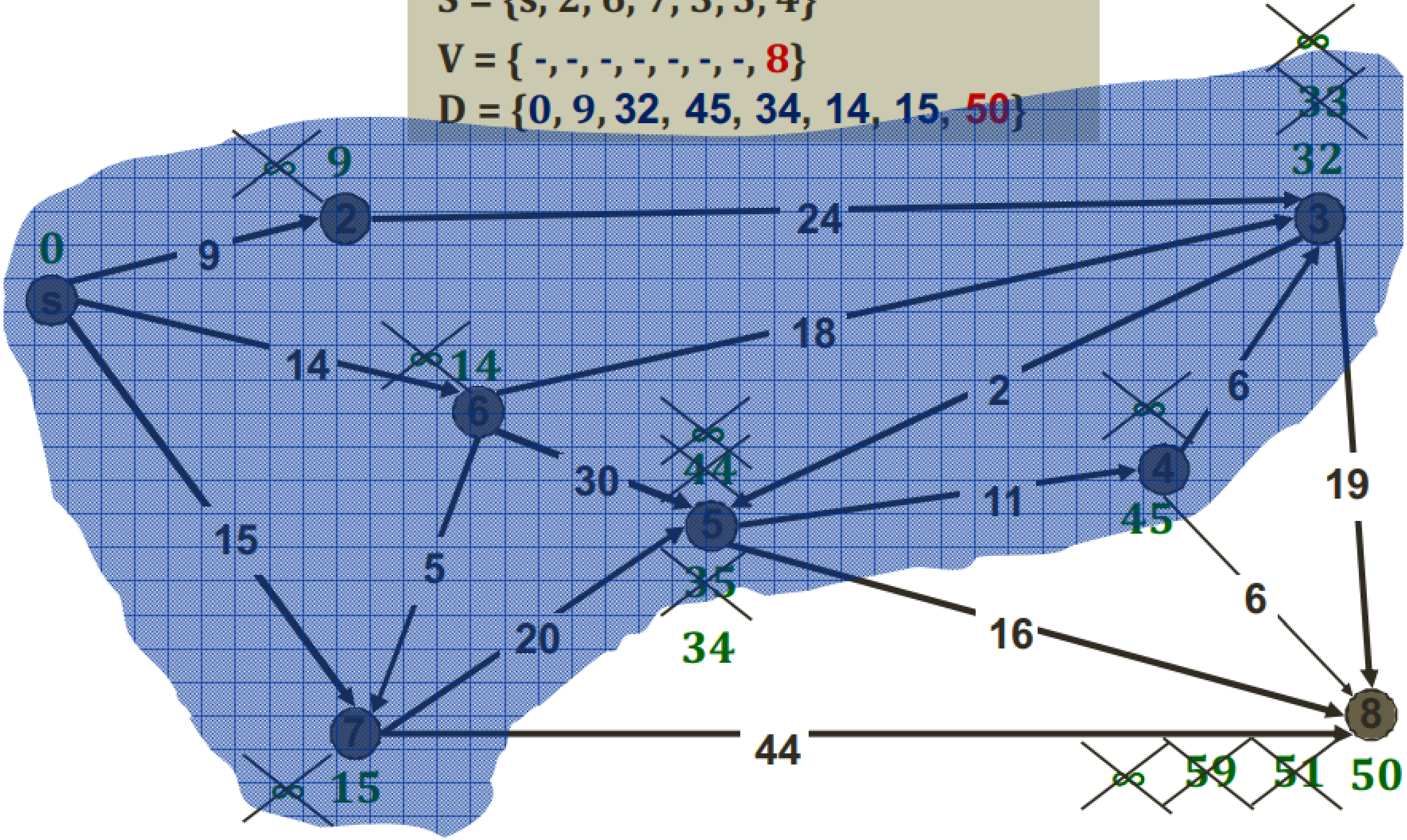
Dijkstra algorithm[14]

