Graphs

Introduction

A graph is an abstract data structure that is used to implement the mathematical concept of graphs.

Definition

A graph G is defined as an ordered set (V, E), where $V(G)$ represents the set of vertices and $E(G)$ represents the edges that connect these vertices.

Graph Terminology

Adjacent nodes or neighbors: For every edge, $e = (u, v)$ that connects nodes u and v, the nodes u and v are the end-points and are said to be the adjacent nodes or neighbors.

Degree of a node: Degree of a node u, $deg(u)$, is the total number of edges containing the node u. If $deg(u) = 0$, it means that u does not belong to any edge and such a node is known as an isolated node.

Regular Graph: It is a graph where each vertex has the same number of neighbors. That is, every node has the same degree. A regular graph with vertices of degree k is called a k-regular graph or a regular graph of degree k.

Path: A path P written $P = \{v_0, v_1, v_2, \ldots, v_n\}$, of length n from a node u to v is defined as a sequence of $(n + 1)$ nodes.

Closed Path: A path P is known as a closed path if the edge has the same end-points. That is, if $v_0 = v_n$.

Simple Path: A path P is known as a simple path if all the nodes in the path are distinct with an exception that v_0 may be equal to v_n. If $v_0 = v_n$, then the path is called a closed simple path.

Cycle: A path in ich the first and the last vertices are the same. A simple cycle has no repeated edges or vertices (except the first and last vertices).

Connected Graph: A graph is said to be connected if for any two vertices (u, v) in V there is a path from u to v. That is to say there is no isolated nodes in the graph. A connected graph that does not have any cycle is called a tree.

Complete Graph: A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has $\frac{n(n-1)}{2}$ edges, where n is the number of nodes in G.

Clique: In an undirected graph $G = (V, E)$, clique is a subset of the vertex $C \subseteq V$, such that for every two vertices in C, there is an edge that connects two vertices.

Labelled Graph or Weight Graph: A graph is said to be labelled if every edge in the graph is assigned some data. In a weighted graph, the edges of the graph are assigned some weight or length. The weight of an edge is denoted by $w(e)$ is a positive value which indicates the cost of traversing the edge.

Multiple Edges: Distinct edges which connect the same end-points are called multiple edges. That is, $e = (u, v)$ and $e' = (u, v)$ are known as multiple edges of G

Loop: An edge that has identical end-points is called a loop. That is $e = (u, u)$.

Multi-Graph: A graph with multiple edges and/or loops is called a multi-graph.

Size of a Graph: The size of a graph is the total number of edges in it.

Directed Graphs

A directed graph G, also known as a **digraph**, is a graph in which every edge has a direction assigned to it. An edge of a directed graph is given as an ordered pair (u, v) of nodes in G. For an edge (u, v),

- The edge begins at u and terminates at v.
- u is known as the origin or initial point of e. Correspondingly, v is known as the destination or terminal point of e.
- u is the predecessor of v. Correspondingly, v is the successor of u.
- Nodes u and v are adjacent to each other

Terminology of a Directed Graph

Out-degree of a node: The out-degree of a node u, written as `outdeg(u)`, is the number of edges that originate at u.

In-degree of a node: The in-degree of a node u, written as `indeg(u)`, is the number of edges that terminate at u.

Degree of a node: The degree of a node, written as `deg(u)`, is equal to the sum of in-degree and out-degree of that node. Therefore, `deg(u) = indeg(u) + outdeg(u)`.

Isolated Vertex: A vertex with degree zero. Such a vertex is not an end-point of any edge.

Pendant Vertex: (also known as leaf vertex) A vertex with degree one.

Cut vertex: A vertex which when deleted would disconnect the remaining graph.

Source: A node u is known as a source of it has a positive out-degree but a zero in-degree.

Sink: A node $u$is known as a sink if it has a positive in-degree but a zero out-degree.

Reachability: A node v is said to be reachable from node u, if and only if there exists a (directed) path from node u to node v.

Strongly Conncected Directed Graph: A digraph is said to be strongly connected if and only if there exists a path between every pair of nodes in G. That is if there is a path from node u to v, then there must be a path from node v to u.

