Greedy Approach
Minimum Spanning Tree Problem

R. Kanchana
Dept of CSE
SSNCE

15th Dec 2014

Greedy - A simple example

- Problem: Pick k numbers out of n numbers such that the sum of these k numbers is the largest.
- Algorithm:

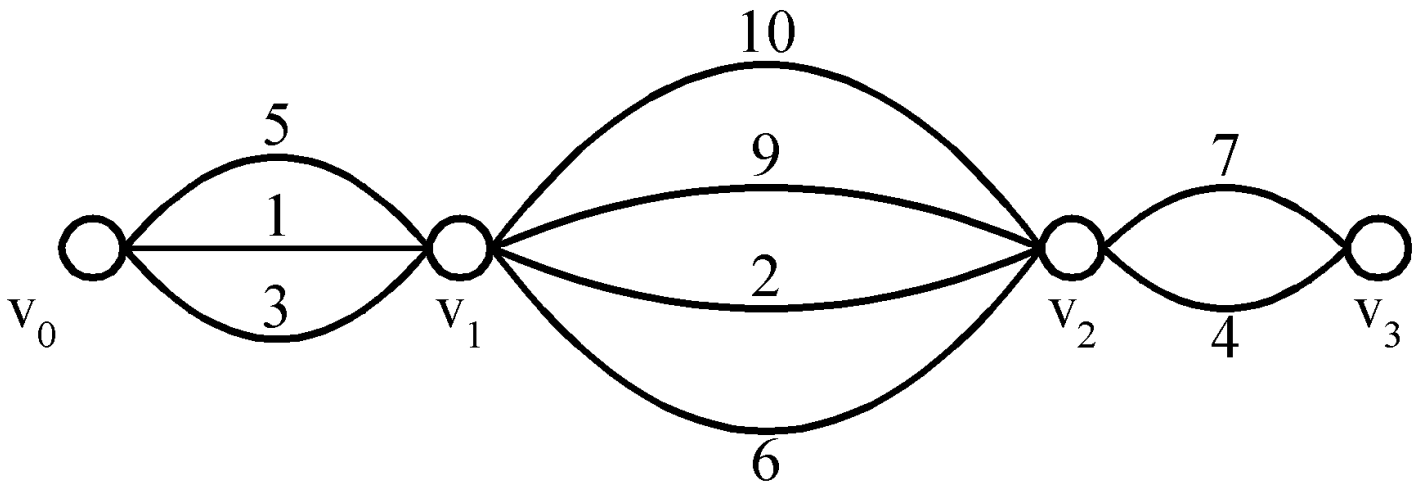
```
FOR i = 1 to k
        pick out the largest number and
        delete this number from the input.
ENDFOR
```

The greedy method

- Suppose that a problem can be solved by a sequence of decisions.
- Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.
- Only a few optimization problems can be solved by the greedy method. (Eg. Scrabble, MST)
- Chess – Greedy does not help

Shortest path Problem

- Problem: Find a shortest path from v_0 to v_3.
- The greedy method can solve this problem.
- The shortest path: 1 + 2 + 4 = 7.



Greedy Approach

X = { } (edges picked so far)

repeat until |X| = |V| -1

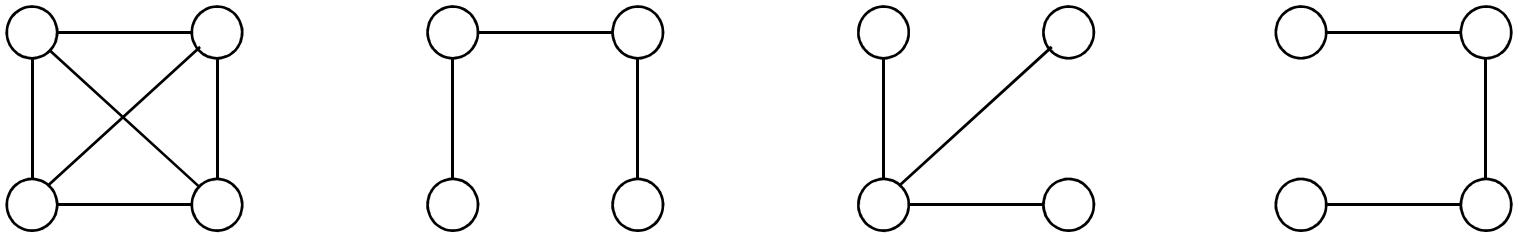
pick a set S from V for which X has no edges between S and V - S

let e ϵ E be the minimum-weight edge between S and V - S

X = X U {e}

Spanning Trees

- Let $G=(V, E)$ be at undirected connected graph.
- A subgraph $t=(V, E')$ of G is a *spanning tree* of G iff t is a tree.
- Example



