Graphs 1

Sunday, 12 November 2023 10:06 PM

1) Directed Graphs → edges are unweighted
2) Undirected Graphs → edges and weighted

Strongly connected Components (directed graphs)
↳ space should be a path b/w any two nodes

[Simple Graphs]
1) no self loop
2) no multiple edges

Graphs Representation

1) Adjacency matrix

- adj matrix for an undirected graph is a transpose of itself

unweighted → 1/0 to indicate (2 or 0)
weighted → 1/0 are weights

Pros of Adj Matrix
O(1) edge check, add or remove an edge

Cons
space required is large
Sparse matrix are often problem
Adj/remove nodes is expensive

2) Adjacency List

SC = O(V+E) → for undirected
: O(V+E) → for directed

- **Breadth First Search (BFS)**

Explore are breadth first.
→ neighboring nodes first

- **Depth First Search (DFS)**

Explore until nodes first
→ go as many consecutive neighbour successively

→ queue → out : queue order in out Importantly.

DFS is preferred using BFS easy to code.

Recap — 18 Nov 2023 6:34 PM

- Graphs can be:
  - Weighted & Unweighted
  - Directed & Undirected
  - Cyclic Graph
    If we have a simple path from a node to itself via Distinct edges.

In a CC → there have a path b/w any two nodes.

- For Directed graphs, we calculated concept in Strongly Connected Component
  which mean every node has a path from everything node.

- Simple Graphs
  - no self loop
  - no multiple edges

- How to Represent Graphs in code

Directed Graphs

- In an set of undirected edges b/w 2 nodes
  $= {}^{V}C_{2}$

$\sim O(V^2)$

- Max # of directed edges
  2 nodes → $2 \times {}^{V}C_{2}$

$\sim O(V^2)$

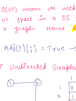
O(V²) means we need at least space in a DS to store a graph. hence **adj matrix**

$Adj[i][j] = True \rightarrow edge\ b/w\ i, j$

Undirected Graphs

$Adj[i][j] = Adj[j][i]$

- Some problems are usually given in terms of numerical nodes. 1 → n.
  If symmetric DS is given as nodes, create a mapping for those values.



{ P: 1
D: 2
C: 3 }

- For a weighted graph, simply fill weights in one that cell.
  - Fill zero for no edges.
  - Or create another matrix for edges.

D: [0, 6, 5] SC: O(V²)

→ Adj mat pros:
  1) check edge → O(1)
  2) Add edge → O(1)
  3) Remove edge → O(1)

→ Cons
  1) large space Complexity O(V²)
  2) one mat is usually sparse
  3) Adding a node is a lot of overhead

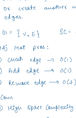
Adjacency List

1 → 2
2 → 1, 3, 4
3 → 2, 4
4 → 2, 3

- One input for graph questions is generally given as edges (2 parts)

SC = O(V+2E)
or SC = O(V+E)

- Directed



1 → 3
1 → 2
3 → 2
2 → 4

SC= O(V+E)

→ 1 : adj[1].add(2)

- Given an undirected unweighted graph find a source node S that has the shortest distance from (C) in every other node.

Question weighted nodes are LCS nodes many from cannot (1+3)

In contrast that we need to implement BFS here. (Count as a level)

Only count adding to set if we are adding using FIFO in BFS.

Adj[1] = Adj[1].add(2)
Adj[2] = Adj[2].add(1)

TC → for every node, we are traversing over the adjacency list

→ TC= O(V+2E)
: O(V+E)

SC= O(V)
+
O(V+E, Adj)

we are count the Adj as the no. space we are storing in the queue.

- DFS : Depth First Search

Adj[1]
Adj[2] = True
↳ [-1, -1, -1, -1, -1]
If not vis[1]:
Adj[1]
vis[1] = True

→ Forward method to traverse using graph.
→ Enable read from that graph
→ Backward not to read from graph.
→ Backward not to read also in Recursion

TC = O(V+E)

- Given a graph, Return a list of one of nodes in each CC

Adj[1] → True
Adj[2] → True
Adj[3] → True

→ we can call dfs only only once to get CC.
But where calls should come from.

- Given a graph, Return a list of one of nodes in each CC

Adj[1] → dfs(1) called 3 times (or times) [-1, -1, -1, -1, -1]
Adj[2] → called and count count count
Adj[3] → 3 times

Adj[1] → 3

3 [-1, -1, -1, -1, -1]
If not vis[1]:
count = 1
Adj[1]
Adj[2] count = 2
Adj[3] count = 3

Adj[V]:
count = 0
If not vis[1]:
count += 1
Adj[1]
If not vis[2]:
count += 1
Adj[2]

TC = O(V+E) SC= O(V)