Unit 7 Applications of Graphs

Structure:

7.1 Introduction

In the previous unit we discussed the directed graph, types of directed graphs, binary relation as a diagraph, Euler's diagraphs, and Matrix representation of diagraphs. In this unit we will be discussing about the sorting techniques, finding shortest path and some algorithms in the graph structures which are extensively used for varieties of applications. The applications that use graph algorithms are so many. Among them some applications are used for Project scheduling where a technique known as PERT or CPM; Computer graphics systems, topological sorting etc. We will also deal, in this unit, about the problems which are considered to be intractable and discuss about the nondeterministic polynomial complete problems (NPC).

Objectives:

After studying this unit, you should be able to:

- describe topological sorting.
- explain Dijkstra's algorithm of finding shortest paths
- explain the minimum spanning tree and prim's algorithm
- discuss NP-complete problems.

7.2 Topological Sorting

A topological sort defines a linear ordering on those nodes of a directed graph that follows the property: if node **u** is predecessor of node **v** then **v** cannot be the predecessor of node **u**. This property must hold even if u = v, i.e. **u** cannot be its own immediate predecessor and successor. In other words, a topological sort cannot order the nodes in cycle. We can also say that Topological sort is a process of assigning a linear ordering to the vertices of a directed acyclic graph (DAG) so that if there is an arc from vertex ***u*** to vertex **v** then ***u*** appears before **v** in the linear ordering. Topological sorting cannot be applied for linear ordering in the graph with cycles.

Topological Sort was used for network analysis with technique like Program Evaluation and Review Technique (PERT). The main purpose of Topological sort is to order the events in linear manner i.e., first, second, third and so on, with the condition that an event cannot precede other events that must first take place. For e.g., when washing clothes, the washing machine must finish before we put the clothes to dry.

In this section we will deal with linear-time algorithms for computing topological order for a given DAG. It should be noted that DAGs may have more than one possible topological order. The algorithms for computing *all* types of possible orders are more advanced and may not be justifiable in this section to introduce about it. We can see from the following example that the more edge a DAG has, the fewer number of topological orders it has. This is because each edge (u, v) forces u to occur before v, which restricts the number of valid permutations of the vertices.

Let us look at an example of the results of a topological sort given in figure 7.1 and 7.2 below. Figure 7.1 shows the graph with no cycles i.e. the graph is directed acyclic graph (DAG).

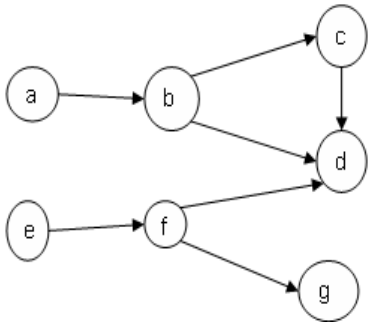

Figure 7.1: Graph with no cycles

Now let us order this graph using topological sorting. In the linear ordering, the successors of each node **u** always appear to the right of **u**. The resulting graph is given below in figure 7.2.

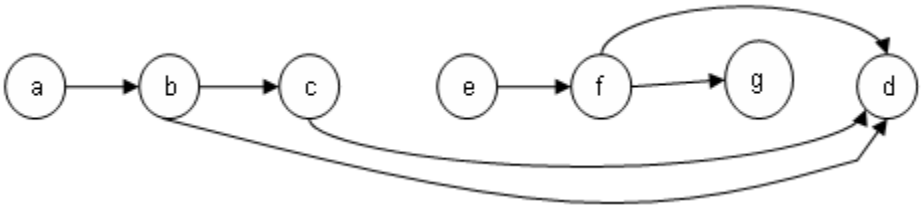

Figure 7.2: Linear ordering of the graph of figure 7.1

Note that all of the edges are pointing in the same direction. *In general, the vertex order produced by a* ***topological sort*** *is not unique i.e. other linear ordering could also be defined to make different order of topological sort.* We could reorder the nodes while maintaining the property of a topologically sorted DAG and get different type of linear orders for sorting.

In this section, Let us apply topological sorting for ordering of terms defined in the given way. Suppose that we have a number of terms T_1, T_2, T_3 ... T_n that are related by pairs (T_i , T_j) such that T_i is used in the definition of T_j . Now to order the terms so that each term T_i appears before each term T_j that uses T_i directly or indirectly in its definition, it is possible to have circular definition, for example, (T_i , T_j), (T_j , T_k) and (T_k , T_i). In such a case, a complete linear ordering of the terms cannot be made.

Let us consider the example given below in figure 7.3 and 7.4. In a graph of figure 7.3, a linear ordering cannot be defined because as already stated that topological sort cannot order the nodes in a cycle.

