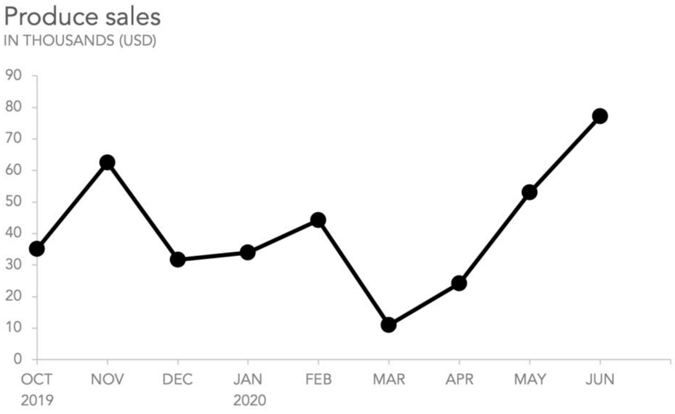
Fu-Yin Cherng

National Taiwan University

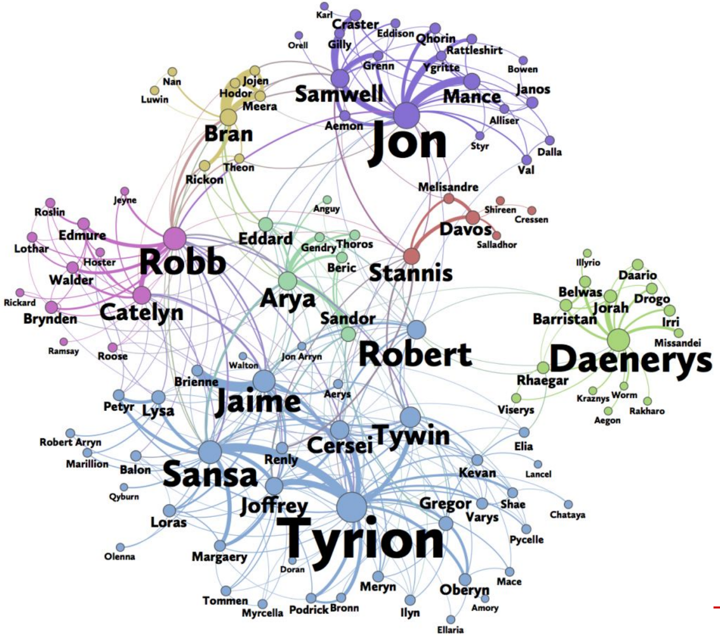
Graph

What is graph?

- a way to illustrate data & represent the relationships among data items



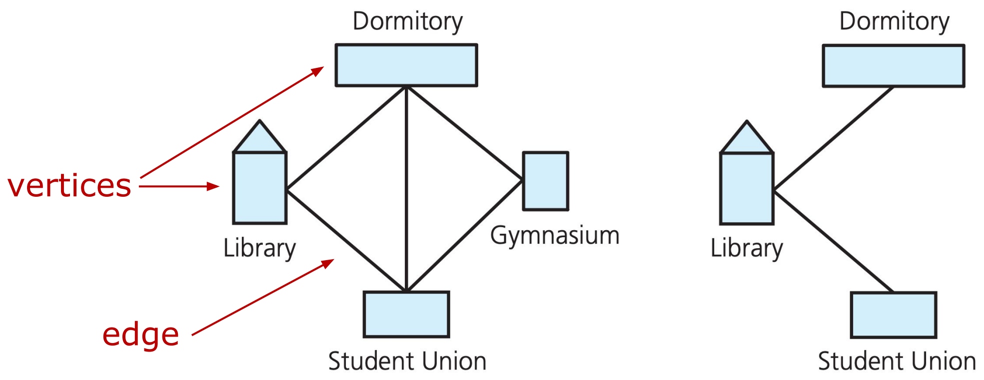
https://www.storytellingwithdata.com/blog/2020/3/24/what-is-a-line-graph



https://predictivehacks.com/social-network-analysis-of-game-of-thrones/

What is graph?

- General definition of graph G consists of 2 sets
  - ***V***: a set of vertices/nodes
  - ***E***: a set of edges that connect the vertices
- Subgraph: a subset of a graph vertices and its edges



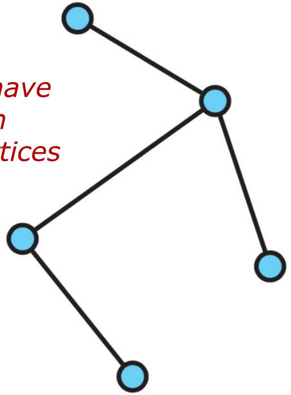
subgraph

Terminology of Graph

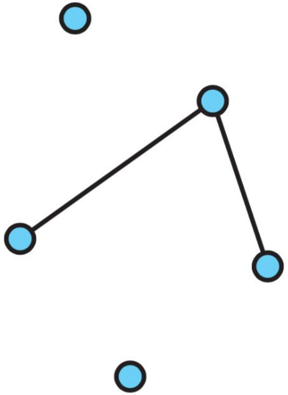
- Adjacent vertices
  - 2 vertices are adjacent if joined by an edge
- Path
  - a path between two vertices is a sequence of edges the begins at one vertex and ends at another vertex
- Simple Path
  - a path passes through a vertex only once
- Cycle
  - a path that begins and ends at the same vertex

Terminology of Graph

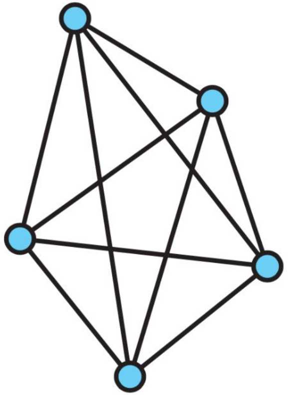
not necessarily have an edge between every pair of vertices



(a)



(b)



(c)

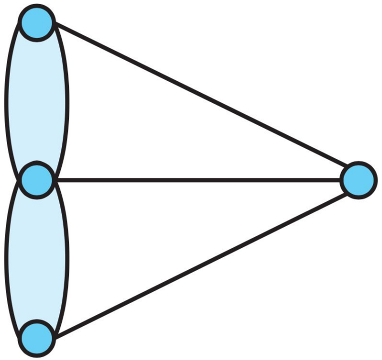
Connected Graph
each pair of distinct vertices has a path between them

Disconnected Graph

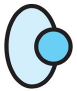
Complete Graph
each pair of distinct vertices has an edge between them

Terminology of Graph

- A graph
  - edge cannot begin and end at the same vertex (self edge)
  - cannot have duplicate edges between vertices
- Multigraph is not a graph and can have multiple edges



