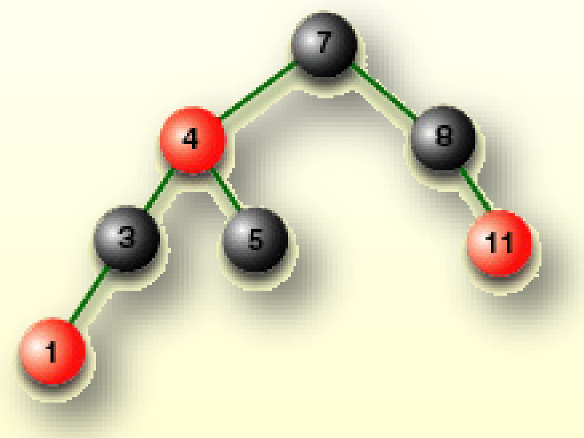
CS161:
Design and Analysis of Algorithms



Lecture 14
Leonidas Guibas

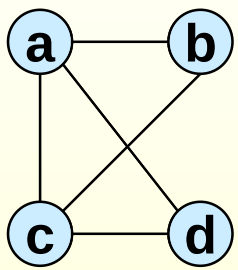
Outline

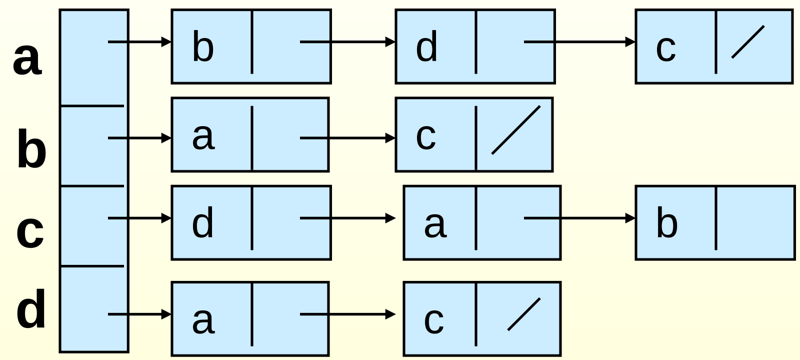
- Last lecture: Graph traversal – breadth and depth first search
- Today: Minimum spanning trees (MSTs)
  - Kruskal’s algorithm
  - Prim’s algorithm
  - Boruvka’s algorithm

Slides modified from
- *http://delab.csd.auth.gr/.../Lec9_MST.ppt*
- *http://www.cse.unr.edu/~bebis/.../MinimumSpanni...*

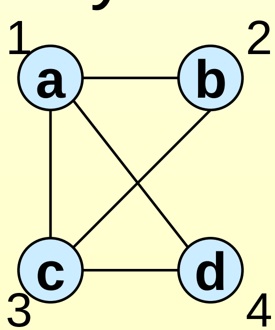
Representation of Graphs

- Two standard ways
  - Adjacency Lists





  - Adjacency Matrix



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Graph-Searching Algorithms

- Searching a graph:
  - Systematically follow the edges of a graph to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

Breadth-First Search (BFS)

- Expands the frontier between discovered and undiscovered vertices uniformly across the length of the frontier by using a queue.
  - A vertex is “discovered” the first time it is encountered during the search.
  - A vertex is “finished” if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.

Depth-First Search (DFS)

- Explore edges out of the most recently discovered vertex *v*.
- When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered (its *predecessor*).
- "Search as deep as possible first" – using a stack.
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Graph Search Algorithms

- BFS, DFS often used for their “by-products” – certain node annotations
- BFS provides shortest path distances to the source, and the BFS tree is a shortest path tree
- BFS selects a set of graph edges with useful properties

DFS Classification of Edges

- **Tree edge:** in the depth-first forest. Found by exploring (*u, v*).
- **Back edge:** (*u, v*), where *u* is a descendant of *v* (in the depth-first tree).
- **Forward edge:** (*u, v*), where *v* is a descendant of *u*, but not a tree edge.
- **Cross edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

Theorem:
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Topological Sort

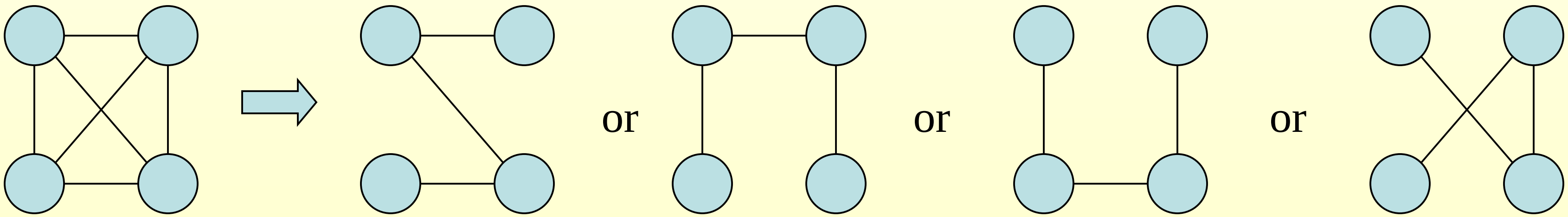
- *Topological sort* of a DAG (directed acyclic graph):
  - Linear ordering of all vertices in graph G such that vertex *u* comes before vertex *v* if edge $(u, v) \in G$
- Real-world example: getting dressed

Spanning Trees

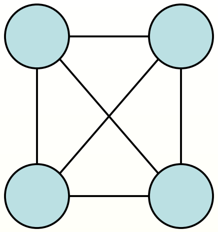
A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree.

A graph may have many spanning trees.

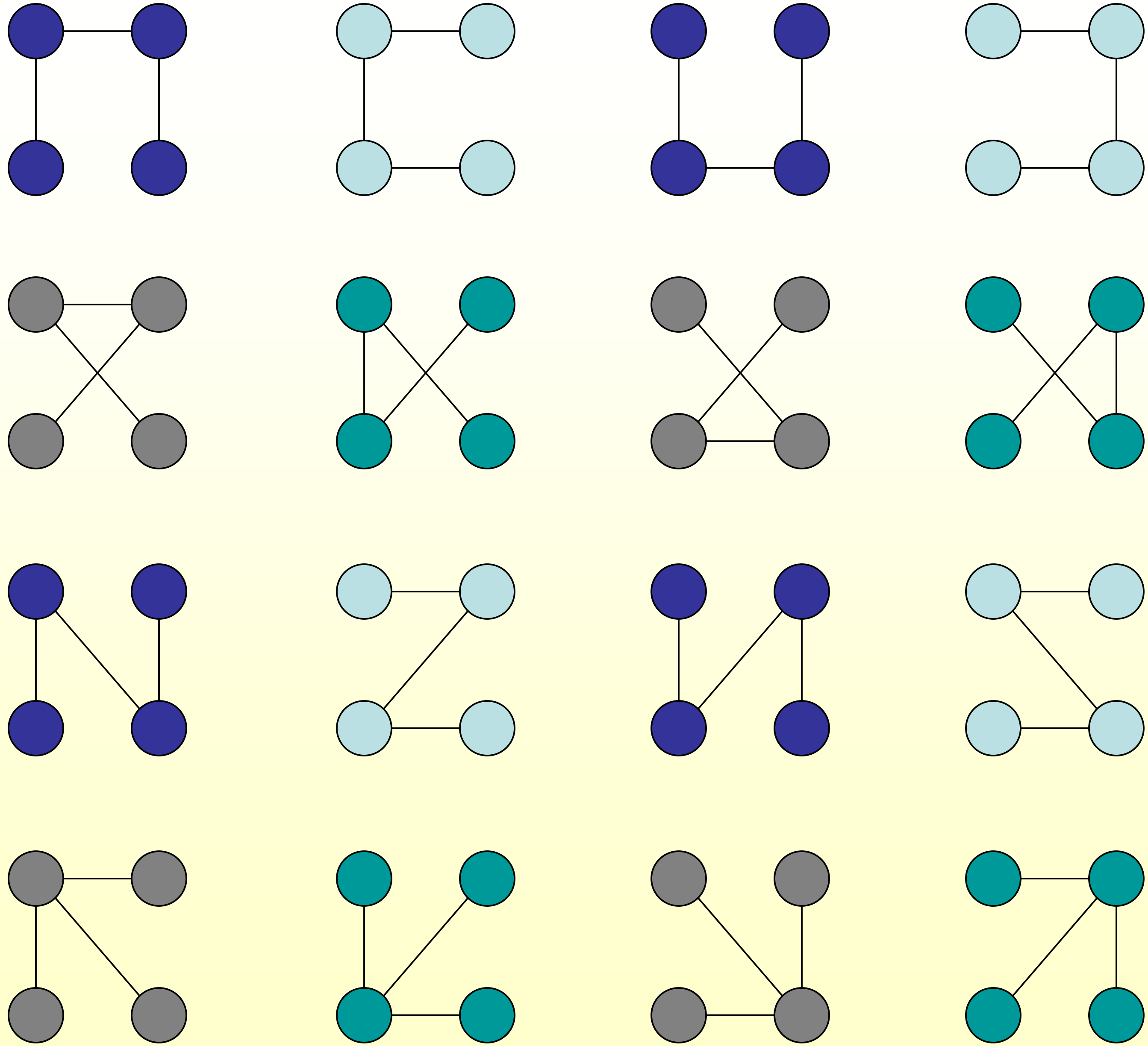
Graph A   Some Spanning Trees from Graph A



Complete Graph



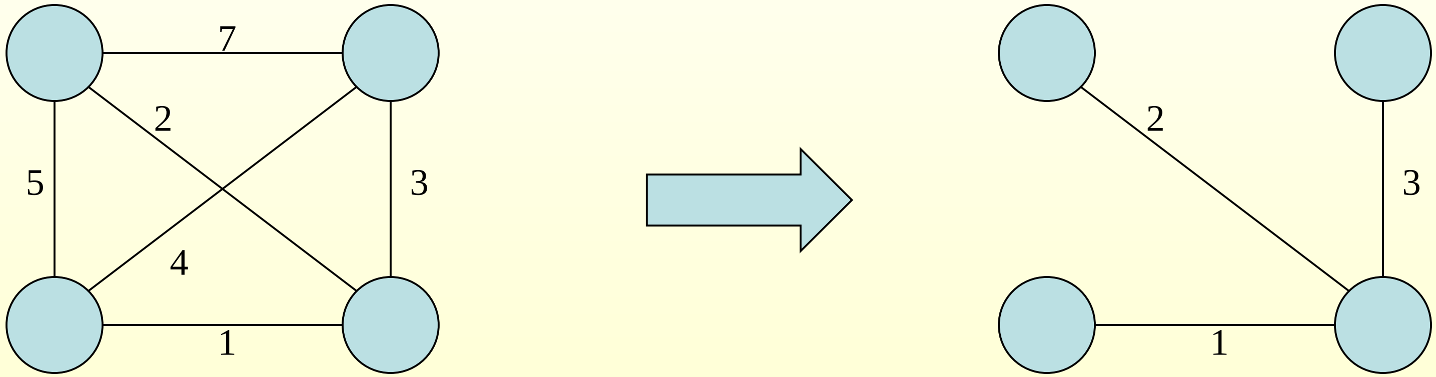
All 16 of its Spanning Trees



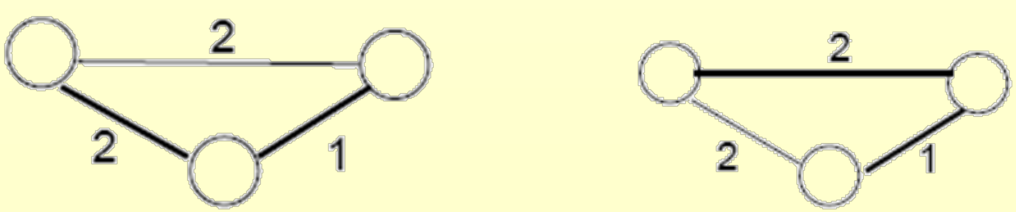
Minimum Spanning Trees

The Minimum Spanning Tree for a given graph is the Spanning Tree of minimum cost for that graph.

Complete Graph　　　　　　　　Minimum Spanning Tree

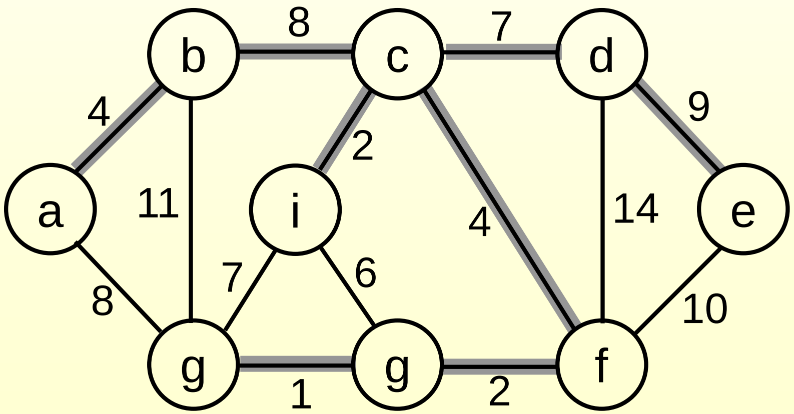


The Minimum Spanning Tree for a given graph is not unique.



Minimum Spanning Trees

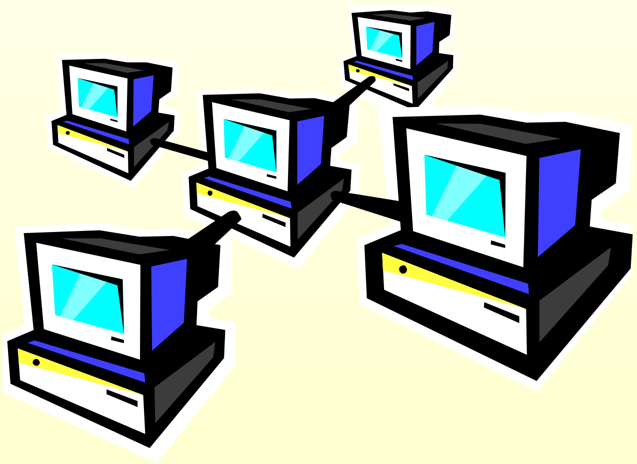
- Spanning Tree
  - A tree (i.e., connected, acyclic graph) which **contains all the vertices** of the graph
- Minimum Spanning Tree
  - Spanning tree with the **minimum sum of edge weights**

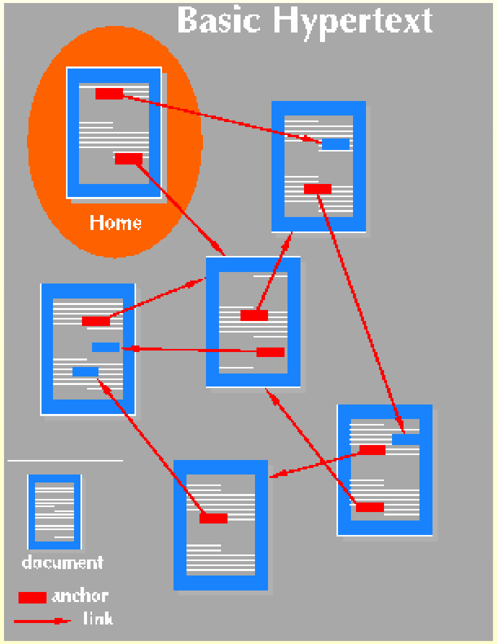


- Spanning forest
  - If a graph is not connected, then there is a spanning tree for each connected component of the graph

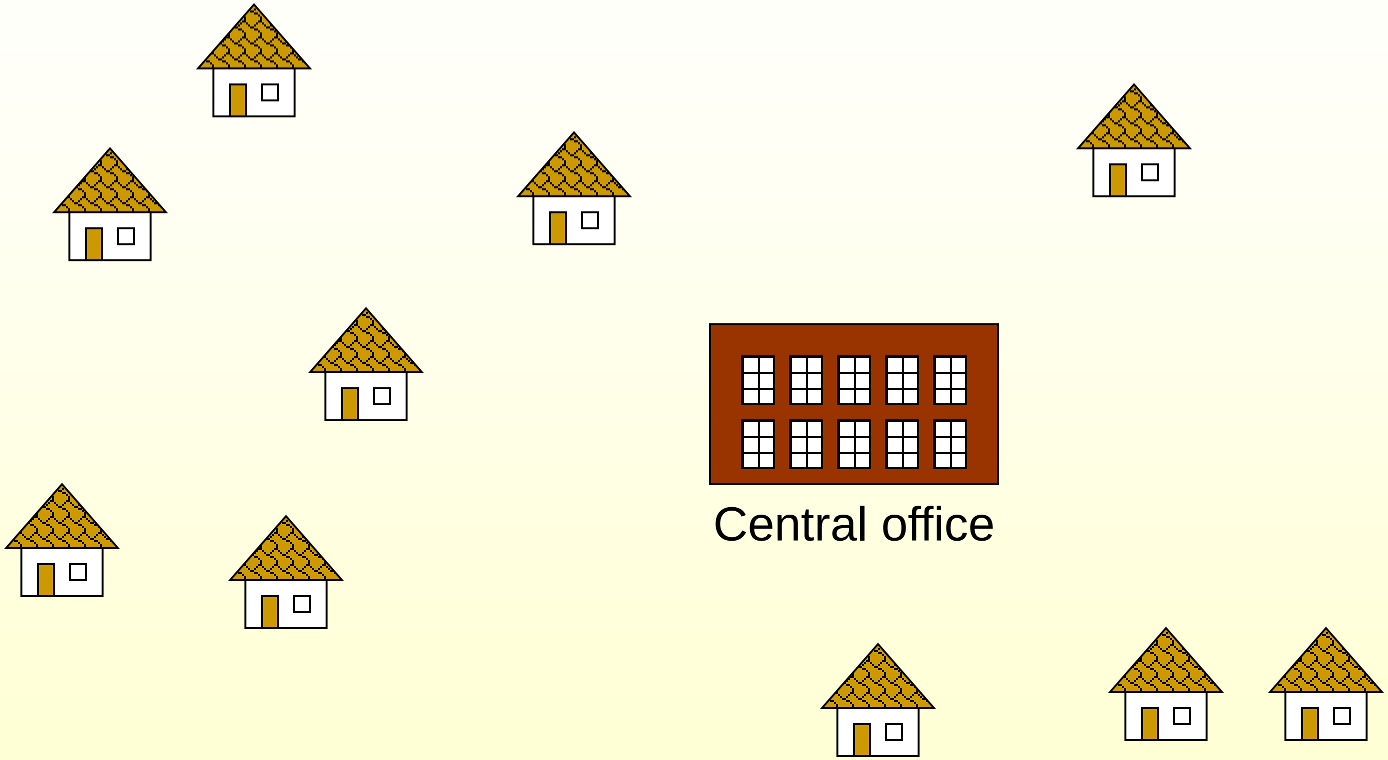
Applications of MSTs

- Find the least expensive way to connect a set of cities, terminals, computers, etc.

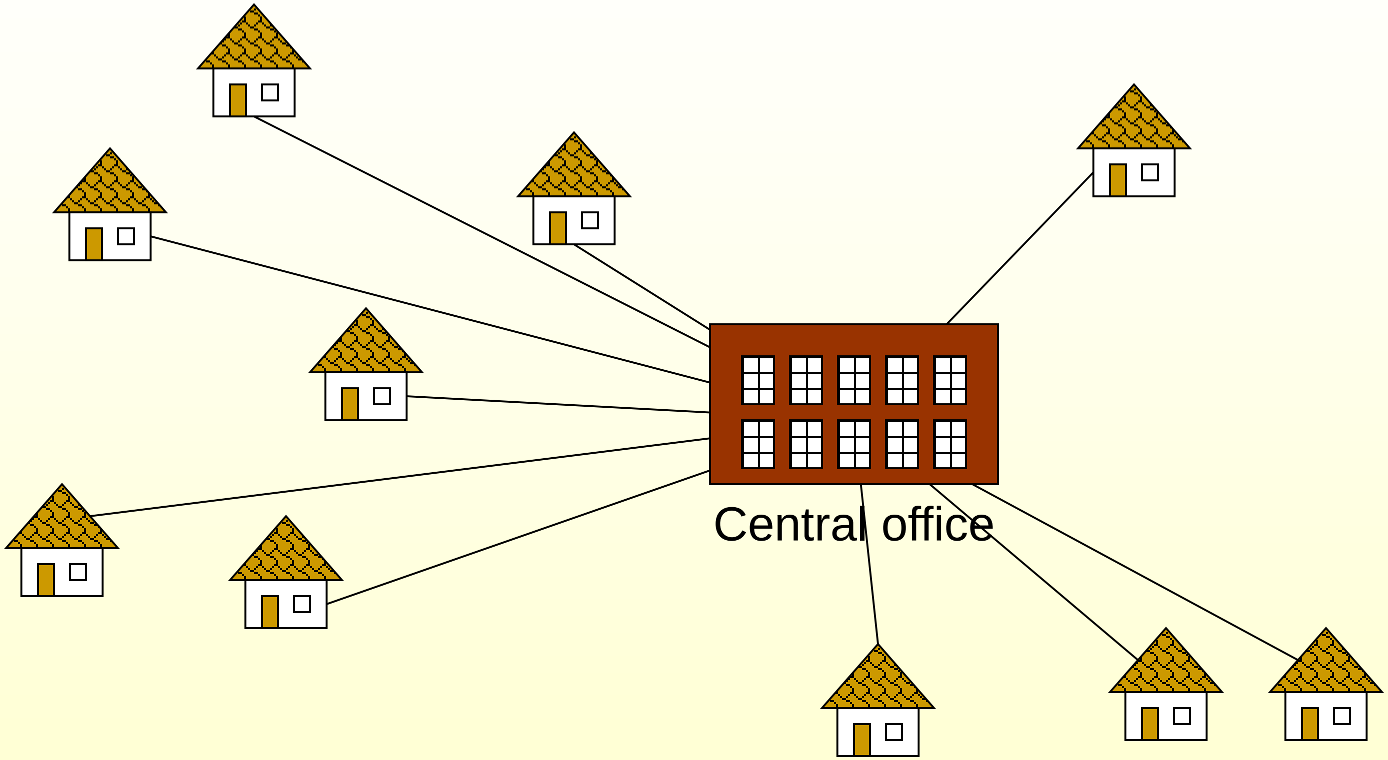




Problem: Laying Telephone Wire

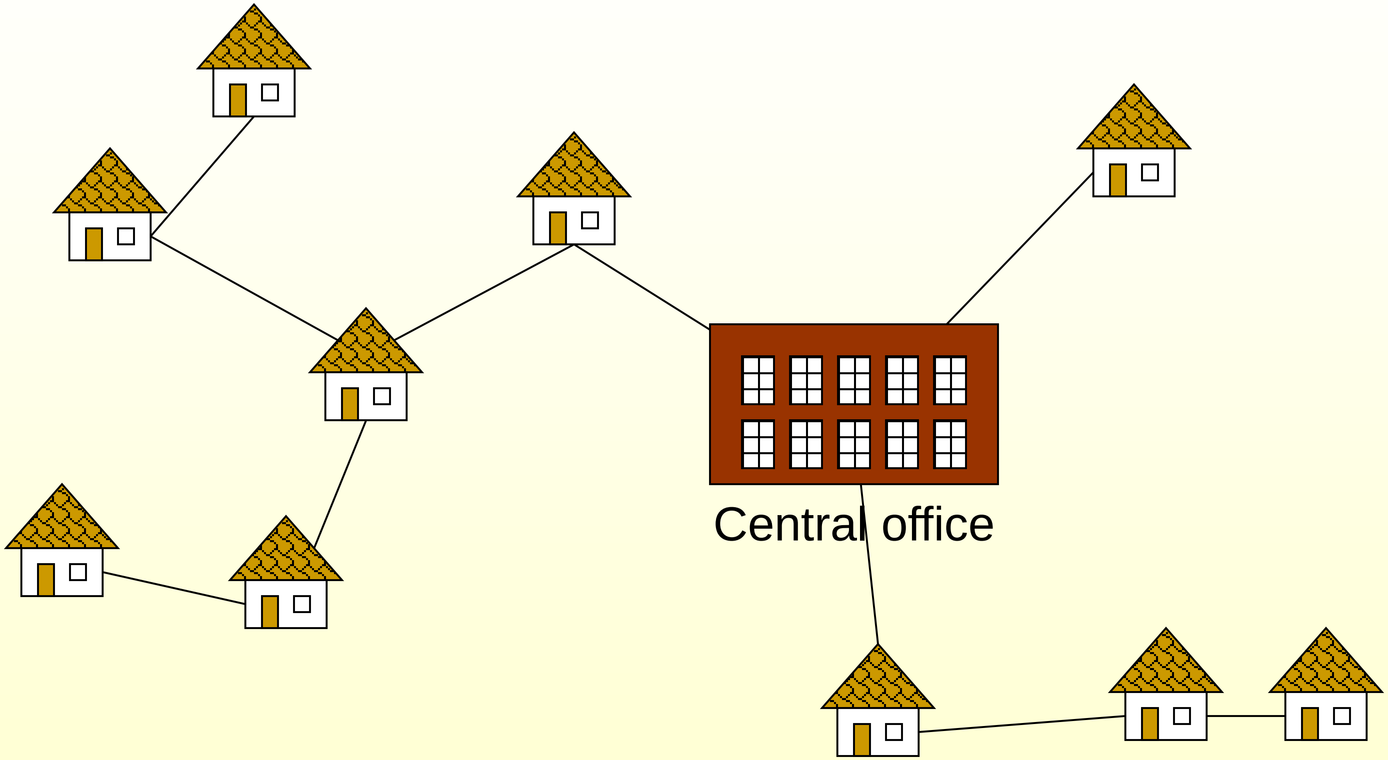


Wiring: Naive Approach



Expensive!

