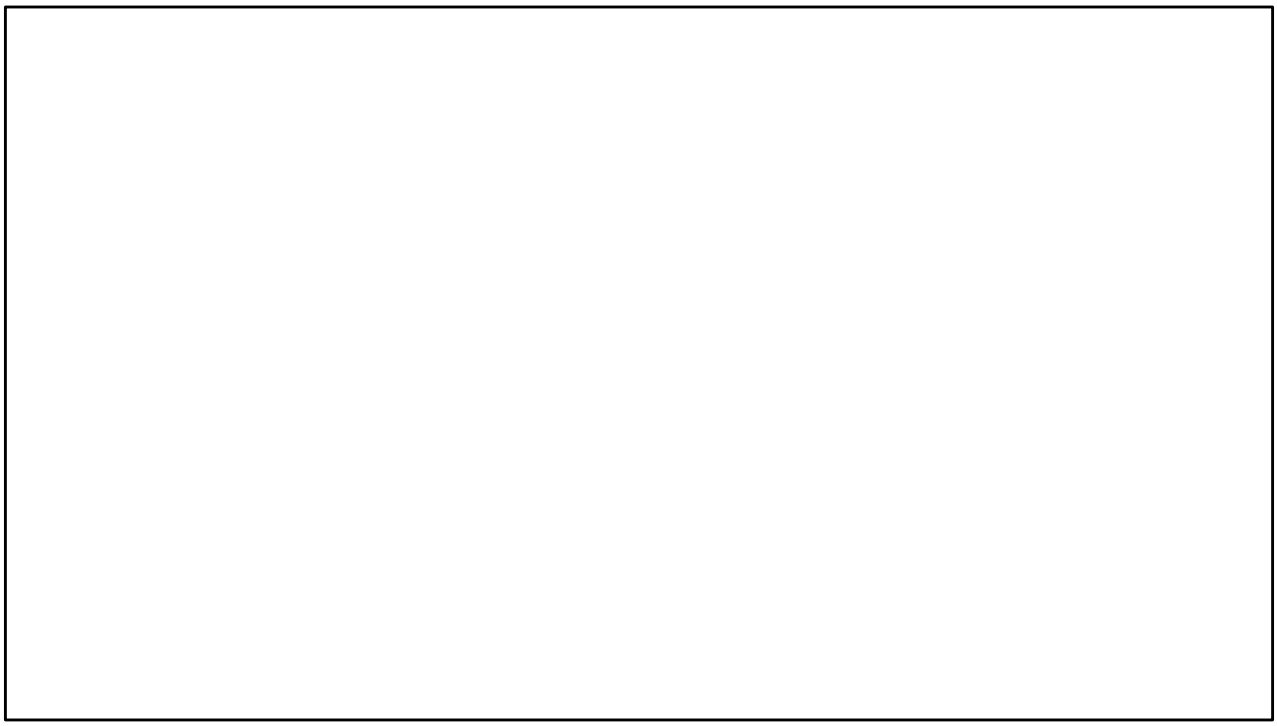
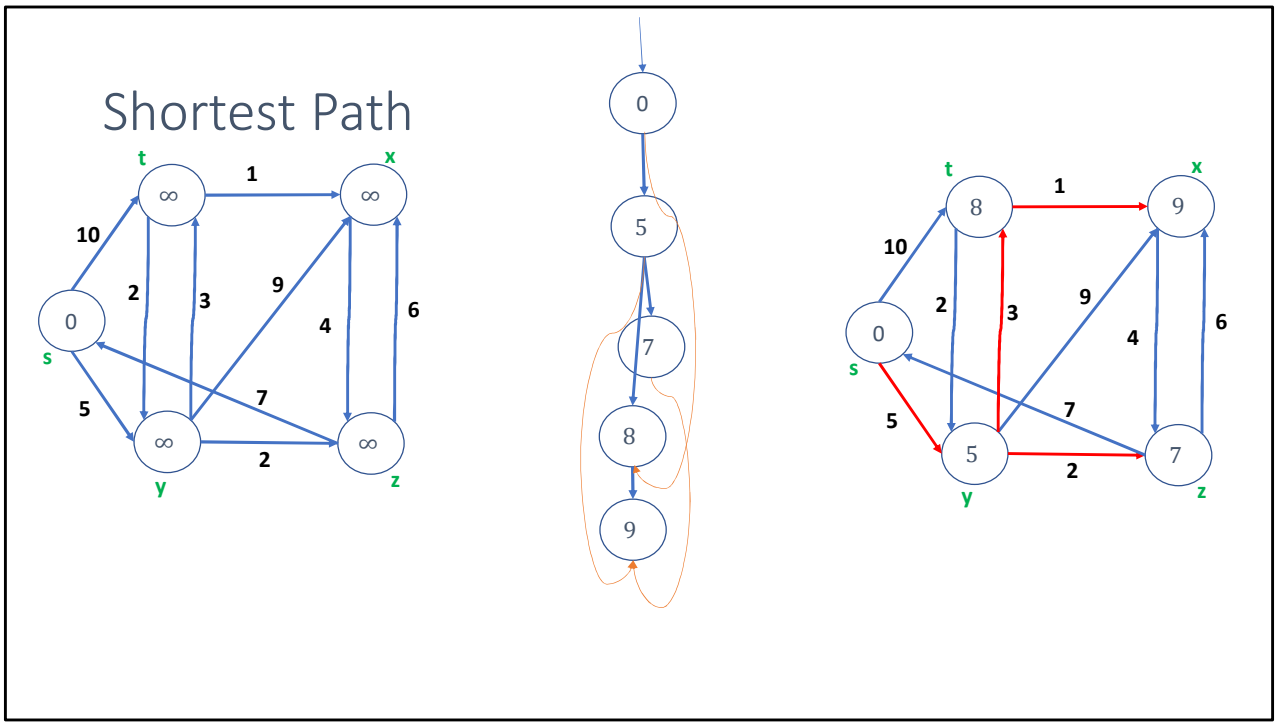


