Graph Data Structure And Algorithms

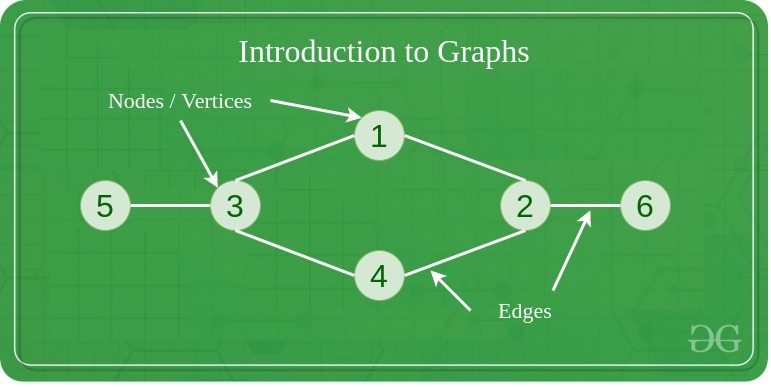
[**Learn more about Graph in DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?utm_source=geeksforgeeks&utm_medium=articles+graph_lp+header_link_click&utm_campaign=dsa+course+tracker)

[**Practice Problems on Graphs**](https://practice.geeksforgeeks.org/explore/?category%5B%5D=Graph&page=1&category%5B%5D=Graph&utm_source=geeksforgeeks&utm_medium=articles+graph_lp+header_link_click&utm_campaign=practice+tracker)

[**Recent Articles on Graph**](https://www.geeksforgeeks.org/category/graph/?ref=graph_lp)

[**What is Graph Data Structure?**](https://www.geeksforgeeks.org/introduction-to-graphs-data-structure-and-algorithm-tutorials/)

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices( V ) and a set of edges( E ). The graph is denoted by G(E, V).



**Components of a Graph**

* **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
* **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

**Topics**:

* [Introduction](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#introduction)
* [BFS & DFS in Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#bfsndfs)
* [Cycles in Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#cycle)
* [Shortest Paths in Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#shortest)
* [Minimum Spanning Tree](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#MST)
* [Topological Sorting](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#topo)
* [Connectivity](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#connectivity)
* [Maximum Flow](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#maxflow)
* [Some must do problems on Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#mustdo)
* [Some Quizzes](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#quiz)

**Introduction:**

1. [Introduction to Graphs](https://www.geeksforgeeks.org/introduction-to-graphs/)
2. [Graph and its representations](https://www.geeksforgeeks.org/graph-and-its-representations/)
3. [Types of Graphs with Examples](https://www.geeksforgeeks.org/graph-types-and-applications/)
4. [Basic Properties of a Graph](https://www.geeksforgeeks.org/basic-properties-of-a-graph/)
5. [Applications, Advantages and Disadvantages of Graph](https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-graph/)
6. [Transpose graph](https://www.geeksforgeeks.org/transpose-graph/)
7. [Difference between graph and tree](https://www.geeksforgeeks.org/difference-between-graph-and-tree/)

**BFS and DFS in Graph:**

1. [Breadth First Traversal for a Graph](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/)
2. [Depth First Traversal for a Graph](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)
3. [Applications of Depth First Search](https://www.geeksforgeeks.org/applications-of-depth-first-search/)
4. [Applications of Breadth First Traversal](https://www.geeksforgeeks.org/applications-of-breadth-first-traversal/)
5. [Iterative Depth First Search](https://www.geeksforgeeks.org/iterative-depth-first-traversal/)
6. [BFS for Disconnected Graph](https://www.geeksforgeeks.org/bfs-disconnected-graph/)
7. [Transitive Closure of a Graph using DFS](https://www.geeksforgeeks.org/transitive-closure-of-a-graph-using-dfs/)
8. [Difference between BFS and DFS](https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/)

**Cycles in Graph:**

1. [Detect Cycle in a Directed Graph](https://www.geeksforgeeks.org/detect-cycle-in-a-graph/)
2. [Detect cycle in an undirected graph](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/)
3. [Detect cycle in a direct graph using colors](https://www.geeksforgeeks.org/detect-cycle-direct-graph-using-colors/)
4. [Detect a negative cycle in a Graph | (Bellman Ford)](https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/)
5. [Cycles of length n in an undirected and connected graph](https://www.geeksforgeeks.org/cycles-of-length-n-in-an-undirected-and-connected-graph/)
6. [Detecting negative cycle using Floyd Warshall](https://www.geeksforgeeks.org/detecting-negative-cycle-using-floyd-warshall/)
7. [Clone a Directed Acyclic Graph](https://www.geeksforgeeks.org/clone-directed-acyclic-graph/)
8. [Union By Rank and Path Compression in Union-Find Algorithm](https://www.geeksforgeeks.org/union-by-rank-and-path-compression-in-union-find-algorithm/)
9. [Introduction to Disjoint Set Data Structure or Union-Find Algorithm](https://www.geeksforgeeks.org/introduction-to-disjoint-set-data-structure-or-union-find-algorithm/)

Advertisement

**Shortest Path in Graph:**

1. [Dijkstra’s shortest path algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/)
2. [Bellman–Ford Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/)
3. [Floyd Warshall Algorithm](https://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/)
4. [Johnson’s algorithm for All-pairs shortest paths](https://www.geeksforgeeks.org/johnsons-algorithm/)
5. [Shortest Path in Directed Acyclic Graph](https://www.geeksforgeeks.org/shortest-path-for-directed-acyclic-graphs/)
6. [Dial’s Algorithm](https://www.geeksforgeeks.org/dials-algorithm-optimized-dijkstra-for-small-range-weights/)
7. [Multistage Graph (Shortest Path)](https://www.geeksforgeeks.org/multistage-graph-shortest-path/)
8. [Shortest path in an unweighted graph](https://www.geeksforgeeks.org/shortest-path-unweighted-graph/)
9. [Karp’s minimum mean (or average) weight cycle algorithm](https://www.geeksforgeeks.org/karps-minimum-mean-average-weight-cycle-algorithm/)
10. [0-1 BFS (Shortest Path in a Binary Weight Graph)](https://www.geeksforgeeks.org/0-1-bfs-shortest-path-binary-graph/)
11. [Find minimum weight cycle in an undirected graph](https://www.geeksforgeeks.org/find-minimum-weight-cycle-undirected-graph/)

**Minimum Spanning Tree:**

1. [Prim’s Minimum Spanning Tree (MST))](https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/)
2. [Kruskal’s Minimum Spanning Tree Algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/)
3. [Difference between Prim’s and Kruskal’s algorithm for MST](https://www.geeksforgeeks.org/difference-between-prims-and-kruskals-algorithm-for-mst/)
4. [Applications of Minimum Spanning Tree Problem](https://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/)
5. [Minimum cost to connect all cities](https://www.geeksforgeeks.org/minimum-cost-connect-cities/)
6. [Total number of Spanning Trees in a Graph](https://www.geeksforgeeks.org/total-number-spanning-trees-graph/)
7. [Minimum Product Spanning Tree](https://www.geeksforgeeks.org/minimum-product-spanning-tree/)
8. [Reverse Delete Algorithm for Minimum Spanning Tree](https://www.geeksforgeeks.org/reverse-delete-algorithm-minimum-spanning-tree/)
9. [Boruvka’s algorithm for Minimum Spanning Tree](https://www.geeksforgeeks.org/greedy-algorithms-set-9-boruvkas-algorithm/)

**Topological Sorting:**

1. [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/)
2. [All topological sorts of a Directed Acyclic Graph](https://www.geeksforgeeks.org/all-topological-sorts-of-a-directed-acyclic-graph/)
3. [Kahn’s Algorithm for Topological Sorting](https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/)
4. [Maximum edges that can be added to DAG so that is remains DAG](https://www.geeksforgeeks.org/maximum-edges-can-added-dag-remains-dag/)
5. [Longest Path in a Directed Acyclic Graph](https://www.geeksforgeeks.org/find-longest-path-directed-acyclic-graph/)
6. [Topological Sort of a graph using departure time of vertex](https://www.geeksforgeeks.org/topological-sorting-using-departure-time-of-vertex/)

**Connectivity:**

1. [Articulation Points (or Cut Vertices) in a Graph](https://www.geeksforgeeks.org/articulation-points-or-cut-vertices-in-a-graph/)
2. [Biconnected Components](https://www.geeksforgeeks.org/biconnected-components/)
3. [Bridges in a graph](https://www.geeksforgeeks.org/bridge-in-a-graph/)
4. [Eulerian path and circuit](https://www.geeksforgeeks.org/eulerian-path-and-circuit/)
5. [Fleury’s Algorithm for printing Eulerian Path or Circuit](https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/)
6. [Strongly Connected Components](https://www.geeksforgeeks.org/strongly-connected-components/)
7. [Count all possible walks from a source to a destination with exactly k edges](https://www.geeksforgeeks.org/count-possible-paths-source-destination-exactly-k-edges/)
8. [Euler Circuit in a Directed Graph](https://www.geeksforgeeks.org/euler-circuit-directed-graph/)
9. [Length of shortest chain to reach the target word](https://www.geeksforgeeks.org/length-of-shortest-chain-to-reach-a-target-word/)
10. [Find if an array of strings can be chained to form a circle](https://www.geeksforgeeks.org/given-array-strings-find-strings-can-chained-form-circle/)
11. [Tarjan’s Algorithm to find strongly connected Components](https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/)
12. [Paths to travel each nodes using each edge (Seven Bridges of Königsberg)](https://www.geeksforgeeks.org/paths-travel-nodes-using-edgeseven-bridges-konigsberg/)
13. [Dynamic Connectivity | Set 1 (Incremental)](https://www.geeksforgeeks.org/dynamic-connectivity-set-1-incremental/)

**Maximum Flow**

1. [Max Flow Problem Introduction](https://www.geeksforgeeks.org/max-flow-problem-introduction/)
2. [Ford-Fulkerson Algorithm for Maximum Flow Problem](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/)
3. [Find maximum number of edge disjoint paths between two vertices](https://www.geeksforgeeks.org/find-edge-disjoint-paths-two-vertices/)
4. [Find minimum s-t cut in a flow network](https://www.geeksforgeeks.org/minimum-cut-in-a-directed-graph/)
5. [Maximum Bipartite Matching](https://www.geeksforgeeks.org/maximum-bipartite-matching/)
6. [Channel Assignment Problem](https://www.geeksforgeeks.org/channel-assignment-problem/)
7. [Introduction to Push Relabel Algorithm](https://www.geeksforgeeks.org/introduction-to-push-relabel-algorithm/)
8. [Karger’s Algorithm- Set 1- Introduction and Implementation](https://www.geeksforgeeks.org/kargers-algorithm-for-minimum-cut-set-1-introduction-and-implementation/)
9. [Dinic’s algorithm for Maximum Flow](https://www.geeksforgeeks.org/dinics-algorithm-maximum-flow/)

**Some must do Problems on Graph:**

1. [Find length of the largest region in Boolean Matrix](https://www.geeksforgeeks.org/find-length-largest-region-boolean-matrix/)
2. [Count number of trees in a forest](https://www.geeksforgeeks.org/count-number-trees-forest/)
3. [A Peterson Graph Problem](https://www.geeksforgeeks.org/peterson-graph/)
4. [Clone an Undirected Graph](https://www.geeksforgeeks.org/clone-an-undirected-graph/)
5. [Graph Coloring (Introduction and Applications)](https://www.geeksforgeeks.org/graph-coloring-applications/)
6. [Traveling Salesman Problem (TSP) Implementation](https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/)
7. [Vertex Cover Problem | Set 1 (Introduction and Approximate Algorithm)](https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/)
8. [K Centers Problem | Set 1 (Greedy Approximate Algorithm)](https://www.geeksforgeeks.org/k-centers-problem-set-1-greedy-approximate-algorithm/)
9. [Erdos Renyl Model (for generating Random Graphs)](https://www.geeksforgeeks.org/erdos-renyl-model-generating-random-graphs/)
10. [Chinese Postman or Route Inspection | Set 1 (introduction)](https://www.geeksforgeeks.org/chinese-postman-route-inspection-set-1-introduction/)
11. [Hierholzer’s Algorithm for directed graph](https://www.geeksforgeeks.org/hierholzers-algorithm-directed-graph/)
12. [Check whether a given graph is Bipartite or not](https://www.geeksforgeeks.org/bipartite-graph/)
13. [Snake and Ladder Problem](https://www.geeksforgeeks.org/snake-ladder-problem-2/)
14. [Boggle (Find all possible words in a board of characters)](https://www.geeksforgeeks.org/boggle-find-possible-words-board-characters/)
15. [Hopcroft Karp Algorithm for Maximum Matching-Introduction](https://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-1-introduction/)
16. [Minimum Time to rot all oranges](https://www.geeksforgeeks.org/minimum-time-required-so-that-all-oranges-become-rotten/)
17. [Construct a graph from given degrees of all vertices](https://www.geeksforgeeks.org/construct-graph-given-degrees-vertices/)
18. [Determine whether a universal sink exists in a directed graph](https://www.geeksforgeeks.org/determine-whether-universal-sink-exists-directed-graph/)
19. [Number of sink nodes in a graph](https://www.geeksforgeeks.org/number-sink-nodes-graph/)
20. [Two Clique Problem (Check if Graph can be divided in two Cliques)](https://www.geeksforgeeks.org/two-clique-problem-check-graph-can-divided-two-cliques/)

**Some Quizzes:**

* [Quizzes on Graph Traversal](https://www.geeksforgeeks.org/algorithms-gq/top-mcqs-on-graph-traversals-with-answers/?ref=graph_lp)
* [Quizzes on Graph Shortest Path](https://www.geeksforgeeks.org/algorithms-gq/top-mcqs-on-shortest-paths-in-graphs-with-answers/?ref=graph_lp)
* [Quizzes on Graph Minimum Spanning Tree](https://www.geeksforgeeks.org/algorithms-gq/top-mcqs-on-minimum-spanning-tree-mst-in-graphs-with-answers/?ref=graph_lp)
* [Quizzes on Graphs](https://www.geeksforgeeks.org/data-structure-gq/top-mcqs-on-graph-data-strcuture-with-answers/?ref=graph_lp)

**Quick Links :**

* [Top 10 Interview Questions on Depth First Search (DFS)](https://www.geeksforgeeks.org/top-10-interview-question-depth-first-search-dfs/)
* [Some interesting shortest path questions](https://www.geeksforgeeks.org/interesting-shortest-path-questions-set-1/)
* [Practice Problems on Graphs](https://practice.geeksforgeeks.org/topics/Graph/?ref=taocp)
* [Videos on Graphs](https://www.youtube.com/playlist?list=PLqM7alHXFySEaZgcg7uRYJFBnYMLti-nh)

**Introduction to Graphs – Data Structure and Algorithm Tutorials**

**Introduction:**

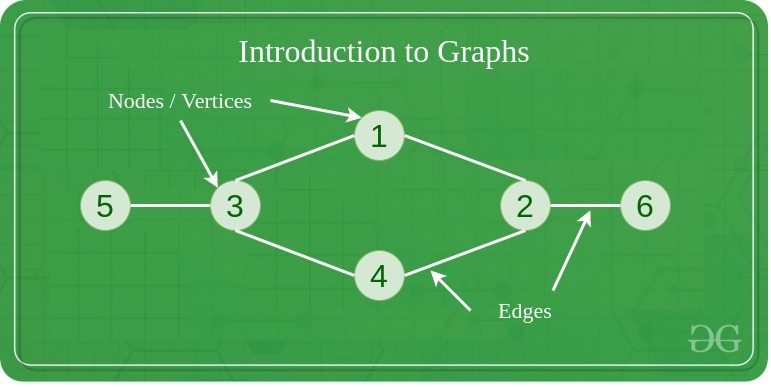
A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a [Graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/) is composed of a set of vertices( **V**) and a set of edges( **E**). The graph is denoted by **G(V, E).**

Graph data structures are a powerful tool for representing and analyzing complex relationships between objects or entities. They are particularly useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structures can be used to analyze and understand the dynamics of team performance and player interactions on the field.

Imagine a game of football as a web of connections, where players are the nodes and their interactions on the field are the edges. This web of connections is exactly what a graph data structure represents, and it’s the key to unlocking insights into team performance and player dynamics in sports.

**Components of a Graph**

* **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
* **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200630111809/graph18.jpg)

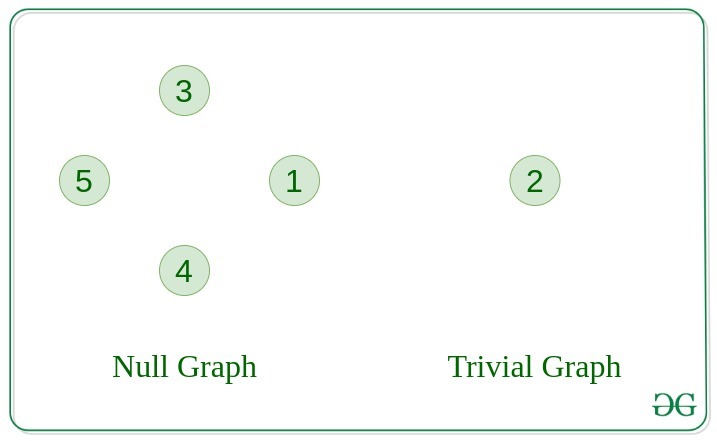
**Types Of Graph**

**1. Null Graph**

A graph is known as a null graph if there are no edges in the graph.

**2. Trivial Graph**

Graph having only a single vertex, it is also the smallest graph possible.

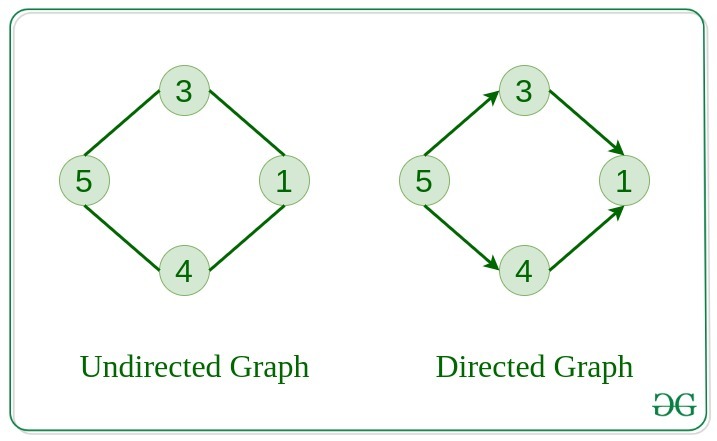
[](https://media.geeksforgeeks.org/wp-content/uploads/20200630113942/null_graph_trivial.jpg)

**3. Undirected Graph**

A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

**4. Directed Graph**

A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.

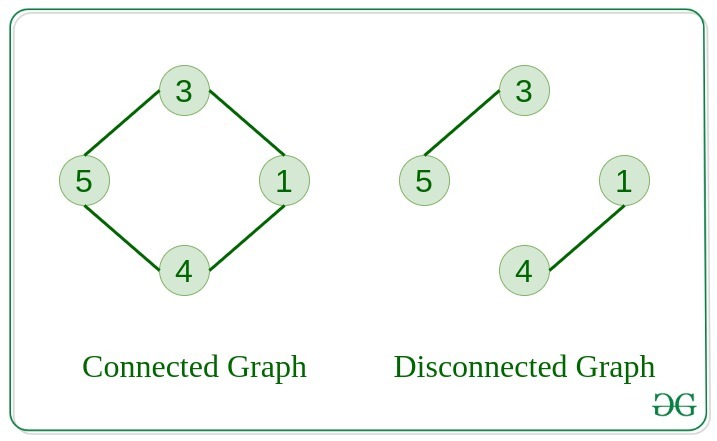
[](https://media.geeksforgeeks.org/wp-content/uploads/20200630114438/directed.jpg)

**5. Connected Graph**

The graph in which from one node we can visit any other node in the graph is known as a connected graph.

**6. Disconnected Graph**

The graph in which at least one node is not reachable from a node is known as a disconnected graph.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200630121400/connected1.jpg)

Advertisement

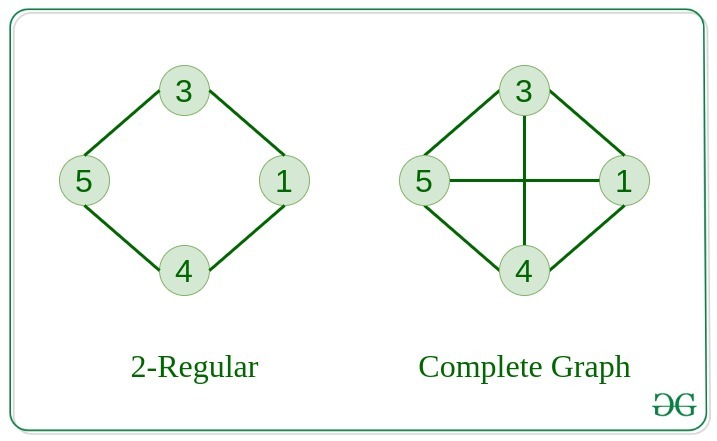
**7. Regular Graph**

The graph in which the degree of every vertex is equal to K is called K regular graph.

**8. Complete Graph**

The graph in which from each node there is an edge to each other node.

.

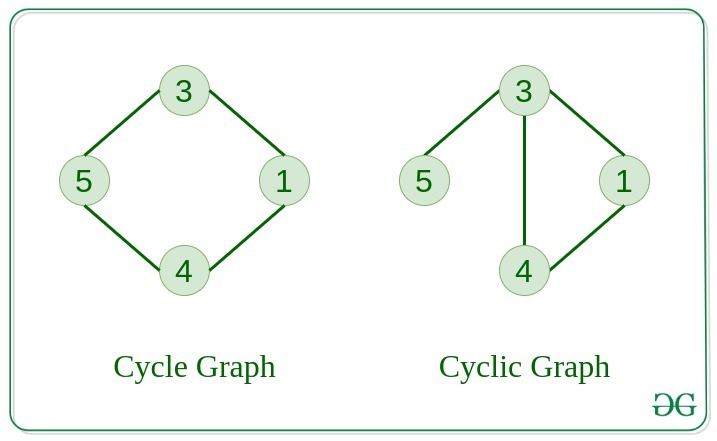
[](https://media.geeksforgeeks.org/wp-content/uploads/20200630122008/regular.jpg)

**9. Cycle Graph**

The graph in which the graph is a cycle in itself, the degree of each vertex is 2.

**10. Cyclic Graph**

A graph containing at least one cycle is known as a Cyclic graph.

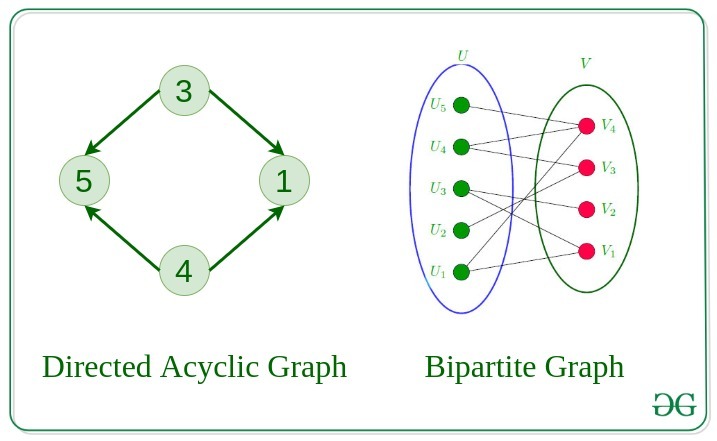
[](https://media.geeksforgeeks.org/wp-content/uploads/20200630122225/cyclic.jpg)

**11. Directed Acyclic Graph**

A Directed Graph that does not contain any cycle.

**12. Bipartite Graph**

A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.

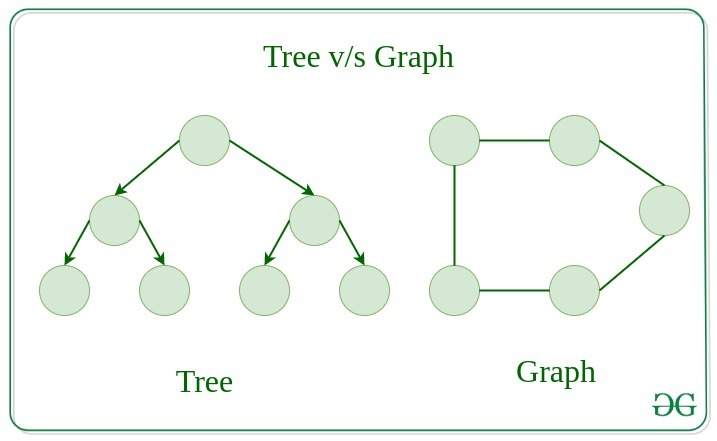
[](https://media.geeksforgeeks.org/wp-content/uploads/20200630122552/bipartite1.jpg)

**13. Weighted Graph**

* A graph in which the edges are already specified with suitable weight is known as a weighted graph.
* Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

**Tree v/s Graph**

Trees are the restricted types of graphs, just with some more rules. Every tree will always be a graph but not all graphs will be trees. [Linked List](https://www.geeksforgeeks.org/data-structures/linked-list/), [Trees](https://www.geeksforgeeks.org/binary-tree-data-structure/), and [Heaps](https://www.geeksforgeeks.org/heap-data-structure/) all are special cases of graphs.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200630123458/tree_vs_graph.jpg)

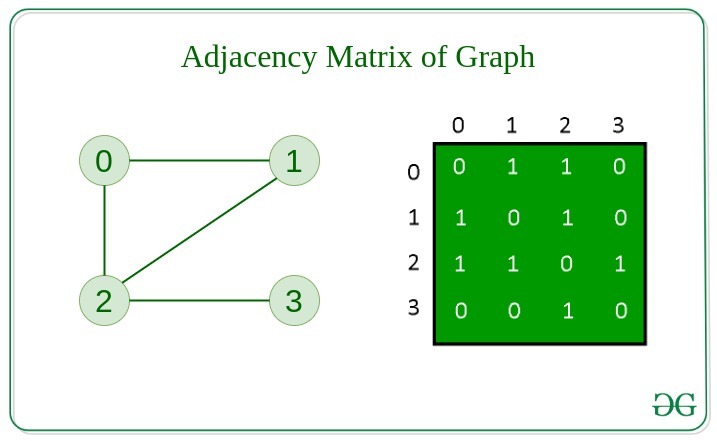
**Representation of Graphs**

There are two ways to store a graph:

* Adjacency Matrix
* Adjacency List

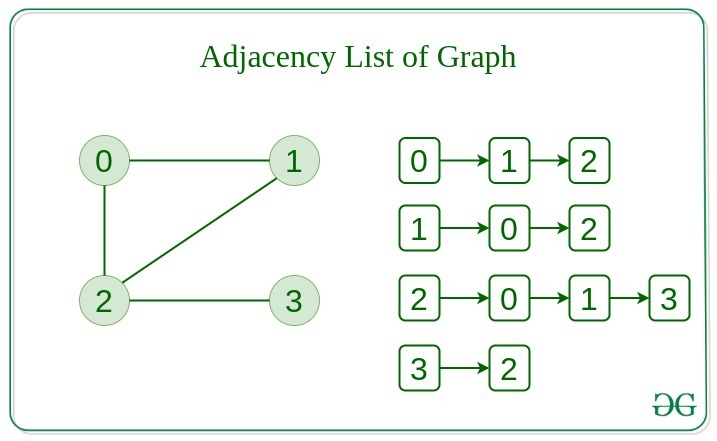
**Adjacency Matrix**

In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200630124726/adjacency_mat1.jpg)

**Adjacency List**

This graph is represented as a collection of linked lists. There is an array of pointer which points to the edges connected to that vertex.

[](https://media.geeksforgeeks.org/wp-content/uploads/20200630125356/adjacency_list.jpg)

**Comparison between Adjacency Matrix and Adjacency List**

When the graph contains a large number of edges then it is good to store it as a matrix because only some entries in the matrix will be empty. An algorithm such as [Prim’s](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/) and [Dijkstra](https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/) adjacency matrix is used to have less complexity.

|  |  |  |
| --- | --- | --- |
| **Action** | **Adjacency Matrix** | **Adjacency List** |
| Adding Edge | O(1) | O(1) |
| Removing an edge | O(1) | O(N) |
| Initializing | O(N\*N) | O(N) |

**Basic Operations on Graphs**

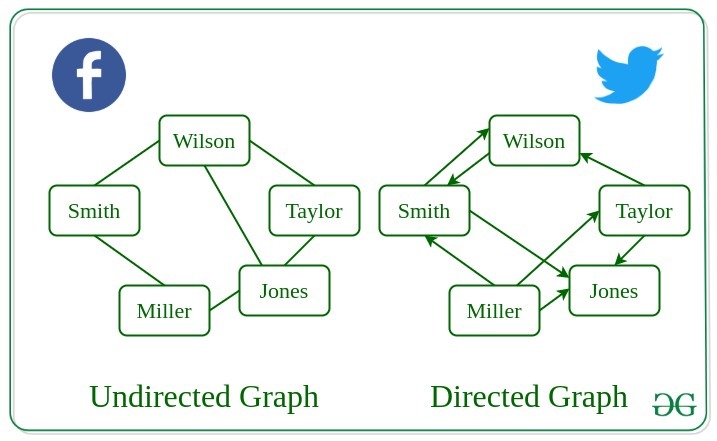
Below are the basic operations on the graph:

* Insertion of Nodes/Edges in the graph – Insert a node into the graph.
* Deletion of Nodes/Edges in the graph – Delete a node from the graph.
* Searching on Graphs – Search an entity in the graph.
* Traversal of Graphs – Traversing all the nodes in the graph.