Unilaterally Connected Graph: A digraph is said to be unilaterally connected if there exists a path between any pair of nodes u, v in G such that there is a path fro u to v or a path from v to u, but not both.

Weakly Connected Digraph: A directed graph is said to be weakly connected if it is connected by ignoring the direction of edges. In such a graph, it is possible to reach any node from any other node by traversing edges in any direction. (may not be in the direction they point). The nodes in a weakly connected directed graph must have either out-degree or in-degree of at least 1.

Parallel/Multiple edges: Distinct edges which connect the same end-points are called multiple edges. That is, $e = (u, v)$ and $e' = (u, v)$ are known as multiple edges of G.

Simple Directed Graph: A directed graph G is said to be a simple directed graph if and only if it has no parallel edges. However, a simple directed graph may contain cycles with an exception that is cannot have more than one loop at a given node.

Transitive Closure of a Directed Graph

A transitive closure of a graph is constructed to answer reachability questions. Like the adjacency list, transitive closure is also stored as a matrix T.

Definition:

For a directed graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, the transitive closure of G is a graph $G^* = (V, E^*)$. In G^*, for every vertex pair v, w in V there is an edge (v, w) in E^* if and only if there is a valid path from v to w in G.

Where and Why is it Needed?

Finding the transitive closure of a directed graph is an important problem in the following computational tasks:

- Transitive closure is used to find the reachability analysis of transition networks representing distributed and parallel systems.

- It is used in the construction of parsing automata in compiler construction.

- Recently, transitive closure computation is being used to evaluate recursive databases queries (because almost all practical recursive queries are transitive in nature).

Algorithm

In order to determine the transitive closure of a graph, we define a matricx t where $t_{ij}^k = 1$ for $i, j, k = 1, 2, 3, \ldots, n$ if there exists a path in G from the vertex i to vertex j with intermediate vertices in the set $(1, 2, 3, \ldots, k)$ and 0 otherwise. That is, G^* is constructed by adding an edge (i, j) into E^* if and only if $t_{ij}^k = 1$.

Bi-Connected Components

A vertex v of G is called an articulation point, if removing v along with the edges incident on v, result in a graph that has at least two connected components.

A bi-connected graph is defined as a connected graph that has no articulation vertices. That is, bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected. By definition,

- A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deleting any single vertex.

- In a bi-connected directed graph, for any two vertices v and w, there are two directed paths from v to w which have no vertices in common other than v and w.

As for vertices, there is a related concept for edges. An edge in a graph is called a bridge if removing that edge results in a disconnected graph. Also, an edge in a graph that does not lie on a cycle is a bridge. This means that a bridge has at least one articulation point at its end, although it is not necessary that the articulation point is linked to a bridge.

Representation of Graphs

There are three common ways of storing graphs in the computer's memory. There are:

- **Sequential representation** by using an adjacency matrix.
- **Linked Representation** by using an adjacency list that stores the neighbors of a node using a linked list.
- **Adjacency multi-list** which is an extenstion of linked representation.

Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.

In a directed graph G, if node v is adjacent to node u, then there is an edge from u to v. For any graph G havig n nodes, the adjacency maxtrix will have the dimension of $n \times n$.

In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry a_{ij} in the adjacency matrix will contain 1, if vertices v_i and v_j are adjacent to each other. However if nodes are not adjacent, a_{ij} will be set to zero. Since an adjacency matrix contains only 0 and 1, it is called a **bit matrix** or a **Boolean matrix**. The entries in the matrix depend on the ording of the ndoes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

We can draw the following conclusions about Adjacnecy Matrices

- For a simple graph (that has no loops), the adjacency matrix has 0s on the main diagonal.
- The adjacency marix of an undirected graph is symmetric.
- The memory use of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.
- Number of 1s (or non-zerp entries) in an adjacency matrix is equal to the number of edges in the graph.
- The adjacency matrix for a weighted graph contains the wights of the edges connecting the nodes.

In general terms, we can conclude that every entry in the ith row and jth colum of A^n (where n is the number of nodes in the graph) gives the number of paths of length n from node v_i to v_j.