Wiring: Better Approach

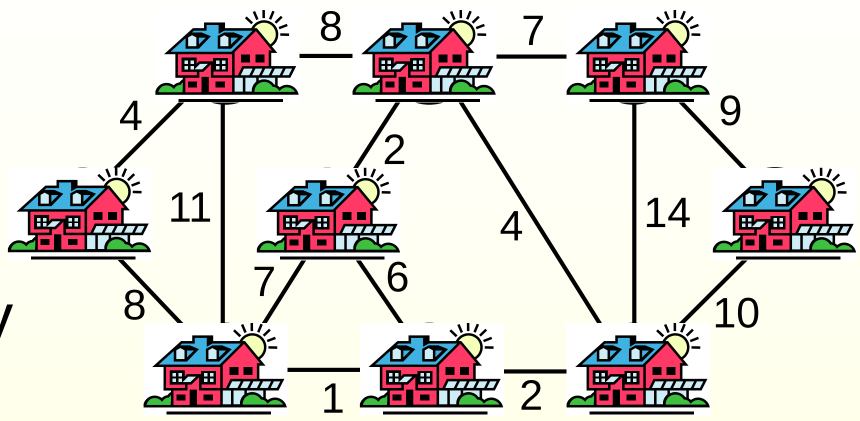


Minimize the total length of wire connecting the customers

Another Example

Problem

- A town has a set of houses and a set of roads
- A road connects two and only two houses
- A road connecting houses u and v has a repair cost w(u, v)



Goal: Repair enough (and no more) roads so that:

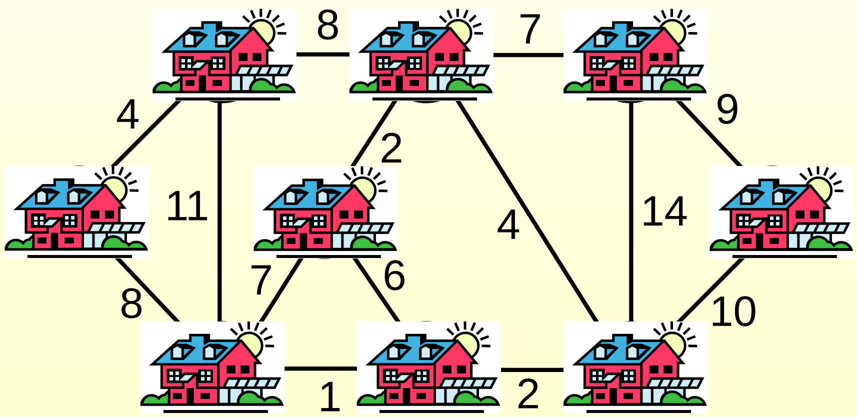
1. Everyone stays connected
   i.e., can reach every house from all other houses
2. Total repair cost is minimum

Minimum Spanning Trees

- A connected, undirected graph:
  - Vertices = houses, Edges = roads
- A **weight** $w(u, v)$ on each edge $(u, v) \in E$

Find $T \subseteq E$ such that:

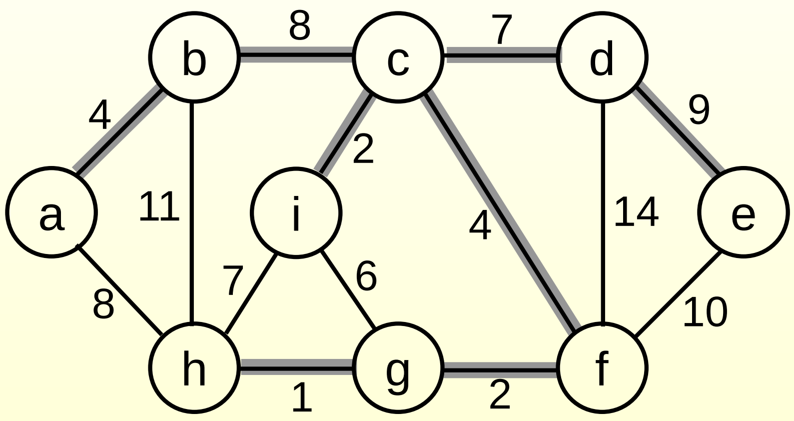
1. T connects all vertices
2. $w(T) = \Sigma_{(u,v)\in T}\, w(u, v)$ is minimized



Growing a MST – Generic Approach

- Grow a set A of edges from G (initially empty)
- Incrementally add edges to A such that they belong

to a MST



Idea: add only "safe" edges

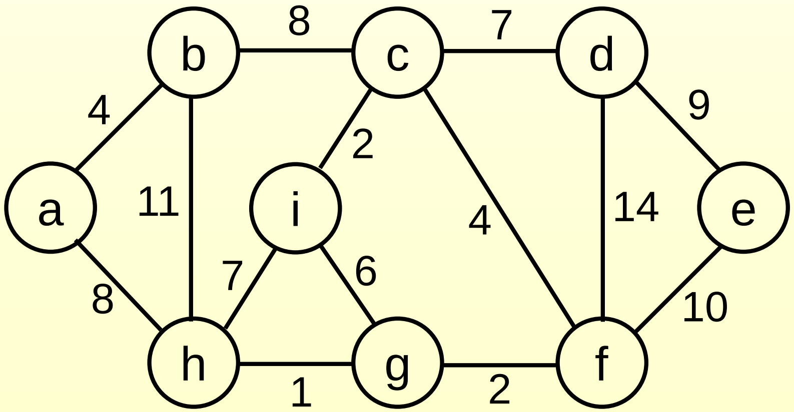
– An edge (u, v) is **safe** for A if and only if $A \cup \{(u, v)\}$ is also a subset of **some** MST

Greedy MST Algorithms

- Greedy algorithms
  - iteratively make "myopic" decisions – aimed at locally optimal choice
  - but somehow everything works out to yield the global optimum at the end – because, as we grow the local solution, we are always consistent with some global solution
- While growing a partial MST, an edge not currently in the tree is safe, if it can be added while still being part of some MST

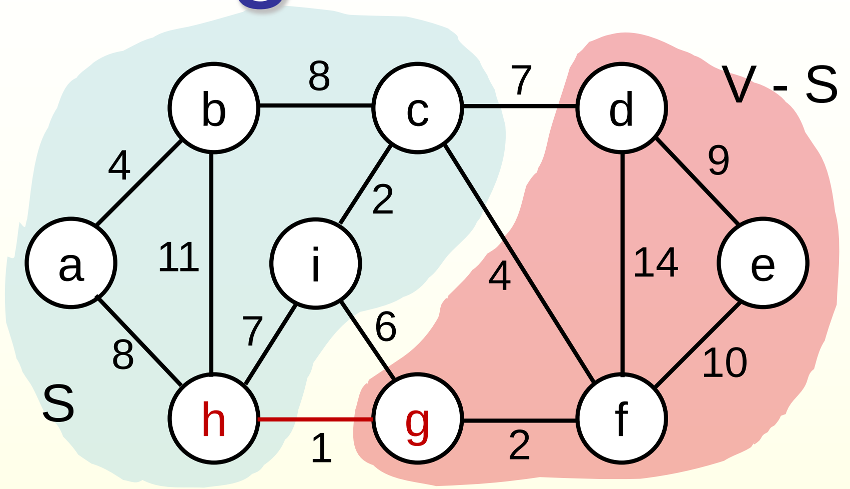
Generic MST algorithm

1. $A \leftarrow \emptyset$
2. **while** A is not a spanning tree
3. **do** find an edge (u, v) that is safe for A
4. $A \leftarrow A \cup \{(u, v)\}$
5. **return** A



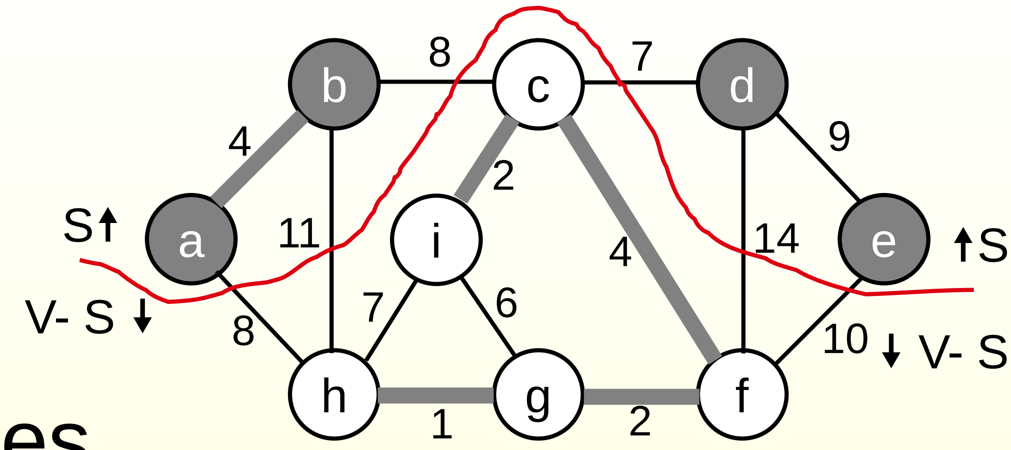
- Key: how do we find safe edges?

Finding Safe Edges



- Let's look at edge (h, *g*)
  - Is it safe for A initially?
- Later on:
  - Let S ⊂ V be any set of vertices that includes h but not *g* (so that *g* is in V - S)
  - In any MST, there has to be one edge (at least) that connects S with V - S
  - Why not choose the edge with **minimum weight** (h,*g*)?

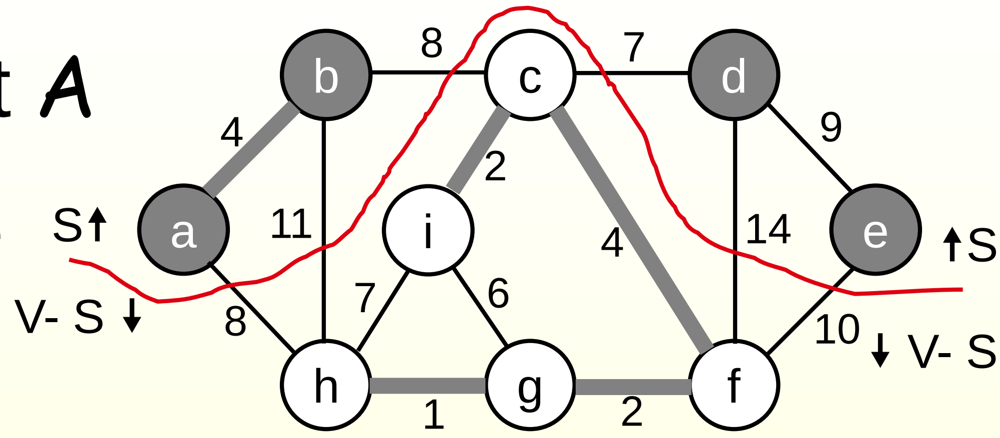
Definitions



- A **cut** (S, V - S) is a partition of vertices into disjoint sets S and V - S
- An edge **crosses** the cut (S, V - S) if one endpoint is in S and the other in V – S

Definitions (cont'd)

- A cut **respects** a set *A* of edges ⇔ no edge in *A* crosses the cut



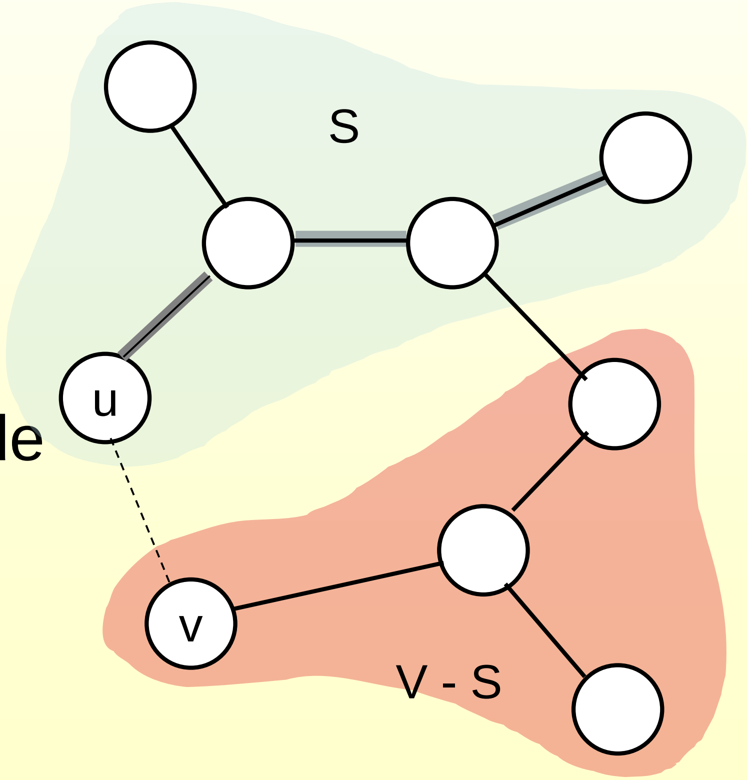
- An edge is a **light edge** crossing a cut ⇔ its weight is minimum over all edges crossing the cut
  - Note that, for a given cut, there can be multiple light edges crossing it

Theorem

- Let *A* be a subset of some MST (i.e., T), (S, V - S) be a **cut** that respects A, and (u, v) be a **light edge** crossing (S, V-S). Then (u, v) is safe for *A* .

Proof:

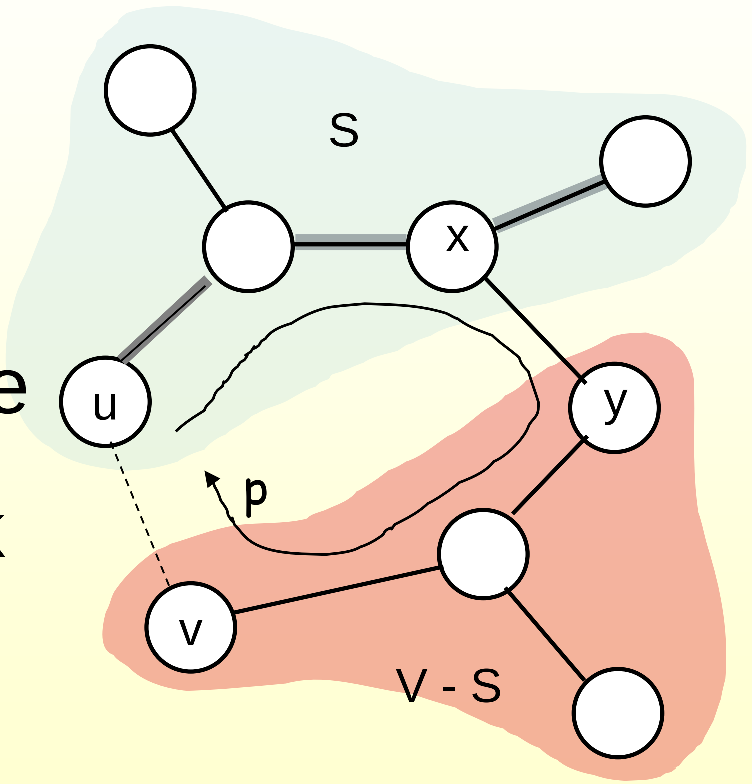
- Let T be an MST that includes A
  - edges in A are shaded
- Case1: If T includes (u,v), then it would be safe for A
- Case2: Suppose T does not include the edge (u, v)
- **Idea**: construct another MST T' that includes $A \cup \{(u, v)\}$



Theorem - Proof

- T contains a unique path **p** between u and v
- Path p must cross the cut (S, V - S) at least once: let (x, y) be that edge
- Let's remove (x,y) ⇒ break T into two components.
- Adding (u, v) reconnects the components

$$T' = T - \{(x, y)\} \cup \{(u, v)\}$$



Theorem – Proof (cont.)

$T' = T - \{(x, y)\} \cup \{(u, v)\}$

Have to show that T' is an MST:

- (u, v) is a light edge

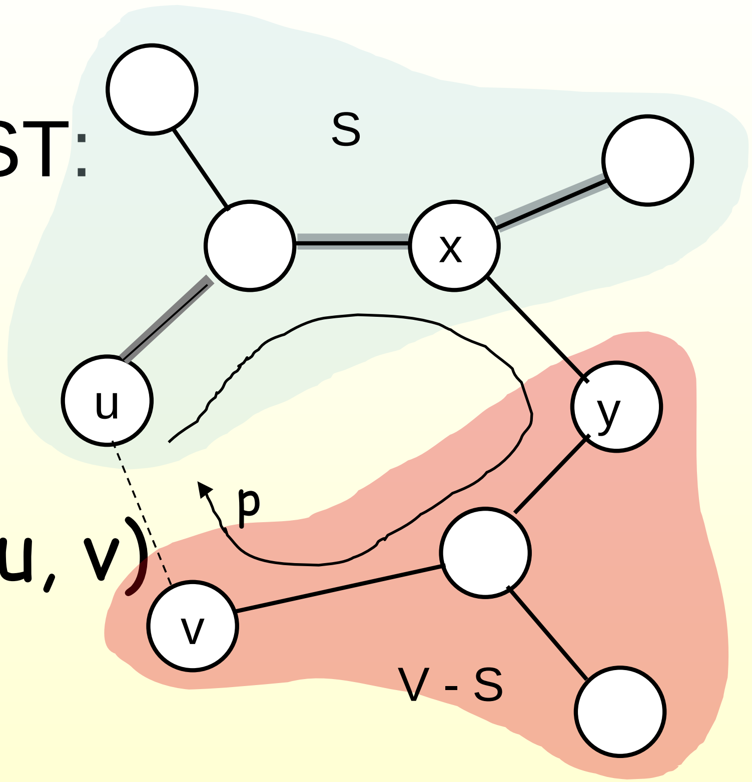
  $\Rightarrow w(u, v) \leq w(x, y)$

- $w(T') = w(T) - w(x, y) + w(u, v)$

  $\leq w(T)$

- Since T is a spanning tree

  $w(T) \leq w(T') \Rightarrow$ T' must be an MST as well



Theorem – Proof (cont.)

Need to show that (u, v) is safe for A:

i.e., (u, v) can be a part of an MST

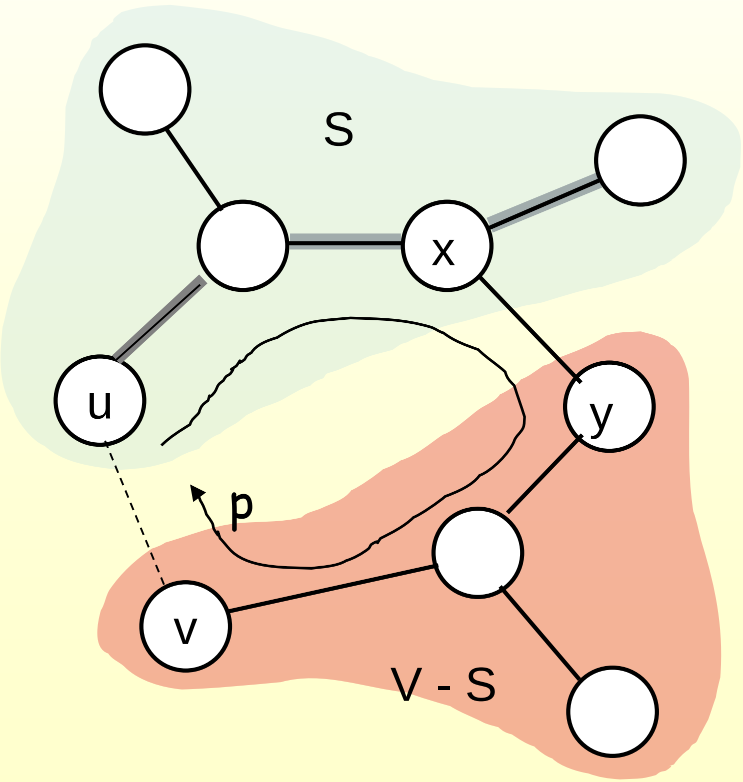
- $A \subseteq T$ and $(x, y) \notin T \Rightarrow$

  $(x, y) \notin A \Rightarrow A \subseteq T'$

- $A \cup \{(u, v)\} \subseteq T'$

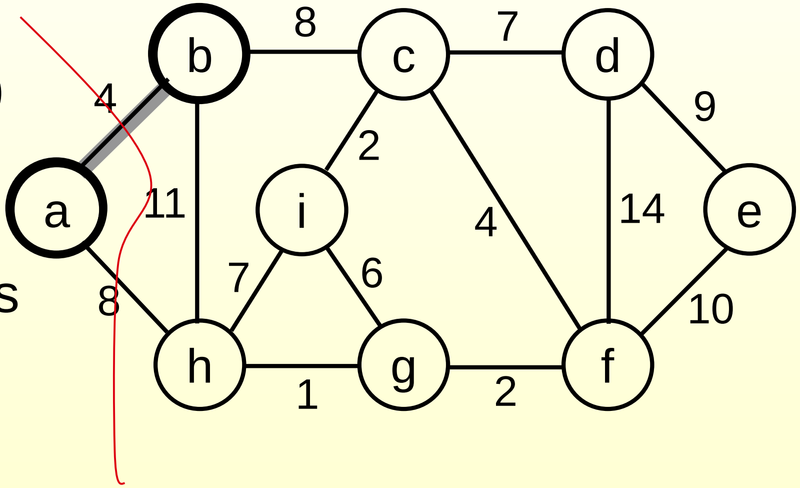
- Since T' is an MST

$\Rightarrow$ (u, v) is safe for A



Prim's Algorithm

- The edges in set A always <u>form a single tree</u>
- Start from an arbitrary "root": $V_A = \{a\}$
- At each step:
  - Find a light edge crossing $(V_A, V - V_A)$
  - Add this edge to A
  - Repeat until the tree spans all vertices

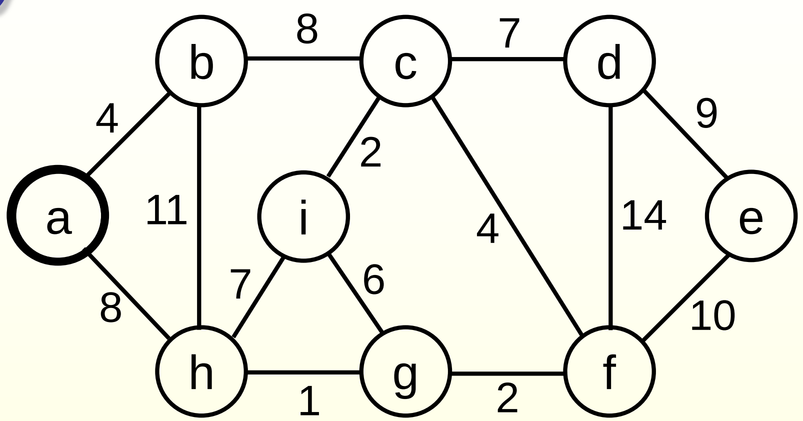


Greedy approach

How to Find Light Edges Quickly?

Use a priority queue Q:

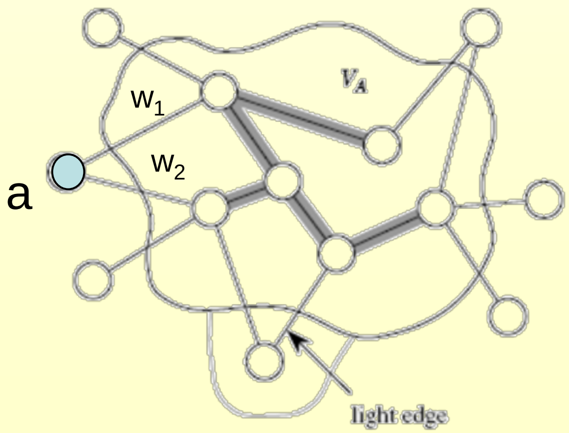
- Contains vertices not yet included in the tree, i.e., ($V – V_A$)
  - V_A = {a}, Q = {b, c, d, e, f, g, h, i}



- We associate a key with each vertex v:

  key[v] = minimum weight of any edge (u, v) connecting v to V_A

Key[a]=min(w_1,w_2)

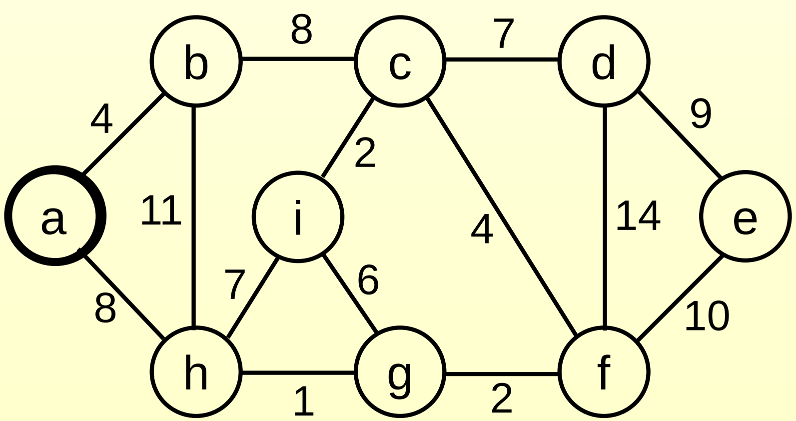


How to Find Light Edges Quickly? (cont.)

