Graphs

Minimum Spanning Trees

Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Problem definition

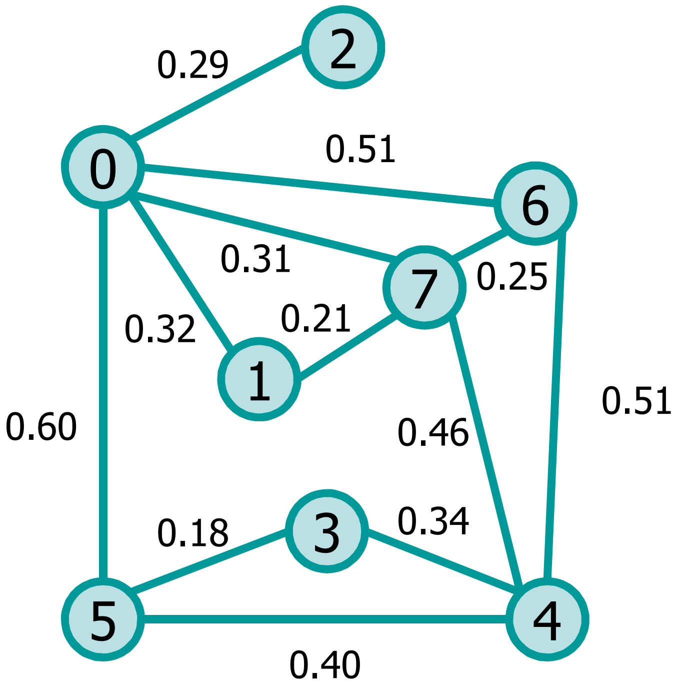
- Example
  - Given an electronic circuit, designers often need to make the pins of several components elettrically equivalent by wiring them togheter
  - To interconnect n pins we can use n-1 connections
  - Of all such arrangements the one that uses the least amount of wire is usually the most desiderable
- Such a problem can be mapped as a **Minimum Spanning Tree** problem

Minimum Spanning Trees

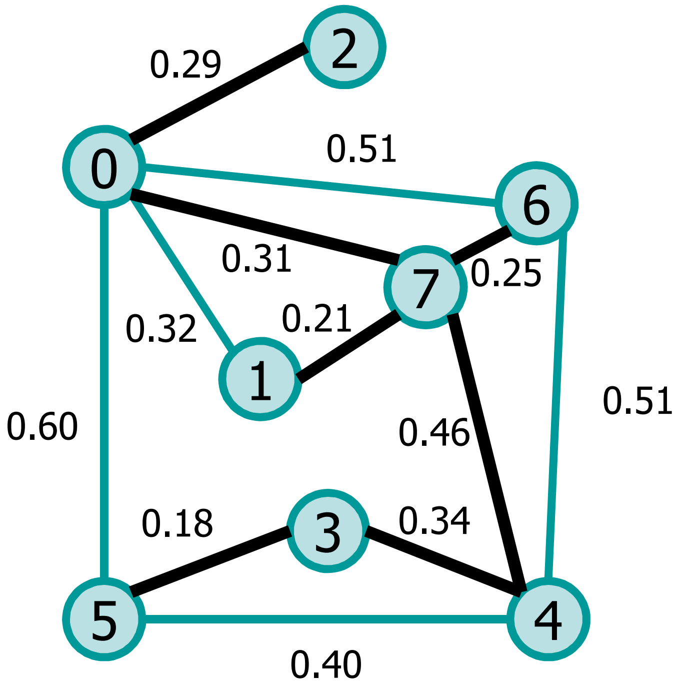
- Given a graph G=(V,E)
  - Connected
  - Undirected
  - Weighted
    - With a positive real-value weight function w: E→R
- A Minimum-weight Spanning Tree (MST) G’ is a graph such that
  - G'=(V, T) with $T \subseteq E$
  - G' is acyclic
  - G' minimizes
    - $w(T)=\Sigma_{(u,v) \in T} w(u,v)$

Example

G = (V, E)

