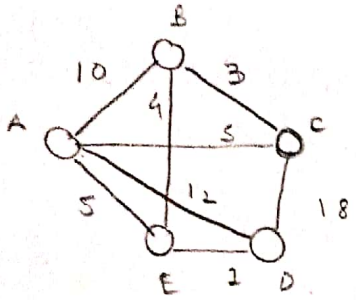
Day 39 Graphs 2

Shortest path: i) When edges have same weights & graph is bidirectional

ii) edges could have weights <= k (where k is very small), graph is bidirectional.

iii) lifting restriction on k, graph is still bidirectional



↓

Dijkstra's algorithm

↓

similar strategy to BFS

↓

picking the node at the smallest distance from the current node.

(Basic intuition) ← it assumes that popped element's min distance has been reached.

↑ Doesn't work with -ve weights.

(max edges possible)

T: O(E log E) ≤ O(E log V²)

S: O(E)

visited = { }, distance = ∞ (to sort on)

q = min heap

q.push(make_pair(0, 0)); (distance, node)

while(!q.empty()){

top = q.front();

q.pop();

dist = top.first();

node = top.second();

(if node is repeated)

if(visited[node]) continue;

visited[node] = T;

for(neighbour in node.neighbours):

if(visited[neighbour]) continue;

if((dist + edge) < distance[neighbour]) {

(the distance for the current node)

(edge between node & neighbour)

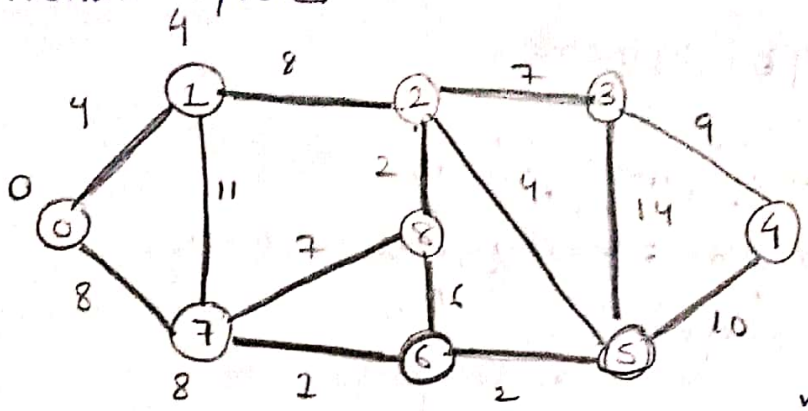
q.push(make_pair(dist + e, neighbour));

distance[neighbour] = dist + e;

}

when we are popping an element, we are saying we would achieved the least amount of distance

Djiketra's dry run



Same node may get added to the priority queue, but will do with different weights & will get processed for the min pose.

queue :

1 0,0
2 4,1
3 8,7
6 12,2
4 9,6
8 15,8
5 11,5

21,4
7 25,3
14,8
9 19,3

visited : 1 1.

↓

{ 0 }.

↓

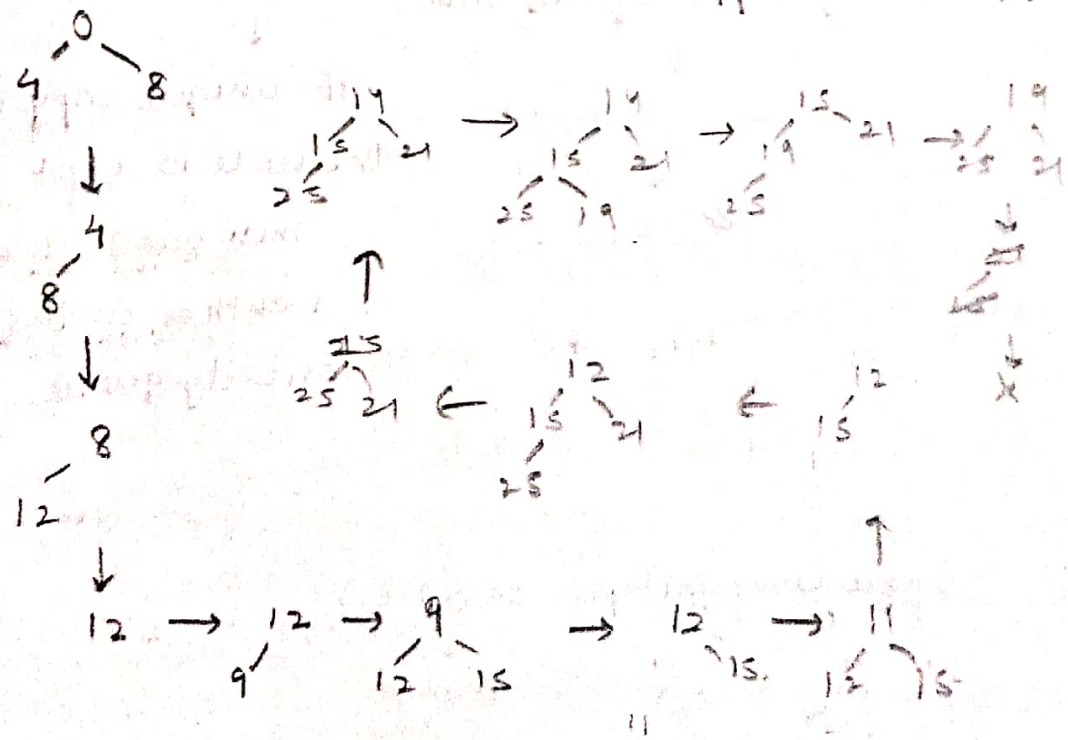
{ 0, 1, 7, 6, 5, 2, 8, 3, 4 }

0	4	12	19	21	11	9	8	14

top = (0,0) (9,6) (14,8)
(4,1) (11,5) (15,8)
(8,7) (12,2) (19,3)

distance :

