Minimum Spanning Trees

1. Prim's Algorithm
   - Similar to Dijkstra's Algorithm?
2. Kruskal's Algorithm
   - Focuses on edges, rather than nodes

Definition

- A **Minimum Spanning Tree (MST)** is a subgraph of an undirected graph such that
  - the subgraph spans (includes) all nodes,
  - is connected,
  - is acyclic, and
  - has minimum total edge weight

Algorithm Characteristics

- Both Prim's and Kruskal's Algorithms work with undirected graphs
- Both work with **weighted and unweighted** graphs but are more interesting when edges are weighted
- Both are greedy algorithms that produce optimal solutions

Why undirected?

- A spanning tree is defined only for **undirected graphs** because it is a subset of edges that connects **all vertices without cycles**, and direction does not make sense in this context.
- In a **directed graph**, the equivalent concept is called an **arborescence*** or a **minimum spanning arborescence** (handled by algorithms like Edmonds' algorithm), not an MST.

- In graph theory, an **arborescence** is a directed graph where there exists a vertex *r* (called the *root*) such that, for any other vertex *v*, there is exactly one directed walk from *r* to *v* (noting that the root *r* is unique).

Prim's Algorithm

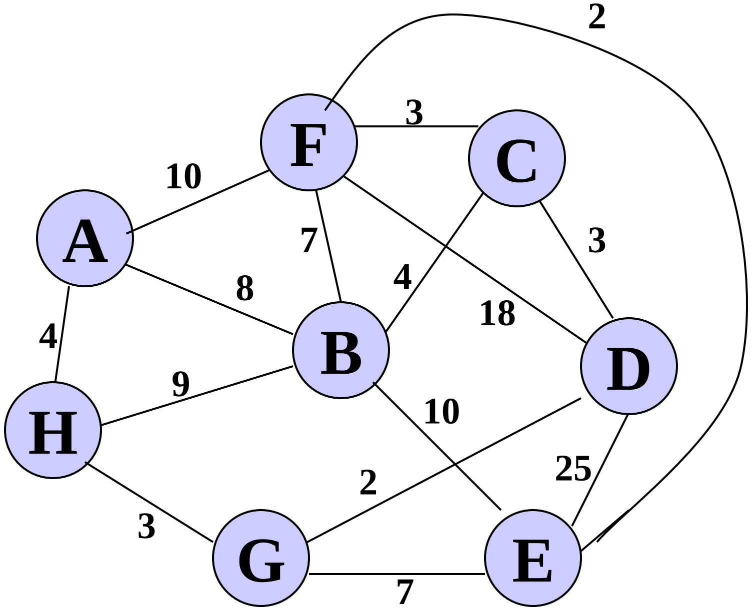
- Similar(not same) to Dijkstra's Algorithm except that it records edge weights, not path lengths

- Time complexity : $O(n^2)$, $n = |V|$.

Algorithm Prim's

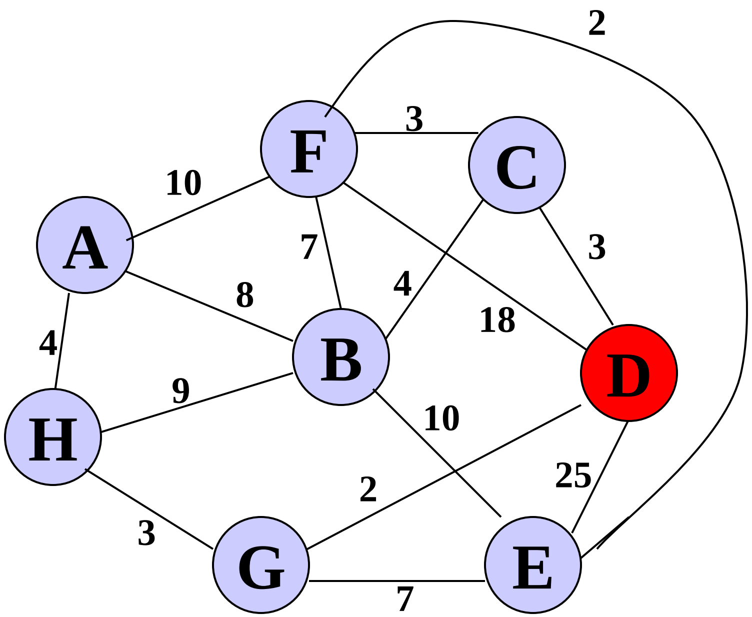
- **Input:** Two sets of nodes from the graph:
    1. **in the** current tree (*T*),
    2. not-yet-in the current tree (*U*).
- **Step1:** Start with any node and put it in *T*.
- **Step2:** At every stage pick up a node *u* from *U* such that it has the minimum distance-edge (d_v) from any node in *T*.
- **Step3:** Take it off from *U*, put it in *T*, and update min distances (direct arc from *u* or any node in *T*, not the shortest path from a source) of all nodes in *U* from this node *u*.
- **Step 4:** If $U = \varnothing$, stop; otherwise, go to **Step2**

