Today's topics

- Strongly connected components
- Single-source shortest-paths

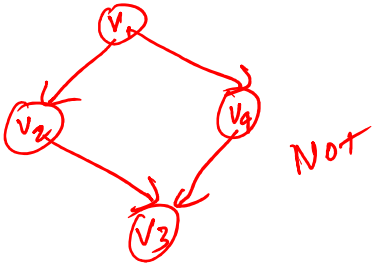
Strong connectivity

Strong connectivity

- Section-20.5 from the CLRS book
- Section-6.4 from the Goodrich and Tamassia book
- https://en.wikipedia.org/wiki/Strongly_connected_component

Strong connectivity

- A directed graph is strongly connected if every vertex is reachable from all other vertices



Strongly connected components (SCC)

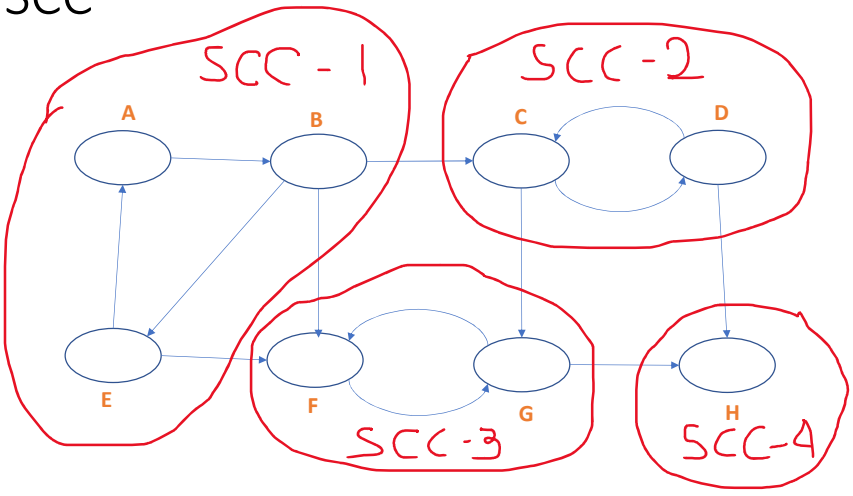
SCC

- A strongly connected component of a directed graph G is a strongly connected maximal subgraph

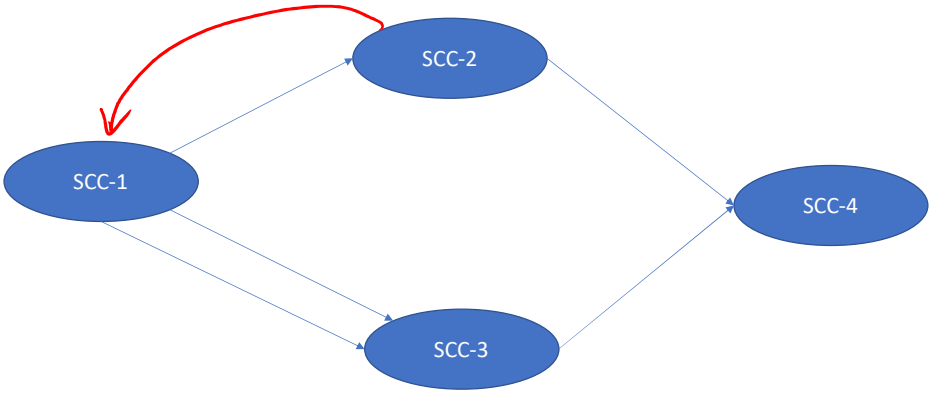
SCC

- Find all strongly connected components of a graph

SCC



SCC



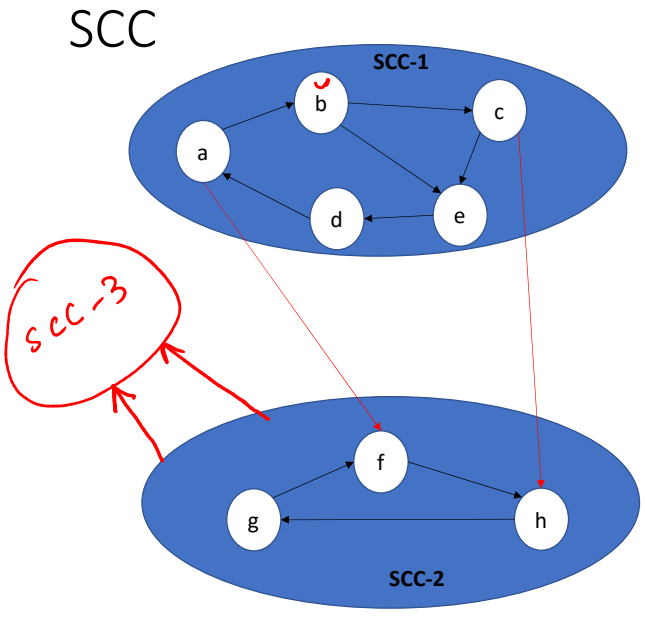
If we construct a graph whose vertices are the SCCs and edges are the edges between SCCs, then the resulting graph must be acyclic.

SCC

- If we consider all vertices in an SCC as a single vertex and draw edges between SCCs, then the resulting graph G^{SCC} must be acyclic
- If the graph G^{SCC} is cyclic then there would be at least two vertices that are reachable from each other
- Notice that the vertices in G^{SCC} are strongly connected components, and two strongly connected components can't reach from each other

SCC

- Every acyclic graph has a topologically sorted order
- If an SCC, say S1, comes before another SCC, say S2, in the topologically sorted order, then the finish time of all the vertices in S1 must be greater than all vertices in S2



If DFS starts from SCC-1, then none of the vertices in SCC-1 will be finished until all vertices in SCC-2 are finished. Notice that all vertices in SCC-2 are reachable from all vertices in SCC-1; however, none of the vertices in SCC-1 are reachable from any vertex in SCC-2.

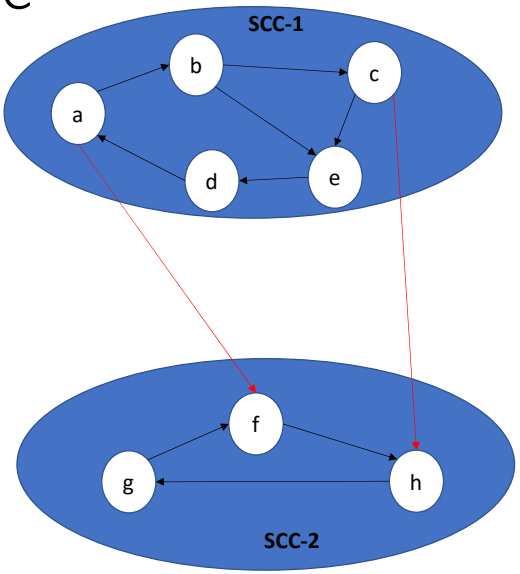
If DFS starts from SCC-2, then none of the vertices of SCC-1 will be visited because they are not reachable from any vertex in SCC-2. Vertices in SCC-1 will only be visited when DFS is called later on a vertex in SCC-1 or a vertex in a different SCC from which SCC-1 is reachable.

The finish time of all the vertices in SCC-1 must be higher than that of the vertices in SCC-2.

SCC

- If we reverse all edges
  - the destination of an outgoing edge from an SCC will always have a larger finish time than the corresponding origin of the edge
  - The other edges that were not going outside the SCC will remain inside the SCC

SCC



Because the direction of all the edges between SCC-1 and SCC-2 are from SCC-1 to SCC-2, the direction of the reversed edges will be from SCC-2 to SCC-1. As discussed earlier, the finish times of all vertices in SCC-1 are higher than the vertices in SCC-2; therefore, the finish time of the destination end of a reversed edge between two strongly connected components will always be higher than its origin.

If we reverse the edges, the edge originated from SCC-2 will be either in SCC2 or will go to an SCC that comes before SCC-2 in a topologically sorted order.

