Basic Graph Theory

Arpit Kumar(05CS1032)

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur, India
arpitk.iitkgp@gmail.com

1 Introduction

This lecture deals with basic graph theory. In the subsequent lectures we would frequently use the terminologies defined here.

A graph is a symbolic representation of a network and of its connectivity. It implies an abstraction of the reality and can be simplified as a set of linked nodes. *Graph theory is a branch of mathematics concerned about how networks can be encoded and their properties measured.*

1.1 Origin of graph theory:

The origin of graph theory can be traced back to Euler's work on the Konigsberg bridges problem (1735), which subsequently led to the concept of an Eulerian graph. The study of cycles on polyhedra by the Thomas P. Kirkman (1806 - 95) and William R. Hamilton (1805-65) led to the concept of a Hamiltonian graph. The development of graph theory is very similar to the development of probability theory, where much of the original work was motivated by efforts to understand games of chance. The large portions of graph theory have been motivated by the study of games and recreational mathematics.

1.2 Utilities of graph theory:

How can we lay cable at minimum cost to make every telephone reachable from from every other? What is the fastest route from national capital to each state capital? How can n jobs be filled by na people with maximum total utility? What is the maximum flow per unit time from source to sink in a network of pipes? How many layers does a computer chip need so that wires in the same layer don't cross? In what order should a travelling salesman visit cities to minimize travel time? Can we color the regions of every map

using four colors so that neighboring regions receive different colors? These and many other practical problems involve graph theory.

Section 2 introduces some of the terms, which are used frequently in graph theory. These form the basic concepts of graph theory. Section 3 explains some of the very important algorithms of graphs, each with a suitable example. Sections 4 and 5 describe the important properties of adjacency and incidence matrices respectively. Section 6 deals with **Planar graphs**, which have a lot of practical applications. In section 7, we see **Flow Networks**, which enable us to determine what can be the maximum flow from a source to sink in a network of pipes and reservoirs. Finally we conclude our chapter in section 8.

2 Terms and Definitions

2.1 Graph

A **graph** G is a triple consisting of a **vertex set** $V(G) = \{v_1....v_n\}$, an **edge set** $E(G) = \{e_1....e_m\}$, and a relation that associates with each edge two vertices (not necessarily distinct) called its **endpoints**.

A **loop** is an edge whose endpoints are equal. **Multiple edges** are edges having the same pair of endpoints.

A **simple graph** is a graph having no loops or multiple edges. We specify a simple graph by its vertex set and edge set, treating the edge set as a set of unordered pairs of vertices and writing (u, v) or (v, u) for an edge e with endpoints u and v.

When u and v are the endpoints of an edge, they are **adjacent** and are **neighbors**. We write $u \leftrightarrow v$ for "u is adjacent to v".

If vertex v is and endpoint of edge e, then v and e are **incident**. The **degree** of vertex v (in a loopless graph) is the number of incident edges.

Figure 1 shows a simple undirected graph.

2.2 Walk, Trail, Path and Cycle

A **walk** is a list $v_0, e_1, v_1, ..., e_k, v_k$ of vertices and edges such that ,for $1 \leq i \leq k$,the edge e_i has endpoints v_{i-1} and v_i. A **trail** is a walk with no repeated edge. A walk or trail is **close** if its endpoint are the same. A u, v**-path** in a graph is a sequence of vertices such that from each vertices there is an edge to the next vertex in the sequence. The first vertex u is called the **start** vertex and v is called the **end** vertex. The other vertices in the path are **internal** vertices. A **cycle** is a closed path, in which first = last is the only vertex repetition.

In figure 1:
1, (1, 2), 2, (2, 4), 4, (4, 3), 3, (3, 1), 1, (1, 2), 2 is a **walk** of length 5.

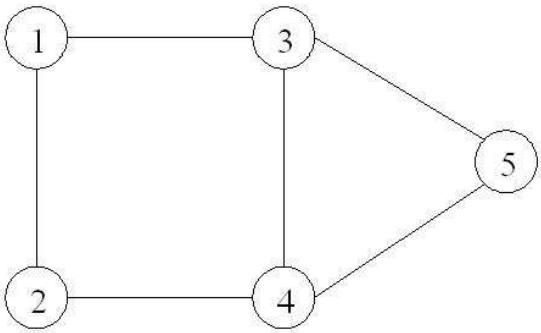


Fig. 1. A Simple Graph

1, (1, 2), 2, (2, 4), 4, (4, 3), 3, (3, 5), 5, (5, 4), 4 is a **trail** of length 5.
1, (1, 2), 2, (2, 4), 4, (4, 3), 3 is a **path** of length 3.
1, (1, 2), 2, (2, 4), 4, (4, 3), 3, (3, 1), 1 is a **cycle** of length 4.

Theorem 1.1: *Let G be a graph in which the degree of every vertex is at least 2. Then G contains a cycle.*

2.3 Diameter

If G has a u, v-path, then the distance from u to v, written as $d_G(u, v)$ or simply $d(u, v)$, is the least length of a u, v-path. If G has no such path, then $d(u, v) = \infty$. The **diameter** is the maximum of all such distances, when considered for each possible pair of vertices in the graph. Mathematically, it can be written as $max_{u,v \in V(G)} d(u, v)$.

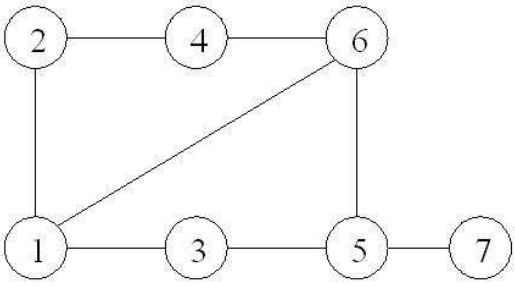


Fig. 2. Diameter of a graph

Consider the graph shown in Figure 2. The number of links (edges) between the furthest nodes (2 and 7) of this graph is 4. Consequently, the diameter of this graph is 4.

2.4 Graph Isomorphism

In graph theory, an isomorphism of graphs G and H is a bijection between the vertex sets of G and H

$f : V(G) \rightarrow V(H)$

such that any two vertices u and v of G are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H.