**Usage of graphs**

* Maps can be represented using graphs and then can be used by computers to provide various services like the shortest path between two cities.
* When various tasks depend on each other then this situation can be represented using a Directed Acyclic graph and we can find the order in which tasks can be performed using topological sort.
* State Transition Diagram represents what can be the legal moves from current states. In-game of tic tac toe this can be used.

**Real-Life Applications of Graph**

[](https://media.geeksforgeeks.org/wp-content/uploads/20200630130949/applications_graph.jpg)

**Following are the real-life applications:**

* Graph data structures can be used to represent the interactions between players on a team, such as passes, shots, and tackles. Analyzing these interactions can provide insights into team dynamics and areas for improvement.
* Commonly used to represent social networks, such as networks of friends on social media.
* Graphs can be used to represent the topology of computer networks, such as the connections between routers and switches.
* Graphs are used to represent the connections between different places in a transportation network, such as roads and airports.
* **Neural Networks:**Vertices represent neurons and edges represent the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^11 neurons and close to 10^15 synapses.
* **Compilers:**Graphs are used extensively in compilers. They can be used for type inference, for so-called data flow analysis, register allocation, and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.
* **Robot planning:**Vertices represent states the robot can be in and the edges the possible transitions between the states. Such graph plans are used, for example, in planning paths for autonomous vehicles.

**When to use Graphs:**

* When you need to represent and analyze the relationships between different objects or entities.
* When you need to perform network analysis.
* When you need to identify key players, influencers or bottlenecks in a system.
* When you need to make predictions or recommendations.
* Modeling networks: Graphs are commonly used to model various types of networks, such as social networks, transportation networks, and computer networks. In these cases, vertices represent nodes in the network, and edges represent the connections between them.
* Finding paths: Graphs are often used in algorithms for finding paths between two vertices in a graph, such as shortest path algorithms. For example, graphs can be used to find the fastest route between two cities on a map or the most efficient way to travel between multiple destinations.
* Representing data relationships: Graphs can be used to represent relationships between data objects, such as in a database or data structure. In these cases, vertices represent data objects, and edges represent the relationships between them.
* Analyzing data: Graphs can be used to analyze and visualize complex data, such as in data clustering algorithms or machine learning models. In these cases, vertices represent data points, and edges represent the similarities or differences between them.

However, there are also some scenarios where using a graph may not be the best approach. For example, if the data being represented is very simple or structured, a graph may be overkill and a simpler data structure may suffice. Additionally, if the graph is very large or complex, it may be difficult or computationally expensive to analyze or traverse, which could make using a graph less desirable.

**Advantages and Disadvantages:**

**Advantages:**

1. Graphs are a versatile data structure that can be used to represent a wide range of relationships and data structures.
2. They can be used to model and solve a wide range of problems, including pathfinding, data clustering, network analysis, and machine learning.
3. Graph algorithms are often very efficient and can be used to solve complex problems quickly and effectively.
4. Graphs can be used to represent complex data structures in a simple and intuitive way, making them easier to understand and analyze.

**Disadvantages:**

1. Graphs can be complex and difficult to understand, especially for people who are not familiar with graph theory or related algorithms.
2. Creating and manipulating graphs can be computationally expensive, especially for very large or complex graphs.
3. Graph algorithms can be difficult to design and implement correctly, and can be prone to bugs and errors.
4. Graphs can be difficult to visualize and analyze, especially for very large or complex graphs, which can make it challenging to extract meaningful insights from the data.

**Summary:**

* Graph data structures are a powerful tool for representing and analyzing relationships between objects or entities.
* Graphs can be used to represent the interactions between different objects or entities, and then analyze these interactions to identify patterns, clusters, communities, key players, influencers, bottlenecks and anomalies.
* In sports data science, graph data structures can be used to analyze and understand the dynamics of team performance and player interactions on the field.
* They can be used in a variety of fields such as Sports, Social media, transportation, cybersecurity and many more.

**Graph and its representations**

A graph is a data structure that consists of the following two components:

**1.** A finite set of vertices also called as nodes.

**2.** A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, and locale. See [this](http://en.wikipedia.org/wiki/Graph_theory#Applications)for more applications of graph.

Following is an example of an undirected graph with 5 vertices.

C:\Users\qj771f\AppData\Local\Temp\msohtmlclip1\02\clip_image012.png

The following two are the most commonly used representations of a graph.

**1.** Adjacency Matrix

**2.** Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

**Adjacency Matrix:**

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

In case of an undirected graph, we need to show that there is an edge from vertex i  to vertex j and vice versa. In code, we assign adj[i][j] = 1  and adj[j][i] = 1

In case of a directed graph, if there is an edge from vertex i to vertex j then we just assign adj[i][j]=1

The adjacency matrix for the above example graph is:

Adjacency Matrix Representation

*Pros:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

*Cons:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.  Computing all neighbors of a vertex takes O(V) time (Not efficient).

Please see [this](https://ide.geeksforgeeks.org/9je5j6jJ13) for a sample Python implementation of adjacency matrix.

**Implementation of taking input for adjacency matrix**

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    #  n is the number of vertices

    #  m is the number of edges

    n, m **=** map(int, input().split())

    adjMat **=** [[0 **for** i **in** range(n)]**for** j **in** range(n)]

**for** i **in** range(n):

        u, v **=** map(int, input().split())

        adjMat[u][v] **=** 1

        adjMat[v][u] **=** 1

        # for a directed graph with an edge pointing from u to v,we just assign

        # adjMat[u][v] as 1

**Output**

**Adjacency List:**

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the***i***th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

Adjacency List Representation of Graph

Advertisement

Recommended Problem

Print adjacency list

[Graph](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Graph&sortBy=submissions)

[Data Structures](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Data%20Structures&sortBy=submissions)

[Solve Problem](https://practice.geeksforgeeks.org/problems/print-adjacency-list-1587115620/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 74.3K

Note that in the below implementation, we use dynamic arrays (vector in C++/ArrayList in Java) to represent adjacency lists instead of the linked list. The vector implementation has advantages of cache friendliness.

"""

A Python program to demonstrate the adjacency

list representation of the graph

"""

# A class to represent the adjacency list of the node

**class** AdjNode:

**def** \_\_init\_\_(self, data):

        self.vertex **=** data

        self.next **=** None

# A class to represent a graph. A graph

# is the list of the adjacency lists.

# Size of the array will be the no. of the

# vertices "V"

**class** Graph:

**def** \_\_init\_\_(self, vertices):

        self.V **=** vertices

        self.graph **=** [None] **\*** self.V

    # Function to add an edge in an undirected graph

**def** add\_edge(self, src, dest):

        # Adding the node to the source node

        node **=** AdjNode(dest)

        node.next **=** self.graph[src]

        self.graph[src] **=** node

        # Adding the source node to the destination as

        # it is the undirected graph

        node **=** AdjNode(src)

        node.next **=** self.graph[dest]

        self.graph[dest] **=** node

    # Function to print the graph

**def** print\_graph(self):

**for** i **in** range(self.V):

**print**("Adjacency list of vertex {}\n head".format(i), end**=**"")

            temp **=** self.graph[i]

**while** temp:

                print(" -> {}".format(temp.vertex), end**=**"")

                temp **=** temp.next

            print(" \n")

# Driver program to the above graph class

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    V **=** 5

    graph **=** Graph(V)

    graph.add\_edge(0, 1)

    graph.add\_edge(0, 4)

    graph.add\_edge(1, 2)

    graph.add\_edge(1, 3)

    graph.add\_edge(1, 4)

    graph.add\_edge(2, 3)

    graph.add\_edge(3, 4)

    graph.print\_graph()

# This code is contributed by Kanav Malhotra

**Output**

Adjacency list of vertex 0  
 head -> 1-> 4

Adjacency list of vertex 1  
 head -> 0-> 2-> 3-> 4

Adjacency list of vertex 2  
 head -> 1-> 3

Adjacency list of vertex 3  
 head -> 1-> 2-> 4

Adjacency list of vertex 4  
 head -> 0-> 1-> 3

*Pros:* Saves space O(|V|+|E|). In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier. Computing all neighbors of a vertex takes optimal time.

*Cons:* Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done O(V).

 In Real-life problems,  graphs are sparse(|E| <<|V|2). That’s why adjacency lists Data structure is commonly used for storing graphs. Adjacency matrix will enforce (|V|2) bound on time complexity for such algorithms.

Reference:

<http://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29>

**Related Post:**

[Graph representation using STL for competitive programming | Set 1 (DFS of Unweighted and Undirected)](https://www.geeksforgeeks.org/graph-representation-using-stl-for-competitive-programming-set-1-dfs-of-unweighted-and-undirected/)

[Graph implementation using STL for competitive programming | Set 2 (Weighted graph)](https://www.geeksforgeeks.org/graph-implementation-using-stl-for-competitive-programming-set-2-weighted-graph/)

This article is compiled by [Aashish Barnwal](https://www.facebook.com/barnwal.aashish) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Applications of Depth First Search**

In this article we will deep dive into the world of application of Depth-First Search (DFS), the algorithm that traverses the depth of a graph before exploring its breadth. From topological sorting to pathfinding, cycle detection to maze generation, DFS is a versatile tool for solving a wide range of problems.

Following are the problems that use DFS as a building block.

**1) Detecting cycle in a graph**

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See [this](http://people.csail.mit.edu/thies/6.046-web/recitation9.txt)for details)

**2) Path Finding**

We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

iii) As soon as destination vertex z is encountered, return the path as the

contents of the stack

See [this](http://ww3.algorithmdesign.net/handouts/DFS.pdf)for details.

**3)**[**Topological Sorting**](https://www.geeksforgeeks.org/topological-sorting/)

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

**4) To test if a graph is**[**bipartite**](http://en.wikipedia.org/wiki/Bipartite_graph)

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn’t link two vertices of the same color. The first vertex in any connected component can be red or black! See [this](http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/LEC/LECTUR16/NODE16.HTM)for details.

**5) Finding Strongly Connected Components of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See [this](https://www.geeksforgeeks.org/strongly-connected-components/)for DFS-based algo for finding Strongly Connected Components)

**6) Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

**7) Web crawlers:** Depth-first search can be used in the implementation of web crawlers to explore the links on a website.

8) **Maze generation:** Depth-first search can be used to generate random mazes.

9)**Model checking:** Depth-first search can be used in model checking, which is the process of checking that a model of a system meets a certain set of properties.

10) **Backtracking:**Depth-first search can be used in backtracking algorithms.

**Sources:**

<http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/LEC/LECTUR16/NODE16.HTM>

<http://en.wikipedia.org/wiki/Depth-first_search>

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/depthSearch.htm>

<http://ww3.algorithmdesign.net/handouts/DFS.pdf>

**Basic Properties of a Graph**

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

**The basic properties of a graph include:**

1. Vertices (nodes): The points where edges meet in a graph are known as vertices or nodes. A vertex can represent a physical object, concept, or abstract entity.
2. Edges: The connections between vertices are known as edges. They can be undirected (bidirectional) or directed (unidirectional).
3. Weight: A weight can be assigned to an edge, representing the cost or distance between two vertices. A weighted graph is a graph where the edges have weights.
4. Degree: The degree of a vertex is the number of edges that connect to it. In a directed graph, the in-degree of a vertex is the number of edges that point to it, and the out-degree is the number of edges that start from it.
5. Path: A path is a sequence of vertices that are connected by edges. A simple path does not contain any repeated vertices or edges.
6. Cycle: A cycle is a path that starts and ends at the same vertex. A simple cycle does not contain any repeated vertices or edges.
7. Connectedness: A graph is said to be connected if there is a path between any two vertices. A disconnected graph is a graph that is not connected.
8. Planarity: A graph is said to be planar if it can be drawn on a plane without any edges crossing each other.
9. Bipartiteness: A graph is said to be bipartite if its vertices can be divided into two disjoint sets such that no two vertices in the same set are connected by an edge.

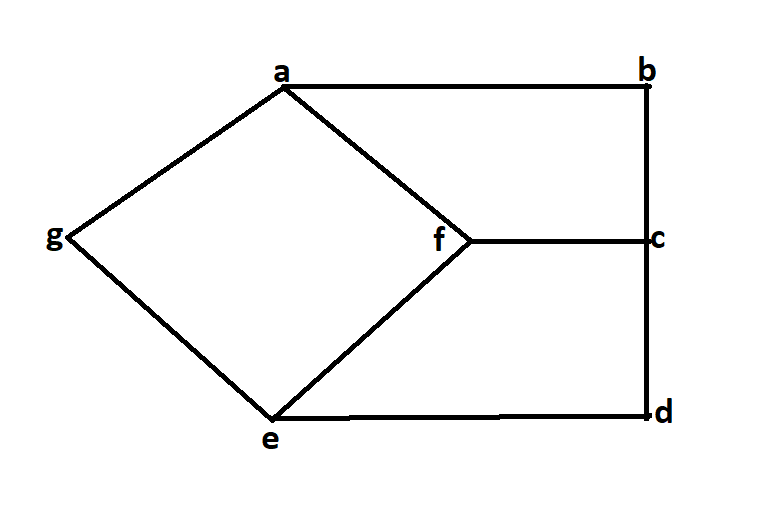
Properties of Graphs are basically used for the characterization of graphs depending on their structures. We defined these properties in specific terms that pertain to the domain of graph theory. In this article, we are going to discuss some properties of Graphs these are as follows:

**Distance between two Vertices:**

 It is basically the number of edges that are available in the shortest path between vertex A and vertex B.If there is more than one edge that is used to connect two vertices then we basically considered the shortest path as the distance between these two vertices.

Notation used :  
d(A, B)  
here function d is basically showing the distance between vertex A and vertex B.

* Let us understand this using an example:



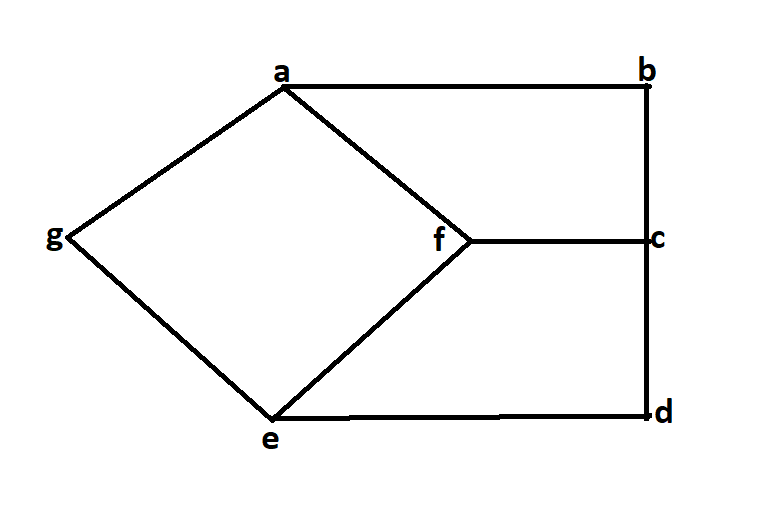
* In the above diagram, let’s try to find the distance between vertices b and d.

d(b, d)  
We can go from vertex b to vertex d in different ways such as  
1.ba, af, fe, ed here the d(b, d) will be 4.  
2.ba, af, fc, cd here the d(b, d) will be 4.  
3.bc, cf, fe, ed here the d(b, d) will be 4.  
4.bc, cd here the d(b, d) will be 2.  
hence the minimum distance between vertex b and vertex d is 2.

**The eccentricity of a Vertex:** Maximum distance from a vertex to all other vertices is considered as the Eccentricity of that vertex.

Notation used:  
e(V)  
here e(v) determines the eccentricity of vertex V.

* Let us try to understand this using following example.

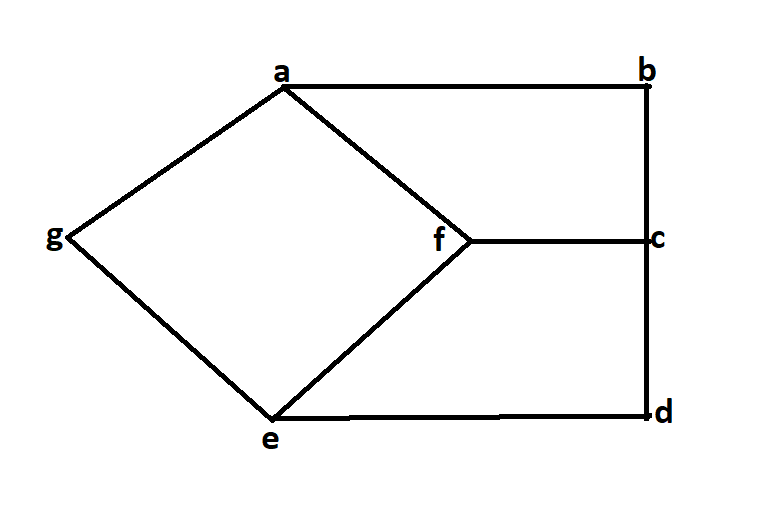


From the above diagram lets try to find the eccentricity of vertex b.  
e(b)  
d(b, a)=1  
d(b, c)=1  
d(b, d)=2  
d(b, e)=3  
d(b, f)=2  
d(b, g)=2  
Hence the eccentricity of vertex b is 3

**Radius of a Connected Graph:** The minimum value of eccentricity from all vertices is basically considered as the radius of connected graph.

Notation used:  
r(G)  
here G is the connected graph.

Let us try to understand this using following example.

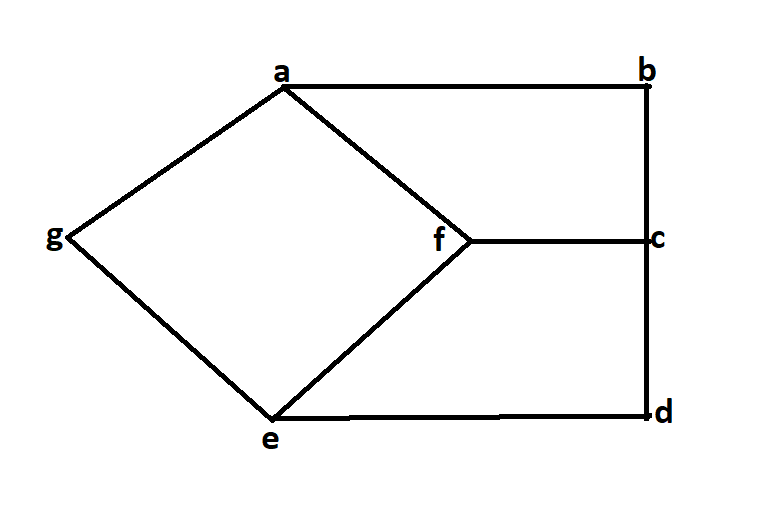


From the above diagram:  
r(G) is 2.  
Because the minimum value of eccentricity from all vertices is 2.

**Diameter of A Connected Graph:** Unlike the radius of the connected graph here we basically used the maximum value of eccentricity from all vertices to determine the diameter of the graph.

Notation used:  
d(G)  
where G is the connected graph.

* Let us try to understand this using following example.

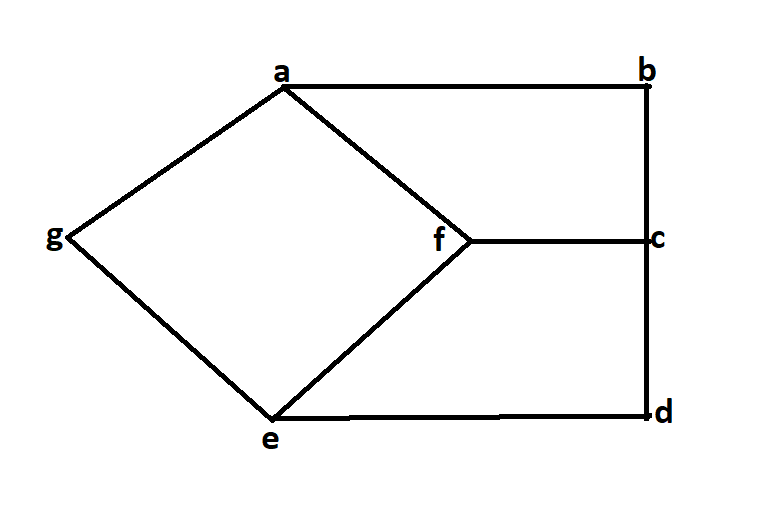


From the above diagram:  
d(G) is 3.  
Because the maximum value of eccentricity from all vertices is 3.

**Central Point and Centre:**The vertex having minimum eccentricity is considered as the central point of the graph.And the sets of all central point is considered as the centre of Graph.

if  
e(V)=r(G)  
then v is the central point.

* Let us try to understand this using following example.



In the above diagram the central point will be f.  
because   
e(f)=r(G)=2  
hence f is considered as the central point of graph.  
Hence f is also the centre of the graph.

**Applications, Advantages and Disadvantages of Graph**

[**Graph**](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/) is a non-linear data structure that contains nodes (vertices) and edges. A graph is a collection of set of vertices and edges (formed by connecting two vertices). A graph is defined as **G = {V, E}** where **V** is the set of vertices and **E** is the set of edges.

 Graphs can be used to model a wide variety of real-world problems, including social networks, transportation networks, and communication networks. They can be represented in various ways, such as by a set of vertices and a set of edges, or by a matrix or an adjacency list. The two most common types of graphs are directed and undirected graphs.

**Graph Representation**:

Graph can be represented in the following ways:

1. **Set Representation:**Set representation of a graph involves two sets: Set of vertices **V = {V1, V2, V3, V4}** and set of edges**E = {{V1, V2}, {V2, V3}, {V3, V4}, {V4, V1}}**. This representation is efficient for memory but does not allow parallel edges.
2. **Sequential Representation:**This representation of a graph can be represented by means of matrices: Adjacency Matrix, Incidence matrix and Path matrix.

* **Adjacency Matrix:** This matrix includes information about the adjacent nodes. Here, **aij = 1** if there is an edge from**Vi** to **Vj**otherwise **0**. It is a matrix of order **V×V**.
* **Incidence Matrix:** This matrix includes information about the incidence of edges on the nodes. Here, **aij = 1** if the**jth** edge **Ej**is incident on **ith** vertex **Vi** otherwise**0**. It is a matrix of order**V×E.**
* **Path Matrix:**This matrix includes information about the simple path between two vertices. Here, **Pij = 1** if there is a path from **Vi**to **Vj** otherwise **0**. It is also called as reachability matrix of graph **G**.

1. **Linked Representation:**This representation gives the information about the nodes to which a specific node is connected i.e. adjacency lists. This representation gives the adjacency lists of the vertices with the help of array and linked lists. In the adjacency lists, the vertices which are connected with the specific vertex are arranged in the form of lists which is connected to that vertex.

**Real-Time Applications of Graph:**

* **Social media analysis**: Social media platforms generate vast amounts of data in real-time, which can be analyzed using graphs to identify trends, sentiment, and key influencers. This can be useful for marketing, customer service, and reputation management.
* **Network monitoring:** Graphs can be used to monitor network traffic in real-time, allowing network administrators to identify potential bottlenecks, security threats, and other issues. This is critical for ensuring the smooth operation of complex networks.
* **Financial trading:** Graphs can be used to analyze real-time financial data, such as stock prices and market trends, to identify patterns and make trading decisions. This is particularly important for high-frequency trading, where even small delays can have a significant impact on profits.
* **Internet of Things (IoT) management:**IoT devices generate vast amounts of data in real-time, which can be analyzed using graphs to identify patterns, optimize performance, and detect anomalies. This is important for managing large-scale IoT deployments.
* **Autonomous vehicles:** Graphs can be used to model the real-time environment around autonomous vehicles, allowing them to navigate safely and efficiently. This requires real-time data from sensors and other sources, which can be processed using graph algorithms.
* **Disease surveillance**: Graphs can be used to model the spread of infectious diseases in real-time, allowing health officials to identify outbreaks and implement effective containment strategies. This is particularly important during pandemics or other public health emergencies.

**Advantages of Graph:**

* **Representing complex data:** Graphs are effective tools for representing complex data, especially when the relationships between the data points are not straightforward. They can help to uncover patterns, trends, and insights that may be difficult to see using other methods.
* **Efficient data processing:** Graphs can be processed efficiently using graph algorithms, which are specifically designed to work with graph data structures. This makes it possible to perform complex operations on large datasets quickly and effectively.
* **Network analysis:**Graphs are commonly used in network analysis to study relationships between individuals or organizations, as well as to identify important nodes and edges in a network. This is useful in a variety of fields, including social sciences, business, and marketing.
* **Pathfinding:**Graphs can be used to find the shortest path between two points, which is a common problem in computer science, logistics, and transportation planning.
* **Visualization**: Graphs are highly visual, making it easy to communicate complex data and relationships in a clear and concise way. This makes them useful for presentations, reports, and data analysis.
* **Machine** **learning**: Graphs can be used in machine learning to model complex relationships between variables, such as in recommendation systems or fraud detection.

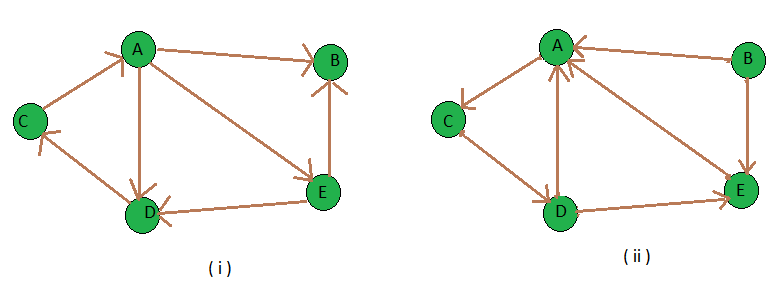
**Disadvantages of Graph:**

* **Limited representation:**Graphs can only represent relationships between objects, and not their properties or attributes. This means that in order to fully understand the data, it may be necessary to supplement the graph with additional information.
* **Difficulty in interpretation:** Graphs can be difficult to interpret, especially if they are large or complex. This can make it challenging to extract meaningful insights from the data, and may require advanced analytical techniques or domain expertise.
* **Scalability issue**s: As the number of nodes and edges in a graph increases, the processing time and memory required to analyze it also increases. This can make it difficult to work with large or complex graphs.
* **Data quality issues**: Graphs are only as good as the data they are based on, and if the data is incomplete, inconsistent, or inaccurate, the graph may not accurately reflect the relationships between objects.
* **Lack of standardization**: There are many different types of graphs, and each has its own strengths and weaknesses. This can make it difficult to compare graphs from different sources, or to choose the best type of graph for a given analysis.
* **Privacy concerns**: Graphs can reveal sensitive information about individuals or organizations, which can raise privacy concerns, especially in social network analysis or marketing.

**Transpose graph**

[Transpose](https://en.wikipedia.org/wiki/transposeGraph) of a directed graph G is another directed graph on the same set of vertices with all of the edges reversed compared to the orientation of the corresponding edges in G. That is, if G contains an edge (u, v) then the converse/transpose/reverse of G contains an edge (v, u) and vice versa. Given a [graph (represented as adjacency list)](https://www.geeksforgeeks.org/graph-and-its-representations/), we need to find another graph which is the transpose of the given graph.

**Example:**

[](https://media.geeksforgeeks.org/wp-content/uploads/transpose_both.png)

*Transpose Graph*

**Input :** figure (i) is the input graph.  
**Output :** figure (ii) is the transpose graph of the given graph.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

We traverse the adjacency list and as we find a vertex v in the adjacency list of vertex u which indicates an edge from u to v in main graph, we just add an edge from v to u in the transpose graph i.e. add u in the adjacency list of vertex v of the new graph. Thus traversing lists of all vertices of main graph we can get the transpose graph. Thus the total time complexity of the algorithm is O(V+E) where V is number of vertices of graph and E is the number of edges of the graph. Note : It is simple to get the transpose of a graph which is stored in adjacency matrix format, you just need to get the [transpose](https://www.geeksforgeeks.org/program-to-find-transpose-of-a-matrix/) of that matrix.