Walk-Through



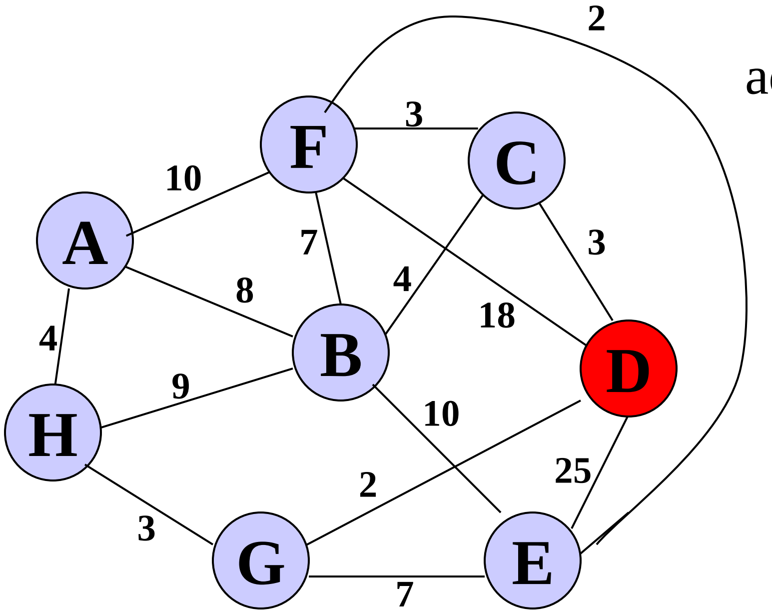
Initialize array

	T	d_v	p_v	U
A	F	∞	–	A
B	F	∞	–	B
C	F	∞	–	C
D	F	∞	–	D
E	F	∞	–	E
F	F	∞	–	F
G	F	∞	–	G
H	F	∞	–	H



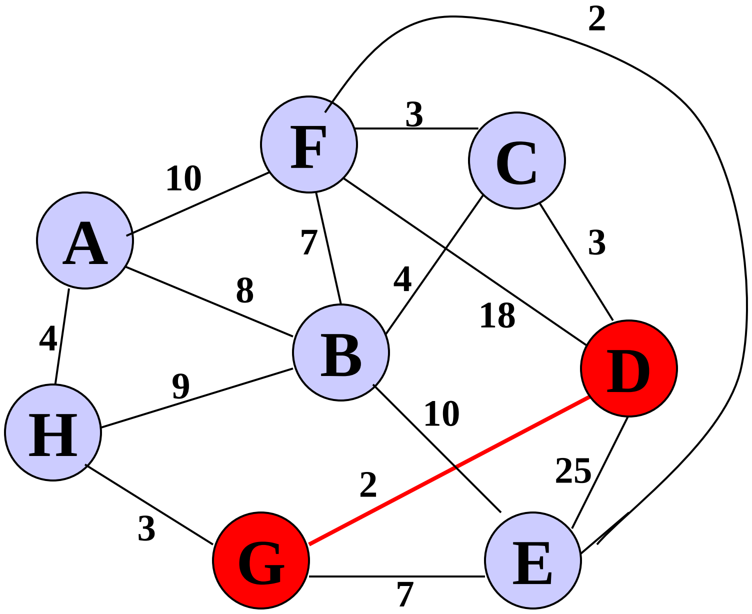
Start with any node, say D

	T	d_v	p_v	*U*
A				A
B				B
C				C
D	T	0	–	
E				E
F				F
G				G
H				H



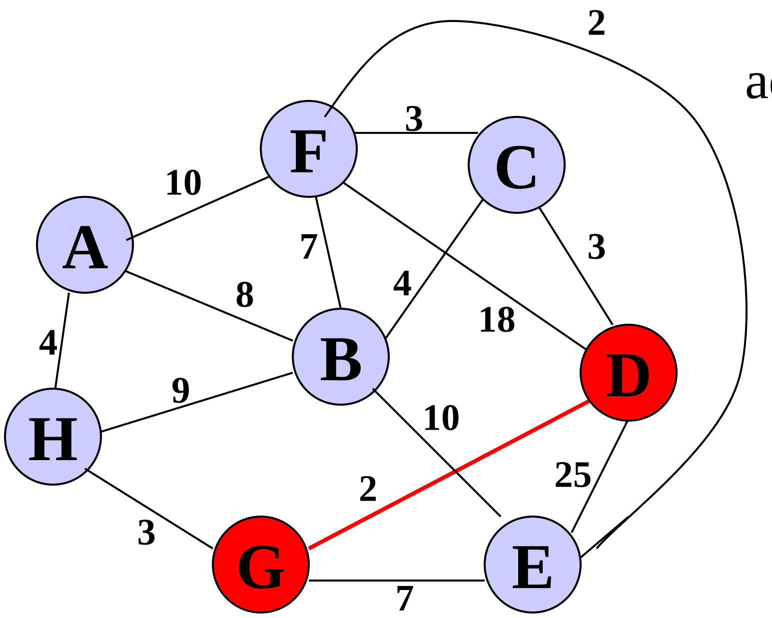
Update distances of adjacent, unselected nodes

	T	d_v	p_v	U
A				A
B				B
C		3	D	C
D	T	0	–	
E		25	D	E
F		18	D	F
G		2	D	G
H				H



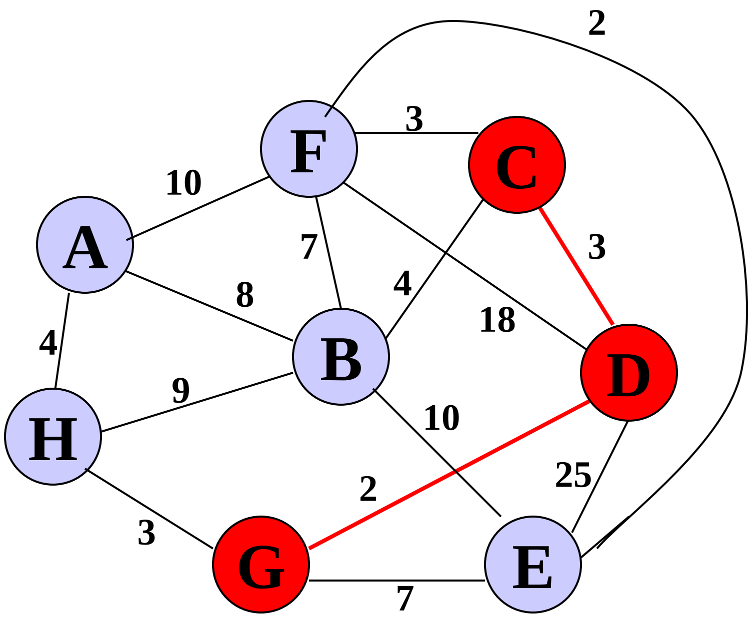
Select node with minimum distance

	T	d_v	p_v	*U*
A				A
B				B
C		3	D	C
D	T	0	–	
E		25	D	E
F		18	D	F
G	T	2	D	
H				H



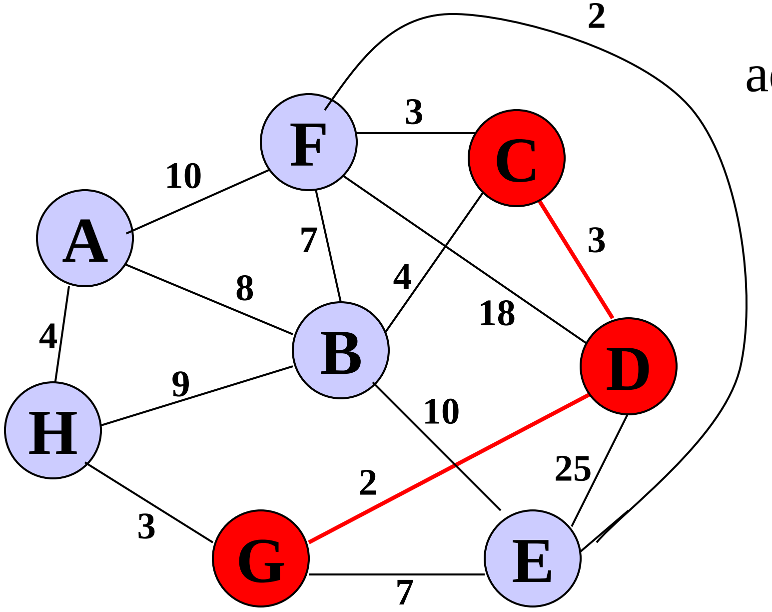
Update distances of
adjacent, unselected nodes

