SoC 2022 : Competitive Coding

Week-3

Graphs and BFS, DFS

Contents

Graph Terminology

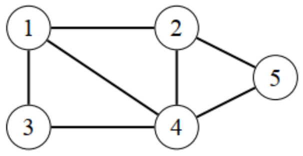


Fig. 7.1 A graph with 5 nodes and 7 edges

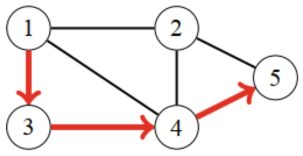


Fig. 7.2 A path from node 1 to node 5

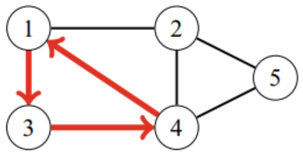


Fig. 7.3 A cycle of three nodes

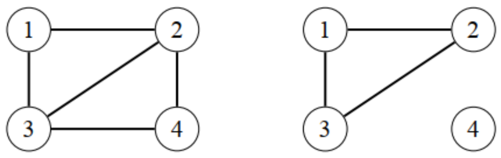


Fig. 7.4 The left graph is connected, the right graph is not

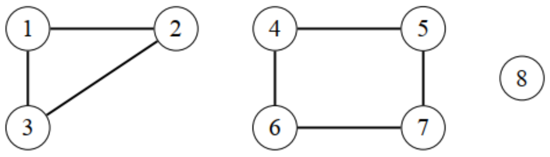


Fig. 7.5 Graph with three components

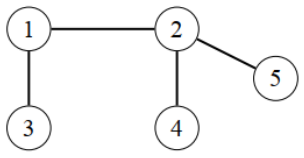
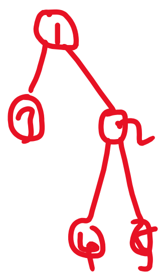


Fig. 7.6 A tree



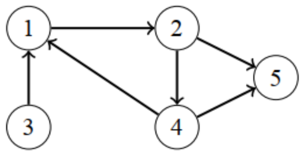


Fig. 7.7 Directed graph

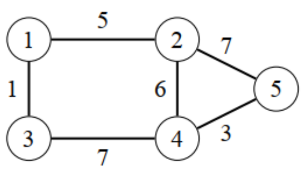
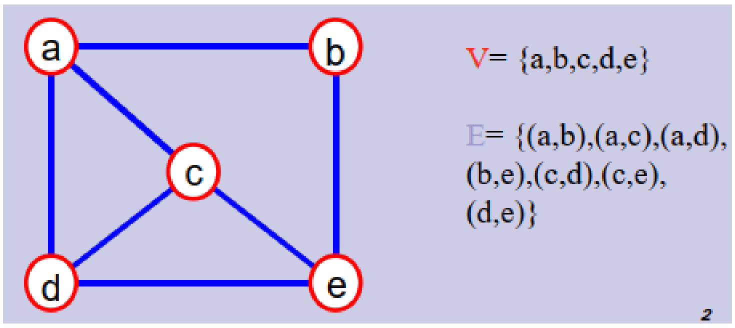


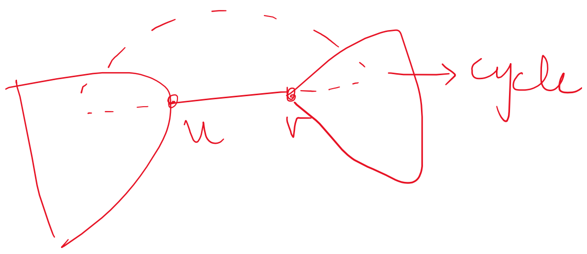
Fig. 7.8 Weighted graph

- G = (V,E)
- Nodes (vertices). Labelled, e.g., 1,2,…,n. Set of vertices = V, e.g. {1,2,…,n}
- Set of edges = E, e.g. E={(1,2),(2,5),…}.



- Adjacent vertices: vertices connected by an edge.
- Degree of vertex = #edges incident on it = #vertices adjacent to it.
  In-degree, out-degree in case of directed graphs.
- We are not considering multiple edges between 2 vertices, and also not considering self-loops here. Such graphs are called **simple graphs**.
  So, if n vertices and m edges, then $0 \leq m \leq {}^{n}C_2 = \frac{n(n-1)}{2}$.
- Path from vertex u to vertex v. [diagram↑]
  Sequence of vertices v1,v2,…,vk such that consecutive vertices vi and vi+1 are adjacent.
- Length of a path (#edges in this path)
- Simple path (no cycle) versus Cycle
- Connected graph, connected components

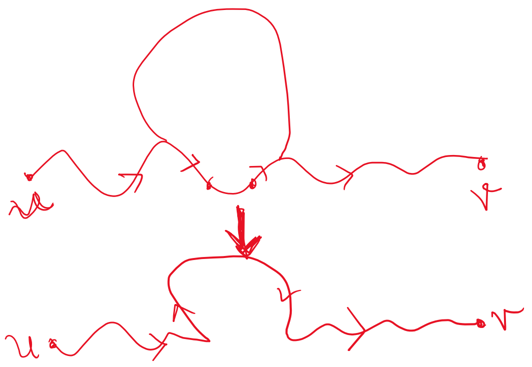
- Tree: connected graph with no cycles [diagram ↑]
  Properties:
  - Tree with n nodes has exactly (n-1) edges.
    Proof by induction:
    Consider any edge (u,v) in the tree. Remove this edge.
    We get exactly 2 components: cannot have the entire graph to be connected even after removing an edge. If this is the case, then the original graph has a cycle. Contradiction to the fact that the original graph was a tree.
    Cannot have more than 2 connected components, because the only change made was: removal of (u,v). If we have more than 2 connected components, then edges other than (u,v) must also have been removed. Contradiction.



