-: Graph Coloring and Partitioning :-

→ Suppose that you are given a graph G with n-vertices and are asked to paint its vertices such that no two adjacent vertices have the same color. What ~~ot~~ is the minimum number of colors that you would require? This constitutes a coloring problem.

→ Having painted the vertices, you can group them into different sets - each set containing vertices of same color. This is a partitioning problem.

→ The coloring and ~~partion~~ partitioning can, of course, be performed on edges or vertices of a graph. In case of a ~~plats~~ planar graph, one may be interested in coloring the regions.

→ Earlier we came across the subject of ~~partioning~~ partitioning the edges of a given graph into sets with some ~~spei~~ specified properties. Example:-

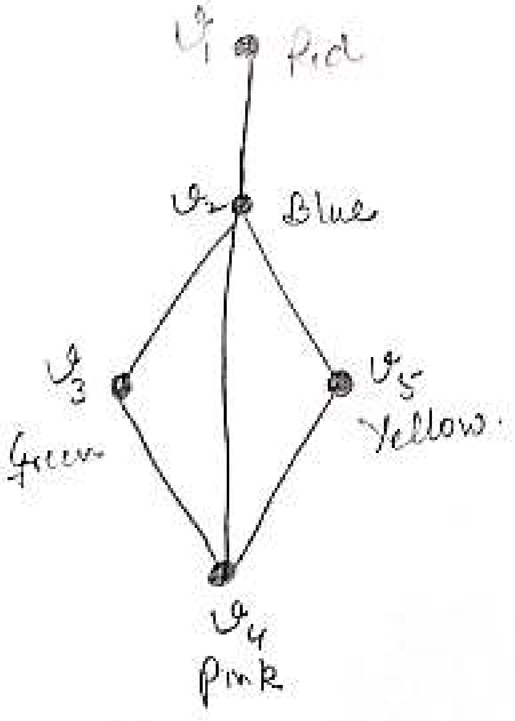
- Finding a Spanning tree in connected graph is equivalent to partioning the ~~of~~ edges into two sets one containing branches of a tree while the other containing the remaining edges.

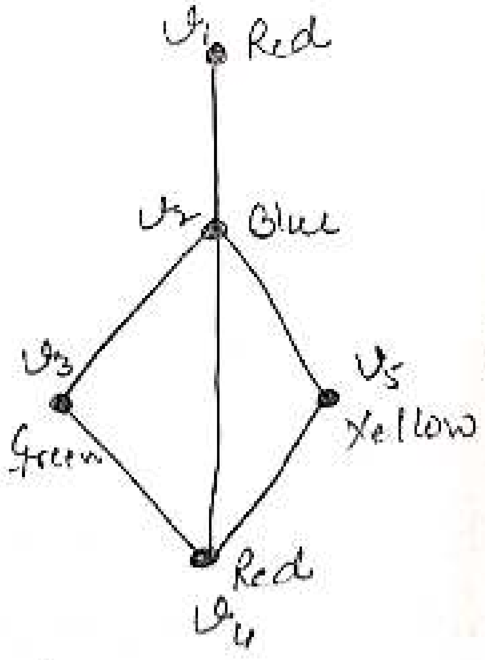
Scanned by CamScanner

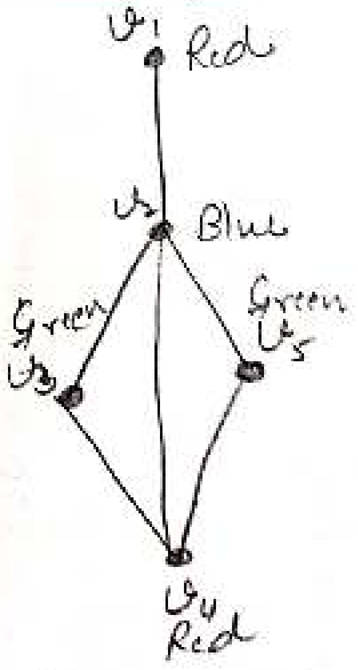
→ ~~Partitioning~~ Partitioning is applicable to many practical problems, such as coding theory, partitioning of logic in digital computers, and state reduction of sequential machines.

Definition:- • Painting all the vertices of a graph with colors such that no two adjacent vertices have the same color is called the proper ~~colour~~ coloring.

• A graph in which every vertex has been assigned a color according to a proper coloring is called a properly colored graph.







(5- Colors are used) (4- Colors are used) (3- Colors are used).

• Three different proper colorings of a graph.

→ The proper coloring which is of interest to us is one that requires the minimum number of colors. A graph G that requires K-different colors for its proper coloring, and no less, is called a K-Chromatic graph, and the number K is called the chromatic number of G.

- Previous example is 3- chromatic graph.

→ • Vertex coloring of one component of a disconnec-ted graph has no effect on the coloring of the other components.

- Parallel edges b/w two vertices can be replaced by a single edge without affecting adjacency of vertices.

- Self-loops must be disregarded.

Hence it is sufficient to consider coloring problems we need to consider only simple, connected graphs.

→ Some observations that follow directly from the definition are.

1) A graph consisting of only isolated vertices is 1- chromatic.

Scanned by CamScanner

2). A graph with one or more edges is atleast 2- chromatic.

3). A complete graph of n-vertices is n-chromatic, ~~at~~ as all its vertices are adjacent.

4). A graph consisting of simply one circuit with $n \geq 3$ vertices is 2-chromatic if n is even, and 3-chromatic if n is odd.

Theorem:- Every tree with two or more vertices is 2- chromatic.

Proof:-

- Select any vertex v in the given tree T. Consider T as a rooted tree at vertex v.
- Paint v with color 1. Paint all vertices adjacent to v with color 2. Next paint the vertices adjacent to these using color 1. Continue this process ~~##~~ till every vertex in T has been painted.
- Now in T we find that all vertices at odd distances from v have color 2, while v and vertices at even distances from v have color 1.
- Now along any path in T the vertices are

of alternating colors. Since there is one and only one path b/w any two vertices in a tree, no two adjacent vertices have the same color.

- Thus T has been properly colored with two colors. One color would not have been enough. Hence the chromatic no. of tree is 2.

#. Though a tree is 2-chromatic, not every 2-chromatic graph is a tree. Example:- Simple circuit with even vertices.

Theorem:- A graph with at least one edge is 2-chromatic iff it has no circuits of odd length.

Proof:- • Let G be a connected graph with circuits of only even lengths. Consider a spanning tree T in G.

- Using previous theorem, this spanning tree T is properly colored with two colors.

- Now add the chords to T one by one. Since G had no circuits of odd ~~leng~~ length, the end vertices of every chord being replaced are differently colored in T.

- Thus G is colored with two colors, with no adjacent vertices having the same color. That is G is 2-chromatic.

- Conversely, if G has a circuit of odd length, we would need at least three colors just for that circuit. Thus the theorem.

Theorem:- (only statement)
- If d_{max} is the maximum degree of the vertices in a graph G,

  Chromatic no. of G $\leq 1 + d_{max}$,

- Brooks showed that this upper bound can be improved by 1 if G has no complete graph of $d_{max}+1$ vertices. In that case,

  Chromatic no. of G $\leq d_{max}$.

