```
//Graph represented in adj matrix.
int adjMatrix [256][256], table[256];
vector <int> st ;
int counter = 0 ;
//This table contains the DFS Search number
int dfsnum [256], num = 0, low[256] ;
void StronglyConnectedComponents( int u ) {
      low[u] = dfsnum[ u ] = num++;
      Push(st, u) ;
      for( int v = 0 ; v < 256; ++v ) {
            if(graph[u][v] && table[v] == -1) {
                  if( dfsnum[v] == -1)
                     StronglyConnectedComponents(v) ;
                  low[u] = min(low[u] , low[v]) ;
            }
      }
      if(low[u] == dfsnum[u]) {
            while( table[u] != counter) {
                  table[st.back()] = counter;
                  Push(st) ;
            }
            ++ counter;
      }
}
```

Problem-19 Count the number of connected components of Graph *G* which is represented in the adjacent matrix.

Solution: This problem can be solved with one extra counter in *DFS*.

```
//Visited[] is a global array.
int Visited[G→V];
void DFS(struct Graph *G, int u) {
    Visited[u] = 1;
    for( int v = 0; v < G→V; v++ ) {
        /* For example, if the adjacency matrix is used for representing the
              graph, then the condition to be used for finding unvisited adjacent
              vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */
        for each unvisited adjacent node v of u {
            DFS(G, v);
        }
    }
}
void DFSTraversal(struct Graph *G) {
      int count = 0;
      for (int i = 0; i< G→V;i++)
              Visited[i]=0;
      //This loop is required if the graph has more than one component
      for (int i = 0; i< G→V;i++)
              if(!Visited[i]) {
                      DFS(G, i);
                      count++;
              }
      return count;
}
```

Time Complexity: Same as that of DFS and it depends on implementation. With adjacency matrix the complexity is $O(|E| + |V|)$ and with adjacency matrix the complexity is $O(|V|^2)$.

Problem-20 Can we solve the Problem-19, using BFS?

Solution: Yes. This problem can be solved with one extra counter in BFS.

```
void BFS(struct Graph *G, int u) {
    int v,
    Queue Q = CreateQueue();
    EnQueue(Q, u);
    while(!IsEmptyQueue(Q)) {
            u = DeQueue(Q);
            Process u; //For example, print
            Visited[s]=1;
            /* For example, if the adjacency matrix is used for representing the
                  graph, then the condition be used for finding unvisited adjacent
                  vertex of u  is: if( !Visited[v] && G→Adj[u][v] ) */
           for each unvisited adjacent node v of u {
                EnQueue(Q, v);
           }
    }
}
void BFSTraversal(struct Graph *G) {
      for (int i = 0; i< G→V;i++)
              Visited[i]=0;
      //This loop is required if the graph has more than one component
      for (int i = 0; i< G→V; i++)
              if(!Visited[i])
                      BFS(G, i);
}
```

Time Complexity: Same as that of *BFS* and it depends on implementation. With adjacency matrix the complexity is $O(|E| + |V|)$ and with adjacency matrix the complexity is $O(|V|^2)$.

Problem-21 Let us assume that $G(V,E)$ is an undirected graph. Give an algorithm for finding a spanning tree which takes $O(|E|)$ time complexity (not necessarily a minimum spanning tree).

Solution: The test for a cycle can be done in constant time, by marking vertices that have been added to the set *S*. An edge will introduce a cycle, if both its vertices have already been marked.

Algorithm:

```
S = {}; //Assume S is a set
for each edge e ∈ E {
    if(adding e to S doesn't form a cycle) {
        add e to S;
        mark e;
    }
}
```

Problem-22 Is there any other way of solving 0?

Solution: Yes. We can run *BFS* and find the *BFS* tree for the graph (level order tree of the graph). Then start at the root element and keep moving to the next levels and at the same time we have to consider the nodes in the next level only once. That means, if we have a node with multiple input edges then we should consider only one of them; otherwise they will form a cycle.

Problem-23 Detecting a cycle in an undirected graph

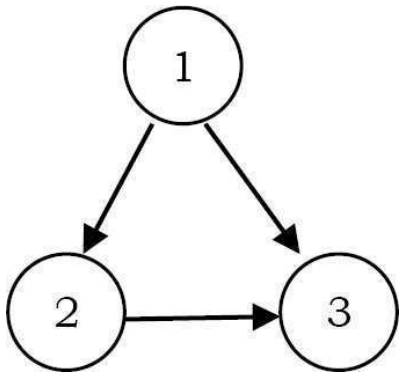
Solution: An undirected graph is acyclic if and only if a *DFS* yields no back edges, edges (u, v) where v has already been discovered and is an ancestor of u.

- Execute *DFS* on the graph.
- If there is a back edge – the graph has a cycle.

If the graph does not contain a cycle, then $|E| < |V|$ and *DFS* cost $O(|V|)$. If the graph contains a cycle, then a back edge is discovered after $2|V|$ steps at most.