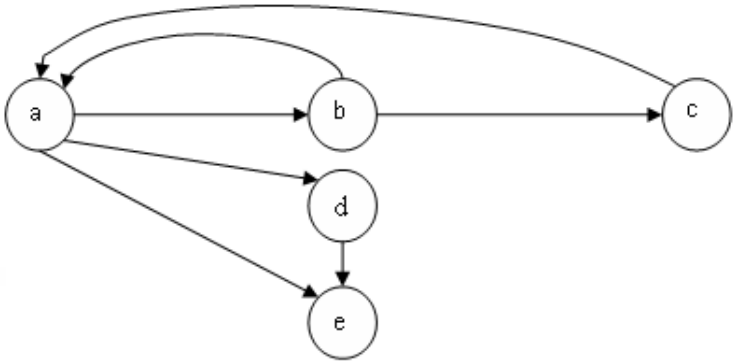

Figure 7.3: Graph with cycles

But if you consider for graph of figure 7.4, topological sort can define the linear ordering as **a – b – d – c – e** as shown in figure 7.5**.**

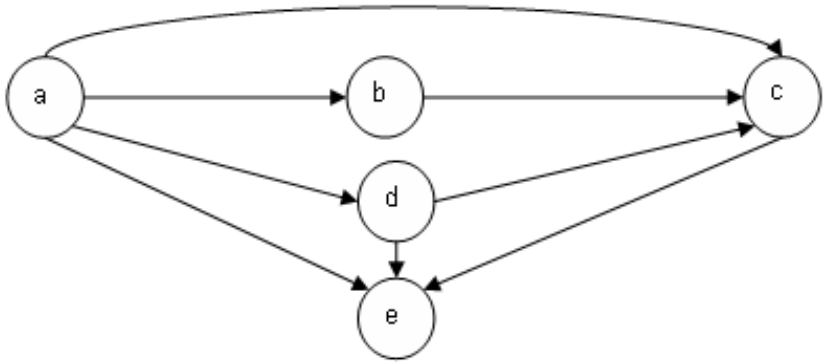

Figure 7.4: Graph with no cycles

In the figure 7.5, the linear ordering of graph of figure 7.4 has been shown. Alternatively, we can also order graph as: **a – d – b – c – e.**

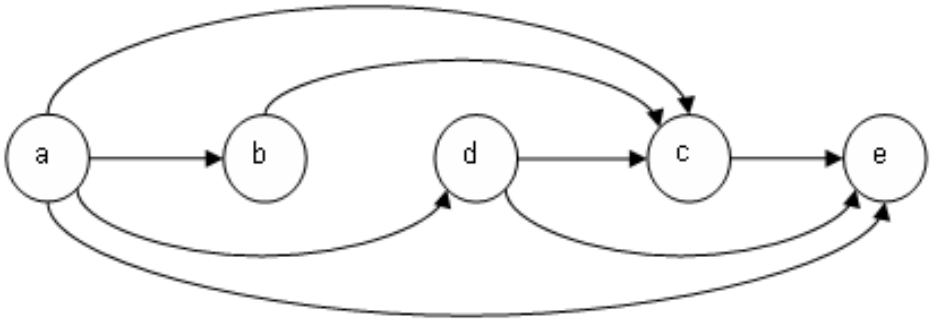

Figure 7.5: linear ordering as a – b – d – c – e

In general, topological sorting specifies a linear ordering for a set of nodes identified by descriptors if a number of edges are identified by ordered pairs of descriptors. Descriptor's successor must appear after that descriptor in linear ordering for topological sort. If it is not possible to specify a complete linear ordering, then the loops that exist among the nodes or descriptors must be accounted for.

Self Assessment Questions

1. DAGs may have more than one possible topological order. (True/False)

2. Descriptor's successor must appear after that descriptor in _____ ordering for topological sort. (Choose correct option)
   a) linear
   b) alphabetical
   c) random
   d) unique

7.3 Weighted Shortest Paths - Dijkstra's Algorithm

This algorithm is used to find the shortest path between the two vertices in a weighted directed graph and it is also very popular and efficient to find each and every path from starting (source) to terminal vertices.

Let $w(v_i , v_j)$ be the weight associated with every edge (v_i , v_j) in a given weighted directed graph G. Let us define that the weights are such that the total weight from vertex $\mathbf{v_i}$ to $\mathbf{v_k}$ via vertex $\mathbf{v_j}$ is $w(v_i , v_j) + w(v_j , v_k)$. Using this form of definition , the weight from a vertex $\mathbf{v_s}$ (source) to vertex $\mathbf{v_t}$ (end of path) in the graph G for a given path (v_s , v_1), (v_1 , v_2), (v_2 , v_3)…… (v_i , v_t) is given by $w(v_s , v_1) + w(v_1 , v_2) + w(v_2 , v_3) +$ ……..$+ w(v_i , v_t)$. Hence for such a graph there may be many possible paths between $\mathbf{v_s}$ and $\mathbf{v_t}$. .