	T	d_v	p_v	U
A				A
B				B
C		3	D	C
D	T	0	–	
E		7	G	E
F		18	D	F
G	T	2	D	
H		3	G	H



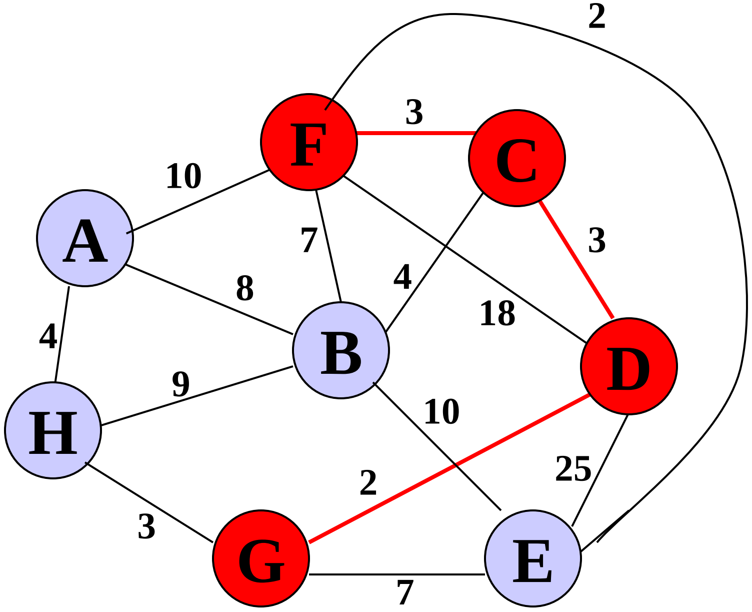
Select node with
minimum distance

	T	d_v	p_v	U
A				A
B				B
C	T	3	D	
D	T	0	–	
E		7	G	E
F		18	D	F
G	T	2	D	
H		3	G	H



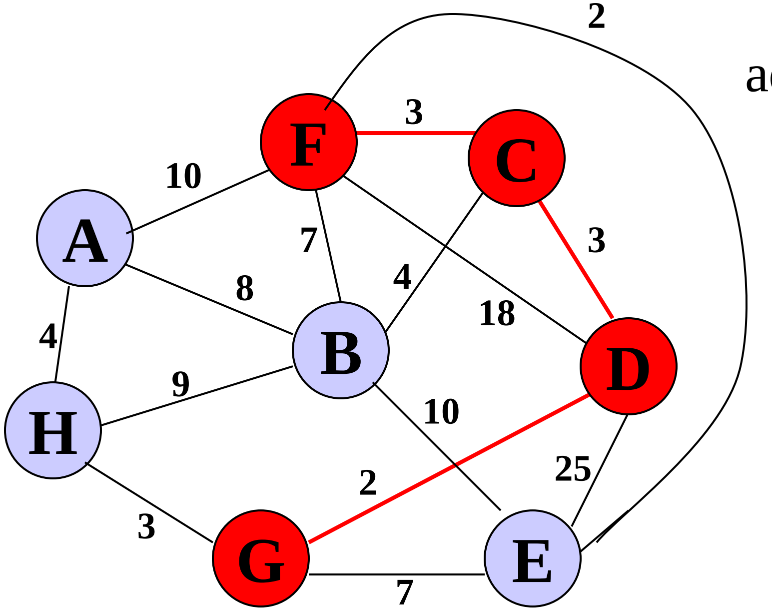
Update distances of
adjacent, unselected nodes

	T	d_v	p_v	*U*
A				A
B		4	C	B
C	T	3	D	
D	T	0	–	
E		7	G	E
F		3	C	F
G	T	2	D	
H		3	G	H



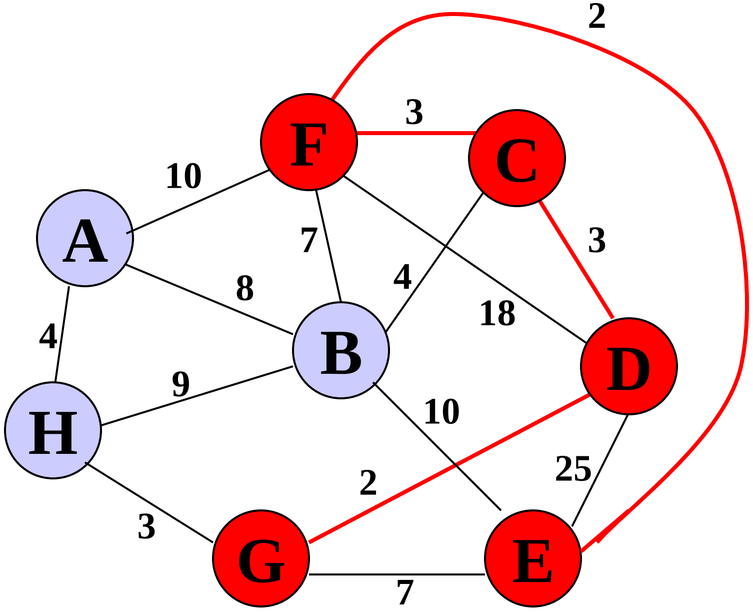
Select node with minimum distance

	T	d_v	p_v	*U*
A				A
B		4	C	B
C	T	3	D	
D	T	0	–	
E		7	G	E
F	T	3	C	
G	T	2	D	
H		3	G	H



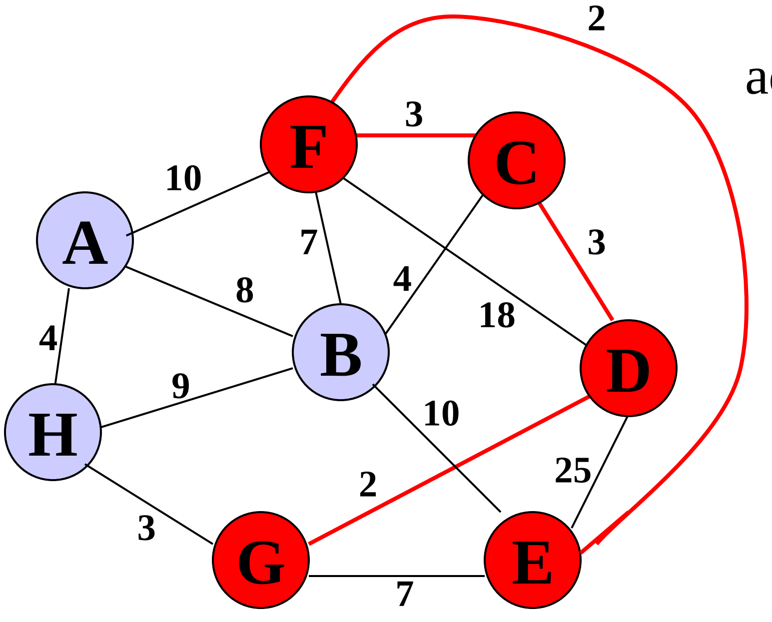
Update distances of adjacent, unselected nodes