- **Dijkstra algorithm demonstration**

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

V = { -, -, -, -, -, -, -, 8}

D = {0, 9, 32, 45, 34, 14, 15, 50}



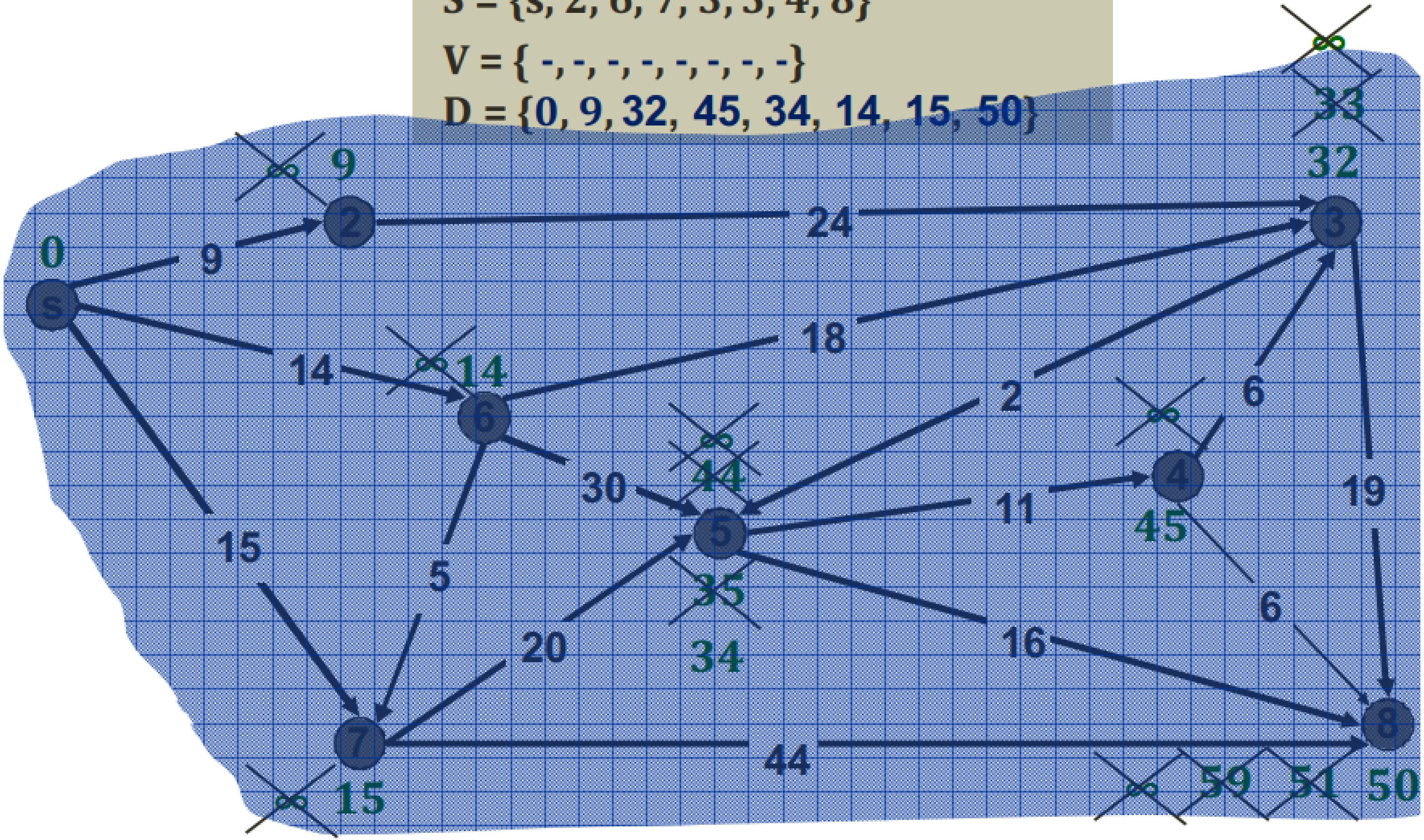
Dijkstra algorithm[15]