In this algorithm, labels are assigned to each vertex. Length is considered to be the weight of the edge if there is an edge between two vertices. And if there are several edges then the shortest length edge is used. If no edge actually exits then the length is set to infinity. Edge (v_i , v_j) does not necessarily have the same length as edge (v_j , v_i). This fact allows that the different paths with different weights can be there in between two vertices depending upon their direction of travel. The time required by Dijkstra's algorithm is $O(|V|^2)$.

Since label is assigned to each vertex, the label is equal to the distance (weight) from the starting vertex to the end vertex. The starting vertex v_s is supposed to have label 0. A label can be either temporary or permanent. Temporary label is one that has uncertainty along the shortest path where as permanent label lies along the shortest path.

Dijkstra's algorithm is given below in stepwise manner:

Step 1: Assign a temporary label 1 (v_i) = ∞ to all vertices except v_s
Step 2: Mark v_s as permanent by assigning 0 label to it
$1(v_s) = 0$
Step 3: Assign value of v_s to v_r where v_r is last vertex to be madepermanent. $V_r = V_s$

Step 4: If $1(v_i) > 1(v_k) + w(v_k, v_i)$.
$1(v_i) = 1(v_k) + w(v_k, v_i)$.
Step 5: $v_r = v_i$
Step 6: If v_t has temporary label, repeat step 4 to step 5 otherwise the valueof v_t is permanent label and is equal to the shortest path v_s to v_t.
Step 7: Exit.

This technique gradually changes the temporary labels to permanent labels until the end vertex has a permanent label. At each step, the aim is to make the temporary labels shorter by finding paths to the associated vertices using the shortest paths to the permanent labelled vertices. After this the temporary label with the smallest value is made permanent. This process eliminates one temporary vertex at each step, ensuring that the shortest path from $v_{s.}$ to v_t will eventually be found.

Let us consider one simple example of the graph G given in figure 7.6 where we need to find out the shortest path between V_1 to V_2.

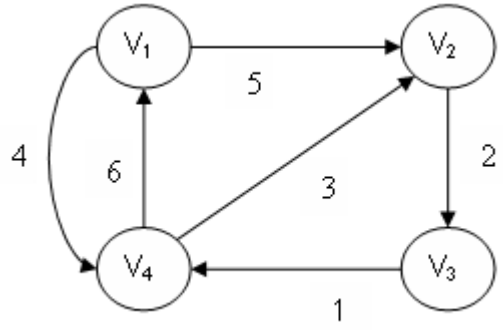

Figure 7.6: Weighted graph

And let the weight matrix of graph be **W=**

	V_1	V_2	V_3	V_4
V_1	0	5	0	4
V_2	0	0	2	0
V_3	0	0	0	1
V_4	6	3	0	0

If we replace the elements, which have 0 with 9999 (with very large value) except diagonal elements, then we get a distance matrix **D=**

	V_1	V_2	V_3	V_4
V_1	0	5	9999	4
V_2	9999	0	2	9999
V_3	9999	9999	0	1
V_4	6	3	9999	0

When we travel from V_1 to V_2 in the graph G then the possible route can be either from vertex V_1 to V_2 or from vertex V_1 to V_4 and then from V_4 to V_2. (Depending upon the graph there can be more than one route). The distances for the possible routes then is represented as 5 and 7 (i.e. 4 + 3) respectively. Similarly to travel from V_1 to V_3, V_1 to V_4 and so on can be found out.

Using Dijkstra's algorithm, if we find out the shortest path then the steps are:
Step 1, Step 2 and Step 3: assigning labels and setting $V_r = V_s = V_1$

	V_s	V_3	V_4	V_t
Label	0	9999	9999	9999
Permanent	Y	N	N	N

Step 4 and Step 5: Redefining temporary labels as:
Label (V_3)= min (9999, 0+9999)= 9999
Label (V_4)= min (9999, 0+ 4)= 4
Label (V_t)= min (9999, 0+5)= 5
Making label (V_4) permanent, $V_r = V_4$

	V_s	V_3	V_4	V_t
Label	0	9999	4	9999
Permanent	Y	N	Y	N

Again repeating step 4 and step 5 and redefining labels as:
Label (V_3)= min (9999, 6+0)= 6
Label (V_t)= min (5, 6+ 3)= 5
Making label (V_3) permanent, $V_r = V_3$

	V_s	V_3	V_4	V_t
Label	0	6	4	9999
Permanent	Y	Y	Y	N

Repeating step 4 and step 5 and redefining
Label (V_t)= min (4, 0 +9999)= 4
Make label (V_t) permanent set $V_r = V_t$

	V_s	V_3	V_4	V_t
Label	0	6	5	4
Permanent	Y	Y	Y	Y

Now label **(**V_t) is permanent. And the shortest path length is 4.
Hence we can find the shortest path between the two vertices using Dijkstra's algorithm.