Spanning trees

- Applications
  - To network a set of computers with minimum cost
  - Obtaining an independent set of circuit equations for an electric network

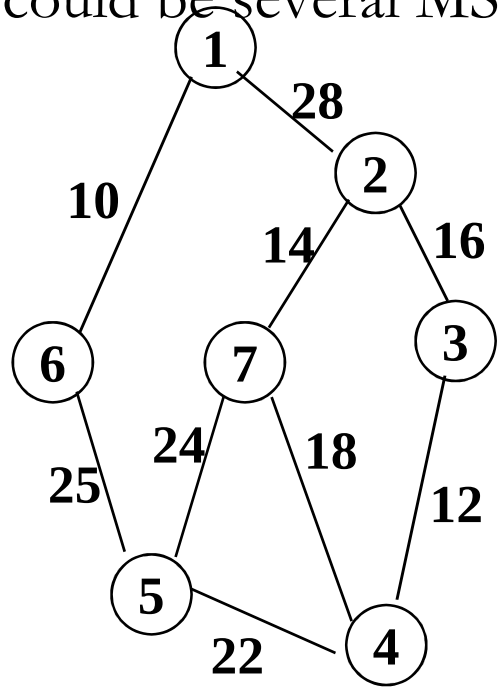
Minimum-cost Spanning Trees

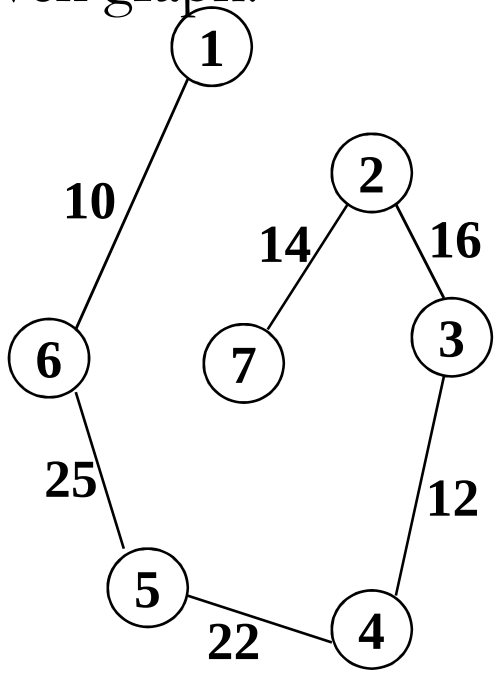
Input: An undirected graph G = (V,E); edge weights w_e.

Output: A tree T = (V,E'), with E' is a subset of E, that minimizes

weight(T) = $\sum w_e$ e ϵ E'

There could be several MSTs for a given graph!





Kruskal's MST Algorithm

- Start with an empty graph and then select edges from E according to the following rule.

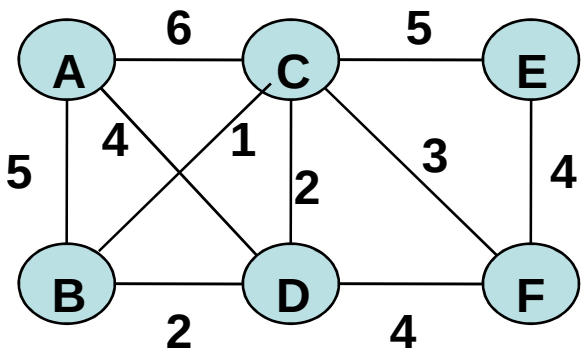
  Repeatedly add the next lightest edge that doesnot produce a cycle