Problem-24 Detecting a cycle in DAG

Solution:



Cycle detection on a graph is different than on a tree. This is because in a graph, a node can have multiple parents. In a tree, the algorithm for detecting a cycle is to do a depth first search, marking nodes as they are encountered. If a previously marked node is seen again, then a cycle exists. This won't work on a graph. Let us consider the graph shown in the figure below. If we use a tree cycle detection algorithm, then it will report the wrong result. That means that this graph has a cycle in it. But the given graph does not have a cycle in it. This is because node 3 will be seen twice in a *DFS* starting at node 1.

The cycle detection algorithm for trees can easily be modified to work for graphs. The key is that in a *DFS* of an acyclic graph, a node whose descendants have all been visited can be seen again without implying a cycle. But, if a node is seen for the second time before all its descendants have been visited, then there must be a cycle. Can you see why this is? Suppose there is a cycle containing node A. This means that A must be reachable from one of its descendants. So when the *DFS* is visiting that descendant, it will see *A* again, before it has finished visiting all of *A*'s descendants. So there is a cycle. In order to detect cycles, we can modify the depth first search.

```
int DetectCycle(struct Graph *G) {
    for (int i = 0; i< G→V; i++) {
        Visited[s]=0;
        Predecessor[i] = 0;
    }
    for (int i = 0; i < G→V;i++) {
        if(!Visited[i] && HasCycle(G, i))
            return 1;
    }
    return false;;
}
int HasCycle(struct Graph *G, int u) {
    Visited[u]=1;
    for (int i = 0; i< G→V; i++) {
      if(G→Adj[s][i]) {
        if(Predecessor[i] != u && Visited[i])
            return 1;
         else {
            Predecessor[i] = u;
            return  HasCycle(G, i);
        }
      }
   }
   return 0;
}
```

Time Complexity: O($V + E$).

Problem-25 Given a directed acyclic graph, give an algorithm for finding its depth.

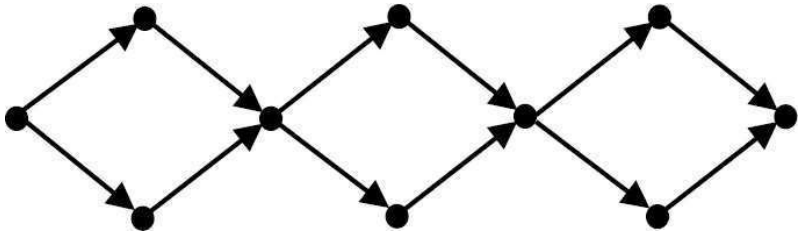
Solution: If it is an undirected graph, we can use the simple unweighted shortest path algorithm (check *Shortest Path Algorithms* section). We just need to return the highest number among all distances. For directed acyclic graph, we can solve by following the similar approach which we used for finding the depth in trees. In trees, we have solved this problem using level order

traversal (with one extra special symbol to indicate the end of the level).

```
//Assuming the given  graph is a DAG
int DepthInDAG( struct Graph *G ) {
        struct Queue *Q;
        int counter;
        int v, w;
        Q = CreateQueue();

        counter =  0;
        for (v = 0; v< G→V; v++)
                if( indegree[v] ==  0 )
                        EnQueue( Q , v );
        EnQueue( Q, '$' );

        while( !IsEmptyQueue( Q )) {
                v = DeQueue( Q );
                if(v == '$') {
                        counter++;
                        if(!IsEmptyQueue( Q ))
                                EnQueue( Q , '$' );

                }
                for each w adjacent to v
                        if( --indegree[w] ==  0 )
                                EnQueue ( Q , w );
        }
        DeleteQueue( Q );
        return counter;
}
```

Total running time is $O(V + E)$.

Problem-26 How many topological sorts of the following dag are there?



Solution: If we observe the above graph there are three stages with 2 vertices. In the early

discussion of this chapter, we saw that topological sort picks the elements with zero indegree at any point of time. At each of the two vertices stages, we can first process either the top vertex or the bottom vertex. As a result, at each of these stages we have two possibilities. So the total number of possibilities is the multiplication of possibilities at each stage and that is, $2 \times 2 \times 2 = 8$.

Problem-27 **Unique topological ordering:** Design an algorithm to determine whether a directed graph has a unique topological ordering.

Solution: A directed graph has a unique topological ordering if and only if there is a directed edge between each pair of consecutive vertices in the topological order. This can also be defined as: a directed graph has a unique topological ordering if and only if it has a Hamiltonian path. If the digraph has multiple topological orderings, then a second topological order can be obtained by swapping a pair of consecutive vertices.

Problem-28 Let us consider the prerequisites for courses at *IIT Bombay*. Suppose that all prerequisites are mandatory, every course is offered every semester, and there is no limit to the number of courses we can take in one semester. We would like to know the minimum number of semesters required to complete the major. Describe the data structure we would use to represent this problem, and outline a linear time algorithm for solving it.

