1st Class, 2nd Course

2016-2017

Discrete Structures

الهياكل المتقطعة

أستاذة المادة : م.د. إقباس عز الدين

Logic and propositional calculus

A *proposition* (or *statement*) is a declarative statement which is true or false, but not both.

Example: the following six sentences:
(1) Ice floats in water.
(2) China is in Europe.
(3) 2 + 2 = 4
(4) 2 + 2 = 5
(5) Where are you going?
(6) Do your homework.

The first four are propositions, the last two are not. Also, (1) and (3) are true, but (2) and (4) are false.

Compound Propositions

It is the proposition that composed of sub propositions and various connectives.
Primitive proposition is the proposition that cannot be broken down into simpler propositions.
For example, the above propositions are primitive propositions, while:
"Roses are red **and** violets are blue." and
"John is smart **or** he studies every night.", Are compound.

Basic Logical Operations

1- Conjunction, $p \wedge q$
2- Disjunction, $p \vee q$
3- Negation, $\neg p$

Conjunction (AND/∧)

Let A and B are two statements, then conjunction of A and B is denoted as A ∧ B (read as "A and B") and the truth value of the statement A ∧ B is true if, truth values of both the statements A & B are true. Otherwise, it is false.

Disjunction (OR/∨)
Let A and B are two statements then disjunction of A and B is denoted as A ∨ B (read as "A Or B") and the truth value of the statement A ∨ B is *true* if the truth value of the statement A or B or both are *true*. Otherwise it is *false*.

Negation (¬)
Connective Negation is used with unary statement mode. The negation of the statement inverts its logic sense. That is similar to the introducing "not" at the appropriate place in the statement so that its meaning is reverse or negate.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

(*a*) "*p* and *q*"

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

(*b*) "*p* or *q*"

p	$\neg p$
T	F
F	T

(*c*) "not *p*"

EXAMPLE :
Consider the following four statements:
(i) Ice floats in water **and** 2 + 2 = 4.
(ii) Ice floats in water **and** 2 + 2 = 5.
(iii) China is in Europe **and** 2 + 2 = 4.
(iv) China is in Europe **and** 2 + 2 = 5.
Only the first statement is true. Each of the others is false since at least one of its substatements is false.

EXAMPLE:
Consider the following four statements:
(i) Ice floats in water **or** 2 + 2 = 4.
(ii) Ice floats in water **or** 2 + 2 = 5.
(iii) China is in Europe **or** 2 + 2 = 4.
(iv) China is in Europe **or** 2 + 2 = 5.

Only the last statement (iv) is false. Each of the others is true since at least one of its sub-statements is true.

EXAMPLE :
Consider the following six statements:
(*a*1) Ice floats in water.
(*a*2) It is false that ice floats in water.
(*a*3) Ice does not float in water.

(*b*1) 2 + 2 =5
(*b*2) It is false that 2 + 2 = 5.
(*b*3) 2 + 2 ≠5

Then (*a*2) and (*a*3) are each the negation of (*a*1); and (*b*2) and (*b*3) are each the negation of (*b*1). Since (*a*1) is true, (*a*2) and (*a*3) are false; and since (*b*1) is false, (*b*2) and (*b*3) are true.

PROPOSITIONS AND TRUTH TABLES
The truth table for the compound proposition ¬(p∧¬q) is:

p	q	$\neg q$	$p \wedge \neg q$	$\neg(p \wedge \neg q)$
T	T	F	F	T
T	F	T	T	F
F	T	F	F	T
F	F	T	F	T

TAUTOLOGIES AND CONTRADICTIONS
Some propositions P(p, q, . . .) contain only T in the last column of their truth tables or, in other words, they are true for any truth values of their variables. Such propositions are called tautologies.
Analogously, a proposition P(p, q, . . .) is called a contradiction if it contains only F in the last column of its truth table or, in other words, if it is false for any truth values of its variables.

For example, the proposition “p or not p,” that is, p ∨¬p, is a tautology, and the proposition “p and not p,” that is, p∧¬p, is a contradiction.

p	$\neg p$	$p \vee \neg p$
T	F	T
F	T	T

(a) $p \vee \neg p$

p	$\neg p$	$p \wedge \neg p$
T	F	F
F	T	F

(b) $p \wedge \neg p$

Note that the negation of a tautology is a contradiction since it is always false, and the negation of a contradiction is a tautology since it is always true.

LOGICAL EQUIVALENCE

Two propositions P(p, q, . . .) and Q(p, q, . . .) are said to be logically equivalent, or equal, denoted by P(p, q, . . .) ≡ Q(p, q, . . .) if they have identical truth tables.

For example: the truth tables of ¬(p ∧ q) and ¬p∨¬q, both truth tables are the same, that is, both propositions are false in the first case and true in the other three cases. Accordingly, we can write:

¬(p ∧ q) ≡ ¬p ∨¬q

p	q	$p \wedge q$	$\neg(p \wedge q)$
T	T	T	F
T	F	F	T
F	T	F	T
F	F	F	T

(a) $\neg(p \wedge q)$

p	q	$\neg p$	$\neg q$	$\neg p \vee \neg q$
T	T	F	F	F
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

(b) $\neg p \vee \neg q$

ALGEBRA OF PROPOSITIONS:

Idempotent laws:	(1a) $p \vee p \equiv p$	(1b) $p \wedge p \equiv p$
Associative laws:	(2a) $(p \vee q) \vee r \equiv p \vee (q \vee r)$	(2b) $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Commutative laws:	(3a) $p \vee q \equiv q \vee p$	(3b) $p \wedge q \equiv q \wedge p$
Distributive laws:	(4a) $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	(4b) $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Identity laws:	(5a) $p \vee F \equiv p$ (6a) $p \vee T \equiv T$	(5b) $p \wedge T \equiv p$ (6b) $p \wedge F \equiv F$
Involution law:	(7) $\neg\neg p \equiv p$	
Complement laws:	(8a) $p \vee \neg p \equiv T$ (9a) $\neg T \equiv F$	(8b) $p \wedge \neg p \equiv F$ (9b) $\neg F \equiv T$
DeMorgan's laws:	(10a) $\neg(p \vee q) \equiv \neg p \wedge \neg q$	(10b) $\neg(p \wedge q) \equiv \neg p \vee \neg q$

Implication (→)

Let A and B are two statements then the statement A → B (read as "A implies B" or "if A then B") is an implication statement (conditional statement).

The truth value of A → B is false only when truth value of B is false; Otherwise it is true, (The conditional A → B is false only when the first part A is true and the second part B is false. Accordingly, when A is false, the conditional A → B is true regardless of the truth value of B)

In the implicative statement (A → B), statement A is known as antecedent or predecessor and statement B is known as consequent or resultant.

A	B	A → B
F	F	T
F	T	T
T	F	F
T	T	T

Note that the truth table of $p \rightarrow q$ and $\neg p \vee q$ and are identical, that is, they are both false only in the second case. Accordingly, $p \rightarrow q$ is logically equivalent to $\neg p \vee q$; that is,

$p \rightarrow q \equiv \neg p \vee q$

p	q	$\neg p$	$\neg p \vee q$
T	T	F	T
T	F	F	F
F	T	T	T
F	F	T	T

(c) $\neg p \vee q$

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

(a) $p \rightarrow q$

Arguments

A valid argument is a finite set of propositions P1, ..., Pr called premises, together with a proposition C, the conclusion, such that the propositional form (P1 ^ P2 ^ ... ^ Pr) C is a tautology.

We say C is a logical consequence of the premises.

We write P1, ..., Pr $\vdash$ C. The symbol ` is called the turnstile.

If an argument is not valid we say that it is invalid.

Method to prove validity

Example :

Let Q1 = "John graduates"

Q2 = "Mary graduates"

Q3 = "John gets a job"

Q4 = "Mary gets a job"

Q5 = "Mary earns money"

(i) Consider the following argument:

"If John graduates then he gets a job".

"John graduates".

"Therefore John gets a job".

To see the "form" of this argument we symbolize it as:

Q1 $\rightarrow$Q3, Q1 $\vdash$ Q3.

Now we check that ((Q1 $\rightarrow$Q3) ^ Q1) $\rightarrow$ Q3 is a tautology:

Q_1	Q_3	$Q_1 \rightarrow Q_3$	$(Q_1 \rightarrow Q_3) \wedge Q_1$	$((Q_1 \rightarrow Q_3) \wedge Q_1) \rightarrow Q_3$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

This tautology shows that the argument is valid.

Example (ii) Consider the following argument:

"If Mary graduates then she gets a job".

"Mary does not get a job".

"Therefore Mary does not graduate".

Symbolized, this becomes: Q2 →Q4, (¬ Q4) ⊢ (¬ Q2).

Now check that ((Q2 →Q4) ^ (¬ Q4)) ⊢ (¬ Q2) is a tautology.

Q_2	Q_4	$Q_2 \rightarrow Q_4$	$\neg Q_4$	$(Q_2 \rightarrow Q_4) \wedge (\neg Q_4)(\equiv A)$	$\neg Q_2$	$A \rightarrow (\neg Q_2)$
T	T	T	F	F	F	T
T	F	F	T	F	F	T
F	T	T	F	F	T	T
F	F	T	T	T	T	T

Example (iii) Consider the following argument:

"Either Mary or John graduate".

"John does not graduate".

"Therefore Mary graduates".

Symbolized, this becomes Q2 vQ1, (¬ Q1) ⊢ Q2.

*Now check that ((Q2 vQ1) ^ (¬ Q1)) → Q2 is a tautology.

Q_1	Q_2	$Q_2 \vee Q_1$	$\neg Q_1$	$(Q_2 \vee Q_1) \wedge (\neg Q_1)(\equiv B)$	$B \rightarrow Q_2$
T	T	T	F	F	T
T	F	T	F	F	T
F	T	T	T	T	T
F	F	F	T	F	T

Example (iv) Consider the following argument:

"If Mary graduates then she gets a job".

"If Mary gets a job then she earns money".
"Therefore if Mary graduates then she earns money".

Symbolized, this becomes Q2 →Q4, Q4 →Q5 ⊢ (Q2 →Q5).

*Now check that ((Q2 →Q4) ∧ (Q4 →Q5)) →(Q2 →Q5) is a tautology.

Q_2	Q_4	Q_5	$Q_2 \to Q_4$ A	$Q_4 \to Q_5$ B	$A \wedge B$	$Q_2 \to Q_5$ C	$(A \wedge B) \to C$
T	T	T	T	T	T	T	T
T	T	F	T	F	F	F	T
T	F	T	F	T	F	T	T
T	F	F	F	T	F	F	T
F	T	T	T	T	T	T	T
F	T	F	T	F	F	T	T
F	F	T	T	T	T	T	T
F	F	F	T	T	T	T	T

We can sum up the above by saying the following are all valid:

(i) p → q, p ⊢ q,

(ii) p → q, ¬ q ⊢ ¬ p,

(iii) p ∨q, ¬ q ⊢ p,

(iv) p → q, q → r ⊢ p → r.

Example

Show that p → q, q ∨p ⊢ (¬ q) ∨ (¬ p) is valid.

p	q	$p \to q$	$q \vee p$	$((p \to q) \wedge (q \vee p))$ P	$((\neg q) \vee (\neg p))$ C	$P \to C$	
T	T	T	T	T	F	F	←
T	F	F	T	F	T	T	
F	T	T	T	T	T	T	
F	F	T	F	F	T	T	

We do not have a tautology in the last column so the argument is invalid.

Homework:

Is p →q, q ∨ p ⊢ (¬ q) ∨(¬ p) valid?

Is $p \rightarrow (s \rightarrow (\neg r)), p \rightarrow r, p \vdash \neg s$ valid?

Is $(p \vee q) \rightarrow s, q \rightarrow s \vdash s$ valid?

Graphs:

Graphs are discrete structures consisting of vertices and edges that connect these vertices, so a graph G(V,E) consists of:

(i) V, a nonempty set of *vertices* (or *nodes*).

(ii) *E*, a set of *edges*. Each *edge* has either one or two vertices associated with it, called its *endpoints*.

Graphs are used in a wide variety of models with computer science such as communication network, logical design, transportation networks, formal languages, compiler writing and retrieval.
For example: in a communication network, where computers can be represented by vertices and communication links by edges. A graph in which each edge connects two different vertices and where no two edges connect the same pair of vertices is called a **simple graph**.

