COMP20003
Algorithms and Data Structures

Greedy Algorithms and the Minimum Spanning Tree (MST)

Nir Lipovetzky
Department of Computing and Information Systems
University of Melbourne
Semester 2

Greedy Algorithms

Greedy algorithms are used in optimization problems

Greedy algorithms keep taking the next best step repeatedly, until the best solution is reached

- Dijkstra's algorithm is greedy: takes the next best edge to add to the path tree

Minimum Spanning Tree

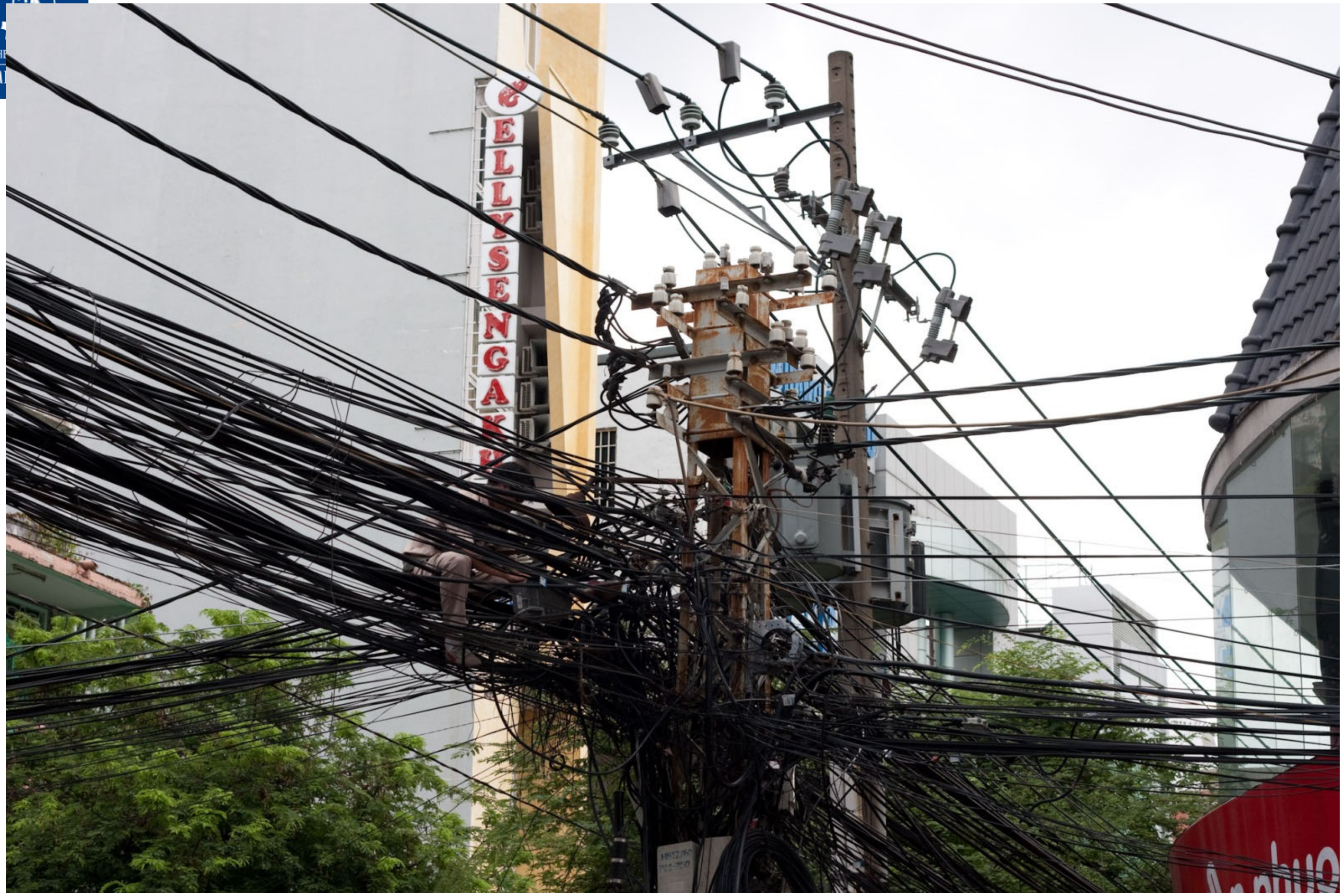
Undirected weighted graphs

Minimum spanning tree = subgraph that is:

- A tree (no cycles)
- Contains every vertex (spans)
- Minimum sum of edge weights

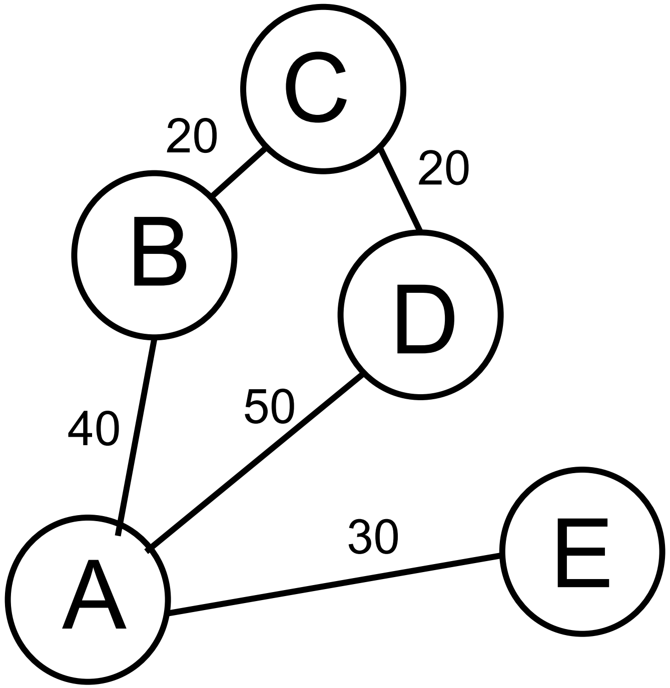
Also called:

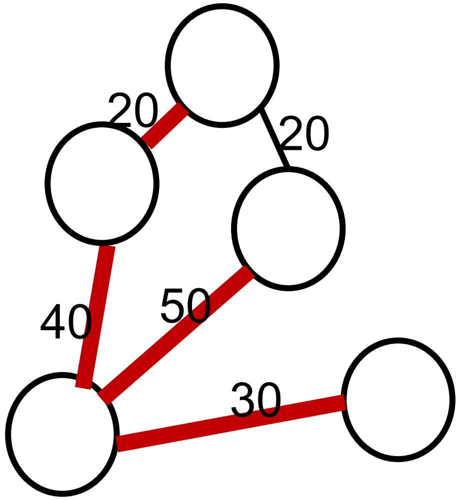
- Minimum weight spanning tree (sum of weights)
- Minimal spanning tree (might be more than one)



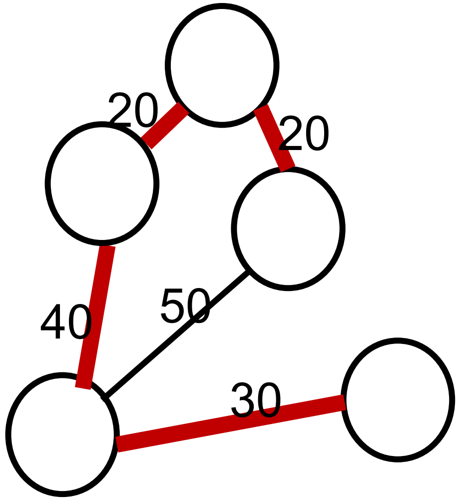
Reena Mahtani
Creative Commons License

MST *vs.* shortest path





Shortest paths



Min Span Tree

MST and Graph characteristics

Graph must be connected

MST *must* have exactly *V-1* edges

No cycles in MST

Building a MST: General approach

Start with isolated vertices (all), **no edges**

Begin with **any vertex** (Prim's) **or** the **least cost edge** (Kruskal's)

- This is a MST subtree

Keep adding vertices/edges to **extend** this MST **subtree**

- Shortest connections
- No cycles

Famous MST algorithms

- Prim's
  - **Shortest connection networks and some generalizations**. R.C. Prim, *Bell System Technical Journal* ***36****(6),* 1389-1401, 1957.
- Kruskal's
  - **On the shortest spanning subtree of a graph and the traveling salesman problem**. J.B. Kruskal, *Proceedings of the American Mathematical Society* **7**, 48-50, 1956.
- Borůvka's (1926, published in Czech)
  - **Otakar Borůvka on minimum spanning tree problem: translation of both the 1926 papers, comments, history**. Nešetřil, Jaroslav; Milková, Eva; Nešetřilová, Helena (2001). *Discrete Mathematics* **233** (1–3): 3–36

Prim's MST algorithm

Preferred method for dense graphs

Easiest with matrix representation

Prim's algorithm relies on picking the next best edge that joins two set of vertices:

- Vertices already in the tree (S)
- Vertices not yet in the tree (V-S)

These two sets form a "cut"

Definitions

A Cut $(V,V-S)$ of G is a partition of V

Cross: an edge (u,v) in E with one endpoint in S and the other in $V-S$

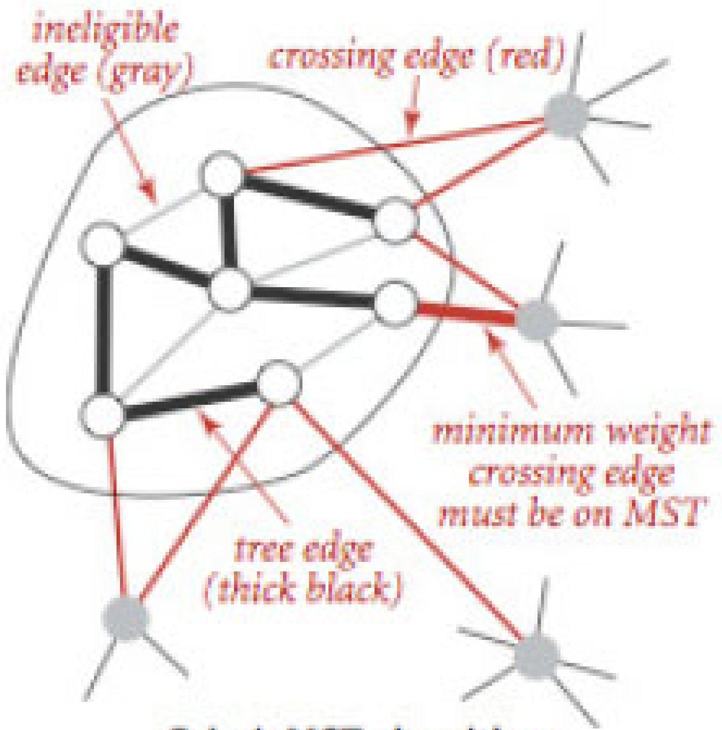
Light edge: the minimum weight edge crossing the cut

Respect: a cut respects a set A of edges if no edge in A crosses the cut

Cut during MST construction

Cut:

- *S*: set of vertices already **in** the MST
- *V-S*: **not** yet in the MST
  - Fringe: part of *V-S* **one step away** from the MST
  - **Vertices in *V-S*** have a cost (**distance**) from the MST subtree so far constructed
  - **Distances** between non-MST vertices and MST vertices **are updated** as vertices are added to MST