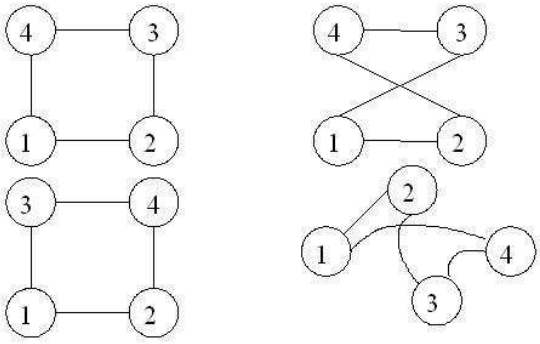


Fig. 3. Isomorphism

The graphs shown in Figure 3 are isomorphic, despite their different looking drawings.

2.5 Strongly and Weakly connected components

A graph G is **connected** if it has a u, v-path whenever $u, v \in V(G)$ (otherwise, G is **disconnected**). If G has a u, v-path, then u is connected to v in G.

The **components** of a graph G are its maximal connected subgraphs. A component(or graph) is **trivial** if it has no edges; otherwise it is **nontrivial**. An **isolated vertex** is a vertex of degree 0.

A directed graph is **weakly connected** if it would be connected by ignoring the direction of edges. Thus a weakly connected graph consists of a single piece.

A directed graph is called **strongly connected** if for every pair of vertices u and v there is a path from u to v and a path from v to u. The **strongly**

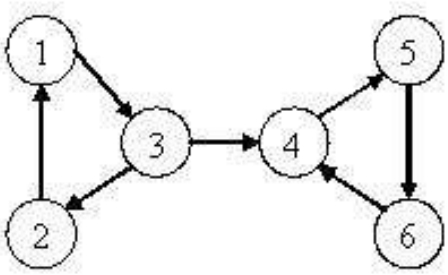


Fig. 4. Strongly Connected Components

connected components (SCC) of a directed graph are its maximal strongly connected subgraphs.

In figure 4, there are two strongly connected components: {1, 2, 3} and {4, 5, 6}. Note that if we ignore the direction of edges, the graph shown is weakly connected.

2.6 Bipartite Graphs

A **bipartite graph** is a graph consisting of two set of vertices, such that edges only run from one set to another, and there is no edge connecting two vertices from the same set. The two sets are known as the partitions. Figure 5 shows a bipartite graph with partitions X and Y. A **complete bipartite** graph is a bipartite graph in which two vertices are adjacent if and only if they are in different partite sets. We denote a complete bipartite graph as $K_{r,s}$, where r and s are the sizes of two sets.

Bipartite graphs have many real world applications. For example, in the figure shown, X can be the set of movie actors, and Y can be the set of movies. Then an edge from some X_i to Y_j means that actor X_i acted in movie Y_j. Of course, an edge from some X_i to X_j or from some Y_i to Y_j makes no sense.

Similarly, X can be the set of men, Y the set of women, and edges from X_i to Y_j would imply sexual relationships. If we assume a heterosexual society, then again there can be no edge between two members of X or two members of Y.

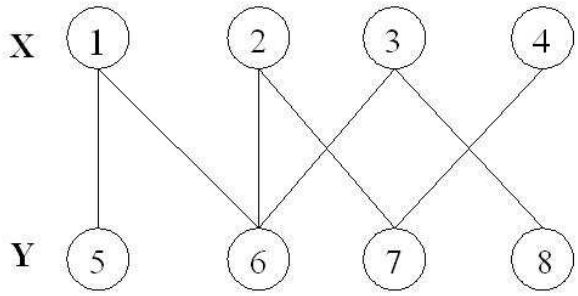


Fig. 5. Bipartite graph

2.7 Eulerian Circuits and Eulerian Trails

A graph is **Eulerian** if it has a closed trail containing all the edges. We call a closed trail a **circuit** when we do not specify the first vertex but keep the list in cyclic order. An **Eulerian circuit** or **Eulerian trail** in a graph is a circuit or trail containing all the edges.

Theorem 1.2: *A connected graph G has an Eulerian circuit iff the degree of every vertex is even.*

Theorem 1.3: *A connected graph G has an Eulerian trail iff it has at most 2 odd vertices, i.e. it has either no vertex of odd degree or exactly 2 vertices of odd degree.*

In 6(a), both an Eulerian trail(1-6-2-3-4-5-3-6-4) and an Eulerian circuit(1-6-2-3-4-5-3-6-4-1) exists. This is because each vertex in the graph has an even degree.

In 6(b), an Eulerian trail exists(1-6-2-3-4-5-3-6-4), since there are only two vertices of odd degree(1 and 4). But there is no Eulerian circuit.

In figure 6(c), neither an Eulerian circuit, nor an Eulerian train exists. We can see that 4 of the vertices(1, 3, 4 and 6) have odd degrees.

2.8 Hamiltonian Circuits

A **Hamiltonian circuit**, also called a **Hamiltonian cycle** is cycle (i.e., closed loop) through a graph that visits each node exactly once.

By convention, the trivial graph on a single node is considered to possess a Hamiltonian circuit, but the connected graph on two nodes is not. A graph possessing a Hamiltonian circuit is said to be a Hamiltonian graph. Alternatively, **Hamiltonian graph** is a graph with a spanning cycle also called **Hamiltonian cycle**. Figure 7 shows a Hamiltonian graph, having a Hamiltonian cycle as 1-2-3-4-5-6-7-8-9-1.

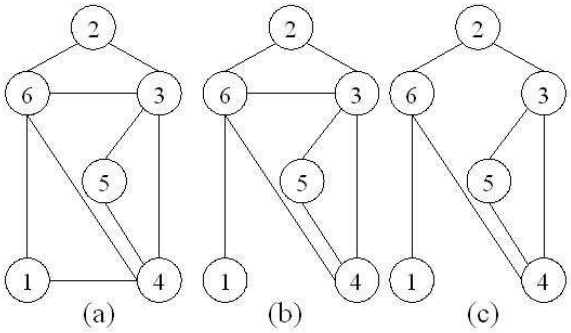


Fig. 6. Eulerian trail and circuit

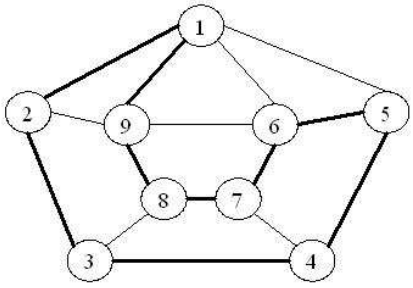


Fig. 7. Hamiltonian graph

Determining whether such paths and cycles exist in graphs is the *Hamiltonian path problem* which is *NP-complete*. Hamiltonian Path problem is a special case of *Traveling salesman problem*.

