Graphs

Minimum Spanning Trees

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

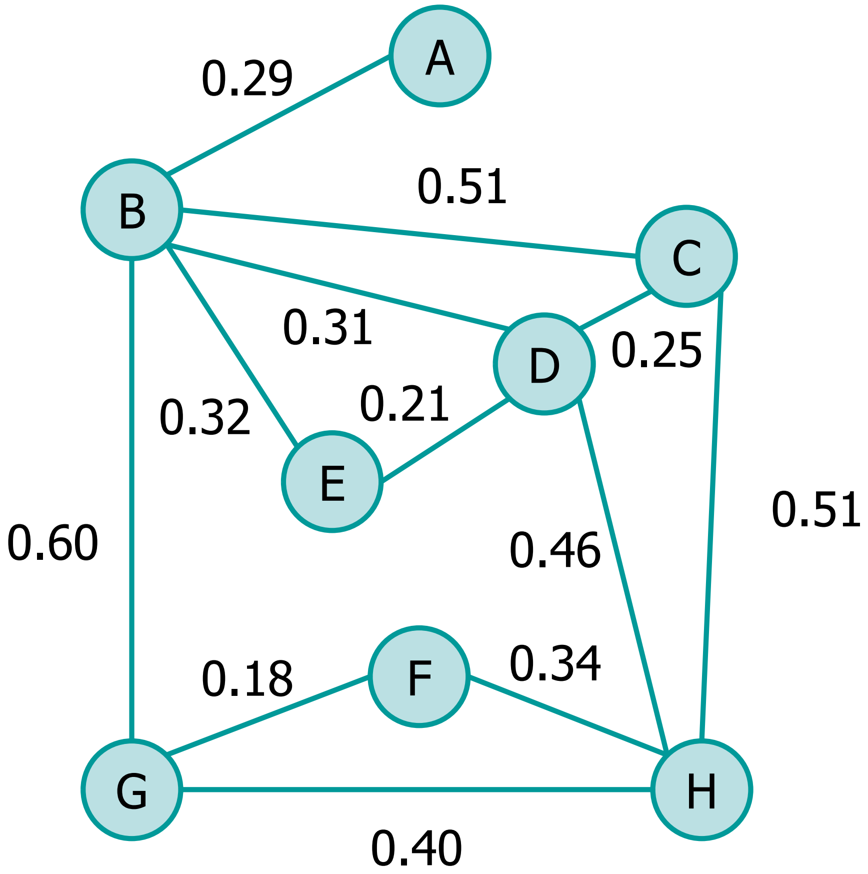
Minimum Spanning Trees

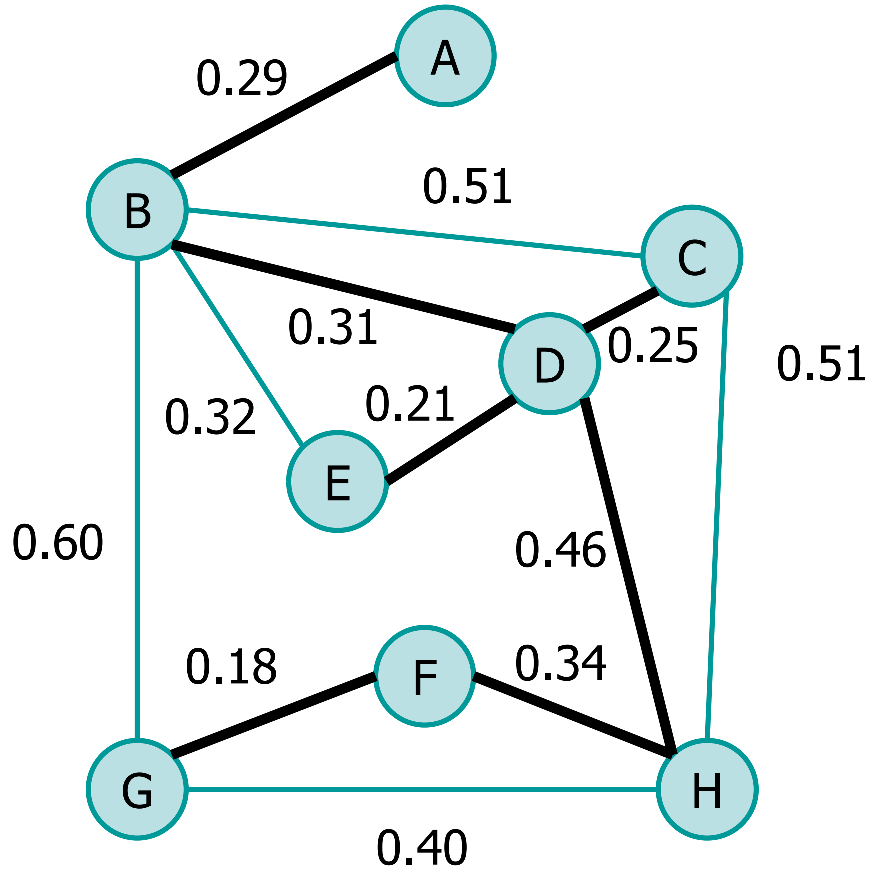
- Given a connected, undirected, weighted graph $G = (V, E)$ in which positive real-value weight function $w: E \rightarrow R$ defines the weights
- A Minimum-weight Spanning Tree (MST) G' is a subset of the edges that connect all the vertices togheter, has no cycles, and has the minimum possibile total edge weight
  - As the subset is acyclic and connects all edges it is a tree, and we call it spanning tree

Example

$$G = (V, E)$$
$$w: E \rightarrow R \quad \text{defines the weights}$$

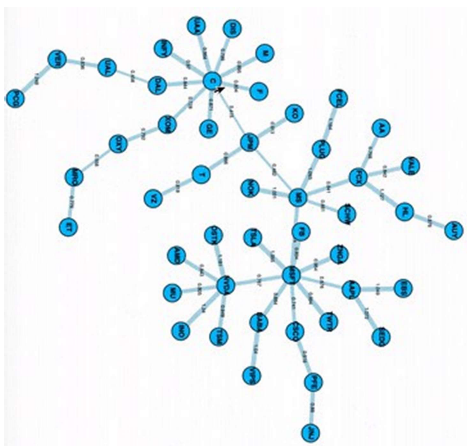
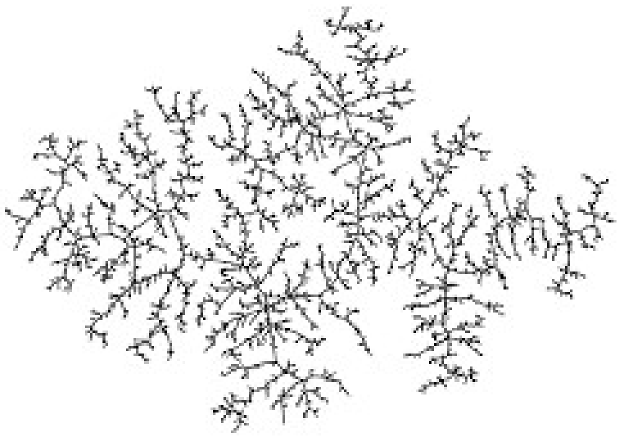
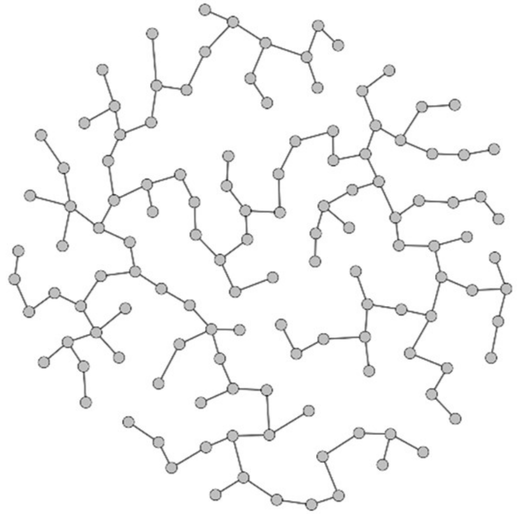
$$G' = (V, \mathrm{T}) \text{ with } T \subseteq E$$
$$G' \text{ is acyclic}$$
$$w(T) = \sum_{(u,v) \in T} w(u, v) \quad \text{is minimum}$$