#. A graph G is called bipartite if its vertex ~~#~~ set V can be decomposed into two disjoint subsets V_1 and V_2 s.t. every

Scanned by CamScanner

edge in G joins a vertex in V_1 with a vertex in V_2.
Thus every tree is a bipartite graph.

- A bipartite graph can have no self-loops.
- A set of $\parallel^{lel}$ edges b/w a pair of vertices can all be replaced with one edge without affecting bipartiteness of a graph.
- Every 2-chromatic graph is bipartite.
- Every bipartite graph is 2-chromatic, with one trivial exception; a graph of two or more isolated vertices and with no edges is bipartite but is 1-chromatic.

-: Chromatic Partitioning :-

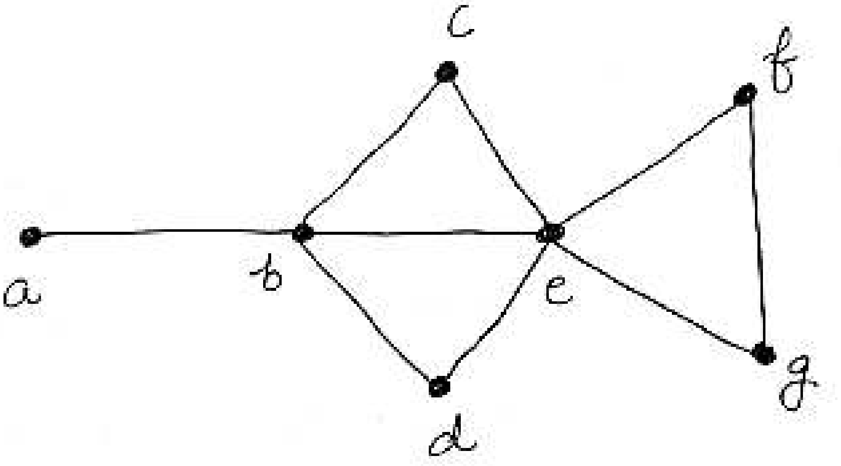
Definition:- A set of vertices in a graph is said to be independent set of vertices or simply and independent set (or an internally stable set) If no two vertices in the set are adjacent.
In example on page 1:-

$$\{v_1, v_4\}, \{v_2\} \text{ and } \{v_3, v_5\}.$$

are Independent sets.

- A maximal independent set (or maximal internally stable set) is an independent set to which no other vertex can be added without destroying its independence property.

Example.



→ {a, c, d} is an independent set.

→ {a, c, d, f} is a maximal independent set. {b, f} is another maximal independent set.

→ • The problem at hand now is to find all the maximal independent set, ~~with~~ with ~~largest no. of~~ largest no. of vertices.

- The number of vertices in the largest independent set of a graph G is called the independence number (or coefficient of

Scanned by CamScanner

internal stability), $\beta(G)$.

→ Consider a K-chromatic graph G of n-vertices properly colored with K different colors. Since the largest no. of vertices in G with the same color cannot exceed the independence no. $\beta(G)$, we have the inequality:-

$$\beta(G) \geq n/K.$$

→ <u>Finding a Maximal Independent Set</u>:-

~~This prog problem~~

- A reasonable method of finding a maximal independent set in a graph will be to start with any vertex v of G in the set.
- Add more vertices to the set, selecting at each stage a vertex that is not adjacent to any of those already selected.
- This procedure will ultimately produce a maximal independent set. This set, however, is not necessarily a maximal independent set with a largest no. of vertices.

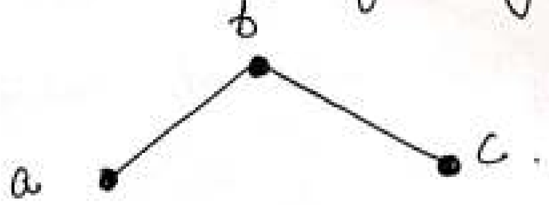
Scanned by CamScanner

→ The next task is to find all the maximal Independent sets :-

- We shall use Boolean arithmetic on the vertices. Let each vertex in the graph be treated as a Boolean variable.
- Let the logical (or Boolean) sum $a+b$ denote the operation of including vertex a or b or both; Let the logical multiplication ab denote the operation of including both vertices a and b, and let the Boolean complement a' denote that vertex a is not included.
- We shall use the following identities :-

$$aa = a$$
$$a + a = a$$
$$a + ab = a$$

Algorithm :- We shall example the algorithm with help of following example :-



Scanned by CamScanner

→ Write the sum ~~Φ~~ ϕ Such that If there is edge b/w a & b vertices then product ab is included in ϕ, In above example:-

$$\phi = ab + bc$$

→ then write ϕ' (complement of ϕ) s.t. If ab is in ϕ the a+b is included in ϕ ie sum is changed to product and product is changed to sum. ie.

$$\phi' = (a'+b')(b'+c')$$

→ Open ϕ' using the identities given before,

$$\phi' = a'b' + a'c' + \underbrace{b'b'}_{\downarrow} + b'c'$$

$$= a'b' + a'c' + \underbrace{b' + b'c'}_{b'}$$

$$= \underbrace{a'b'} + a'c' + \underbrace{b'}$$

$$\phi' = b' + a'c'$$

→ Each term in sum gives a maximally

independent set.

- If b' is in ϕ' then b is not in maximal independent set, a' and c' are not in ϕ' then a & c is in maximal independent set.

Hence the maximal independent sets are.

$\{a, c\}$ and $\{b\}$.

→ • Once all the maximal independent sets of G have been obtained, we find the size of the one with the largest number of vertices to get the independence number $\beta(G)$.

- The independence no. of the above example is 2

→ • To find the chromatic no. of G, we must find the minimum no. of these (maximal independent) sets, which collec- -tively include all the vertices of G.

- In example above,

$\{(a,c), (b)\}$ are all the

Scanned by CamScanner

~~Is~~ vertices of Graph in example.

- ~~Since~~ The no. of (maximal) independent sets which forms the graph is the chromatic no. of graph. In above example, the chromatic no. is 2.

→ Given a simple, connected graph G, partition all vertices of G into the smallest possible no. of disjoint, independent sets. This problem, known as the chromatic partitioning of graphs.

{(a,c),(b)} is the Chromatic partitioning of the graph in example.

→ Chromatic partition ~~ini~~ of a graph may not be unique. ~~always~~

→ A graph that has only one chromatic partition is called a uniquely colorable graph.

Scanned by CamScanner

-: Chromatic Polynomial :-

→ In general, a given graph G of n-vertices can be properly colored in many different ways using a sufficiently large no. of colors. This property of a graph is expressed elegantly by means of a polynomial. This polynomial is called the chromatic polynomial.

→ Definition:- The value of the chromatic polynomial $P_n(\lambda)$ of a graph with n-vertices gives the no. of ways of properly coloring the graph, using λ or fewer colors.

→ Let C_i be the different ways of properly coloring G using i different colors. Since i colors can be chosen out of λ colors in

$\binom{\lambda}{i}$ different ways.

there are $C_i\binom{\lambda}{i}$ different ways of properly coloring G using exactly i colors out of λ colors.

