Chapter 8

Graphs

Objectives

Discuss the following topics:

- Graphs
- Graph Representation
- Graph Traversals
- Shortest Paths
- Cycle Detection
- Spanning Trees – Phủ cây
- Connectivity
- Eulerian and Hamiltonian Graphs
- Graph Coloring

Introduction to Graphs

- A **graph** is a collection of vertices (or nodes) and the connections between them
- A **simple graph** $G = (V, E)$ consists of a nonempty set V of **vertices** and a possibly empty set E of **edges**, each edge being a set of two vertices from V
- A **directed graph**, or a **digraph**, $G = (V, E)$ consists of a nonempty set V of vertices and a set E of edges (also called **arcs**), where each edge is a pair of vertices from V

Graphs (continued)

- A **multigraph** is a graph in which two vertices can be joined by multiple edges
- A **pseudograph** is a multigraph with the condition $v_i \neq v_j$ removed, which allows for loops to occur
- If all vertices in a circuit are different, then it is called a **cycle**
- A graph is called a **weighted graph** if each edge has an assigned number

Graphs (continued)

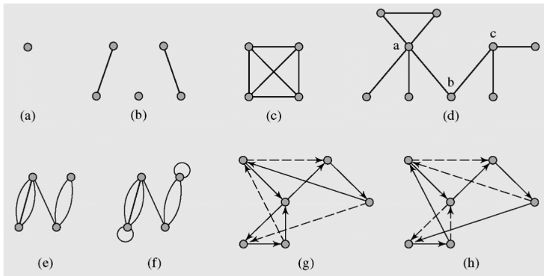


Figure 8-1 Examples of graphs: (a–d) simple graphs; (c) a complete graph K_4; (e) a multigraph; (f) a pseudograph; (g) a circuit in a digraph; (h) a cycle in the digraph

8.1- Graph Representation

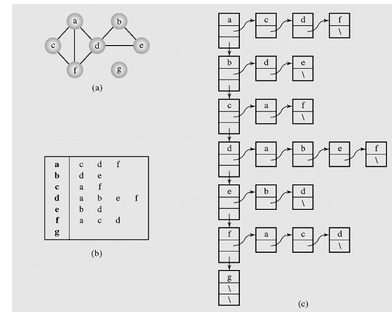


Figure 8-2 Graph representations (a) A graph represented as (b–c) an adjacency list

Graph Representation (continued)

	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

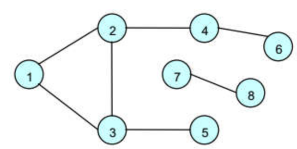
(d)

	ac	ad	af	bd	be	cf	de	df
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	0	1
g	0	0	0	0	0	0	0	0

(e)

Figure 8-2 Graph representations (d) an adjacency matrix, and (e) an incidence matrix (continued)

Graph Traversals



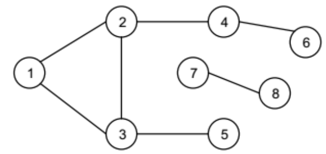
GRAPH.INP	GRAPH.OUT
8 7 1 5 1 2 1 3 2 3 2 4 3 5 4 6 7 8	1, 2, 3, 5, 4, 6, 5<-3<-2<-1

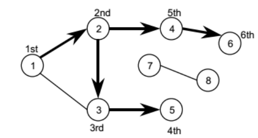
8.2- Graph Traversals

- In the **depth-first search algorithm**, each vertex *v* is visited and then each unvisited vertex adjacent to *v* is visited

```
void DFS (Vertex v)
    num(v) = i++;  // number  represents the order in traversal,  it marks that whether this vertex is visited  nor not
    for all vertices u adjacent to v
        if (num(u) is 0)
            attach edge uv to edges;
            DFS(u);
void depthFirstSearch()
    for all vertices v  num(v) = 0;
    edges = null;
    i = 1;
    while there is a vertex v such that num(v) is 0 DFS(v);
    output edges;
```


Example





Graph Traversals

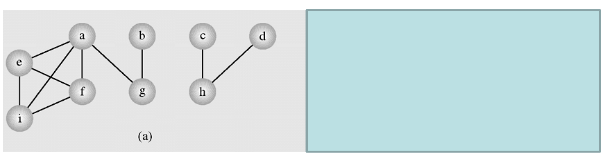


Figure 8-3 An example of application of the `depthFirstSearch()` algorithm to a graph

Graph Traversals (continued)

- Depth First Search guarantees generating a tree (or a forest, a set of trees) that includes or spans over all vertices of the original graph

Graph Traversals (continued)

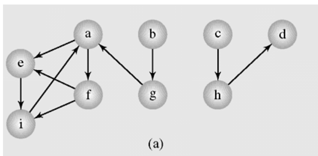
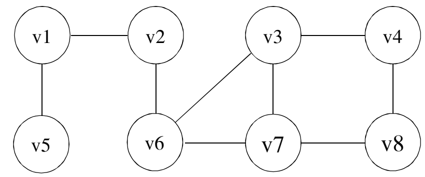


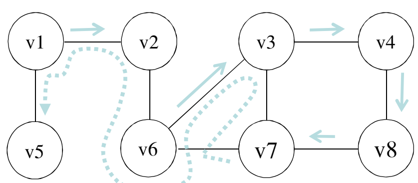
Figure 8-4 The `depthFirstSearch()` algorithm applied to a digraph

Example



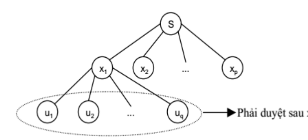
Solution

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
v1	v2	v6	v3	v4	v8	**v7**	v5



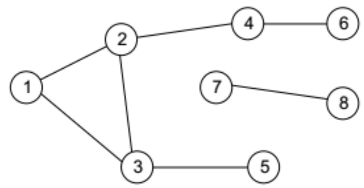
Graph Traversals (continued)

```
breadthFirstSearch()
    for all vertices v  num(v) = 0;
    edges = null;
    i = 1;
    while there is a vertex v such that num(v) is 0
        num(v) = i++;
        enqueue (v)
        while queue is not empty
            v=dequeue();
            for all vertices u adjecent to v
                if num(u) is 0
                    num(u) = i++;
                    enqueue(u);
```



Example

Xét đồ thị dưới đây, Đỉnh xuất phát S = 1.



Hàng đợi	**Đỉnh u (lấy ra từ hàng đợi)**	Hàng đợi (sau khi lấy u ra)	Các đỉnh v kề u mà chưa lên lịch	Hàng đợi sau khi đẩy những đỉnh v vào
(1)	**1**	∅	2, 3	(2, 3)
(2, 3)	**2**	(3)	4	(3, 4)
(3, 4)	**3**	(4)	5	(4, 5)
(4, 5)	**4**	(5)	6	(5, 6)
(5, 6)	**5**	(6)	Không có	(6)
(6)	**6**	∅	Không có	∅

Graph Traversals (continued)

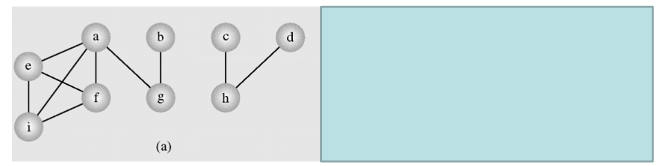


Figure 8-5 An example of application of the `breadthFirstSearch()` algorithm to a graph

Graph Traversals (continued)

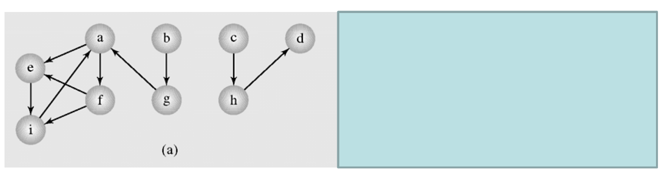
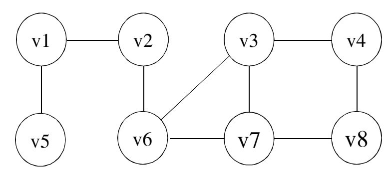
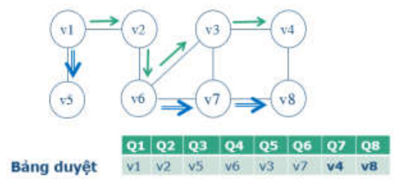


Figure 8-6 The `breadthFirstSearch()` algorithm applied to a digraph

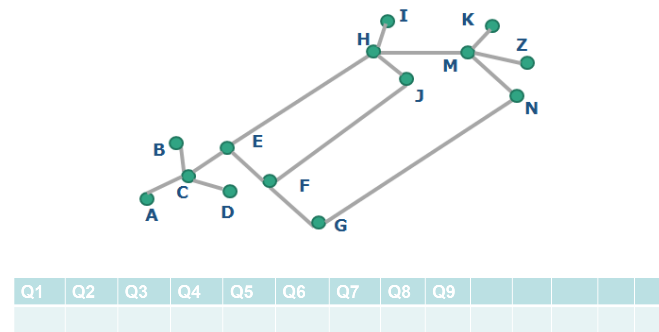
Example



Solution



Exercise



Algorithm Complexity

- Trong trường hợp ta biểu diễn đồ thị bằng danh sách kề, cả hai thuật toán BFS và DFS đều có độ phức tạp tính toán là O(n + m) = O(max(n, m)).

- Nếu ta biểu diễn đồ thị bằng ma trận kề như ở trên thì độ phức tạp tính toán trong trường hợp này là $O(n + n^2) = O(n^2)$.

- Nếu ta biểu diễn đồ thị bằng danh sách cạnh, thao tác duyệt những đỉnh kề với đỉnh u sẽ dẫn tới việc phải duyệt qua toàn bộ danh sách cạnh, đây là cài đặt tồi nhất, nó có độ phức tạp tính toán là O(n.m).

8.3- Shortest Paths

- The methods solving the shortest path problem are divided in two classes:
  - **Label-setting** methods
    - In each pass through the vertices still to be processed, one vertex is set to a value that remains unchanged to the end of the execution
  - **Label-correcting** methods
    - Allow for the changing of *any* label during application of the method
- **Negative cycle** is a cycle composed of edges with weights adding up to a negative number

Some Algorithm

- **Dijkstra**
- **Ford-Bellman**
- **Floyd-Warshall**

Shortest Paths

```
genericShortestPathAlgorithm (weighted simple digraph, vertex first)
  for all vertices v  currDist(v) = ∞
  currDist(first)=0;
  initialize toBeChecked; // set of vertices will be checked
  while (toBeChecked is not empty)
      v= a vertex in toBeChecked;
      remove v from toBeChecked;
      for all vertices u adjacent to v
          if currDist(u) > currDist(v) + weight(edge(vu))
              currDist(u) = currDist(v) + weight(edge(vu));
              predecessor(u)= v;
              Add u to toBeChecked if it is not there;
```

Dijkstra
toBeChecked= all vertices

Ford
Examine edges instead of vertices

Label(v) = (currDist(v), predecessor(v))
Two above things open will be specified by some authors.