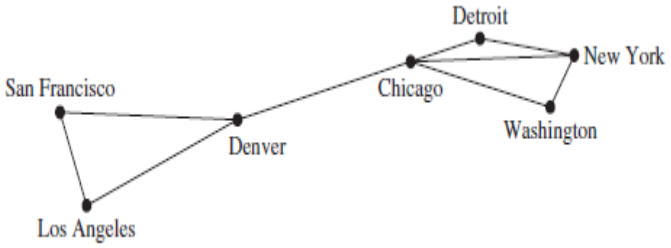


Figure (1): simple graph

A computer network may contain multiple links between data centers, as shown in Figure 2. To model such networks we need graphs that have more than one edge connecting the same pair of vertices. Graphs that may have **multiple edges** connecting the same vertices are called **multigraphs**.

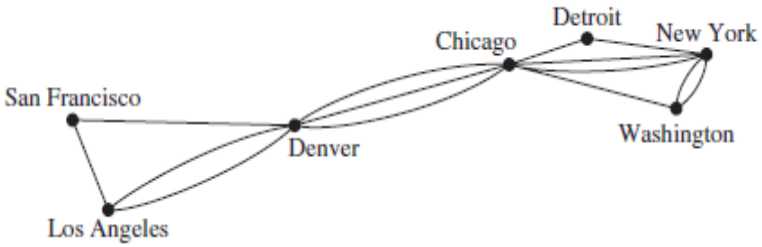


Figure (2): multigraphs

Sometimes a communications link connects a data center with itself, perhaps a feedback loop for diagnostic purposes. Such a network is illustrated in Figure 3. To model this network we need to include edges that connect a vertex to itself. Such edges are called **loops**,

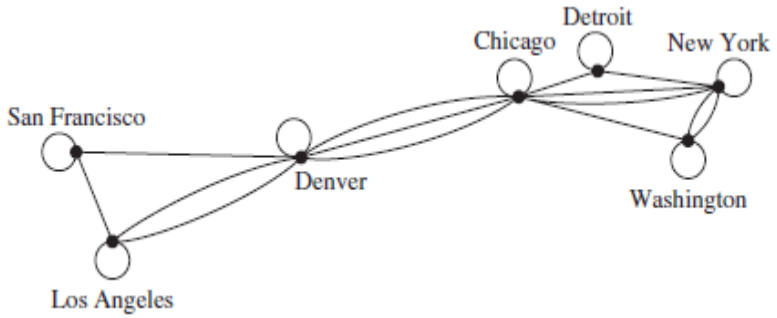


Figure (3): multigraphs with loops

In a computer network, some links may operate in only one direction (such links are called single duplex lines). This may be the case if there is a large amount of traffic sent to some data centers, with little or no traffic going in the opposite direction. Such a network is shown in Figure 4. To model such a computer network we use a **directed graph**. Each edge of a directed graph is associated to an ordered pair.

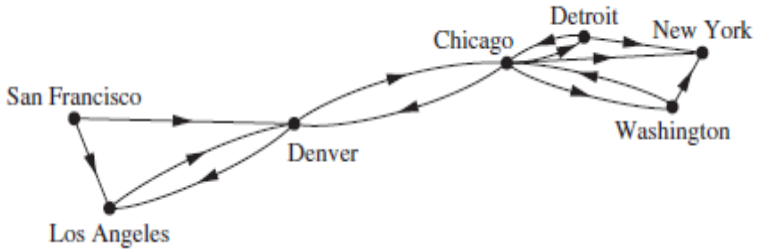


Figure (4): directed graph

For example we have in Figure (5) the graph G(V,E) where: V consists of four vertices A, B, C, D ; and, E consists of five edges

e1 ={A,B},
e2 = {B,C},
e3 = {C, D},
e4 = {A, C},
e5 = {B, D}.

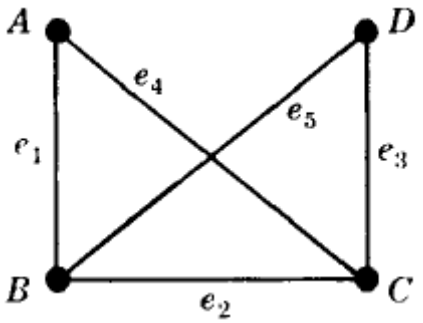


Figure (5)

Vertices u and v are said to be **adjacent** if there is an edge e={u,v}. In such a case, u and v are called the endpoints of e, and e is said to connect u and v. Also, the edge e is said to be **incident** on each of its endpoints u and v.

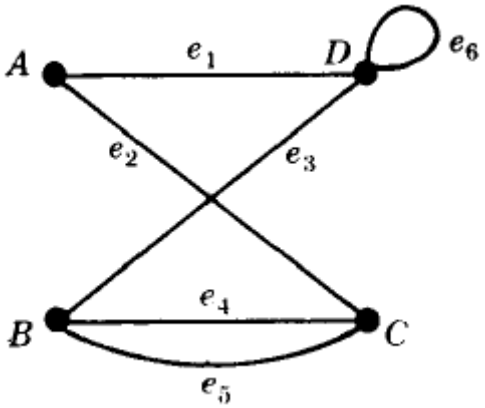


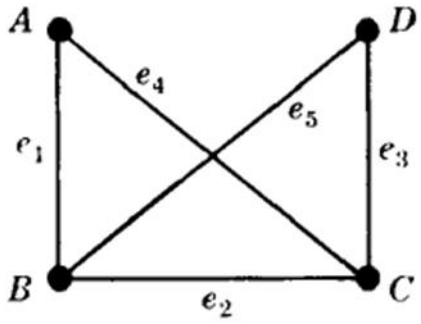
Figure 6: multigraph with: 1) multiple edges e4 & e5
2) a loop e6

Degree :
The degree of a vertex v [deg(v)], is equal to the number of edges which are incident on v. since each edge is counted twice in counting the degrees of the vertices of a graph.

Theorem: The sum of the degrees of the vertices of a graph is equal to twice the number of edges. Let $G = (V, E)$ be an undirected graph with m edges. Then

$$2m = \sum_{v \in V} deg(v).$$

For example, in the figure (5) we have



deg(A) = 2,
deg(B) = 3,
deg(C) = 3,
deg(D) = 2

The sum of the degrees = twice the number of edges = 2×5=10

EXAMPLE 1: How many edges are there in a graph with 10 vertices each of degree six?
Solution: Because the sum of the degrees of the vertices is $6 \times 10 = 60$, it follows that $2m = 60$
where m is the number of edges. Therefore, $m = 30$.

A vertex is said to be **even** or **odd** according as its degree is an even or odd number. Thus A and D are even vertices whereas B and C are odd vertices.
This theorem also holds for multigraphs where a loop is counted twice towards the degree of its endpoint. For example, in Fig (6) we have deg (D) = 4 since the edge e6 is counted twice; hence D is an even vertex.
A vertex of degree zero is called an isolated vertex.

Subgraphs
Consider a graph G = G(V,E) and a graph H = H(V', E') is called a subgraph of G if the vertices and edges of H are contained in the vertices and edges of G, that is, if $V' \subseteq V$ and $E' \subseteq E$.

Sometimes we need only part of a graph to solve a problem. For instance, we may care only about the part of a large computer

network that involves the computer centers in New York, Denver, Detroit, and Atlanta. Then we can ignore the other computer centers and all telephone lines not linking two of these specific four computer centers. In the graph model for the large network, we can remove the vertices corresponding to the computer centers other than the four of interest, and we can remove all edges incident with a vertex that was removed. When edges and vertices are removed from a graph, without removing endpoints of any remaining edges, a smaller graph is obtained. Such a graph is called a **subgraph** of the original graph.

EXAMPLE 2: The graph G shown in Figure 7 is a subgraph of $K5$. If we add the edge connecting a, b, c and e to G, we obtain the subgraph induced by $W = \{a, b, c, e\}$.

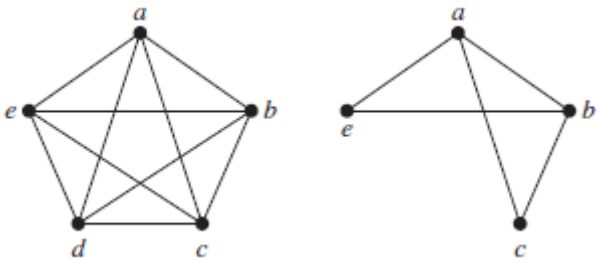


Figure 7

Connectivity :

Many problems can be modeled with paths formed by traveling along the edges of graphs. For instance, the problem of determining whether a message can be sent between two computers using intermediate links can be studied with a graph model. Problems of efficiently planning routes for mail delivery, garbage pickup, diagnostics in computer networks, and so on can be solved using models that involve paths in graphs.

a walk is a sequence of edges that begins at a vertex of a graph and travels from vertex to vertex along edges of the graph. As the path travels along its edges, it visits the vertices along this walk, that is, the endpoints of these edges.

A **walk** in a multigraph G consists of an alternating sequence of vertices and edges of the form:

v0, e1, v1, e2, v2,……., en-1,vn-1,en,vn

where each edge e_i contains the vertices v_{i-1} and v_i (which appear on the sides of ei in the sequence).

Length of walk : is the number n of edges. When there is no ambiguity, we denote a path by its sequence of vertices

(v0, v1, . . . , vn).

Closed walk: the walk is said to be closed if v0 = vn .
Otherwise, we say that the walk is from v0 to vn.

Trail: is a walk in which all edges are distinct.
Path: is a walk in which all vertices are distinct.
Cycle: is a closed walk such that all vertices are distinct except v1 = vn, A cycle of length k is called a k-cycle.

EXAMPLE 1
In the simple graph shown in Figure 8:

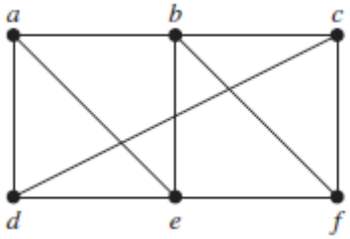


Figure 8

a, *d*, *c*, *f* , *e* is a path of length 4, because {*a*, *d*}, {*d*, *c*}, {*c*, *f* }, and {*f*, *e*} are all edges. However,

d, *e*, *c*, *a* is not a path, because {*e*, *c*} is not an edge. Note that

b, *c*, *f* , *e*, *b* is a circuit of length 4 because {*b*, *c*}, {*c*, *f* }, {*f*, *e*}, and {*e*, *b*} are edges, and this path begins and ends at *b*.

The walk *a*, *b*, *e*, *d*, *a*, *b*, which is of length 5, is not path because it contains the edge {*a*, *b*} twice.

Example: Consider the graph in figure (9), then

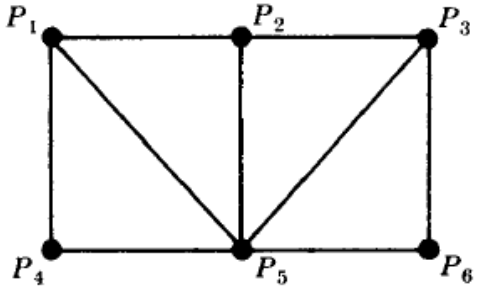


Figure (9)

The sequence: (P4, P1, P2, P5, P1, P2, P3, P6) is a walk from P4 to P6. It is not a trail since the edge {P1,P2} is used twice.

The sequence: (P4, P1, P5, P3, P2, P6) Is not a walk since there is no edge {P2, P6}.

The sequence: (P4, P1, P5, P2, P3, P5, P6) is a trail since no edge is used twice; but it is not a path since the vertex P5 is used twice.
The sequence: (P4, P1, P5, P3, P6) Is a path from P4 to P6.

The shortest path from P4 to P6 is (P4, P5, P6) which has length = 2 (2 edges only)
The distance between vertices u & v d(u,v) is the length of the shortest path d(P4,P6) = 2

Connectivity, Connected Components

A graph G is connected if there is a path between any two of its vertices. The graph in Fig.(9) is connected, but the graph in Fig.(10) is not connected since, for example, there is no path between vertices D and E.