Scanned by CamScanner

Since i can be any +ve integer from 1 to n (it is not possible to use more than n colors on n-vertices), the chromatic polynomial is a sum of these terms, that is

$$P_n(\lambda) = \sum_{i=1}^{n} C_i \binom{\lambda}{i} = C_1 \frac{\lambda}{1!} + C_2 \frac{\lambda(\lambda-1)}{2!} + \text{---} + C_n \frac{\lambda(\lambda-1) \text{---} (\lambda-(n-1))}{n!}$$

→ Each C_i has to be evaluated individually for the given graph. For example, any graph with even one edge requires atleast two colors for proper coloring and therefore

$$C_1 = 0$$

→ A graph with n-vertices and using n-different colors can be properly colored in $n!$ ways, that is,

$$C_n = n!$$

Following ~~glimpse~~ theorems provide a glimpse into the colorful world of chromatic polynomials.

Scanned by CamScanner

Theorem:- A graph of n-vertices is a complete graph iff its chromatic polynomial is

$$P_n(\lambda) = \lambda(\lambda-1)(\lambda-2) \text{ --- } (\lambda-n+1).$$

Proof:-
- With λ colors, there are λ different ways of coloring any selected vertex of a graph.
- A second vertex can be colored properly in exactly $(\lambda-1)$ ways, the third in $(\lambda-2)$ ways, the fourth in $(\lambda-3)$ ways, —, and the nth in $(\lambda-n+1)$ ways iff every vertex is adjacent to every other. ~~Th~~ That is, iff the graph is complete.

Theorem:- (only statement) An n-vertex graph is a tree iff its chromatic polynomial is

$$P_n(\lambda) = \lambda(\lambda-1)^{n-1}.$$

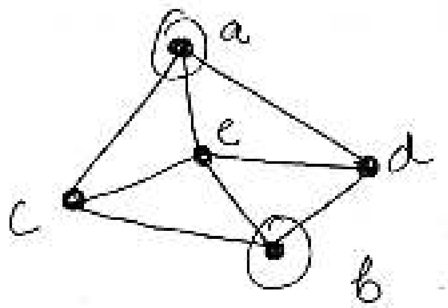
Theorem:- (only statement) Let a and b be two non-adjacent vertices in a graph G. Let G' be a graph obtained by adding an edge b/w a and b. Let G'' be a ~~stay~~ simple

Scanned by CamScanner

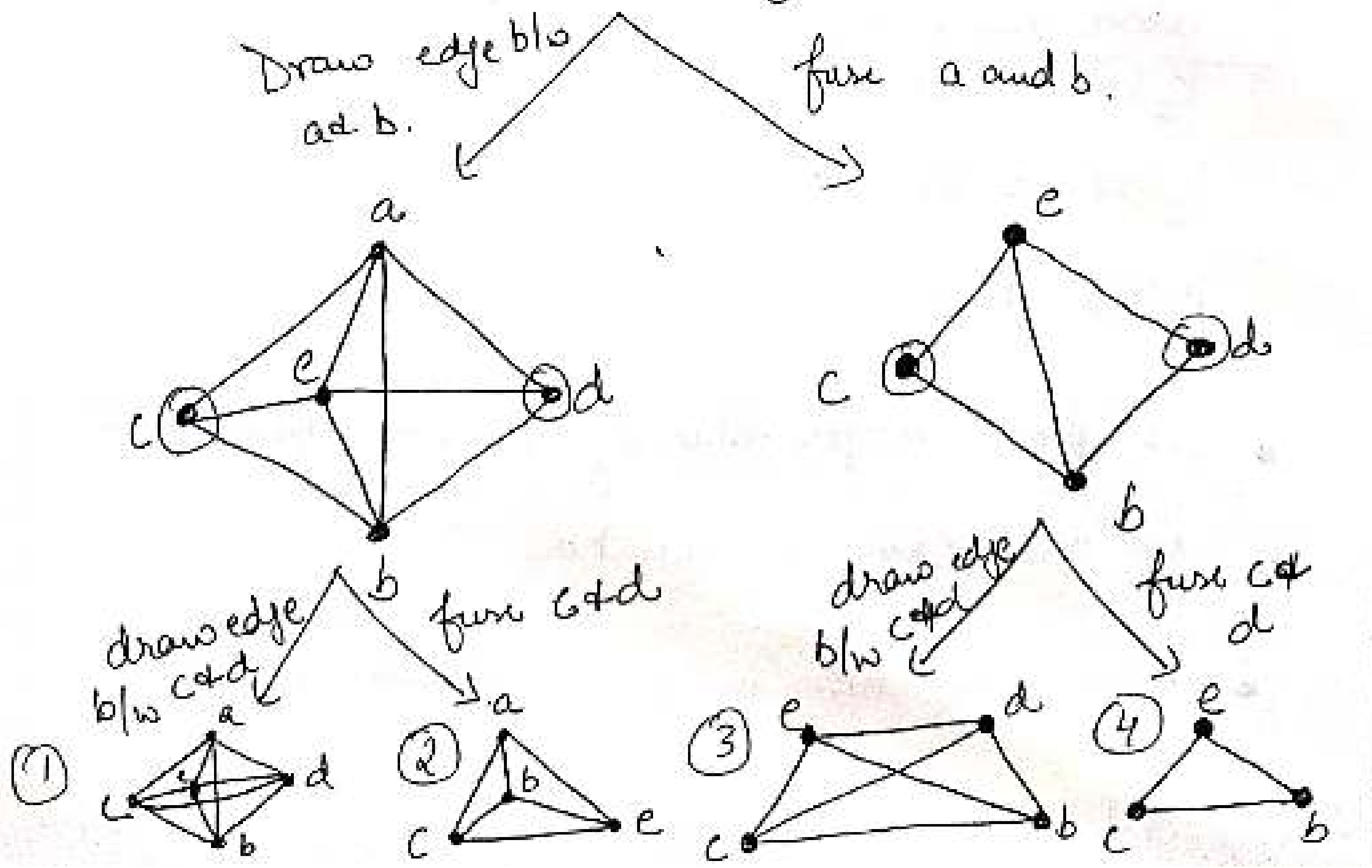
graph obtained from G by fusing the vertices a and b together and replacing sets of $\parallel$ lel e.g. edges with single edges. Then

$$P_n(\lambda) \text{ of } G = P_n(\lambda) \text{ of } G' + P_{n-1}(\lambda) \text{ of } G''$$

Example.
- Example to illustrate the use of above thm.
- Write the chromatic polynomial of the following graph:-



a & b are not adjacent.



All these are complete graphs; and we stop here.
Hence the chromatic polynomial is

$$P_5(\lambda) = \underset{\text{(① complete graph)}}{\lambda(\lambda-1)(\lambda-2)(\lambda-3)(\lambda-4)} + \underset{\text{((②+③) complete graph)}}{2\lambda(\lambda-1)(\lambda-2)(\lambda-3)}$$

$$+ \underset{\text{(④ complete graph)}}{\lambda(\lambda-1)(\lambda-2)}.$$

$$= \lambda(\lambda-1)(\lambda-2)\left[1 + 2(\lambda-3) + (\lambda-3)(\lambda-4)\right]$$

$$= \lambda(\lambda-1)(\lambda-2)(\lambda^2 - 5\lambda + 7)$$

is the chromatic polynomial.