Today's topics

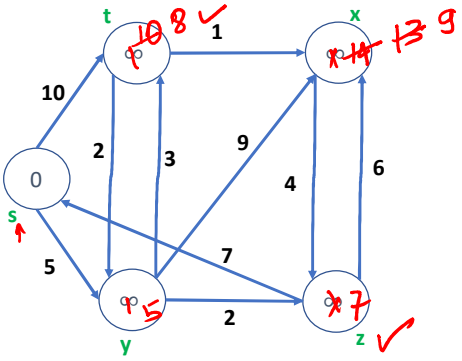
- Shortest paths
- Minimum spanning tree

Single-source shortest-paths

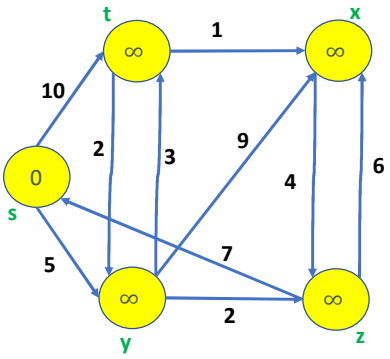


We can also use gravity to find the shortest path. Let's say vertices are some solid objects. If we connect the adjacent objects using a thread of length equal to the weight of the corresponding edge and hang the source object to a hook in the ceiling, then the order in which the objects will appear 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.

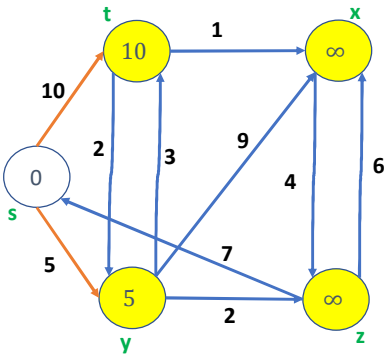
Red edges are edges on the shortest path.

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.

```
All the vertices are initially inserted in a min-heap. The distance of the
source vertex is zero. The distances of other vertices are set to infinity.
```

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.

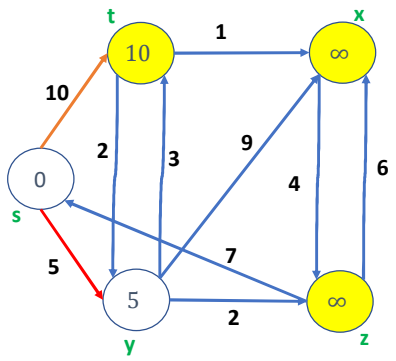
Red edges are edges on the shortest path.

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.

```
We extract the minimum element from the min-heap. This is the vertex whose
shortest path has been computed. We also update the distances of adjacent
vertices using decrease-key operations if a shorter path is available from
the extracted vertex.
```

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.

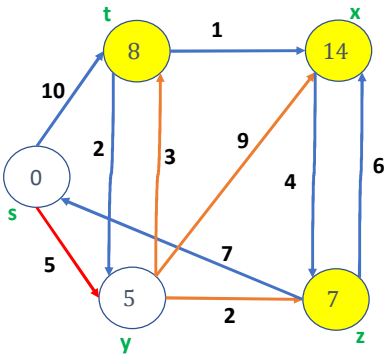
Red edges are edges on the shortest path.

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.

```
Extracting minimum element from the min-heap. The yellow vertices are still
part of the min heap. The white ones are extracted 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.

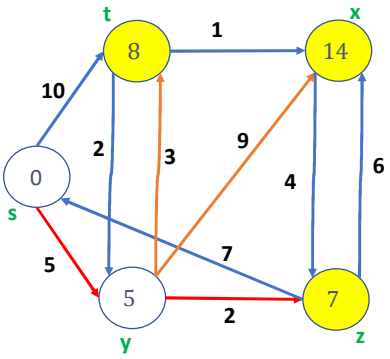
Red edges are edges on the shortest path.

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.

