

# Chapter 9 - Graph

- A **Graph**  $G$  consists of a set  $V$ , whose members are called the **vertices** of  $G$ , together with a set  $E$  of pairs of distinct vertices from  $V$ .
- The pairs in  $E$  are called the **edges** of  $G$ .
- If the pairs are unordered,  $G$  is called an **undirected graph** or a **graph**. Otherwise,  $G$  is called a **directed graph** or a **digraph**.
- Two vertices in an undirected graph are called **adjacent** if there is an edge from the first to the second.

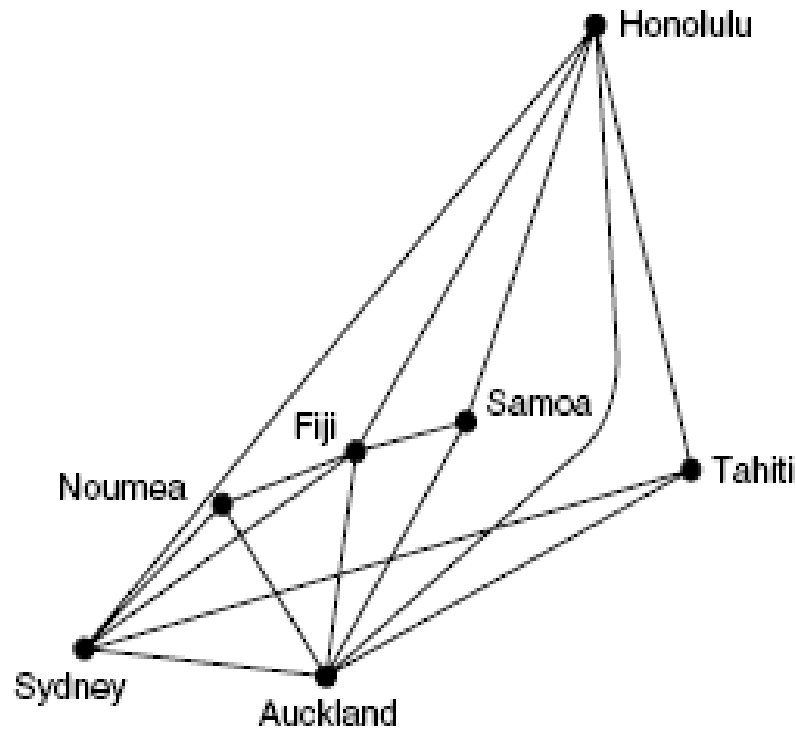
# Chapter 9 - Graph

- A **path** is a sequence of distinct vertices, each adjacent to the next.
- A **cycle** is a path containing at least three vertices such that the last vertex on the path is adjacent to the first.
- A graph is called **connected** if there is a path from any vertex to any other vertex.
- A **free tree** is defined as a connected undirected graph with no cycles.

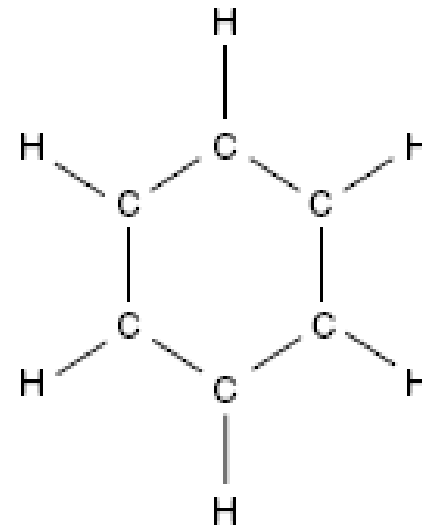
# Chapter 9 - Graph

- In a *directed graph* a path or a cycle means **always moving in the direction** indicated by the arrows.
- A directed graph is called **strongly connected** if there is a directed path from any vertex to any other vertex.
- If we suppress the direction of the edges and the resulting undirected graph is connected, we call the directed graph **weakly connected**

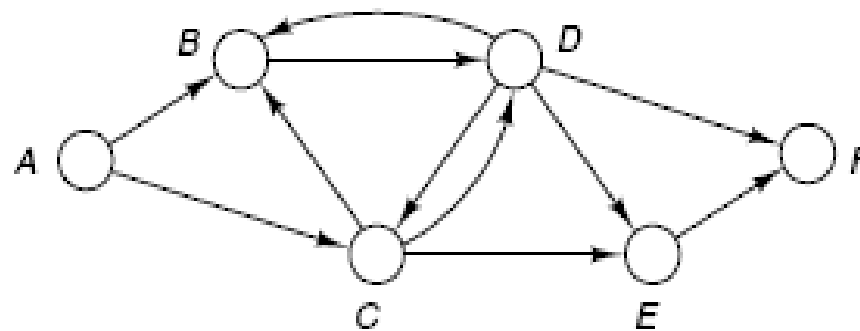
# Examples of Graph



Selected South Pacific air routes

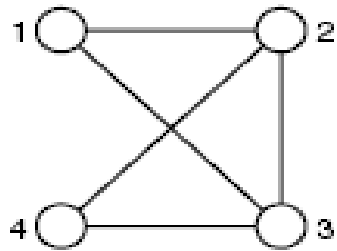


Benzene molecule



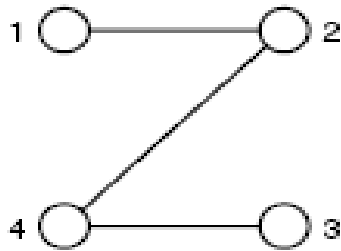
Message transmission in a network

# Examples of Graph



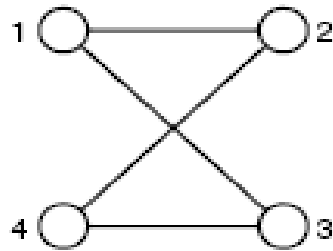
Connected

(a)



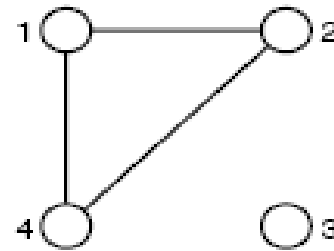
Path

(b)



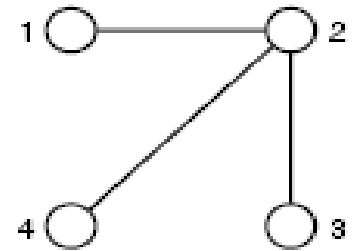
Cycle

(c)



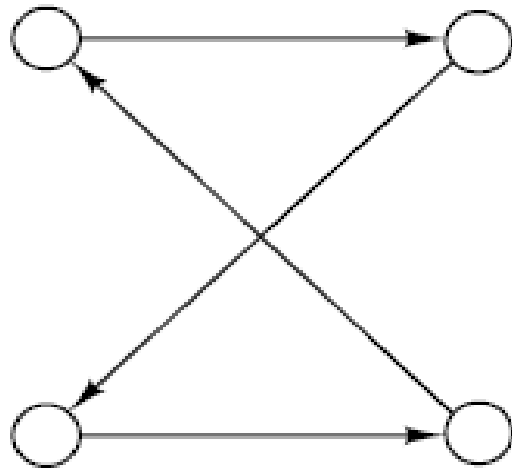
Disconnected

(d)



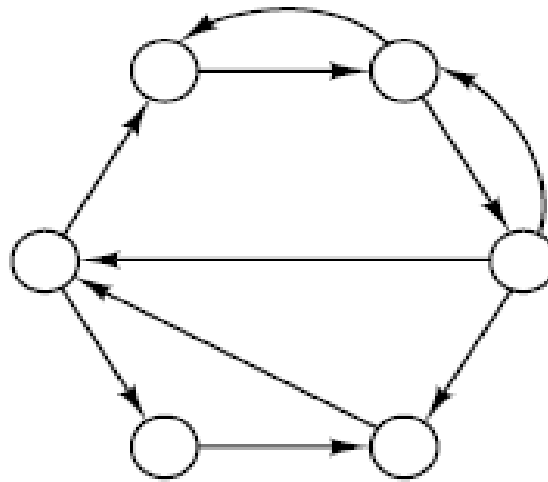
Tree

(e)



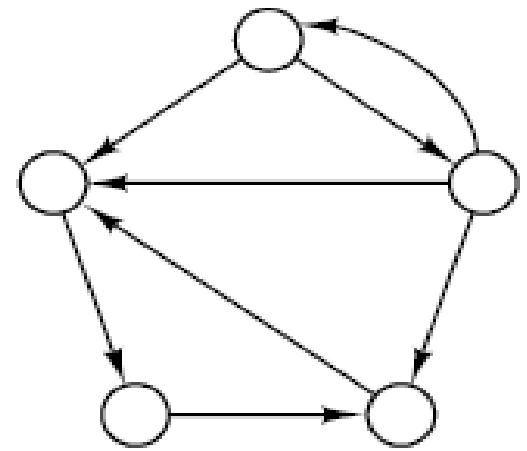
Directed cycle

(a)



Strongly connected

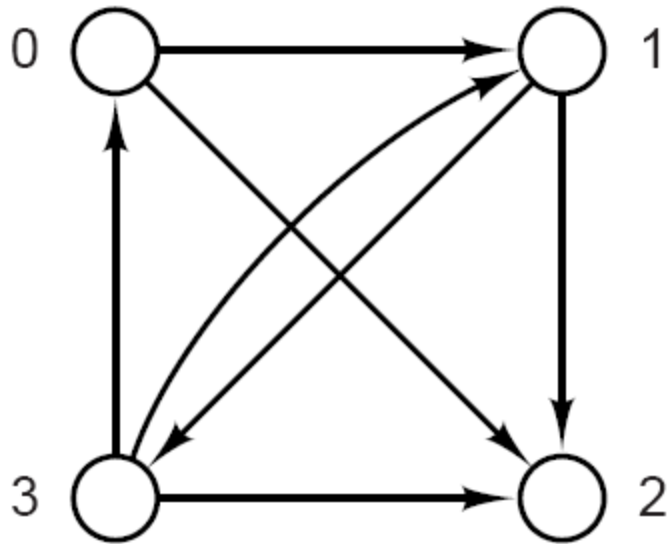
(b)



Weakly connected

(c)

# Digraph as an adjacency table



Directed graph

vertex	Set
0	{ 1, 2 }
1	{ 2, 3 }
2	$\emptyset$
3	{ 0, 1, 2 }

Adjacency set

	0	1	2	3
0	F	T	T	F
1	F	F	T	T
2	F	F	F	F
3	T	T	T	F

Adjacency table

## Digraph

count <integer>

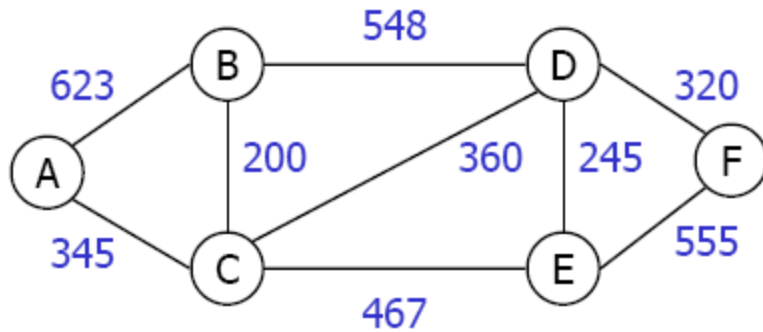
// Number of vertices

edge <array of <array of <boolean> > >

// Adjacency table

End Digraph

# Weighted-graph as an adjacency table



Weighted-graph

A
B
C
D
E
F

vertex vector

	A	B	C	D	E	F
A	0	623	345	0	0	0
B	623	0	200	548	0	0
C	345	200	0	360	467	0
D	0	548	360	0	245	320
E	0	0	467	245	0	555
F	0	0	0	320	555	0

adjacency table

## WeightedGraph

count <integer>

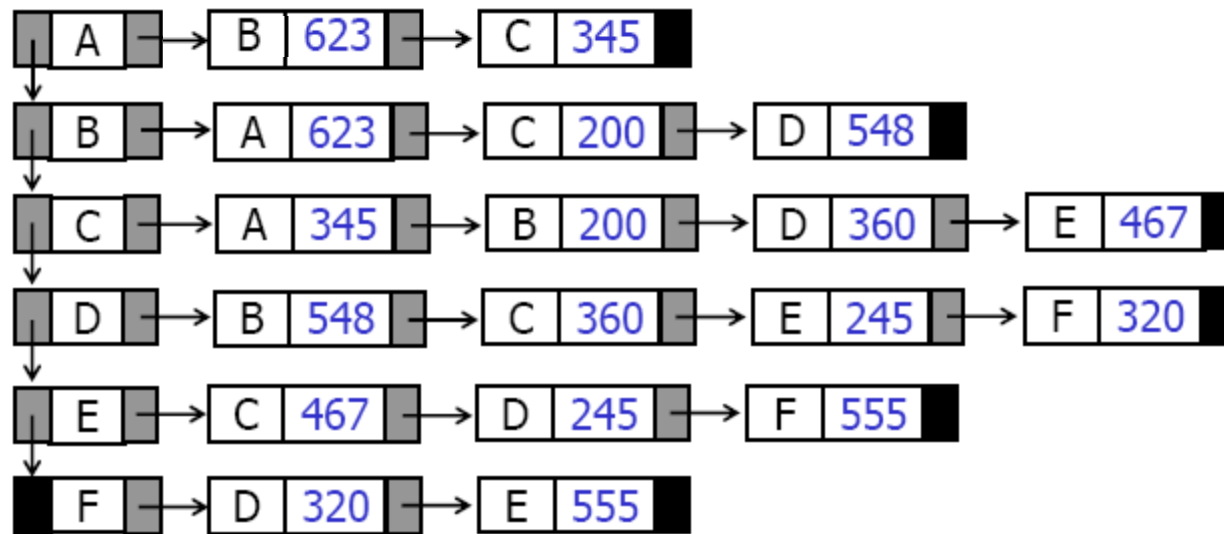
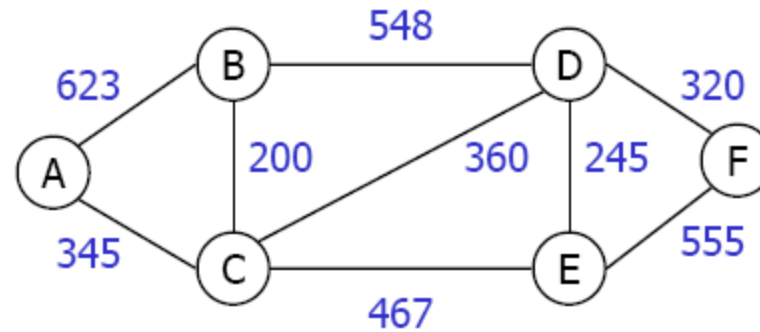
edge <array of<array of<WeightType>>>