	T	d_v	p_v	*U*
A		10	F	A
B		4	C	B
C	T	3	D	
D	T	0	–	
E		2	F	E
F	T	3	C	
G	T	2	D	
H		3	G	H



Select node with minimum distance

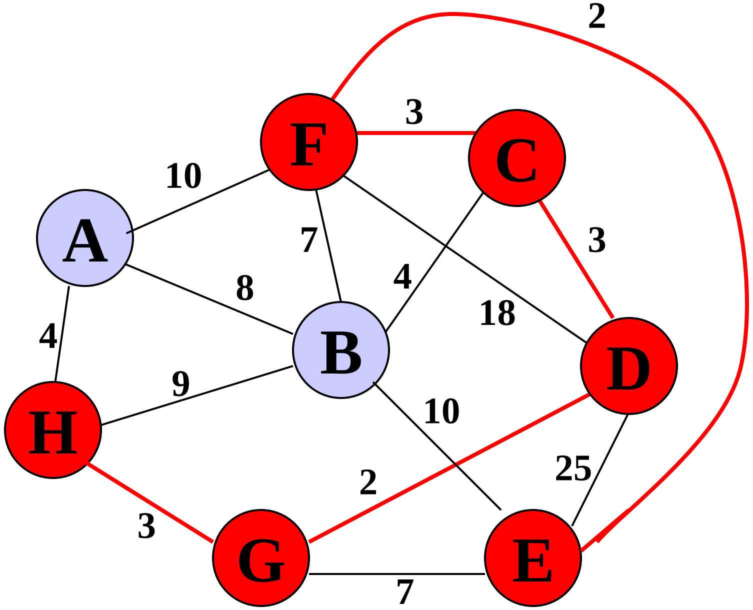
	T	d_v	p_v	U
A		10	F	A
B		4	C	B
C	T	3	D	
D	T	0	–	
E	T	2	F	
F	T	3	C	
G	T	2	D	
H		3	G	H



Update distances of adjacent, unselected nodes

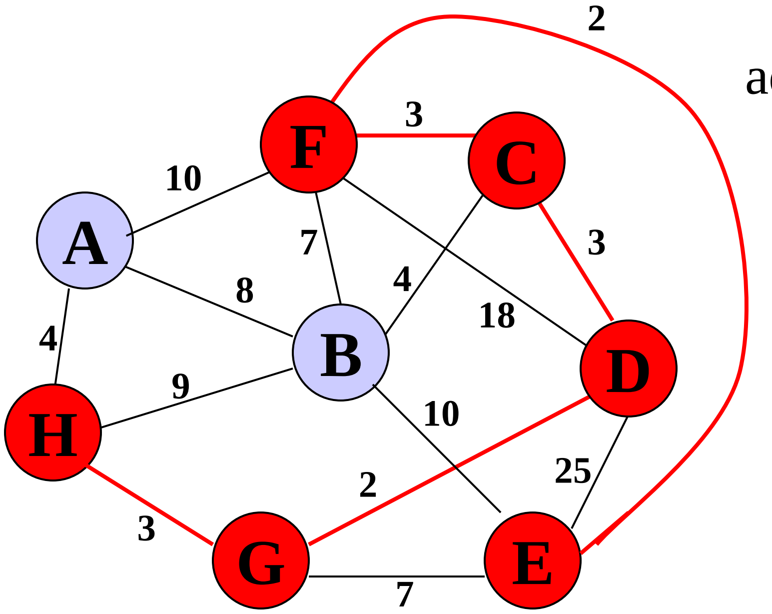
	T	d_v	p_v	U
A		10	F	A
B		4	C	B
C	T	3	D	
D	T	0	–	
E	T	2	F	
F	T	3	C	
G	T	2	D	
H		3	G	H

Table entries unchanged



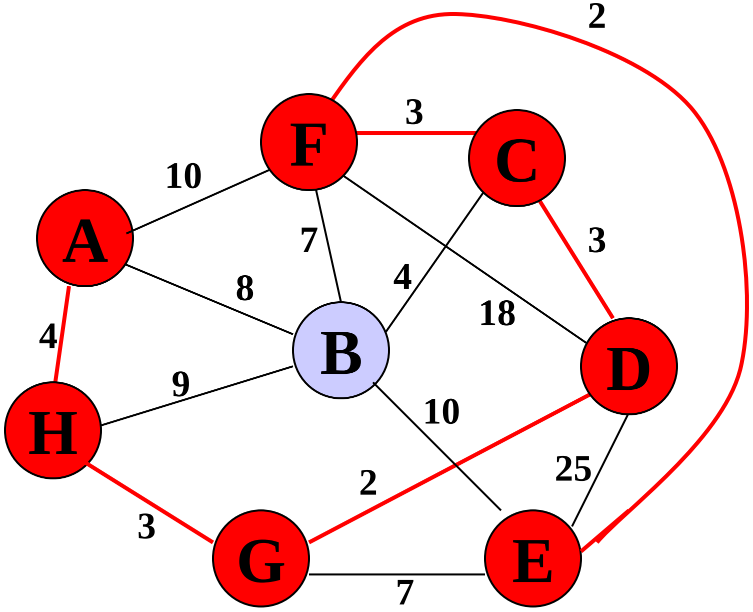
Select node with minimum distance

	T	d_v	p_v	*U*
A		10	F	A
B		4	C	B
C	T	3	D	
D	T	0	–	
E	T	2	F	
F	T	3	C	
G	T	2	D	
H	T	3	G	



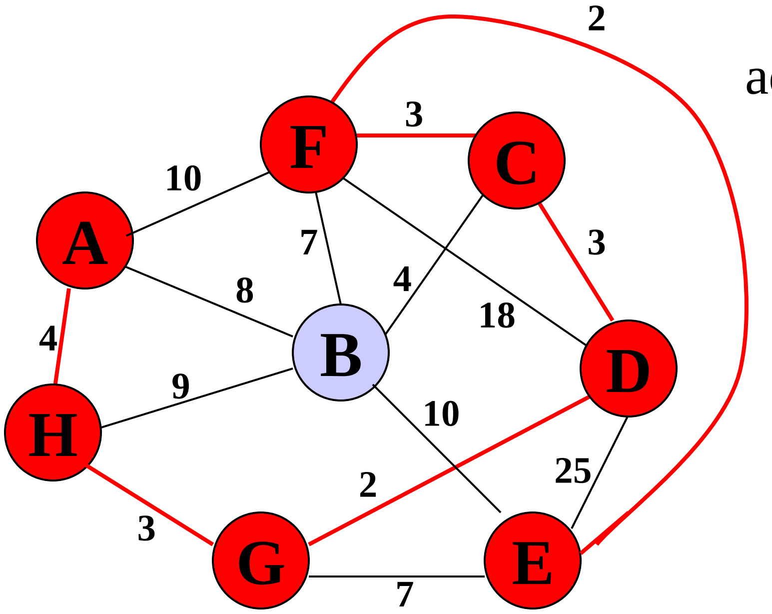
Update distances of adjacent, unselected nodes