Self Assessment Questions

3. Dijkstra's algorithm can be applied for unweighted directed graph. (True/False)
4. In Dijkstra's algorithm, _____ are assigned to each vertex.
   a) weight
   b) length
   c) cost
   d) labels

7.4 Minimum Spanning Tree (MST)

We have already discussed in the previous unit, about the spanning tree T which is defined as an undirected tree of a connected graph G which is

composed of all the vertices and the edges necessary to connect all the nodes of graph *G*. Spanning tree has all the vertex that lies in the tree, but without any loop. Hence only one path exits between two nodes. We also found out that a single graph can have many spanning trees depending upon the criteria used to generate it.

We know that a graph may contain redundancy i.e. it can have multiple paths between two vertices. This redundancy may be desirable because in some cases we may need to offer alternative routes to overcome breakdown or overloading of an edge (e.g. road, connection, phone line) in a network. But having redundancies, we always want the cheapest sub-network that connects the vertices of a given graph. So we need to find out the tree of a graph which has a minimum cost or weight.

The total cost or weight of a tree is given by the sum of the weights of the edges in the tree. We assume that the weight of every edge is greater than zero. Then for a given connected, undirected graph G=(V,E), the *minimum spanning tree problem* is to find a tree T=(V,E') such that E' is a subset of E and the cost of T is minimal.

Note that a minimum spanning tree is not necessarily unique. Recall that a tree over n vertices contains (n-1) edges and it can be represented by an array of this many edges. A *minimum spanning tree* would be one with the lowest total cost.

There are few algorithms which are used to get the minimum spanning tree. The name of such algorithms are *Borůvka's algorithm* developed by Czech scientist Otakar Borůvka in 1926, *Prim's algorithm* and *Kruskal's algorithm* which are also known as *greedy algorithms* that run in polynomial time. Another greedy algorithm not as commonly used is the *reverse-delete algorithm*, which is the reverse of Kruskal's algorithm. In this Unit, we will be discussing only about the Prim's minimum spanning tree algorithm.

Prim's Minimum Spanning Tree Algorithm

In Prim's algorithm for the minimum spanning tree problem, the strategy is to begin with a single node in a graph. We start with ($\{v_1\},\{\ \}$), and grow further until it includes all vertices in the given graph. Initially the tree contains just an arbitrary starting node v_1. At each stage a vertex not yet in the tree but closest (lower in weight) to some vertex that is in the tree is found. This

closest vertex is added to the tree. In this way we go on finding the vertex so that knowledge of the distances of the remaining vertices to the tree is known.

Prim's algorithm is very similar to Dijkstra's single-source shortest path algorithm. The principal difference is that the criterion for choosing a new vertex is proximity to the tree rather than to a source. Prim's algorithm also clearly takes $O(|V|^2)$ time.

It's a **greedy** style algorithm and produces a correct result. In the example discussed below, let the distance from each node not in the tree to the tree be the edge of minimal weight between that node and some node in the tree. If there is no such edge then we assume the distance is infinity (but this shouldn't happen).

The steps to obtain minimum spanning tree are as follows:

1) Start with a tree which contains only one node.
2) Identify a node (outside the tree) which is closest to the tree and add the minimum weight edge from that node to some node in the tree and incorporate the additional node as a part of the tree.
3) If there are less then *n – 1* edges in the tree, go to **2** (where n is the number of edges in a graph).

Let us see one example to get a minimum spanning tree for the graph shown in figure 7.7.

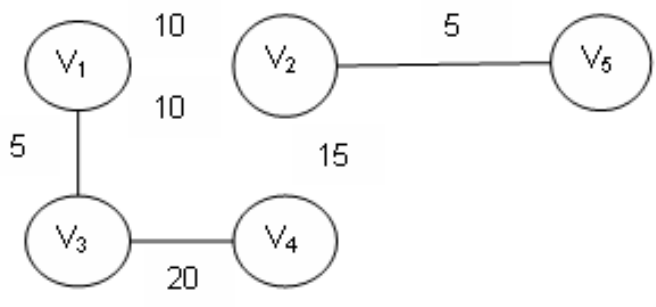

Figure 7.7: A graph with 5 nodes and 6 edges that connect them with the weight associated with each edges

We follow the steps of obtaining spanning tree as said above and obtain minimum spanning tree as depicted in figures 7.8 to 7.12.

Start with only node v_1 in the tree as shown in figure 7.8.