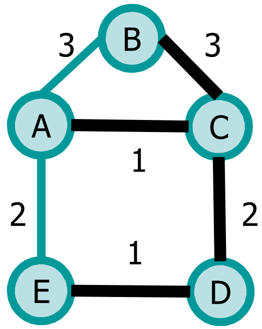


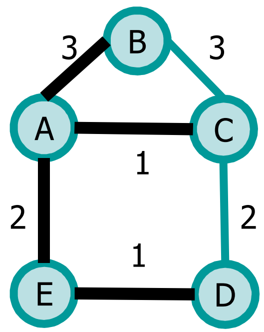
G′ = (V, T) with $T \subseteq E$



Properties

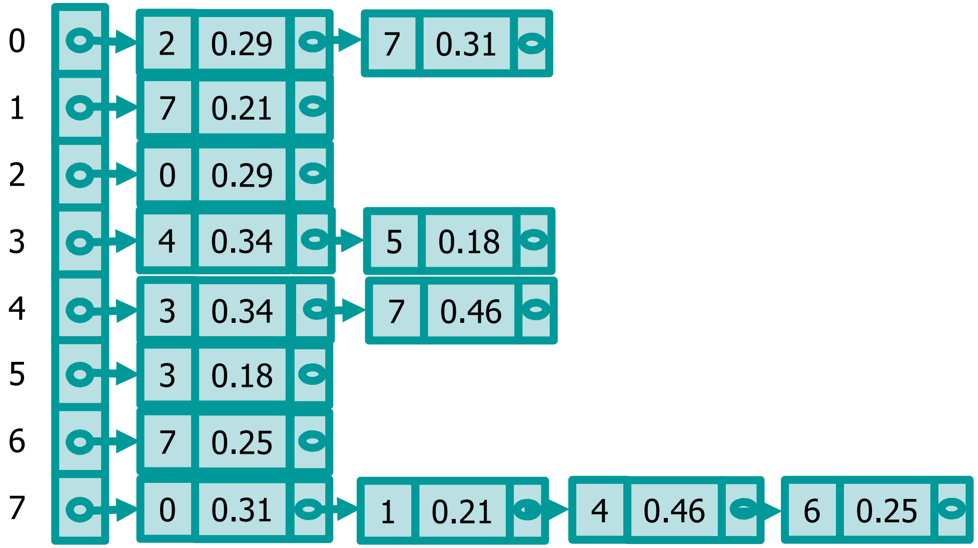
- MST properties
  - As G' is acyclic and cover all vertices
    - G' is a tree
  - The MST is generally not unique
    - It is unique only iff all weights are distinct
  - A MST may be represented as
    - An adjacency matrix or list
    - A list of edges plus weights
    - A list of parents plus weights