2.9 Tournament

An **orientation** of an undirected graph G is a digraph D obtained from G by choosing an orientation ($x \longrightarrow y$ or $y \longrightarrow x$) for each edge $(x, y) \in E(G)$. A **tournament** is an orientation of a complete graph. Figure 8(a) shows a complete graph, and figure 8(b) shows a tournament, which is an orientation of 8(a).

2.10 Connectivity

- A **seperating set** or **vertex cut** of a graph is a set of vertices $S \subseteq V(G)$ such that $G - S$ has more than one component. A vertex is said to be

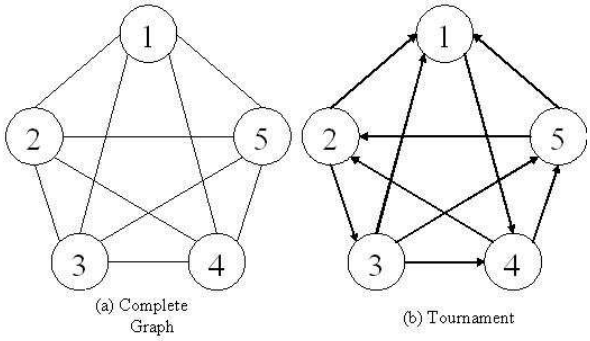


Fig. 8. Tournament

cut-vertex if deletion of that vertex leaves graph disconnected i.e have more than one component.

- The **connectivity** of G, written $\kappa(G)$, is the minimum size of a vertex cut. Connectivity of K_n and $K_{n,m}$ are $n-1$ and $min(m,n)$ respectively.
- A graph is $k-connected$ if its vertex connectivity is at least k.
- An **edge cut** is a set of edges $F \subseteq E(G)$ such that $G-F$ has more than one component. An edge is said to be **cut-edge** if deletion of that edge leaves graph disconnected i.e having more than one component.
- **Edge connectivity** of G, written $\kappa^{'}(G)$, is the minimum size of an edge cut.
- A graph is k-**edge-connected** if its edge connectivity is at least k.

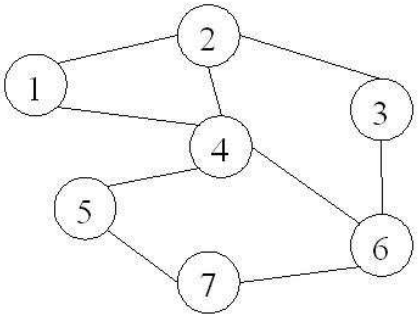


Fig. 9. Connectivity

Example (figure 9):

Vertex cuts:{1, 2, 6} is a vertex cut. This is because on removal of nodes 1, 2 and 6, the graph has 2 components, which are {3} and {4, 5, 7}. By similar arguments, there are many other vertex cuts, like {4, 5, 6}, {4, 6}, {3, 4} ...

Minimum size vertex cuts: {2, 4}, {2, 6}, {4, 6}, {3, 4} and {5, 6} are all minimum size vertex cuts, with size of 2 each.

Vertex connectivity: 2

Graph is *2-connected and 1-connected.*

Edge cuts: {(1, 2), (1, 4)} is an edge cut, because on removal of these edges, the number of components changes from one to two, consisting of {1} and {2, 3, 4, 5, 6, 7}. By similar arguments, there are many other edge cuts, like {(2, 3), (3, 6)}, {(1, 4), (2, 4), (3, 6)}, {(1, 2), (1, 4), (2, 4), (2, 3)} ....

Minimum size edge cuts: {(1, 2), (1, 4)}, {(2, 3), (3, 6)}, {(5, 7), (6, 7)} and {(5, 7), (4, 5)} are all minimum size edge cuts, with size of 2 each.

Edge connectivity: 2

Graph is *2-edge-connected and 1-edge-connected.*

Theorem 1.4: *The edge connectivity of a graph G cannot exceed the minimum degree of any node in G.*

Proof: Consider the node v with the minimum degree in the graph G. Let the degree of v be $\delta(G)$. If we delete all those $\delta(G)$ edges whose one end point is v, we get a disconnected graph. Thus,

$$\kappa^{'}(G) \leq \delta(G).$$

Theorem 1.5: *The vertex connectivity of a graph G can never exceed the edge connectivity of G.*

Proof: First let us understand the meaning of notation $[S, T]$. Given $S, T \subseteq V(G)$, we write [S, T] for the set of edges having one end point in S and the other in T.

Now, given any edge cut $[S, S']$ of size k, we can find a vertex cut set of size k. It can be obtained by removing all the vertices that are the end points of one of the edges in $[S, S']$ keeping in mind not to make S or S' null. The number of vertices removed is at most the number of edges in the edge cut. Thus,

$$\kappa(G) \leq \kappa^{'}(G).$$

Also, since in any graph the average degree of a node is $2e/n$ thus minimum degree $\delta(G) \leq \lfloor 2e/n \rfloor$. Thus,

$$\kappa(G) \leq \kappa^{'}(G) \leq \delta(G) \leq \lfloor 2e/n \rfloor.$$

Here, n: no. of vertices, e: no. of edges, $\lfloor x \rfloor$: the greatest integer smaller than x.

3 Algorithms

3.1 Breadth-first search(BFS)

In graph theory, **BFS** is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

Algorithm for BFS:

1. Initialize the color of all nodes as white.
2. Color the root node as grey and enqueue it.
3. Dequeue a node and examine it. If the element sought sought is found this node, quit the search and return the result. Otherwise color the dequeued node as black and enqueue all those direct child nodes that are colored white. Color them as grey.
4. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
5. Repeat from Step 3.

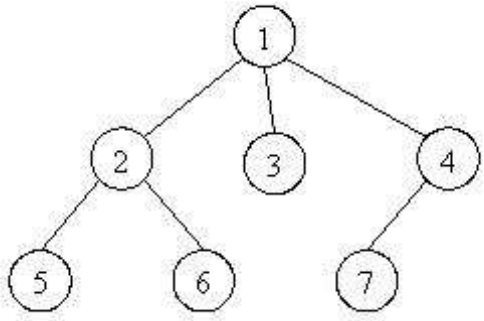


Fig. 10. Breadth-first search

Example:

Let us illustrate the working of BFS algorithm in figure 10. Suppose we are searching for an element which is in node 5.