Solution: Use a directed acyclic graph (DAG). The vertices represent courses and the edges represent the prerequisite relation between courses at *IIT Bombay*. It is a DAG, because the prerequisite relation has no cycles.

The number of semesters required to complete the major is one more than the longest path in the dag. This can be calculated on the DFS tree recursively in linear time. The longest path out of a vertex x is 0 if x has outdegree 0, otherwise it is $1 + max$ {*longest path out of* y | (x,y) is *an edge of* G}.

Problem-29 At a university let's say *IIT Bombay*), there is a list of courses along with their prerequisites. That means, two lists are given:
A – Courses list
B – Prerequisites: B contains couples (x,y) where $x,y \in A$ indicating that course x can't be taken before course y.

Let us consider a student who wants to take only one course in a semester. Design a schedule for this student.

Example: A = {C-Lang, Data Structures, OS, CO, Algorithms, Design Patterns, Programming}. B = { (C-Lang, CO), (OS, CO), (Data Structures, Algorithms), (Design Patterns, Programming) }. *One possible schedule could be*:

Semester 1:	Data Structures
Semester 2:	Algorithms
Semester 3:	C-Lang

Semester 4: OS
Semester 5: CO
Semester 6: Design Patterns
Semester 7: Programming

Solution: The solution to this problem is exactly the same as that of topological sort. Assume that the courses names are integers in the range [1..*n*], *n* is known (*n* is not constant). The relations between the courses will be represented by a directed graph $G = (V,E)$, where V are the set of courses and if course i is prerequisite of course j, E will contain the edge (i,j). Let us assume that the graph will be represented as an Adjacency list.

First, let's observe another algorithm to topologically sort a DAG in $O(|V| + |E|)$.

- Find in-degree of all the vertices - $O(|V| + |E|)$
- Repeat:

  Find a vertex v with in-degree=0 - $O(|V|)$

  Output v and remove it from G, along with its edges - $O(|V|)$

  Reduce the in-degree of each node u such as (v, u) was an edge in G and keep a list of vertices with in-degree=0 – $O(degree(v))$

  Repeat the process until all the vertices are removed

The time complexity of this algorithm is also the same as that of the topological sort and it is $O(|V| + |E|)$.

Problem-30 In Problem-29, a student wants to take all the courses in *A*, in the minimal number of semesters. That means the student is ready to take any number of courses in a semester. Design a schedule for this scenario. *One possible schedule is:*
Semester 1: C-Lang, OS, Design Patterns
Semester 2: Data Structures, CO, Programming
Semester 3: Algorithms

Solution: A variation of the above topological sort algorithm with a slight change: In each semester, instead of taking one subject, take all the subjects with zero indegree. That means, execute the algorithm on all the nodes with degree 0 (instead of dealing with one source in each stage, all the sources will be dealt and printed).

Time Complexity: $O(|V| + |E|)$.

Problem-31 **LCA of a DAG:** Given a DAG and two vertices *v* and *w*, find the *lowest common ancestor* (LCA) of *v* and *w*. The LCA of *v* and w is an ancestor of *v* and *w* that has no descendants that are also ancestors of *v* and *w*.

Hint: Define the height of a vertex *v* in a DAG to be the length of the longest path from *root* to *v*. Among the vertices that are ancestors of both *v* and *w*, the one with the greatest height is an LCA

of *v* and *w*.

Problem-32 **Shortest ancestral path:** Given a DAG and two vertices *v* and *w*, find the *shortest ancestral path* between *v* and *w*. An ancestral path between *v* and *w* is a common ancestor *x* along with a shortest path from *v* to *x* and a shortest path from *w* to *x*. The shortest ancestral path is the ancestral path whose total length is minimized.

Hint: Run BFS two times. First run from *v* and second time from *w*. Find a DAG where the shortest ancestral path goes to a common ancestor *x* that is not an LCA.

Problem-33 Let us assume that we have two graphs G_1 and G_2. How do we check whether they are isomorphic or not?

Solution: There are many ways of representing the same graph. As an example, consider the following simple graph. It can be seen that all the representations below have the same number of vertices and the same number of edges.

Definition: Graphs $G_1 = \{V_1, E_1\}$ and $G_2 = \{V_2, E_2\}$ are isomorphic if

1) There is a one-to-one correspondence from V_1 to V_2 and
2) There is a one-to-one correspondence from E_1 to E_2 that map each edge of G_1 to G_2.

Now, for the given graphs how do we check whether they are isomorphic or not?

