Graph Algorithms

Graph Algorithm

Content:

ELEMENTARY GRAPHS ALGORITHMS

INTRODUCTION

It is a mathematical object that defines such situations. It is a collection of vertices, V, and edges, E. An edge connect two vertices.

It is a path with no vertex repeated. A simple cycle is a simple path except the first and last vertex is repeated and, for an undirected graph, number of vertices >= 3. A tree is a graph with no cycle. A complete graph is a graph is a graph in which all edges are present. A sparse graph is a graph with relatively few edges. A dense graph is a graph with many edges. An undirected graph has no specific direction between the vertices. A directed graph has edges that are "one way". We can go from one vertex to another, but not vice versa. A weighted graph has weights associated with each edge. The weights can represent distances, costs, etc.

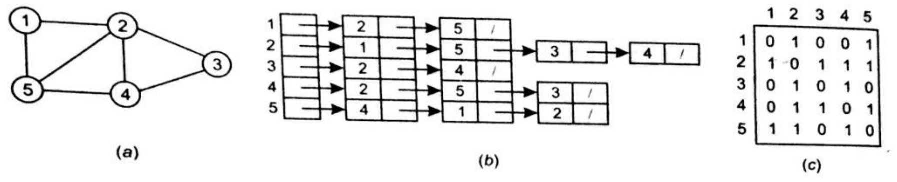
HOW IS A GRAPH REPRESENTED

REPRESENTING GRAPH ASA AN ADJACENCY LIST

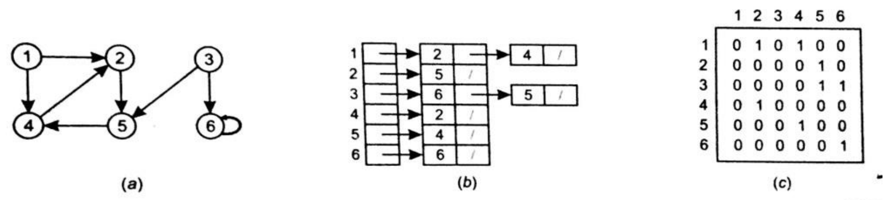
It is an array, which contains a list for each vertex. The list for a given vertex holds all the other vertices that are connected to the first vertex by a single edge.

REPRESENTING GRAPHS AS AN ADJACENCY MATRIX

It is a matrix with a row and column for each vertex. It is 1 if there is an edge between the row vertex and the column vertex. The diagonal may be zero or one. Every edge is represented twice.



Undirected graph : Matrix is symmetric



Directed graph : Matrix is asymmetric

TWO COMMON ELEMENTARY ALGORITHMS FOR TREE SEARCHING ARE:

- Breadth-first search (BFS)
- Depth-first search (DFS).

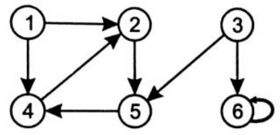
BREADTH FIRST SEARCH

It have a nice property: Every edge of G can be classified into one of three groups. Some edges are in T themselves. Few connect two vertices at the same level of T. and the remaining ones connect two vertices on two adjacent levels. It is not possible for an edge to skip a level.

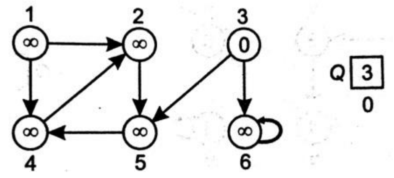
Therefore, It really is a shortest path tree starting from its root. Every vertex has path to the root, with path length equal to its level (just follow the tree itself), and no path can skip a level so this really is a shortest path.

Example: Examine the graph G in Fig. Describe the whole process of breadth first search.

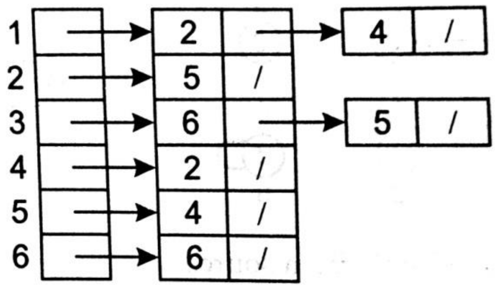
Using vertex 3 as the source.