1. All nodes of the set {1, 2, 3, 4, 5, 6, 7} colored as white.
2. Color 1, the root node grey and enqueue it. Queue is now [1].

3. Dequeue 1. It is not the goal node. Color it black and enqueue 2, 3, 4. Set the color of 2, 3, and 4 as grey. Queue is now [2, 3, 4].
4. Dequeue 2. It is not the goal node. Color it black and enqueue 5, 6. Set the color of 5, 6 as grey. Queue is now [3, 4, 5, 6].
5. Dequeue 3. It is not the goal node. Color it black and enqueue nothing (no children of 3). Queue is now [4, 5, 6].
6. Dequeue 4. It is not the goal node. Color it black and enqueue 7. Set the color of 7 as grey. Queue is now [5, 6, 7].
7. Dequeue 5. It is the goal node. Return the result.

3.2 Depth-first search(DFS)

DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring.

Algorithm for DFS:

1. Initialize the color of all nodes as white.
2. Color the root node as grey and push it in an empty stack.
3. Pop a node and examine it. If the element sought sought is found this node, quit the search and return the result. Otherwise color the popped node as black and push all those direct child nodes that are colored white. Color them as grey.
4. If the stack is empty, every node on the graph has been examined – quit the search and return "not found".
5. Repeat from Step 3.

Example:

Let us illustrate the working of DFS algorithm in the same figure as used for BFS, i.e., figure 10. Suppose that we are searching for an element which is in node 3.

1. All nodes of the set {1, 2, 3, 4, 5, 6, 7} colored as white.
2. Color 1, the root node grey and push it to an empty stack. Stack is now [1].
3. Pop 1. It is not the goal node. Color it black and push 4, 3, 2 in order. Set the color of 2, 3, and 4 as grey. Stack is now [4, 3, 2], with 2 on top.
4. Pop 2. It is not the goal node. Color it black and push 6, 5. Set the color of 5, 6 as grey. Stack is now [4, 3, 6, 5].
5. Pop 5. It is not the goal node. Color it black and push nothing (no children of 5). Stack is now [4, 3, 6].
6. Pop 6. It is not the goal node. Color it black and push nothing (no children of 6). Stack is now [4, 3].
7. Pop 3. It is the goal node. Return the result.

3.3 Strongly connected components (SCC)

We saw in Section 2 what a strongly connected component (SCC) means. We use the DFS algorithm (already discussed) for this purpose, but with a slight modification. Apart from the normal DFS algorithm, we add two more variables for each node, and we call them *s* and *f*. *s*[u] is the time at which a node *u* is first visited (start time, the time at which u is popped out of stack in the DFS algorithm), and f is the time at which all the children (and the children of children, recursively) of *u* have been visited (the time at which all the children of u have been popped out). Now we see the algorithm to compute SCC.

Algorithm for SCC:

1. Call $DFS(G)$ to compute finishing times $f[u]$ for each vertex u.
2. Compute G^T, where G^T is the transpose graph with all edges reversed.
3. Call $DFS(G^T)$, but consider the vertices in order of decreasing $f[u]$.
4. Produce as output the vertices of each tree in the DFS forest formed in point 3 as a separate SCC.

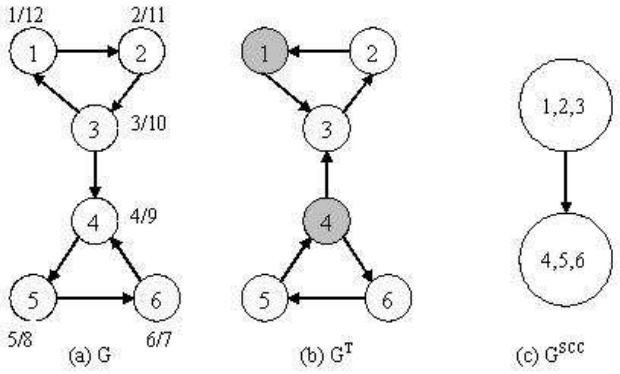


Fig. 11. Strongly connected components (SCC)

Example:

Let us illustrate the working of SCC algorithm in figure 11.

1. 11(a) shows a directed graph G, in which the nodes are marked by their start/finish times after running DFS on the graph.

2. 11(b) shows the transpose G^T of G. We call DFS on G^T, considering nodes in decreasing order of their finish times *f*. So, we start with node 1, and make one of the trees of DFS forest consisting of nodes 1, 2 and 3. Then we start with node 4, and make another tree of DFS forest consisting of nodes 4, 5 and 6.
3. In 11(c), we output the two SCC's as {1, 2, 3} and {4, 5, 6}.

3.4 Shortest path algorithms

Number of algorithms are available to compute shortest path between any two nodes. Some of them include Dijkstra's, Bellman-Ford, A^* search, Floyd-Warshall etc. We shall discuss Dijkstras and Bellman Ford algorithms here. Bellman Ford algorithm can work for negative edge weights as well, while Dijkstra's algorithm can't. Interested readers may refer any standard textbook for details about these algorithm.

Dijkstra's Algorithm

1. For each vertex v in the graph, initialize dist[v] = ∞ and previous[v] = undefined. Mark all the nodes as unvisited.
2. Make distance of source s = 0, i.e., dist[s] = 0.
3. Pick an unvisited node u having minimum value of dist[u], and mark it as visited.
4. for each unvisited neighbor v of u, check if dist[u] + length(u,v) is smaller than dist[v]. If so, make dist[v] = dist[u] + length(u,v). Also, make previous[v] = u.
5. If all the nodes are visited, return previous[ ]. Otherwise repeat from Step 3.

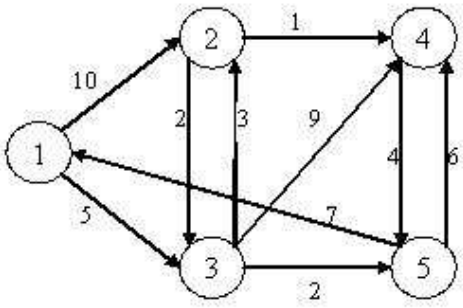


Fig. 12. Dijkstra algorithm

Example:

We illustrate the working of Dijkstra's algorithm using figure 12.

