Graph ds

Types

(i) directed Graph : These are graph with directed edges i.e. edges have same direction

(ii) Undirected Graph: Edges are not directed

(iii) Weighted (labelled) : - Each of the edges in the graph holds some value (weight) to indicate cost of traversal through the edge.

(iv) Cyclic graph :- Graph is cyclic when one can reach it's own with traversal. A simple graph with n vertices (>=3) and 'n' edges is cyclic if all its edges form a cycle of length n.

(v) Simple graph :- Graph with no loops or parallel edges.

=> The max no. of edges possible in a simple graph with n vertices is $^nC_2$

=> No. of simple graphs possible with n vertices

$= 2^{^nC_2} = 2^{\frac{n(n-1)}{2}}$

Applications of bfs

Physical applications :-

(i) P2P network

(ii) Crawling in search engine.

Coding applications (ds/algo) ⟶ path finding

|

Cycle detection

Cycle detection

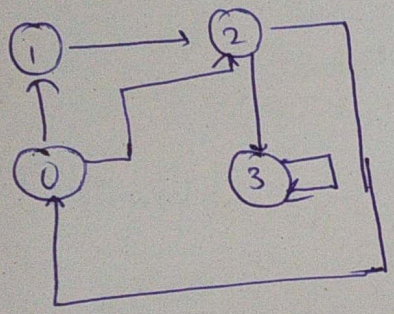
(i) Undirected Graphs
(ii) Directed Graphs

There must be a single path from one node to another i.e. a unique path from u to v, for the graph to be acyclic

i.e. If more than 1 way to visit a node from another, then there is a cycle.

If we encounter a node which is already visited in the current dfs, then there is a cycle

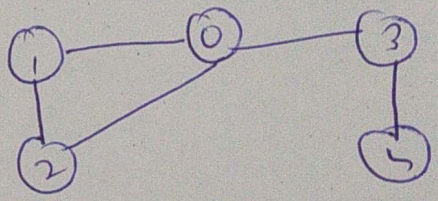
directed



there is a cycle

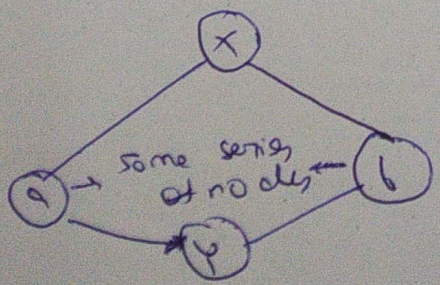
$0 \rightarrow 2 \rightarrow 0$

Undirected



0 - 2 - 1

cycle.



Cycle in Undirected Graph

Other practical applications of BFS

(i) GPS navigation :- BFS is used to find all neighbouring locations.

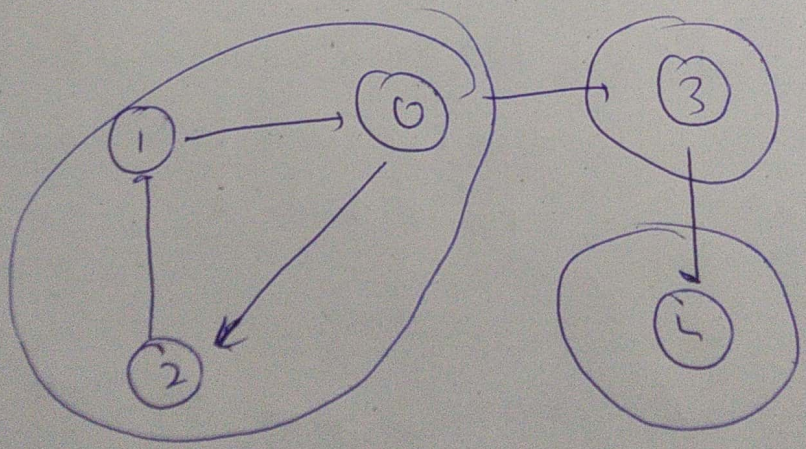
(ii) Social networking ~~devices~~ websites :- To find people with a distance 'K' from a person, we use BFS till 'K' levels.

Definitions

(i) Topological sorting

Topological sorting for directed Acyclic graph (DAG) is a linear ordering of vertices such that for every directed edge $u, v \Rightarrow$ vertex u comes before v in the ordering. Topological sorting is only possible for DAG.

(ii) Strongly connected components - A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph.



3 SCCs

Single source shortest path algorithms

(i) Dijkstra Algorithm (only works in cases where there are no negative edges ~~cycles~~)

(ii) Bellman-ford Algorithm

↓

works even for graphs containing negative ~~cycles~~ edges

Bellman ford is slower than dijkstra however it works in case of negative edges.