End WeightedGraph

// Number of vertices

// Adjacency table

# Weighted-graph as an adjacency list

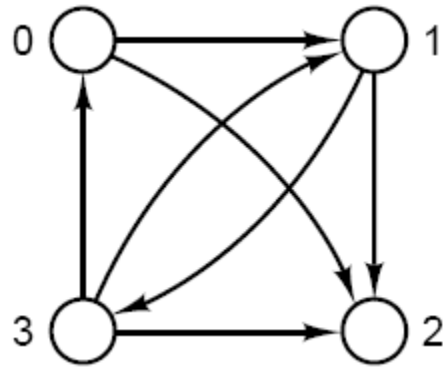


vertex  
vector

adjacency  
list



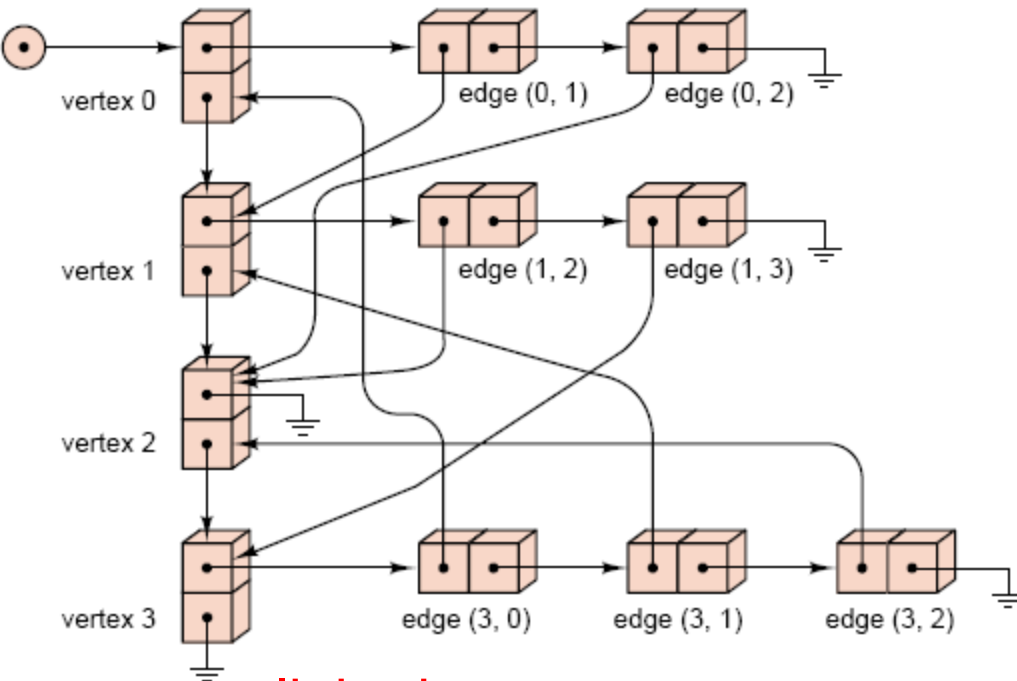
# Digraph as an adjacency list



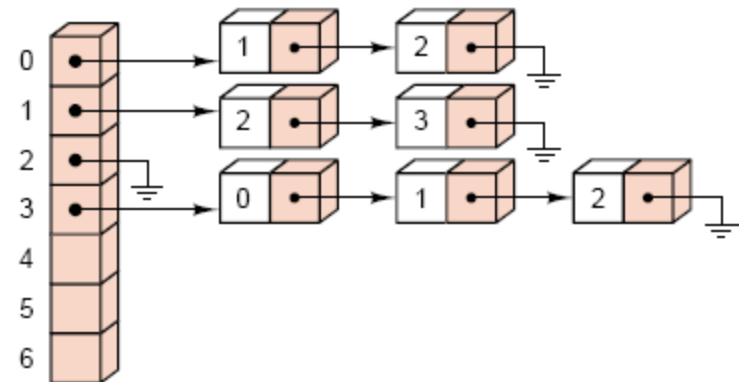
Directed graph

0	1	2	-	-	-	-
1	2	3	-	-	-	-
2	-	-	-	-	-	-
3	0	1	2	-	-	-
4	-	-	-	-	-	-
5	-	-	-	-	-	-
6	-	-	-	-	-	-

contiguous structure

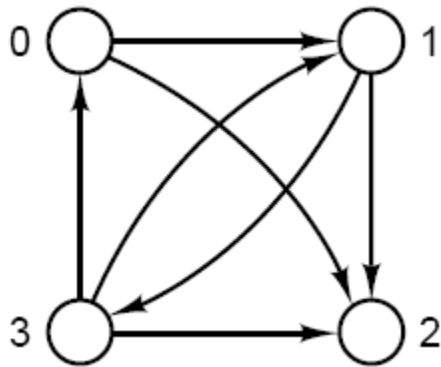


linked structure



mixed structure

# Digraph as an adjacency list (not using List ADT)



Directed graph

## VertexNode

first\_edge <pointer to EdgeNode>  
next\_vertex <pointer to VertexNode>

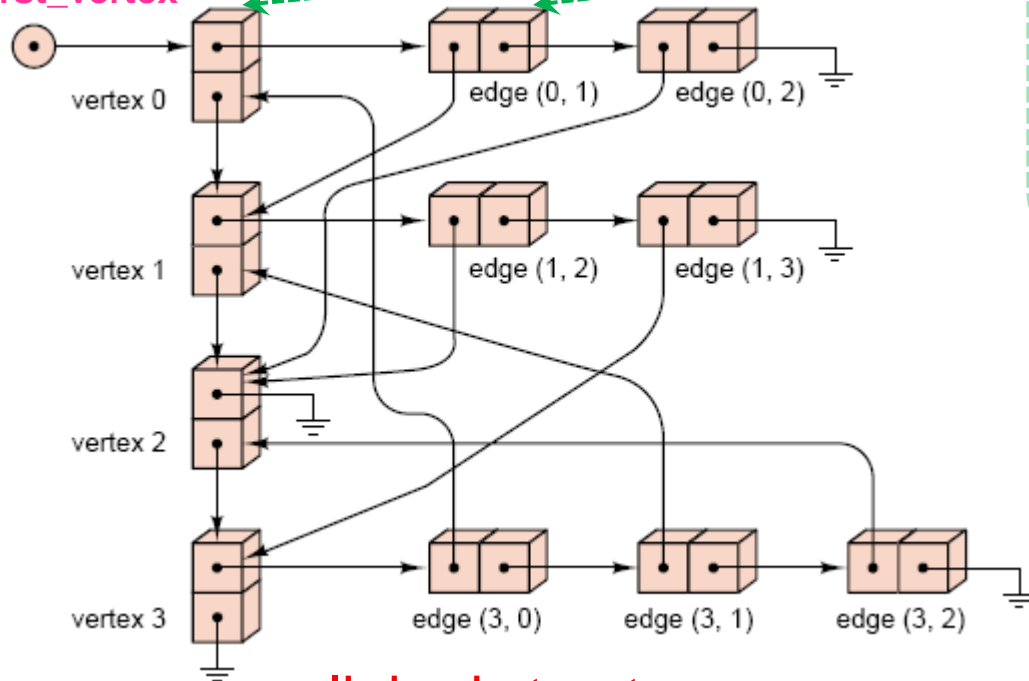
End VertexNode

## EdgeNode

vertex\_to <pointer to VertexNode>  
next\_edge <pointer to EdgeNode>

End EdgeNode

first\_vertex



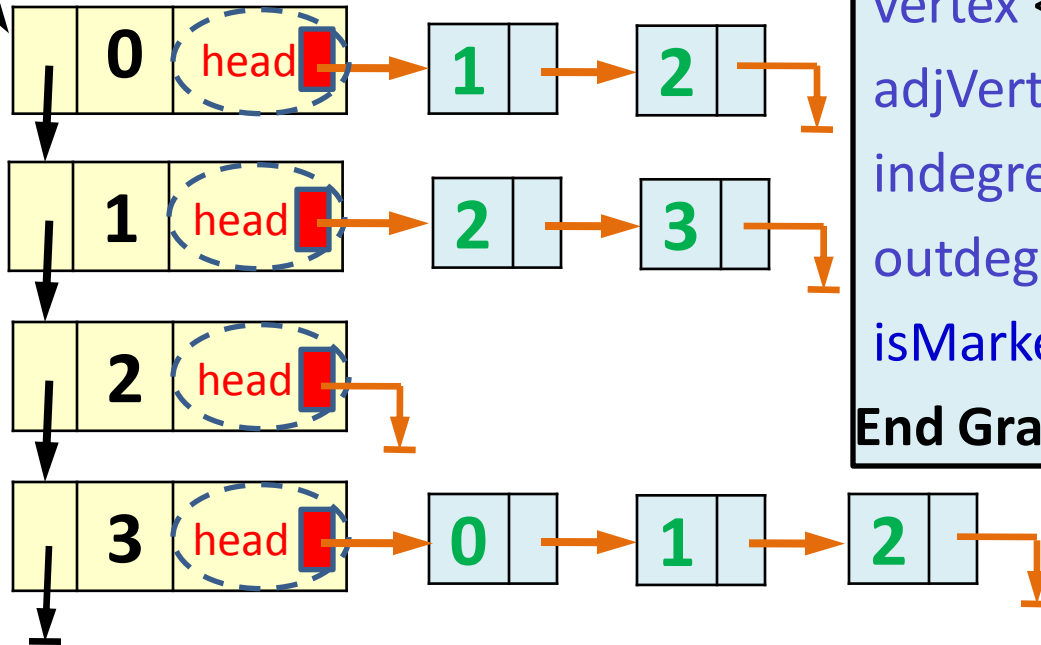
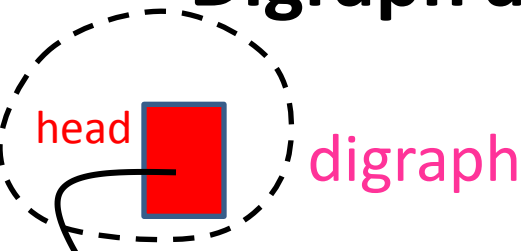
linked structure

## DiGraph

first\_vertex <pointer to VertexNode>

End DiGraph

# Digraph as an adjacency list (using List ADT)



## GraphNode

```
vertex <VertexType> // (key field)
adjVertex<LinkedList of< VertexType >>
indegree <int>
outdegree <int>
isMarked <boolean>
```

*are hidden from the image below*

End GraphNode

**ADT List is linked list:**

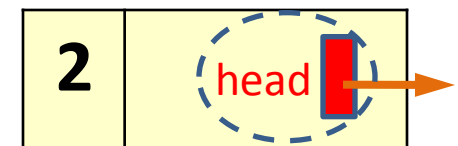
**DiGraph**

**digraph** <LinkedList<of<GraphNode>>

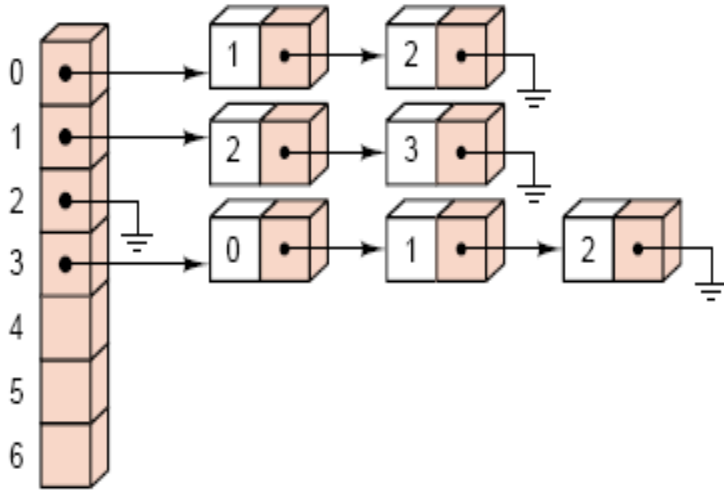
End DiGraph

## GraphNode

vertex adjVertex



# Digraph as an adjacency list (using List ADT)



mixed list

## GraphNode

`vertex` <VertexType> // (*key field*)

`adjVertex` <LinkedList of< VertexType >>

`indegree` <int>

`outdegree` <int>

`isMarked` <boolean>

End GraphNode

**ADT List is contiguous list:**

## DiGraph

`digraph` <ContiguousList<of<GraphNode>>>

End DiGraph

# Digraph as an adjacency list (using List ADT)

0	1	2	-	-	-	-	-
1	2	3	-	-	-	-	-
2	-	-	-	-	-	-	-
3	0	1	2	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	-	-
6	-	-	-	-	-	-	-

contiguous list

## GraphNode

`vertex` <VertexType> // (*key field*)

`adjVertex`<**ContiguousList** of< VertexType >>

`indegree` <int>

`outdegree` <int>

`isMarked` <boolean>

End GraphNode

***ADT List is contiguous list:***

## DiGraph

`digraph` <**ContiguousList**<of<GraphNode>>

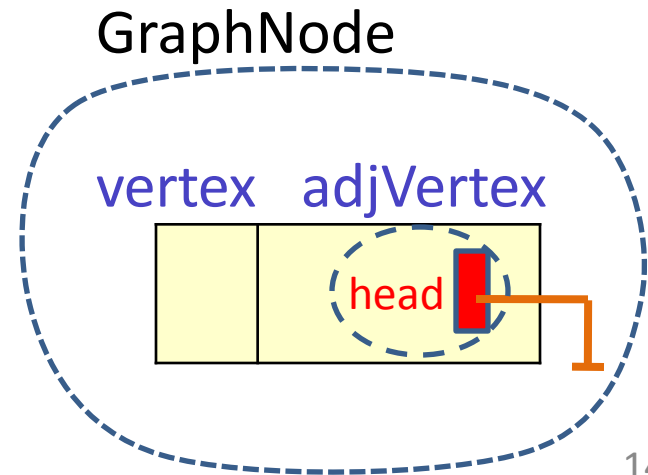
End DiGraph

# GraphNode

<void> GraphNode() *// constructor of GraphNode*

1. indegree = 0
2. outdegree = 0
3. adjVertex.clear() *// By default, constructor of adjVertex made it empty.*

End GraphNode



# Operations for Digraph

- Insert Vertex
- Delete Vertex
- Insert edge
- Delete edge
- Traverse

# Digraph

## Digraph

*private:*

**digraph** <List of <GraphNode> > *// using of List ADT.*

<void> **Remove\_EdgesToVertex**(val **VertexTo** <VertexType>)

*public:*

<ErrorCode> **InsertVertex** (val **newVertex** <VertexType>)

<ErrorCode> **DeleteVertex** (val **Vertex** <VertexType>)

<ErrorCode> **InsertEdge** (val **VertexFrom** <VertexType>,  
val **VertexTo** <VertexType>)

<ErrorCode> **DeleteEdge** (val **VertexFrom** <VertexType>,  
val **VertexTo** <VertexType>)

*// Other methods for Graph Traversal.*

**End Digraph**



# Methods of List ADT

*Methods of Digraph will use these **methods of List ADT**:*

```
<ErrorCode> Insert (val DataIn <DataType>)          // (success, overflow)
<ErrorCode> Search (ref DataOut <DataType>)         // (found, notFound)
<ErrorCode> Remove (ref DataOut <DataType>)         // (success, notFound)
<ErrorCode> Retrieve (ref DataOut <DataType>)        // (success, notFound)
<ErrorCode> Retrieve (ref DataOut <DataType>, position <int>)
                                                    // (success, range_error)
<ErrorCode> Replace (val DataIn <DataType>, position <int>)
                                                    // (success, range_error)
<ErrorCode> Replace (val DataIn <DataType>, val DataOut <DataType>)
                                                    // (success, notFound)

<boolean> isFull()
<boolean> isEmpty()
<integer> Size()
```

# Insert New Vertex into Digraph

<ErrorCode> **InsertVertex** (val *newVertex* <VertexType>)

Inserts new vertex into digraph.

**Pre**        *newVertex* is a vertex needs to be inserted.

**Post**      if the vertex is not in digraph, it has been inserted and no edge is involved with this vertex.

**Return**   *success*, *overflow*, or *duplicate\_error*

# Insert New Vertex into Digraph

<ErrorCode> **InsertVertex** (val newVertex <VertexType>)

1. DataOut.vertex = newVertex

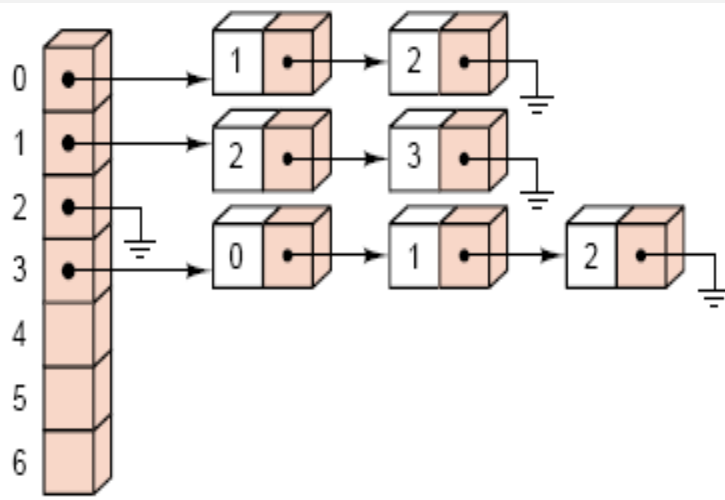
2. if (digraph.Search(DataOut) = *success*)

1. return *duplicate\_error*

3. else

1. return digraph.Insert(DataOut) // *success* or *overflow*

End InsertVertex



## GraphNode

vertex <VertexType> // (*key field*)

adjVertex<List of< VertexType >>

indegree <int>

outdegree <int>

isMarked <boolean>

End GraphNode

# Delete Vertex from Digraph

<ErrorCode> **DeleteVertex** (val **Vertex** <VertexType>)

Deletes an existing vertex.

**Pre**      **Vertex** is the vertex needs to be removed .

**Post**     if **Vertex** 's indegree  $\neq 0$ , the edges ending at this vertex have been removed. Finally, this vertex has been removed.