```
Updating distances in the min-heap.
```

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



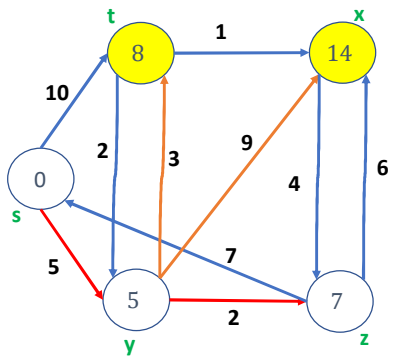
Vertices that are in the min-heap are shown in yellow.

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



Extracting minimum.

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

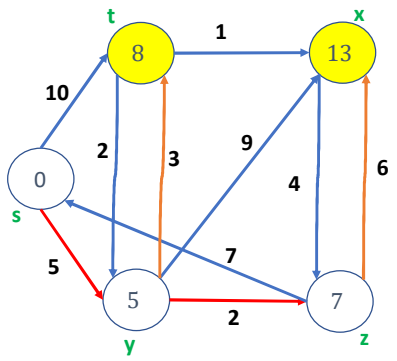
Vertices whose shortest paths are known are shown in white.

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



Updating distances.

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

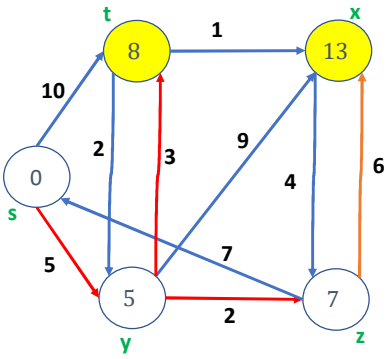
Vertices whose shortest paths are known are shown in white.

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



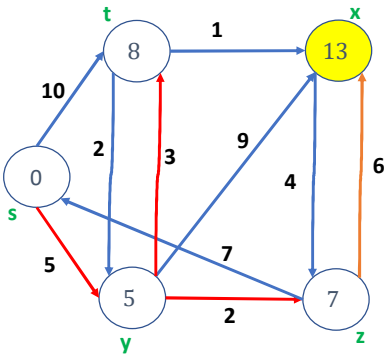
Vertices that are in the min-heap are shown in yellow.

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



Extracting minimum.

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

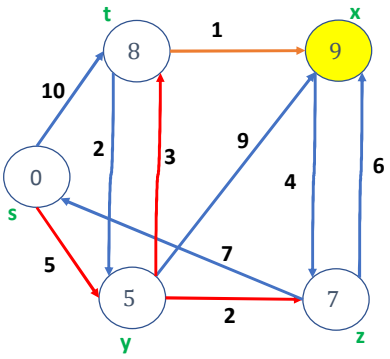
Vertices whose shortest paths are known are shown in white.

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



Updating distances.

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

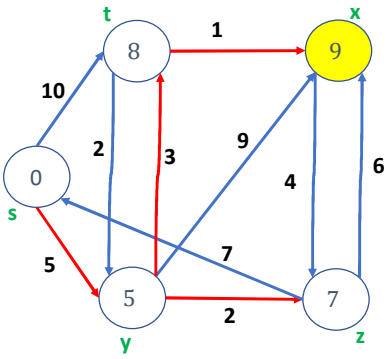
Vertices whose shortest paths are known are shown in white.

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



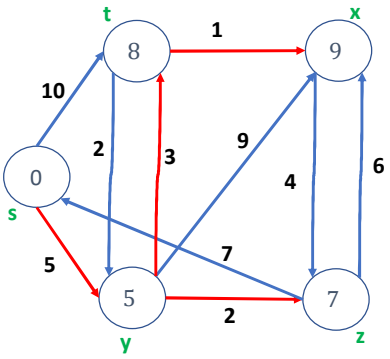
Vertices that are in the min-heap are shown in yellow.

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.

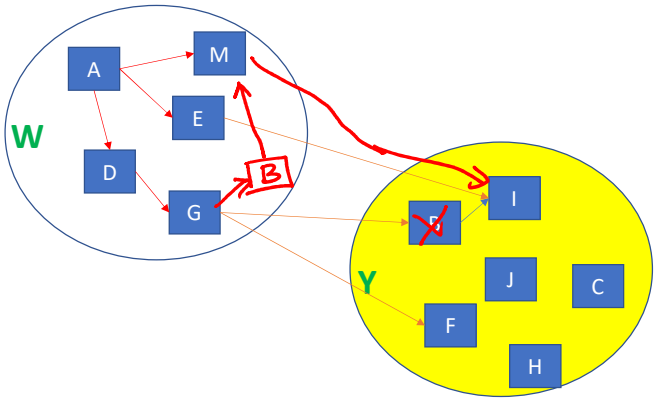
Vertices whose shortest paths are known are shown in white.

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.