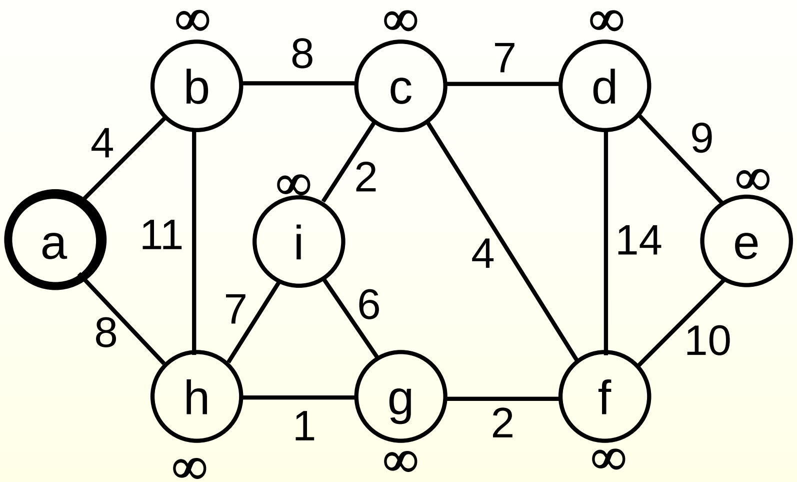
- After adding a new node to V_A we update the weights of all the nodes adjacent to it

  e.g., after adding **a** to the tree, k[b]=4 and k[h]=8

- Key of **v** is ∞ if v is not adjacent to any vertices in V_A



Example

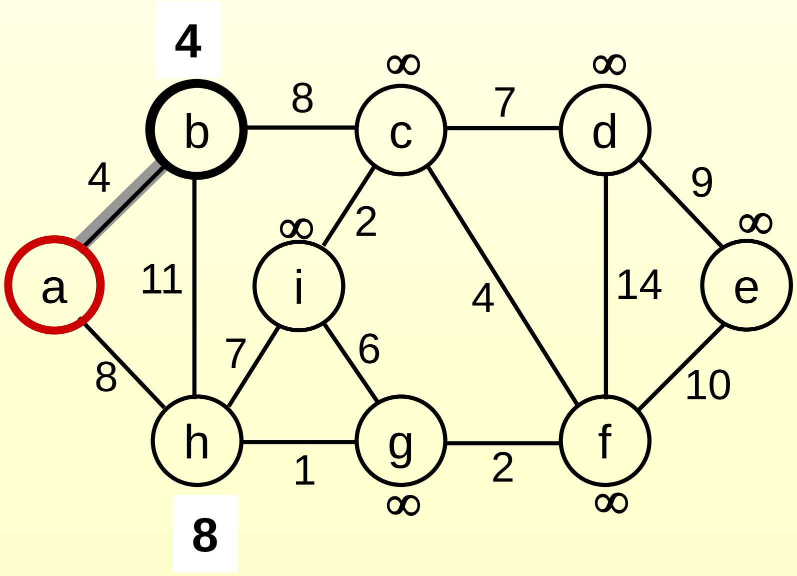


0 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞

Q = {a, b, c, d, e, f, g, h, i}

$V_A = \varnothing$

Extract-MIN(Q) ⇒ a



key [b] = 4   π [b] = a

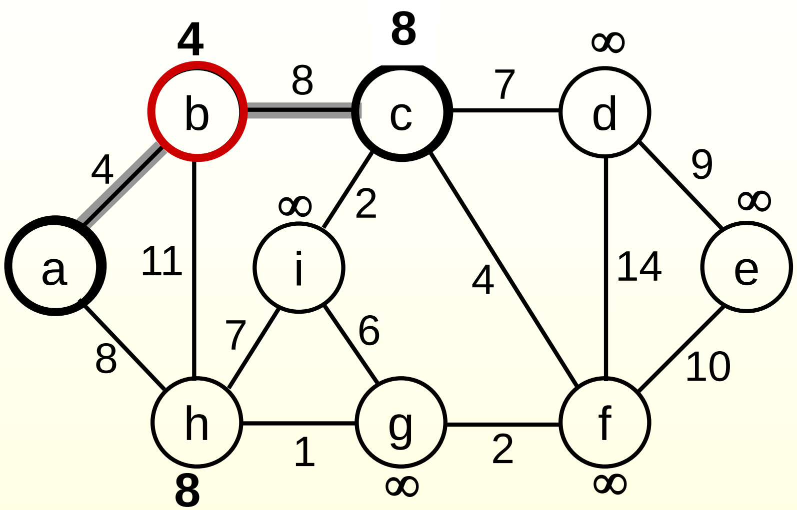
key [h] = 8   π [h] = a

4 ∞ ∞ ∞ ∞ ∞ **8** ∞

Q = {b, c, d, e, f, g, h, i} V_A = {a}

Extract-MIN(Q) ⇒ b

Example



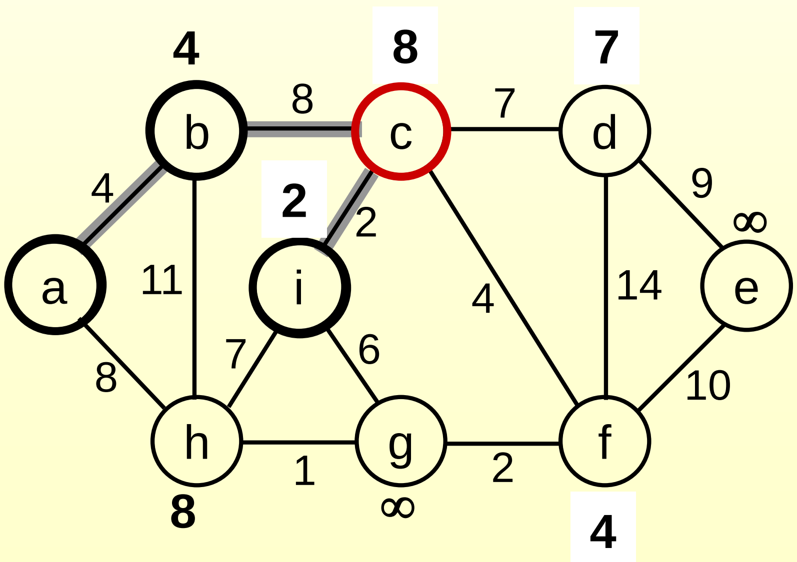
key [c] = 8 π [c] = b

key [h] = 8 π [h] = a - unchanged

8 ∞ ∞ ∞ ∞ 8 ∞

Q = {c, d, e, f, g, h, i} V_A = {a, b}

Extract-MIN(Q) $\Rightarrow$ c



key [d] = 7 π [d] = c

key [f] = 4 π [f] = c

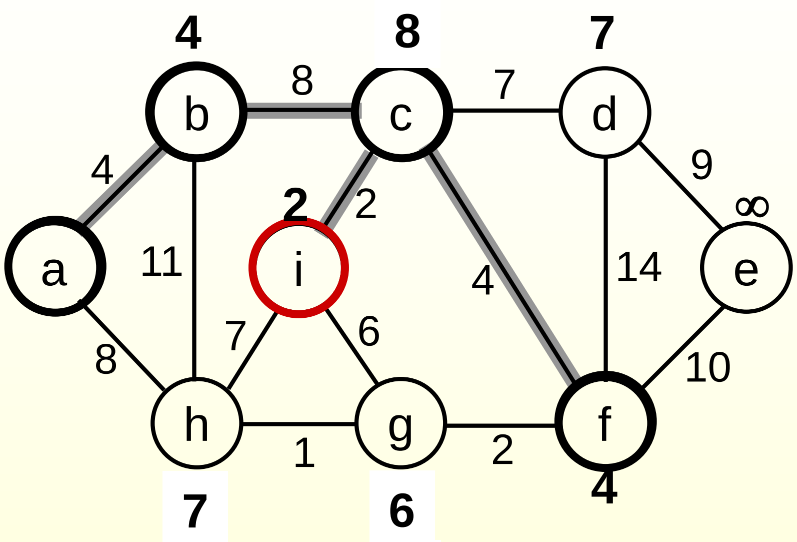
key [i] = 2 π [i] = c

7 ∞ 4 ∞ 8 2

Q = {d, e, f, g, h, i} V_A = {a, b, c}

Extract-MIN(Q) $\Rightarrow$ i

Example



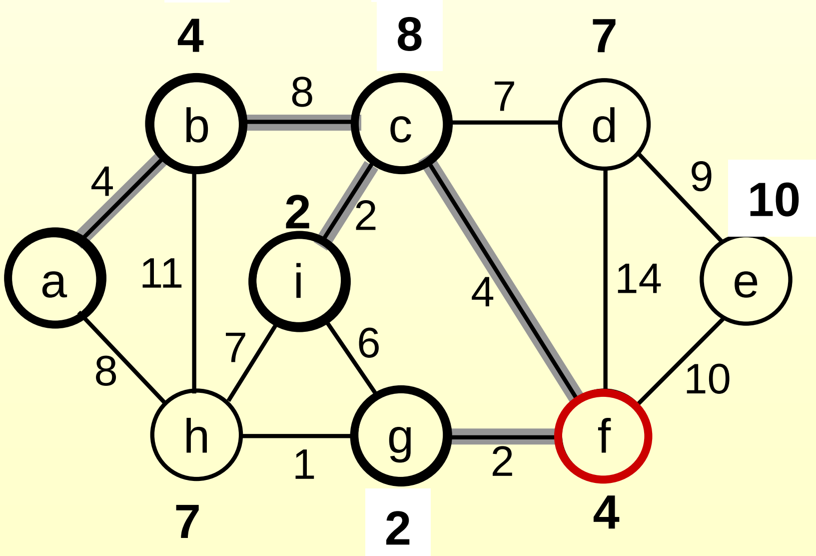
key [h] = 7 π [h] = i

key [g] = 6 π [g] = i

7 ∞ 4 6 8

Q = {d, e, f, g, h} V_A = {a, b, c, i}

Extract-MIN(Q) ⇒ f



key [g] = 2 π [g] = f

key [d] = 7 π [d] = c unchanged

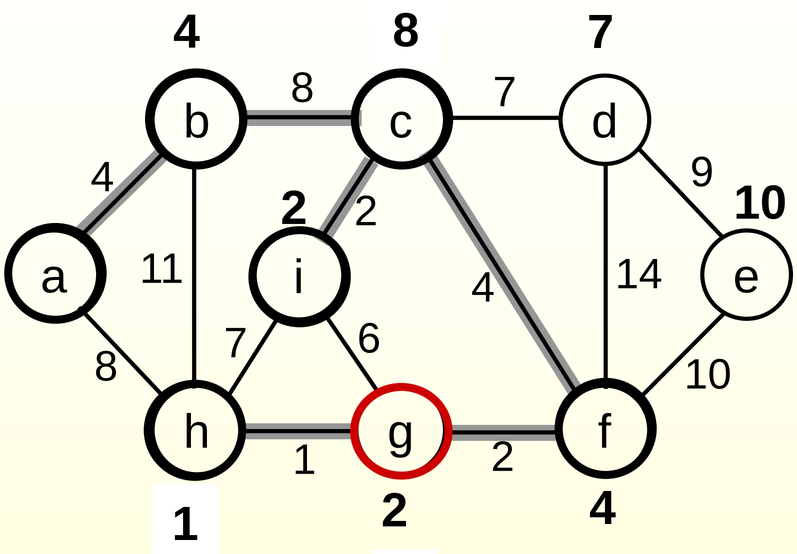
key [e] = 10 π [e] = f

7 10 2 8

Q = {d, e, g, h} V_A = {a, b, c, i, f}

Extract-MIN(Q) ⇒ g

Example

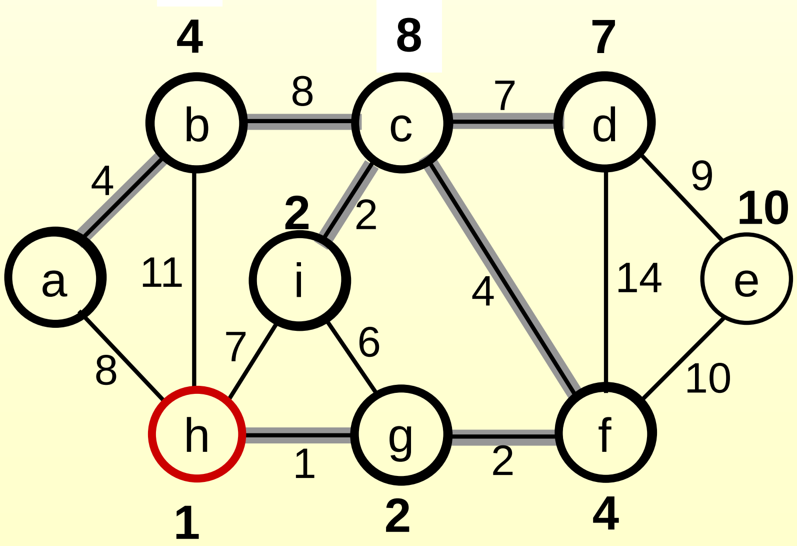


key [h] = 1 π [h] = g

7 10 1

Q = {d, e, h} V_A = {a, b, c, i, f, g}

Extract-MIN(Q) $\Rightarrow$ h

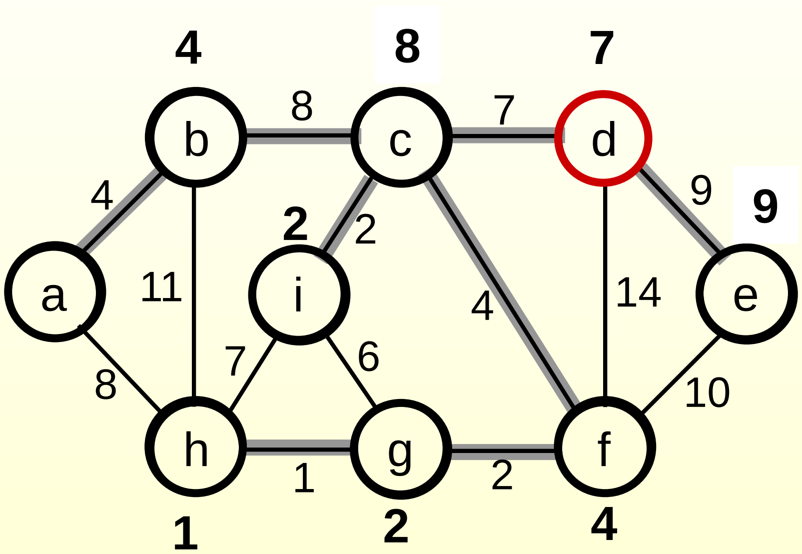


7 10

Q = {d, e} V_A = {a, b, c, i, f, g, h}

Extract-MIN(Q) $\Rightarrow$ d

Example



key [e] = 9 π [e] = f

9

Q = {e} V_A = {a, b, c, i, f, g, h, d}

Extract-MIN(Q) $\Rightarrow$ e

Q = $\varnothing$ V_A = {a, b, c, i, f, g, h, d, e}

PRIM(V, E, w, r)

Total time: $O(V\lg V + E\lg V) = O(E\lg V)$

```
Q ← ∅
for each u ∈ V
    do key[u] ← ∞
        π[u] ← NIL
        INSERT(Q, u)
DECREASE-KEY(Q, r, 0)        ► key[r] ← 0
while Q ≠ ∅
    do u ← EXTRACT-MIN(Q)
        for each v ∈ Adj[u]
            do if v ∈ Q and w(u, v) < key[v]
                then π[v] ← u
                    DECREASE-KEY(Q, v, w(u, v))
```

$O(V)$ if Q is implemented as a min-heap

$O(\lg V)$

Executed |V| times

Takes $O(\lg V)$

Min-heap operations: $O(V\lg V)$

Executed O(E) times total

Constant

Takes $O(\lg V)$

$O(E\lg V)$

Advanced: Using Fibonacci Heaps (CLRS, Ch. 19)

- Depending on the heap implementation, running time could be improved!

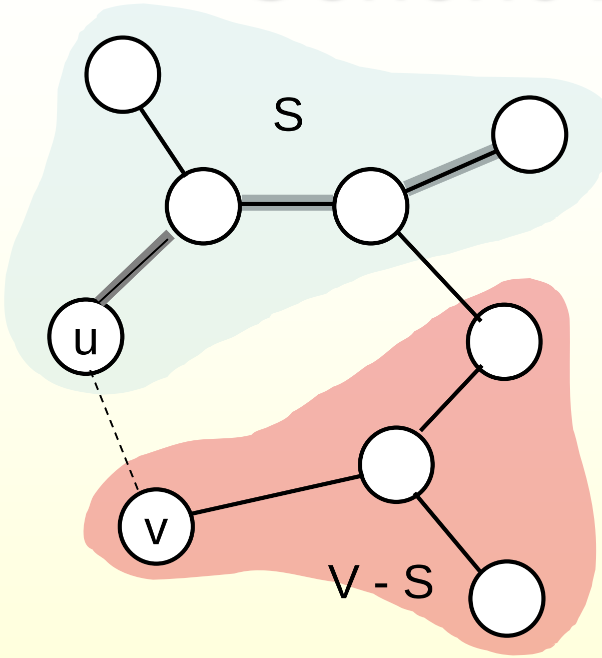
	EXTRACT-MIN	DECREASE-KEY	Total
binary heap	O(lgV)	O(lgV)	O(ElgV)
Fibonacci heap	O(lgV)	O(1)	O(VlgV + E)

Prim’s Algorithm

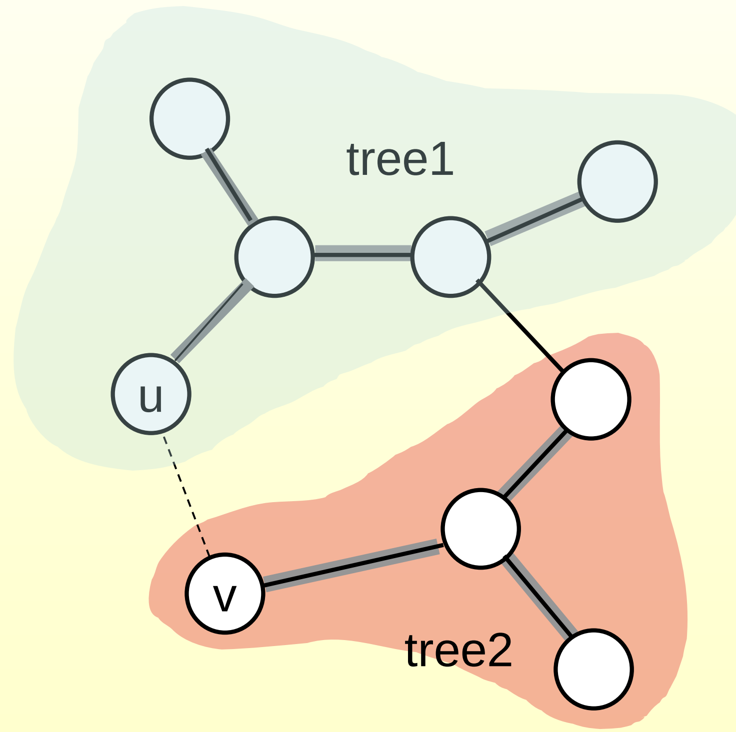
- Prim’s algorithm is a **“greedy”** algorithm
  - Greedy algorithms find solutions based on a sequence of choices which are **“locally”** optimal at each step.
- Nevertheless, Prim’s greedy strategy produces a globally optimum solution
  - See proof for generic approach

Another Instance of the Generic Approach

(instance 1)



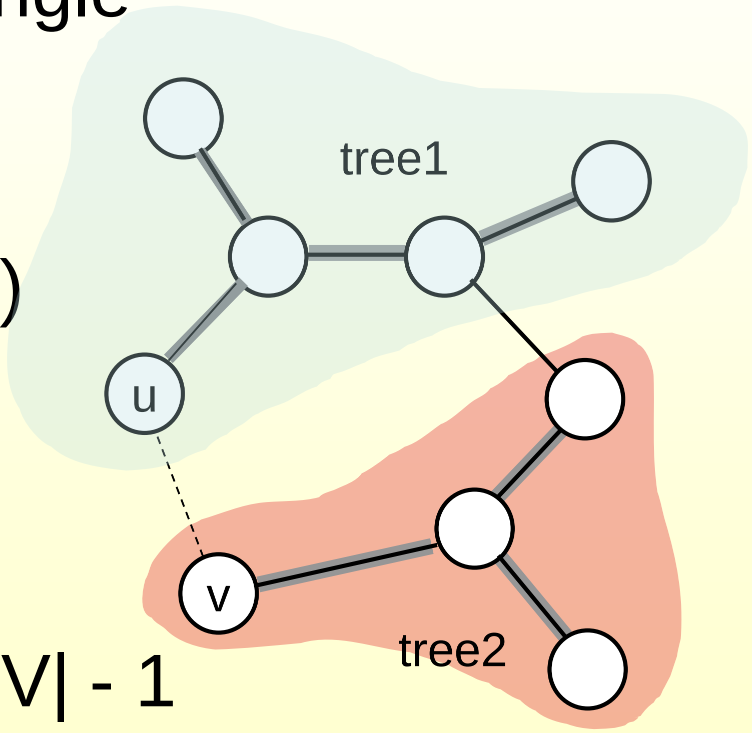
(instance 2)



- *A* is a forest containing connected components
  - Initially, each component is a single vertex
- Any safe edge merges two of these components into one
  - Each component remains a tree

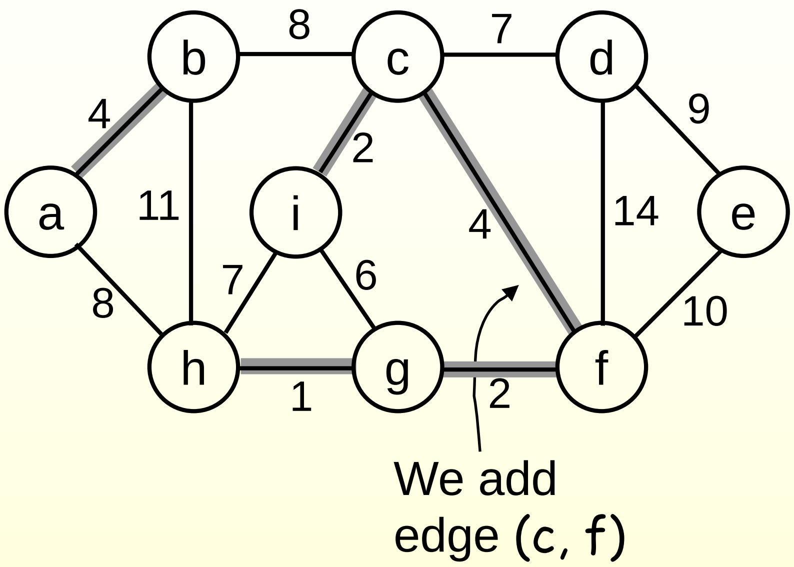
Kruskal's Algorithm

- How is it different from Prim's algorithm?
  - Prim's algorithm grows a single tree at all times
  - Kruskal's algorithm grows multiple trees (i.e., a forest) all at the same time.
  - Trees are merged together using **safe** edges
  - Since an MST has exactly |V| - 1 edges, after |V| - 1 merges, we have only one component

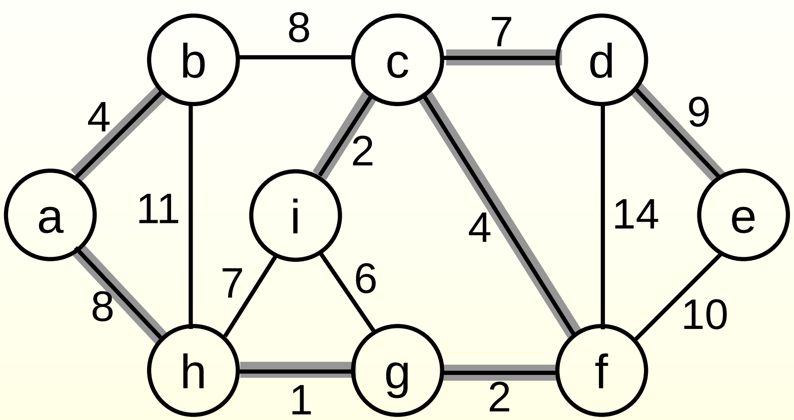


Kruskal's Algorithm

- Start with each vertex being its own connected component
- Repeatedly merge two components into one by choosing a **light** edge that connects them
- Which components to consider at each iteration?
  - Scan the set of edges in monotonically increasing order by weight (guarantees lightness)