**Return**   *success*, or *notFound*

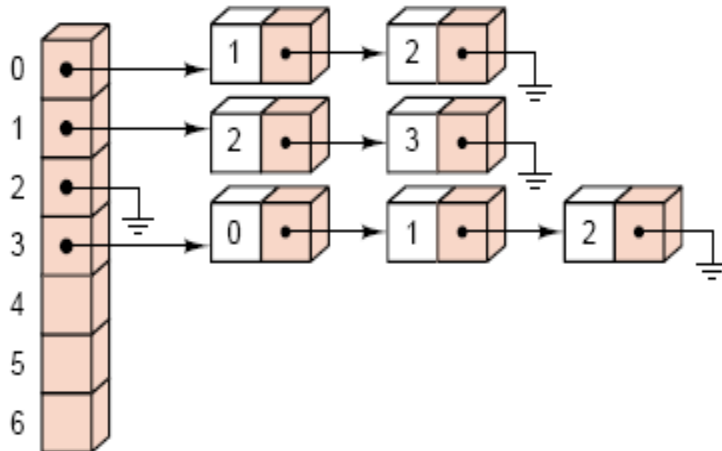
**Uses**     Function **Remove\_EdgeToVertex**.

# Delete Vertex from Digraph

<ErrorCode> **DeleteVertex** (val **Vertex** <VertexType>)

1. DataOut.vertex = **Vertex**
2. if (**digraph**.Retrieve(DataOut) = *success*)
  1. if (DataOut.indegree>0)
    1. **digraph.Remove\_EdgeToVertex**(**Vertex**)
    2. **digraph.Remove**(DataOut)
    3. return *success*
  2. **digraph.Remove**(DataOut)
  3. return *success*
3. else
  1. return *notFound*

End DeleteVertex



## GraphNode

**vertex** <VertexType> // (*key field*)  
**adjVertex**<**List** of< VertexType >>  
**indegree** <int>  
**outdegree** <int>  
**isMarked** <boolean>

End GraphNode

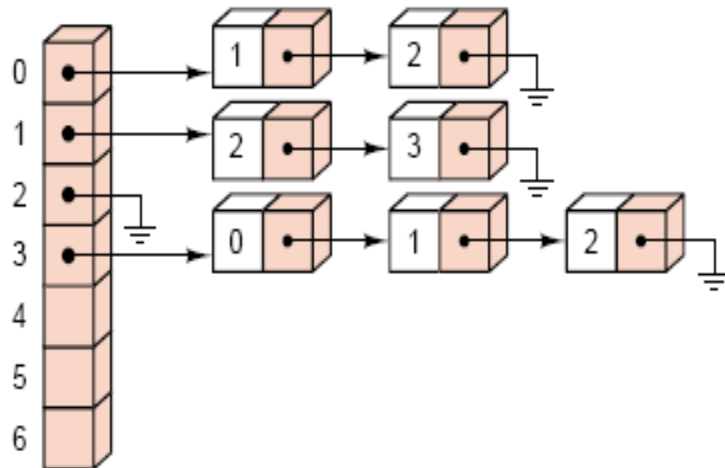
# Auxiliary function Remove all Edges to a Vertex

```
<void> Remove_EdgesToVertex(val VertexTo <VertexType>)
```

Removes all edges from any vertex to **VertexTo** if exist.

1. position = 0
2. **loop** (**digraph**.Retrieve(DataFrom, position) = *success*)
  1. **if** (DataFrom.outdegree>0)
    1. **if** (DataFrom.adjVertex.Remove(**VertexTo**) = *success*)
      1. DataFrom.outdegree = DataFrom.outdegree - 1
      2. **digraph**.Replace(DataFrom, position)
    2. position = position + 1

End Remove\_EdgesToVertex



## GraphNode

**vertex** <VertexType> // (*key field*)

**adjVertex**<List of< VertexType >>

**indegree** <int>

**outdegree** <int>

**isMarked** <boolean>

End GraphNode

# Insert new Edge into Digraph

<ErrorCode> **InsertEdge** (val **VertexFrom**<VertexType>,  
val **VertexTo** <VertexType>)

Inserts new edge into digraph.

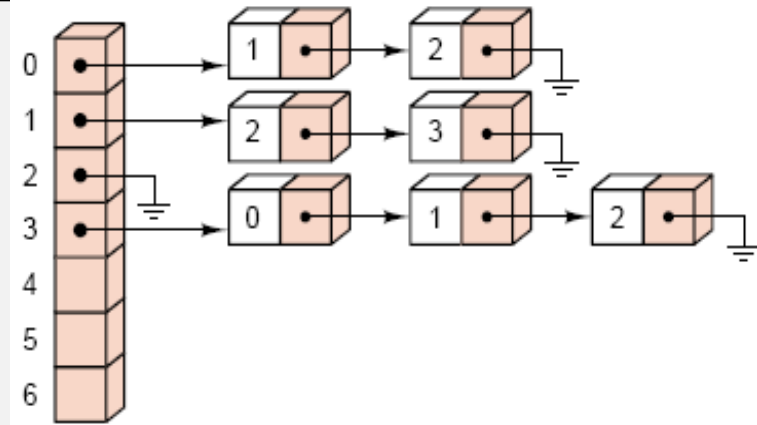
Post if **VertexFrom** and **VertexTo** are in the digraph, and the edge from **VertexFrom** to **VertexTo** is not in the digraph, it has been inserted.

Return *success, overflow, notFound\_VertexFrom, notFound\_VertexTo*  
or *duplicate\_error*

- ```
1. if ( digraph.Retrieve(DataTo) = success )
```

- ```
2. else
    1. return notFound_Ver
```

- ```
1. return notFound_VertexFrom
```



## GraphNode

```
vertex <VertexType> // (key field)
adjVertex<List of< VertexType >>
indegree <int>
outdegree <int>
isMarked <boolean>
```

## End GraphNode

## End InsertEdge



# Delete Edge from Digraph

<ErrorCode> **DeleteEdge** (val **VertexFrom** <VertexType>,  
val **VertexTo** <VertexType>)

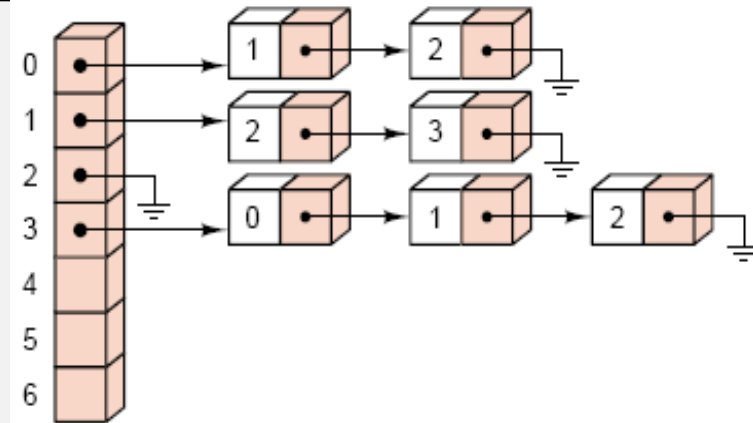
Deletes an existing edge in the digraph.

Post if **VertexFrom** and **VertexTo** are in the digraph, and the edge from **VertexFrom** to **VertexTo** is in the digraph, it has been removed

Return *success*, *notFound\_VertexFrom*, *notFound\_VertexTo* or *notFound\_Edge*

1. DataFrom.vertex = VertexFrom
2. DataTo.vertex = VertexTo
3. if ( digraph.Retrieve(DataFrom) = success )
  1. if ( digraph.Retrieve(DataTo) = success )
    1. newData = DataFrom
    2. if ( newData.adjVertex.Remove(VertexTo) = success )
      1. newData.outdegree = newData.outdegree - 1
      2. digraph.Replace(newData, DataFrom)
      3. return success
    3. else
      1. return notFound\_Edge
  2. else
    1. return notFound\_VertexTo
4. else
  1. return notFound\_VertexFrom

End DeleteEdge



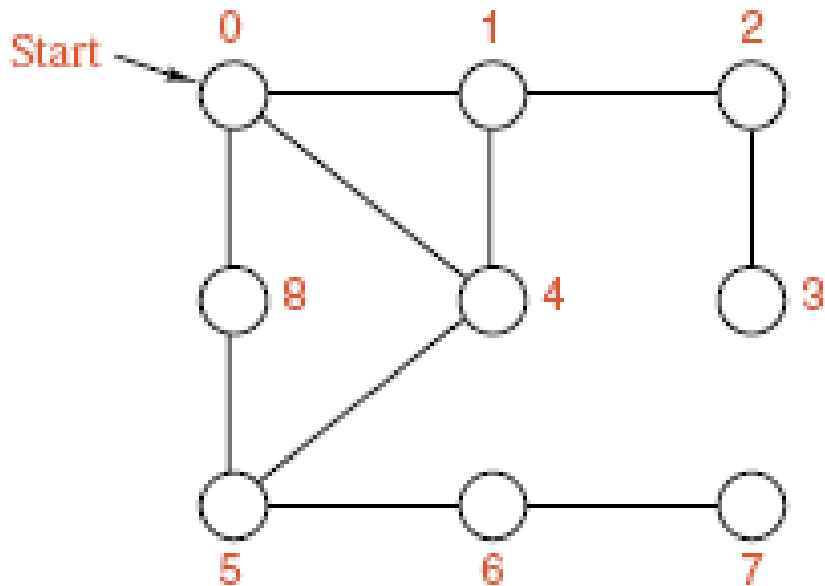
### GraphNode

```
vertex <VertexType> // (key field)
adjVertex<List of< VertexType >>
indegree <int>
outdegree <int>
isMarked <boolean>
```

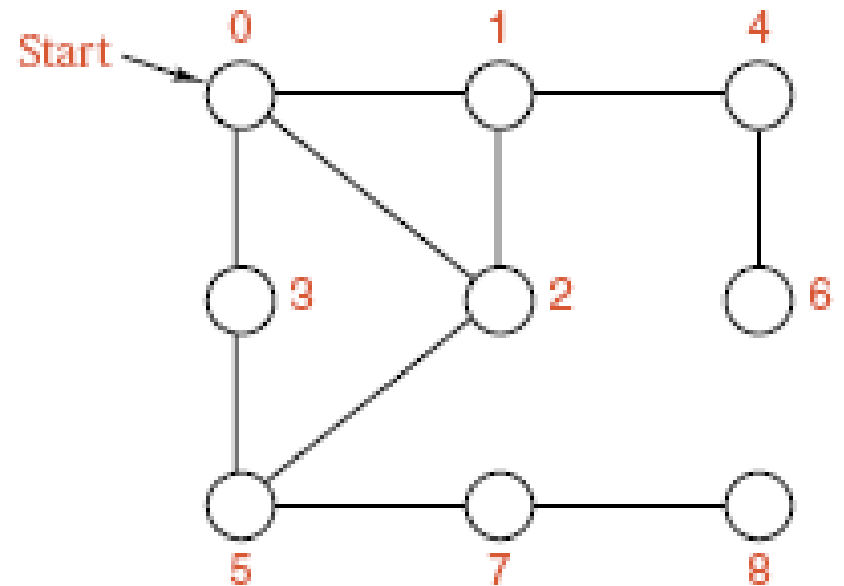
End GraphNode

# Graph Traversal

- **Depth-first traversal:** analogous to preorder traversal of an ordered tree.
- **Breadth-first traversal:** analogous to level-by-level traversal of an ordered tree.

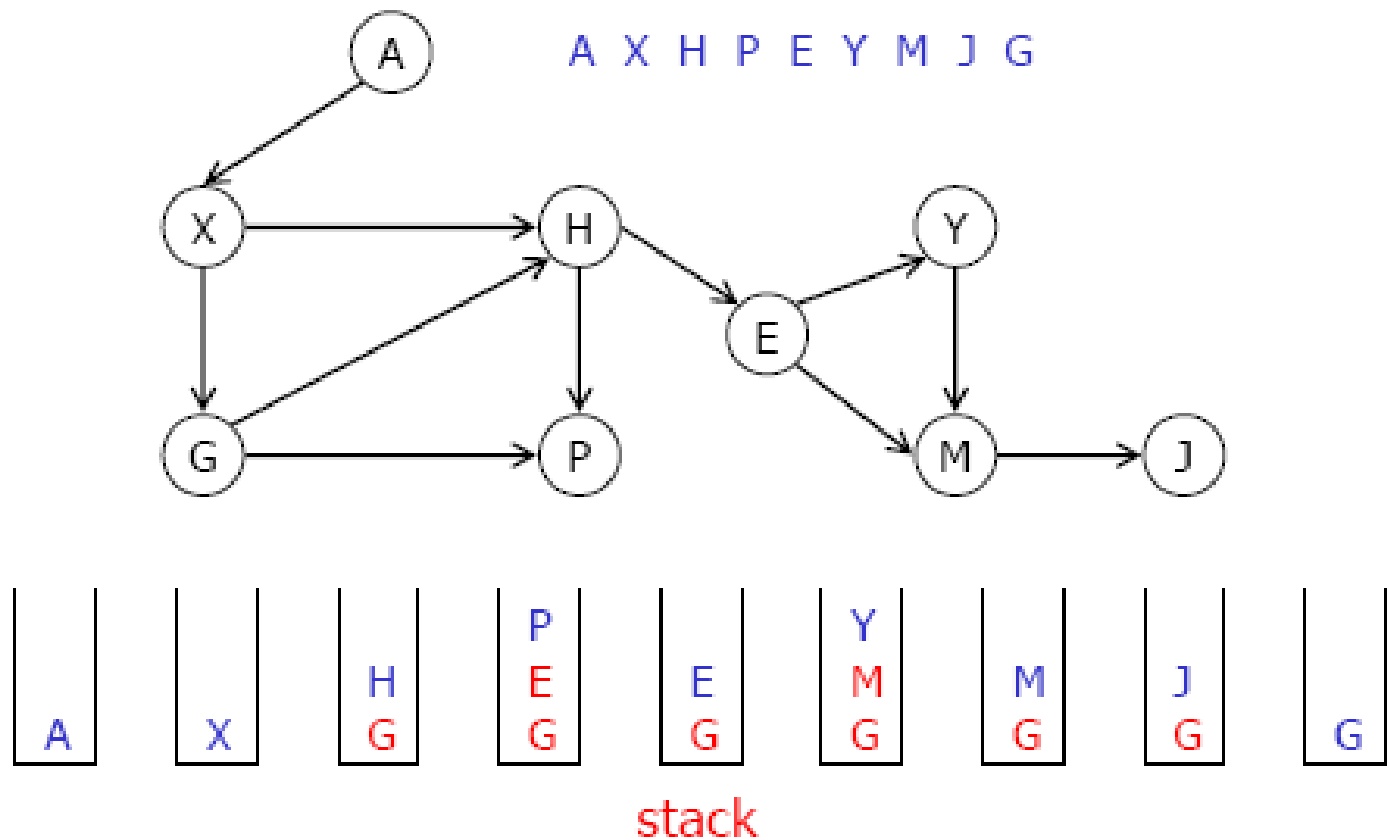


Depth-first traversal

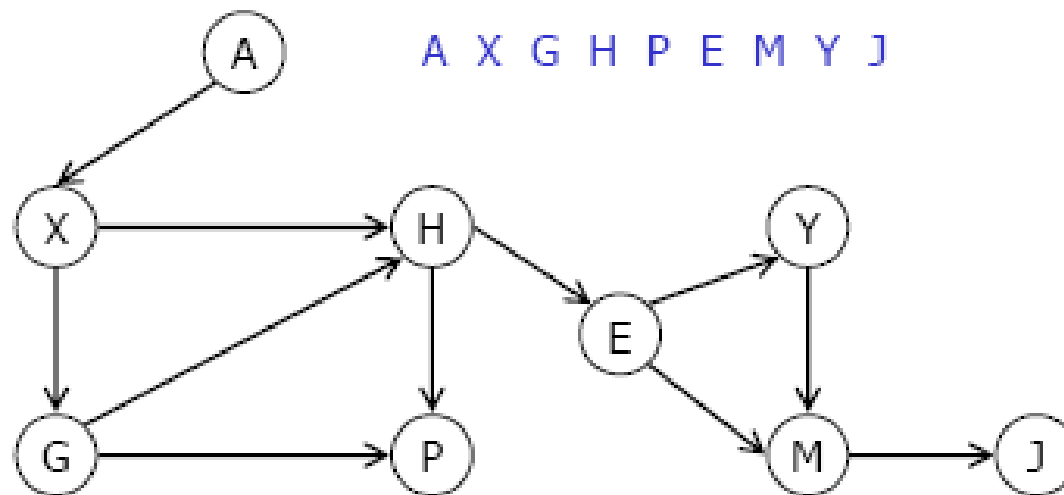


Breadth-first traversal

# Depth-first traversal



# Breadth-first traversal



A X G H P E M Y J



queue

# Depth-first traversal

<void> **DepthFirst**

(ref <void> **Operation** ( ref Data <DataType>))

Traverses the digraph in depth-first order.