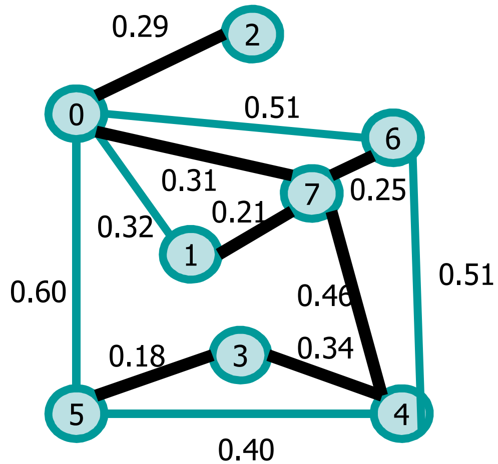




Representation

- Adjacency list
  - Array of lists of list of lists



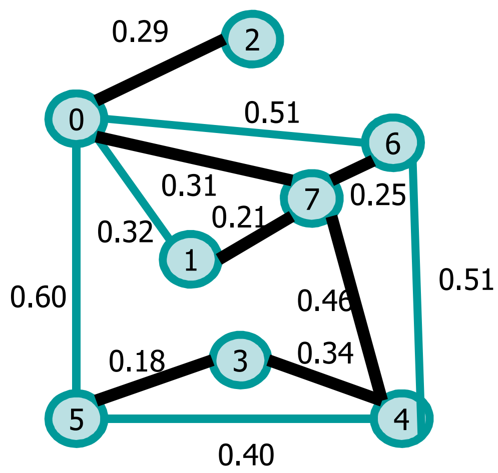


Representation

- List of edges (and weights)
  - Static or dynamic array

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

Specifically used for the Kruskal's algorithm

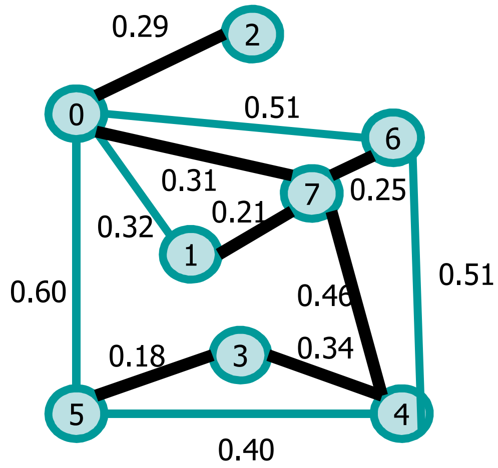


Representation

- List of parents (and weights)
  - Static or dynamic array

	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

Specifically used for the Prim's algorithm



Algorithms

- We will analyze two greedy algorithms
  - Greedy algorithms do not generally guarantee globally optimal soluzions
  - 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)

```
generic_MST (G, w)
  A = ϕ
  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

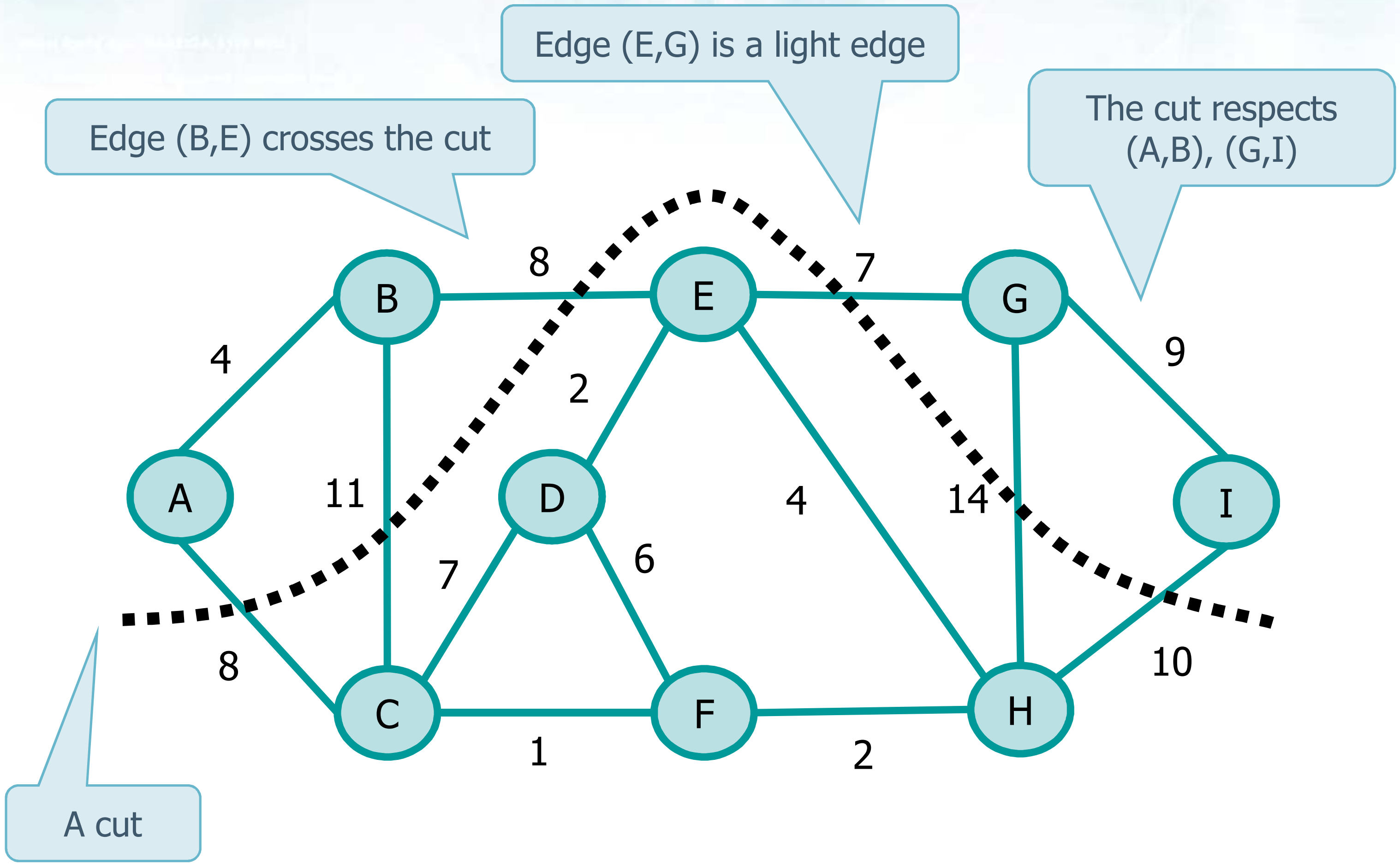
Generic algorithm

- 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

- G=(V,E) connected, undirected, and weighted
  - Cut
    - A partition of V into S and V-S such that
    - $V = S \cup (V-S)$ && $S \cap (V-S) = \varnothing$
  - Crossing edge
    - An edge $(u,v) \in E$ crosses the cut if and only if
    - $u \in S$ && $v \in (V-S)$ or vice-versa
  - A cut respecting a set of edges
    - A cut respect a set A of edges if no edge of A crosses the cut
  - A light edge
    - An edge if a light edge if its weight is minimum among the edges crossing the cut

Example


Edge (E,G) is a light edge
Edge (B,E) crosses the cut
The cut respects (A,B), (G,I)
A cut
A
B
C
D
E
F
G
H
I
4
8
11
8
7
2
6
1
4
7
14
2
9
10

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
  - (u, v) be a light edge crossing the cut (S, V-S)
- Then
  - Edge (u,v) is **safe** for A

Prim's Algorithm

- Known as DJP algorithm, Jarnik's algorithm, Prim-Jarnik algorithm, Prim-Dijkstra algorithm
  - Developed in 1930 by Vojtech Jarnik
  - Rediscovered in 1957 by Robert Prim
  - Rediscovered in 1959 by Edsger Dijkstra
- Based on the generic algorithm
- Use the theorem to select the safe edge

Pseudo-code

Pseudo-code

Source = starting vertex

