WORKSHEET, WEEK10: GRAPH TRAVERSALS

8.1 DFS in Oriented Graphs

Discovery/Finishing times DFS traversal algorithm can mark each vertex with two numbers `d/f`, where the first number `d` is the discovery time, and the second number `f` is the finishing time. All these numbers should be different and all of them belong to the interval $[1, 2N]$, where $N = |V|$ is the number of vertices.

Discovery/Finishing times can be stored in the graph nodes (augmented nodes); they are useful in various derived algorithms.

Edges after a DFS traversal DFS traversal turns all edges into four groups:

Tree edges Edges in the depth-first forest G_{DFS}. Edge (u, v) is a tree edge iff (u, v) was first discovered when u was gray (visited, not finished) and v was white (not yet visited).

Back edges Edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Edge is a back edge iff it was first discovered when u was gray and also v was gray (in process of DFS processing, not finished). Also self-loops, which may occur in directed graphs are considered to be back edges.

Forward edges Edges (u, v) that connect a vertex u to a descendant v in a depth-first tree (but they did not become tree edges – since v was first discovered through another path.

Cross edges All other edges – they can go between vertices in the same DFS tree as long as one vertex is not an ancestor of the other. Or they can go between vertices in different depth-first trees.

8.2 Topological Sorting

Definition Given an acyclic directed graph, a sequence of all its vertices $v_1, v_2, \ldots, v_N$ is called a *topological sorting* if for each edge (v_i, v_j) in this graph, vertex v_i precedes the vertex v_j.

Problem 1 (Topological Sorting) Consider the following graph:

(A) Run the DFS traversal algorithm on the graph shown in the figure, mark each vertex with two numbers `d/f`, where the first number `d` is the discovery time, and the second number `f` is the finishing time. Separate both numbers with a slash. All these numbers should be different and all of them belong to the interval $[1, 28]$, since the graph has 14 vertices.

If there are multiple ways how to pick a vertex to visit next in the DFS order, always pick the vertex with the alphabetically smallest label. Namely, your DFS traversal should start from the vertex A; every time there is a choice where to go deeper – pick the alphabetically first label not visited. Whenever the DFS traversal runs out of vertices to visit in a given discovery tree (but some nodes are still unvisited), pick the alphabetically smallest node as the root for the next DFS tree, and so on.

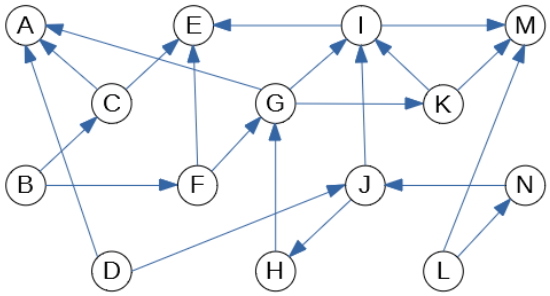


Fig. 1: Directed graph to run DFS and Topological Sorting

(B) In case if the graph shown in the Figure is not a DAG (directed acyclic graph), explain why it is not a DAG and remove some edge so that it becomes a DAG. On the other hand, if the graph in the Figure is already a DAG, explain why it is the case and do not remove any edges.

(C) Produce a topological sorting of the graph obtained in **(B)** – list the vertices in their topological sorting order.

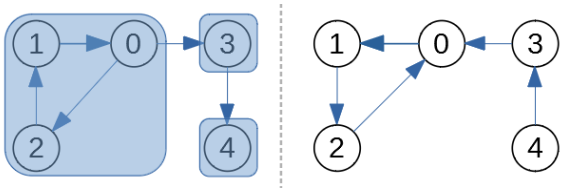
8.3 Strongly Connected Components

DFS traversal of a directed graph can be used to find strongly connected components – Kosaraju's algorithm. https://bit.ly/3lI20ec, https://bit.ly/3mNU2la.

Definition: A subset of vertices in a directed graph $S \subseteq G.V$ makes a strongly connected component, iff for any two distinct vertices u, v there is a path $u \rightsquigarrow v$ (one or more and also another path $v \rightsquigarrow u$ that goes back from v to u.