Also it finds -ve weight cycle in graph.

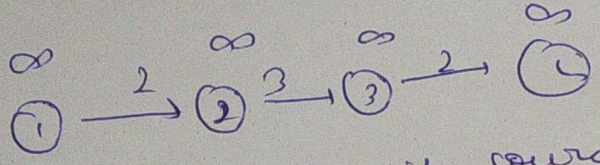
If there is -ve weight cycle in graph, then we cannot find shortest distance as we can keep going through that cycle again and again and keep reducing the weight between the 2 vertices.

```
relax ( Edge u v) {
    if (dist [v] > dist [u] + weight) {
        dist [v] = dist [u] + weight
        parent [v] = u
    }
}
```

To perform relax after on all edges V-1 times in any order.

If we perform relaxation once more with same edges and it reduces the distances then it means there is a negative weight cycle in the graph.

Why we repeat the process V-1 times



In this graph, if source vertex is 1 and destination '4', in worst case it will V-1 edges.

Now, depending on the order in which we process the edges, it might take V-1 iterations to discover 4.

3 → 4
2 → 3
1 → 2

3 iterations if edges are processed in this order.

1 → 2
2 → 3
3 → 4

} 1 iteration only required to discover 4.

↗ outer loop

So, at max V-1 iterations needed to process all edges.

All pair shortest path (Shortest distances b/w every pair of vertices in a given edge weighted graphs)

(i) Floyd Warshall algorithm

```
if (dist[i][j] > dist[i][k] + dist[k][j])
    { dist[i][j] = dist[i][k] + dist[k][j];
parent ← Path[i][j] = Path[k][j]; }
```

for every pair of vertices, do we get a shorter distance by going through K

$0 \leq K \leq V$

Time and space complexities

Graph representation

Adjacency matrix

Space complexity : $O(V^2)$

Traversal time : $O(V^2)$

Advantages :

(i) Easier to implement

(ii) Remove an edge takes $O(1)$ time.

(iii) Queries like whether there is an edge from u to v can be efficiently done in $O(1)$.

Disadvantages :

(i) Consumes more space $O(V^2)$ (even in case of sparse graph)

(ii) Adding a vertex is $O(V^2)$ time.

difference b/w Adjacency matrix and list on CS FC

Adjacency list

Space : $O(V+E)$

Advantages :

(i) Saves space $O(V+E)$

(ii) Adding a vertex is easier. $O(1)$

Disadvantages

(i) Queries like whether there is an edge b/w u to v are not efficient and can be done in $O(V)$

BFS:

Time Complexity

$O(V+E)$

$O(V+E)$ for sparse graph

$O(E)$ or $O(V^2)$ for dense graph

Explanation -

We have to first iterate through the vertices ~~in this the loop in BFS~~ in the loop and this loop in BFS is executed at most $|V|$ times. The reason is that every vertex is put into queue only once. This amounts $O(V)$.

For each vertex, we counted the edges it has counting all vertices in the graph, this count is at most $|E|$ times if G is directed or $2|E|$ if undirected.

this gives it an overall complexity of $O(|V|+|E|)$

$O(V^2)$ when adjacency matrix is used

Time Complexity

$O(V+E)$

Space $O(V)$

(for visited array)

$O(V^2)$ when adjacency ~~matrix~~ is used.

Explanation

V_1 + (incident edges) + V_2 + (incident edges)
+ --- + V_n + (incident edges)

$= (V_1 + V_2 + -- + V_n)$
+ [(incident edges V_1)
+ (incident edges V_2)
+ -- -+ (incident edges V_n)]

$O(V) + O(E)$

In case of dfs we visit each node only once and in case of no cycles we cross all edges only once

Single source shortest path (i) dijkstra algorithm

$O(V^2)$ for adjacency list representation

$O(E \log V)$ for binary heap

$O(V+E)$ for traversal (like BFS)

$O(\log V)$ for decrease Key() operation

$O((E+V) + \log V)$

$= O(E \log V)$

→ $O(E + V \log V)$ for fibonacci heap

02254

Time complexity for STL priority queue representation

N → size of priority queue

~~empty: O(1)~~
~~Pop()~~

empty(): O(1)
pop() : O(log N)
push() : O(log N)
size() : O(1)
top() : O(1)

(ii) Bellman ford algorithm

$O(VE)$

All pair shortest path algorithm

(i) Floyd Warshall → $O(V^3)$

(ii) Johnson's algorithm → $O(V^2 \log V + VE)$

↓

dijkstra for all vertices

↓

Bellman ford

Kosaraju algorithm

$O(V+E)$ time complexity

~~*~~ Tarjan's Algorithm

Johnson's Algorith

(all pair shortest path

$O(V^2 \log V + VE)$

⇒ It uses Dijkstra & Bellman ford as subroutines.