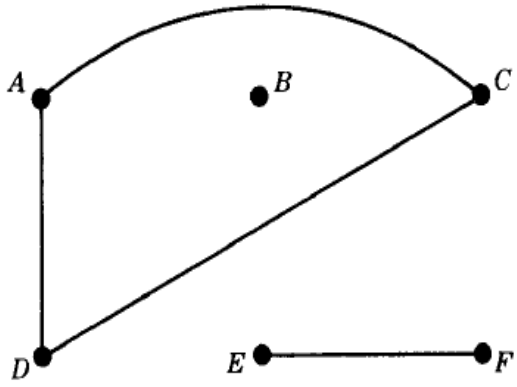


Figure (10)

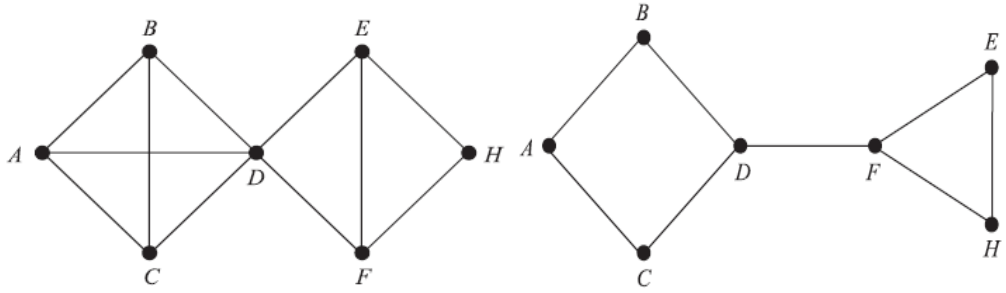
Suppose G is a graph. A connected subgraph H of G is called a connected component of G. It is clear that any graph G can be partitioned into its connected components. For example, the graph G in Fig. (10) has three connected components, the subgraphs induced by the vertex sets {A,C,D}, {E,F}, and {B}.

The vertex B in Fig. (10) is called an isolated vertex since B does not belong to any edge or, in other words, deg(B) = 0

Distance

Consider a connected graph G. The distance between vertices u and v in G, written d(u,v), is the length of the shortest path between u and v.

For example, in Fig. 11*(a)*, *d(A,F)* = 2,
whereas in Fig. 11(b), *d(A,F)* = 3.



a b

Figure 11

Cutpoints and Bridges

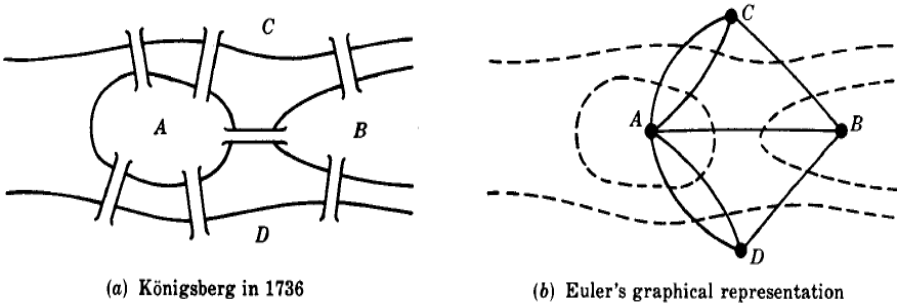
Let G be a connected graph. A vertex v in G is called a cutpoint if G-v is disconnected. (G-v is the graph obtained from G by deleting v and all edges containing v.)

An edge e of G is called a bridge if G - e is disconnected. (G - e is the graph obtained from G by simply deleting the edge e). In Fig. 11(a), the vertex D is a cutpoint and there are no bridges.

In Fig. 11(b), the edge = {D,F} is a bridge. (Its endpoints D and F are cutpoints.)

The Bridges of konigsberg, traversable multigraphs

The eighteenth-century East Prussian town of Konigsberg included two islands and seven bridges. Question: beginning anywhere and ending anywhere, can a person walk through town crossing all seven bridges but not crossing any bridge twice? The people of Konigsberg wrote to the celebrated Swiss mathematician L. Euler about this question. Euler proved in 1736 that such a walk is impossible. He replaced the islands and two side of the river by points and the bridges by curves, obtaining Fig 12 (b).



(a) Königsberg in 1736 (b) Euler's graphical representation

Figure 12

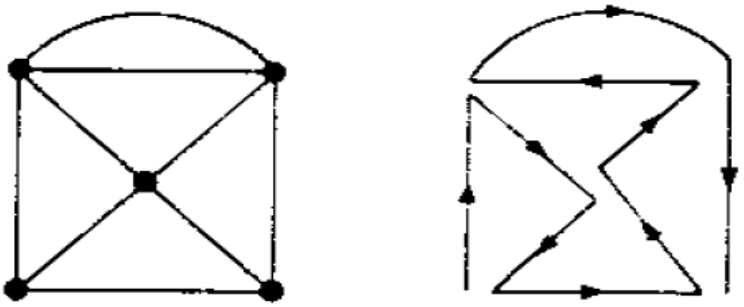
Konigsberg graph is a multigraph, A multigraph is said to traversable if it can be drawn without any breaks and without repeating any edge. That is if there is a walk includes all vertices and uses each edge exactly once. Such a walk must be a trail (no edge is used twice) and will be called a *traversable trail*.

We now show how Euler proved that the konigsberg multigraph is not traversable and the walk in it is impossible. Suppose a multigraph is traversable and that a traversable trail does not begin or end at vertex P. thus the edges in the trail incident with P must appear in pairs, and so P is an even vertex. Therefore if a

vertex Q is odd, the traversable trail must begin or end at Q. Consequently, a multigraph with more than two odd vertices cannot be traversable. Observe that the multigraph corresponding to the Konigsberg bridge problem has four odd vertices. Thus one cannot walk through Konigsberg so that each bridge is crossed exactly once.

Theorem (Euler) : A finite connected graph is eulerian if and only if each vertex has even degree.

Corollary: Any finite connected graph with two odd vertices is traversable. A traversable trail may begin at either odd vertex and will end at the other odd vertex.



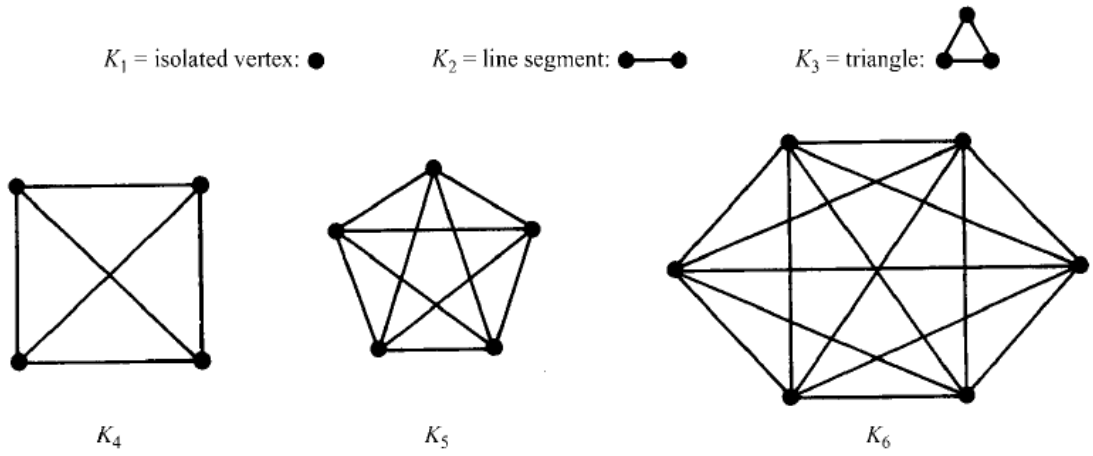
COMPLETE, REGULAR, AND BIPARTITE GRAPHS:

There are many different types of graphs:

Complete graph:

A graph G is said to be complete if every vertex in G is connected to every other vertex.

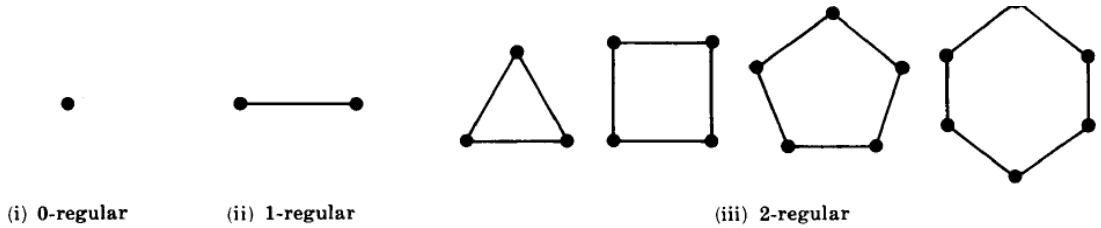
The complete graph G must be connected. The complete graph with n vertices is denoted by Kn



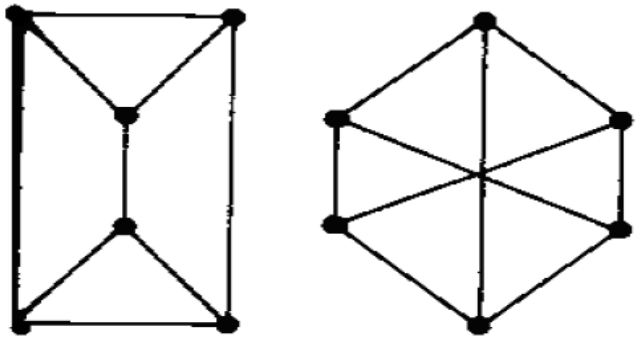
Regular Graph

Every vertex has the same degree. A graph G is regular of degree K or K- regular if every vertex has degree K.

Example: 2-regular graph with every vertex has degree 2.



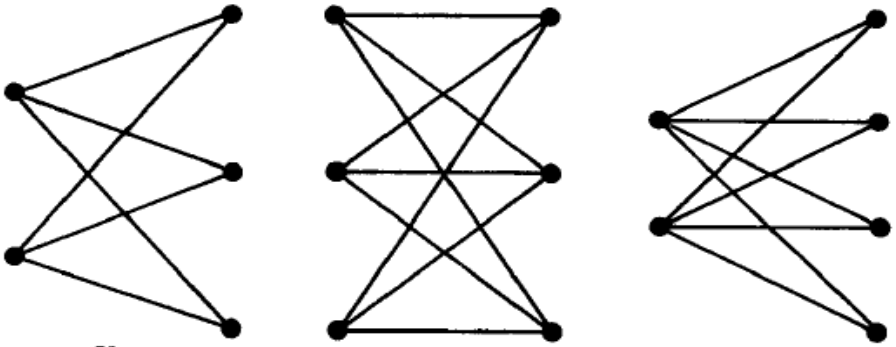
(i) 0-regular (ii) 1-regular (iii) 2-regular



(vi) 3-regular

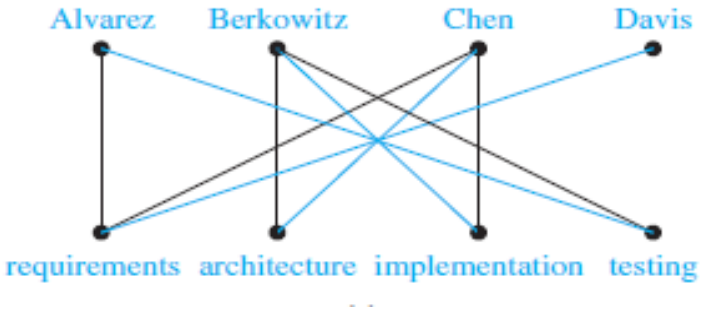
Bipartite graph :

Graph G is said to be bipartite if its vertices V can be partitioned into two subsets M and N such that each edge of G connects a vertex of M to a vertex of N.



$K_{2,3}$ $K_{3,3}$ $K_{2,4}$

Bipartite graphs can be used to model many types of applications that involve matching the elements of one set to elements of another, example: as **in Job Assignments** that used to assign each employees in a group with *n* different jobs.



Tree graph:

A graph T is called a *tree* if T is connected and T has no cycles. Consider a tree T . Clearly, there is only one simple path between two vertices of T ; otherwise, the two paths would form a cycle. Also:

(a) Suppose there is no edge {u, v} in T and we add the edge e = {u, v} to T . Then the simple path from u to v in T and e will form a cycle; hence T is no longer a tree.

(b) suppose there is an edge e = {u, v} in T , and we delete e from T . Then T is no longer connected; hence T is no longer a tree.