Solution:



First, we create adjacency-list representation



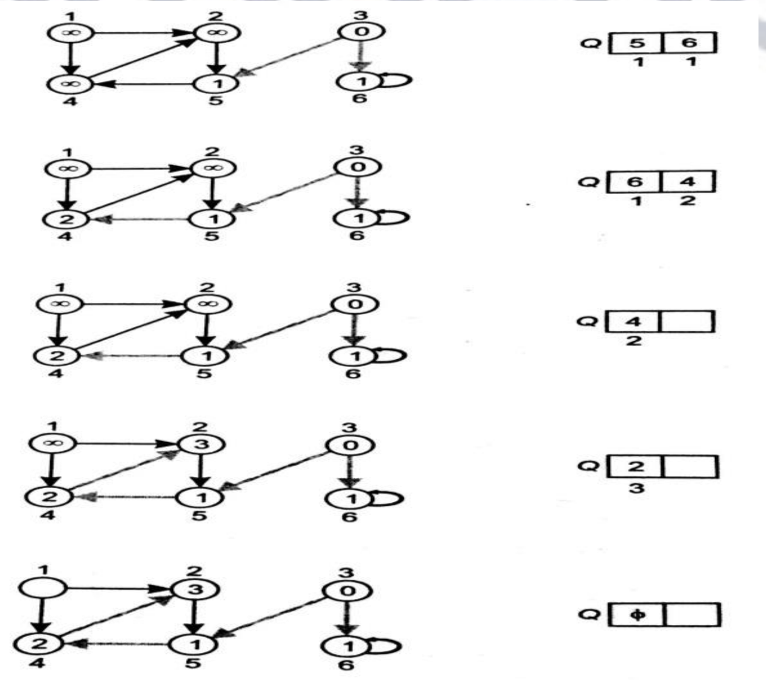
So, adj[4] = {6, 5} so color[6] = GRAY

Color[5] = GRAY

And, d[6] = 1 d[5] = 1

$\pi[6] \leftarrow 3$ $\pi[5] \leftarrow 3$

Now,



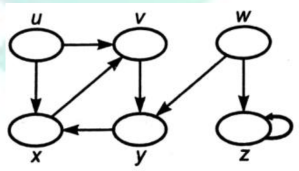
Thus vertex 1 cannot be reachable from source.

DEPTH FIRST SEARCH

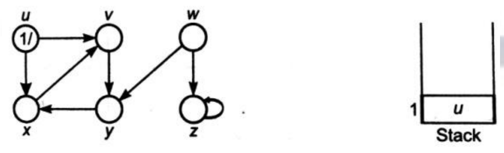
It edges explored out of the most recently discovered vertex v that still has unexplored edges living it. When all of v's edges have been explored, the search still has unexplored edges leaving the vertex from which v was discovered. This method runs until we have launched all the vertices that are reachable from the original source vertex. If any undetected vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire method is repeated until all vertices are discovered.

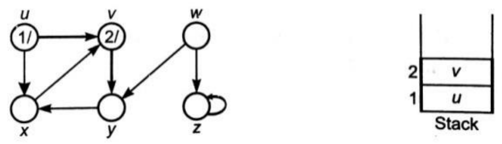
In this, each vertex v has two timestamps the first timestamp d[v] i.e. discovery time records when v is first discovered i.e. grayed, and the second timestamp f[v] i.e. finishing time records when the search finishes examining v's adjacency list i.e, blacked. For every vertex d[u] < f[u]. The running time of DFS is θ (V + E)

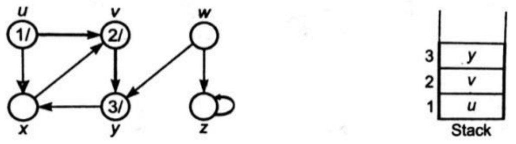
Example: Show the progress of the depth – first – search (DFS) algorithm on a directed graph.

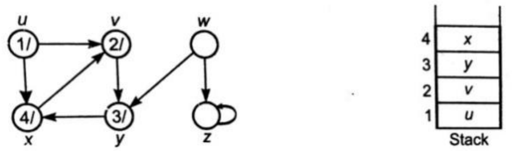


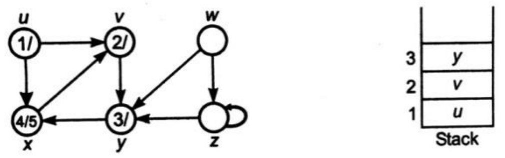
Solution:



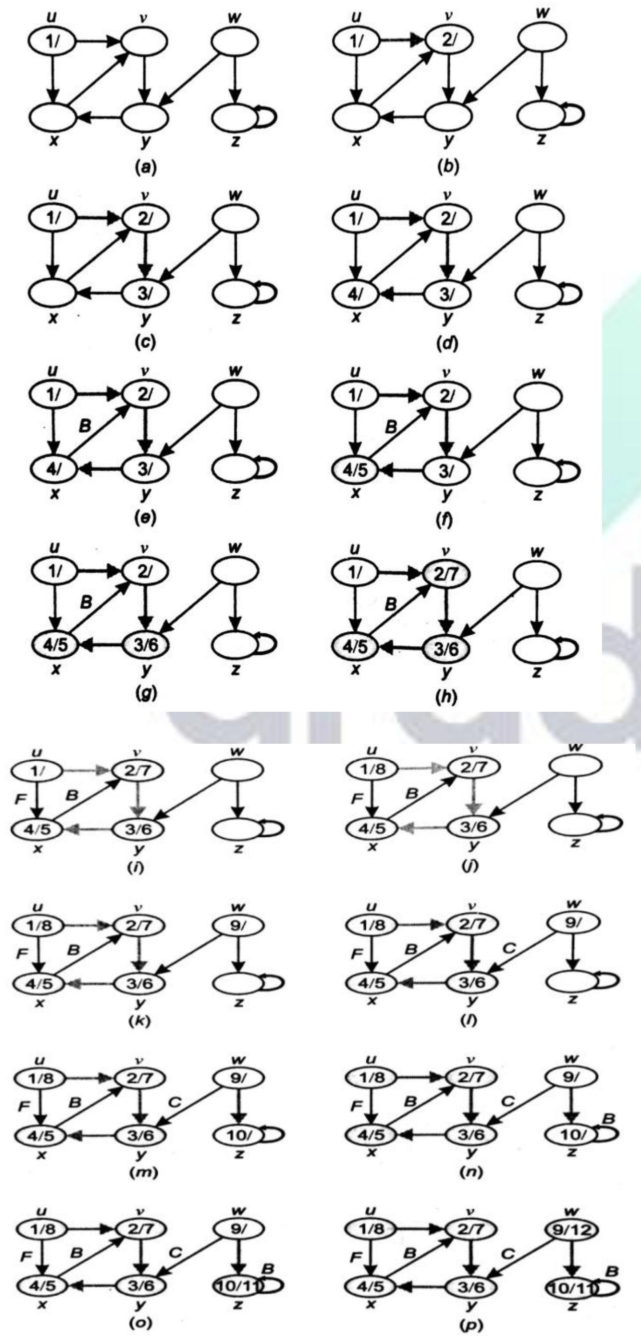








All the steps in DFS are as follows:



TOPOLOGICAL SORT

Directed acyclic graphs or DAG's are used for topological sorts. A topological sort of a directed a cyclic graph G = (V, E) is a linear ordering of $u, v \epsilon V$ such that if (u, v) ∈ E then u appears before v in this ordering. If G is cyclic, no such ordering exists.

TOPOLOGICAL-SORT (G)

1. Call DFS (G) to determine finishing times f[v] for each vertex v
2. Every vertex is finished, insert it onto the front of a linked list.
3. Return the linked list of vertices.

It can also be viewed as placing all the vertices along a horizontal line so that all directed edges go from left to right. DAG's are used in larger applications to determine precedence. We can perform a topological sort in time θ(V + E).

MINIMUM SPANNING TREE

SPANNING TREE

It is a graph just a sub graph that contains all the vertices and is a tree. A graph may have many spanning trees. We can also allocate a weight to every edge, which is a number representing how adverse it is, and use this to allocate a weight to a spanning tree by calculating the sum of the weights of the edges in tree. A lower spanning tree or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree, any undirected graph has a minimum spanning forest.

To describe how to find a lower Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both different in their methodology, but both in the end, end up with the MST. Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connection in determining the MST. Both algorithms are greedy algorithms that run in polynomial time. At every step, one of several possible choices must be made.

KURSKAL'S ALGORITHM

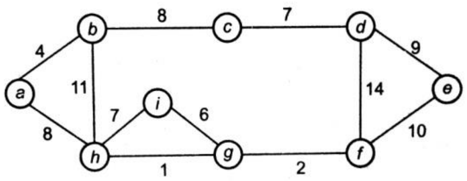
It is an algorithm that finds a minimum spanning tree for a connected weighted graph. It finds a secure edges to add to the developing forest by finding, of all the edges that connect any two trees in the forst, an edge (u, v) of least weight. It means it observe a subset of the edges that forms a tree that involves each vertex, where the total weight of all the edges in the tree is minimized. If graph is not attached, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).

First sort the edges by weight using a comparison sort in O(E log E) time; Next, we use a disjoint-set data structure to keep track of which vertices are in which components. We need to execute O(E) operations, two search operations and possibly one union for every edge.

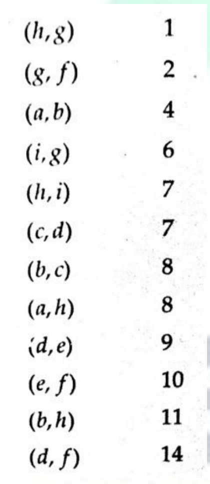
Even a simple disarrange-set data structure such as disjoint-set forests with union by rank can perform O(E) operations in O(E log V) time. Thus the total time is

$$O(E \log E) = O(E \log V).$$

Example: Find the minimum spanning tree of the following graph using Krushal's algorithm.