Prim's MST algorithm

From R. Sedgewick, Algorithms 4th edition

Prim's MST construction

Start:

- *S* = {any vertex}
- *S-V* = {all the others}
- The **cut** *S/V-S* **respects** edges **in** the **MST** as it is being constructed
- The cut itself changes

Prim’s MST construction

Respect:

- The cut S/V-S respects edges in the MST being constructed
  - –Fringe: vertices in V-S one step away from the MST
  - –Vertices in V-S have a cost(distance) from the MST subtree so far constructed (some may be ∞)

Prim’s MST construction

Pick **lightest** edge **crossing** the cut:

- Crossing edge *(u,v)* has *u* in *S* and *v* in *V-S*
  - Add v to S
  - Keep track of path (pred[ ])
  - Update distances between non-MST vertices and MST vertices (could be closer now) (w[ ])

Repeat until V-S = {0}

Reconstruct connections and distances from `pred`[ ] and `w`[ ]

Prim's: Pseudocode

```
void prim(G,w,root)
{
   for every u in V { dist[u] = ∞;  inmst[u] = FALSE;}
   dist[root] = 0; pred[root] = NULL;
   PQ = makePQ(V); /* all vertices in PQ */
   while(!empty PQ){
     u = deletemin(PQ);
     for every (v adjacent to u){
       if ((inmst[v]==FALSE)&& (w[u][v]< dist[v])){
            dist[v] = w[u][v]; /* update distance */
            decreasewt(PQ,v,dist[v]);/* update PQ dist*/
              pred[v]=u; /* update path information */
       }
     }
     inmst[u] = TRUE;
   }
 } /* at end: MST = {{v,pred[v]}: v in V - {root}} */
```

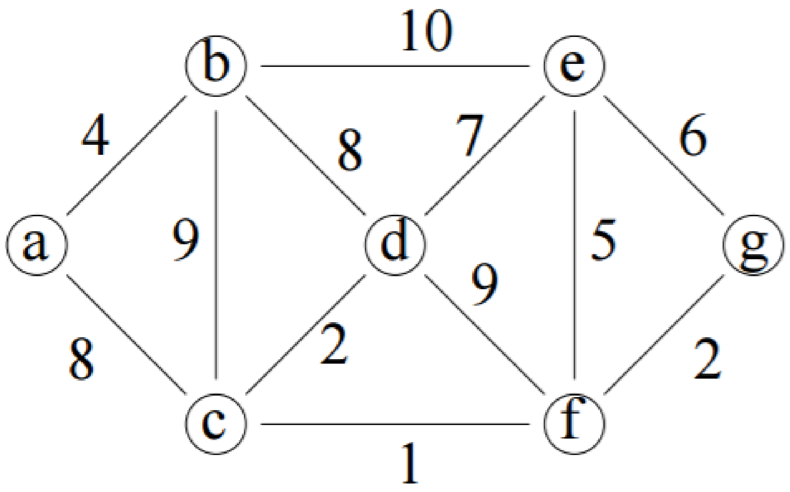
Fringe vertices are in a priority queue

This is a Priority-First Search

Prim's example

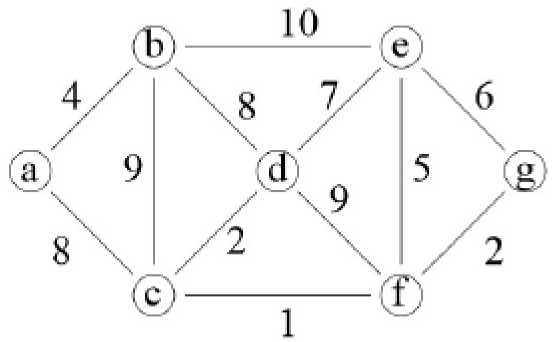
	a	b	c	d	e	f	g	
Dist = [								]
pred = [								]
inMST =[								]

PQ = { }

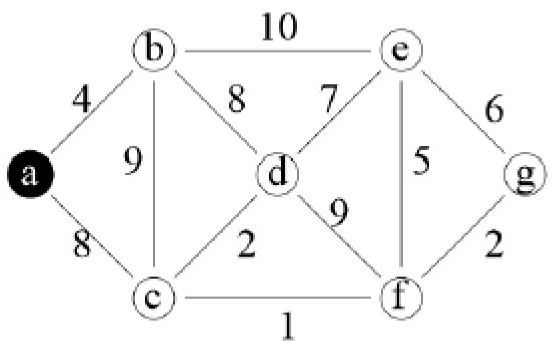



```
void prim(G,wt,root)
{
   for every u in V { dist[u] = ∞;  inmst[u] = FALSE;}
   dist[root] = 0; pred[root] = NULL;
   PQ = makePQ(V); /* all vertices in PQ */
   while(!empty PQ){
     u = deletemin(PQ);
     for every (v adjacent to u){
            if ((inmst[v]==FALSE)&& (w[u][v]< dist[v])){
               dist[v] = w[u][v]; /* update distance */
               decreasewt(PQ,v,dist[v]);/* update PQ dist*/
               pred[v]=u; /* update path information */
            }
     }
     inmst[u] = TRUE;
   }
 } /* at end: MST = {{v,pred[v]}: v in V - {root}} */
```