Let the number of vertices in the 2 connected components (say c1,c2) be n1,n2. We have n1+n2 = n.
Since c1,c2 are connected (because exactly 2 conn. comp.), and no cycle can be introduced by removing an edge, so c1,c2 are **trees**.
Now, by induction hypothesis, #edges in c1 = n1-1, #edges in c2 = n2-1.

Total edges finally = n1+n2-2.
Total edges initially = n1+n2-2 + 1 = n1+n2-1 = n-1.
Done.

- Any **connected** (i.e., all vertices have at least one edge incident on them) graph on n nodes and n-1 edges is a tree.
  [note that multiple edges between 2 vertices, and self-loops aren't being considered here, otherwise you can have n-1 edges b/w 2 vertices only!]
  Proof: Show that there can't be any cycle.

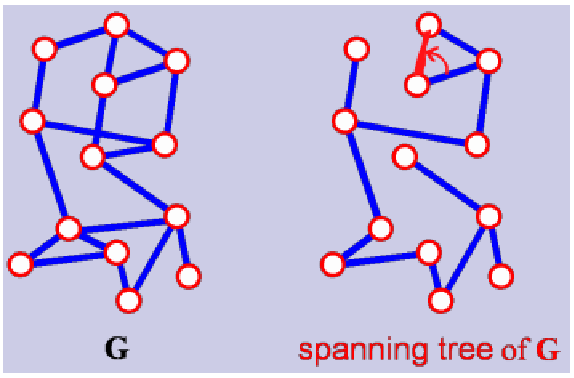
- Any graph with n vertices, and < n-1 edges is disconnected.
  Proof:
  This graph isn't a tree (because we know that tree on n nodes has exactly n-1 edges).
  So, why is it not a tree?
  There can be 2 reasons:
    - Not connected, and/or
    - Has a cycle
  Assume that it is connected. Then, why is it not a tree?
  Must have a cycle (or cycles). So, remove some edge from one cycle. Repeat this process for any remaining cycles. Even then, it will remain connected.
  

  Then, we have a case where we had < n-1 edges, and then we removed some edge(s), and then the final graph is connected, and has no cycles.
  So, we have reached at a **connected** graph (n nodes) with **no cycles**. This is a tree, by definition. But it has < n-1 edges. Contradiction.

- Between any 2 vertices of a tree, there's a **unique** path. [If there are ≥2 paths, then can clearly see a cycle.]

- Rooted tree. [Parent, Children]
- Leaves of a tree : vertices with degree 1.

- A spanning tree of G is a subgraph which is a tree and which contains all vertices of G.



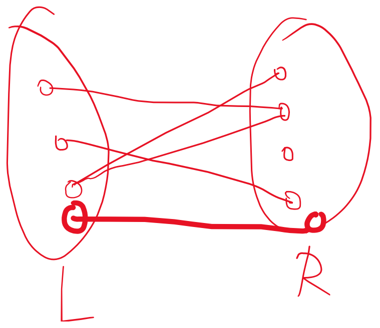
(not unique, at least here)

- Directed Graphs: edges have directions; diagram at the top ↑ (e.g., one-way roads)
- Weighted Graphs: edges have some weights assigned to them
  e.g., consider a road map – the lengths of roads b/w 2 cities can be mentioned as weights.

Some special types of graphs:

- Complete graph:
  Any two vertices have an edge between them. So, #edges = ${}^{n}C_2 = \frac{n(n-1)}{2}$.

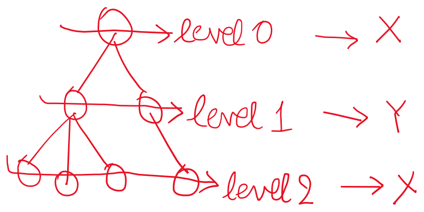
- Bipartite Graph:



$G = (V, E)$
$V = L \cup R$

No edge (u,v) such that both u,v in L, or both in R.

Someone pointed this out in the session: Any tree is bipartite.
A simple proof can be by rooting the tree at any vertex. Then we will have vertices in "levels".



Since there's no cycle, can see that putting vertices in alternate levels, into sets L and R, works.