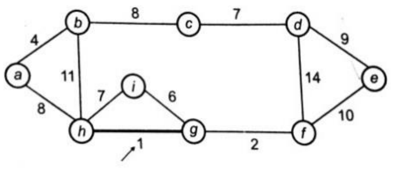
Solution: Find we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight, i.e,.



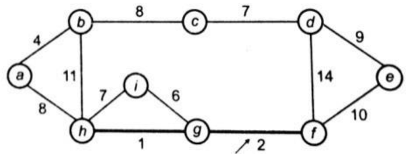
(h, g)	1
(g, f)	2
(a, b)	4
(i, g)	6
(h, i)	7
(c, d)	7
(b, c)	8
(a, h)	8
(d, e)	9
(e, f)	10
(b, h)	11
(d, f)	14

Now, check for each edge (u, v) whether the end points u and v belong to the same tree, then the edge (u, v) cannot be added. Otherwise, the two vertices belongs to different trees and the edge (u, v) is added to A and the vertices in the two tress are merged in by UNION procedure.

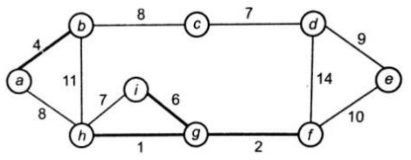
So, first take (h, g) edge



The (g, f) edge.

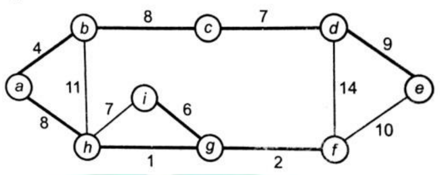


Then (a, b) and (I, g) edges are considered and forest becomes.



Now, edge (h, i). Both h and I vertices are in same set, thus it creates a cycle. So this edge is discarded.

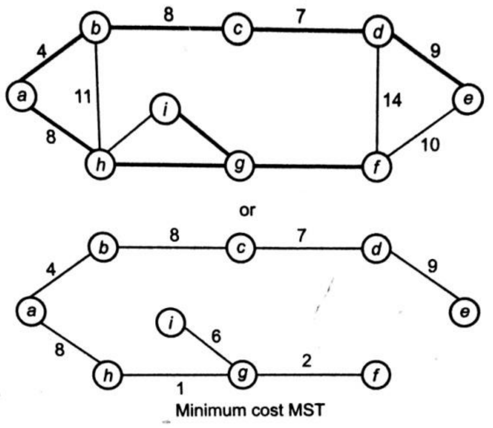
edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered and forest becomes.



In (e, f) edge both end points e and f exist in same tree so discarded this edge.

Then (b, h) edge, it also produce a cycle.

After edge (d, f) and the final spanning tree is shown as in dark lines.

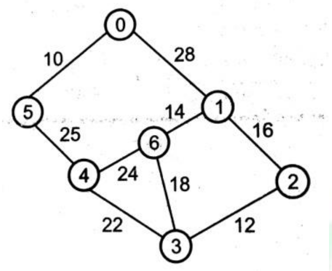


Minimum cost MST

PRIM'S ALGORITHM

The main idea of Prim's algorithm is similar to that of Dijkstra's algorithm (discussed later) for finding shortest path in a given graph. It has the property that the edges in the set A always form a single tree. We being with some vertex v in a given graph G = (V, E), defining the initial set of vertices A. then, in each emphasis, we select a minimum-weight edge (u, v), connecting a vertex v in the set A to the vertex u outside of set A. then vertex u is brought in to A. this method is frequent until a spanning tree is formed. Like krushkal's algorithm, here too, the important fact about MSTs is we always choose the smallest-weight edge joining a vertex inside set A to the one outside the set A.

Example: Create minimum cost spanning tree for the following graph using Prim's algorithm.



Solution: First we initialize the priority queue Q. to carry all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXRACT-MIN(Q) produce, now u =r and Adj[u] = [5, 1].

Removing u from the set Q and adds it to the set V – Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in the tree.