1. Assuming that 1 is the source, the algorithm starts with dist[1] = 0 and distance of other vertices = ∞.
2. The unvisited node with minimum value of distance is 1, with dist[1] = 0. 1 is marked as visited. Neighbors of 1 are 2 and 3, with dist[2] = dist[3] = ∞. Both are unvisited. dist[2] is modified to 10; dist[3] modified to 5. previous[2] and previous[3] both modified to 1.
3. The unvisited node with minimum value of distance is 3, with dist[3] = 5. 3 is marked as visited. Neighbors of 3 are 2, 4 and 5, with dist[2] = 10 and dist[4] = dist[4] = ∞. All are unvisited. dist[2] is modified to 8; dist[4] modified to 14; dist[5] modified to 7. previous[2], previous[4] and previous[5] all modified to 3.
4. The unvisited node with minimum value of distance is 5, with dist[5] = 7. 5 is marked as visited. Neighbors of 5 are 4 and 1, with dist[4] = 14. 1 is visited so need not be considered. 4 is unvisited. dist[4] is modified to 13. previous[4] modified to 5.
5. The unvisited node with minimum value of distance is 2, with dist[2] = 8. 2 is marked as visited. Neighbors of 2 are 3 and 4, with dist[4] = 13. 3 is visited so need not be considered. 4 is unvisited. dist[4] is modified to 9. previous[4] modified to 2.
6. The unvisited node with minimum value of distance is 4, with dist[4] = 9. All its neighbors are visited.
7. No unvisited node left. The algorithm returns with previous[1] = undefined; previous[2] = 3; previous[3] = 1; previous[4] = 2; previous[5] = 3;

Bellman Ford Algorithm

1. *Initialize graph.*
   For each vertex v in the graph, except for the source s, dist[v] = ∞ and previous[v] = undefined. Make dist[s] = 0.
2. *Relax edges repeatedly.*
   Repeat the following procedure n-1 times (where n = number of vertices): For each edge (u, v) (where u = (u, v).source and v = (u, v).destination), check if dist[v] is greater than dist[u] + weight((u, v)). If so, make previous[v] = u.
3. *Check for negative weight-cycles.*
   For each edge (u, v) (where u = (u, v).source and v = (u, v).destination), check if dist[v] is greater than dist[u] + weight((u, v)). If so, print the error message that the graph has a negative weight cycle.

Example:

We illustrate the working of Bellman Ford algorithm using the same figure as used for Dijkstra's, i.e., figure 12.

1. Assuming that 1 is the source, the algorithm starts with dist[1] = 0 and distance of other vertices = ∞. Also for each vertex v, previous[v] = undefined.
2. *Iteration 1*: dist[2] is modified to 10 and dist[3] to 5 when we consider edges (1, 2) and (1, 3) respectively; previous[2] and previous[3] are set to 1. dist[4] and dist[5] remain ∞; previous[4] and previous[5] remain undefined. This is because no other edge relaxation modifies these values.
3. *Iteration 2*: dist[2] changes to 8 and previous[2] changes to 3, because edge (3, 2) has a value of 3, which decreases distance of 2 from 1. dist[3] remains 5 and previous[3] remains 1; dist[4] is modified to 11 (=dist[2]+1) and previous[4] is modified to 2; dist[5] is modified to 7 and previous[5] is modified to 3.
4. *Iteration 3*: dist[2], dist[3], dist[5] and previous[2], previous[3], previous[5] undergo no change. Only change occurs for 4, making dist[4] = 9 and previous[4] = 2. This is clear, because the value of 2 in previous iteration had got modified to 8.
5. *Iteration 4*: No change.
6. Check for negative weight cycle. None found.

4 Adjacency Matrix

The **adjacency matrix** of G, written as $A(G)$, is the n-by-n matrix in which entry $\{a_{i,j}\}$ is the number of edges in G with endpoint $\{v_i, v_j\}$.

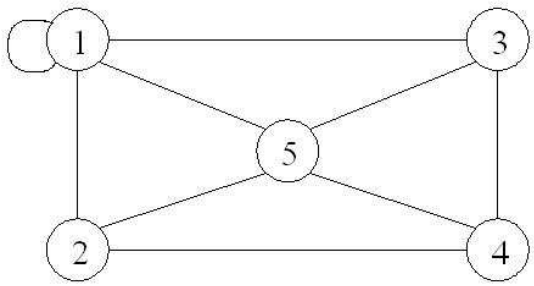


Fig. 13. Figure for adjacency matrix

The adjacency matrix of Figure 13 is:

$$\begin{pmatrix} 1\,1\,1\,0\,1 \\ 1\,0\,0\,1\,1 \\ 1\,0\,0\,1\,1 \\ 0\,1\,1\,0\,1 \\ 1\,1\,1\,1\,0 \end{pmatrix}$$

Properties of Adjacency matrix:

1. Adjacency matrix of an undirected graph is **symmetric**($a_{i,j} = a_{j,i}$ for all i, j).
2. By Adjacency Matrix we can represent self loops but can't represent parallel edges.
3. The adjacency matrix of a complete graph is all 1's except for 0's on the diagonal.
4. The adjacency matrix of a complete bipartite graph $K_{r,s}$ has the form

$$\begin{pmatrix} O & J \\ J^T & O \end{pmatrix}$$

where J is an $r \times s$ matrix of all ones and O denoted all-zero matrix.

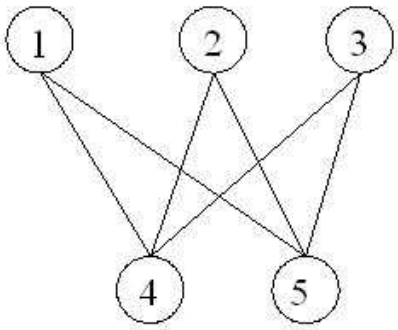


Fig. 14. Complete bipartite graph $K_{3,2}$

In figure 14, $r = 3, s = 2$. Hence J is 3×2 matrix.

$$J = \begin{pmatrix} 1\,1 \\ 1\,1 \\ 1\,1 \end{pmatrix}$$

and

$$A = \begin{pmatrix} 0\,0\,0\,1\,1 \\ 0\,0\,0\,1\,1 \\ 0\,0\,0\,1\,1 \\ 1\,1\,1\,0\,0 \\ 1\,1\,1\,0\,0 \end{pmatrix}$$

5. The degree of v is the sum of the entries in the row for v in $A(G)$.
6. Suppose two directed or undirected graphs G_1 and G_2 with adjacency matrices A_1 and A_2 are given. G_1 and G_2 are *isomorphic* if and only if there exists a permutation matrix P such that $PA_1P^{-1} = A_2$
   Consider graphs of figure 15 which are isomorphic.