If you can travel only in one direction (say, from u to v), but cannot return, then u, v should be in different strongly connected components. (Same thing, if u and v are mutually unreachable.) Every vertex is strongly connected to itself – in a graph with n vertices there are at most n strongly connected components.

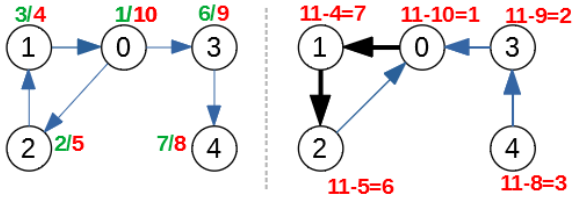
Figure shows an example of a graph with $n = 5$ vertices having 3 strongly connected components. Next to that graph is the *transposed graph* G^T where all the edges are reversed.



Kosaraju's algorithm to find strongly connected components in an arbitrary graph by running DFS twice (i.e. it works in linear time $O(n + m)$).

```
    STRONGLY_CONNECTED(G)
    (compute all finishing times u.f)
1   call DFS(G)
    (G^T is transposed G, all edges reversed)
2   compute G^T
    (visit vertices in decreasing u.f order)
3   call DFS(G^T)
4   for each tree T in the forest DFS(G^T)
5       Output T as a component
```

To see how this works, we can run it on the example graph shown earlier. After the DFS on graph G is run, we get the finishing times for the vertices $0, 1, 2, 3, 4$ (all shown in red on the left side of Figure below). After that we replace G by G^T (to the right side of the same figure), and assign priorities in the decreasing sequence of $u.f$ (the finishing times when running DFS(G)).



To make this reverse order obvious, we assign new priorities to the vertices in G^T. The new priorities in G^T are the following:

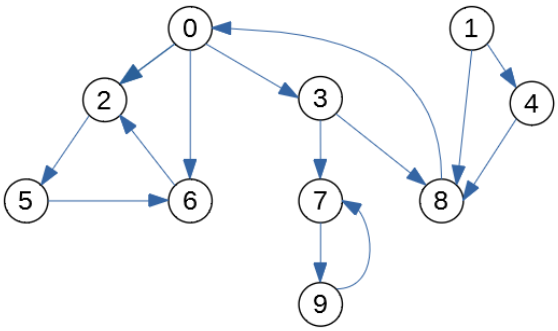
- Vertex 0 has priority $11 - 10 = 1$.
- Vertex 1 has priority $11 - 4 = 7$.
- Vertex 2 has priority $11 - 5 = 6$.
- Vertex 3 has priority $11 - 9 = 2$.
- Vertex 4 has priority $11 - 8 = 3$.

Now run DFS(G^T). It turns out that the DFS algorithm starts in the vertex `"0"` once again (since it was finished last in DFS(G)). But unlike the DFS algorithm in G itself (it produced just one DFS tree), we get a DFS forest with 3 components (tree/discovery edges shown bold and black in the previous Figure).

- $\{0, 1, 2\}$ (DFS tree has root "0").
- $\{3\}$ (DFS tree has root "3").
- $\{4\}$ (DFS tree has root "4").

They represent the strongly connected components in G (they are also strongly connected in G^T).

Problem 2(Kosaraju's algorithm) We start with the graph shown in Figure below.



(A) Run the DFS traversal algorithm on the graph G. Mark each vertex with the pair of numbers `d/f`, where the first number `d` is the discovery time, and the second number `f` is the finishing time.

(B) Draw the transposed directed graph (same vertices, but each arrow points in the opposite direction). Run the DFS traversal algorithm on G^T. Make sure that the DFS outer loop visits the vertices in the reverse order by $u.f$ (the finishing time for the DFS algorithm in step **(A)**). In this case you do not produce the discovery/finishing times once again, just draw the discovery edges used by the DFS on G^T – you can highlight them (show them in bold or use a different color).

(C) List all the strongly connected components (they are the separate pieces in the forest obtained by running DFS on G^T).

8.4 Single-Source Shortest Paths

Definition Let $G(V, E)$ be an (undirected or directed) graph, where each edge is assigned a weight – some real number. One of the vertices $v \in V$ is selected as the source. The Single-Source Shortest Path problem finds the shortest paths between the given vertex v and all other vertices in the graph.