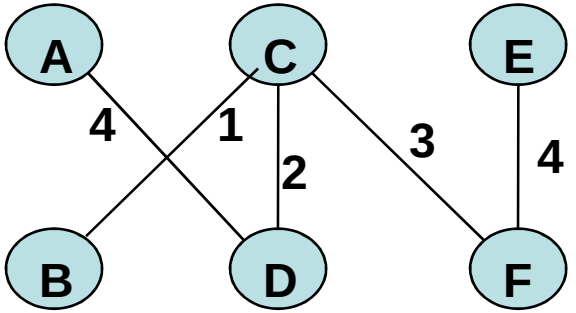
Kruskal's Algorithm

- Start with an empty graph and then attempt to add edges in increasing order of weight

BC; CD; BD; CF; DF; EF; AD; AB; CE; AC:

- The first two succeed, but the third, BD, would produce a cycle if added. So ignore it and move along.
- The final result is a tree with cost 14, the minimum possible.

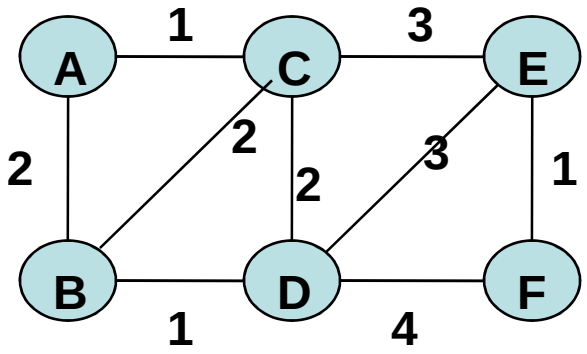




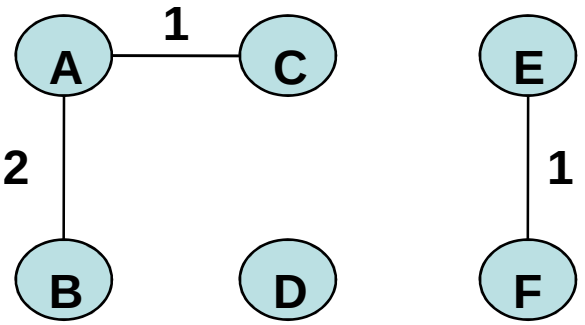
Kruskal's Algorithm – Cut Property

- The correctness of Kruskal's method follows from *cut property*
- A cut is any partition of the vertices into two groups, S and V-S.
- Suppose edges X are part of a minimum spanning tree of G = (V,E). Pick any subset of nodes S for which X does not cross between S and V – S.
- Let e be the lightest edge across this partition. Then X U{e} is part of some MST
- Cut property ensures that it is always safe to add the lightest edge across any cut (between a vertex in S and one in V-S), provided X has no edges across the cut)

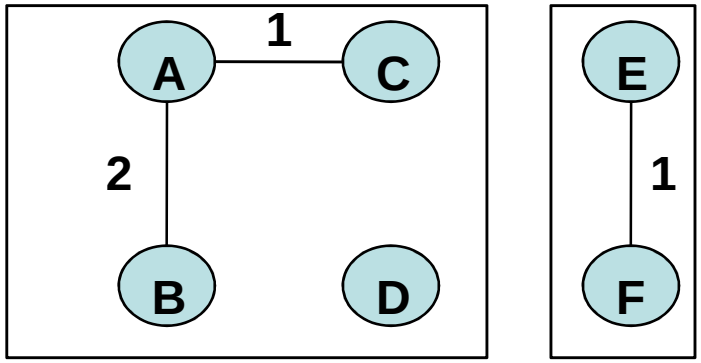
Kruskal's Algorithm – Cut Property



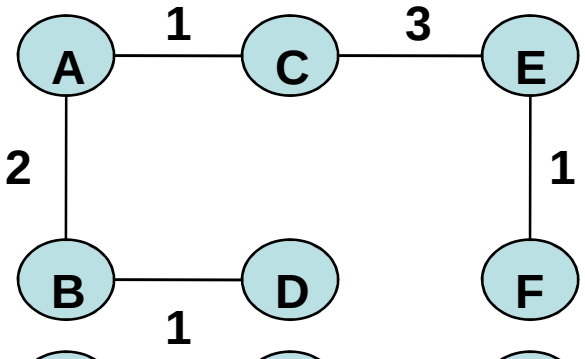
Edges X:



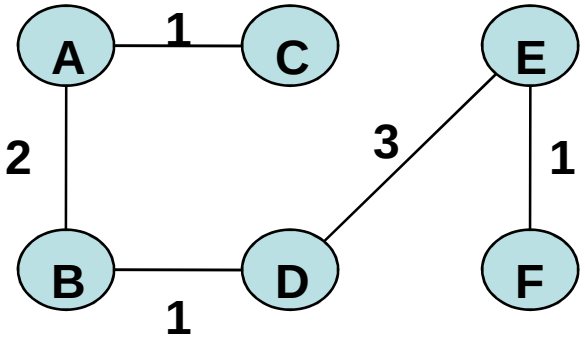
Cut



S V-S



MST T



MST T'

Kruskal's Algorithm - General

```
X = { };
while ((X has fewer than n-1 edges) && E <> { })
{
    choose an edge (v, w) from E of lowest cost
    delete (v, w) from E;
    if (v, w) dose not create a cycle in X
        add (v, w) to X
    else discard (v, w)
}
```

Kruskal's Algorithm

procedure kruskal(G,w)
Input: A connected undirected graph G = (V,E) with edge weights w_e
Output: A minimum spanning tree defined by the edges X
for all u ϵ V
    makeset(u)
X = { }; mincost = 0
Sort the edges E by weight
for all edges {u,v} ϵ E, in increasing order of weight
    if find(u) <> find(v)
        add edge {u,v}to X
        mincost = mincost + w_{uv}
        union(u, v)

Kruskal's Algorithm

- At each stage, the algorithm chooses an edge to add to its current partial solution.
- It tests each candidate edge *uv* to see whether the endpoints u and v lie in different components; otherwise the edge produces a cycle.
- Once an edge is chosen, the corresponding components need to be merged.
- What kind of data structure supports such operations?
- Can model the algorithm's state as a collection of *disjoint sets, each of which contains* the nodes of a particular component.
- Initially each node is in a component by itself

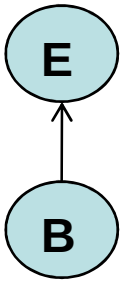
Disjoint Set

One way to store a set is as a *directed tree*.

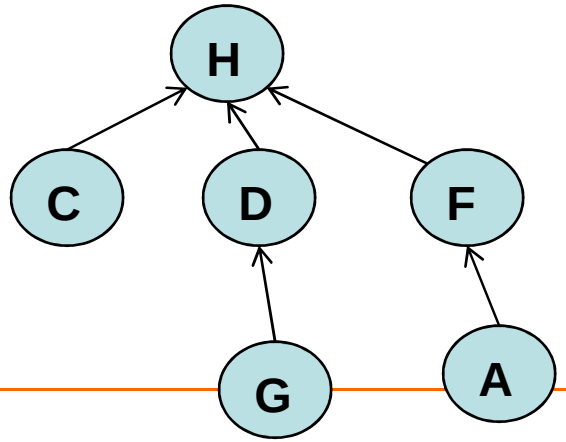
Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree.

This root element is distinguished from the other elements by the fact that its parent pointer is a self-loop.

Set {B,E}



set {A,C,D,F,G,H}



Disjoint Set Operations

makeset(x): create a singleton set containing just x.

```
procedure makeset(x)
x.parent = x
rank(x) = 0  // height of the subtree hanging from node x
```

To repeatedly test pairs of nodes to see if they belong to the same set.

find(x): to which set does x belong?

```
function find(x)
while x <> x.parent
        x = x.parent
return x
```

Disjoint Set Operations

whenever an edge is added, two components are merged

union(x, y): merge the sets containing x and y.

procedure union(x, y)

$root_x$ = find(x)

$root_y$ = find(y)

if $root_x = root_y$

return

if rank($root_x$) > rank($root_y$)