Theorem: Let G be a graph with $n > 1$ vertices. Then the following are equivalent:

(i) G is a tree.

(ii) G is a cycle-free and has $n - 1$ edges.

(iii) G is connected and has $n - 1$ edges.

This theorem also tells us that a finite tree T with n vertices must have $n-1$ edges. For example, the tree in Fig. 13*(a)* has 9 vertices and 8 edges, and the tree in Fig. 13*(b)* has 13 vertices and 12 edges.

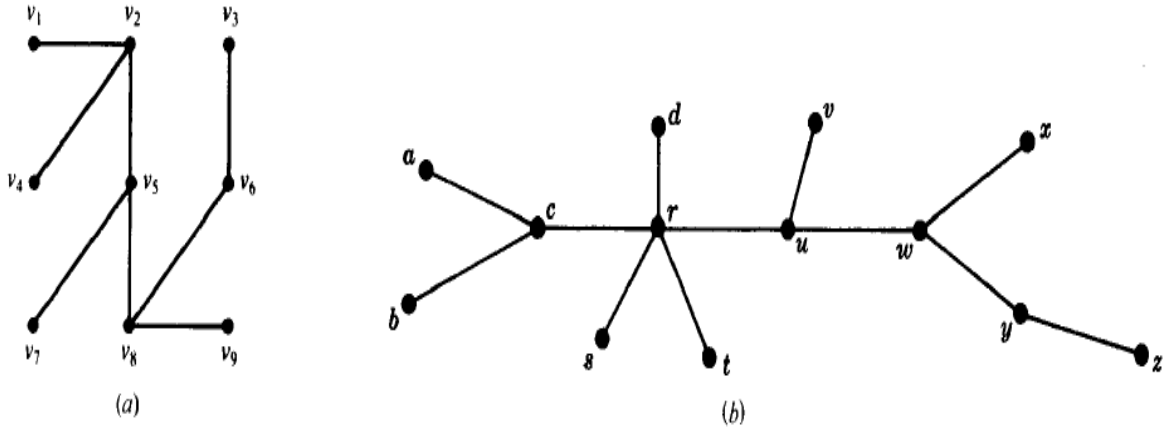


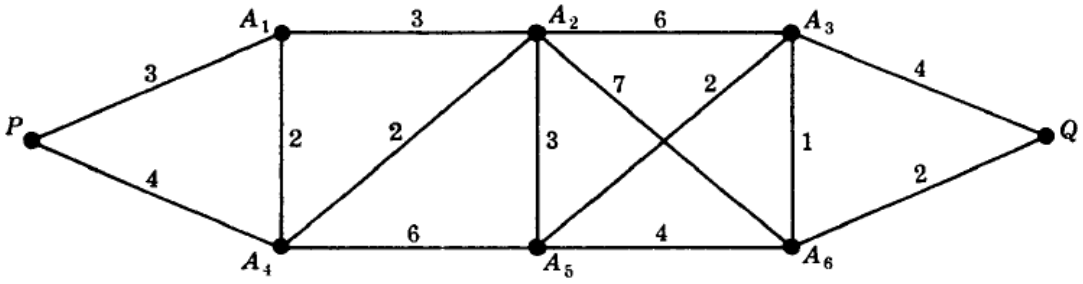
Figure 13

Labeled And weighted graphs:

A graph G is called a labeled graph if its edges and/or vertices are assigned data. If each edge (e) is assigned a non-negative number L(e). Then L(e) is called the weight or length of e. The weight of a path in such a weighted graph G is defined to be the sum of the weights of the edges in the path.

One important problem in graph theory is to find a shortest path, that is, a path of minimum weight (length), between any two given vertices.

Example: find the minimum path between P & Q:



(P, A1, A2, A5, A3, A6, Q)

$$\sum_{P}^{Q} L(e) = 3 + 3 + 3 + 2 + 1 + 2 = 14$$

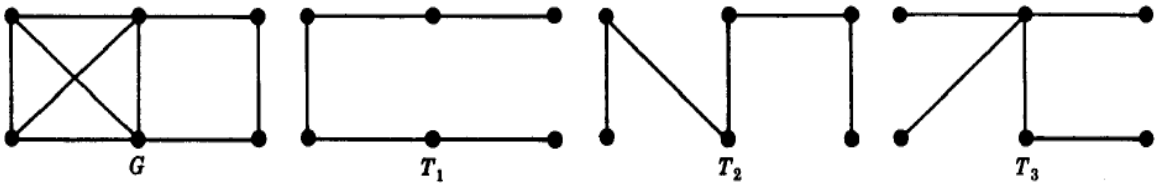
Another minimum path:

(P, A4, A2, A5, A3, A6, Q)

$$\sum_{P}^{Q} L(e) = 4 + 2 + 3 + 2 + 1 + 2 = 14$$

Spanning Trees

A subgraph T of a connected graph G is called a spanning tree of G if T is a tree and T includes all the vertices of G.



Minimum Spanning Trees

Suppose G is a connected weighted graph. That is, each edge of G is assigned a nonnegative number called the weight of the edge. Then any spanning tree T of G is assigned a total weight obtained by adding the weights of the edges in T . A minimal

spanning tree of G is a spanning tree whose total weight is as small as possible.

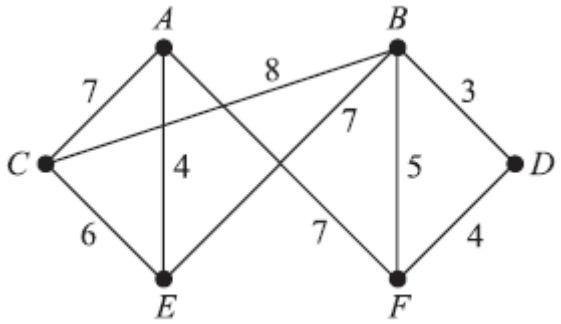
Algorithm 1 : The input is a connected weighted graph G with n vertices.

Step 1. Arrange the edges of G in the order of decreasing weights.

Step 2. Proceeding sequentially, delete each edge that does not disconnect the graph until $n-1$ edges remain.

Step 3. Exit.

EXAMPLE: Find a minimal spanning tree of the weighted graph Q, Note that Q has six vertices, so a spanning tree will have five edges.

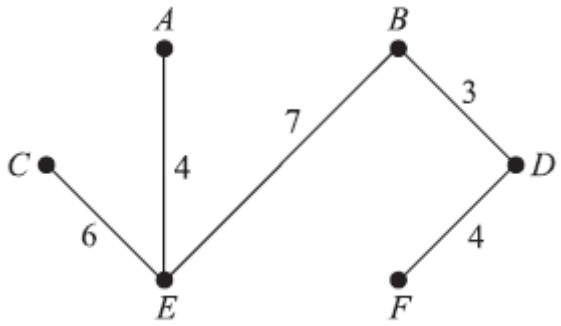


(a)

First we order the edges by decreasing weights, and then we successively delete edges without disconnecting Q until five edges remain. This yields the following data:

Edges:	BC	AF	AC	BE	CE	BF	AE	DF	BD
Weight	8	7	7	7	6	5	4	4	3
Delete	Yes	Yes	Yes	No	No	Yes			

Thus the minimal spanning tree of Q which is obtained contains the edges:
BE, CE, AE, DF, BD The spanning tree has weight 24



(b)

Algorithm 2: (Kruskal): The input is a connected weighted graph G with n vertices.

Step 1. Arrange the edges of G in order of increasing weights.

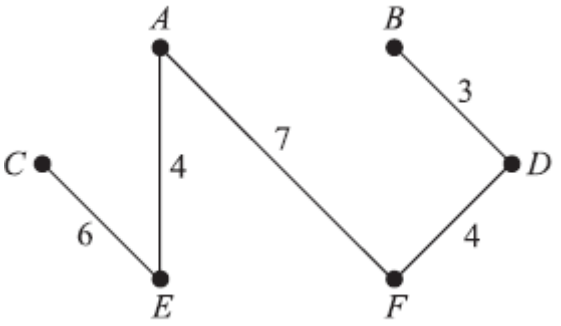
Step 2. Starting only with the vertices of G and proceeding sequentially, add each edge which does not result in a cycle until $n - 1$ edges are added.

Step 3. Exit.

First we order the edges by increasing weights, and then we successively add edges without forming any cycles until five edges are included. This yields the following data:

Edges	BD	AE	DF	BF	CE	AC	AF	BE	BC
Weight	3	4	4	5	6	7	7	7	8
Add?	Yes	Yes	Yes	No	Yes	No	Yes		

Thus the minimal spanning tree of Q which is obtained contains the edges:
BD, AE, DF, CE, AF
Observe that this spanning tree is not the same as the one obtained using Algorithm 1 as expected it also has weight 24.



(c)

REPRESENTING GRAPHS IN COMPUTER MEMORY:

There are many useful ways to represent graphs where in working with a graph it is helpful to be able to choose its most convenient representation.

(1)adjacency lists

EXAMPLE 1 Use adjacency lists to describe the simple graph given in Figure 14.

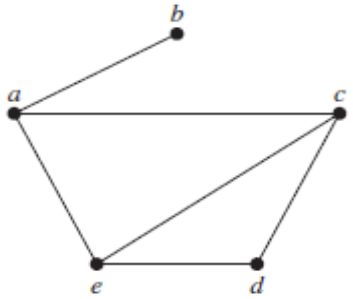


Figure 14

Solution: Table 1 lists those vertices adjacent to each of the vertices of the graph.

TABLE 1 An Adjacency List for a Simple Graph.

Vertex	*Adjacent Vertices*
a	b, c, e
b	a
c	a, d, e
d	c, e
e	a, c, d

EXAMPLE 2

Represent the directed graph shown in Figure 15 by adjacency lists

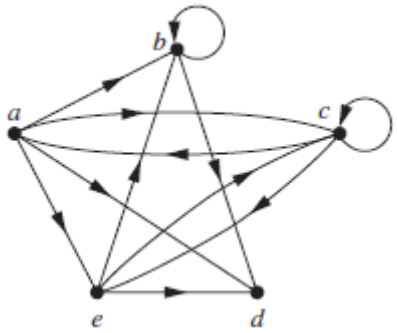


Figure 15

Solution: Table 2 represents the directed graph shown in Figure 15.

TABLE 2 An Adjacency List for a Directed Graph.

Initial Vertex	*Terminal Vertices*
a	b, c, d, e
b	b, d
c	a, c, e
d	
e	b, c, d

(2)Adjacency Matrices

Carrying out graph algorithms using the representation of graphs by adjacency lists, can be cumbersome if there are many edges in the graph. To simplify computation, graphs can be represented using matrices. Two types of matrices commonly used to represent graphs will be presented here. One is based on the adjacency of vertices, and the other is based on incidence of vertices and edges.

Suppose that $G = (V, E)$ is a simple graph where $|V| = n$. The **adjacency matrix A** of G, is the n x n zero–one matrix with 1 as its (i, j)th entry when v_i and v_j are adjacent, and 0 as its (i, j)th entry when they are not adjacent.

$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise.} \end{cases}$$

EXAMPLE 3 Use an adjacency matrix to represent the graph shown in Figure 16.

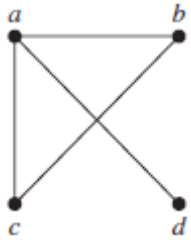


Figure 16

Solution:
We order the vertices as *a*, *b*, *c*, *d*. The matrix representing this graph is

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

EXAMPLE 4 Draw a graph with the following adjacency matrix

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

Solution: A graph with this adjacency matrix is shown in Figure 17.

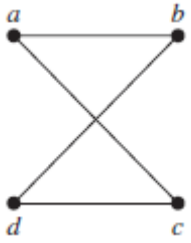


Figure 17

The adjacency matrix of a simple graph is symmetric, that is, $a_{ij} = a_{ji}$, because both of these entries are 1 when v_i and v_j are adjacent, and both are 0 otherwise. Furthermore, because a simple graph has no loops, each entry a_{ii}, $i = 1, 2, 3, \ldots, n$, is 0.

Adjacency matrices can also be used to represent undirected graphs with loops and with multiple edges. A loop at the vertex v_i is represented by a 1 at the (i, i)th position of the adjacency matrix. When multiple edges connecting the same pair of vertices v_i and v_j, or multiple loops at the same vertex, are present, the adjacency matrix is no longer a zero–one matrix,

because the *(i, j)*th entry of this matrix equals the number of edges that are associated to {*vi* , *vj* }.