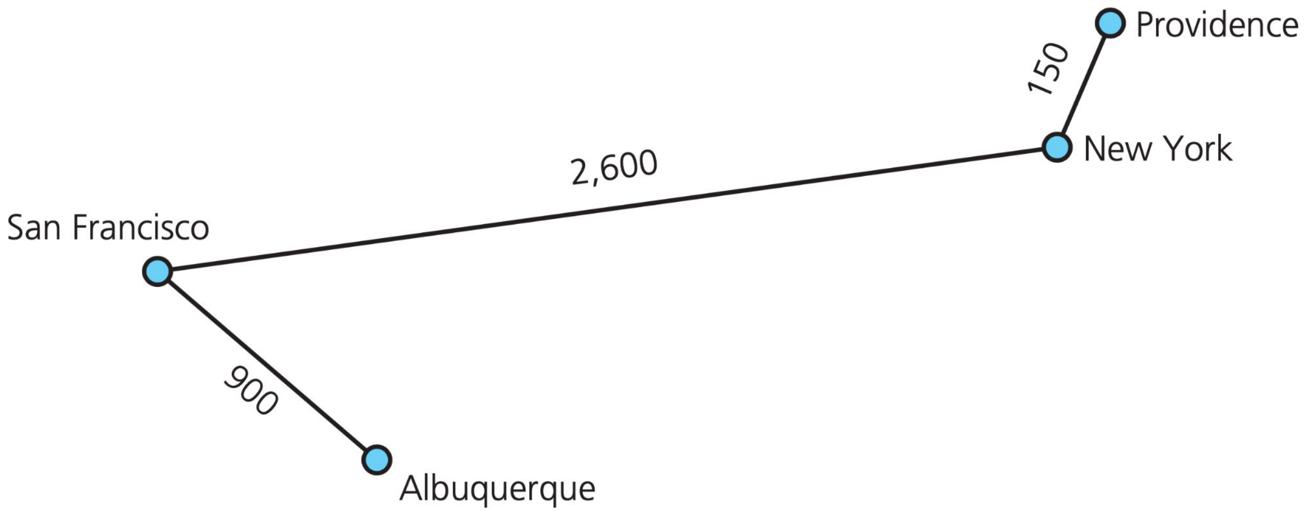
Multigraph



self edge
not allowed in a graph

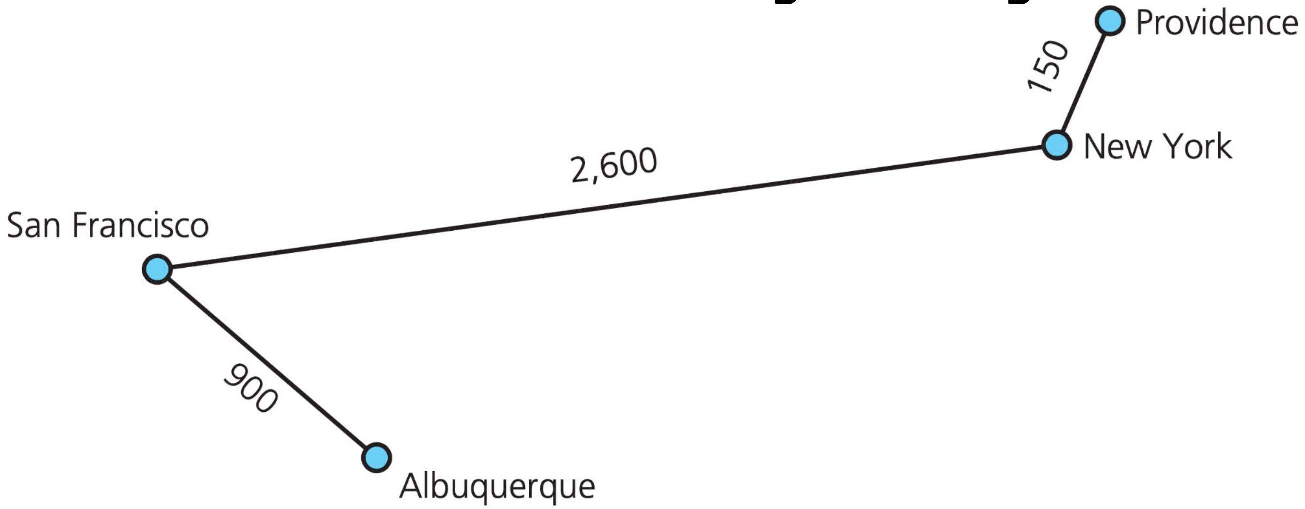
Terminology of Graph

- Weighted graph: edges with numeric labels



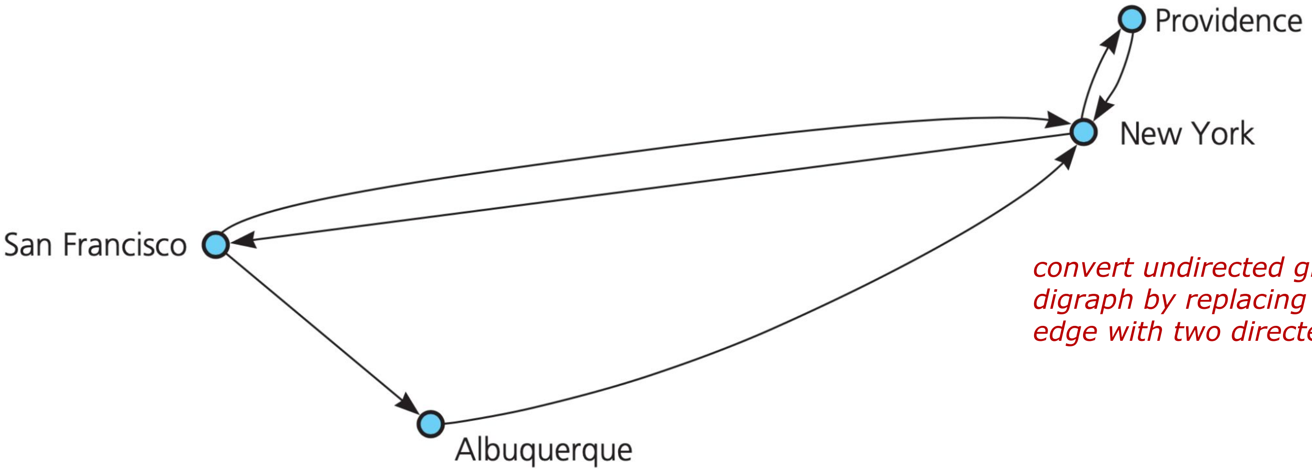
Terminology of Graph

- Undirected graphs: edges do not indicate direaction, can travel either direction along the edges



Terminology of Graph

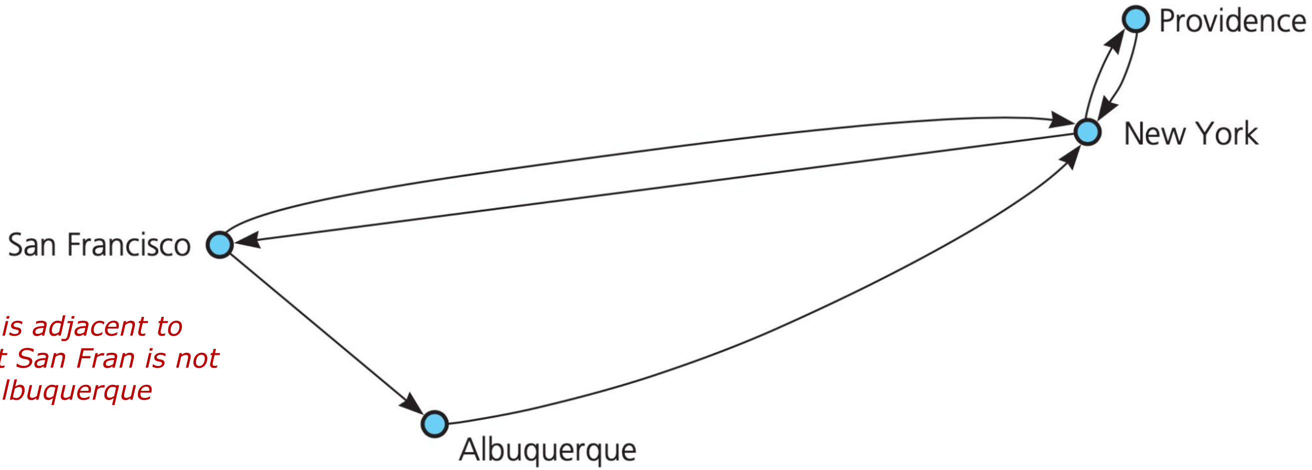
- Directed graph or digraph: graph with directed edge
  - can have 2 edges with different directions between a pair of vertices



convert undirected graph to digraph by replacing each edge with two directed edges

Terminology of Graph

- Directed graph or digraph
  - adjacent vertices: if there is a directed edge from vertex x to vertex y, then y is adjacent to x (x -> y)
  - y is successor of x, x is predecessor of y



Albuquerque is adjacent to San Fran, but San Fran is not adjancet to Albuquerque

Abstract data type: Graph

- define following ADT graph’s vertices contain value
  - you can also design another ADT graph whose vertices don’t contain value.
- many operations, for example
  - Test whether a graph is empty.
  - Get the number of vertices in a graph.
  - Insert an edge between two given vertices in a graph
  - Retrieve from a graph the vertex that contains a given value
  - …

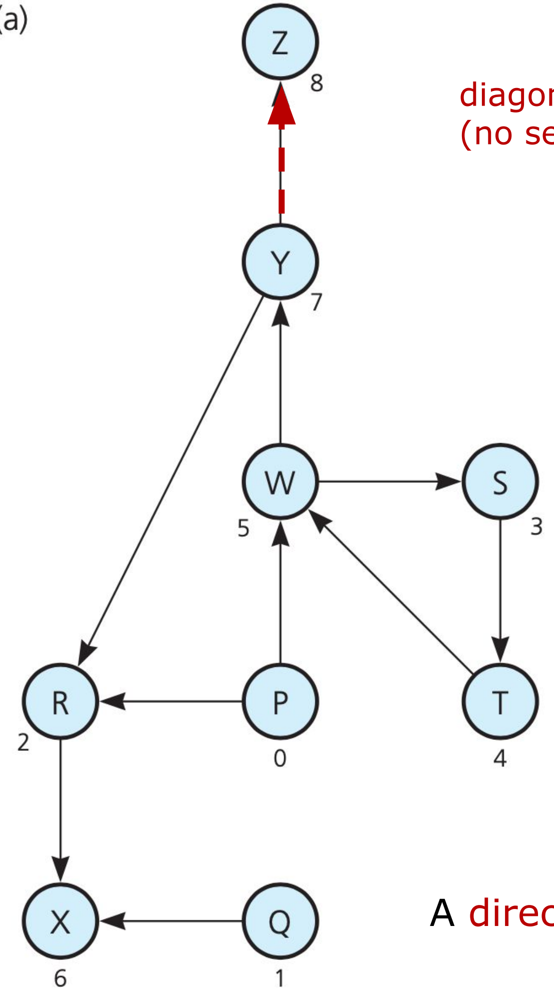
Abstract data type: Graph

```
//An interface for the ADT undirected, connected graph
template<class LabelType>
class GraphInterface{
     public:
          virtual int getNumVertices() const = 0;
          virtual int getNumEdges() const = 0;
          virtual bool add(LabelType start, LabelType end, int edgeWeight) = 0;
          irtual bool remove(LabelType start, LabelType end) = 0;
          virtual int getEdgeWeight(LabelType start, LabelType end) const = 0;
          virtual void depthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;
          virtual void breadthFirstTraversal(LabelType start, void visit(LabelType&)) =
     0;
}
```