($root_y$).parent = $root_x$

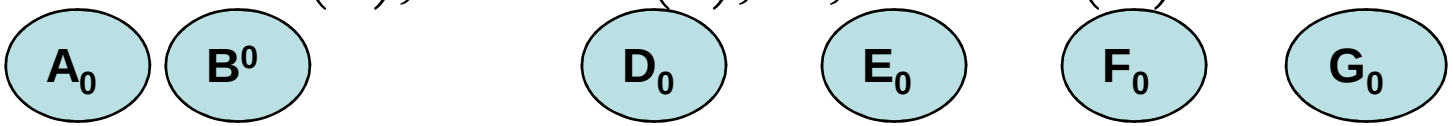
else ($root_x$).parent = $root_y$

if rank($root_x$) = rank($root_y$)

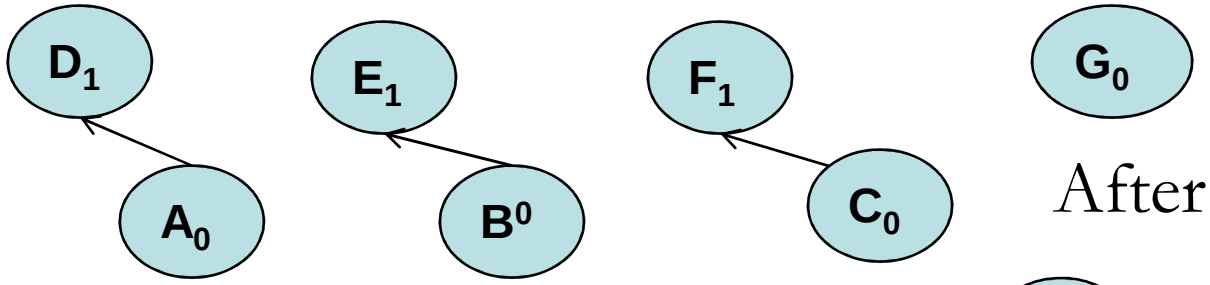
rank($root_y$) = rank($root_y$) + 1

Disjoint Set Operations

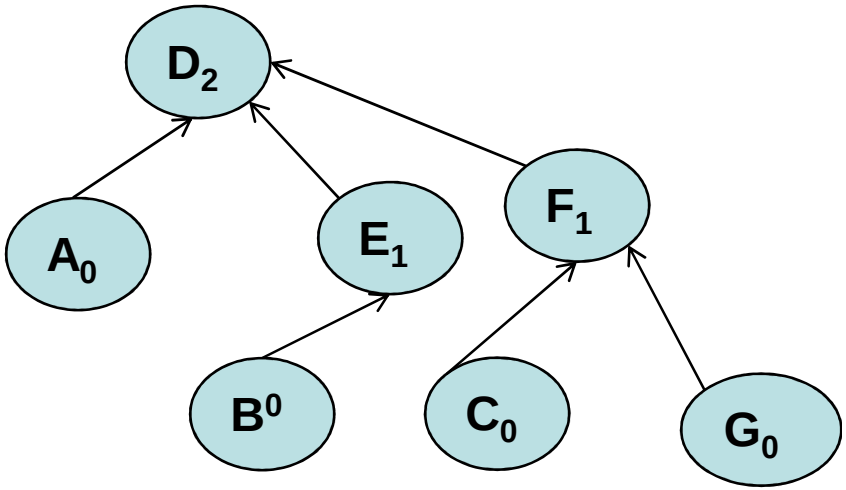
After makeset(A),makeset(B),…,makeset(G)



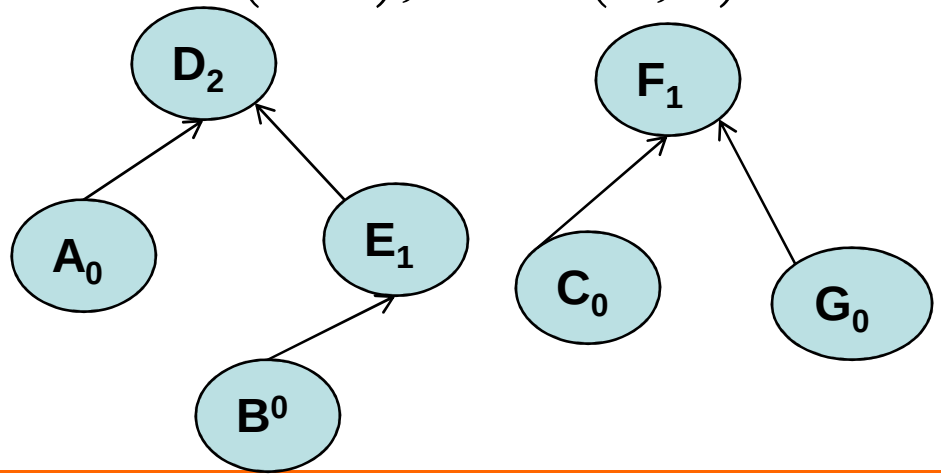
After union(A,D), union(B,E),union(C,F)