EXAMPLE 5: Use an adjacency matrix to represent the multigraph shown in Figure 18.

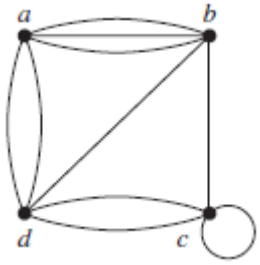


Figure 18

Solution: The adjacency matrix using the ordering of vertices *a, b, c, d* is:

$$\begin{bmatrix} 0 & 3 & 0 & 2 \\ 3 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \\ 2 & 1 & 2 & 0 \end{bmatrix}.$$

The adjacency matrix for a directed graph does not have to be symmetric, because there may not be an edge from *vj* to *vi* when there is an edge from *vi* to *vj* .

TRADE-OFFS BETWEEN ADJACENCY LISTS AND ADJACENCY MATRICES

When a simple graph contains relatively few edges, that is, when it is sparse, it is usually preferable to use adjacency lists rather than an adjacency matrix to represent the graph.

(3)Incidence Matrices

Another common way to represent graphs is to use **incidence matrices**. Let *G = (V ,E)* be an undirected graph. Suppose that *v*1, *v*2, . . . , *vn* are the vertices and *e*1, *e*2, . . . , *em* are the edges of *G*. Then the incidence matrix with respect to this ordering of *V* and *E* is the $n \times m$ matrix **M** = [*mij*], where:

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise.} \end{cases}$$

EXAMPLE 6: Represent the graph shown in Figure 19 with an incidence matrix.

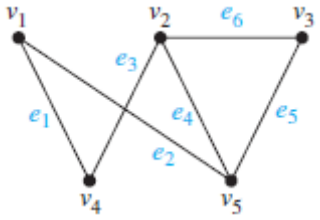
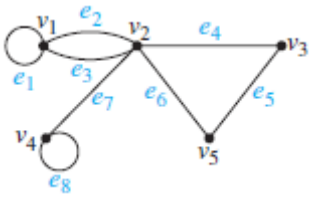


Figure 19

Solution: The incidence matrix is

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \end{array}.$$

EXAMPLE 7: Represent the multigraph shown in the following figure using an incidence matrix.



Solution: The incidence matrix for this graph is

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{cccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}.$$

Rooted tree:

Recall that a tree graph is a connected cycle-free graph, that is, a connected graph without any cycles. A *rooted tree* T is a tree graph with a designated vertex r called the *root* of the tree.

Consider a rooted tree *T* with root *r*. The length of the path from the root *r* to any vertex *v* is called the *level* (or *depth*) of *v*, and the maximum vertex level is called the *depth* of the tree.

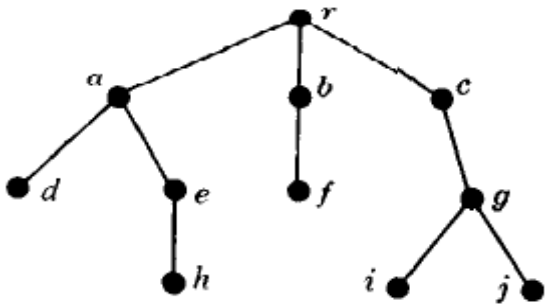


Figure 20

Those vertices with degree 1, other than the root *r*, are called the *leaves* of *T*.

One usually draws a picture of a rooted tree *T* with the root at the top of the tree.

Figure 20 shows a rooted tree *T* with root *r* and 10 other vertices. The tree has five leaves, *d,f, h, i,* and *j* . Observe that: *level(a)* = 1, *level(f)* = 2, *level(j)* = 3. Furthermore, the depth of the tree is 3.

EXAMPLE 8:

Suppose Marc and Erik are playing a tennis tournament such that the first person to win two games in a row or who wins a total of three games wins the tournament. Find the number of ways the tournament can proceed.

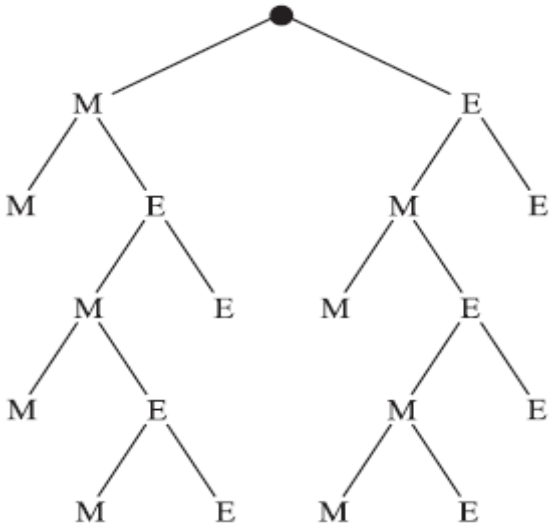
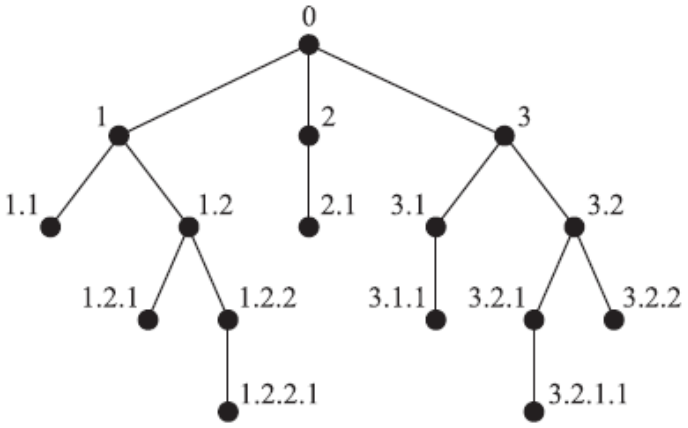


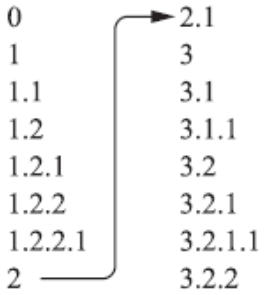
Figure 21

The rooted tree in Fig.21 shows the various ways that the tournament could proceed. There are 10 leaves which correspond to the 10 ways that the tournament can occur:
MM, **MEMM**, **MEMEM**, **MEMEE**, **MEE**, **EMM**, **EMEMM**, **EMEME**, **EMEE**, **EE**
Specifically, the path from the root to the leaf describes who won which games in the particular tournament.

Order Rooted Tree (ORT):

Whenever draw the digraph of a tree, we assume some ordering at each level, by arranging children from left to right. Where identical to the order obtained by moving down the leftmost branch of the tree, then the next branch to the right, then the second branch to the right, and so on.





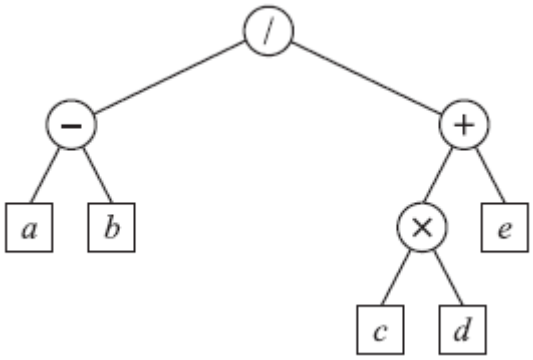
Degree of tree: The largest number of children in the vertices of the tree
Binary tree : every vertex has at most 2 children

Algebraic Expressions and Polish Notation

Any algebraic expression involving binary operations +, -, ×, ÷ can be represented by an order rooted tree (ORT).
Let *E* be any algebraic expression which uses only binary operations, such as:

$$E = (a - b)/((c \times d) + e)$$

Then *E* can be represented by a tree as



where the variables in *E* appear as the external nodes, and the operations in *E* appear as internal nodes.

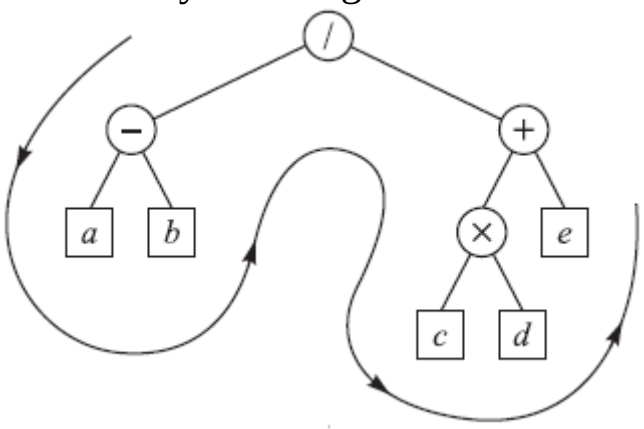
The Polish mathematician Lukasiewicz observed that by placing the binary operation symbol before its arguments, e.g.:

$+ab$ instead of $a + b$ and $/cd$ instead of c/d

one does not need to use any parentheses. This notation is called *Polish notation in prefix form*. (one can place the symbol after its arguments, called *Polish notation in postfix form*.) Rewriting *E* in prefix form we obtain:

$$E = / - a\ b + \times c\ d\ e$$

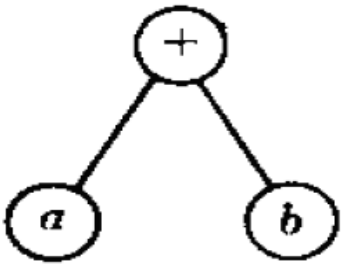
Observe that this is precisely the order of the vertices in its tree which can be obtained by scanning the tree as :



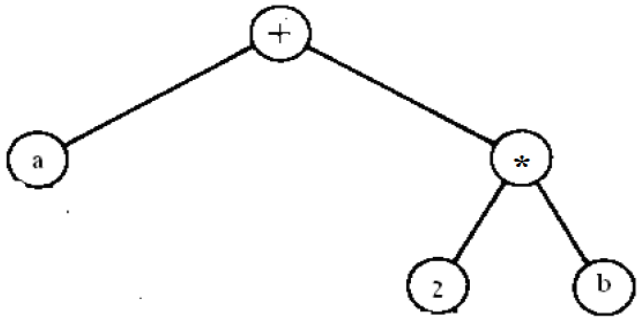
The polish notation form of an algebraic expression represents the expression unambiguously without the need for parentheses

1) a + b (infix)
2) + a b (prefix)
3) a b + (postfix)

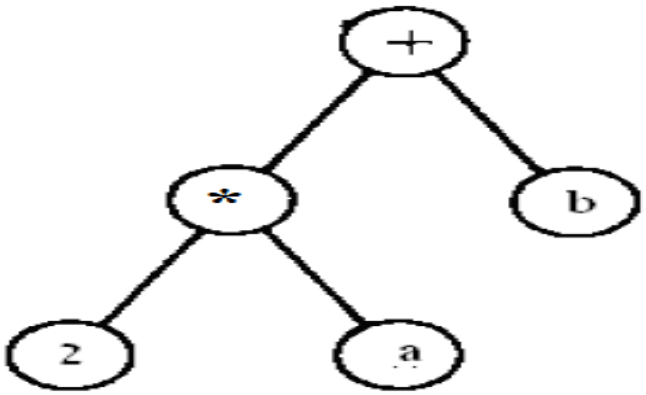
Example 1:
infix polish notation is : a + b
prefix polish notation : + a b