Adjacency List Representation

An adjacency list structure consists of a list of all node in G. Every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it.

The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.

- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.

- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered.

For a directed graph, the sum of the lengths of all adjacency lists is equal to the number of edges in G. However, for an undirected graph, the sum of the lengths of all adjacency lists is equal to twice the number of edges in G because an edge (u, v) means an edge from node u to v as well as an edge from v to u. Adjacency lists can also be modified to store weighted graphs.

Adjacency Multi-list Representation

Graphs can also be represented using multi-lists which can be said to be modified version of adjacency lists. Adjacency multi-list is an edge-based rather than vertex-based representation of graphs. A multi-list representation basically consists of two parts—a directory of nodes' information and a set of linked list storing information about edges. While there is a single entry for each node in the node directory, every node, on the other hand, appears in two adjacency lists (one for the node at each end of the edge). For example, the directory entry for node i points to the adjacency node for node i. This means that the nodes are shared among several lists.

In a multi-list representation, the information about an edge (v_i, v_j) of an undirected graph can be stored using the following attributes:

M: A single bit field to indicate whether the edge has been examined or not.

v_1: A vertex in the graph that is connected to vertex v_j by an edge.

v_j: A vertex that is connected to vertex v_i by an edge.

Link i for v_1: A link that points to another node that has an edge incident on v_i.

Link j for v_i: A link that points to another node that has an edge incident on v_j.

Graph Traversal Algorithms

There are two standard methods of graph traversal:

(1) Breadth-first search

(2) Depth-first search

While the breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack. Both these algorithms make use of a variable `STATUS`. During execution of the algorithm, every node in the graph will have the variable `STATUS` set to 1 or 2, depending on its current state.

Breadth-First Search Algorithm

Breadth-first search (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, the algorithm explores their unexplored neighbor nodes, and so on until it finds the goal.

That is, we start examining the node A and then all the neghbor nodes of A are examined. In the next step, we examine the neighbors of A, so on and so fourth. This means that we need to track the neighbors of the nodes and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing and a variable `STATUS` to represent the current state of the node.

Features of Breadth-First Search Algorithm

Space Complexity In the breadth-first search algorithm, all the nodes at a particular level must be saved until their child nodes in the next level have been generated. The space complexity is proportional to the number of nodes at the deepest level of the graph. Given a graph with a branching factor b (number of children at each node) and depth d, the asymptotic space complexity is the number of nodes at the deepest level $O(b^d)$. If the number of vertices in the graph are known ahead of time, the space complexity can also be expressed as $O(|E| + |V|)$, where $|E|$ is the total number of edges in G and $|V|$ is the number of nodes or vertices.

Time Complexity In the worst case, breadth-first search has to traverse through all paths to all possible nodes, thus the time complexity if this algorithm asymptotically approaches $O(b^d)$. However, the time complexity can also be expressed as $O(|E| + |V|)$, since every vertex and every edge will be explored in the worst case.

Completeness Breadth-first search is said to be a complete algorithm because if there is a solution, breadth-first search will find it regardless of the kind of graph. But in case of an infinite graph where there is no possible solution, it will diverge.

Optimality Breadth-first search is optimal for a graph that has edges of equal length, since it always returns the result with the fewest edges between the start node and the goal node. But generally, in real-world applications, we have weighted graphs that have costs associated with each edge, so the goal next to the start does not have to be the cheapest goal available.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.
- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Depth-First Search Algorithm

The depth-first seach algorithm progresses by expanding the starting node of G and then going deeper and deeper until the goal node is found, or until a node that has no children is encountered. When a dead-end is reached, the algorithm backtracks, returning to the most recent node that has not been completely explored.

In other words, depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbor of A, then a neighbor of neighbor of A and so on. During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node.

The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the dead-end, we backtrack to find another path P'. The algorithm terminates when backtracking leads back to the starting node A. In this algorithm, edges that lead to a new vertex are called **discovery edges** and edges that lead to an already visited vertex are called **back edges**.