Examples: There are some well-known solutions to the single-source shortest paths problem in special cases:

- BFS (Breath-First-Search) by itself finds the shortest distances from the root to all the other vertices, if every edge has weight 1. Every time we discover a new vertex w (not visited by the BFS earlier) we assign its distance to the root $d_v(w)$ to be $d_v(u) + 1$, where $d_v(u)$ denotes the distance of its parent u.
- Dijkstra's algorithm can be used to find the shortest distances from the root to all the other vertices, if every edge has a positive weight.

8.4.1 Positive Edge Weights

Dijkstra's algorithm requires $O((m + n) \log_2 n)$ time, if we use priority queues; here $m = |E|$ is the number of edges and $n = |V|$ is the number of vertices in a graph.

In this example we do not implement a priority queue; assume that you can always pick the vertex with the smallest distance and add it to the set S of visited vertexes (those having distances already computed).

Example (Dijkstra's Algorithm): We start with the graph shown in Figure below:

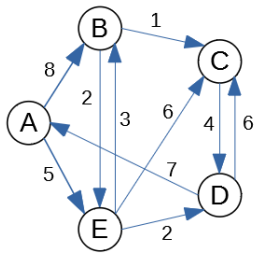


Fig. 2: Graph Diagram for Dijkstra's Algorithm

Vertex A will be your source vertex. (You can assume that the distance from A to itself is 0; initially all the other distances are infinite, but then Dijkstra's algorithm relaxes them).

(A) Run the Dijkstra's algorithm: At every phase write the current vertex v; the set of finished vertices and also a table showing the new distances to all A, B, C, D, E (and their parents) after the relaxations from v are performed. At the end of every phase highlight which vertex (among those not yet finished) has the minimum distance. This will become the current vertex in the next phase.

(B) After the algorithm finishes, summarize the answer: For each of the five vertices tell what is its minimum distance from the source. Also show what is the shortest path how to achieve that minimum distance.

Solution Draw Dijkstra's algorithm step by step; show results in tables.

(A) At every phase we select the minimum-distance vertex in the priority queue of vertices (not yet added to the set of finished vertices S). This becomes the current vertex v. After that we relax all the edges that go out from the current vertex v (if some distance decreases, we change the parent of this new vertex to become v).

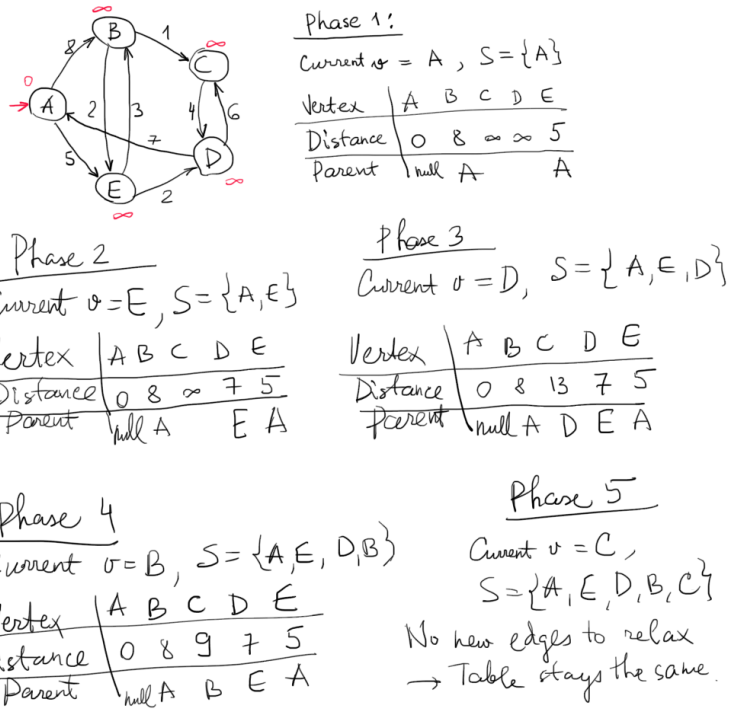


Fig. 3: **(A)** – Five Phases of Dijkstra's Algorithm

(B) The result of Dijkstra's algorithm can be summarized as shown below. For each vertex we specify the distance from A to that vertex (and also what is the shortest path to achieve it).

