Minimum Spanning Trees; Shortest Paths

Note Title 3/22/2012

Graphs and Digraphs

Definitions and Representations

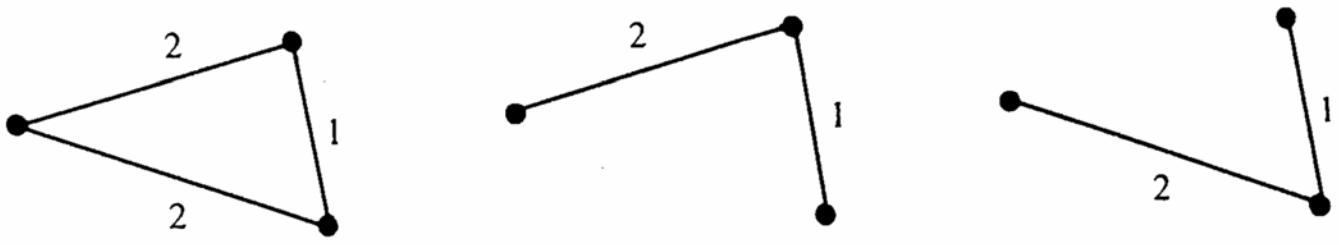
A Minimum Spanning Tree Algorithm

A Shortest-Path Algorithm

Traversing Graphs and Digraphs

Spanning trees

A Minimum Spanning Tree Algorithm



(a) A graph.

(b) Two minimum spanning trees.

Figure 4.12 Minimum spanning trees.

Kruskal Algorithm

Given a connected weighted graph $G = (V, E, W)$ construct a weighted graph $T = (V', E', W')$ that satisfies the following properties:

(i) T is a connected & acyclic graph (a tree)

(ii) T is a subgraph of G (so: $V' \subseteq V$, $E' \subseteq E$, $W' = W \restriction E'$)

that covers vertices of G (so: $V' = V$)

(a spanning tree of G)

(iii) T has the minimum weight out of all graphs that satisfy conditions (i) and (ii)

The algorithm.

1. Order increasingly all edges E of G with respect to their weights:

$L = (e_1, e_2, \dots, e_m)$, where $m = |E|$ and

$$W(e_i) \leq W(e_{i+1}) \text{ for } i = 1, \dots m-1$$

2. Begin with a weighted graph $T_0 = (V, E_0, W|E_0)$,

putting $E_0 = \emptyset$ (the empty set of edges).
The algorithm will construct a sequence of graphs $T_k = (V, E_k, w|E_k)$ such that the last (n-th) graph T_{n-1} in that sequence is the required minimum spanning tree of G

At this point, T_0 is but a bunch of n 1-element connected subgraphs of G and <u>not</u> a connected

graph. Obviously, it is acyclic and minimal.

2. Select the next edge e_i from the list L (if this is the first-time of execution of this step, it will be the first edge e_1)
3. Check if e_i has ends in the same connected component of $T_k = (V, E_k, W|E_k)$.
4. If "yes" then go back to Step 2.

(We already know that adding e_i to E_k would produce a cycle, an then bye-bye spanning tree).

5. If "no" then make $T_{k+1} = (V, E_{k+1}, \cup \Gamma E_{k+1})$ where

$$E_{k+1} = E_k \cup \{e_i\}$$

Now, T_{k+1} is still acyclic but has one less connected component. So the number of connected components of T_{k+1} is $(n-k)-1$.

6. If T_{k+1} is still not connected (has more than 1 connected component, that is, $n - k - 1 > 1$, or $k + 1 < n - 1$ then go back to step 2.

7. If T_{k+1} is connected then it is the required spanning tree of G:

- its connected (has only 1 connected component)
- its acyclic (we constructed it that way)

A note on implementation

The connected components constructed along by Kruskal define an equivalence relation $\equiv$ by:

$v \equiv w$ iff v and w belong to the same connected component
iff there is a path from v to w in F (Kruskal forest constructed so far

Find-Union algorithm

Find(1) = 1 Find(3) = 3 {1} {2} {3} {4}
Union(1,3) {1,3} {2} {4}
Union(2,4) {1,3} {2,4}
$1 \equiv 3$? yes $1 \equiv 2$ no

1) connected components constructed by Kruskal are pairwise disjoint

2) Their union covers the entire graph

3) They are all non-empty

So, at any point, these connected components constitute a partition of the set of vertices of entire graph G

This is why we could use Union-Find operations.