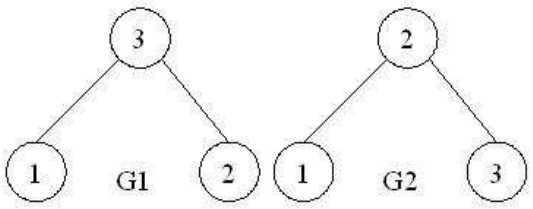


Fig. 15. Isomorphic graphs

$$A_1 = \begin{pmatrix} 0\ 0\ 1 \\ 0\ 0\ 1 \\ 1\ 1\ 0 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} 0\ 1\ 0 \\ 1\ 0\ 1 \\ 0\ 1\ 0 \end{pmatrix}$$

We will see that for

$$P = \begin{pmatrix} 3\ 2\ 1 \\ 1\ 1\ 5 \\ 2\ 3\ 1 \end{pmatrix}$$

they satisfy the equation $PA_1P^{-1} = A_2$. Isomorphism of G_1 and G_2 can also be verified by mapping $f(1) = 1, f(2) = 3, f(3) = 2$.

7. The main diagonal of every adjacency matrix corresponding to a graph without loops has all zero entries.
8. *Calculating paths from Adjacency Matrix:*
   If A is the adjacency matrix of the directed or undirected graph G, then the matrix A^n (i.e., the matrix product of n copies of A) has an interesting interpretation: the entry in row i and column j gives the number of (directed or undirected) paths of length n from vertex i to vertex j.

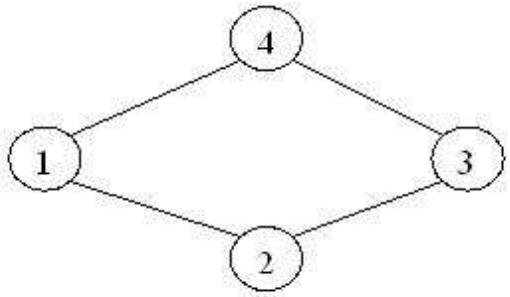


Fig. 16. Path using Adjacency Matrix

In figure 16

$$A^1 = \begin{pmatrix} 0\ 1\ 0\ 1 \\ 1\ 0\ 1\ 0 \\ 0\ 1\ 0\ 1 \\ 1\ 0\ 1\ 0 \end{pmatrix}$$

$$A^2 = \begin{pmatrix} 2\ 0\ 2\ 0 \\ 0\ 2\ 0\ 2 \\ 2\ 0\ 2\ 0 \\ 0\ 2\ 0\ 2 \end{pmatrix}$$

In matrix A^2, the entries represent the number of paths of length 2 in the above graph. Consider the [r1, c1] of A^2. The entry in this cell is 2. It means there two paths of length 2 in the graph from $1 \rightarrow 1$. These are $1 \rightarrow 2 \rightarrow 1$ and $1 \rightarrow 4 \rightarrow 1$. Similarly, consider the [r1, c3]. Entry in this cell is also 2, and the two paths of length 2 from $1 \rightarrow 3$ are $1 \rightarrow 4 \rightarrow 3$ and $1 \rightarrow 2 \rightarrow 3$.
Now, consider the [r1, c2], whose entry is 0. We can verify that there are no paths of length 2 from $1 \rightarrow 2$.

9. *Calculating Diameter from Adjacency Matrix:*
Diameter of a graph is the maximum shortest distance between any two nodes in the graph. As we have already seen, if A is the adjacency matrix, then A^r[i,j] represents the number of r-hop distant paths from vertex i to j. The smallest r for which A^r[i,j] is non-zero gives us the shortest distance from vertex i to j. Assuming the graph has 1 component, if we go on adding the matrices $A^1 + A^2 + ... + A^r$, then at some point, all the entries in this some will be positive.

So, diameter D = r, where all entries in $\sum_{k=1}^{r} A^k$ are positive, while not in $\sum_{k=1}^{r-1} A^k$

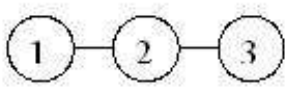


Fig. 17. Diameter calculation using adjacency matrix

In figure 17

$$A^1 = \begin{pmatrix} 0\ 1\ 0 \\ 1\ 0\ 1 \\ 0\ 1\ 0 \end{pmatrix}$$

$$A^2 = \begin{pmatrix} 1\ 0\ 1 \\ 0\ 2\ 0 \\ 1\ 0\ 1 \end{pmatrix}$$

$$A^1 + A^2 = \begin{pmatrix} 1\ 1\ 1 \\ 1\ 2\ 1 \\ 1\ 1\ 1 \end{pmatrix}$$

We see that for r = 2, all entries in $A^1 + A^2$ are positive. So the diameter is 2.

5 Incidence Matrix

The **incidence matrix** $M(G)$, is the n-by-m matrix in which entry $\{m_{i,j}\}$ is 1 if v_i is an endpoint of e_j and otherwise is 0.

The incidence matrix of graph shown in figure 18 is:

	$e_{(1,3)}$	$e_{(1,2)}$	$e_{(2,4)}$	$e_{(3,4)}$	$e_{(3,5)}$	$e_{(4,5)}$
$v1$	1	1	0	0	0	0
$v2$	0	1	1	0	0	0
$v3$	1	0	0	1	1	0
$v4$	0	0	1	1	0	1
$v5$	0	0	0	0	1	1

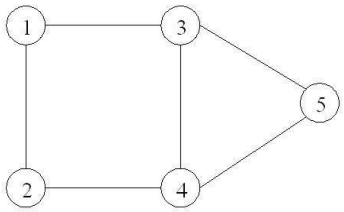
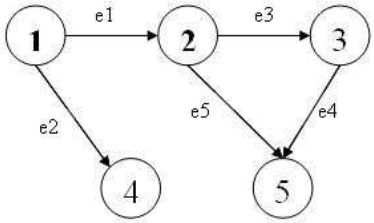