```
mst_Prim (G, w, source)
  for each v ∈ V
    v.key = ∞
    v.pred = NULL
  source.key = 0
  Q = V
  while Q ≠ ϕ
    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)
```

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

Pseudo-code

Pseudo-code

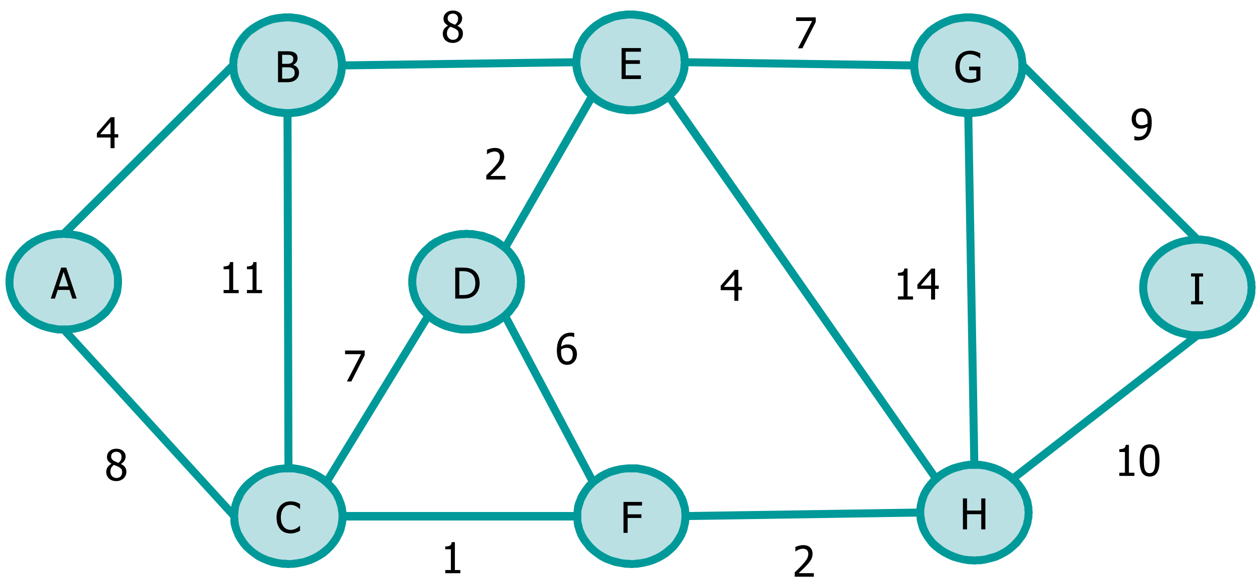
```
mst_Prim (G, w, source)
  for each v ∈ V
    v.key = ∞
    v.pred = NULL
  source.key = 0
  Q = V
  while Q ≠ ϕ
    u = extract_min (Q)
    for each v ∈ adjacency list of u
      if v ∈ Q and w(u,v)<v.key
        v.pred = u