Example



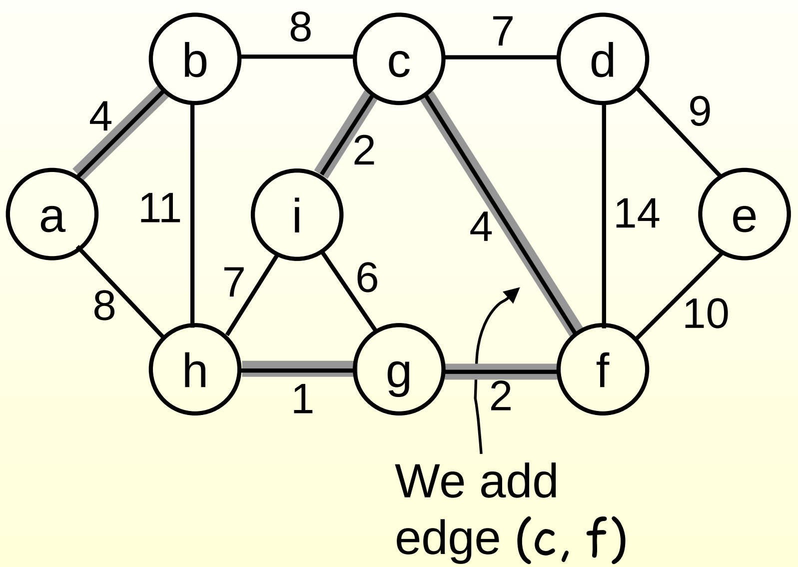
1: (h, g)
2: (c, i), (g, f)
4: (a, b), (c, f)
6: (i, g)
7: (c, d), (i, h)
8: (a, h), (b, c)
9: (d, e)
10: (e, f)
11: (b, h)
14: (d, f)

{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

1.	Add (h, g)	{g, h}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
2.	Add (c, i)	{g, h}, {c, i}, {a}, {b}, {d}, {e}, {f}
3.	Add (g, f)	{g, h, f}, {c, i}, {a}, {b}, {d}, {e}
4.	Add (a, b)	{g, h, f}, {c, i}, {a, b}, {d}, {e}
5.	Add (c, f)	{g, h, f, c, i}, {a, b}, {d}, {e}
6.	Ignore (i, g)	{g, h, f, c, i}, {a, b}, {d}, {e}
7.	Add (c, d)	{g, h, f, c, i, d}, {a, b}, {e}
8.	Ignore (i, h)	{g, h, f, c, i, d}, {a, b}, {e}
9.	Add (a, h)	{g, h, f, c, i, d, a, b}, {e}
10.	Ignore (b, c)	{g, h, f, c, i, d, a, b}, {e}
11.	Add (d, e)	{g, h, f, c, i, d, a, b, e}
12.	Ignore (e, f)	{g, h, f, c, i, d, a, b, e}
13.	Ignore (b, h)	{g, h, f, c, i, d, a, b, e}
14.	Ignore (d, f)	{g, h, f, c, i, d, a, b, e}

Implementation of Kruskal's Algorithm

- Use a disjoint-set data structure (see Ch. 21 in CLRS) to determine whether an edge connects vertices in the same or different components



Operations on Disjoint Data Sets

- MAKE-SET(u) – creates a new set whose only member is u
- FIND-SET(u) – returns a representative element from the set that contains u
  - Any of the elements of the set that has a particular property
  - *E.g.:* $S_u = \{r, s, t, u\}$, the property is that the element be the first one alphabetically

    FIND-SET(u) = r FIND-SET(s) = r
  - FIND-SET has to return the same value for any element of a given set

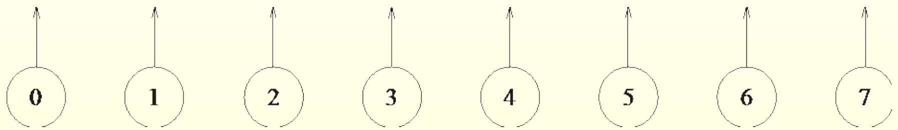
Operations on Disjoint Data Sets

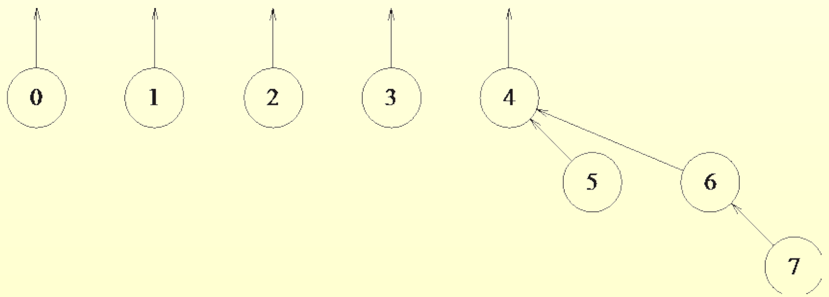
- UNION(u, v) – unites the dynamic sets that contain u and v, say S_u and S_v
  - *E.g.:* $S_u = \{r, s, t, u\}$, $S_v = \{v, x, y\}$

    UNION (u, v) = {r, s, t, u, v, x, y}
- Running time for FIND-SET and UNION depends on specific implementation.
- Can be shown to be amortized **α(n)=o(lg n)** where **α()** is a very slowly growing function – here we just need **O(lg n)** [union by rank]

Quick Union: Tree Implementation

- Each set a tree: Root serves as SetName
  - To Find, follow parent pointers to the root
  - Initially parent pointers set to self
  - To union(u,v), make v's root point to u's root





- After union(4,5), union(6,7), union(4,6)

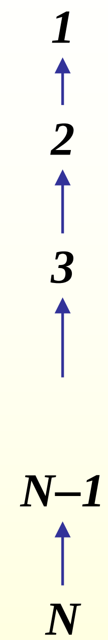
Analysis of Quick Union

```
Initialize(int N)
  parent = new int [N+1];
  for (int e=1; e<=N; e++)
    parent[e] = 0;

int Find(int e)
  while (parent[e] != 0)
    e = parent[e];
  return e;

Union(int i, int j)
  parent[j] = i;
```

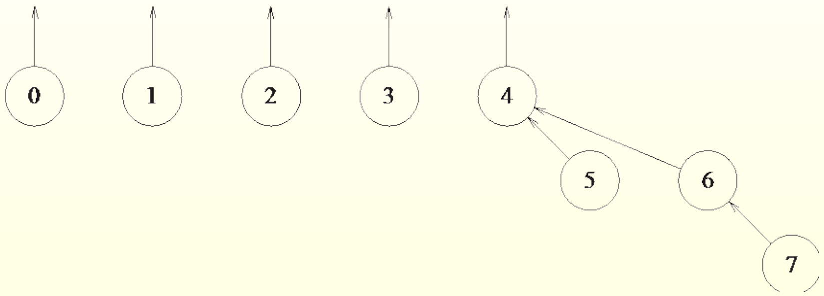
```
Union(N-1, N);
Union(N-2, N-1);
Union(N-3, N-2);
  …
Union(1, 2);
Find(1);
Find(2);
…
Find(N);
```

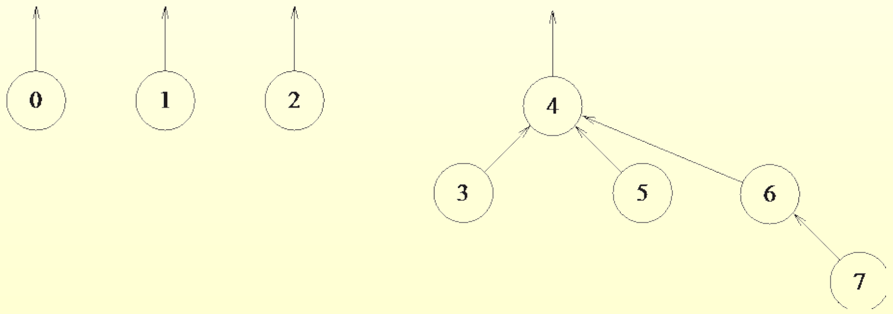


- Complexity in the worst case:
  - Union is $O(1)$ but Find is $O(n)$
  - u Union, f Find : $O(u + f\,n)$
  - N operations: $\Theta(N^2)$ total time

Smart Union (Union by Size)

- union(u,v): make smaller tree point to bigger one's root
- That is, make v's root point to u's root if v's tree is smaller.
- Union(4,5), union(6,7), union(4,6) .





- Now perform union(3, 4). Smaller tree made the child node.

Union by Size:
Link Small Tree to Large One

```
Initialize(int N)
  setsize = new int[N+1];
  parent = new int [N+1];
  for (int e=1; e <= N; e++)
    parent[e] = 0;
    setsize[e] = 1;

int Find(int e)
  while (parent[e] != 0)
    e = parent[e];
  return e;

Union(int i, int j)
  if setsize[i] < setsize[j]
  then
    setsize[j] += setsize[i];
    parent[i] = j;
  else
    setsize[i] += setsize[j];
    parent[j] = i ;
```

Lemma: After n union ops, the tree height is at most log n.

Union by Size: Analysis

- Find(u) takes time proportional to u's depth in its tree.
- Show that if u's depth is h, then its tree has at least 2^h nodes.

- When union(u,v) performed, the depth of u only increases if its root becomes the child of v.
  - That only happens if v's tree is larger than u's tree.

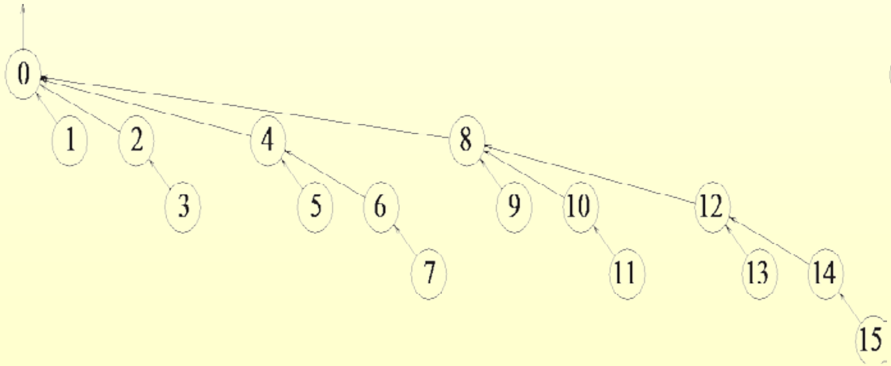
- If u's depth grows by 1, its (new) treeSize is > 2 * oldTreeSize
  - Each increment in depth doubles the size of u's tree.
  - After n union operations, size is at most n, so depth at most log n.

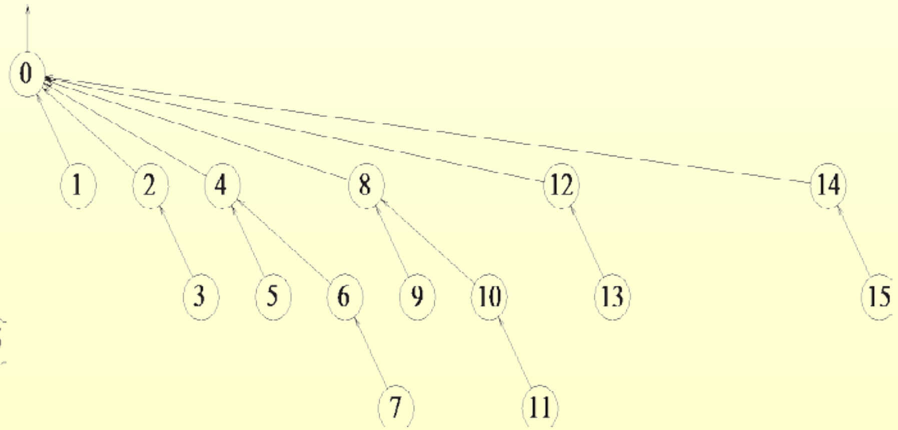
- Theorem: With Union-By-Size, we can do find in O(log n) time and union in O(1) time (assuming roots of u, v known).
- *N-1* Unions, *O(N)* Finds: *O*(*N log N*) total time

The Ultimate Union-Find: Path Compression

```
int Find(int e)
  if (parent[e] == 0)
    return e
  else
    parent[e] = Find(parent[e])
    return parent[e]
```

- While performing Find, direct all nodes on the path to the root.
- Example: Find(14)





The Ultimate Union-Find: Path Compression

```
int Find(int e)
  if (parent[e] == 0)
    return e
  else
    parent[e] = Find(parent[e])
    return parent[e]
```

- Any single find can still be O(log N),
  but later finds on the same path are faster
- Analysis of UF with Path Compression a tour de force [Robert Tarjan]

- u Unions, f Finds: $O(u + f\,\alpha(f, u))$
- $\alpha(f, u)$ is a functional inverse of Ackermann's function
- N-1 Unions, $O(N)$ Finds: "almost linear" total time

KRUSKAL(V, E, w)

```
A ← ∅
for each vertex v ∈ V                            O(V)
        do MAKE-SET(v)
sort E into non-decreasing order by w            O(ElgE)
for each (u, v) taken from the sorted list       ← O(E)
        do if FIND-SET(u) ≠ FIND-SET(v)
                then A ← A ∪ {(u, v)}            ← O(lgV)
                        UNION(u, v)
return A
```

Running time: O(V+ElgE+ElgV)=O(ElgE) – depending on the implementation of the disjoint-set data structure

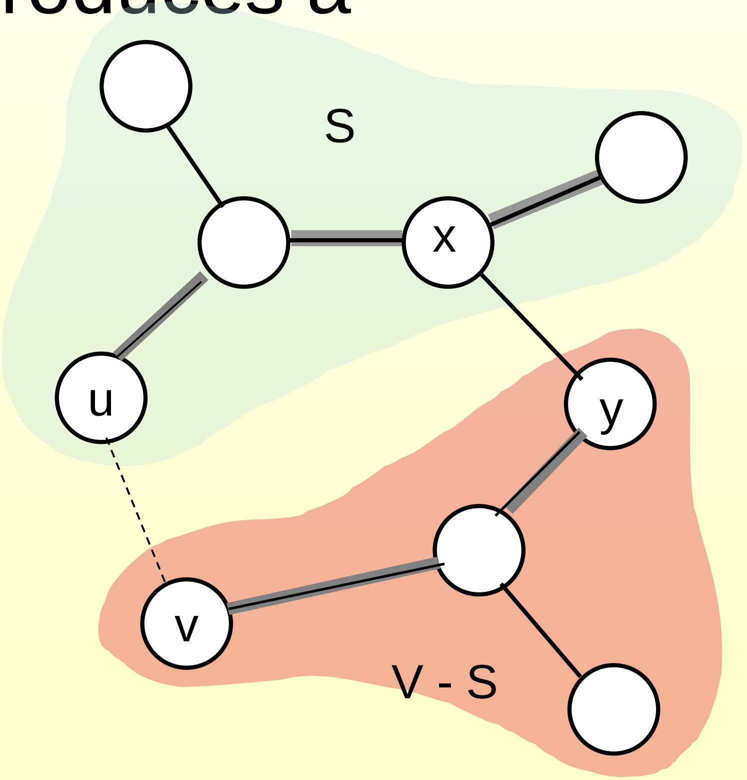
KRUSKAL(V, E, w)

$A \leftarrow \varnothing$
for each vertex $v \in V$
    do MAKE-SET(v) **O(V)**
sort E into non-decreasing order by w **O(ElgE)**
for each (u, v) taken from the sorted list **O(E)**
    do if FIND-SET(u) $\neq$ FIND-SET(v)
        then $A \leftarrow A \cup \{(u, v)\}$
            UNION(u, v) **O(lgV)**
return A

- Running time: O(V+ElgE+ElgV)=O(ElgE)
- Since E=O(V^2), we have lgE=O(2lgV)=O(lgV)

Kruskal's Algorithm

- Kruskal's algorithm is also a **"greedy"** algorithm
- Kruskal's greedy strategy produces a globally optimum solution
- Proof for generic approach applies to Kruskal's algorithm too



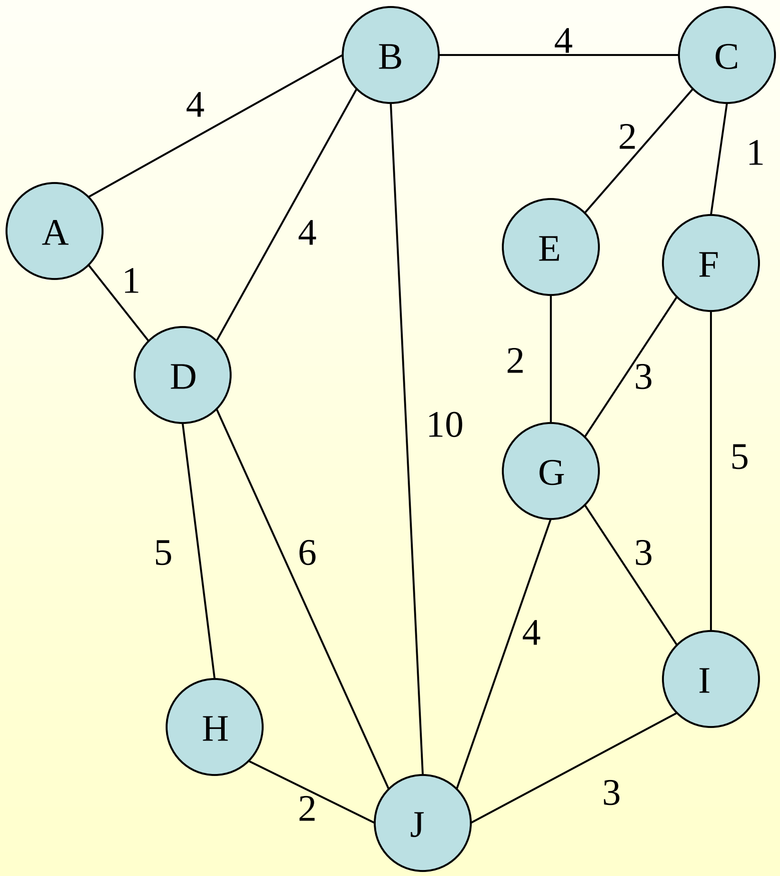
Baruvka's Algorithm

- Like Kruskal's Algorithm, Baruvka's algorithm grows many "clouds" at once, but is more "parallel".

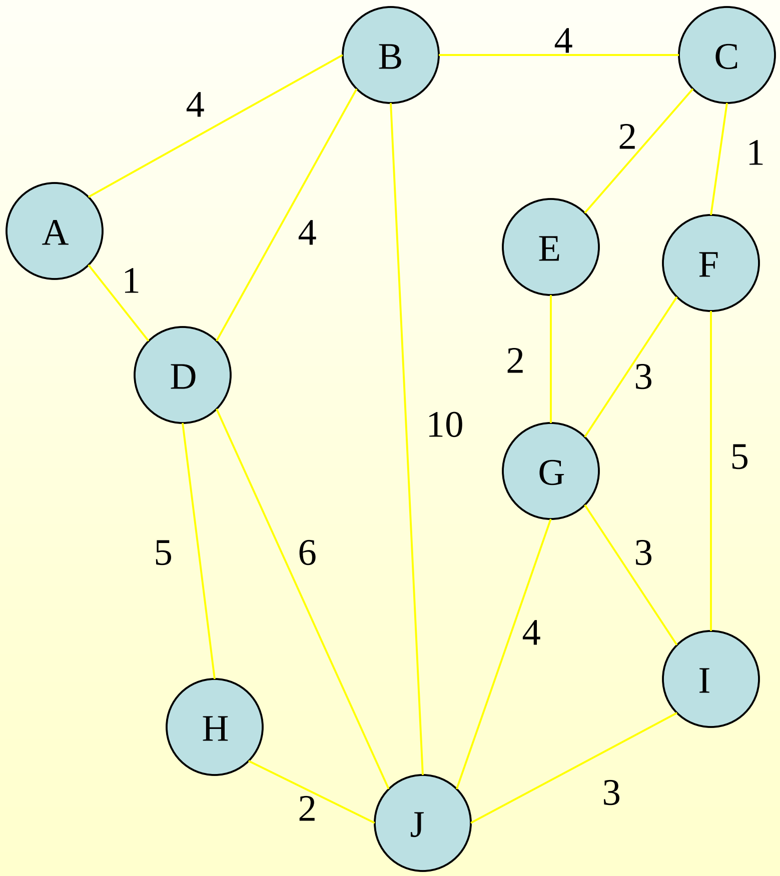
```
Algorithm BaruvkaMST(G)
    T ← V  {just the vertices of G}
  while T has fewer than V|-1 edges do
    for each connected component C in T do
      Let edge e be the smallest-weight edge from C to another component in T.
      if e is not already in T then
        Add edge e to T
  return T
```

- Each iteration of the while-loop halves the number of connected compontents in T.
  - The running time is O(E log V).

Complete Graph



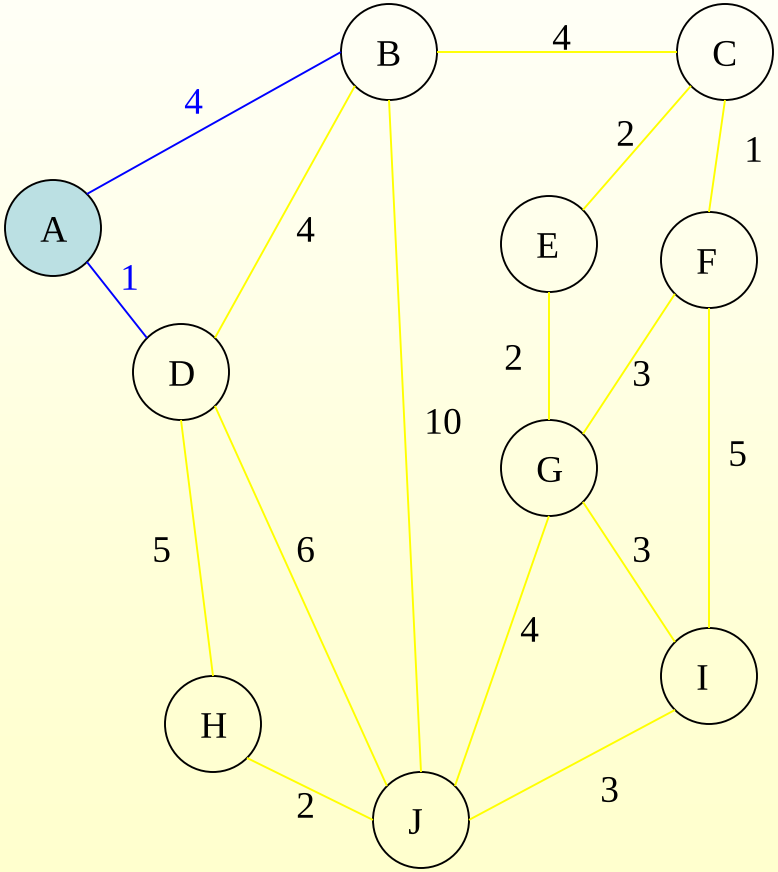
Trees of the Graph at Beginning of Round 1