**Implementation:**

* C++
* Java
* Python3
* C#
* Javascript

# Python3 program to find transpose of a graph.

# function to add an edge from vertex

# source to vertex dest

**def** addEdge(adj, src, dest):

    adj[src].append(dest)

# function to print adjacency list

# of a graph

**def** displayGraph(adj, v):

**for** i **in** range(v):

        print(i, "--> ", end **=** "")

**for** j **in** range(len(adj[i])):

**print**(adj[i][j], end **=** " ")

**print**()

# function to get Transpose of a graph

# taking adjacency list of given graph

# and that of Transpose graph

**def** transposeGraph(adj, transpose, v):

    # traverse the adjacency list of given

    # graph and for each edge (u, v) add

    # an edge (v, u) in the transpose graph's

    # adjacency list

**for** i **in** range(v):

**for** j **in** range(len(adj[i])):

            addEdge(transpose, adj[i][j], i)

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    v **=** 5

    adj **=** [[] **for** i **in** range(v)]

    addEdge(adj, 0, 1)

    addEdge(adj, 0, 4)

    addEdge(adj, 0, 3)

    addEdge(adj, 2, 0)

    addEdge(adj, 3, 2)

    addEdge(adj, 4, 1)

    addEdge(adj, 4, 3)

    # Finding transpose of graph represented

    # by adjacency list adj[]

    transpose **=** [[]**for** i **in** range(v)]

    transposeGraph(adj, transpose, v)

    # displaying adjacency list of

    # transpose graph i.e. b

    displayGraph(transpose, v)

# This code is contributed by PranchalK

**Output**

0--> 2   
1--> 0 4   
2--> 3   
3--> 0 4   
4--> 0

**Difference between graph and tree**

[**Graph**](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/) :

A [graph](https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/) is a collection of two sets V and E where V is a finite non-empty set of vertices and E is a finite non-empty set of edges.

* Vertices are nothing but the nodes in the graph.
* Two adjacent vertices are joined by edges.
* Any graph is denoted as G = {V, E}.

For Example:



G = {{V1, V2, V3, V4, V5, V6}, {E1, E2, E3, E4, E5, E6, E7}}

A graph is a collection of vertices (also known as nodes) and edges that connect these vertices. Each edge represents a relationship or connection between two vertices.

Graphs can be directed or undirected, meaning that edges have a specific direction or they do not.

A directed graph is also known as a digraph. Graphs can also have weighted edges, where each edge has a weight or cost associated with it. Graphs can be represented in various ways, such as adjacency matrix or adjacency list.

[**Tree**](https://www.geeksforgeeks.org/binary-tree-data-structure/)**:**

A tree is a special type of graph that is connected and acyclic, meaning that there are no cycles in the graph.

 In a tree, there is a unique path between any two vertices, and there is a single vertex called the root that is used as the starting point for traversing the tree.

Trees can be used to model hierarchical relationships, such as the structure of a file system or the organization of a company.

Trees can be binary or non-binary. In a binary tree, each node has at most two children, while in a non-binary tree, each node can have any number of children.

Binary trees can be used to solve problems such as searching and sorting, as well as to represent expressions and parse trees.

A tree is a finite set of one or more nodes such that –

1. There is a specially designated node called root.
2. The remaining nodes are partitioned into n>=0 disjoint sets T1, T2, T3, …, Tn   
   where T1, T2, T3, …, Tn are called the subtrees of the root.

The concept of a tree is represented by following Fig.



**Graph vs Tree**

|  |  |  |
| --- | --- | --- |
| **The basis of Comparison** | **Graph** | **Tree** |
| **Definition** | **Graph is a non-linear data structure.** | **Tree is a non-linear data structure.** |
| **Structure** | **It is a collection of vertices/nodes and edges.** | **It is a collection of nodes and edges.** |
| **Structure cycle** | **A graph can be connected or disconnected, can have cycles or loops, and does not necessarily have a root node.** | **A tree is a type of graph that is connected, acyclic (meaning it has no cycles or loops), and has a single root node.** |
| **Edges** | **Each node can have any number of edges.** | **If there is n nodes then there would be n-1 number of edges** |
| **Types of Edges** | **They can be directed or undirected** | **They are always directed** |
| **Root node** | **There is no unique node called root in graph.** | **There is a unique node called root(parent) node in trees.** |
| **Loop Formation** | **A cycle can be formed.** | **There will not be any cycle.** |
| **Traversal** | **For graph traversal, we use**[Breadth-First Search (BFS)](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)**, and**[Depth-First Search (DFS)](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/)**.** | **We traverse a tree using**[in-order, pre-order, or post-order](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/)**traversal methods.** |
| **Applications** | **For finding shortest path in networking graph is used.** | **For game trees, decision trees, the tree is used.** |
| **Node relationships** | **In a graph, nodes can have any number of connections to other nodes, and there is no strict parent-child relationship.** | **In a tree, each node (except the root node) has a parent node and zero or more child nodes.** |
| **Commonly used for** | **Graphs are commonly used to model complex systems or relationships, such as social networks, transportation networks, and computer networks.** | **Trees are commonly used to represent data that has a hierarchical structure, such as file systems, organization charts, and family trees.** |
| **Connectivity** | **In a graph, nodes can have any number of connections to other nodes.** | **In a tree, each node can have at most one parent, except for the root node, which has no parent.** |

**Breadth First Search or BFS for a Graph**

*The breadth-first search (BFS) algorithm is used to search a tree or graph data structure for a node that meets a set of criteria. It starts at the tree’s root or graph and searches/visits all nodes at the current depth level before moving on to the nodes at the next depth level. Breadth-first search can be used to solve many problems in graph theory.*

[Breadth-First Traversal (or Search)](http://en.wikipedia.org/wiki/Breadth-first_search) for a graph is similar to the Breadth-First Traversal of a tree (See method 2 of [this post](https://www.geeksforgeeks.org/level-order-tree-traversal/)).

The only catch here is, that, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we divide the vertices into two categories:

1. Visited and
2. Not visited.

A boolean visited array is used to mark the visited vertices. For simplicity, it is assumed that all vertices are reachable from the starting vertex. BFS uses a [**queue data structure**](https://www.geeksforgeeks.org/queue-data-structure/) for traversal.

**Example:**

*In the following graph, we start traversal from vertex 2.*



*When we come to****vertex 0****, we look for all adjacent vertices of it.*

1. *2 is also an adjacent vertex of 0.*
2. *If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process.*

*There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :*

*2, 3, 0, 1*

*2, 0, 3, 1*

Recommended Problem

BFS of graph

[Graph](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=Graph&sortBy=submissions)

[BFS](https://practice.geeksforgeeks.org/explore?page=1&category%5b%5d=BFS&sortBy=submissions)

+2 more

[Flipkart](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Flipkart&sortBy=submissions)

[Amazon](https://practice.geeksforgeeks.org/explore?page=1&company%5b%5d=Amazon&sortBy=submissions)

+5 more

[Solve Problem](https://practice.geeksforgeeks.org/problems/bfs-traversal-of-graph/1?utm_source=gfg&utm_medium=article&utm_campaign=bottom_sticky_on_article)

Submission count: 2.3L

**Implementation of BFS traversal on Graph:**

**Pseudocode:**

*Breadth\_First\_Search( Graph, X ) // Here, Graph is the graph that we already have and X is the source node*

*Let Q be the queue*

*Q.enqueue( X ) // Inserting source node X into the queue*

*Mark X node as visited.*

*While ( Q is not empty )*

*Y = Q.dequeue( ) // Removing the front node from the queue*

*Process all the neighbors of Y, For all the neighbors Z of Y*

*If Z is not visited, Q. enqueue( Z ) // Stores Z in Q*

*Mark Z as visited*

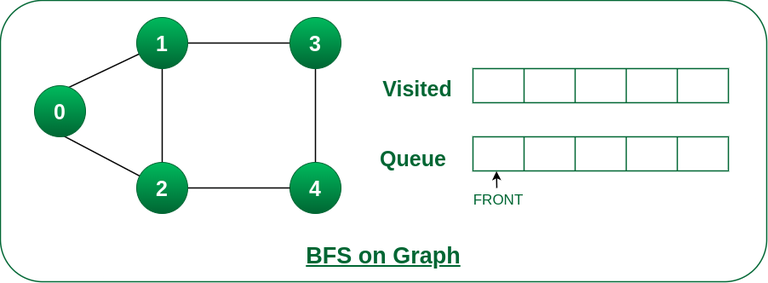
*Advertisement*

Follow the below method to implement BFS traversal.

1. Declare a queue and insert the starting vertex.
2. Initialize a **visited** array and mark the starting vertex as visited.
3. Follow the below process till the queue becomes empty:
4. Remove the first vertex of the queue.
5. Mark that vertex as visited.
6. Insert all the unvisited neighbors of the vertex into the queue.

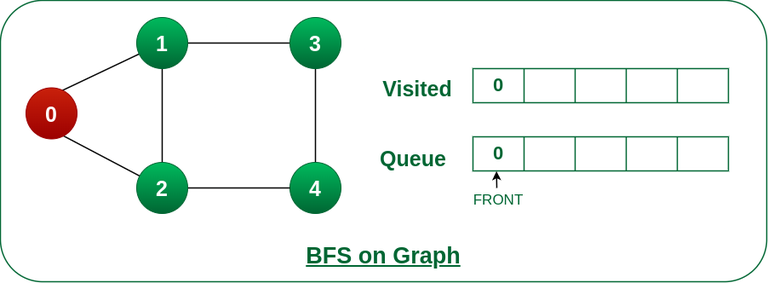
**Illustration:**

***Step1:****Initially queue and visited arrays are empty.*



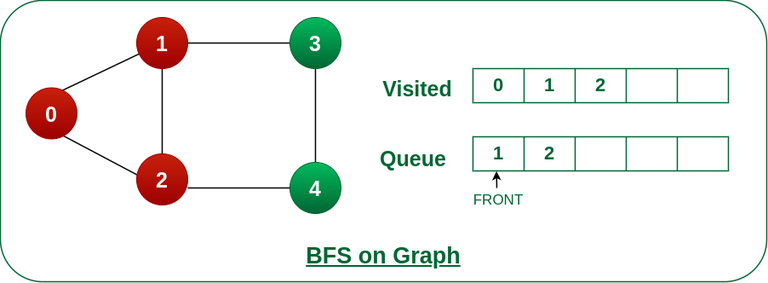
*Queue and visited arrays are empty initially.*

***Step2:****Push node 0 into queue and mark it visited.*



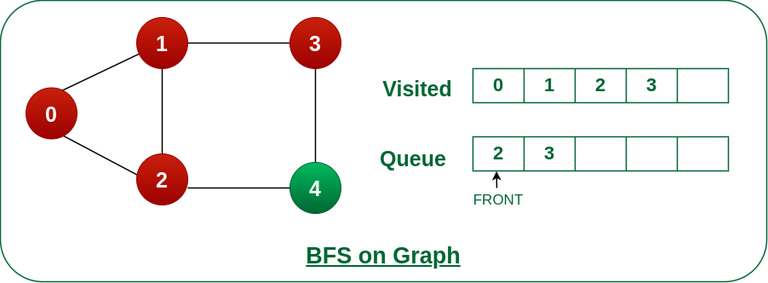
*Push node 0 into queue and mark it visited.*

***Step 3:****Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



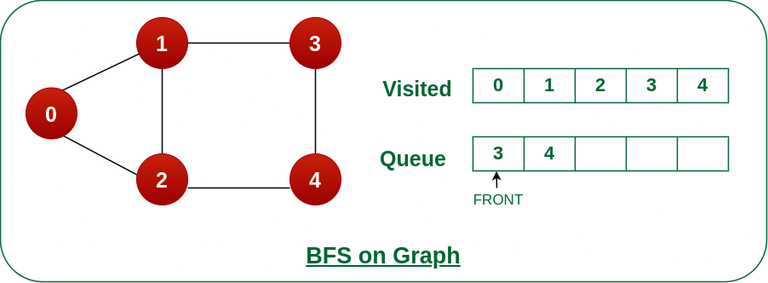
*Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.*

***Step 4:****Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*



*Remove node 1 from the front of queue and visited the unvisited neighbours and push*

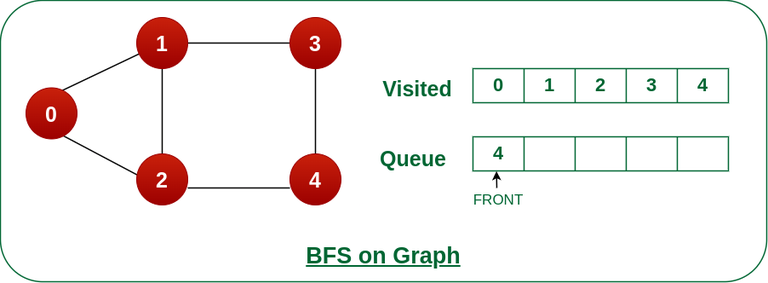
***Step 5:****Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*



*Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*

***Step 6:****Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

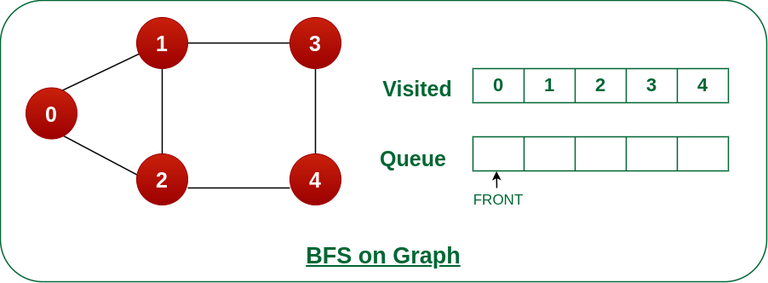
*As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.*



*Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

***Steps 7:****Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.*

*As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.*



*Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.*

*Now, Queue becomes empty, So, terminate these process of iteration.*

The implementation uses an [adjacency list representation](http://en.wikipedia.org/wiki/Adjacency_list) of graphs. [STL](http://en.wikipedia.org/wiki/Standard_Template_Library)‘s [list container](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST) stores lists of adjacent nodes and the queue of nodes needed for BFS traversal.

# Python3 Program to print BFS traversal

# from a given source vertex. BFS(int s)

# traverses vertices reachable from s.

**from** collections **import** defaultdict

# This class represents a directed graph

# using adjacency list representation

**class** Graph:

    # Constructor

**def** \_\_init\_\_(self):

        # default dictionary to store graph

        self.graph **=** defaultdict(list)

    # function to add an edge to graph

**def** addEdge(self, u, v):

        self.graph[u].append(v)

    # Function to print a BFS of graph

**def** BFS(self, s):

        # Mark all the vertices as not visited

        visited **=** [False] **\*** (max(self.graph) **+** 1)

        # Create a queue for BFS

        queue **=** []

        # Mark the source node as

        # visited and enqueue it

        queue.append(s)

        visited[s] **=** True

**while** queue:

            # Dequeue a vertex from

            # queue and print it

            s **=** queue.pop(0)

            print(s, end**=**" ")

            # Get all adjacent vertices of the

            # dequeued vertex s. If a adjacent

            # has not been visited, then mark it

            # visited and enqueue it

**for** i **in** self.graph[s]:

**if** visited[i] **==** False:

                    queue.append(i)

                    visited[i] **=** True

# Driver code

# Create a graph given in

# the above diagram

g **=** Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

**print**("Following is Breadth First Traversal"

      " (starting from vertex 2)")

g.BFS(2)

# This code is contributed by Neelam Yadav

**Output**

Following is Breadth First Traversal (starting from vertex 2)   
2 0 3 1

**Time Complexity:**O(V+E), where V is the number of nodes and E is the number of edges.

**Auxiliary Space:**O(V)

**BFS for Disconnected Graph:**

Note that the above code traverses only the vertices reachable from a given source vertex. In every situation, all the vertices may not be reachable from a given vertex (i.e. for a disconnected graph).

*To print all the vertices, we can modify the BFS function to do traversal starting from all nodes one by one (Like the*[*DFS modified version*](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/)*).*

Below is the implementation for BFS traversal for the entire graph (valid for directed as well as undirected graphs) with possible multiple disconnected components:

'''

Generic Function for BFS traversal of a Graph

 (valid for directed as well as undirected graphs

 which can have multiple disconnected components)

-- Inputs --

-> V - represents number of vertices in the Graph

-> adj[] - represents adjacency list for the Graph

-- Output --

-> bfs\_traversal - a vector containing bfs traversal

for entire graph

'''

**def** bfsOfGraph(V, adj):

    bfs\_traversal **=** []

    vis **=** [False]**\***V

**for** i **in** range(V):

        # To check if already visited

**if** (vis[i] **==** False):

            q **=** []

            vis[i] **=** True

            q.append(i)

            # BFS starting from ith node

**while** (len(q) > 0):

                g\_node **=** q.pop(0)

                bfs\_traversal.append(g\_node)

**for** it **in** adj[g\_node]:

**if** (vis[it] **==** False):

                        vis[it] **=** True

                        q.append(it)

**return** bfs\_traversal

  # This code is contributed by Abhijeet Kumar(abhijeet19403)

**Time Complexity: O(V+E)**

Where V is the number of vertices and E is the number of edges in the graph.

**Space Complexity: O(V)**

We used an array of size V to store the BFS traversal. We also used an array of size V to keep track of visited vertices. We used a queue of size V to store the vertices.

**Problems related to BFS:**

|  |  |  |
| --- | --- | --- |
| **S.no** | **Problems** | **Practice** |
| 1. | [Find the level of a given node in an Undirected Graph](https://www.geeksforgeeks.org/find-the-level-of-given-node-in-an-undirected-graph/) | [Link](https://ide.geeksforgeeks.org/) |
| 2. | [Minimize maximum adjacent difference in a path from top-left to bottom-right](https://www.geeksforgeeks.org/minimize-maximum-adjacent-difference-in-a-path-from-top-left-to-bottom-right/) | [Link](https://ide.geeksforgeeks.org/) |
| 3. | [Minimum jump to the same value or adjacent to reach the end of an Array](https://www.geeksforgeeks.org/minimum-jumps-to-same-value-or-adjacent-to-reach-end-of-array/) | [Link](https://ide.geeksforgeeks.org/) |
| 4. | [Maximum coin in minimum time by skipping K obstacles along the path in Matrix](https://www.geeksforgeeks.org/maximum-coin-in-minimum-time-by-skipping-k-obstacles-along-path-in-matrix/) | [Link](https://ide.geeksforgeeks.org/) |
| 5. | [Check if all nodes of the Undirected Graph can be visited from the given Node](https://www.geeksforgeeks.org/check-if-all-nodes-of-undirected-graph-can-be-visited-from-given-node/) | [Link](https://ide.geeksforgeeks.org/) |
| 6. | [Minimum time to visit all nodes of a given Graph at least once](https://www.geeksforgeeks.org/minimum-time-to-visit-all-nodes-of-given-graph-at-least-once/) | [Link](https://ide.geeksforgeeks.org/) |
| 7. | [Minimize moves to the next greater element to reach the end of the Array](https://www.geeksforgeeks.org/minimize-moves-to-next-greater-element-to-reach-end-of-array/) | [Link](https://ide.geeksforgeeks.org/) |
| 8. | [Shortest path by removing K walls](https://www.geeksforgeeks.org/shortest-path-by-removing-k-walls/) | [Link](https://ide.geeksforgeeks.org/) |
| 9. | [Minimum time required to infect all the nodes of the Binary tree](https://www.geeksforgeeks.org/minimum-time-required-to-infect-all-the-nodes-of-binary-tree/) | [Link](https://ide.geeksforgeeks.org/) |
| 10. | [Check if destination of given Matrix is reachable with required values of cells](https://www.geeksforgeeks.org/check-if-destination-of-given-matrix-is-reachable-with-required-values-of-cells/) | Link |

**Applications of BFS:**

1. **Shortest Path and Minimum Spanning Tree for unweighted graph:**In an unweighted graph, the shortest path is the path with the least number of edges. With Breadth First, we always reach a vertex from a given source using the minimum number of edges. Also, in the case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
2. **Peer-to-Peer Networks:** In Peer-to-Peer Networks like [**BitTorrent**](https://www.geeksforgeeks.org/how-bittorrent-works/), Breadth First Search is used to find all neighbor nodes.
3. **Crawlers in Search Engines:** Crawlers build an index using Breadth First. The idea is to start from the source page and follow all links from the source and keep doing the same. Depth First Traversal can also be used for crawlers, but the advantage of Breadth First Traversal is, the depth or levels of the built tree can be limited.
4. **Social Networking Websites:**In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.
5. **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
6. **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
7. **In Garbage Collection:** Breadth First Search is used in copying garbage collection using [**Cheney’s algorithm**](http://en.wikipedia.org/wiki/Cheney%27s_algorithm). Refer [**this**](https://lambda.uta.edu/cse5317/notes/node48.html)and for details. Breadth First Search is preferred over Depth First Search because of the better locality of reference:
8. [**Cycle detection in the undirected graph:**](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/) In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use [BFS to detect cycle in a directed graph](https://www.geeksforgeeks.org/detect-cycle-in-a-directed-graph-using-bfs/) also,
9. [**Ford–Fulkerson algorithm**](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/)**:**In the Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst-case time complexity to O(VE2).
10. [**To test if a graph is Bipartite**](https://www.geeksforgeeks.org/bipartite-graph/)**:** We can either use Breadth First or Depth First Traversal.
11. **Path Finding:** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
12. **Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.

**Advantages of Breadth First Search:**

1. BFS will never get trapped exploring the useful path forever.
2. If there is a solution, BFS definitely find it out.
3. If there is more than one solution then BFS can find the minimal one that requires less number of steps. If there is a solution then BFS is guaranteed to find it.
4. Low storage requirement: linear with depth.
5. Easily programmed.

**Disadvantages of Breadth First Search:**

1. The main drawback of BFS is its memory requirement. Since each level of the tree must be saved in order to generate the next level and the amount of memory is proportional to the number of nodes stored the space complexity of BFS is O(bd ). As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.

**Depth First Search or DFS for a Graph**

1. Difficulty Level : [Easy](https://www.geeksforgeeks.org/easy/)
2. Last Updated : 24 Mar, 2023
3. Read
4. Discuss(110+)
5. Courses
6. Practice
7. Video

[Depth First Traversal (or Search)](http://en.wikipedia.org/wiki/Depth-first_search) for a graph is similar to [Depth First Traversal of a tree.](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/) The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

**Example:**

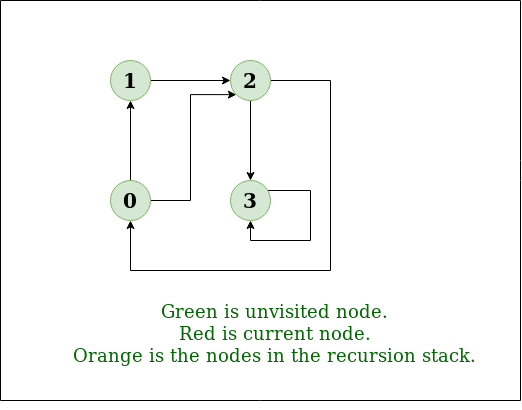
***Input:****n = 4, e = 6*

*0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3*

***Output:****DFS from vertex 1 : 1 2 0 3*

***Explanation:***

*DFS Diagram:*



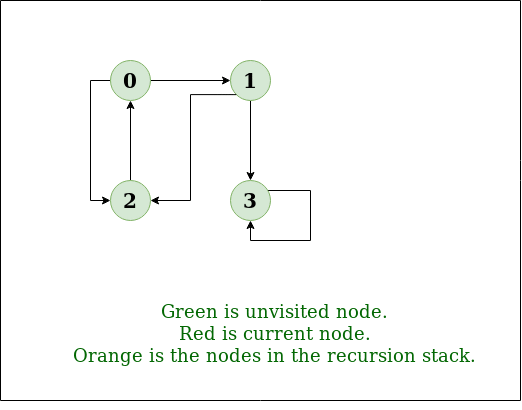
***Input:****n = 4, e = 6*

*2 -> 0, 0 -> 2, 1 -> 2, 0 -> 1, 3 -> 3, 1 -> 3*

***Output:****DFS from vertex 2 : 2 0 1 3*

***Explanation:***

*DFS Diagram:*



Recommended Problem

DFS of Graph

**Prerequisites:**  See [this post](https://www.geeksforgeeks.org/archives/11644) for all applications of Depth First Traversal.

*Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.*

*So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.*

Follow the below steps to solve the problem:

1. Create a recursive function that takes the index of the node and a visited array.
2. Mark the current node as visited and print the node.
3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

Below is the implementation of the above approach:

# Python3 program to print DFS traversal

# from a given  graph

**from** collections **import** defaultdict

# This class represents a directed graph using

# adjacency list representation

**class** Graph:

    # Constructor

**def** \_\_init\_\_(self):

        # default dictionary to store graph

        self.graph **=** defaultdict(list)

    # function to add an edge to graph

**def** addEdge(self, u, v):

        self.graph[u].append(v)

    # A function used by DFS

**def** DFSUtil(self, v, visited):

        # Mark the current node as visited

        # and print it

        visited.add(v)

**print**(v, end**=**' ')

        # Recur for all the vertices

        # adjacent to this vertex

**for** neighbour **in** self.graph[v]:

**if** neighbour **not in** visited:

                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses

    # recursive DFSUtil()

**def** DFS(self, v):

        # Create a set to store visited vertices

        visited **=** set()

        # Call the recursive helper function

        # to print DFS traversal

        self.DFSUtil(v, visited)

# Driver's code

# Create a graph given

# in the above diagram

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    g **=** Graph()

    g.addEdge(0, 1)

    g.addEdge(0, 2)

    g.addEdge(1, 2)

    g.addEdge(2, 0)

    g.addEdge(2, 3)

    g.addEdge(3, 3)

    print("Following is DFS from (starting from vertex 2)")

    # Function call

    g.DFS(2)

# This code is contributed by Neelam Yadav

**Output**

Following is Depth First Traversal (starting from vertex 2)   
2 0 1 3

**Time complexity:**O(V + E), where V is the number of vertices and E is the number of edges in the graph.

**Auxiliary Space:** O(V + E), since an extra visited array of size V is required, And stack size for iterative call to DFS function.

Advertisement

**Handling A Disconnected Graph:**

*This will happen by handling a corner case.*

*The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex, as in a Disconnected graph. To do a complete DFS traversal of such graphs, run DFS from all unvisited nodes after a DFS. The recursive function remains the same.*

Follow the below steps to solve the problem:

1. Create a recursive function that takes the index of the node and a visited array.
2. Mark the current node as visited and print the node.
3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.
4. Run a loop from 0 to the number of vertices and check if the node is unvisited in the previous DFS, then call the recursive function with the current node.

Below is the implementation of the above approach:

1. C++
2. Java
3. Python3
4. C#
5. Javascript

'''Python3 program to print DFS traversal for complete graph'''

**from** collections **import** defaultdict

# this class represents a directed graph using adjacency list representation

**class** Graph:

    # Constructor

**def** \_\_init\_\_(self):

        # default dictionary to store graph

        self.graph **=** defaultdict(list)

    # Function to add an edge to graph

**def** addEdge(self, u, v):

        self.graph[u].append(v)

    # A function used by DFS

**def** DFSUtil(self, v, visited):

        # Mark the current node as visited and print it

        visited.add(v)

        print(v, end**=**" ")

    # recur for all the vertices adjacent to this vertex

**for** neighbour **in** self.graph[v]:

**if** neighbour **not in** visited:

                self.DFSUtil(neighbour, visited)

    # The function to do DFS traversal. It uses recursive DFSUtil

**def** DFS(self):

        # create a set to store all visited vertices

        visited **=** set()

    # call the recursive helper function to print DFS traversal starting from all

    # vertices one by one

**for** vertex **in** self.graph:

**if** vertex **not in** visited:

                self.DFSUtil(vertex, visited)

# Driver's code

# create a graph given in the above diagram

**if** \_\_name\_\_ **==** "\_\_main\_\_":

**print**("Following is Depth First Traversal \n")

    g **=** Graph()

    g.addEdge(0, 1)

    g.addEdge(0, 2)

    g.addEdge(1, 2)

    g.addEdge(2, 0)

    g.addEdge(2, 3)

    g.addEdge(3, 3)

    # Function call

    g.DFS()

# This code is contributed by Priyank Namdeo

**Output**

Following is Depth First Traversal   
0 1 2 3

**Time complexity:**O(V + E), where V is the number of vertices and E is the number of edges in the graph.

**Auxiliary Space:**O(V), since an extra visited array of size V is required.

**Advantages of Depth First Search:**

1. Memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
2. The time complexity of a depth-first Search to depth d is O(bd) since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.
3. If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.
4. DFS requires less memory since only the nodes on the current path are stored. By chance DFS may find a solution without examining much of the search space at all.