**Post**      The function **Operation** has been performed at each vertex of the digraph in depth-first order.

**Uses**     Auxiliary function **recursiveTraverse** to produce the recursive depth-first order.

# Depth-first traversal

<void> **DepthFirst**

(ref <void> **Operation** ( ref Data <DataType>))

- 1. loop** (more vertex v in Digraph)
  - 1.** unmark (v)
- 2. loop** (more vertex v in Digraph)
  - 1. if** (v is unmarked)
    - 1. recursiveTraverse** (v, **Operation**)

End DepthFirst

# Depth-first traversal

```
<void> recursiveTraverse (ref v <VertexType>,  
                           ref <void> Operation ( ref Data <DataType> ) )
```

Traverses the digraph in depth-first order.

**Pre**      v is a vertex of the digraph.

**Post**     The depth-first traversal, using function **Operation**, has been completed for v and for all vertices that can be reached from v.

**Uses**    function **recursiveTraverse** recursively.



# Depth-first traversal

```
<void> recursiveTraverse(ref v <VertexType>,  
                        ref <void> Operation ( ref Data <DataType> ) )  
  
1. mark(v)  
2. Operation(v)  
3. loop (more vertex w adjacent to v)  
    1. if (vertex w is unmarked)  
        1. recursiveTraverse (w, Operation)  
End Traverse
```

# Breadth-first traversal

**<void> BreadthFirst**

(ref <void> **Operation** ( ref Data <DataType> ) )

Traverses the digraph in breadth-first order.

**Post**      The function **Operation** has been performed at each vertex of the digraph in breadth-first order.

**Uses**      Queue ADT.

## // BreadthFirst

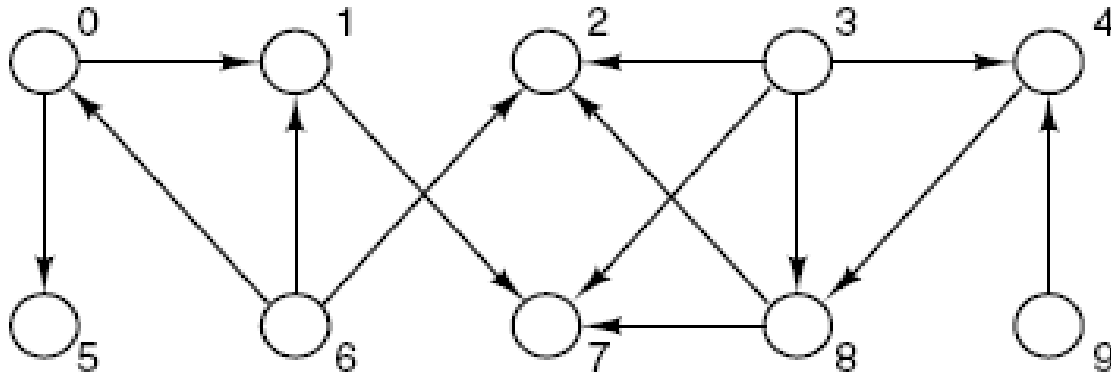
1. `queueObj <Queue>`
2. **loop** (more vertex `v` in digraph)
  1. `unmark(v)`
3. **loop** (more vertex `v` in Digraph)
  1. **if** (vertex `v` is unmarked)
    1. `queueObj.Enqueue(v)`
    2. **loop** (NOT `queueObj.IsEmpty()`)
      1. `queueObj.QueueFront(w)`
      2. `queueObj.DeQueue()`
      3. **if** (vertex `w` is unmarked)
        1. `mark(w)`
        2. `Operation(w)`
        3. **loop** (more vertex `x` adjacent to `w`)
          1. `queueObj.Enqueue(x)`

End BreadthFirst

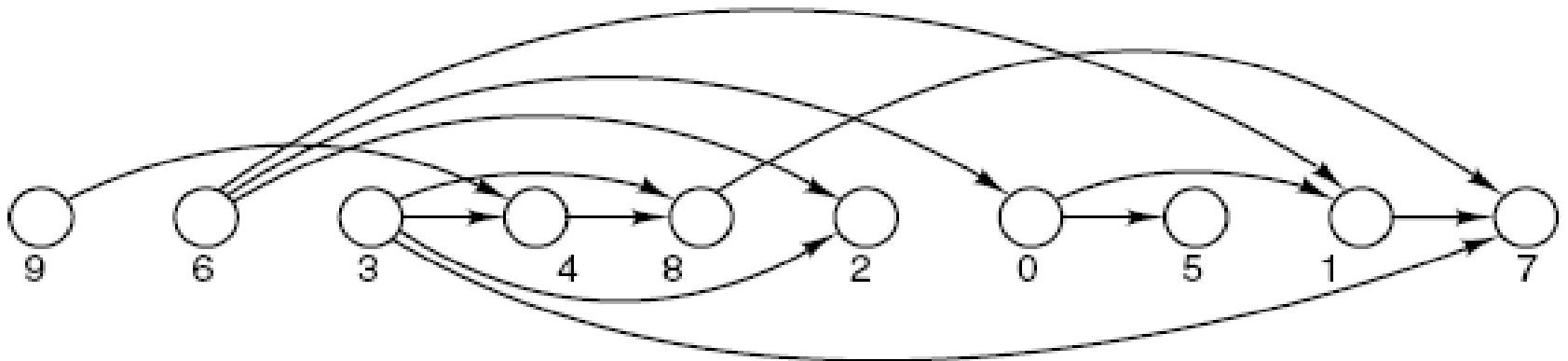
# Topological Order

A **topological order** for  $G$ , a **directed graph with no cycles**, is a sequential listing of all the vertices in  $G$  such that, for all vertices  $v, w \in G$ , if there is an edge from  $v$  to  $w$ , then  $v$  precedes  $w$  in the sequential listing.

# Topological Order

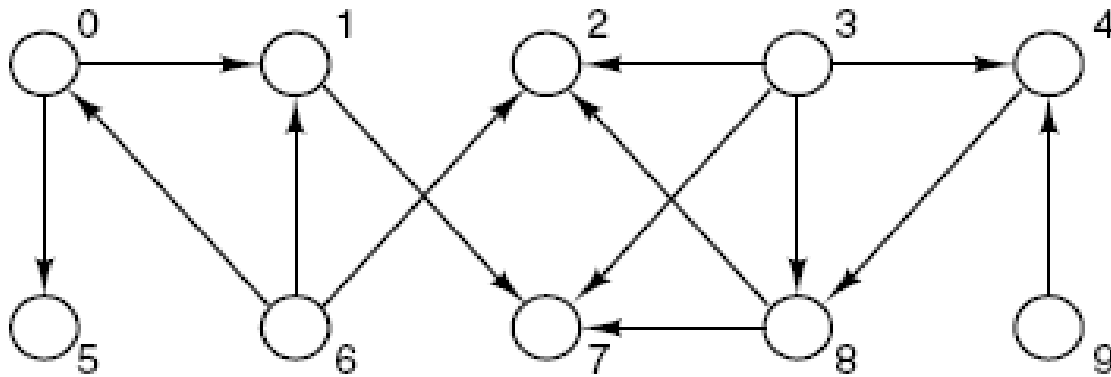


Directed graph with no directed cycles

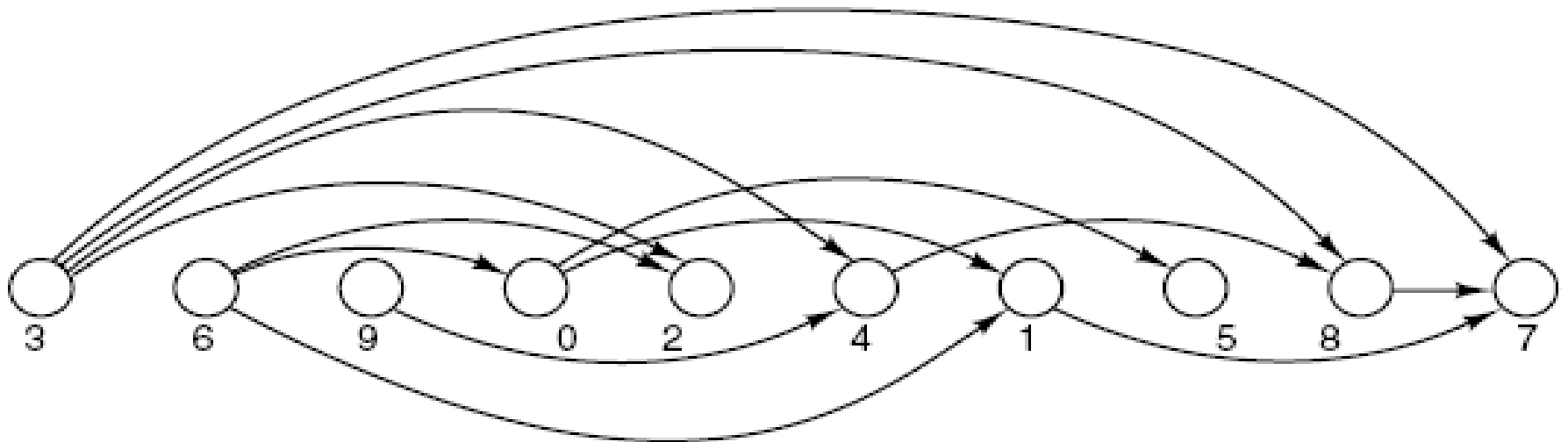


Depth-first ordering

# Topological Order



Directed graph with no directed cycles



Breadth-first ordering

# Applications of Topological Order

Topological order is used for:

- Courses available at a university,
  - Vertices: course.
  - Edges:  $(v,w)$ ,  $v$  is a prerequisite for  $w$ .
  - A **topological order** is a listing of all the courses such that all prerequisites for a course appear before it does.
- A glossary of technical terms: no term is used in a definition before it is itself defined.
- The topics in the textbook.

# Topological Order

<void> **DepthTopoSort** (ref **TopologicalOrder** <List>)

Traverses the digraph in depth-first order and made a list of topological order of digraph's vertices.

**Pre**      Acyclic digraph.

**Post**     The vertices of the digraph are arranged into the list **TopologicalOrder** with a depth-first traversal of those vertices that do not belong to a cycle.

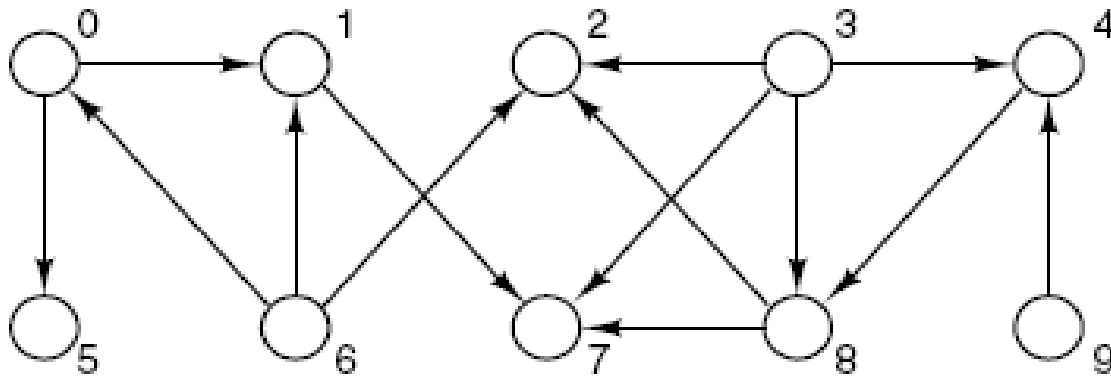
**Uses**    List ADT and function **recursiveDepthTopoSort** to perform depth-first traversal.

## Idea:

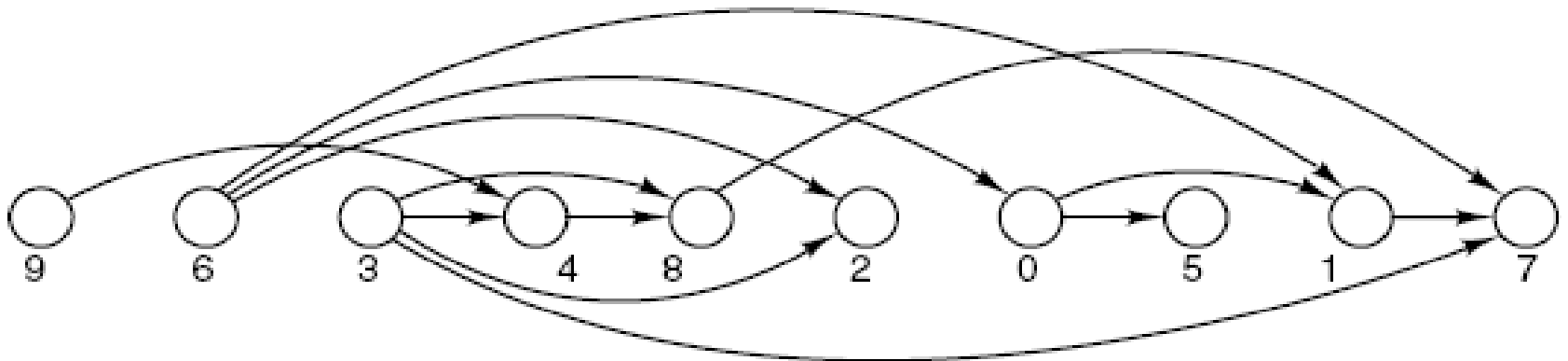
- *Starts by finding a vertex that has no successors and place it last in the list.*
- *Repeatedly add vertices to the beginning of the list.*
- *By recursion, places all the successors of a vertex into the topological order.*
- *Then, place the vertex itself in a position before any of its successors.*



# Topological Order



Directed graph with no directed cycles



Depth-first ordering

# Topological Order

<void> **DepthTopoSort** (ref TopologicalOrder <List>)

1. **loop** (more vertex v in digraph)
  1. unmark(v)
2. TopologicalOrder.clear()
3. **loop** (more vertex v in Digraph)
  1. **if** (vertex v is unmarked)
    1. **recursiveDepthTopoSort**(v, TopologicalOrder)

End DepthTopoSort

# Topological Order

```
<void> recursiveDepthTopoSort (val v <VertexType>,  
                                ref TopologicalOrder <List>)
```

**Pre** Vertex **v** in digraph does not belong to the partially completed list **TopologicalOrder**.

**Post** All the successors of **v** and finally **v** itself are added to **TopologicalOrder** with a depth-first order traversal.

**Uses** List ADT and the function **recursiveDepthTopoSort**.

## Idea:

- *Performs the recursion, based on the outline for the general function traverse.*
- *First, places all the successors of **v** into their positions in the topological order.*
- *Then, places **v** into the order.*

# Topological Order

```
<void> recursiveDepthTopoSort (val v <VertexType>,  
                                ref TopologicalOrder <List>)
```

1. mark(v)
  2. **loop** (more vertex w adjacent to v)
    1. **if** (vertex w is unmarked)
      1. **recursiveDepthTopoSort**(w, **TopologicalOrder**)
  3. **TopologicalOrder.Insert**(0, v)
- End recursiveDepthTopoSort

# Topological Order

<void> **BreadthTopoSort** (ref **TopologicalOrder** <List>)

Traverses the digraph in depth-first order and made a list of topological order of digraph's vertices.

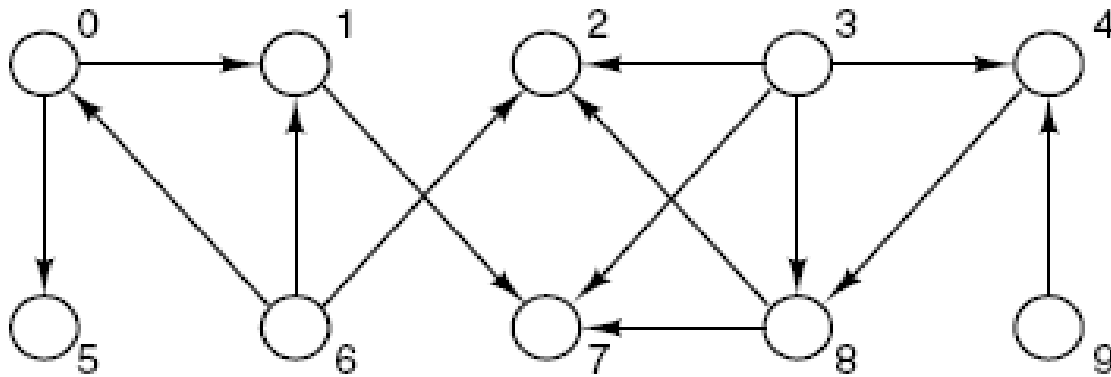
**Post** The vertices of the digraph are arranged into the list **TopologicalOrder** with a breadth-first traversal of those vertices that do not belong to a cycle.