Vertex	Distance	Path
A	$d(A, A) = 0$	A
B	$d(A, B) = 8$	$A \to B$
C	$d(A, C) = 9$	$A \to B \to C$
D	$d(A, D) = 7$	$A \to E \to D$
E	$d(A, E) = 5$	$A \to E$

8.4.2 Negative Edge Weights

The Bellman-Ford algorithm solves the single source shortest paths problem in the case in which edge weights may be negative. It can work with directed graphs (and also undirected graphs; not discussed in this exercise). The algorithm initializes the distances to all the vertices u by $u.d = +\infty$. The only exception is the *source vertex*) which gets distance $s.d = 0$ (the distance to itself is 0).

After this initialization in a graph with n vertices it will perform $n - 1$ identical iterations. In every iteration it considers all the edges in some order, and "relaxes" all the edges. After that you can perform one last iteration with Bellman-Ford algorithm: If there are still relaxations that reduce distances even after n steps, this means that there is a negative loop in the original graph (and the shortest paths are not possible to compute as the distances can be reduced infinitely).

Let $G(V, E)$ be a directed graph. Let $w : E \to \mathbf{Z}$ be a function assigning integer weights to all the graph's edges and let $s \in V$ be the source vertex. Every vertex $v \in V$ stores $v.d$ – the current estimate of the distance from the source. A vertex also stores $v.p$ – its "parent" (the last vertex on the shortest path before reaching v). Bellman-Ford algorithm to find the minimum distance from s to all the other vertices is given by the following pseudocode:

BELLMANFORD(G, w, s):
    for each vertex $v \in V$: *(initialize vertices to run shortest paths)*
        $v.d = \infty$
        $v.p =$ NULL
    $s.d = 0$ *(the distance from source vertex to itself is 0)*
    for $i = 1$ **to** $|V| - 1$ *(repeat $|V| - 1$ times)*
        for each edge $(u, v) \in E$
            if $v.d > u.d + w(u, v)$: *(relax an edge, if necessary)*
                $v.d = u.d + w(u, v)$
                $v.p = u$

Example (Bellman-Ford): Consider the graph in Figure:

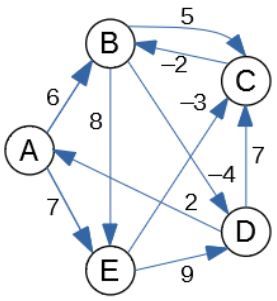


Fig. 4: Graph Diagram for Bellman Ford Algorithm

Let us pick vertex B as the *source vertex* for Bellman-Ford algorithm. (You could pick the source vertex differently, but then all the distance computations would be different as well.)

(A) Create a table showing all the changes to all the distances to A, B, C, D, E as the relaxations are performed. In a single iteration the same distance can be relaxed/improved multiple times (and you can use distances computed in the current phase to relax further edges). The table should display all $n - 1$ iterations (where $n = 5$ is the number of vertices). (*Sometimes it is worth running one more iteration to find possible negative loops*).

Note: Please make sure to release the edges in the alphabetical/lexicographical order: Regardless of which is your source, in every iteration the edges are always relaxed in this order:

$$AB, AE, BD, BE, CB, DA, DC, EC, ED.$$

In fact, any order can work; the only thing that matters is that you consider all the edges. But alphabetical ordering of edges makes the solution deterministic.

(B) Summarize the result: For each of the 5 vertices tell what is its minimum distance from the source. Also tell what is the shortest path how to get there. For example, if your source is E then you could claim that the shortest path $E \rightsquigarrow B$ is of length -5 and it consists of two edges $(E, C), (C, B)$.

Solution Show the Bellman-Ford algorithm in stages; results are shown in tables.

(A) In this case we only need to run three phases (not $n - 1 = 4$ phases), since all the distances become stable and do not change anymore after Phase 3. The tables show only those relaxed edges that lead to decreased distances.

(B) The result of Bellman-Ford's algorithm can be summarized as shown below. For each vertex we specify the distance from A to that vertex (and also what is the shortest path to achieve it).