**Disadvantages of Depth First Search:**

1. The disadvantage of Depth-First Search is that there is a possibility that it may down the left-most path forever. Even a finite graph can generate an infinite tre One solution to this problem is to impose a cutoff depth on the search. Although ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.
2. Depth-First Search is not guaranteed to find the solution.
3. And there is no guarantee to find a minimal solution, if more than one solution.

[**Applications of DFS.**](https://www.geeksforgeeks.org/applications-of-depth-first-search/)

Breadth-First[**Traversal for a Graph**](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/)

[**Recent Articles on DFS**](https://www.geeksforgeeks.org/tag/dfs/)

Would you please write comments if you find anything incorrect or share more information about the topic discussed above?

**Applications of Depth First Search**

In this article we will deep dive into the world of application of Depth-First Search (DFS), the algorithm that traverses the depth of a graph before exploring its breadth. From topological sorting to pathfinding, cycle detection to maze generation, DFS is a versatile tool for solving a wide range of problems.

Following are the problems that use DFS as a building block.

**1) Detecting cycle in a graph**

A graph has cycle if and only if we see a back edge during DFS. So we can run DFS for the graph and check for back edges. (See [this](http://people.csail.mit.edu/thies/6.046-web/recitation9.txt)for details)

**2) Path Finding**

We can specialize the DFS algorithm to find a path between two given vertices u and z.

i) Call DFS(G, u) with u as the start vertex.

ii) Use a stack S to keep track of the path between the start vertex and the current vertex.

iii) As soon as destination vertex z is encountered, return the path as the

contents of the stack

See [this](http://ww3.algorithmdesign.net/handouts/DFS.pdf)for details.

**3)**[**Topological Sorting**](https://www.geeksforgeeks.org/topological-sorting/)

Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers [2].

**4) To test if a graph is**[**bipartite**](http://en.wikipedia.org/wiki/Bipartite_graph)

We can augment either BFS or DFS when we first discover a new vertex, color it opposite its parents, and for each other edge, check it doesn’t link two vertices of the same color. The first vertex in any connected component can be red or black! See [this](http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/LEC/LECTUR16/NODE16.HTM)for details.

**5) Finding Strongly Connected Components of a graph** A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. (See [this](https://www.geeksforgeeks.org/strongly-connected-components/)for DFS-based algo for finding Strongly Connected Components)

**6) Solving puzzles with only one solution**, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

**7) Web crawlers:** Depth-first search can be used in the implementation of web crawlers to explore the links on a website.

8) **Maze generation:** Depth-first search can be used to generate random mazes.

9)**Model checking:** Depth-first search can be used in model checking, which is the process of checking that a model of a system meets a certain set of properties.

10) **Backtracking:**Depth-first search can be used in backtracking algorithms.

**Sources:**

<http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/LEC/LECTUR16/NODE16.HTM>

<http://en.wikipedia.org/wiki/Depth-first_search>

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/GraphAlgor/depthSearch.htm>

<http://ww3.algorithmdesign.net/handouts/DFS.pdf>

**Applications of Breadth First Traversal**

We have earlier discussed [Breadth First Traversal Algorithm](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/) for Graphs. We have also discussed [Applications of Depth First Traversal](https://www.geeksforgeeks.org/applications-of-depth-first-search/). In this article, applications of Breadth First Search are discussed.

1. **Shortest Path and Minimum Spanning Tree for unweighted graph**In an unweighted graph, the shortest path is the path with least number of edges. With Breadth First, we always reach a vertex from given source using the minimum number of edges. Also, in case of unweighted graphs, any spanning tree is Minimum Spanning Tree and we can use either Depth or Breadth first traversal for finding a spanning tree.
2. **Minimum Spanning Tree for weighted graphs:**We can also find Minimum Spanning Tree for weighted graphs using BFT, but the condition is that the weight should be non-negative and same for each pair of vertices.
3. **Peer to Peer Networks.** In Peer-to-Peer Networks like [**BitTorrent**](https://www.geeksforgeeks.org/how-bittorrent-works/), Breadth First Search is used to find all neighbor nodes.
4. **Crawlers in Search Engines:** Crawlers build index using Breadth First. The idea is to start from source page and follow all links from source and keep doing same. Depth First Traversal can also be used for crawlers, but the advantage with Breadth First Traversal is, depth or levels of the built tree can be limited.
5. **Social Networking Websites:**In social networks, we can find people within a given distance ‘k’ from a person using Breadth First Search till ‘k’ levels.
6. **GPS Navigation systems:** Breadth First Search is used to find all neighboring locations.
7. **Broadcasting in Network:** In networks, a broadcasted packet follows Breadth First Search to reach all nodes.
8. **In Garbage Collection:** Breadth First Search is used in copying garbage collection using [**Cheney’s algorithm**](http://en.wikipedia.org/wiki/Cheney%27s_algorithm). Refer [**this**](https://lambda.uta.edu/cse5317/notes/node48.html)and for details. Breadth First Search is preferred over Depth First Search because of better locality of reference:
9. [**Cycle detection in undirected graph:**](https://www.geeksforgeeks.org/detect-cycle-undirected-graph/) In undirected graphs, either Breadth First Search or Depth First Search can be used to detect cycle. We can use [**BFS to detect cycle in a directed graph**](https://www.geeksforgeeks.org/detect-cycle-in-a-directed-graph-using-bfs/) also,
10. [**Ford–Fulkerson algorithm**](https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/)In Ford-Fulkerson algorithm, we can either use Breadth First or Depth First Traversal to find the maximum flow. Breadth First Traversal is preferred as it reduces worst case time complexity to O(VE2).
11. [**To test if a graph is Bipartite**](https://www.geeksforgeeks.org/bipartite-graph/) We can either use Breadth First or Depth First Traversal.
12. **Path Finding** We can either use Breadth First or Depth First Traversal to find if there is a path between two vertices.
13. **Finding all nodes within one connected component:** We can either use Breadth First or Depth First Traversal to find all nodes reachable from a given node.
14. **AI:**In AI, BFS is used in traversing a game tree to find the best move.
15. **Network Security:**In the field of network security, BFS is used in traversing a network to find all the devices connected to it.
16. **Connected Component:**We can find all connected components in an undirected graph.
17. **Topological sorting:** BFS can be used to find a topological ordering of the nodes in a directed acyclic graph (DAG).
18. **Image processing:** BFS can be used to flood fill an image with a particular color or to find connected components of pixels.
19. **Recommender systems:** BFS can be used to find similar items in a large dataset by traversing the items’ connections in a similarity graph.

Many algorithms like [Prim’s Minimum Spanning Tree](https://www.geeksforgeeks.org/greedy-algorithms-set-5-prims-minimum-spanning-tree-mst-2/) and [Dijkstra’s Single Source Shortest Path](https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/) use structure similar to Breadth First Search.

There can be many more applications as Breadth First Search is one of the core algorithms for Graphs.

**Iterative Depth First Traversal of Graph**

[Depth First Traversal (or Search)](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) for a graph is similar to [Depth First Traversal (DFS) of a tree](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/). The only catch here is, unlike trees, graphs may contain cycles, so a node might be visited twice. To avoid processing a node more than once, use a boolean visited array.

**Example:**

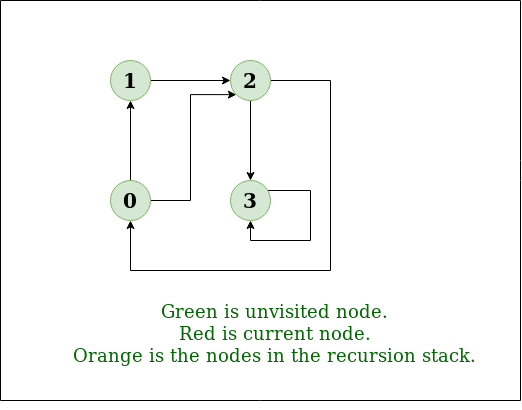
***Input:****n = 4, e = 6*

*0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3*

***Output:****DFS from vertex 1 : 1 2 0 3*

***Explanation:***

*DFS Diagram:*



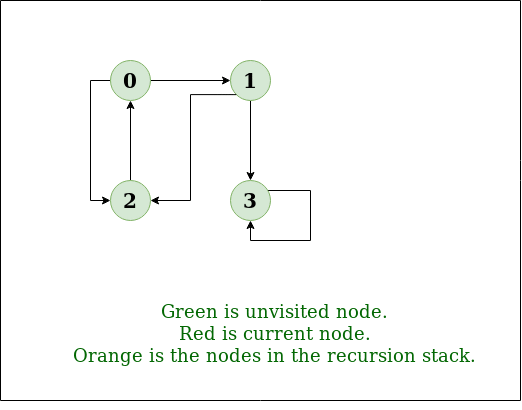
***Input:****n = 4, e = 6*

*2 -> 0, 0 -> 2, 1 -> 2, 0 -> 1, 3 -> 3, 1 -> 3*

***Output:****DFS from vertex 2 : 2 0 1 3*

***Explanation:***

*DFS Diagram:*



The recursive implementation of DFS is already discussed: [previous post](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/).

**Solution:**

1. **Approach:**Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.   
   The only difference between iterative DFS and recursive DFS is that the recursive stack is replaced by a stack of nodes.
2. **Algorithm:**
3. Created a stack of nodes and visited array.
4. Insert the root in the stack.
5. Run a loop till the stack is not empty.
6. Pop the element from the stack and print the element.
7. For every adjacent and unvisited node of current node, mark the node and insert it in the stack.

* **Implementation of Iterative DFS:** *This is similar to*[*BFS*](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/)*, the only difference is queue is replaced by stack.*

# An Iterative Python program to do DFS traversal from

# a given source vertex. DFS(int s) traverses vertices

# reachable from s.

# This class represents a directed graph using adjacency

# list representation

**class** Graph:

**def** \_\_init\_\_(self,V): # Constructor

        self.V **=** V        # No. of vertices

        self.adj  **=** [[] **for** i **in** range(V)]  # adjacency lists

**def** addEdge(self,v, w):     # to add an edge to graph

        self.adj[v].append(w)    # Add w to v’s list.

    # prints all not yet visited vertices reachable from s

**def** DFS(self,s):            # prints all vertices in DFS manner from a given source.

                                # Initially mark all vertices as not visited

        visited **=** [False **for** i **in** range(self.V)]

        # Create a stack for DFS

        stack **=** []

        # Push the current source node.

        stack.append(s)

**while** (len(stack)):

            # Pop a vertex from stack and print it

            s **=** stack[**-**1]

            stack.pop()

            # Stack may contain same vertex twice. So

            # we need to print the popped item only

            # if it is not visited.

**if** (**not** visited[s]):

**print**(s,end**=**' ')

                visited[s] **=** True

            # Get all adjacent vertices of the popped vertex s

            # If a adjacent has not been visited, then push it

            # to the stack.

**for** node **in** self.adj[s]:

**if** (**not** visited[node]):

                    stack.append(node)

# Driver program to test methods of graph class

g **=** Graph(5); # Total 5 vertices in graph

g.addEdge(1, 0);

g.addEdge(0, 2);

g.addEdge(2, 1);

g.addEdge(0, 3);

g.addEdge(1, 4);

print("Following is Depth First Traversal")

g.DFS(0)

# This code is contributed by ankush\_953

**Output**

Following is Depth First Traversal  
0 3 2 1 4

* **Complexity Analysis:**
* **Time complexity:** O(V + E), where V is the number of vertices and E is the number of edges in the graph.
* **Space Complexity:**O(V). Since an extra visited array is needed of size V.

**Modification of the above Solution:** Note that the above implementation prints only vertices that are reachable from a given vertex. For example, if the edges 0-3 and 0-2 are removed, then the above program would only print 0. To print all vertices of a graph, call DFS for every unvisited vertex.

**Implementation:**

# An Iterative Python3 program to do DFS

# traversal from a given source vertex.

# DFS(s) traverses vertices reachable from s.

**class** Graph:

**def** \_\_init\_\_(self, V):

        self.V **=** V

        self.adj **=** [[] **for** i **in** range(V)]

**def** addEdge(self, v, w):

        self.adj[v].append(w) # Add w to v’s list.

    # prints all not yet visited vertices

    # reachable from s

**def** DFSUtil(self, s, visited):

        # Create a stack for DFS

        stack **=** []

        # Push the current source node.

        stack.append(s)

**while** (len(stack) !**=** 0):

            # Pop a vertex from stack and print it

            s **=** stack.pop()

            # Stack may contain same vertex twice.

            # So we need to print the popped item only

            # if it is not visited.

**if** (**not** visited[s]):

**print**(s, end **=** " ")

                visited[s] **=** True

            # Get all adjacent vertices of the

            # popped vertex s. If a adjacent has not

            # been visited, then push it to the stack.

            i **=** 0

**while** i < len(self.adj[s]):

**if** (**not** visited[self.adj[s][i]]):

                    stack.append(self.adj[s][i])

                i **+=** 1

    # prints all vertices in DFS manner

**def** DFS(self):

        # Mark all the vertices as not visited

        visited **=** [False] **\*** self.V

**for** i **in** range(self.V):

**if** (**not** visited[i]):

                self.DFSUtil(i, visited)

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    g **=** Graph(5) # Total 5 vertices in graph

    g.addEdge(1, 0)

    g.addEdge(2, 1)

    g.addEdge(3, 4)

    g.addEdge(4, 0)

    print("Following is Depth First Traversal")

    g.DFS()

# This code is contributed by PranchalK

**Output**

Following is Depth First Traversal  
0 1 2 3 4

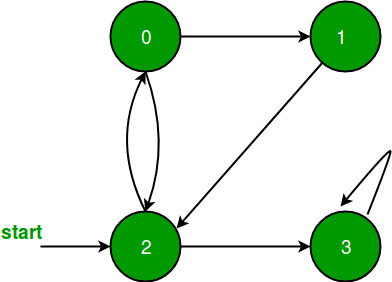
**Time complexity:**O(V+E), The time complexity of DFS is O (V+E). Here V is the number of vertices and E is the number of edges.

**Auxiliary Space:**O(V),The space complexity of DFS is O(V). The space is consumed by the recursion stack and the visited array.

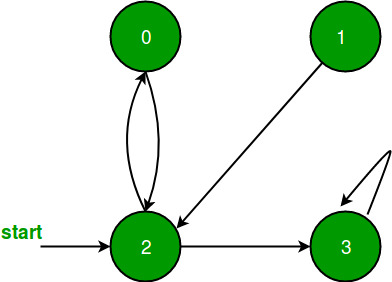
*From <*[*https://www.geeksforgeeks.org/iterative-depth-first-traversal/*](https://www.geeksforgeeks.org/iterative-depth-first-traversal/)*>*

**BFS for Disconnected Graph**

In the previous [post](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/), BFS only with a particular vertex is performed i.e. it is assumed that all vertices are reachable from the starting vertex. But in the case of a disconnected graph or any vertex that is unreachable from all vertex, the previous implementation will not give the desired output, so in this post, a modification is done in BFS.



 All vertices are reachable. So, for the above graph, simple [BFS](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/) will work.

[](https://media.geeksforgeeks.org/wp-content/uploads/graph4.png)

 As in the above graph vertex 1 is unreachable from all vertex, so simple BFS wouldn’t work for it.

Just to modify BFS, perform simple BFS from each   
unvisited vertex of given graph.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

Following is the code when adjacency matrix representation is used for the graph.

**import** queue

**def** add\_edge(edges, f, s):

    edges[f][s] **=** 1

**def** print\_bfs(edges, V, start, visited):

**if** V **==** 0:

**return**

    bfs **=** queue.Queue()

    bfs.put(start)

    visited[start] **=** 1

**while not** bfs.empty():

        data **=** bfs.get()

**print**(data, end**=**' ')

**for** i **in** range(V):

**if** edges[data][i] **==** 1:

**if** visited[i] **==** 0:

                    bfs.put(i)

                    visited[i] **=** 1

**def** bfs\_helper(edges, V):

**if** V **==** 0:

**return**

    visited **=** [0] **\*** V

**for** i **in** range(V):

**if** visited[i] **==** 0:

            print\_bfs(edges, V, i, visited)

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    V **=** 5

    E **=** 6

**if** E **==** 0:

**for** i **in** range(V):

            print(i, end**=**' ')

        exit()

    edges **=** [[0 **for** \_ **in** range(V)] **for** \_ **in** range(V)]

    add\_edge(edges, 0, 4)

    add\_edge(edges, 1, 2)

    add\_edge(edges, 1, 3)

    add\_edge(edges, 1, 4)

    add\_edge(edges, 2, 3)

    add\_edge(edges, 3, 4)

    bfs\_helper(edges, V)

**Output**

0 4 1 2 3

The **time complexity** of this algorithm is O(V + E), where V is the number of vertices and E is the number of edges. This is because we traverse each vertex and each edge once.

The **space complexity** is O(V), since we use an array to store the visited vertices.

Following is the code when adjacency list representation is used for the graph.

# Python3 implementation of modified BFS

**import** queue

# A utility function to add an edge

# in an undirected graph.

**def** addEdge(adj, u, v):

    adj[u].append(v)

# A utility function to do BFS of

# graph from a given vertex u.

**def** BFSUtil(u, adj, visited):

    # Create a queue for BFS

    q **=** queue.Queue()

    # Mark the current node as visited

    # and enqueue it

    visited[u] **=** True

    q.put(u)

    # 'i' will be used to get all adjacent

    # vertices 4 of a vertex list<int>::iterator i

**while**(**not** q.empty()):

        # Dequeue a vertex from queue

        # and print it

        u **=** q.queue[0]

**print**(u, end **=** " ")

        q.get()

        # Get all adjacent vertices of the

        # dequeued vertex s. If an adjacent

        # has not been visited, then mark

        # it visited and enqueue it

        i **=** 0

**while** i !**=** len(adj[u]):

**if** (**not** visited[adj[u][i]]):

                    visited[adj[u][i]] **=** True

                    q.put(adj[u][i])

            i **+=** 1

# This function does BFSUtil() for all

# unvisited vertices.

**def** BFS(adj, V):

    visited **=** [False] **\*** V

**for** u **in** range(V):

**if** (visited[u] **==** False):

            BFSUtil(u, adj, visited)

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    V **=** 5

    adj **=** [[] **for** i **in** range(V)]

    addEdge(adj, 0, 4)

    addEdge(adj, 1, 2)

    addEdge(adj, 1, 3)

    addEdge(adj, 1, 4)

    addEdge(adj, 2, 3)

    addEdge(adj, 3, 4)

    BFS(adj, V)

# This code is contributed by PranchalK

**Output**

0 4 1 2 3

 This article is contributed by [**Sahil Chhabra (akku)**](https://www.facebook.com/sahil.chhabra.965). If you like GeeksforGeeks and would like to contribute, you can also write an article using [write.geeksforgeeks.org](http://write.geeksforgeeks.org/) or mail your article to review-team@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

**Transitive Closure of a Graph using DFS**

Given a directed graph, find out if a vertex v is reachable from another vertex u for all vertex pairs (u, v) in the given graph. Here reachable means that there is a path from vertex u to v. The reach-ability matrix is called transitive closure of a graph.

For example, consider below graph

Transitive closure of above graphs is   
 1 1 1 1   
 1 1 1 1   
 1 1 1 1   
 0 0 0 1

We have discussed an O(V3) solution for this [here](https://www.geeksforgeeks.org/transitive-closure-of-a-graph/). The solution was based on [Floyd Warshall Algorithm](https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/). In this post, an O(V(V+E)) algorithm for the same is discussed. So for dense graph, it would become O(V3) and for sparse graph, it would become O(V2).

Below are the abstract steps of the algorithm.

1. Create a matrix tc[V][V] that would finally have transitive closure of the given graph. Initialize all entries of tc[][] as 0.
2. Call DFS for every node of the graph to mark reachable vertices in tc[][]. In recursive calls to DFS, we don’t call DFS for an adjacent vertex if it is already marked as reachable in tc[][].

Below is the implementation of the above idea. The code uses adjacency list representation of input graph and builds a matrix tc[V][V] such that tc[u][v] would be true if v is reachable from u.

**Implementation:**

# Python program to print transitive

# closure of a graph.

**from** collections **import** defaultdict

**class** Graph:

**def** \_\_init\_\_(self,vertices):

        # No. of vertices

        self.V **=** vertices

        # default dictionary to store graph

        self.graph **=** defaultdict(list)

        # To store transitive closure

        self.tc **=** [[0 **for** j **in** range(self.V)] **for** i **in** range(self.V)]

    # function to add an edge to graph

**def** addEdge(self, u, v):

        self.graph[u].append(v)

    # A recursive DFS traversal function that finds

    # all reachable vertices for s

**def** DFSUtil(self, s, v):

        # Mark reachability from s to v as true.

**if**(s **==** v):

**if**( v **in** self.graph[s]):

              self.tc[s][v] **=** 1

**else**:

            self.tc[s][v] **=** 1

        # Find all the vertices reachable through v

**for** i **in** self.graph[v]:

**if** self.tc[s][i] **==** 0:

**if** s**==**i:

                   self.tc[s][i]**=**1

**else**:

                   self.DFSUtil(s, i)

    # The function to find transitive closure. It uses

    # recursive DFSUtil()

**def** transitiveClosure(self):

        # Call the recursive helper function to print DFS

        # traversal starting from all vertices one by one

**for** i **in** range(self.V):

            self.DFSUtil(i, i)

        print(self.tc)

# Create a graph given in the above diagram

g **=** Graph(4)

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

g.transitiveClosure()

**Output**

Transitive closure matrix is   
1 1 1 1   
1 1 1 1   
1 1 1 1   
0 0 0 1

**Time Complexity : O(V^2)** where V is the number of vertexes .

**Space complexity : O(V^2)** where V is number of vertices.

*From <*[*https://www.geeksforgeeks.org/transitive-closure-of-a-graph-using-dfs/*](https://www.geeksforgeeks.org/transitive-closure-of-a-graph-using-dfs/)*>*

**Difference between BFS and DFS**

**Breadth-First Search:**

**BFS, Breadth-First Search,** is a vertex-based technique for finding the shortest path in the graph. It uses a [Queue data structure](https://www.geeksforgeeks.org/queue-data-structure/) that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

**Example**:

**Input:**  
 A  
 / \  
 B C  
 / / \  
 D E F

**Output:**

A, B, C, D, E, F

**Depth First Search:**

**DFS,** [**Depth First Search**](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/), is an edge-based technique. It uses the [Stack data structure](https://www.geeksforgeeks.org/stack-data-structure/) and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped.

**Example:**

**Input:**  
 A  
 / \  
 B D  
 / / \  
 C E F

**Output:**

A, B, C, D, E, F

**BFS vs DFS**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No.** | **Parameters** | **BFS** | **DFS** |
| **1.** | **Stands for** | **BFS stands for Breadth First Search.** | **DFS stands for Depth First Search.** |
| **2.** | **Data Structure** | **BFS(Breadth First Search) uses Queue data structure for finding the shortest path.** | **DFS(Depth First Search) uses Stack data structure.** |
| **3.** | **Definition** | **BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.** | **DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.** |
| **4.** | **Technique** | **BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex.** | **In DFS, we might traverse through more edges to reach a destination vertex from a source.** |
| **5.** | **Conceptual Difference** | **BFS builds the tree level by level.** | **DFS builds the tree sub-tree by sub-tree.** |
| **6.** | **Approach used** | **It works on the concept of FIFO (First In First Out).** | **It works on the concept of LIFO (Last In First Out).** |
| **7.** | **Suitable for** | **BFS is more suitable for searching vertices closer to the given source.** | **DFS is more suitable when there are solutions away from source.** |
| **8.** | **Suitability for Decision-Trees** | **BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles.** | **DFS is more suitable for game or puzzle problems. We make a decision, and the then explore all paths through this decision. And if this decision leads to win situation, we stop.** |
| **9.** | **Time Complexity** | **The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.** | **The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.** |
| **10.** | **Visiting of Siblings/ Children** | **Here, siblings are visited before the children.** | **Here, children are visited before the siblings.** |
| **11.** | **Removal of Traversed Nodes** | **Nodes that are traversed several times are deleted from the queue.** | **The visited nodes are added to the stack and then removed when there are no more nodes to visit.** |
| **12.** | **Backtracking** | **In BFS there is no concept of backtracking.** | **DFS algorithm is a recursive algorithm that uses the idea of backtracking** |
| **13.** | **Applications** | **BFS is used in various applications such as bipartite graphs, shortest paths, etc.** | **DFS is used in various applications such as acyclic graphs and topological order etc.** |
| **14.** | **Memory** | **BFS requires more memory.** | **DFS requires less memory.** |
| **15.** | **Optimality** | **BFS is optimal for finding the shortest path.** | **DFS is not optimal for finding the shortest path.** |
| **16.** | **Space complexity** | **In BFS, the space complexity is more critical as compared to time complexity.** | **DFS has lesser space complexity because at a time it needs to store only a single path from the root to the leaf node.** |
| **17.** | **Speed** | **BFS is slow as compared to DFS.** | **DFS is fast as compared to BFS.** |
| **18.** | **When to use?** | **When the target is close to the source, BFS performs better.** | **When the target is far from the source, DFS is preferable.** |

*From <*[*https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/*](https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/)*>*

**Easy Questions:**

**Find size of the largest region in Boolean Matrix**

Consider a matrix, where each cell contains either a ‘0’ or a ‘1’, and any cell containing a 1 is called a filled cell. Two cells are said to be connected if they are adjacent to each other horizontally, vertically, or diagonally. If one or more filled cells are also connected, they form a region. find the size of the largest region.

**Examples:**

***Input:****M[][5] = { {0, 0, 1, 1, 0}, {1, 0, 1, 1, 0}, {0, 1, 0, 0, 0}, {0, 0, 0, 0, 1}}*

***Output:****6*

***Explanation:****In the following example, there are 2 regions.*

*One with size 1 and the other as 6. So largest region: 6*

***Input:****M[][5] = { {0, 0, 1, 1, 0}, {0, 0, 1, 1, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 0 1} }*

***Output:****4*

***Explanation:****In the following example, there are 2 regions.*

*One with size 1 and the other as 4. So largest region: 4*

Recommended Problem

Unit Area of largest region of 1's

**Approach:** To solve the problem follow the below idea:

*The idea is based on the problem of*[*finding number of islands in Boolean 2D-matrix*](https://www.geeksforgeeks.org/find-number-of-islands/)

**Find the length of the largest region in Boolean Matrix using**[**DFS**](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/)**:**

Follow the given steps to solve the problem:

1. A cell in the 2D matrix can be connected to at most 8 neighbors.
2. So in DFS, make recursive calls for 8 neighbors of that cell.
3. Keep a visited Hash-map to keep track of all visited cells.
4. Also, keep track of the visited 1’s in every DFS and update the maximum size region.

Below is the implementation of the above approach:

# Python3 program to find the length of the

# largest region in boolean 2D-matrix

# A function to check if a given cell

# (row, col) can be included in DFS

**def** isSafe(M, row, col, visited):

**global** ROW, COL

    # row number is in range, column number is in

    # range and value is 1 and not yet visited

**return** ((row >**=** 0) **and** (row < ROW) **and**

            (col >**=** 0) **and** (col < COL) **and**

            (M[row][col] **and not** visited[row][col]))

# A utility function to do DFS for a 2D

# boolean matrix. It only considers

# the 8 neighbours as adjacent vertices

**def** DFS(M, row, col, visited, count):

    # These arrays are used to get row and column

    # numbers of 8 neighbours of a given cell

    rowNbr **=** [**-**1, **-**1, **-**1, 0, 0, 1, 1, 1]

    colNbr **=** [**-**1, 0, 1, **-**1, 1, **-**1, 0, 1]

    # Mark this cell as visited

    visited[row][col] **=** True

    # Recur for all connected neighbours

**for** k **in** range(8):

**if** (isSafe(M, row **+** rowNbr[k],

                   col **+** colNbr[k], visited)):

            # increment region length by one

            count[0] **+=** 1

            DFS(M, row **+** rowNbr[k],

                col **+** colNbr[k], visited, count)

# The main function that returns largest

# length region of a given boolean 2D matrix

**def** largestRegion(M):

**global** ROW, COL

    # Make a bool array to mark visited cells.

    # Initially all cells are unvisited

    visited **=** [[0] **\*** COL **for** i **in** range(ROW)]

    # Initialize result as 0 and traverse

    # through the all cells of given matrix

    result **= -**999999999999

**for** i **in** range(ROW):

**for** j **in** range(COL):

            # If a cell with value 1 is not

**if** (M[i][j] **and not** visited[i][j]):

                # visited yet, then new region found

                count **=** [1]

                DFS(M, i, j, visited, count)

                # maximum region

                result **=** max(result, count[0])

**return** result

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

  ROW **=** 4

  COL **=** 5

  M **=** [[0, 0, 1, 1, 0],

       [1, 0, 1, 1, 0],

       [0, 1, 0, 0, 0],

       [0, 0, 0, 0, 1]]

  # Function call

  print(largestRegion(M))

# This code is contributed by PranchalK

**Output**

6

**Time complexity:** O(ROW \* COL). In the worst case, all the cells will be visited so the time complexity is O(ROW \* COL).