Prim's example



Connected graph



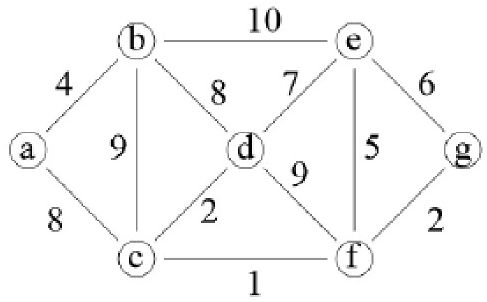
Step 0

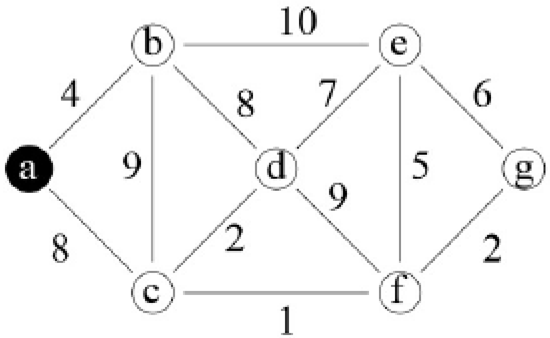
S={a}

V \ S = {b,c,d,e,f,g}

lightest edge = {a,b}

Example from Mordechai Golin, Hong Kong University of Science and Technology
http://www.cse.ust.hk/faculty/golin/COMP271Sp03/Notes/MyL10.pdf