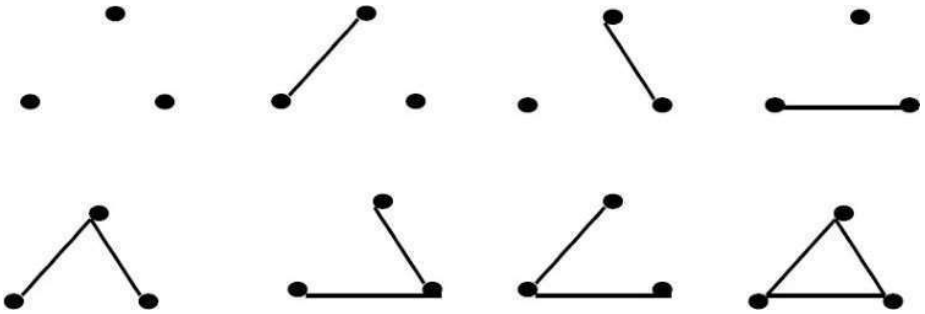
In general, it is not a simple task to prove that two graphs are isomorphic. For that reason we must consider some properties of isomorphic graphs. That means those properties must be satisfied if the graphs are isomorphic. If the given graph does not satisfy these properties then we say they are not isomorphic graphs.

Property: Two graphs are isomorphic if and only if for some ordering of their vertices their adjacency matrices are equal.

Based on the above property we decide whether the given graphs are isomorphic or not. I order to check the property, we need to do some matrix transformation operations.

Problem-34 How many simple undirected non-isomorphic graphs are there with *n* vertices?

Solution: We will try to answer this question in two steps. First, we count all labeled graphs. Assume all the representations below are labeled with {1,2,3} as vertices. The set of all such graphs for $n = 3$ are:



There are only two choices for each edge: it either exists or it does not. Therefore, since the maximum number of edges is $\binom{n}{2}$ (and since the maximum number of edges in an undirected graph with *n* vertices is $\frac{n(n-1)}{2} = n_{C_2} = \binom{n}{2}$), the total number of undirected labeled graphs is $2^{\binom{n}{2}}$.

Problem-35 **Hamiltonian path in DAGs:** Given a DAG, design a linear time algorithm to determine whether there is a path that visits each vertex exactly once.

Solution: The *Hamiltonian* path problem is an NP-Complete problem (for more details ref *Complexity Classes* chapter). To solve this problem, we will try to give the approximation algorithm (which solves the problem, but it may not always produce the optimal solution).

Let us consider the topological sort algorithm for solving this problem. Topological sort has an interesting property: that if all pairs of consecutive vertices in the sorted order are connected by edges, then these edges form a directed *Hamiltonian* path in the DAG. If a *Hamiltonian* path exists, the topological sort order is unique. Also, if a topological sort does not form a *Hamiltonian* path, the DAG will have two or more topological orderings.

Approximation Algorithm: Compute a topological sort and check if there is an edge between each consecutive pair of vertices in the topological order.

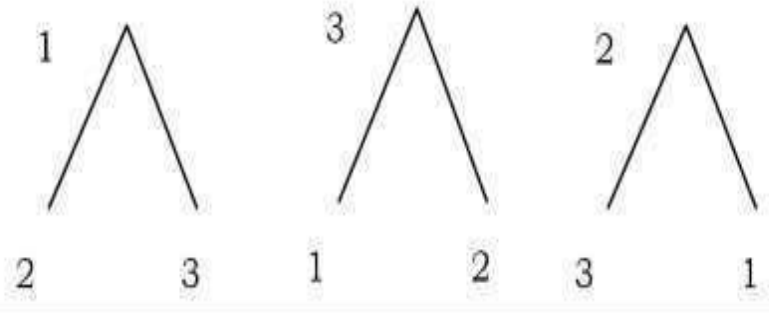
In an unweighted graph, find a path from **s** to **t** that visits each vertex exactly once. The basic solution based on backtracking is, we start at s and try all of its neighbors recursively, making sure we never visit the same vertex twice. The algorithm based on this implementation can be given as:

```
bool seenTable[32];
void HamiltonianPath( struct Graph *G, int u ) {
    if( u == t )
        /* Check that we have seen all vertices. */
    else {
        for( int v = 0; v < n; v++ )
            if( !seenTable[v] && G→Adj[u][v] ) {
                seenTable[v] = true;
                HamiltonianPath( v );
                seenTable[v] = false;
            }
    }
}
```

Note that if we have a partial path from s to u using vertices s = v_1, v_2,..., v_k = u, then we don't care about the order in which we visited these vertices so as to figure out which vertex to visit next. All that we need to know is the set of vertices we have seen (the seenTable[] array) and which vertex we are at right now (u). There are 2^n possible sets of vertices and n choices for u. In other words, there are 2^n possible *seenTable*[] arrays and n different parameters to Hamiltonian_path(). What Hamiltonian_path() does during any particular recursive call is completely determined by the *seenTable*[ ] array and the parameter u.

Problem-36 For a given graph *G* with *n* vertices how many trees we can construct?