Implementing Graphs: adjacency matrix

- use adjacency matrix or adjaceny list
- for a graph with *n* vertices numberred *0, 1, …, n-1*
  - adjacent matrix: `matrix[n][n]`
    - unweighted:
      - `matrix[i][j]` is 1 (true) if there is an edge from vertex i to vertex j, and 0 (false) otherwise
    - weighted
      - `matrix[i][j]` is the weight that labels the edge from vertex i to vertex j, and ∞ otherwise

(a)



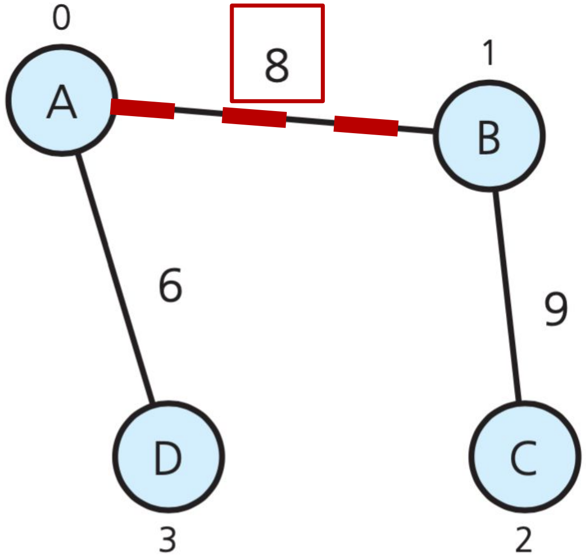
(b)

diagonal entries is 0
(no self edge)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

A directed + unweighted graph and its adjacent matrix undirected graph is symmertical

(a)



(b)

		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

A undirected + weighted graph and its adjancet matrix

Implementing Graphs: adjacency matrix

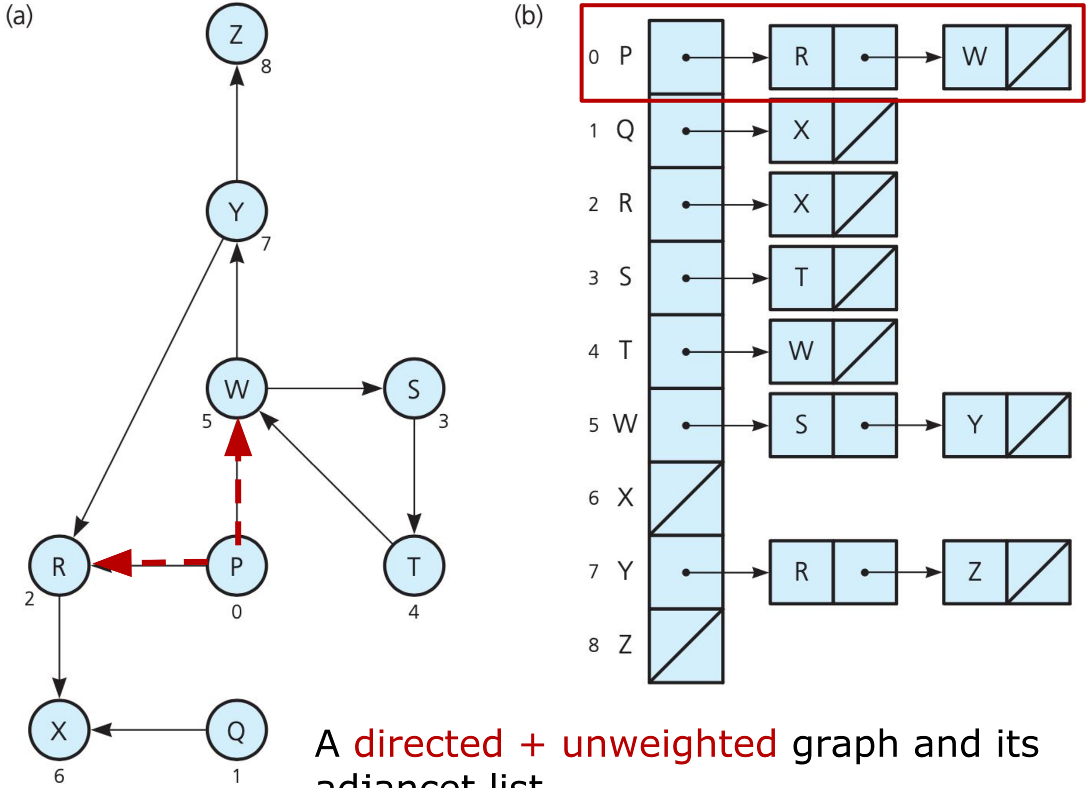
- use second array to represent the n vertex values
- `values[i]` is the value in vertex i

Implementing Graphs: adjacency list

- use adjacency matrix or adjacency list
- for a graph with *n* vertices numberred *0, 1, …, n-1*
  - adjancey list: consists of `n` linked chains
  - see example directly

Implementing Graphs: adjacency list

- the `ith` linked chain has nodes for vertices with edge to vertex i
- If the vertex has no value, the node needs to contain some indication of the vertex's identity

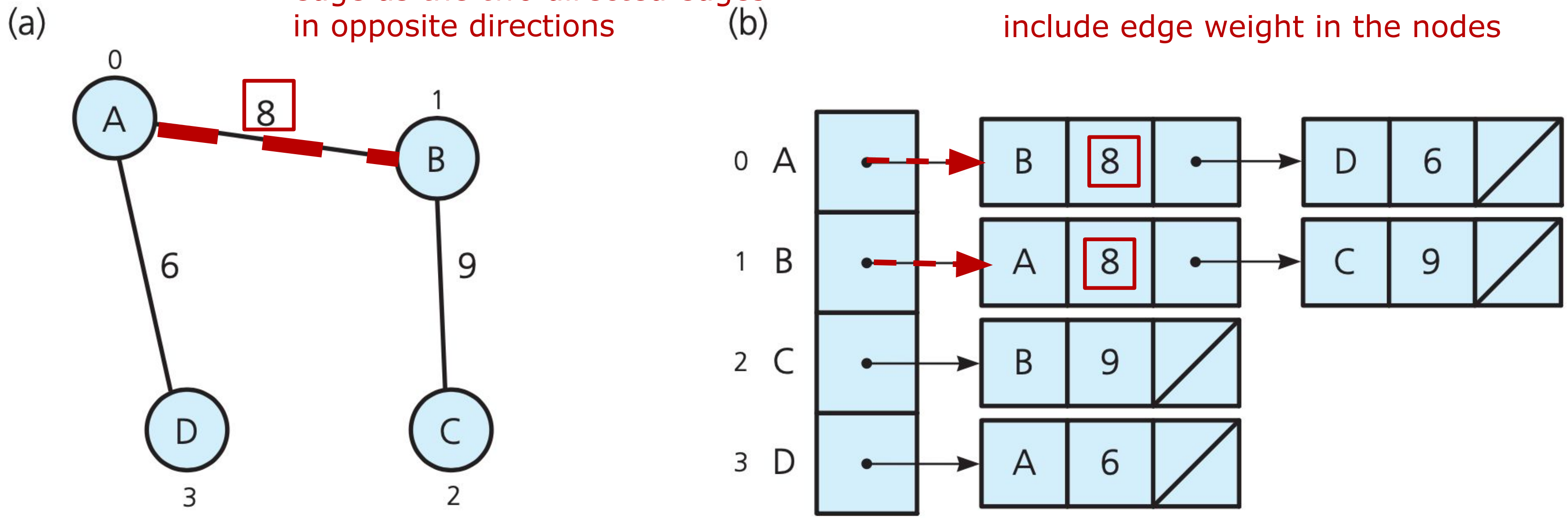


A directed + unweighted graph and its adjancet list

Implementing Graphs: adjacency list

for undirected graph, treat each edge as the two directed edges in opposite directions

include edge weight in the nodes



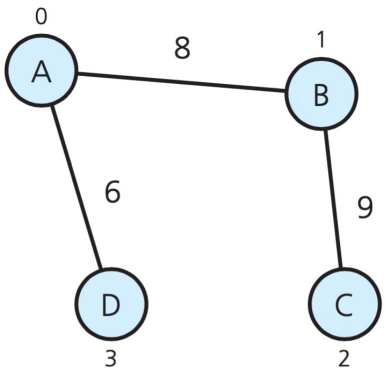
A undirected + weighted graph and its adjancet list

Implementing Graphs: comparison

- Which one is better? adjacency matrix or list?
- depend on applications and operations

Implementing Graphs: comparison

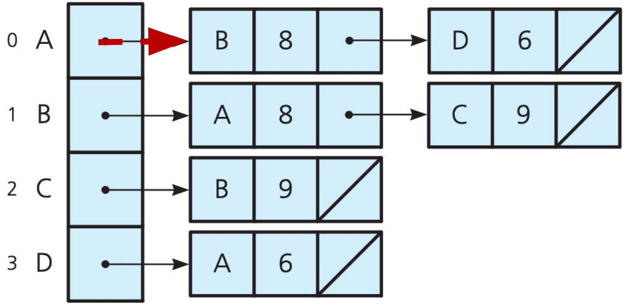
- For example, adjacency matrix is more efficient for determining whether there is an edge from vertex i to vertex j
  - only need to check the value of `matrix[i][j]`
  - if an adjaceny list, need to traverse ith linked chain



edge between A and C?