This algorithm is similar to the in-order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm byt here we use a stack instead of a queue. Again, we use a variable `STATUS` to represent the current state of the node.

Features of Depth-First Search Algorithm

Space Complexity The space complexity if a depth-first search is lower than that of a breadth-first search.

Time Complexity The time complexity of a depth-first search is proportional to the number of vertices plus the number of edges in the graphs that are traversed. The time complexity can be given as $O(|V|+|E|)$.

Completeness Depth-first search is said to be a complete algorithm. If there is a solution, depth-first search will find it regardless of the kind of graph. But in case of an infinite graph, where there is no possible solution, it will diverge.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Topological Sorting

Topological sort of a directed acyclic graph (DAG) G is defined as a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more number of topological sorts.

A topolgical sort of a DAG G is an ordering of the vertices of G such tat if G contains an edge (u, v), then u appears before v in the ordering. Note that topological sort is possible only on directed acyclic graphs that do not have any cycles. For a Dag that contains cycles, no linear ordering of its vertices is possible. In other words, a topological ordering of a DAG G is an ordering of its vertices such that any directed path G is an ordering of its vertices such that any directed path in G traverses the vertices in increasing order.

Topological sorting is widely used in scheduling applications, jobs or tasks. The jobs that have to be completed are represented by nodes, and there is an edge from node u to v if job u must be completed before job v can be started. A topological sort of such a graph gives an order inwhich the given jobs must be performed.

Algorithm

The algorithm for the topological sort of a graph that has no cycles focuses on selecting a node N with zero in-degree, that is, a node that has no predecessor. The two main steps in volved in the topological sort algorithm include:

- Selecting a node with zero in-degree
- Deleting N from the graph along with its edges.

We will use a `QUEUE` to hold the nodes with zero in-degree. The order in which the nodes will be deleted from the graph will depend on the sequence in whic the nodes are inserted in the `QUEUE`. Then, we will use a variable `INDEG`, where `INDEG(N)` will represent the in-degree of node N.

Shortest Path Algorithms

Three algorithms to calculate the shortest path between the vertices of a graph G. These algorithms include:

- Minimum Spanning Tree
- Dijkstra's algorithm
- Warshall's algorithm

The first two use an adjacency list to find the shortest path, Warshall's algorithm uses an adjacency matrix to do the same.

Minimum Spanning Trees

A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together. A graph G can have many different spanning trees. We can assign weights to each edge (which is a number that represents how unfavorable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree. A **minimum spanning tree** (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Properties

Possible multiplicity There can be multiple spanning trees of the same weight. Particulary, if all the weights are the same, then every spanning tree will be minimum.

Uniqueness When each edge in the graph is assigned a different weight, then there will be only one unique minimum spanning tree.

Minimum-cost subgraph If the edges of a graph are assigned non-negative weights, then a minimum spanning tree is in fact the minimum-cost subgraph or a tree that connects all vertices.

Cycle property If there exists a cycle C in the graph G that has a weight larger than that of other edges of C, then this edge cannot belong to an MST.

Usefulness Minimum spanning trees can be computed quickly and easily to provide optimal solutions. These trees create a sparse subgraph that reflects a lot about the original graph.

Simplicity The minimum spanning tree of a weighted graph is nothing but a spanning tree of the graph which comprises of $n - 1$ edges of minimum total weight. Note that for an unweighted graph, any spanning tree is a minimum spanning tree.

Applications of Minimum Spanning Trees

1. MSTs are widely used for designing networks. For instance, people separated by varying distances wish to be connected together through a telephone network. A minimum spanning tree is used to determine the least costly path with no cycles in this network, thereby providing a connection that has the minimum cost involved.

2. MSTs are useful to find airline routes. While the vertices in the graph denoted cities, edges represent the routes between these cities. The more the distance between the cities, the higher the amount charged. Therefore, MSTs are used to optimize airline routes by finding the least costly path with no cycles.

3. MSTs are also used to find the cheapest way to connect terminals, such as cities, electronic components or computers via roads, airlines, railways, wires, or telephone lines.

MSTs are applied in routing algorithms for finding the most efficient path.