Applying dijkstra algorithm for every vertex, considering every vertex as source, we can find all shortest paths in $O(V * V \log V)$.

However dijkstra does not work for -ve weight edges.

The idea of Johnson's algo is to assign some weight to every vertex, let weight assigned to vertex v be $h(v)$. For eg. an edge (u, v) of weight $w(u, v)$ the new weight becomes $w(u, v) + h(u) - h(v)$.

⇒ All set of paths b/w any two vertices are increased by same amount.

⇒ All negative weights become non-negative

We calculate $h[]$ values using bellman ford algo

Algo :-

(i) Add some vertex S to graph, connect it to all the vertices of G with weight 0.

(ii) Run Bellman ford on G with S as source and let distances computed by Bellman ford be $h[0], h[1], \ldots$ $h[V-1]$. (there is no edge to S, only from it).

(iv) Remove the added vertex S and run dijkstra for every vertex.

Spanning Tree

Given a connected and undirected graph, a spanning tree of that graph is a subgraph which contains all vertices with minimum number of edges.

=) A spanning subgraph

=) A tree [(N-1) edges]

MST has (V-1) edges where V = no. of vertices in graph

=) A subgraph that is a tree and connects all vertices together

A minimum spanning tree or minimum weight spanning tree for a weighted, connected and undirected graph is a subgraph with weight less than or equal to the weight of every other spanning tree

↓

Total sum of weights of edges must be minimum.

Famous algo for MST

Kruskal (Greedy) $O(V \log E)$

Prim's (Greedy) $O(E \log V)$ with heap

prim's algorithm

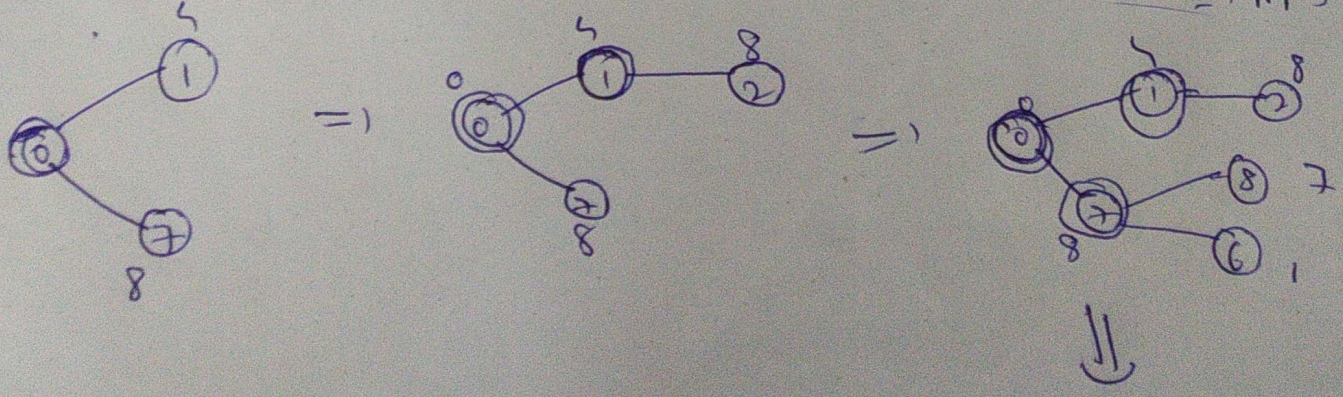
It starts with an empty spanning tree. The idea is to maintain 2 sets of vertices. The first set contains vertices already included in MST, the other set contains the vertices not yet included. At every step it considers all the edges that connects 2 sets and picks minimum weight edge from these edges.

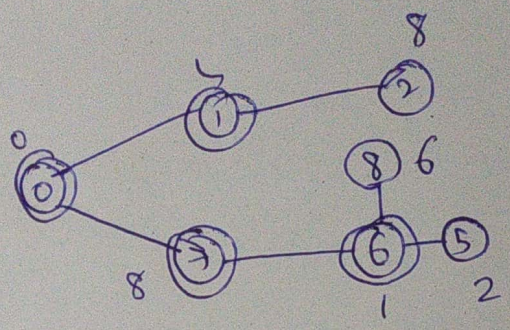
Eg.