		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	$\boxed{\infty}$	9	∞	∞
3	D	6	∞	∞	∞

check value in [A][C] or [C][A]

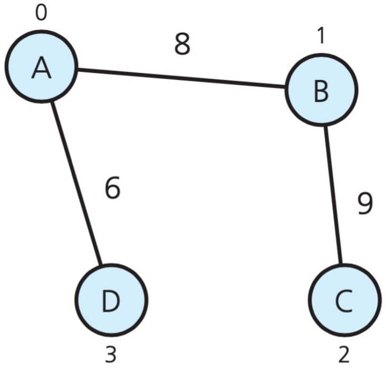


traverse 2th linked chain to check if there is node for C

Implementing Graphs: comparison

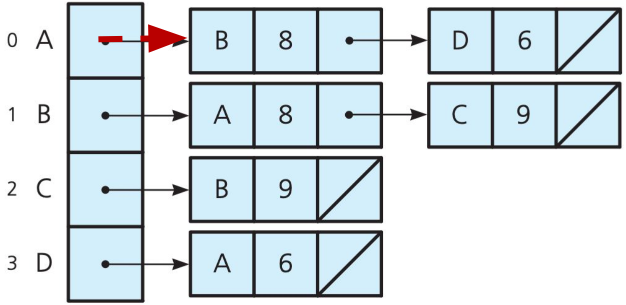
- On the other hand, adjacency list is more efficient for finding all vertices adjacent to a given vertex i
  - directly traverse the ith linked chain.

(a)



(b)

		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞



__ find adjacency of A

_____ traverse all columns in 0th row

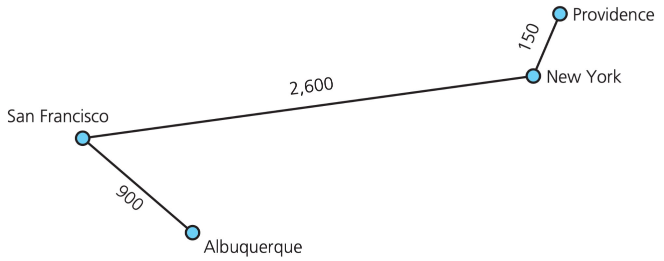
____ traverse 0th linked chain with 2 nodes

Implementing Graphs: comparison

- storage space requirement of a directed graph with n node
  - adjacency matrix always has *n*n* entries
  - adjacency list has number of nodes equals the number of edges in a directed graph
- adjacency list often requires less storage than an adjacency matrix.

Implementing Graphs: comparison

- choosing a graph implementation for a particular application
  - what operations you will perform most frequently
- for example, the flight map problem, most frequent operation was to find all cities (vertices) adjacent to given city
  - adjacency list would be more efficient



Graph Traversal

- traverse all vertices that it can reach
- mark each vertex during visit and only visit a vertex once
  - to prevent loop indefinitely due to a cycle in a graph

- 2 basic graph-traversal algorithms
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)

Depth-First Search (DFS)

- 深度優先
- From vertex v, DFS traversal goes as far (deep) as possible from the vertex v before backing up
- After visiting a vertex, a DFS vists an unvisited adjacent vertex if possible.

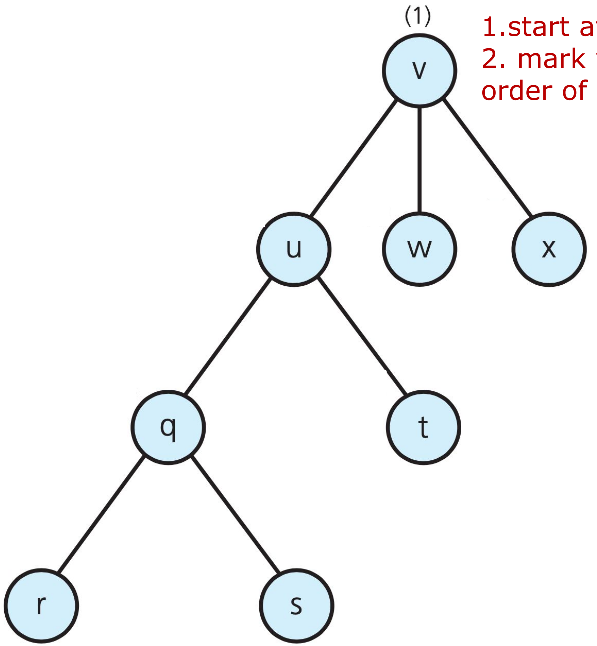
Depth-First Search (DFS)

DFS strategy has a recursive form

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Recursive version.
dfs(v: Vertex)
    Mark v as visited
    for (each unvisited vertex u adjacent to v)
        dfs(u)
```

- don't specify the order of which it should visit the vertices adjacent to v
- Could visit the adjacent vertices in sorted order alphabetically or numerically

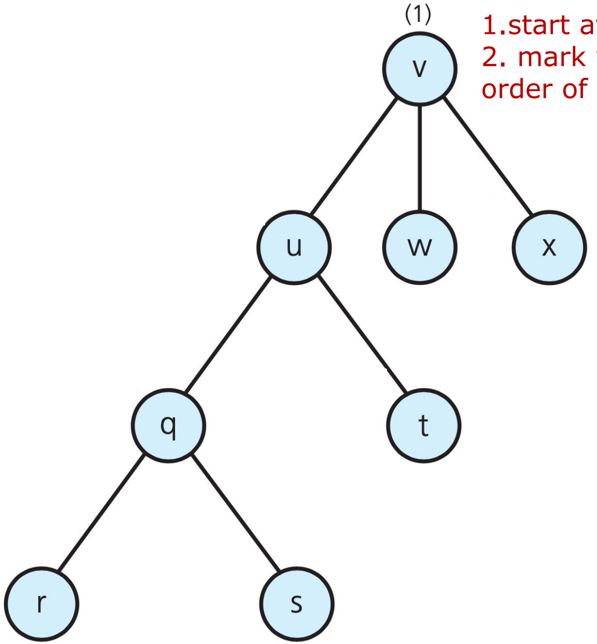
Depth-First Search (DFS): example



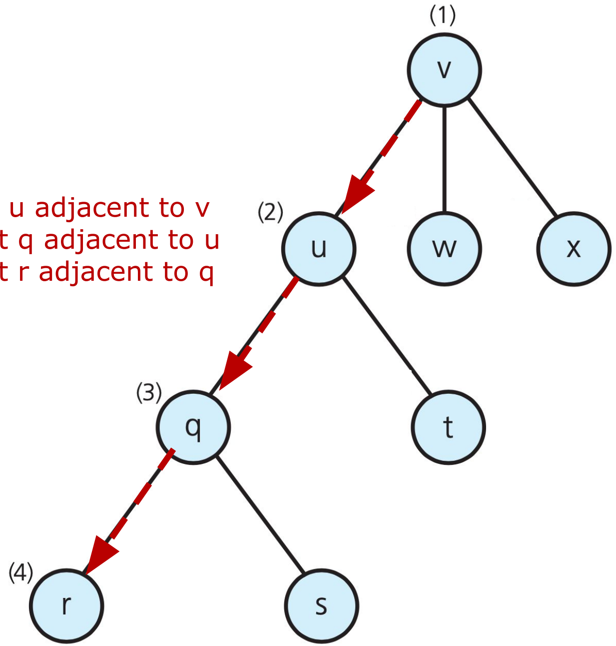
1.start at vertex v
2. mark v (1 means the order of visit)

undirected + unweighted **graph**

Depth-First Search (DFS): example



1.start at vertex v
2. mark v (1 means the order of visit)

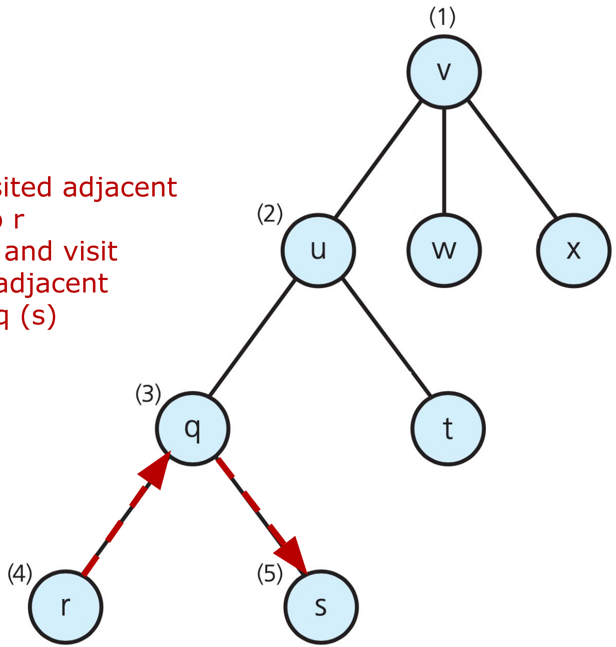


3.visit u adjacent to v
4. visit q adjacent to u
5. visit r adjacent to q

– undirected + unweighted **graph**

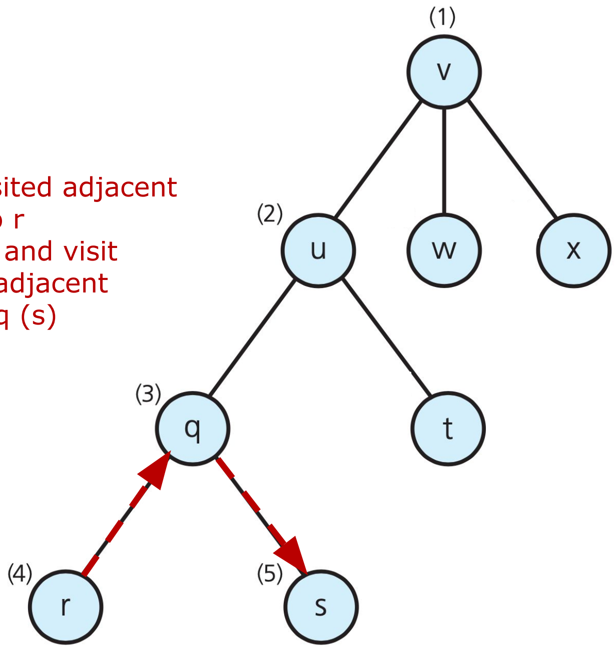
Depth-First Search (DFS): example

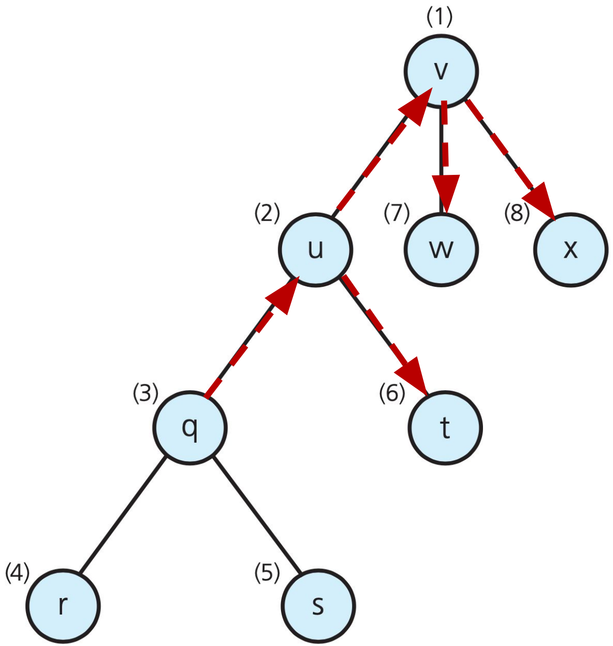
6.no unvisited adjacent vertices to r
7.back up and visit unvisited adjacent vertex to q (s)



Depth-First Search (DFS): example

6.no unvisited adjacent vertices to r
7.back up and visit unvisited adjacent vertex to q (s)



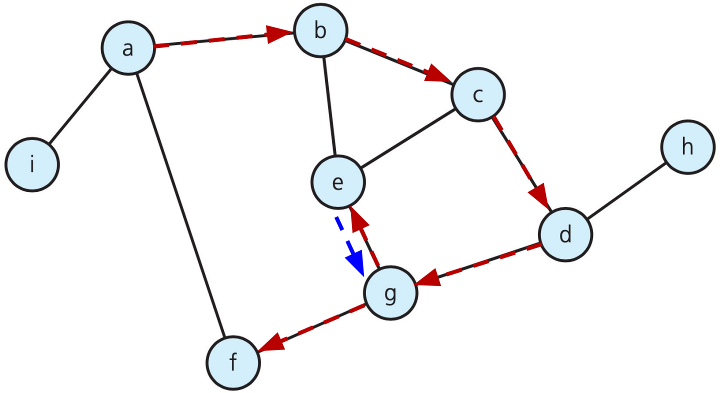


Depth-First Search (DFS)

DFS strategy has a iterative version using a stack