→ Chromatic no. from Chromatic polynomial:-

In above example:-

$P_5(1) = 0$

$P_5(2) = 0$

$P_5(3) \neq 0$

- The first integer value of λ s.t. $P_n(\lambda) \neq 0$ is the chromatic number

- Chromatic no. in above example is 3.

Scanned by CamScanner

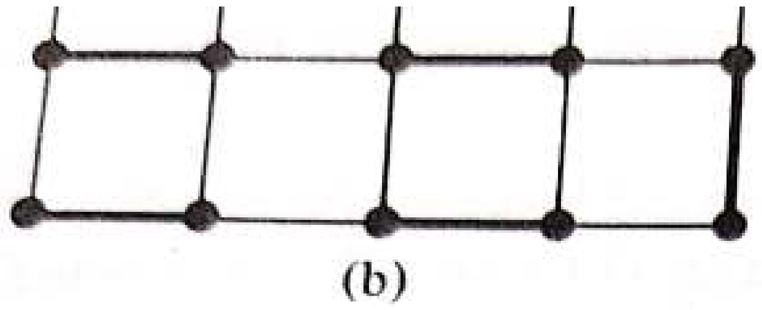


Fig. 8-13 Two dimer coverings of a graph.

8-6. FOUR-COLOR PROBLEM

So far we have considered proper coloring of vertices and proper coloring of edges. Let us briefly consider the *proper coloring of regions* in a planar graph (embedded on a plane or sphere). Just as in coloring of vertices and

Scanned by CamScanner

edges, the regions of a planar graph are said to be properly colored if no two contiguous or *adjacent regions* have the same color. (Two regions are said to be adjacent if they have a common edge between them. Note that one or more vertices in common does not make two regions adjacent.) The proper coloring of regions is also called *map coloring*, referring to the fact that in an atlas different countries are colored such that countries with common boundaries are shown in different colors.

Once again we are not interested in just properly coloring the regions of a given graph. We are interested in a coloring that uses the minimum number of colors. This leads us to the most famous conjecture in graph theory. The conjecture is that every map (i.e., a planar graph) can be properly colored with four colors. The *four-color conjecture*, already referred to in Chapter 1, has been worked on by many famous mathematicians for the past 100 years. No one has yet been able to either prove the theorem or come up with a map (in a plane) that requires more than four colors.

That at least four colors are necessary to properly color a graph is immediate from Fig. 8-14, and that five colors will suffice for any planar graph will be shown shortly.

Two remarks may be made here in passing. Paradoxically, for surfaces more complicated than the plane (or sphere) corresponding theorems have been proved. For example, it has been proved that seven colors are necessary and sufficient for properly coloring maps on the surface of a torus.† Second, it has been proved that all maps containing less than 40 regions can be properly colored with four colors. Therefore, if in general the four-color conjecture is false, the counterexample has to be a very complicated and large one.

Vertex Coloring Versus Region Coloring: From Chapter 5 we know that a graph has a dual if and only if it is planar. Therefore, coloring the regions of a planar graph G is equivalent to coloring the vertices of its dual G^*, and

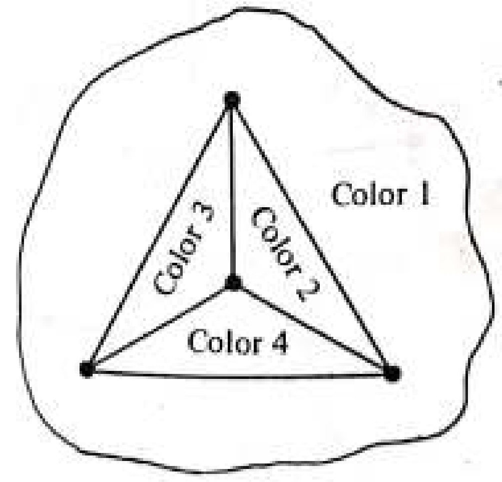


Fig. 8-14 Necessity of four colors.

†In fact, the Heawood map-coloring theorem gives the exact number of colors required for every orientable surface more complicated than that of a sphere. See page 136, [1-5], or page 94, [1-2].

Scanned by CamScanner

vice versa. Thus the four-color conjecture can be restated as follows: *Every planar graph has a chromatic number of four or less.*

Five-Color Theorem: We shall now show that every planar map can be properly colored with five colors:

THEOREM 8-11

The vertices of every planar graph can be properly colored with five colors.

Proof: The theorem will be proved by induction. Since the vertices of all graphs (self-loop-free, of course†) with 1, 2, 3, 4, or 5 vertices can be properly colored with five colors, let us assume that vertices of every planar graph with $n - 1$ vertices can be properly colored with five colors. Then, if we prove that any planar graph G with n vertices will require no more than five colors, we shall have proved the theorem.

Consider the planar graph G with n vertices. Since G is planar, it must have at least one vertex with degree five or less (Problem 5-4). Let this vertex be v.

Let G' be a graph (of $n - 1$ vertices) obtained from G by deleting vertex v (i.e., v and all edges incident on v). Graph G' requires no more than five colors, according to the induction hypothesis. Suppose that the vertices in G' have been properly colored, and now we add to it v and all edges incident on v. If the degree of v is 1, 2, 3, or 4, we have no difficulty in assigning a proper color to v.

This leaves only the case in which the degree of v is five, and all the five colors have been used in coloring the vertices adjacent to v, as shown in Fig. 8-15(a). (Note that Fig. 8-15 is part of a planar representation of graph G'.)

Suppose that there is a path in G' between vertices a and c colored alternately with colors 1 and 3, as shown in Fig. 8-15(b). Then a similar path between b and d, colored alternately with colors 2 and 4, cannot exist; otherwise, these two paths

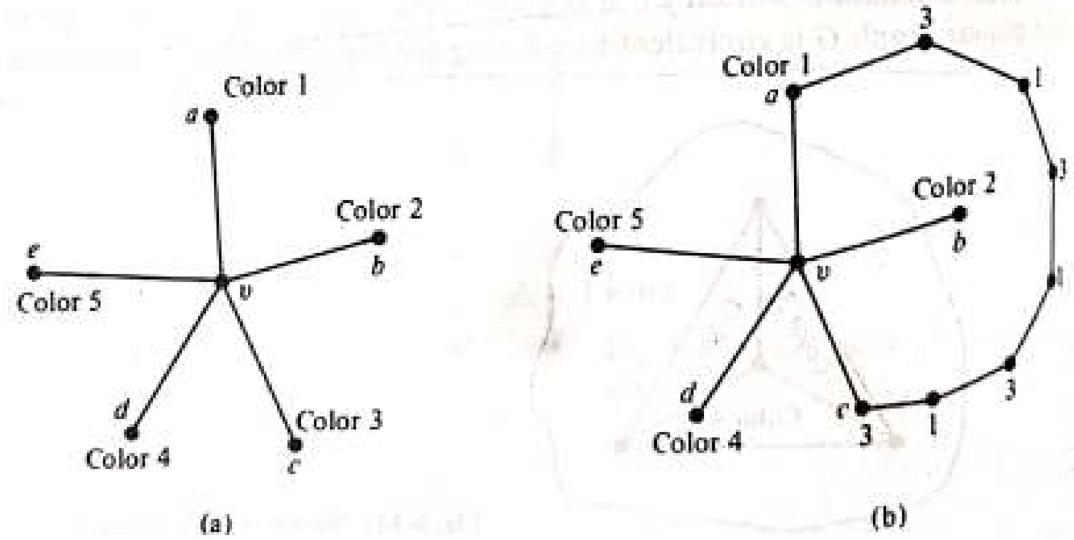


Fig. 8-15 Reassigning of colors.

