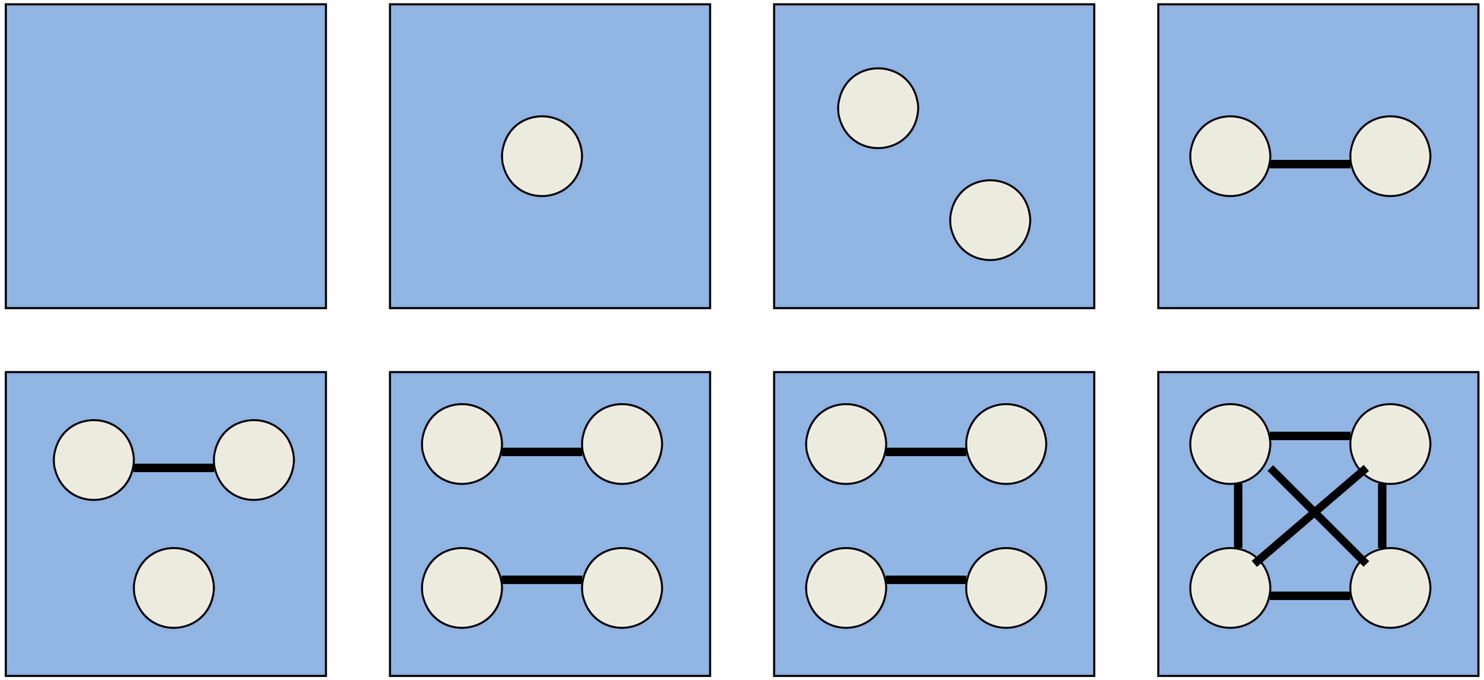
Undirected Graphs

- An **undirected graph** G = (V, E)
  - V a set of vertices
  - E a set of **unordered edges** (v,w) where v, w in V

- USE: to model **symmetric** relationships between entities
- vertices v and w are **adjacent** if there is an edge (v,w) **[or (w,v)]**
- the edge (v,w) is **incident** upon vertices v and w
- an edge may be (v,w,c) where c is a **cost component** **(e.g. distance)**

Examples



Terminology

PATH: a sequence of vertices v_1, v_2, …v_n such that (v_1 ,v_2), (v_2 ,v_3), … (v_{n-1} ,v_n) are edges

LENGTH: number of edges in a path
(v denotes a path length 0 from v to v)

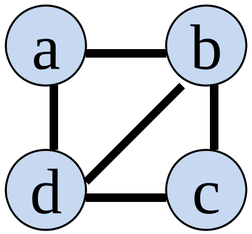
SIMPLE PATH: **all vertices are distinct**
(except possibly the first and the last)

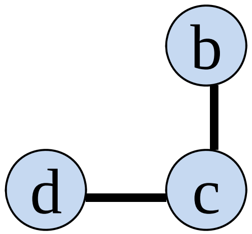
SIMPLE CYCLE: (undirected graph) a simple path of **length 3 or more** that connects a vertex to itself

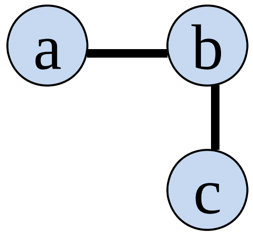
Sub-graph

- G = (V, E)
- a **sub-graph** of G is a graph G' = (V', E') where
  - V' is a subset of V
  - E' consists of edges (v,w) such that both v and w are in V'
- if E' consists of all edges (v,w) in E such that both v, w in V' then G' is an **INDUCED SUB-GRAPH** of G
- a **connected component** of a graph G is a maximal connected induced sub-graph that is not itself a proper sub-graph of any other connected sub-graph of G

Sub-graph: example







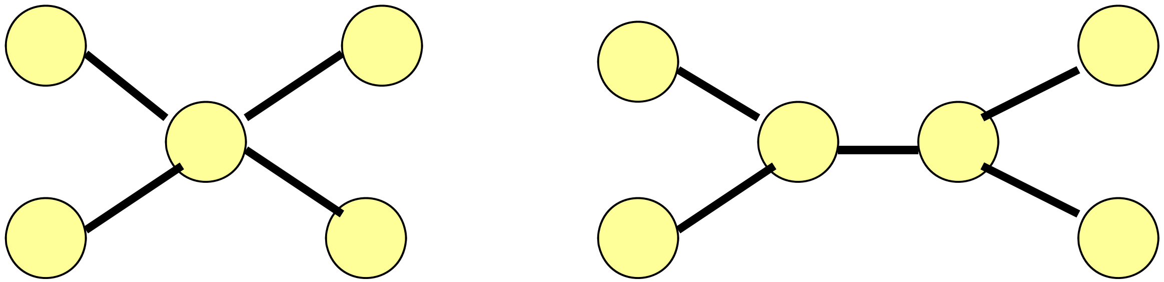
graph G

sub-graph G’

(an) induced sub-graph

One connected component - namely G itself

An Unconnected Graph

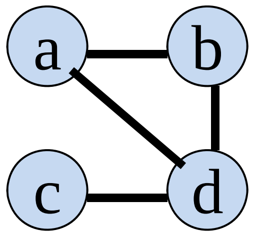


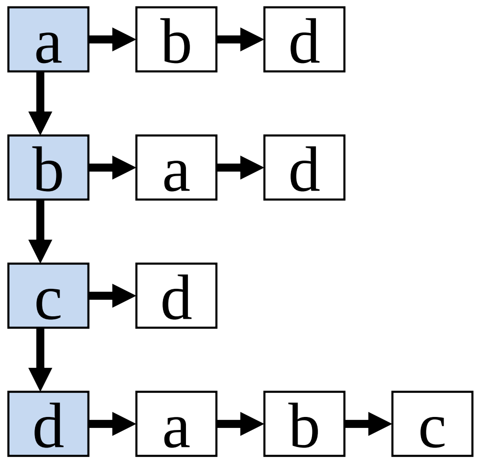
- **two connected components** (each a free tree)
- connected acyclic graph is a FREE TREE
  - every free tree with n >=1 vertices contains exactly (n-1) edges
  - any edge added to a free tree gives a cycle