**Uses** List and Queue ADT.

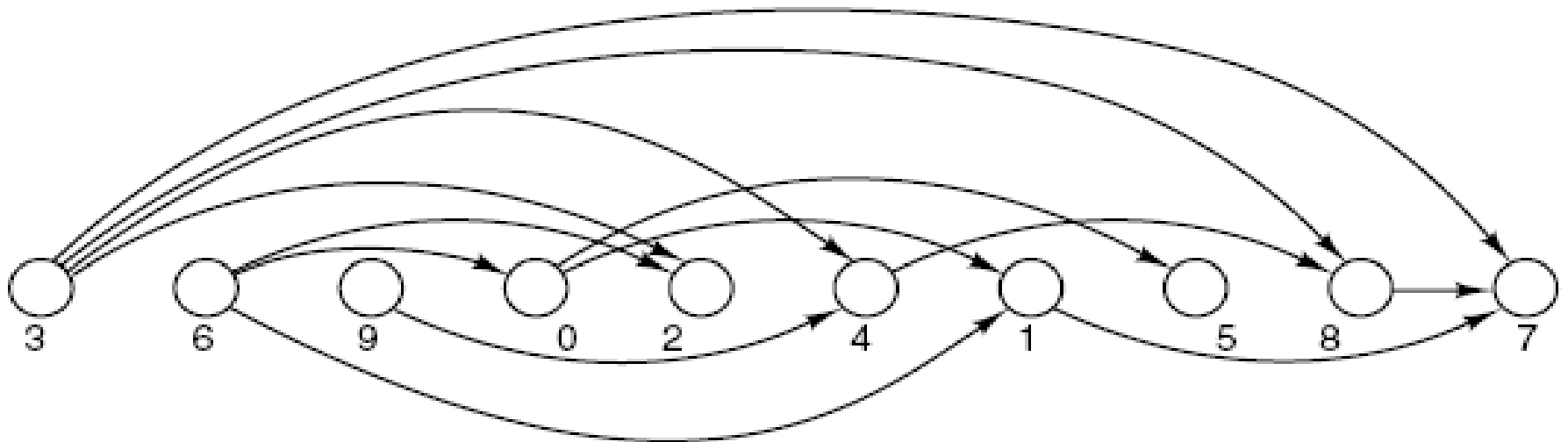
## Idea:

- *Starts by finding the vertices that are not successors of any other vertex.*
- *Places these vertices into a queue of vertices to be visited.*
- *As each vertex is visited, it is removed from the queue and placed in the next available position in the topological order (starting at the beginning).*
- *Reduces the indegree of its successors by 1.*
- *The vertex having the zero value indegree is ready to processed and is places into the queue.*

# Topological Order



Directed graph with no directed cycles



Breadth-first ordering

<void> **BreadthTopoSort** (ref TopologicalOrder <List>)

1. TopologicalOrder.clear()
2. queueObj <Queue>
3. **loop** (more vertex v in digraph)
  1. **if** (indegree of v = 0)
    1. queueObj.Enqueue(v)
4. **loop** (NOT queueObj.isEmpty())
  1. queueObj.QueueFront(v)
  2. queueObj.DeQueue()
  3. TopologicalOrder.Insert(TopologicalOrder.size(), v)
  4. **loop** (more vertex w adjacent to v)
    1. decrease the indegree of w by 1
    2. **if** (indegree of w = 0)
      1. queueObj.Enqueue(w)

End BreadthTopoSort

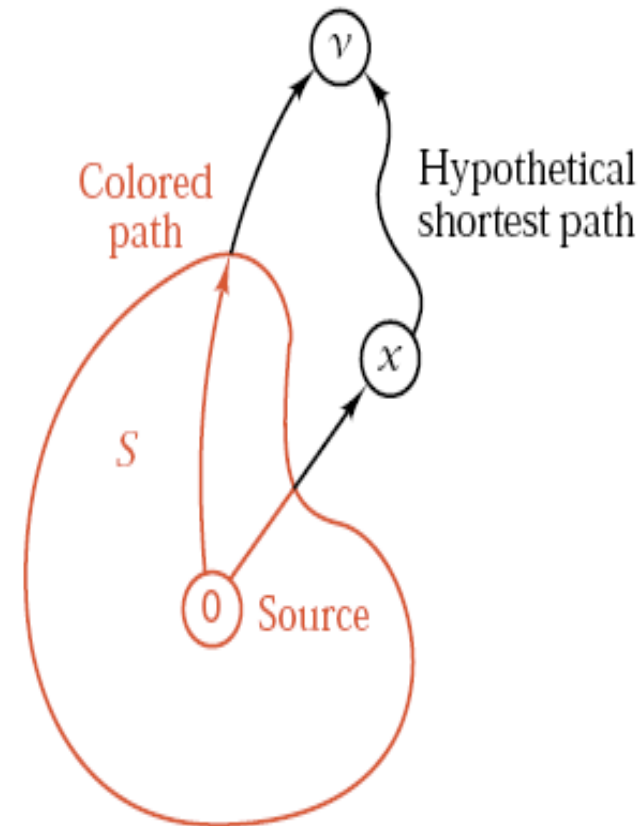
# Shortest Paths

- Given a directed graph in which each edge has a nonnegative weight.
- Find a path of least total weight from a given vertex, called the source, to every other vertex in the graph.
- A greedy algorithm of Shortest Paths: Dijkstra's algorithm (1959).



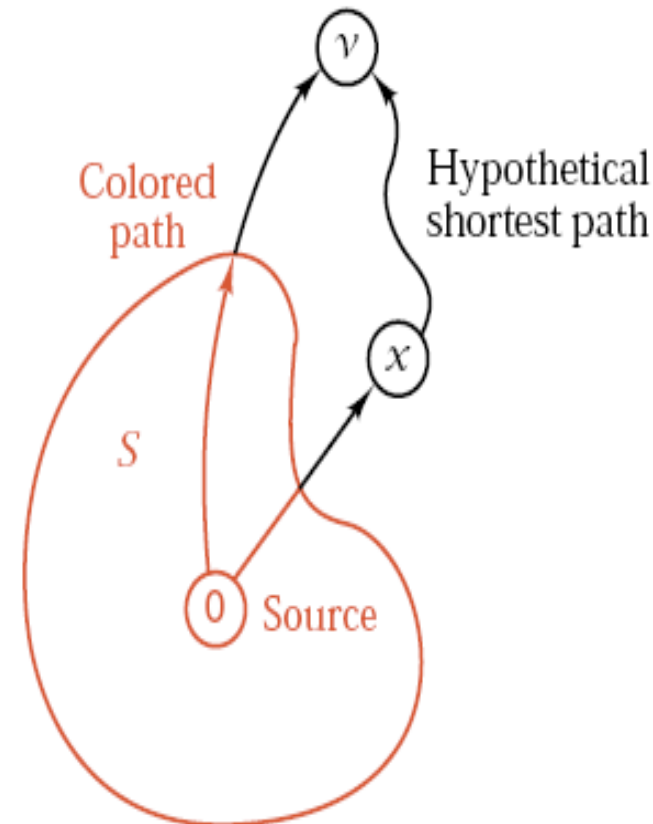
# Dijkstra's algorithm

- Let tree is the subgraph contains the shortest paths from the source vertex to all other vertices.
- At first, add the source vertex to the tree.
- Loop until all vertices are in the tree:
  - Consider the **adjacent vertices** of the vertices already in the tree.
  - Examine all **the paths from those adjacent vertices to the source vertex**.
  - Select the **shortest path** and insert the corresponding adjacent vertex into the tree.

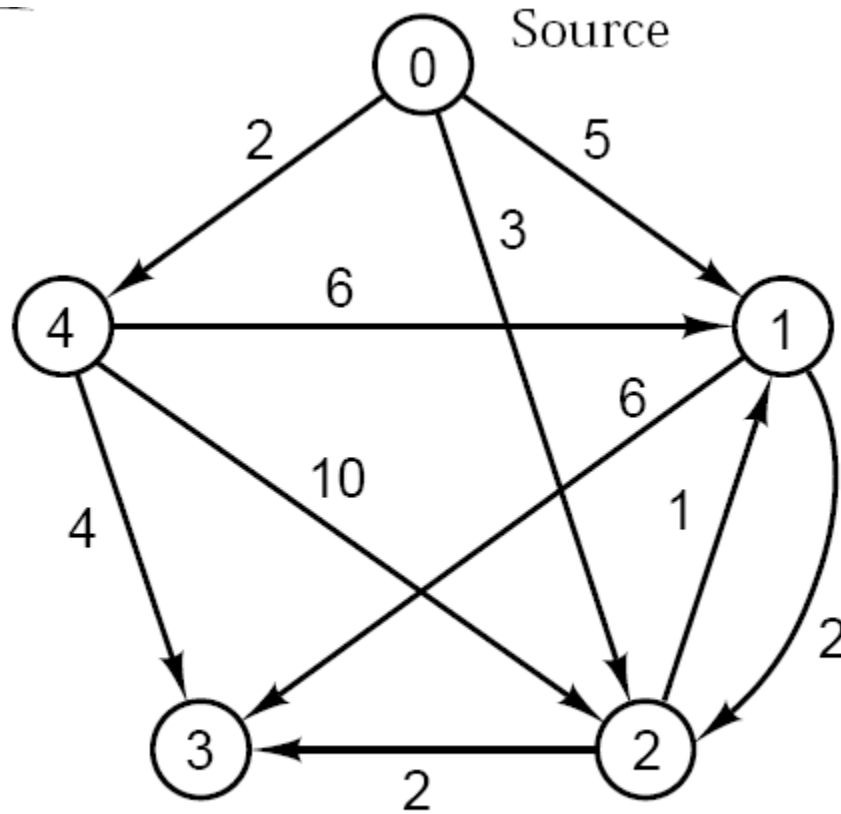


# Dijkstra's algorithm in detail

- S: Set of vertices whose closest distances to the source are known.
- Add one vertex to S at each stage.
- For each vertex  $v$ , maintain the distance from the source to  $v$ , along a path all of whose vertices are in S, except possibly the last one.
- To determine what vertex to add to S at each step, apply the **greedy criterion** of choosing the vertex  $v$  with **the smallest distance**.
- Add  $v$  to S.
- Update distance from the source for all  $w$  not in S, if the **path through  $v$  and then directly to  $w$**  is shorter than the previously recorded distance to  $w$ .

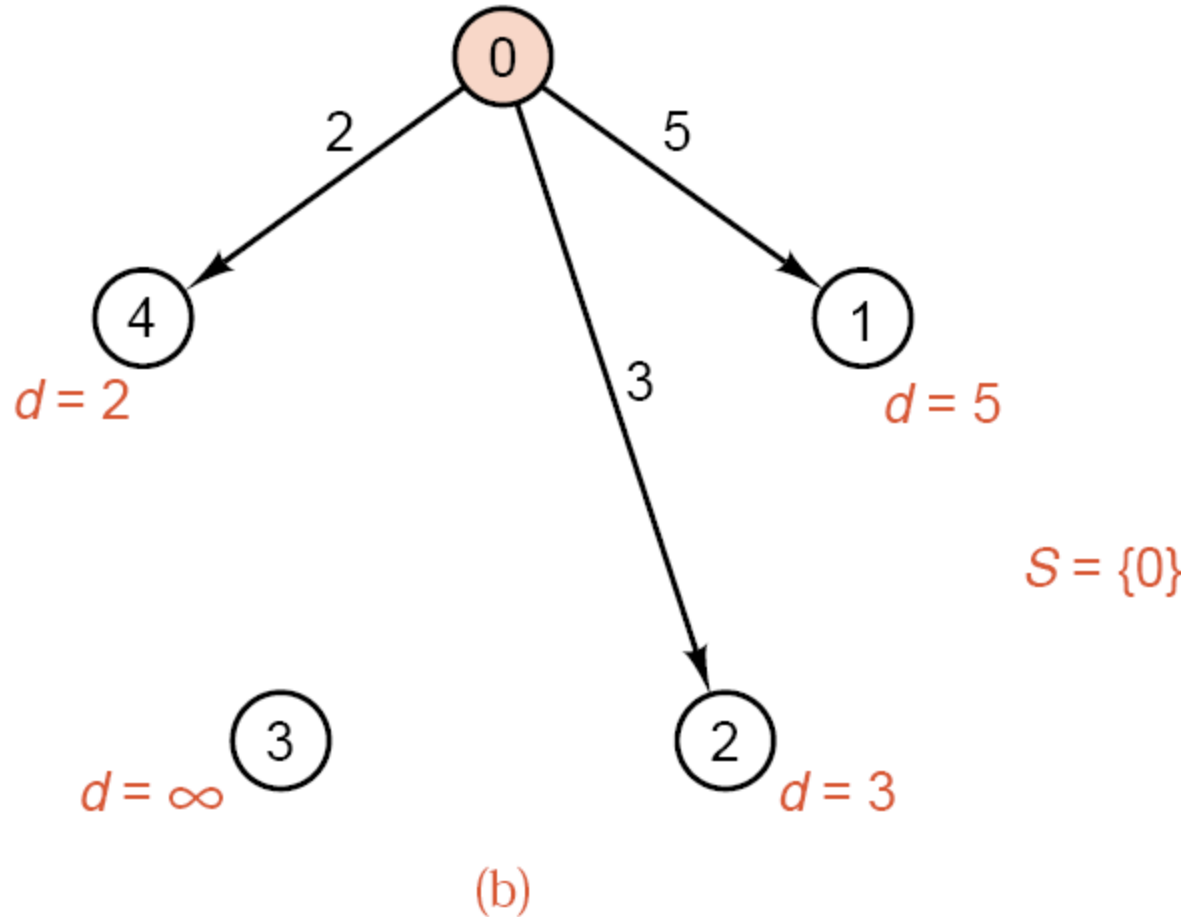


# Dijkstra's algorithm

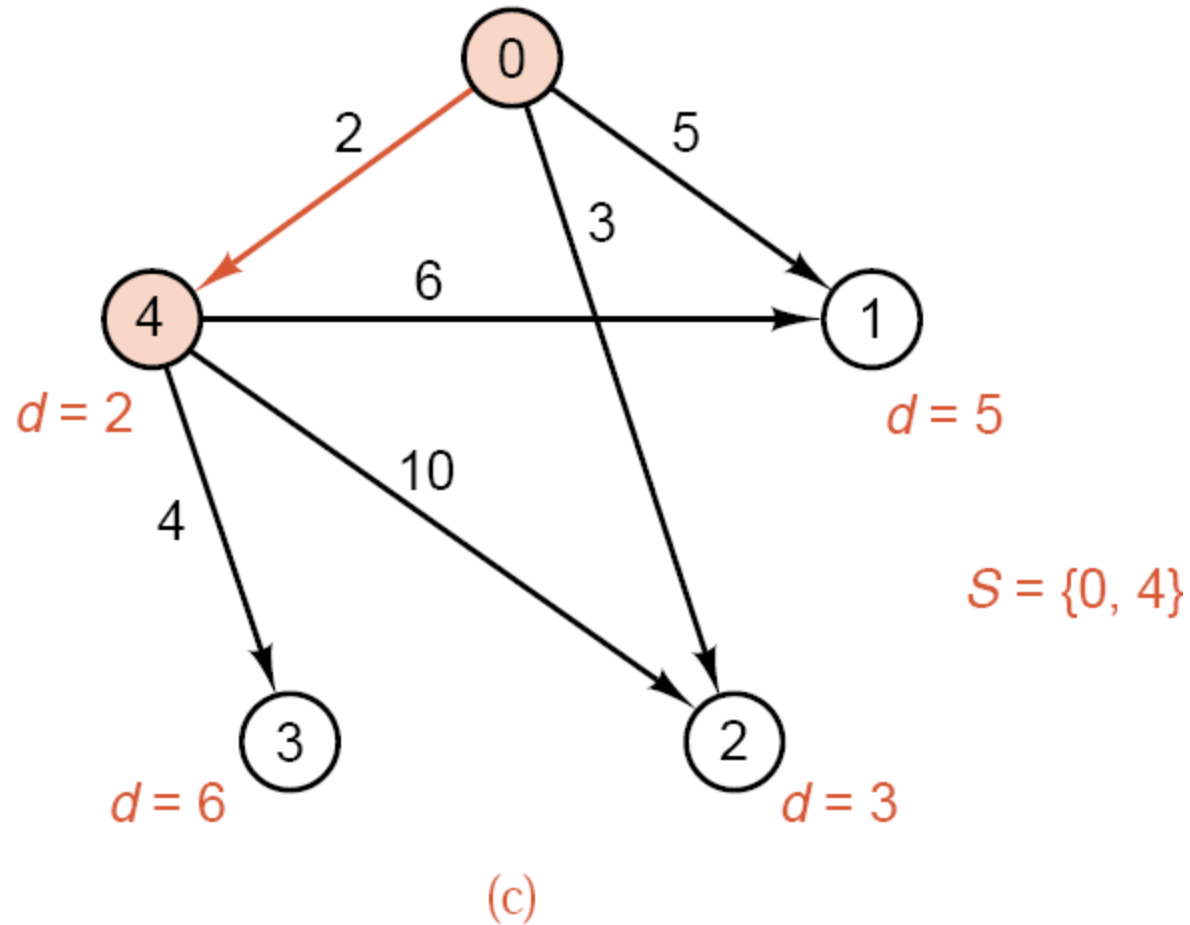


(a)