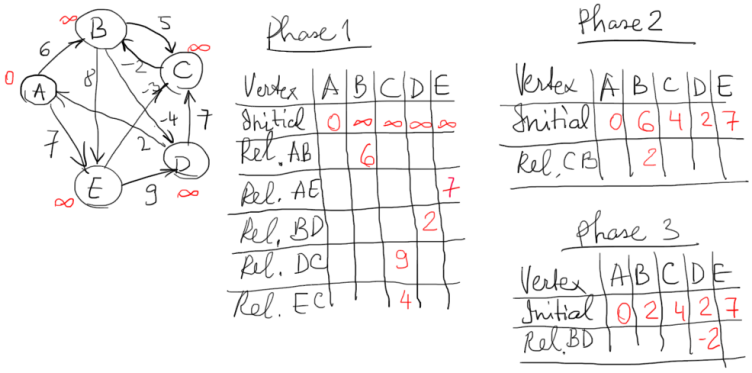


Fig. 5: **(A)** – Phases of Bellman-Ford's Algorithm

Vertex	Distance	Path
A	$d(A, A) = 0$	A
B	$d(A, B) = 2$	$A \to E \to C \to B$
C	$d(A, C) = 4$	$A \to E \to C$
D	$d(A, D) = -2$	$A \to E \to C \to B \to D$
E	$d(A, E) = 7$	$A \to E$

Problem 3 (Bellman-Ford) Consider the input graph shown in Fig.1.

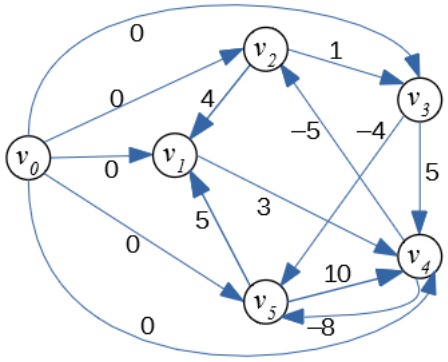


Fig. 6: A directed graph for Bellman-Ford Algorithm

(A) In your graph use the vertex $s = v_0$ as the *source vertex* for Bellman-Ford algorithm. Create a table showing the changes to all the distances to the vertices of the given graph every time a successful edge relaxing happens and some distance is reduced. You should run $n-1$ phases of the Bellman-Ford algorithm (where n is the number of vertices). You can also stop earlier, if no further edge relaxations can happen.

Note: Please make sure to release the edges in the lexicographical order. For example, in a single phase the edge (v_1, v_4) is relaxed before the edge (v_2, v_1), since v_1 precedes v_2.

(B) Summarize the result: For each vertex tell what is its minimum distance from the source. Also tell what is the shortest path how to get there.

(C) Does the input graph contain negative cycles? Justify your answer.

8.5 Minimum Spanning Trees

Definition Let $G(V, E)$ be a connected undirected graph where each edge is assigned a non-negative weight. A *minimum spanning tree* is a subset of edges $MST \subseteq E$ that keeps graph connected, and the total weight of all the edges in MST is the smallest possible.

Prim's Algorithm Let $G(V, E)$ be an *undirected* graph. Let $w : E \to \mathbf{Z}$ be a function assigning integer weights to all the graph's edges and let r be the root vertex that will start to grow the minimum spanning tree (MST). Every vertex $v \in V$ stores $v.key$ – the key for a priority queue (initially containing all the vertices). A vertex also stores $v.p$ – its "parent" (the parent vertex in the ultimate MST; it is assigned only once). Prim's algorithm to find the minimum spanning tree in G is given by the following pseudocode:

MSTPRIM(G, w, r):
    for each vertex $u \in V$:
        $u.key = \infty$
        $u.p =$ NULL
    $r.d = 0$
    $Q =$ MINIMUMHEAP(V) *(Insert all vertices in a priority queue)*
    while $Q \neq \emptyset$:
        $u =$ EXTRACTMIN(Q) *(pick a vertex closest to the MST built so far)*
        for each $v \in$ ADJ(G, u):
            if $v \in Q$ **and** $w(u, v) < v.key$
                $v.p = u$
                $v.key = w(u, v)$

It is an efficient algorithm; it requires $O((m + n) \log_2 n)$ time, if we use priority queues as heaps.