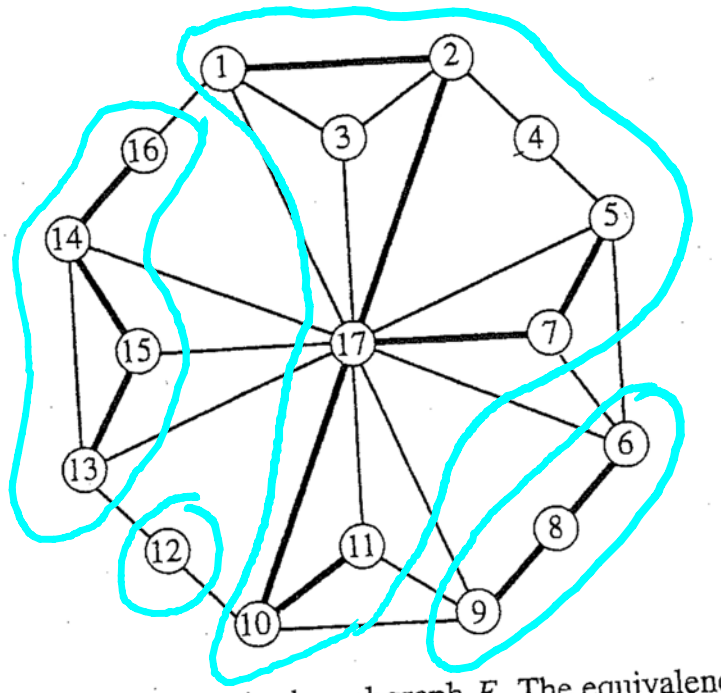
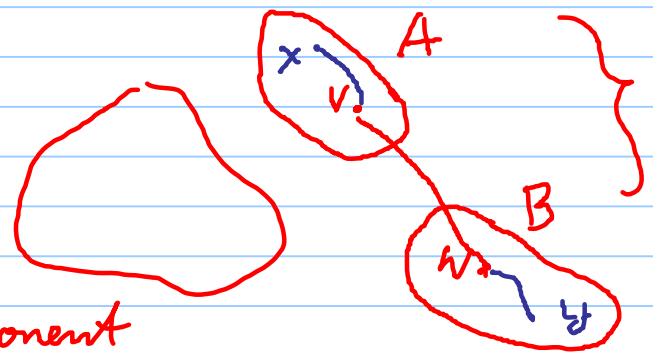


Figure 8.11 The darkened edges are in the subgraph F. The equivalence classes are $\{1, 2, 5, 7, 10, 11, 17\}$, $\{6, 8, 9\}$, $\{13, 14, 15, 16\}$, $\{3\}$, $\{4\}$, and $\{12\}$.

Lemma 8.8 Let F be a forest; that is, any ~~undirected~~ acyclic graph. Let $e = vw$ be an edge that is not in F. There is a cycle consisting of e and edges in F if and only if v and w are in the same connected component of F. □

Case 1.
v and w are not in the same connected component



C is a connected graph because there is a path between x and y for any $x \in A$, $y \in B$.

A has n_A vertices and $n_A - 1$ edges

B has n_B vertices and $n_B - 1$ edges

So C has $n_A + n_B$ vertices and

$$(n_A - 1) + (n_B - 1) + 1 = (n_A + n_B) - 1 \text{ edges}$$

We showed already that C is connected.

So C is a tree. Therefore, C is acyclic

Now, suppose v and w are in the same tree D of F.