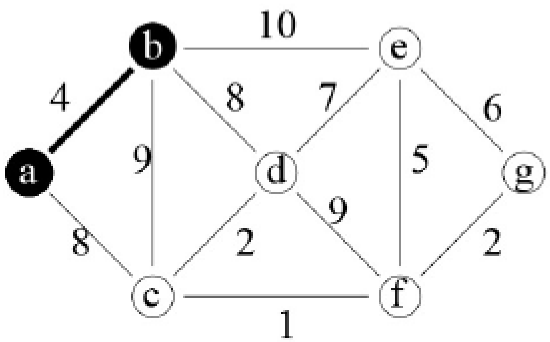
Step 1.1 before

S={a}

V \ S = {b,c,d,e,f,g}

A={}

lightest edge = {a,b}



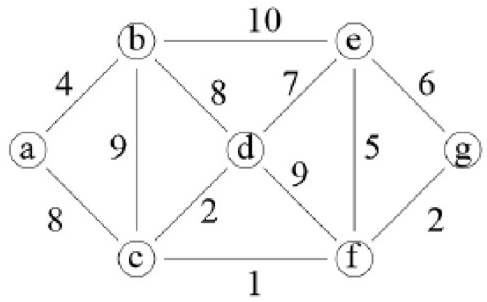
Step 1.1 after

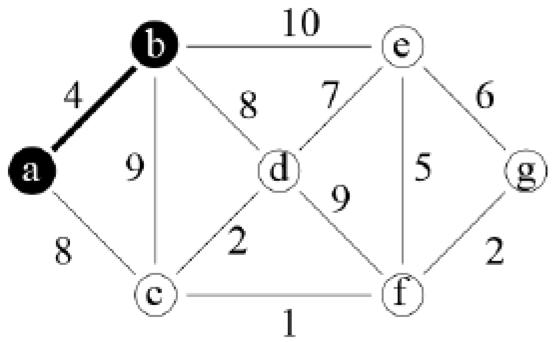
S={a,b}

V \ S = {c,d,e,f,g}

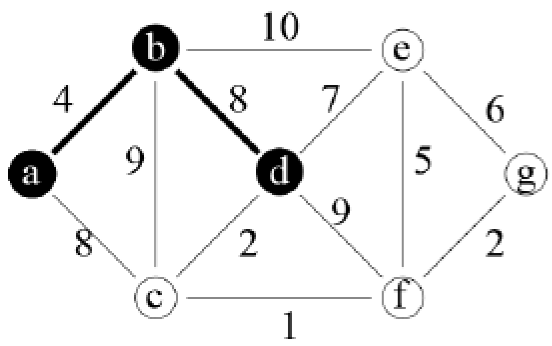
A={{a,b}}

lightest edge = {b,d}, {a,c}

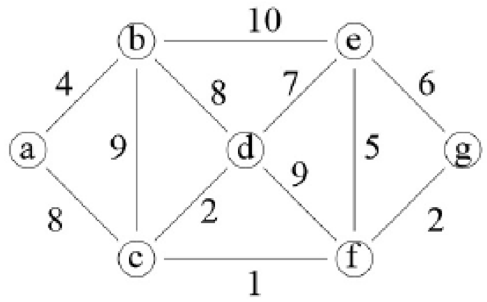


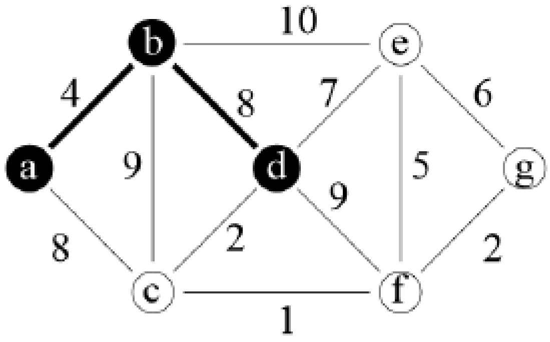


Step 1.2 before
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}



Step 1.2 after
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}





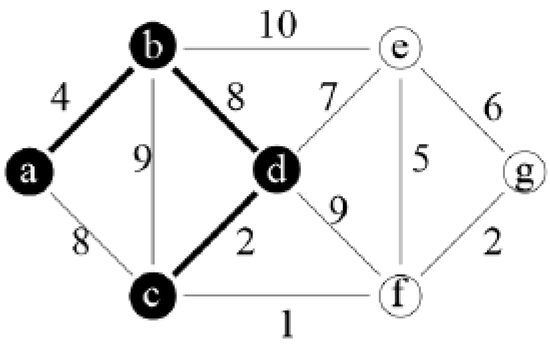
Step 1.3 before

S={a,b,d}

V \ S = {c,e,f,g}

A={{a,b},{b,d}}

lightest edge = {d,c}



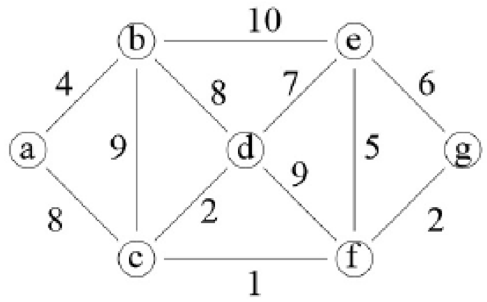
Step 1.3 after

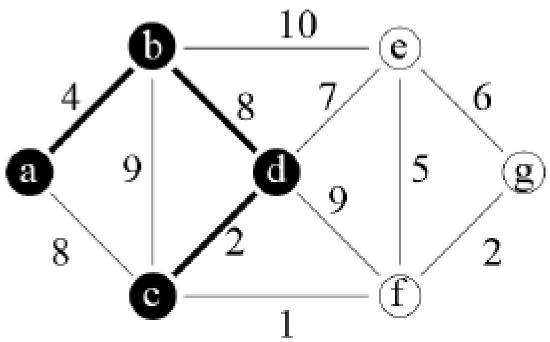
S={a,b,c,d}

V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}





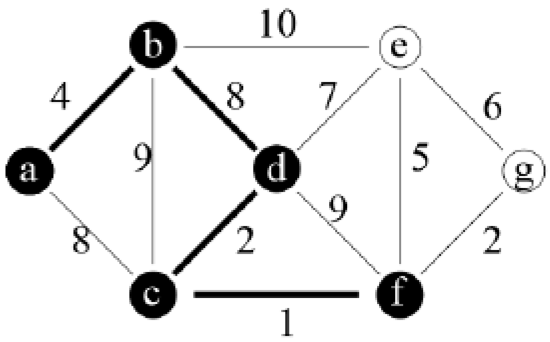
Step 1.4 before

S={a,b,c,d}

V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}



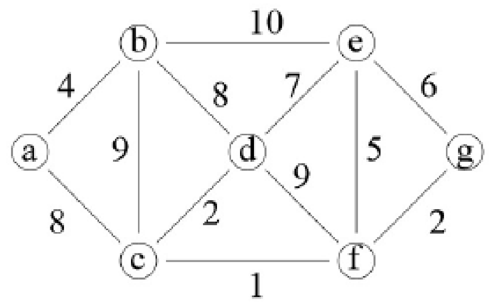
Step 1.4 after

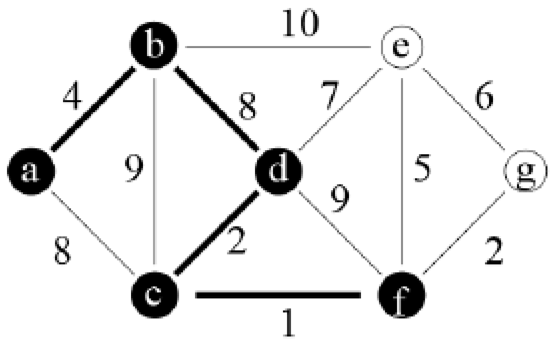
S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}





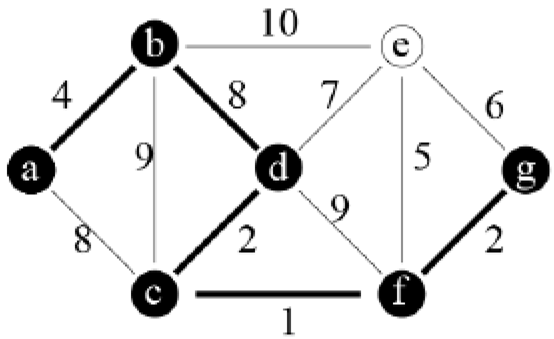
Step 1.5 before

S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}



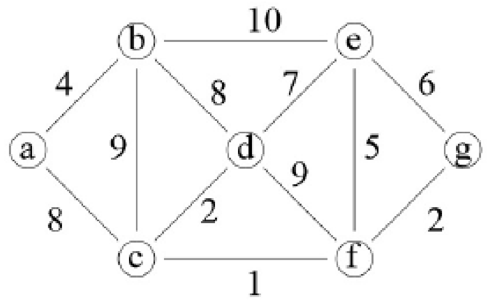
Step 1.5 after

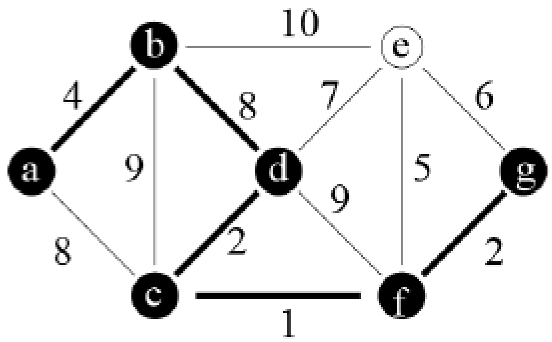
S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}





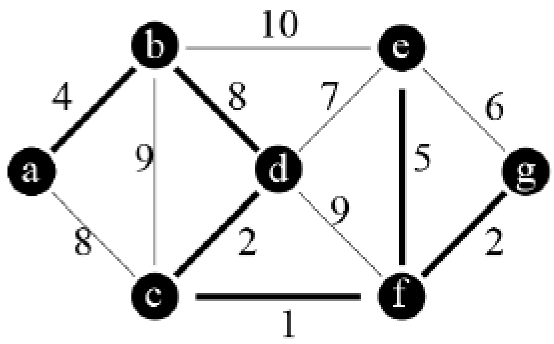
Step 1.6 before

S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}



Step 1.6 after

S={a,b,c,d,e,f,g}

V \ S = {}

A={{a,b},{b,d},{c,d},{c,f},
{f,g},{f,e}}

MST completed

Prim's: Analysis

Initialize arrays pred[] dist[]: O(v)