Kruskal's Algorithm You start out with a bunch of one-node isolated components. At each step you pick the cheapest edge between any two components and join them together. Let F denote the *forest* containing the little trees used to build the MST. Here is the pseudocode:

KRUSKAL(G)
    $F = \emptyset$
    for each $v \in G.V$:
        makeSet(v)
    for each $(u, v) \in G.E$ ordered by $weight(u, v)$ increasing:
        if FINDSET$(u) \neq$ FINDSET(v):
            $F = F \cup \{(u, v)\} \cup \{(v, u)\}$
            UNION(FINDSET(u), FINDSET(v))
    return F

Example for MSTs We start with the graph shown in Figure:

(A) Vertex A will be your source vertex. It is the first vertex added to the MST vertice set S. At every step you find the lightest edge that connects some vertex in S to some vertex not in S. Add this new vertex to a graph and remember the edge you added. Show how the Prim's MST (Minimum Spanning Tree grows) one edge at a time.

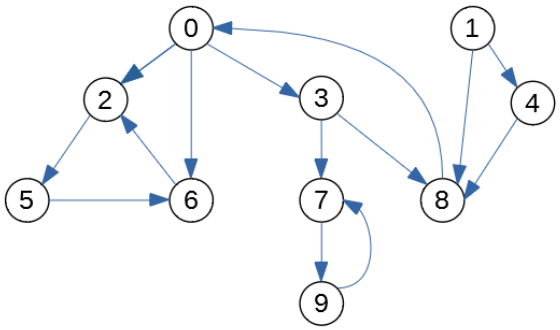


Fig. 7: Graph Diagram for Prim's Algorithm.

Note: In cases when there is a choice between multiple lightest edges of the same weight, pick the edge (v, w) with $v \in S$ and $w \notin S$ such that (v, w) lexicographically precedes any other lightest edge.

(B) Redraw the graph, highlight the edges selected for MST (make them bold or color them differently). Add up the total weight of the obtained MST and write this in your answer (it should be the minimum value among all the possible spanning trees in this graph).

Solution We show the subsequent steps of Prim's algorithm.

(A) At each step we show the current set of vertices in MST (denoted by S) and which edge is being added.

1. $S = \{A\}$, adding edge AB
2. $S = \{A, B\}$, adding edge BH
3. $S = \{A, B, H\}$, adding edge HG
4. $S = \{A, B, G, H\}$, adding edge GF
5. $S = \{A, B, F, G, H\}$, adding edge FC
6. $S = \{A, B, C, F, G, H\}$, adding edge CI
7. $S = \{A, B, C, F, G, H, I\}$, adding edge FE
8. $S = \{A, B, C, E, F, G, H, I\}$, adding edge CD

(B) Solution shows the MST edges added in previous step colored blue:

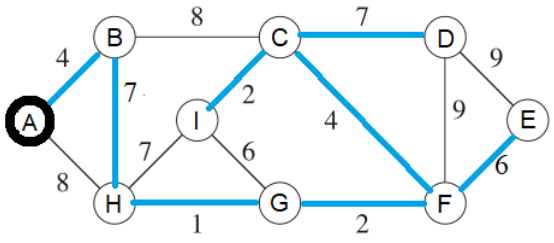


Fig. 8: MST obtained by Prim's Algorithm.

The total weight of this MST is $4 + 7 + 1 + 2 + 4 + 2 + 6 + 7 = 33$. (*In this case the MST is unique. In general case there is no guarantee that there are no other MSTs of the same weight, but the one we found with Prim's algorithm is among the lightest ones.*)

Example Continued Run Kruskal's algorithm on the same graph as before.

(A) After each step when there is an edge connecting two sets of vertices, write that edge and show the partition where that edge connects two previously disjoined pieces in the forest of trees.

Note:

If there are multiple lightest edges that can be used to connect two disjoined pieces, pick edge (v, w) which lexicographically precedes any other.

(B) Redraw the given graph (show the order how you added the edges in parentheses). Also compute the total weight of this MST.