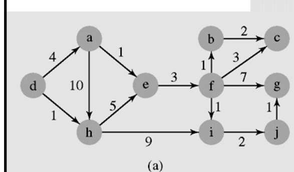
Shortest Paths (continued)

```
DijkstraAlgorithm(weighted simple digraph, vertex first)
   for all vertices v
      currDist(v) = ∞;
   currDist(first) = 0;
   toBeChecked = all vertices;
   while toBeChecked is not empty
      v = a vertex in toBeChecked with minimal currDist(v);
      remove v from toBeChecked;
      for all vertices u adjacent to v and in toBeChecked
         if currDist(u) > currDist(v)+ weight(edge(vu))
            currDist(u) = currDist(v)+ weight(edge(vu));
            predecessor(u) = v;
```

Use: Positive weight

Time complexity depends on the implementation:
- Can be $O(n^2+m)$, $O(m \log n)$, or $O(m+n \log n)$

Shortest Paths (continued)



iteration:	init	1	2	3	4	5	6	7	8	9	10
active vertex:		d	h	a	e	f	b	i	c	j	g
a	∞	4	4								
b	∞	∞	∞	∞	∞	9					
c	∞	∞	∞	∞	∞	11	11	11			
d	0										
e	∞	∞	6	5							
f	∞	∞	∞	∞	8						
g	∞	∞	∞	∞	∞	15	15	15	15	12	
h	∞	1									
i	∞	∞	10	10	10	9	9				
j	∞	∞	∞	∞	∞	∞	∞	11	11		

(b)

Figure 8-7 An execution of DijkstraAlgorithm() (continued)

Shortest Paths (continued)

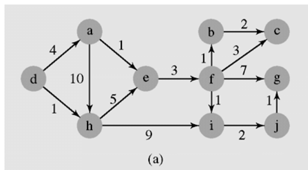
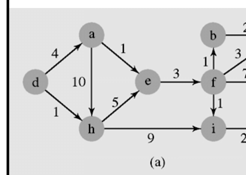


Figure 8-7 An execution of `DijkstraAlgorithm()`

Shortest Paths (continued)



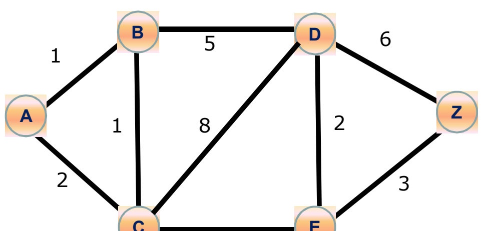
iteration:	init	1	2	3	4	5	6	7	8	9	10
active vertex:		d	h	a	e	f	b	i	c	j	g
a	∞	4	4								
b	∞	∞	∞	∞	∞	9					
c	∞	∞	∞	∞	∞	11	11	11			
d	0										
e	∞	∞	6	5							
f	∞	∞	∞	∞	8						
g	∞	∞	∞	∞	∞	15	15	15	15	12	
h	∞	1									
i	∞	∞	10	10	10	9	9				
j	∞	∞	∞	∞	∞	∞	∞	11	11		

(b)

Figure 8-7 An execution of `DijkstraAlgorithm()` (continued)

Bài tập 1

Tìm đường đi ngắn nhất từ A đến mỗi đỉnh khác của đồ thị G dưới đây



Đáp án

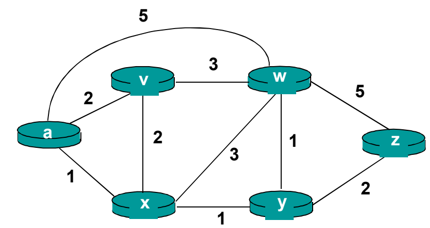
Bước	Tập T	A	B	C	D	E	Z
0 (initial)	A,B,C,D,E,Z	(0,-)	(∞, -)	(∞, -)	(∞, -)	(∞, -)	(∞, -)
1	B,C,D,E,Z	(0,-)*	(1,A)	(2,A)	(∞, -)	(∞, -)	(∞, -)
2	C,D,E,Z	-	(1,A)*	(2,A)	(6,B)	(∞, -)	(∞, -)
3	D,E,Z	-	-	(2,A)*	(6,B)	(12,C)	(∞, -)
4	E,Z	-	-	-	(6,B)*	(8,D)	(12,D)
5	Z	-	-	-	-	(8,D)*	(11,E)
6	№	-	-	-	-	-	(11,E)*

Đáp án

Bước	Tập T	a	b	d	c	e	z
0	ø	(0,-)	(∞,-)	(∞,-)	(∞,-)	(∞,-)	(∞,-)
1	a	(0,-)*	(1,a)	(∞,-)	(2,a)	(∞,-)	(∞,-)
2	ba	-	(1,a)*	(6,b)	(2,a)	(∞,-)	(∞,-)
3	cba	-	-	(6,b)	(2,a) *	(12,c)	(∞,-)
4	dcba	-	-	(6,b)*	-	(8,d)	(12,d)
5	edcba	-	-	-	-	(8,d)*	(11,e)
6	zedcba	-	-	-	-	-	(11,e)*

Bài tập 2

- Tìm đường đi ngắn nhất giữa A và W trong đồ thị G dưới đây

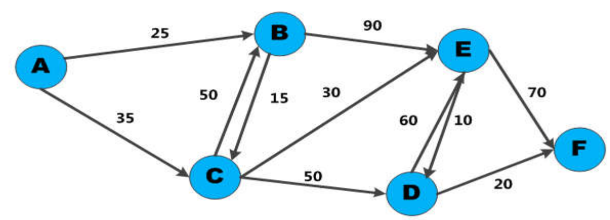


Đáp án

Bước	Tập T	a	v	x	y	w	z
0	ø	(0.-)	(∞,-)	(∞,-)	(∞,-)	(∞,-)	(∞,-)
1	a	(0,-)*	(2,a)	(1,a)	(∞,-)	(5,a)	(∞,-)
2	xa	-	(2,a)	(1,a)*	(2,x)	(4,x)	(∞,-)
3	yxa	-	(2,a)	-	(2,x)*	(3,y)	(4,y)
4	vyxa	-	(2,a)*	-	-	(3,y)	(4,y)
5	wvyxa	-	-	-	-	(3,y)*	(4,y)
6	Kết thúc vì đỉnh đến w chứa trong tập T						

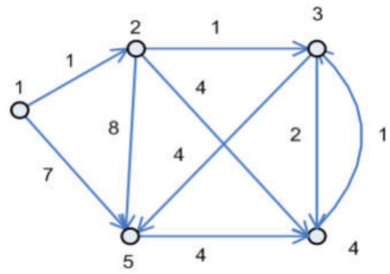
Bài tập 3

Use Dijkstra's algorithm to find shortest paths from vertex C to others



What is "current distance of A" after executing of the algorithm end?

Bài 3.1

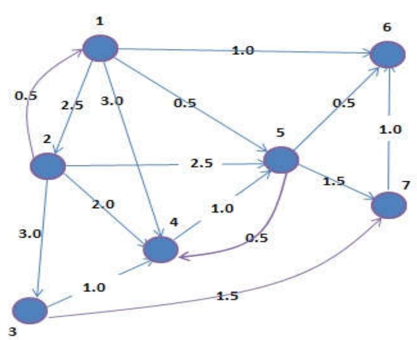


Đáp án

T	Đỉnh 2	Đỉnh 3	Đỉnh 4	Đỉnh 5
2, 3, 4, 5	1, 1	∞,1	∞, 1	7, 1
3, 4, 5		2, 2	5, 2	7, 1
4, 5			4, 3	6, 3
E				6, 3
∅	1, 1	2, 2	4, 3	6, 3

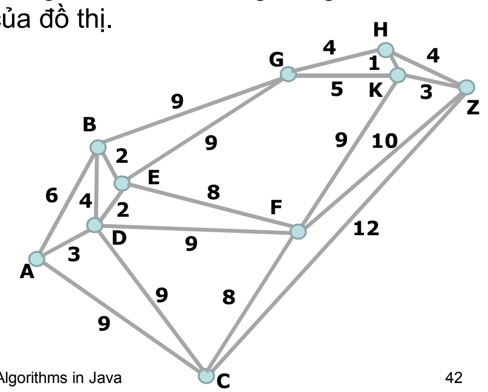
Bài tập 4

Use Dijkstra's algorithm to find the shortest path from vertex 1 to others.



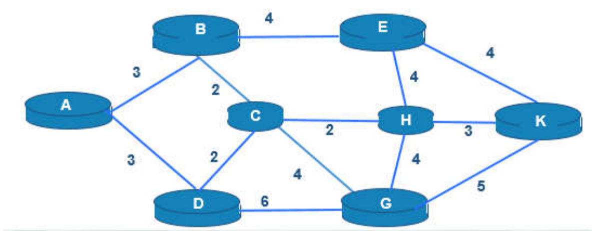
Bài tập 5

Cho đồ thị có trọng số G.Tìm đường đi ngắn nhất từ A đến mỗi đỉnh của đồ thị.



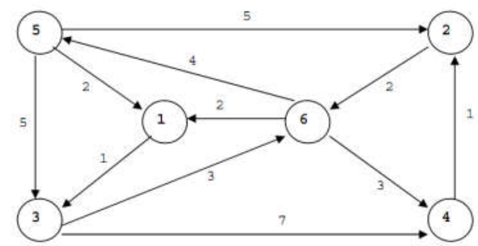
Bài tập 6

- Cho đồ thị có trọng số G = (V, E), tìm đường đi ngắn nhất giữa A và H.