Make PQ of vertices: if heap O(v)

Loop while PQ not empty V*

Deletemin (if heap) O(log v)

Update adjacent weights,

adjust wt in PQ **O(degree of u * log v)**

= O(V* (log V + deg(u) * logV))

= O(V log V + V deg(u) * log V)

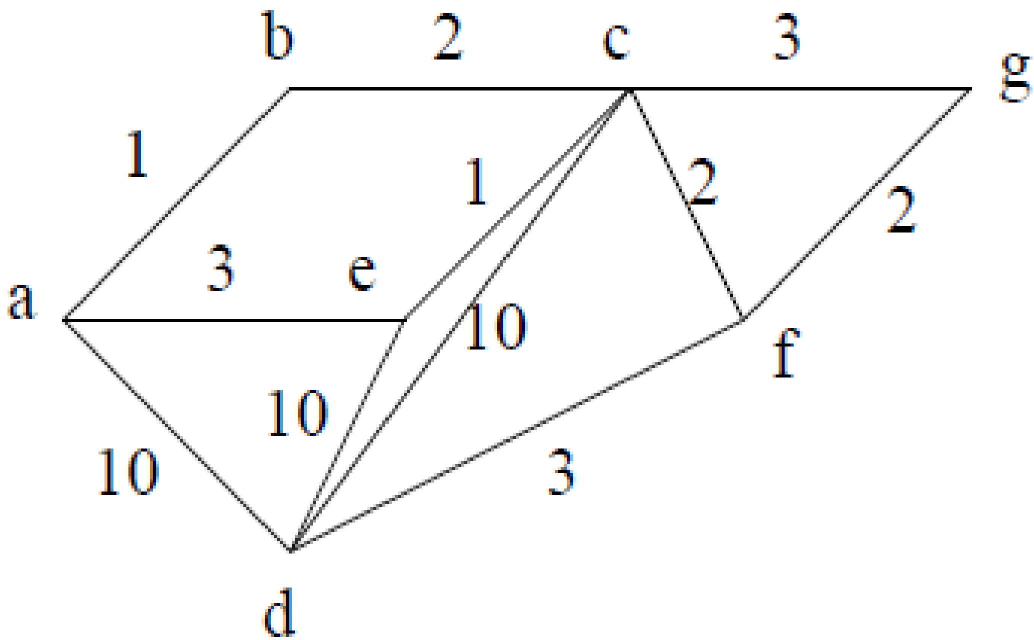
Noting that V deg(u) = E,

= O(V log V + E log V)

= O((V+E)logV)

= O(E log V) for dense graphs with heap PQ

Prim's *vs.* Dijkstra's



Quizz! Which ones are correct?

a) Shortest paths From (a) == MST

b) Shortest paths From (f) == MST

c) Is there more than 1 MST solution for this graph

Kruskal's MST algorithm

Prim's algorithm adds the next closest vertex.

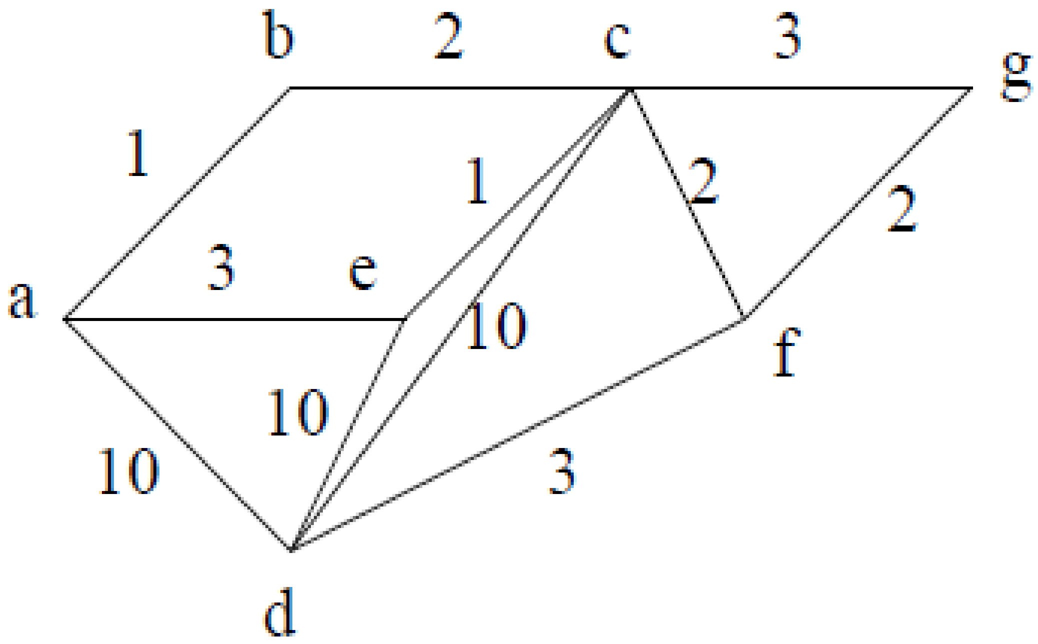
Kruskal's algorithm adds the next lowest weight edge *that **doesn't form a cycle***.