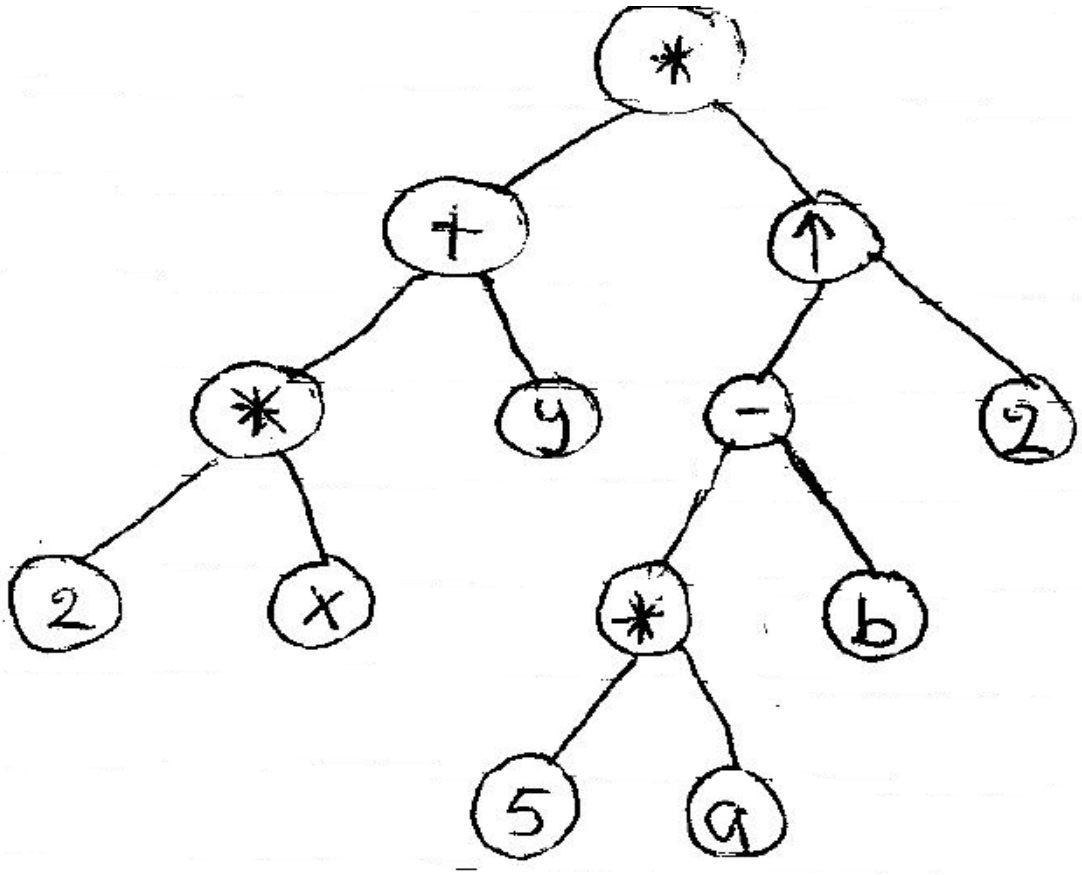
example 2:
infix polish notation is : a + 2 * b
prefix polish notation : + a * 2 b



example 3:
infix polish notation is : 2 * a + b
prefix polish notation : + * 2 a b



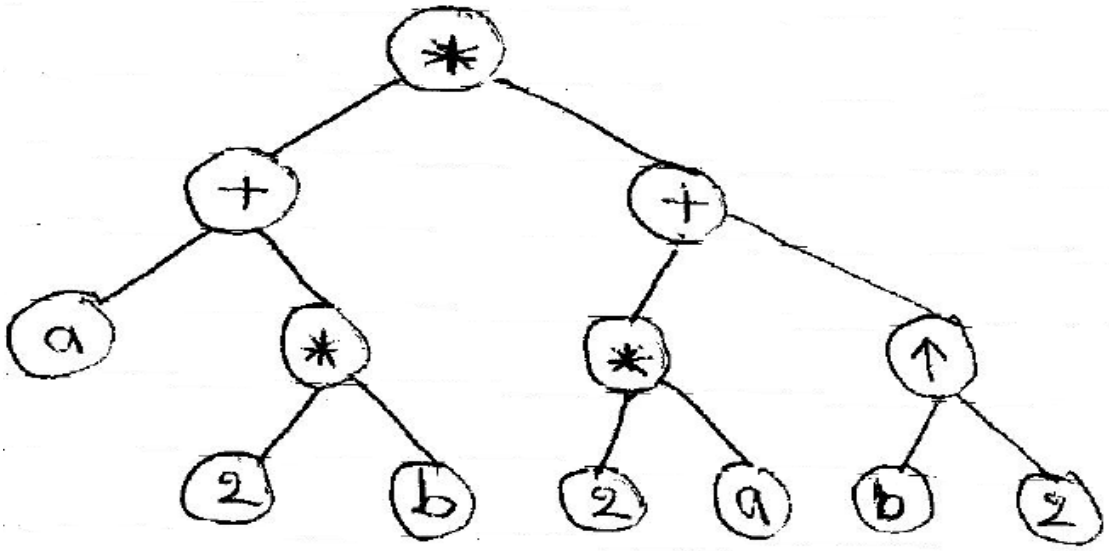
example 4:
infix polish notation is : (2 * x + y).(5 * a – b)^2
prefix polish notation : * + * 2 x y ^ - * 5 a b 2



example 5:

infix polish notation is : (a + 2 * b) (2 * a + b^2)

prefix polish notation : * + a * 2 b + * 2 a ^ b 2



To evaluate an expression in polish form proceed as follows:

a) move from left to right until we find a simple string of the form Pxy, where P is the symbol for a binary operations: (+,-,× , /) and x & y are numbers.
b) Evaluate xPy and substitute the answer.
c) Continue this procedure until only one number remains.

Example:
evaluate the value of the expression (a-b)×(c+(d/e)), if a=6, b=4, c=5, d=2 and e=2

Prefix: * - a b+ c /d e
To evaluate: * - 6 4+ 5 /2 2

a) *- 6 4 +5 / 2 2
b) *2 + 5 / 2 2
c) * 2 + 5 1
d) * 2 6
e) 12

Homework:
Rewrite the following expressions into prefix polish notation form, construct their corresponding ORT and evaluate their value

(3*(1-x))/((4+(7-(y+2)))*(7+(x/y)))
(3-(2+x))+((x-2) –(3+x))

Finite state machines (FSM):

We may view a digital computer as a machine which is in a certain "internal state" at any given moment. The computer "reads" an input symbol, and then "prints" an output symbol and changes its "state". The output symbol depends solely upon the input symbol and the internal state of the machine, and the internal state of the machine depends solely upon the preceding state of the machine and the preceding input symbol.

A finite state machine FSM (or complete sequential machine) M consists of five things:

(1) A finite set A of input symbols.
(2) A finite set S of internal states.
(3) A finite set Z of output symbols.
(4) An initial state *s*0 in *S*.
(5) A next-state function f $f: S \times A \rightarrow S$
(6) An output function g $g: S \times A \rightarrow Z$

This machine M is denoted by M = (A, S, Z, q0, f, g) where q0 is the initial state.

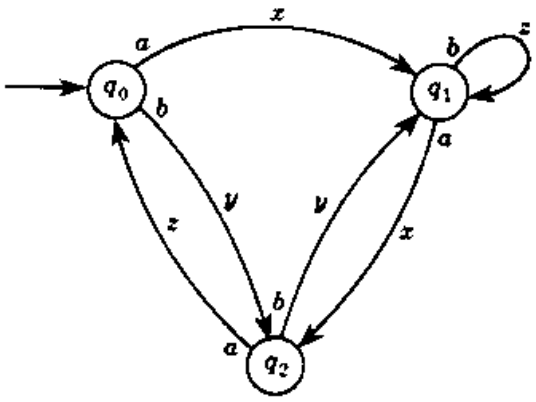
Example 1:

The following defines a FSM with two input symbols, three internal states and three output symbols:

(1) A = {a, b}
(2) S = {q0, q1, q2}
(3) Z = {x, y, z}
(4) Next-state function $f: S \times A \rightarrow S$ defined by :
    f(q0, a) = q1 f(q1,a) = q2
    f(q2,a) = q0 f(q0, b) = q2
    f(q1, b) = q1 f(q2, b) = q1
(5) Output function $g: S \times A \rightarrow Z$ defined by
    g(q0,a) = x g(q1,a) = x
    g(q2,a) = z g(q0, b) = y
    g(q1, b) = z g(q2, b) = y

There are two ways of representing a finite state machine in compact form. One way is by a table called the **state table** of

machine, and the other way is by a labeled directed graph called the **state diagram** of the machine.



	a	b
q_0	q_1, x	q_2, y
q_1	q_2, x	q_1, z
q_2	q_0, z	q_1, y

State diagraph **State table**

We visualize these symbols on an "input tape." The machine *M* "reads" these input symbols one by one and, simultaneously, changes through a sequence of states.

If the input string**:** **abaab ,** is given to the machine in example (1), and suppose q0 is the initial state of the machine.

We calculate the string of states and the string of output symbols from the state diagram by beginning at the vertex q0 and following the arrows which are labeled with the input symbols:

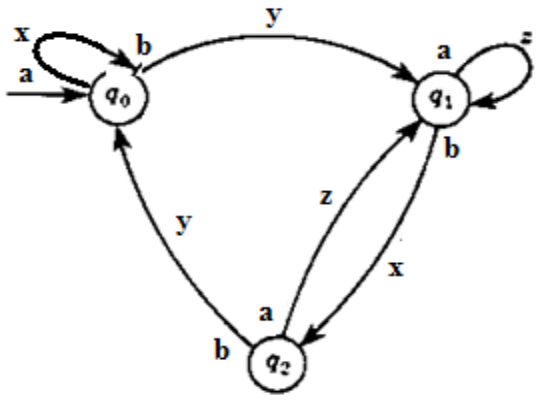
$$q_0 \xrightarrow{a,x} q_1 \xrightarrow{b,z} q_1 \xrightarrow{a,x} q_2 \xrightarrow{a,z} q_0 \xrightarrow{b,y} q_2$$

This yields the following strings of states and output symbols:

State : q0 q1 q1 q2 q0 q2

Output symbols : x z x z y

Homework:
Draw the state table for the following FSM and Trace it with the input: aaabbb, and abaab.



Example 2:
Design a **FSM** which can do binary addition
we can assume that our numbers have the same number of digits.
If the machine is given the input:

1101011
+ 0111011

then we want the output to be the binary sum 10100110.
Specifically, the input is the string of pairs of digits to be added:

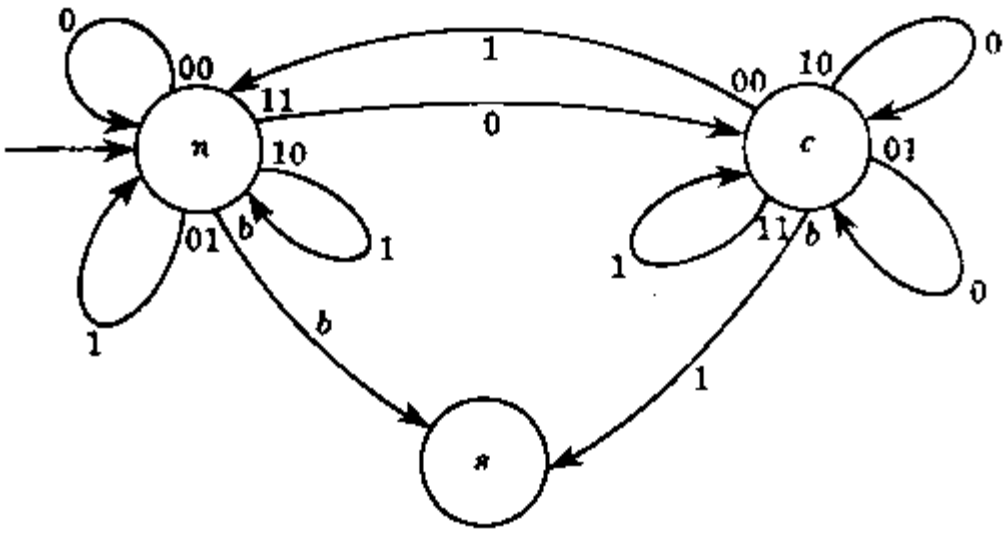
11, 11, 00, 11, 01, 11, 10, b (where b denotes blank spaces)

and the output should be the string:

0, 1, 1, 0, 0, 1, 0, 1

We also want the machine to enter a state called "stop" when the machine finishes the addition.
The input symbols and output symbols are, respectively, as follows:
$A = \{00, 01, 10, 11, b\}$ and $Z = \{0, 1, b\}$

The machine *M* that we "construct" will have three states:
$S = \{$carry *(c)*, no carry *(n)*, stop *(s)*$\}$
Here *n* is the initial state.



FINITE AUTOMATA

A finite automaton is similar to a finite state machine except that an automation has "accepting" and rejecting" states rather than an output. Specifically, a finite automaton M consists of five things:

(1) A finite set A of input symbols
(2) A finite set S of internal states
(3) A subset T of S (whose elements called accepting states)
(4) An initial state q0 in S
(5) A next-state function f from S × A ino S.