- **Dijkstra algorithm demonstration**

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

V = { -, -, -, -, -, -, -, -}

D = {0, 9, 32, 45, 34, 14, 15, 50}



Dijkstra algorithm[16]

Dijkstra algorithm demonstration:
Summarize steps with table:

D(s)	D(2)	D(3)	D(4)	D(5)	D(6)	D(7)	D(8)
0	∞	∞	∞	∞	∞	∞	∞
	9(s)	∞	∞	∞	14(s)	15(s)	∞
		33(2)	∞	∞	**14(s)**	15(s)	∞
		32(6)	∞	44(6)		**15(s)**	∞
		32(6)	∞	35(7)			59(7)
			∞	**34(3)**			51(3)
			45(5)				50(5)
							50(5)

Dijkstra algorithm[17]

- **Dijkstra algorithm analysis**
  - Textbook **page 379**, supplementary textbook **page 658**.
  - The time complexity of Dijkstra algorithm is:
    - **$O(|E| + |V|^2)$ by using array to store V and D[v].**
    - **$O(|E|\log V + |V|\log V)$ by using a priority queue to store V and D[v].**
  - Dijkstra algorithm doesn't work with negative-weighted edge.
  - Graph can be directed or un-directed.

Bellman-Ford algorithm [1]

- A SSSP algorithm
  - Find shortest paths from **vertex s** to all other vertices.
  - Works even with **negative-weight edges**.
  - Can detect the existence of **negative-weight cycle** reachable from s.

- Assumption

- The graph is connected.
- The edges **are directed**.

Bellman-Ford algorithm [2]

- Definition of **D[i][v]**:
  - D[i][v] is the total weight of the shortest path **that use i or fewer edges**, from **s** to **v**.
  - Recall that the distance from vertex u to vertex v is the total weight of the shortest path from u to v.
  - Then D[i][v] is the **distance** from **s** to **v** **using i or fewer edges**.
    - D[i][s]=0
    - D[i][v]=∞ if you can't get to v within i edges.
  - Bellman-Ford algorithm **computes the D[i][v] for any v in V and i=1 .. |V|**.

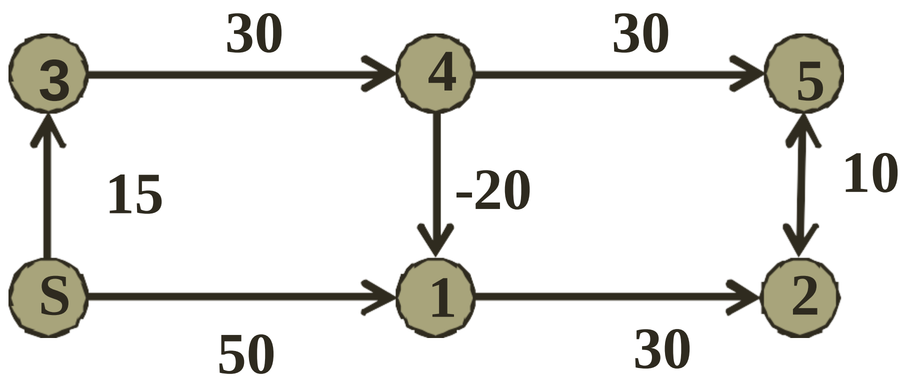
Bellman-Ford algorithm [3]

- **Algorithm idea**
  - To compute D[i][v], we **calculate i_{th} row from $i-1_{th}$ row**.
  - Consider a vertex u so that there is an edge from u to v.
  - We know the shortest path from s to u using i-1 or fewer edges, D[i-1][u].
  - Then:

$$D[i][v] = \min_{w[u][v] \neq \infty} \{D[i-1][u] + w[u][v]\}$$

Bellman-Ford algorithm [4]