```
Extracting minimum. Nothing in the min-heap. Algorithm terminates.
```

Shortest path invariants



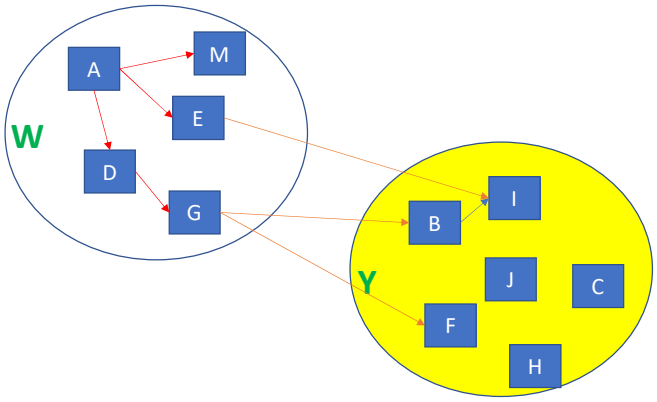
The vertices can be divided into two sets, W and Y.

d[v] stores the current known distance from the source.

Invariant-1: The vertices in the set Y are the ones for which d[v] contains the length of a shortest path that doesn't include vertices in Y – {v}, for each v ∈ Y.

Invariant-2: The vertices in set W are the ones for which d[v] contains the shortest distance from the source, for each v ∈ W.

Shortest path invariants



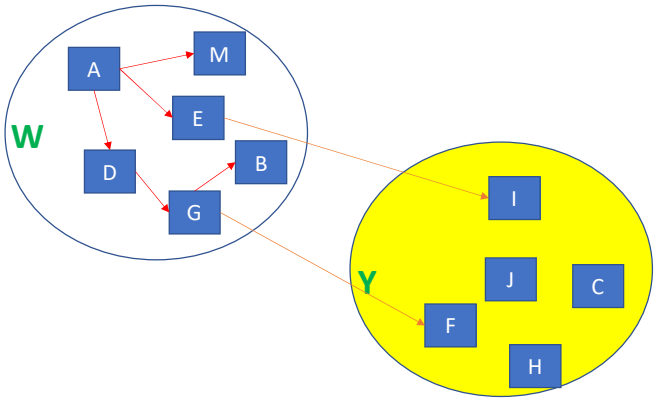
The vertices can be divided into two sets, W and Y.

d[v] stores the current known distance from the source.

Invariant-1: The vertices in the set Y are the ones for which d[v] contains the length of a shortest path that doesn't include vertices in Y – {v}, for each v ∈ Y.

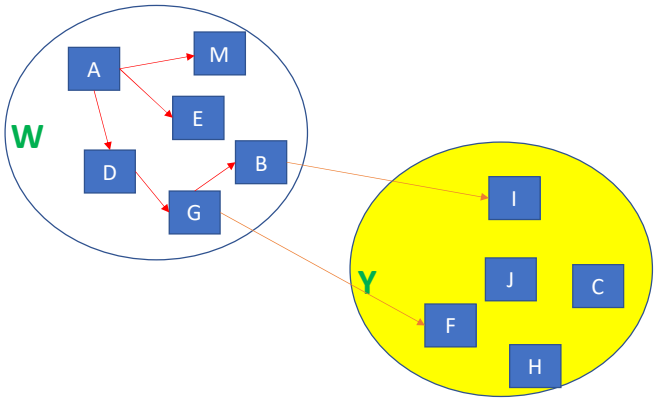
Invariant-2: The vertices in set W are the ones for which d[v] contains the shortest distance from the source, for each v ∈ W.

Shortest path invariants



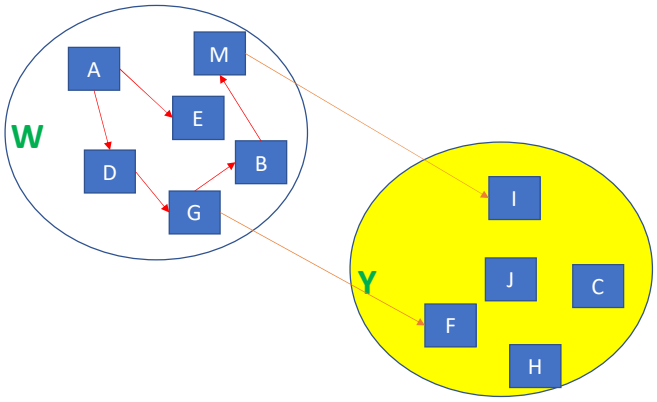
After moving a vertex, say B to W, let's say B is not on the shortest path to any other vertex in Y. In that case, invariant-1 holds because the distance of I will not be updated.

Shortest path invariants



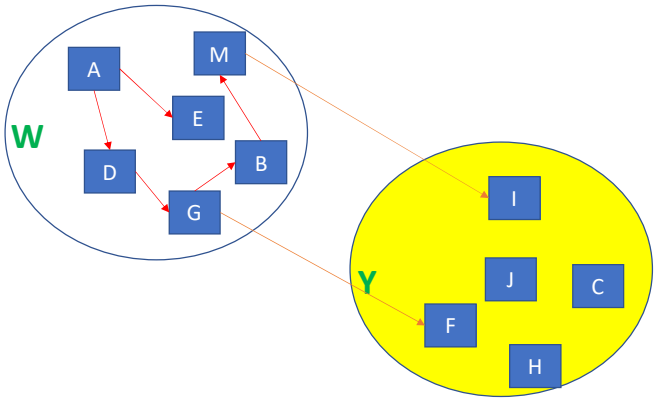
After moving a vertex, say B to W, let's say B is on the shortest path; in that case, invariant-1 still holds because the distance of I will be updated to the min(d[I], d[B]+w(B,I)).

Shortest path invariants



After moving a vertex, say B to W, let's say B is on the shortest path to I via M, where M belongs to set W. Does invariant-1 hold in this case?

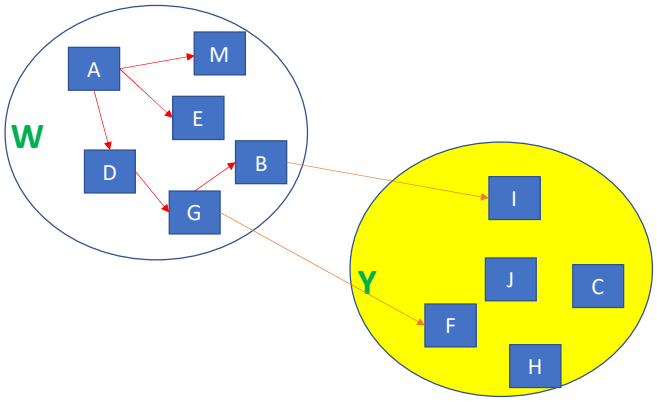
Shortest path invariants



After moving a vertex, say B to W, let's say B is on the shortest path to I via M, where M belongs to set W. Does invariant-1 hold in this case?

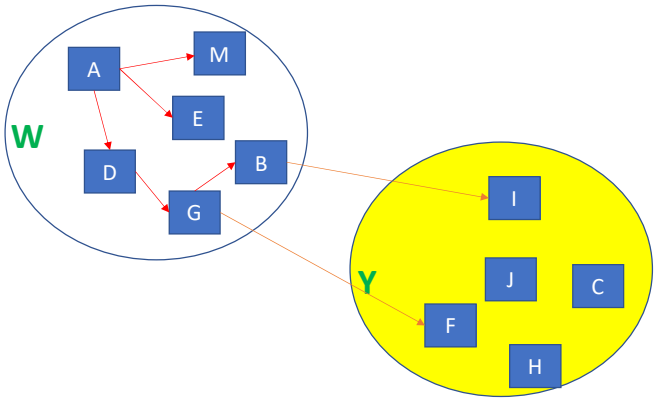
This is not possible. We already know the shortest path to M from the source because of the invariant-2. It is possible that B is indeed on the shortest path to I via M; in that case, the length of the shortest path to M via B must be equal to the length of the previous shortest path without B. Therefore, nothing needs to be changed, and invariant-1 still holds.

Shortest path invariants



After moving a vertex, say B to W, does invariant-2 hold?

Shortest path invariants



After moving a vertex, say B to W, does invariant-2 hold?

Let's say invariant-2 doesn't hold. Because of invariant-1, we know that the d[B] is the length of a shortest path to B, excluding the vertices in set Y – {B}. It means there must be another vertex from set Y – {B} on the shortest path to B if invariant-2 doesn't hold. Let's say x is the first vertex that belongs to set Y – {B} and lies on the shortest path to B. But this can only be true if length(x, B) == 0 and d[x] = d[B]. This is because we move a vertex y from Y to W only if d[y] >= d[z], for all z ∈ Y. Therefore, the path to B without x is indeed a shortest path.

Dijkstra's Algorithm