**Auxiliary Space:**O(ROW \* COL). To store the visited nodes O(ROW \* COL) space is needed.

**Find the length of the largest region in Boolean Matrix using**[**BFS**](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)**:**

Follow the given steps to solve the problem:

1. If the value at any particular cell is 1 then from here we need to do the BFS traversal
2. Push the pair<i,j> in the queue
3. Marking the value 1 to -1 so that we don’t again push the same cell again
4. We will check in all 8 directions and if we encounter the cell having a value of 1 then we will push it into the queue and we will mark the cell to -1

Below is the implementation of the above approach:

**from** typing **import** List, Tuple

**from** collections **import** deque

**def** largestRegion(grid: List[List[int]]) **-**> int:

    m **=** len(grid)

    n **=** len(grid[0])

    # creating a queue that will help in bfs traversal

    q **=** deque()

    area **=** 0

    ans **=** 0

**for** i **in** range(m):

**for** j **in** range(n):

            # if the value at any particular cell is 1 then

            # from here we need to do the BFS traversal

**if** grid[i][j] **==** 1:

                ans **=** 0

                # pushing the pair(i,j) in the queue

                q.append((i, j))

                # marking the value 1 to -1 so that we

                # don't again push this cell in the queue

                grid[i][j] **= -**1

**while** len(q) > 0:

                    t **=** q.popleft()

                    ans **+=** 1

                    x, y **=** t[0], t[1]

                    # now we will check in all 8 directions

**if** x **+** 1 < m:

**if** grid[x **+** 1][y] **==** 1:

                            q.append((x **+** 1, y))

                            grid[x **+** 1][y] **= -**1

**if** x **-** 1 >**=** 0:

**if** grid[x **-** 1][y] **==** 1:

                            q.append((x **-** 1, y))

                            grid[x **-** 1][y] **= -**1

**if** y **+** 1 < n:

**if** grid[x][y **+** 1] **==** 1:

                            q.append((x, y **+** 1))

                            grid[x][y **+** 1] **= -**1

**if** y **-** 1 >**=** 0:

**if** grid[x][y **-** 1] **==** 1:

                            q.append((x, y **-** 1))

                            grid[x][y **-** 1] **= -**1

**if** x **+** 1 < m **and** y **+** 1 < n:

**if** grid[x **+** 1][y **+** 1] **==** 1:

                            q.append((x **+** 1, y **+** 1))

                            grid[x **+** 1][y **+** 1] **= -**1

**if** x **-** 1 >**=** 0 **and** y **+** 1 < n:

**if** grid[x **-** 1][y **+** 1] **==** 1:

                            q.append((x **-** 1, y **+** 1))

                            grid[x **-** 1][y **+** 1] **= -**1

**if** x **-** 1 >**=** 0 **and** y **-** 1 >**=** 0:

**if** grid[x **-** 1][y **-** 1] **==** 1:

                            q.append((x **-** 1, y **-** 1))

                            grid[x **-** 1][y **-** 1] **= -**1

**if** x **+** 1 < m **and** y **-** 1 >**=** 0:

**if** grid[x **+** 1][y **-** 1] **==** 1:

                            q.append((x **+** 1, y **-** 1))

                            grid[x **+** 1][y **-** 1] **= -**1

                area **=** max(area, ans)

**return** area

**def** main():

    grid **=** [

        [0, 0, 1, 1, 0],

        [1, 0, 1, 1, 0],

        [0, 1, 0, 0, 0],

        [0, 0, 0, 0, 1]

    ]

    result **=** largestRegion(grid)

    print(f'Largest region of 1s has an area of {result}')

main()

**Output**

6

**Time complexity:** O(ROW \* COL). In the worst case, all the cells will be visited so the time complexity is O(ROW \* COL).

**Auxiliary Space:** O(ROW \* COL). To store the visited nodes O(ROW \* COL) space is needed.

*From <*[*https://www.geeksforgeeks.org/find-length-largest-region-boolean-matrix/*](https://www.geeksforgeeks.org/find-length-largest-region-boolean-matrix/)*>*

**Count number of trees in a forest**

Given n nodes of a forest (collection of trees), find the number of trees in the forest.

**Examples :**

**Input :** edges[] = {0, 1}, {0, 2}, {3, 4}  
**Output :** 2  
**Explanation :** There are 2 trees  
 0 3  
 / \ \  
 1 2 4

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

**Approach :**

1. Apply DFS on every node.
2. Increment count by one if every connected node is visited from one source.
3. Again perform DFS traversal if some nodes yet not visited.
4. Count will give the number of trees in forest.

**Implementation:**

# Python3 program to count number

# of trees in a forest.

# A utility function to add an

# edge in an undirected graph.

**def** addEdge(adj, u, v):

    adj[u].append(v)

    adj[v].append(u)

# A utility function to do DFS of graph

# recursively from a given vertex u.

**def** DFSUtil(u, adj, visited):

    visited[u] **=** True

**for** i **in** range(len(adj[u])):

**if** (visited[adj[u][i]] **==** False):

            DFSUtil(adj[u][i], adj, visited)

# Returns count of tree is the

# forest given as adjacency list.

**def** countTrees(adj, V):

    visited **=** [False] **\*** V

    res **=** 0

**for** u **in** range(V):

**if** (visited[u] **==** False):

            DFSUtil(u, adj, visited)

            res **+=** 1

**return** res

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    V **=** 5

    adj **=** [[] **for** i **in** range(V)]

    addEdge(adj, 0, 1)

    addEdge(adj, 0, 2)

    addEdge(adj, 3, 4)

    print(countTrees(adj, V))

# This code is contributed by PranchalK

**Output**

2

**Time Complexity :** **O(V + E),**where V is the number of vertices and E is the number of edges.

**Space Complexity: O(V)**. We use an array of size V to store the visited nodes.

**Approach** :- Here’s an implementation of counting the number of trees in a forest using BFS in C++

    Define a Node type using a typedef statement to represent a node in the forest. A Node is a pair of integers that represents the row and column indices of a node in the forest.

   Define a bfs function that takes the forest, a start node, and a visited array as inputs. The function performs BFS starting from the start node and marks all visited nodes in the visited array.

   Inside the bfs function, create a queue q to store the nodes that are to be visited in the BFS. Initially, push the start node onto the queue and mark it as visited in the visited array.

#include <iostream>

#include <queue>

#include <vector>

**using namespace** std;

// define a pair to represent a node in the forest

**typedef** pair<**int**, **int**> Node;

// function to perform BFS from a given node and mark all visited nodes

**void** bfs(vector<vector<**int**>>& forest, Node start, vector<vector<**bool**>>& visited) {

    // create a queue for BFS

    queue<Node> q;

    q.push(start);

    visited[start.first][start.second] = **true**;

    // BFS loop

**while** (!q.empty()) {

        Node curr = q.front();

        q.pop();

        // add unvisited neighboring nodes to the queue

**int** dx[] = {-1, 0, 1, 0};

**int** dy[] = {0, 1, 0, -1};

**for** (**int** i = 0; i < 4; i++) {

**int** nx = curr.first + dx[i];

**int** ny = curr.second + dy[i];

**if** (nx >= 0 && nx < forest.size() && ny >= 0 && ny < forest[0].size() && forest[nx][ny] == 1 && !visited[nx][ny]) {

                q.push(make\_pair(nx, ny));

                visited[nx][ny] = **true**;

            }

        }

    }

}

// function to count the number of trees in a forest using BFS

**int** count\_trees\_in\_forest(vector<vector<**int**>>& forest) {

**int** count = 0;

**int** n = forest.size();

**int** m = forest[0].size();

    // create a 2D boolean array to keep track of visited nodes

    vector<vector<**bool**>> visited(n, vector<**bool**>(m, **false**));

    // iterate over all nodes in the forest and perform BFS from each unvisited tree

**for** (**int** i = 0; i < n; i++) {

**for** (**int** j = 0; j < m; j++) {

**if** (forest[i][j] == 1 && !visited[i][j]) {

                bfs(forest, make\_pair(i, j), visited);

                count++;

            }

        }

    }

**return** count;

}

**int** main() {

    // example usage

    vector<vector<**int**>> forest = {

        {0, 1, 1, 0, 0},

        {0, 0, 0, 0, 0},

        {0, 0, 0, 0, 0},

        {0, 0, 0, 0, 1},

        {0, 0, 0, 0, 0}

    };

**int** num\_trees = count\_trees\_in\_forest(forest);

    cout << "The forest has " << num\_trees << " trees." << endl;

**return** 0;

}

**Output**

The forest has 2 trees.

Time complexity : – O(NM)

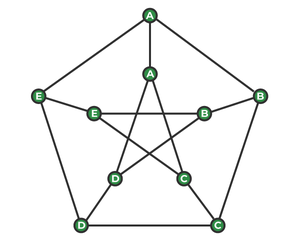
Auxiliary Space :- O(MN)

**A Peterson Graph Problem**

The following graph G is called a Petersen graph and its vertices have been numbered from 0 to 9. Some letters have also been assigned to vertices of G, as can be seen from the following picture:

Let’s consider a walk W in graph G, which consists of L vertices W1, W2, …, WL. A string S of L letters ‘A’ – ‘E’ is realized by walking W if the sequence of letters written along W is equal to S. Vertices can be visited multiple times while walking along W.

For example, S = ‘ABBECCD’ is realized by W = (0, 1, 6, 9, 7, 2, 3). Determine whether there is a walk W that realizes a given string S in graph G and if so then find the lexicographically least such walk. The only line of input contains one string S. If there is no walk W which realizes S, then output -1 otherwise, you should output the least lexicographical walk W which realizes S.



*Example of a Petersen Graph*

**Examples:**

Input : s = 'ABB'  
Output: 016  
Explanation: As we can see in the graph  
 the path from ABB is 016.  
Input : s = 'AABE'  
Output :-1  
Explanation: As there is no path that  
 exists, hence output is -1.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

**Algorithm for a Peterson Graph Problem:**

**petersonGraphWalk(S, v):**

*begin*

*res := starting vertex*

*for each character c in S except the first one, do*

*if there is an edge between v and c in outer graph, then*

*v := c*

*else if there is an edge between v and c+5 in inner graph, then*

*v := c + 5*

*else*

*return false*

*end if*

*put v into res*

*done*

*return true*

*end*

Below is the implementation of the above algorithm:

1. C++
2. Java
3. Python3
4. C#
5. Javascript

# Python3 program to find the

# path in Peterson graph

# path to be checked

# adjacency matrix.

adj **=** [[False **for** i **in** range(10)] **for** j **in** range(10)]

# resulted path - way

result **=** [0]

# we are applying breadth first search

# here

**def** findthepath(S, v):

    result[0] **=** v

**for** i **in** range(1, len(S)):

        # first traverse the outer graph

**if** (adj[v][ord(S[i]) **-** ord('A')] **or**

            adj[ord(S[i]) **-** ord('A')][v]):

            v **=** ord(S[i]) **-** ord('A')

        # then traverse the inner graph

**else if** (adj[v][ord(S[i]) **-** ord('A') **+** 5] **or**

               adj[ord(S[i]) **-** ord('A') **+** 5][v]):

            v **=** ord(S[i]) **-** ord('A') **+** 5

        # if the condition failed to satisfy

        # return false

**else**:

**return** False

        result.append(v)

**return** True

# driver code

# here we have used adjacency matrix to make

# connections between the connected nodes

adj[0][1] **=** adj[1][2] **=** adj[2][3] **=** \

adj[3][4] **=** adj[4][0] **=** adj[0][5] **=** \

adj[1][6] **=** adj[2][7] **=** adj[3][8] **=** \

adj[4][9] **=** adj[5][7] **=** adj[7][9] **=** \

adj[9][6] **=** adj[6][8] **=** adj[8][5] **=** True

# path to be checked

S**=** "ABB"

S**=**list(S)

**if** (findthepath(S, ord(S[0]) **-** ord('A')) **or**

    findthepath(S, ord(S[0]) **-** ord('A') **+** 5)):

    print(**\***result, sep **=** "")

**else**:

**print**("-1")

# This code is contributed by SHUBHAMSINGH10

**Output**

016

**Time complexity: O(N)**

The time complexity of the above program is O(N), where N is the length of the given string S. We are applying Breadth First Search here, which runs in linear time.

**Space complexity: O(N)**

The space complexity of the above program is O(N), where N is the length of the given string S. We are using two auxiliary arrays – result[] and S[] to store the path and the given string, respectively. Both of them require linear space.

**Clone an Undirected Graph**

Cloning of a [LinkedList](https://www.geeksforgeeks.org/clone-linked-list-next-arbit-pointer-set-2/)and a [Binary Tree](https://www.geeksforgeeks.org/clone-binary-tree-random-pointers/)with random pointers has already been discussed. The idea behind cloning a graph is pretty much similar.

Clone Graph

The idea is to do a [BFS traversal](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/)of the graph and while visiting a node make a clone node of it (a copy of original node). If a node is encountered which is already visited then it already has a clone node.

**How to keep track of the visited/cloned nodes?** A HashMap/Map is required in order to maintain all the nodes which have already been created. *Key stores*: Reference/Address of original Node *Value stores*: Reference/Address of cloned Node A copy of all the graph nodes has been made,

**how to connect clone nodes?** While visiting the neighboring vertices of a node *u*get the corresponding cloned node for u , let’s call that *cloneNodeU*, now visit all the neighboring nodes for *u*and for each neighbor find the corresponding clone node(if not found create one) and then push into the neighboring vector of *cloneNodeU*node.

**How to verify if the cloned graph is a correct?** Do a BFS traversal before and after the cloning of graph. In BFS traversal display the value of a node along with its address/reference. Compare the order in which nodes are displayed, if the values are same but the address/reference is different for both the traversals then the cloned graph is correct.

**Implementation:**

**from** collections **import** deque

**class** GraphNode:

**def** \_\_init\_\_(self, val**=**0, neighbors**=**[]):

        self.val **=** val

        self.neighbors **=** neighbors

**def** cloneGraph(src: GraphNode) **-**> GraphNode:

    # A Map to keep track of all the

    # nodes which have already been created

    m **=** {}

    q **=** deque()

    # Enqueue src node

    q.append(src)

    node **=** None

    # Make a clone Node

    node **=** GraphNode()

    node.val **=** src.val

    # Put the clone node into the Map

    m[src] **=** node

**while** q:

        # Get the front node from the queue

        # and then visit all its neighbors

        u **=** q.popleft()

        v **=** u.neighbors

**for** neighbor **in** v:

            # Check if this node has already been created

**if** neighbor **not in** m:

                # If not then create a new Node and

                # put into the HashMap

                node **=** GraphNode()

                node.val **=** neighbor.val

                m[neighbor] **=** node

                q.append(neighbor)

            # Add these neighbors to the cloned graph node

            m[u].neighbors.append(m[neighbor])

    # Return the address of cloned src Node

**return** m[src]

# Build the desired graph

**def** buildGraph() **-**> GraphNode:

    """

    Given Graph:

    1--2

    | |

    4--3

    """

    node1 **=** GraphNode(1)

    node2 **=** GraphNode(2)

    node3 **=** GraphNode(3)

    node4 **=** GraphNode(4)

    node1.neighbors **=** [node2, node4]

    node2.neighbors **=** [node1, node3]

    node3.neighbors **=** [node2, node4]

    node4.neighbors **=** [node3, node1]

**return** node1

# A simple bfs traversal of a graph to

# check for proper cloning of the graph

**def** bfs(src: GraphNode):

    visit **=** {}

    q **=** deque()

    q.append(src)

    visit[src] **=** True

**while** q:

        u **=** q.popleft()

**print**(f"Value of Node {u.val}")

        print(f"Address of Node {u}")

        v **=** u.neighbors

**for** neighbor **in** v:

**if** neighbor **not in** visit:

                visit[neighbor] **=** True

                q.append(neighbor)

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    src **=** buildGraph()

**print**("BFS Traversal before cloning")

    bfs(src)

    clone **=** cloneGraph(src)

    print("\nBFS Traversal after cloning")

    bfs(clone)

    # This code is contributed by vikramshirsath177

**Output**

BFS Traversal before cloning  
Value of Node 1  
Address of Node 0x1b6ce70  
Value of Node 2  
Address of Node 0x1b6cea0  
Value of Node 4  
Address of Node 0x1b6cf00  
Value of Node 3  
Address of Node 0x1b6ced0

BFS Traversal after cloning  
Value of Node 1  
Address of Node 0x1b6e5a0  
Value of Node 2  
Address of Node 0x1b6e5d0  
Value of Node 4  
Address of Node 0x1b6e620  
Value of Node 3  
Address of Node 0x1b6e670

**Time Complexity:** **O(V+E)** where V is the number of vertices and E is the number of edges in the graph.

**Auxiliary Space:** **O(V),** since a map is used to store the graph nodes which can grow upto V.

**Graph Coloring | Set 1 (Introduction and Applications)**

[Graph coloring](http://en.wikipedia.org/wiki/Graph_coloring) problem is to assign colors to certain elements of a graph subject to certain constraints.

**Vertex coloring** is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using same color. The other graph coloring problems like ***Edge Coloring*** (No vertex is incident to two edges of same color) and ***Face Coloring***(Geographical Map Coloring) can be transformed into vertex coloring.

**Algorithm for graph coloring**

Algorithm GRAPH COLORING(G, COLOR, i)

Description: Solve the graph coloring problem using backtracking   
//Input: Graph G with n vertices, list of colors, initial vertex i  
COLOR(1...n] is the array of n different colors

//Output: Colored graph with minimum color  
if CHECK\_VERTEX(i)==1 then  
if i == N then

print COLOR[1...n]  
else  
j <- 1  
while (S<=M) do  
COLOR(i+1) <- j  
j +j <- 1  
end

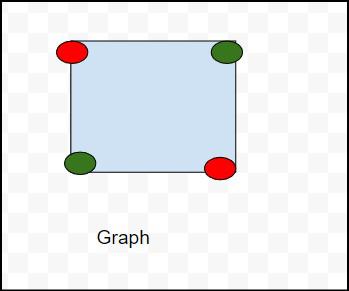
end  
end

Function CHECK\_VERTEX(i)  
for j <- 1to i -1 do  
if Adjacent(i,j) then  
if COLOR(i)==COLOR(j) then  
return 0  
end

end

end  
return 1

**Chromatic Number:** The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored minimum 2 colors.



Chromatic number of this graph is 2 because  in this above diagram we can use to color red and green .

so chromatic number of this graph is 2 and is denoted x(G)  ,means x(G)=2 .

Chromatic number define as the least no of colors needed for coloring the graph .

and types of chromatic number are:

1) Cycle graph

2) planar graphs

3) Complete graphs

4) Bipartite Graphs:

5) Trees

 The problem to find chromatic number of a given graph is [NP Complete](https://www.geeksforgeeks.org/np-completeness-set-1/).

The chromatic number is denoted by X(G). Finding the chromatic number for the graph is NP-complete problem. Graph coloring problem is both, a decision problem as well as an optimization problem. A decision problem is stated as, “With given M colors and graph G, whether a such color scheme is possible or not?”.

The optimization problem is stated as, “Given M colors and graph G, find the minimum number of colors required for graph coloring.” Graph coloring problem is a very interesting problem of graph theory and it has many diverse applications.

**Applications of Graph Coloring:**

The graph coloring problem has huge number of applications.

***1) Making Schedule or Time Table:***Suppose we want to make am exam schedule for a university. We have list different subjects and students enrolled in every subject. Many subjects would have common students (of same batch, some backlog students, etc). *How do we schedule the exam so that no two exams with a common student are scheduled at same time? How many minimum time slots are needed to schedule all exams?* This problem can be represented as a graph where every vertex is a subject and an edge between two vertices mean there is a common student. So this is a graph coloring problem where minimum number of time slots is equal to the chromatic number of the graph.

***2)***[***Mobile Radio Frequency Assignment***](http://www.zib.de/groetschel/teaching/SS2012/GraphCol%20and%20FrequAssignment.pdf)***:*** When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the minimum number of frequencies needed? This problem is also an instance of graph coloring problem where every tower represents a vertex and an edge between two towers represents that they are in range of each other.

***3) Sudoku:***Sudoku is also a variation of Graph coloring problem where every cell represents a vertex. There is an edge between two vertices if they are in same row or same column or same block.

***4)***[***Register Allocation***](http://en.wikipedia.org/wiki/Register_allocation)***:***In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers. This problem is also a graph coloring problem.

***5) Bipartite Graphs:***We can check if a graph is Bipartite or not by coloring the graph using two colors. If a given graph is 2-colorable, then it is Bipartite, otherwise not. See [this](https://www.geeksforgeeks.org/bipartite-graph/)for more details.

***6) Map Coloring:***Geographical maps of countries or states where no two adjacent cities cannot be assigned same color. Four colors are sufficient to color any map (See [Four Color Theorem](http://en.wikipedia.org/wiki/Four_color_theorem))

**There can be many more applications:** For example the below reference video lecture has a case study at 1:18.

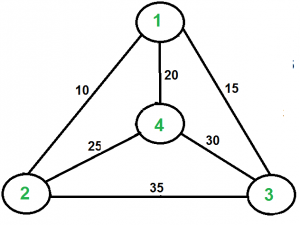
**Traveling Salesman Problem (TSP) Implementation**

[Travelling Salesman Problem (TSP) :](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/) Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between [Hamiltonian Cycle](https://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/) and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in the figure on the right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP-hard problem. There is no polynomial-time known solution for this problem.



**Examples:**

Output of Given Graph:  
minimum weight Hamiltonian Cycle :  
10 + 25 + 30 + 15 := 80

[Recommended: Please try your approach on ***{Practice}*** first, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/travelling-salesman-problem2732/1)

In this post, the implementation of a simple solution is discussed.

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting point.
2. Generate all (n-1)! permutations of cities.
3. Calculate the cost of every permutation and keep track of the minimum cost permutation.
4. Return the permutation with minimum cost.

Below is the implementation of the above idea

# Python3 program to implement traveling salesman

# problem using naive approach.

**from** sys **import** maxsize

**from** itertools **import** permutations

V **=** 4

# implementation of traveling Salesman Problem

**def** travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex

    vertex **=** []

**for** i **in** range(V):

**if** i !**=** s:

            vertex.append(i)

    # store minimum weight Hamiltonian Cycle

    min\_path **=** maxsize

    next\_permutation**=**permutations(vertex)

**for** i **in** next\_permutation:

        # store current Path weight(cost)

        current\_pathweight **=** 0

        # compute current path weight

        k **=** s

**for** j **in** i:

            current\_pathweight **+=** graph[k][j]

            k **=** j

        current\_pathweight **+=** graph[k][s]

        # update minimum

        min\_path **=** min(min\_path, current\_pathweight)

**return** min\_path

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # matrix representation of graph

    graph **=** [[0, 10, 15, 20], [10, 0, 35, 25],

            [15, 35, 0, 30], [20, 25, 30, 0]]

    s **=** 0

    print(travellingSalesmanProblem(graph, s))

**Output**

80

**Time complexity:**  O(n!) where n is the number of vertices in the graph. This is because the algorithm uses the next\_permutation function which generates all the possible permutations of the vertex set.

**Auxiliary Space:**O(n) as we are using a vector to store all the vertices.

**Introduction and Approximate Solution for Vertex Cover Problem**

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either ‘u’ or ‘v’ is in the vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. ***Given an undirected graph, the vertex cover problem is to find minimum size vertex cover***.

The following are some examples.

VertexCover

[Vertex Cover Problem](http://en.wikipedia.org/wiki/Vertex_cover) is a known [NP Complete problem](https://www.geeksforgeeks.org/np-completeness-set-1/), i.e., there is no polynomial-time solution for this unless P = NP. There are approximate polynomial-time algorithms to solve the problem though. Following is a simple approximate algorithm adapted from [CLRS book](http://www.flipkart.com/introduction-algorithms-english-3rd/p/itmdwxyrafdburzg?pid=9788120340077&affid=sandeepgfg).

**Naive Approach:**

Consider all the subset of vertices one by one and find out whether it covers all edges of the graph. For eg. in a graph consisting only 3 vertices the set consisting of the combination of vertices are:{0,1,2,{0,1},{0,2},{1,2},{0,1,2}} . Using each element of this set check whether these vertices cover all the edges of the graph. Hence update the optimal answer. And hence print the subset having minimum number of vertices which also covers all the edges of the graph.

**Approximate Algorithm for Vertex Cover:**

1) Initialize the result as {}  
2) Consider a set of all edges in given graph. Let the set be E.  
3) Do following while E is not empty  
...a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result  
...b) Remove all edges from E which are either incident on u or v.  
4) Return result

Below diagram to show the execution of the above approximate algorithm:

vertexCover

**How well the above algorithm perform?**

It can be proved that the above approximate algorithm never finds a vertex cover whose size is more than twice the size of the minimum possible vertex cover (Refer [this](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/AproxAlgor/vertexCover.htm)for proof)

**Implementation:**

The following are C++ and Java implementations of the above approximate algorithm.

# Python3 program to print Vertex Cover

# of a given undirected graph

**from** collections **import** defaultdict

# This class represents a directed graph

# using adjacency list representation

**class** Graph:

**def** \_\_init\_\_(self, vertices):

        # No. of vertices

        self.V **=** vertices

        # Default dictionary to store graph

        self.graph **=** defaultdict(list)

    # Function to add an edge to graph

**def** addEdge(self, u, v):

        self.graph[u].append(v)

    # The function to print vertex cover

**def** printVertexCover(self):

        # Initialize all vertices as not visited.

        visited **=** [False] **\*** (self.V)

        # Consider all edges one by one

**for** u **in** range(self.V):

            # An edge is only picked when

            # both visited[u] and visited[v]

            # are false

**if not** visited[u]:

                # Go through all adjacents of u and

                # pick the first not yet visited

                # vertex (We are basically picking

                # an edge (u, v) from remaining edges.

**for** v **in** self.graph[u]:

**if not** visited[v]:

                        # Add the vertices (u, v) to the

                        # result set. We make the vertex

                        # u and v visited so that all

                        # edges from/to them would

                        # be ignored

                        visited[v] **=** True

                        visited[u] **=** True

**break**

        # Print the vertex cover

**for** j **in** range(self.V):

**if** visited[j]:

                print(j, end **=** ' ')

        print()

# Driver code

# Create a graph given in

# the above diagram

g **=** Graph(7)

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(3, 4)

g.addEdge(4, 5)

g.addEdge(5, 6)

g.printVertexCover()

# This code is contributed by Prateek Gupta

Output:

0 1 3 4 5 6

The **Time Complexity** of the above algorithm is **O(V + E)**.

The **space complexity** of this solution is **O(V)**, where V is the number of vertices of the graph. This is because we are using an array of size V to store the visited vertices.

**Exact Algorithms:**

Although the problem is NP complete, it can be solved in polynomial time for the following types of graphs.

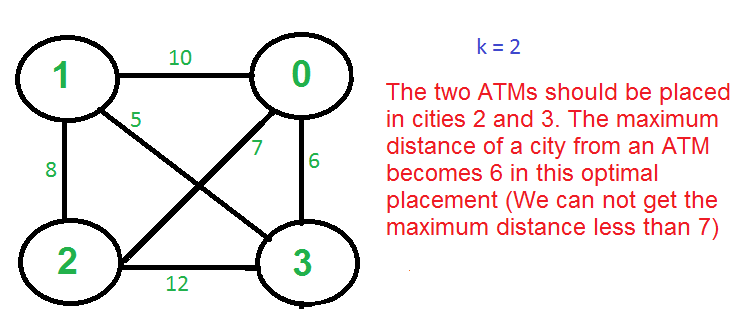
1) [Bipartite Graph](https://www.geeksforgeeks.org/bipartite-graph/)

2) [Tree Graph](https://www.geeksforgeeks.org/check-given-graph-tree/)

**Greedy Approximate Algorithm for K Centers Problem**

Given n cities and distances between every pair of cities, select k cities to place warehouses (or ATMs or Cloud Server) such that the maximum distance of a city to a warehouse (or ATM or Cloud Server) is minimized.

For example consider the following four cities, 0, 1, 2, and 3, and the distances between them, how to place 2 ATMs among these 4 cities so that the maximum distance of a city to an ATM is minimized.



There is no polynomial-time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial-time Greedy approximate algorithm, the greedy algorithm provides a solution that is never worse than twice the optimal solution. The greedy solution works only if the distances between cities follow [Triangular Inequality](http://en.wikipedia.org/wiki/Triangle_inequality) (The distance between two points is always smaller than the sum of distances through a third point).