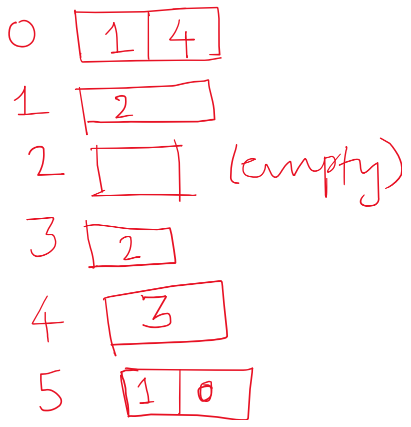
Graph Representation

- **Adjacency Lists**: (Quite popular) In the adjacency list representation, each node v of the graph is assigned an adjacency list that consists of nodes to which there is an edge from v.

$$\begin{tikzcd}
 & 1 \arrow[r] & 2 \\
0 \arrow[ur] \arrow[drr, bend right] & 5 \arrow[u] \arrow[l] & 3 \arrow[u] \\
 & & 4 \arrow[u]
\end{tikzcd}$$

$$\begin{tikzcd}
 & 1 \arrow[r, "1"] & 2 \\
0 \arrow[ur, "2"] \arrow[drr, bend right, "4"'] & 5 \arrow[u, "9"'] \arrow[l, "5"'] & 3 \arrow[u, "8"'] \\
 & & 4 \arrow[u, "7"']
\end{tikzcd}$$

Can use array/vector of vectors:
See code adj-list.cpp.



For weights, can use an array/vector of vectors of pairs.
See code adj-list-wgts.cpp.

- **Adjacency Matrix**: An adjacency matrix indicates the edges that a graph contains.
bool adj[n][n];
where adj[i][j] is 1 iff there's an edge i→j.
Can use int adj[n][n] to store weights too.

	0	1	2	3	4	5
0	0	1	0	0	0	0
1	0	0	1	0	0	0
2	0	0	0	0	1	0
3	0	0	1	0	0	0
4	0	0	0	1	0	0
5	1	1	0	0	0	0

Drawback: O(n^2) space.
For adjacency list, can do with space ~ O(m), where m = no. of edges. If the graph doesn't have many edges (e.g., see the number of 0s in the above matrix), then this is a wastage. So, this representation cannot be used if the graph is large.

- **Edge List**: An edge list contains all edges of a graph in some order. This is a convenient way to represent a graph if the algorithm processes all its edges, and it is not needed to find edges that start at a given node.

  ```
  vector<pair<int,int> > edges;
  edges.push_back({0,1});
  and so on…
  ```

  Can store weights as

  ```
  vector<pair<pair<int,int>,int> > edges;
  edges.push_back({{0,1},2});          // 0→1, with wgt 2
  ```

Graph Traversal

- We will discuss BFS (breadth-first-search) and DFS (depth-first-search).
- Both algorithms are given a **starting node** in the graph, and they visit all nodes that can be reached from the starting node. The difference in the algorithms is the order in which they visit the nodes.

- NOTE: considering undirected graphs here.

- **Depth First Search (DFS)**
  Depth-first search always follows a single path in the graph as long as it finds new nodes. After this, it returns to previous nodes and begins to explore other parts of the graph. The algorithm keeps track of visited nodes, so that it processes each node only once.

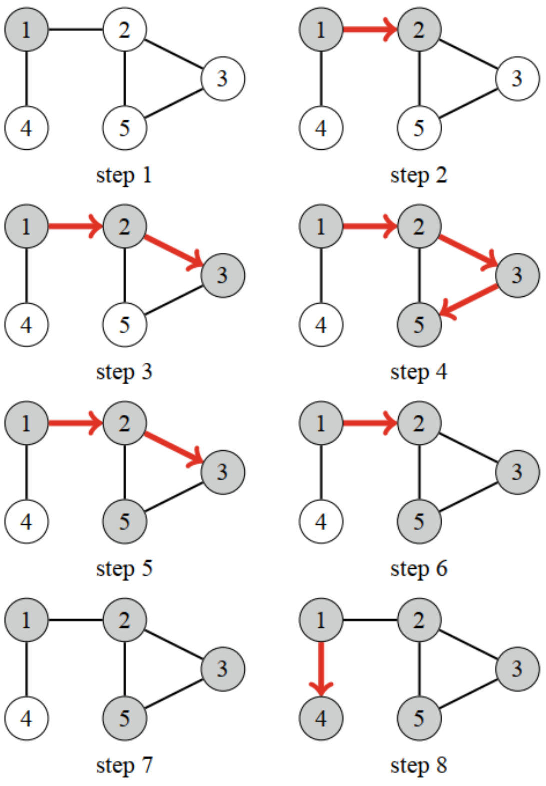


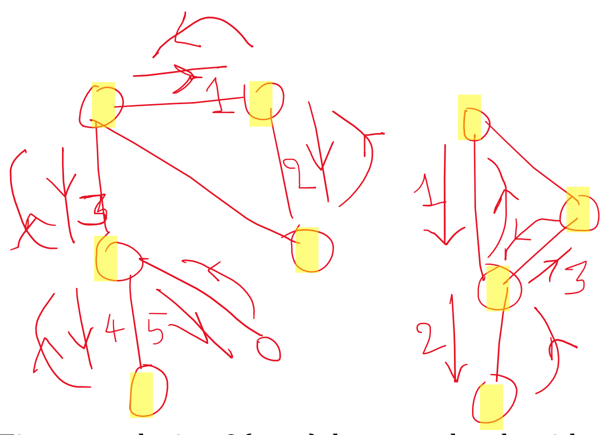
Fig. 7.13 Depth-first search

Can be conveniently implemented using recursion.
See code 3_dfs.cpp.

```
5 5
1 2
1 4
3 5
2 5
3 2
1 0 5 0
2 1 4 1
3 3 4 5
4 4 5 1
5 2 4 2
```

Can see different arrival and departure times, and parents (of 5,3) from above diagram. Both are correct DFS runs.