SCC

- Computing SCC
  - The outgoing reversed edges within an SCC will always go to a vertex in another SCC with a higher finish time or go to a vertex within the SCC

SCC

- Let's say a graph G = (V, E) has k SCCs numbered from 1 to k, such that SCC-i comes before SCC-j in topologically sorted order for all 1 <= i < j <= k
  - If vertex v has the highest finish time, then v belongs to SCC-1
    - If we run DFS at v on the reversed graph, which all vertices will be visited?

SCC

- Let's say a graph G = (V, E) has k SCCs numbered from 1 to k, such that SCC-i comes before SCC-j in topologically sorted order for all 1 <= i < j <= k
  - If vertex v has the highest finish time, then v belongs to SCC-1
    - If we run DFS at v on the reversed graph, which all vertices will be visited?
      - All vertices that belong to SCC-1

SCC

- Let's say a graph G = (V, E) has k SCCs numbered from 1 to k, such that SCC-i comes before SCC-j in topologically sorted order for all 1 <= i < j <= k
  - If vertex v has the highest finish time among the vertices that don't belong to SSC-1, then v belongs to SCC-2
    - If we run DFS at v on the reversed graph, which all vertices will be visited?

SCC

- Let's say a graph G = (V, E) has k SCCs numbered from 1 to k, such that SCC-i comes before SCC-j in topologically sorted order for all 1 <= i < j <= k
  - If vertex v has the highest finish time among the vertices that don't belong to SSC-1, then v belongs to SCC-2
    - If we run DFS at v on the reversed graph, which all vertices will be visited?
      - All vertices that belong to SCC-2. Vertices in SCC-1 may also be visited.

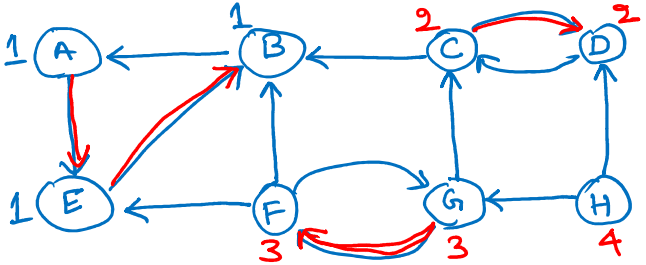
SCC

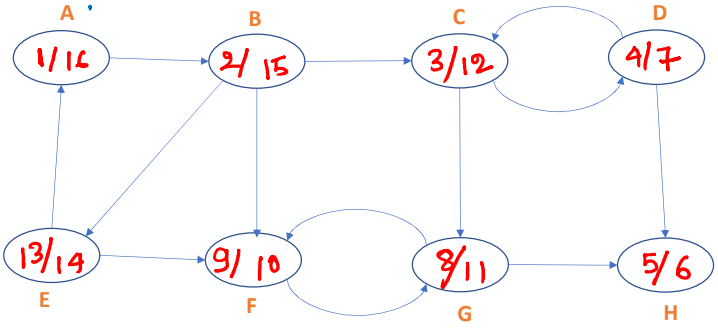
- Let's say a graph G = (V, E) has k SCCs numbered from 1 to k, such that SCC-i comes before SCC-j in topologically sorted order for all 1 <= i < j <= k
  - If vertex v has the highest finish time among the vertices that don't belong to SSC-1 and SSC-2, then v belongs to SCC-2
    - If we run DFS at v on the reversed graph, which all vertices will be visited?

SCC

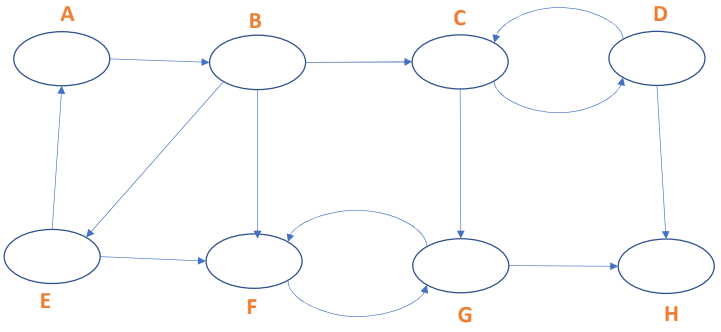
- Let's say a graph G = (V, E) has k SCCs numbered from 1 to k, such that SCC-i comes before SCC-j in topologically sorted order for all 1 <= i < j <= k
  - If vertex v has the highest finish time among the vertices that don't belong to SSC-1 and SSC-2, then v belongs to SCC-2
    - If we run DFS at v on the reversed graph, which all vertices will be visited?
      - All vertices that belong to SCC-2. Vertices in SCC-1 and SCC-2 may also be visited.

SCC



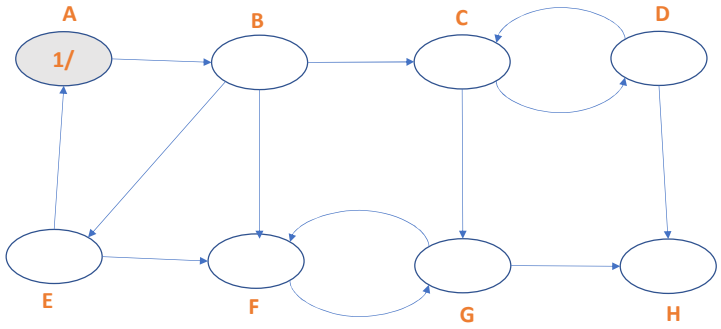


SCC

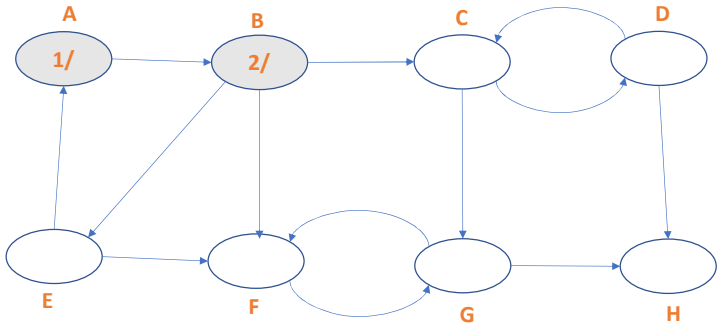


The first step in computing SCCs is to run DFS on the graph. Here, we are starting DFS from vertex A.