***The 2-Approximate Greedy Algorithm:***

1. *Choose the first center arbitrarily.*
2. *Choose remaining k-1 centers using the following criteria.*

* *Let c1, c2, c3, … ci be the already chosen centers. Choose*
* *(i+1)’th center by picking the city which is farthest from already*
* *selected centers, i.e, the point p which has following value as maximum*
* *Min[dist(p, c1), dist(p, c2), dist(p, c3), …. dist(p, ci)]*

greedyAlgo

**Example (k = 3 in the above-shown Graph):**

1. *Let the first arbitrarily picked vertex be 0.*
2. *The next vertex is 1 because 1 is the farthest vertex from 0.*
3. *Remaining cities are 2 and 3. Calculate their distances from already selected centers (0 and 1). The greedy algorithm basically calculates the following values.*

* *Minimum of all distanced from 2 to already considered centers*
* *Min[dist(2, 0), dist(2, 1)] = Min[7, 8] = 7*
* *Minimum of all distanced from 3 to already considered centers*
* *Min[dist(3, 0), dist(3, 1)] = Min[6, 5] = 5*
* *After computing the above values, city 2 is picked as the value corresponding to 2 is maximum.*

Note that the greedy algorithm doesn’t give the best solution for k = 2 as this is just an approximate algorithm with a bound as twice optimal.

**Proof that the above greedy algorithm is 2 approximate.**

*Let OPT be the maximum distance of a city from a center in the Optimal solution. We need to show that the maximum distance obtained from the Greedy algorithm is 2\*OPT.*

*The proof can be done using contradiction.*

1. *Assume that the distance from the furthest point to all centers is > 2·OPT.*
2. *This means that distances between all centers are also > 2·OPT.*
3. *We have k + 1 points with distances > 2·OPT between every pair.*
4. *Each point has a center of the optimal solution with distance <= OPT to it.*
5. There exists a pair of points with the same center X in the optimal solution (pigeonhole principle: k optimal centers, k+1 points)
6. The distance between them is at most 2·OPT (triangle inequality) which is a contradiction.

**Implementation:**

# Python3 program for the above approach

**def** maxindex(dist, n):

    mi **=** 0

**for** i **in** range(n):

**if** (dist[i] > dist[mi]):

            mi **=** i

**return** mi

**def** selectKcities(n, weights, k):

    dist **=** [0]**\***n

    centers **=** []

**for** i **in** range(n):

        dist[i] **=** 10**\*\***9

    # index of city having the

    # maximum distance to it's

    # closest center

    max **=** 0

**for** i **in** range(k):

        centers.append(max)

**for** j **in** range(n):

            # updating the distance

            # of the cities to their

            # closest centers

            dist[j] **=** min(dist[j], weights[max][j])

        # updating the index of the

        # city with the maximum

        # distance to it's closest center

        max **=** maxindex(dist, n)

    # Printing the maximum distance

    # of a city to a center

    # that is our answer

    # print()

**print**(dist[max])

    # Printing the cities that

    # were chosen to be made

    # centers

**for** i **in** centers:

**print**(i, end **=** " ")

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    n **=** 4

    weights **=** [ [ 0, 4, 8, 5 ],

              [ 4, 0, 10, 7 ],

              [ 8, 10, 0, 9 ],

              [ 5, 7, 9, 0 ] ]

    k **=** 2

    # Function Call

    selectKcities(n, weights, k)

# This code is contributed by mohit kumar 29.

**Output**

5  
0 2

**Time Complexity:**O(n\*k), as we are using nested loops to traverse n\*k times.

**Auxiliary Space:**O(k), as we are using extra space for the array **center.**

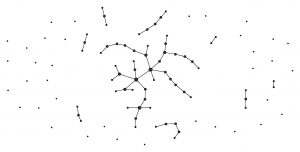
**Erdos Renyl Model (for generating Random Graphs)**

In graph theory, the Erdos–Rényi model is either of two closely related models for generating random graphs.

There are two closely related variants of the Erdos–Rényi (ER) random graph model.

In the G(n, M) model, a graph is chosen uniformly at random from the collection of all graphs which have n nodes and M edges. For example, in the G(3, 2) model, each of the three possible graphs on three vertices and two edges are included with probability 1/3.

In the G(n, p) model, a graph is constructed by connecting nodes randomly. Each edge is included in the graph with probability p independent from every other edge. Equivalently, all graphs with n nodes and M edges have equal probability of



 A graph generated by the binomial model of Erdos and Rényi (p = 0.01)

The parameter p in this model can be thought of as a weighting function; as p increases from 0 to 1, the model becomes more and more likely to include graphs with more edges and less and less likely to include graphs with fewer edges. In particular, the case p = 0.5 corresponds to the case where all

graphs on n vertices are chosen with equal probability.

The article will basically deal with the G (n,p) model where n is the no of nodes to be created and p defines the probability of joining of each node to the other.

**Properties of G(n, p)**

With the notation above, a graph in G(n, p) has on average

edges. The distribution of the degree of any particular vertex is binomial:

Where n is the total number of vertices in the graph.

Since

as

and np= constant This distribution is Poisson for large n and np = const. In a 1960 paper, Erdos and Rényi described the behaviour of G(n, p) very precisely for various values of p. Their results included that:

1. If np < 1, then a graph in G(n, p) will almost surely have no connected components of size larger than O(log(n)).
2. If np = 1, then a graph in G(n, p) will almost surely have a largest component whose size is of order

.

1. If np

c > 1, where c is a constant, then a graph in G(n, p) will almost surely have a unique giant component containing a positive fraction of the vertices. No other component will contain more than O(log(n)) vertices.

1. If

, then a graph in G(n, p) will almost surely contain isolated vertices, and thus be disconnected.

1. If

, then a graph in G(n, p) will almost surely be connected.

Thus

is a sharp threshold for the connectedness of G(n, p). Further properties of the graph can be described almost precisely as n tends to infinity. For example, there is a k(n) (approximately equal to 2log2(n)) such that the largest clique in G(n, 0.5) has almost surely either size k(n) or k(n) + 1. Thus, even though finding the size of the largest clique in a graph is NP-complete, the size of the largest clique in a “typical” graph (according to this model) is very well understood. Interestingly, edge-dual graphs of Erdos-Renyi graphs are graphs with nearly the same degree distribution, but with degree correlations and a significantly higher clustering coefficient.

Next I’ll describe the code to be used for making the ER graph. For implementation of the code below, you’ll need to install the netwrokx library as well you’ll need to install the matplotlib library. Following you’ll see the exact code of the graph which has been used as a function of the networkx library lately in this article.

**Erdos\_renyi\_graph(n, p, seed=None, directed=False)**

Returns a G(n,p) random graph, also known as an Erdos-Rényi graph or a binomial graph.

The G(n,p) model chooses each of the possible edges with probability p. The functions binomial\_graph() and erdos\_renyi\_graph() are aliases of this function.

*Parameters: n (int) – The number of nodes.*

*p (float) – Probability for edge creation.*

*seed (int, optional) – Seed for random number generator (default=None).*

*directed (bool, optional (default=False)) – If True, this function returns a directed graph.*

1. Python

#importing the networkx library

>>> **import** networkx as nx

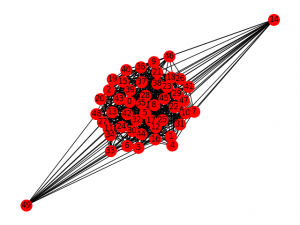
#importing the matplotlib library for plotting the graph

>>> **import** matplotlib.pyplot as plt

>>> G**=** nx.erdos\_renyi\_graph(50,0.5)

>>> nx.draw(G, with\_labels**=**True)

>>> plt.show()



*Figure 1: For n=50, p=0.5*

The above example is for 50 nodes and is thus a bit unclear.

When considering the case for lesser no of nodes (for example 10), you can clearly see the difference.

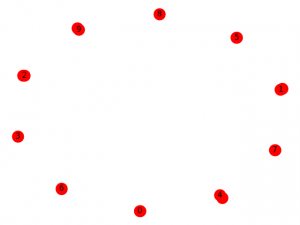
Using the codes for various probabilities, we can see the difference easily:

1. Python

>>> I**=** nx.erdos\_renyi\_graph(10,0)

>>> nx.draw(I, with\_labels**=**True)

>>> plt.show()



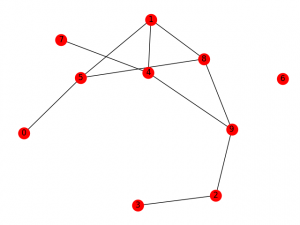
*Figure 2: For n=10, p=0*

1. Python

>>> K**=**nx.erdos\_renyi\_graph(10,0.25)

>>> nx.draw(K, with\_labels**=**True)

>>> plt.show()



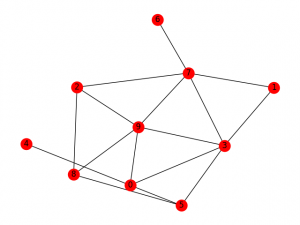
*Figure 3: For n=10, p=0.25*

1. Python

>>>H**=** nx.erdos\_renyi\_graph(10,0.5)

>>> nx.draw(H, with\_labels**=**True)

>>> plt.show()



*Figure 4: For n=10, p=0.5*

This algorithm runs in O(

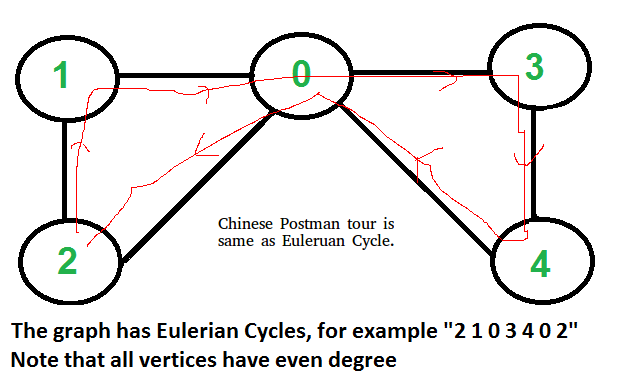
) time. For sparse graphs (that is, for small values of p), fast\_gnp\_random\_graph() is a faster algorithm. Thus the above examples clearly define the use of erdos renyi model to make random graphs and how to use the foresaid using the networkx library of python. Next we will discuss the ego graph and various other types of graphs in python using the library networkx.

**Chinese Postman or Route Inspection | Set 1 (introduction)**

[Chinese Postman Problem](https://en.wikipedia.org/wiki/Route_inspection_problem) is a variation of [Eulerian circuit](https://www.geeksforgeeks.org/eulerian-path-and-circuit/) problem for undirected graphs. An Euler Circuit is a closed walk that covers every edge once starting and ending position is same. Chinese Postman problem is defined for connected and undirected graph. The problem is to find shortest path or circuity that visits every edge of the graph at least once.

**If input graph contains Euler Circuit, then a solution of the problem is Euler Circuit**

An undirected and connected graph has Eulerian cycle if “all vertices have even degree“.

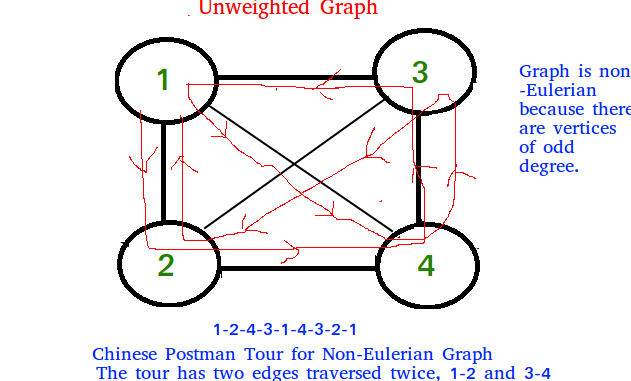


It doesn’t matter whether graph is weighted or unweighted, the Chinese Postman Route is always same as Eulerian Circuit if it exists. In weighted graph the minimum possible weight of Postman tour is sum of all edge weights which we get through Eulerian Circuit. We can’t get a shorter route as we must visit all edges at-least once.

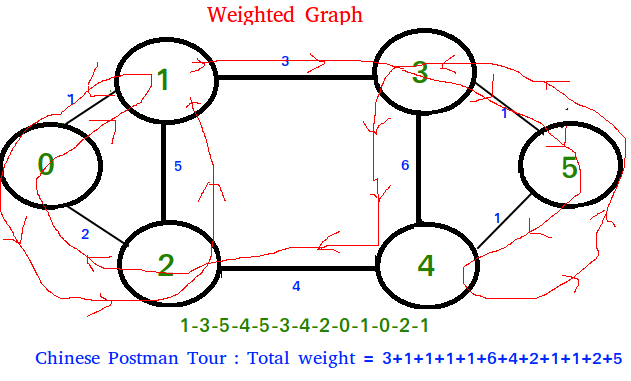
**If input graph does NOT contain Euler Circuit**

In this case, the task reduces to following.

**1)** In unweighted graph, minimum number of edges to duplicate so that the given graph converts to a graph with Eulerian Cycle.



**2)**In weighted graph, minimum total weight of edges to duplicate so that given graph converts to a graph with Eulerian Cycle.



Algorithm to find shortest closed path or optimal   
Chinese postman route in a weighted graph that may  
not be Eulerian.

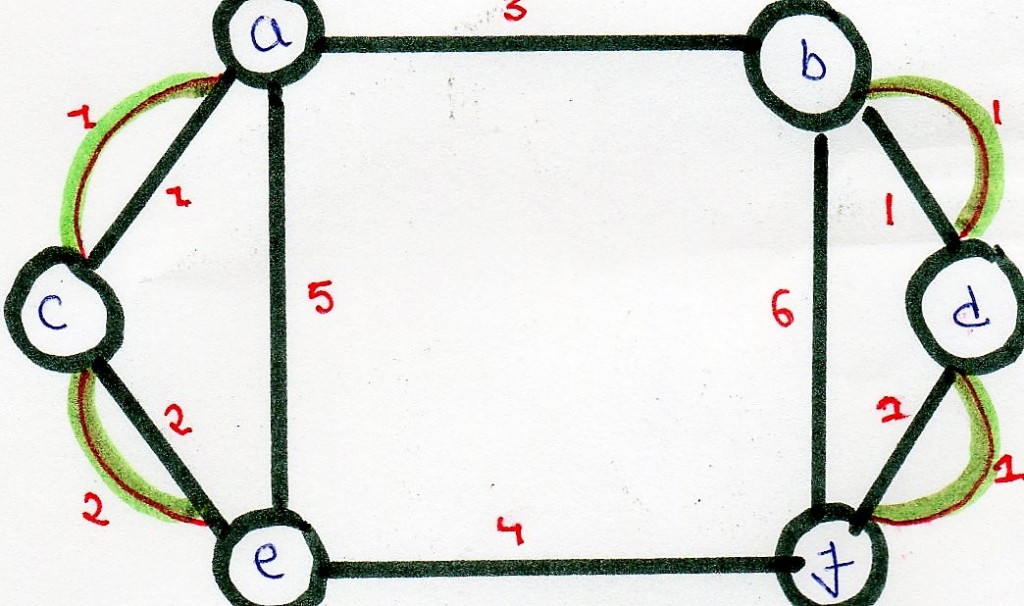
step 1 : If graph is Eulerian, return sum of all   
 edge weights.Else do following steps.  
step 2 : We find all the vertices with odd degree   
step 3 : List all possible pairings of odd vertices   
 For n odd vertices total number of pairings   
 possible are, (n-1) \* (n-3) \* (n -5)... \* 1  
step 4 : For each set of pairings, find the shortest   
 path connecting them.  
step 5 : Find the pairing with minimum shortest path   
 connecting pairs.  
step 6 : Modify the graph by adding all the edges that   
 have been found in step 5.  
step 7 : Weight of Chinese Postman Tour is sum of all   
 edges in the modified graph.  
step 8 : Print Euler Circuit of the modified graph.   
 This Euler Circuit is Chinese Postman Tour.

**Illustration :**

3  
 (a)-----------------(b)  
 1 / | | \1  
 / | | \  
 (c) | 5 6| (d)  
 \ | | /  
 2 \ | 4 | /1  
 (e)------------------(f)  
As we see above graph does not contain Eulerian circuit  
because is has odd degree vertices [a, b, e, f]  
they all are odd degree vertices .

First we make all possible pairs of odd degree vertices  
[ae, bf], [ab, ef], [af, eb]   
so pairs with min sum of weight are [ae, bf] :  
ae = (ac + ce = 3 ), bf = ( bd + df = 2 )   
Total : 5

We add edges ac, ce, bd and df to the original graph and  
create a modified graph.



Optimal chinese postman route is of length : 5 + 23 =   
28 [ 23 = sum of all edges of modified graph ]

Chinese Postman Route :   
a - b - d - f - d - b - f - e - c - a - c - e - a   
This route is Euler Circuit of the modified graph.

**Hierholzer’s Algorithm for directed graph**

Given a directed Eulerian graph, print an [Euler circuit](https://www.geeksforgeeks.org/eulerian-path-and-circuit/). Euler circuit is a path that traverses every edge of a graph, and the path ends on the starting vertex. **Examples:**

Input : Adjacency list for the below graph

Output : 0 -> 1 -> 2 -> 0

Input : Adjacency list for the below graph

Output : 0 -> 6 -> 4 -> 5 -> 0 -> 1   
 -> 2 -> 3 -> 4 -> 2 -> 0   
Explanation:  
In both the cases, we can trace the Euler circuit   
by following the edges as indicated in the output.

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

We have discussed the [problem of finding out whether a given graph is Eulerian or not](https://www.geeksforgeeks.org/eulerian-path-and-circuit/). In this post, an algorithm to print the Eulerian trail or circuit is discussed. The same problem can be solved using [Fleury’s Algorithm](https://www.geeksforgeeks.org/fleurys-algorithm-for-printing-eulerian-path/), however, its complexity is O(E\*E). Using Hierholzer’s Algorithm, we can find the circuit/path in O(E), i.e., linear time. Below is the Algorithm: ref ( [wiki](https://en.wikipedia.org/wiki/Eulerian_path#Hierholzer.27s_algorithm) ). Remember that a directed graph has a Eulerian cycle if the following conditions are true (1) All vertices with nonzero degrees belong to a single strongly connected component. (2) In degree and out-degree of every vertex is the same. The algorithm assumes that the given graph has a Eulerian Circuit.

1. Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because indegree and outdegree of every vertex must be same, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
2. As long as there exists a vertex u that belongs to the current tour, but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.

Thus the idea is to keep following unused edges and removing them until we get stuck. Once we get stuck, we backtrack to the nearest vertex in our current path that has unused edges, and we repeat the process until all the edges have been used. We can use another container to maintain the final path. Let’s take an example:

Let the initial directed graph be as below

Let's start our path from 0.  
Thus, curr\_path = {0} and circuit = {}  
Now let's use the edge 0->1

Now, curr\_path = {0,1} and circuit = {}  
similarly we reach up to 2 and then to 0 again as

Now, curr\_path = {0,1,2} and circuit = {}  
Then we go to 0, now since 0 haven't got any unused  
edge we put 0 in circuit and back track till we find  
an edge

We then have curr\_path = {0,1,2} and circuit = {0}  
Similarly, when we backtrack to 2, we don't find any   
unused edge. Hence put 2 in the circuit and backtrack   
again.

curr\_path = {0,1} and circuit = {0,2}

After reaching 1 we go to through unused edge 1->3 and   
then 3->4, 4->1 until all edges have been traversed.

The contents of the two containers look as:  
curr\_path = {0,1,3,4,1} and circuit = {0,2}

now as all edges have been used, the curr\_path is   
popped one by one into the circuit.  
Finally, we've circuit = {0,2,1,4,3,1,0}

We print the circuit in reverse to obtain the path followed.  
i.e., **0->1->3->4->1->1->2->0**

Below is the implementation for the above approach:

# Python3 program to print Eulerian circuit in given

# directed graph using Hierholzer algorithm

**def** printCircuit(adj):

    # adj represents the adjacency list of

    # the directed graph

    # edge\_count represents the number of edges

    # emerging from a vertex

    edge\_count **=** dict()

**for** i **in** range(len(adj)):

        # find the count of edges to keep track

        # of unused edges

        edge\_count[i] **=** len(adj[i])

**if** len(adj) **==** 0:

**return** # empty graph

    # Maintain a stack to keep vertices

    curr\_path **=** []

    # vector to store final circuit

    circuit **=** []

    # start from any vertex

    curr\_path.append(0)

    curr\_v **=** 0 # Current vertex

**while** len(curr\_path):

        # If there's remaining edge

**if** edge\_count[curr\_v]:

            # Push the vertex

            curr\_path.append(curr\_v)

            # Find the next vertex using an edge

            next\_v **=** adj[curr\_v][**-**1]

            # and remove that edge

            edge\_count[curr\_v] **-=** 1

            adj[curr\_v].pop()

            # Move to next vertex

            curr\_v **=** next\_v

        # back-track to find remaining circuit

**else**:

            circuit.append(curr\_v)

            # Back-tracking

            curr\_v **=** curr\_path[**-**1]

            curr\_path.pop()

    # we've got the circuit, now print it in reverse

**for** i **in** range(len(circuit) **-** 1, **-**1, **-**1):

**print**(circuit[i], end **=** "")

**if** i:

**print**(" -> ", end **=** "")

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Input Graph 1

    adj1 **=** [0] **\*** 3

**for** i **in** range(3):

        adj1[i] **=** []

    # Build the edges

    adj1[0].append(1)

    adj1[1].append(2)

    adj1[2].append(0)

    printCircuit(adj1)

    print()

    # Input Graph 2

    adj2 **=** [0] **\*** 7

**for** i **in** range(7):

        adj2[i] **=** []

    adj2[0].append(1)

    adj2[0].append(6)

    adj2[1].append(2)

    adj2[2].append(0)

    adj2[2].append(3)

    adj2[3].append(4)

    adj2[4].append(2)

    adj2[4].append(5)

    adj2[5].append(0)

    adj2[6].append(4)

    printCircuit(adj2)

    print()

# This code is contributed by

# sanjeev2552

**Output:**

0 -> 1 -> 2 -> 0  
0 -> 6 -> 4 -> 5 -> 0 -> 1 -> 2 -> 3 -> 4 -> 2 -> 0

**Time complexity :**O(V+E), where V is the number of vertices in the graph and E is the number of edges. This is because the code uses a stack to store the vertices and the stack operations push and pop have a time complexity of O(1).

**Space complexity :** O(V+E), as the code uses a stack to store the vertices and a vector to store the circuit, both of which take up O(V) space. Additionally, the code also uses an unordered\_map to store the count of edges for each vertex, which takes up O(E) space.

**Alternate Implementation:**Below are the improvements made from the above code

The above code kept a count of the number of edges for every vertex. This is unnecessary since we are already maintaining the adjacency list. We simply deleted the creation of edge\_count array. In the algorithm we replaced `if edge\_count[current\_v]` with `if adj[current\_v]`

The above code pushes the initial node twice to the stack. Though the way he coded the result is correct, this approach is confusing and inefficient. We eliminated this by appending the next vertex to the stack, instead of the current one.

In the main part, where the author tests the algorithm, the initialization of the adjacency lists `adj1` and `adj2`were a little weird. That potion is also improved.

# Python3 program to print Eulerian circuit in given

# directed graph using Hierholzer algorithm

**def** printCircuit(adj):

    # adj represents the adjacency list of

    # the directed graph

**if** len(adj) **==** 0:

**return** # empty graph

    # Maintain a stack to keep vertices

    # We can start from any vertex, here we start with 0

    curr\_path **=** [0]

    # list to store final circuit

    circuit **=** []

**while** curr\_path:

        curr\_v **=** curr\_path[**-**1]

        # If there's remaining edge in adjacency list

        # of the current vertex

**if** adj[curr\_v]:

            # Find and remove the next vertex that is

            # adjacent to the current vertex

            next\_v **=** adj[curr\_v].pop()

            # Push the new vertex to the stack

            curr\_path.append(next\_v)

        # back-track to find remaining circuit

**else**:

            # Remove the current vertex and

            # put it in the circuit

            circuit.append(curr\_path.pop())

    # we've got the circuit, now print it in reverse

**for** i **in** range(len(circuit) **-** 1, **-**1, **-**1):

        print(circuit[i], end **=** "")

**if** i:

            print(" -> ", end **=** "")

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    # Input Graph 1

    adj1 **=** [[] **for** \_ **in** range(3)]

    # Build the edges

    adj1[0].append(1)

    adj1[1].append(2)

    adj1[2].append(0)

    printCircuit(adj1)

    print()

    # Input Graph 2

    adj2 **=** [[] **for** \_ **in** range(7)]

    adj2[0].append(1)

    adj2[0].append(6)

    adj2[1].append(2)

    adj2[2].append(0)

    adj2[2].append(3)

    adj2[3].append(4)

    adj2[4].append(2)

    adj2[4].append(5)

    adj2[5].append(0)

    adj2[6].append(4)

    printCircuit(adj2)

    print()

**Output:**

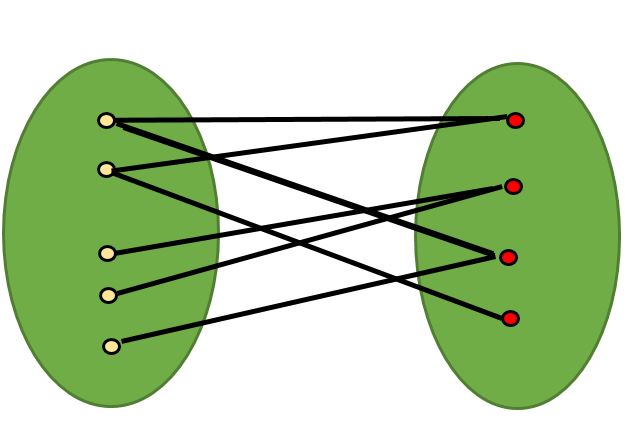
0 -> 1 -> 2 -> 0  
0 -> 6 -> 4 -> 5 -> 0 -> 1 -> 2 -> 3 -> 4 -> 2 -> 0

**Time Complexity :**  O(V + E), where V is the number of vertices and E is the number of edges in the graph. The reason for this is because the algorithm performs a depth-first search (DFS) and visits each vertex and each edge exactly once. So, for each vertex, it takes O(1) time to visit it and for each edge, it takes O(1) time to traverse it.

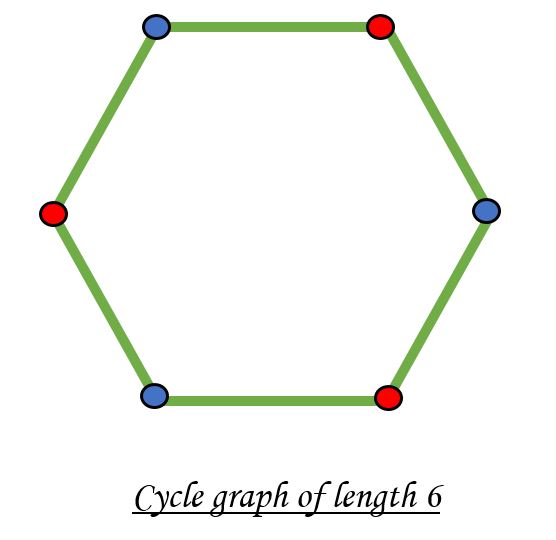
**Space complexity  :**O(V + E), as the algorithm uses a stack to store the current path and a list to store the final circuit. The maximum size of the stack can be V + E at worst, so the space complexity is O(V + E).

**Check whether a given graph is Bipartite or not**

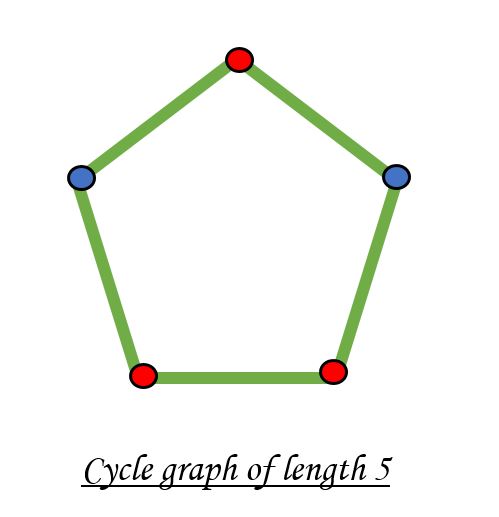
A [Bipartite Graph](http://en.wikipedia.org/wiki/Bipartite_graph) is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.



A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors. For example, see the following graph.



It is not possible to color a cycle graph with odd cycle using two colors.



*Algorithm to check if a graph is Bipartite:*

One approach is to check whether the graph is 2-colorable or not using [backtracking algorithm m coloring problem](https://www.geeksforgeeks.org/backttracking-set-5-m-coloring-problem/).

Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).

1. Assign RED color to the source vertex (putting into set U).

2. Color all the neighbors with BLUE color (putting into set V).

3. Color all neighbor’s neighbor with RED color (putting into set U).

4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where m = 2.