†See "Regularization of a Planar Graph" in this section.

Scanned by CamScanner

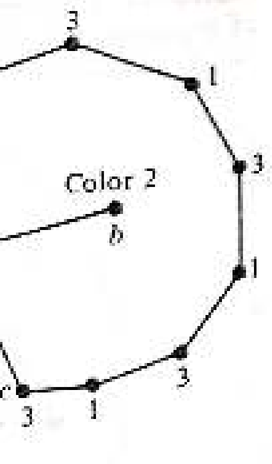
as follows: *Every*

anar map can be

with five colors.

rtices of all graphs
perly colored with
ith $n - 1$ vertices
ny planar graph G
have proved the

anar, it must have
is vertex be v.
eting vertex v (i.e.,
e colors, according
ave been properly
the degree of v is
to v.
d all the five colors
wn in Fig. 8-15(a).
h G'.)
colored alternately
ath between b and
ise, these two paths



(b)

intersect and cause G to be nonplanar. (This is a consequence of the Jordan curve theorem, used in Section 5-3, also.)
If there is no path between b and d colored alternately with colors 2 and 4, starting from vertex b we can interchange colors 2 and 4 of all vertices connected to b through vertices of alternating colors 2 and 4. This interchange will paint vertex b with color 4 and yet keep G' properly colored. Since vertex d is still with color 4, we have color 2 left over with which to paint vertex v.
Had we assumed that there was no path between a and c of vertices painted alternately with colors 1 and 3, we would have released color 1 or 3 instead of color 2. And thus the theorem. ■

Regularization of a Planar Graph: Removing every vertex of degree one (together with the pendant edge) from the graph G does not affect the regions of a planar graph. Nor does the elimination of every vertex of degree two, by merging the two edges in series (Fig. 5-6), have any effect on the regions of a planar graph.

Now consider a typical vertex v of degree four or more in a planar graph. Let us replace vertex v by a small circle with as many vertices as there were incidences on v. This results in a number of vertices each of degree three (see Fig. 8-16).

Performing this transformation on every vertex of degree four or more in a planar graph G will produce another planar graph H in which every vertex is of degree three. When the regions of H have been properly colored, a proper coloring of the regions of G can be obtained simply by shrinking each of the new regions back to the original vertex.

Such a transformation may be called *regularization* of a planar graph, because it converts a planar graph G into a regular planar graph H of degree three. Clearly, if H can be colored with four colors, so can G. Thus, for map-coloring problems, it is sufficient to confine oneself to (connected) planar, regular graphs of degree three. And the four-color conjecture may be restated as follows:

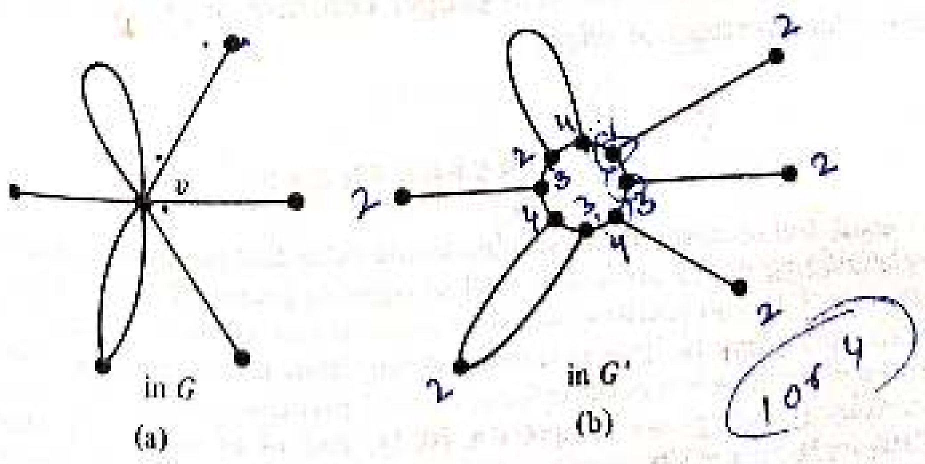


Fig. 8-16 Regularization of a graph.

Scanned by CamScanner

The regions of every planar, regular graph of degree three can be colored properly with four colors.

If, in a planar graph G, every vertex is of degree three, its dual G^* is a planar graph in which every region is bounded by three edges; that is, G^* is a triangular graph. Thus the four-color conjecture may again be restated as follows: The chromatic number of every triangular, planar graph is four or less.

Scanned by CamScanner

-: Shortest - Path Problems :-

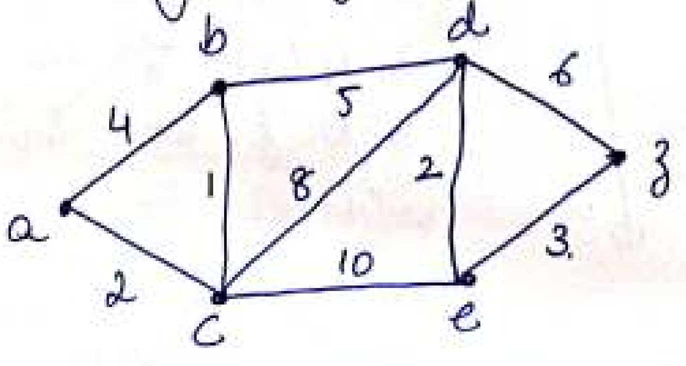
→ Major example (or application) of shortest path problems is solving the ~~network~~ travelling salesman problem or equivalent problems.

→ Algorithm we shall study for this is Dijkstra's Algorithm.

→ This will gives us shortest-path b/w any two given vertices. By shortest-path we mean that in a weighted graph, the shortest-path b/w two pts. 'a' and 'b' is the ~~small~~ path with ~~smallest~~ least weight.

→ We shall discuss the algrorithim along with the example.

Q Use Dijkstra's algorithm to find the length of a shortest path b/w the vertices a and z in the following weighted graph.



Scanned by CamScanner

→ Starting from 'a' we go to next vertex in path using the following assertion :-

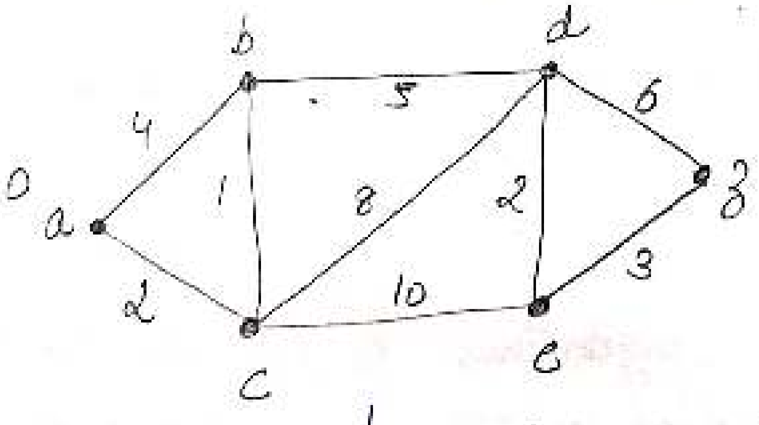
At the k th iteration:-

(i) the label of every vertex in S is the length of a shortest path from 'a' ~~the~~ this vertex.