Fig. 18. Graph for Incidence Matrix

5.1 More on Incidence Matrix

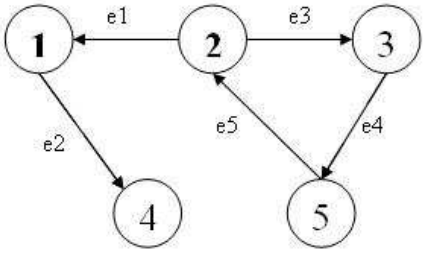
1. By Incidence Matrix we can't represent self loops but can represent parallel edges.
2. The incidence matrix of a directed graph D is a $p \times q$ matrix (b_{ij}) where p and q are the number of vertices and edges respectively, such that $b_{ij} = -1$ if the edge x_j leaves vertex v_i, 1 if it enters vertex v_i and 0 otherwise.(Figure 18)



	e1	e2	e3	e4	e5
v1	-1	-1	0	0	0
v2	1	0	-1	0	-1
v3	0	0	1	-1	0
v4	0	1	0	0	0
v5	0	0	0	1	1

Fig. 19. Incidence Matrix of a directed graph

3. The incidence matrix of one orientation of some undirected graph G can be obtained from the incidence matrix of some other orientation of G only by negating some of the columns. Consider the graphs of figure 19 and figure 20. We see that the direction of edges e1 and e5 have been reversed. If we negate columns 1 and 5 of figure 19, we get the incidence matrix of figure 20.



	e1	e2	e3	e4	e5
v1	1	-1	0	0	0
v2	-1	0	-1	0	1
v3	0	0	1	-1	0
v4	0	1	0	0	0
v5	0	0	0	1	-1

Fig. 20. Another orientation of previous graph

6 Planarity

6.1 Meaning of planar graph

A graph is planar if it has a drawing without crossings. Such a drawing is a planar embedding of G. A plane graph is a particular planar embedding of a planar graph.

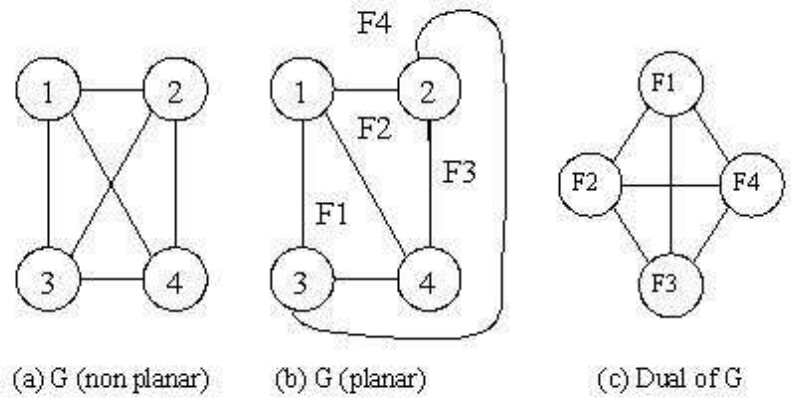


Fig. 21. Planarity

Figures 21(a) and 21(b) show two different embeddings of same graph G. 21(b) is a planar embedding of G, while 21(a) is not, since there are crossings

in 21(a), but no crossings in 21(b). Since G has a planar embedding, we say it is a planar graph.

The **faces** of a plane graph are the maximal regions of the plane that contain no point used in the embedding. Boundary of a face is a set of closed walks of edges that enclose the face. Referring back to figure 21(b), there are four faces F1, F2, F3 and F4 having boundaries 1-4-3-1, 1-2-4-1, 2-3-4-2 and 1-2-3-1 respectively.

Theorem 1.6: *If a connected plane graph G has exactly n vertices, e edges and f faces, then n - e + f = 2.*

6.2 Application of planar graphs

There are many practical applications of a planar graph. Some of the major among them being:

1. VLSI - e.g. laying out circuits on computer chips.
2. Vehicle routing - e.g. planning routes on roads without underpasses.
3. Telecommunications - e.g. spanning trees.

6.3 Dual graph

The **dual graph** G^* of a plane graph G is a plane graph whose vertices correspond to the faces of G. The edges of G^* corresponds to the edges of G as follows: if e is an edge of G with face X on one side and face Y on other side, then the endpoints of the dual edge $e^* \in E(G^*)$ are the vertices x, y of G^* that represent the faces X, Y of G. Figure 21(c) shows the dual graph corresponding to the plane graph shown in 21(b).

Theorem 1.7: (Kuratowski [1930]) *A graph is planar if and only if it does not contain a subdivision of K_5 or $K_{3,3}$.*

Subdivision of a graph is a graph obtained from it by replacing edges with pairwise internally-disjoint paths. K_5 is the complete graph of size 5 and $K_{3,3}$ is the complete bipartite graph with bipartition size $3, 3$.

K_5 is the smallest non-planar graph is terms of node($n = 5, e = 10$) while $K_{3,3}$ is in terms of edges ($n = 6, e = 9$).

7 Flow networks

A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E(G)$ has a nonnegative capacity $c(u, v) \geq 0$. If $(u, v) \notin E(G)$, we assume that $c(u, v) = 0$. We distinguish two vertices in a flow network: a source s and a

sink t. For convenience, we assume that every vertex lies on some path from the source to the sink. That is, for every vertex $v \in V(G)$, there is a path $s \to v \to t$. The graph is therefore connected, and $|E| \geq |V| - 1$.

Maximum flow problem: Find flow with maximum value

First let us understand the meaning of a Residual graph R_f. For each pair u, v of vertices:

1. If e_1 = (u, v) is in E and e_2 = (v, u) is also in E, then $r(u, v) = C(e_1) - f(e_1) + f(e_2)$ and $r(v, u) = C(e_2) - f(e_2) + f(e_1)$.
2. If e_1 = (u, v) is in E but (v, u) is not in E, then $r(u, v) = C(e_1) - f(e_1)$ and $r(v, u) = f(e_1)$.
3. If e_2 = (v, u) is in E but (u, v) is not in E, then $r(v, u) = C(e_2) - f(e_2)$ and $r(u, v) = f(e_2)$

Residual graph $R_f = (V, E_f)$, where $E_f = (u, v)$ such that $r(u, v) > 0$ and weight $w(e) = r(e)$ for e in E_f. Augmenting path = path from s to t in R_f. Now we see the Ford-Fulkerson algorithm to compute maximum flow in a network.

Ford Fulkerson Algorithm:

1. Compute R_f.
2. If there is some augmenting path P in R_f, then find the edge with minimum weight W in P and augment(increase) the flow along all edges in P by W. Then compute new R_f.
3. If there is no such path, the current flow is a max-flow.
4. Repeat from step 2.

Example:

Let us illustrate this algorithm with example shown in figure 22.