Bài tập 7

Find the shortest path from the vertex 5 to the vertex 4



Bài tập 8

Ví dụ 5:

Tìm đường đi ngắn nhất từ v1 đến các đỉnh khác của đồ thị có trọng số được biểu diễn trong ma trận M hình bên.

	v1	v2	v3	v4	v5	v6	v7
v1	0	5	30	40	∞	∞	∞
v2	∞	0	27	∞	73	∞	∞
v3	∞	26	0	8	40	25	38
v4	∞	∞	∞	0	∞	16	20
v5	∞	70	∞	∞	0	20	12
v6	∞	∞	22	∞	22	12	∞
v7	50	∞	∞	∞	10	∞	∞

Ford-Bellman

```
// Step 1: initialize graph
for each vertex v in vertices:
    distance[v] := inf             // At the beginning , all vertices have a weight of infinity
    predecessor[v] := null         // And a null predecessor

distance[source] := 0              // Except for the Source, where the Weight is zero
```

```
// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
    for each edge (u, v) with weight w in edges:
        if distance[u] + w < distance[v]:
            distance[v] := distance[u] + w
            predecessor[v] := u
```

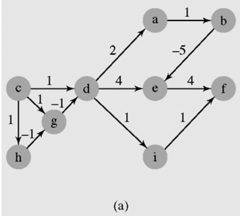
```
// Step 3: check for negative-weight cycles
for each edge (u, v) with weight w in edges:
    if distance[u] + w < distance[v]:
        error "Graph contains a negative-weight cycle"
return distance[], predecessor[]
```

Sử dụng khi:

1. **Cạnh có trọng lượng âm**
2. **Đồ thị không có chu trình âm**
3. **O(E*V)**

4. → Có thể dùng Bellman-Ford để kiểm tra tồn tại chu trình âm trong đồ thị

Shortest Paths (continued)



the order of edges: *ab be cd cg ch da de di ef gd hg if*

		iteration				
	init	1		2	3	4
a	∞	3		2	1	
b	∞			4	3	2
c	0					
d	∞	1	0	−1		
e	∞	5		−1	−2	−3
f	∞	9	3	2	1	
g	∞	1	0			
h	∞	1				
i	∞	2		1	0	

(b)

-Adjacency list
-All edges are checked every time → Drawback

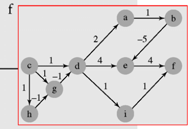
Figure 8-8 `FordAlgorithm()` **applied to a digraph with negative weights**

Shortest Paths (continued)

Re-visit is permited

active vertex

		c	d	g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e
queue		d	g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e	
		g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e		
		h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f				
			e	i	d	g	b	f	a	e	i	d	b			i	e							
			i	d	g	b	f		e	i	d													
									i	d														
a	∞	∞	3	3	3	3	3	3	2	2	2	2	2	2	2	1								
b	∞	∞	∞	∞	∞	4	4	4	4	4	4	4	3	3	3	3	3	3	2					
c	0																							
d	∞	1	1	0	0	0	0	0	0	−1														
e	∞	∞	5	5	5	5	5	5	4	4	−1	−1	−1	−1	−1	−1	−2	−2	−2	−2	−2	−3		
f	∞	∞	∞	∞	∞	∞	9	3	3	3	3	3	3	3	2	2	2	2	2	1				
g	∞	1	1	1	0																			
h	∞	1																						
i	∞	∞	2	2	2	2	2	2	1	1	1	1	1	1	1	0								



Only adjacence vertices of active nertex are checked → Advantage

Figure 8-9 An execution of `labelCorrectingAlgorithm()`**, which uses a queue**

Shortest Paths (continued)

Re-visit is not permited

active vertex

		c	d	g	d	h	g	d	a	e	i	b	e	f
deque		d	g	d	h	g	d	a	e	i	b	e	f	
		g	h	h	a	a	a	e	i	b	f	f		
		h	a	a	e	e	e	i	b	f				
			e	e	i	i	i							
			i	i										
a	∞	∞	3	3	2	2	2	1						
b	∞	∞	∞	∞	∞	∞	∞	∞	2					
c	0													
d	∞	1	1	0	0	0	−1							
e	∞	∞	5	5	4	4	4	3	3	3	3	−3		
f	∞	∞	∞	∞	∞	∞	∞	∞	∞	7	1			
g	∞	1	1	1	1	0								
h	∞	1												
i	∞	∞	2	2	1	1	1	0						

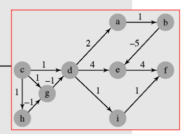
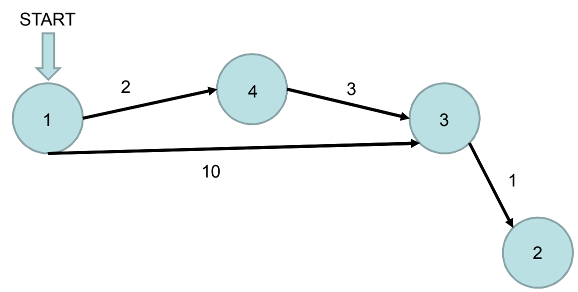


Figure 8-10 An execution of `labelCorrectingAlgorithm()`**, which applies a deque**

Simple Exercise



Solution-case-1 (Chú ý cách xếp trong tập E)

K	1	2	3	4
0	(0,-)	(∞,-)	(∞,-)	(∞,-)
1	(0,-)	(∞,-)	(10,1)	(2,1)
2	(0,-)	(11,3)	(5,4)	(2,1)
3	(0,-)	(6,3)	(5,4)	(2,1)

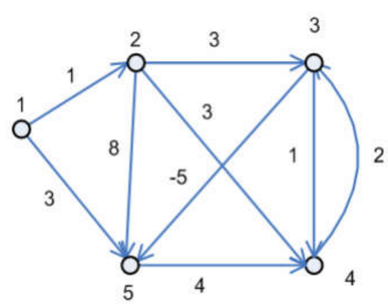
Tập hợp E(Edges) = {(1,4) = 2, (1,3) = 10, **(3,2) = 1, (4,3) = 3**}

Solution-case-2 (Chú ý cách xếp trong tập E)

K	1	2	3	4
0	(0,-)	(∞,-)	(∞,-)	(∞,-)
1	(0,-)	(∞,-)	(10,1)	(2,1)
2	(0,-)	(6,3)	(5,4)	(2,1)
3	(0,-)	(6,3)	(5,4)	(2,1)

Tập hợp E(Edges) = {(1,4) = 2, (1,3) = 10, **(4,3) = 3** , **(3,2) = 1**}

Bài 1

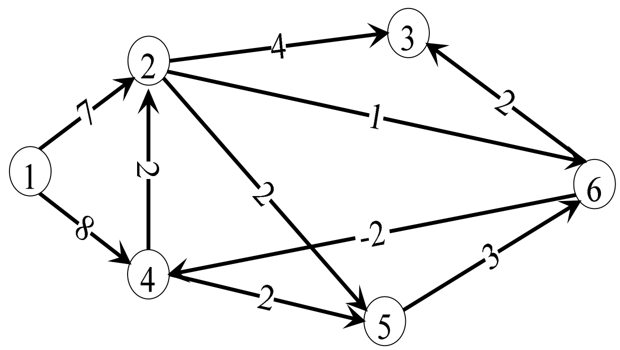


Đáp án

k	**1**	**2**	**3**	**4**	**5**
0 (initial)	**(0,-)**	**(∞,-)**	**(∞,-)**	**(∞,-)**	**(∞,-)**
1	**(0,-)**	**(1,1)**	**(∞,-)**	**(∞,-)**	**(3,1)**
2	**(0,-)**	**(1,1)**	**(4,2)**	**(4,2)**	**(3,1)**
3	**(0,-)**	**(1,1)**	**(4,2)**	**(4,2)**	**(-1,3)**
4	**(0,-)**	**(1,1)**	**(4,2)**	**(3,5)**	**(-1,3)**
5	**(0,-)**	**(1,1)**	**(4,2)**	**(3,5)**	**(-1,3)**

Tập hợp cạnh
1->2: 1
1->5: 3
……

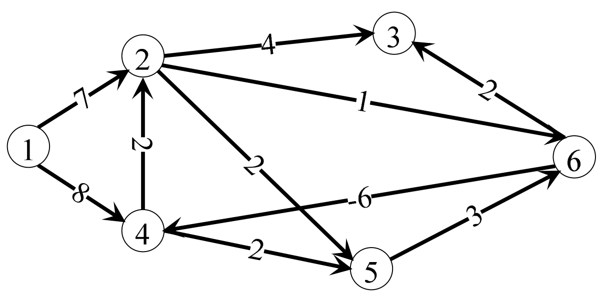
Bài tập 2



Đáp án

k	1	2	3	4	5	6
0	(0,1)	(∞,-1)	(∞,-1)	(∞,-1)	(∞,-1)	(∞,-1)
1	(0,1)	(7,1)	(∞,-1)	(8,1)	(∞,-1)	(∞,-1)
2	(0,1)	(7,1)	(11,2)	(8,1)	(9,2)	(8,2)
3	(0,1)	(7,1)	(10,6)	(6,6)	(9,2)	(8,2)
4	(0,1)	(7,1)	(10,6)	(6,6)	(8,4)	(8,2)
5	(0,1)	(7,1)	(10,6)	(6,6)	(8,4)	(8,2)

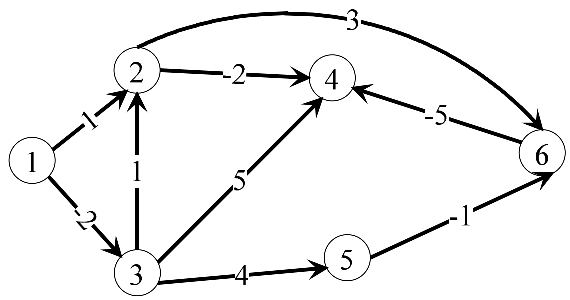
Bài tập 3



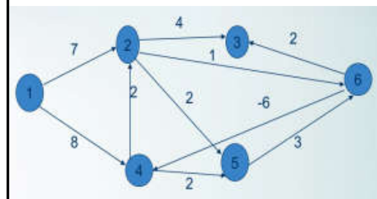
Đáp án

k	1	2	3	4	5	6
0	(0,1)	(∞,-1)	(∞,-1)	(∞,-1)	(∞,-1)	(∞,-1)
1	(0,1)	(7,1)	(∞,-1)	(8,1)	(∞,-1)	(∞,-1)
2	(0,1)	(7,1)	(11,2)	(8,1)	(9,2)	(8,2)
3	(0,1)	(7,1)	(10,6)	(2,6)	(9,2)	(8,2)
4	(0,1)	(4,4)	(10,6)	(2,6)	(4,4)	(8,2)
5	(0,1)	(4,4)	(8,2)	(2,6)	(4,4)	(5,2)
6	(0,1)	(4,4)	(7,6)	(-1,6)	(4,4)	(5,2)