5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

Recommended Problem

Bipartite Graph

# Python program to find out whether a

# given graph is Bipartite or not

**class** Graph():

**def** \_\_init\_\_(self, V):

        self.V **=** V

        self.graph **=** [[0 **for** column **in** range(V)] \

**for** row **in** range(V)]

    # This function returns true if graph G[V][V]

    # is Bipartite, else false

**def** isBipartite(self, src):

        # Create a color array to store colors

        # assigned to all vertices. Vertex

        # number is used as index in this array.

        # The value '-1' of  colorArr[i] is used to

        # indicate that no color is assigned to

        # vertex 'i'. The value 1 is used to indicate

        # first color is assigned and value 0

        # indicates second color is assigned.

        colorArr **=** [**-**1] **\*** self.V

        # Assign first color to source

        colorArr[src] **=** 1

        # Create a queue (FIFO) of vertex numbers and

        # enqueue source vertex for BFS traversal

        queue **=** []

        queue.append(src)

        # Run while there are vertices in queue

        # (Similar to BFS)

**while** queue:

            u **=** queue.pop()

            # Return false if there is a self-loop

**if** self.graph[u][u] **==** 1:

**return** False;

**for** v **in** range(self.V):

                # An edge from u to v exists and destination

                # v is not colored

**if** self.graph[u][v] **==** 1 **and** colorArr[v] **== -**1:

                    # Assign alternate color to this

                    # adjacent v of u

                    colorArr[v] **=** 1 **-** colorArr[u]

                    queue.append(v)

                # An edge from u to v exists and destination

                # v is colored with same color as u

**elif** self.graph[u][v] **==** 1 **and** colorArr[v] **==** colorArr[u]:

**return** False

        # If we reach here, then all adjacent

        # vertices can be colored with alternate

        # color

**return** True

# Driver program to test above function

g **=** Graph(4)

g.graph **=** [[0, 1, 0, 1],

            [1, 0, 1, 0],

            [0, 1, 0, 1],

            [1, 0, 1, 0]

            ]

**print** ("Yes" **if** g.isBipartite(0) **else** "No")

# This code is contributed by Divyanshu Mehta

**Output**

Yes

**Time Complexity**: O(V\*V) as adjacency matrix is used for graph but can be made O(V+E) by using adjacency list

**Auxiliary Space:** O(V) due to queue and color vector.

**The above algorithm works only if the** **graph is connected**. In above code, we always start with source 0 and assume that vertices are visited from it. One important observation is a graph with no edges is also Bipartite. Note that the Bipartite condition says all edges should be from one set to another.

We can extend the above code to handle cases when a graph is not connected. The idea is repeatedly called above method for all not yet visited vertices.

# Python3 program to find out whether a

# given graph is Bipartite or not

**class** Graph():

**def** \_\_init\_\_(self, V):

        self.V **=** V

        self.graph **=** [[0 **for** column **in** range(V)]

**for** row **in** range(V)]

        self.colorArr **=** [**-**1 **for** i **in** range(self.V)]

    # This function returns true if graph G[V][V]

    # is Bipartite, else false

**def** isBipartiteUtil(self, src):

        # Create a color array to store colors

        # assigned to all vertices. Vertex

        # number is used as index in this array.

        # The value '-1' of self.colorArr[i] is used

        # to indicate that no color is assigned to

        # vertex 'i'. The value 1 is used to indicate

        # first color is assigned and value 0

        # indicates second color is assigned.

        # Assign first color to source

        # Create a queue (FIFO) of vertex numbers and

        # enqueue source vertex for BFS traversal

        queue **=** []

        queue.append(src)

        # Run while there are vertices in queue

        # (Similar to BFS)

**while** queue:

            u **=** queue.pop()

            # Return false if there is a self-loop

**if** self.graph[u][u] **==** 1:

**return** False

**for** v **in** range(self.V):

                # An edge from u to v exists and

                # destination v is not colored

**if** (self.graph[u][v] **==** 1 **and**

                        self.colorArr[v] **== -**1):

                    # Assign alternate color to

                    # this adjacent v of u

                    self.colorArr[v] **=** 1 **-** self.colorArr[u]

                    queue.append(v)

                # An edge from u to v exists and destination

                # v is colored with same color as u

**elif** (self.graph[u][v] **==** 1 **and**

                      self.colorArr[v] **==** self.colorArr[u]):

**return** False

        # If we reach here, then all adjacent

        # vertices can be colored with alternate

        # color

**return** True

**def** isBipartite(self):

        self.colorArr **=** [**-**1 **for** i **in** range(self.V)]

**for** i **in** range(self.V):

**if** self.colorArr[i] **== -**1:

**if not** self.isBipartiteUtil(i):

**return** False

**return** True

# Driver Code

g **=** Graph(4)

g.graph **=** [[0, 1, 0, 1],

           [1, 0, 1, 0],

           [0, 1, 0, 1],

           [1, 0, 1, 0]]

print ("Yes" **if** g.isBipartite() **else** "No")

# This code is contributed by Anshuman Sharma

**Output**

Yes

**Time complexity**: **O(V+E)**.

**Auxiliary Space: O(V),**because we have a V-size array.

**If Graph is represented using Adjacency List** .Time Complexity will be O(V+E).

Works for connected as well as disconnected graph.

**def** isBipartite(V, adj):

    # vector to store colour of vertex

    # assigning all to -1 i.e. uncoloured

    # colours are either 0 or 1

    # for understanding take 0 as red and 1 as blue

    col **=** [**-**1]**\***(V)

    # queue for BFS storing {vertex , colour}

    q **=** []

    #loop incase graph is not connected

**for** i **in** range(V):

        # if not coloured

**if** (col[i] **== -**1):

            # colouring with 0 i.e. red

            q.append([i, 0])

            col[i] **=** 0

**while** len(q) !**=** 0:

                p **=** q[0]

                q.pop(0)

                # current vertex

                v **=** p[0]

                # colour of current vertex

                c **=** p[1]

                # traversing vertexes connected to current vertex

**for** j **in** adj[v]:

                    # if already coloured with parent vertex color

                    # then bipartite graph is not possible

**if** (col[j] **==** c):

**return** False

                    # if uncoloured

**if** (col[j] **== -**1):

                        # colouring with opposite color to that of parent

**if** c **==** 1:

                            col[j] **=** 0

**else**:

                            col[j] **=** 1

                        q.append([j, col[j]])

    # if all vertexes are coloured such that

    # no two connected vertex have same colours

**return** True

V, E **=** 4, 8

# adjacency list for storing graph

adj **=** []

adj.append([1,3])

adj.append([0,2])

adj.append([1,3])

adj.append([0,2])

ans **=** isBipartite(V, adj)

# returns 1 if bipartite graph is possible

**if** (ans):

    print("Yes")

# returns 0 if bipartite graph is not possible

**else**:

**print**("No")

    # This code is contributed by divyesh072019.

**Output**

Yes

**Time Complexity:**O(V+E)

**Auxiliary Space:**O(V)

**Exercise:**

**1.** Can DFS algorithm be used to check the bipartite-ness of a graph? If yes, how?

Solution :

# Python3 program to find out whether a given

# graph is Bipartite or not using recursion.

V **=** 4

**def** colorGraph(G, color, pos, c):

**if** color[pos] !**= -**1 **and** color[pos] !**=** c:

**return** False

    # color this pos as c and all its neighbours and 1-c

    color[pos] **=** c

    ans **=** True

**for** i **in** range(0, V):

**if** G[pos][i]:

**if** color[i] **== -**1:

                ans &**=** colorGraph(G, color, i, 1**-**c)

**if** color[i] !**=-**1 **and** color[i] !**=** 1**-**c:

**return** False

**if not** ans:

**return** False

**return** True

**def** isBipartite(G):

    color **=** [**-**1] **\*** V

    #start is vertex 0

    pos **=** 0

    # two colors 1 and 0

**return** colorGraph(G, color, pos, 1)

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    G **=** [[0, 1, 0, 1],

         [1, 0, 1, 0],

         [0, 1, 0, 1],

         [1, 0, 1, 0]]

**if** isBipartite(G): print("Yes")

**else**: **print**("No")

# This code is contributed by Rituraj Jain

**Output**

Yes

**Time Complexity:**O(V+E)

**Auxiliary Space:**O(V)

**References:**

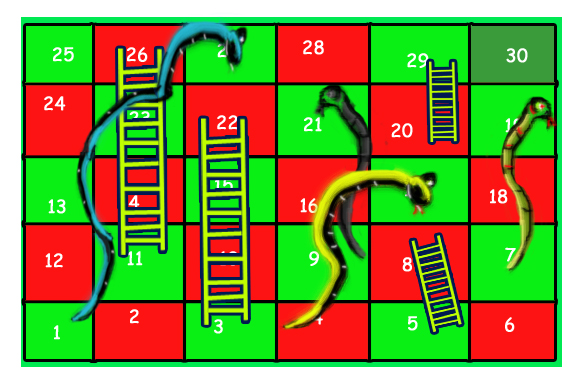
<http://en.wikipedia.org/wiki/Graph_coloring>

<http://en.wikipedia.org/wiki/Bipartite_graph>

**Snake and Ladder Problem**

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from the source or 1st cell. Basically, the player has total control over the outcome of the dice throw and wants to find out the minimum number of throws required to reach the last cell.

If the player reaches a cell which is the base of a ladder, the player has to climb up that ladder and if reaches a cell is the mouth of the snake, and has to go down to the tail of the snake without a dice throw.



For example, consider the board shown, the minimum number of dice throws required to reach cell 30 from cell 1 is 3.

Following are the steps:

a) First throw two dice to reach cell number 3 and then ladder to reach 22

b) Then throw 6 to reach 28.

c) Finally through 2 to reach 30.

There can be other solutions as well like (2, 2, 6), (2, 4, 4), (2, 3, 5).. etc.

Recommended Problem

Snake and Ladder Problem

The idea is to consider the given snake and ladder board as a directed graph with a number of vertices equal to the number of cells in the board. The problem reduces to finding the shortest path in a graph. Every vertex of the graph has an edge to next six vertices if the next 6 vertices do not have a snake or ladder. If any of the next six vertices has a snake or ladder, then the edge from the current vertex goes to the top of the ladder or tail of the snake. Since all edges are of equal weight, we can efficiently find the shortest path using Breadth-First [Search](https://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/) of the graph.

Following is the implementation of the above idea. The input is represented by two things, the first is ‘N’ which is a number of cells in the given board, second is an array ‘move[0…N-1]’ of size N. An entry move[i] is -1 if there is no snake and no ladder from i, otherwise move[i] contains index of destination cell for the snake or the ladder at i.

# Python3 program to find minimum number

# of dice throws required to reach last

# cell from first cell of a given

# snake and ladder board

# An entry in queue used in BFS

**class** QueueEntry(object):

**def** \_\_init\_\_(self, v**=**0, dist**=**0):

        self.v **=** v

        self.dist **=** dist

'''This function returns minimum number of

dice throws required to. Reach last cell

from 0'th cell in a snake and ladder game.

move[] is an array of size N where N is

no. of cells on board. If there is no

snake or ladder from cell i, then move[i]

is -1. Otherwise move[i] contains cell to

which snake or ladder at i takes to.'''

**def** getMinDiceThrows(move, N):

    # The graph has N vertices. Mark all

    # the vertices as not visited

    visited **=** [False] **\*** N

    # Create a queue for BFS

    queue **=** []

    # Mark the node 0 as visited and enqueue it

    visited[0] **=** True

    # Distance of 0't vertex is also 0

    # Enqueue 0'th vertex

    queue.append(QueueEntry(0, 0))

    # Do a BFS starting from vertex at index 0

    qe **=** QueueEntry()  # A queue entry (qe)

**while** queue:

        qe **=** queue.pop(0)

        v **=** qe.v  # Vertex no. of queue entry

        # If front vertex is the destination

        # vertex, we are done

**if** v **==** N **-** 1:

**break**

        # Otherwise dequeue the front vertex

        # and enqueue its adjacent vertices

        # (or cell numbers reachable through

        # a dice throw)

        j **=** v **+** 1

**while** j <**=** v **+** 6 **and** j < N:

            # If this cell is already visited,

            # then ignore

**if** visited[j] **is** False:

                # Otherwise calculate its

                # distance and mark it

                # as visited

                a **=** QueueEntry()

                a.dist **=** qe.dist **+** 1

                visited[j] **=** True

                # Check if there a snake or ladder

                # at 'j' then tail of snake or top

                # of ladder become the adjacent of 'i'

                a.v **=** move[j] **if** move[j] !**= -**1 **else** j

                queue.append(a)

            j **+=** 1

    # We reach here when 'qe' has last vertex

    # return the distance of vertex in 'qe

**return** qe.dist

# driver code

N **=** 30

moves **=** [**-**1] **\*** N

# Ladders

moves[2] **=** 21

moves[4] **=** 7

moves[10] **=** 25

moves[19] **=** 28

# Snakes

moves[26] **=** 0

moves[20] **=** 8

moves[16] **=** 3

moves[18] **=** 6

**print**("Min Dice throws required is {0}".

      format(getMinDiceThrows(moves, N)))

# This code is contributed by Ajitesh Pathak

**Output**

Min Dice throws required is 3

**The time complexity** of the above solution is O(N) as every cell is added and removed only once from the queue. And a typical enqueue or dequeue operation takes O(1) time.

**Auxiliary Space : O(N)**

Another approach we can think of is **recursion** in which we will be going to each block, in this case, which is from 1 to 30, and keeping a count of a minimum number of throws of dice at block i and storing it in an array t.

So, basically, we will:

1. Create an array, let’s say ‘t’, and initialize it with -1.
2. Now we will call a recursive function from block 1, with variable let’s say ‘i’, and we will be incrementing this.
3. In this we will define the base condition as whenever block number reaches 30 or beyond we will return 0 and we will also check if this block has been visited before, this we will do by checking the value of t[i], if this is -1 then it means its not visited and we move forward with the function else its visited and we will return value of t[i].
4. After checking base cases we will initialize a variable ‘min’ with a max integer value.
5. Now we will initiate a loop from 1 to 6, i.e the values of a dice, now for each iteration we will increase the value of i by the value of dice(eg: i+1,i+2….i+6) and we will check if any increased value has a ladder on it if there is then we will update the value of i to the end of the ladder and then pass the value to the recursive function, if there is no ladder then also we will pass the incremented value of i based on dice value to a recursive function, **but**if there is a snake then we won’t pass this value to recursive function as we want to reach the end as soon as possible, and the best of doing this would be not to be bitten by a snake. And we would be keep on updating the minimum value for variable ‘min’.
6. Finally we will update t[i] with min and return t[i].

Below is the implementation of the above approach:

**from** typing **import** List, Dict

**def** min\_throw(n: int, arr: List[int]) **-**> int:

    # Initialise an array t of length 31, we will use from

    # index to 1 to 30

    t **=** [**-**1] **\*** 31

    # create a dictionary to store snakes and ladders start

    # and end for better efficiency

    h **=** {}

**for** i **in** range(0, 2 **\*** n, 2):

        # store start as key and end as value

        h[arr[i]] **=** arr[i **+** 1]

    # final ans

**return** sol(1, h, t)

# recursive function

**def** sol(i: int, h: Dict[int, int], t: List[int]) **-**> int:

    # base condition

**if** i >**=** 30:

**return** 0

    # checking if block is already visited or

    # not(memoization).

**elif** t[i] !**= -**1:

**return** t[i]

    # initialising min as max int value

    min\_value **=** float("inf")

    # for loop for every dice value from 1 to 6

**for** j **in** range(1, 7):

        # incrementing value of i with dice value i.e j

        # taking new variable k

        # ->taking new variable so that we dont change i

        # as we will need it again in another iteration

        k **=** i **+** j

**if** k **in** h:

            # checking if this is a snake or ladder

            # if a snake then we continue as we dont

            # need a snake

**if** h[k] < k:

**continue**

            # updating if it's a ladder to ladder end value

            k **=** h[k]

        # updating min in every iteration for getting

        # minimum throws from this particular block

        min\_value **=** min(min\_value, sol(k, h, t) **+** 1)

    # updating value of t[i] to min

    # memoization

    t[i] **=** min\_value

**return** t[i]

# Given a 5x6 snakes and ladders board

# You are given an integer N denoting the total

# number of snakes and ladders and a list arr[]

# of 2\*N size where 2\*i and (2\*i + 1)th values

# denote the starting and ending point respectively

# of ith snake or ladder

N **=** 8

arr **=** [3, 22, 5, 8, 11, 26, 20, 29, 17, 4, 19, 7, 27, 1, 29, 9]

print("Min Dice throws required is", min\_throw(N, arr))

# This code is contributed by sanjanasikarwar24

**Output**

Min Dice throws required is 3

**Time complexity:** O(N).

**Auxiliary Space** O(N)

This article is contributed by **Siddharth**and **Sahil Srivastava**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

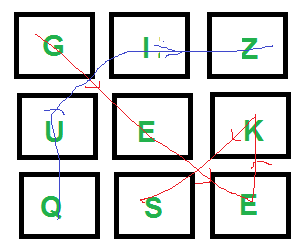
**Boggle (Find all possible words in a board of characters) | Set 1**

Given a dictionary, a method to do lookup in dictionary and a M x N board where every cell has one character. Find all possible words that can be formed by a sequence of adjacent characters. Note that we can move to any of 8 adjacent characters, but a word should not have multiple instances of same cell.

**Example:**

Input: dictionary[] = {"GEEKS", "FOR", "QUIZ", "GO"};  
 boggle[][] = {{'G', 'I', 'Z'},  
 {'U', 'E', 'K'},  
 {'Q', 'S', 'E'}};  
 isWord(str): returns true if str is present in dictionary  
 else false.

Output: Following words of dictionary are present  
 GEEKS  
 QUIZ



[We strongly recommend that you click here and practice it, before moving on to the solution.](https://practice.geeksforgeeks.org/problems/word-boggle4143/1)

The idea is to consider every character as a starting character and find all words starting with it. All words starting from a character can be found using [Depth First Traversal](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/). We do depth-first traversal starting from every cell. We keep track of visited cells to make sure that a cell is considered only once in a word.

# Python3 program for Boggle game

# Let the given dictionary be following

dictionary **=** ["GEEKS", "FOR", "QUIZ", "GO"]

n **=** len(dictionary)

M **=** 3

N **=** 3

# A given function to check if a given string

# is present in dictionary. The implementation is

# naive for simplicity. As per the question

# dictionary is given to us.

**def** isWord(Str):

    # Linearly search all words

**for** i **in** range(n):

**if** (Str **==** dictionary[i]):

**return** True

**return** False

# A recursive function to print all words present on boggle

**def** findWordsUtil(boggle, visited, i, j, Str):

    # Mark current cell as visited and

    # append current character to str

    visited[i][j] **=** True

    Str **=** Str **+** boggle[i][j]

    # If str is present in dictionary,

    # then print it

**if** (isWord(Str)):

        print(Str)

    # Traverse 8 adjacent cells of boggle[i,j]

    row **=** i **-** 1

**while** row <**=** i **+** 1 **and** row < M:

        col **=** j **-** 1

**while** col <**=** j **+** 1 **and** col < N:

**if** (row >**=** 0 **and** col >**=** 0 **and not** visited[row][col]):

                findWordsUtil(boggle, visited, row, col, Str)

            col**+=**1

        row**+=**1

    # Erase current character from string and

    # mark visited of current cell as false

    Str **=** "" **+** Str[**-**1]

    visited[i][j] **=** False

# Prints all words present in dictionary.

**def** findWords(boggle):

    # Mark all characters as not visited

    visited **=** [[False **for** i **in** range(N)] **for** j **in** range(M)]

    # Initialize current string

    Str **=** ""

    # Consider every character and look for all words

    # starting with this character

**for** i **in** range(M):

**for** j **in** range(N):

        findWordsUtil(boggle, visited, i, j, Str)

# Driver Code

boggle **=** [["G", "I", "Z"], ["U", "E", "K"], ["Q", "S", "E"]]

print("Following words of", "dictionary are present")

findWords(boggle)

#  This code is contributed by divyesh072019.

**Output**

Following words of dictionary are present  
GEEKS  
QUIZ

Note that the above solution may print the same word multiple times. For example, if we add “SEEK” to the dictionary, it is printed multiple times. To avoid this, we can use hashing to keep track of all printed words.

To improve time complexity, we can use unordered\_set(in C++) or dictionary(in Python) which takes constant search time. Now Time Complexity, Since we are doing depth-first traversal for every position in the array so n\*m( time for one DFS) = n\*m( |V| + |E|) where |V| is the total number of nodes and |E| is the total number of edges which are equal to n\*m. So,

**Time Complexity:**O(N2 \*M2)

**Auxiliary Space:**O(N\*M)

**Optimised Approach :**

Instead of generating all strings from the grid and the checking whether it exists in dictionary or not , we can simply run a DFS on all words present in dictionary and check whether we can make that word from grid or not. This Approach is more optimised then the previous one.

Below is the implementation of above Approach.

**def** dfs(board, s, i, j, n, m, idx):

**if** i < 0 **or** i >**=** n **or** j < 0 **or** j >**=** m:

**return** False

**if** s[idx] !**=** board[i][j]:

**return** False

**if** idx **==** len(s) **-** 1:

**return** True

    temp **=** board[i][j]

    board[i][j] **=** '\*'

    a **=** dfs(board, s, i, j**+**1, n, m, idx**+**1)

    b **=** dfs(board, s, i, j**-**1, n, m, idx**+**1)

    c **=** dfs(board, s, i**+**1, j, n, m, idx**+**1)

    d **=** dfs(board, s, i**-**1, j, n, m, idx**+**1)

    e **=** dfs(board, s, i**+**1, j**+**1, n, m, idx**+**1)

    f **=** dfs(board, s, i**-**1, j**+**1, n, m, idx**+**1)

    g **=** dfs(board, s, i**+**1, j**-**1, n, m, idx**+**1)

    h **=** dfs(board, s, i**-**1, j**-**1, n, m, idx**+**1)

    board[i][j] **=** temp

**return** a **or** b **or** c **or** e **or** f **or** g **or** h **or** d

**def** wordBoggle(board, dictionary):

    n **=** len(board)

    m **=** len(board[0])

    store **=** set()

    #     Let the given dictionary be following

**for** word **in** dictionary:

**for** i **in** range(n):

**for** j **in** range(m):

**if** dfs(board, word, i, j, n, m, 0):

                    store.add(word)

**for** word **in** store:

        print(word)

boggle **=** [['G', 'I', 'Z'],

          ['U', 'E', 'K'],

          ['Q', 'S', 'E']]

dictionary **=** ["GEEKS", "FOR", "QUIZ", "GO"]

**print**("Following words of dictionary are present:")

wordBoggle(boggle, dictionary)

# This code is contributed by vikramshirsath177

**Output**

Following words of dictionary are present  
GEEKS  
QUIZ

**Time Complexity: O(N\*W + R\*C^2)**

**Auxiliary Space: O(N\*W + R\*C)**

In below set 2, we have discussed Trie based optimized solution:

[**Boggle | Set 2 (Using Trie)**](https://www.geeksforgeeks.org/boggle-set-2-using-trie/)

**Hopcroft–Karp Algorithm for Maximum Matching | Set 1 (Introduction)**

A matching in a [Bipartite Graph](https://www.geeksforgeeks.org/bipartite-graph) is a set of the edges chosen in such a way that no two edges share an endpoint. A maximum matching is a matching of maximum size (maximum number of edges). In a maximum matching, if any edge is added to it, it is no longer a matching. There can be more than one maximum matching for a given Bipartite Graph.

We have discussed importance of maximum matching and [Ford Fulkerson Based approach for maximal Bipartite Matching](https://www.geeksforgeeks.org/maximum-bipartite-matching/) in [previous post](https://www.geeksforgeeks.org/maximum-bipartite-matching/). Time complexity of the Ford Fulkerson based algorithm is O(V x E). Hopcroft Karp algorithm is an improvement that runs in O(√V x E) time. Let us define few terms before we discuss the algorithm

***Free Node or Vertex:*** Given a matching M, a node that is not part of matching is called free node. Initially all vertices as free (See first graph of below diagram). In second graph, u2 and v2 are free. In third graph, no vertex is free.

***Matching and Not-Matching edges:*** Given a matching M, edges that are part of matching are called Matching edges and edges that are not part of M (or connect free nodes) are called Not-Matching edges. In first graph, all edges are non-matching. In second graph, (u0, v1), (u1, v0) and (u3, v3) are matching and others not-matching.

***Alternating Paths:*** Given a matching M, an alternating path is a path in which the edges belong alternatively to the matching and not matching. All single edges paths are alternating paths. Examples of alternating paths in middle graph are u0-v1-u2 and u2-v1-u0-v2.

***Augmenting path:*** Given a matching M, an augmenting path is an alternating path that starts from and ends on free vertices. All single edge paths that start and end with free vertices are augmenting paths. In below diagram, augmenting paths are highlighted with blue color. Note that the augmenting path always has one extra matching edge. The Hopcroft Karp algorithm is based on below concept. *A matching M is not maximum if there exists an augmenting path. It is also true other way, i.e, a matching is maximum if no augmenting path exists* So the idea is to one by one look for augmenting paths. And add the found paths to current matching.

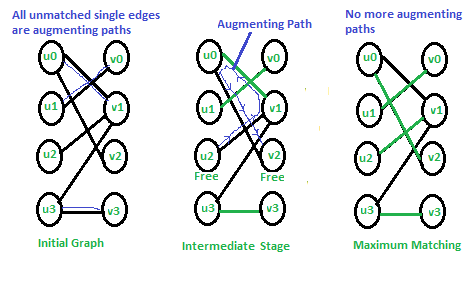
**Hopcroft Karp Algorithm:**

1. *Initialize Maximal Matching M as empty.*
2. *While there exists an Augmenting Path p*

* *Remove matching edges of p from M and add not-matching edges of p to M*
* *(This increases size of M by 1 as p starts and ends with a free vertex)*

1. *Return M.*

Below diagram shows working of the algorithm.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/HopcroftKarp1.png)

In the initial graph all single edges are augmenting paths and we can pick in any order. In the middle stage, there is only one augmenting path. We remove matching edges of this path from M and add not-matching edges. In final matching, there are no augmenting paths so the matching is maximum.

Implementation of Hopcroft Karp algorithm is discussed in set 2.

[Hopcroft–Karp Algorithm for Maximum Matching | Set 2 (Implementation)](https://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-2-implementation/)

**Minimum time required to rot all oranges**

Given a matrix of dimension M \* N where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:

1. 0: Empty cell
2. 1: Cells have fresh oranges
3. 2: Cells have rotten oranges

Determine what is the minimum time required so that all the oranges become rotten. A rotten orange at index (i,j ) can rot other fresh oranges which are its neighbours (up, down, left and right). If it is impossible to rot every orange then simply return -1.

**Examples:**

***Input:****arr[][C] = { {2, 1, 0, 2, 1}, {1, 0, 1, 2, 1}, {1, 0, 0, 2, 1}};*

***Output:****2*

***Explanation:****At 0th time frame:*

*{2, 1, 0, 2, 1}*

*{1, 0, 1, 2, 1}*

*{1, 0, 0, 2, 1}*

*At 1st time frame:*

*{2, 2, 0, 2, 2}*

*{2, 0, 2, 2, 2}*

*{1, 0, 0, 2, 2}*

*At 2nd time frame:*

*{2, 2, 0, 2, 2}*

*{2, 0, 2, 2, 2}*

*{2, 0, 0, 2, 2}*

***Input:****arr[][C] = { {2, 1, 0, 2, 1}, {0, 0, 1, 2, 1}, {1, 0, 0, 2, 1}}*

***Output:****-1*

***Explanation:****At 0th time frame:*

*{2, 1, 0, 2, 1}*

*{0, 0, 1, 2, 1}*

*{1, 0, 0, 2, 1}*

*At 1st time frame:*

*{2, 2, 0, 2, 2}*

*{0, 0, 2, 2, 2}*

*{1, 0, 0, 2, 2}*

*At 2nd time frame:*

*{2, 2, 0, 2, 2}*

*{0, 0, 2, 2, 2}*

*{1, 0, 0, 2, 2}*

*The 1 at the bottom left corner of the matrix is never rotten.*

Recommended Problem

Rotten Oranges

**Naive Approach:**

*The idea is very basic. Traverse through all oranges in multiple rounds. In every round, rot the oranges to the adjacent position of oranges that were rotten in the last round.*

Follow the steps below to solve the problem:

1. Create a variable **no = 2** and **changed = false**.
2. Run a loop until there is no cell of the matrix which is changed in an iteration.
3. Run a nested loop and traverse the matrix:
4. If the element of the matrix is equal to **no** then assign the adjacent elements to **no + 1** if the adjacent element’s value is equal to 1, i.e. not rotten, and update **changed** to true.
5. Traverse the matrix and check if there is any cell that is **1**.
6. If 1 is present return -1
7. Else return **no – 2**.