Prim's Algorithm

Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph. In other words, the algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized. For this, the algorithm maintains three sets of vertices which can be given as:

- **Tree vertices** Verties that are part of the minimum spanning tree T.
- **Fringe vertices** Vertices that are currently not part of T, but are adjacent to some tree vertex.
- **Unseen vertices** Vertices that are neither tree vertices nor fringe vertices. fall under this category.

The steps involved in the Prim's algorithm are:

- Choose a starting vertex.
- Branc out from the starting vertex and during each iteration, select a new vertex and an edge. Basically, during each iteration of the algorithm, we have to select a vertex from the fringe vertices in such a way that the edge connecting the tree vertex and the new vertex has the minimum weight assigned to it.

The running time of Prim's algorithm can be gien as $O(E \log V)$ where E is the number of edges and V is the number of vertices in the graph.

Kruskal's Algorithm

Kruskal's algorithm is sued to find the minimum spanning tree for a connected weighted graph. The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized. However, if the graph is not connected, then it finds a minimum spanning forest. Note that a forest is a collection of trees. Similarily, a minimum spanning forest is a collection of minimum spanning trees.

Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choich at each stage with the hope of finding the global optimum.

In the algorithm, we use a priority queue Q in which edges that have a minimum weight takes a priority over any other edge in the graph. When the Kruskal's algorithm terminates, the forest has only one component and forms a minimum spanning tree of the graph. The running time of the Kruskal's algorithm can be given as $O(E \log V)$, where E is the number of edges and V is the number of vertices in the graph.

Dijkstra's Algorithm

Dijkstra's algorithm, is used to find the shortest path tree. This algorithm is widely used in network routing protpcols, most notably IS-IS and OSPf9open Shrtest Path First).

Given a graph G and a source A, the algorithm is used to find the shortest path (one having the lowest cost) betwen A (source node) and every other node. Moreover, Dijkstra's algorithm is also used for finding the costs of the shortest paths froma source node to a destination node.

Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph. Ther term optimal can mean anything, shortest, cheapest, or fastest. If we start the algorith with an initial node, then the distance of a node Y can be given a the distance from the initial node to that node.

Dijkstra's algorithm labels every node in the graph where the labels represent the distance (cost) from the source node to that node. There are two kinds of labels **temporary** and **permanent**. Temporary labels are assigned to nodes that have not been reached, while permanent labels are given to nodes that have been reached and their distance (cost) to the source node is known. A node must be a permanent label or a temporary label, but not both.

The execution of this algorithm will produce either of the following two results:

1. If the destination node is labelled, then the label will in turn represent the distance from the source node to the destination node.

2. If the destination node is not labelled, then there is no path from the source to the destination node.

Difference between Dijkstra's Algorithm and Minimum Spanning Tree

Minimum spanning tree algorithm is used to traverse a graph in the mose efficient manner, but Dijkstra's calculates the distance from a given node to ever other vertex in the graph.

Dijkstra's algorithm is very similar to Prim's algorithm. Both the algorithms begin at a specific node and extend outward within the graph, until all other nodes in the graph have been reached. The point where these algorithms differ is that while Prim's algorithm stores a minimum cost edge, Dijkstra's stores the total cost from a source to the current node. Moreover, Dijkstra's algorithm is used to store the summation of minimum costs edges, while Prim's algorithm stores at most one minimum cost edge.

Warshall's Algorithm

If a graph G is given as $G = (V, E)$, where V is the set of vertices and E is the set of edges, the path matrix of G can be found as, $P = A + A^2 + A^3 + \cdots + A^n$. This is a lengthy process, so Warshall has given a very efficient algorithm to calculate the path matrix. Warshall's algorithm defines matrices $P_0, P_1, P_2, \ldots, P_n$

This means that if $P_0[i][j] = 1$, then there exists an edge from node v_i to v_j.

If $P_1[i][j] = 1$, then there exists an edge from v_i to v_j that does not use any other vertex except v_1

If $P_2[i][j] = 1$, then there exists an edge from v_i to v_j that does not use any other vertex except v_1 and v_2.