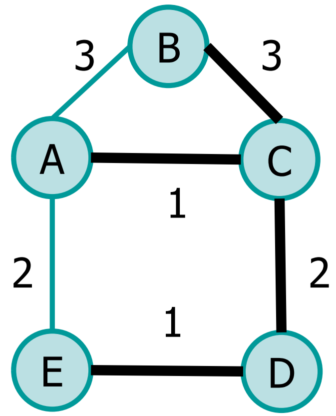
Problem definition

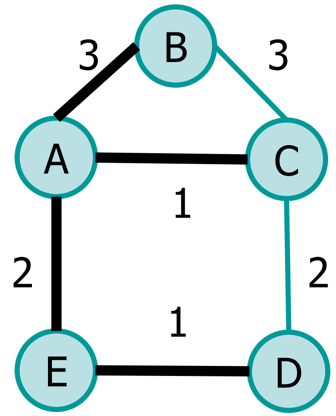
- Application
  - Given an electronic circuit, designers often need to make the pins of several components elettrically equivalent by wiring them togheter
  - To interconnect n pins we can use n connections
  - Of all such arrangements the one that uses the least amount of wire is usually the desired one

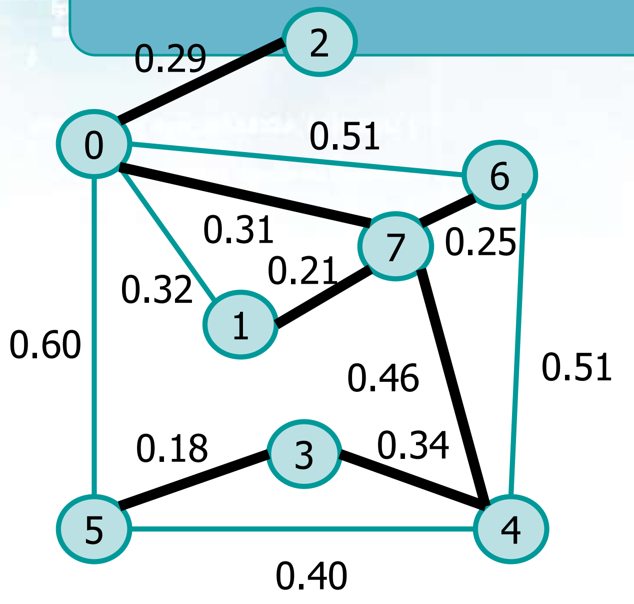
Properties

- MST properties
  - As G' is acyclic and cover all vertices
    - G' is a tree
  - The MST is generally not unique
    - It is unique only iff all weights are distinct
  - A MST may be represented as
    - An adjacency matrix
    - An adjancecy list
    - A list of edges (with their weights)
    - A list of parents (with their weights)

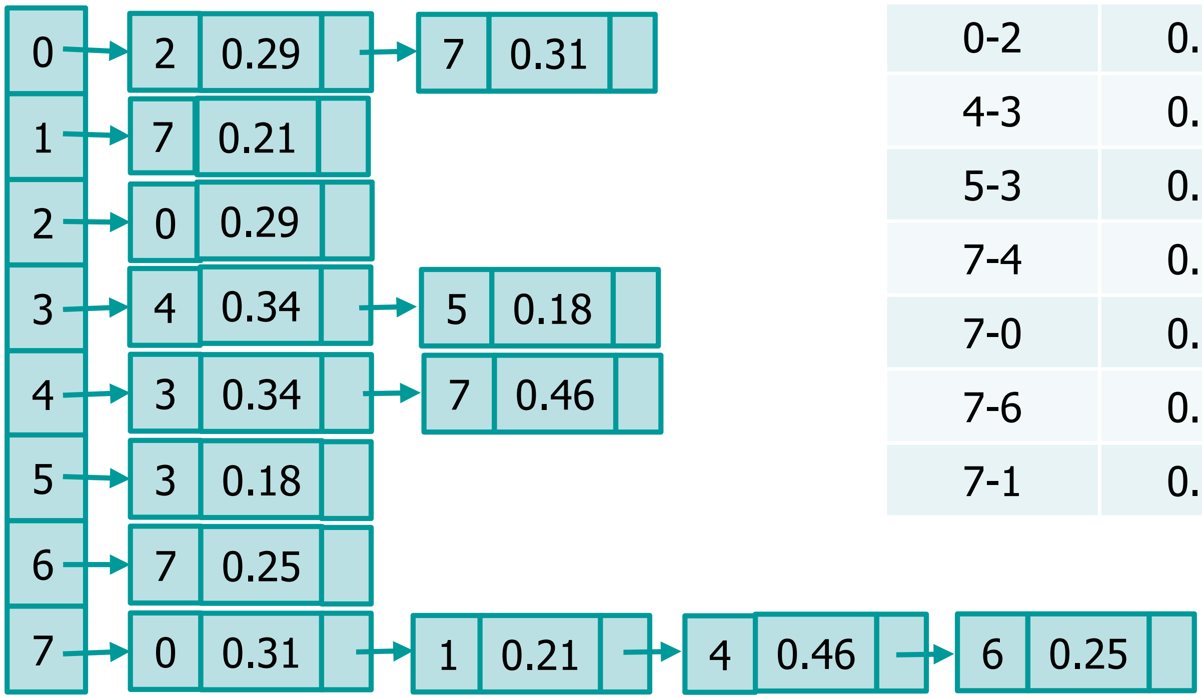




Example



Adjacency matrix or list



List of edges
Specifically used for the Kruskal's algorithm

edge	weight
0-2	0.29
4-3	0.34
5-3	0.18
7-4	0.46
7-0	0.31
7-6	0.25
7-1	0.21

List of parents
Specifically used for the Prim's algorithm

node	parent	weight
0	0	0
1	7	0.21
2	0	0.29
3	4	0.34
4	7	0.46
5	3	0.18
6	7	0.25
7	0	0.31

Algorithms

- We analyze two greedy algorithms
  - Greedy algorithms do not generally guarantee globally optimal solutions
  - Fortunately, for the MST problem they do
- Both algorithms
  - Kruskal's algorithm
  - Prim's algorithm

  are based on a generic method
- The generic method grows a spanning tree by adding one edge at a time

Generic algorithm

Pseudo-code

A is a subset of the MST (initially empty)

graph weighted edges

```
generic_MST (G, w)
  A = ∅
  while A is not a MST do
    find a safe edge (u,v) for A
    A = A ∪ (u, v)
  return A
```

While A is not a MST

Add a safe edge (u,v) to A

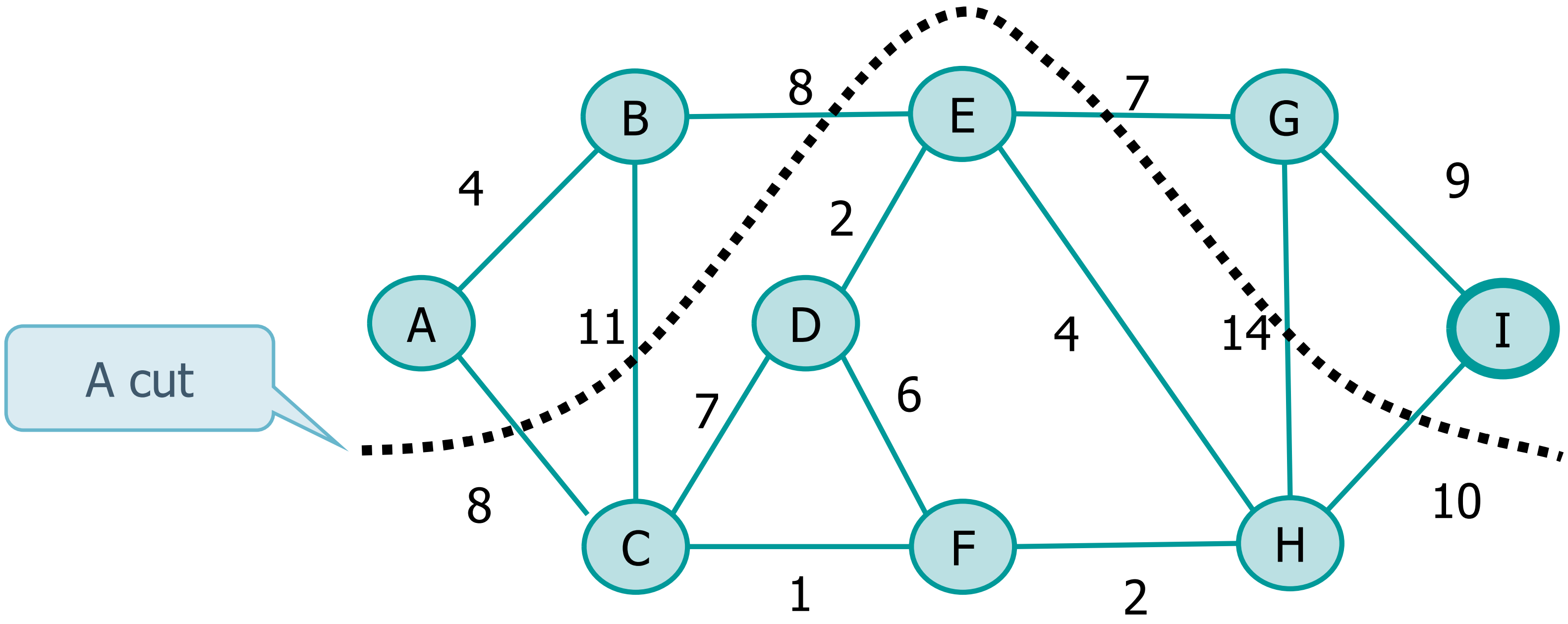
IFF edge (u,v) is safe, adding (u,v) to a subset A of the MST let A as a subset of the MST

Generic algorithm

- Given a set A
  - Set of edges, i.e., a sub-set of a MST
  - Initially empty
- While A is not a MST
  - Find a **safe edge**
  - Add this edge to A
- Invariant
  - The edge (u,v) is **safe** if and only if added to a sub-set of the MST it produces another sub-set of the MST

Definitions

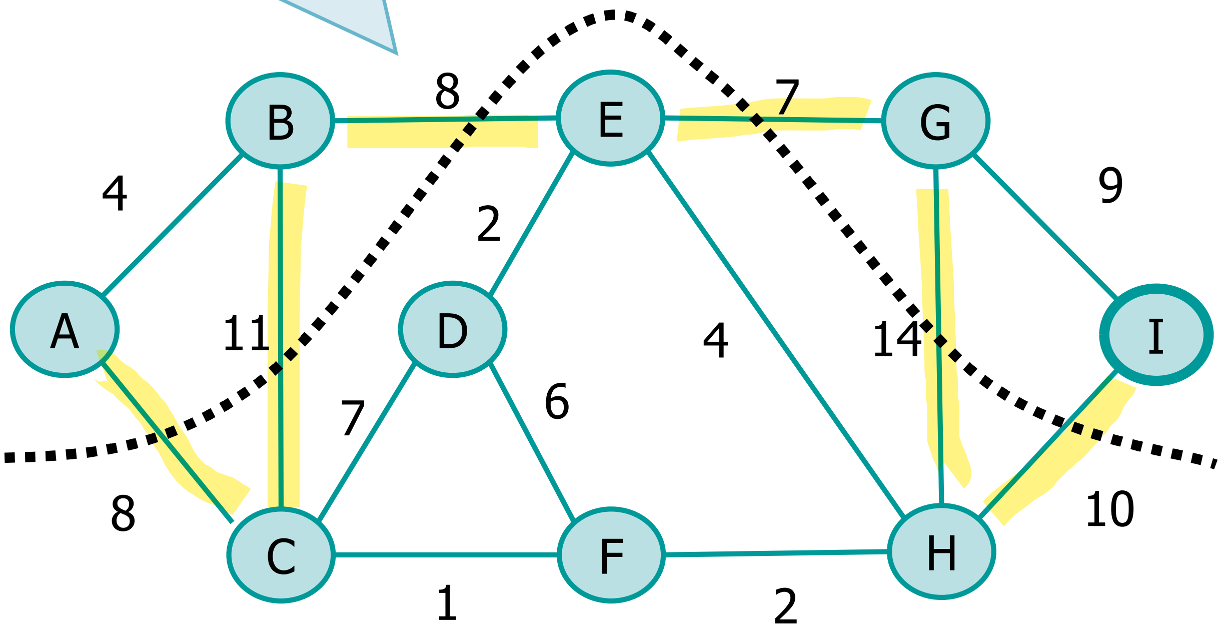
- Given a connected, undirected, and weighted graph $G = (V, E)$, we define
  - Cut
    - A partition of V into S and $V - S$ such that $V = S \cup (V - S)$ and $S \cap (V - S) = \emptyset$



Definitions

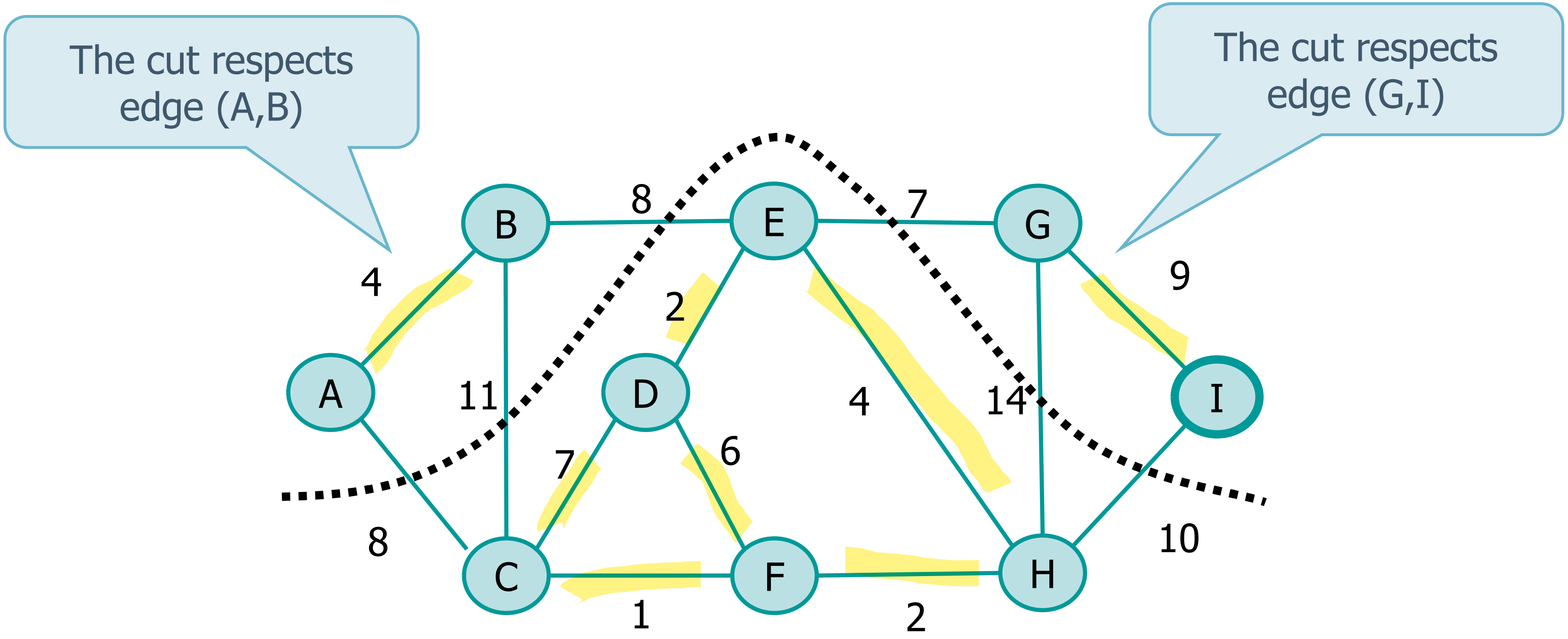
- **Crossing edge**
  - An edge $(v, u) \in E$ crosses the cut if and only if $u \in S$ and $v \in (V - S)$ or vice-versa