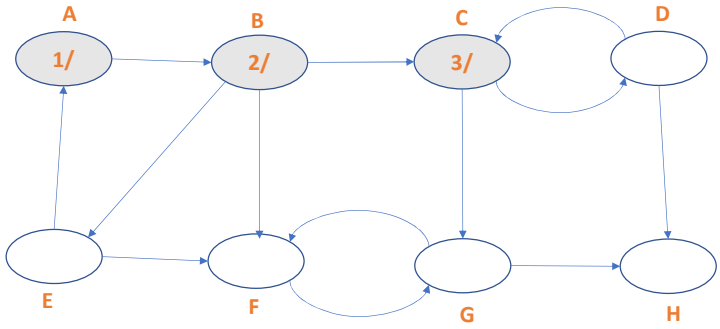
SCC



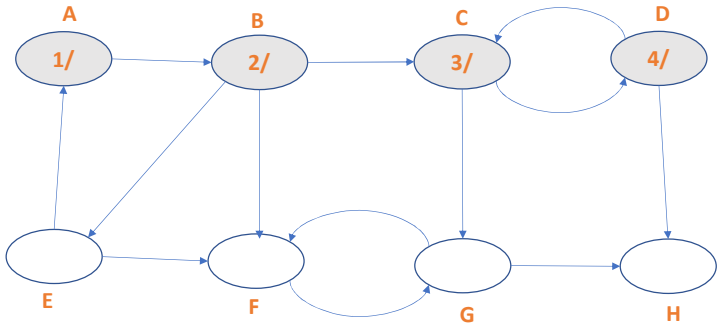
SCC



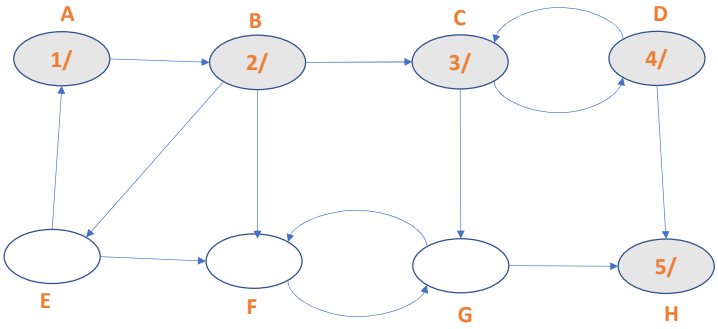
SCC



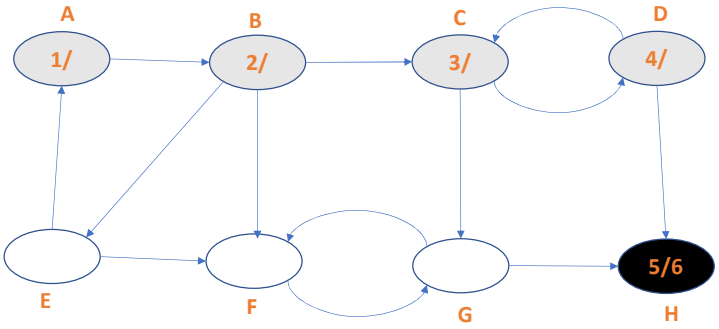
SCC



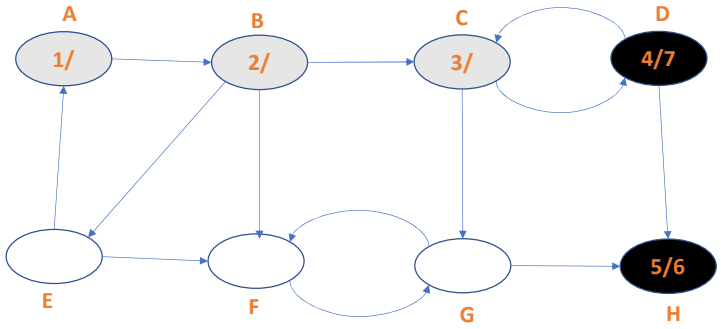
SCC



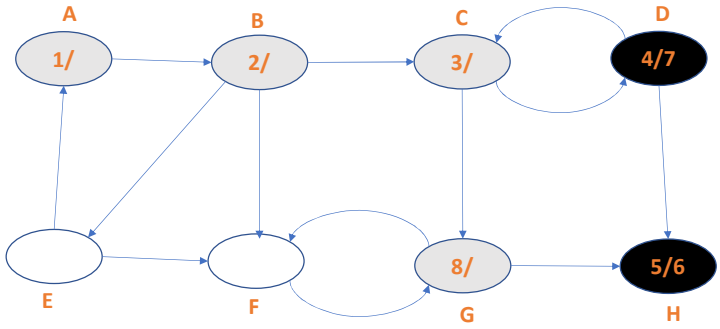
SCC



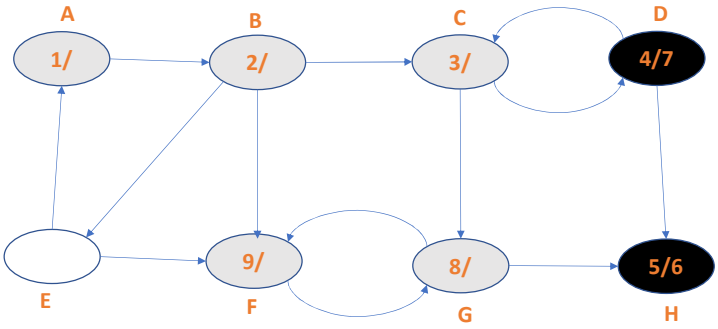
SCC



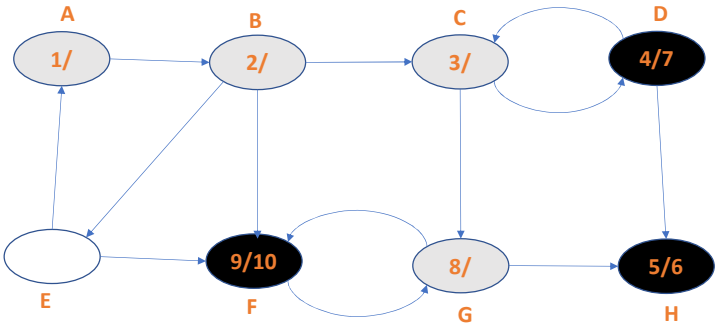
SCC



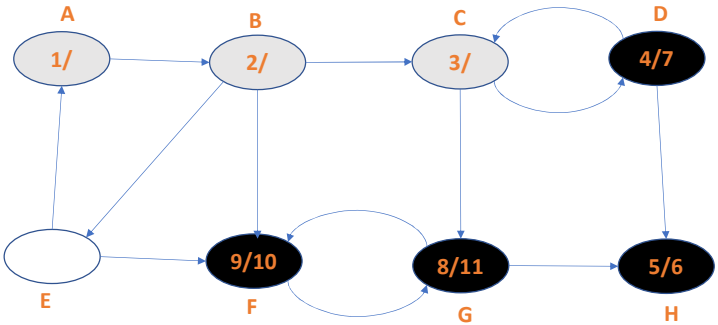
SCC



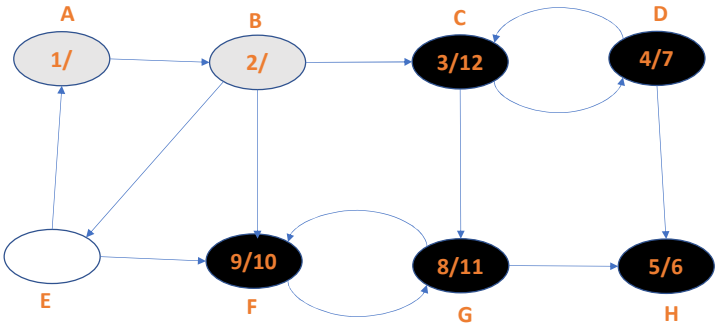
SCC



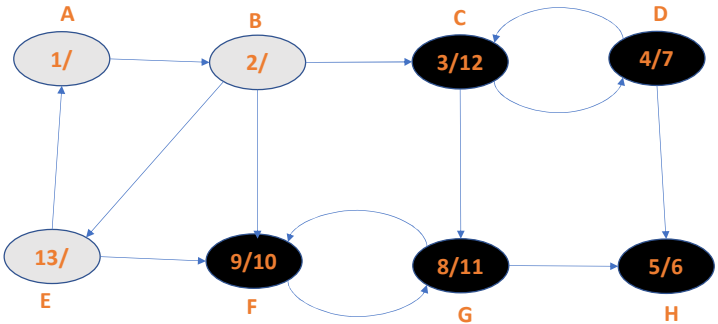
SCC



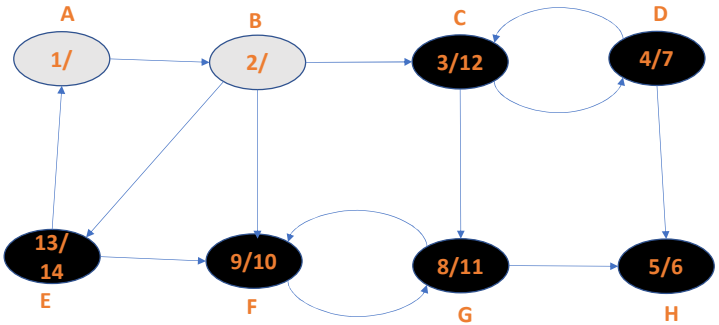
SCC