(ii) the label of every vertex not in S is the length of a shortest path from 'a' ~~to~~ to this vertex that contains only (besides the vertex itself) vertices in S.

~~(iii)~~ (iii) At each step circle the vertex if all the possibilities to reach that ~~graph~~ vertex are exhausted.

→ At 'a' we start with zero length and we have the following graph:-

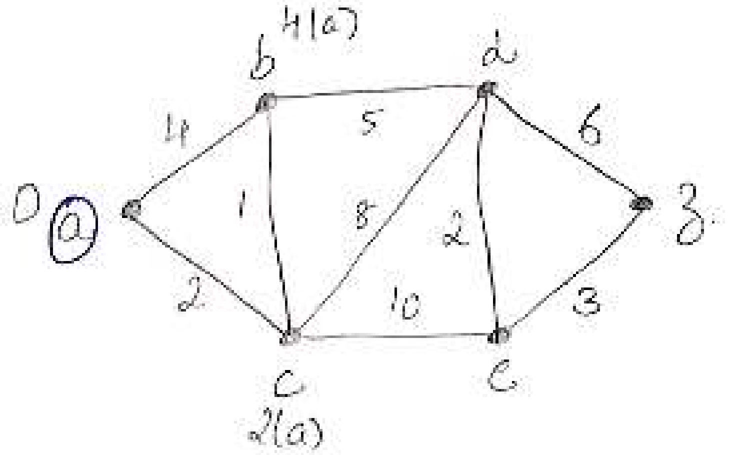


Circle 'a' ↓

From a the ~~adjacent~~ adjacent vertices are b and c. ~~The~~ path length on b will be 4(a). ∴ in going from a to b we have the weight of edge as 4.

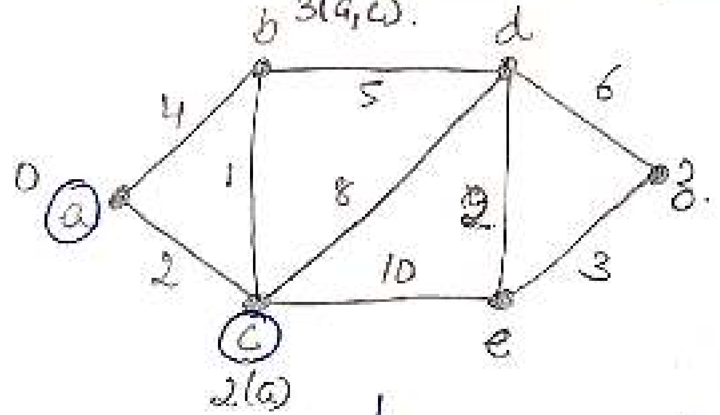
Similarly, on c we have 2(a).

Scanned by CamScanner



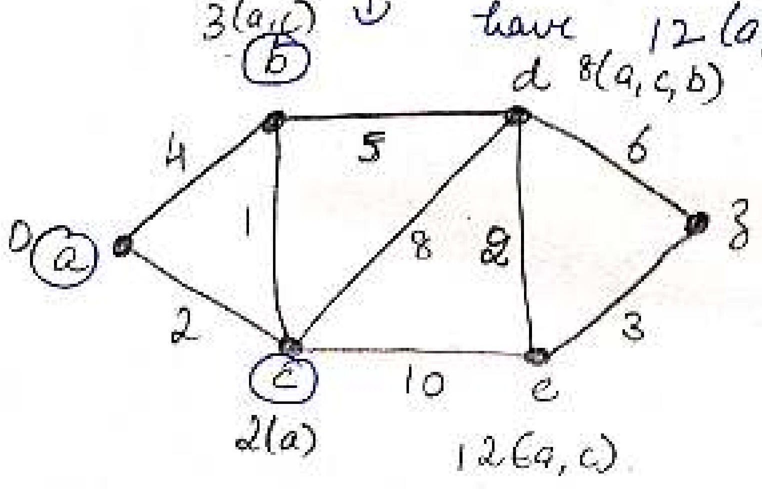
Circle 'c'

In going from a to b, we also have an alternate path $a \to c \to b$ and the weight of this path is 3(a, c), which is less than 4(a) hence replace.



Circle 'b'

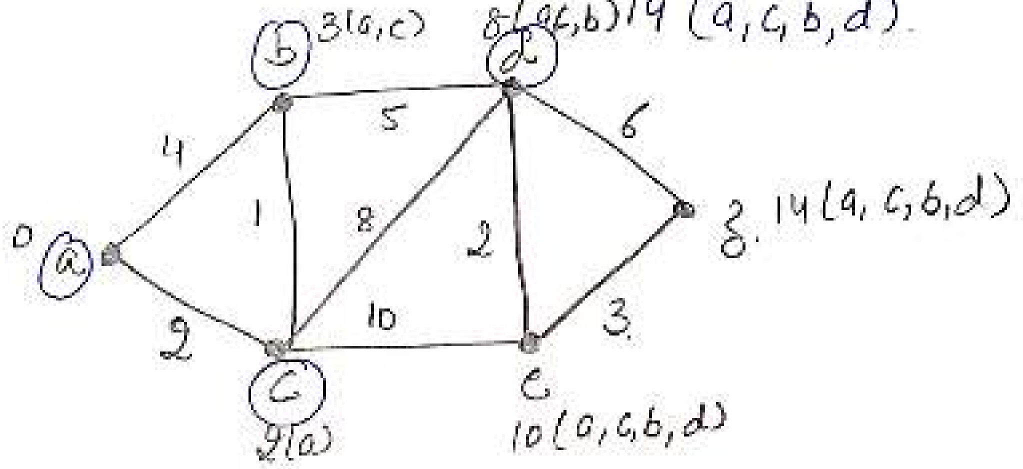
- Next we have d and e as adjacent to b and c.
- For d we will have min{8(a, c, b), 10(a, c)} (∵ d has both b & c adjacent to it).
  = 8(a, c, b) and for e we will have 12(a, c)