```
DIJKSTRA(G, s)

// G is a graph (V, E, w)
// s is the source vertex
// each vertex contains two
fields, d and v
// d contains the distance of the
best known path at a given point
// π contains the predecessor
vertex on the best known path at a
given point

// Output: shortest distance to
each vertex v from s in v.d; and
the predecessor on the shortest
path to vertex v in v.π
```

```
1. INITIALIZE-SINGLE-SOURCE(G, s)
2. for each vertex v ∈ G.V
3.   v.d = ∞
4.   v.π = NIL
5. s.d = 0
```

```
1. DIJKSTRA(G, u)
2. INITIALIZE-SINGLE-SOURCE(G, s)
3. S = ϕ  // S is a set
4. Q = ϕ  // Q is min-heap
5. for each vertex u ∈ G.V
6.   INSERT(Q, u)
7. while Q != ϕ
8.   u = EXTRACT-MIN(Q)
9.   S = S ∪ {u}
10.  for each vertex v in G.Adj[u]
11.    if v.d > u.d + G.w(u, v)
12.      v.d = u.d + G.w(u, v)
13.      v.π = u
14.      DECREASE-KEY(Q, v, v.d)
```

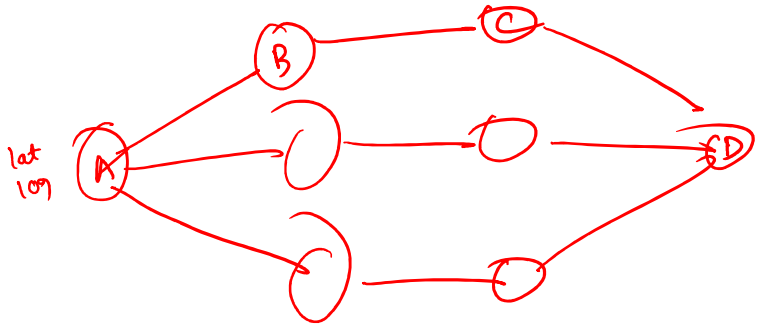
Time complexity

Time complexity

- Inserting all vertices at line-6 in the min-heap will take $O(|V|)$ operations; if we use Build-Min-Heap
- Extract-min at line-8 will be performed $|V|$ times; thus, it will require $O(|V|*\log(|V|))$ operations
- Decrease key at line-14 will be performed $|E|$ times; thus, it will require $O(|E|*\log(|V|))$ operations
- Therefore, the time complexity is $O((|V|+ |E|)*\log(|V|))$

Assignment-4

Computing SSSP



Computing SSSP

- Each user data is stored in an object of type struct record
- struct record also stores the location of the user
- Users can have friends
- For every user, a linked list of references to friends is maintained
  - The head of the linked list is stored in the struct record of the user

Computing SSSP

- For a given user U
  - Each friend of a U is reachable from U