Graph Representation

- Adjacency Matrix
- Adjacency List

	a	b	c	d
a	0	1	0	1
b	1	0	0	1
c	0	0	0	1
d	1	1	1	0



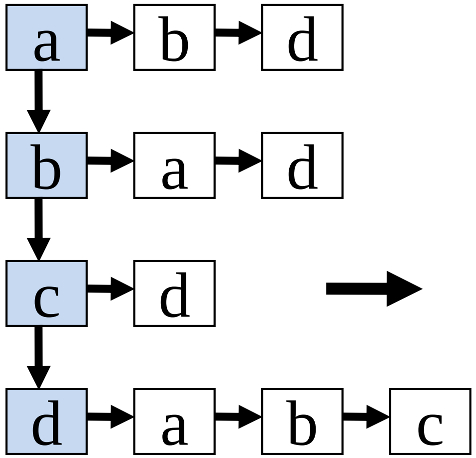


Operations

insert
remove
find

vertex
edge

navigate



list
operations

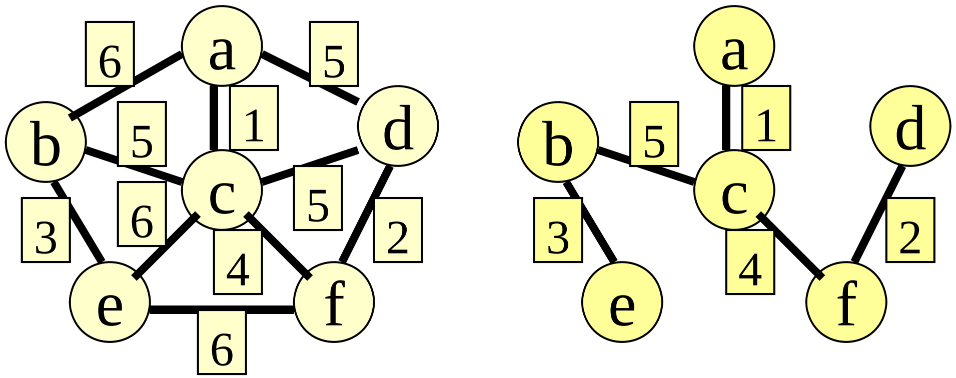
is_path
is_cycle

shortest
path

spanning
forest

Minimum-cost Spanning Trees

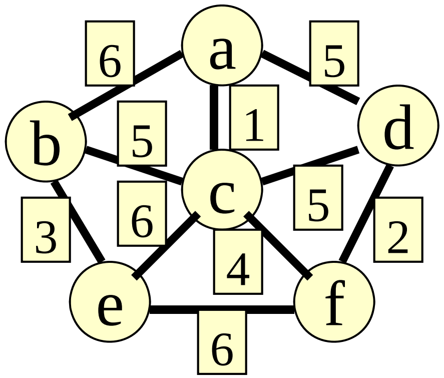
- G = (V,E) where each edge (v,w) has an associated cost
- a **SPANNING TREE** for G is a **free tree** that connects all the vertices in G **(n nodes and (n-1) edges; no cycles)**
- the **cost of the spanning tree** is the sum of the costs of the edges in the tree

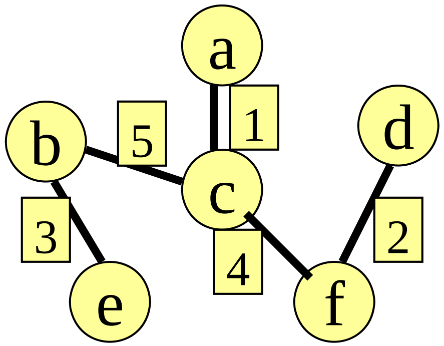


- **application areas**: communication networks (transport/computer)

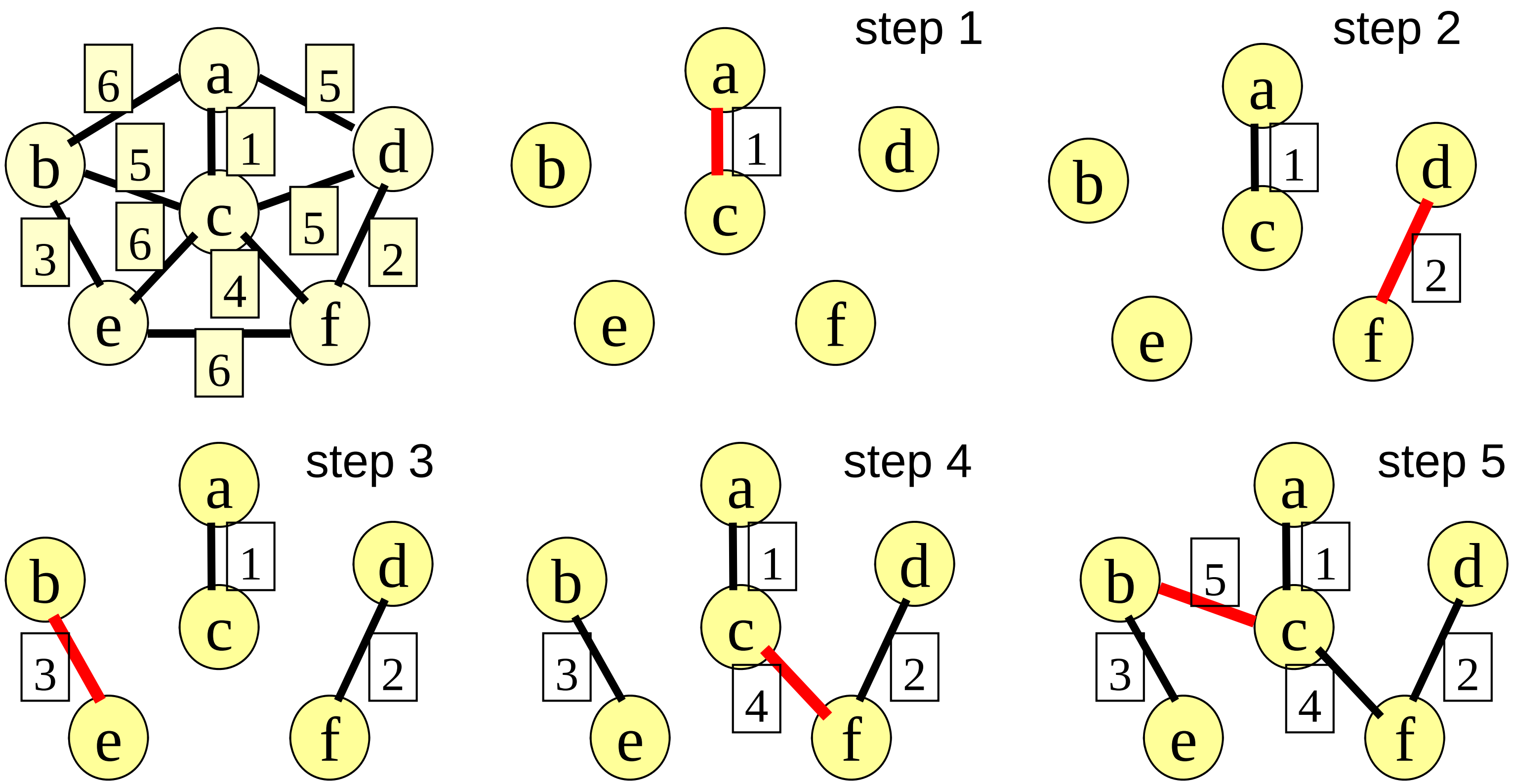
MST Property

- G = (V,E)
  - a **connected graph** with a cost function on the edges
  - let U be a proper subset of V
  - if (u,v) is an edge of lowest cost such that
    - **u in U and v in V-U then there is a MST that includes (u,v) as an edge**