```
// Traverses a graph beginning at vertex v by using a
// depth-first search: Iterative version.
dfs(v: Vertex)
     s = a new empty stack
     s.push(v)
     Mark v as visited
     while (!s.isEmpty()){
          if (no unvisited vertices are adjacent to the vertex on the top of the stack)
               s.pop() // Backtrack
          else{
               Select an unvisited vertex u adjacent to vertex on the top of the stack
               s.push(u) // vertex u on the top of the stack
               Mark u as visited
          }
     }
```

Depth-First Search (DFS): example 2



order of visit: a, b, c, d, e, f, h, i

Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e *no unvisited vertex adjacent to e*
(backtrack)	a b c d g *pop e, so the top vertex in stack is g*
f	a b c d g f *visit f adjacent to g*
(backtrack)	a b c d g *pop f, so the top vertex is g*
(backtrack)	a b c d *pop g, so the top vertex is d*
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

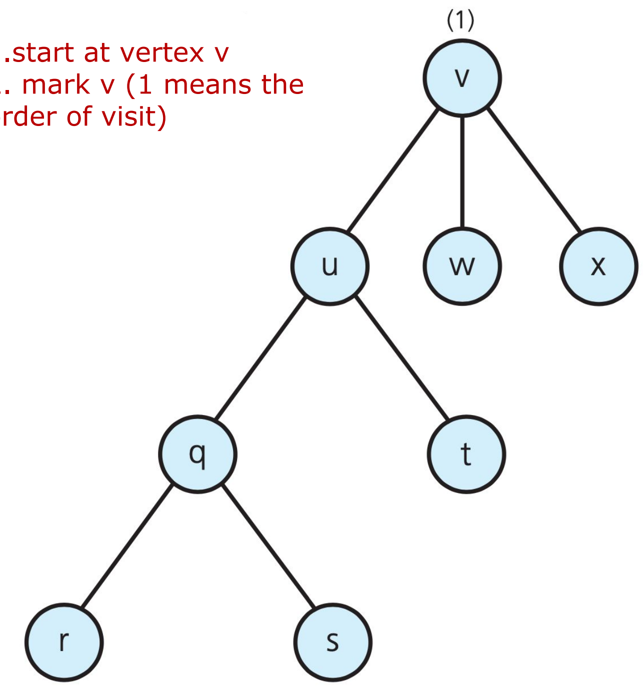
The results of a DFS traversal, beginning at vertex a

Breadth-First Search (BFS)

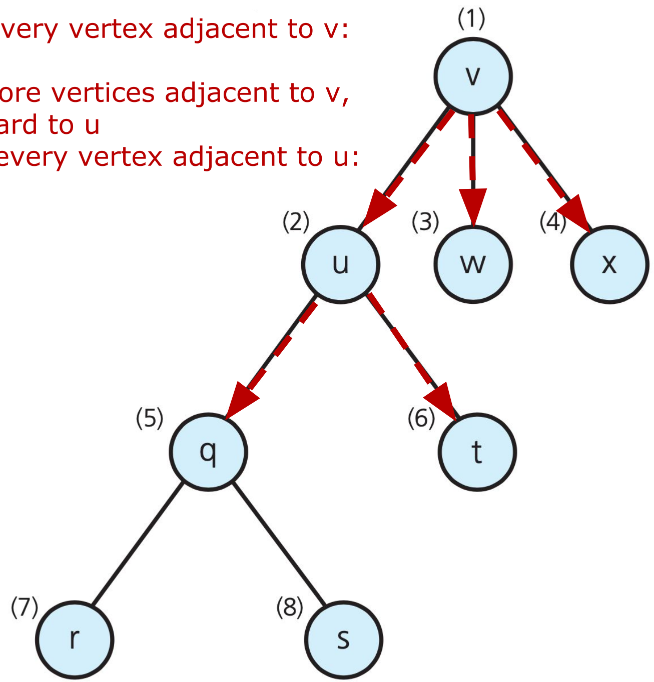
- 廣度優先
- BFS traversal visits all vertices adjacent to a vertex before going forward (deeper)

Breadth-First Search (BFS): example

1.start at vertex v
2. mark v (1 means the order of visit)



3.visit every vertex adjacent to v: u, w, x
4. no more vertices adjacent to v, go forward to u
5. visit every vertex adjacent to u: q, t



undirected + unweighted **graph**

Breadth-First Search (BFS)

- BFS strategy has a recursion version using a queue
- not as simple as the recursive version of DFS traversal

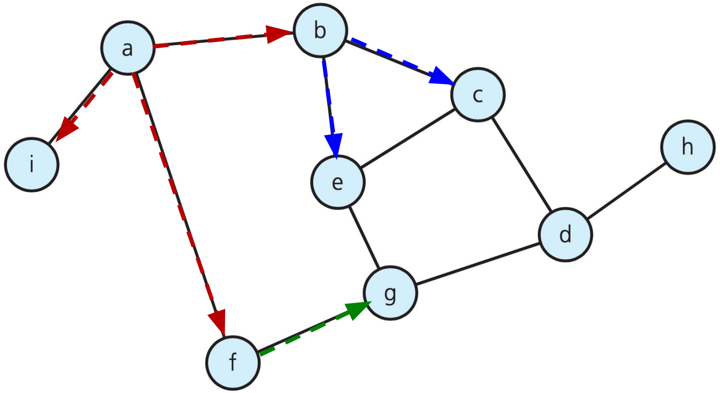
```
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Recursive version.
bfs(q: Queue)
      if (q.empty()) return

      v = q.dequeue(the front node)

      for(each unvisited vertex u  adjacent to v){ // do for every edge `v –> u
            Mark u as visited
            q.enqueue(u)
      }
      bfs(q)
```

Breadth-First Search (BFS)

BFS strategy has a iterative version using a queue