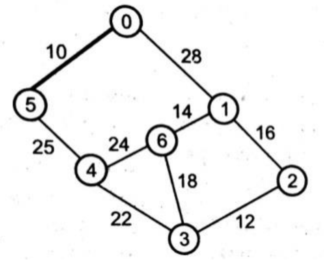
Key [5] = ∞

w [0, 5) = 10 i.e., w(u, v) < key [5]

so, π[5] = 0 and key [5] = 10

and key [1] = ∞

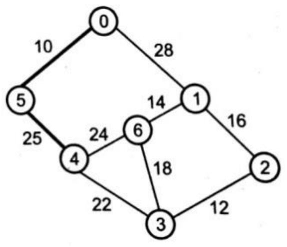
w(0, 1) = 28 i.e., w(u, v) < key[5] so π[1] = 0 key[1] = 28



Now, by EXTEACT_MIN (Q) Remove 5 because key [5] = 10 which is minimum so u = 5

Adj[5] = [4]

w(u, v) < key[v] then key[4] = 25



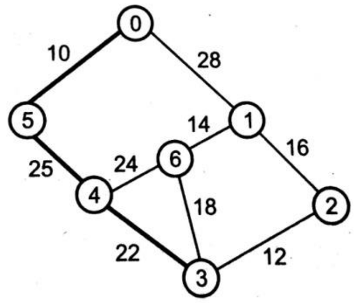
Now remove 4 because key [4] = 25 which is minimum so u =4

Adj[4] = [6, 3]

w(u, v) < key[v] then key [6] = 24

key[3] = 22

Now remove 3 because key[3] = 22 is minimum so u = 3.



Adj[3] = {4, 6, 2}

4 ≠ Q key [6] = 24 now becomes key [6] = 18

Key [2] = 12

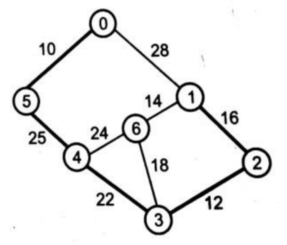
Now, in Q, key[2] = 12, key[6] = 18, key[1] = 28

By EXTRACT_MIN(Q) Remove 2, because key[2] = 12 is minimum.

Adj[2] = {3, 1}

3 ≠ Q.

Now key[1] = 16.

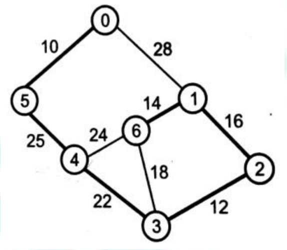


By EXTRACT_MIN(Q) Remove 1 because key[1] = 16 is minimum.

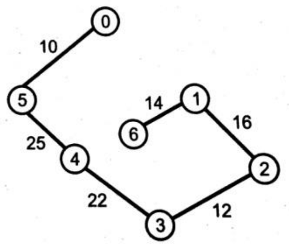
Adj[1] = {0, 6, 2}

{0, 2} ∉ Q key[6] = 14.

Now becomes Q contains only one vertex 6. By EXTRACT_MIN remove it



Thus, the final spanning tree is



SINGLE-SOURCE SHORTEST PATHS

DIJKSTRA'S ALGORITHM

It is named behind its discoverer, Dutch computer scientist Edsger Dijkastra, is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights i.e. we suppose that w(u, v) ≥ 0 for each edge (u, v) ∈ E.

It keeps a set S of vertices whose final shortest-path weights from the sources have already been examined, for all vertices v ∈ S, we have d[v] = δ(s, v). The algorithm frequently choose the vertex u ∈ V – S with the minimum shortest – path estimate, inserts u into S, and relaxes all edges leaving u. we preserve a priority queue Q that holds all the vertices in v – s, keyed by their d values. Graph G is represented by adjacency lists.

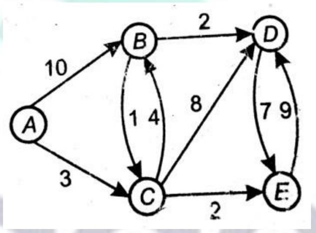
Because it always selects the "lightest" or "closest" vertex in V – S to insert into set S, we say that it uses a greedy strategy.

It bears some similarity to both breadth-first search and Prime's algorithm for computing minimum spanning trees. It is like breadth-first search in that set S corresponds to the group of black vertices in a breadth-first search; just as vertices in S have their final shortest-path weights, so black vertices in a breadth-first search have their correct breadth-first distances. It is like Prim's algorithm in both algorithms use a priority line to find the "lightest" vertex outside a given set, insert this vertex into the set, and adjust the weights of the remaining vertices outside the set accordingly.