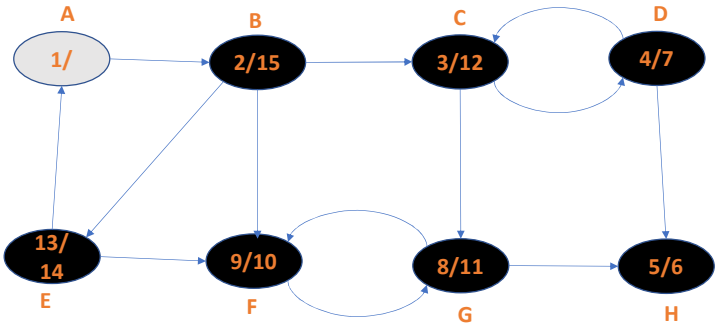
SCC



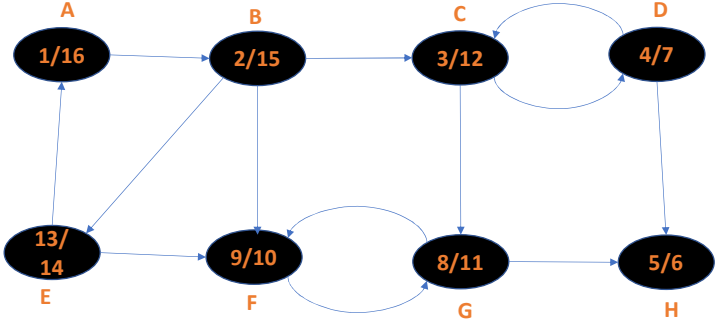
SCC



SCC



SCC



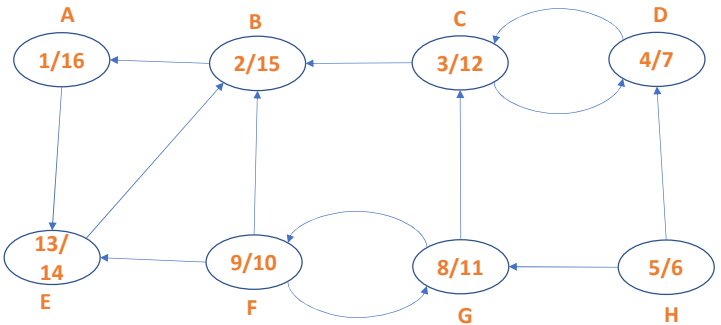
Now we have computed the finish times of all vertices.

SCC

- Reversing edges

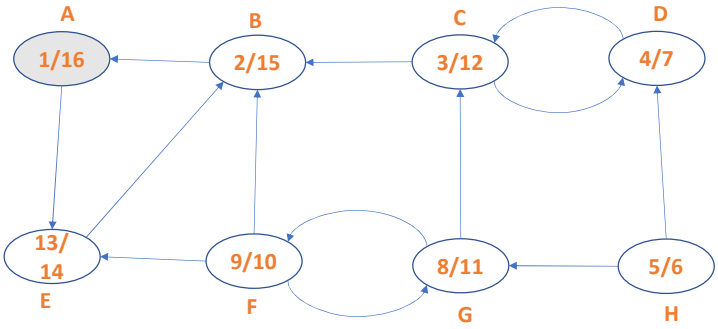
SCC

Run DFS at the vertex with the highest finish time, i.e., A.

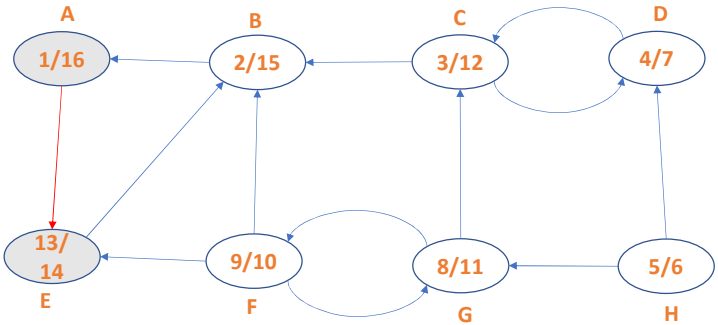


This is the reversed graph. Now we will run DFS-VISIT at the vertex that has the maximum finish time. At the end of the DFS-VISIT, all the nodes that belong to SSC-1 will be marked as black.