```
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Iterative version.
bfs(v: Vertex)
     q = a new empty queue
     q.enqueue(v) // Add v to queue and mark it
     Mark v as visited

     while (!q.isEmpty()){
          q.dequeue(the front node)

          for (each unvisited vertex u adjacent to v){ // do for every edge v -> u
               Mark u as visited
               q.enqueue(u)
          }
     }
```

Breadth-First Search (BFS): example 2



order of visit: a, b, f, i, c, e, g, d, h

Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f *put unvisited vertices adjacent to a in*
i	b f i *queue: b, f, i*
	f i *dequeue b*
c	f i c *put unvisited vertices adjacent to b in*
e	f i c e *queue: c, e*
	i c e *dequeue f*
g	i c e g *put unvisited vertices adjacent to f in*
	c e g *queue: g*
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

The results of a BFS traversal, beginning at vertex a

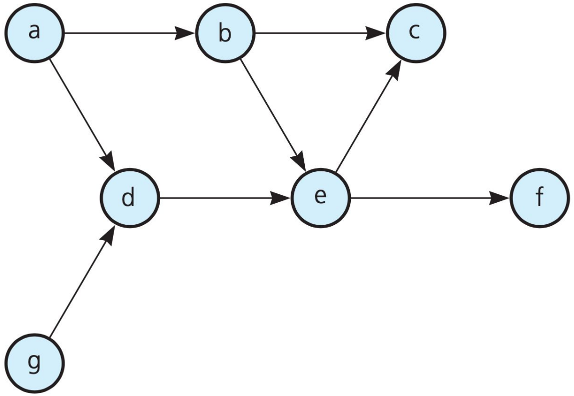
Application of Graphs

- Some common applications of graphs
  - Topological Sorting
  - Shortest Paths

Topological Sorting

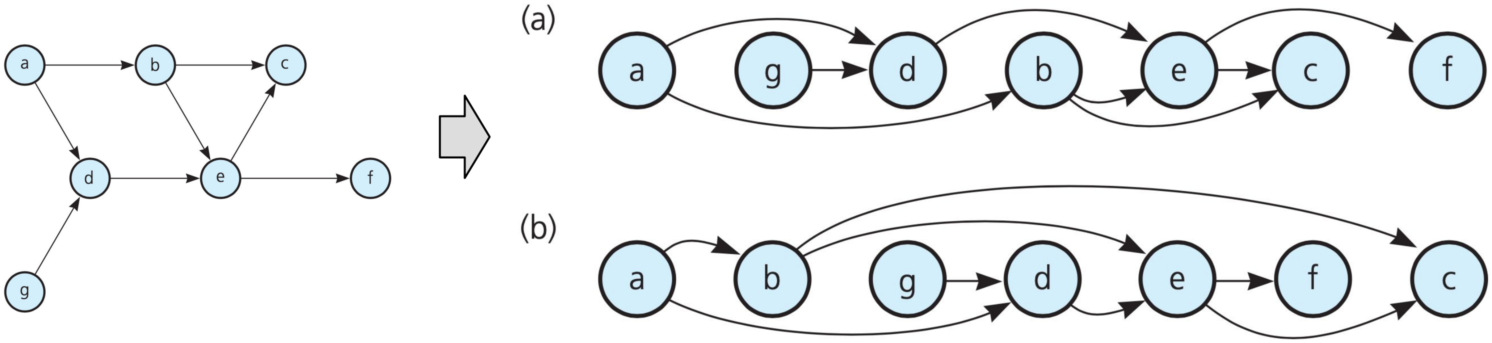
- A directed graph without cycle has a natural order
- If the vertices represent courses, the graph represent the prerequisite structure for the courses
- In what order should you take all seven courses so that you will satisfy all prerequisites? **Topological order**

a is prerequisite of b, b is prerequisite of c and e



Topological Sorting

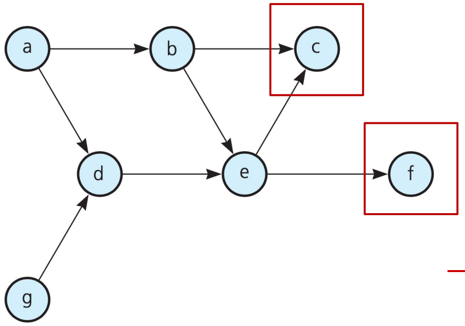
- The vertices in a given graph may have several topological orders
  - a, g, d, b, e, c, f
  - a, b, g, d, e, f, c



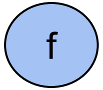
arrange the vertices of a directed graph in a topological order, edges will all point in one direction

Topological Sorting

- Arranging the vertices into a topological order is called topological sorting
- Several ways to do topolgical sorting
  - find a vertex that has no successor
  - remove the vertex and all edges lead to it from graph
  - add the vertex to the beginning of the list
  - repeatedly the above steps until the graph is empty
  - the list of vertices is in topological order

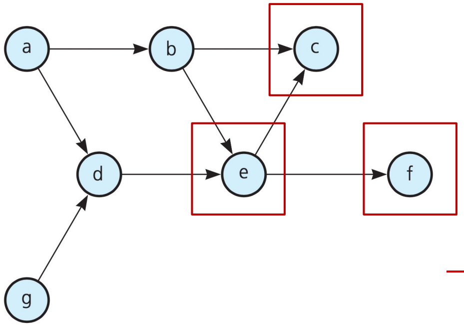


list:

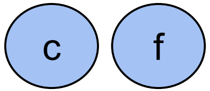


Topological Sorting

- Arranging the vertices into a topological order is called topological sorting
- Several ways to do topolgical sorting
  - find a vertex that has no successor
  - remove the vertex and all edges lead to it from graph
  - add the vertex to the beginning of the list
  - repeatedly the above steps until the graph is empty
  - the list of vertices is in topological order

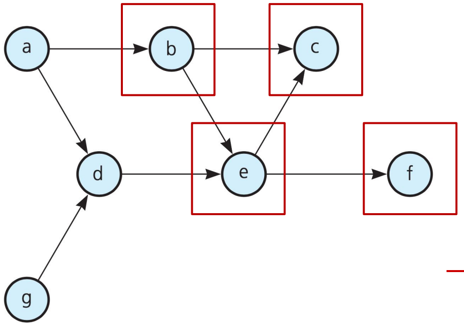


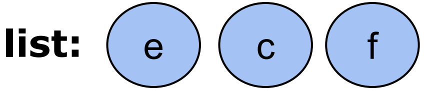
list:



Topological Sorting

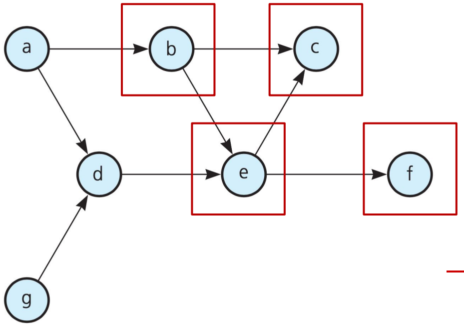
- Arranging the vertices into a topological order is called topological sorting
- Several ways to do topolgical sorting
  - find a vertex that has no successor
  - remove the vertex and all edges lead to it from graph
  - add the vertex to the beginning of the list
  - repeatedly the above steps until the graph is empty
  - the list of vertices is in topological order





Topological Sorting

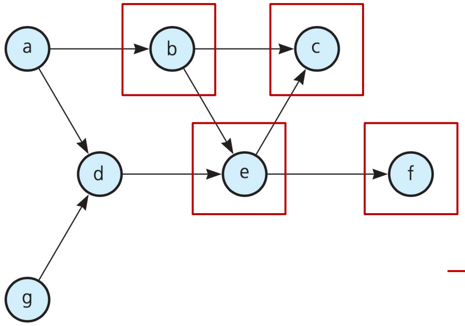
- Arranging the vertices into a topological order is called topological sorting
- Several ways to do topolgical sorting
  - find a vertex that has no successor
  - remove the vertex and all edges lead to it from graph
  - add the vertex to the beginning of the list
  - repeatedly the above steps until the graph is empty
  - the list of vertices is in topological order



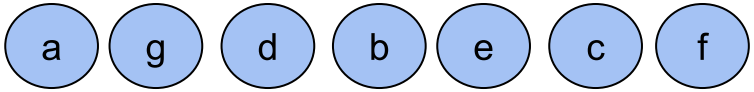
list: b e c f

Topological Sorting

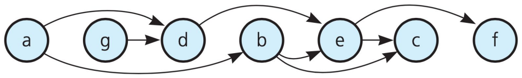
- Arranging the vertices into a topological order is called topological sorting
- Several ways to do topolgical sorting
  - find a vertex that has no successor
  - remove the vertex and all edges lead to it from graph
  - add the vertex to the beginning of the list
  - repeatedly the above steps until the graph is empty
  - the list of vertices is in topological order



list:



(a)



Topological Sorting: pseudocode