Building an MST: creative guess §1



Kruskal's principles

- **Build a priority queue (PQ) with the edges, shortest edges first**
- **Each node in the graph becomes a component**
- **Choose an edge from the PQ such that the edge connects 2 distinct components until there is only one component – this is the MST**

Kruskal's principles - example

PQ: **(a c 1), (d f 2), (b e 3), (c f 4), (a d 5), (b c 5), (c d 5), (a b 6), (c e 6), (e f 6)**

Components: [a], [b], [c], [d], [e], [f] - 6 components

- **(a c 1) ➔ [a-c], [b], [d], [e], [f] - 5 components**
- **(d f 2) ➔ [a-c], [b], [d-f], [e] - 4 components**
- **(b e 3) ➔ [a-c], [b-e], [d, f] - 3 components**
- **(c f 4) ➔ [a-c, c-f, f-d], [b-e] - 2 components**
- **(a d 5) ➔ not chosen - a & d in same component**
- **(b c 5) ➔ [a-c, c-b, b-e, c-f, f-d] - 1 component (MST)**

MST – explanation (Kruskal)

priority queue

a c 1

d f 2

b e 3

c f 4

~~a d 5~~

b c 5

~~c d 5~~

~~a b 6~~

~~c e 6~~

~~e f 6~~

Comments

- The edges are stored in a PQ (lowest values first)
- Each node becomes a component
- Each edge should connect 2 components
- NB: **a d 5** does not connect 2 components
  - a and d are in the same component (step 5 above)
  - adding **a d 5** would also create a cycle
  - An MST is a free tree and therefore has no cycles
- **b c 5** completes the MST
- An MST with n nodes has (n-1) edges
- An MST is a **Free Tree** (no cycles)

Kruskal’s Algorithm (creative guess §1)

- One method of constructing an MST is **Kruskal’s**
- start with a graph T = (V, ¤) i.e. only the vertices of G = (V, E)
- **each vertex is a connected component** (in the graph T)
- to construct the MST, T examine the edges in E in order of increasing cost **(implementation - priority queue)**
- **if the edge connects two vertices in two connected components then add the edge to T** (otherwise discard the edge)
- when all the edges are in one component, T is a MST for G

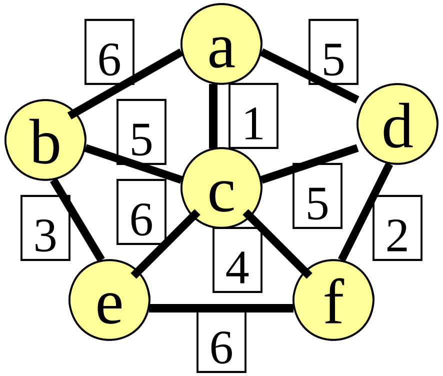
Kruskal's Algorithm

- S = set of connected components (V from G=(V,E))
- **merge(A, B, S)** -- merge components A & B in S - rename A
- **find(v, S)** -- return name of component X in S : v in X
- **initial('A', v, S)** -- make A the name of component in S containing only vertex v initially
- **insert(e, S)** -- add a given edge to S
- **remove_pq()** -- remove an edge from the PQ
- **(x, y, c)** -- edge (x, y) in PQ with cost c

Kruskal's Algorithm

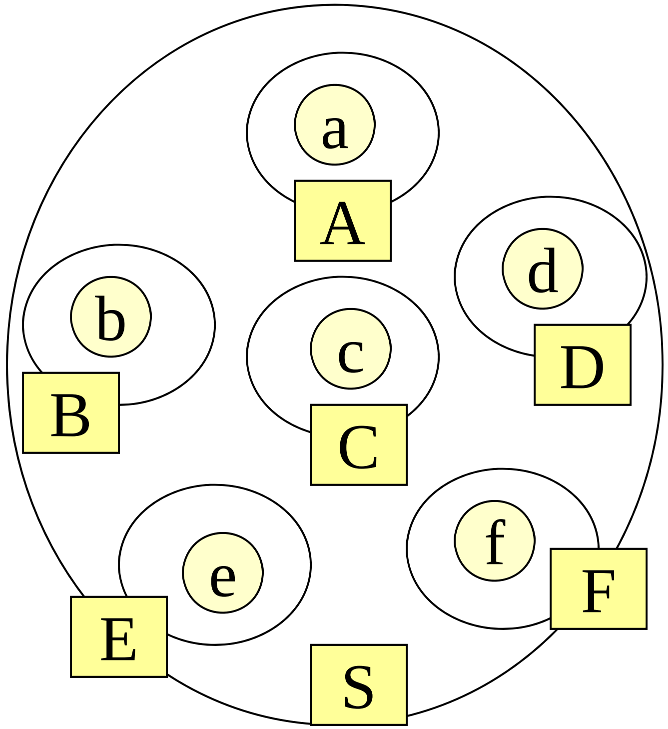
```
for each v in S initial ( next(name), v, S) -- initiliase
while (size(S) > 1 {            -- size = number of components
    get_PQ ( );                 -- get (x, y, c) from PQ
    if ( find(x, S) != find(y, S) ) { -- x, y in different components
        merge ( find (x, S), find (y, S), S );
        insert (get_PQ ( ), S);
        }
    remove_pq( );
    }
```

Kruskal: example



a	**c**	**1**
d	**f**	**2**
b	**e**	**3**
c	**f**	**4**
a	d	5
b	**c**	**5**
c	d	5
a	b	6
c	e	6
e	f	6