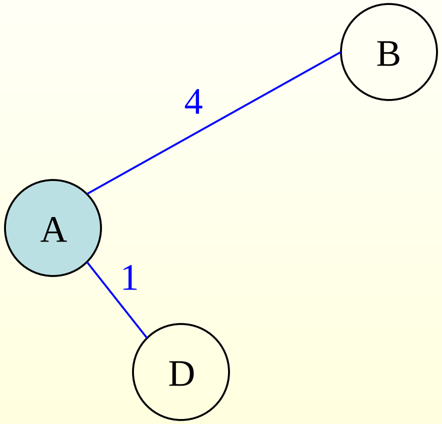
List of Trees

- A
- B
- C
- D
- E
- F
- G
- H
- I
- J

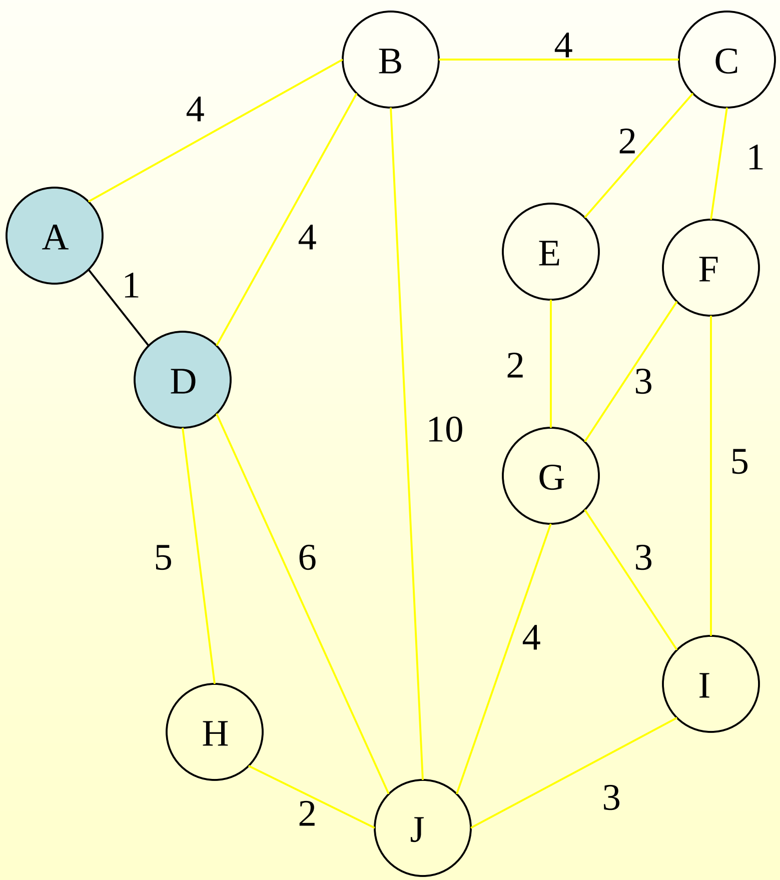
Round 1



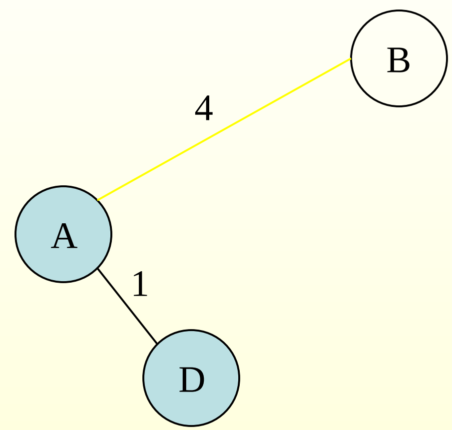
Tree A



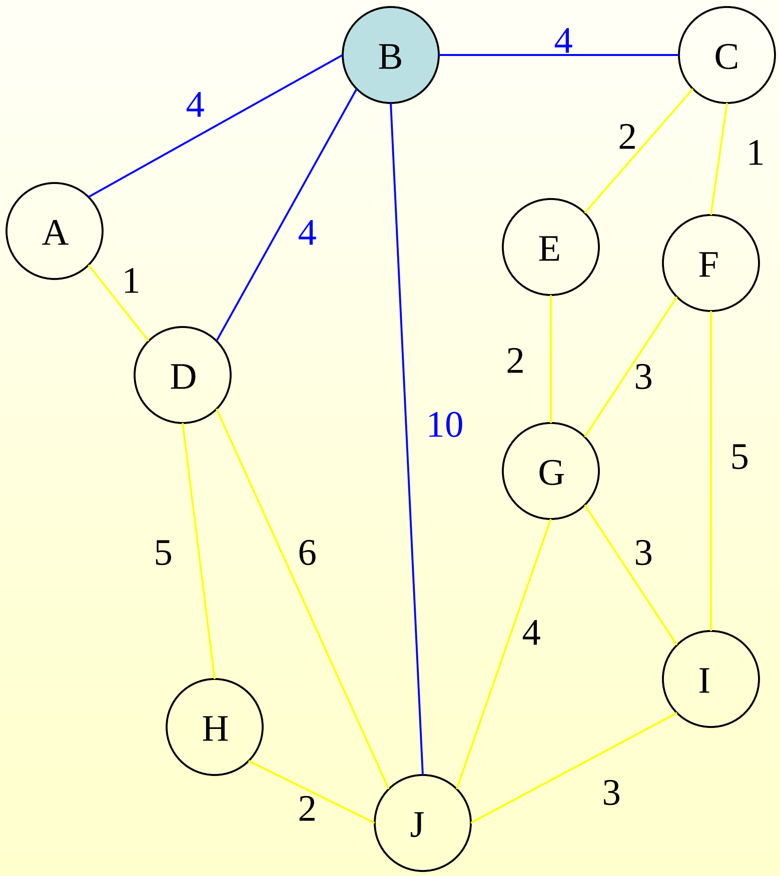
Round 1



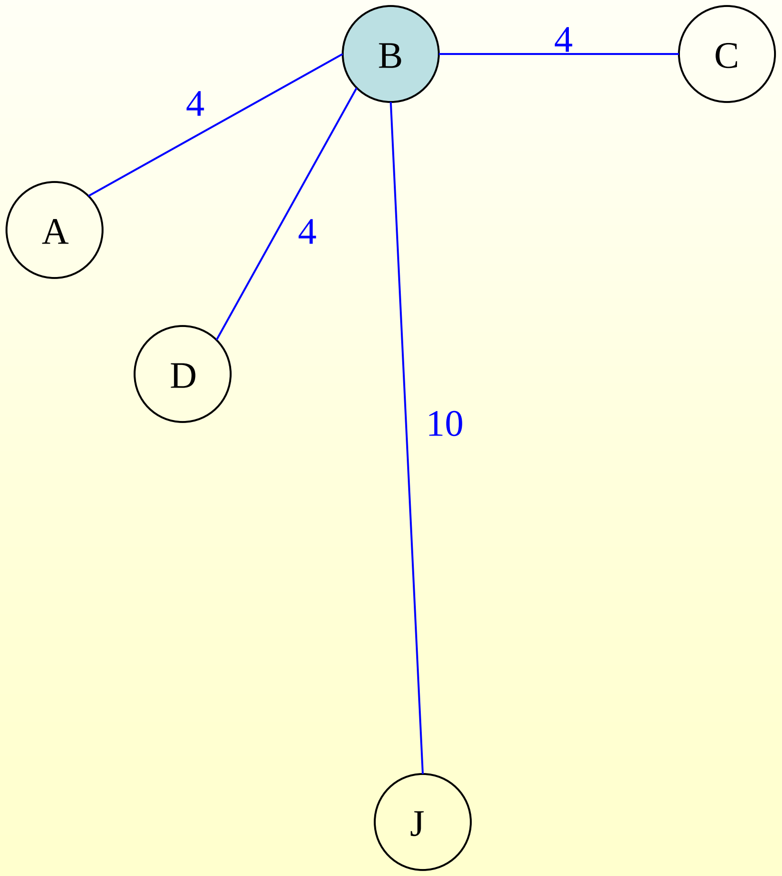
Edge A-D



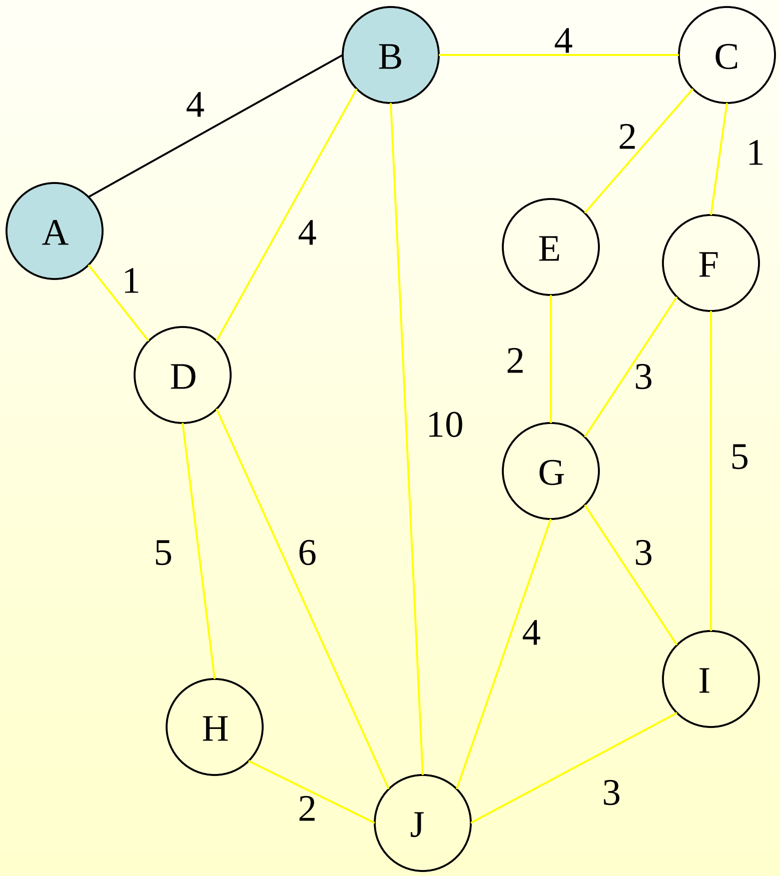
Round 1



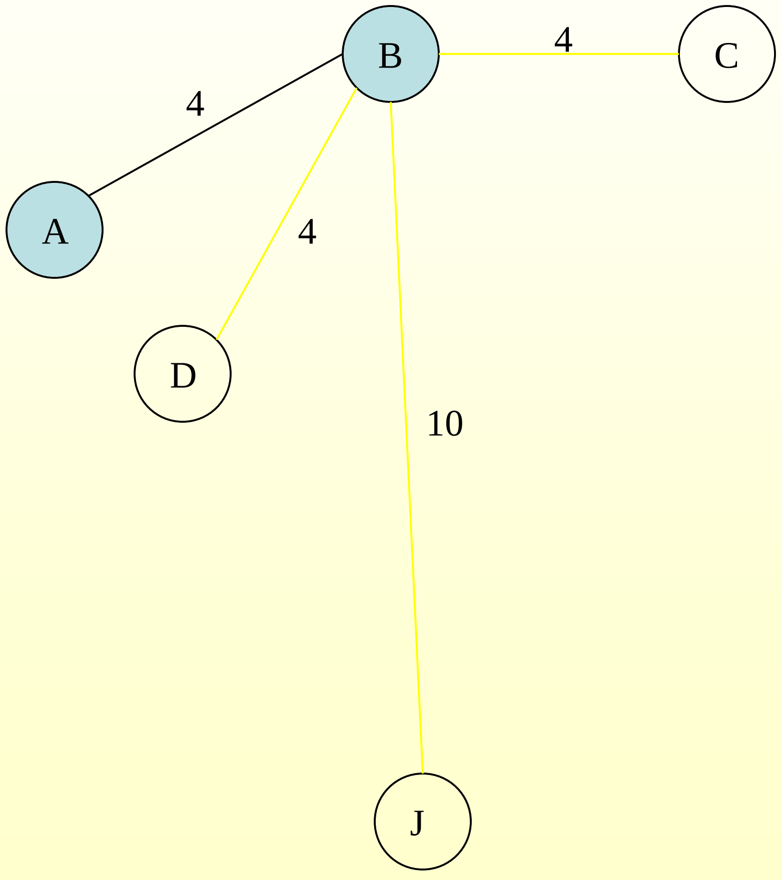
Tree B



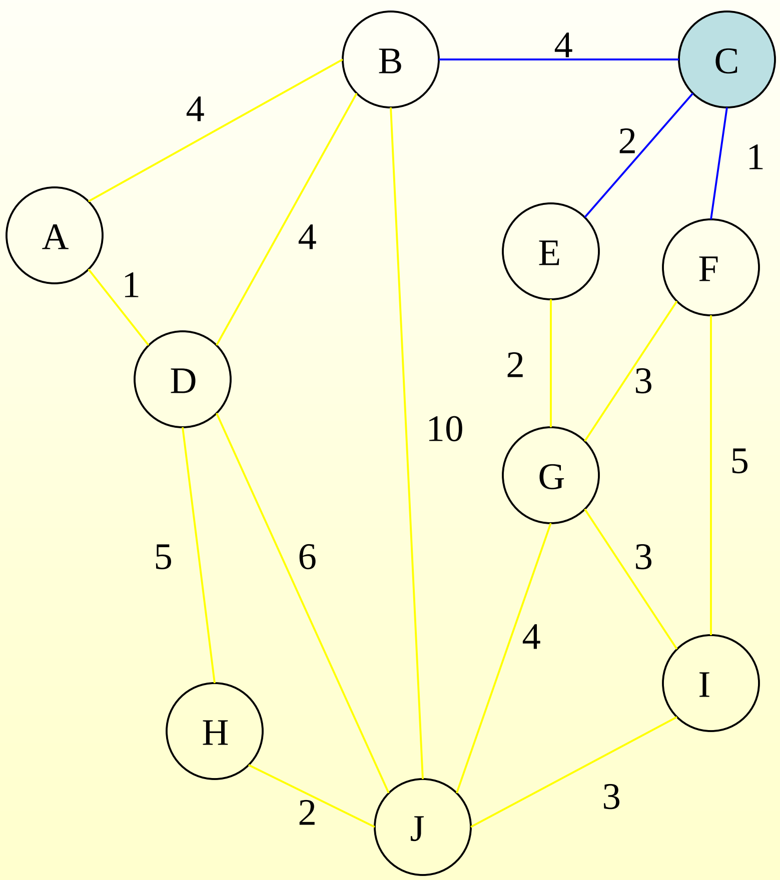
Round 1



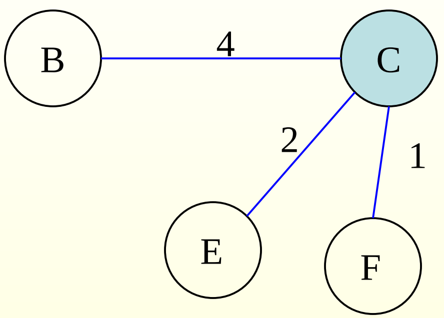
Edge B-A



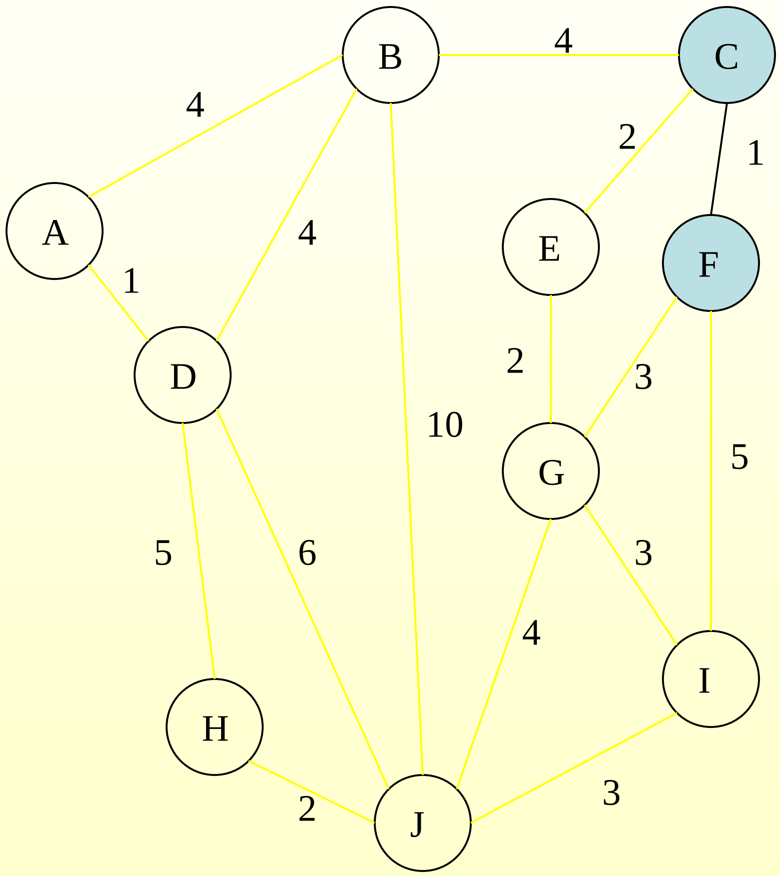
Round 1



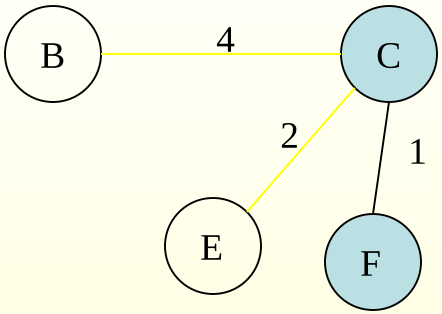
Tree C



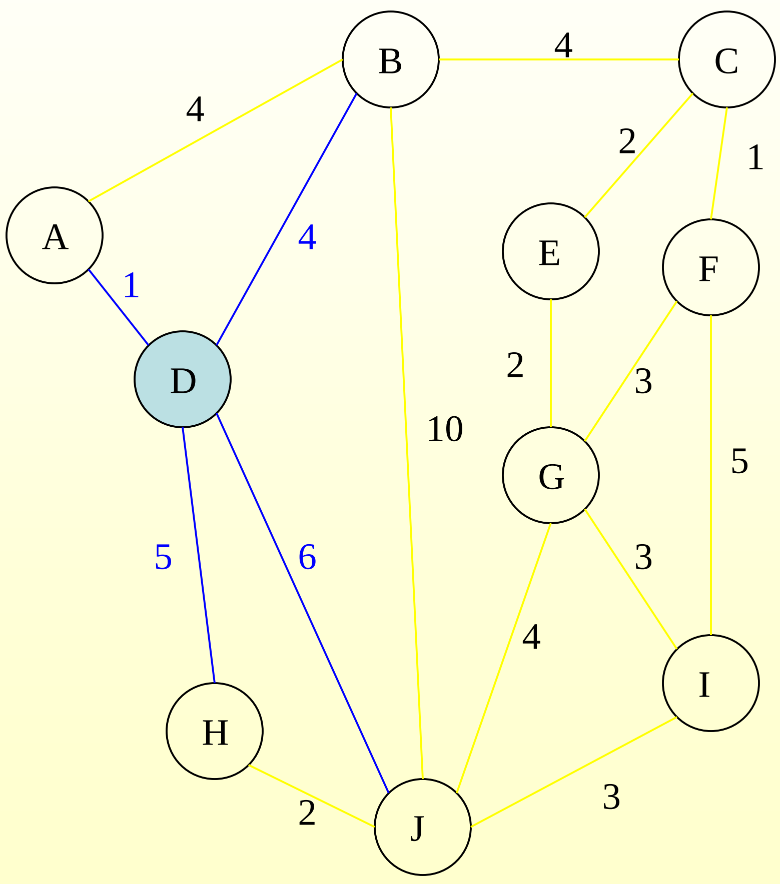
Round 1



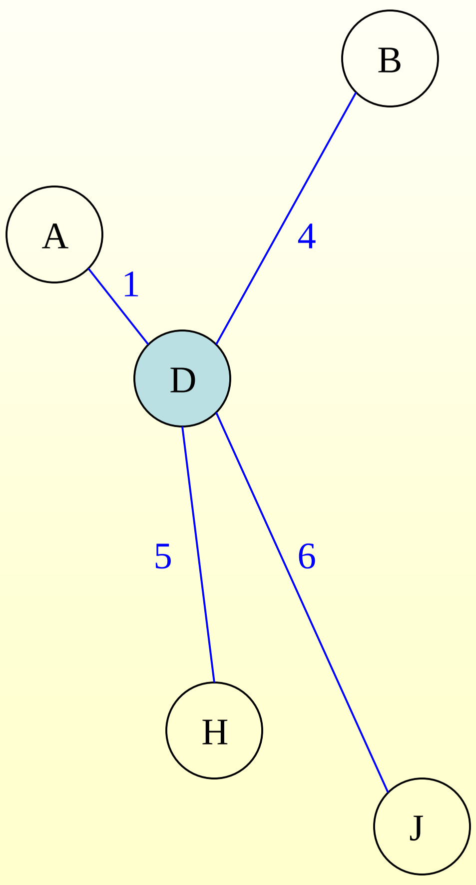
Edge C-F



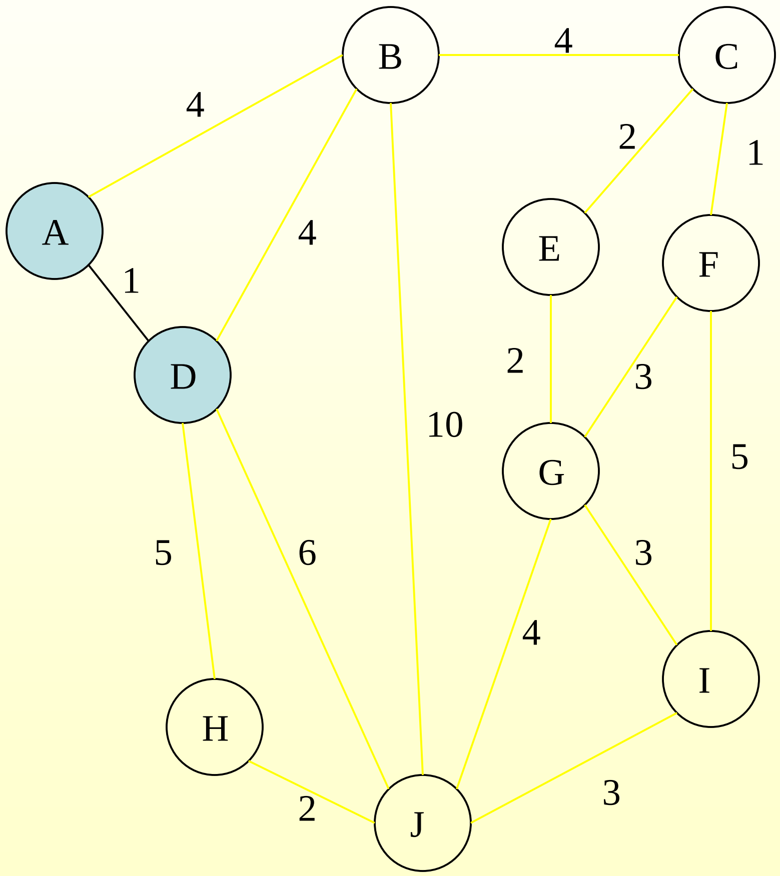
Round 1