	T	d_v	p_v	*U*
A		4	H	A
B		4	C	B
C	T	3	D	
D	T	0	–	
E	T	2	F	
F	T	3	C	
G	T	2	D	
H	T	3	G	



Select node with
minimum distance

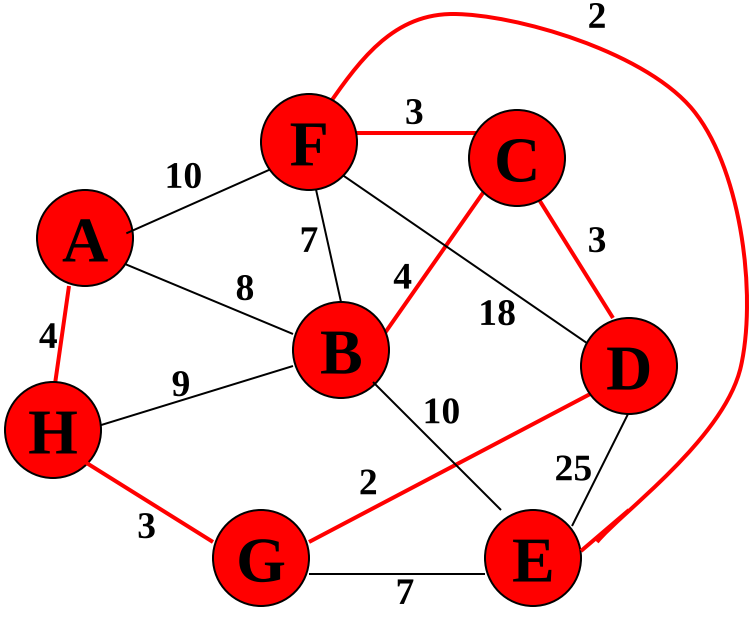
	T	d_v	p_v	*U*
A	T	4	H	
B		4	C	B
C	T	3	D	
D	T	0	–	
E	T	2	F	
F	T	3	C	
G	T	2	D	
H	T	3	G	



Update distances of
adjacent, unselected nodes

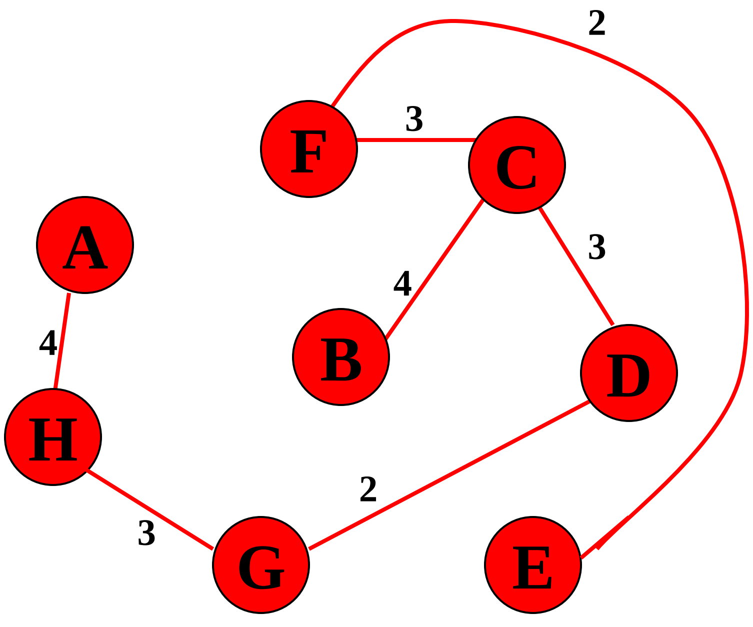
	T	d_v	p_v	*U*
A	T	4	H	
B		4	C	B
C	T	3	D	
D	T	0	–	
E	T	2	F	
F	T	3	C	
G	T	2	D	
H	T	3	G	

Table entries unchanged



Select node with minimum distance

	T	d_v	p_v	*U*
A	T	4	H	
B	T	4	C	
C	T	3	D	
D	T	0	–	
E	T	2	F	
F	T	3	C	
G	T	2	D	
H	T	3	G	

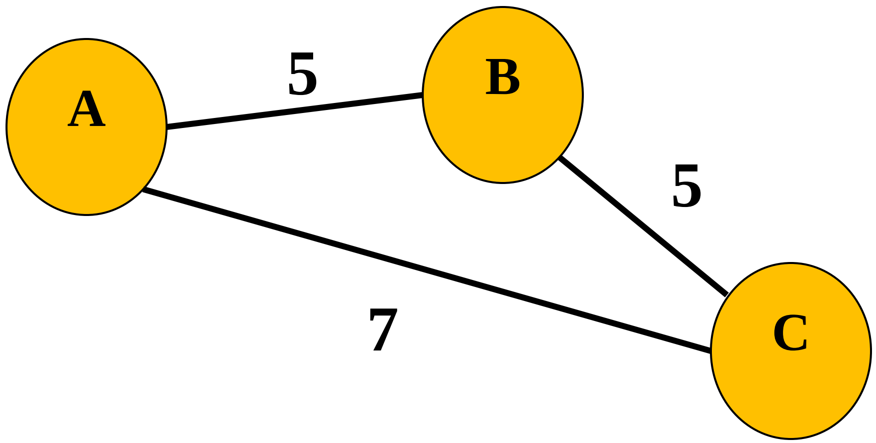


Cost of Minimum
Spanning Tree = Σ d_v = **21**

	T	d_v	p_v	U
A	T	4	H	
B	T	4	C	
C	T	3	D	
D	T	0	–	
E	T	2	F	
F	T	3	C	
G	T	2	D	
H	T	3	G	

Done

- **Dijkstra** selects as next edge the one that leads out from the tree to a node not yet chosen closest to the starting node (Then with this choice, distances are recalculated).
- **Prim** choses as edge the shortest one leading out of the tree constructed so far. So, both algorithms chose a "minimal edge". The main difference is the value chosen to be minimal.
- For **Dijkstra** it is the length of the complete path from start node to the candidate node, for Prim it is just the weight of that single edge.



- In MST case, edges (A→B), (B→C) will be on MST with total weight of 10. So cost of reaching A to C in MST is 10.
- But in Shortest Path case, shortest path between A to C is (A→C) which is 7. (A→C) was never on MST.