        v.key = w(u,v)
```

End when all vertices belong to the same tree

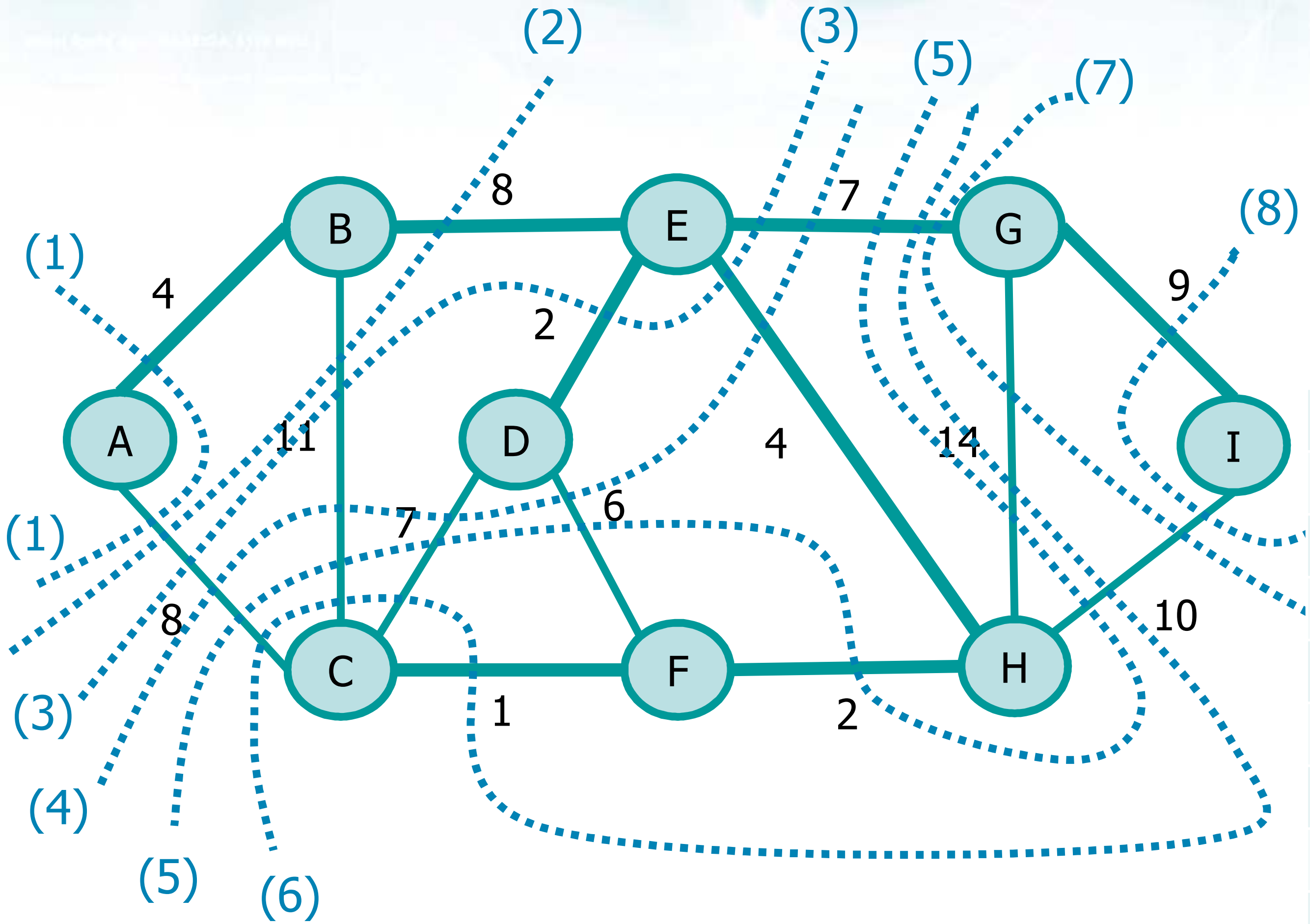
Select all edges crossing the cut Among those, select the edge with minimun weight and add it to A

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

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 ≠ ϕ
    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(lg |V|) → **O(|V| log|V|)**

Executed |E| times altogether

O(lg |V|) → **O(|E| log|V|)**

Decrease key → log|V|

Overall running time complexity