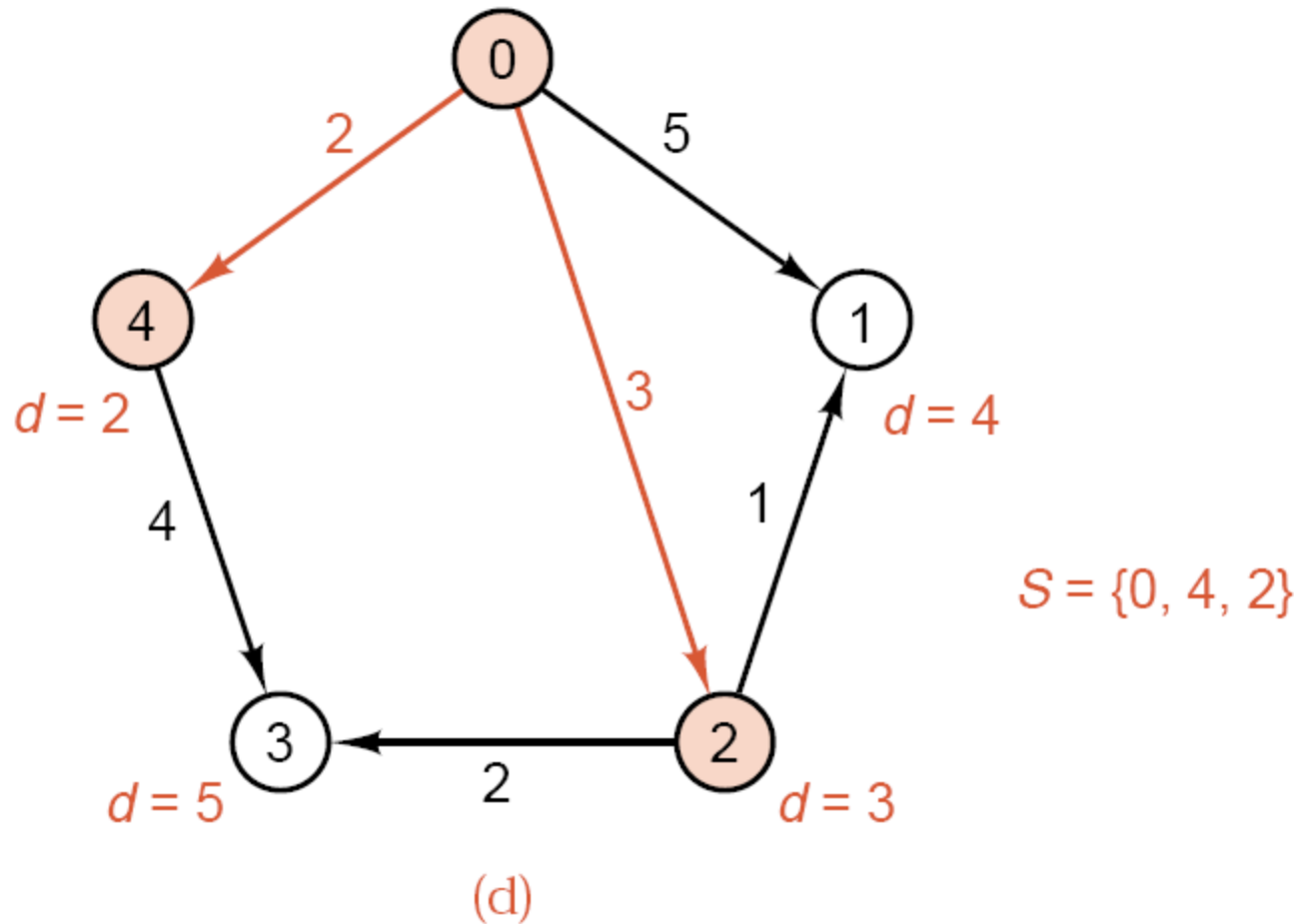
# Dijkstra's algorithm



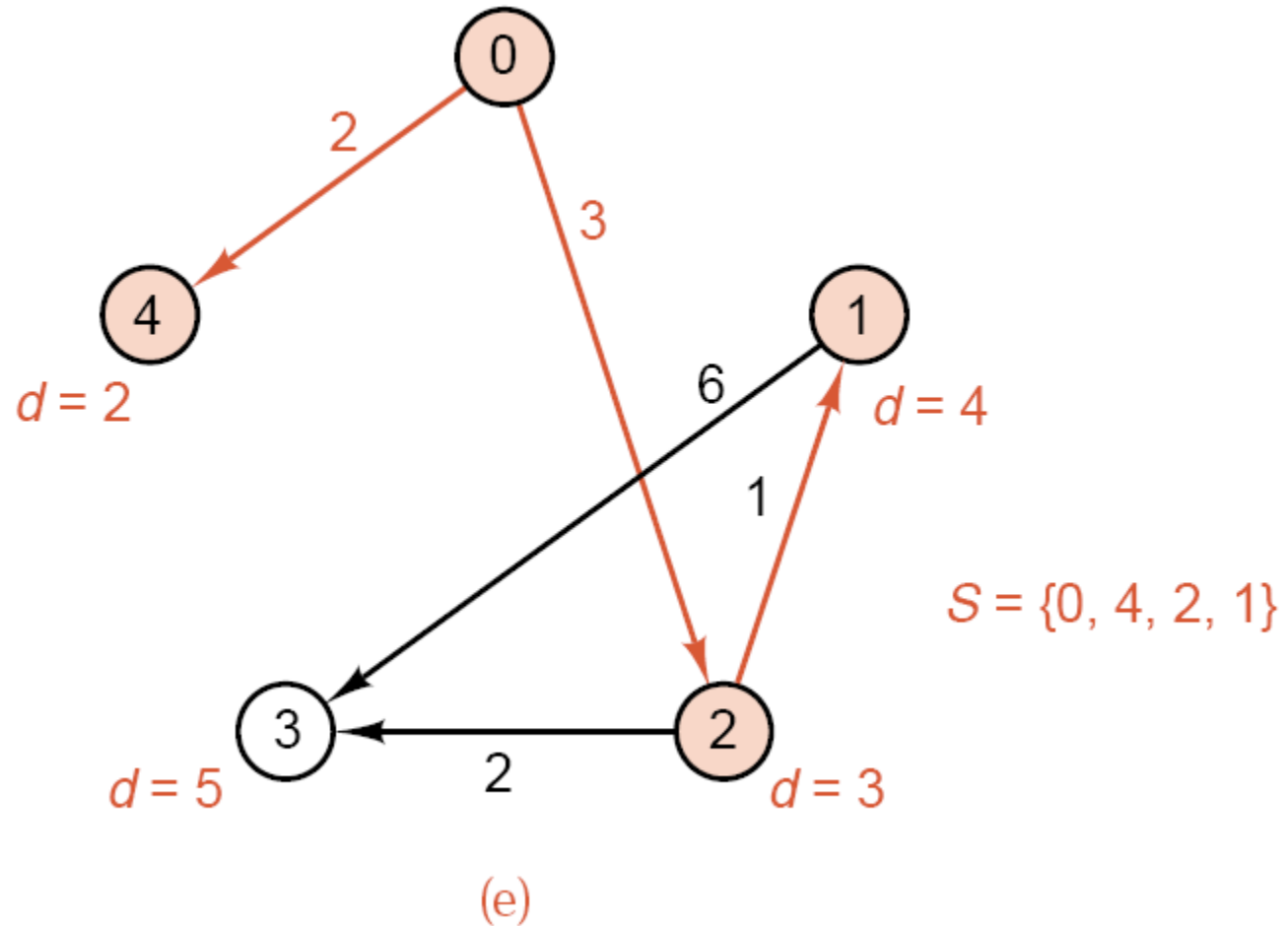
# Dijkstra's algorithm



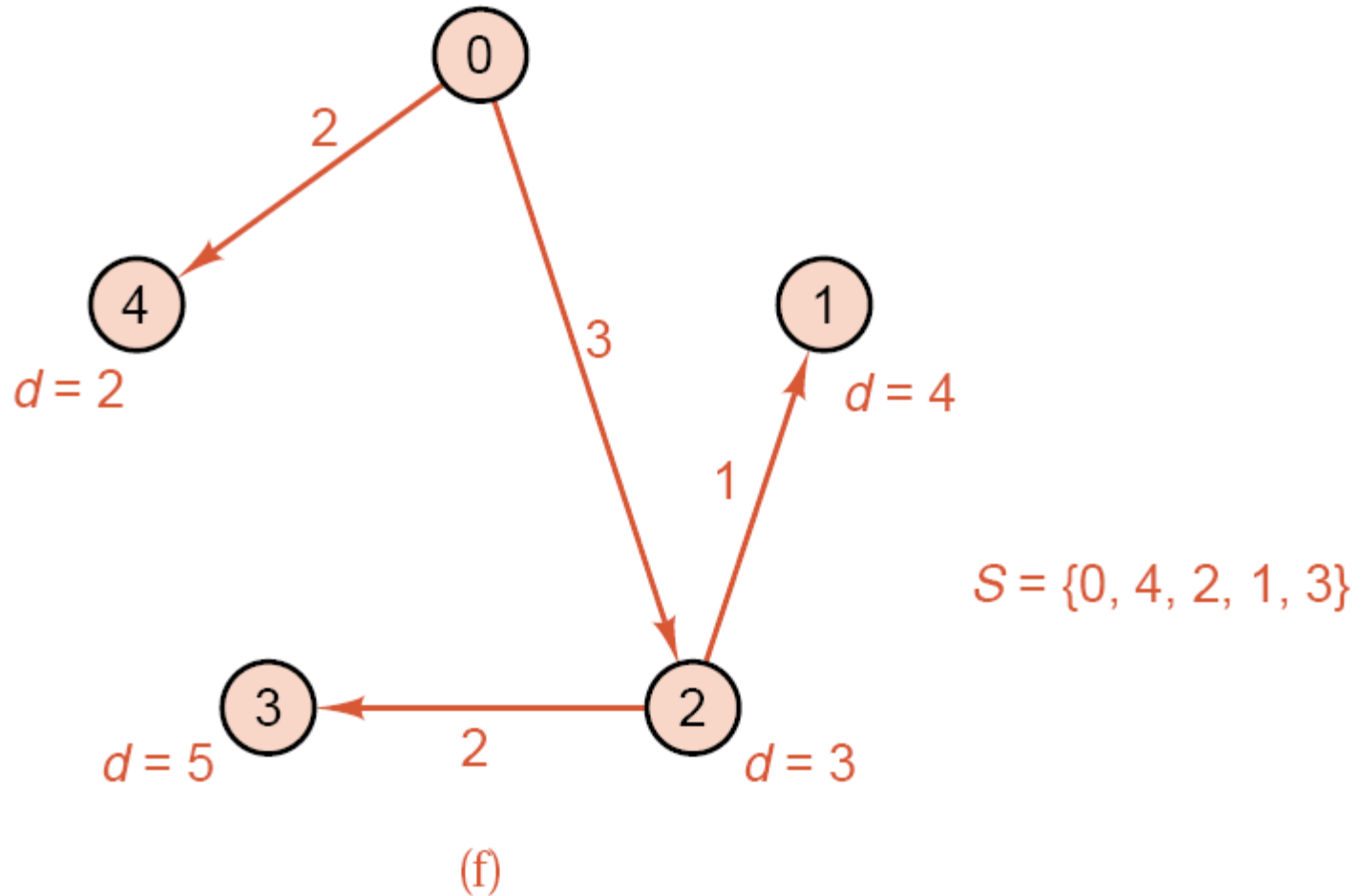
# Dijkstra's algorithm



# Dijkstra's algorithm



# Dijkstra's algorithm





# Dijkstra's algorithm

```
<void> ShortestPath (val source <VertexType>,  
    ref listOfShortestPath <List of <DistanceNode>>)
```

Finds the shortest paths from source to all other vertices in digraph.

**Post** Each node in **listOfShortestPath** gives the minimal path weight from vertex **source** to vertex **destination** in **distance** field.

DistanceNode

**destination** <VertexType>

**distance** <int>

End DistanceNode

## // ShortestPath

1. listOfShortestPath.clear()
2. Add source to set S
3. **loop** (more vertex v in digraph) *// Initiate all distances from source to v*
  1. distanceNode.destination = v
  2. distanceNode.distance = weight of edge(source, v) *// = infinity if edge(source,v) isn't in digraph.*
  3. listOfShortestPath.Insert(distanceNode)
4. **loop** (more vertex not in S) *// Add one vertex v to S on each step.*
  1. minWeight = infinity *// Choose vertex v with smallest distance.*
  2. **loop** (more vertex w not in S)
    1. Find the distance x from source to w in listOfShortestPath
    2. **if** (x < minWeight)
      1. v = w
      2. minWeight = x
  3. Add v to S.

DistanceNode

destination <VertexType>

distance <int>

End DistanceNode

*// ShortestPath (continue)*

4. **loop** (more vertex  $w$  not in  $S$ ) *// Update distances from source  
// to all  $w$  not in  $S$* 
  1. Find the distance  $x$  from source to  $w$  in **listOfShortestPath**
  2. **if** (  $(\text{minWeight} + \text{weight of edge from } v \text{ to } w) < x$  )
    1. Update distance from source to  $w$  in **listOfShortestPath**  
to  $(\text{minWeight} + \text{weight of edge from } v \text{ to } w)$

End ShortestPath

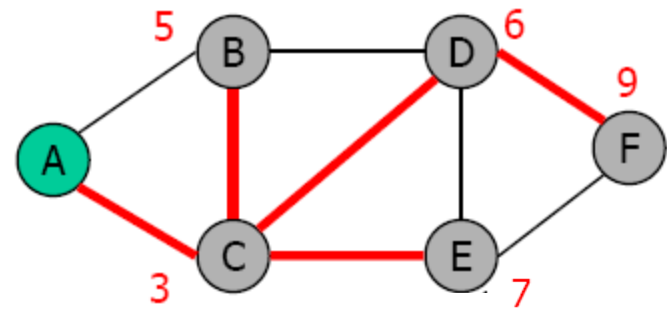
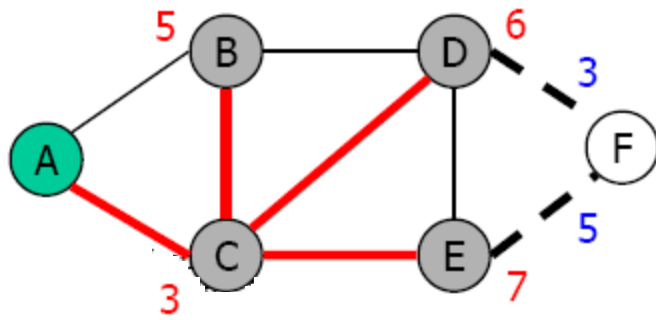
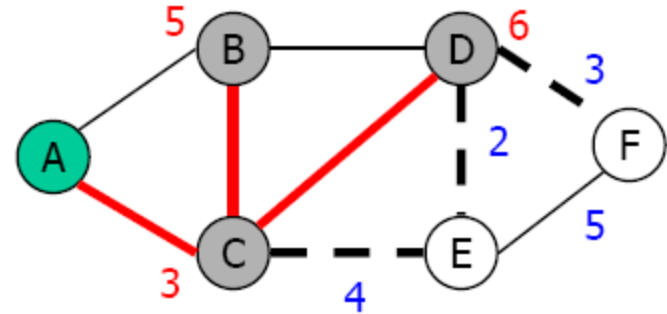
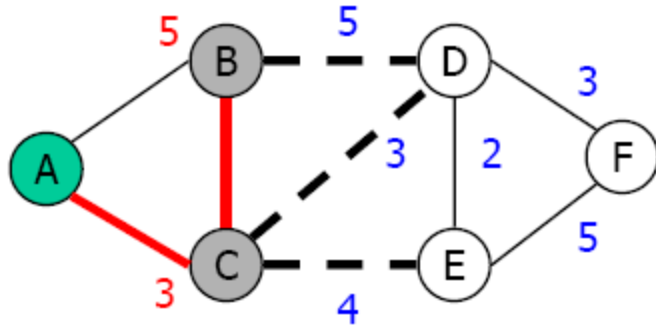
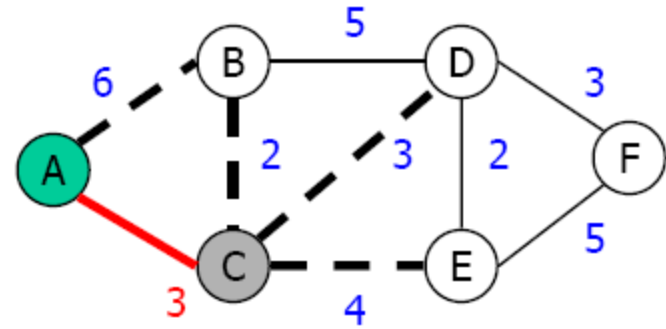
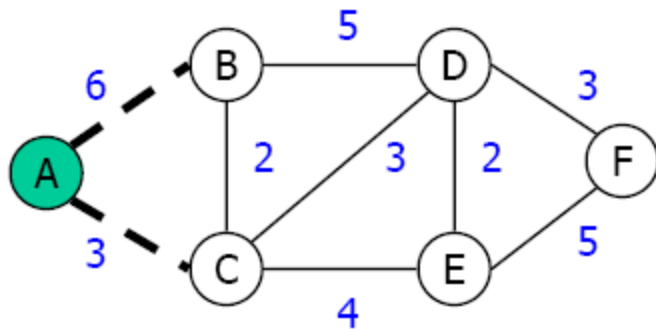
DistanceNode