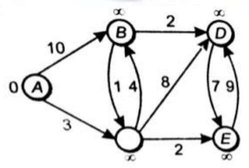
ANALYSIS

The running time of Dijkstra's algorithm is O(V^2)

Example:



Initialize:

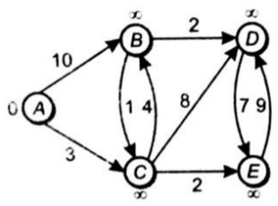


Q:

A	B	C	D	E
0	∞	∞	∞	∞

S : { }

"A" ← EXTRACT – MIN(Q):

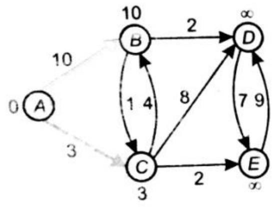


Q:

A	B	C	D	E
0	∞	∞	∞	∞

S : {A}

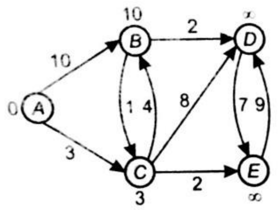
Relax all edges leaving A:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

S : {A}

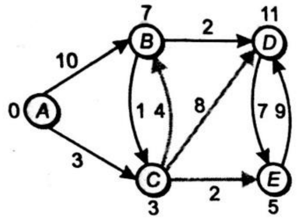
“C” ← EXTRACT – MIN(Q):



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–

S : {A,C}

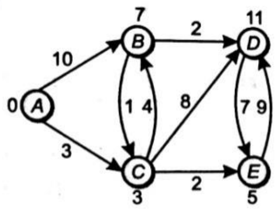
Relax all edges leaving C:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

S : {A,C}

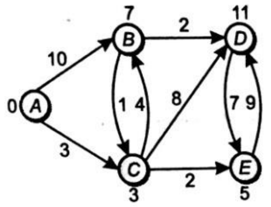
“E” ← EXTRACT-MIN(Q):



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	–	–
		7		11	5

S : {A,C, E}

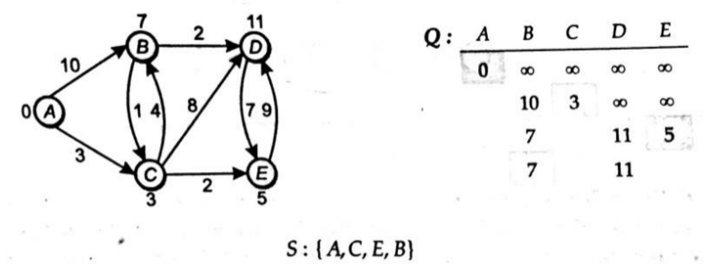
Relax all edges leaving E:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

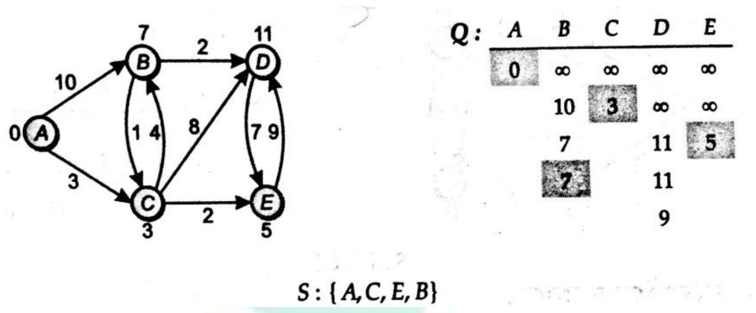
S : {A,C, E}

"B" ← EXTRACT – MIN(Q):



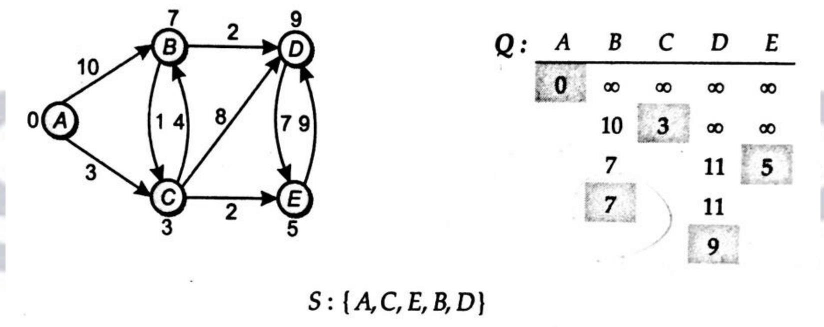
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

Relax all edges leaving B:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

"D" ← EXTRACT – MIN(Q):



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

ALL – PAIRS SHORTEST PATHS

INTRODUCTION

The all-parts shortest path problem can be considered the mother of all routing problems. If aims to compute the shortest path from each vertex v to every other u. Using standard single source algorithms, we can expect to get a naïve implementation of O(n^3) if we use Dijkstra for example, i.e. running a O(n^2) process n times. Likewise, if we use the Bellman-Ford-Moore algorithm on a dense graph, it'll take about O(n^4), but handle negative arc-length too.