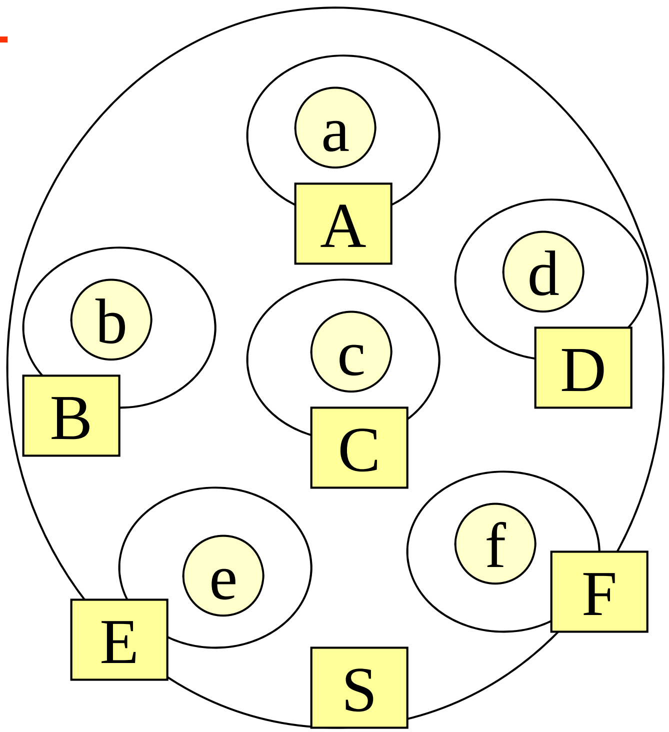
PQ

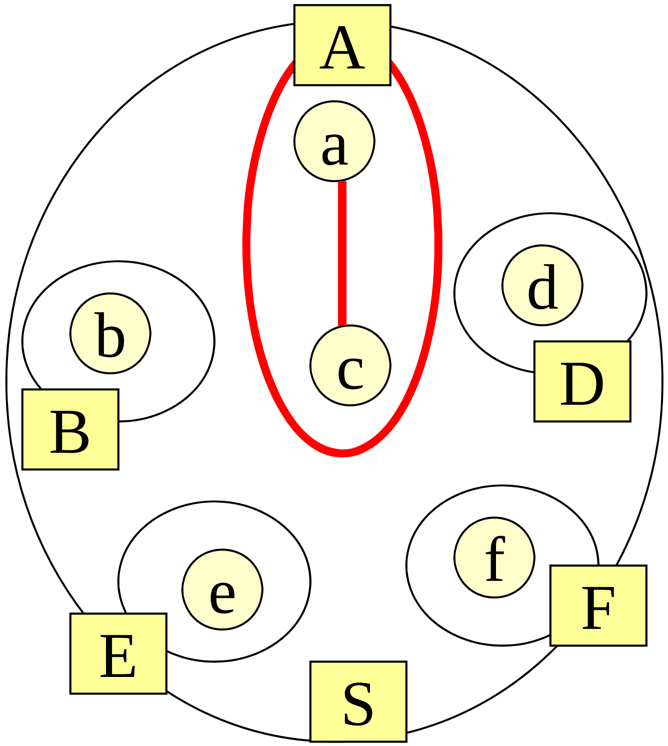


Kruskal: example

~~a~~	~~c~~	~~1~~
d	f	2
b	e	3
c	f	4
a	d	5
b	c	5
c	d	5
a	b	6
c	e	6
e	f	6

PQ

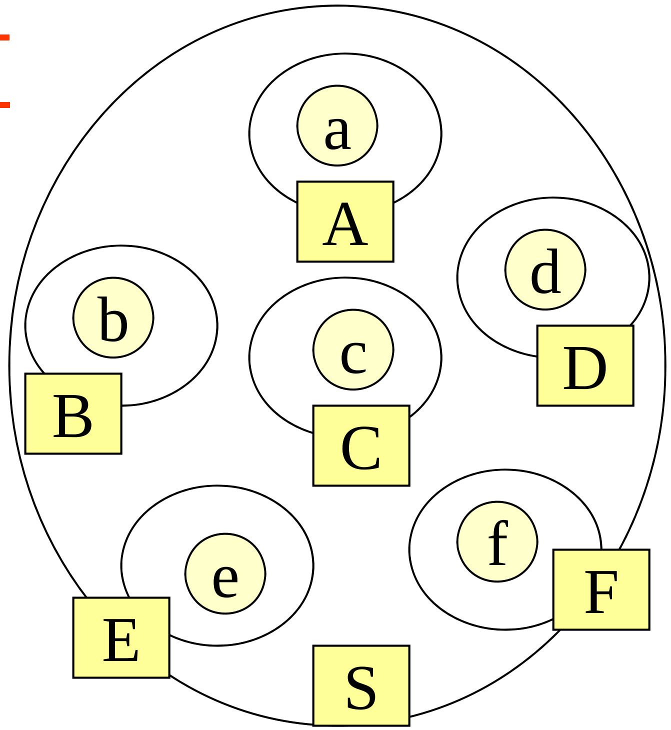


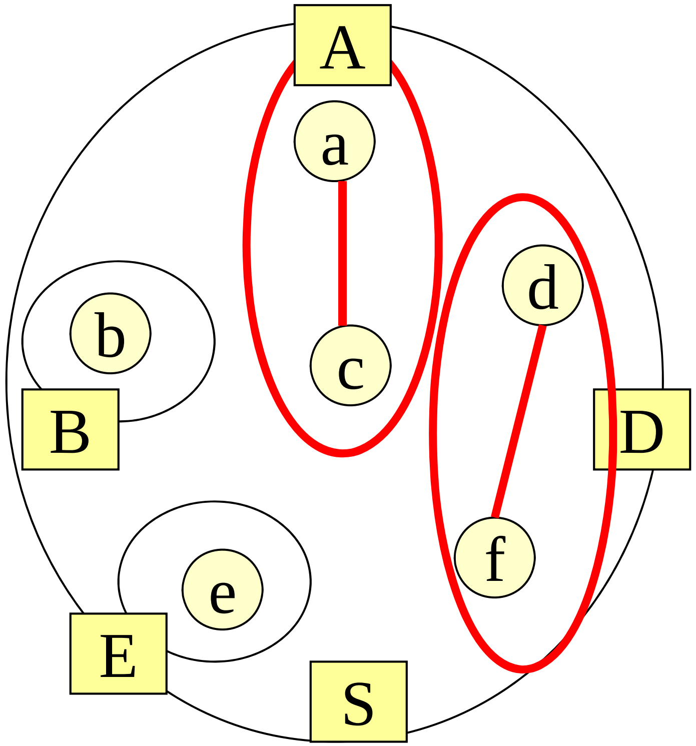


Kruskal: example

~~a~~	~~c~~	~~1~~
~~d~~	~~f~~	~~2~~
b	e	3
c	f	4
a	d	5
b	c	5
c	d	5
a	b	6
c	e	6
e	f	6

PQ

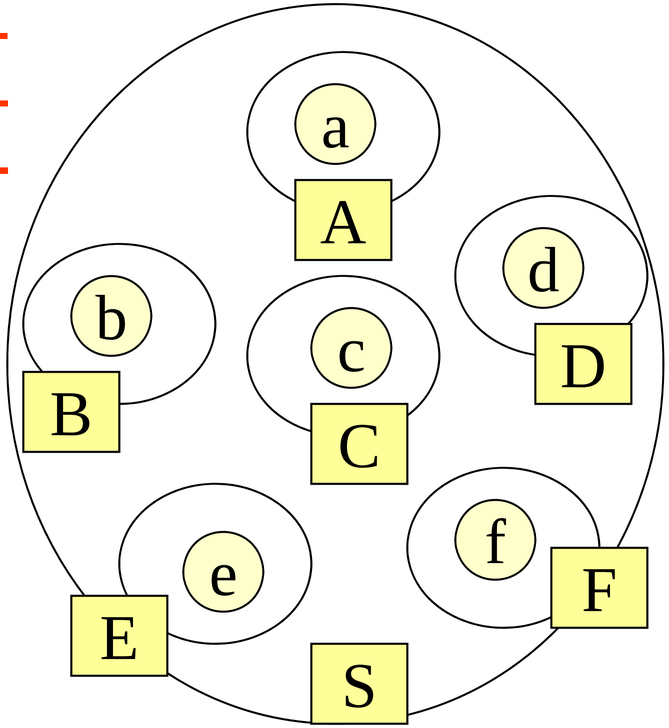


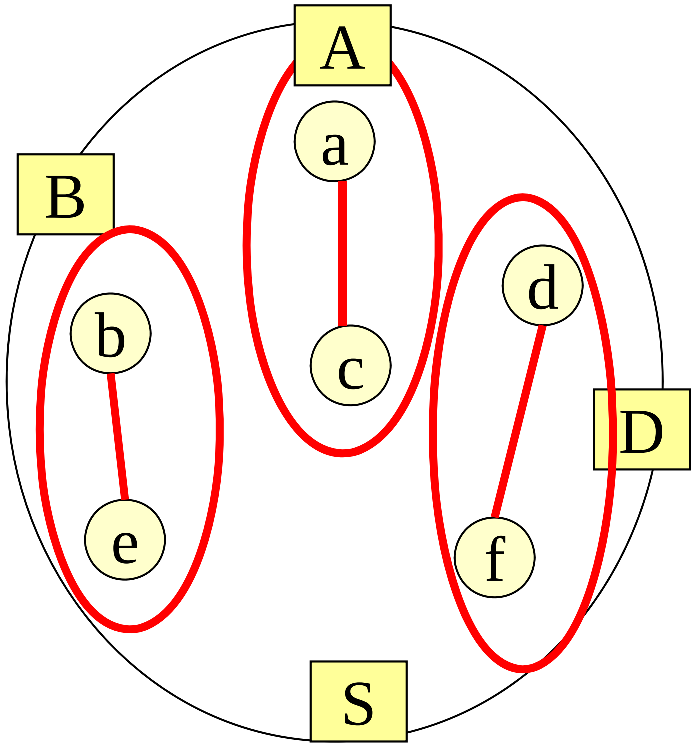


Kruskal: example

~~a~~	~~c~~	~~1~~
~~d~~	~~f~~	~~2~~
~~b~~	~~e~~	~~3~~
c	f	4
a	d	5
b	c	5
c	d	5
a	b	6
c	e	6
e	f	6

PQ

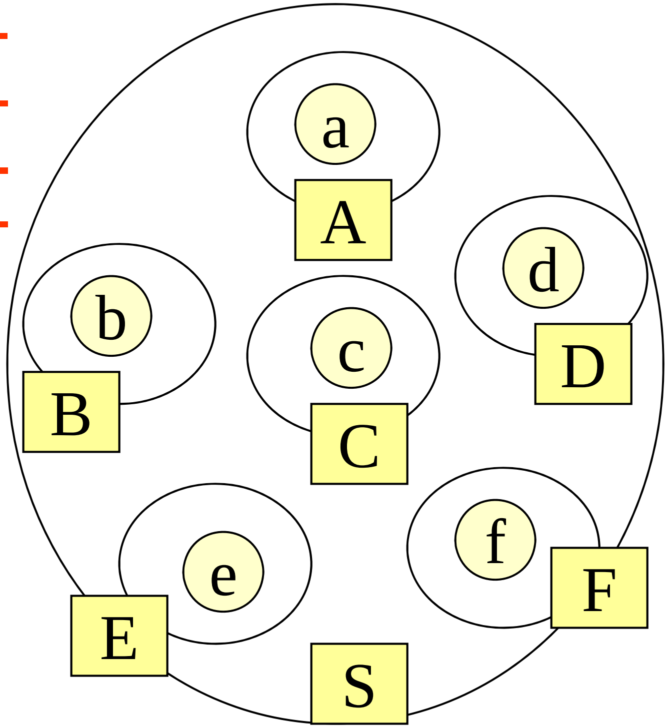


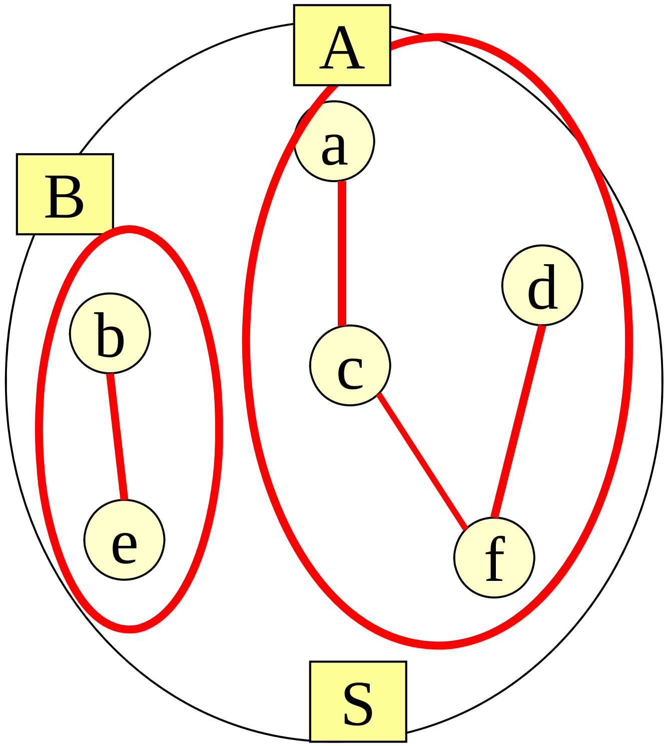


Kruskal: example

~~a~~	~~c~~	~~1~~
~~d~~	~~f~~	~~2~~
~~b~~	~~e~~	~~3~~
~~c~~	~~f~~	~~4~~
a	d	5
b	c	5
c	d	5
a	b	6
c	e	6
e	f	6