  - Every user that is reachable from a friend of U is also reachable from U
  - If U is reachable from V, then V is also reachable from U
- There is an undirected edge between users U and V if they are friends
  - The weight of the edge is the distance between U and V
  - You can use the distance function provided to you to compute the distance between two users
- If two users are reachable from each other, then a path must exist between them

Computing SSSP

- The goal of this assignment is to compute a shortest path and the shortest distance from a given user U to all other users that are reachable from U

- The user's data is stored in struct record
  - You can use status, distance, and pred fields in the struct record
  - Initially, the status is set to zero in all records reachable from the source
  - The distance and pred may contain garbage values
  - At the end of your algorithm, distance and pred must contain the shortest distance and the predecessor (address of struct record) on a shortest path

Computing SSSP

- min-heap
  - The min-heap implementation is optional
  - You can also use a linear time algorithm to find a vertex with the minimum distance
- For the min-heap or linear time algorithm, you can use min_heap_arr
  - min_heap_arr is large enough to store references to all vertices in the graph
  - Notice that the references to vertices (i.e., users) are of type "struct record*"

Making friends

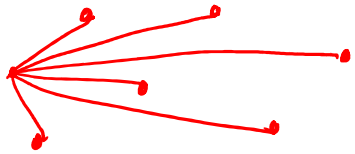
- The friends field in struct record contains the head of the linked list that stores references to friends' data
  - You are not allowed to allocate memory except for the nodes in this linked list

Minimum spanning tree (MST)

Minimum spanning tree

- Read chapter-21 from the CLRS book

Electronic circuit design

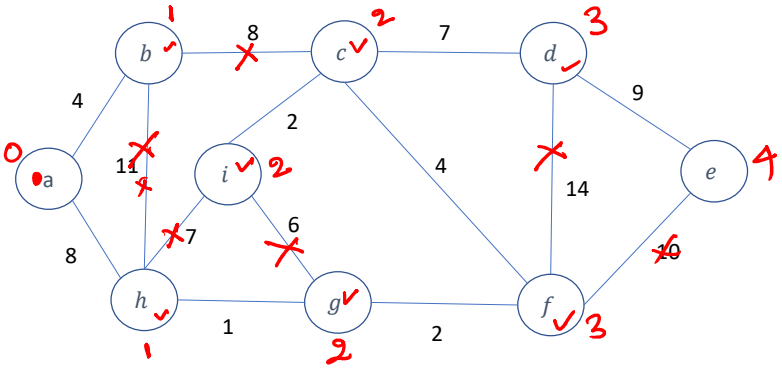


- We often need to connect the pins in electronic circuit design using wires
- To interconnect a set of n pins, the designer can use n-1 wires
- There are several ways to connect the pins; the one that requires the least amount of wire is the desirable design

Minimum spanning tree

- Every connected undirected graph has a spanning tree
- The weight of a spanning tree is the sum of the weights of all the edges
- Out of all possible spanning trees for a connected undirected graph, the one with the minimum weight is the minimum spanning tree

Minimum spanning tree

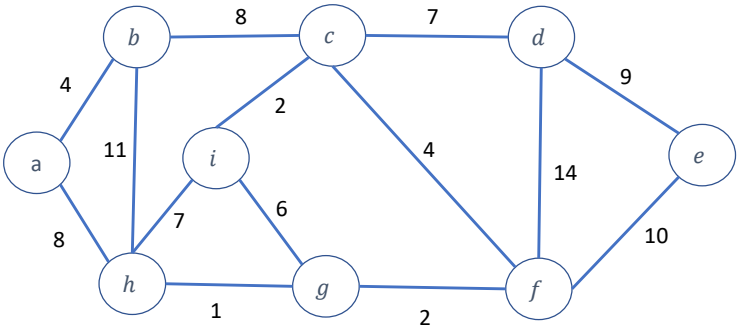


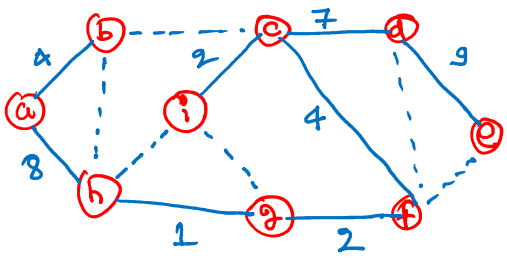
To find the minimum spanning tree, we can use BFS to find a cycle, and whenever a cycle is encountered, we can walk all edges in the cycle and remove an edge with the largest weight. If we keep doing this until all the cycles are removed, then we will end up with an MST. However, the problem with this approach is that every time we encounter a cycle, we need to walk all the edges in the cycle that could be O(|V|); therefore, the total time complexity could be O(|E|*|V|), which is very high.

Minimum spanning tree

- An approach that tries to remove edges from the graph to obtain a spanning tree may need to remove $O(|V|^2)$ edges because $|E|$ is $O(|V|^2)$
- On the other hand, an approach that focuses on identifying edges that are part of a spanning tree may need to identify only $|V|-1$ edges
- The efficiency of the algorithm depends on how quickly we can select an edge for removal or identify an edge that belongs to a spanning tree

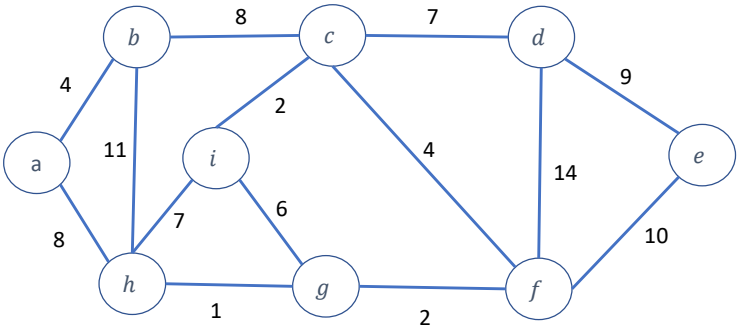
Kruskal's algorithm