Solution: There is a simple formula for this problem and it is named after Arthur Cayley. For a given graph with *n* labeled vertices the formula for finding number of trees on is n^{n-2}. Below, the number of trees with different *n* values is shown.

n value	Formula value: n^{n-2}	Number of Trees
2	1	
3	3	

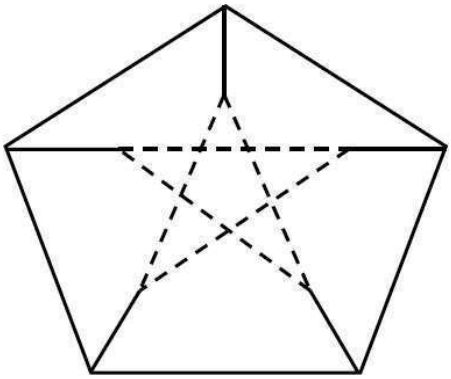
Problem-37 For a given graph G with *n* vertices how many spanning trees can we construct?

Solution: The solution to this problem is the same as that of Problem-36. It is just another way of asking the same question. Because the number of edges in both regular tree and spanning tree are the same.

Problem-38 The *Hamiltonian cycle* problem: Is it possible to traverse each of the vertices of a graph exactly once, starting and ending at the same vertex?

Solution: Since the *Hamiltonian* path problem is an NP-Complete problem, the *Hamiltonian* cycle problem is an NP-Complete problem. A *Hamiltonian* cycle is a cycle that traverses every vertex of a graph exactly once. There are no known conditions in which are both necessary and sufficient, but there are a few sufficient conditions.

- For a graph to have a *Hamiltonian* cycle the degree of each vertex must be two or more.
- The Petersen graph does not have a *Hamiltonian* cycle and the graph is given below.



- In general, the more edges a graph has, the more likely it is to have a *Hamiltonian* cycle.
- Let G be a simple graph with $n \geq 3$ vertices. If every vertex has a degree of at least $\frac{n}{2}$, then G has a *Hamiltonian* cycle.
- The best known algorithm for finding a *Hamiltonian* cycle has an exponential worst-case complexity.

Note: For the approximation algorithm of *Hamiltonian* path, refer to the *Dynamic Programming* chapter.

Problem-39 What is the difference between *Dijkstra's* and *Prim's* algorithm?

Solution: *Dijkstra's* algorithm is almost identical to that of *Prim's*. The algorithm begins at a specific vertex and extends outward within the graph until all vertices have been reached. The only distinction is that *Prim's* algorithm stores a minimum cost edge whereas *Dijkstra's* algorithm stores the total cost from a source vertex to the current vertex. More simply, *Dijkstra's* algorithm stores a summation of minimum cost edges whereas *Prim's* algorithm stores at most one minimum cost edge.

Problem-40 **Reversing Graph:** : Give an algorithm that returns the reverse of the directed graph (each edge from *v* to *w* is replaced by an edge from *w to v*).

Solution: In graph theory, the reverse (also called *transpose*) of a directed graph G is another directed graph on the same set of vertices with all the edges reversed. That means, if G contains an edge (u, v) then the reverse of G contains an edge (v, u) and vice versa.

Algorithm:

```
Graph ReverseTheDirectedGraph(struct Graph *G) {
        Create new graph with name ReversedGraph and
                let us assume that this will contain the reversed graph.
        //The reversed graph also will contain same number of vertices and edges.
        for each vertex of given graph G {
                for each vertex w adjacent to v {
                        Add the w to v edge in ReversedGraph;
                        //That means we just need to reverse the bits in adjacency matrix.
                }
        }
        return ReversedGraph;
}
```

Problem-41 **Travelling Sales Person Problem:** Find the shortest path in a graph that visits each vertex at least once, starting and ending at the same vertex?

Solution: The Traveling Salesman Problem (*TSP*) is related to finding a Hamiltonian cycle. Given a weighted graph G, we want to find the shortest cycle (may be non-simple) that visits all the vertices.

Approximation algorithm: This algorithm does not solve the problem but gives a solution which is within a factor of 2 of optimal (in the worst-case).

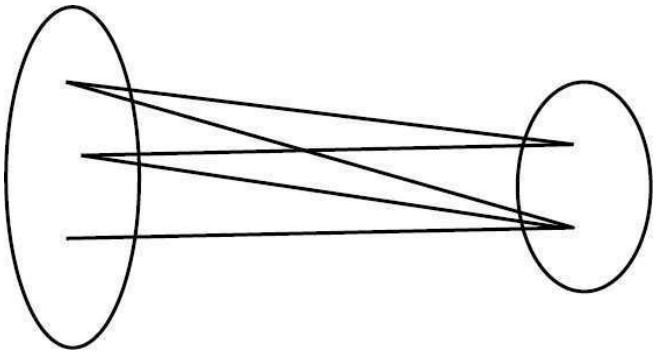
1) Find a Minimal Spanning Tree (MST).
2) Do a DFS of the MST.

For details, refer to the chapter on *Complexity Classes*.

Problem-42 Discuss Bipartite matchings?

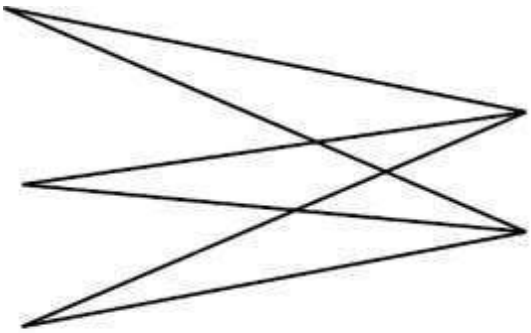
Solution: In Bipartite graphs, we divide the graphs in to two disjoint sets, and each edge connects a vertex from one set to a vertex in another subset (as shown in figure).

Definition: A simple graph $G = (V, E)$ is called a *bipartite graph* if its vertices can be divided into two disjoint sets $V = V_1 \cup V_2$, such that every edge has the form $e = (a,b)$ where $a \in V_1$ and $b \in V_2$. One important condition is that no vertices both in V_1 or both in V_2 are connected.

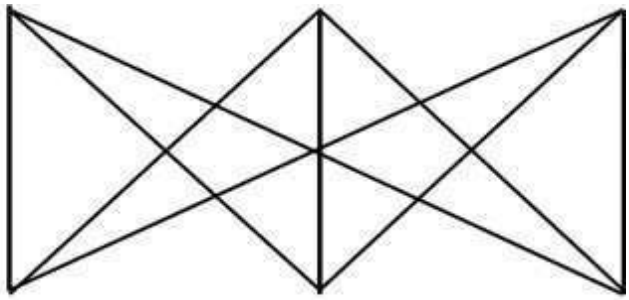


Properties of Bipartite Graphs

- A graph is called bipartite if and only if the given graph does not have an odd length cycle.
- A *complete bipartite graph* $K_{m,n}$ is a bipartite graph that has each vertex from one set adjacent to each vertex from another set.

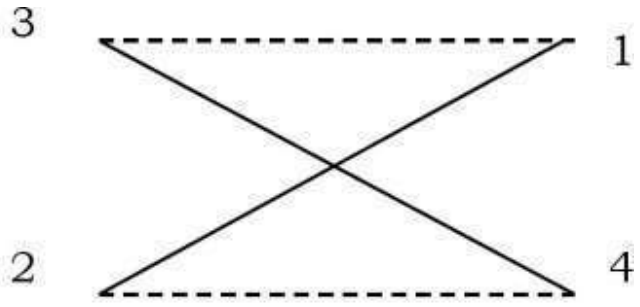


$K_{2,3}$



$K_{3,3}$

- A subset of edges $M \subset E$ is a *matching* if no two edges have a common vertex. As an example, matching sets of edges are represented with dotted lines. A matching M is called *maximum* if it has the largest number of possible edges. In the graphs, the dotted edges represent the alternative matching for the given graph.



- A matching M is *perfect* if it matches all vertices. We must have $V_1 = V_2$ in order to have perfect matching.
- An *alternating path* is a path whose edges alternate between matched and unmatched edges. If we find an alternating path, then we can improve the matching. This is because an alternating path consists of matched and unmatched edges. The number of unmatched edges exceeds the number of matched edges by one.

Therefore, an alternating path always increases the matching by one.

The next question is, how do we find a perfect matching? Based on the above theory and definition, we can find the perfect matching with the following approximation algorithm.

Matching Algorithm (Hungarian algorithm)

1) Start at unmatched vertex.
2) Find an alternating path.
3) If it exists, change matching edges to no matching edges and conversely. If it does not exist, choose another unmatched vertex.
4) If the number of edges equals $V/2$, stop. Otherwise proceed to step 1 and repeat, as long as all vertices have been examined without finding any alternating paths.

Time Complexity of the Matching Algorithm: The number of iterations is in O(V). The complexity of finding an alternating path using BFS is O(E). Therefore, the total time complexity is O($V \times E$).

Problem-43 Marriage and Personnel Problem?

Marriage Problem: There are X men and Y women who desire to get married. Participants indicate who among the opposite sex could be a potential spouse for them. Every woman can be married to at most one man, and every man to at most one woman. How can we marry everybody to someone they like?

Personnel Problem: You are the boss of a company. The company has M workers and N jobs. Each worker is qualified to do some jobs, but not others. How will you assign jobs to each worker?

Solution: These two cases are just another way of asking about bipartite graphs, and the solution is the same as that of Problem-42.

Problem-44 How many edges will be there in complete bipartite graph $K_{m,n}$?

Solution: $m \times n$. This is because each vertex in the first set can connect all vertices in the second set.

Problem-45 A graph is called a regular graph if it has no loops and multiple edges where each vertex has the same number of neighbors; i.e., every vertex has the same degree. Now, if $K_{m,n}$ is a regular graph, what is the relation between m and n?

Solution: Since each vertex should have the same degree, the relation should be $m = n$.

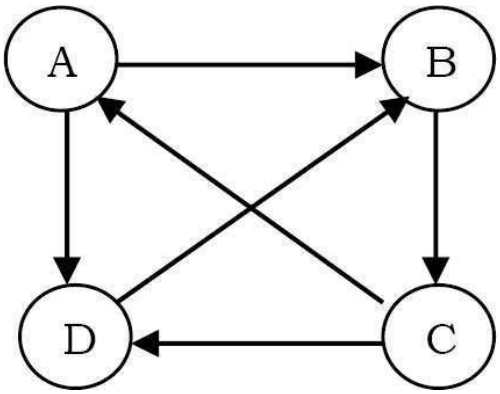
Problem-46 What is the maximum number of edges in the maximum matching of a bipartite graph with n vertices?

Solution: From the definition of *matching*, we should not have edges with common vertices. So

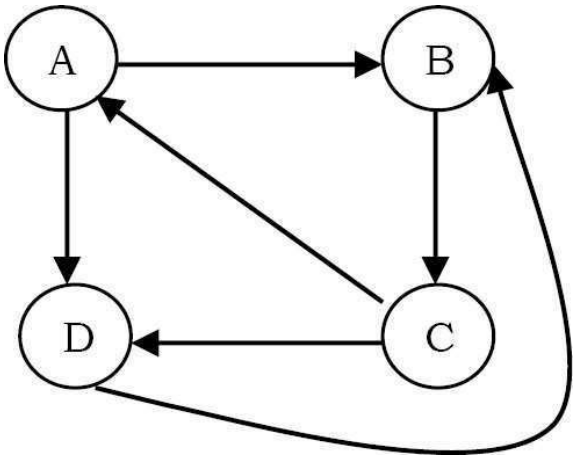
in a bipartite graph, each vertex can connect to only one vertex. Since we divide the total vertices into two sets, we can get the maximum number of edges if we divide them in half. Finally the answer is $\frac{n}{2}$.

Problem-47 Discuss Planar Graphs. *Planar graph:* Is it possible to draw the edges of a graph in such a way that the edges do not cross?

Solution: A graph G is said to be planar if it can be drawn in the plane in such a way that no two edges meet each other except at a vertex to which they are incident. Any such drawing is called a plane drawing of G. As an example consider the below graph:



This graph we can easily convert to a planar graph as below (without any crossed edges).



How do we decide whether a given graph is planar or not?

The solution to this problem is not simple, but researchers have found some interesting properties that we can use to decide whether the given graph is a planar graph or not.

Properties of Planar Graphs

- If a graph *G* is a connected planar simple graph with *V* vertices, where *V* = 3 and *E* edges, then *E* = 3*V* – 6.
- K_5 is non-planar. [K_5 stands for complete graph with 5 vertices].
- If a graph *G* is a connected planar simple graph with *V* vertices and *E* edges, and no

triangles, then $E = 2V - 4$.

- $\mathbf{K_{3,3}}$ is non-planar. [$K_{3,3}$ stands for bipartite graph with 3 vertices on one side and the other 3 vertices on the other side. $K_{3,3}$ contains 6 vertices].
- If a graph G is a connected planar simple graph, then G contains at least one vertex of 5 degrees or less.
- A graph is planar if and only if it does not contain a subgraph that has K_5 and $K_{3,3}$ as a contraction.
- If a graph G contains a nonplanar graph as a subgraph, then G is non-planar.
- If a graph G is a planar graph, then every subgraph of G is planar.
- For any connected planar graph $G = (V,E)$, the following formula should hold: $V + F - E = 2$, where F stands for the number of faces.
- For any planar graph $G = (V, E)$ with K components, the following formula holds: $V + F - E = 1 + K$.

In order to test the planarity of a given graph, we use these properties and decide whether it is a planar graph or not. Note that all the above properties are only the necessary conditions but not sufficient.

Problem-48 How many faces does $K_{2,3}$ have?

Solution: From the above discussion, we know that $V + F - E = 2$, and from an earlier problem we know that $E = m \times n = 2 \times 3 = 6$ and $V = m + n = 5$. $\therefore 5 + F - 6 = 2 \Rightarrow F = 3$.

Problem-49 Discuss Graph Coloring

Solution: A k –coloring of a graph G is an assignment of one color to each vertex of G such that no more than k colors are used and no two adjacent vertices receive the same color. A graph is called k –colorable if and only if it has a k –coloring.

Applications of Graph Coloring: The graph coloring problem has many applications such as scheduling, register allocation in compilers, frequency assignment in mobile radios, etc.

Clique: A *clique* in a graph G is the maximum complete subgraph and is denoted by $\omega(G)$.

Chromatic number: The chromatic number of a graph G is the smallest number k such that G is k –colorable, and it is denoted by $X(G)$.