PQ

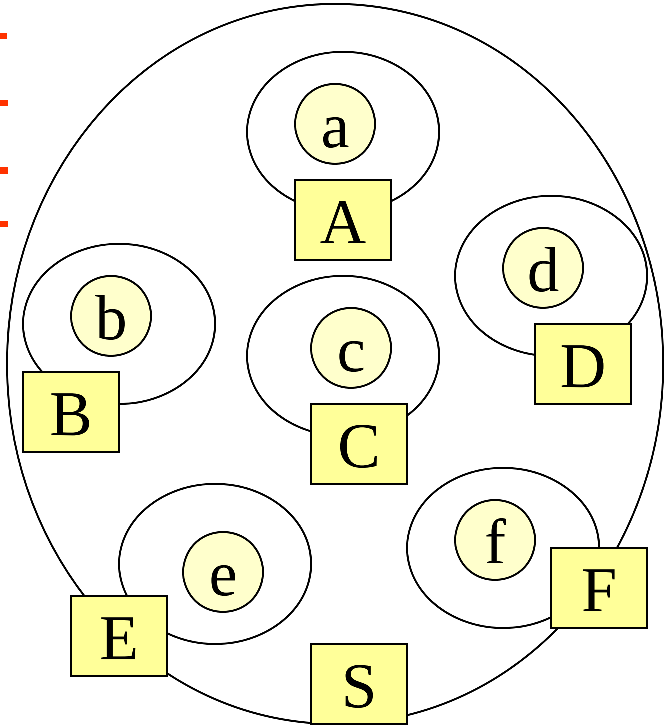


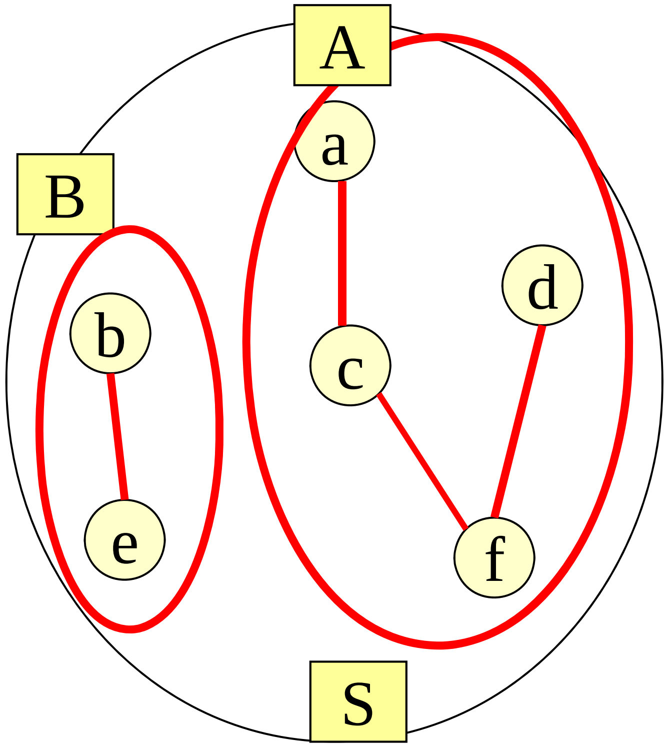


Kruskal: example

~~a~~	~~c~~	~~1~~
~~d~~	~~f~~	~~2~~
~~b~~	~~e~~	~~3~~
~~c~~	~~f~~	~~4~~
a	**d**	**5**
b	c	5
c	d	5
a	b	6
c	e	6
e	f	6

PQ

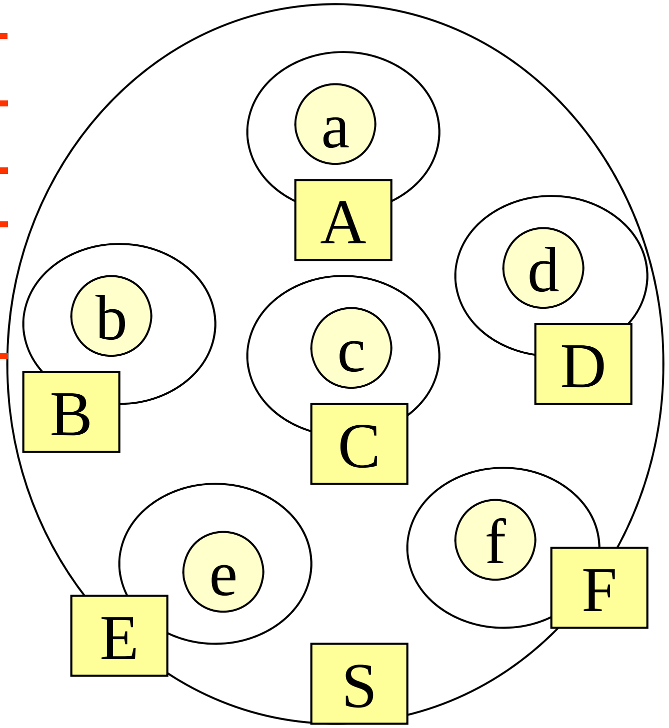


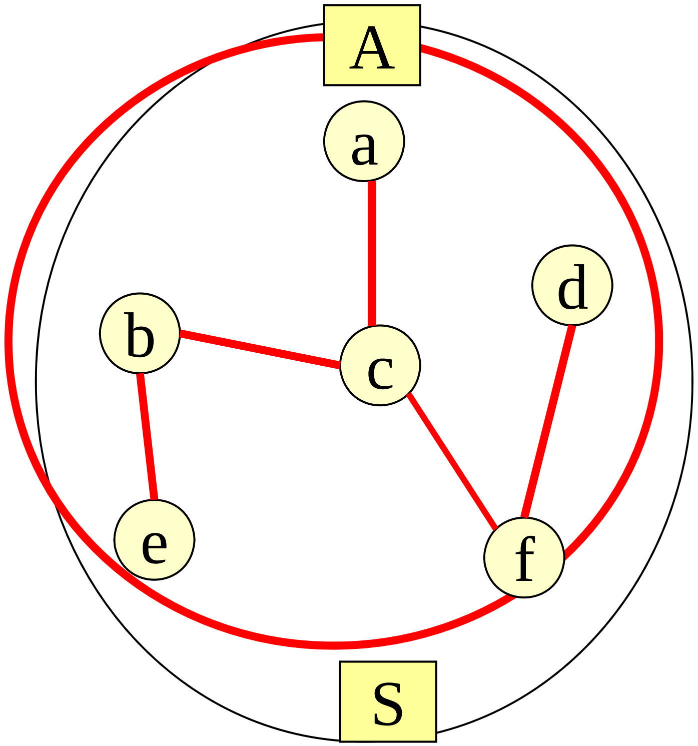


Kruskal: example

~~a~~	~~c~~	~~1~~
~~d~~	~~f~~	~~2~~
~~b~~	~~e~~	~~3~~
~~c~~	~~f~~	~~4~~
a	**d**	**5**
~~b~~	~~c~~	~~5~~
c	d	5
a	b	6
c	e	6
e	f	6