Note that P_0 is equal to the adjacency matrix of G. If G contains n nodes, then $P_n = P$ which is the path matrix of the graph G.

We can conclude that $P_k[i][j]$ is equal to 1 only when either of the two following cases occur:

- There is a path from v_1 to v_j that does not use any other node except $v_1, v_2, \ldots, v_{k-1}$. Therefore, $P_{k-1}[i][j] = 1$

- There is a path from v_i to v_k and a path from v_k to v_j where all the nodes use $v_1, v_2, \ldots, v_{k-1}$ Therefore

$$P_{k-1}[i][j] = 1 \quad \text{AND} \quad P_{k-1}[k][j] = 1$$

Hence, the path matrix P_n can be calculated with the formula

$$P_k[i][j] = P_{k-1}[i][j] \vee (P_{k-1}[i][j] \wedge P_{k-1}[k][j])$$

where $\vee$ indicates logical `OR` operation and $\wedge$ indicates logical `AND` operation.

Modified Warshall's Algorithm

Warshall's algorithm can be modified to obtain a matrix that gives the shortest paths between the nodes in a graph G. As an input to the algorithm, we take the adjacency matrix A of G and replace all the values of A which are zero by infinity (∞) denotes a very large number and indicates that there is no path between the vertices. In Warshall's modified algorithm, we obtain a set of matrices $Q_0, Q_1, Q_2, \ldots, Q_m$ using the formula below:

$$Q_k[i][j] = \texttt{Minimum}\Big(M_{k-1}[i][j], M_{k-1}[i][k] + M_{k-1}[k][j]\Big)$$

Q_0 is exactly yhe same as A with differece that every element having a zero value in A is replaced by (∞ in Q_0. Using the given formula, the matrix Q_n will give the path matrix that has the shortest path between the vertices of the graph.

Applications of Graphs

Graphs are constructed for various types of applications such as:

• In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.

• On transport networks where stations are drawn as vertices and routes become the edges of the graph.

• In maps that draw cities/states/regions as vertices and adjacency relations as edges.

• In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.

• Once we have a graph or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by edges.

• In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.

• Graphs are also used to draw activity network diagrams. these diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.

An Activity Network Diagram (AND) also known as an Arrow Diagram or a PERT (Program Evaluation Review Technique) is used to identify time sequences of events which are pivotal to objectives. It is also helpful when a project has multiple activities which need simultaneous management. It is also helpful when a project has multiple activities which need simultaneous management. ANDs help the project development team to create a realistic project schedule by drawing graphs that exhibit:

• The total amount of time needed to complete the project

• The sequence in which activites must be performed

• The activities that must be monitored on a regular basis.

Points to Remeber

• A graphi is basically a collection of vertices (also called nodes) and edges that connect these vertices.

• Degree of a node u is the total number of edges contaiing the node u. When the degree of a node is zero, it is also called an isolated node. A path P is known as a closed path if the edge has the same end-points. A closed simple path with length 3 or more is known as a cycle.

• A graph in which there exists a path between any two of its nodes is called a connected graph. An edge that has identical end-points is called a loop. The size of a graph is the total number of edges in it.

• The out-degree of a node is the number of edges that originate at u.

• The in-degree of a node is the number of edges that terminate at u. A node u is known as a sink if it has a positive in-degree but a zero-out degree.

• Breadth-first search is a graph search algorithm that begins at the root node and explores all te neighboring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbor nodes, and so on, until it finds the goal.

• The epth-first search algorithm progresses by expanding the starting node of G and thus going deeper and deeper until a goal node is found, or until a node that has no children is encountered.

• A transitive closure of a graph is constructed to ansert reachability questions.

• Since an adjacnecy matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The memory used of an adjacency matrix is $O(n^2)$, where n is the number of nodes in the graph.

• Topological sort of a directed acyclic graph G is defined as a linear ordering of its nodes in which each node comes before all the node to which it has outbound of topological sorts.

• A vertex v of G is called an articulation point if removing v along with the edges incident to v results in a graph that has at least two connected components.

• A biconnected graph is defined as a connected graph that has no articulation vertices.

• A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together.

• Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.

• Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph.