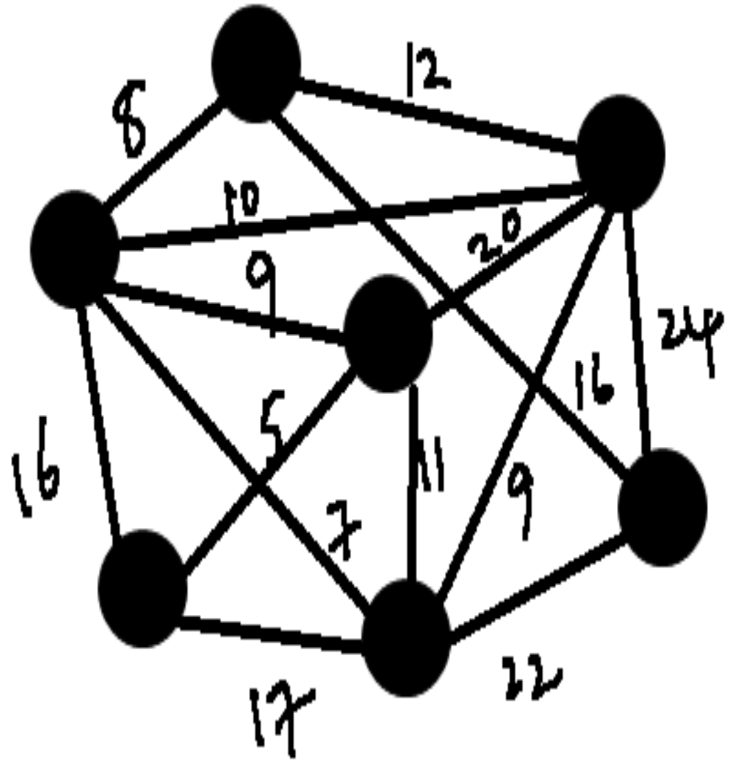
Data Structures & Algorithms for Problem Solving (CS1.304)

Lecture # 16 : MST

Avinash Sharma
Center for Visual Information Technology (CVIT),
IIIT Hyderabad

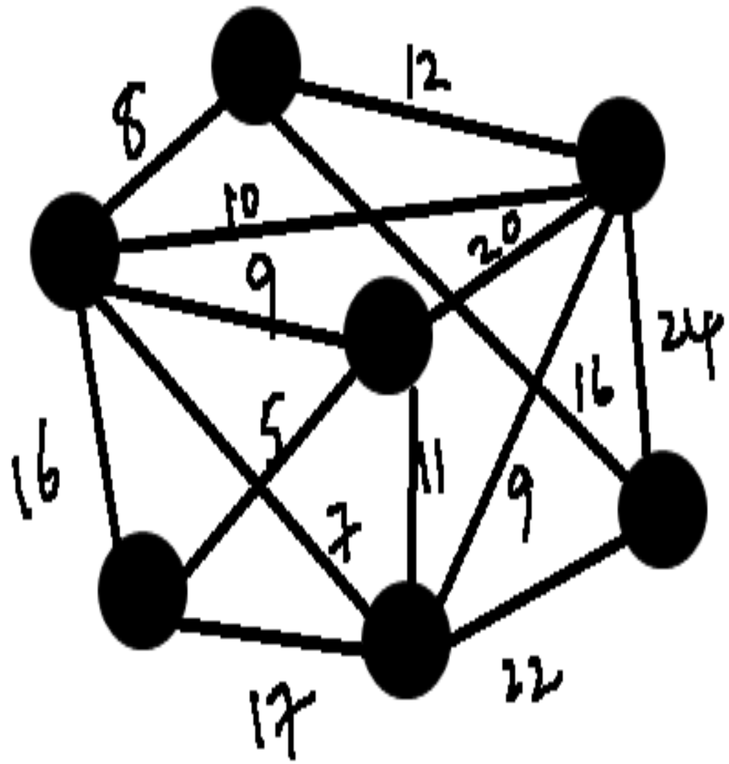
Spanning Trees

- We will now consider another famous problem in graphs.
- Imagine providing connectivity to a set of cities.
- Each highway connects two cities
- In reality, each highway requires a certain cost to be built.



Spanning Trees

- So, there is a trade-off here.
- How to provide connectivity while minimizing the total cost of building the highways.
- The weights on the edges indicate the cost of building that highway.
- The total cost of connectivity = sum of all the built up highway.
- Minimize this cost.



Spanning Trees

- Formalize the problem as follows.
- Let G = (V, E, W) be a weighted graph.
- Find a subgraph G' of G that is connected and has the smallest cost
  - Cost is defined as the sum of the edge weights of edges in G'.

Spanning Trees

- Observation I : If G' has a cycle and is connected, then there exists a G'', which is also a subgraph of G and is connected so that

$$\text{cost}(G'') < \text{cost}(G')$$

- To get G'', simply break at least one cycle of G'.

- Hence, the optimal G' shall have no cycles and is connected.
  - Suggests that G' is a tree.

Spanning Trees

- Two keywords : spanning and tree.
- Some notation: A subgraph G' of G is called a spanning subgraph if V(G') = V(G).
- A spanning subgraph G' of G that is also a tree is called as a spanning tree of G.

Spanning Trees

- Consider the problem: Find a spanning tree of G that has the least cost.
- Such a spanning tree is also called as a minimum cost spanning tree of G. Often one refers to this as the minimum spanning tree, or MST for short.

MST

- Let us now think of devising an algorithm to construct an MST of a given weighted graph G.
- There are several approaches, but let us consider a bottom-up approach.
- Let us start with a graph (tree) that has no edges and add edges successively.
- Every new edge we add should not create a cycle.
- Further, the total cost of the final tree should be the least possible.

MST

- Suggests that we should prefer edges of smaller weight.
  - But should not add edges that create cycles.
- Indeed, that is intuitive and turns out that is correct too.
  - we will skip the proof of this.

MST Algorithm

Algorithm MST(G)

begin

sort the edges of G in increasing order of weight as $e_1, e_2, ..., e_m$

$k = 1; V(T) = V(G); E(T) = \Phi$

while $|E(T)| < n-1$ do

if **$E(T) \cup e_k$ does not have a cycle** then

$E(T) = E(T) \cup e_k$

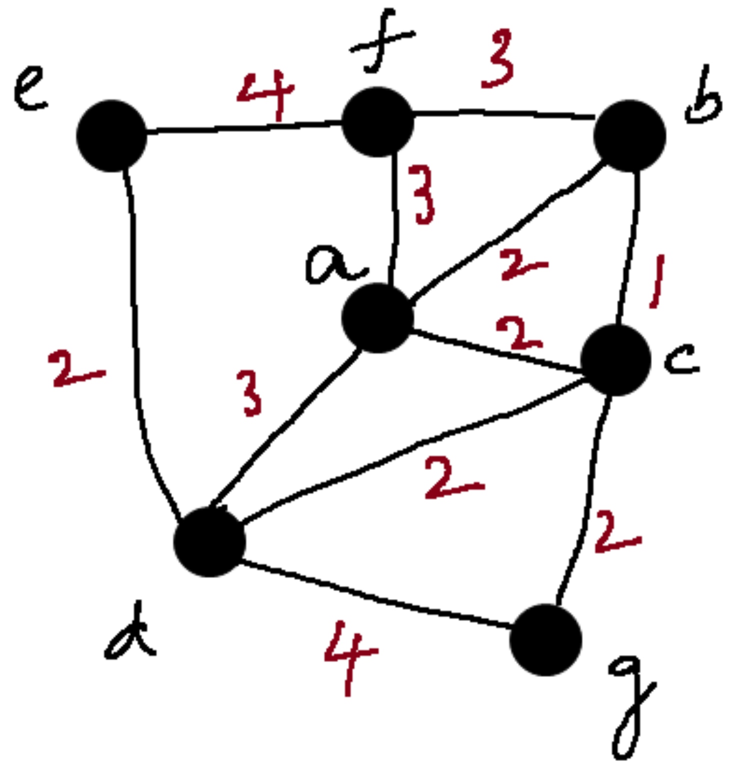
end-if

$k = k + 1;$

end-while

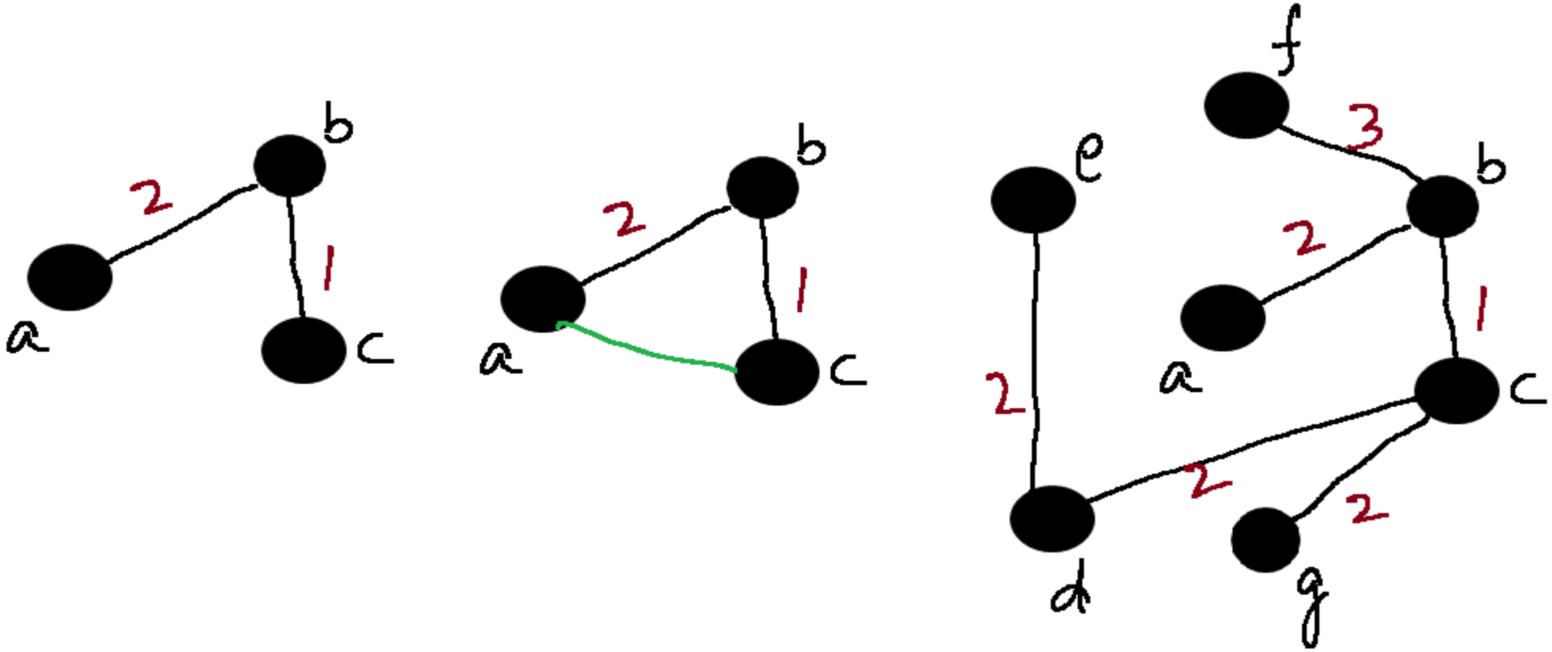
End.

MST Practice Problem



MST

- List of edges by weight
  - bc, ab, ac, cg, cd, de, bf, af, ad, ef, dg



MST Algorithm Analysis

- The algorithm we devised is called the Kruskal's algorithm.
- Belongs to a class of algorithms called greedy algorithms.
- How do we analyze our algorithm?
  - Need to know how to implement the cycle checker.

MST Algorithm Analysis

- How quickly can we find if a given graph has a cycle?
  - O(m+n) is possible using DFS.
- Notice that if the graph is a forest, then m = O(n).
- So, can be done in O(n) time.
- Also, need to try all m edges in the worst case.
- So the time required in this case is O(mn).

MST Algorithm Analysis

- Too high in general.
- But, advanced data structures exist to bring the time down very close to O(m+n).
  - Cannot be covered in this class.
  - We will show an approach that takes us almost there.

Advanced Data Structures

- An abstract problem:
- Given n elements, grouped into a collection of disjoint sets S_1, S_2, …, S_k, design a data structure to:
  - Find the set to which an element belongs
  - Combine two sets
- The abstract problem finds applications in several settings:
  - Spanning tree algorithm of Kruskal
  - Graph connected components
  - Least common ancestors
  - …

Notations for Disjoint Sets

- Imagine a collection $S = \{S_1, S_2, \ldots, S_k\}$ of sets.
- Each set has a **representative** element
  - Some member of the set, typically.
  - Depending on application, can be
    - The smallest numbered element
    - A number …
- Typical operations
  - **MakeSet(x)** - Creates new set whose only member is x. The representative is x
  - **Union(x, y)** - Unites set S_x containing x and set S_y containing y into a new set S and removes S_x and S_y from the collection.
  - **FindSet(x)** - Returns representative of the set holding x

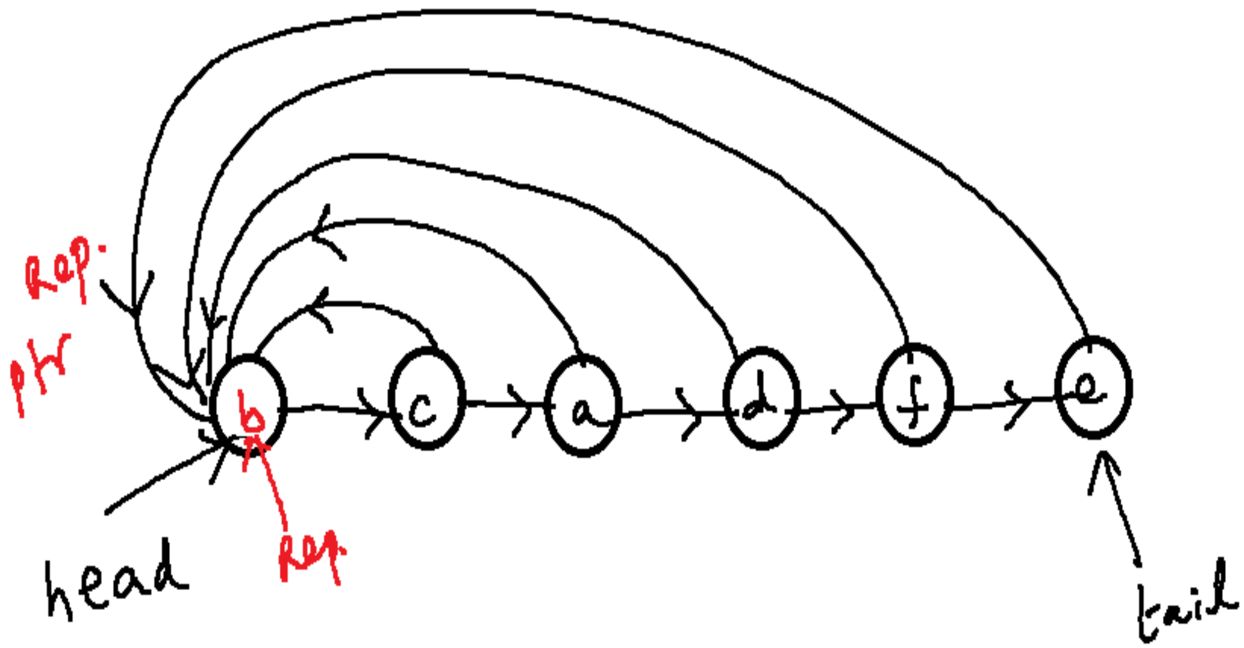
Some Notations

- Two parameters
  - n : The number of MakeSet operations.
  - m : The total number of MakeSet, Union, and Find operations.
- Some observations
  - Each Union operation reduces the number of sets by 1.
  - When starting with n elements, at most n-1 Union operations.
  - Also, m >= n.
- Assume that the n MakeSet operations are the first n operations.

How to Implement the Operations?

- Option 1 : Use linked lists.
- For every set, there is a linked list.
- The representative of a set is the head of the list.
- Every element also stores a pointer to the representative.
- There is a tail pointer indicating where to append.

Example

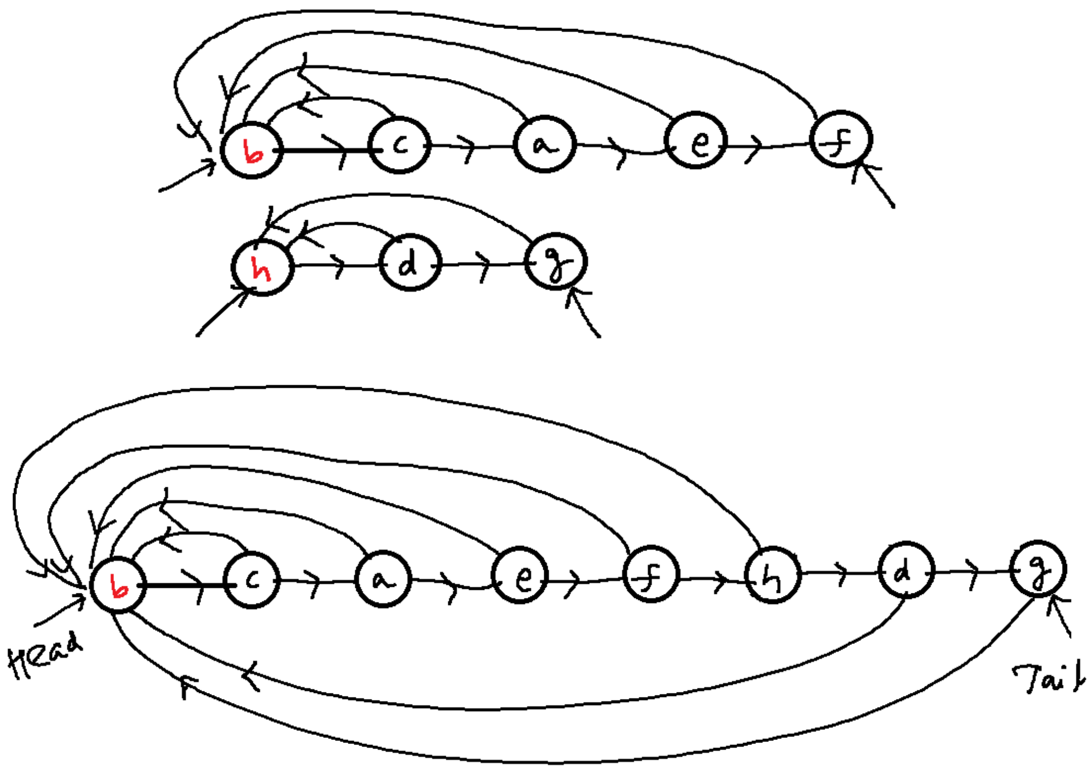


Operations

- MakeSet(x): Create a new linked list.
- FindSet(x) : Can be answered via the direct pointer

- Union(x, y) : Can append the list of x to the list of y.
- But have to update the pointer for each element in the list of x.

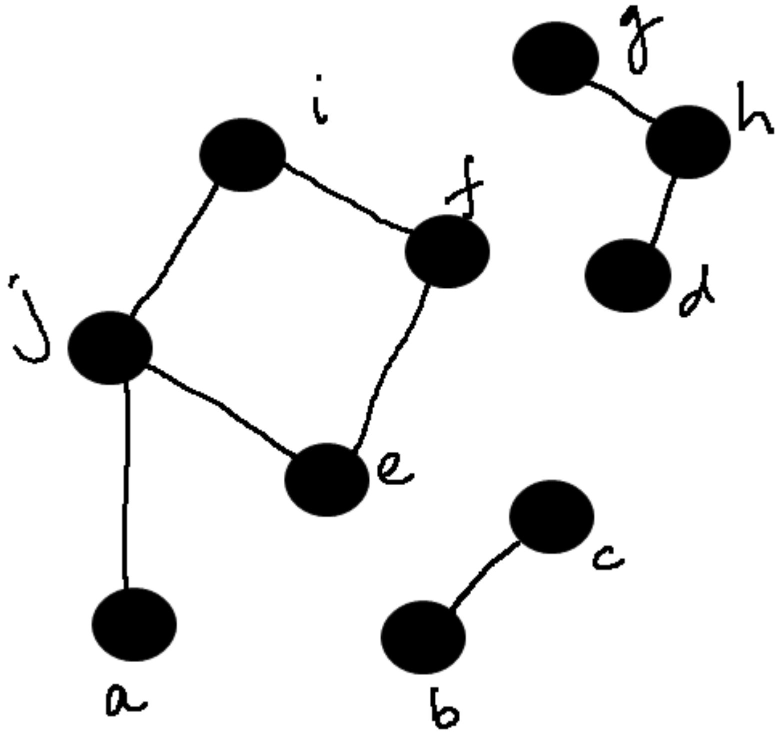
Adjacency List Example



Application to Connected Components

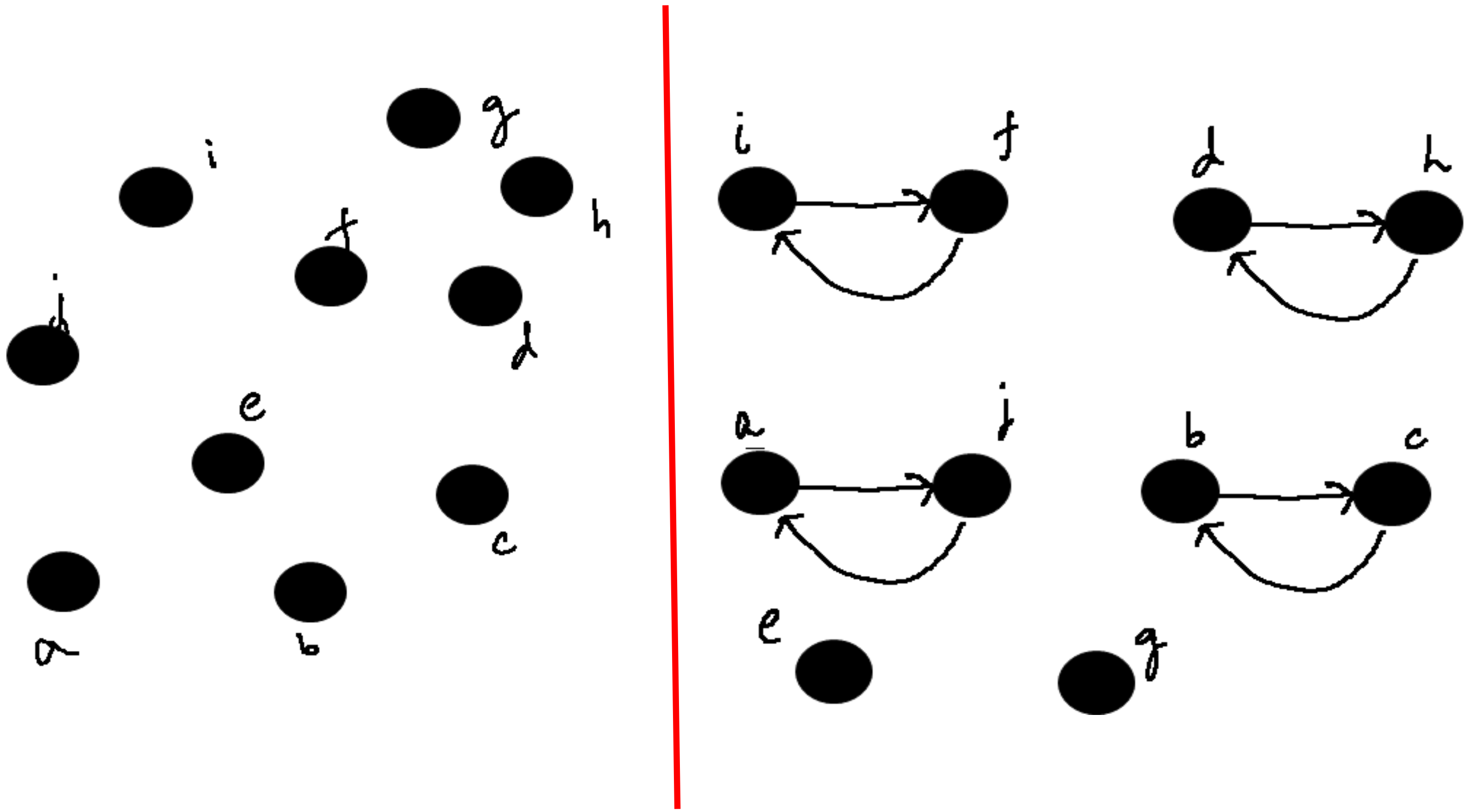
- Problem: Given an undirected graph G = (V, E), partition V into disjoint sets V_1, V_2, …, V_k, so that two vertices u and v are in the same partition if and only if there is a path between u and v.
- Several ways to solve this problem
  - This may not be the best way!
- Example follows.

Example

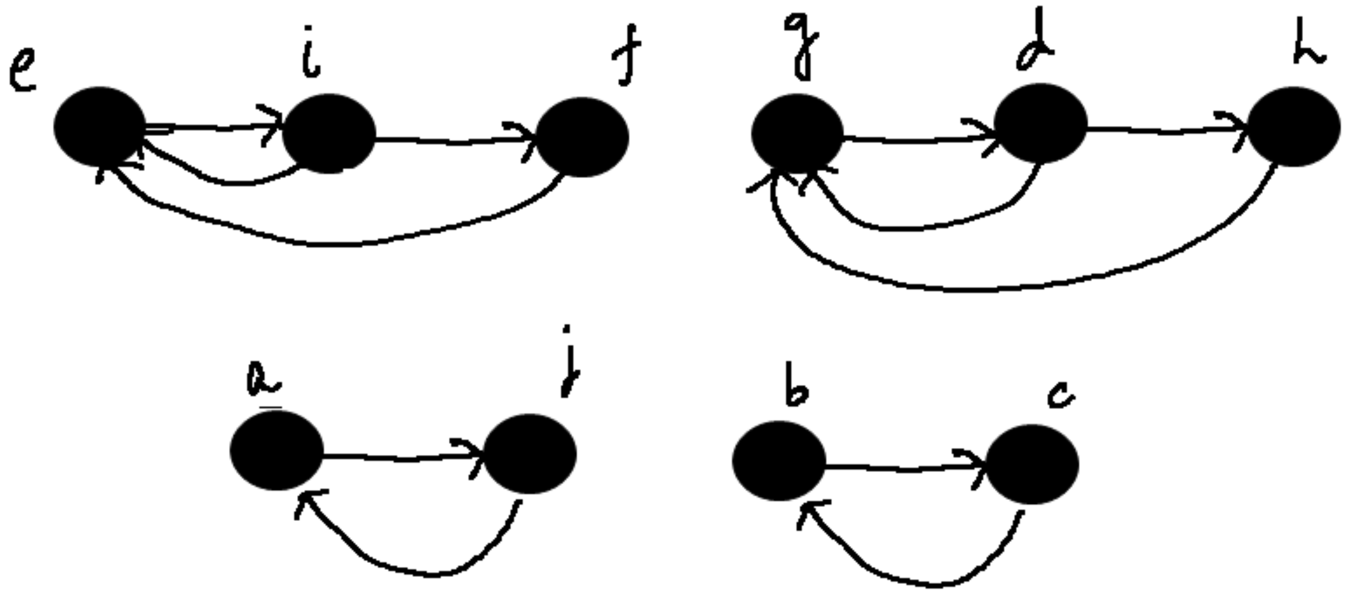


- Algorithm:
  - For each vertex v
    - MakeSet(v)
  - For each edge vw
    - Union(v,w)

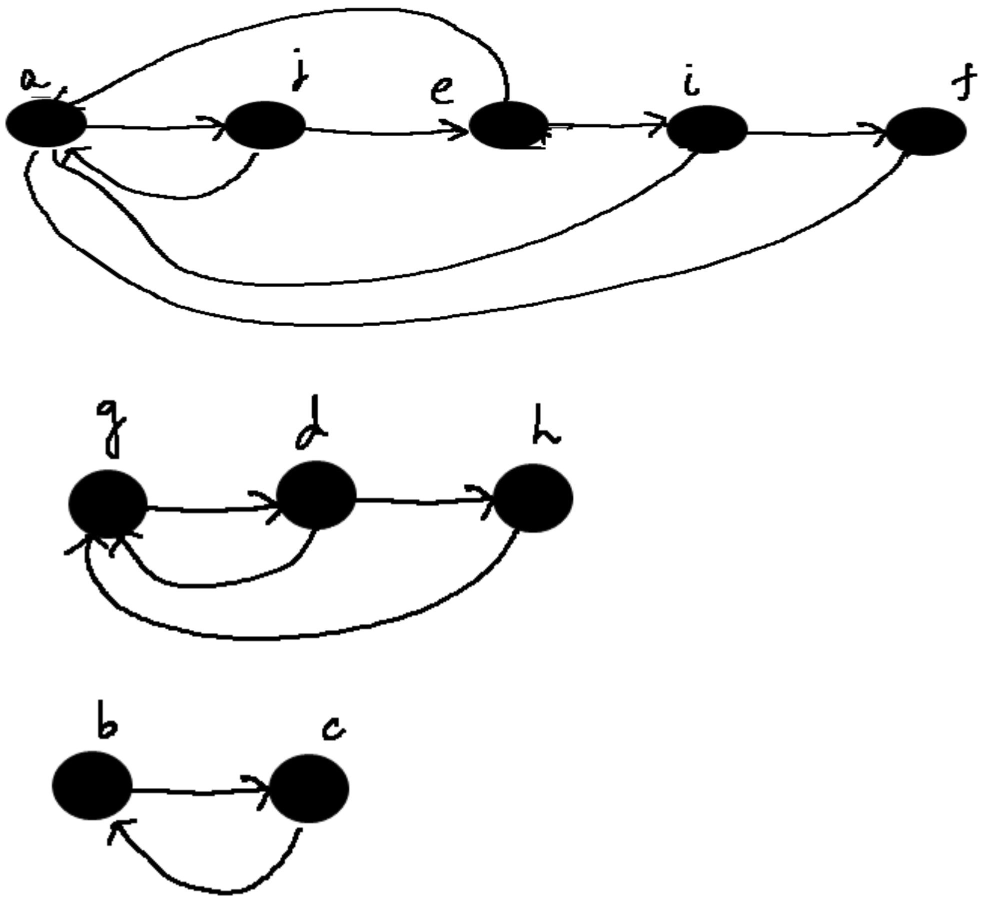
Example



Example



Example



Operations

- How difficult is it to append the lists?
- Claim: There exists a sequence of m operations on n objects so that the total time required for the entire sequence of operations is $O(n^2)$.
- After the first n MakeSet operations, call Union(x1,x2), Union(x2, x3), Union(x3, x4), …, Union(x_{n-1}, x_n).
- The kth union call takes time proportional to k.
- Total time is therefore $O(n^2)$.

- The average time per operation is also O(n).

Application to Kruskal's Algorithm

- An average time of O(n) is not helpful for Kruskal's algorithm.
- We have several Union calls and several FindSet calls.

Better Solution

- Most of the time spent is in the Union operation.
- Can we modify the operation slightly?
- Intuitively, it is easier to append a smaller list to a larger list.
  - Requires fewer updates.
  - Will the overall time decrease?
- We will show that indeed it does.

The Weighted Union Heuristic

- Maintain the length of each list. Corresponds to the size of the set.
- To perform Union(x, y):
  - Append the list of x to the list of y if len(x) < len(y)
  - Append the list of y to the list of x otherwise.

- A single Union operation can still take lot of time.
  - Union of two large lists, say of size n/10 each.
- But, a sequence of operations may be not so expensive.
  - Hopefully.

Analysis

- How many times can an element change its representative?
- Consider any element x.
- If in an Union operation, the representative of x changes, then x is in the smaller list.
  - Why?
- The first time this happens, the resulting list has at least 2 elements.
- Next time, the resulting list has at least 4 elements.

Analysis

- In general, if the representative of x changes k times, then the resulting list has size at least 2^k.
- The largest set can have a size of n.
- Therefore, the representative of x cannot change more than log n times, over all the Union operations.
- This applies to every element.
- Therefore, over all Union operations, the total time spent is O(n log n).

Analysis

- Now, consider a sequence of m operations.
- MakeSet and Find are $O(1)$ time operations.
- Therefore, the total time is $O(m + n\log n)$.
- The average time per operation is $O(\log n)$.

Application to Kruskal’s Algorithm

- How does the above apply?

Application to Kruskal's Algorithm

- Do n MakeSet operations indicating that each vertex is in its own tree/set.
- To check if e = uv creates a cycle, check if FindSet(u) = FindSet(v).
- If not, add e to the current tree. Perform Union(u, v) to merge the trees of u and v.

- There are at most m FindSet operations.
- Overall time is therefore bound by $O(m+n\log n)$.

MST – Another Approach

- The previous approach has to check for cycles every iteration.
- Another approach that has a smaller runtime even with basic data structures.
- Largely simplifies the solution.

MST – Another Approach

- The current approach is characterized by having a single tree T at any time.
- In each iteration, T is extended by adding one vertex v not in T and one edge from v to some vertex in T.
- Starting from a tree of one node, this process is repeated n-1 times.

MST – Another Approach

- Two questions:
  - How to pick the new vertex v?
  - How to pick the edge to be added from v to some other vertex in T?

MST – Another Approach

- The answers are provided by the following claims.

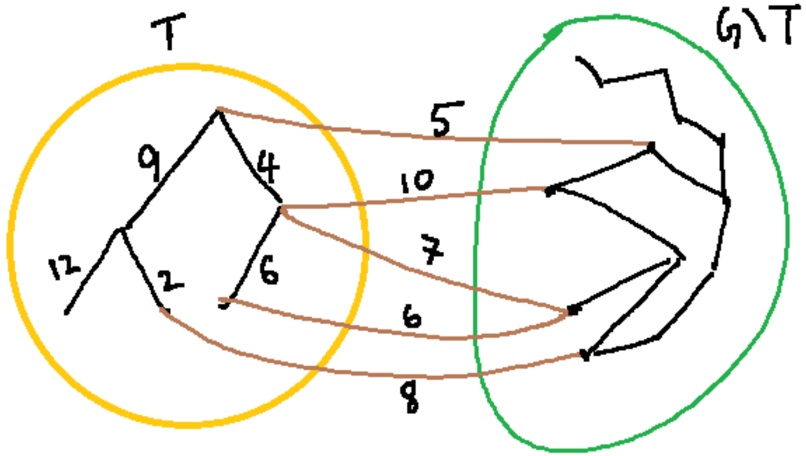
- Claim 1: Let G = (V, E, W) be a weighted undirected graph. Let v be any vertex in G. Let vw be the edge of smallest weight amongst all edges with one endpoint as v. Then vw is always contained in any MST of G.

MST – Another Approach

- Claim 1 can be shown in the following way.
- For each vertex v in G, there must be at least one edge in any MST.
- Considering the edge of the smallest weight is useful as it can decrease the cost of the spanning tree.

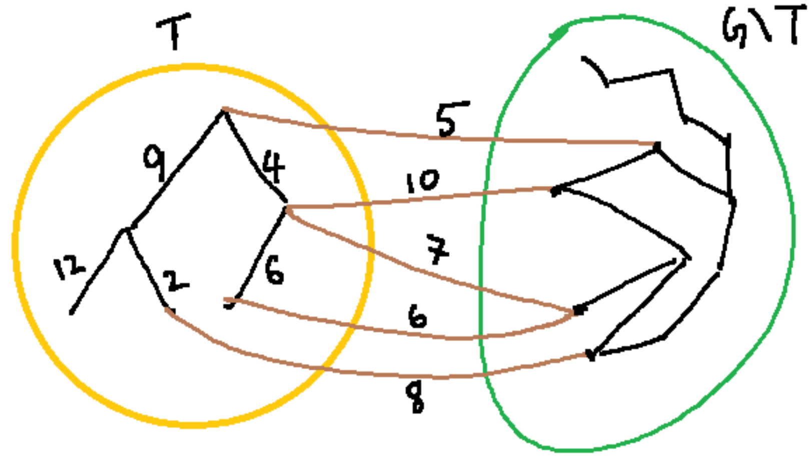
Generalizing Claim 1

- Let T be a subtree of some MST of an undirected weighted graph G.
- Consider edges uv in G such that u is in T and v is not in T.
- Of all such edges, let e = xy be the edge with the smallest weight.
- Then T U {e} is also a subtree of some MST of G.



Generalizing Claim 1

- Claim 2 allows us to expand a given sub-MST T.
- We can use Claim 2 to expand the current tree T.
- How to Start ?



Towards an Algorithm

- Let v be any vertex in the graph G. Pick v as the starting vertex to be added to T.
- T now contains one vertex and no edges.
- T is a subtree of some MST of G.
- Now, apply Claim 2 and extend T.

Towards an Algorithm

```
Algorithm MST(G, v)
Begin
    Add v to T;
    While T has less than n – 1 edges do
      w = vertex s.t. vw has the smallest weight
      amongst edges with one endpoint in T and
      another not in T.
      Add vw to T.
    End
End
```

Towards an Algorithm

- How to find w in the algorithm?
- Need to maintain the weight of edges that satisfy the criteria.
- A better approach:
  - Associate a key to every vertex
  - key[v] is the smallest weight of edges with v as one endpoint and another in the current tree T.
  - key[v] changes only when some vertex is added to T.
  - Vertex with the smallest key[v] is the one to be added to T.

Towards an Algorithm

- Suggests that key[v] need to be updated only when a new vertex is added to T.
- Further, not all key[v] may change in every iteration.
  - Only the neighbors of the vertex added to T.
  - Similar to Dijkstra's algorithm.

Towards an Algorithm

- Therefore, can maintain a heap of vertices with their key[ ] values.
- Initially, key[v] = infinity for every vertex except the start vertex for which key value can be 0.
- Perform DeleteMin on the heap. Let v be the result.
- Update the key[ ] value for neighbors w of v as:
  - key[w] = min{key[w], W(vw)}

Algorithm using a Heap

```
Algorithm MST(G, u)
begin
    for each vertex v do key[v] = infty.
    key[u] = 0;
    Add all vertices to a heap H.
    While T has less than n-1 edges do
        v = deleteMin();
        Add v to T via uv s.t. u is in T
        For each neighbor w of v do
            if W(vw) > key[w] then DecreaseKey(w)
        end
    end
end
```

Algorithm using a Heap

- The algorithm is called as Prim's algorithm.
- Runtime easy to analyze;
  - Each vertex deleted once from the heap. Each DeleteMin() takes O(log n) time. So, this accounts for a time of O(nlog n).
  - Each edge may result in one call to DecreaseKey(). Over m edges, this accounts for a time of O(mlog n).
  - Total time = O((n+m)log n).

Thank You