It deals with multiple connected components too.



Time complexity: O(m+n), because the algorithm processes each node and edge once.

- **Breadth First Search (BFS)**
  - BFS visits the nodes of a graph in increasing order of their distance from the starting node.
  - Thus, we can calculate the **distance** (length of **shortest** path) from the starting node to all other nodes using breadth-first search.
  - Note that BFS is guaranteed to give correct shortest distances only in case of **unweighted** graphs. This is same as saying that all edges have equal weights, so WLOG can take them all to be 1.
  - For weighted graphs, we have other algorithms.

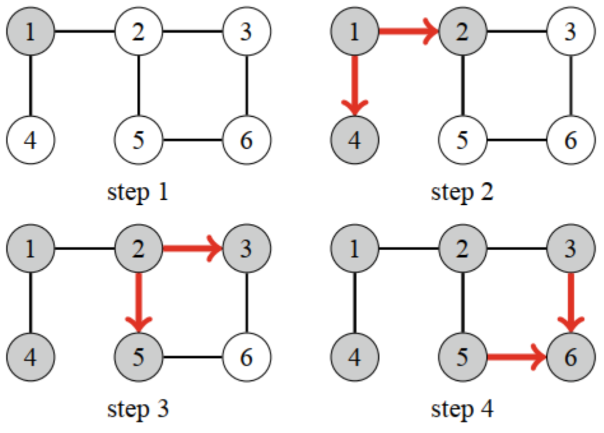


Fig. 7.14 Breadth-first search

Implementation: use a queue.
See 4_bfs.cpp. Shortest distances have been computed.
For DFS, a stack is used. The recursion basically models the stack as a stack of function calls.
e.g. (previous diagram)

dfs(5)
dfs(3)
dfs(2)
dfs(1)
main()

Applications

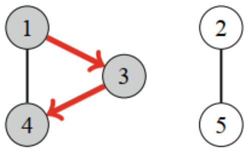
[here, considering undirected graphs]

If both traversals are applicable, usually DFS is preferred, since it is easier to implement.

- **Connectivity Check**

  A graph is connected if there is a path between any two nodes of the graph. Thus, we can check if a graph is connected by starting at an arbitrary node and finding out if we can reach all other nodes.

  Fig. 7.15 Checking the connectivity of a graph

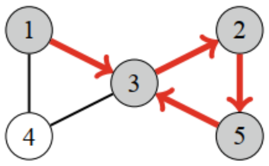
  

  Method:
  - Do DFS from any 1 vertex, and after this function, check if all nodes have been visited (using the visited array/vector).
  - Graph is connected iff all nodes have been visited by a single BFS/DFS.

- **Cycle Detection**

  A graph contains a cycle if during a graph traversal, we find a node whose neighbor (other than the previous node in the current path) has already been visited.

  Fig. 7.16 Finding a cycle in a graph

  Code: 5_cycle-check.cpp.

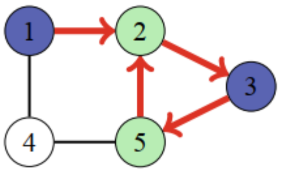
- **Bipartiteness Check**

  Recall definition.

  Equivalently, a graph is bipartite if its nodes can be colored using two colors so that there are no adjacent nodes with the same color; i.e., it is "2-colorable".

  Easy to check using BFS/DFS.

  Fig. 7.17 Conflict when checking bipartiteness

  - Pick a starting vertex. Color it "X". Color all its neighbors "Y".
  - Recurse: if some vertex has color X, then color its neighbors Y, and vice versa.
  - If at any point we encounter a node, which we colored X(Y), and its neighbor has already the same color X(Y), then declare the graph non-bipartite.

Shortest Paths

- We need to find shortest path between some 2 vertices.
- In an unweighted graph, the length of a path equals the number of its edges, and we can simply use breadth-first search to find a shortest path.
- Now dealing with weighted graphs.

- A point to note: If G has <u>no negative cycles</u>, then there is a shortest path between any 2 vertices that is simple (i.e., does not repeat nodes), and hence has at most $n - 1$ edges.
  Proof: Suppose we need to compute the shortest path $s \to t$. Since every cycle has nonnegative cost, the shortest path P from s to t with the fewest number of edges does not repeat any vertex v. If P did repeat a vertex v, we could remove the portion of P between consecutive visits to v, resulting in a path of no greater cost [← cycle weight ≥ 0 used here] and fewer edges.