Storing all paths explicitly can be very memory expensive indeed, as we need one spanning tree for each vertex, this is often impractical in terms of memory consumption, so these are considered as all-pairs shortest distance problems, which aim to find just the distance from each to each node to another. We want the output in tabular form: the entry in u's row and v's column should be the weight of a shortest path from u to v.

Three approaches for improvement:

Algorithm	Cost
Matrix multiplication	$O(V^3 \lg V)$
Floyed-Warshall	$O(V^3)$
johnson	$O(V^2 \lg V + VE)$

NP - COMPLETENESS

CLASSES OF PROBLEMS

We can categorize the problems into the following broad classes:

1. Problem which cannot even be defined formally.
2. Problem which can be formally defined but cannot be solved by computational means.
3. Problem which, though theoretically can be solved by computational means, yet are infeasible, i.e, these problems require so large amount of computational resources that practically is not feasible to solve these problems by computational means. These problems are called intractable or infeasible problems.
4. Problems that are called practical or theoretically not difficult to solve by computational means. The determine feature of the problem is that for every instance of any of these problems, there exists a Deterministic Turing Machine that solves the problem having time-complexity as a polynomial function of the area of problem. The case of problem is denoted by P.
5. Last, but probably most interesting class includes large number of problems for each of which it is not known whether it is in P or not in P.

These problems fall somewhere between class (3) and class (4) given above. However, for every o the problems in the class, it is known that it is in NP, i.e., each can be solved by at least one-Non-Deterministic Turning Machine, the difficulty time of which is a polynomial function of the size of the problem. Now, we can go still further and categorize the problems as.

OPTIMIZATION PROBLEM vs. DECISION PROBLEMS

Decision Problems have Yes/No answers. Optimization Problems require answers that are optimal configurations. Decision problems are "easier" than optimization problems; if we can show that a decision problem is hard that will imply that its corresponding optimization problem is also hard.

THE CLASSES P AND NP

We define P(Polynomial) as the class of decision problems that are solvable by algorithms that runs in time polynomial in the length of the input. That, is a decision question as in P if there exists an exponent k and an algorithm for the question that runs in time $O(n^k)$ where n is the length of the input.

P forcefully catch the class of practically solvable problems. Or at least that is the conventional wisdom. Individual that runs in time 2^n requires double the time if one adds one character to the input. Something that runs in polynomial time does not endure from this problem.

Class P: The set of all polynomially solvable problems.

- P is closed under addition, multiplication and composition.
- P is independent of particular formal models of computation or implementations.
- Any problem not in P is hard.
- Problem in P does not automatically have an structured algorithm.

THE CLASS NP (NON DETERMINISTIC POLYNOMIAL)

The set of all problems that can be solved if we always guess correctly what computation path we should follow. Roughly speaking, it includes problems with exponential algorithms but has not proved that they cannot have polynomial time algorithms. A language L ∈ NP is and only if there exists a polynomial-tree verification algorithm A for L.

Note that P _ NP. At this time we do not know if P = NP or P _ NP.

EXAMPLE OF LANGUAGES IN NP

Let us examine the following examples of decision problems.

- HAM-CYCLE = {(G) | G is a Hamiltonian graph}
- CIRCUIT-SAT = {(C) | C is a satisfiable Boolean circuit}
- SAT = {(ϕ | ϕ is satisfiable bollean formula}
- CNF - SAT = {(ϕ) ϕ is a satisfiable boolean formula in CNF}
- 3 - CNF - SAT = {(ϕ) | ϕ is a satsifiable Boolean formula in CNF}
- CLIQUE = {(G, K) | G is an undirected graph with a clique of size k}.
- IS = {(G, K) | G is an undirected graph with an independent set of size k}
- VERTEX - COVER = {(G, K) | undirected graph G has a vertex cover of size k}
- TSP = {(G, c, k) | G = (V, E) is a complete graph, c: V × V → Z is a cost function, k ∈ Z and G has a travelling salesman tour with cost at most k}
- SUBNET - SUM = {(S, t) | there is a subnet S’ _ S such that t = Σ S}.

Theorem

If $L_1 \leq {}_pL_2$ and $L_2 \in P$, then $L_1 \in P$.

Proof. $L_2 \in P$ means that we have a polynomial time algorithm A_2 for L_2. Since $L_1 \leq {}_pL_2$, we have a polynomial-time transformation f mapping input x for L_1 to an input for L_2. Combining these, we get the following polynomial-time algorithm for solving L_1:

1. Take input x for L_1 and compute f(x);
2. Run A_2 on input f(x), and return the answer found (for L_2 on f(x)) as the answer for L_1 on x.

Every steps (1) and (2) takes polynomial time. So the combine algorithm takes polynomial time. Hence $L_1 \in P$.