After union(B,G)



After union(C.G),union(E,A)



Kruskal's Algorithm

- Time Complexity
- It uses
  - |V| makeset
  - O(|E| log |V|) for sorting the edges (log |E|≈ log |V|)
  - O(|E| log |V|) for the union and find operations that dominate the rest of the algorithm. (2 * |E| find
  - |V| - 1 union operations)
- Hence O(|E| log |V|)

Prim's MST Algorithm

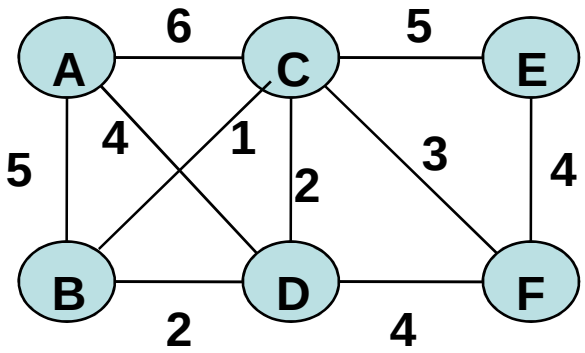
- Alternative to Kruskal's algorithm
- The intermediate set of edges X always forms a subtree, and S is chosen to be the set of this tree's vertices.
- On each iteration, the subtree defined by X *grows by one edge, namely, the lightest edge* between a vertex in S and a vertex outside S
- S grows to include the vertex v (not ϵ S) of smallest cost
- cost(v) = min w(u, v) u ϵ s
- It differs from Dijkstra's algorithm in the key values by which the priority queue is ordered.
- In Prim's algorithm, the value of a node is the weight of the lightest incoming edge from set S, whereas in Dijkstra's it is the length of an entire path to that node from the starting point

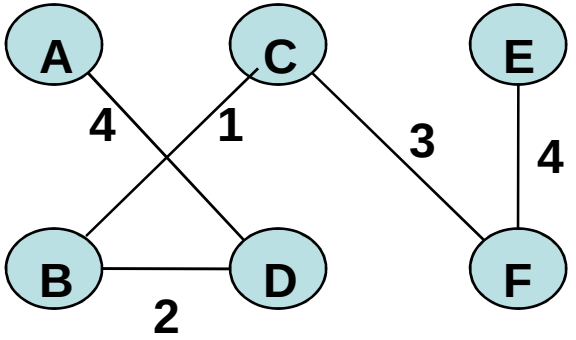
Prim's Algorithm

```
procedure prim( G, w)
Input: A connected undirected graph G = (V,E) with edge weights w_e
Output: A minimum spanning tree defined by the array prev
for all u ∈ V :
        cost(u) = ∞
        prev(u) = NIL
Pick any initial node u_0
cost(u_0) = 0
H = makequeue(V )     //priority queue, using cost-values as keys
while H is not empty
        v = deletemin(H)
        for each {v, z} ∈ H
                if cost(z) > w(v, z)
                        cost(z) = w(v, z)
                        prev(z) = v
                        decreasekey(H, z)
```

Prim's Algorithm

Trace





Set S	A	B	C	D	E	F
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		5/A	6/A	4/A	∞/nil	∞/nil
A,D		2/D	2/D		∞/nil	4/D
A,D,B			1/B		∞/nil	4/D
A,D,B,C					5/C	3/C
A,D,B,C,F					4/F	

Prim's Algorithm

Time complexity

- Since makequeue takes at most |V| insert operations, a total of |V| deletemin are required and |V|+ |E| insert/decreasekey operations.

- The time needed for these varies by implementation;

- For instance, a binary heap gives an overall running time of O((|V| + |E|) log |V|)