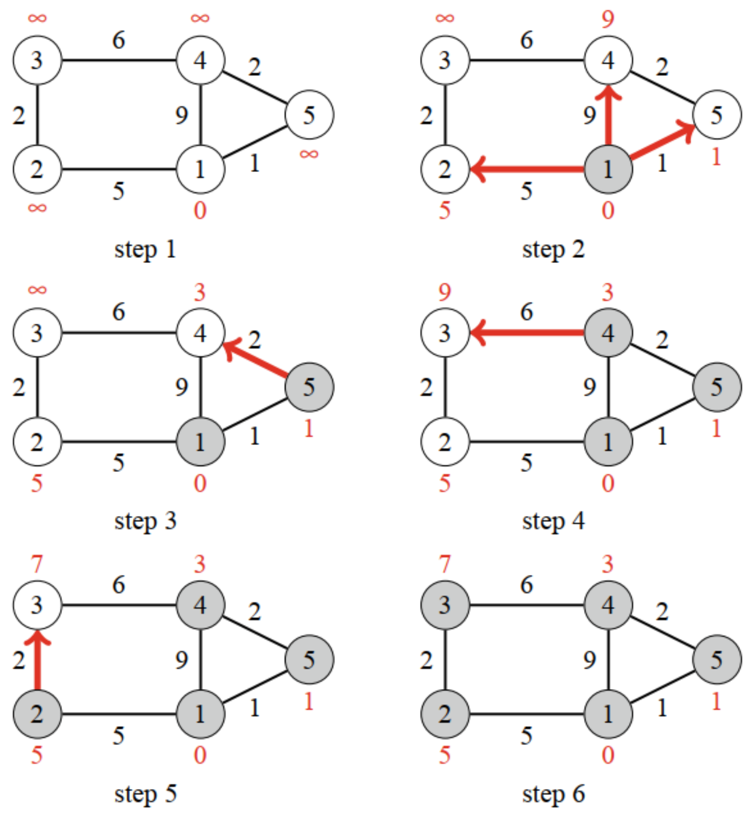
1. Dijkstra's Algorithm [Single Source Shortest Paths]

Finds shortest paths from a given node (source) to **all** vertices.
This algorithm requires that there are <u>no negative weight edges</u> in the graph.

Consider two sets of vertices. S and V\S.

- At each step, Dijkstra's algorithm selects a node "v" that has not been processed yet (i.e., a node from V\S) and whose current distance is as small as possible. Then, the algorithm goes through all edges that start at the node and reduces the distances using them. After this processing, "v" is put into set S.
- This is a greedy strategy, since we are making locally optimal decisions. [just checking current distances, and modifying S.] A proof of correctness can be found in CLRS Section 24.3.
- All vertices in S [i.e., all processed ones] have "final" smallest distances from the source (at any point of the run).

In the step 1 below, S=ϕ. In step 1, S={1}, V\S={2,3,4,5,6}.

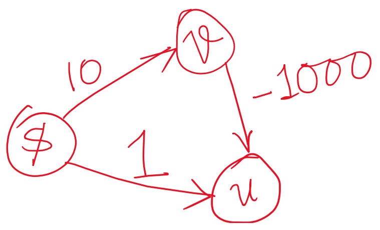


Implementation: We need to search for this "v" efficiently. "v" is an unprocessed node, that currently has the least distance. Use a priority_queue (~ min heap, it has the least element at its top).

Time complexity: O(n log n) [log n due to use of min heap / priority_queue]
In each iteration, one vertex removed from V\S, and added to S.

See code and comments in 9_dijkstra.cpp.

Note that the "path" itself can be obtained by keeping track of the edges corresponding to the vertex movement from V\S to S. These edges result in a <u>tree</u>. This tree can be used to obtain the (unique) paths.



$s \rightarrow u$: wrong min. dist.

2. Bellman-Ford Algorithm

- As Dijkstra's algo, this also finds shortest paths from a given node (source) to **all** vertices.
- However, can handle negative weights also [in directed graphs].
- Note that if there is a negative cycle in the graph, then the concept of shortest paths breaks down.

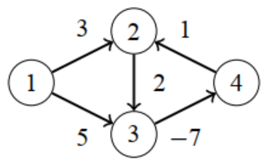
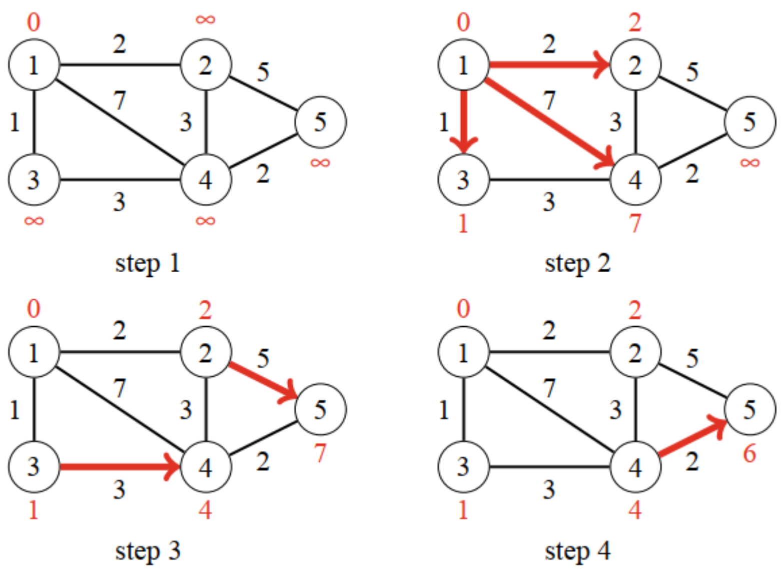