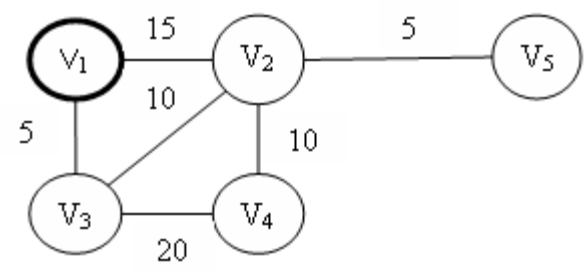

Figure 7.8: 1st step in obtaining minimum spanning tree

Find the closest node to the tree which has edge with lower cost (or weight), and add it. So we get the figure as depicted in figure 7.9.

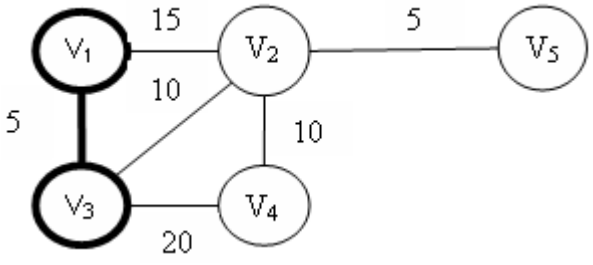

Figure 7.9: 2nd step in obtaining minimum spanning tree

We repeat the above said steps until there are *n – 1* edges in the tree. In this case n=5, and the corresponding figures are shown from figures 7.10 to 7.12.

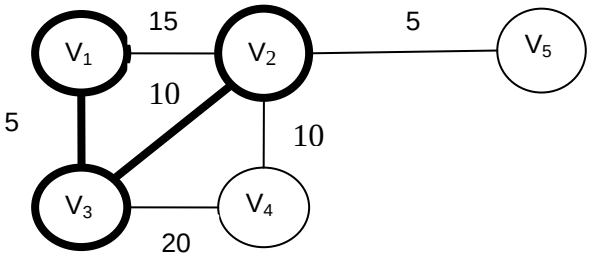

Figure 7.10: 3rd step in obtaining minimum spanning tree

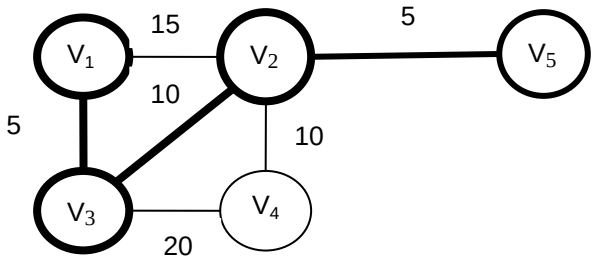

Figure 7.11: 4th step in obtaining minimum spanning tree

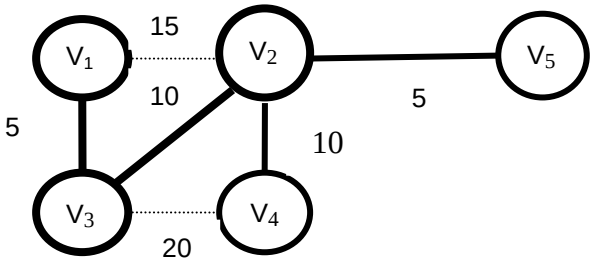

Figure 7.12: Last step: Minimum spanning tree

Self Assessment Questions:

5. Minimum spanning tree is always unique. (True/False)
6. Minimum spanning tree is a graph with weight _____ than or _____ to the weight of every other spanning tree. (Pick the right option)
   a) less, greater
   b) less, equal
   c) greater, equal
   d) greater, always greater.
7. Prim's minimum spanning tree algorithm is known as _____ style algorithm. (Choose correct option)
   a) greedy
   b) unique
   c) linear
   d) sparse

7.5 Introduction to NP-Completeness

The 'NP' stands for 'nondeterministic polynomial time', which says about the fact that a solution for certain type of a problem can be checked (but may not be found) in polynomial time. This class of algorithms informally indicates that there is a polynomial time for checking a solution for a problem whose solution may be difficult to be found out.

In studying the theory of *NP*-Completeness, we do not provide a method of obtaining polynomial time algorithms, nor do we say that algorithms of this complexity do not exist.

The focus of this topic is to show that the problems for which we have no known polynomial time algorithms are computationally related.

A problem is said to be polynomial if there exists an algorithm that solves the problem in time for the expression like $T(n) = O(n^c)$, where c is a constant.

Few of the examples of polynomial problems are (i) Sorting: $O(n \log n) = O(n^2)$ (ii) All-pairs shortest path: $O(n^3)$ (iii) Minimum spanning tree: $O(E \log E) = O(E^2)$ etc.

A problem is said to be exponential if no polynomial-time algorithm can be developed for it and if we can find an algorithm that solves it in $O(n^{u(n)})$, where u(n) goes to infinity as n goes to infinity.