T(n) = O(|V| · log|V|+|E| · lg |V|)

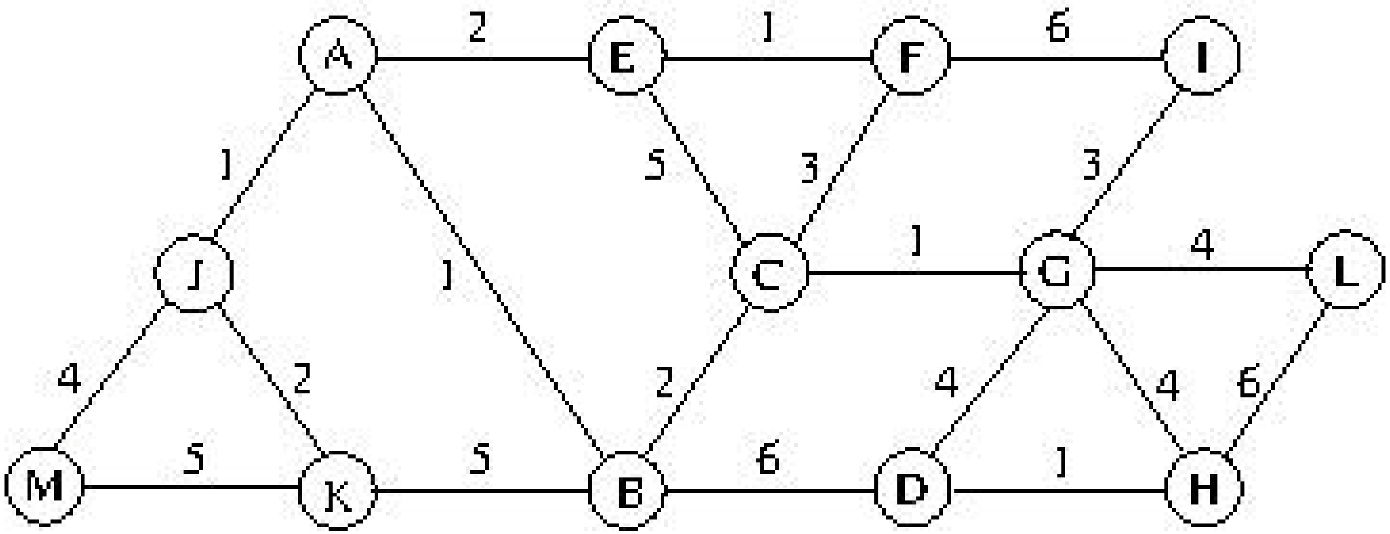
Complexity

- In general
  - $T(n) = O(|V| \cdot \lg |V| + |E| \cdot \lg |V|)$

  that is
  - $T(n) = O(|E| \cdot \lg |V|)$
- 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 $O(1)$
    - $T(n) = O(|E| + |V| \cdot \lg |V|)$

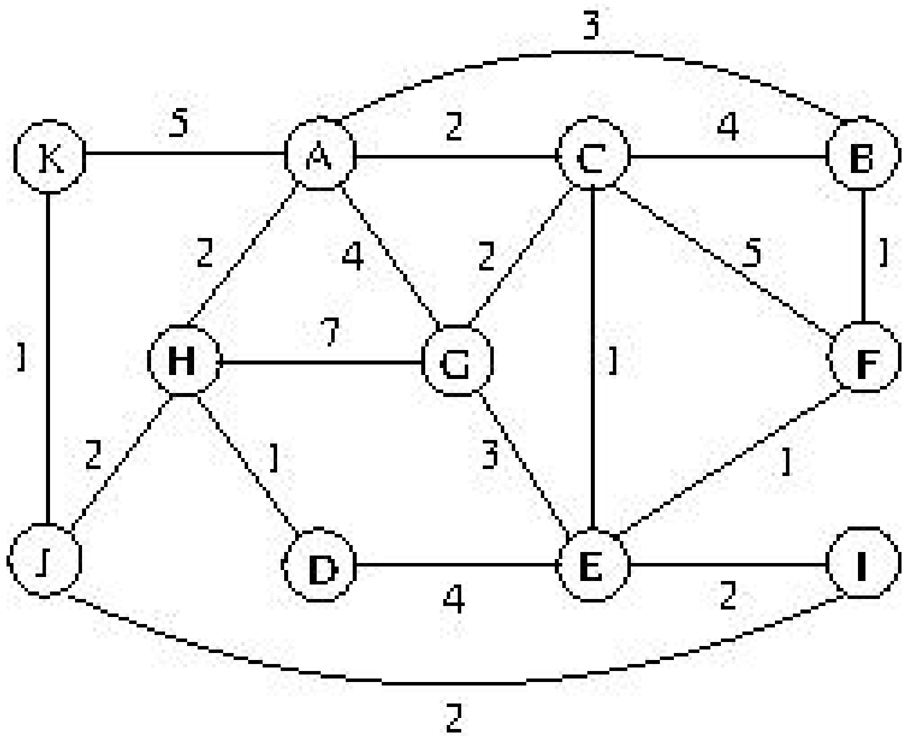
Exercise

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



Exercise

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



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)