0	1	2	3	4	5	6	7	8
0	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~	~~∞~~
	4	12 ~~8~~ 12 19	~~21~~	11	9	8	~~15~~ 14	



Time complexity of Dijkstra's

query, priority queue : O(1)

insertion, priority queue = O(log E)

(upper bound)

a node at max can be pushed to queue : deg. of adjacent nodes or degree times.

(disconnected nodes will not be considered)

(+ ∞ will be the distance)

summation : O(E)

the operation on the priority queue.

total no. of pushes to queue at max O(E)

~~O(E log E)~~ ← every push op : O(log E)

since a vertex can be push multiple times.

E ≈ V² (in worst case)

O(E log V²)

O(2E log V)

~~O(E log V)~~

if unique copy of the node is kept (the min one), then V entries in the priority queue

O(E log V)

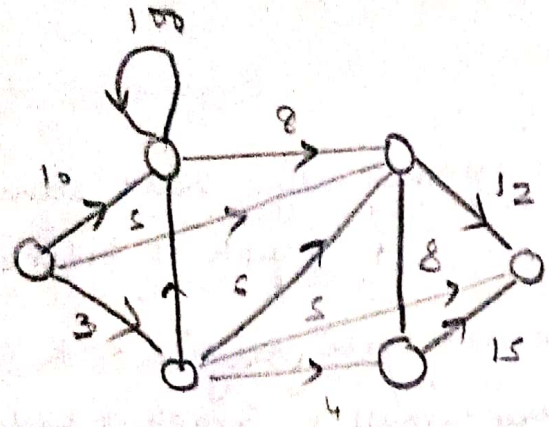
space complexity : O(V + E)

visited array

min heap size

Q.

cycle detection using Dijkstra's in a directed graph



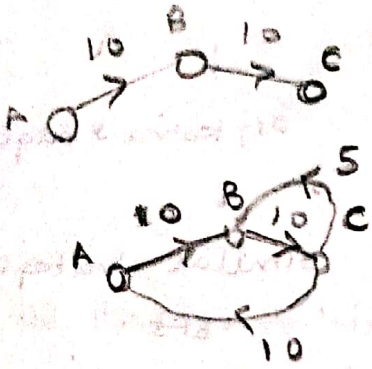
Detect cycle in graph, and the minimum weight cycle in the graph

↓

Edges in the graph are directed.

only

(1) hashing to detect cycle



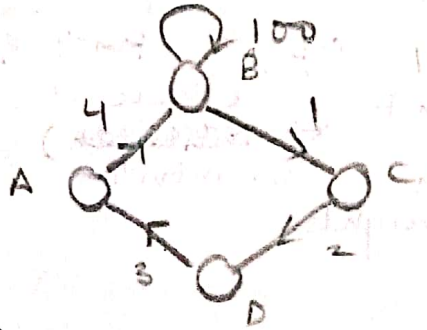
Dijkstra's: min distance between the starting node & other nodes.

↓

Dijkstra's for each node, to calculate the distance to the same node.

↓

The minimum distance out of all these would be the answer



min distance cycle:

A → B → C → D

A

rather than A → A

visited at first pop not be marked for each node, and don't mark the distance for start node as 0, rather make it ∞.

use a flag maybe ←

↓

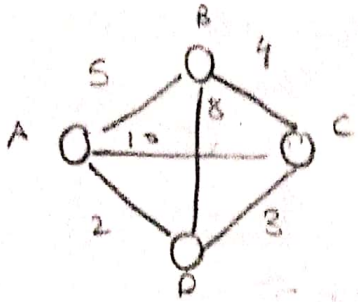
At the end if all distances are ∞ then no cycles.

* Time complexity

V * E log V → E ← (V, V^2)

$[V^2 \log V \;,\; V^3 \log V]$

Shortest path for every pair of nodes in a graph



(pair wise shortest distance)

↓

Floyd warshal's algorithm

↓

$O(V^3)$

↓

uses dynamic programming

Algorithm:

Base condition: initialize a $V \times V$ (prev) matrix, where if there is an edge between i, j then the edge weight is its corresponding value, else mark the values as $+\infty$.

can be done using two matrices → prev, current as well.

↓

extra $O(N)$ is not really req.

Now, for K → 1 to V:

for i → 1 to V:

for j → 1 to V:

current[i][j] = min(prev[i][j], prev[i][k-1] + prev[k-1][j]);

* at any iteration (min distance with atmost k intermediate nops)

{the current distance ← and taking distance as the current node intermediate ?

return dist[N] → shortest distance between any pair of nodes.

↓

At any point of time, no. of edges between i & j will never be more than v.