Solution Here is the Kruskal's algorithm showing node clusters:

(A) We list the steps that add edges and join two previously disconnected pieces:

1. Add edge GH, the partition becomes $\{A, B, C, D, E, F, GH, I\}$.
2. Add edge CI, the partition becomes $\{A, B, CI, D, E, F, GH\}$.
3. Add edge FG, the partition becomes $\{A, B, CI, D, E, FGH\}$.
4. Add edge AB, the partition becomes $\{AB, CI, D, E, FGH\}$.
5. Add edge CF, the partition becomes $\{AB, CFGHI, D, E\}$.
6. Add edge FE, the partition becomes $\{AB, CEFGHI, D\}$.
7. Add edge BH, the partition becomes $\{ABCEFGHI, D\}$.
8. Add edge CD, the partition becomes $\{ABCEFGHID\}$.

(B) Solution shows the MST edges added in previous step colored blue. The total weight is 33. The order of their addition is shown in red in parentheses.

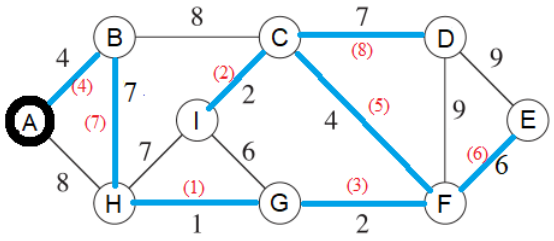


Fig. 9: MST obtained by Kruskal's Algorithm.

Note: In some cases Prim's and Kruskal's algorithm can yield different MSTs even for the same input graph, but they are both optimal in such cases.

Problem 4 (Prim's Algorithm): Denote the last three digits of your Student ID by a, b, c. Student ID often looks like this: `201RDBabc`, where a, b, c are digits. Compute three more digits x, y, z:

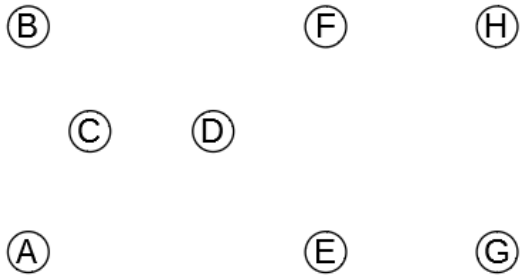
$$\begin{cases} x = (b + 4) \bmod 10 \\ y = (c + 4) \bmod 10 \\ z = (a + b + c) \bmod 10 \end{cases}$$

In this task the input graph $G = (V, E)$ is given by its adjacency matrix:

$$M_G = \begin{pmatrix} 0 & 0 & 5 & 8 & y & 0 & 0 & 0 \\ 0 & 0 & 3 & 7 & 0 & z & 0 & 0 \\ 5 & 3 & 0 & 3 & 0 & 0 & 0 & 0 \\ 8 & 7 & 3 & 0 & 1 & 7 & 0 & 0 \\ y & 0 & 0 & 1 & 0 & 6 & 9 & 6 \\ 0 & z & 0 & 7 & 6 & 0 & x & 2 \\ 0 & 0 & 0 & 0 & 9 & x & 0 & 7 \\ 0 & 0 & 0 & 0 & 6 & 2 & 7 & 0 \end{pmatrix}.$$

(A) Draw the graph as a diagram with nodes and edges. Replace x, y, z with values calculated from your Student ID. Label the vertices with letters A, B, C, D, E, F, G, H (they correspond to the consecutive rows and columns in the matrix).

If you wish, you can use the following layout (edges are not shown, but the vertice positions allow to draw the edges without much intersection). But you can use any other layout as well.



(B) Run Prim's algorithm to find MST using $r = A$ as the root. If you do not have time to redraw the graph many times, just show the table with $v.key$ values after each phase. (No need to show $v.p$, as the parents do not change and they are easy to find once you have the final rooted tree drawn.) The top of the table would look like this (it shows Phase 0 – the initial state before any edges have been added).

Phase	A	B	C	D	E	F	G	H
0 (initial state)	0	∞	∞	∞	∞	∞	∞	∞

(C) Summarize the result: Draw the MST obtained as the result of Prim's algorithm, find its total weight.