PQ



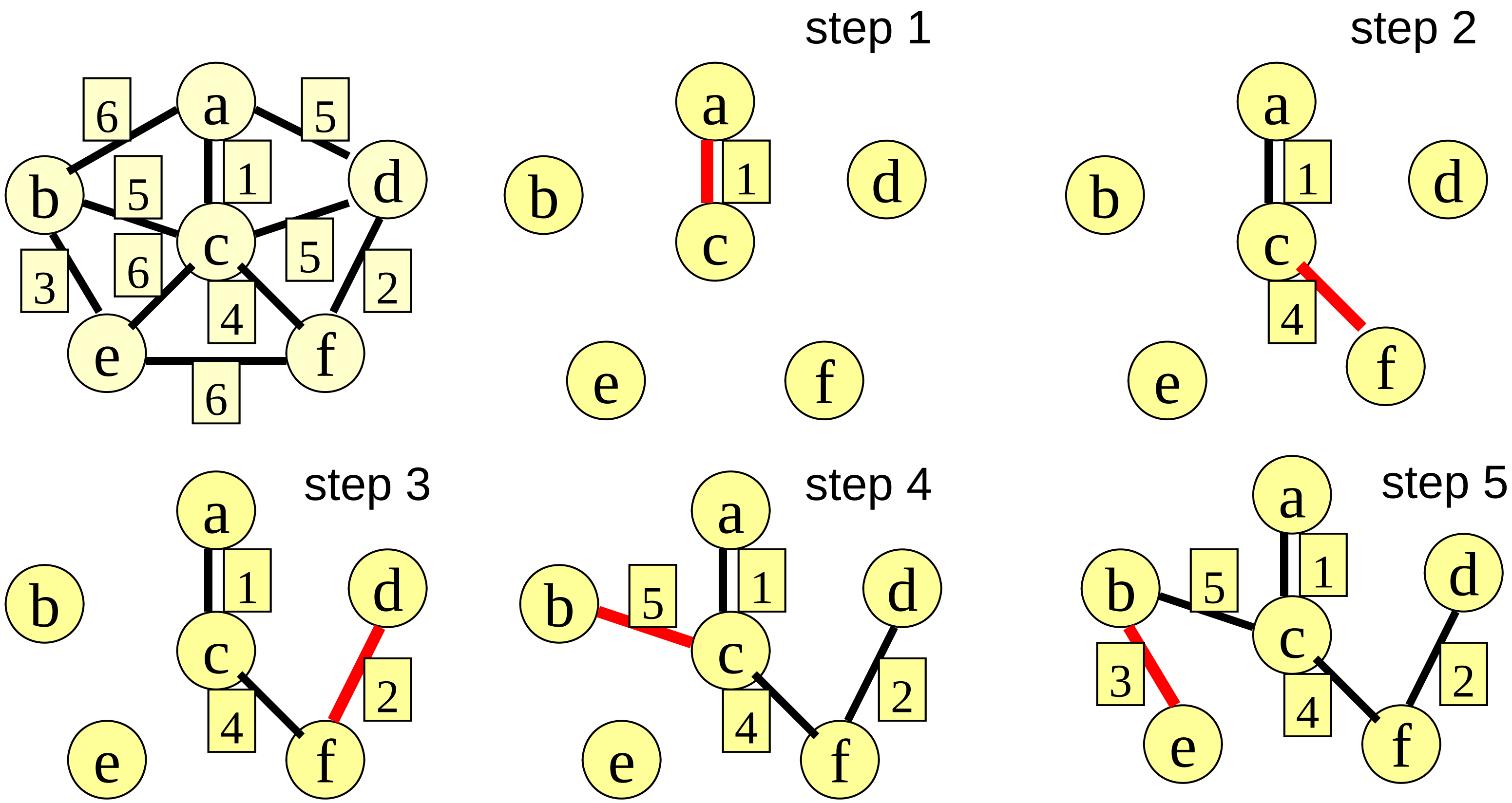


Kruskal: Comment

- Using the PQ, the algorithm is reasonably easy to understand in principle **(the pictorial representation is easy to follow)**

- In general it is worth looking at the problem and its solution before going through any algorithm in detail
- look at each line of the pseudo code and be sure that you can relate the code to the action required i.e. that you can **interpret** the code

Building an MST: creative guess §2



Prim's principles

- given start node x mark as visited;
- note the edge values from x to the remaining nodes; this uses 2 arrays L for the edge lengths and C for the node name;
- find the shortest edge from x to y; mark y as visited;
- build a COMPONENT (x y) i.e. y is then added to the component (i.e. the visited nodes);
- now examine the edge costs from y to the remaining nodes; if this edge is cheaper, replace the current edge with this edge. The new node is added to the component.
- Repeat for the unvisited nodes. The component grows node by node and cheaper edges replace those edges previously found as cheaper.

Prim's principles example

- **(a b 6), (a c 1), (a d 5), (b c 5), (b e 3), (c d 5), (c e 6), (c f 4), (d f 2), (e f 6)**
- Start node a – visited {a} – **unvisited {b, c, d, e, f}**
- **L = [6, 1, 5, §, §] C = [a, a, a, a, a]**
- Shortest edge **(a c 1)** – visited {a, c} – **unvisited {b, d, e, f}**
- (c b 5) is cheaper ➔ **L = [5, 1, 5, §, §] C = [c, a, a, a, a]**
- (c d 5) not cheaper ➔ no change
- (c e 6) is **cheaper** ➔ **L = [5, 1, 5, 6, §] C = [c, a, a, c, a]**
- (c f 4) is **cheaper** ➔ **L = [5, 1, 5, 6, 4] C = [c, a, a, c, c]**

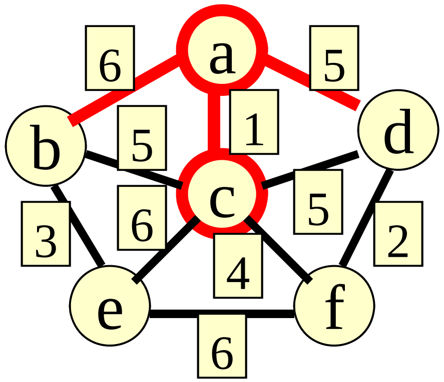
Prim’s principles example

- **(a b 6), (a c 1), (a d 5), (b c 5), (b e 3), (c d 5), (c e 6), (c f 4), (d f 2), (e f 6)**
- **L = [5, 1, 5, 6, 4] C = [c, a, a, c, c]**
- Shortest edge **(c f 4)** – visited {a, c, f} – **unvisited {b, d, e}**
- (f b §) not cheaper ➔ no change
- (f d 2) is **cheaper ➔** **L = [5, 1, 2, 6, 4] C = [c, a, f, c, c]**
- (f e 6) not cheaper ➔ no change
- Shortest edge **(f d 2)** – visited {a, c, d, f} – **unvisited {b, e}**
- (d b §) not cheaper ➔ no change
- (d e §) not cheaper ➔ no change
- **L = [5, 1, 2, 6, 4] C = [c, a, f, c, c]**

Prim's principles example

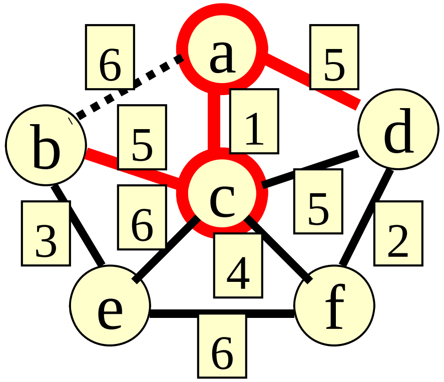
- **(a b 6), (a c 1), (a d 5), (b c 5), (b e 3), (c d 5), (c e 6), (c f 4), (d f 2), (e f 6)**
- **L = [5, 1, 2, 6, 4] C = [c, a, f, c, c]**
- Shortest edge **(c b 5)** – visited {a, b, c, d, f} – **unvisited {e}**
- (b e 3) is cheaper ➔ **L = [5, 1, 2, 3, 4] C = [c, a, f, b, c]**
- Shortest edge **(b e 3)** – visited {a, b, c, d, e, f} – **unvisited {}** empty – STOP
- Result **L = [5, 1, 2, 3, 4] C = [c, a, f, b, c]**

Prim’s principles - pictures



L = [6, 1, 5, §, §]

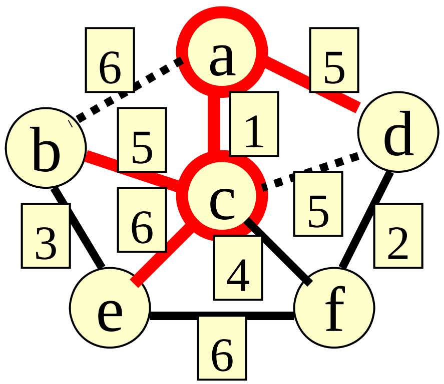
C = [a, a, a, a, a]



L = [5, 1, 5, §, §]

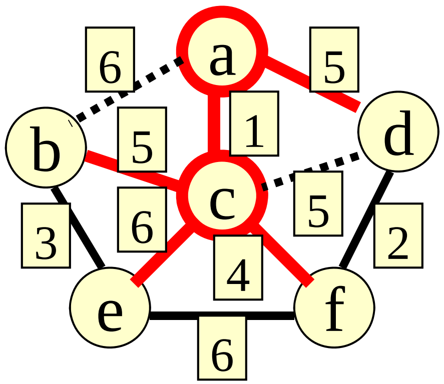
C = [c, a, a, a, a]

Prim’s principles - pictures



L = [5, 1, 5, 6, §]