Edge (B,E) crosses the cut



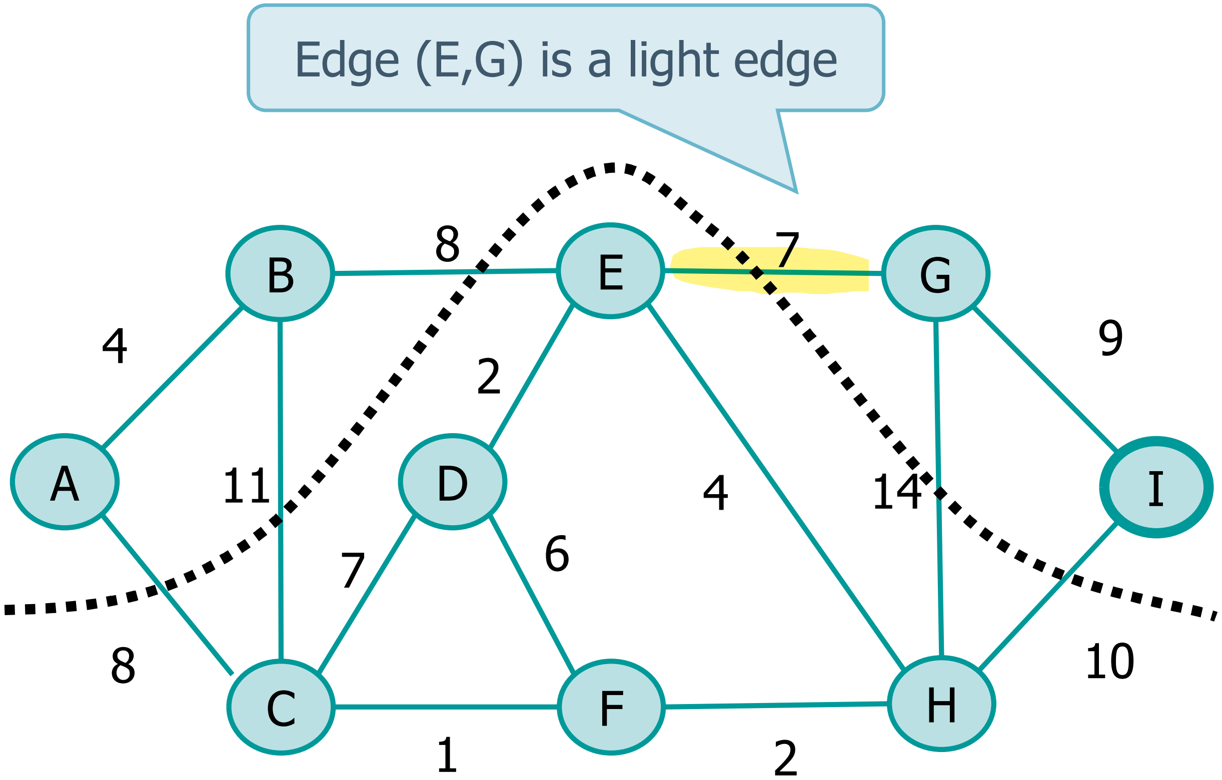
Definitions

- A cut respecting a set of edges
  - A cut respect a set A of edges if no edge of A crosses the cut



Definitions

- A light edge
  - An edge if a light edge if its weight is minimum among the edges crossing the cut

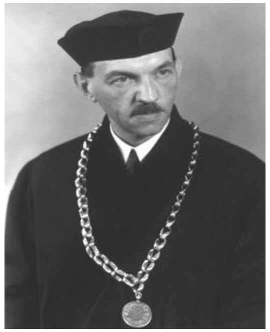


Safe Edges: Theorem

- Let $G = (V, E)$ be a connected, undirected, and weighted graph
- Let
  - A be a subset of E including a MST
    - Initially A is empty
  - $(S, V - S)$ be any cut of G that respects A
  - (v, u) be a light edge crossing the cut $(S, V - S)$
- Then
  - Edge (v, u) is **safe** for A

Prim's Algorithm

- Known as DJP algorithm, Jarnik's algorithm, Prim-Jarnik algorithm, Prim-Dijkstra algorithm
  - Developed in 1930 by Volteci Jarnik (1897-1970)
  - Rediscovered in 1957 by Robert C. Prim (1921-today)
  - Rediscovered in 1959 by Edsger Dijkstra (1930-2002)



- Based on the generic algorithm
- Use the theorem to select the safe edge

Pseudo-code

Pseudo-code

Source = starting vertex

v.key is the minimum weight of any edge connecting v to a vertex in the tree

v.pred is the vertex parent

Extract the vertex from Q and insert it in the MST

Update the key and pred fields of all adjacency nodes

```
mst_Prim (G, w, source)
  for each v ∈ V
    v.key = ∞
    v.pred = NULL
  source.key = 0
  Q = V
  while Q ≠ Ø
    u = extract_min (Q)
    for each v ∈ adjacency list of u
      if v ∈ Q and w(u,v)<v.key
        v.pred = u
        v.key = w(u,v)
```

Pseudo-code

Pseudo-code

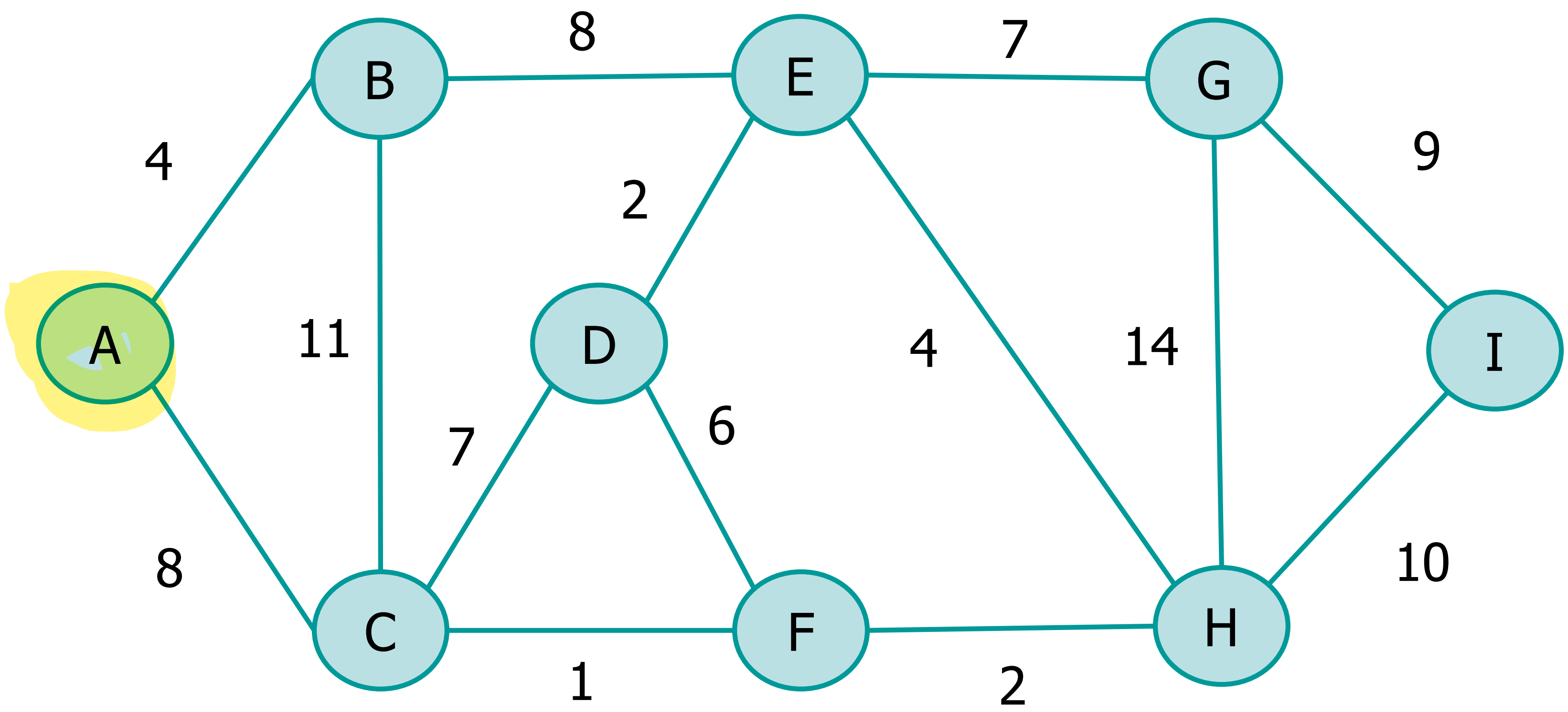
```
mst_Prim (G, w, source)
  for each v ∈ V
    v.key = ∞
    v.pred = NULL
  source.key = 0
  Q = V
  while Q ≠ Ø
    u = extract_min (Q)
    for each v ∈ adjacency list of u
      if v ∈ Q and w(u,v)<v.key
        v.pred = u
        v.key = w(u,v)
```

End when all vertices belong to the same tree

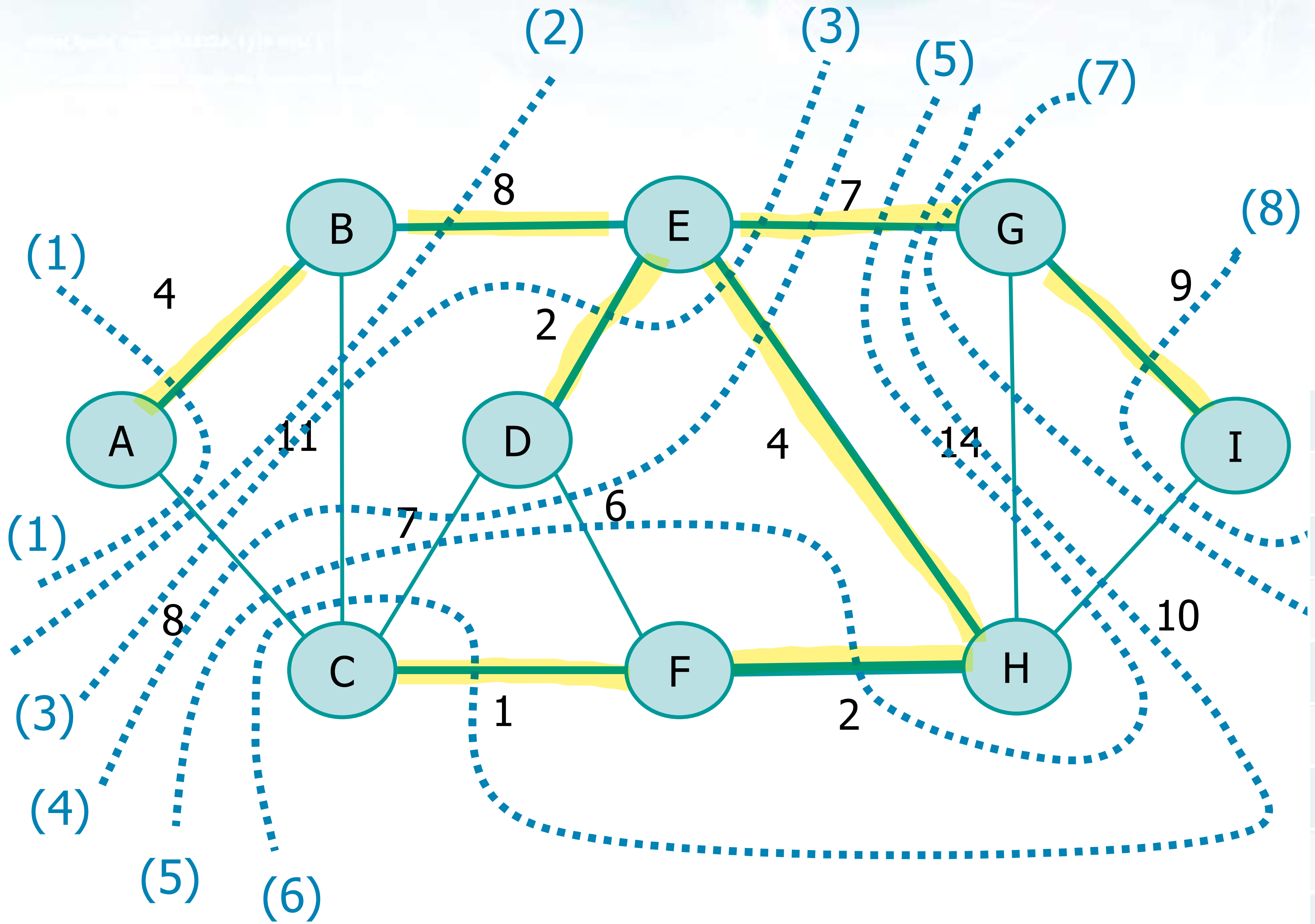
Select all edges crossing the cut Among those, select the edge with minimun weight and add it to A

Adjust S and the set of edges crossing the cut depending on the selected edge

Example



Solution



AB	4
BE	8
DE	2
EH	4
FH	2
CF	1
EG	7
GI	9
	37

or could be BC

Implementation

Graph ADT

```
typedef struct graph_s graph_t;
typedef struct vertex_s vertex_t;
typedef struct edge_s edge_t;


struct graph_s {
  vertex_t *g;
  int nv;
};
```

```
struct edge_s {
  int weight;
  int dst;
  edge_t *next;
};

struct vertex_s {
  int id;
  int color;
  int dist;
  int disc_time;
  int endp_time;
  int pred;
  int scc;
  edge_t *head;
};
```

Array of vertex of lists of edges

Implementation

Client (code extract)

```
g = graph_load (argv[1]);

weight = mst_prim (g);
fprintf (stdout, "Total tree weight: %d\n", weight);

graph_dispose(g);
```

Prim's algorithm

```
int mst_prim (graph_t *g) {
  int i, j, min, weight=0;
  int *fringe;
  edge_t *e;

  fringe = (int *) util_malloc (g->nv * sizeof(int));
  for (i=0; i<g->nv; i++) {
    fringe[i] = i;
  }
```

Implementation

```
fprintf (stdout, "List of edges making an MST:\n");
min = 0;
g->g[min].dist = 0;

while (min != -1) {
  i = min;
  g->g[i].pred = fringe[i];
  weight += g->g[i].dist;
  if (g->g[i].dist != 0) {
    printf("Edge %d-%d (w=%d)\n",
      fringe[i], i, g->g[i].dist);
  }
  min = -1;
  e = g->g[i].head;
```

Consider vertex 0 as a starting one

Implementation

```
    while (e != NULL) {
      j = e->dst;
      if (g->g[j].pred == -1) {
        if (e->weight < g->g[j].dist) {
          g->g[j].dist = e->weight;
          fringe[j] = i;
        }
      }
      e = e->next;
    }
    for (j=0; j<g->nv; j++) {
      if (g->g[j].pred == -1) {
        if (min==-1 || g->g[j].dist<g->g[min].dist) {
          min = j;
        }
      }
    }
  }
  free(fringe);
  return weight;
}
```

Complexity

```
mst_Prim (G, w, source)
  for each v ∈ V
    v.key = ∞
    v.pred = NULL
  source.key = 0
  Q = V
  while Q ≠ Ø
    u = extract_min (Q)
    for each v ∈ adjacency list of u
      if v ∈ Q and w(u,v)<v.key
        v.pred = u
        v.key = w(u,v)
```

$O(|V|)$

Executed $|V|$ times

$O(log_2|V|) \rightarrow O(|V| \cdot log_2|V|)$

Executed $|E|$ times altogether

$O(log_2|V|) \rightarrow O(|E| \cdot log_2|V|)$

Decrease key → log|V|

Overall running time complexity
$T(n) = O(|V| \cdot log_2|V| + |E| \cdot log_2|V|)$

Complexity

- In general

$$T(n) = O(|V| \cdot log_2|V| + |E| \cdot log_2|V|)$$

  - that is

$$T(n) = O(|E| \cdot log_2|V|)$$

- Using an efficient data structure the running time can be improved
  - With a Fibonacci-Heap decrease key is no longer of cost $O(|V|)$ but becomes of cost $O(1)$

$$T(n) = O(|E| + |V| \cdot log_2|V|)$$

Safe Edges: Corollary

- Let $G = (V, E)$ be a connected, undirected, and weighted graph
- Let
  - A be a subset of E including a MST
    - Initially A is empty
  - C is a tree in the forest $G_A = (V, A)$
  - (v, u) is a light edge connecting C to another component of G_A
- Then
  - Edge (v, u) is **safe** for A

Kruskal's Algorithm

- Algorithm proposed by Joseph Kruskal (1928-2010) in 1956
- Based on the generic algorithm
- Use the corollary to select the safe edge
  - Forest of tree, initially single vertices
  - Sort edges into nondecreasing order by weigth w
  - Iteration
    - Select a safe edge, i.e., an edge with minimum weight connecting two trees and generating one single tree (Union-Find)
  - End
    - All vertices belong to the same tree

Pseudo-code

Pseudo-code

```
mst_Kruskal (G, w)
  A = ∅
  for each vertex v ∈ V
    make_set (v)
  sort E into non-decreasing order by weight w
  for each edge (u,v)∈E
  if find (u) ≠ find (v)
    A = A ∪ (u,v)
    union (u,v)
return A
```

A is initially the empty set

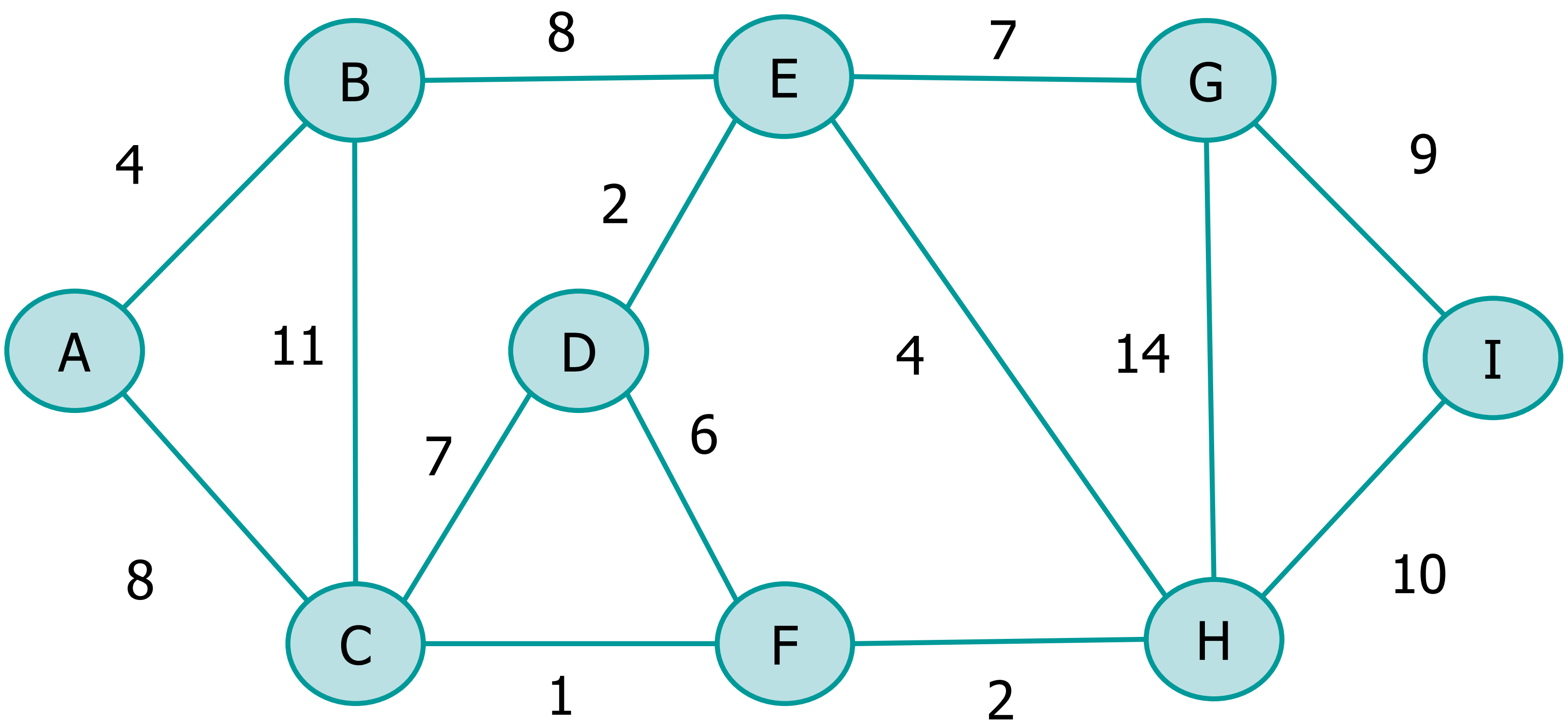
For each v create a set

taken in nondecreasing order by weight

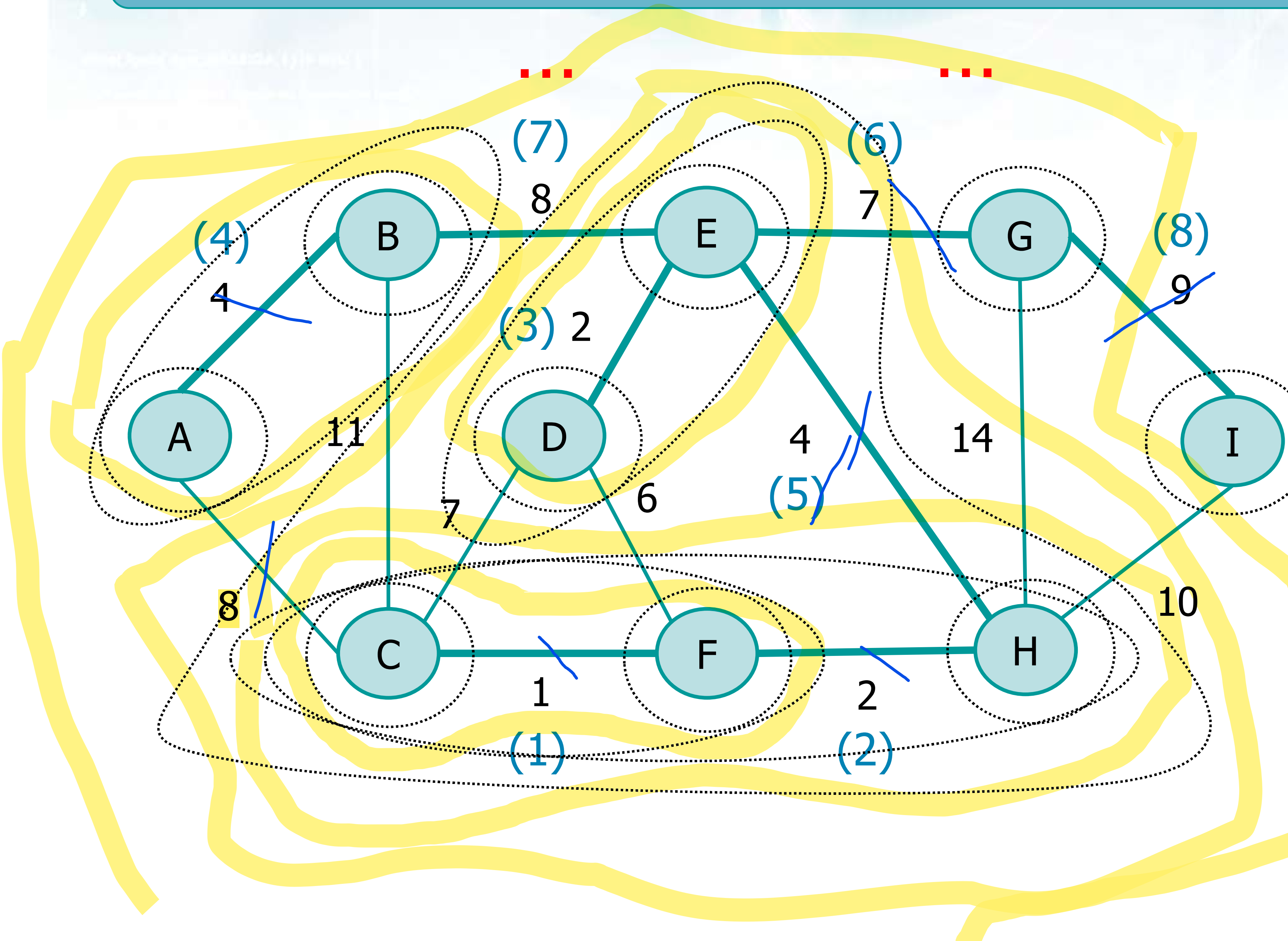
Find representative of u and v

Union set

Example



Solution



CF	1
FH	2
DE	2
AB	4
EH	4
EG	7
BE	8
GI	9
	37

Implementation

Graph ADT

```
struct graph_s {
  vertex_t *g;
  int nv;
};
struct edge_s {
  int weight;
  int dst;
};
struct vertex_s {
  int id;
  int ne;
  int color;
  int dist;
  int scc;
  int disc_time;
  int endp_time;
  int pred;
  edge_t *edges;
};
```

Array of vertex of array of edges

ADT to store edges and order them in ascending order by weight

```
typedef struct {
  int src, dst, weight;
} link;
```

Implementation

Client
(code extract)

```
g = graph_load (argv[1]);

weight = mst_kruskal (g);
fprintf (stdout, "Total tree weight: %d\n", weight);

graph_dispose(g);
```

Kruskal's
algorithm

```
int mst_kruskal (graph_t *g) {
  int i, j, k, weight, ne, nl;
  link *edges;

  for (nl=i=0; i<g->nv; i++) {
    nl += g->g[i].ne;
  }
  nl /= 2;
  edges = (link *)util_calloc(nl, sizeof(link));
  nl = 0;
```

Count the total
number of edges

Implementation

Create array of link elements
AND
Order elements by weight

```
for (i=0; i<g->nv; i++) {
  for (j=0; j<g->g[i].ne; j++) {
    if (i < g->g[i].edges[j].dst) {
      k = nl - 1;
      while (k>=0 &&
             edges[k].weight>g->g[i].edges[j].weight) {
        edges[k+1] = edges[k];
        k--;
      }
      edges[k+1].src = i;
      edges[k+1].dst = g->g[i].edges[j].dst;
      edges[k+1].weight = g->g[i].edges[j].weight;
      nl++;
    }
  }
}
```

Implementation

```
  /* build the tree */
  fprintf(stdout, "List of edges making an MST:\n");
  for (i=0; i<g->nv; i++) {
    g->g[i].pred = i;
  }
  weight = ne = 0;
  for (k=0; k<nl && ne<g->nv-1; k++) {
    i = union_find_find (g, edges[k].src);
    j = union_find_find (g, edges[k].dst);

    union_find_union (g, edges, i, j, k, &weight, &ne);
  }

  free(edges);

  return weight;
}
```