The world of computation can be classified as: (i) Polynomial problems (P), (ii) Exponential problems (E), (iii) Intractable (non-computable) problems (I)
There are so many important classes of problems that we know how to solve it exponentially. We also know that there is no polynomial algorithm known for solving them or we don't know if they can be solved in polynomial methods at all.

However, there exists no non-polynomial lower bound on the solution time. Thus, the question whether they can be solved in polynomial time is still open. But as methods in this class can be translated into each other and if there is a solution for one method implies that solution exist for all of them. And this also means that none of the problems in this class have been solved so far.

However, according to Cormen, Leiserson, and Rivest there are certain polynomial-time algorithms which can be considered tractable because of the following reasons:

1) There are problems which has a running time like $O(n^{20})$ or $O(n^{100})$ and can be called intractable but very few practical problems with such orders of polynomial complexity are seen.
2) If by modelling a problem in an easy form, a problem that can be solved in polynomial time for one model can also be solved in polynomial time for another.
3) The class of polynomial-time solvable problems has nice closure properties (since polynomials are closed under addition, multiplication, etc.)

7.5.1 Definition of NP

A problem is said to be Non-deterministic Polynomial (NP) if

1) Its solution comes from a finite set of possibilities, and
2) It takes polynomial time to verify the correctness of a candidate solution.

In other words, we can also say that a problem is said to be NP if there exists an NP algorithm for it.

The class of NP-complete (Non-deterministic polynomial time complete) problems is a very important and interesting class of problems in Computer Science. This class of problems are interesting due to the following reasons:

1) No polynomial-time algorithm has yet been discovered for any NP-complete problem; at the same time no NP-complete problem has been shown to have a super polynomial-time (for example exponential time) lower bound.
2) If a polynomial-time algorithm is discovered for even one NP-complete problem, then all NP-complete problems will be solvable in polynomial-time.

But it is believed, without any proof so far, that NP-complete problems do not have polynomial-time algorithms and therefore are intractable. The belief is so because if any single NP-complete problem can be solved in polynomial time, then it means every NP-complete problem has a polynomial-time algorithm.

If one can establish a problem as NP-complete, then it can be believed to be intractable. And for such problem it is better to try to design a good *approximation algorithm* rather than finding endlessly an exact solution.

7.5.2 Optimization versus decision problems

Any problem for which the answer is either in the form of YES or NO is called a *decision problem*. An algorithm for a decision problem is termed a *decision algorithm*.

Any problem that involves the identification of an optimal solution (where some value needs to be found out as minimized or maximized) is known as an *optimization problem*. An *optimization algorithm* is used to solve an optimization problem.

We can observe that some problems are optimization problems and some are decision problems. It is very difficult to find the solution for some of the optimization problems where the methodology of finding its solution may be endless. So if we want to apply the theory of NP-completeness to optimization problems then it is better to transform it into decision problems so that it becomes easier to find its solution in Yes-No form. Let us look at an example of how an optimization problem can be transformed into a decision problem.

Example: Consider the problem SHORTEST-PATH where we need to find a shortest path between two given vertices in an unweighted, undirected graph *G = (V, E)*. An instance of such problem consists of a particular graph and two vertices of that graph. A solution is a sequence of vertices in the graph, with perhaps an empty sequence denoting that no path exists. Thus such problem gives a relation that associates each instance of a graph and two vertices with a solution (namely a shortest path in this case). It should be noted that a given instance may have no solution, exactly one solution, or multiple solutions.

A decision problem says PATH can be transformed related to the SHORTEST-PATH problem above as: Given a graph *G = (V, E)*, two vertices v_1, $v_2 \in V$, and a non-negative integer k, does a path exist in G between v_1 and v_2 whose length is at most k?

It should be noted that decision problem PATH is one way of casting the original optimization problem as a decision problem. This has been done by

imposing a bound on the value to be optimized. This is a popular way of transforming an optimization problem into a decision problem.

If an optimization problem is easy then its related decision problem is easy as well. Similarly, if we can provide evidence that a decision problem is hard then we can also provide evidence that its related optimization problem is hard.

Hence it is always better to typecast Optimization problems into decision problems and so better to focus on Yes-No type of answer for Yes-No type of problems.

7.5.3 Classes P and NP

An algorithm is said to be polynomially bounded if its worst-case complexity is bounded by a polynomial function of the input size. A problem is said to be polynomially bounded if there is a polynomially bounded algorithm for it.

P is the class of all decision problems that are polynomially bounded. The implication is that a decision problem $X \in P$ can be solved in polynomial time on a deterministic computation model (such as a deterministic Turing machine).