Kruskal's Algorithm

- Work with edges, rather than nodes
- Two steps:
  – Sort edges by increasing edge weight
  – Select the first |V| – 1 edges that do not generate a cycle

Algorithm Kruskal

Input: graph G = (V, E).

1. E1 = sort edges in G; // or better use a heap
2. repeat for **N-1** edges // any Spanning Tree has
   //exactly N-1 edges, |V|=N
3. pick up next shortest edge **e** from the ordered **E1**;
4. E1 = E1 – {e};
5. if (T ∪ {e}) does not have a cycle then
   // by set matching algorithm (Union-find)
6. T = T ∪ {e};

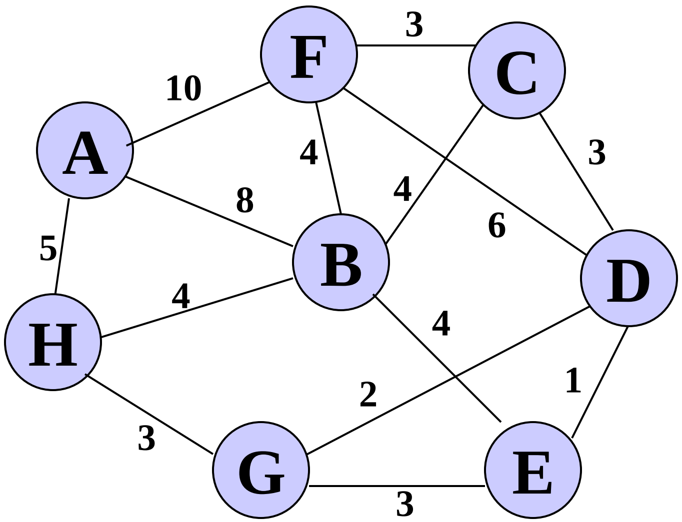
   end repeat;

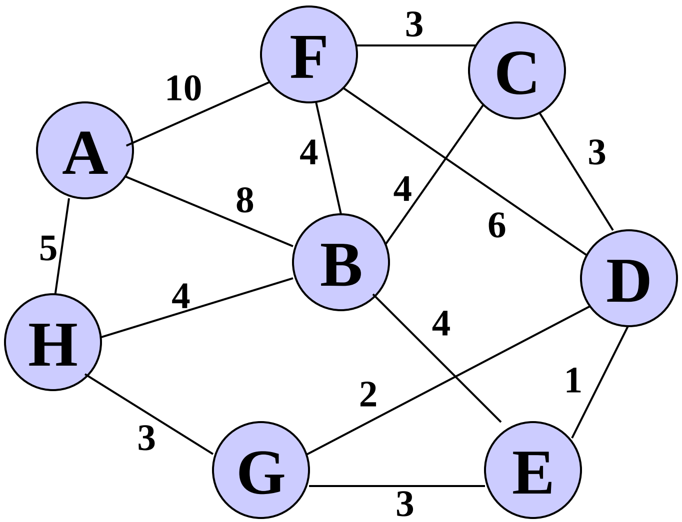
7. return graph (V, T);

End algorithm.

Walk-Through

Consider an undirected, weight graph

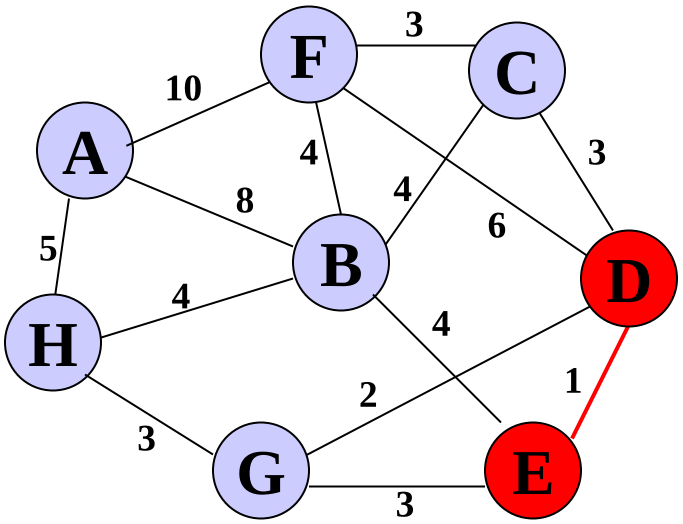




Sort the edges by increasing edge weight

E1	d_v	*T*
(D,E)	1	
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

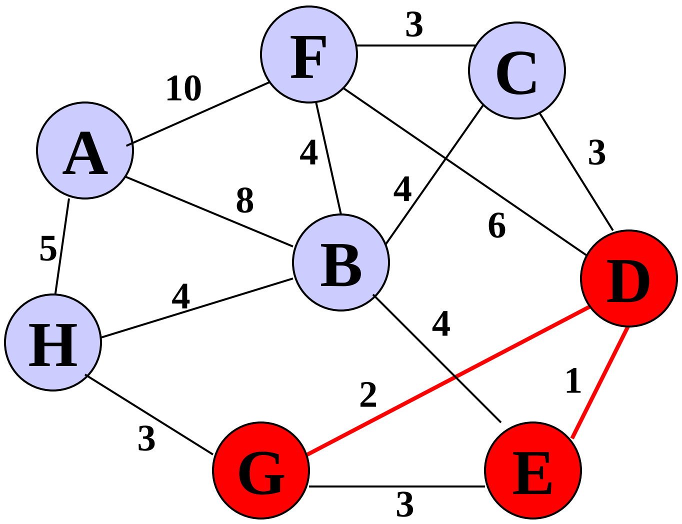
E1	d_v	*T*
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

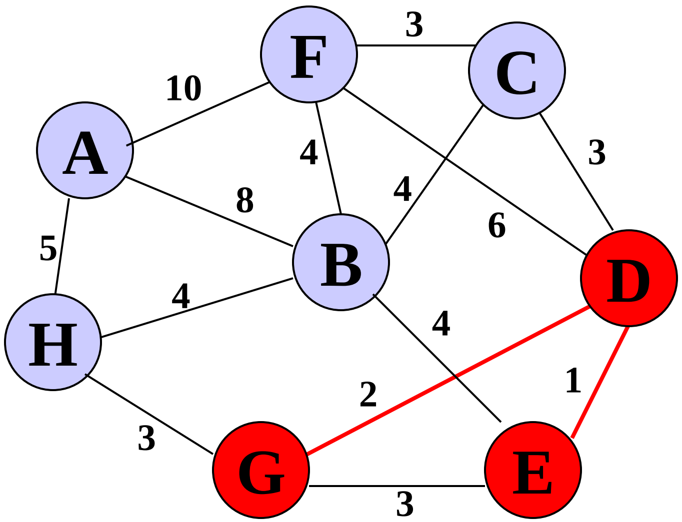
E1	d_v	*T*
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