Create the tree

Implementation

Union-Find Algorithms

```
static int union_find_find (graph_t *g, int k) {
  int i = k;
  while (i != g->g[i].pred) {
    i = g->g[i].pred;
  }
  return i;
}
static void union_find_union (graph_t *g, link *edges,
  int i, int j, int k, int *weight, int *ne
) {
  if (i != j) {
      fprintf (stdout, "Edge %d-%d (w=%d)\n",
      edges[k].src, edges[k].dst, edges[k].weight);
    g->g[j].pred = i;
    *weight += edges[k].weight;
    *ne = *ne + 1;
  }
  return;
}
```

Find

Union

Complexity

```
mst_Kruskal (G, w)
  A = ∅
  for each vertex v ∈ V
    make_set (v)
  sort E into non-decreasing order by weight w
  for each edge (u,v)∈E
  if find (u) ≠ find (v)
    A = A ∪ (u,v)
    union (u,v)
return A
```

$O(1)$

Executed V times

$O(1) \rightarrow \boldsymbol{O(|V|)}$

$O(|E| \cdot log_2|E|)$

Executed E times

Union and find takes
$O(log_2|E|) \rightarrow O(|E| \cdot log_2|E|)$

Overall running time complexity
$T(n) = O(|E| \cdot log_2|E|)$

Complexity

- In general

$$T(n) = O(|E| \cdot log_2|E|)$$

- Asintotically, for dense graph

$$E = \frac{|V| \cdot (|V|-1)}{2} \quad \rightarrow \quad |E| > |V|$$

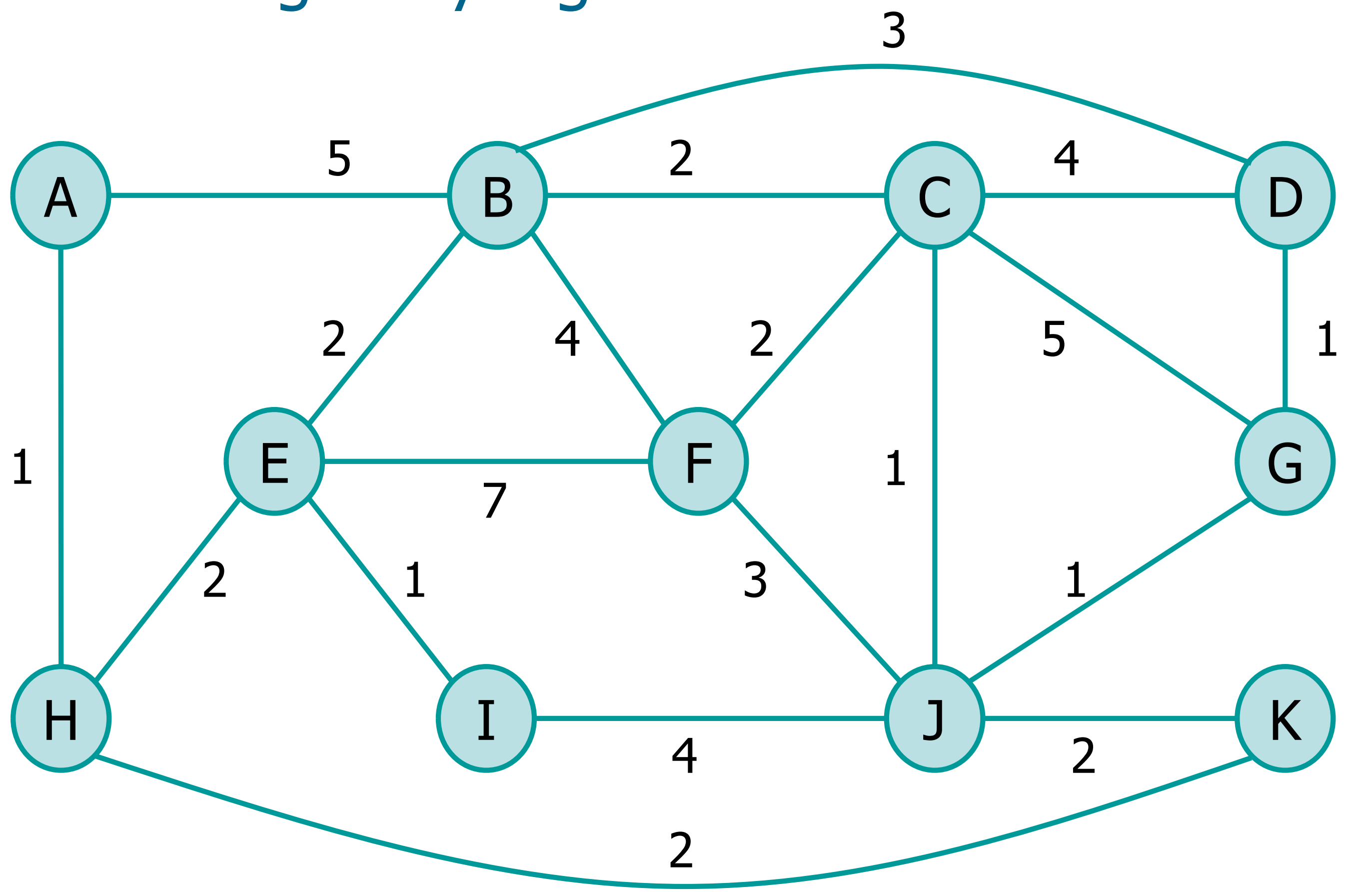
  - Then, Prim is more efficient than Kruskal

$$Prim \quad T(n) = O(|E| + |V| \cdot log_2|V|)$$

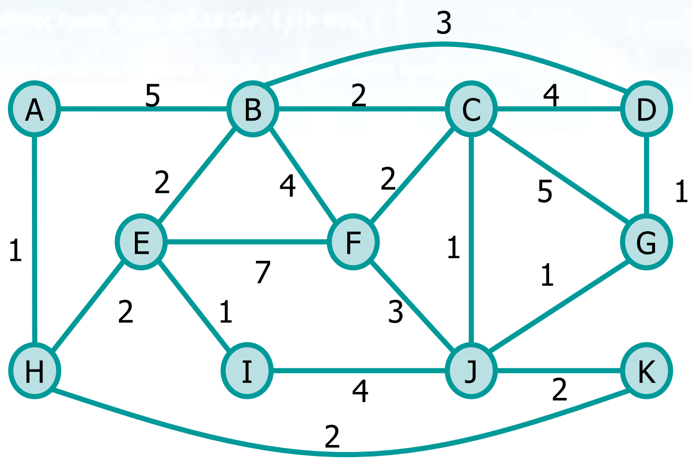
$$Kruskal \quad T(n) = O(|E| \cdot log_2|E|)$$

Exercise

- Given the following graph apply
  - Prim's greedy algorithm from vertex A
  - Kruskal's greedy algorithm



Solution



Prim

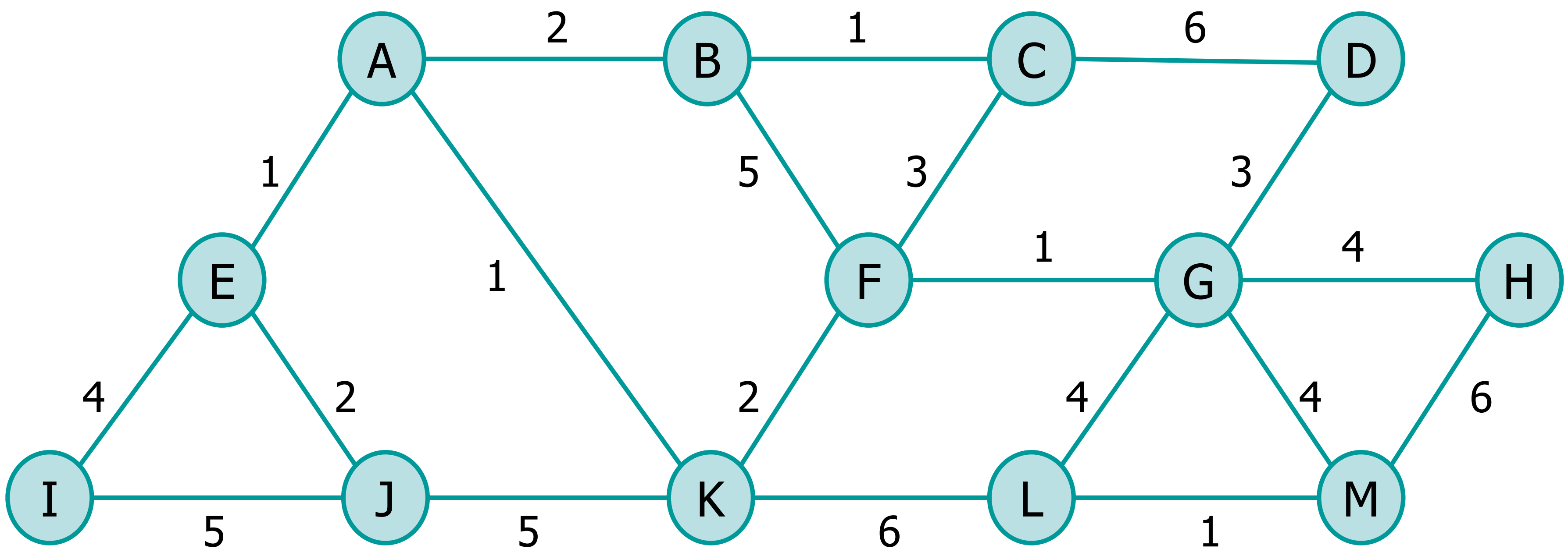
```
Edge [ 0]  A - [ 7]  H ==> weight =  1
Edge [ 7]  H - [ 4]  E ==> weight =  2
Edge [ 4]  E - [ 8]  I ==> weight =  1
Edge [ 4]  E - [ 1]  B ==> weight =  2
Edge [ 1]  B - [ 2]  C ==> weight =  2
Edge [ 2]  C - [ 9]  J ==> weight =  1
Edge [ 9]  J - [ 6]  G ==> weight =  1
Edge [ 6]  G - [ 3]  D ==> weight =  1
Edge [ 2]  C - [ 5]  F ==> weight =  2
Edge [ 7]  H - [10]  K ==> weight =  2
Total tree weight               = 15
```