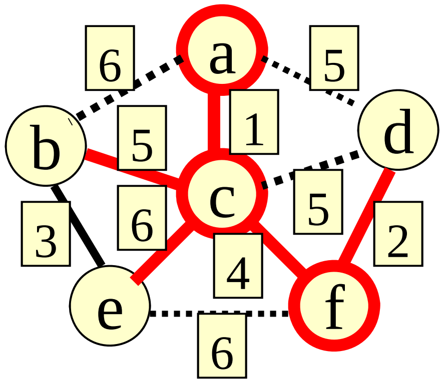
C = [c, a, a, c, a]



L = [5, 1, 5, 6, 4]

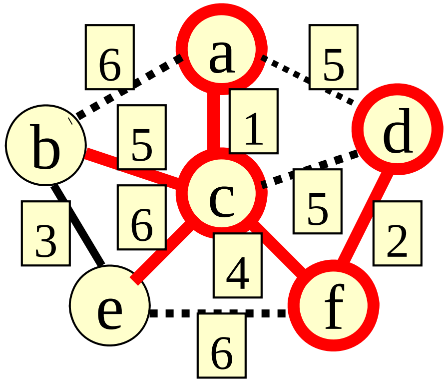
C = [c, a, a, c, c]

Prim's principles - pictures



L = [5, 1, 2, 6, 4]

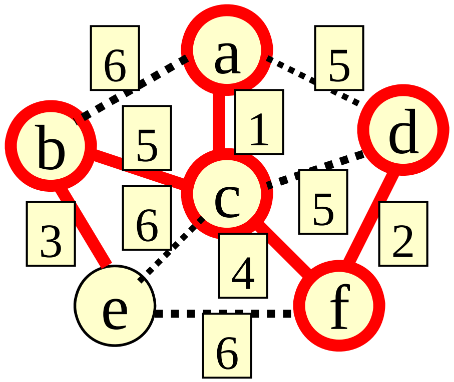
C = [c, a, f, c, c]



L = [5, 1, 2, 6, 4]

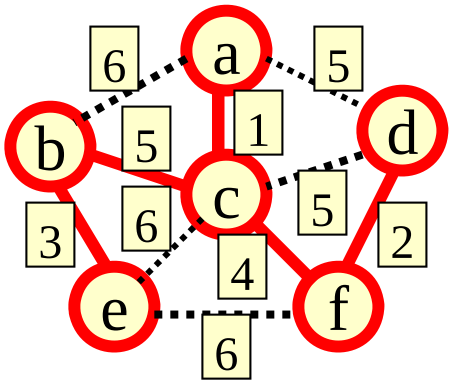
C = [c, a, f, c, c]

Prim's principles - pictures



L = [5, 1, 2, 3, 4]

C = [c, a, f, b, c]



L = [5, 1, 2, 3, 4]

C = [c, a, f, b, c]

MST – explanation (Prim)

- Prim's algorithm is a **greedy** algorithm
  - greedy = takes the locally best solution
  - The MST "grows" the MST as one component (similar to Dijkstra)
- Process
  - Choose the **cheapest edge** from the component to an **unvisited node**, add edge to the MST and **mark the node as visited (U)**
  - **Start at node a** – choose the cheapest edge **a c 1** **mark c**
  - Now choose the cheapest edge U = {a,c} **c f 4** **mark f**
  - Now choose the cheapest edge U = {a,c,f} **f d 2** **mark d**
  - Now choose the cheapest edge U = {a,c,f,d} **c b 5** **mark b**
  - Now choose the cheapest edge U = {a,c,f,d,b} **b e 5** **mark e**
  - All nodes have now been visited U = {a,c,f,d,b,e} **stop**.

Prim's Algorithm (creative guess §2)

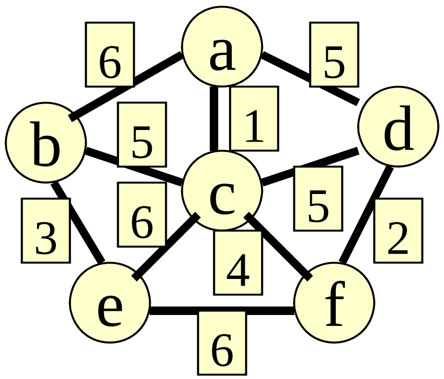
- V = {a,b,c,d,…}
- initialise U to {a}
- **the spanning tree grows one edge at a time**
- each step:
  - find the shortest **edge** (u,v) that connects U and V-U
  - add v to U
  - until U = V   i.e. V-U = ¤
- cost matrix C gives the costs of each edge

Prim’s Algorithm

```
Prim ( node v)                                          -- v is the start node
    { U = {v};  for i in (V-U) { low-cost[i] = C[v,i]; closest[i] = v; }
    while (!is_empty (V-U) ) {                          -- find the closest vertex in V-U
        i = first(V-U); min = low-cost[i]; k = i; -- minimum cost edge
        for j in (V-U-k) if (low-cost[j] < min) {min = low-cost[j]; k = j; }
        display(k, closest[k]);                         -- display edge
        U = U + k;                                      -- k added to U
        for j in (V-U) if ( C[k,j] < low-cost[j] ) )    -- readjust costs
                {low-cost[j] = C[k,j]; closest[j] = k; }
        }
    }
```

See http://www.cs.kau.se/cs/education/courses/dvgb03/revision/index.php?PrimEx=1

Prim: example



	a	b	c	d	e	f
a	§	6	1	5	§	§
b	6	§	5	§	3	§
c	1	5	§	5	6	4
d	5	§	5	§	§	2
e	§	3	6	§	§	6
f	§	§	4	2	6	§

§	infinity

Prim: example

Init: U	V-U	low-cost	closest	k / min
{a}	{b,c,d,e,f}	(-,6, 1, 5, §, §)	(-,a,a,a,a,a)	c / 1
display ((a,c))				
{a,c}	{b,d,e,f}	(-,5, 1, 5, 6, 4)	(-,c,a,a,c,c)	f / 4
display ((c,f))				
{a,c,f}	{b,d,e}	(-,5, 1, 2, 6, 4)	(-,c,a,f,c,c)	d / 2
display ((f,d))				
{a,c,f,d}	{b,e}	(-,5, 1, 2, 6, 4)	(-,c,a,f,c,c)	b / 5
display ((c,b))				
{a,c,f,d,b}	{e}	(-, 5, 1, 2, 3, 4)	(-,c,a,f,b,c)	e / 3
display ((b,e))				
{a,c,f,d,b,e}	{ }	(-, 5, 1, 2, 3, 4)	(-,c,a,f,b,c)	

See http://www.cs.kau.se/cs/education/courses/dvgb03/revision/index.php?PrimEx=1

Prim: Comment

- Since the MST is a free tree, there are n-1 edges, hence n-1 iterations
- the following code finds the least cost edge between U and V-U **(min, k)**

  min = low-cost[i]; k = i;

  for j in (V-U-k) if (low-cost[j] < min) {min = low-cost[j]; k = j; }
- the following code may be replaced by for e.g. ***Add_MST(u,v)***

  display(k, closest[k]); -- **display edge**
- the re-adjustment of the costs is perhaps the trickiest to understand but is in effect similar to Dijkstra - finding the cheapest (i,j) or (i,k) (k,j)

  for j in (V-U) if (C[k,j] < low-cost[j]))

  {low-cost[j] = C[k,j]; closest[j] = k; }