Assign key value to all vertices in graph. initialize all key values as infinite. Assign key value as 0 for first vertex so that it is picked first.

while :
pick a vertex u which is not in mst set.
Include u to mst set
Update key values of all adjacent vertices of u.

mst : {0, INF, INF, INF, INF, INF, INF, INF, INF}

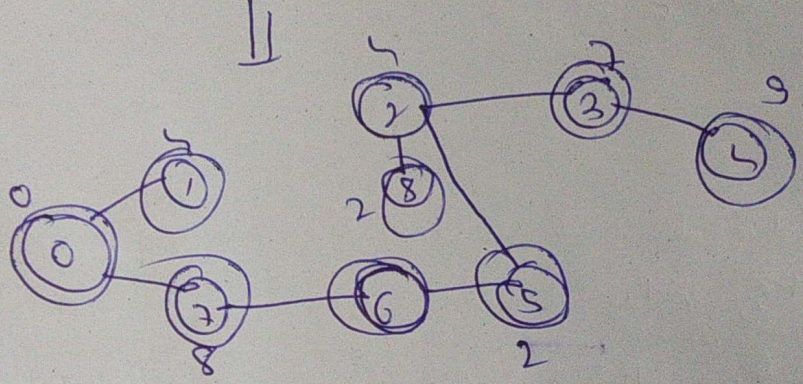




⇓

After some steps, until graph has V-1 edges

V-1 = 8 edges



Time complexity : $O(V^2)$ → Adjacency matrix representation

$O(E \log V)$ → With the help of binary heap.

more to do

SCC

(i) Tarjan Algorithm

MST

(ii) Kruskal algorithm

properties of Graph

Sum of degrees of vertices theorem

(i) Undirected graph

$G = (V, E)$

$V = \{v_1, v_2 \cdots v_n\}$

$\sum_{i=1}^{n} \deg(v_i) = 2|E|$

(ii) directed graph

$G = (V, E)$

$V = \{v_1, v_2, v_3 \cdots v_n\}$

$\sum_{i=1}^{n} \deg(v_i) = |E|$

$\therefore \sum_{i=1}^{n} \deg^-(v_i)$

Types of graphs

(i) Dense Graph :- Is a graph in which the number of edges is close to maximal number of edges.

maximum number of edges in Graph

In an undirected graph $\frac{n(n-1)}{2}$ $[nC_2]$

In a directed graph an edge may occur in both directions between two nodes

$n(n-1)$ $[2 \times nC_2]$

Why are there $\frac{n(n-1)}{2}$ edges in undirected graph

Consider n Points

→ We can draw n-1 edges from first point

→ we can draw n-2 edges from second point now that we are connected to first point

Sum $= (n-1) + (n-2) + \cdots + 3 + 2 + 1$

$= \frac{n(n-1)}{2}$

In a dense graph $E = O(V^2)$

जबलपुर मंडल
JABALPUR DIVISION

(ii) Sparse graph

It is a graph in which the number of edges is close to the minimal number of edges

$$E = O(V)$$

The minimum number of edges for undirected and connected graph with n vertices is $(n-1)$ edges

~~Ex~~

=> Since graph is connected there must be a unique path from every vertex to every other vertex and removing any edge will make graph disconnected.

Representation used for different types of Graphs

(i) Sparse Graph : If we use matrix representation for most of the matrix cells remain unused which leads to wastage of memory. Thus we prefer adjacency list.

(ii) But if graph is dense number of edges is close to $\frac{n(n-1)}{2}$ [complete] or n^2, then there is no advantage of using adjacency list over matrix.

Complete Graph

A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge. A complete digraph is a directed graph in which every pair of distinct vertices is connected by a pair of unique edges.

A complete graph with n vertices has

$$\frac{n(n-1)}{2} \text{ edges}$$

Types of edges in a graph

(i) Tree edge :- It is an edge which is present in a tree obtained after applying DFS on graph.

(ii) Forward edge :- It is an edge (u, v) such that v is descendent but not part of DFS tree.

(iii) Back edge :- It is an edge (u, v) such that v is ancestor of edge u but not part of DFS tree.

(iv) Cross edge :- It is a edge which connects two nodes such that they do not have any ancestor and a descendant relationship between them.

Hamiltonian Cycle

Hamiltonian path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle is a hamiltonian path such that there is an edge from last vertex to the first vertex of Hamiltonian path.

Articulation point :- A vertex in an undirected connected graph is an articulation point if removing it (and edges through it) disconnects the graph. Articulation point represents vulnerabilities in a connected network - single point where failures would split the network into 2 or more components.

Bridge graph :- An edge in an undirected connected graph is a bridge if removing it disconnects the graph. For a disconnected undirected graph, definition is similar, a bridge is an edge removing which increases number of disconnected components.