Fig. 7.19 A graph with a negative cycle

Can see that (shortest) distance from 1 to 4 keeps on decreasing if we keep on traversing the cycle 2→3→4→2…, as it has a negative total weight.
If we have an undirected graph with a negative edge, then it is same as a negative cycle in a directed graph.

- Bellman-Ford can detect such cases.



- The algorithm consists of $n-1$ rounds, and on each round, the algorithm goes through all edges of the graph and attempts to reduce the distances. We can show that if there's no negative cycle, then the distance values will stabilize in <u>at most</u> $n-1$ rounds.
  If in some round we find that no dist[v] has been modified (for any v∈V), then we can break out of this loop.
- For detecting negative cycles, let the loop run for n rounds. If in the n^{th} round too, any dist[v] changes, then declare that a neg. cycle is present.
- Time complexity: O(m*n) [in each of the n rounds, all m edges are being considered].

- See code 6_bellman-ford.cpp.

3. Floyd-Warshall Algorithm
[APSP: All Pairs Shortest Paths]

- The algorithm maintains a matrix that contains distances between the nodes. The initial matrix is directly constructed based on the adjacency matrix of the graph.
- Then, the algorithm consists of consecutive rounds, and on each round, it selects a new node that can act as an intermediate node in paths from now on, and reduces distances using this node.

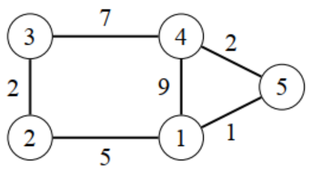


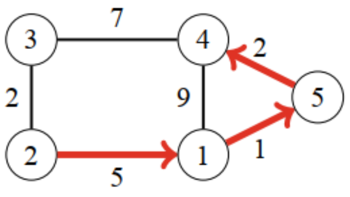
Fig. 7.22 Input for the Floyd–Warshall algorithm

-

$$\begin{bmatrix} 0 & 5 & \infty & 9 & 1 \\ 5 & 0 & 2 & \infty & \infty \\ \infty & 2 & 0 & 7 & \infty \\ 9 & \infty & 7 & 0 & 2 \\ 1 & \infty & \infty & 2 & 0 \end{bmatrix}$$

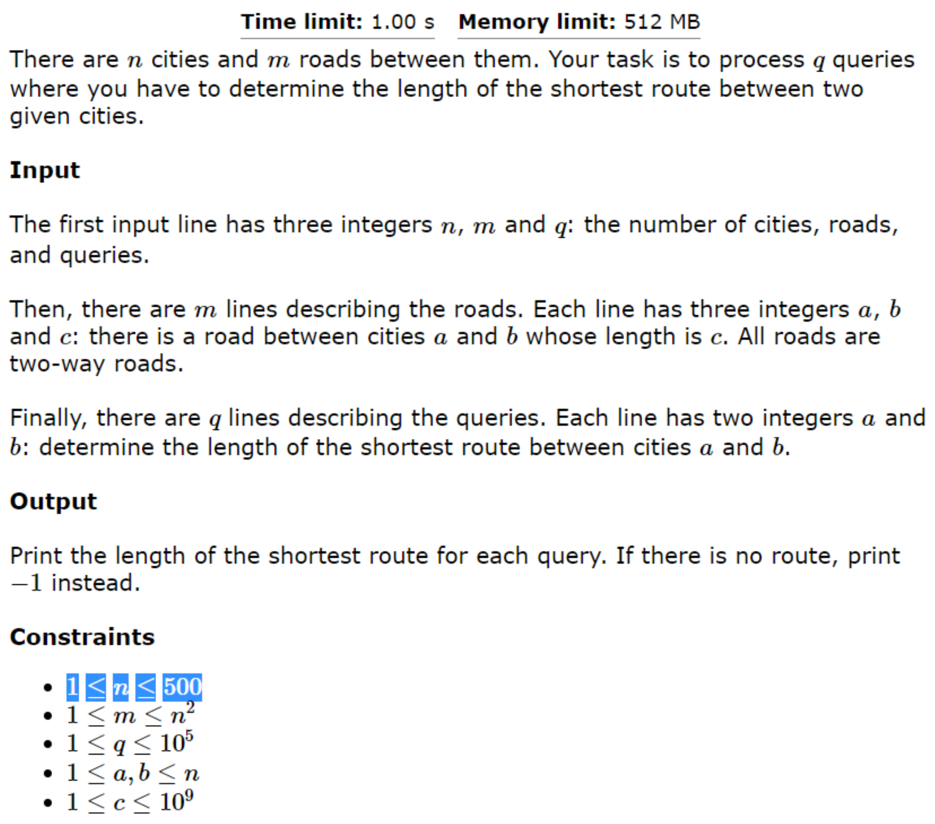
Initial adjacency matrix [with weights of *direct* edges]