  - (u,v) is a light edge connecting C to another component of G_A
- Then
  - Edge (u,v) is **safe** for A

Kruskal's Algorithm

- Algorithm proposed by Joseph Kruskal 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 = ϕ
  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

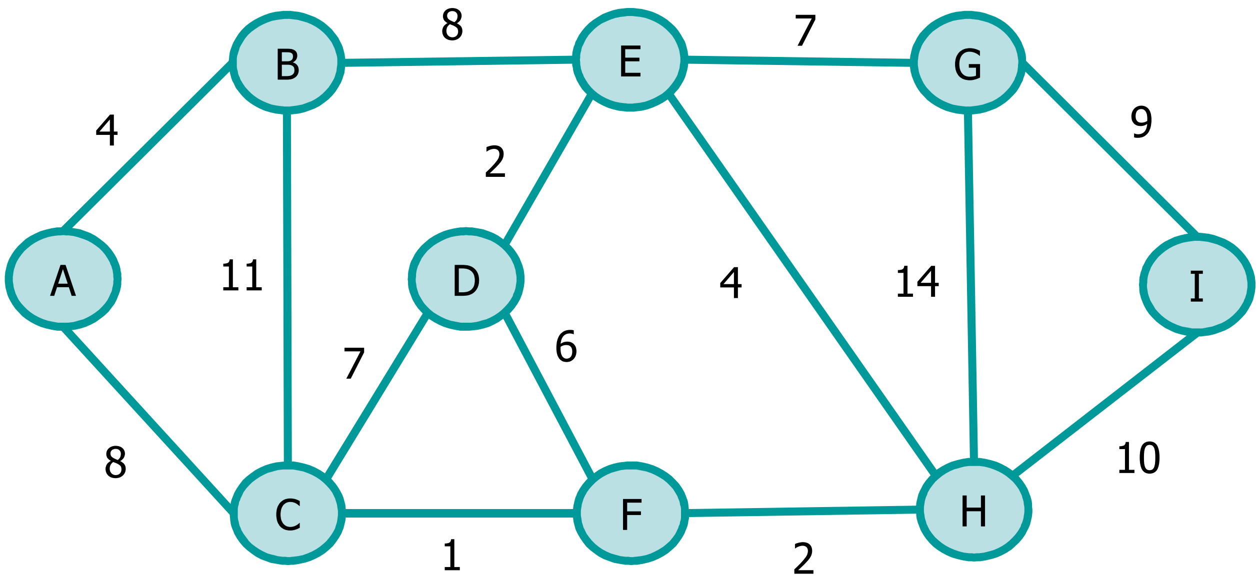
For each v create a set

taken in nondecreasing order by weight

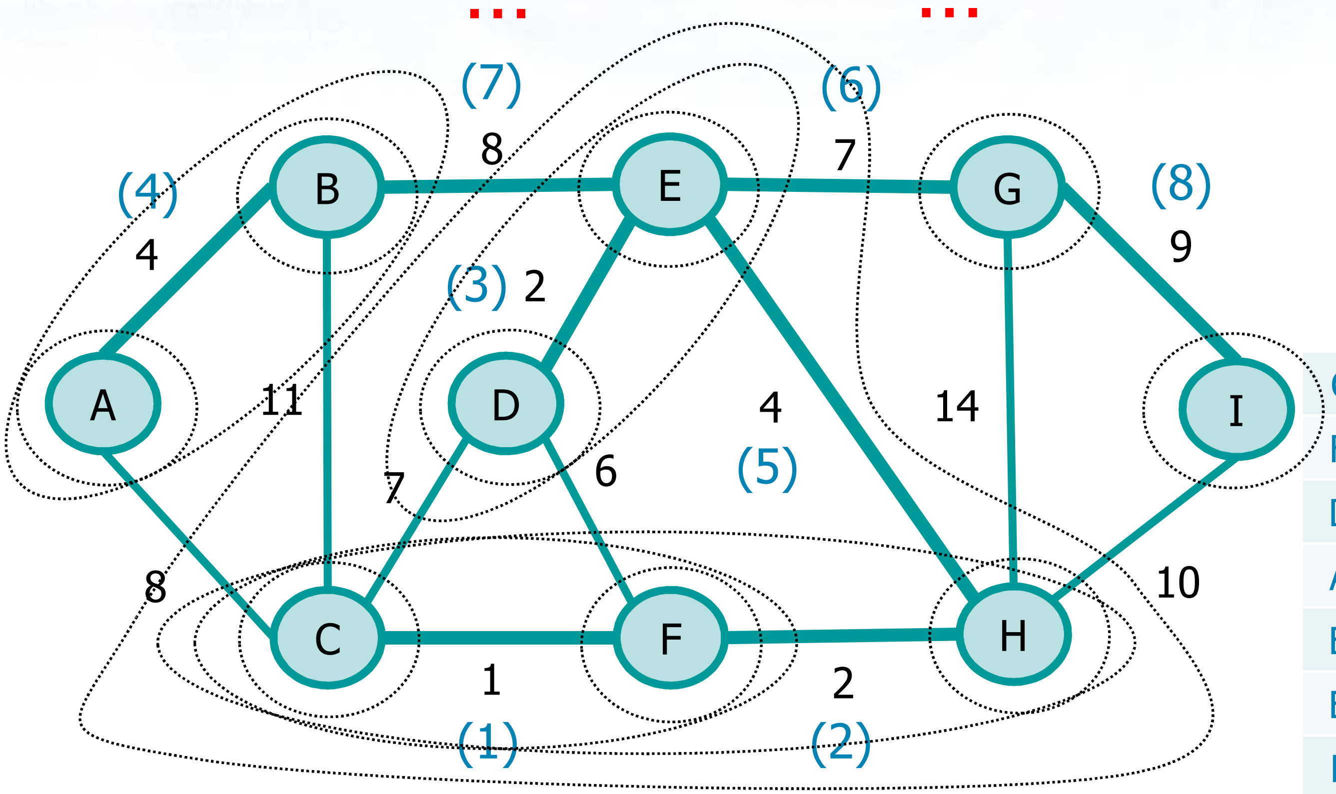
Find representative of u and v

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

Find

Union