E1	d_v	***T***
(D,E)	1	√
(D,G)	2	√
(E,G)	3	
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

E1	d_v	***T***
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

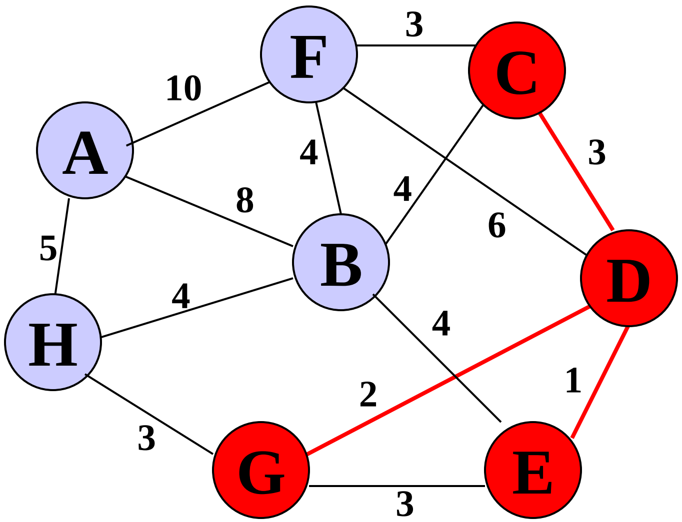


Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	
(G,H)	3	
(C,F)	3	
(B,C)	4	

E1	d_v	*T*
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

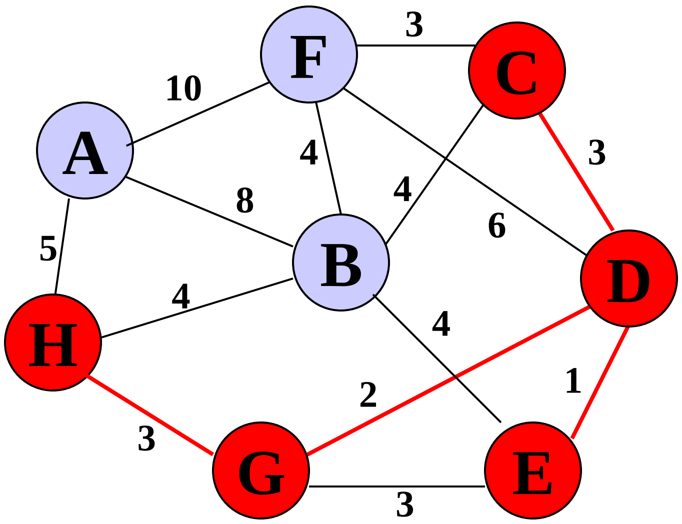
Accepting edge (E,G) would create a cycle



Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	
(C,F)	3	
(B,C)	4	

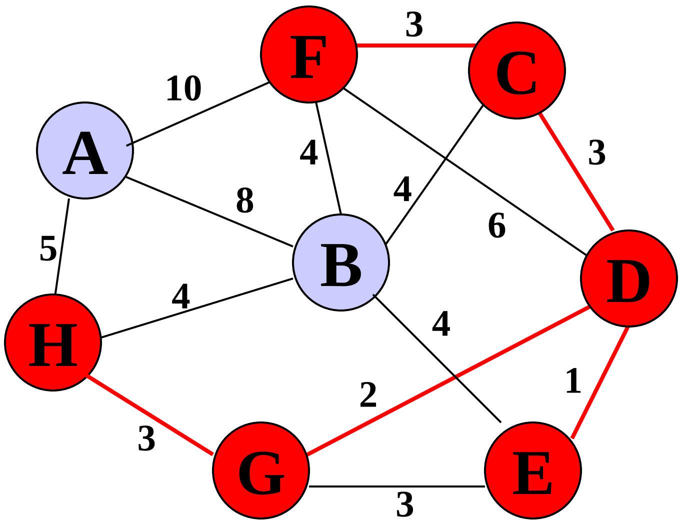
E1	d_v	*T*
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	
(B,C)	4	

E1	d_v	*T*
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

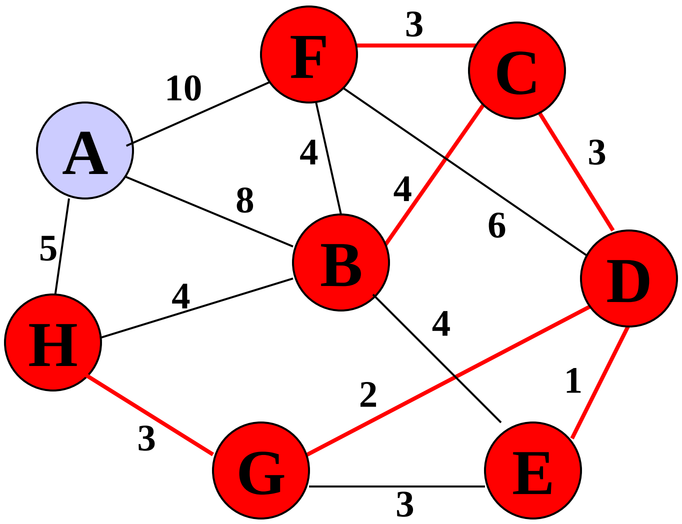


Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	

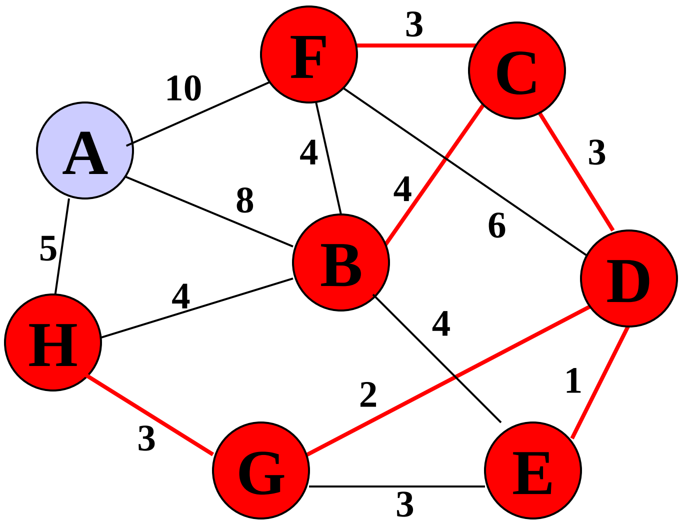
E1	d_v	*T*
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	