On the first round, node 1 is the new intermediate node. There is a new path between nodes 2 and 4 with length 14, because node 1 connects them. There is also a new path between nodes 2 and 5 with length 6.



$$\begin{bmatrix} 0 & 5 & \infty & 9 & 1 \\ 5 & 0 & 2 & \mathbf{14} & \mathbf{6} \\ \infty & 2 & 0 & 7 & \infty \\ 9 & \mathbf{14} & 7 & 0 & 2 \\ 1 & \mathbf{6} & \infty & 2 & 0 \end{bmatrix}$$

And so on.



Time limit: 1.00 s **Memory limit:** 512 MB

There are n cities and m roads between them. Your task is to process q queries where you have to determine the length of the shortest route between two given cities.

Input

The first input line has three integers n, m and q: the number of cities, roads, and queries.

Then, there are m lines describing the roads. Each line has three integers a, b and c: there is a road between cities a and b whose length is c. All roads are two-way roads.

Finally, there are q lines describing the queries. Each line has two integers a and b: determine the length of the shortest route between cities a and b.

Output

Print the length of the shortest route for each query. If there is no route, print -1 instead.

Constraints

- $1 \leq n \leq 500$
- $1 \leq m \leq n^2$
- $1 \leq q \leq 10^5$
- $1 \leq a, b \leq n$
- $1 \leq c \leq 10^9$

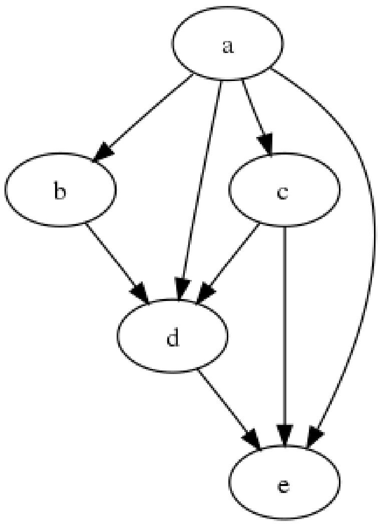
O(n^3) works here.

See code: shortest-paths-2.cpp.

Directed Acyclic Graphs

Aka DAGs.

- As the name suggests, **directed**, and no (directed) **cycles**

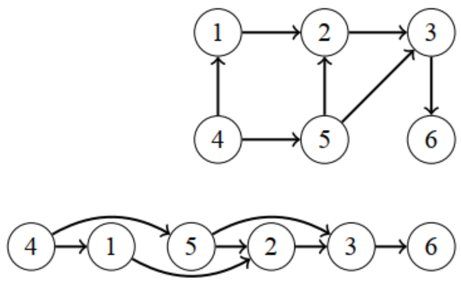


-
- Many problems are easier to solve if we are given this. In particular, we can always construct a topological sort for the graph.

Topological Sort

- A topological sort of a DAG G=(V,E) is a linear ordering of all its vertices such that if G contains an edge u→v, then u appears before v in the ordering. (If the graph contains a cycle, then no linear ordering is possible.)
- We can view a topological sort of a graph as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.

Fig. 7.24 Graph and a topological sort



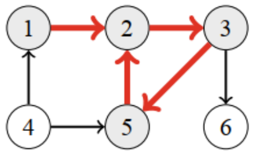
-
- Topological sort isn't necessarily unique.

$$\begin{tikzcd} 4 \arrow[r] \arrow[rr, bend left] & 5 \arrow[rr, bend right] \arrow[rrr, bend right] & 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 6 \end{tikzcd}$$

is also a correct top-sort.

-

(note that in the above diagram, 2-3-5 isn't a "cycle")



- Directed graph with (directed) cycle ⇒ no topological sort. Because, if we put 2,3,5, in any order, then the definition of topological sort will be violated.
e.g., keeping it as 2→3→5 is wrong because the edge 5→2 is present.

- Can use DFS to get the topological sort.

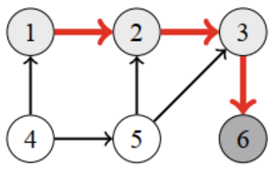


Fig. 7.25 First search adds nodes 6, 3, 2, and 1 to the list

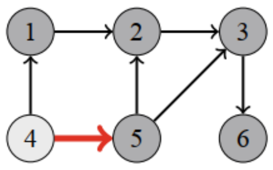


Fig. 7.26 Second search adds nodes 5 and 4 to the list

-
- During the run, each node can have 3 possible states:
  - state 0: the node has not been processed (white)
  - state 1: the node is under processing (light gray)
  - state 2: the node has been processed (dark gray)
- Initially, the state of each node is 0. When a search reaches a node for the first time, its state becomes 1. Finally, after all edges from the node have been processed, its state becomes 2.
- We can construct a topological sort by pushing each node into a vector when its state becomes 2. Finally, we reverse the list and get a topological sort for the graph.
- If the graph contains a cycle, we will discover this during the search, because sometime in the run, we will arrive at a node whose state is 1. [and use this to detect a (directed) cycle]

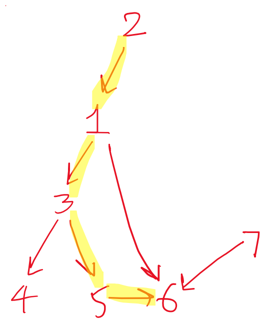
- Since DFS is being used (more than once, though just covering each vertex and edge ~ once), so O(n+m), i.e., O(|V|+|E|).