```
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;
}
```

Complexity

```
mst_Kruskal (G, w)
  A = φ
  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) → **O(|V|)**

O(|E| lg |E|)

Executed E times

Union and find takes O(lg |E|)
→ **O(E log |E|)**

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

Complexity

- In general
  - $T(n) = (|E| \cdot \lg |E|)$
- Asintotically, for dense graph, Prim is more efficient than Kruskal
  - Prim
    - $T(n) = (|E| + |V| \cdot \lg |V|)$
  - Kruskal
    - $T(n) = (|E| \cdot \lg |E|)$

For dense graph

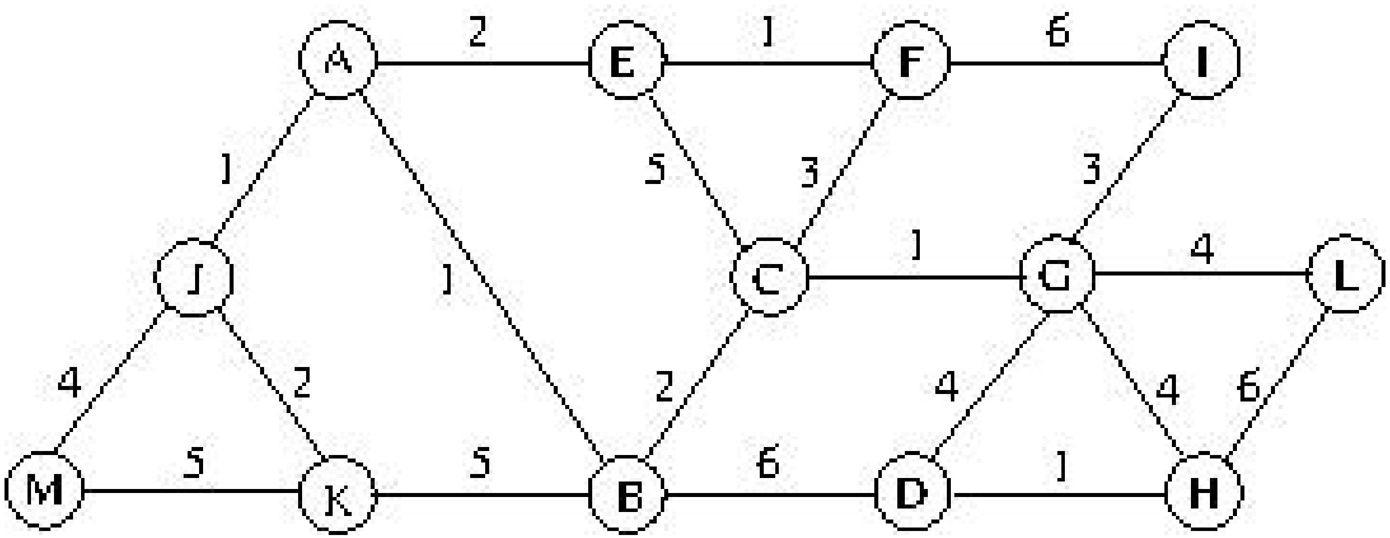
$$E = \frac{|V| \cdot (|V| - 1)}{2}$$

then

$|E| > |V|$

Exercise

- Given the following graph apply Kruskal's greedy algorithm



Exercise

- Given the following graph apply Kruskal's greedy algorithm