NOTE: This does not imply that if $L_1 \leq {}_pL_2$ and $L_1 \in P$, then $L_2 \in P$. This statement is not true.

NP - COMPLETENESS

Polynomial - time depletion provide a formal means for showing that one problem is at least as hard as another, to within polynomial - time factor. That is, if $L_1 \leq {}_pL_2$, then L_1 is not more than polynomial factor harder than L_2, which is why the "less than or equal to" notation for reduction is mnemonic. We can define the set of NP - Complete languages, which are the hardest problems in NP.

A langue L _ $\{0, 1\}*$ is NP - Complete if it satisfies the following two properties:

1. $L \in NP$; and
2. For every $L' \in NP$, $L' \leq {}_pL$

If a language L satisfies property 2, but not necessarily property 1, we say that L is NP - hard.

We use the notation $L \in NPC$ to denote that L is NP - complete.

Theorem

If any NP - complete problem is polynomial time solvable, then P = NP. If any problem in NP is not polynomial time solvable, then all NP complete problems are not polynomial time solvable.

PROVING NP - COMPLETENESS

To prove that a problem P is NP - complete, we have following methods:

Method 1 (direct proof):

a. P is in NP.
b. All problems in NP - complete can be reduced to P.

Example: Conjunctive Normal Form (CNF) Satisfiability problem

Method 2 (equally general but potentially easier):

a. P is in NP.
b. Find a problem P’ that has already been proven to be in NP - Complete.
c. Show that P’ $\leq_p$P.

TECHNIQUES FOR NP - COMPLETE PROBLEM

Techniques for dealing with NP - complete problems in practice are:

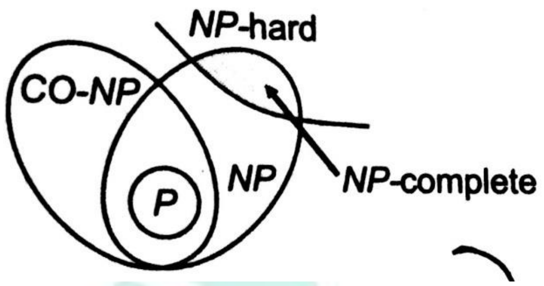
1. Backtracking 2. Branch and Bound 3Approximation lgorithms

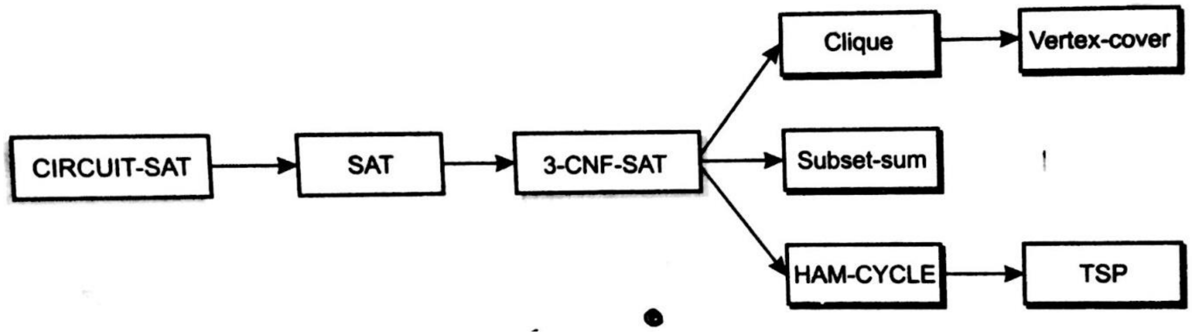
Examples of NP - Complete problems:

1. Circuit satisfiablilty	2. Formula satisfiability
3. 3 - CNF satisfiability	4. Clique
5. Vertex cover	6. Subnet - sum
6. Hamiltonian cycle	8. Traveling salesman
9. Sub graph isomorphism	10. Integer programming

11. Set - partition	11. Graph coloring

A problem is NP - Complete if it is both NP - hard and an element for NP. NP - complete problems are the hardest problems in NP. If anyone finds a polynomial - time algorithm for even one NP - complete problem, then that would imply a polynomial - time algorithm for every NP - complete program. Literally thousands of problems have been shown to be NP - complete, as a polynomial - time algorithm for one (i.e. all) of them seems incredibly unlikely.





All proofs ultimately follow by reduction from the NP - completeness of CIRCUIT - SAT.

gradeup SUPER

Gradeup UGC NET
Super Superscription

Features:

1. 7+ Structured Courses for UGC NET Exam
2. 200+ Mock Tests for UGC NET & MHSET Exams
3. Separate Batches in Hindi & English
4. Mock Tests are available in Hindi & English
5. Available on Mobile & Desktop

Gradeup Super Subscription, Enroll Now

gradeup.co