```
// Arranges the vertices in graph the Graph into a
// topological order and places them in list aList

topSort1(theGraph: Graph, aList: List)
     n = number of vertices  in theGraph
     for (step = 1 through n){ //all vertices
          Select a vertex v that has no successors
          aList.insert(1, v) //insert at the beginning of list
          Remove from theGraph vertex v and its edges
     }
```

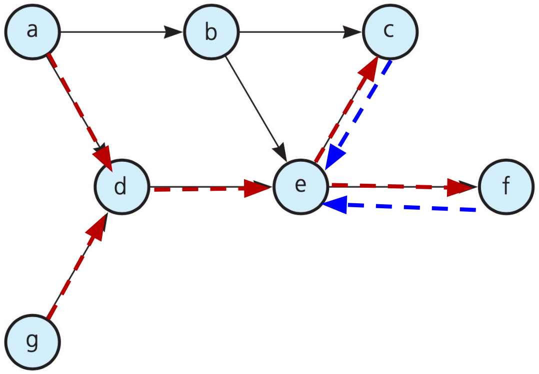
Topological Sorting: DFS topological sorting

- depth-first search (DFS) topological sorting
  - push all vertices that have no predecessor onto a stack
  - Each time you pop a vertex from the stack
  - add it to the beginning of a list of vertices
- modified from DFS traversal

Topological Sorting: DFS pseudocode

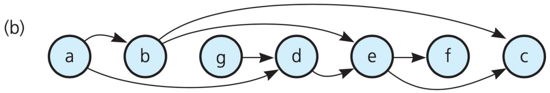
```
topSort2(theGraph: Graph, aList: List)
      s = a new empty stack
      for (all vertices v in the graph){
            if (v has no predecessors){
                  s.push(v)
                  Mark v as visited
            }
      }