Kruskal

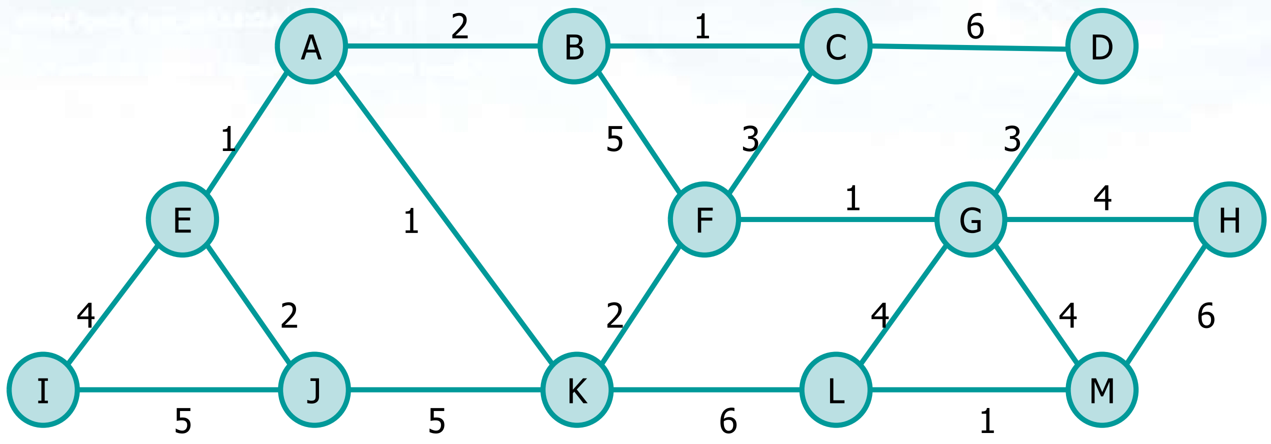
```
Edge [ 0]  A - [ 0]  H ==> weight =  1
Edge [ 2]  C - [ 2]  J ==> weight =  1
Edge [ 3]  D - [ 3]  G ==> weight =  1
Edge [ 4]  E - [ 4]  I ==> weight =  1
Edge [ 6]  G - [ 6]  J ==> weight =  1
Edge [ 1]  B - [ 1]  C ==> weight =  2
Edge [ 1]  B - [ 1]  E ==> weight =  2
Edge [ 2]  C - [ 2]  F ==> weight =  2
Edge [ 4]  E - [ 4]  H ==> weight =  2
Edge [ 7]  H - [ 7]  K ==> weight =  2
Total tree weight               = 15
```

Exercise

- Given the following graph apply
  - Prim's greedy algorithm from vertex A
  - Kruskal's greedy algorithm



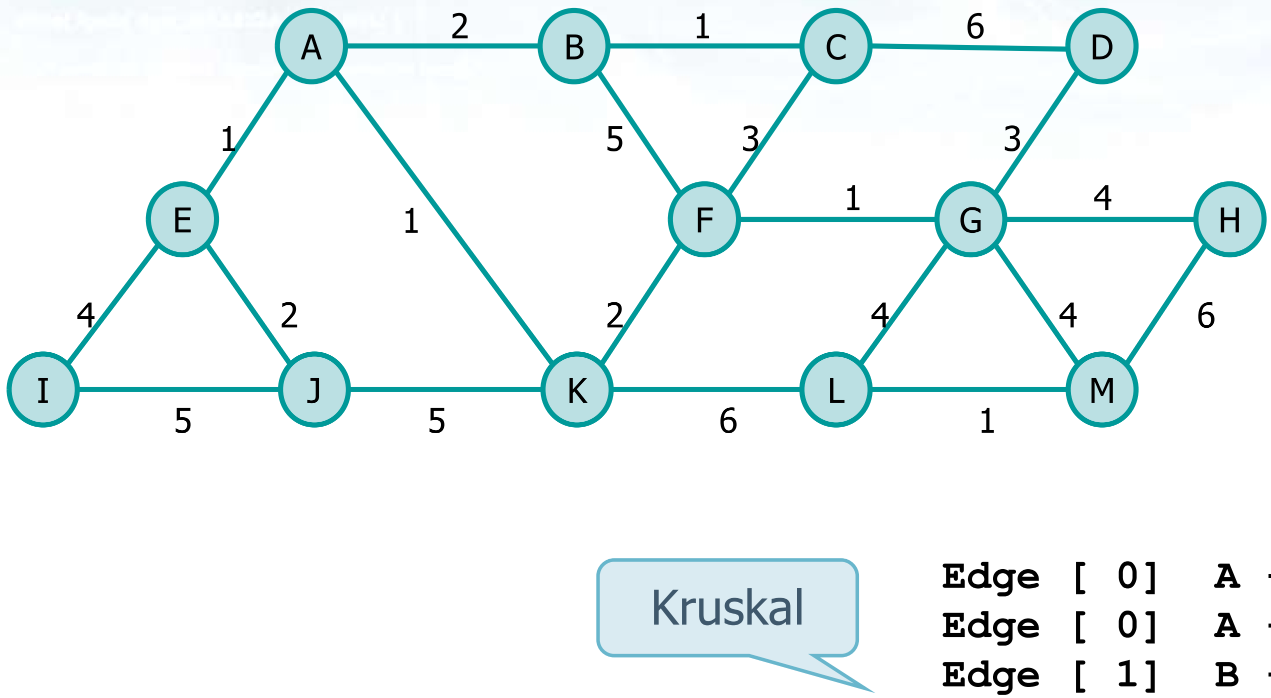
Solution



Prim

```
Edge [ 0]  A - [ 4]  E ==> weight =  1
Edge [ 0]  A - [10]  K ==> weight =  1
Edge [ 0]  A - [ 1]  B ==> weight =  2
Edge [ 1]  B - [ 2]  C ==> weight =  1
Edge [10]  K - [ 5]  F ==> weight =  2
Edge [ 5]  F - [ 6]  G ==> weight =  1
Edge [ 4]  E - [ 9]  J ==> weight =  2
Edge [ 6]  G - [ 3]  D ==> weight =  3
Edge [ 6]  G - [ 7]  H ==> weight =  4
Edge [ 4]  E - [ 8]  I ==> weight =  4
Edge [ 6]  G - [11]  L ==> weight =  4
Edge [11]  L - [12]  M ==> weight =  1
Total tree weight                  = 26
```

Solution




```
Edge [ 0]  A - [ 0]  E ==> weight =  1
Edge [ 0]  A - [ 0]  K ==> weight =  1
Edge [ 1]  B - [ 1]  C ==> weight =  1
Edge [ 5]  F - [ 5]  G ==> weight =  1
Edge [11]  L - [11]  M ==> weight =  1
Edge [ 0]  A - [ 0]  B ==> weight =  2
Edge [ 4]  E - [ 4]  J ==> weight =  2
Edge [ 5]  F - [ 5]  K ==> weight =  2
Edge [ 3]  D - [ 3]  G ==> weight =  3
Edge [ 4]  E - [ 4]  I ==> weight =  4
Edge [ 6]  G - [ 6]  H ==> weight =  4
Edge [ 6]  G - [ 6]  L ==> weight =  4
Total tree weight                 = 26
```