Kruskal's Algorithm for MST

```
E1: edges in MST so far
E2: remaining edges

E1=EMPTYSET,E2=E
Sort edges in E2 by weight
while |E1| < |V|-1 edges and E2 not EMPTYSET
    Pick min cost edge e(i,j) from E2
    E2 = E2 \ e(i,j)
    if V(i),V(j) are not in same MST-so far, then
        E1 = E1 Union e(I,j)
        unite MSTs with V(i) and V(j)
```

Kruskal's



```
if V(i),V(j) are not in same MST-so far, then
         unite MSTs with V(i) and V(j)
```

Prevents cycles (not in same MST-so far)

Unites MSTs (new edge in MST-so far)

Kruskal's algorithm

Sort edges:

- E log E

E* get next edge and check for cycle

E * merge subsets

Kruskal's Algorithm for MST

E1: edges in MST so far
E2: remaining edges

```
E1=EMPTYSET,E2=E
Sort edges in E2 by weight
while |E1| < |V|-1 edges and E2 not EMPTYSET
     Pick min cost edge e(i,j) from E2
     E2 = E2 \ e(I,j)
     if V(i),V(j) are not in same MST-so far, then
          E1 = E1 Union e(I,j)
          unite MSTs with V(i) and V(j)
```

??

```
if V(i),V(j) are not in same MST-so far, then
         unite MSTs with V(i) and V(j)
```

Prevents cycles (not in same MST-so far)

Unites MSTs (new edge in MST-so far)

Sounds easy, but…

…. **requires new data structure and algorithm**

- Disjoint-set data structure
- Union-find algorithm

Union-find

Have disjoint (non-overlapping) subsets

- Find: Which subset is an element in?
- Union: Join two subsets into a single subset

For Kruskal's algorithm:

- **Find**: Are the two vertexes of the new edge in the same subset?
  - If yes, this is a cycle! – don't use!
- **Union**: join two subsets, with the new edge, into a single subset

Union-find

Have disjoint (non-overlapping) subsets

- Find: Which subset is an element in?
- Union: Join two subsets into a single subset

Union-find

Have disjoint (non-overlapping) subsets

- Find: Which subset is an element in?
- Union: Join two subsets into a single subset

Naïve union-find: array

Union-find: array

Have disjoint (non-overlapping) subsets

- Find: Which subset is an element in?
- Union: Join two subsets into a single subset

Naïve union-find: array

	0	1	2	3	4	5	6	7
id[]								

Union-find: array

Start: Singleton Sets

	0	1	2	3	4	5	6	7
id[]	0	1	2	3	4	5	6	7

Example: Union (2, 3) in same set, and (4,6):

–change entry in id[]: choose representatives

0	1	2	3	4	5	6	7
0	1	2	2	4	5	4	7

Union-find: array

0	1	2	3	4	5	6	7
0	1	2	2	4	5	4	7
4	2	2	2	4	4	4	2
2	2	2	2	2	2	2	2

Union-find: array

Naïve algorithm, using array:

Find:

- id[p] == id[q]
- O(?)

Union:

- id[p and all in same subset] = id[q]
- O(?)

Speeding up the Union in Union-Find

Speed up union: tree-based approach

- id[] is a parent array
- **Root** is the **representative of the subset**

To union two subsets – make the root of one the parent of the root of the other

- O(?)

0	1	2	3	4	5	6	7

Find in Tree-based Union-Find

Find:

- Traverse back through parent array to root
- Nodes are in the same subset if they have the same root
- O(?)

  Time for trace depends on depth of tree

0	1	2	3	4	5	6	7

Improvements in Union-Find

Find:

- **Time** for trace depends on **depth of tree**
- Weighted: merge smaller tree into larger
  - **keeps tree broader**
- Path compression

Analysis: E union-finds on V vertices

- Naïve: O(EV)
- Weighted or path compress: O(V + E log V)
- Weighted AND path compress: O(E+V) α(V)

  $\approx$ O(E+V)

Union-Find Analysis

Analysis: E union-finds on V vertices

- Naïve: O(EV)
  - Array: O(1) find; O(n) union
  - Tree: O(1) union; O(n) find
- Weighted OR path compress: O(V + E log V)
- Weighted AND path compression:
  - $O(E*\alpha(E,V) + V)$
  - $\alpha(n)$: inverse Ackermann function, small constant
  - $\approx O(E+V)$

Kruskal’s: Analysis with best union-find

Sort edges:

- E log E

E finds and E unions:

- E+V

$O(E \log E + E + V) = O(E \log E)$

Time is dominated by sorting the edges!

Kruskal’s: Analysis with best union-find

Time is dominated by sorting the edges!

Any ideas for what we might do?

Improvement to Kruskals: Partial sort

When sorting dominates performance, partial sorting can help…

… only need the smallest V-1 edges

e.g. quicksort-like partition, but

- Works if graph is connected
- Doesn't work if longest edge needs to be in MST
  - *e.g.* tight clusters connected by one or more long edges

	Prim	Kruskal
General	(E+V) log V	E log E
Dense Graph	E log V	E log E
V<<E, Prim's is faster		
Sparse Graph	V log V	V log V
Kruskal's is faster because of the data structures		