The automaton M is denoted by M = (A, S, T, q0, f) when we want to designate its five parts

We can concisely describe a finite automaton M by its state diagram as was done with finite state machines, except that here we use double circles for accepting states and each edge is labeled only by the input symbol. Specifically, the state diagram D of M is a labeled directed graph whose vertices are the states of S where accepting states are labeled by having a double circle, and if f(qj, ai) = qk then there is an arc from qj to qk which is labeled with ai. Also the initial state q0 is denoted by having an arrow entering the vertex q0.

We say that M recognizes or accepts the string W if the final state sn is an accepting state, i. e. if sn $\in$T. We will let L(M) denote the set of all strings which are recognized by M.

Example

The following defines a finite automaton with two input symbols and three states:

(1) A = {a,b}, input symbols
(2) S = {q0, q1, q2}, states
(3) T = {q0, q1}, accepting states
(4) q0, the initial state.
(5) Next-state function f : S × A → S defined by:

f(q0,a) = q0, f(q1,a)=q0, f(q2,a)=q2
f(q0,b)=q1, f(q1,b)=q2, f(q2,b)=q2
or by the table:

f	a	b
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_2	q_2

The automaton M will recognize those strings which do not have two successive b's. Thus M will accept:

aababaaba, aaa, baab, abaaababab, b, aabaaab

But will reject :

aabaabba, bbaaa, ababbaab, bb, abbbbaa

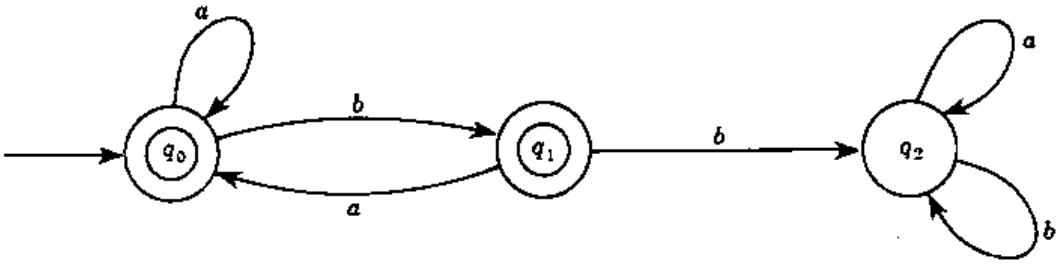


Fig. 1

Language *L(M)* Determined by an Automaton *M*

Each automaton *M* with input alphabet *A* defines a language over *A*, denoted by *L(M)*. We say that *M recognizes* the word *w* if the final state s_m is an accepting state in *Y* . The language *L(M)* of *M* is the collection of all words from *A* which are accepted by *M*.

EXAMPLE

Determine whether or not the automaton *M* in Fig. 1 accepts the words:

$w1 = ababba$; $w2 = baab$; $w3 = \lambda$ (empty word)

Use Fig. 1 and the words *w*1 and *w*2 to obtain the following respective paths:

$$P1 = s0 \xrightarrow{a} s0 \xrightarrow{b} s1 \xrightarrow{a} s0 \xrightarrow{b} s1 \xrightarrow{b} s2 \xrightarrow{a} s2$$

$$P2 = s0 \xrightarrow{b} s1 \xrightarrow{a} s0 \xrightarrow{a} s0 \xrightarrow{b} s1$$

The final state in P_1 is s_2 which is not in *Y* ; hence w_1 is not accepted by *M*. On the other hand, the final state in P_2 is s_1 which is in *Y* ; hence w_2 is accepted by *M*. The final state determined by w_3 is the initial state s_0 since $w_3 = \lambda$ is the empty word. Thus w_3 is accepted by *M* since $s_0 \in Y$.

EXAMPLE

Describe the language *L(M)* of the automaton *M* in Fig. 1.

L(M) will consist of all words *w* on *A* which do not have two successive *b*'s. This comes from the following facts:

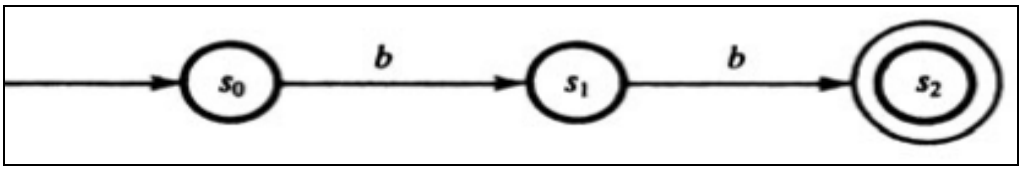
(1) We can enter the state q_2 if and only if there are two successive *b*'s.

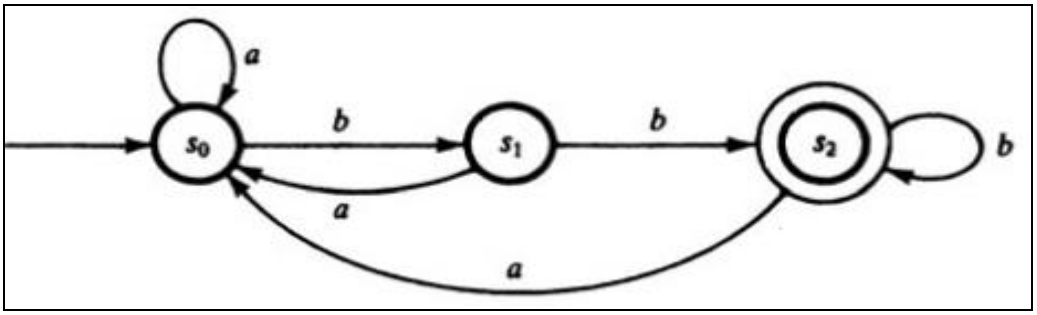
(2) We can never leave *q*2.

(3) The state q_2 is the only rejecting (nonaccepting) state.

EXAMPLE

Let A = {a, b}. Construct an automaton M which will accept precisely those words from A which end in two b's.





Since *bb* is accepted, but not λ or *b*, we need three states, s_0, the initial state, and s_1 and s_2 with an arrow labeled *b* going from s_0 to s_1 and one from s_1 to s_2. Also, s_2 is an accepting state, but not s_0 nor s_1.

On the other hand, if there is an *a*, then we want to go back to s_0, and if we are in s_2 and there is a *b*, then we want to stay in s_2. These additional conditions give the required automaton *M*.

EXAMPLE

Construct an automaton M with input symbols a and b, which only accept those string such that the number of b's is divisible by 3.

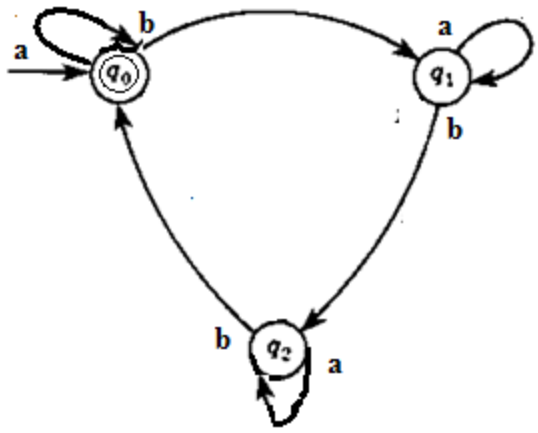
A= {a, b}

S= {q0, q1, q2}

T= {q0}

Accepted symbols: ababaab, baabab, bbabbbba, aa, aabbaab

Rejected symbols: ab, ababbb



	a	b
q0	q0	q1
q	q1	q2
q2	q2	q0

Some Examples of FSM

We study examples of finite state machines that are designed to recognize given patterns.

As there is essentially no standard way of constructing such machines, we shall illustrate the underlying ideas by examples.

Example 1:

Suppose that *A (input)* = Z (output) = *{*0, 1*}*, and that we want to design a finite state machine that recognizes the sequence pattern 11 in the input string $x \in A*$. An example of an input string $x \in A*$ and its corresponding output string $y \in Z*$ is shown below:

x = 10111010101111110101

y = 00011000000111110000

Note that the output digit is 0 when the sequence pattern 11 is not detected and 1 when the sequence pattern 11 is detected. In order to achieve this, we must ensure that the finite state machine has at least two states, a "passive" state when the previous entry is 0 (or

when no entry has yet been made), and an "excited" state when the previous entry is 1. Furthermore, the finite state machine has to observe the following and take the corresponding actions:

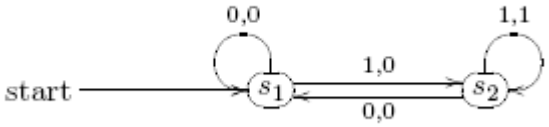
(1) If it is in its "passive" state and the next entry is 0, it gives an output 0 and remains in its "passive" state.

(2) If it is in its "passive" state and the next entry is 1, it gives an output 0 and switches to its "excited" state.

(3) If it is in its "excited" state and the next entry is 0, it gives an output 0 and switches to its "passive" state.

(4) If it is in its "excited" state and the next entry is 1, it gives an output 1 and remains in its "excited" state.

It follows that if we denote by $s1$ the "passive" state and by $s2$ the "excited" state, then we have the state diagram below:



We then have the corresponding transition table:

	g 0 1	f 0 **1**
$+s_1+$	0 0	s_1 s_2
s_2	0 1	s_1 s_2

Example 2:

Suppose again that *A (input)* = Z (output) = *{*0, 1*}*, and that we want to design a finite state machine that recognizes the sequence pattern 111 in the input string $x \in A*$. An example of the same input string $x \in A*$ and its corresponding output string $y \in Z*$ is shown below:

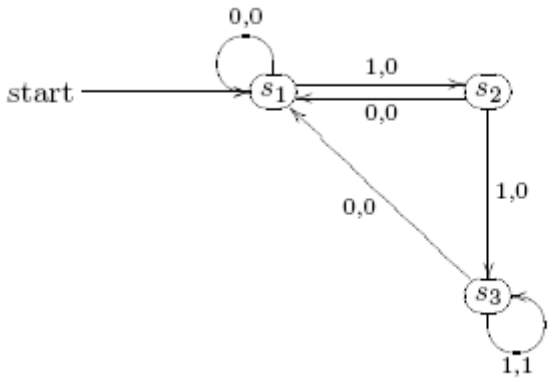
x = 10111010101111110101

y = 00001000000011110000

In order to achieve this, the finite state machine must now have at least three states, a "passive" state when the previous entry is 0 (or when no entry has yet been made), an "expectant" state when the previous two entries are 01 (or when only one entry has so far been made and it is 1), and an "excited" state when the previous two entries are 11. Furthermore, the finite state machine has to observe the following and take the corresponding actions:

(1) If it is in its "passive" state and the next entry is 0, it gives an output 0 and remains in its "passive" state.
(2) If it is in its "passive" state and the next entry is 1, it gives an output 0 and switches to its "expectant" state.
(3) If it is in its "expectant" state and the next entry is 0, it gives an output 0 and switches to its "passive" state.
(4) If it is in its "expectant" state and the next entry is 1, it gives an output 0 and switches to its "excited" state.
(5) If it is in its "excited" state and the next entry is 0, it gives an output 0 and switches to its "passive" state.
(6) If it is in its "excited" state and the next entry is 1, it gives an output 1 and remains in its "excited" state.