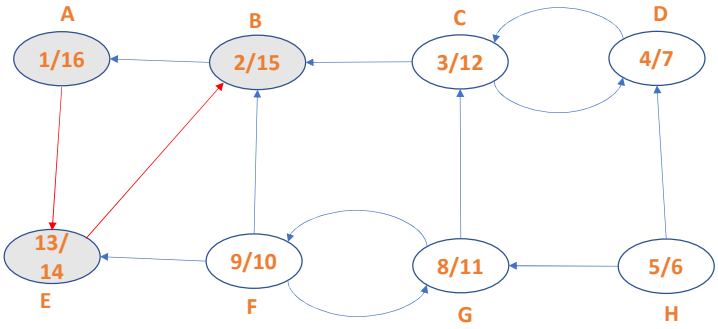
SCC



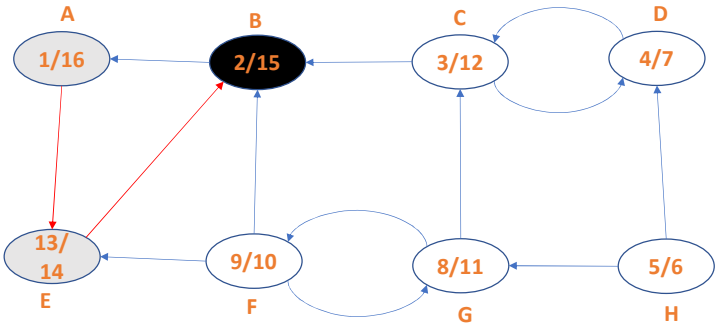
SCC



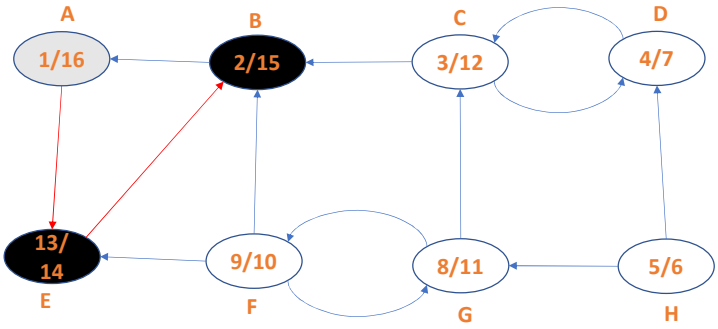
SCC



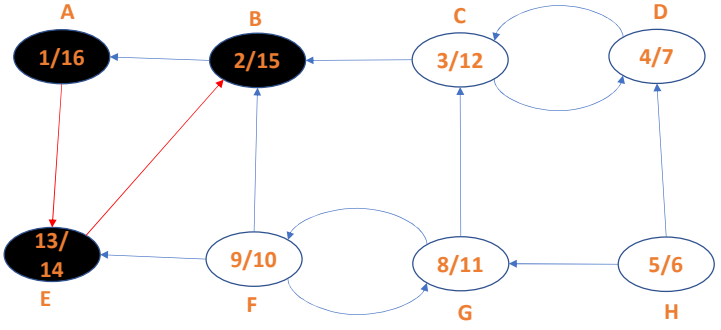
SCC



SCC



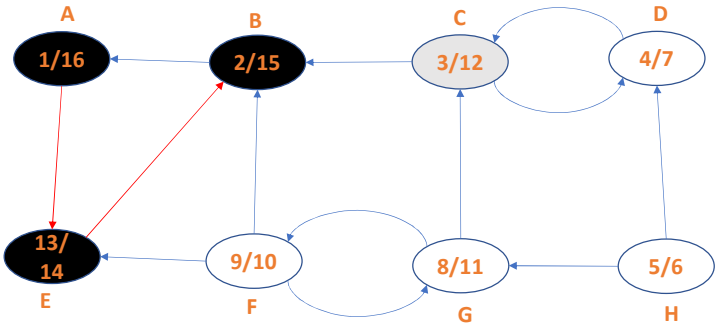
SCC



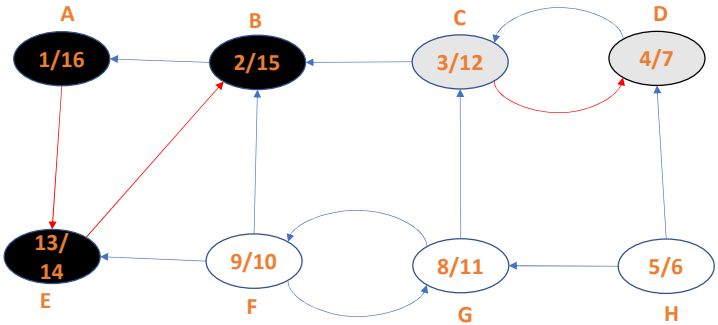
We have identified the vertices in SSC-1. Now from the remaining vertices, the vertex that has the highest finish time, i,e., C must belong to SSC-2. If we run DFS-VISIT at C, all the nodes in SCC-2 will be marked as black. During DFS, we may also encounter vertices from SCC-1. We can easily identify them because they are already black.

SCC

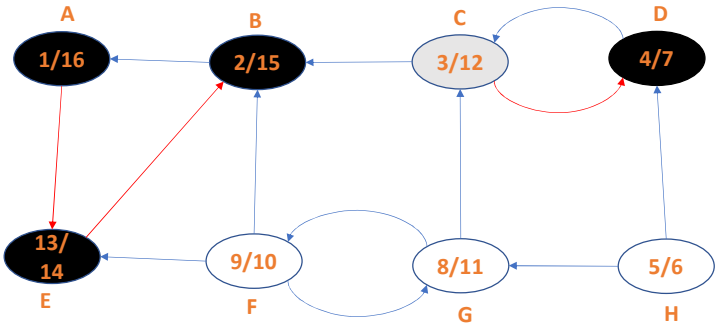
Run DFS at the vertex with the highest finish time among the remaining vertices, i.e., C.



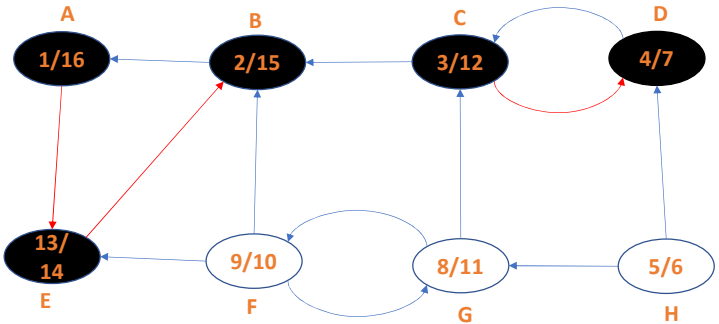
SCC



SCC



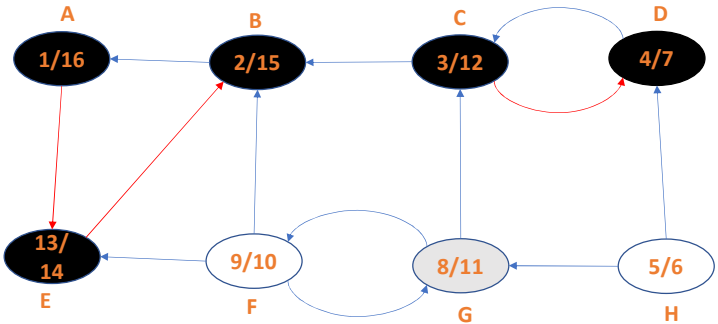
SCC



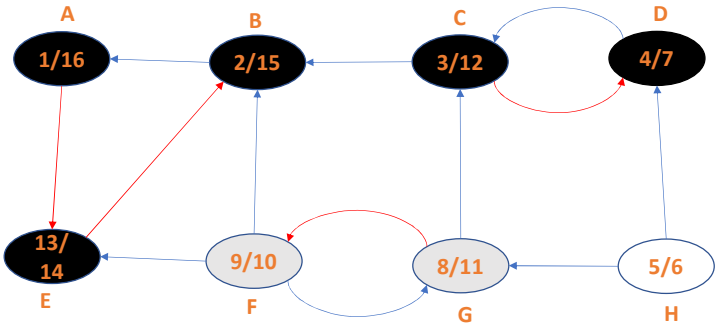
We have identified SCC-2. Vertex G belongs to SCC-3. We will now run DFS-VISIT for G, which will visit all the vertices in SSC-3. We may also encounter some vertices from SCC-1 and SCC-2, but they are already marked as black.

SCC

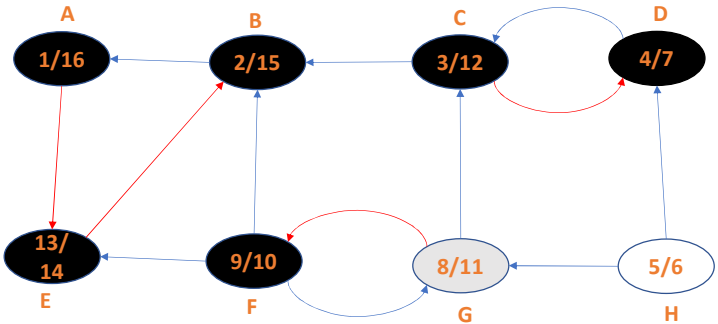
Run DFS at the vertex with the highest finish time among the remaining vertices, i.e., G.



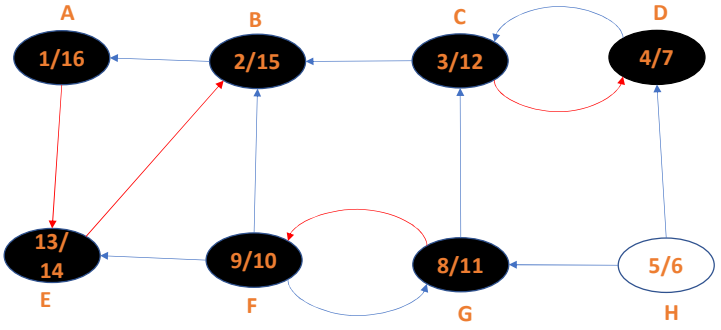
SCC



SCC



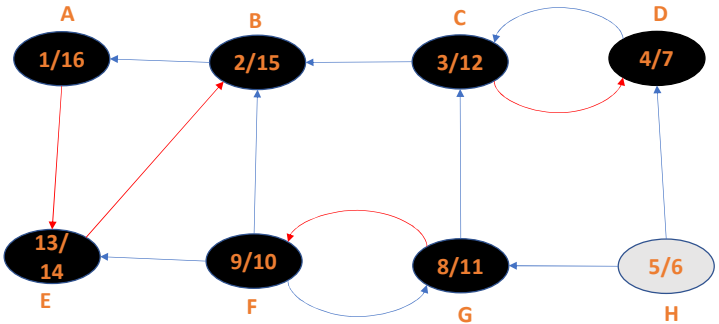
SCC



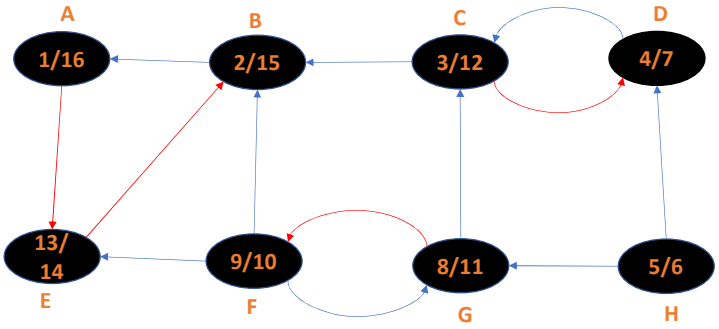
We have found the vertices in SCC-3. Only vertex H is remaining; that will be part of SCC-4.