Scanned by CamScanner

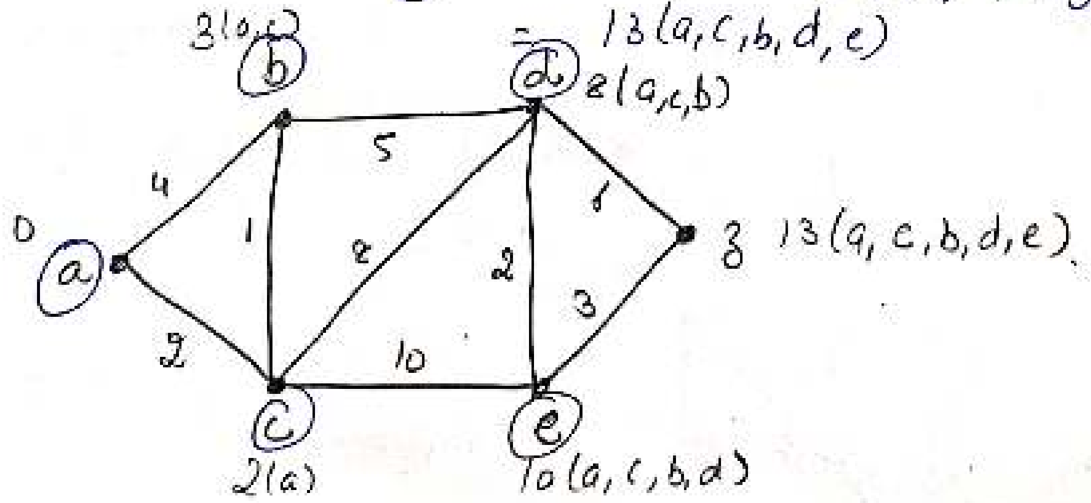
Circle 'd'

- we could have come to e via a-c-b-d-e whose ~~wei~~ weight is 10 (a,c,b,d) < 12(a,c) hence replace.
- Also to z via a-c-b-d-z is 14 (a,c,b,d).



Circle 'c'

for z we will now have min{14,(a,c,b,d), 13(a,c,b,d,e)} = 13(a,c,b,d,e)



No more options for z - are left and z - is our final point. Hence stop.

Final path is

a - c - b - d - e → z

and the weight is 13

-: Algorithm to find Spanning tree in an unweighted graph:-

→ There are two methods:-

→ Breadth first ~~method~~. Search

→ Depth first seach.

Scanned by CamScanner

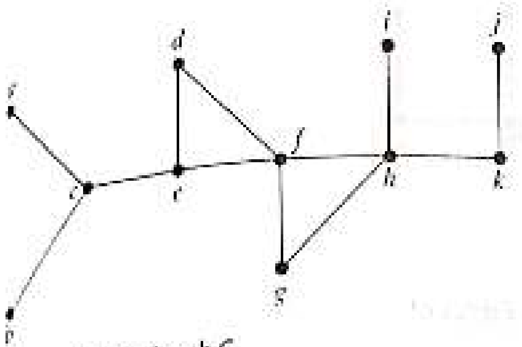


Figure 6 The Graph G

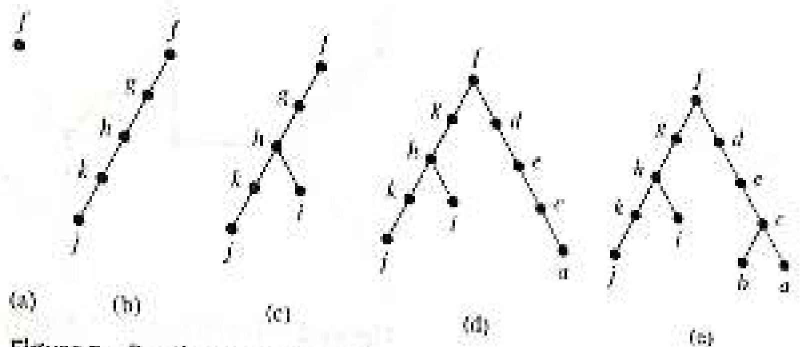


Figure 7 Depth-First Search of G

Depth-First Search The proof of Theorem 1 gives an algorithm for finding spanning trees by removing edges from simple circuits. This algorithm is inefficient, because it requires that simple circuits be identified. Instead of constructing spanning trees by removing edges, spanning trees can be built up by successively adding edges. Two algorithms based on this principle will be presented here.

Links

We can build a spanning tree for a connected simple graph using **depth-first search.** We will form a rooted tree, and the spanning tree will be the underlying undirected graph of this rooted tree. Arbitrarily choose a vertex of the graph as the root. Form a path starting at this vertex by successively adding vertices and edges, where each new edge is incident with the last vertex in the path and a vertex not already in the path. Continue adding vertices and edges to this path as long as possible. If the path goes through all vertices of the graph, the tree consisting of this path is a spanning tree. However, if the path does not go through all vertices, more vertices and edges must be added. Move back to the next to last vertex in the path, and, if possible, form a new path starting at this vertex passing through vertices that were not already visited. If this cannot be done, move back another vertex in the path, that is, two vertices back in the path, and try again.

Demo

Repeat this procedure, beginning at the last vertex visited, moving back up the path one vertex at a time, forming new paths that are as long as possible until no more edges can be added. Because the graph has a finite number of edges and is connected, this process ends with the production of a spanning tree. Each vertex that ends a path at a stage of the algorithm will be a leaf in the rooted tree, and each vertex where a path is constructed starting at this vertex will be an internal vertex.

The reader should note the recursive nature of this procedure. Also, note that if the vertices in the graph are ordered, the choices of edges at each stage of the procedure are all determined when we always choose the first vertex in the ordering that is available. However, we will not always explicitly order the vertices of a graph.

Depth-first search is also called **backtracking,** because the algorithm returns to vertices previously visited to add paths. Example 3 illustrates backtracking.

Example 3 *Use depth-first search to find a spanning tree for the graph G shown in Figure 6.*

Extra Examples

Solution: The steps used by depth-first search to produce a spanning tree of G are shown in Figure 7. We arbitrarily start with the vertex f. A path is built by successively adding edges incident with vertices not already in the path, as long as this is possible. This produces a path f, g, h, k, j (note that other paths could have been built). Next, backtrack to k. There is no path beginning at k containing vertices not already visited. So we backtrack to h. Form the path h, i. Then backtrack to h, and then to f. From f build the path f, d, e, c, a. Then backtrack to c and form the path c, b. This produces the spanning tree. ◀

The edges selected by depth-first search of a graph are called **tree edges.** All other edges of the graph must connect a vertex to an ancestor or descendant of this vertex in the tree. These edges are called **back edges.** (Exercise 39 asks for a proof of this fact.)

Scanned by CamScanner

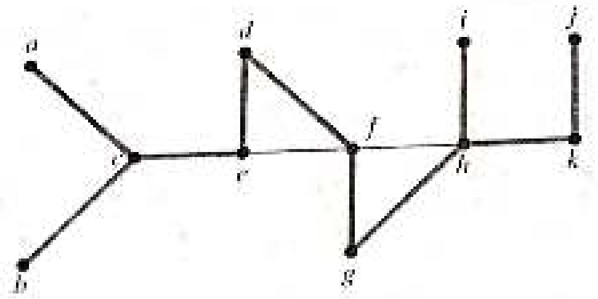


Figure 8 The Tree Edges and Back Edges of the Depth-First Search in Example 4.

Example 4 In Figure 8 we highlight the tree edges found by depth-first search starting at vertex f by showing them with heavy colored lines. The back edges (e, f) and (f, h) are shown with thinner black lines. ◀

We have explained how to find a spanning tree of a graph using depth-first search. However, our discussion so far has not brought out the recursive nature of depth-first search. To help make the recursive nature of the algorithm clear, we need a little terminology. We say that we *explore* from a vertex when we carry out the steps of depth-first search beginning when is added to the tree and ending when we have backtracked back to for the last time. The key observation needed to understand the recursive nature of the algorithm is that when we add an edge connecting a vertex to a vertex w, we finish exploring from w before we return to to complete exploring from .