Select first |V|–1 edges which do not generate a cycle



E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

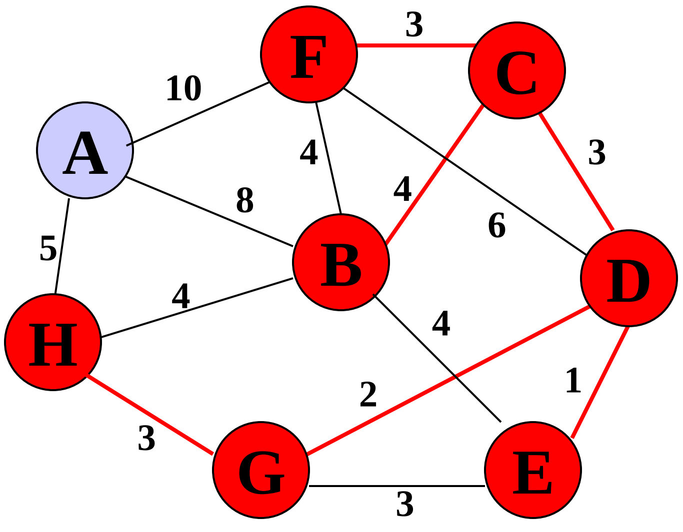
E1	d_v	*T*
(B,E)	4	
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

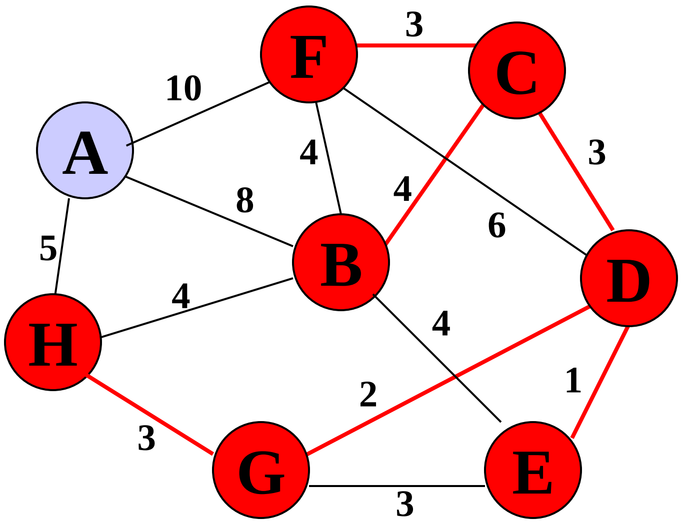
E1	d_v	*T*
(B,E)	4	χ
(B,F)	4	
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

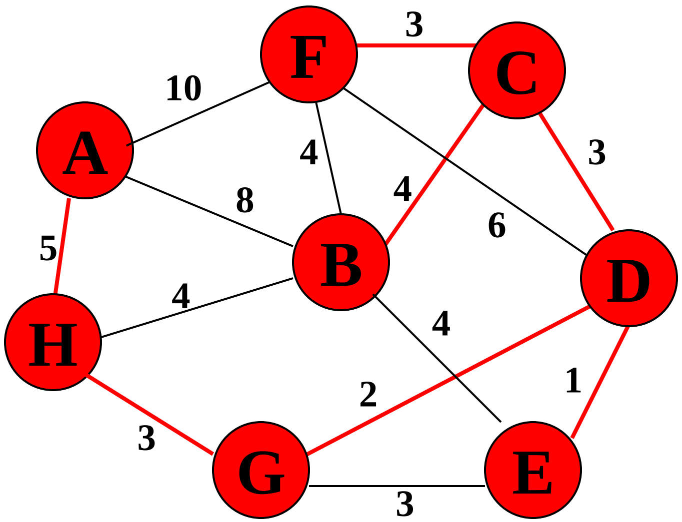
E1	d_v	*T*
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

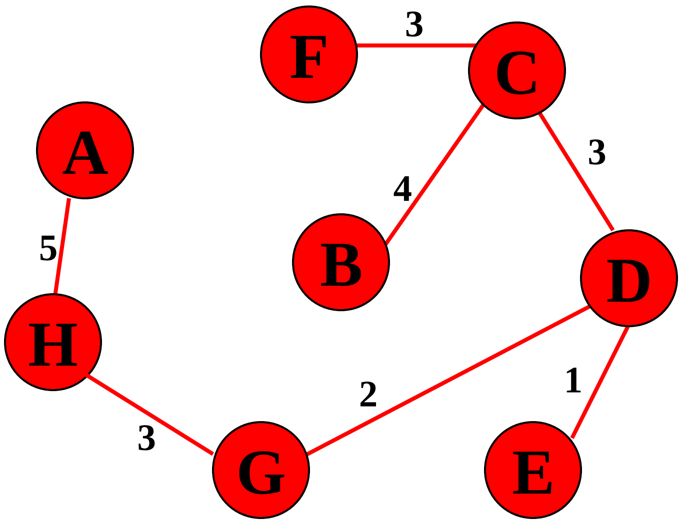
E1	d_v	*T*
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

E1	d_v	*T*
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	



Select first |V|–1 edges which do not generate a cycle

E1	d_v	*T*
(D,E)	1	√
(D,G)	2	√
(E,G)	3	χ
(C,D)	3	√
(G,H)	3	√
(C,F)	3	√
(B,C)	4	√

E1	d_v	*T*
(B,E)	4	χ
(B,F)	4	χ
(B,H)	4	χ
(A,H)	5	√
(D,F)	6	
(A,B)	8	
(A,F)	10	

} not considered

Done

Total Cost = $\Sigma\, d_v = 21$

Detecting Cycles

- Use Disjoint Sets

How Disjoint Sets Help in Cycle Detection

- Initially: Each vertex is its own set (meaning no edges).
  - Example: For 5 vertices {1}, {2}, {3}, {4}, {5}.
- For each edge (u, v):
  - Check whether u and v belong to the same set (using find() operation).
  - If they are in the same set, then adding (u, v) would form a cycle then skip it.
  - If they are in different sets, then no cycle will form add the edge and merge (union) the two sets.
- Repeat until V−1 edges are included (where V = number of vertices).

How Disjoint Sets Help in Cycle Detection

- **Example**
- Suppose we have edges in increasing weight order: (1–2), (2–3), (1–3)
- Start: {1}, {2}, {3}
- Add (1–2): different sets »» union »» {1,2}, {3}
- Add (2–3): different sets »» union »» {1,2,3}
- Add (1–3): both vertices already in same set {1,2,3} »» cycle skip
- Thus DSU helps us detect the cycle efficiently

Time Compexity

- Let v be number of vertices and e the number of edges of a given graph.
- Kruskal's algorithm: $O(e \log e)$
- Prim's algorithm: $O(e \log v)$
- Kruskal's algorithm is preferable on **sparse graphs**, i.e., where *e* is very small compared to the total number of possible edges: $C(v, 2) = v(v-1)/2$.