SCC

Run DFS at the vertex with the highest finish time among the remaining vertices, i.e., H.



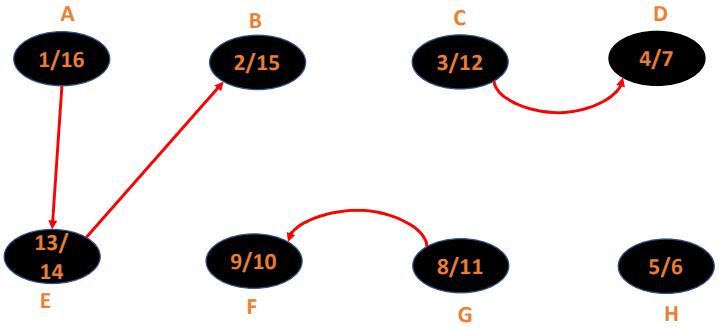
SCC



We are done.

The trees in the depth-first forest are the strongly connected components.

SCC



We are done.

The trees in the depth-first forest are the strongly connected components.

STRONGLY-CONNECTED_COMP

```
STRONGLY-CONNECTED-COMP(G)

// G is a graph (V, E)
// each vertex contains five
fields: color, d, f, π, scc_id
// d is discovery time
// f is finish time
// π is predecessor in the DFS
forest
// scc_id is the scc id of the
vertex

// Output: compute the scc id for
each vertex
```

```
STRONGLY-CONNECTED-COMP(G)

L = DFS-SCC(G)          // Topological sort
REVERSE-ALL-EDGES(G)
RESET-COLORS(G)

scc_id = 0
foreach vertex v in L
   if v.color == WHITE
     scc_id = scc_id + 1
     v.scc_id = scc_id
     DFS-VISIT-SCC-ID(G, v)
```

DFS-SCC

```
DFS-SCC(G)

// G is a graph (V, E)
// each vertex contains five
fields: color, d, f, π, scc_id
// d is discovery time
// f is finish time
// π is predecessor in the DFS
forest
// scc_id is the scc id of the
vertex

// Output: returns a list that
contains all vertices in the
decreasing order of their finish
time
```

```
DFS-SCC(G)
for each vertex u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
L = NIL
for each vertex u ∈ G.V
    if u.color == WHITE
       L = DFS-VISIT-SCC(G, u, L)
return L

DFS-VISIT-SCC(G, u, L)
time = time + 1
u.d = time
u.color = GRAY
for each vertex v ∈ G.Adj[u]
    if v.color == WHITE
       v.π = u
       L = DFS-VISIT-SCC(G, v, L)
time = time + 1
u.f = time
u.color = BLACK
L = insert_front(L, u)
return L
```

DFS-VISIT-SCC-ID

```
DFS_VISIT-SCC-ID(G, s)

// G is a graph (V, E)
// s is the source vertex
// each vertex contains five
fields: color, d, f, π, scc_id
// d is discovery time
// f is finish time
// π is predecessor in the DFS
forest
// scc_id is the scc id of the
vertex

// Output: set the scc id of all
vertices reachable from s to
s.scc_id
```

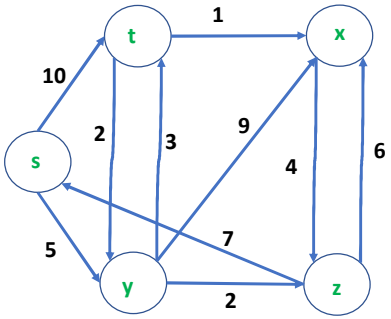
```
1. DFS-VISIT-SCC-ID(G, u)
2. u.color = GRAY
3. for each vertex v ∈ G.Adj[u]
4.     if v.color == WHITE
5.         v.scc_id = u.scc_id
6.         DFS-VISIT-SCC-ID(G, v)
7. u.color = BLACK
```

Weighted graph

Weighted graph

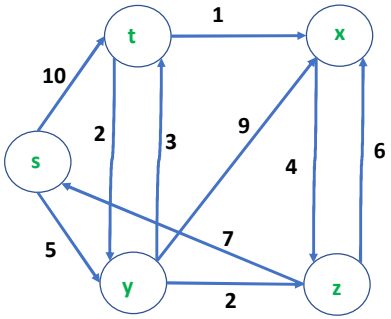
- A weighted graph, G = (V, E, w), contains a weight function, w : E -> R, that maps an edge in the graph to a real numeric value called the weight of the edge

Weighted graph



w(s, t) = 10
w(y, x) = 9
w(x, y) = No such edge
w(x, z) = 4
w(z, x) = 6

Weighted graph

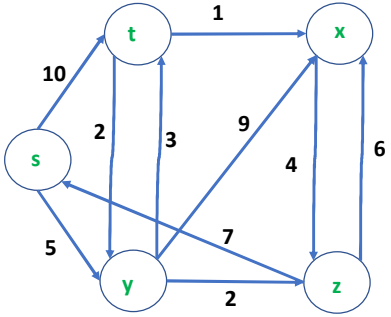


Weight w(p) of path
p = $\langle v_0, v_1, \ldots, v_k \rangle$ is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

w(s, t, x) = 11
w(s, y, z, x) = 13
w(s, y, x, z) = 18

Shortest path



Shortest-path length $\delta(u, v)$ from vertex u to vertex v is defined as

$$\delta(u,v) = \begin{cases} \min(w(p) : p =< u, \dots, v >) & \textbf{\textit{if a path from u to v exists}} \\ \infty & \textbf{\textit{otherwise}} \end{cases}$$

$\delta(s,t) =$ 8
$\delta(s,x) =$ 9
$\delta(z,y) =$ 12
$\delta(t,s) =$ 11

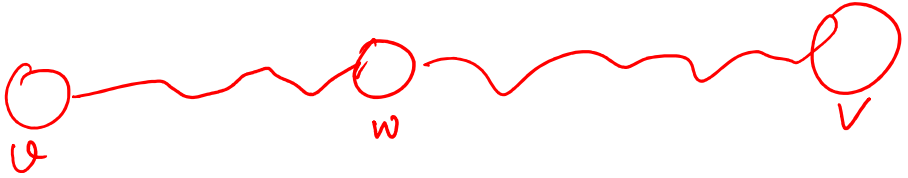
Properties of shortest path

- If all edges have non-negative weights, can a shortest path contain a cycle?

A cycle is always going to add more weight to the path in addition to adding repeating edges. A cyclic shortest path is only possible if the weight of the circular path is zero. Even in that case, we can remove the cycle and get a shortest path. Therefore, we can safely say that if a shortest path exists, then a shortest simple path (i.e., acyclic path) also exists.

Properties of shortest path

- The subpaths of a shortest path are also shortest paths



Let's say the path between vertices u and v is the shortest path. For any intermediate vertex w, the path from u to w (say path-1) is also a shortest path. If this is not true, then there exists another path to w from u (say path-2) with the shortest length. In that case, the path from u to v is not a shortest path because we can take path-2 instead of path-1 to reach w. Hence, there is a contradiction.