NP represents the class of decision problems which can be solved in polynomial time by a non-deterministic model of computation. That is, a decision problem $X \in NP$ can be solved in polynomial-time on a non-deterministic computation model (such as a non-deterministic Turing machine). A non-deterministic model can make the right guesses on every move and race towards the solution much faster than a deterministic model.

A deterministic machine, at each point in time, executes an instruction. Depending on the outcome of executing the instruction, it then executes some next instruction, which is unique. A non-deterministic machine on the other hand has a choice of next steps. It is free to choose any that it wishes. For example, it can always choose a next step that leads to the best solution for the problem. A non-deterministic machine thus has the power of extremely good, optimal guessing.

We use NP to designate the class of all non-deterministic polynomial problems. Are the classes *P* and *NP* identical? This is an open problem. It is not hard to show that every problem in *P* is also in *NP,* but it is unclear whether every problem in *NP* is also in *P.*

It can be shown that P $\subseteq$ NP. However, it is unknown whether P = NP. In fact, this question is perhaps the most celebrated of all open problems in Computer Science.

7.5.4 NP-complete problems

NPC is the standard notation for the class of all NP-complete problems. The definition of NP-completeness is based on reducibility of problems. NP-complete problems have the property that any one of them can be reduced to any other in polynomial time. We say that problem X reduces to problem Y if there exists a transform from X to Y.

A decision problem *E* is ***NP-complete*** if *every* problem in the class *NP* is *polynomial-time* reducible to *E.* It can be shown that if a problem *D* is polynomial-time reducible to a problem *E* and *E* is in the class *P,* then *D* is also in the class *P.* It follows that if there exist a polynomial-time algorithm for the solution of any of the NP-complete problems, then there exist polynomial-time algorithms for all of them and P is NP.

Suppose we wish to solve a problem X and we already have an algorithm for solving another problem Y. Suppose we have a function T that takes an input x for X and produces T(x), an input for Y such that the correct answer for X on x is yes if and only if the correct answer for Y on T(x) is yes. Then by composing T and the algorithm for y, we have an algorithm for X. That is to say that :

- If the function T itself can be computed in polynomially bounded time, we say X is polynomially reducible to Y and we write $X \leq_p Y$.
- If X is polynomially reducible to Y, then the implication is that Y is at least as hard to solve as X. i.e. X is no harder to solve than Y.
- It is easy to see that

  $X \leq_p Y$ and $Y \in \mathbf{P}$ implies $X \in \mathbf{P}$.

There are many problems which are NP-complete problems; some selected examples of problems which are said to be NP-complete problems are given below:

1) **Name: Maximum subgraph matching.** Input to this graph is: *Directed* graphs *G(V,A)*, *H(W,B)*; positive integer *k*. And the question is: Is there a subset *R* of *VxW*, i.e. pairs of nodes where one node is from *V* and one from *W*, having size at least *k* and such that for every < *u,v*> and < *x,y*>

in *R* the edge *(u,x)* belongs to *A* if and only if the edge *(v,y)* belongs to *B*?

2) **Name: Fault-detection in directed graphs.** Input to this graph is: A *directed, acyclic* graph *G(V,A)* such that *G* has a unique node *t* with no out-going edges; *I* the set of nodes in *V* having no incoming edges; a positive integer *K*. And the question is: Is there a '*test set*' of size at most *K* that can detect every `*single fault*' in *G* (**N.B.** not *every single fault*' but every `*single fault*'), i.e. a subset *T* of the nodes in *I* such that (i) *T* contains at most *K* nodes. (ii) For every node *v* in *V* there exists a node *u* in *T* such that *v* lies on a directed path from *u* to *t* in *G*?

3) **Name: Minimum Maximal Matching.** Input to this graph is: *n*-node undirected graph *G(V,E)*; positive integers *k*<=|*E*|. And the question is **:** Is there a subset, *F*, of at most *k* edges from *E* that forms a maximal matching in *G*, i.e. no two edges in *F* have a common endpoint and every edge of *G* that is not in *F* has a common endpoint with at least one edge of *F*?

7.5.5 NP-hardness and NP-completeness

NP-complete is the class of problems which are both NP-hard and themselves members of NP.

A problem that is NP-Complete has the property that it can be solved in polynomial time if and only if all other NP-Complete problems can also be solved in polynomial time.

A problem is **NP-hard** if an algorithm to solve it in (deterministic) polynomial time would make it possible to solve all NP problems in polynomial time. In other words, if an NP-Hard problem can be solved in polynomial time, then all NP-Complete problems can be solved in polynomial time. All NP-Complete problems are NP-Hard, but not all NP-Hard problems are NP-Complete.

Informally, an NP-hard problem is a problem that is at least as hard as any problem in **NP**. To add more to this statement, if the problem belongs to **NP** then it would become NP-complete.