If we now denote by $s1$ the "passive" state, by $s2$ the "expectant" state and by $s3$ the "excited" state, then we have the state diagram below:



We then have the corresponding transition table:

	g		f	
	0	1	0	1
$+s_1+$	0	0	s_1	s_2
s_2	0	0	s_1	s_3
s_3	0	1	s_1	s_3

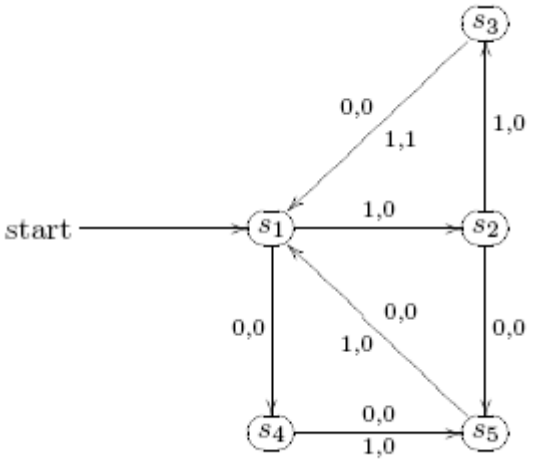
Example 3:

Suppose again that *A (input)* = Z (output) = *{*0, 1*}*, and that we want to design a finite state machine that recognizes the sequence pattern 111 in the input string $x \in A*$, but only when the third 1 in the sequence pattern 111 occurs at a position that is a multiple of 3. An example of the same input string $x \in A*$ and its corresponding output string $y \in Z*$ is shown below:

x = 10111010101111110101

y = 00000000000000100000

In order to achieve this, the finite state machine must as before have at least three states, a "passive" state when the previous entry is 0 (or when no entry has yet been made), an "expectant" state when the previous two entries are 01 (or when only one entry has so far been made and it is 1), and an "excited" state when the previous two entries are 11. However, this is not enough, as we also need to keep track of the entries to ensure that the position of the first 1 in the sequence pattern 111 occurs at a position immediately after a multiple of 3. It follows that if we permit the machine to be at its "passive" state only either at the start or after 3k entries, where $k \in$ N, then it is necessary to have two further states to cater for the possibilities of 0 in at least one of the two entries after a "passive" state and to then delay the return to this "passive" state. We can have the state diagram below:



We then have the corresponding transition table:

	g		f	
	0	1	0	1
$+s_1+$	0	0	s_4	s_2
s_2	0	0	s_5	s_3
s_3	0	1	s_1	s_1
s_4	0	0	s_5	s_5
s_5	0	0	s_1	s_1

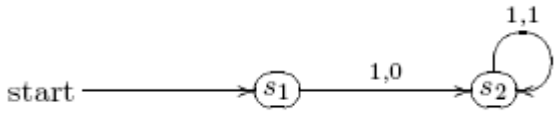
An Optimistic Approach

We construct first of all the part of the machine to take care of the situation when the required pattern occurs repeatly and without interruption. We then complete the machine by studying the situation when the "wrong" input is made at each state.

.

Example 1:

Suppose that *A (input)* = Z (output) = {0, 1*}*, and that we want to design a finite state machine that recognizes the sequence pattern 11 in the input string $x \in A*$. Consider first of all the situation when the required pattern occurs repeatly and without interruption. In other words, consider the situation when the input string is 111111 . . . . To describe this situation, we have the following incomplete state diagram:

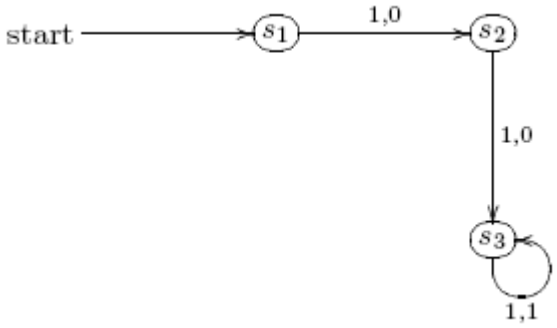


It now remains to study the situation when we have the "wrong" input at each state. Naturally, with a "wrong" input, the output is always 0, so the only unresolved question is to determine the next state.

Note that whenever we get an input 0, the process starts all over again; in other words, we must return to state *s*1.. We therefore obtain the state diagram as in Example 1.

Example 2:

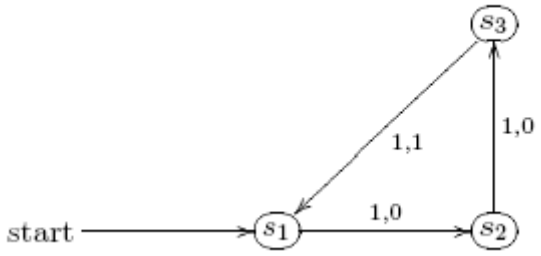
Suppose again that *A (input)* = Z (output) = *{*0, 1*}*, and that we want to design a finite state machine that recognizes the sequence pattern 111 in the input string $x \in A*$. Consider first of all the situation when the required pattern occurs repeatly and without interruption. In other words, consider the situation when the input string is 111111 . . . . To describe this situation, we have the following incomplete state diagram:



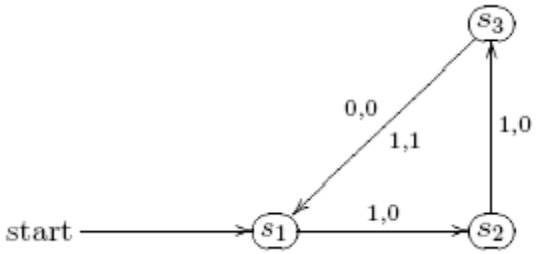
It now remains to study the situation when we have the "wrong" input at each state. As before, with a "wrong" input, the output is always 0, so the only unresolved question is to determine the next state.

Note that whenever we get an input 0, the process starts all over again; in other words, we must return to state *s*1. We therefore obtain the state diagram as Example 2.

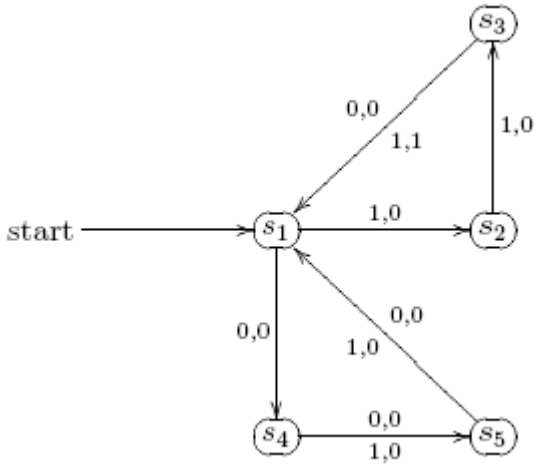
Example 3:
Suppose again that *A (input)* = Z (output) = *{*0, 1*}*, and that we want to design a finite state machine that recognizes the sequence pattern 111 in the input string $x \in A*$, but only when the third 1 in the sequence pattern 111 occurs at a position that is a multiple of 3. Consider first of all the situation when the required pattern occurs repeatly and without interruption. In other words, consider the situation when the input string is 111111 . . . . To describe this situation, we have the following incomplete state diagram:



It now remains to study the situation when we have the "wrong" input at each state. As before, with "wrong" input, the output is always 0, so the only unresolved question is to determine the next state. Suppose that the machine is at state *s*3, and the wrong input 0 is made. We note that if the next three entries are 111, then that pattern should be recognized. It follows that *f*(*s*3, 0) = *s*1. We now have the following incomplete state diagram:
It now remains to study the situation when we have the "wrong" input at each state. As before, with a "wrong" input, the output is always 0, so the only unresolved question is to determine the next state.
Suppose that the machine is at state *s*3, and the wrong input 0 is made. We note that if the next three entries are 111, then that pattern should be recognized. It follows that *f*(*s*3, 0) = *s*1. We now have the following incomplete state diagram:



Suppose that the machine is at state $s1$, and the wrong input 0 is made. We note that the next two inputs cannot contribute to any pattern, so we need to delay by two steps before returning to $s1$. We now have the following incomplete state diagram:



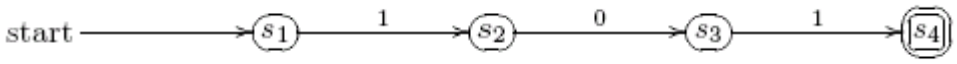
Finally, it remains to examine $f(s2, 0)$. We therefore obtain the state diagram as Example 3.

Note that in Example 3, in dealing with the input 0 at state $s1$, we have actually introduced the extra states $s4$ and $s5$. These extra states are not actually involved with positive identification of the desired pattern, but are essential in delaying the return to one of the states already present. It follows that at these extra states, the output should always be 0. However, we have to investigate the situation for input 0 and 1.

Deterministic Finite State Automata

we discuss a slightly different version of finite state machines which is closely related to regular languages. We begin by an example which helps to illustrate the changes.

Example.1. We shall construct a deterministic finite state automaton which will recognize the
input strings 101 and nothing else. This automaton can be described by the following state
diagram:



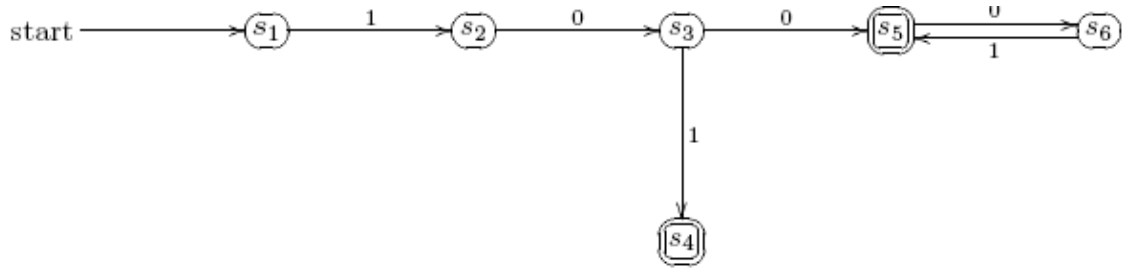
We can also describe the same information in the following transition table:

	ν	
	0	1
$+s_1+$	–	s_2
s_2	s_3	–
s_3	–	s_4
$-s_4-$	–	–

We now modify our definition of a finite state machine accordingly.
Definition. A deterministic finite state automaton is a 5-tuple $A = (S, I, \nu, T, s1)$, where
(a) S is the finite set of states for A;
(b) I is the finite input alphabet for A;
(c) $\nu : S \times I \rightarrow S$ is the next-state function;
(d) T is a non-empty subset of S; and
(e) $s1 \in S$ is the starting state.

Remarks.
(1) The states in T are usually called the accepting states.
(2) If not indicated otherwise, we shall always take state $s1$ as the starting state.

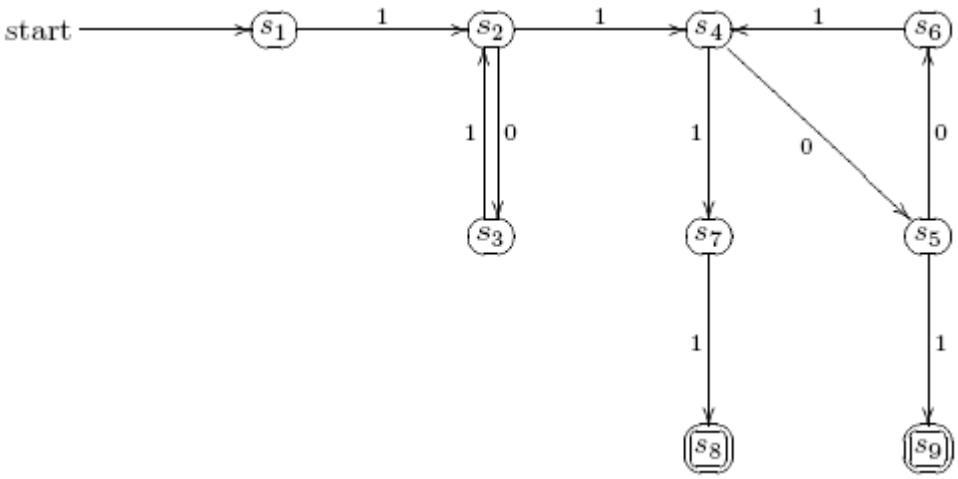
Example.2. We shall construct a deterministic finite state automaton which will recognize the
input strings 101 and $100(01)*$ and nothing else. This automaton can be described by the following state diagram:



We can also describe the same information in the following transition table:

	ν	
	0	1
$+s_1+$	—	s_2
s_2	s_3	—
s_3	s_5	s_4
$-s_4-$	—	—
$-s_5-$	s_6	—
s_6	—	s_5

Example .3. We shall construct a deterministic finite state automaton which will recognize the input strings:
$1(01)*1(001)*(0+1)1$ and nothing else. This automaton can be described by the following state diagram:



We can also describe the same information in the following transition table:

	ν	
	0	1
$+s_1+$	–	s_2
s_2	s_3	s_4
s_3	–	s_2
s_4	s_5	s_7
s_5	s_6	s_9
s_6	–	s_4
s_7	–	s_8
$-s_8-$	–	–
$-s_9-$	–	–

–