Bài tập 4



Ví dụ kiểm tra mạch âm (Ford)

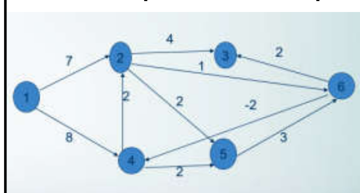


$k = n = 6$. $L_k(i)$ chưa ổn định nên đồ thị có mạch âm. Chẳng hạn:

4→2→6→4 có độ dài -3

k	1	2	3	4	5	6
0	0	(∞,-)	(∞,-)	(∞,-)	(∞,-)	(∞,-)
1	0	(7,1)	(∞,-)	(8,1)	(∞,-)	(∞,-)
2	0	(7,1)	(11,2)	(8,1)	(9,2)	(8,2)
3	0	(7,1)	(10,6)	(2,6)	(9,2)	(8,2)
4	0	(4,4)	(10,6)	(2,6)	(4,4)	(8,2)
5	0	(4,4)	(8,2)	(2,6)	(4,4)	(5,2)
6	0	(4,4)	(7,6)	(-1,6)	(4,4)	(5,2)

Ví dụ kiểm tra mạch âm (Bellman-Ford)



k <= n (n=6)

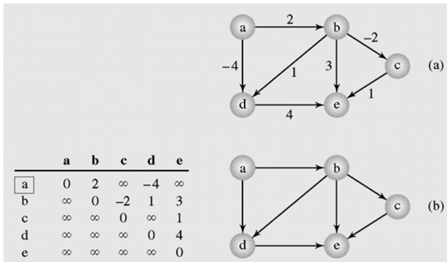
→ Mạch đã ổn định nên không có chu trình âm

k	1	2	3	4	5	6
0	0	(∞,-)	(∞,-)	(∞,-)	(∞,-)	(∞,-)
1	0	(7,1)	(∞,-)	(8,1)	(∞,-)	(∞,-)
2	0	(7,1)	(11,2)	(8,1)	(9,2)	(8,2)
3	0	(7,1)	(10,6)	(6,6)	(9,2)	(8,2)
4	0	(7,1)	(10,6)	(6,6)	(8,4)	(8,2)
5	0	(7,1)	(10,6)	(6,6)	(8,4)	(8,2)

All-to-All Shortest Path Problem Algorithm

```
WFIalgorithm(matrix weight)
  for i = 1 to |V|
    for j = 1 to |V|
      for k = 1 to |V|
        if (weight[j][k] > weight[j][i] + weight[i][k])
            weight[j][k] = weight[j][i] + weight[i][k];
```

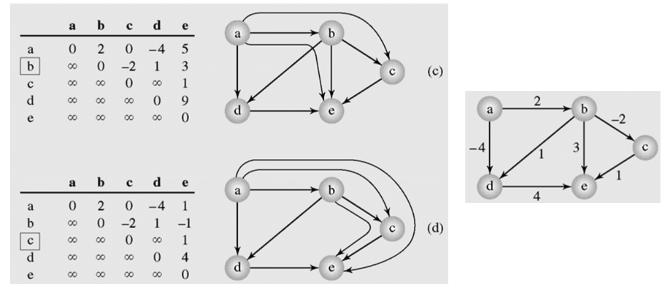
All-to-All Shortest Path Problem



	a	b	c	d	e
$\boxed{a}$	0	2	∞	−4	∞
b	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

Figure 8-11 An execution of `WFIalgorithm()`

All-to-All Shortest Path Problem (continued)

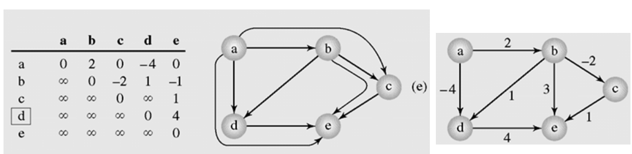


	a	b	c	d	e
a	0	2	0	−4	5
$\boxed{b}$	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	9
e	∞	∞	∞	∞	0

	a	b	c	d	e
a	0	2	0	−4	1
b	∞	0	−2	1	−1
$\boxed{c}$	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

Figure 8-11 An execution of `WFIalgorithm()` (continued)

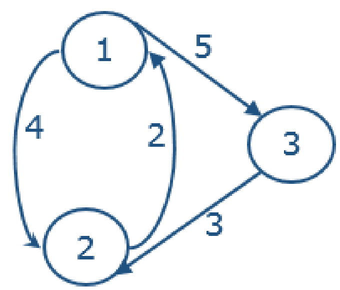
All-to-All Shortest Path Problem (continued)



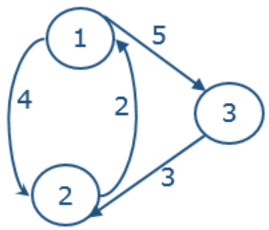
	a	b	c	d	e
a	0	2	0	−4	0
b	∞	0	−2	1	−1
c	∞	∞	0	∞	1
$\boxed{d}$	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

Figure 8-11 An execution of `WFIalgorithm()` (continued)

Bài tập



Thuật toán Floyd (Loop initial: k=0)



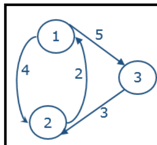
D^0 =

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	3	0

P^0=

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

Loop k=1



D^1 =

	1	2	3
1	0	4	5
2	2	0	7
3	∞	3	0

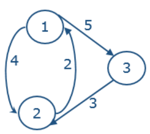
$D^1[2,3] = \min(D^0[2,3], D^0[2,1]+D^0[1,3])$
$= \min (\propto, 7)$
$= 7$
$P[2,3]=1$

P^1=

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

$D^1[3,2] = \min(D^0[3,2], D^0[3,1]+D^0[1,2])$
$= \min (3,\propto)$
$= 3$

Loop k = 2



D^2 =

	1	2	3
1	0	4	5
2	2	0	7
3	5	3	0

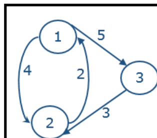
$D^2[1,3] = \min(D^1[1,3], D^1[1,2]+D^1[2,3])$
$= \min (5, 4+7)$
$= 5$

P^2 =

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

$D^2[3,1] = \min(D^1[3,1], D^1[3,2]+D^1[2,1])$
$= \min (\propto, 3+2)$
$= 5$
$P[3,1] = 2$

Loop k = 3



D^3 =

	1	2	3
1	0	4	5
2	2	0	7
3	5	3	0

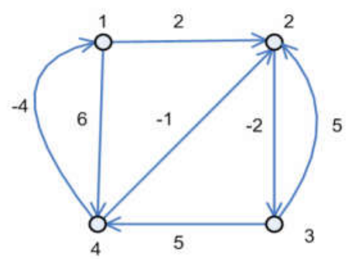
$D^3[1,2] = \min(D^2[1,2], D^2[1,3]+D^2[3,2])$
$= \min (4, 5+3))$
$= 4$

P^3=

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

$D^3[2,1] = \min(D^2[2,1], D^2[2,3]+D^2[3,1])$
$= \min (2, 7+ 5)$
$= 2$

Bài tập 2



Đáp án

- Khởi tạo hai ma trận:

D: Ma trận lưu giá trị đường đi.
P: Ma trận lưu vết.

D0

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

P0

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

Kết quả các bước lặp của thuật toán:

- k=1

D1

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

P1

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

- k=2

D2

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

P2

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

- k=3

D3

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

P3

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

- k=4

D4

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

P4

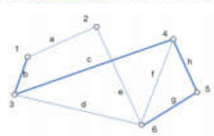
	1	2	3	4
1	0	1	2	3
2	4	0	2	3
3	4	1	0	3
4	4	1	2	0

Vậy đường đi ngắn nhất từ đỉnh 1 đến đỉnh 3 là: 1 → 2 → 3.
Với trọng số = 0.

IV. Đường đi, chu trình

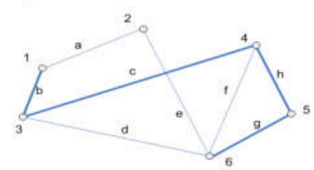
- Đường đi độ dài n từ đỉnh u đến đỉnh v trên đồ thị vô hướng G=(V,E) là dãy(theo đỉnh): $x_0, x_1, \ldots, x_{n-1}, x_n$.
  Trong đó:
  + $u = x_0$
  + $v = x_n$
  + $(x_i, x_{i+1}) \in E$
- Hay theo cạnh: $(x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n)$.
- Khi đó: u gọi là đỉnh đầu, v gọi là đỉnh cuối của đường đi.

Theo đỉnh: (1, 3, 4, 5, 6)
Theo cạnh: (b, c, h, g)



IV. Đường đi, chu trình

- Đường đi có đỉnh đầu và đỉnh cuối trùng nhau gọi là chu trình.
  - Đường đi (hay chu trình) được gọi là đơn nếu nó không đi qua một cạnh nào quá một lần.



Chu trình đơn: (1, 2, 6, 3, 1)
Chu trình không phải chu trình đơn: (2, 6, 4, 3, 6, 2)

8.4- Cycle Detection

- Depth-First Search → Cycle Detection

Nhắc lại: DFS (Vertex v)

```
num(v) = i++;
for all vertices u adjacent to v
    if (num(u) is 0)
        attach edge uv to edges;
        DFS(u);
```

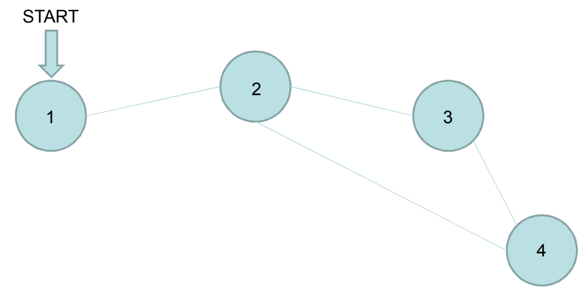
cycleDetectionDFS (Vertex v)

```
num(v) = i++;
for all vertices u adjacent to v
    if (num(u) is 0)
        attach edge uv to edges;
        cycleDetectionDFS(u);
    else if edge(uv) is not in edges cycle detected
```

digraphCycleDetectionDFS (Vertex v)

```
num(v) = i++;
for all vertices u adjacent to v
    if (num(u) is 0)
        pred (u)= v;
        digraphCycleDetectionDFS(u);
    else if num(u) is not ∞
        pred (u) = v;
        cycle detected;
num(v) = ∞;
```