- **Example**:



$$D[i][v] = \min_{w[u][v] \neq \infty} \{D[i-1][u] + w[u][v]\}$$

	S	1	2	3	4	5
0	0	∞	∞	∞	∞	∞
1	0	50	∞	15	∞	∞
2	0	50	80	15	45	∞
3	0	25	80	15	45	75
4	0	25	55	15	45	75
5	0	25 (3,4)	55 (3,4,1)	15	45 (3)	65 (3,4,1,2)

Bellman-Ford algorithm [5]

- **Bellman-Ford algorithm**

Code

Algorithm BellmanFord(V,E,s): boolean

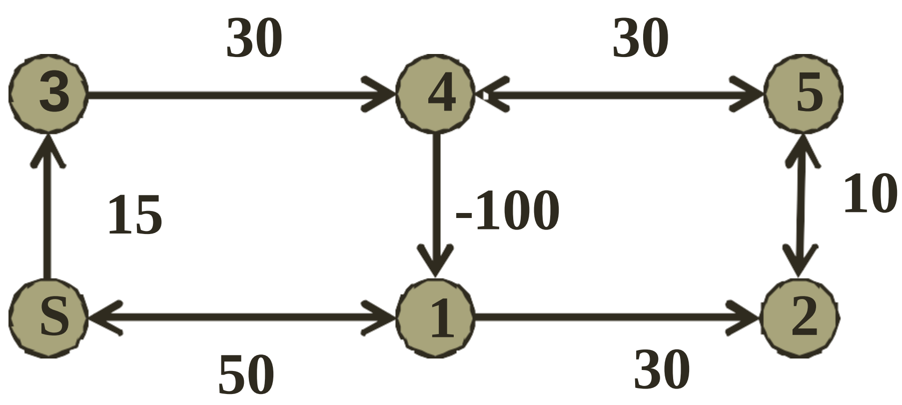
Input: A graph G=(V,E), w is the weighted matrix, s is the source vertex.

Output: The shortest paths from s to other vertices or false if G has a negative-weighted cycle.

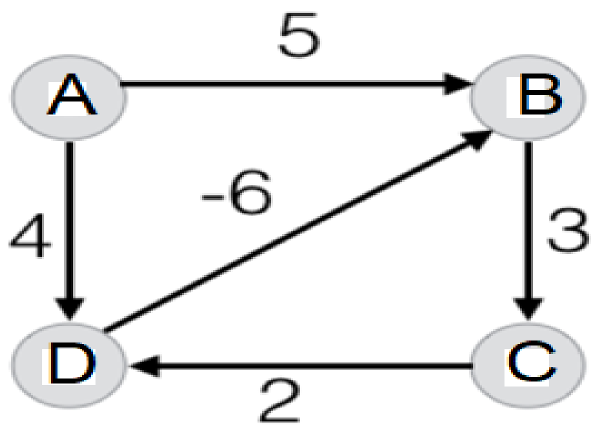
```
For i ← 0 to |V| do
    For each v in V do
        D[i][v] ← ∞
    path[v] ← -1
    D[i][s] ← 0
For i ← 1 to |V| do
    For each v in V do
        For each e=(u,v) in E do
            D[i][v] ← Min{D[i-1][u]+w[u][v]}
            path[v] ← u_min  //  D[i][v]=D[i-1][u_min]+w[u_min][v]
```

Bellman-Ford algorithm [6]

- **Algorithm analysis**
  - If the graph contains no negative-weight cycles reachable from the source vertex s, **after |V| - 1 iterations** all distance estimates represent shortest paths.
  - If the graph contains a negative-weight cycles reachable from s.



Bellman-Ford algorithm [7]



	A	B	C	D	Note
0	0	∞	∞	∞	
1	0	5,A	∞	4,A	
2	0	-2,D	8,B	4,A	
3	0	-2,D (A→D→B =-2)	1,B (A→D→B→C =1)	4,A (A→D=4)	Normally, algorithm stops at step 3.
4	0	-2,D	1,B	3,C (A->D->B->C ->D = 3)	Add step 4: The weight is reduced from (4,A) to (3,C) → Graph contains negative cycle