**destination** <VertexType>

**distance** <int>

End DistanceNode

# Another example of Shortest Paths



*Select the adjacent vertex having minimum path to the source vertex*

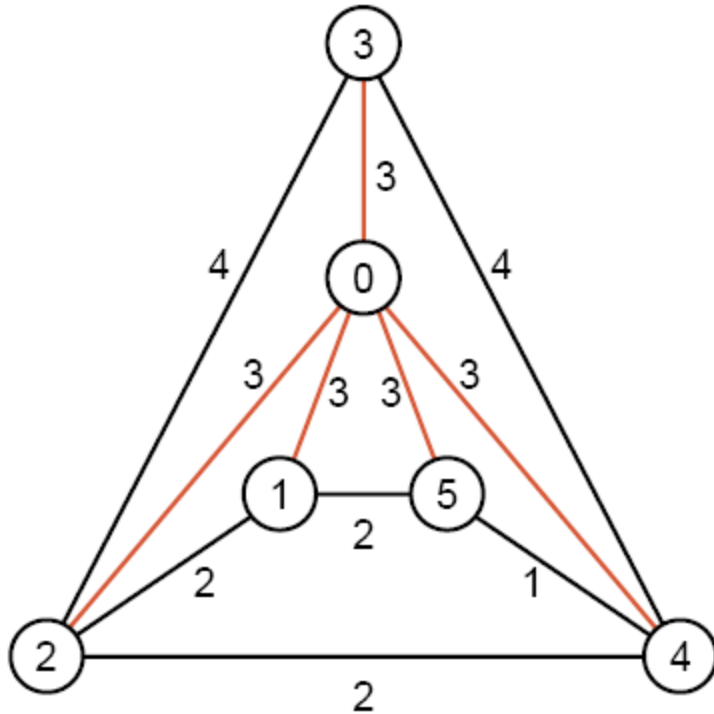
# Minimum spanning tree

## DEFINITION:

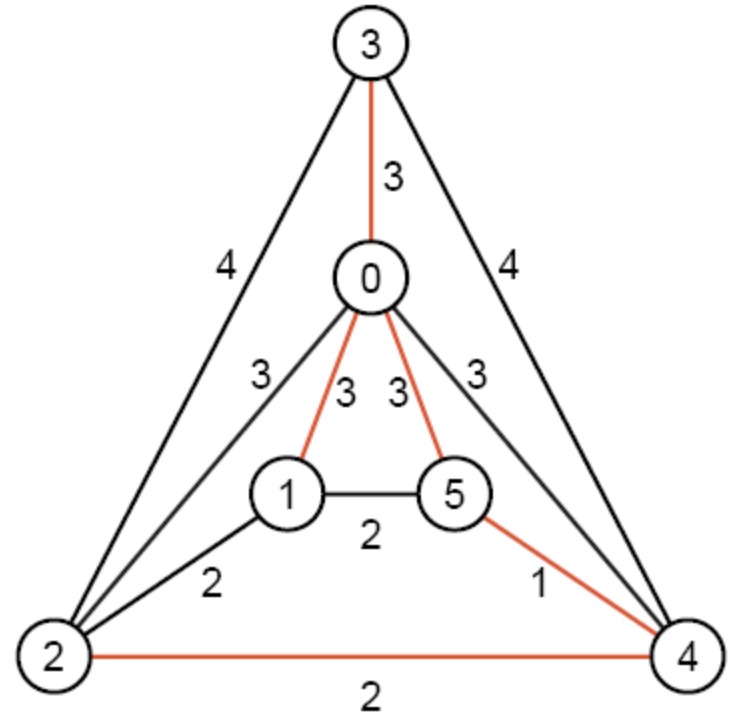
**Spanning tree:** tree that contains all of the vertices in a connected graph.

**Minimum spanning tree:** spanning tree such that the sum of the weights of its edges is minimal.

# Spanning Trees



Weight sum of tree = 15  
(a)



Weight sum of tree = 12  
(b)

Two spanning trees in a network

# A greedy Algorithm: Minimum Spanning Tree

- Shortest path algorithm in a connected graph found an its **spanning tree**.
- What is the algorithm finding the **minimum spanning tree**?
- A small change to shortest path algorithm can find the minimum spanning tree, that is **Prim's algorithm** since 1957.

# Prim's algorithm

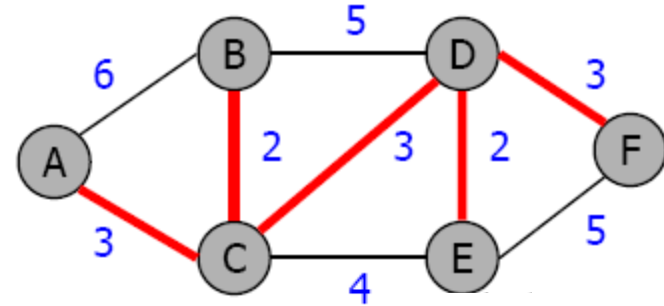
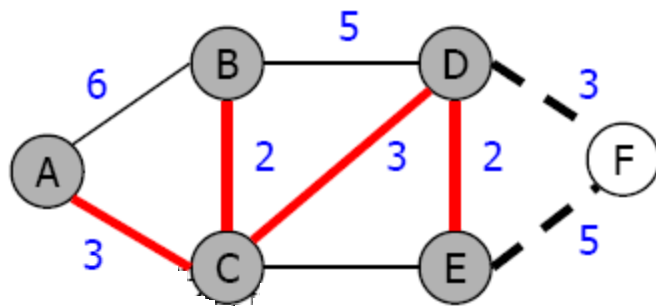
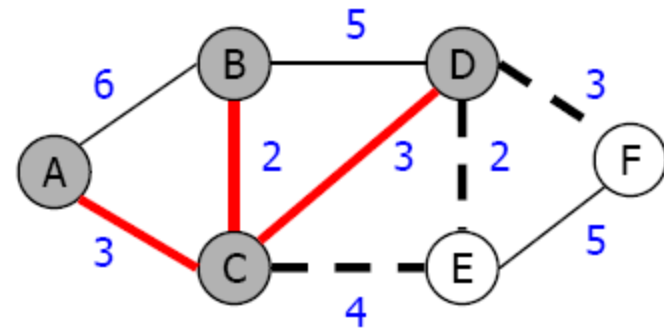
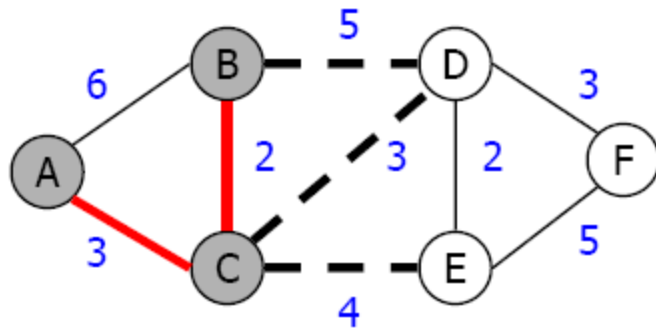
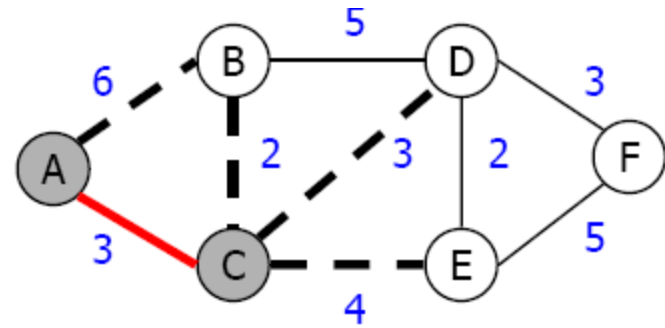
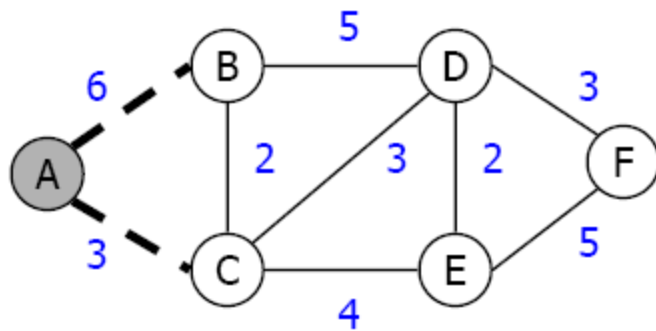
- Let tree is the minimum spanning tree.
- At first, add one vertex to the tree.
- Loop until all vertices are in the tree:
  - Consider the adjacent vertices of the vertices already in the tree.
  - Examine all the edges from each vertices already in the tree to those adjacent vertices.
  - Select the smallest edge and insert the corresponding adjacent vertex into the tree.



# Prim's algorithm in detail

- Let  $S$  is the set of vertices already in the minimum spanning tree.
- At first, add one vertex to  $S$ .
- For each vertex  $v$  not in  $S$ , maintain the distance from a vertex  $x$  to  $v$ , where  $x$  is a vertex in  $S$  and the  $\text{edge}(x,v)$  is the smallest in all edges from another vertices in  $S$  to  $v$  (this  $\text{edge}(x,v)$  is called the distance from  $S$  to  $v$ ). As usual, all edges not being in graph have infinity value.
- To determine what vertex to add to  $S$  at each step, apply the greedy criterion of choosing the vertex  $v$  with the smallest distance from  $S$ .
- Add  $v$  to  $S$ .
- Update distances from  $S$  to all vertices  $v$  not in  $S$  if they are smaller than the previously recorded distances.

# Prim's algorithm



*Select the adjacent vertex having minimum edge to the vertices already in the tree.*

# Prim's algorithm

```
<void> MinimumSpanningTree (val source <VertexType>,  
                             ref tree <Graph>)
```

Finds the minimum spanning tree of a connected component of the original graph that contains vertex **source**.

**Post** **tree** is the minimum spanning **tree** of a connected component of the original graph that contains vertex **source**.

Uses local variables:

- Set S
- listOfDistanceNode
- continue <boolean>

DistanceNode

**vertexFrom** <VertexType>

**vertexTo** <VertexType>

**distance** <WeightType>

End DistanceNode

1. `tree.clear()`
2. `tree.InsertVertex(source)`
3. Add `source` to set `S`
4. `listOfDistanceNode.clear()`
5. `distanceNode.vertexFrom = source`
6. **loop** (more vertex `v` in `graph`) *// Initiate all distances from source to v*
  1. `distanceNode.vertexTo = v`
  2. `distanceNode.distance = weight of edge(source, v)` *// = infinity if*  
*// edge(source, v) isn't in graph.*
  3. `listOfDistanceNode.Insert(distanceNode)`

DistanceNode

`vertexFrom` <VertexType>

`vertexTo` <VertexType>

`distance` <WeightType>

End DistanceNode

7. continue = TRUE
8. **loop** (more vertex not in S) and (continue) *//Add one vertex to S on  
// each step*
  1. minWeight = infinity *//Choose vertex v with smallest distance to S*
  2. **loop** (more vertex w not in S)
    1. Find the node in **listOfDistanceNode** with vertexTo is w
    2. **if** (node.distance < minWeight)
      1. v = w
      2. minWeight = node.distance

DistanceNode

**vertexFrom** <VertexType>

**vertexTo** <VertexType>

**distance** <WeightType>

End DistanceNode

3. **if** (minWeight < infinity)

1. Add v to S.

2. tree.InsertVertex(v)

3. tree.InsertEdge(v,w)

4. **loop** (more vertex w not in S) *// Update distances from v to  
// all w not in S if they are smaller than the  
// previously recorded distances in listOfDistanceNode*

1. Find the node in **listOfDistanceNode** with vertexTo is w

2. **if** ( node.distance > weight of edge(v,w) )

1. node.vertexFrom = v

2. node.distance = weight of edge(v,w) )

3. Replace this node with its old node in **listOfDistance**

4. **else**

1. continue = FALSE

End MinimumSpanningTree

DistanceNode

**vertexFrom** <VertexType>

**vertexTo** <VertexType>

**distance** <WeightType>

End DistanceNode

# Maximum flows

- A network of water pipelines from **one source** to **one destination**.
- Water is pumped thru **many pipes** with **many stations** in between.
- The **amount of water** that can be pumped **may differ** from one pipeline to another.

# Maximum flows

- The flow thru a pipeline cannot be greater than its capacity.
- The total flow coming to a station is the same as the total flow coming from it.