      while(!s.isEmpty()){
            if (all vertices adjacent to the vertex on the top of the stack  have been visited)
{
                  s.pop(v) //backtrace
                  aList.insert(1, v)
            }
            else{
                  Select an unvisited vertex  u adjacent to the vertex on the top  of the stack
                  s.push(u)
                  Mark u as visited
            }
      }
```

Topological Sorting: DFS



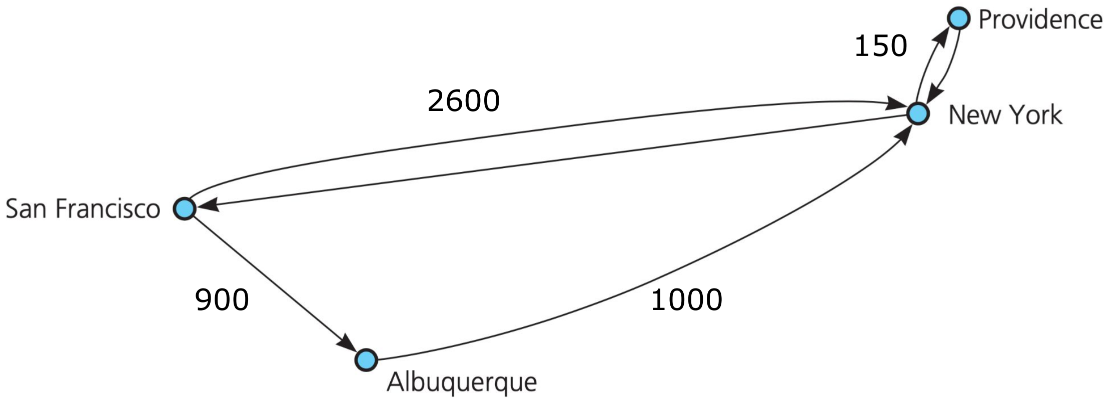
Action	Stack s (bottom to top)	List **aList** (beginning to end)
Push a	a a, g has no predecessors, g at the top of stack	
Push g	a g	
Push d	a g d Select an unvisited vertex d adjacent to g	
Push e	a g d e Select an unvisited vertex e adjacent to d	
Push c	a g d e c Select an unvisited vertex c adjacent to e	
Pop c, add c to **aList**	a g d e	c c has no adjacent & unvisited vertex
Push f	a g d e f	c
Pop f, add f to **aList**	a g d e	f c add f at the biginning of list
Pop e, add e to **aList**	a g d	e f c
Pop d, add d to **aList**	a g	d e f c
Pop g, add g to **aList**	a	g d e f c
Push b	a b	g d e f c
Pop b, add b to **aList**	a	b g d e f c
Pop a, add a to **aList**	(empty)	a b g d e f c

resulting topological order:



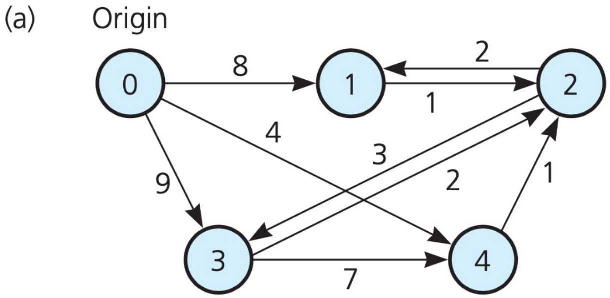
Shortest Path

- find the shortest path between 2 cities (vertices)
  - the path has the smallest edge-weight sum
- the weight can also be the dollars or duration (nonnegative)
- the sum of weights of edges of a path called path's length/weight/cost



Shortest Paths: example

- shortest path between vertex 0 to 1
  - not edge between 0 and 1 (cost 8)
  - path: 0 to 4 to 2 to 1 (cost 7)
- origin labeled 0, other vertices labeled from 1 to n-1



directed + weighted graph

(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

`matrix` of the left graph

Shortest Paths:Dijkstra's shortest-path algorithm

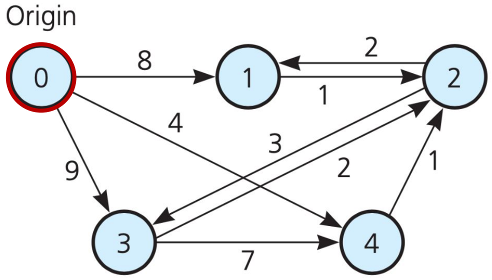
- Dijkstra's shortest-path algorithm
  - determines the shortest paths between a given origin and all other vertices
- `vertexSet`
  - a set of selected vertices
- `weight`
  - array where `weight[v]` is the weight of shortest path from vertex 0 (origin) to vertex v that passes through vertices in `vertexSet`

Shortest Paths:Dijkstra’s shortest-path algorithm

- Initially,
  - `vertexSet` contains only vertex 0 (origin), and `weight` contains the weights of the single-edge paths from vertex 0 to all other vertices v
  - `weights[v] = matrix[0][v]`

initial weight is the first row of adjacency matrix

(a)



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Shortest Paths: Dijkstra’s shortest-path algorithm

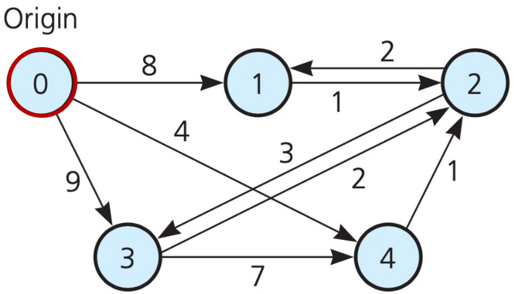
- After this initialization step
  - find a vertex v
    - not in `vertexSet`
    - the samllest value in `weight[v]`
  - You add v to vertexSet
- For all unselected vertices u not in `vertexSet`
  - check value of `weight[u]` (cost of path from 0 to u) to ensure that they are minimums
  - can the value of `weight[u]` be reduced by passing newly selected vertex v?

Shortest Paths:Dijkstra's shortest-path algorithm

- can the value of `weight[u]` be reduced by passing newly selected vertex v?
- how to ensure value of `weight[u]` is minimums?
  - break the path from 0 to u into two pieces and find their weights as follows
    - `weight[v]` = weight of the shortest path from 0 to v
    - `matrix[v][u]` = weight of the edge from v to u
- `weight[u] = min{weight[u], weight[v] + matrix[v][u]}`

Shortest Paths: Dijkstra’s shortest-path algorithm example

(a)



(b)

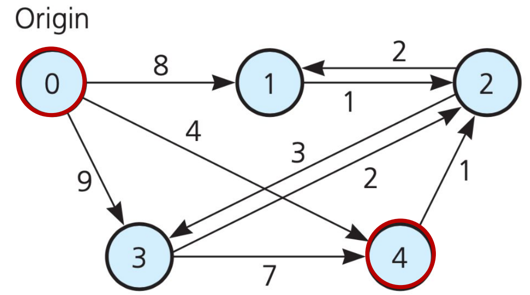
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4

initially, vertexSet contains 0,
weight = adjacency matrix[0][]

Shortest Paths:Dijkstra’s shortest-path algorithm example

(a)



(b)

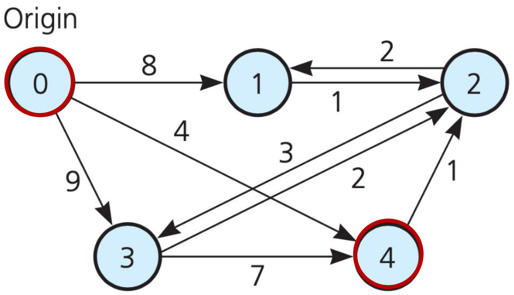
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4					

weight[4] is the smallest value in `weight` and vertex 4 didn’t in `vertexSet`. So, v = 4 & add 4 to `vertexSet`

Shortest Paths: Dijkstra’s shortest-path algorithm example

(a)



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

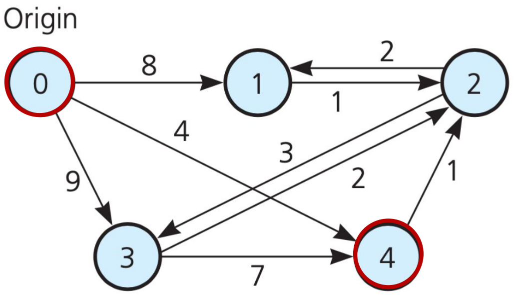
			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0				4

For vertices not in `vertexSet` (for u = 1, 2, 3), check if `weight[u]` (path from 0 to u) is minimums

`weight[u] = min{weight[u], weight[v] + matrix[v][u]}`

Shortest Paths:Dijkstra's shortest-path algorithm example

(a)



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	$\boxed{\infty}$	1	∞	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8			4

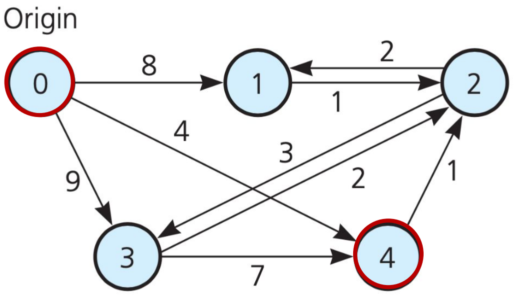
For vertex 1 (u=1),

`weight[1] = 8`

`weight[4] + matrix[4][1] = 4 + ∞`

Shortest Paths: Dijkstra's shortest-path algorithm example

(a)



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5		4

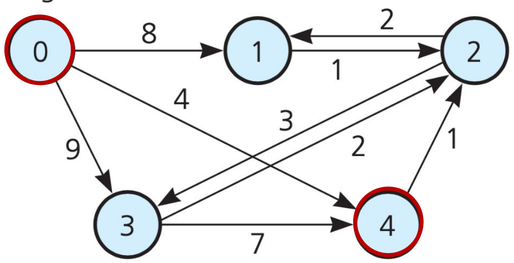
For vertex 2 (u=2),

weight[2] = ∞

weight[4] + matrix[4][2] = 4 + 1

Shortest Paths:Dijkstra’s shortest-path algorithm example

(a)



(b)

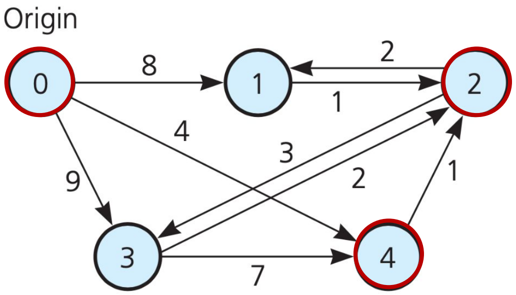
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	$\boxed{\infty}$	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4

For vertex 3 (u=3),
`weight[3] = 9`
`weight[4] + matrix[4][3] = 4 + ∞`

Shortest Paths:Dijkstra’s shortest-path algorithm example

(a)



(b)

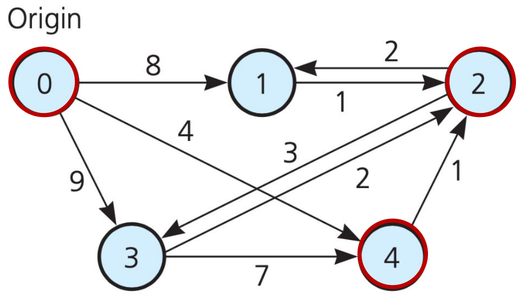
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0		5		4

`weight[2]` is the smallest value in `weight` and vertex 2 didn’t in `vertexSet`.
So, v = 2 & add 2 to `vertexSet`

Shortest Paths: Dijkstra's shortest-path algorithm example

(a)



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

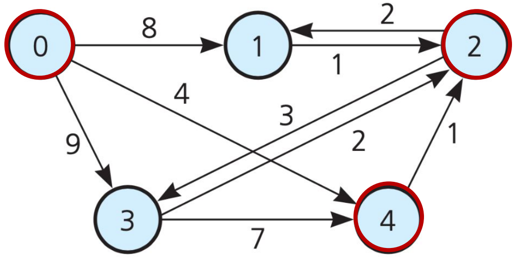
			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0		5		4

For vertices not in `vertexSet` (for u = 1, 3), check if `weight[u]` (path from 0 to u) is minimums

`weight[u] = min{weight[u], weight[v] + matrix[v][u]}`

Shortest Paths:Dijkstra’s shortest-path algorithm example

(a)



(b)

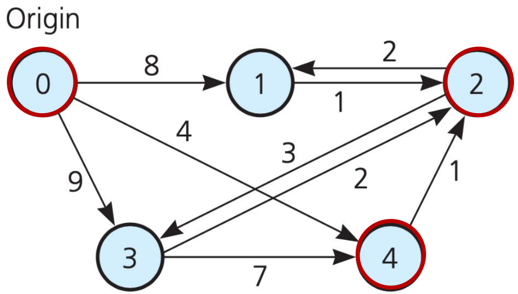
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	$\boxed{2}$	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5		4

For vertex 1 (u=1),
weight[1] = 8
weight[2] + matrix[2][1] = 5 + 2

Shortest Paths:Dijkstra's shortest-path algorithm example

(a)



(b)

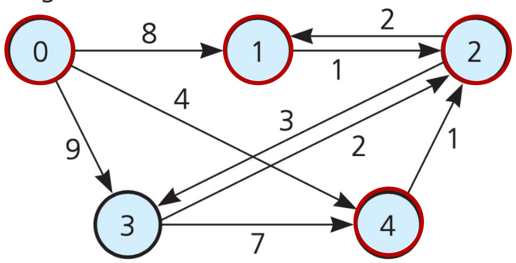
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	$\boxed{3}$	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4

For vertex 3 (u=3),
weight[3] = 9
weight[2] + matrix[2][3] = 5 + 3

Shortest Paths: Dijkstra's shortest-path algorithm example

(a)



(b)

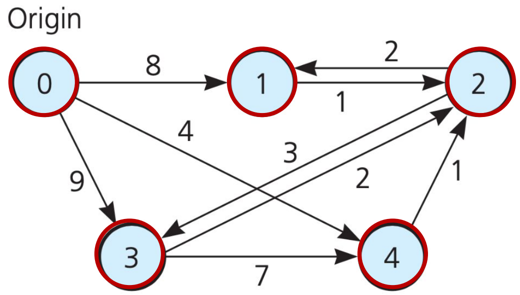
	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

			weight				
Step	v	vertexSet	[0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4

`weight[1]` is the smallest value in `weight` and vertex 1 didn't in `vertexSet`.
So, v = 1 & add 1 to `vertexSet`

Shortest Paths: Dijkstra's shortest-path algorithm example

(a)



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

Step	v	vertexSet	weight [0]	[1]	[2]	[3]	[4]
1	–	0	0	8	∞	9	4
2	4	0, 4	0	8	5	9	4
3	2	0, 4, 2	0	7	5	8	4
4	1	0, 4, 2, 1	0	7	5	8	4
5	3	0, 4, 2, 1, 3	0	7	5	8	4

repeat the same process, update the `weight[u]` (u = 3)

the final values in `weight` are the weight of shortest path from vertex 0 to other vertices

Shortest Paths:Dijkstra’s shortest-path algorithm pseudocode

```
// Finds the minimum-cost paths between an origin vertex
// (vertex 0) and all other vertices in a weighted directed
shortestPath(theGraph: Graph, weight: WeightArray)

      // Step 1: initialization
      Create a set vertexSet that contains only vertex 0
      n = number of vertices in theGraph
      for (v = 0 through n - 1)
            weight[v] = matrix[0][v]

      // Steps 2:
      for (step = 2 through n){ //for the rest vertex 1 to n-1
            Find the smallest weight[v] such that v is not in vertexSet
            Add v to vertexSet

            for (all vertices u not in vertexSet){// update weight[u] for all u not in vertexSet
                  if (weight[u] > weight[v] + matrix[v][u])
                        weight[u] = weight[v] + matrix[v][u]
            }
      }
```

Summary of Graph

- Graph implementation
  - adjacency matrix and the adjacency list
  - advantages and disadvantages
- Graph Traversal
  - Depth-first search (DFS)
  - Breadth-first search (BFS)
  - the order of visit
- Applications
  - Topological sorting
  - Dijkstra’s shortest-path algorithm