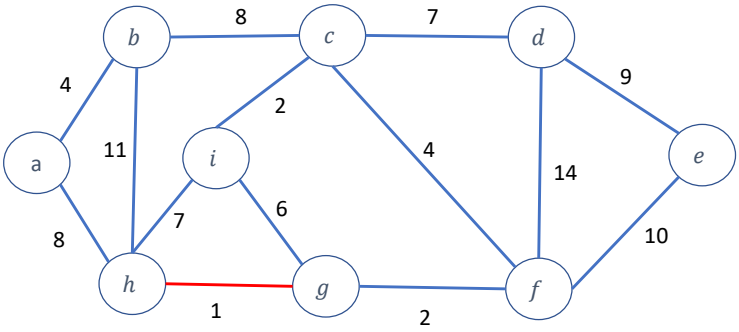
Kruskal's algorithm computes an MST T from graph G. Initially, T contains all the vertices of G without any edges. Afterward, we remove an edge with the minimum weight and add it to T if it doesn't create a cycle. We can ignore the edge if it creates a cycle. We keep removing and adding edges until |V|-1 edges are added to T. After the algorithm terminates, T is actually an MST. One challenging part of this algorithm is to find out whether adding an edge creates a cycle. We will discuss this next.

Kruskal's algorithm

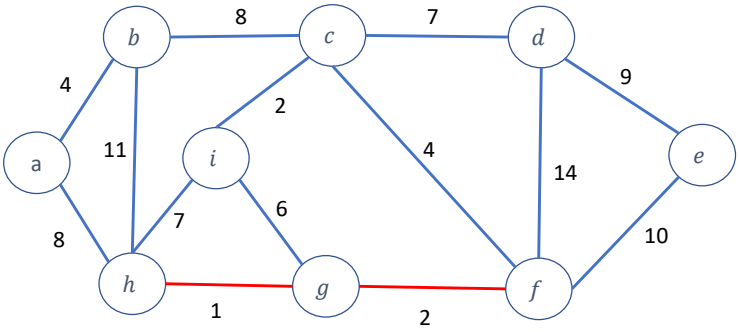


Kruskal's algorithm

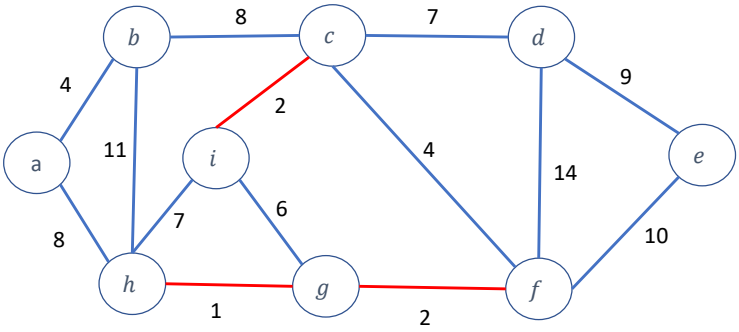
Red edges are part of a MST.



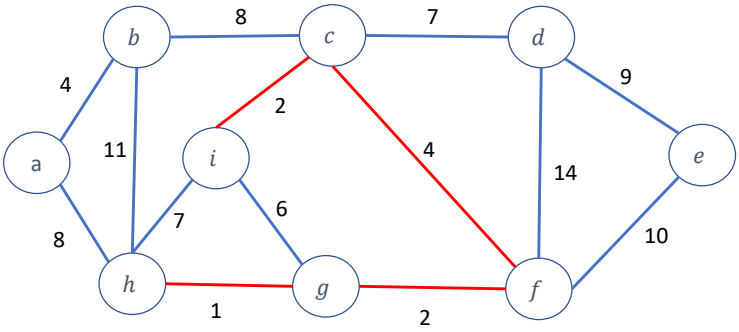
Kruskal's algorithm



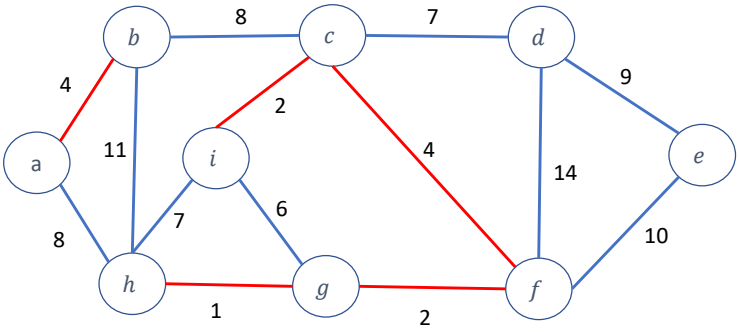
Kruskal's algorithm



Kruskal's algorithm

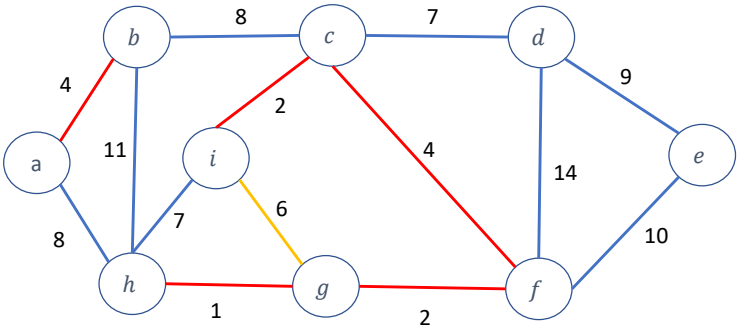


Kruskal's algorithm

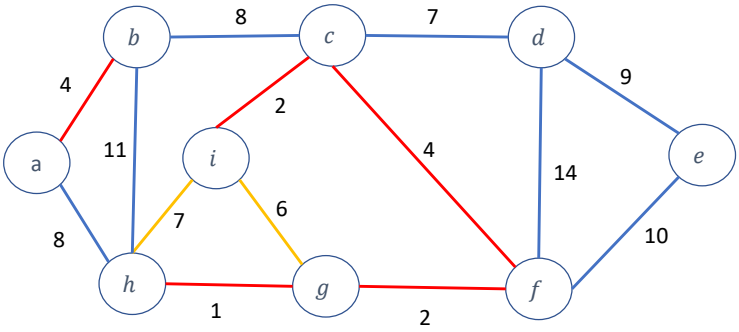


Kruskal's algorithm

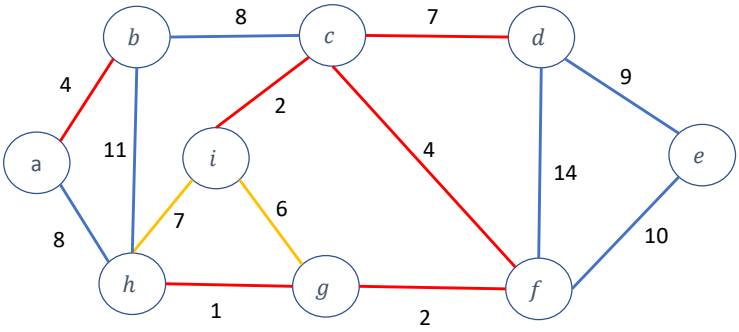
Orange edges are ignored.



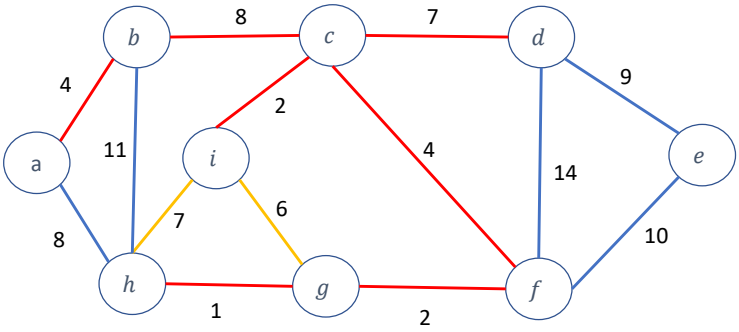
Kruskal's algorithm



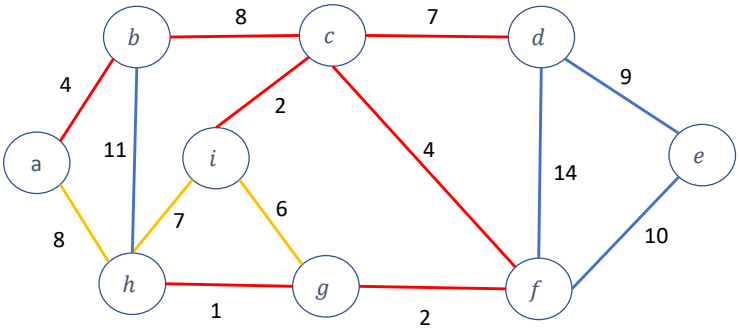
Kruskal's algorithm



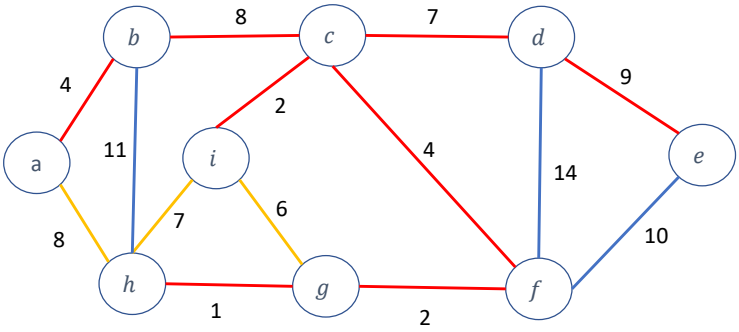
Kruskal's algorithm



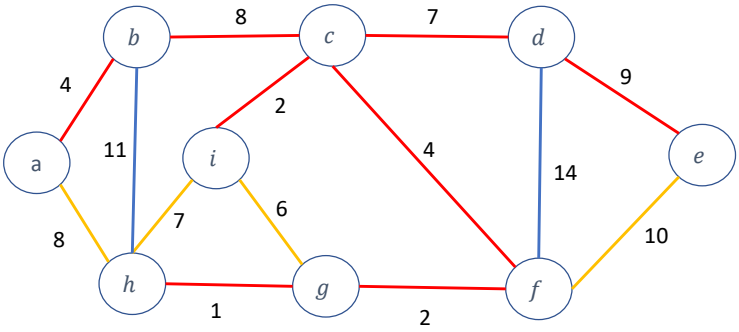
Kruskal's algorithm



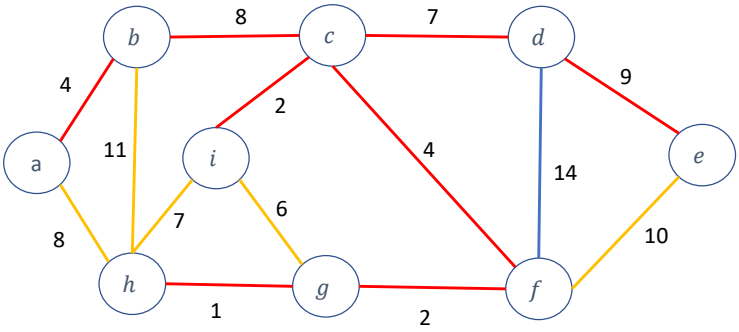
Kruskal's algorithm



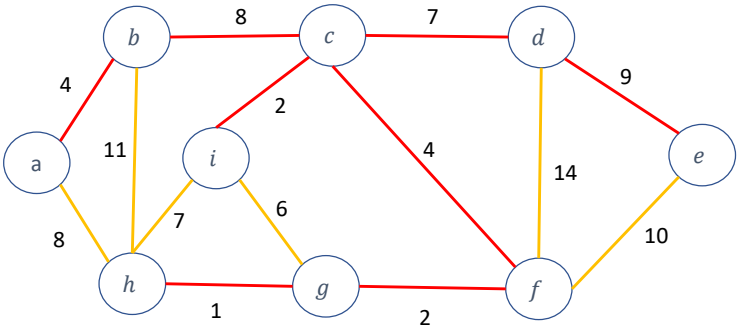
Kruskal's algorithm



Kruskal's algorithm



Kruskal's algorithm



Kruskal's algorithm

- How can we cheaply find the orange edges during the Kruskal's algorithm?

Kruskal's algorithm

- How can we cheaply find the orange edges during Kruskal's algorithm?
  - We can use union and find algorithms that we will discuss in a while