- See code: 7_top-sort.cpp.

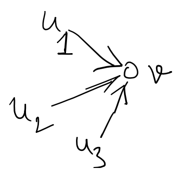
Applications

- A practical application of top-sort: package dependency (building packages). For example, in the below graph, we have that for any u→v, package u must be available before starting the build of package v.
  Then, the top-sort gives a correct order of building packages (without running into dependency issues).

- Longest Path in DAG



$$\begin{tikzcd} \Rightarrow & 2 \arrow[r] & 1 \arrow[r] \arrow[rrr, bend left] & 3 \arrow[r] \arrow[rr, bend right] & 4 & 5 \arrow[rr, bend left] & 7 \arrow[r] & 6 \end{tikzcd}$$



Length of longest path till v = max (length of longest paths till u1, … u2, … u3) + 1
Use recursion [base case: a vertex with no incoming edges: this is the 1st vertex in the topological sort], and then find the max length for any v∈V.

Instead of using the above recurrence, we can (equivalently) do the following:
For each vertex, given longest[v], we update all its out-neighbors. Note that the effect is same as the recurrence. Above example: longest[2]=0, so longest[1]=0+1=1, then longest[3] and longest[6] = 1+1 = 2, and so on (longest[6] will be modified later).

Code: longest-dag.cpp.

Another method: give negative weights to all edges, and find shortest path (say, using Bellman-Ford (Dijkstra won't work)).

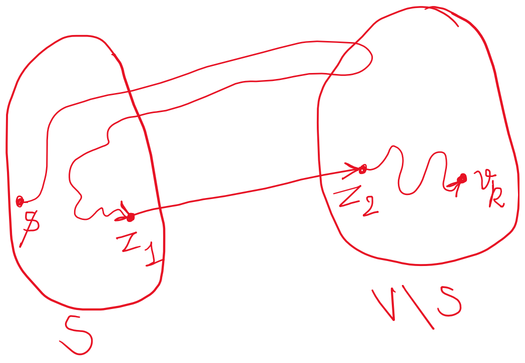
- Single source shortest paths in DAG [in just O(m+n)]
  Idea similar to above:
  Shortest distance from src to v = min (shortest distances from src to u1, … u2, … u3) + 1
  [the one out of u1,u2,u3 that gives the minimum can be used to retrieve the shortest "path" from src to v]

Problems

- First 5 problems from CSES 'Graph Algorithms'. Will put up other problems on the group and the sheet.

Extras

- **Proof of correctness of Dijkstra's Algorithm**
  Claim: P(k) : For the k^{th} vertex v_k added to the set S, we will have correct distance to all nodes in S.
  We use induction. Base case: S={s} (s is the source), dist[s]=0. Correct.
  Induction Hypothesis: Let P(0),P(1),…,P(k-1) be true. [Base case is P(0), $v_0 = s$.]
  Induction Step: Want to prove P(k).



$z_1 \rightarrow z_2$ is one single edge

  Let v_k be the node that is in V\S, and has the current smallest distance. We show that this current distance of v_k is indeed the **shortest** distance from source s to v_k.

  Suppose not.

  Then, there is a path P from s to v_k, that has a path length < this current distance. (1)

  Note that since $s \in S$ and $v_k \in V\backslash S$, this path will necessarily have to make a jump from S to V\S. Let z_1 be the last vertex in P to be part of S (see diagram), and let z_2 be its corresponding neighbor in V\S (that's part of path P).

  By induction hypothesis, dist[z_1] (by our algo) is equal to the correct smallest distance from s to z_1. So, the part of P, that is from s to z_1, has length ≥ dist[z_1].
  Moreover, the part of P that is from z_2 to v_k has length ≥ 0 [**note that we are using the assumption of non-negative edges here**].

  Then, on combining, length(P) ≥ dist[z_1] + (weight of $z_1 \rightarrow z_2$) + (something non-negative)
  ⇒ length(P) ≥ dist[z_1] + (weight of $z_1 \rightarrow z_2$)

  But, v_k was chosen to be that vertex in V\S, which has smallest current distance, and current distances are of the form same as RHS of the above inequality! [because we update current distances for neighbors of the vertex being added into S, as dist[v_i]+(weight of edge v_i→(neighbour))]

  Thus, v_k is the vertex that minimizes RHS of the above inequality. Hence, length(P) ≥ dist[v_k] = current distance of v_k. This contradicts (1).

- Bellman-Ford: Theorem 24.4 (CLRS)
- Floyd-Warshall: Section 25.2 (CLRS)