The lower bound for $X(G)$ is $\omega(G)$, and that means $\omega(G) \leq X(G)$.

Properties of Chromatic number: Let G be a graph with n vertices and G′ is its complement. Then,

- $X(G) \leq \Delta(G) + 1$, where Δ (G) is the maximum degree of G.
- $X(G)\ \omega(G') \geq n$
- $X(G) + \omega(G') \leq n + 1$

- $X(G) + (G') \leq n + 1$

K-colorability problem: Given a graph G = (*V,E*) and a positive integer $k \leq V$. Check whether G is *k* –colorable?

This problem is NP-complete and will be discussed in detail in the chapter on *Complexity Classes*.

Graph coloring algorithm: As discussed earlier, this problem is *NP*-Complete. So we do not have a polynomial time algorithm to determine $X(G)$. Let us consider the following approximation (no efficient) algorithm.

- Consider a graph G with two non-adjacent vertices *a* and *b*. The connection G_1 is obtained by joining the two non-adjacent vertices *a* and *b* with an edge. The contraction G_2 is obtained by shrinking {*a*,*b*} into a single vertex *c*(*a, b*) and by joining it to each neighbor in G of vertex *a* and of vertex *b* (and eliminating multiple edges).
- A coloring of G in which *a* and *b* have the same color yields a coloring of G_1. A coloring of *G* in which *a* and *b* have different colors yields a coloring of G_2.
- Repeat the operations of connection and contraction in each graph generated, until the resulting graphs are all cliques. If the smallest resulting clique is a *K* –clique, then (*G*) = K.

Important notes on Graph Coloring

- Any simple planar graph G can be colored with 6 colors.
- Every simple planar graph can be colored with less than or equal to 5 colors.

Problem-50 What is the four coloring problem?

Solution: A graph can be constructed from any map. The regions of the map are represented by the vertices of the graph, and two vertices are joined by an edge if the regions corresponding to the vertices are adjacent. The resulting graph is planar. That means it can be drawn in the plane without any edges crossing.

The *Four Color Problem* is whether the vertices of a planar graph can be colored with at most four colors so that no two adjacent vertices use the same color.

History: The *Four Color Problem* was first given by *Francis Guthrie*. He was a student at *University College London* where he studied under *Augusts De Morgan*. After graduating from London he studied law, but some years later his brother Frederick Guthrie had become a student of *De Morgan*. One day Francis asked his brother to discuss this problem with *De Morgan*.

Problem-51 When an adjacency-matrix representation is used, most graph algorithms require time $O(V^2)$. Show that determining whether a directed graph, represented in an adjacency-

matrix that contains a sink can be done in time O(V). A sink is a vertex with in-degree $|V|$ – 1 and out-degree 0 (Only one can exist in a graph).

Solution: A vertex i is a sink if and only if $M[i,j] = 0$ for all j and $M[j, i] = 1$ for all $j \neq i$. For any pair of vertices i and j:

$$M[i,j] = 1 \rightarrow \text{vertex i can't be a sink}$$
$$M[i,j] = 0 \rightarrow \text{vertex j can't be a sink}$$

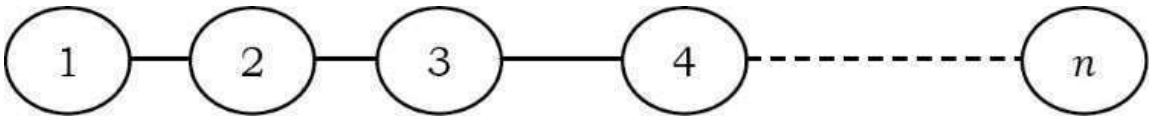
Algorithm:

- Start at $i = 1, j = 1$
- If $M[i,j] = 0 \rightarrow i$ wins, $j++$
- If $M[i,j] = 1 \rightarrow j$ wins, $i++$
- Proceed with this process until $j = n$ or $i = n + 1$
- If $i == n + 1$, the graph does not contain a sink
- Otherwise, check row i – it should be all zeros; and check column i – it should be all but $M[i, i]$ ones; – if so, t is a sink.

Time Complexity: O(V), because at most $2|V|$ cells in the matrix are examined.

Problem-52 What is the worst – case memory usage of DFS?

Solution: It occurs when the O($|V|$), which happens if the graph is actually a list. So the algorithm is memory efficient on graphs with small diameter.



Problem-53 Does DFS find the shortest path from start node to some node w ?

Solution: No. In DFS it is not compulsory to select the smallest weight edge.

Problem-54 True or False: Dijkstra's algorithm does not compute the "all pairs" shortest paths in a directed graph with positive edge weights because, running the algorithm a single time, starting from some single vertex x, it will compute only the min distance from x to y for all nodes y in the graph.

Solution: True.

Problem-55 True or False: Prim's and Kruskal's algorithms may compute different minimum spanning trees when run on the same graph.

Solution: True.