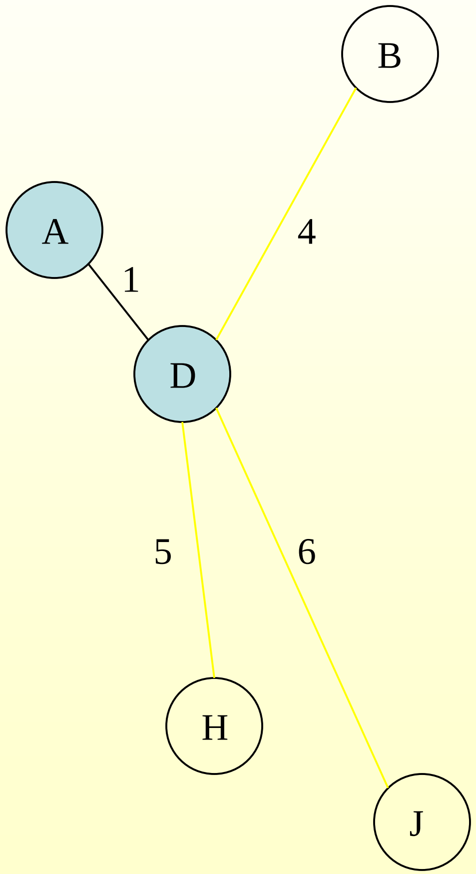
Tree D



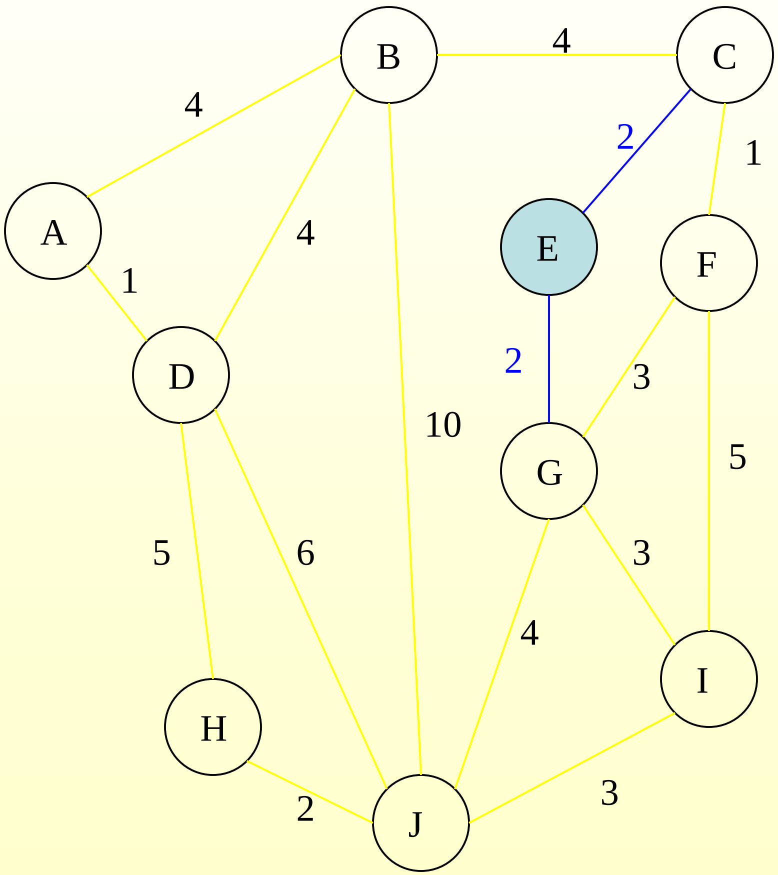
Round 1



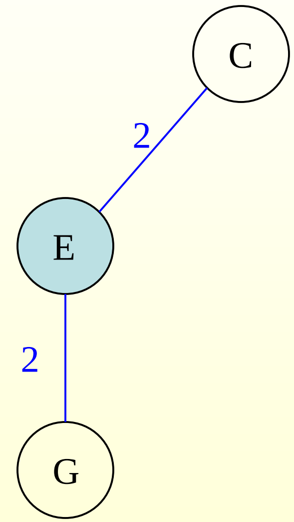
Edge D-A



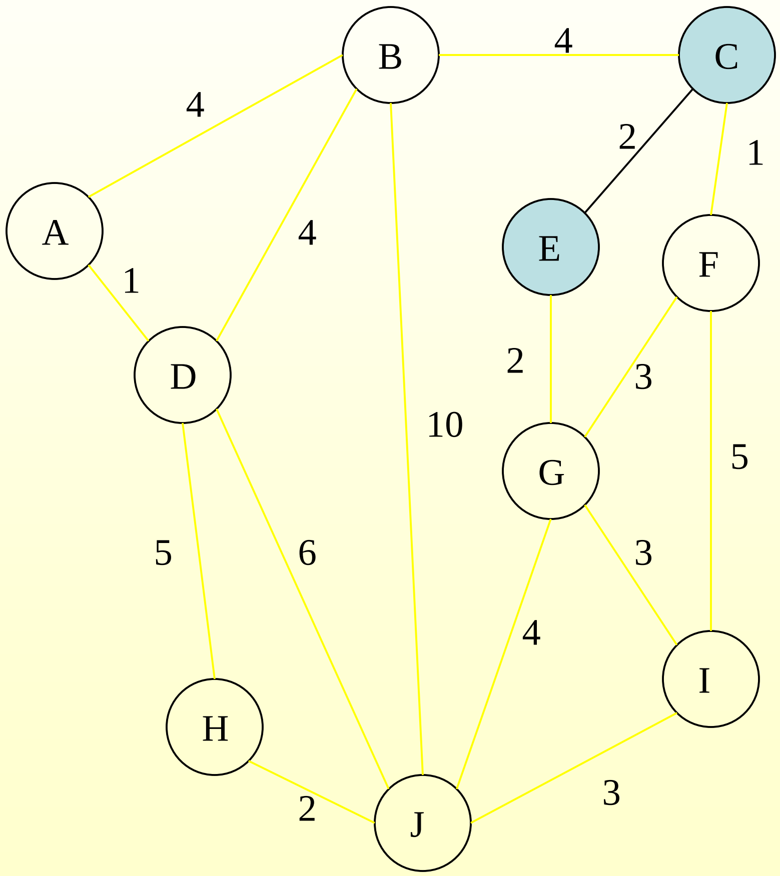
Round 1



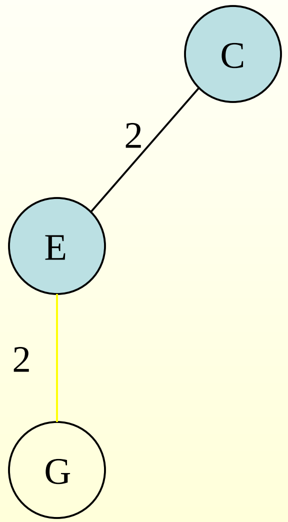
Tree E



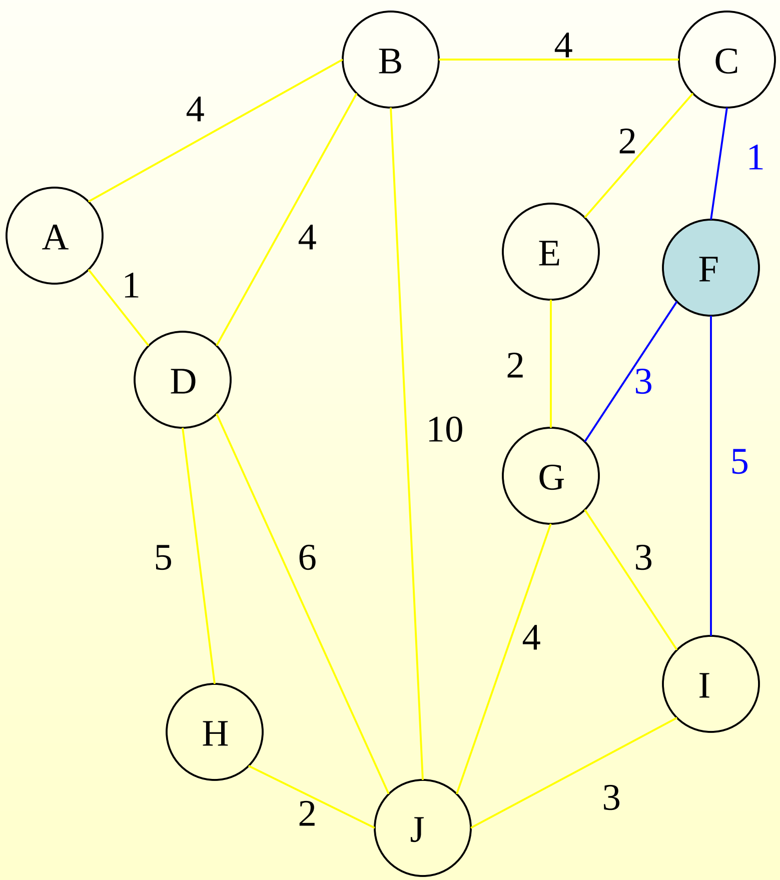
Round 1



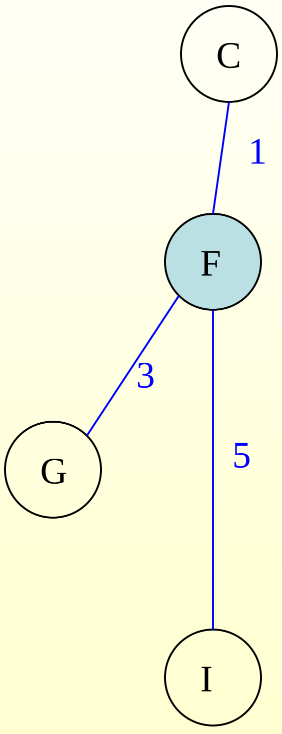
Edge E-C



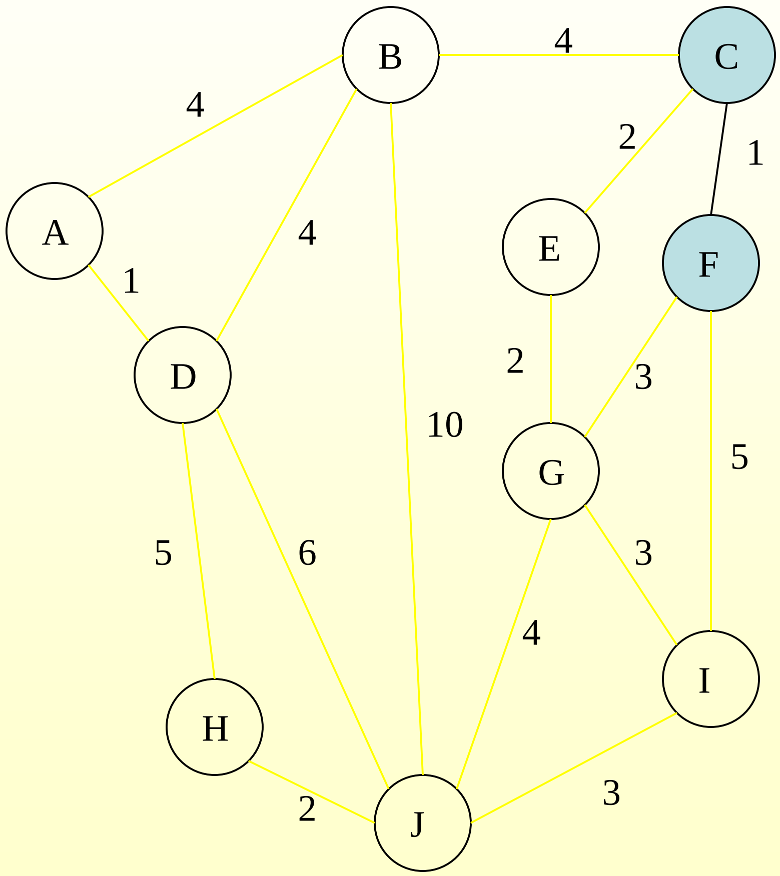
Round 1



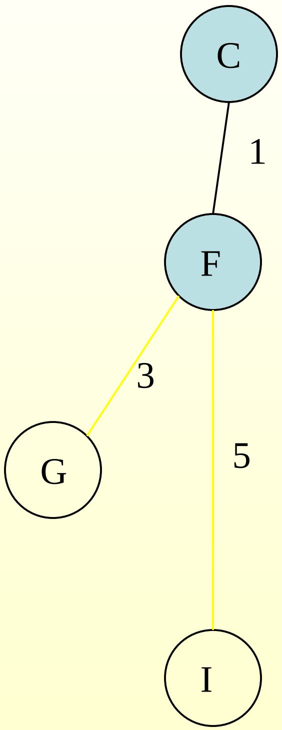
Tree F



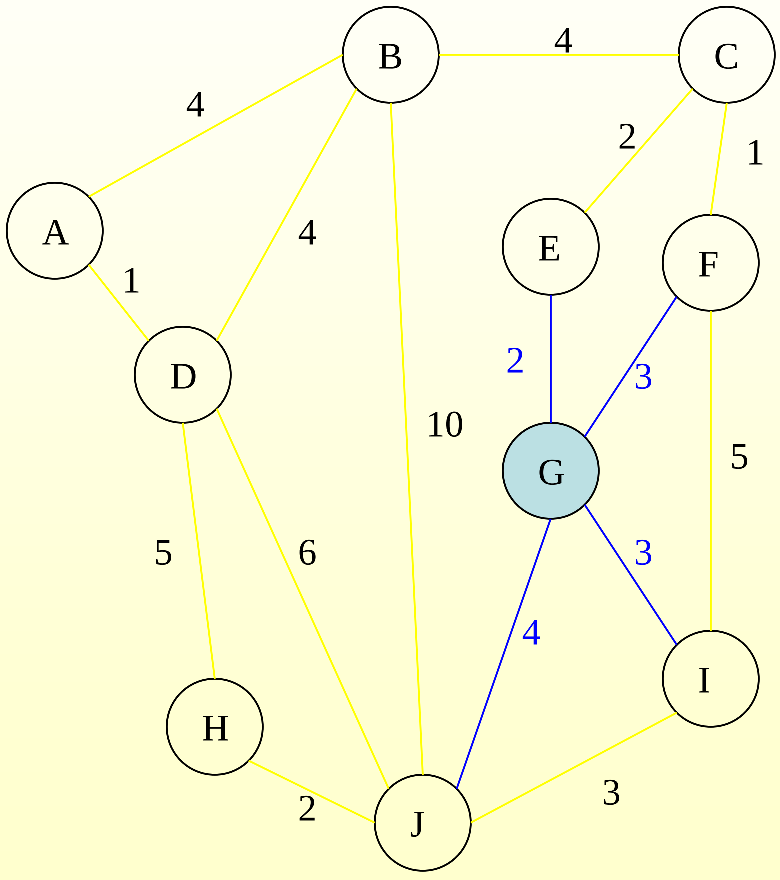
Round 1



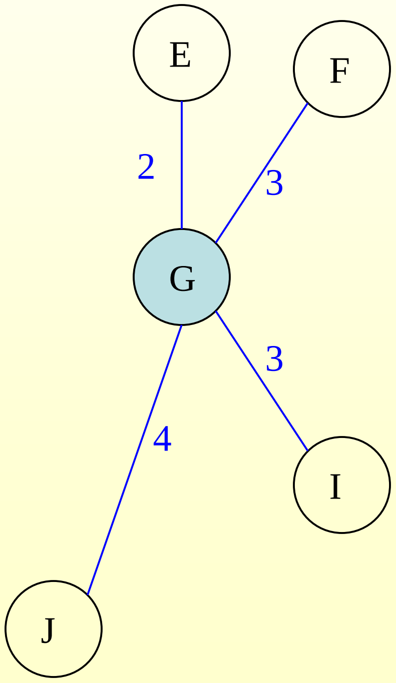
Edge F-C



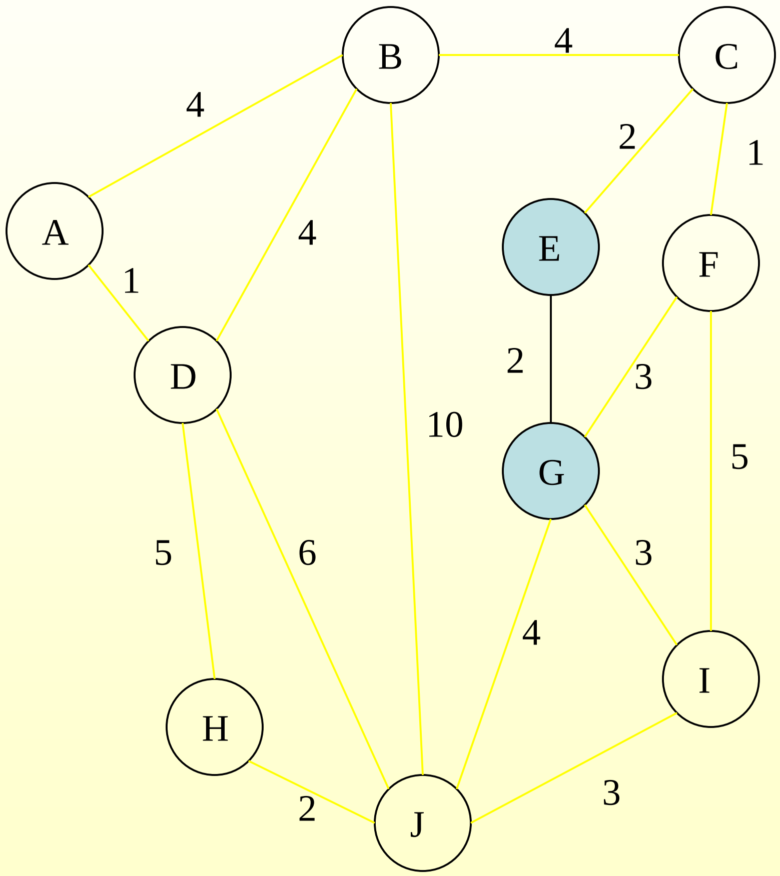
Round 1



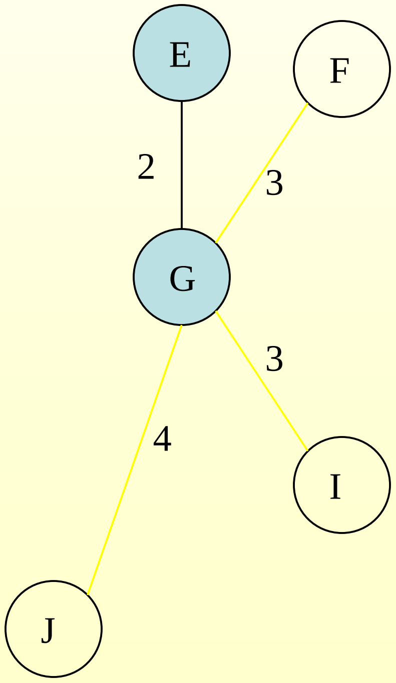
Tree G



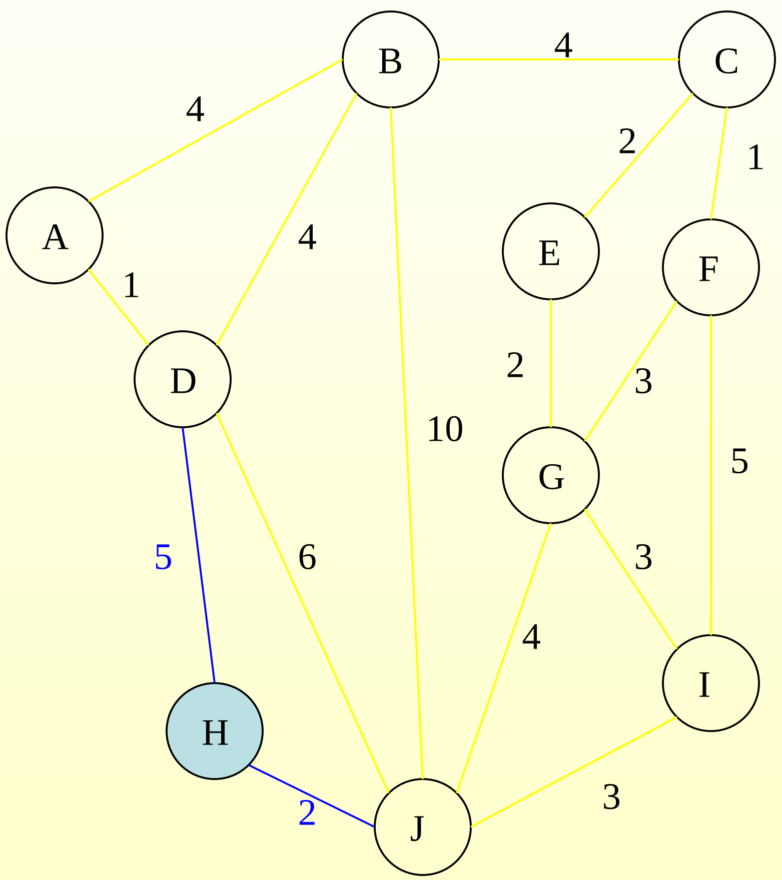
Round 1



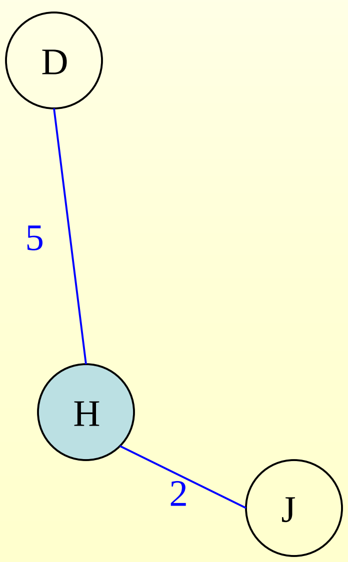
Edge G-E



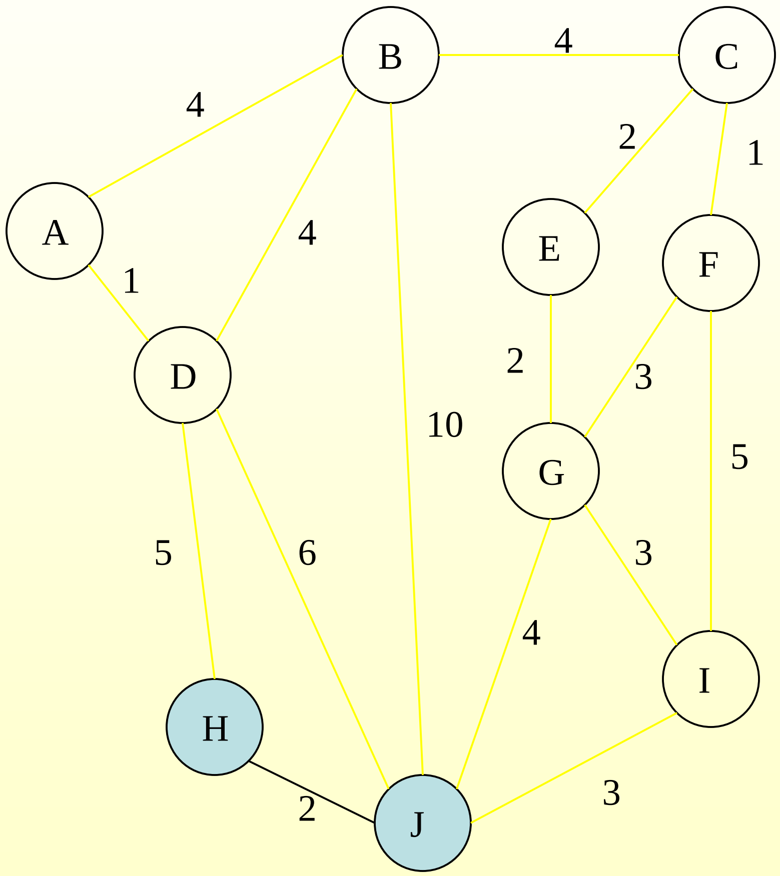