1. The capacities of links in the network are shown in 22(a). Here node 1 is the source, and 4 is the sink. Initially R_f is same as the original graph.
2. We find an augmenting path P as $1 \to 2 \to 3 \to 4$. Minimum weight W = 1 in P. So, we augment the flow along all edges in P by 1 (22(b) flow), and compute the new augmenting graph (22(b) R_f).
3. We find an augmenting path P as $1 \to 3 \to 2 \to 4$. Minimum weight W = 1 in P. So, we augment the flow along all edges in P by 1 (22(c) flow), and compute the new augmenting graph (22(c) R_f).
4. We find an augmenting path P as $1 \to 2 \to 4$. Minimum weight W = 1 in P. So, we augment the flow along all edges in P by 1 (22(d) flow), and compute the new augmenting graph (22(d) R_f).
5. No more augmenting path exists in R_f. So, we return the max flow as 3.

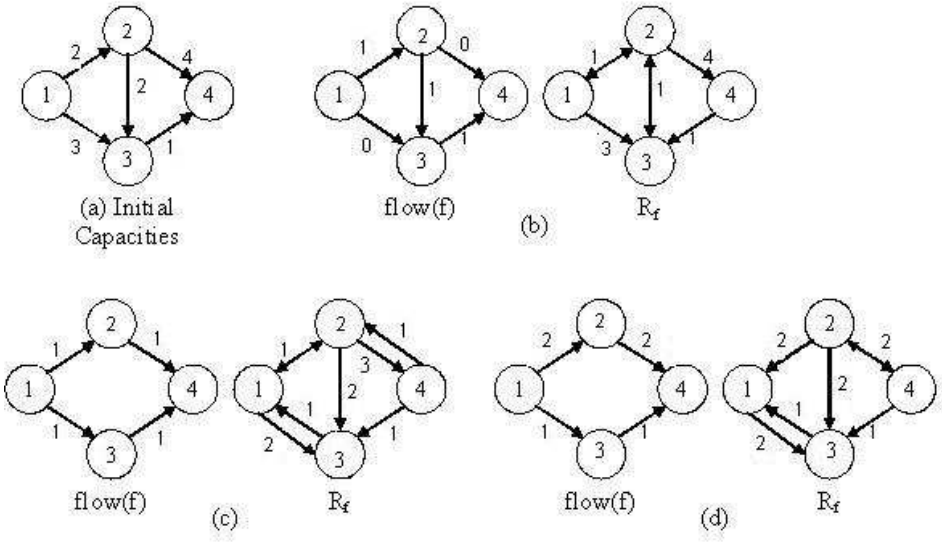


Fig. 22. Network flow

Theorem 1.8 (Max-Flow Min-Cut Theorem): *The maximum amount of flow is equal to the capacity of a minimum cut.*

In other words, the theorem states that the maximum flow in a network is dictated by its bottleneck. Between any two nodes, the quantity of material flowing from one to the other cannot be greater than the weakest set of links somewhere between the two nodes. Using Ford Fulkerson algorithm, we have already seen that the maximal flow of network shown in figure 22 is 3. The minimal cut in this network corresponds to S = {1, 3} and T = {2, 4}. This is indeed equal to the max flow, viz. 3.

Proof for Max-Flow Min-Cut Theorem:

Let D be a directed graph, and let u and v be vertices in D. The maximum weight among all (u, v)-flows in D equals the minimum capacity among all sets of arcs in $A(D)$ whose deletion destroys all directed paths from u to v.

Furthermore, there is a low-order polynomial-time algorithm which will find a maximum (u, v)-flow and a minimum capacity (u, v)-cut (a set of arcs

in $A(D)$ whose deletion destroys all directed paths from u to v).

We sketch the proof for the case in which all capacities are +1. A $(0,1)$-flow is function from the set of arcs to the set $(0,1)$, such that sum of the flow on arcs into any vertex is the same as the sum of the flow on arcs out of the vertex, except for the two special vertices: the source, u, and the sink, v. The 0-flow is the flow which is zero on every arc.

Starting with any $(0,1)$-flow f, for example the 0-flow, we make a new directed graph, D', with $V(D') = V(D)$ and with $A(D') = A_1 \cup A_2$, where A_1 is the set of those arcs of D whose flow is zero, and A_2 is the set of reversals of those arcs of D whose flow is one.

Suppose that there is a directed (u,v)-path, P, in D'. Let F denote the symmetric difference of A_1 and P. Clearly, F is the set of edges of a flow f', where f' is one on arcs of F and zero on arcs not in F. Furthermore, F contains more arcs which are incident with u than A_1 does.

Suppose, instead, that there is no directed (u,v)-path in D'. Let S denote the set of vertices in D' to which there is a directed path from u in D'. Then, in D, the set of arcs, C, leaving S must have cardinality exactly the same as the cardinality of the sets of arcs from A_1 which are incident with u.

The sum of the capacities on the arcs of A_1 which are incident with u is usually called the weight of the flow. Obviously, no $(0,1)$-flow can have weight exceeding $|C|$.

8 Conclusion

After all these discussions, we can see that graph theory is not just theoretical concepts about graphs. It has a lot of practical applications. We saw the ways to represent graphs. Though a picture speaks a thousand words, we saw that adjacency matrix is a very important and useful way to represent graphs. We saw the utilities of adjacency matrix in finding paths and diameter of graph. Adjacency matrix also gives us idea about communities. We saw BFS and DFS search algorithms in graphs. We saw bipartite graphs, which have numerous applications in the chapters to come. Then we saw the Eulerian circuits and the famous Chinese Postman Problem associated with it. We saw hamiltonian circuits, which has the famous Traveling Salesman problem associated with it. Tournaments are widely used in rank aggregation problems. Then we saw how to find single source shortest paths using Dijkstra's and Bellman Ford algorithms. We saw the concepts of planarity, which are very important in laying VLSI circuits. Connectivity of graphs and flow networks give us an

idea about how to find maximum flows in networks, and we also saw the famous max flow min cut theorem associated with them.

Graph Theory is now a major tool in mathematical research, electrical engineering, computer programming and networking, business administration, sociology, economics, marketing, and communications; the list can go on and on. In particular, many problems can be modeled with paths formed by traveling along the edges of a certain graph. For instance, problems of efficiently planning routes for mail delivery, garbage pickup, snow removal, diagnostics in computer networks, and others, can be solved using models that involve paths in graphs.

As you can expect, graphs can be sometimes very complicated. So one needs to find more practical ways to represent them. Matrices are a very useful way of studying graphs, since they turn the picture into numbers, and then one can use techniques from linear algebra.