Simple Exercise



Union-Find Problem

- The task is to determine if two vertices are in the same set by:
  - Finding the set to which a vertex *v* belongs
  - Uniting the two sets into one if vertex *v* belongs to one of them and *w* to another
- The sets used to solve the union-find problem are implemented with circular linked lists
- Each list is identified by a vertex that is the root of the tree to which the vertices in the list belong

Union-Find Problem (continued)

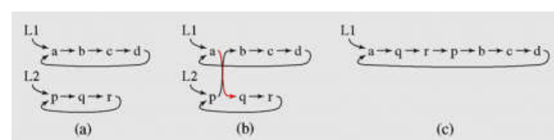


Figure 8-12 Concatenating two circular linked lists

Union-Find Problem (continued)

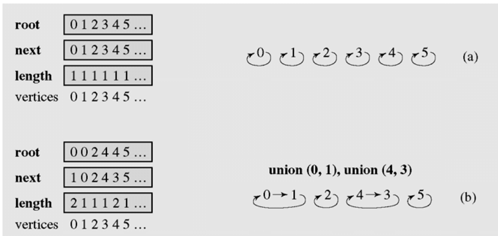


Figure 8-13 An example of application of `union()` to merge lists

Union-Find Problem (continued)

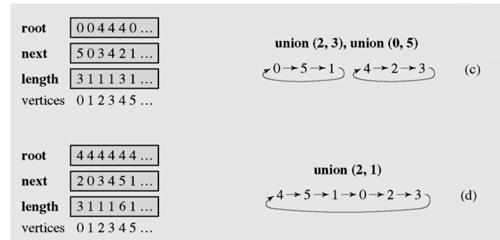
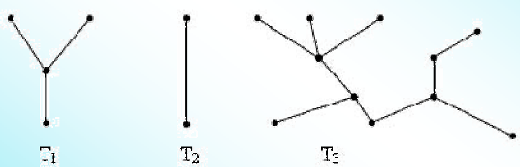


Figure 8-13 An example of application of `union()` to merge lists (continued)

End of this session

Cây

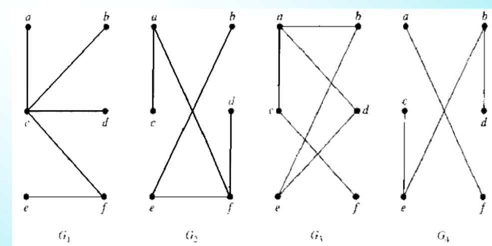
- **Định nghĩa:** **Cây** là một đơn đồ thị **vô hướng**, **liên thông** và **không chứa chu trình**.
- **Ví dụ:** Trong các đồ thị sau, đồ thị nào là cây?



- Cả 3 đồ thị trên đều là cây.

Cây (tt)

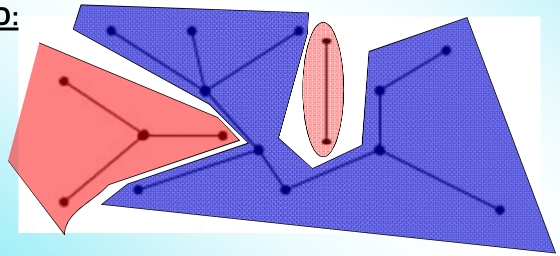
- **VD:** Trong các đồ thị sau, đồ thị nào là cây?



- G1, G2 là cây. G3, G4 không là cây do có chứa chu trình

Cây (tt)

- **Định nghĩa:** Nếu G là một đồ thị vô hướng và không chứa chu trình thì G được gọi là một **rừng**. Khi đó mỗi thành phần liên thông của G sẽ là một cây.
- **VD:**



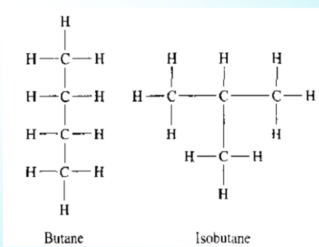
- Đồ thị trên là rừng có 3 cây

Tính chất của cây

- **Định lý:** Cho T là một đồ thị vô hướng. Khi đó, các điều sau đây là tương đương:
  1. T là cây.
  2. **T không chứa chu trình và có n – 1 cạnh.**
  3. T liên thông và có n – 1 cạnh.
  4. T liên thông và mỗi cạnh của T đều là cạnh cắt (cầu).
  5. Hai đỉnh bất kỳ của T được nối với nhau bằng đúng 1 đường đi đơn.
  6. T không chứa chu trình nhưng nếu thêm 1 cạnh bất kỳ vào T thì ta sẽ được thêm đúng 1 chu trình.

Các mô hình dạng cây

- Các Hydrocarbon no:



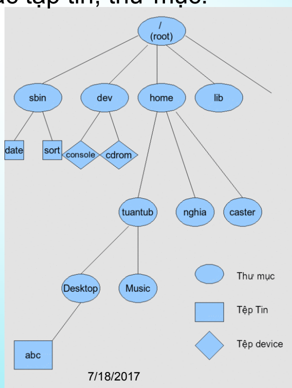
Hai đồng phân của Butane

Các mô hình dạng cây (tt)

- Biểu diễn các tổ chức:

Các mô hình dạng cây (tt)

- Hệ thống các tập tin, thư mục:



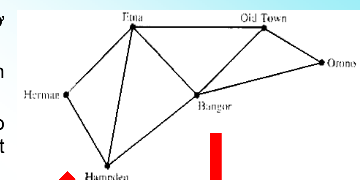
Các ứng dụng của cây

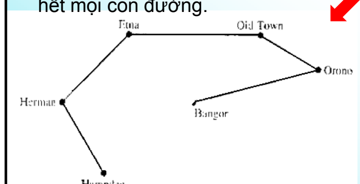
- Cây nhị phân tìm kiếm (đã học trong môn CTDL)
- Cây quyết định.
  - Là cây có gốc
  - Mỗi đỉnh ứng với một quyết định
  - Mỗi cây con tại đỉnh này sẽ ứng với các kết quả có thể của quyết định đó
- Mã tiền tố Huffman. (đề tài nghiên cứu)

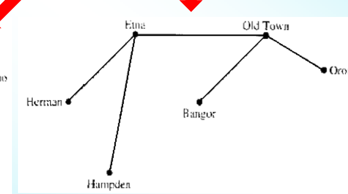
Cây khung
(Spanning Tree)

Bài toán mở đầu

- Hệ thống đường giao thông ở Maine như hình bên.
- Tuyết đang phủ toàn bộ các con đường.
- Cần khôi phục lại hệ thống giao thông bằng cách cào tuyết một số con đường.
- Không nhất thiết phải cào tuyết hết mọi con đường.

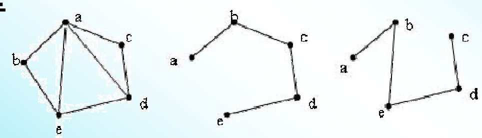






Cây khung

- **Định nghĩa:** Cho G là đơn đồ thị. Một cây T được gọi là cây khung của G nếu và chỉ nếu:
  - T là đồ thị con của G
  - T chứa tất cả các đỉnh của G
- **VD:**



Đồ thị và các cây khung của nó

Graphs- Part 2

8.5- Spanning Trees- Cây phủ

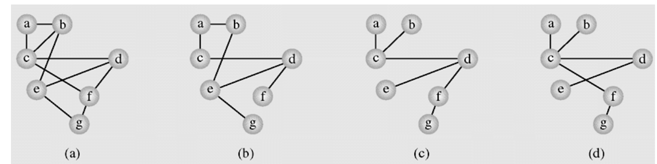
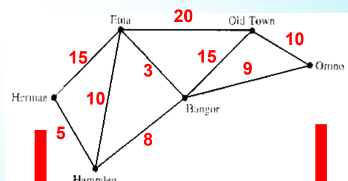
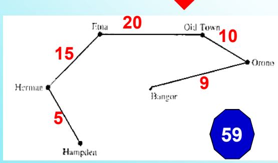


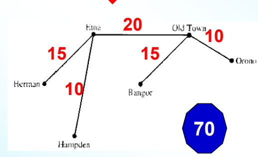
Figure 8-14 A graph representing (a) the airline connections between seven cities and (b–d) three possible sets of connections

Bài toán cây khung nhỏ nhất

- Tìm các con đường để cào tuyết sao cho chi phí là nhỏ nhất







Bài toán cây khung nhỏ nhất

MST: *Minimum Span Tree*

- Ứng dụng
  - Bài toán xây dựng hệ thống đường sắt
  - Bài toán nối mạng máy tính

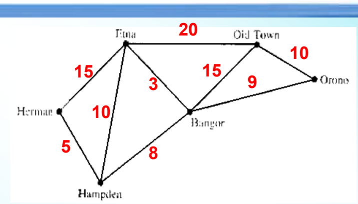
Bài toán cây khung nhỏ nhất (tt)

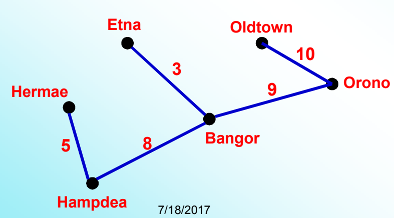
- **Định nghĩa.** Cho đồ thị có trọng số G. Cây khung nhỏ nhất của G (nếu tồn tại) là cây khung có tổng trọng số nhỏ nhất trong số các cây khung của G.
- **Các thuật toán tìm cây khung nhỏ nhất:**
  - Thuật toán Kruskal
  - Thuật toán Prim

Thuật toán Kruskal

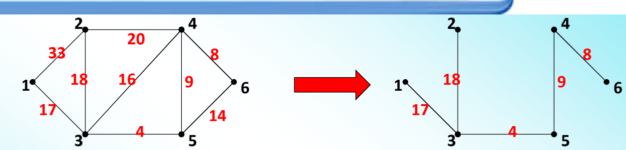
- **Ý tưởng:**
  - Lần lượt xét các cạnh theo thứ tự trọng số tăng dần
  - Ứng với mỗi cạnh đang xét, ta thử đưa nó vào cây khung T:
    - Nếu không tạo thành chu trình với các cạnh đã chọn thì chấp nhận cạnh mới này và đưa vào cây.
    - Nếu tạo thành chu trình với các cạnh đã chọn thì bỏ qua và xét cạnh kế tiếp.
  - Cứ tiếp tục như vậy cho đến khi tìm đủ n-1 cạnh để đưa vào cây T

Thuật toán Kruskal (tt)