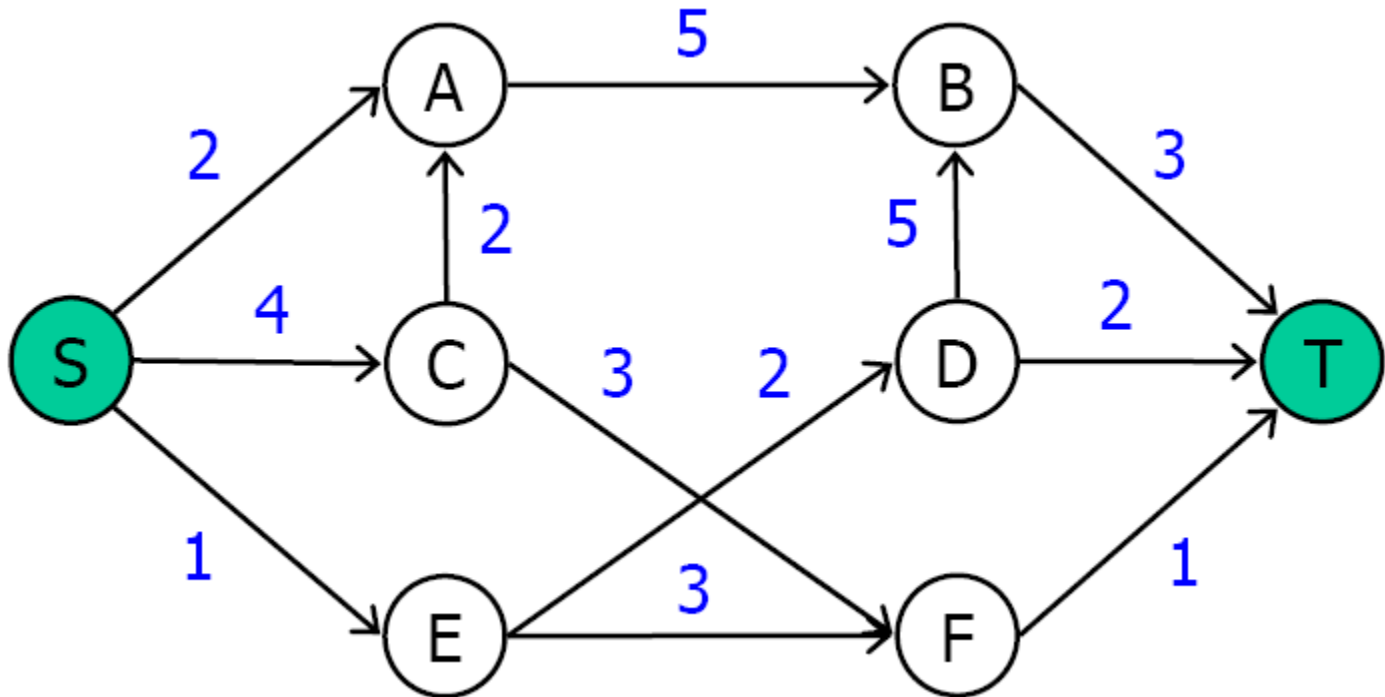


# Maximum flows

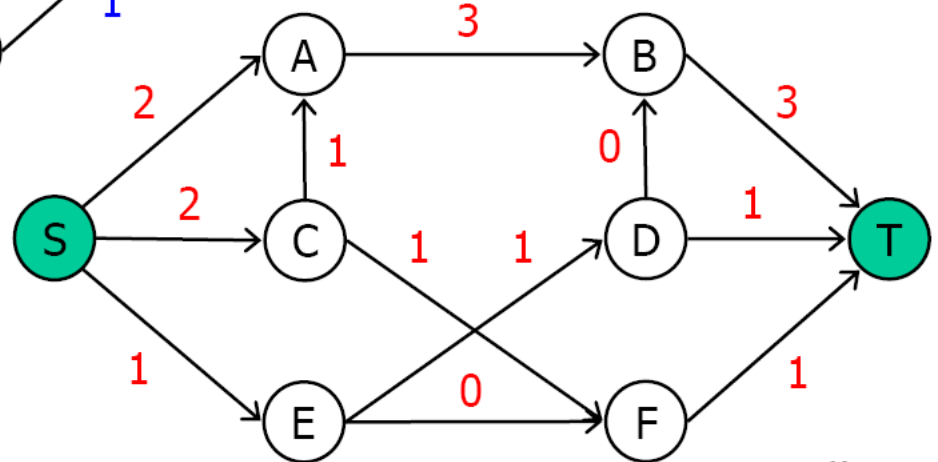
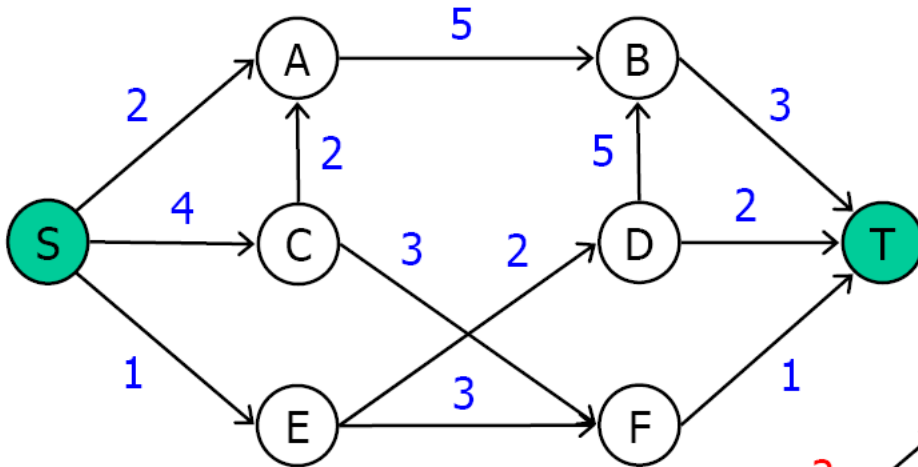
- The flow thru a pipeline cannot be greater than its capacity.
- The total flow coming to a station is the same as the total flow coming from it.

The problem is to maximize the total flow coming to the destination.

# Maximum flows



# Maximum flows



# Matching

- Applicants:        p            q            r            s            t
- Suitable jobs: a b c        b d        a e        e        c d e
- No applicant is accepted for two jobs, and no job is assigned to two applicants.

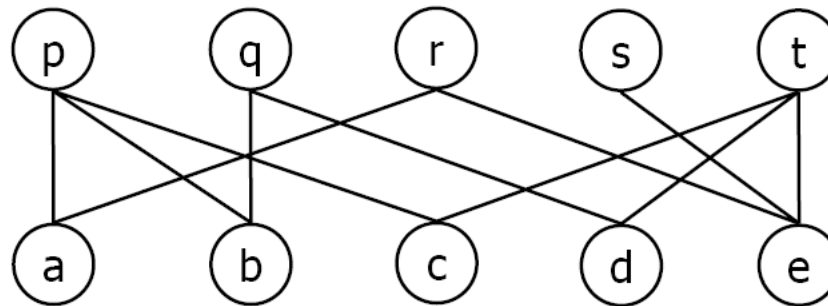
# Matching

- Applicants:        p            q            r            s            t
- Suitable jobs: a b c        b d        a e        e        c d e
- No applicant is accepted for two jobs, and no job is assigned to two applicants.

The problem is to find a worker for each job.

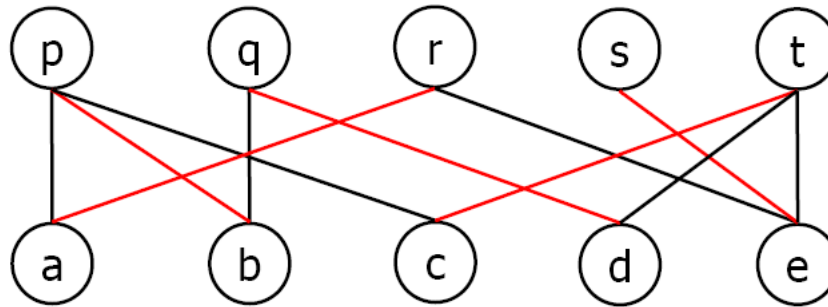
# Matching

- Applicants: p q r s t
- Suitable jobs: a b c b d a e e c d e



# Matching

- Applicants:      p            q            r            s            t
- Suitable jobs: a b c      b d      a e      e      c d e



# Matching

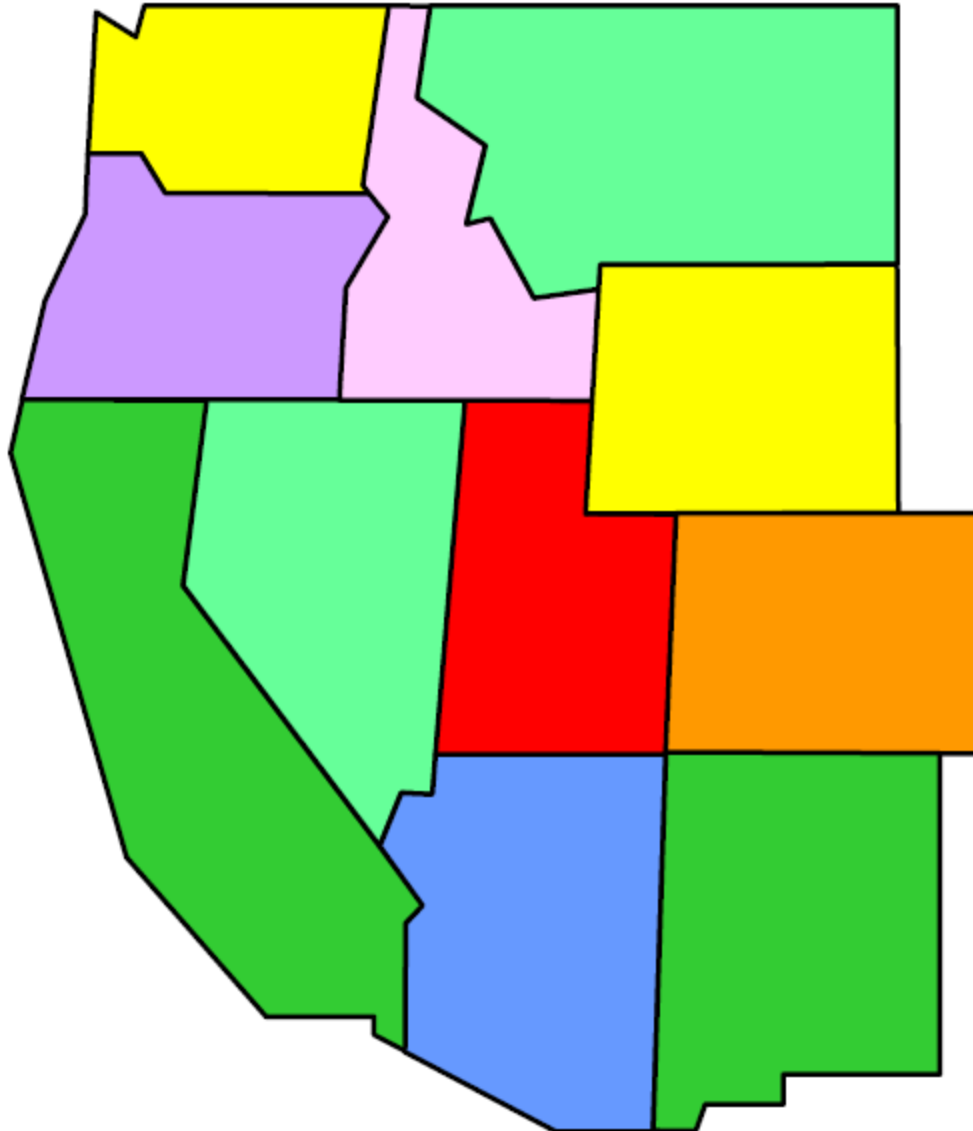
- **Maximum matching**: as many pairs of worker-job as possible.
- **Perfect matching (marriage problem)**: no worker or job left unmatched.



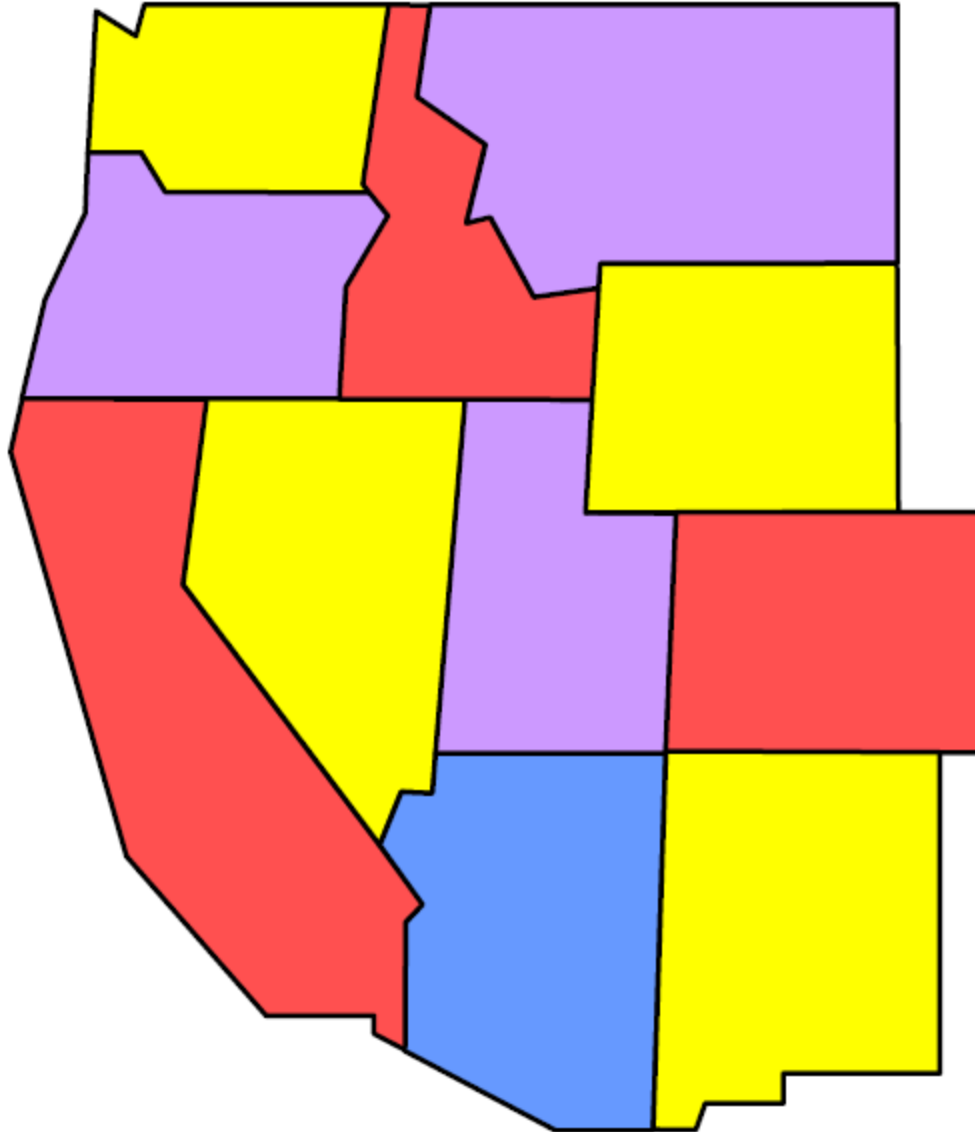
# Graph coloring

- Given a map of adjacent regions.
- Find the minimum number of colors to fill the regions so that no adjacent regions have the same color.

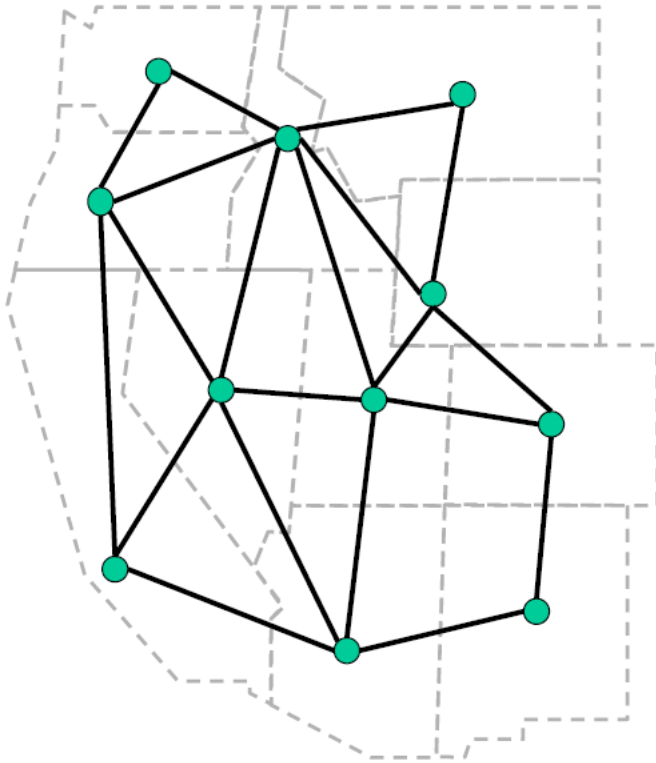
# Graph coloring



# Graph coloring

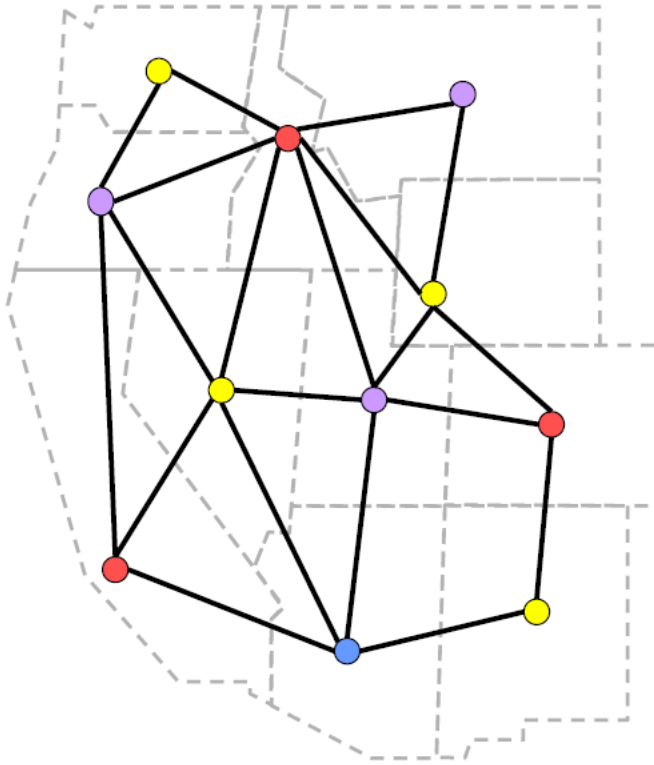


# Graph coloring



The problem is to find the **minimum number** of sets of non-adjacent vertices.

# Graph coloring



The problem is to find the **minimum number of sets of non-adjacent vertices**.