Round 1



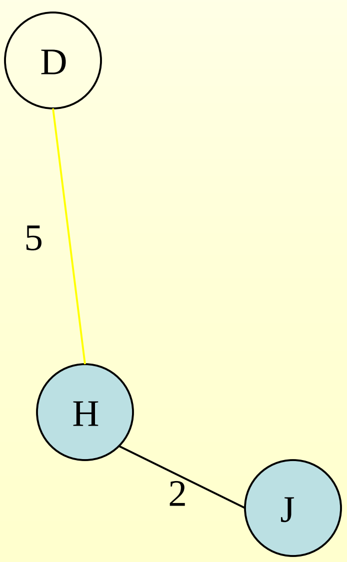
Tree H



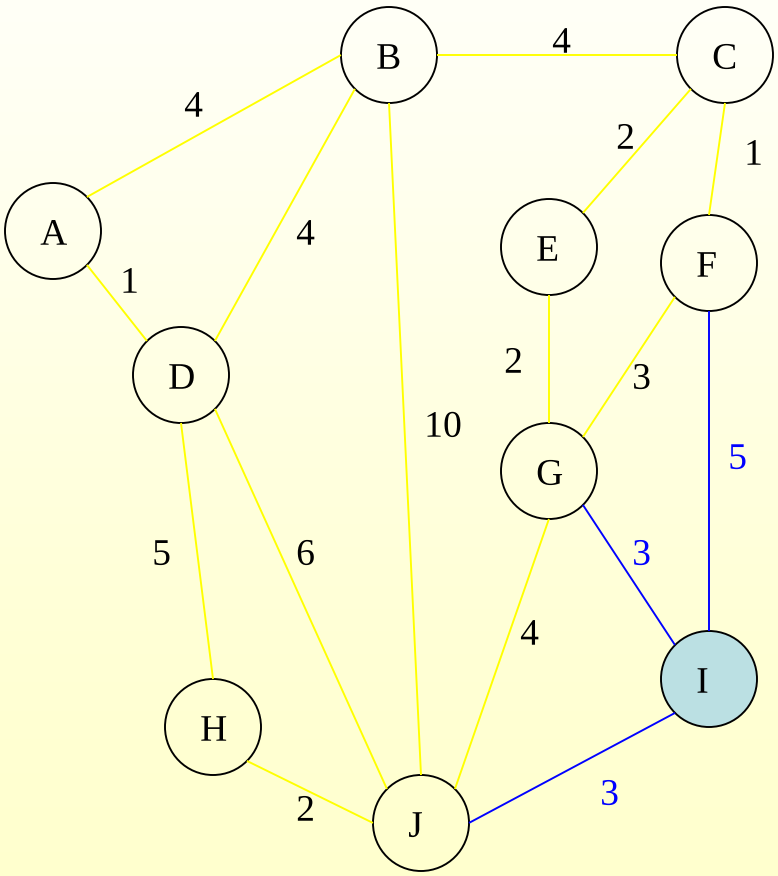
Round 1



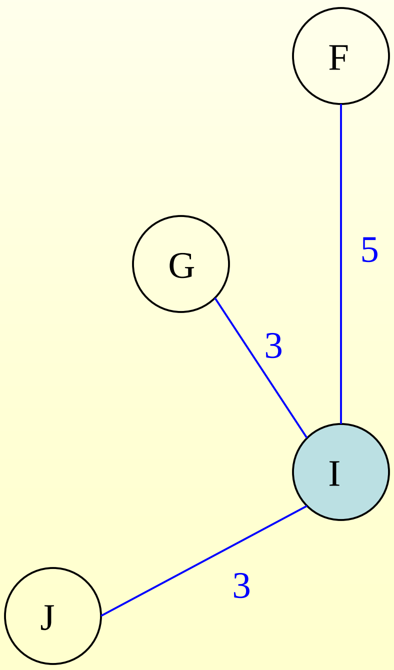
Edge H-J



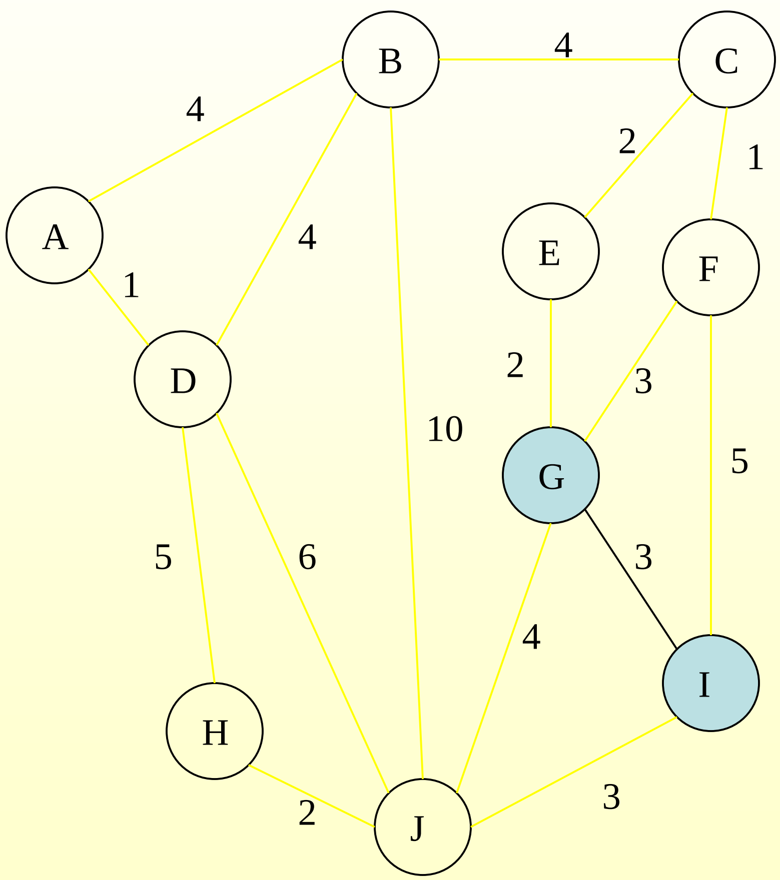
Round 1



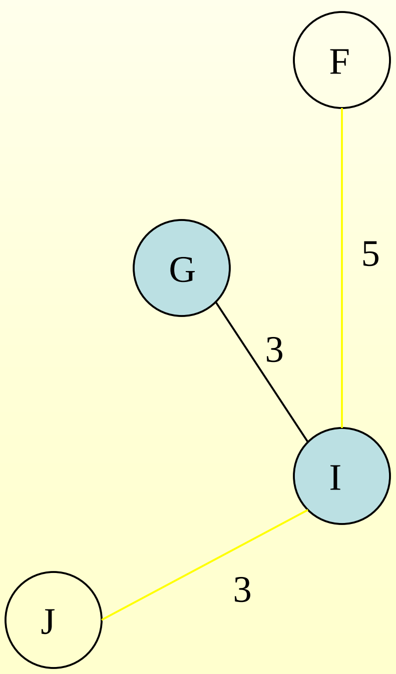
Tree I



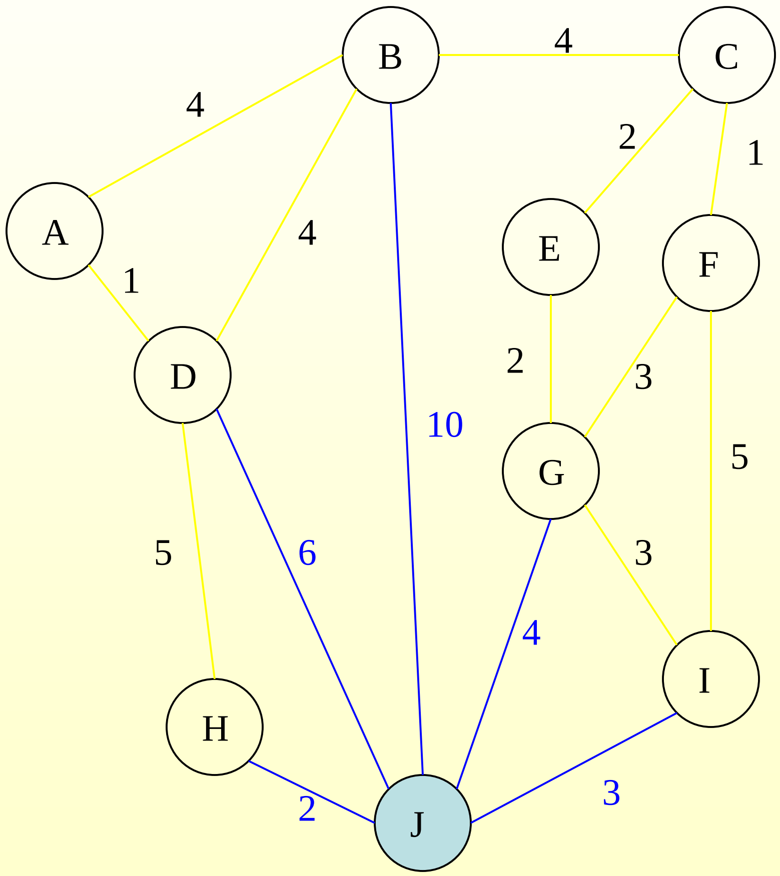
Round 1



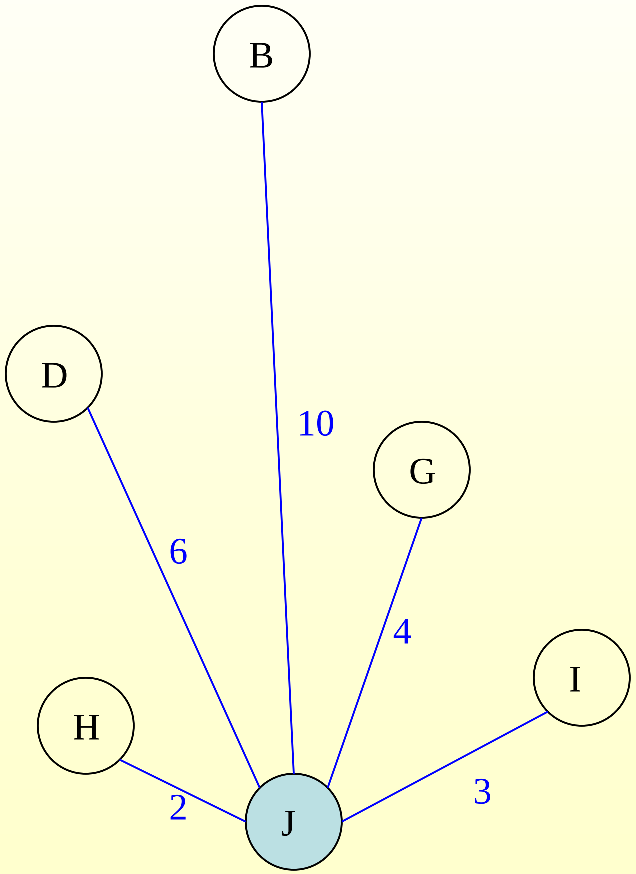
Edge I-G



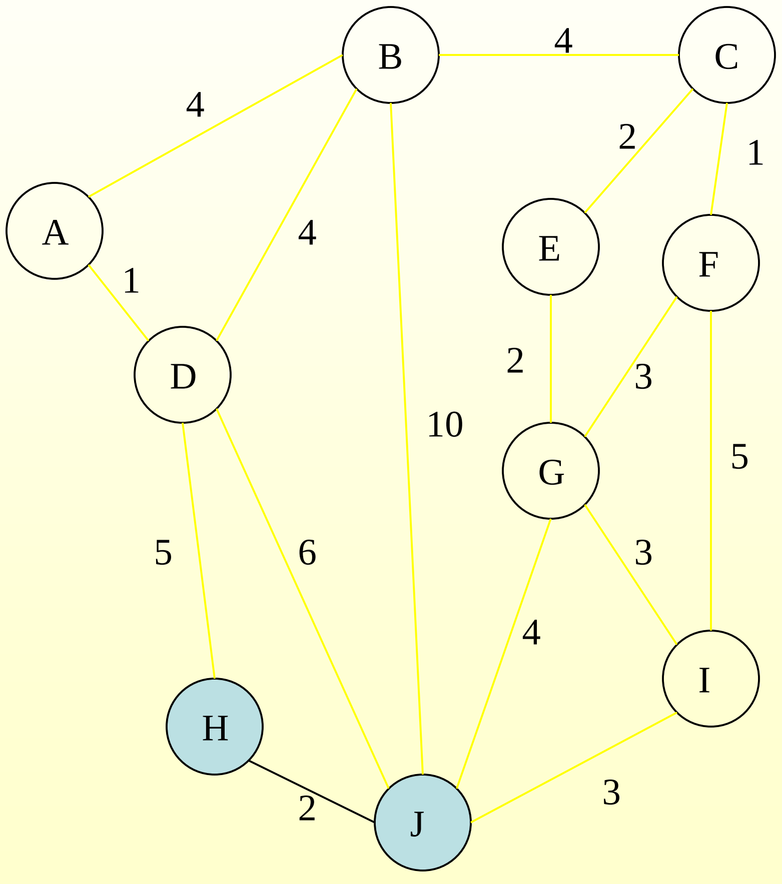
Round 1



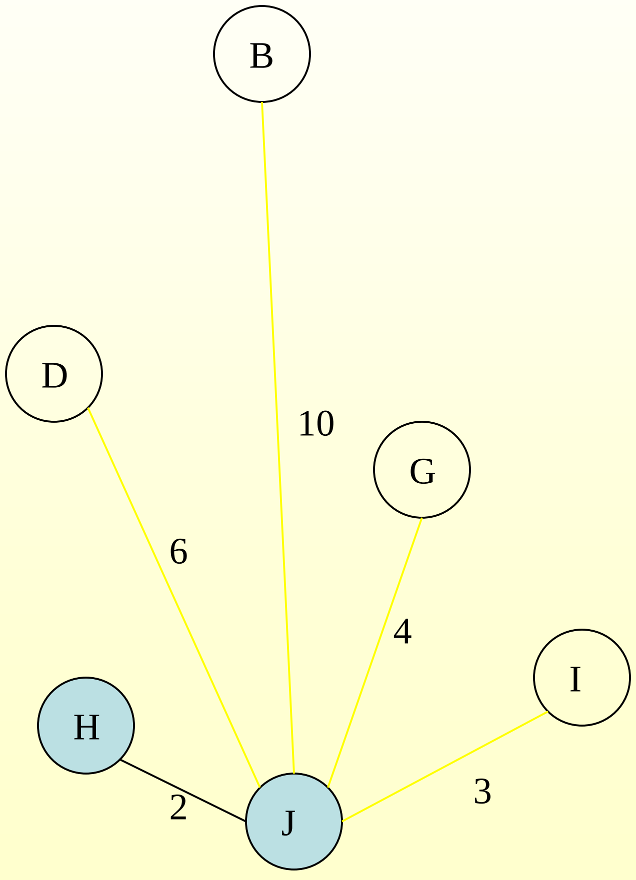
Tree J



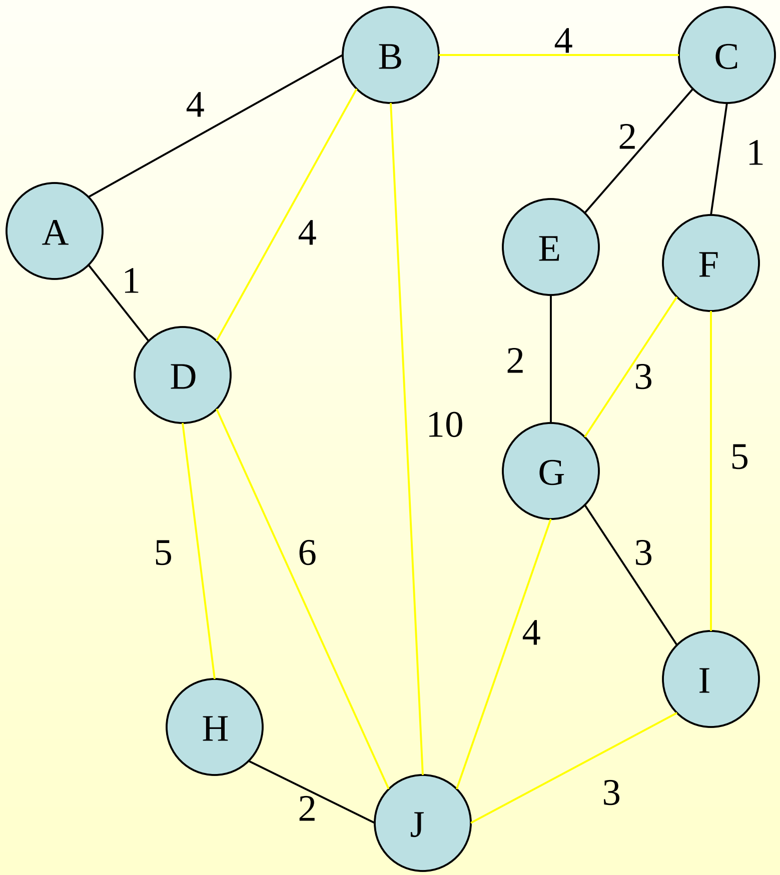
Round 1



Edge J-H



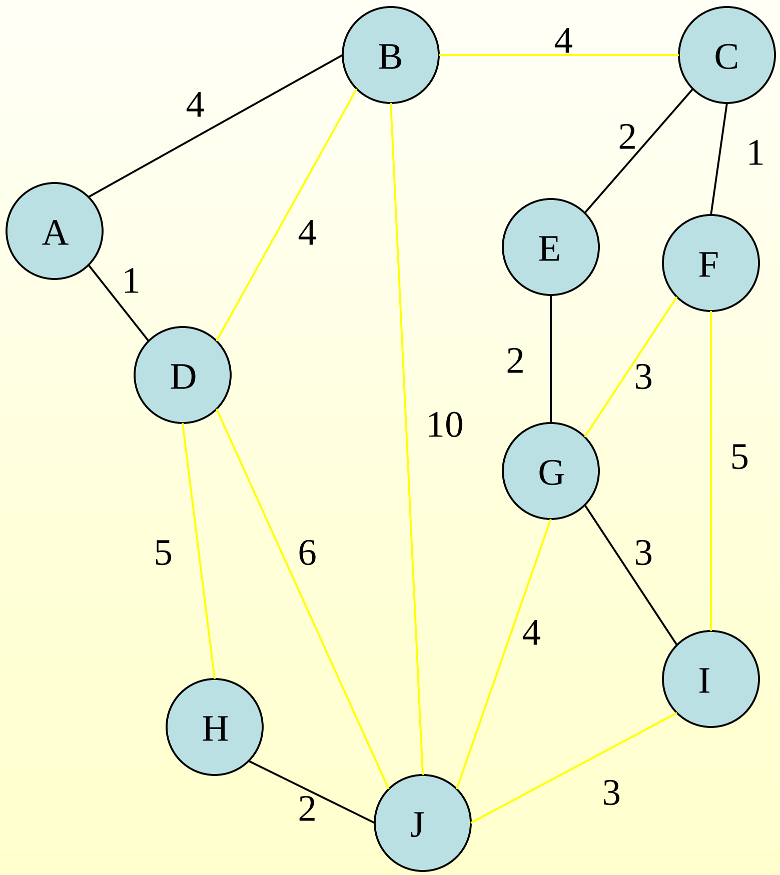
Round 1 Ends - Add Edges



List of Edges to Add

- A-D
- B-A
- C-F
- D-A
- E-C
- F-C
- G-E
- H-J
- I-G
- J-H

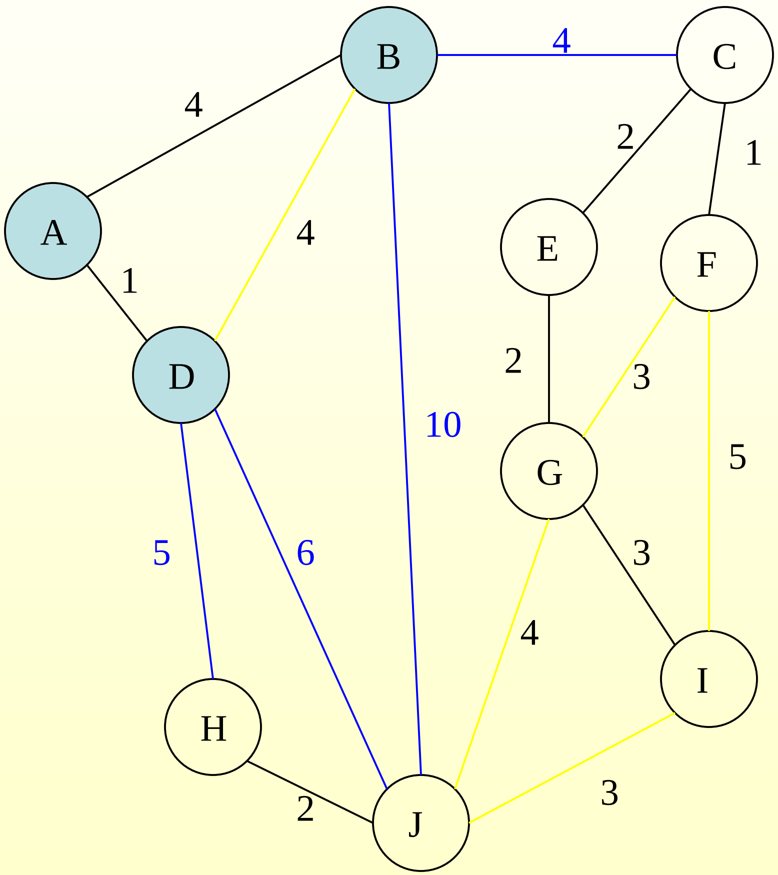
Trees of the Graph at Beginning of Round 2