Single-source shortest-paths (SSSP)

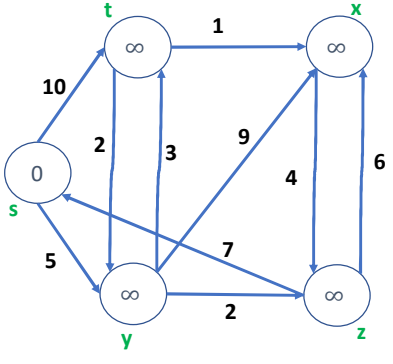
SSSP

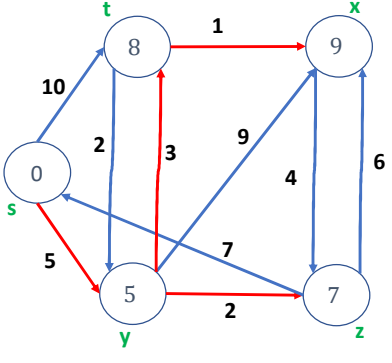
- Read chapter-24 from the CLRS book

SSSP

- Single-source shortest-paths (SSSP)
  - Given a graph G = (V, E, w), find a shortest path from a given source vertex $s \in V$ to every vertex $v \in V$

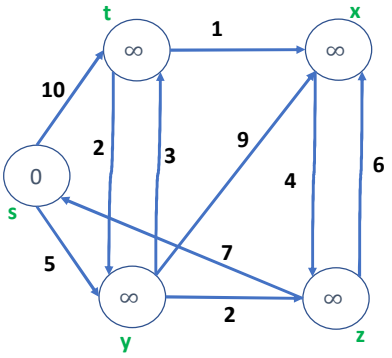
Shortest Path





The goal is to find a shortest path to each vertex from s.

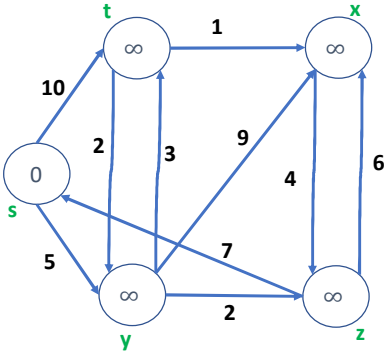
Shortest Path



Using BFS.

We can use BFS to find a shortest path. The BFS algorithm only works if the weight of an edge is one. We can transform this graph to an unweighted graph by adding k-1 additional vertices between two adjacent vertices, where k is the weight of the corresponding edge. However, this solution is inefficient because the weights on the edges can be very large.

Shortest Path




```
Finding all paths.
```

Another solution is to find all possible paths and select a path that has the shortest distance. However, as we show next, the number of possible paths can grow exponentially. Therefore, this solution is also very expensive.

Finding all paths

How many different program paths are possible?

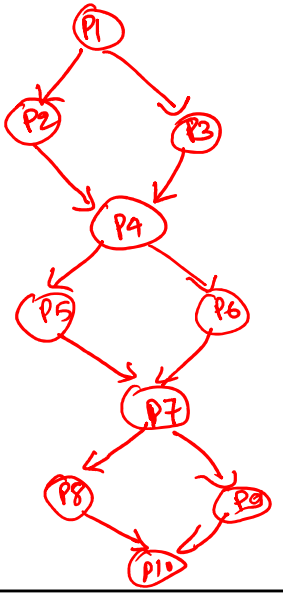
P2 P5 P8
P2 P5 P9
P2 P6 P8
P2 P6 P9
P3 P5 P8
P3 P5 P9
P3 P6 P8
P3 P6 P9

For k if-else number of paths are 2^k

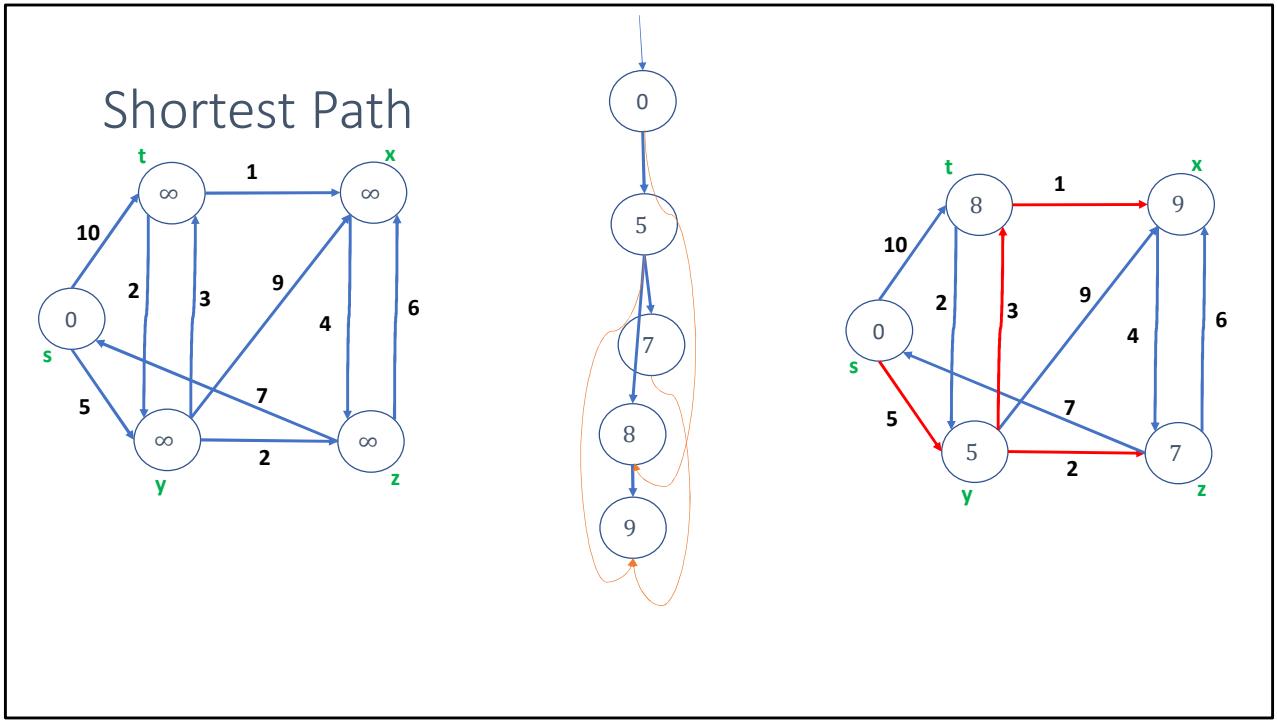
```
if (cond1) {  // P1
  …  // P2
}
else {
  …  // P3
}

if (cond2) { // P4
  …   // P5
}
else {
  …   // P6
}

if (cond3) { // P7
  …   // P8
}
else {
  …   // P9
}
// P10
```

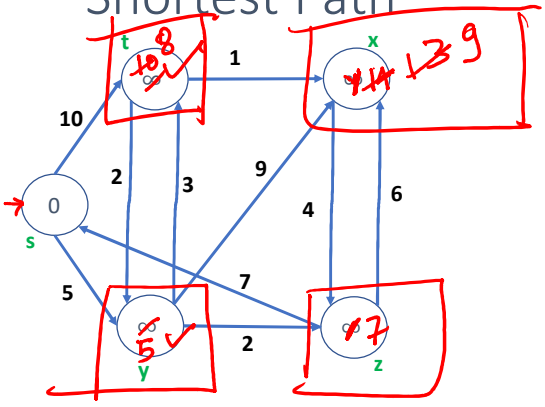


This example contains three if-else conditions. At runtime, the program may take eight different paths. For four if-else conditions, the program may take sixteen different paths at runtime. In general, for k if-else conditions, the program may take 2^k different paths. Therefore the number of paths in a graph can be exponential.

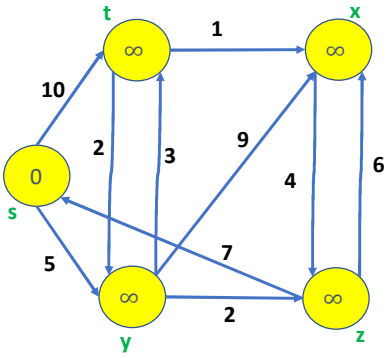


We can also use gravity to find the shortest path. Let's say vertices are some solid objects. If we connect the adjacent objects using a thread of length equal to the weight of the corresponding edge and hang the source object to a hook in the ceiling, then the order in which the objects will appear from top the bottom, would be the order in which we are going to compute the shortest path. All the threads that are not loose are on the shortest path. We also need to remove the threads that are going in the upward direction after hanging them.