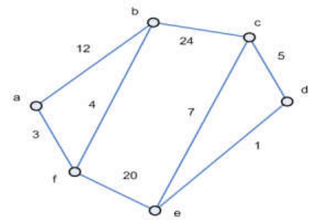
Thuật toán Kruskal (tt)



Trọng số	Cạnh	
4	(3,5)	Chọn
8	(4,6)	Chọn
9	(4,5)	Chọn
14	(5,6)	Không chọn vì tạo chu trình: 4 5 6 4
16	(3,4)	Không chọn vì tạo chu trình: 3 4 5 3
17	(1,3)	Chọn
18	(2,3)	Chọn. Dừng vì đã đủ cạnh.
20	(2,4)	
33	(1,2)	

IV.2. Thuật toán Kruskal

- Ví dụ

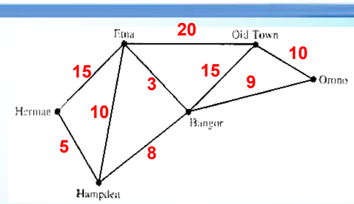


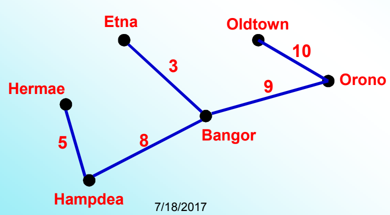
(d, e)	(a, f)	(b, f)	(c, d)	(c, e)	(a, b)	(f, e)	(b, c)
1	3	4	5	7	12	20	24

Thuật toán Prim

- Ý tưởng:
  - Xuất phát từ 1 đỉnh bất kỳ. Đưa đỉnh này vào cây khung T.
  - Tại mỗi bước, luôn chọn cạnh có trọng số nhỏ nhất trong số các cạnh liên thuộc với một đỉnh trong T (đỉnh còn lại nằm ngoài T)
  - Đưa cạnh mới chọn và đỉnh đầu của nó vào cây T
  - Lặp lại quá trình trên cho đến khi đưa đủ n-1 cạnh vào T

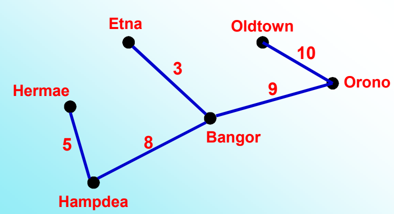
Thuật toán Prim (tt)





Thuật toán Prim (tt)

- Để biểu diễn lời giải, ta sẽ sử dụng 2 mảng:
  - Mảng d: d[v] dùng để lưu độ dài cạnh ngắn nhất nối với v trong số các cạnh chưa xét.
  - Mảng near: near[v] dùng để lưu đỉnh còn lại (ngoài v) của cạnh ngắn nhất nói ở trên.



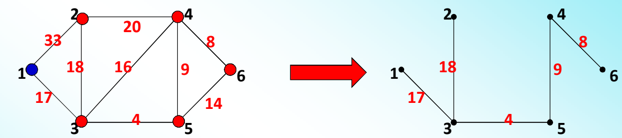
v	d[v]	near[v]

Thuật toán Prim (tt)

(* Khởi tạo *)
Chọn s là một đỉnh nào đó của đồ thị
V_H := {s}; (* Tập những đỉnh đã đưa vào cây *)
T := ∅; (* Tập cạnh của cây *)
d[s] = 0; near[s] = s;
For $v \in V \setminus V_H$ **do**
Begin
d[v] := a[s,v];
near[v] := s;
End;

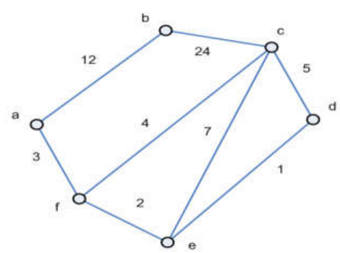
(* Bước lặp *)
Stop := False;
While (not Stop) **do**
Begin
Tìm $u \in V \setminus V_H$ thỏa mãn d[u] = min{d[v]: $v \in V \setminus V_H$};
$V_H := V_H \cup \{u\}$;
T := T ∪ { (u, near[u]) };
If $|V_H|$ = n **then**
Begin
H := (V_H, T) là cây khung của đồ thị.
Stop := True;
End;
Else
For v $\in V \setminus V_H$ **do**
If d[v] > a[u,v] **then**
Begin
d[v] := c[u,v];
near[v] := u;
End;
End;

Thuật toán Prim (tt)

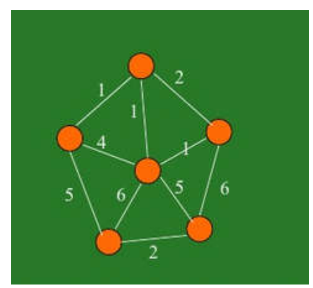


Bước lặp	Đỉnh 1	Đỉnh 2	Đỉnh 3	Đỉnh 4	Đỉnh 5	Đỉnh 6	V_H	T
Khởi tạo								
1								
2								
3								
4								
5								

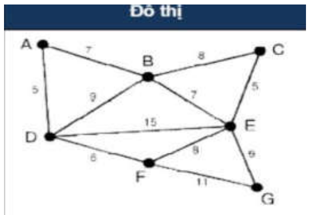
Exercise Prim



Exercise Prim

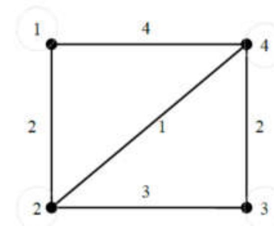


Exercise Prim

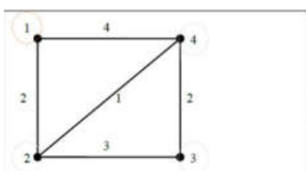


Với A ≡ 1; B ≡ 2; C ≡ 3; D ≡ 4;
E ≡ 5; F ≡ 6; G ≡ 7; A là gốc.

Exercise(i) Prim



Solution(i)-1

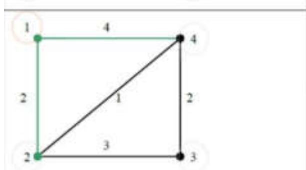


Bước 1: *Chọn tùy ý v ∈ X và khởi tạo Y:= {v}; T := Ø*

Tập X là tập các đỉnh của đồ thị nên

X = {1,2,3,4 }

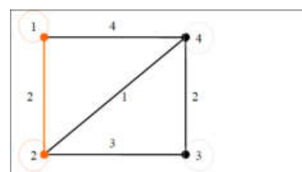
Ta chọn đỉnh **1** làm đỉnh xét đến đầu tiên: Y = {1}; T = Ø. Như vậy X\Y = {2,3,4 }



Bước 2: *Trong số những cạnh e nối đỉnh w với đỉnh v trong Y với w ∈ X\Y và v ∈ Y ta chọn cạnh có trọng lượng nhỏ nhất.*

Cạnh (4,1) và (2,1) nối đỉnh 2 và 4 đến đỉnh 1 trong Y, ta chọn cạnh (2,1) vì nó có trọng nhỏ nhất trong hai cạnh (giá trị là 2).

Solution(i)-2



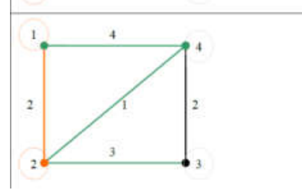
Bước 3: *Gán Y:= Y∪ {w} và T:= T∪ {e}*

Y = Y ∪ {2}

T = T ∪ {(2,1)}

Bước 4: *Nếu T đủ n-1 phần tử thì dừng, ngược lại làm tiếp tục bước 2.*

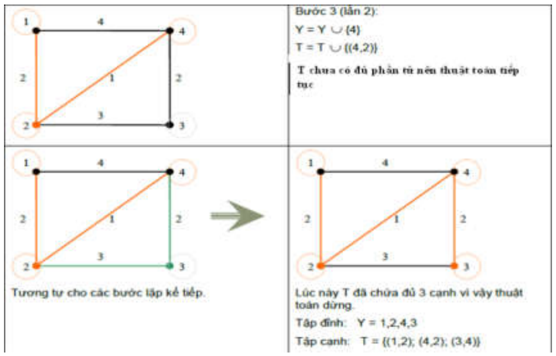
T chỉ có 1 phần tử < n – 1 = 4 – 1 = 3 nên thuật toán chưa dừng.



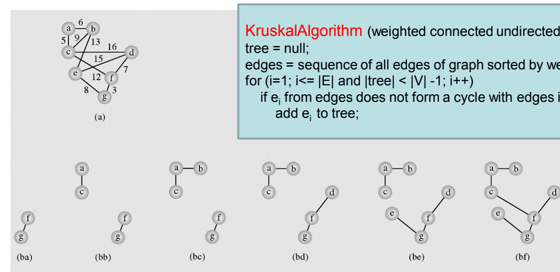
Bước 2 (lần 2):

Cạnh (4,1); (4,2); (3,2) là các cạnh nối đỉnh 4; 3 (tập những đỉnh chưa có trong cây) đến 1;2 (tập các đỉnh đã có trong cây). Tương tự, ta chọn cạnh có trọng lượng nhỏ nhất: (4,2).