In Algorithm 1 we construct the spanning tree of a graph G with vertices $_1, \ldots, _n$ by first selecting the vertex v_1 to be the root. We initially set T to be the tree with just this one vertex. At each step we add a new vertex to the tree T together with an edge from a vertex already in T to this new vertex and we explore from this new vertex. Note that at the completion of the algorithm, T contains no simple circuits because no edge is added that connects a vertex already in the tree. Moreover, T remains connected as it is built. (These last two observations can be easily proved via mathematical induction.) Because G is connected, every vertex in G is visited by the algorithm and is added to the tree (as the reader should verify). It follows that T is a spanning tree of G.

ALGORITHM 1 Depth-First Search.

```
procedure DFS(G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of the vertex $v_1$
visit($v_1$)
procedure visit(: vertex of G)
for each vertex $w$ adjacent to and not yet in T
begin
  add vertex $w$ and edge $\{v, w\}$ to T
  visit($w$)
end
```

Scanned by CamScanner

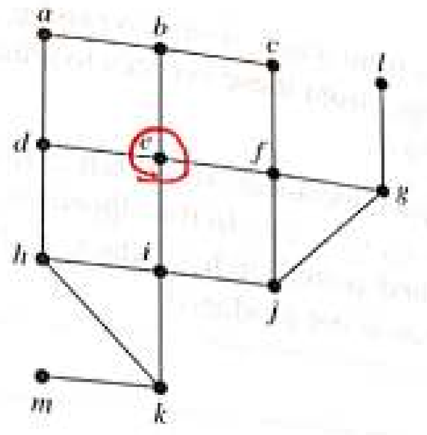


Figure 9 A Graph G.

tree as it is being built and adding this vertex and the corresponding edge if the vertex is not already in the tree. We have also made use of the inequality $e \leq n(n-1)/2$, which holds for any simple graph.]

Depth-first search can be used as the basis for algorithms that solve many different problems. For example, it can be used to find paths and circuits in a graph, it can be used to determine the connected components of a graph, and it can be used to find the cut vertices of a connected graph. As we will see, depth-first search is the basis of backtracking techniques used to search for solutions of computationally difficult problems. (See [GrYe99], [Ma89], and [CoLeRiSt01] for a discussion of algorithms based on depth-first search.)

Breadth-First Search We can also produce a spanning tree of a simple graph by the use of **breadth-first search.** Again, a rooted tree will be constructed, and the underlying undirected graph of this rooted tree forms the spanning tree. Arbitrarily choose a root from the vertices of the graph. Then add all edges incident to this vertex. The new vertices added at this stage become the vertices at level 1 in the spanning tree. Arbitrarily order them. Next, for each vertex at level 1, visited in order, add each edge incident to this vertex to the tree as long as it does not produce a simple circuit. Arbitrarily order the children of each vertex at level 1. This produces the vertices at level 2 in the tree. Follow the same procedure until all the vertices in the tree have been added. The procedure ends because there are only a finite number of edges in the graph. A spanning tree is produced because we have produced a tree containing every vertex of the graph. An example of breadth-first search is given in Example 5.

Demo

Links

Example 5 *Use breadth-first search to find a spanning tree for the graph shown in Figure 9.*

Extra Examples

Solution The steps of the breadth-first search procedure are shown in Figure 10. We choose the vertex e to be the root. Then we add edges incident with all vertices adjacent to e, so edges from e to b, d, f, and i are added. These four vertices are at level 1 in the tree. Next, add the edges from these vertices at level 1 to adjacent vertices not already in the tree. Hence, the edges

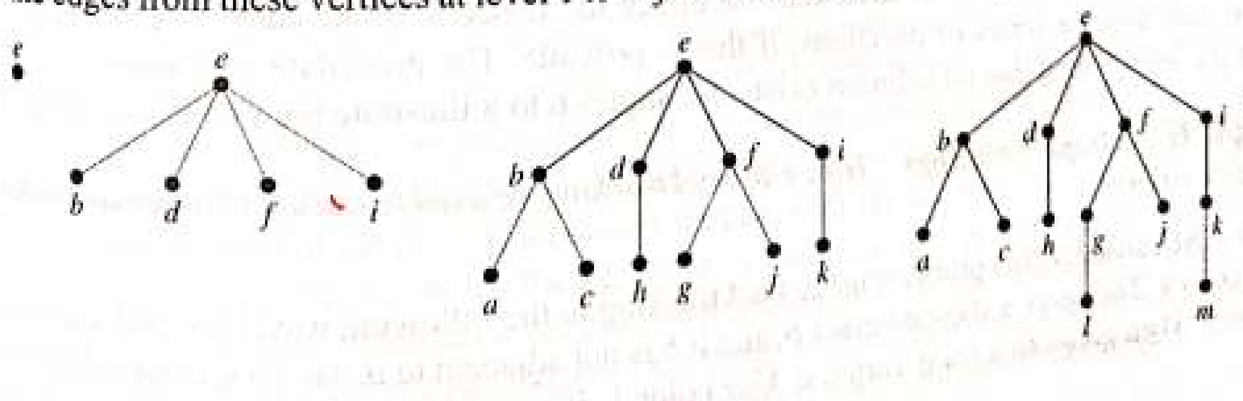


Figure 10 Breadth-First Search of G.

Scanned by CamScanner

from b to a and c are added, as are edges from d to h, from f to j and g, and from i to k. The new vertices a, c, h, j, g, and k are at level 2. Next, add edges from these vertices to adjacent vertices not already in the graph. This adds edges from g to l and from k to m.

We describe breadth-first search in pseudocode as Algorithm 2. In this algorithm, we assume the vertices of the connected graph G are ordered as $v_1, v_2, \ldots, v_n$. In the algorithm we use the term "process" to describe the procedure of adding new vertices, and corresponding edges, to the tree adjacent to the current vertex being processed as long as a simple circuit is not produced.

ALGORITHM 2 Breadth-First Search.

```
procedure BFS (G: connected graph with vertices $v_1, v_2, \ldots, v_n$)
T := tree consisting only of vertex $v_1$
L := empty list
put $v_1$ in the list L of unprocessed vertices
while L is not empty
begin
   remove the first vertex, v, from L
   for each neighbor w of v
      if w is not in L and not in T then
      begin
         add w to the end of the list L
         add w and edge {v, w} to T
      end
end
```

We now analyze the computational complexity of breadth-first search. For each vertex in the graph we examine all vertices adjacent to and we add each vertex not yet visited to the tree T. Assuming we have the adjacency lists for the graph available, no computation is required to determine which vertices are adjacent to a given vertex. As in the analysis of the depth-first search algorithm, we see that we examine each edge at most twice to determine whether we should add this edge and its endpoint not already in the tree. It follows that the breadth-first search algorithm uses $O(e)$ or $O(n^2)$ steps.

... an exhaustive

Scanned by CamScanner