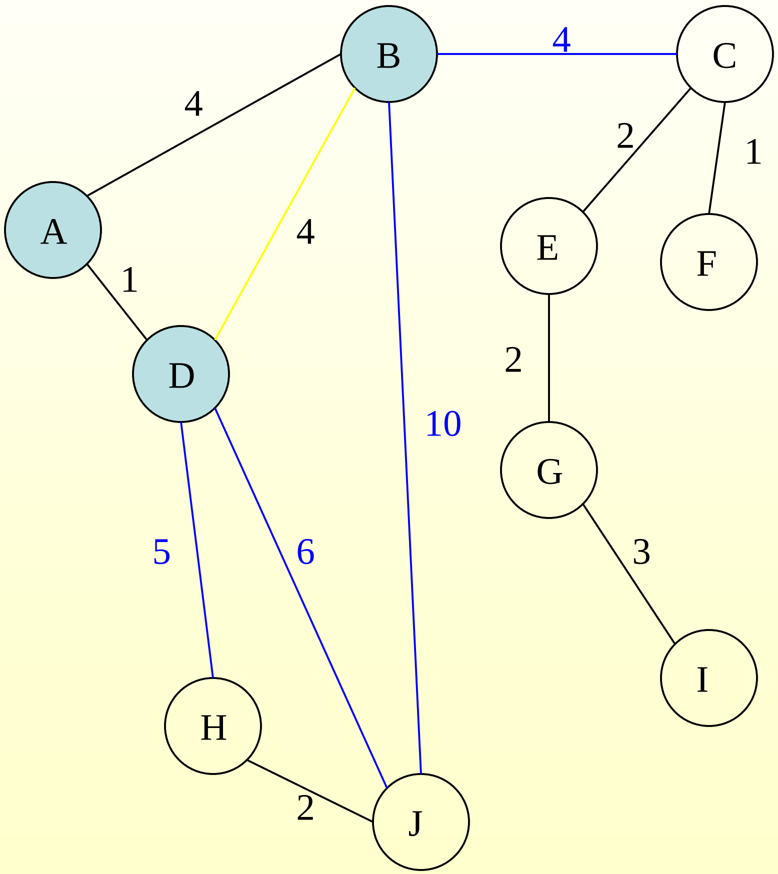
List of Trees

- D-A-B
- F-C-E-G-I
- H-J

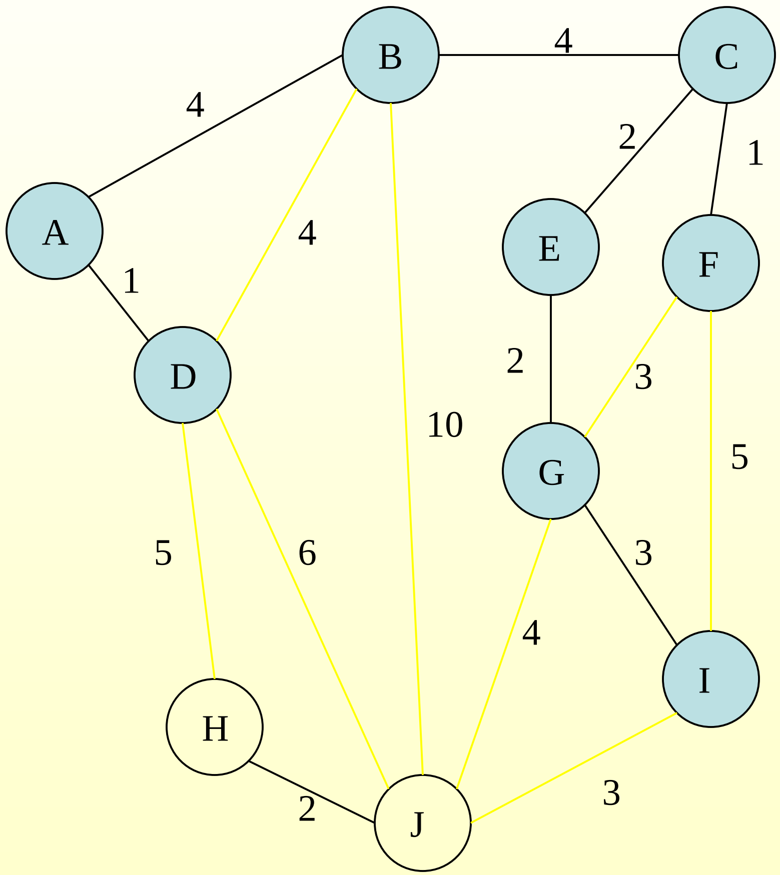
Round 2



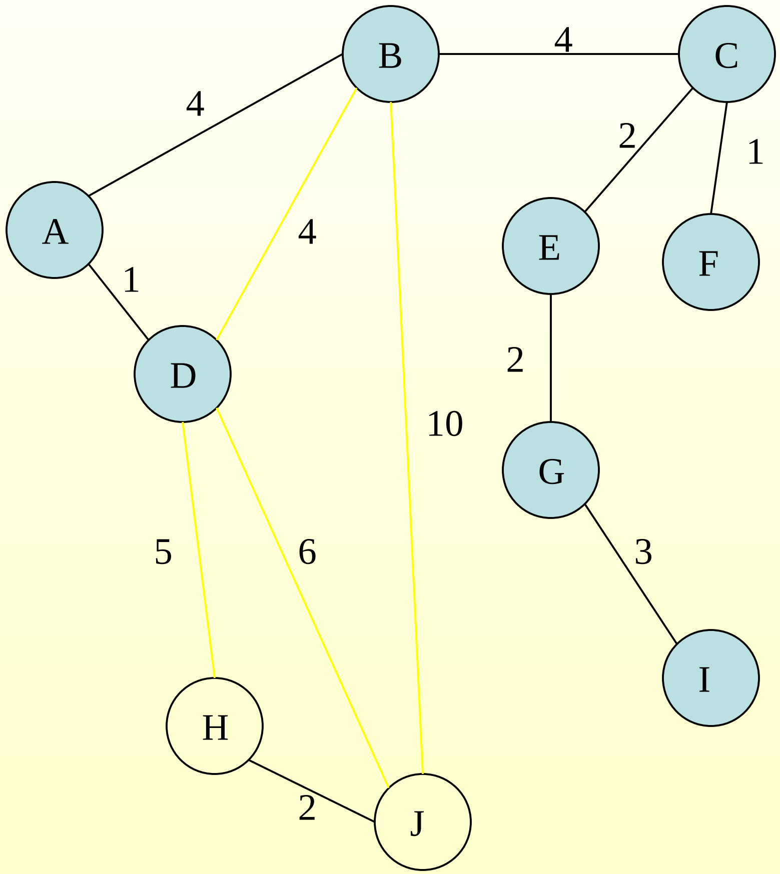
Tree D-A-B



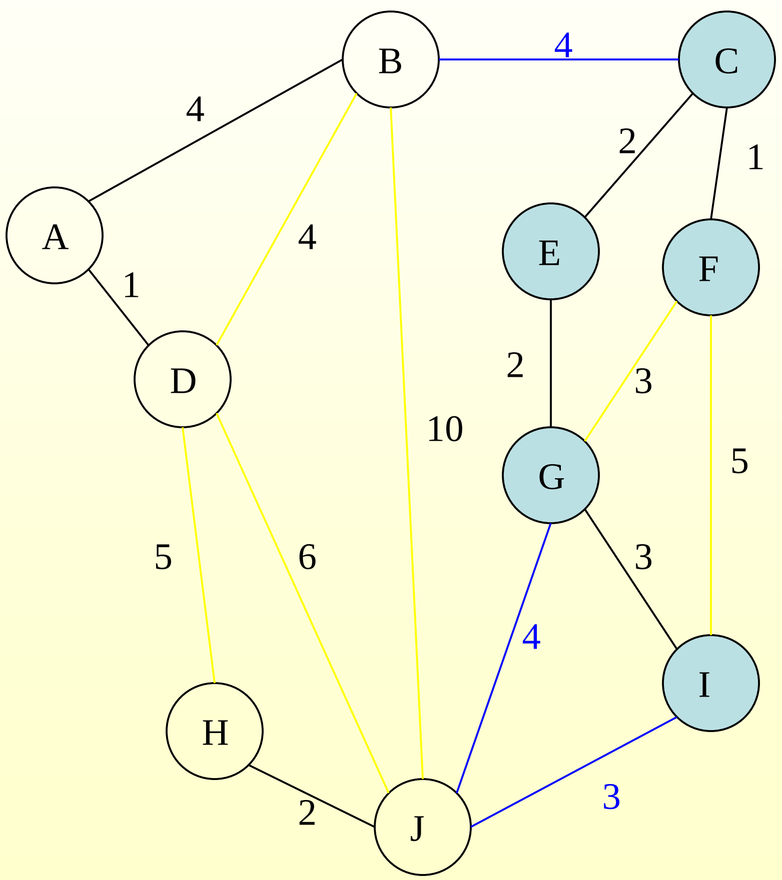
Round 2



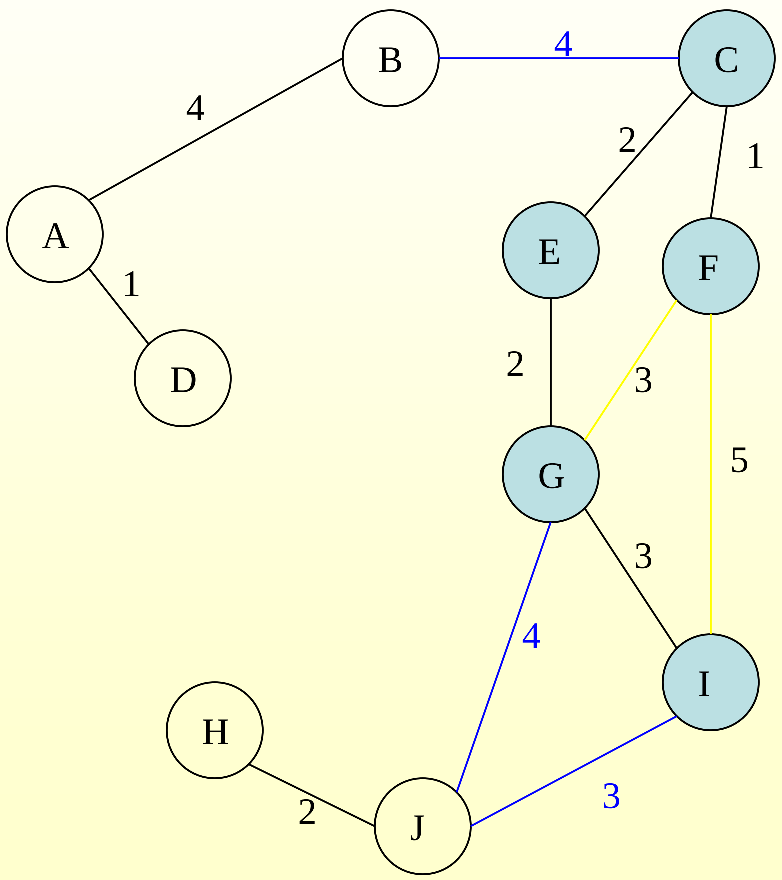
Edge B-C



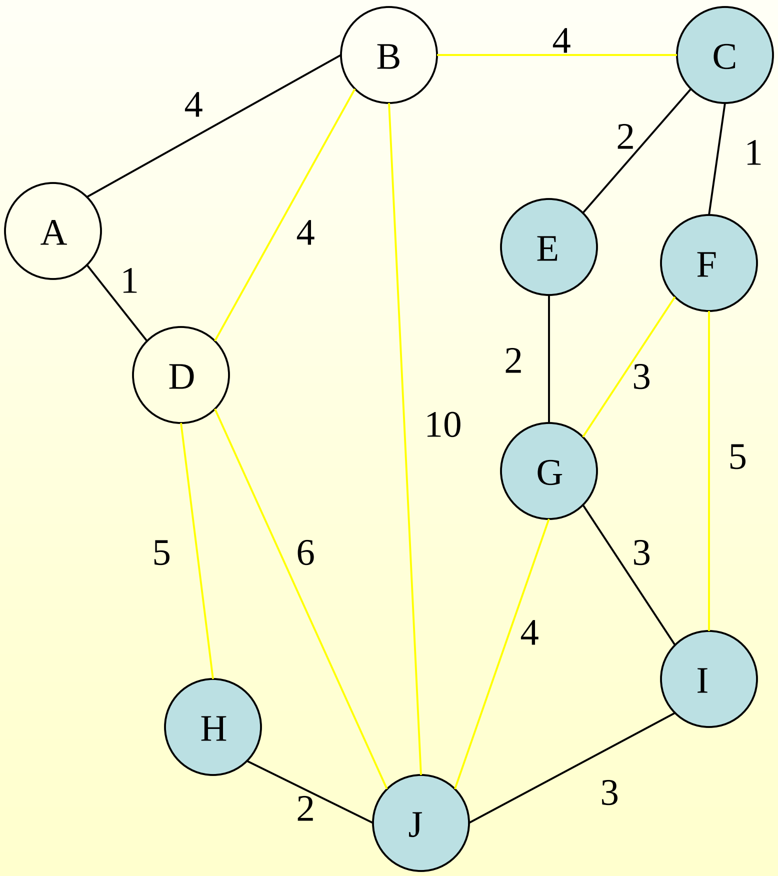
Round 2



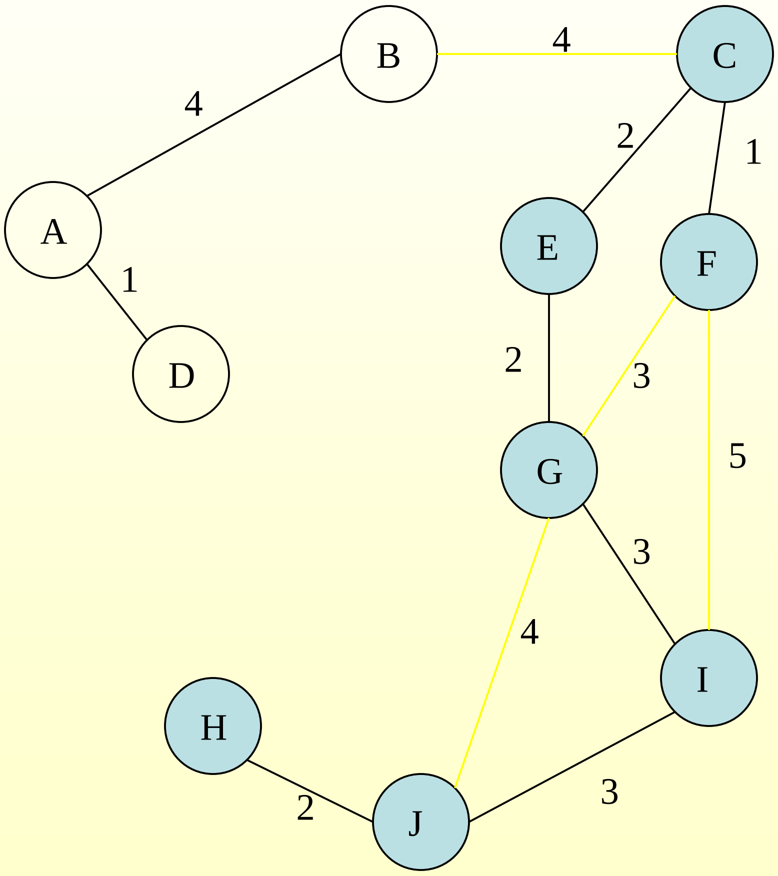
Tree F-C-E-G-I



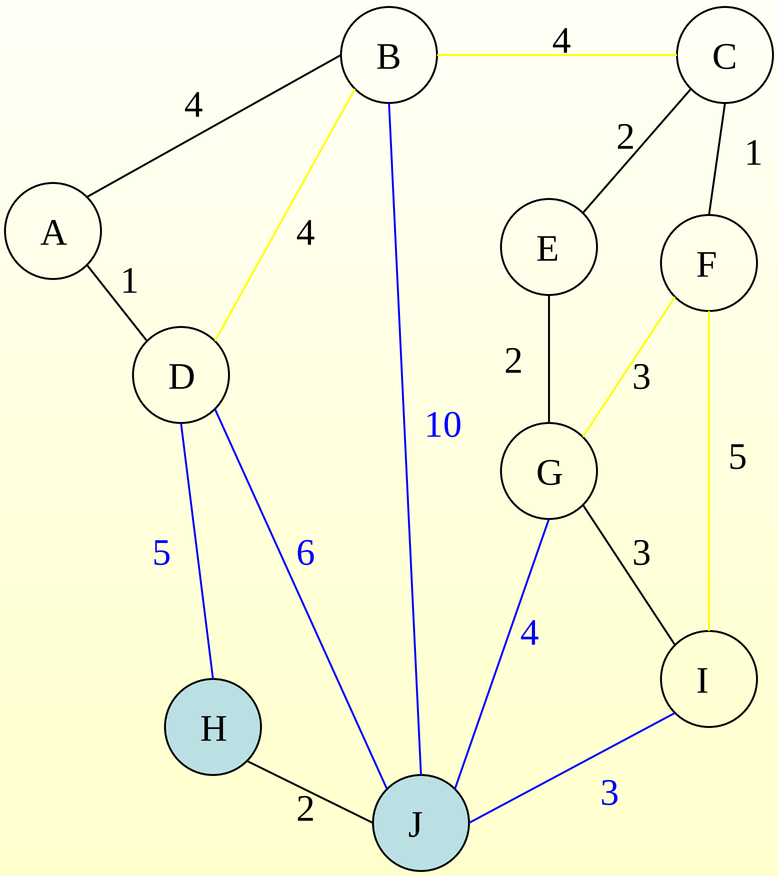
Round 2



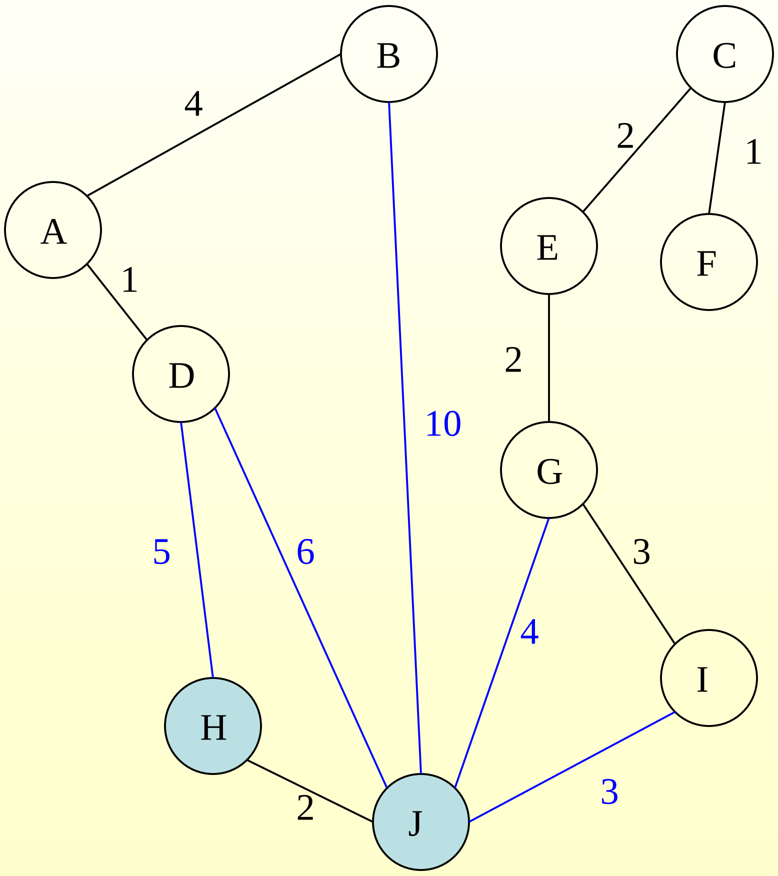
Edge I-J



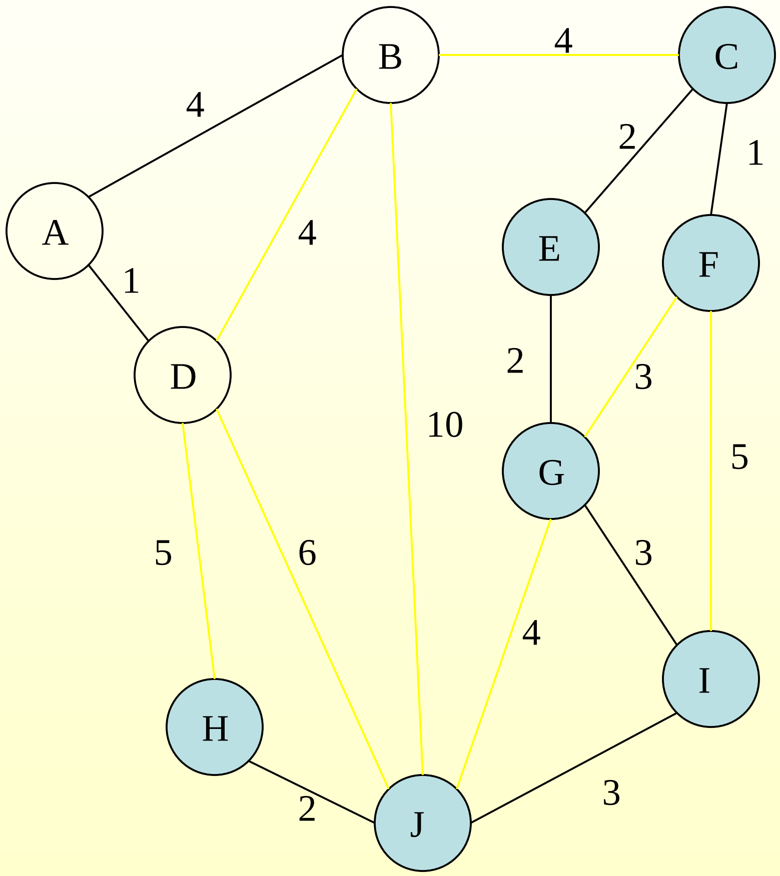
Round 2



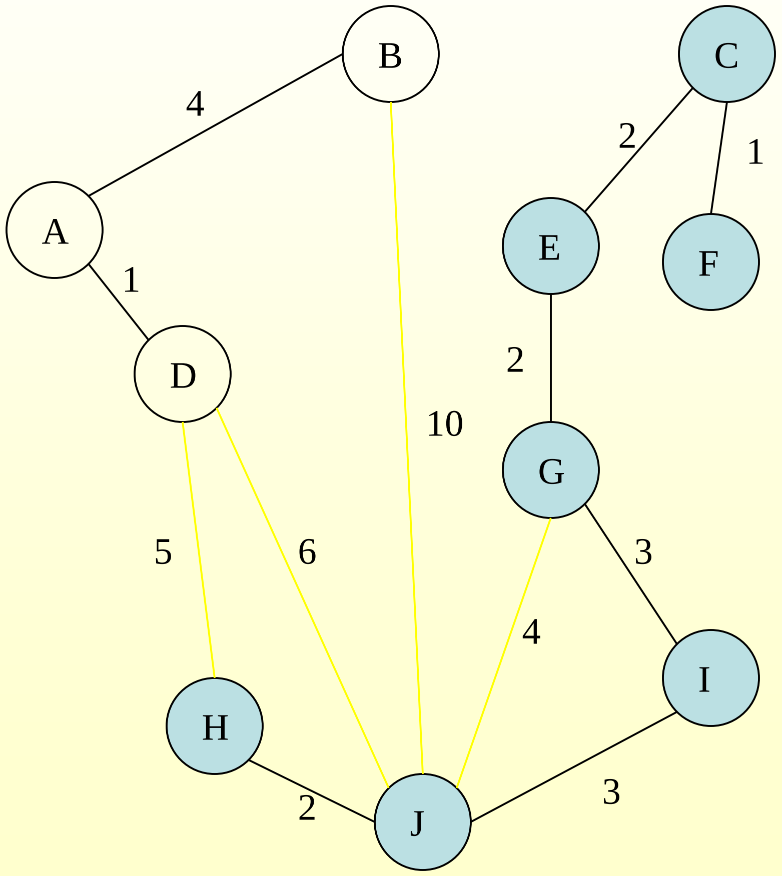
Tree H-J



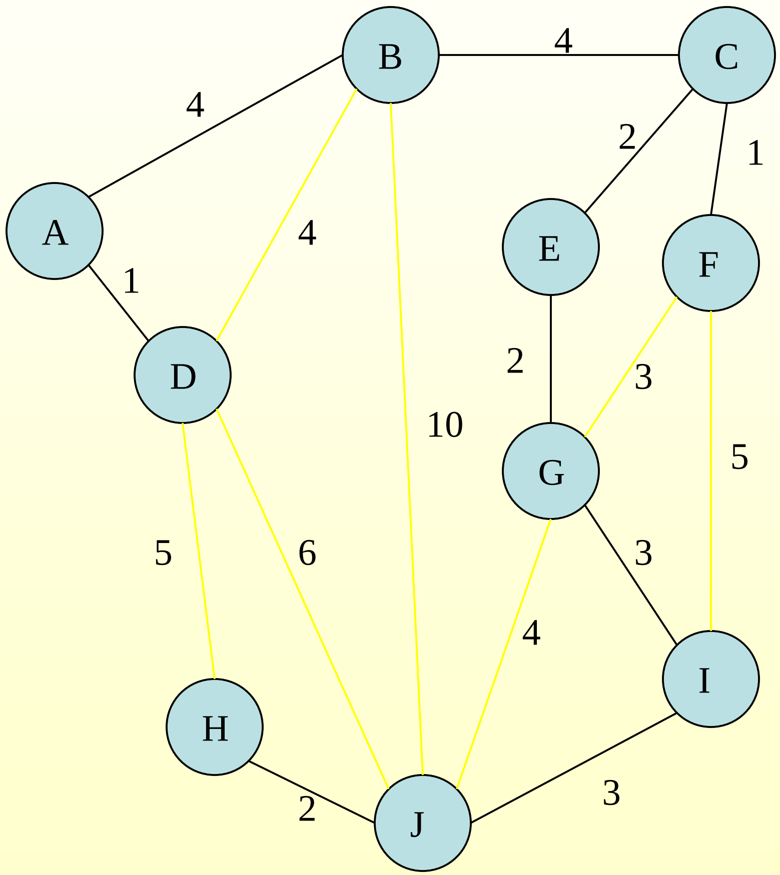
Round 2



Edge J-I



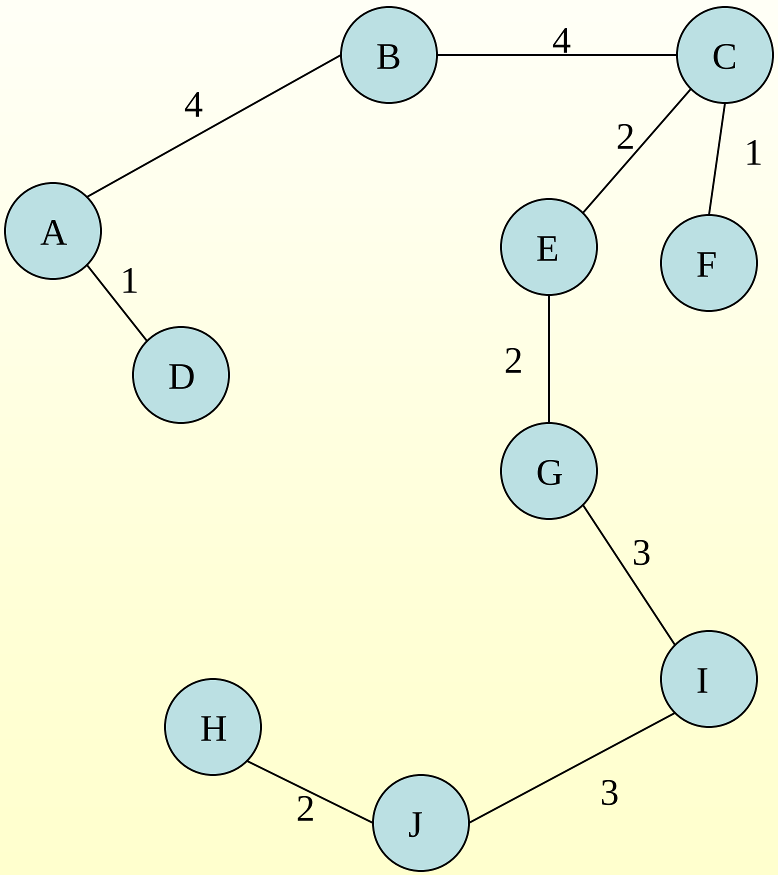
Round 2 Ends - Add Edges



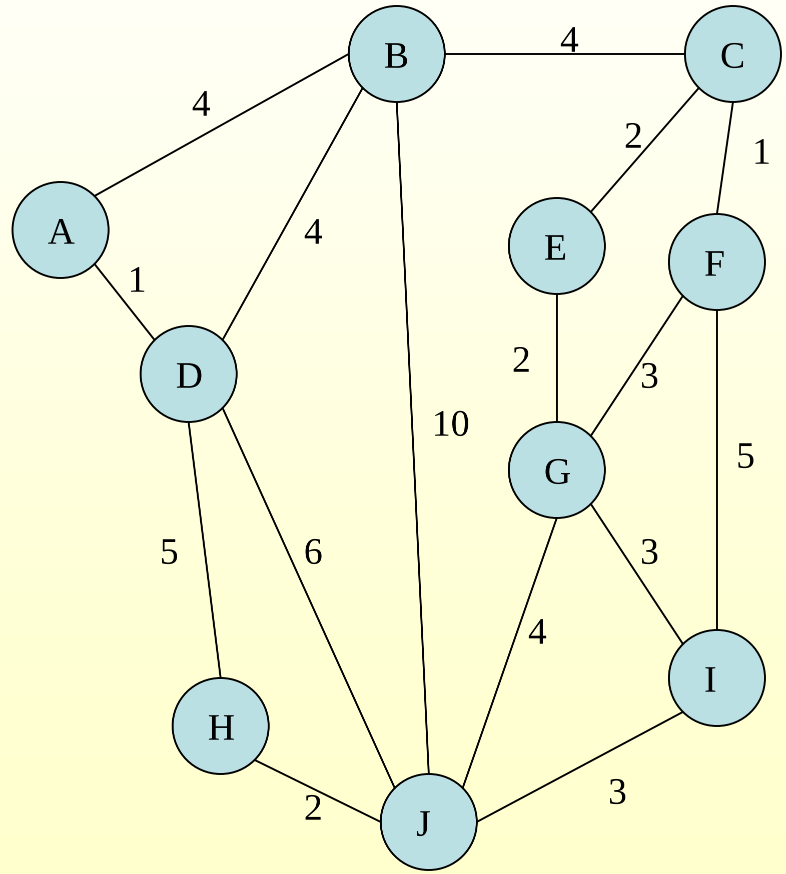
List of Edges to Add

- B-C
- I-J
- J-I

Minimum Spanning Tree



Complete Graph



Clustering

- Clustering. Given a set U of n objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

photos, documents. micro-organisms

- Distance function. Numeric value specifying "closeness" of two objects.

number of corresponding pixels whose intensities differ by some threshold

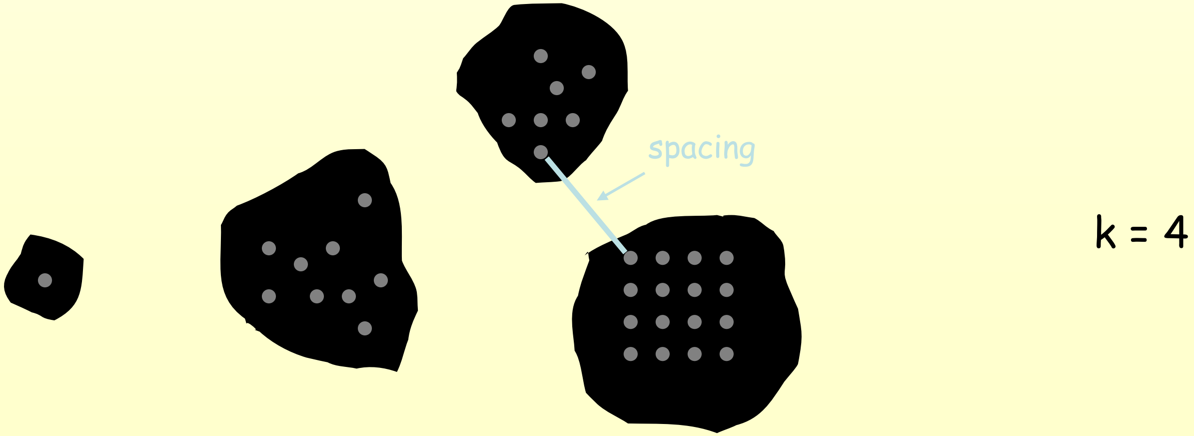
- Fundamental problem. Divide into clusters so that points in different clusters are far apart.
  - Routing in mobile ad hoc networks.
  - Identify patterns in gene expression.
  - Document categorization for web search.
  - Similarity searching in medical image databases
  - Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Clustering of Maximum Spacing

- k-clustering. Divide objects into k non-empty groups.

- Distance function. Assume it satisfies several natural properties.
  - $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
  - $d(p_i, p_j) \geq 0$ (nonnegativity)
  - $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

- Spacing. Min distance between any pair of points in different clusters.
- Clustering of maximum spacing. Given an integer k, find a k-clustering of maximum spacing.



Greedy Clustering Algorithm

- Single-linkage k-clustering algorithm.
  - Form a graph on the vertex set U, corresponding to n clusters.
  - Find the closest pair of objects such that each object is in a different cluster, and add an edge between them.
  - Repeat n-k times until there are exactly k clusters.

- Key observation. This procedure is precisely Kruskal's algorithm
(except we stop when there are k connected components).

- Remark. Equivalent to finding an MST and deleting the k-1 most expensive edges.