It can be shown that if any NP-complete problem is in **P**, then **NP** = **P**. Similarly, if any problem in **NP** is not polynomial-time solvable, then no NP-

complete problem will be polynomial-time solvable. Thus NP-completeness is at the crux of deciding whether or not **NP** = **P**.

If a problem is NP-complete, it does not mean that we cannot find the solution. If the actual input sizes are small in a problem, then an algorithm for such problem with (say) exponential running time may be acceptable. On the other hand, it may still be possible to obtain near-optimal solutions in polynomial-time. Such an algorithm that returns near-optimal solutions (in polynomial time) is called an *approximation algorithm*. Using Kruskal's algorithm to obtain a suboptimal solution for minimum spanning tree for a connected weighted graph is an example of this.

Self Assessment Questions

8. There are certain polynomial-time algorithms which can be considered tractable. (True/False)
9. If the solution is found out to minimize or maximize a given value, then such type of problems are _____ problems.
10. An algorithm that returns near-optimal solutions (in polynomial time) is called _____*algorithm*.
11. A problem is NP-hard if an algorithm to solve it in (deterministic) polynomial time would make it possible to solve all _____ problems in polynomial time. (Pick the right option)
    a) decision
    b) non-deterministic
    c) NP
    d) optimization

7.6 Summary

This unit summaries the fact that the graphs and its algorithms are used in varieties of computer applications. Topological sort is used to order any type of events in a linear fashion where the condition is that the Descriptor's successor must appear after that descriptor. The Topological sort is applicable in DAGs. We also dealt with finding out the shortest path of a Directed weighted graph using Dijkstra's algorithm. Dijkstra's algorithm discusses that there can be many path in between two vertices but the shortest path can be found out between them. This unit also introduced the concept of minimum spanning tree and summarized that the minimum

spanning tree may not be unique. Minimum spanning tree is found out in order to get the least cost in terms of weight of the edges of tree. A *minimum spanning tree* would be one with the lowest total cost.

We also discussed about the NP-complete problems and observed that there are so many problems which may be intractable. This unit discussed that if there exists a polynomial-time algorithm for the solution of *any* of the NP-complete problems, then there exist polynomial-time algorithms for all of them and P is NP. Although the question whether N=NP is still open.

7.7 Terminal Questions

1. Explain topological sorting and its applications.
2. Explain how Dijkstra's algorithm is used to find the shortest path of Directed weighted graph.
3. Explain the Prim's minimum spanning tree algorithm.
4. Explain NPC problems.
5. How can you differentiate NP-Hard and NP problems?
6. Differentiate Decision and optimization problems.

7.8 Answers

Self Assessment Questions

1. True
2. a) linear
3. False
4. d) labels
5. False
6. b) less, equal
7. a) greedy
8. True
9. optimization
10. Approximation
11. c) NP

Terminal Questions

1. Topological sort is a process of assigning a linear ordering to the vertices of a directed acyclic graph (DAG) so that if there is an arc from vertex ***u*** to vertex **v** then ***u*** appears before **v** in the linear ordering. Topological sorting cannot be applied for linear ordering in the graph with cycles. (Refer section 7.2)
2. In Dijkstra's algorithm, label is assigned to each vertex, the label is equal to the distance (weight) from the starting vertex to the end vertex. The starting vertex v_s is supposed to have label 0. A label can be either temporary or permanent. Temporary label is one that has uncertainty along the shortest path where as permanent label lies along the shortest path. (Refer section 7.3 for detail)
3. In Prim's algorithm for the minimum spanning tree problem, the strategy is to begin with a single node in a graph. We start with ($\{v_1\},\{\ \}$), and grow further until it includes all vertices in the given graph. Initially the tree contains just an arbitrary starting node v_1. At each stage a vertex not yet in the tree but closest (lower in weight) to some vertex that is in the tree is found. This closest vertex is added to the tree. In this way we go on finding the vertex so that knowledge of the distances of the remaining vertices to the tree is known. (Refer section 7.4 for detail)
4. NPC is the standard notation for the class of all NP-complete problems. The definition of NP-completeness is based on reducibility of problems. NP-complete problems have the property that any one of them can be reduced to any other in polynomial time. We say that problem X reduces to problem Y if there exists a transform from X to Y. (Refer section 7.5 for detail)
5. All NP-Complete problems are NP-Hard, but not all NP-Hard problems are NP-Complete. Informally, an NP-hard problem is a problem that is at least as hard as any problem in **NP**. To add more to this statement, if the problem belongs to **NP** then it would become NP-complete. (Refer section 7.5)
6. Any problem for which the answer is either in the form of YES or NO is called a *decision problem* whereas any problem that involves the identification of an optimal solution (where some value needs to be found out as minimized or maximized) is known as an *optimization problem.* (Refer section 7.5)