Solution(i)-3

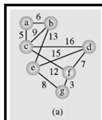


Spanning Trees (continued)

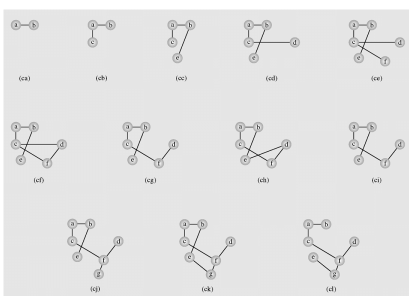


KruskalAlgorithm (weighted connected undirected graph)
tree = null;
edges = sequence of all edges of graph sorted by weight;
for (i=1; i<= |E| and |tree| < |V| -1; i++)
    if e_i from edges does not form a cycle with edges in tree
        add e_i to tree;

Figure 8-15 A spanning tree of graph (a) found, (ba–bf) with Kruskal's algorithm, (ca–cl) and with Dijkstra's method



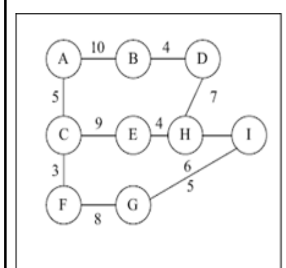
Spanning Trees (continued)



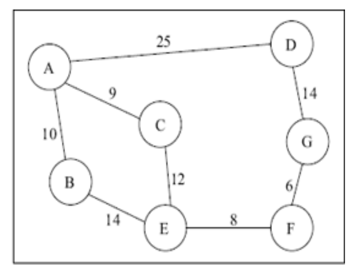
DijkstraMethod(weighted connected undirected graph)
tree = null;
edges = an unsorted sequence of all edges of graph ;
for j=1 to |E|
    add e_j to tree;
    if there is a cycle in tree
        remove an edge with maximum weight from this only cycle;

Figure 8-15 A spanning tree of graph (a) found, (ba–bf) with Kruskal's algorithm, (ca–cl) and with Dijkstra's method (continued)

Exercise Kruskal (i) and Prim (ii)



(i)

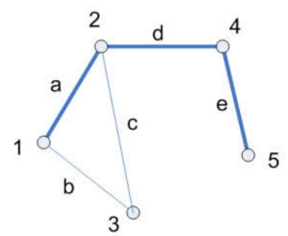


(ii)

V.Đồ thị liên thông

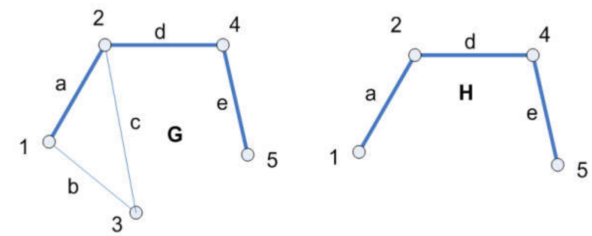
- Đồ thị vô hướng G=(V,E) được gọi là liên thông nếu luôn tìm được đường đi giữa 2 đỉnh bất kỳ của nó.

Đường đi: 1, 3, 2, 4, 5



V.Đồ thị liên thông

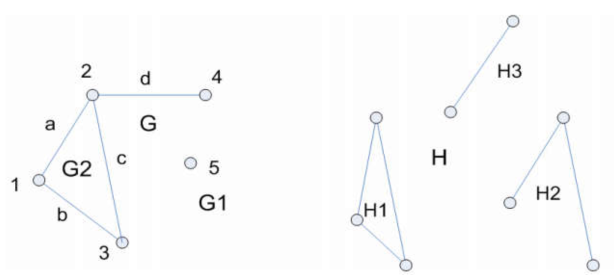
- Đồ thị H=(W,F) được gọi là **đồ thị con** của đồ thị G=(V,E) nếu : $W \subseteq V$ và $F \subseteq E$



V={1, 2, 3, 4, 5} W={1, 2, 4, 5}
E={a, b, c, d, e} F={a, d, e}

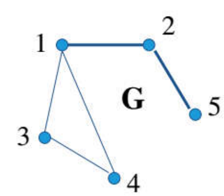
V.Đồ thị liên thông

- Một đồ thị không liên thông sẽ được phân rã thành các thành phần liên thông, và mỗi thành phần liên thông này là một đồ thị con của đồ thị ban đầu.



V.Đồ thị liên thông

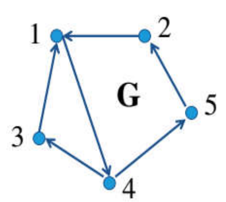
- Đỉnh v được gọi là đỉnh rẽ nhánh nếu việc loại bỏ v cùng các cạnh liên thuộc với nó sẽ làm tăng số thành phần liên thông của đồ thị
- Cạnh e được gọi là cầu nếu việc loại bỏ nó sẽ làm tăng số thành phần liên thông của đồ thị

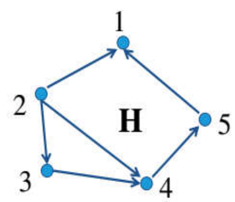


Các đỉnh rẽ nhánh?
Các cạnh là cầu ?

V.Đồ thị liên thông

- Đồ thị có hướng G=(V,E) được gọi là liên thông mạnh nếu luôn tìm được đường đi từ 1 đỉnh bất kỳ đến một đỉnh bất kỳ khác của nó.
- Đồ thị có hướng G=(V,E) được gọi là liên thông yếu nếu đồ thị vô hướng tương ứng với nó là đồ thị vô hướng liên thông.





8.6- Connectivity

- An undirected graph is called **connected** when there is a path between any two vertices of the graph
- A graph is called **n-connected** if there are at least *n* different paths between any two vertices; that is, there are *n* paths between any two vertices that have no vertices in common
- A **2-connected** or **biconnected** graph is when there are at least two nonoverlapping paths between any two vertices

II. TÍNH LIÊN THÔNG TRONG ĐỒ THỊ VÔ HƯỚNG

Một bài toán quan trọng trong lý thuyết đồ thị là bài toán kiểm tra tính liên thông của đồ thị vô hướng hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông của đồ thị vô hướng.

Giả sử đồ thị vô hướng G = (V, E) có n đỉnh đánh số 1, 2, ..., n.

Để liệt kê các thành phần liên thông của G phương pháp cơ bản nhất là:

- Đánh dấu đỉnh 1 và những đỉnh có thể đến từ 1, thông báo những đỉnh đó thuộc thành phần liên thông thứ nhất.
- Nếu tất cả các đỉnh đều đã bị đánh dấu thì G là đồ thị liên thông, nếu không thì sẽ tồn tại một đỉnh v nào đó chưa bị đánh dấu, ta sẽ đánh dấu v và các đỉnh có thể đến được từ v, thông báo những đỉnh đó thuộc thành phần liên thông thứ hai.
- Và cứ tiếp tục như vậy cho tới khi tất cả các đỉnh đều đã bị đánh dấu

⇒ **Dùng thuật toán DFS và BFS**

Connectivity (continued)

- If the vertex is removed from a graph (along with incident edges) and there is no way to find a path from *a* to *b,* then the graph is split into two separate subgraphs called **articulation points**(điểm khớp), or **cut-vertices** (đỉnh cắt))
- If an edge causes a graph to be split into two subgraphs, it is called a **bridge** or **cut-edge**
- Connected subgraphs with no articulation points or bridges are called **blocks**

```
blockDFS(v)
   pred(v) = num(v) = i++;
   for all vertices u adjacent to v
      if edge(uv) has not been processed
         push(edge(uv));
      if num(u) is 0
         blockDFS(u);
         if pred(u) ≥ num(v)                    // if there is no edge from u to a
            e = pop();                          // vertex above v, output a block
            while e ≠ edge(vu)                  // by popping all edges off the
               output e;                        // stack until edge(vu) is
               e = pop();                       // popped off;
            output e;                           // e == edge(vu);
         else pred(v) = min(pred(v),pred(u));   // take a predecessor higher up in
      else if u is not the parent of v          // tree;
         pred(v) = min(pred(v),num(u));         // update when back edge(vu) is
                                                // found;
blockSearch()
   for all vertices v
      num(v) = 0;
   i = 1;
   while there is a vertex v such that num(v) == 0
      blockDFS(v);
```


Connectivity in Undirected Graphs

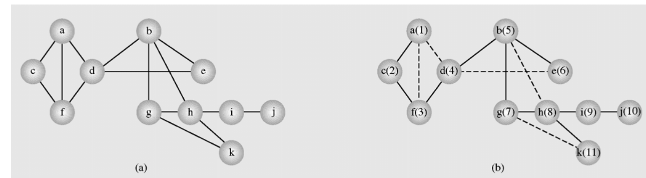


Figure 8-16 Finding blocks and articulation points using the `blockDFS()` algorithm

Connectivity in Undirected Graphs (continued)

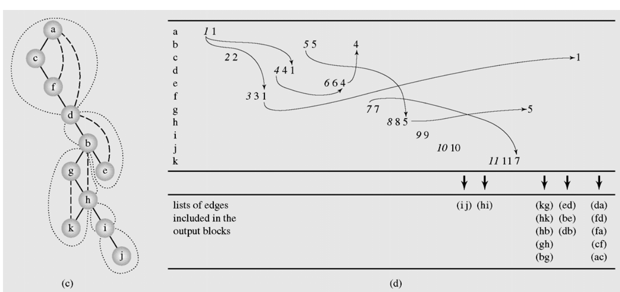


Figure 8-16 Finding blocks and articulation points using the `blockDFS()` algorithm (continued)

IV. CÁC THÀNH PHẦN LIÊN THÔNG MẠNH

Đối với đồ thị có hướng, người ta quan tâm đến bài toán kiểm tra tính liên thông mạnh, hay tổng quát hơn: Bài toán liệt kê các thành phần liên thông mạnh của đồ thị có hướng. Đối với bài toán đó ta có một phương pháp khá hữu hiệu dựa trên thuật toán tìm kiếm theo chiều sâu Depth First Search.

Connectivity in Directed Graphs

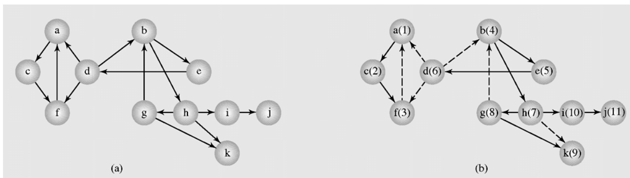


Figure 8-17 Finding strongly connected components with the `strongDFS()` algorithm


```
strongDFS(v)
  pred(v) = num(v) = i++;
  push(v);
  for all vertices u adjacent to v
    if num(u) is 0
      strongDFS(u);
      pred(v) = min(pred(v),pred(u));          // take a predecessor higher up in
    else if num(u) < num(v) and u is on stack  // tree; update if back edge found
      pred(v) = min(pred(v),num(u));           // to vertex u is in the same SCC;
  if pred(v) == num(v)                         // if the root of a SCC is found,
    w = pop();                                 // output this SCC, i.e.,
    while w ≠ v                                // pop all vertices off the stack
      output w;                                // until v is popped off;
      w = pop();
    output w;                                  // w == v;

stronglyConnectedComponentSearch()
  for all vertices v
    num(v) = 0;
  i = 1;
  while there is a vertex v such that num(v) == 0
    strongDFS(v);
```


Connectivity in Directed Graphs (continued)

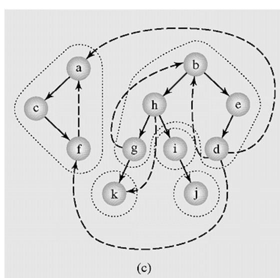
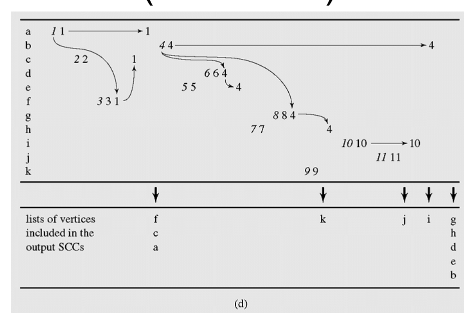


Figure 8-17 Finding strongly connected components with the `strongDFS()` algorithm (continued)

Connectivity in Directed Graphs (continued)



To Eulerian Graphs

Figure 8-17 Finding strongly connected components with the `strongDFS()` algorithm (continued)

8.7-Topological Sort

- Topology: a branch of mathematics which studies the properties of geometrical forms which retain their identity under certain transformations, such as stretching or twisting, which are homeomorphic (Webster dictionary)
- A **topological sort** linearizes a **digraph**
- It labels all its vertices with numbers 1, . . . , |*V*| so that *i* < *j* only if there is a path from vertex v_i to vertex v_j
- The **digraph must not include a cycle**; otherwise, a topological sort is impossible

```
topologicalSort(digraph)
  for i = 1 to |V|
    find a minimal vertex v;
    num(v) = i;
    remove from digraph vertex v and all edges incident with v;
```


Topological Sort (continued)

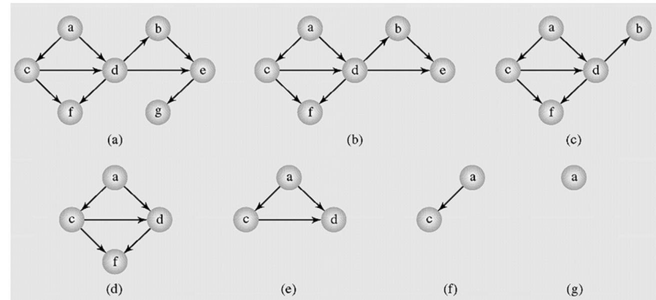


Figure 8-18 Executing a topological sort

A vertex with no outgoing edge: sink, minimal vertex

Base on outgoing edges, result order: g, e, b, f, d, c, a

Topological Sort (continued)

a	1													7
b				4					3					
c		2											6	
d			3									5		
e					5			2						
f										7	4			
g						6	1							

(h)

Figure 8-18 Executing a topological sort (continued)

8.8-Networks

- A **network** is a digraph with one vertex *s*, called the **source**, with no incoming edges, and one vertex *t*, called the **sink**, with no outgoing edges
- Each edge has a capacity, maximum units can be put to this edge.

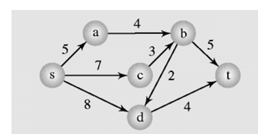
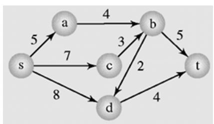


Figure 8-19 A pipeline with eight pipes and six pumping stations

Networks – The Maximum-flow Problem

- Total flow to the vertex v = $\Sigma_u f$ (edge(uv))
- X: set of some vertices including s
- $\bar{X}$: set of other vertices including t
- Cut: set of edges between X and $\bar{X}$
- Capacity of a cut: sum of capacity of its edges.
- Theorem: In any network, the maximal flow from s to t is equal to the minimal capacity of any cut.



If X= { s, a } ➔ $\bar{X}$ ={ b, c, d, t }
Cut = { ab, sc, sb }
Capacity $_{cut}$ = 4 + 7 +8 = 19

Maximum Flows

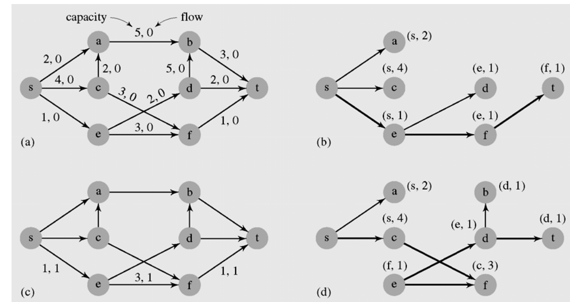


Figure 8-20 An execution of `FordFulkersonAlgorithm()` using depth-first search

Maximum Flows (continued)

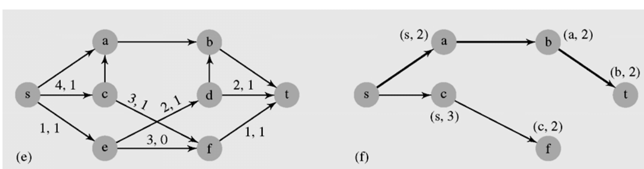


Figure 8-20 An execution of `FordFulkersonAlgorithm()` using depth-first search (continued)

Maximum Flows (continued)

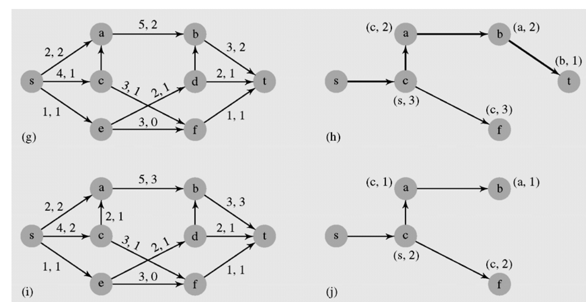


Figure 8-20 An execution of `FordFulkersonAlgorithm()` using depth-first search (continued)

Maximum Flows (continued)

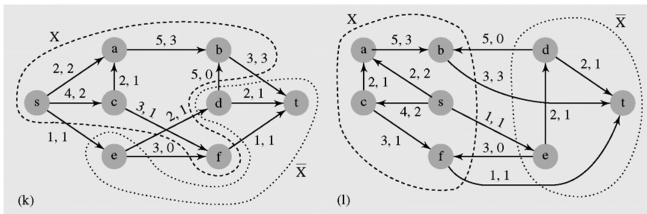


Figure 8-20 An execution of `FordFulkersonAlgorithm()` using depth-first search (continued)

Maximum Flows (continued)

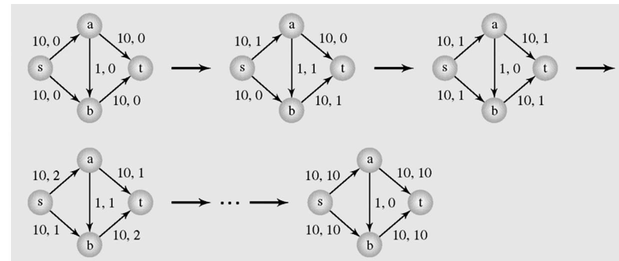


Figure 8-21 An example of an inefficiency of `FordFulkersonAlgorithm()`

Maximum Flows (continued)

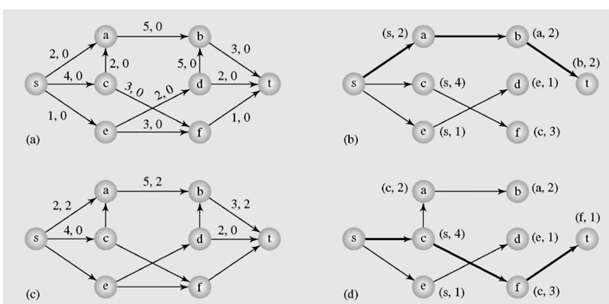


Figure 8-22 An execution of `FordFulkersonAlgorithm()` using breadth-first search

Maximum Flows (continued)

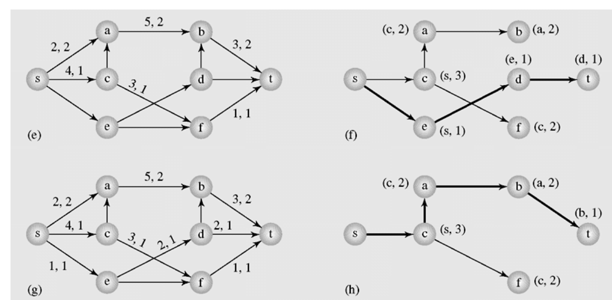


Figure 8-22 An execution of `FordFulkersonAlgorithm()` using breadth-first search (continued)

Maximum Flows (continued)

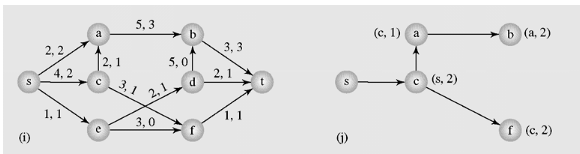


Figure 8-22 An execution of `FordFulkersonAlgorithm()` using breadth-first search (continued)

Maximum Flows (continued)

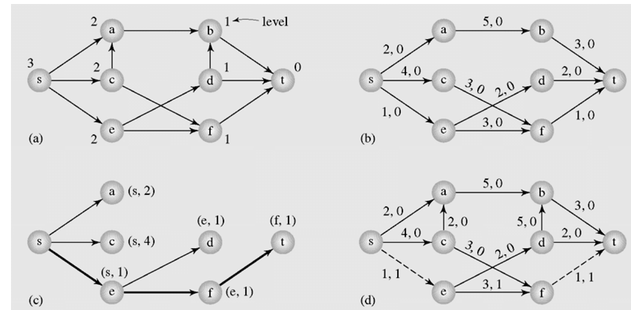


Figure 8-23 An execution of `DinicAlgorithm()`

Maximum Flows (continued)

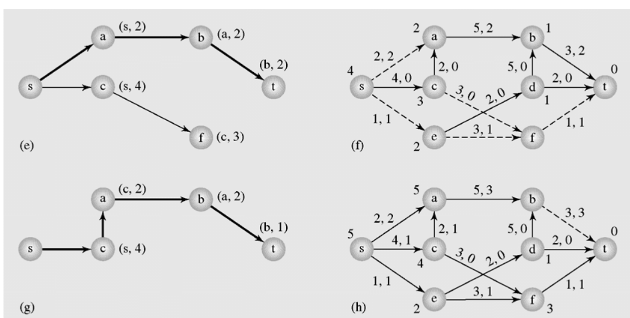


Figure 8-23 An execution of `DinicAlgorithm()` (continued)

Maximum Flows (continued)

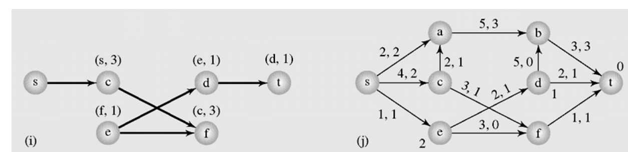


Figure 8-23 An execution of `DinicAlgorithm()` (continued)