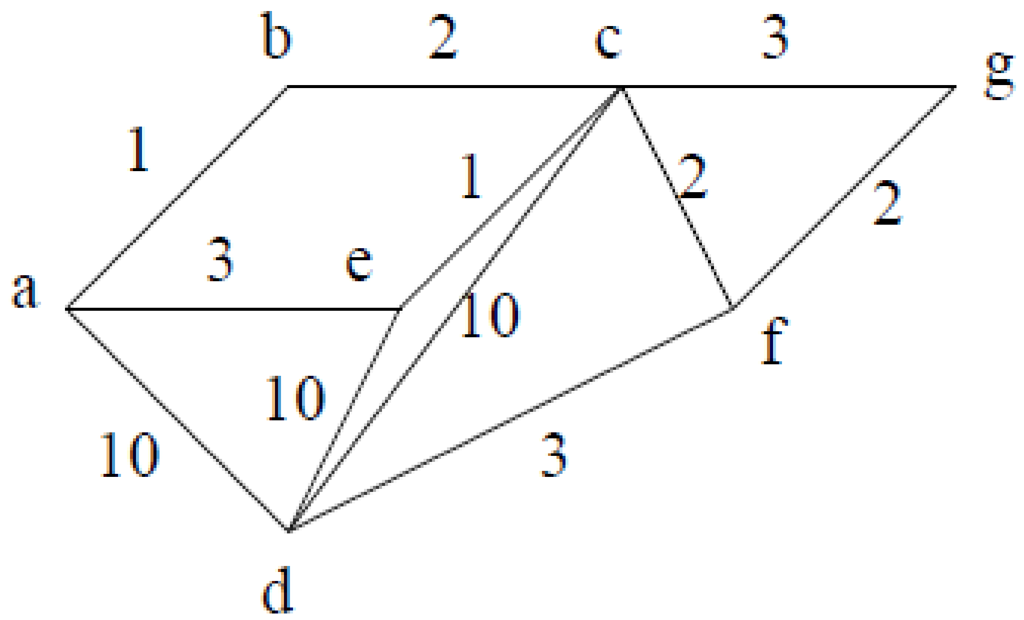
Kruskal's algorithm: an overview (Skiena)

A large-scale view of Kruskal's algorithm:

https://meyavuz.wordpress.com/2017/03/10/prims-algorithm-animation-for-randomly-distributed-points/

https://bost.ocks.org/mike/algorithms/

Kruskal's *vs.* Prim's



More advanced MSTs

Euclidean MSTs:

- Given points on a plane, build MST
- Could construct complete graph, then use Prim's. – Slow!

Other more clever algorithms exist

More advanced MSTs

Randomized MST algorithm

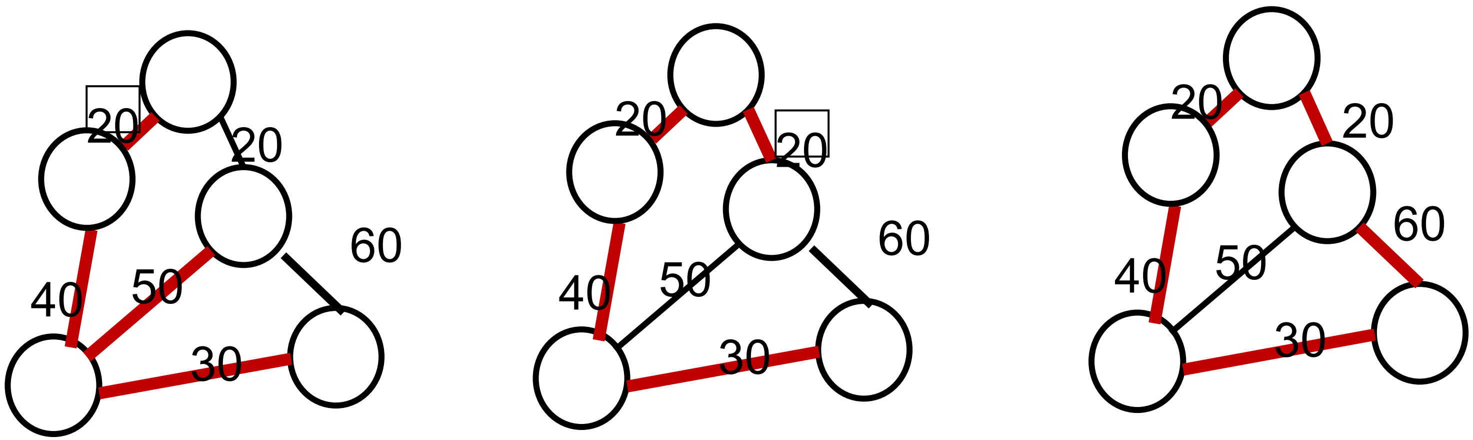
- Random partition of the graph
- Expected time linear, but bad worst case
- Karger, David R.; Klein, Philip N.; Tarjan, Robert E. (1995). "A randomized linear-time algorithm to find minimum spanning trees". *JACM* **42** (2): 321–328.
- Linear MST algorithms exist for restricted types of graphs

The general solution for linear time MST creation is an open research problem

MST and the Travelling Salesperson Problem

Travelling salesperson problem (TSP):

- Given a list of cities and the distances between each pair of cities, find:
  - shortest possible route that
  - visits each city exactly once
  - and returns to the origin city



MST and the Travelling Salesperson Problem

Travelling salesperson problem (TSP):

- Given a list of cities and the distances between each pair of cities, find:
  - shortest possible route that
  - visits each city exactly once
  - and returns to the origin city

Much harder than MST!

Greedy (nearest neighbor) doesn't work!

Graph algorithms

Graph search

Algorithms on undirected graphs

Algorithms on directed graphs

Graph algorithms

Graph search

- Depth-first search
- Breadth-first search
- Priority-first search
- (Connected components)

Algorithms on undirected graphs

Algorithms on directed graphs

Graph algorithms

Graph search

Algorithms on undirected graphs

Algorithms on directed graphs

- Single source shortest path (Dijkstra's)
- Transitive closure (Warshall)
- All pairs shortest path (Floyd-Warshall)

Graph algorithms

Graph search

Algorithms on undirected graphs

- Minimum spanning tree
  - –Prim's
  - –Kruskal's
- Travelling salesperson

Algorithms on directed graphs

Graphs in the real world

Many real-world problems can be modelled as graphs

Many specialized types of graphs allow modelling of complex problems

People have been working on graph algorithms for a long time, so

Huge library of algorithms available

Take away lesson

If you can model a problem as a graph, there is a very good chance that there is already an algorithm to solve the problem…

… or evidence that the problem is intractable