Minimum Spanning Tree

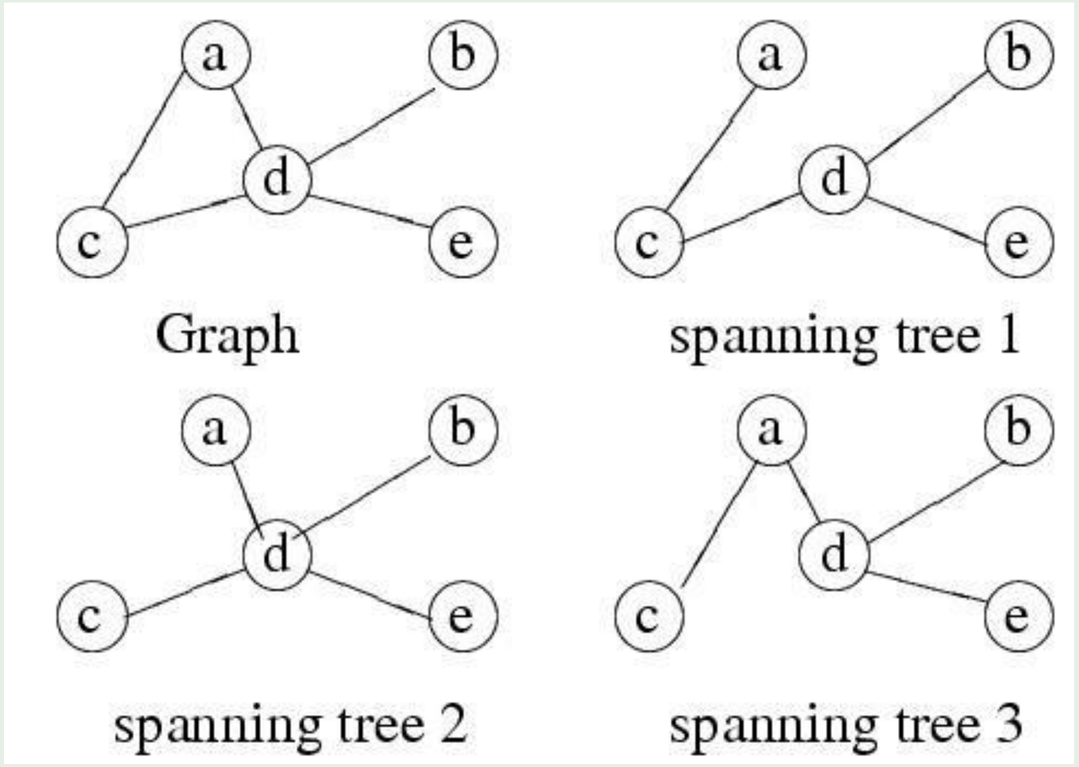
Kruskal’s Algorithm

Spanning Trees

Definition

A subgraph T of a undirected graph $G = (V, E)$ is a spanning tree of G if it is a tree and contains every vertex of G

Example

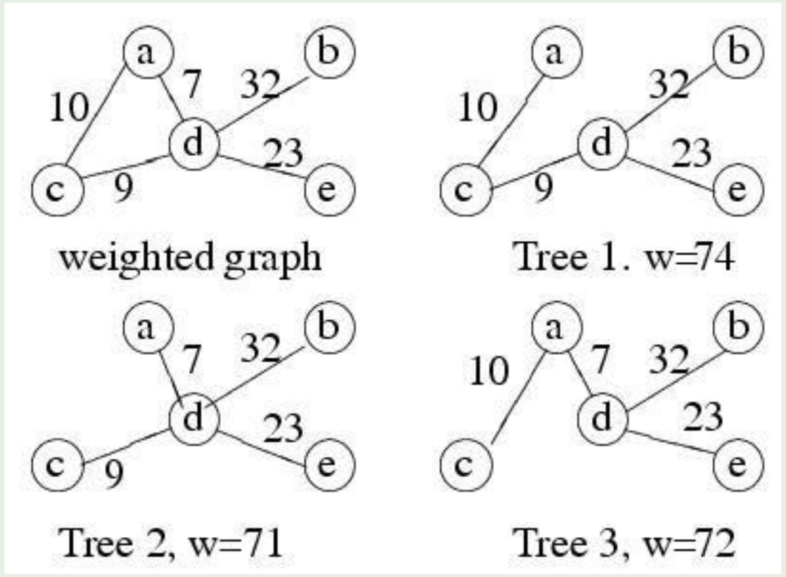


Weighted Graph

Definition

A weighted graph is a graph, in which each edge has a weight (some real number) Could denote length, time, strength, etc.

Example



Definition

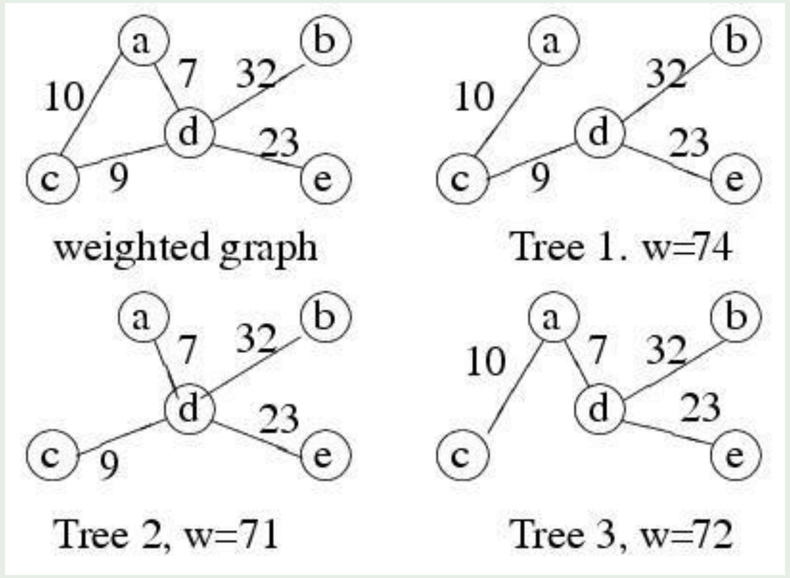
Weight of a graph: The sum of the weights of all edges

Minimum Spanning Trees

Definition

A Minimum spanning tree (MST) of an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees).

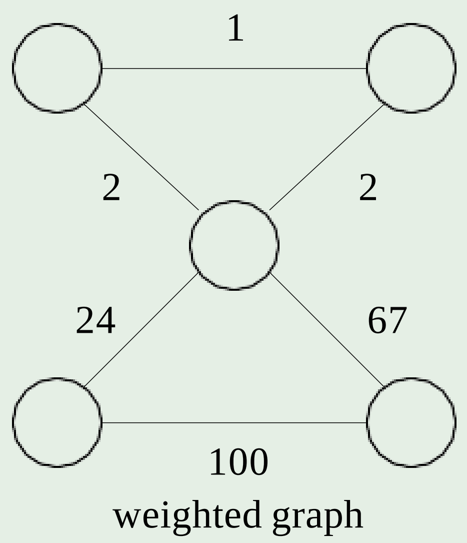
Example

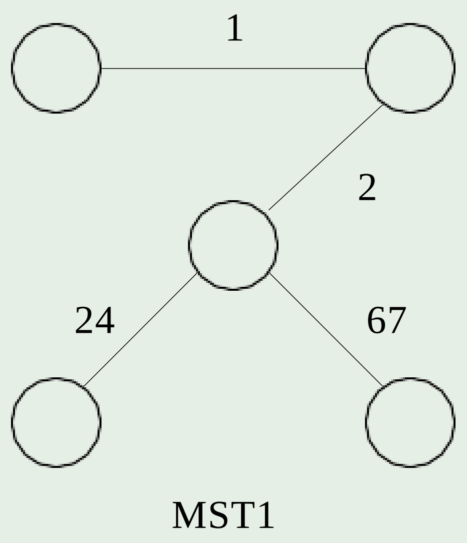


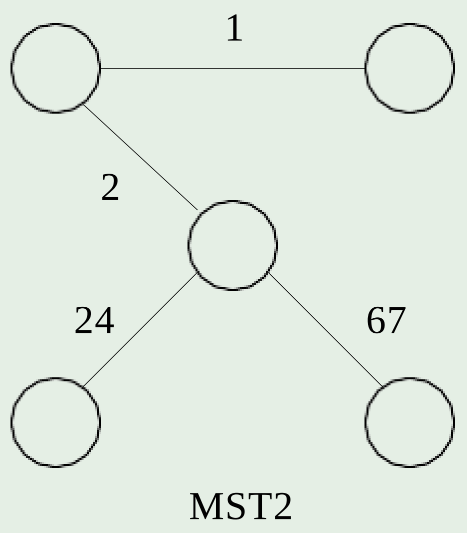
Minimum Spanning Trees

The minimum spanning tree may not be unique

Example







Note: if the weights of all the edges are distinct, MST is unique

Minimum Spanning Tree Problem

Definition (MST Problem)

Given a connected weighted undirected graph G, design an algorithm that outputs a minimum spanning tree (MST) of G.

Minimum Spanning Tree Problem

Input: A graph $G = (V, E)$ and a cost c_e for each edge e.

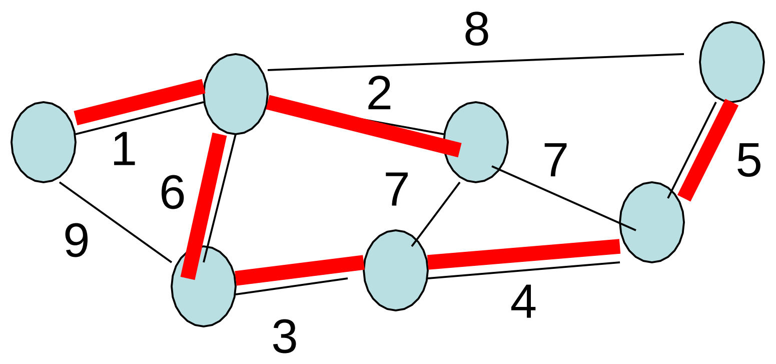
- Adjacency list representation of graph
- Edges can have negative weights

Output: A minimum cost tree $T \subseteq E$, that spans all vertices.

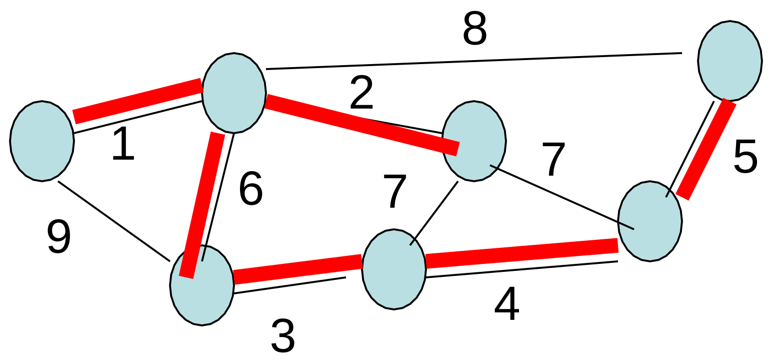
- T has no cycle
- The subgraph (V, T) is connected i.e. contains path between each pair of vertices

Approaches for growing A

1. A grows as *a single tree*
   Loop invariant: *Prior to each iteration, A is a* ***subtree*** *of some minimum spanning tree.*
   (Prim's algorithm)



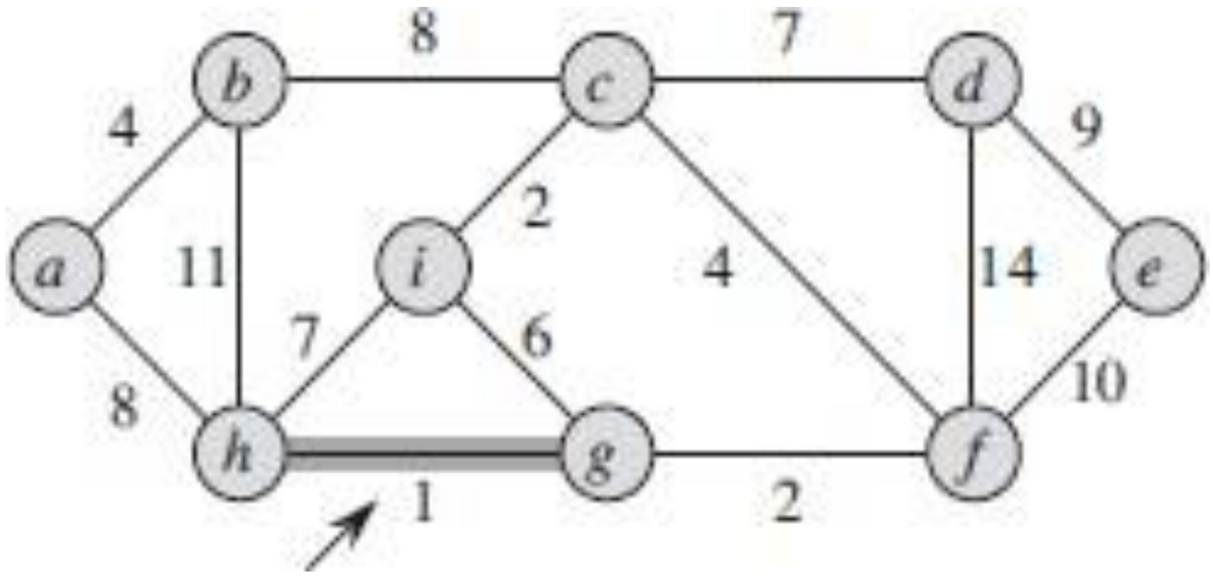
2. A can start as a *forest of trees*.
   Loop invariant: *Prior to each iteration, A is a* ***subset*** *of some minimum spanning tree.*
   (Kruskal's algorithm)



Idea of Kruskal’s Algorithm

- Build the MST starting as a ***forest.***
- Initially, the trees of the forest are the vertexes of the graph (no edges)
- In each step add ***the edge with the smallest weight that does not create a cycle*** (we will prove that this is a safe edge)
- Continue until the forest is a single tree
- This is the minimum spanning tree

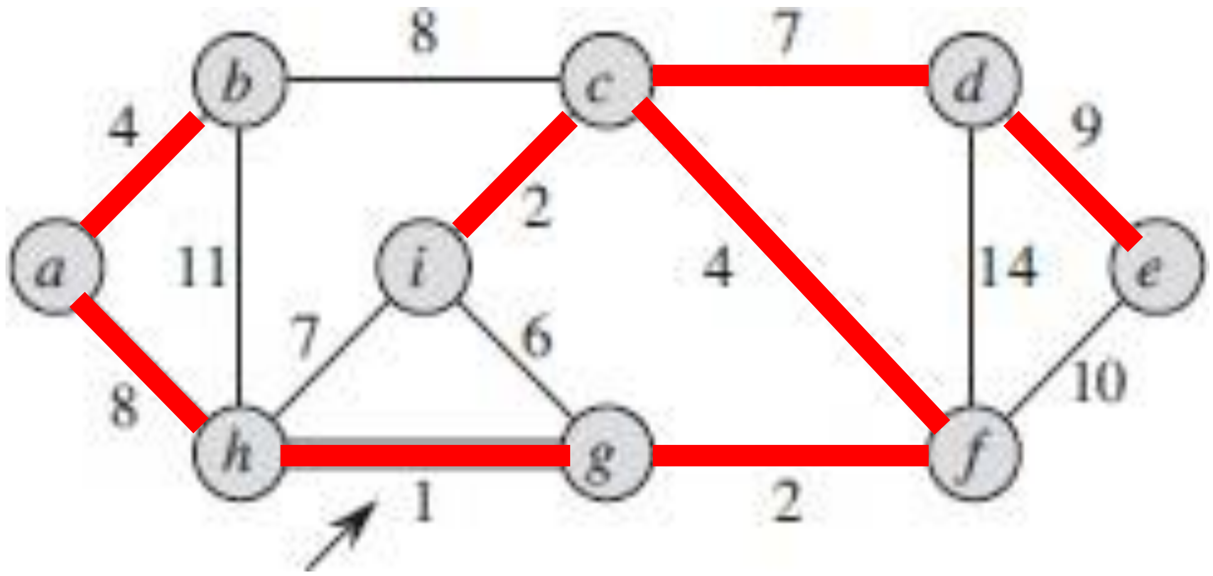
Example



Sorted edges:

- (g,h)
- (c,i)
- (f,g)
- (a,b)
- (c,f)
- (g,i)
- (c,d)
- (h,i)
- (a,h)
- (b,c)
- (d, e)
- (b,h)
- (d,f)

Example



Sorted edges:

- (g,h)
- (c,i)
- (f,g)
- (a,b)
- (c,f)
- (g,i)
- (c,d)
- (h,i)
- (a,h)
- (b,c)
- (d, e)
- (b,h)
- (d,f)

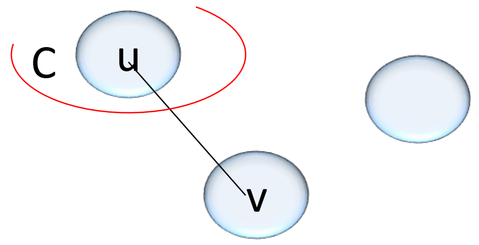
Kruskal's algorithm

```
MST-KRUSKAL(G=(V,E), w)

A={} (initialize as an empty forest)
for each vertex  v in V
     Create-tree(v) and add to forest A
sort the edges of G, E, into increasing order by
  weight w
for each edge (u,v) in E taken in increasing order
  by weight
      if (TreeOf(u)<>TreeOf(v))
             Add (u,v) to A
            (The two trees are united now into a new
             tree which replaces them in the forest)
return A
```

Correctness of Kruskal's algorithm

- We need to prove that the edge added in each step (the edge with the smallest weight that does not create a cycle) is a ***safe edge***
- **Lemma 1:** Let G(V, E) be a connected, weighted, undirected graph. Let A be a subset of E that is included in some MST of G. Also let (S, V\S) be any cut of G that respects A and (u, v) be a light edge crossing the cut (S, V\S) then (u, v) is a safe edge.
- **Corollary 1:** Let G (V, E) be a connected, undirected graph. Let A be a subset of E that is included in some minimum spanning tree for G, and let $C(V_C, E_C)$ be a connected component (tree) in the forest G_A (V, A). If (u,v) is a light edge connecting C to some other component in G_A, then (u,v) is safe for A
- **Proof:** The cut $(V_C, V\backslash V_C)$ respects A, and (u,v) is a light edge for this cut. Therefore, (u,v) is safe for A



Kruskal’s algo - Implementation

- Questions:
  - How to represent the forest of trees?
  - How to test efficiently whether u,v are in the same tree or not?
  - How to replace two trees of the forest by their union ?
- Solution: use an efficient implementation of Disjoint-sets
- We have a collection of disjoint sets that supports following 3 operations:
  - Make-Set(u)
  - Find-Set(u)
  - Union(u,v)

Kruskal’s algorithm

```
KRUSKAL (G=(V,E),w)
A = 0;
for each vertex v in V
        MAKE-SET(v)
sort the edges E of G  into nondecreasing order by
   weight w
for each (u,v) // taken from the sorted list
        if FIND-SET(u)<>FIND-SET(v)
                A  = A + { (u,v)}
                UNION (u,v)
return A
```

Kruskal’s algorithm - Analysis

```
KRUSKAL (G=(V,E),w)
A = 0;
for each vertex v in V
      MAKE-SET(v)
sort the edges E of G  into nondecreasing order by
  weight w
for each (u,v) // taken from the sorted list
      if FIND-SET(u)<>FIND-SET(v)
            A  = A + {(u,v)}
            UNION (u,v)
return A
```

A number of |V| calls of MAKE-SET

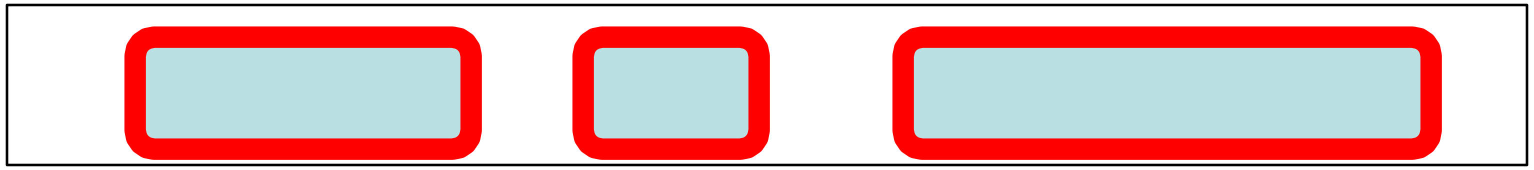
Sorting can be done in |E| * log (|E|)

O(|E|) calls of FIND-SET and O(|V|) UNION

The performace of Kruskal’s algorithm is given by the performance of the Union-Find operations !

Disjoint sets

- Also known as “union-find.”
- Maintain collection S ={S1;…; Sk} of disjoint dynamic (changing over time) sets.
- Each set is identified by a *representative*, which is some member of the set. Doesn’t matter which member is the representative, as long as if we ask for the representative twice without modifying the set, we get the same answer both times.



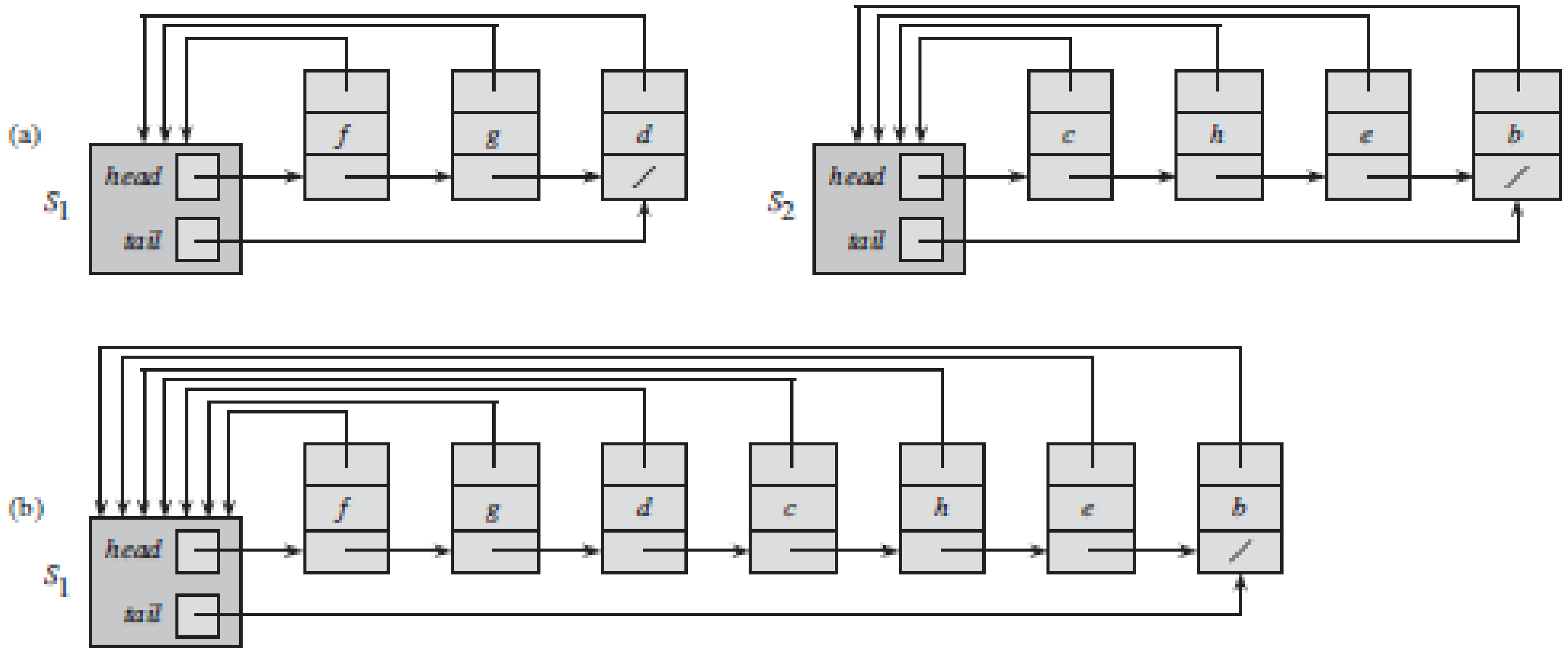
Operations

- MAKE-SET(x): make a new set Si ={x}, and add Si to S.
- UNION(x; y) : makes a new set from the union of set members of Sx and Sy
  - Representative of new set is any member of Sx or Sy, often the representative of one of Sx and Sy.
  - The Union operation destroys Sx and Sy (since sets must be disjoint).
- FIND-SET(x): return representative of set containing x.

Disjoint sets - Implementation with linked lists

- Each set is a singly linked list, represented by an object with attributes
  - *head*: the first element in the list, assumed to be the set's representative, and
  - *tail*: the last element in the list.
- Objects may appear within the list in any order.
- Each object in the list has attributes for
  - the set member,
  - pointer to the set object, and
  - next.

Disjoint sets - Implementation with linked lists



[CLRS – fig 21.2]

Implementing disjoint-sets operations

- MAKE-SET(x): create a new linked list whose only object is x.
- FIND-SET(x): follow the pointer from x back to its set object and then return the member in the object that *head* points to.
- UNION(x; y) : we can append y’s list onto the end of x’s list. The representative of x’s list becomes the representative of the resulting set. We use the *tail* pointer for x’s list to quickly find where to append y’s list. Drawback: we must update the pointer to the set object for each object originally on y’s list, which takes time linear in the length of y’s list.

Weighted-Union on lists

- To improve the situation of UNION, we will ***always append the shorter list at the end of the longer list*** - ***weighted-union heuristic***
- Implementation: each list also includes the length of the list (which we can easily maintain) and we always append the shorter list onto the longer
- **Analysis:**
  - A single UNION operation can take O(n) time if both sets have n members.
  - But *how much time does a sequence of m MAKE-SET, UNION, and FIND-SET operations take, n of which are MAKE-SET operations ?*

Weighted Union Analysis

- ***Theorem***
- With weighted union, a sequence of m operations on n elements takes O(m + n lg n) time.
- ***Sketch of proof***
- Each MAKESET and FIND-SET operation takes O(1) time, and there are O(m) of them
- For UNION: we count how many times can each object's representative pointer be updated? It must be in the smaller set each time.
- Since the largest set has at most n members, The total time spent updating object pointers over all n UNION operations is O.(n lg n).(=> see proof on next slide)
- Elements : e_1, e_2, e_3, ,,,,, e_n

Weighted Union Analysis - Detail

- For UNION: we count how many times can each object's representative pointer be updated?
- Consider a particular object x. We know that each time x's pointer was updated, x must have started in the smaller set.
  - The first time x's pointer was updated, the resulting set had at least 2 members.
  - The next time x's pointer was updated, the resulting set had at least 4 members.
  - The third time x's pointer was updated, the resulting set had at least 8 members
  - After x's pointer has been updated k times, the resulting set must have at least 2^k members
  - Finally, the set will have n members. It's object representative pointer was updated log n times

  - Thus the total time spent updating object pointers over all UNION operations is O(.n * lg n)

Disjoint-sets with linked lists and weighted-union heuristic

- Upper bounds for every single operation:
  - MAKE-SET(x): O(1)
  - FIND-SET(x): O(1)
  - UNION(x; y) : O(n)

- But: **a sequence** of **n** UNION operations in order to build the maximum possible set of n elements takes O(n lg n) time.
- **Amortized analysis**: UNION is an **average** of O(log n) in this case

Amortized analysis

- In an ***amortized analysis***, we average the time required to perform a sequence of data-structure operations over all the operations performed.
- With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.

Kruskal’s algorithm - Analysis

$|V| <= |E| <= |V|^2$

```
KRUSKAL (G=(V,E),w)
A = 0;
for each vertex v in V
        MAKE-SET(v)
sort the edges E of G  into nondecreasing order by
   weight w
for each (u,v); // taken from the sorted list
        if FIND-SET(u)<>FIND-SET(v)
                A  = A + {(u,v)}
                UNION (u,v)
return A
```

A number of |V| calls of MAKE-SET

O(1)

Sorting can be done in |E| * log (|E|)

O(|E|) calls of FIND-SET and O(|V|) calls of UNION

O(1)

O(V * log (V))

Union-Find implemented with linked lists and weighted union

Kruskal’s algorithm - analysis

- If using Union-Find implemented with linked lists and weighted union, the complexity of Kruskal’s algorithm is O(E*log E)
- Actually, O(E*log E)=O(E*log V) (because $E<=V^2$)
- The complexity is given by the sorting of edges
- There are also better implementations of Union-Find (with Forests), but the implementation with linked lists and weighted union is sufficient for Kruskal’s algorithm
- An implementation of Union-Find with arrays would be not OK for Kruskal’s algorithm, since it exceeds the complexity of the sorting!

Slide Credits

- COMP 3711H Design and Analysis of AlgorithmsFall 2016
- https://algorithms.tutorialhorizon.com/prims-minimum-spanning-tree-mst-using-adjacency-list-and-min-heap/
- MIT opencourseware