Shortest Path



Shortest Path



Vertices that are in the min-heap are shown in yellow.

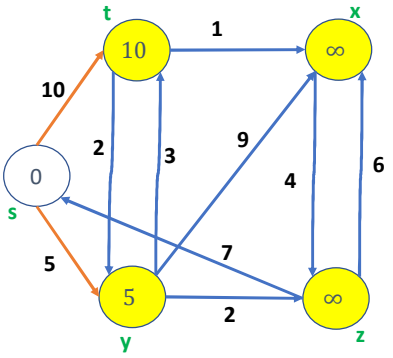
Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

Let's look at this example. Initially, the distance of all vertices except s is infinity. After hanging the source object, the first object that will appear after the source object will be connected to it. Therefore we first need to update the distance of adjacent vertices of s to the shortest known distance from s at this point.

Shortest Path



Vertices that are in the min-heap are shown in yellow.

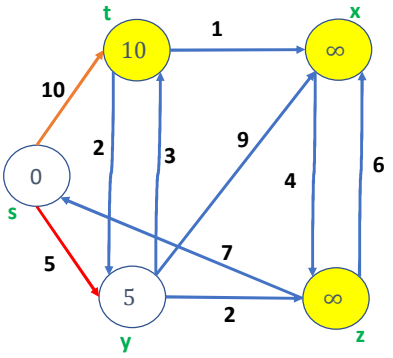
Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

This is the resulting graph after updating the distances. The first vertex that will appear after hanging s will be either y or t. Because y is at a shorter distance than t, therefore, it will appear next.

Shortest Path



Vertices that are in the min-heap are shown in yellow.

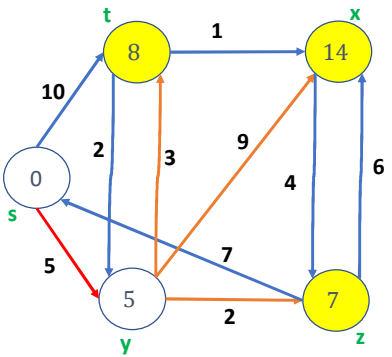
Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

Now we know the first two objects are s and y; the next object that will appear after s and y will be connected to one of them. The next goal is to update the distances of all vertices connected to y. Notice that we have already updated the distance from s previously.

Shortest Path



Vertices that are in the min-heap are shown in yellow.

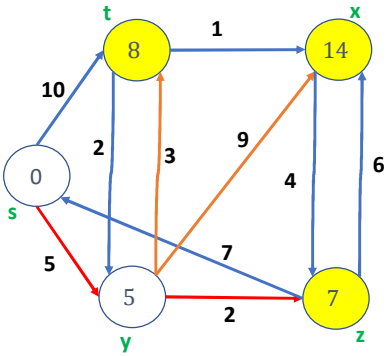
Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

The next vertex that will appear could be connected to either s or y. The one with the shortest distance known so far, i.e., z, will appear next.

Shortest Path



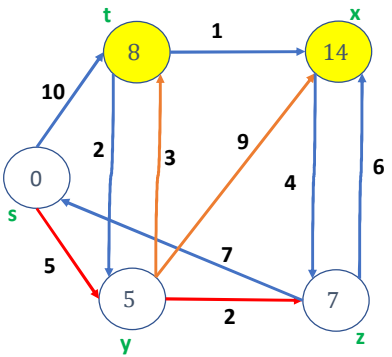
Vertices that are in the min-heap are shown in yellow.

Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

Shortest Path



Vertices that are in the min-heap are shown in yellow.

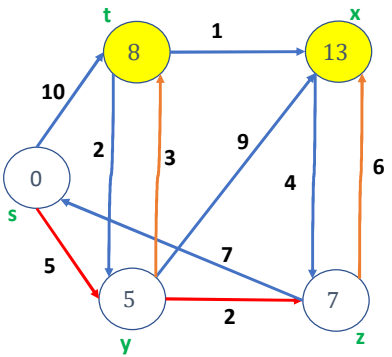
Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

Once we have seen z, the next vertex that may appear could be connected to either s, y, or z. First, we need to update the distances of the adjacent vertices of z if a shorter path exists via z.

Shortest Path



Vertices that are in the min-heap are shown in yellow.

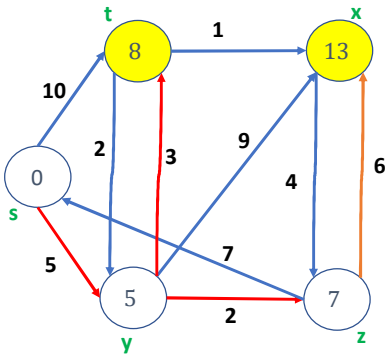
Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

After we have updated the distances of the adjacent vertices of z, the next vertex that will appear in the downward direction would be the vertex with a minimum distance from the remaining vertices. Notice that t has the minimum distance.

Shortest Path



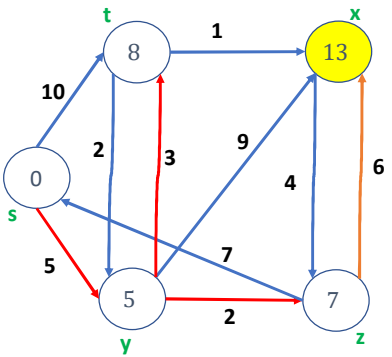
Vertices that are in the min-heap are shown in yellow.

Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

Shortest Path



Vertices that are in the min-heap are shown in yellow.

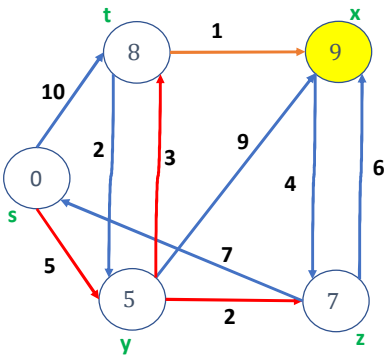
Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

The next object can also be connected to all other objects that we have seen so far. We need to update the distances of the nearby objects of t. We have updated the distance from the other objects that appeared before t.

Shortest Path



Vertices that are in the min-heap are shown in yellow.

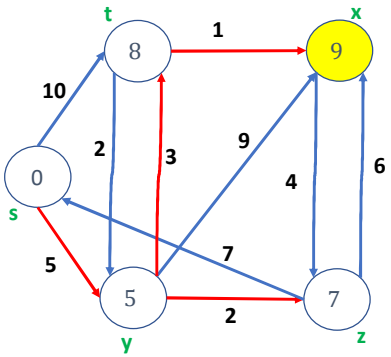
Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

After updating the distance of x, which was also reachable via t, we need to find the vertex that will appear next. Notice that we don't have much of a choice now. Our only choice is x, which will finally appear at the end.

Shortest Path



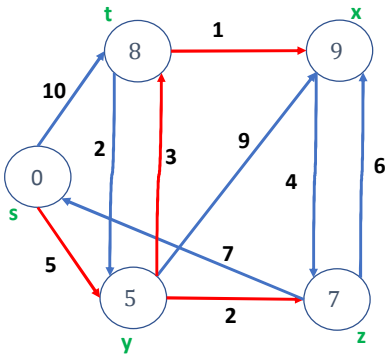
Vertices that are in the min-heap are shown in yellow.

Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.

Shortest Path



Vertices that are in the min-heap are shown in yellow.

Vertices whose shortest paths are known are shown in white.

Red edges are edges on the shortest path.

The origins of the orange edges are the white vertices, and the destinations are the yellow ones. These edges lie on the shortest path to the destination vertex, consisting of only white vertices.