Below is the implementation of the above approach.

# Python3 program to rot all

# oranges when you can move

# in all the four direction

# from a rotten orange

R **=** 3

C **=** 5

# Check if i, j is under the

# array limits of row and

# column

**def** issafe(i, j):

**if** (i >**=** 0 **and** i < R **and**

            j >**=** 0 **and** j < C):

**return** True

**return** False

**def** rotOranges(v):

    changed **=** False

    no **=** 2

**while** (True):

**for** i **in** range(R):

**for** j **in** range(C):

                # Rot all other oranges

                # present at (i+1, j),

                # (i, j-1), (i, j+1),

                # (i-1, j)

**if** (v[i][j] **==** no):

**if** (issafe(i **+** 1, j) **and**

                            v[i **+** 1][j] **==** 1):

                        v[i **+** 1][j] **=** v[i][j] **+** 1

                        changed **=** True

**if** (issafe(i, j **+** 1) **and**

                            v[i][j **+** 1] **==** 1):

                        v[i][j **+** 1] **=** v[i][j] **+** 1

                        changed **=** True

**if** (issafe(i **-** 1, j) **and**

                            v[i **-** 1][j] **==** 1):

                        v[i **-** 1][j] **=** v[i][j] **+** 1

                        changed **=** True

**if** (issafe(i, j **-** 1) **and**

                            v[i][j **-** 1] **==** 1):

                        v[i][j **-** 1] **=** v[i][j] **+** 1

                        changed **=** True

        # if no rotten orange found

        # it means all oranges rottened

        # now

**if** (**not** changed):

**break**

        changed **=** False

        no **+=** 1

**for** i **in** range(R):

**for** j **in** range(C):

            # if any orange is found

            # to be not rotten then

            # ans is not possible

**if** (v[i][j] **==** 1):

**return -**1

    # Because initial value

    # for a rotten orange was 2

**return** no **-** 2

# Driver function

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    v **=** [[2, 1, 0, 2, 1],

         [1, 0, 1, 2, 1],

         [1, 0, 0, 2, 1]]

    print("Max time incurred: ",

          rotOranges(v))

# This code is contributed by Chitranayal

**Output**

Max time incurred: 2

**Time Complexity**: O((R\*C) \* (R \*C)),

1. The matrix needs to be traversed again and again until there is no change in the matrix, that can happen max(R \*C)/2 times.
2. So time complexity is O((R \* C) \* (R \*C)).

**Auxiliary Space:**O(1), No extra space is required.

**Minimum time required to rot all oranges using**[Breadth First Search](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)**:**

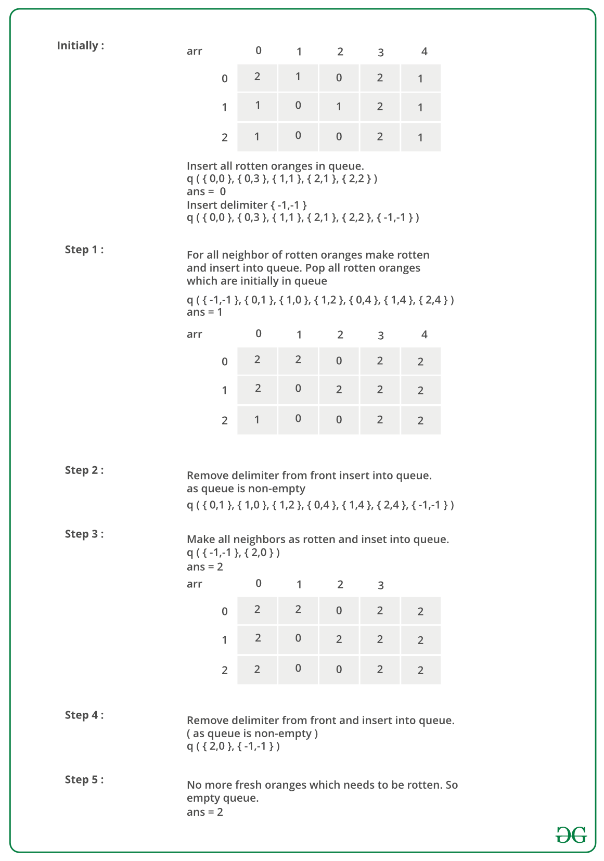
*The idea is to use*[*Breadth First Search*](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)*. The condition of oranges getting rotten is when they come in contact with other rotten oranges. This is similar to a breadth-first search where the graph is divided into layers or circles and the search is done from lower or closer layers to deeper or higher layers.*

*In the previous approach, the idea was based on BFS but the implementation was poor and inefficient. To find the elements whose values are****no****the whole matrix had to be traversed. So time can be reduced by using this efficient approach of BFS.*

Follow the steps below to solve the problem:

1. Create an empty queue **Q**.
2. Find all rotten oranges and enqueue them to **Q**. Also, enqueue a delimiter to indicate the beginning of the next time frame.
3. Run a loop While Q is not empty and do the following while the delimiter in **Q** is not reached
4. Dequeue an orange from the queue, and rot all adjacent oranges.
5. While rotting the adjacent, make sure that the time frame is incremented only once. And the time frame is not incremented if there are no adjacent oranges.
6. Dequeue the old delimiter and enqueue a new delimiter. The oranges rotten in the previous time frame lie between the two delimiters.
7. Return the last time frame.

**Illustration:**



Below is the implementation of the above approach.

# Python3 program to find minimum time required to make all

# oranges rotten

**from** collections **import** deque

# function to check whether a cell is valid / invalid

**def** isvalid(i, j):

**return** (i >**=** 0 **and** j >**=** 0 **and** i < 3 **and** j < 5)

# Function to check whether the cell is delimiter

# which is (-1, -1)

**def** isdelim(temp):

**return** (temp[0] **== -**1 **and** temp[1] **== -**1)

# Function to check whether there is still a fresh

# orange remaining

**def** checkall(arr):

**for** i **in** range(3):

**for** j **in** range(5):

**if** (arr[i][j] **==** 1):

**return** True

**return** False

# This function finds if it is

# possible to rot all oranges or not.

# If possible, then it returns

# minimum time required to rot all,

# otherwise returns -1

**def** rotOranges(arr):

    # Create a queue of cells

    Q **=** deque()

    temp **=** [0, 0]

    ans **=** 1

    # Store all the cells having

    # rotten orange in first time frame

**for** i **in** range(3):

**for** j **in** range(5):

**if** (arr[i][j] **==** 2):

                temp[0] **=** i

                temp[1] **=** j

                Q.append([i, j])

    # Separate these rotten oranges

    # from the oranges which will rotten

    # due the oranges in first time

    # frame using delimiter which is (-1, -1)

    temp[0] **= -**1

    temp[1] **= -**1

    Q.append([**-**1, **-**1])

    # print(Q)

    # Process the grid while there are

    # rotten oranges in the Queue

**while** False:

        # This flag is used to determine

        # whether even a single fresh

        # orange gets rotten due to rotten

        # oranges in current time

        # frame so we can increase

        # the count of the required time.

        flag **=** False

**print**(len(Q))

        # Process all the rotten

        # oranges in current time frame.

**while not** isdelim(Q[0]):

            temp **=** Q[0]

**print**(len(Q))

            # Check right adjacent cell that if it can be rotten

**if** (isvalid(temp[0] **+** 1, temp[1]) **and** arr[temp[0] **+** 1][temp[1]] **==** 1):

                # if this is the first orange to get rotten, increase

                # count and set the flag.

**if** (**not** flag):

                    ans, flag **=** ans **+** 1, True

                # Make the orange rotten

                arr[temp[0] **+** 1][temp[1]] **=** 2

                # append the adjacent orange to Queue

                temp[0] **+=** 1

                Q.append(temp)

                temp[0] **-=** 1  # Move back to current cell

            # Check left adjacent cell that if it can be rotten

**if** (isvalid(temp[0] **-** 1, temp[1]) **and** arr[temp[0] **-** 1][temp[1]] **==** 1):

**if** (**not** flag):

                    ans, flag **=** ans **+** 1, True

                arr[temp[0] **-** 1][temp[1]] **=** 2

                temp[0] **-=** 1

                Q.append(temp)  # append this cell to Queue

                temp[0] **+=** 1

            # Check top adjacent cell that if it can be rotten

**if** (isvalid(temp[0], temp[1] **+** 1) **and** arr[temp[0]][temp[1] **+** 1] **==** 1):

**if** (**not** flag):

                    ans, flag **=** ans **+** 1, True

                arr[temp[0]][temp[1] **+** 1] **=** 2

                temp[1] **+=** 1

                Q.append(temp)  # Push this cell to Queue

                temp[1] **-=** 1

            # Check bottom adjacent cell if it can be rotten

**if** (isvalid(temp[0], temp[1] **-** 1) **and** arr[temp[0]][temp[1] **-** 1] **==** 1):

**if** (**not** flag):

                    ans, flag **=** ans **+** 1, True

                arr[temp[0]][temp[1] **-** 1] **=** 2

                temp[1] **-=** 1

                Q.append(temp)  # append this cell to Queue

            Q.popleft()

        # Pop the delimiter

        Q.popleft()

        # If oranges were rotten in

        # current frame than separate the

        # rotten oranges using delimiter

        # for the next frame for processing.

**if** (len(Q) **==** 0):

            temp[0] **= -**1

            temp[1] **= -**1

            Q.append(temp)

        # If Queue was empty than no rotten oranges left to process so exit

    # Return -1 if all arranges could not rot, otherwise return ans.

**return** ans **+** 1 **if**(checkall(arr)) **else -**1

# Driver program

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [[2, 1, 0, 2, 1],

           [1, 0, 1, 2, 1],

           [1, 0, 0, 2, 1]]

    ans **=** rotOranges(arr)

**if** (ans **== -**1):

**print**("All oranges cannot rotn")

**else**:

**print**("Time required for all oranges to rot => ", ans)

        # This code is contributed by mohit kumar 29

**Output**

Time required for all oranges to rot => 2

**Time Complexity:** O( R \*C), Each element of the matrix can be inserted into the queue only once so the upper bound of iteration is O(R\*C)

**Auxiliary Space:** O(R\*C), To store the elements in a queue.

**Construct a graph from given degrees of all vertices**

This is a C++ program to generate a graph for a given fixed degree sequence. This algorithm generates a undirected graph for the given degree sequence.It does not include self-edge and multiple edges.

**Examples:**

Input : degrees[] = {2, 2, 1, 1}  
Output : (0) (1) (2) (3)  
 (0) 0 1 1 0   
 (1) 1 0 0 1   
 (2) 1 0 0 0   
 (3) 0 1 0 0   
Explanation : We are given that there  
are four vertices with degree of vertex  
0 as 2, degree of vertex 1 as 2, degree  
of vertex 2 as 1 and degree of vertex 3  
as 1. Following is graph that follows  
given conditions.   
 (0)----------(1)  
 | |   
 | |   
 | |  
 (2) (3)

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

***Approach :***

1. *Take the input of the number of vertexes and their corresponding degree.*
2. *Declare adjacency matrix, mat[ ][ ] to store the graph.*
3. *To create the graph, create the first loop to connect each vertex ‘i’.*
4. *Second nested loop to connect the vertex ‘i’ to the every valid vertex ‘j’, next to it.*
5. *If the degree of vertex ‘i’ and ‘j’ are more than zero then connect them.*
6. *Print the adjacency matrix.*

Based on the above explanation, below are implementations:

# Python3 program to generate a graph

# for a given fixed degrees

# A function to print the adjacency matrix.

**def** printMat(degseq, n):

    # n is number of vertices

    mat **=** [[0] **\*** n **for** i **in** range(n)]

**for** i **in** range(n):

**for** j **in** range(i **+** 1, n):

            # For each pair of vertex decrement

            # the degree of both vertex.

**if** (degseq[i] > 0 **and** degseq[j] > 0):

                degseq[i] **-=** 1

                degseq[j] **-=** 1

                mat[i][j] **=** 1

                mat[j][i] **=** 1

    # Print the result in specified form

**print**("      ", end **=** " ")

**for** i **in** range(n):

**print**(" ", "(", i, ")", end **=** "")

**print**()

**print**()

**for** i **in** range(n):

        print(" ", "(", i, ")", end **=** "")

**for** j **in** range(n):

            print("     ", mat[i][j], end **=** "")

**print**()

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    degseq **=** [2, 2, 1, 1, 1]

    n **=** len(degseq)

    printMat(degseq, n)

# This code is contributed by PranchalK

**Output**

(0) (1) (2) (3) (4)

(0) 0 1 1 0 0  
 (1) 1 0 0 1 0  
 (2) 1 0 0 0 0  
 (3) 0 1 0 0 0  
 (4) 0 0 0 0 0

**Time Complexity: O(v\*v).**

**Space complexity :** O(n^2) because it creates a 2-dimensional array (matrix) of size n \* n, where n is the number of vertices in the graph.

**Determine whether a universal sink exists in a directed graph**

Determine whether a universal sink exists in a directed graph. A universal sink is a vertex which has no edge emanating from it, and all other vertices have an edge towards the sink.

Input :   
v1 -> v2 (implies vertex 1 is connected to vertex 2)  
v3 -> v2  
v4 -> v2  
v5 -> v2  
v6 -> v2   
Output :  
Sink found at vertex 2

Input :   
v1 -> v6  
v2 -> v3  
v2 -> v4  
v4 -> v3  
v5 -> v3  
Output :  
No Sink

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

We try to eliminate n – 1 non-sink vertices in **O(n)** time and check the remaining vertex for the sink property.

To eliminate vertices, we check whether a particular index (A[i][j]) in the adjacency matrix is a 1 or a 0. If it is a 0, it means that the vertex corresponding to index j cannot be a sink. If the index is a 1, it means the vertex corresponding to i cannot be a sink. We keep increasing i and j in this fashion until either i or j exceeds the number of vertices.

Using this method allows us to carry out the universal sink test for only one vertex instead of all n vertices. Suppose we are left with only vertex i.

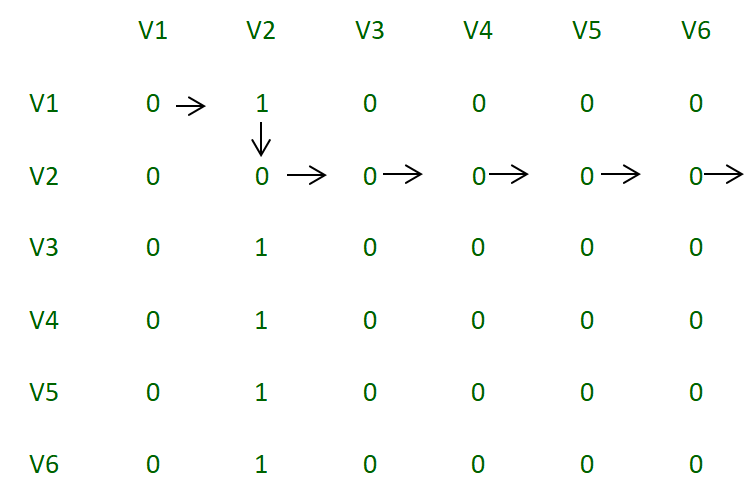
We now check for whether row i has only 0s and whether row j as only 1s except for A[i][i], which will be 0.

**Illustration :**

v1 -> v2   
v3 -> v2  
v4 -> v2  
v5 -> v2  
v6 -> v2   
We can visualize the adjacency matrix for   
the above as follows:  
0 1 0 0 0 0  
0 0 0 0 0 0  
0 1 0 0 0 0  
0 1 0 0 0 0  
0 1 0 0 0 0

We observe that vertex 2 does not have any emanating edge, and that every other vertex has an edge in vertex 2. At A[0][0] (A[i][j]), we encounter a 0, so we increment j and next look at A[0][1]. Here we encounter a 1. So we have to increment i by 1. A[1][1] is 0, so we keep increasing j.

We notice that A[1][2], A[1][3].. etc are all 0, so j will exceed the number of vertices (6 in this example). We now check row i and column i for the sink property. Row i must be completely 0, and column i must be completely 1 except for the index A[i][i]

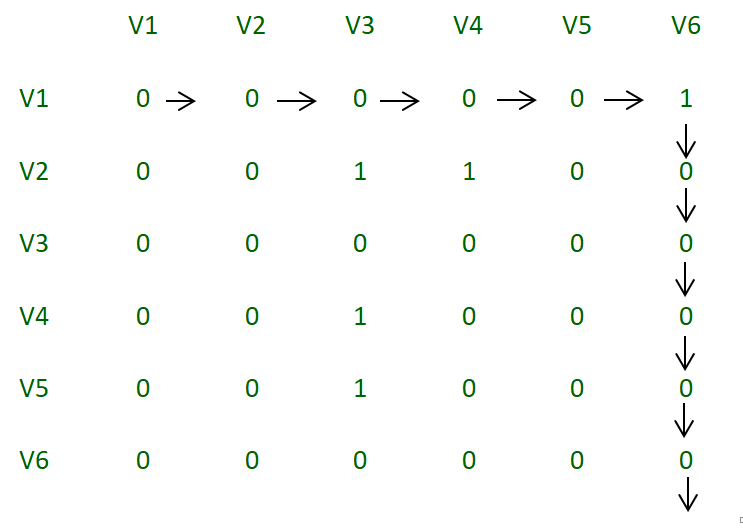
[](https://media.geeksforgeeks.org/wp-content/uploads/Adjacency_matrix_2.png)

*Adjacency Matrix*

**Second Example:**

v1 -> v6  
v2 -> v3  
v2 -> v4  
v4 -> v3  
v5 -> v3  
We can visualize the adjacency matrix  
for the above as follows:  
0 0 0 0 0 1  
0 0 1 1 0 0  
0 0 0 0 0 0  
0 0 1 0 0 0  
0 0 1 0 0 0  
0 0 0 0 0 0

In this example, we observer that in row 1, every element is 0 except for the last column. So we will increment j until we reach the 1. When we reach 1, we increment i as long as the value of A[i][j] is 0. If i exceeds the number of vertices, it is not possible to have a sink, and in this case, i will exceed the number of vertices.

[](https://media.geeksforgeeks.org/wp-content/uploads/Adjacency_matrix.png)

*Adjacency Matrix*

**Implementation:**

# Python3 program to find whether a

# universal sink exists in a directed graph

**class** Graph:

    # constructor to initialize number of

    # vertices and size of adjacency matrix

**def** \_\_init\_\_(self, vertices):

        self.vertices **=** vertices

        self.adjacency\_matrix **=** [[0 **for** i **in** range(vertices)]

**for** j **in** range(vertices)]

**def** insert(self, s, destination):

        # make adjacency\_matrix[i][j] = 1

        # if there is an edge from i to j

        self.adjacency\_matrix[s **-** 1][destination **-** 1] **=** 1

**def** issink(self, i):

**for** j **in** range(self.vertices):

            # if any element in the row i is 1, it means

            # that there is an edge emanating from the

            # vertex, which means it cannot be a sink

**if** self.adjacency\_matrix[i][j] **==** 1:

**return** False

            # if any element other than i in the column

            # i is 0, it means that there is no edge from

            # that vertex to the vertex we are testing

            # and hence it cannot be a sink

**if** self.adjacency\_matrix[j][i] **==** 0 **and** j !**=** i:

**return** False

        # if none of the checks fails, return true

**return** True

    # we will eliminate n-1 non sink vertices so that

    # we have to check for only one vertex instead of

    # all n vertices

**def** eliminate(self):

        i **=** 0

        j **=** 0

**while** i < self.vertices **and** j < self.vertices:

            # If the index is 1, increment the row

            # we are checking by 1

            # else increment the column

**if** self.adjacency\_matrix[i][j] **==** 1:

                i **+=** 1

**else**:

                j **+=** 1

        # If i exceeds the number of vertices, it

        # means that there is no valid vertex in

        # the given vertices that can be a sink

**if** i > self.vertices:

**return -**1

**elif** self.issink(i) **is** False:

**return -**1

**else**:

**return** i

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    number\_of\_vertices **=** 6

    number\_of\_edges **=** 5

    g **=** Graph(number\_of\_vertices)

    # input set 1

    # g.insert(1, 6)

    # g.insert(2, 6)

    # g.insert(3, 6)

    # g.insert(4, 6)

    # g.insert(5, 6)

    # input set 2

    g.insert(1, 6)

    g.insert(2, 3)

    g.insert(2, 4)

    g.insert(4, 3)

    g.insert(5, 3)

    vertex **=** g.eliminate()

    # returns 0 based indexing of vertex.

    # returns -1 if no sink exits.

    # returns the vertex number-1 if sink is found

**if** vertex >**=** 0:

**print**("Sink found at vertex %d" **%** (vertex **+** 1))

**else**:

**print**("No Sink")

# This code is contributed by

# sanjeev2552

**Output**

No Sink

This program eliminates non-sink vertices in **O(n)** complexity and checks for the sink property in **O(n)** complexity.

**Time complexity: O(V^2)**

We have used a 2-D array of size V x V to store the adjacency matrix of the given graph. The time complexity of the algorithm is O(V^2) as we need to traverse the complete adjacency matrix to find the sink vertex.

**Time complexity: O(V^2)**

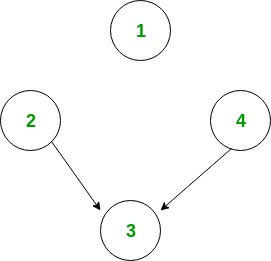
The space complexity of the algorithm is also O(V^2) since we need to store the adjacency matrix.

**Number of sink nodes in a graph**

Given a Directed Acyclic Graph of **n** nodes (numbered from 1 to n) and **m** edges. The task is to find the number of sink nodes. A sink node is a node such that no edge emerges out of it.

**Examples:**

Input : n = 4, m = 2  
 Edges[] = {{2, 3}, {4, 3}}   
Output : 2



Only node 1 and node 3 are sink nodes.

Input : n = 4, m = 2  
 Edges[] = {{3, 2}, {3, 4}}   
Output : 3

[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

The idea is to iterate through all the edges. And for each edge, mark the source node from which the edge emerged out. Now, for each node check if it is marked or not. And count the unmarked nodes.

Algorithm:

1. Make any array A[] of size equal to the  
 number of nodes and initialize to 1.  
2. Traverse all the edges one by one, say,   
 u -> v.  
 (i) Mark A[u] as 1.  
3. Now traverse whole array A[] and count   
 number of unmarked nodes.

Below is implementation of this approach:

# Python3 program to count number if sink nodes

# Return the number of Sink NOdes.

**def** countSink(n, m, edgeFrom, edgeTo):

    # Array for marking the non-sink node.

    mark **=** [0] **\*** (n **+** 1)

    # Marking the non-sink node.

**for** i **in** range(m):

        mark[edgeFrom[i]] **=** 1

    # Counting the sink nodes.

    count **=** 0

**for** i **in** range(1, n **+** 1):

**if** (**not** mark[i]):

            count **+=** 1

**return** count

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    n **=** 4

    m **=** 2

    edgeFrom **=** [2, 4]

    edgeTo **=** [3, 3]

    print(countSink(n, m, edgeFrom, edgeTo))

# This code is contributed by PranchalK

**Output:**

2

**Time Complexity:** O(m + n) where n is number of nodes and m is number of edges.

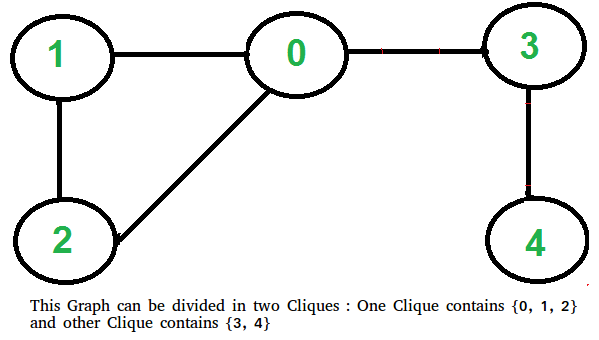
**Space complexity** : O(n) because it uses an array of size n to store the non-sink nodes.

**Two Clique Problem (Check if Graph can be divided in two Cliques)**

A Clique is a subgraph of graph such that all vertices in subgraph are completely connected with each other. Given a Graph, find if it can be divided into two Cliques.

**Examples:**

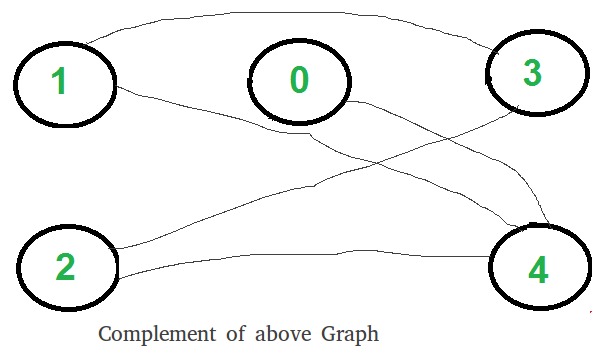
Input : G[][] = {{0, 1, 1, 0, 0},  
 {1, 0, 1, 1, 0},  
 {1, 1, 0, 0, 0},  
 {0, 1, 0, 0, 1},  
 {0, 0, 0, 1, 0}};  
Output : Yes



[Recommended: Please try your approach on ***{IDE}*** first, before moving on to the solution.](https://ide.geeksforgeeks.org/)

This problem looks tricky at first, but has a simple and interesting solution. A graph can be divided in two cliques if its complement graph is [Bipartitite](https://www.geeksforgeeks.org/bipartite-graph/). So below are two steps to find if graph can be divided in two Cliques or not.

* Find the complement of Graph. Below is the complement graph is above shown graph. In complement, all original edges are removed. And the vertices which did not have an edge between them, now have an edge connecting them.



* Return true if complement is Bipartite, else false. The above shown graph is Bipartite. Checking whether a Graph is Bipartite or no is discussed [here](https://www.geeksforgeeks.org/bipartite-graph/).

**How does this work?**

If complement is Bipartite, then graph can be divided into two sets U and V such that there is no edge connecting to vertices of same set. This means in original graph, these sets U and V are completely connected. Hence original graph could be divided in two Cliques.

**Implementation:**

Below is the implementation of above steps.

# Python3 program to find out whether a given

# graph can be converted to two Cliques or not.

**from** queue **import** Queue

# This function returns true if subgraph

# reachable from src is Bipartite or not.

**def** isBipartiteUtil(G, src, colorArr):

**global** V

    colorArr[src] **=** 1

    # Create a queue (FIFO) of vertex numbers

    # and enqueue source vertex for BFS traversal

    q **=** Queue()

    q.put(src)

    # Run while there are vertices in

    # queue (Similar to BFS)

**while** (**not** q.empty()):

        # Dequeue a vertex from queue

        u **=** q.get()

        # Find all non-colored adjacent vertices

**for** v **in** range(V):

            # An edge from u to v exists and

            # destination v is not colored

**if** (G[u][v] **and** colorArr[v] **== -**1):

                # Assign alternate color to this

                # adjacent v of u

                colorArr[v] **=** 1 **-** colorArr[u]

                q.put(v)

            # An edge from u to v exists and destination

            # v is colored with same color as u

**elif** (G[u][v] **and** colorArr[v] **==** colorArr[u]):

**return** False

    # If we reach here, then all adjacent

    # vertices can be colored with alternate color

**return** True

# Returns true if a Graph G[][] is Bipartite or not.

# Note that G may not be connected.

**def** isBipartite(G):

**global** V

    # Create a color array to store colors assigned

    # to all vertices. Vertex number is used as index

    # in this array. The value '-1' of colorArr[i]

    # is used to indicate that no color is assigned

    # to vertex 'i'. The value 1 is used to indicate

    # first color is assigned and value 0 indicates

    # second color is assigned.

    colorArr **=** [**-**1] **\*** V

    # One by one check all not yet

    # colored vertices.

**for** i **in** range(V):

**if** (colorArr[i] **== -**1):

**if** (isBipartiteUtil(G, i, colorArr) **==** False):

**return** False

**return** True

# Returns true if G can be divided into

# two Cliques, else false.

**def** canBeDividedinTwoCliques(G):

**global** V

    # Find complement of G[][]

    # All values are complemented

    # except diagonal ones

    GC **=** [[None] **\*** V **for** i **in** range(V)]

**for** i **in** range(V):

**for** j **in** range(V):

            GC[i][j] **= not** G[i][j] **if** i !**=** j **else** 0

    # Return true if complement is

    # Bipartite else false.

**return** isBipartite(GC)

# Driver Code

V **=** 5

G **=** [[0, 1, 1, 1, 0],

     [1, 0, 1, 0, 0],

     [1, 1, 0, 0, 0],

     [0, 1, 0, 0, 1],

     [0, 0, 0, 1, 0]]

**if** canBeDividedinTwoCliques(G):

**print**("Yes")

**else**:

    print("No")

# This code is contributed by PranchalK

**Output :**

Yes

***Time complexity :****O(V2)*

***Space complexity :****O(V^2),*