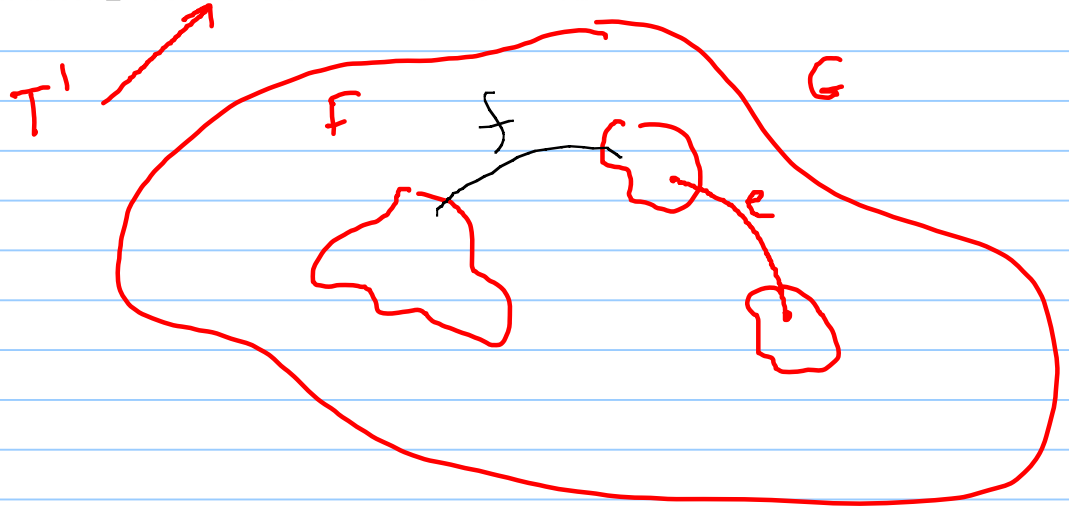
Case 2. v and w are in the same connected component

D had n_D vertices and $n_D - 1$ edges.
So, D is a tree.
After adding edge vw, it has
n_d vertices and n_d edges. So, $D \cup \{vw\}$
has a cycle. (Recall, that a tree is a graph
that becomes cyclic if an edge is added to it.)

The following theorem establishes a loop invariant for the Kruskal algorithm:

If F is a Kruskal forest built so far with the property $F \subseteq T$ for some minimum spanning tree T of G, then $F \cup \{e\} \subseteq T'$ for some minimum spanning tree T' of G, where e is the edge added by the next step of Kruskal algorithm.

Theorem 8.9 Let $G = (V, E, W)$ be a weighted connected graph. Let $F \subseteq E$. If F is contained in a minimum spanning tree for G and if e is an edge of minimum weight in $E - F$ such that $F \cup \{e\}$ has no cycles, then $F \cup \{e\}$ is contained in a minimum spanning tree for G. □



$F \subseteq T'$

$F' = F \cup \{e\} \subseteq T'$

$e \notin F$

(*) Suppose to the contrary, that $F \subseteq T$ but $F \cup \{e\} \not\subseteq T'$ for any minimum spanning tree T' of G.

That means that $e \notin T'$ for any minimum spanning tree T' of G with $F \subseteq T'$.

In particular, $e \notin T$.

Since T is a tree (a maximum acyclic graph on n vertices), $T \cup \{e\}$ has a cycle and e is a part of that cycle.

Let C be the set of edges that belong to that cycle. We have:

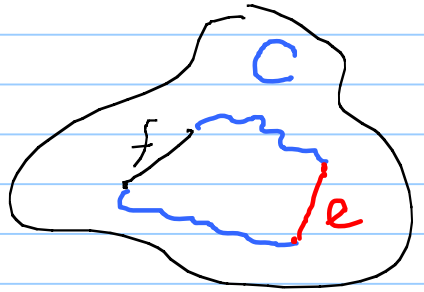
$C \subseteq T \cup \{e\}$ and $C \nsubseteq F \cup \{e\}$ (since $F \cup \{e\}$ is acyclic). Therefore, $C \setminus (F \cup \{e\}) \neq \emptyset$.

Let $f \in C \setminus (F \cup \{e\})$. In particular, $f \in C \setminus \{e\}$. (simply because $\{e\} \subseteq F \cup \{e\}$).

Because $C \subseteq T \cup \{e\}$ then $C \setminus \{e\} \subseteq T$ (exercise: prove it!), so $f \in T$. Since $f \notin F$, $w(f) \geq w(e)$ because e has the minimum

weight of all edges $\notin F$.

Let $T'' = (T \cup \{e\}) \setminus \{f\}$.



We have:

$F \cup \{e\} \subseteq T''$

because $F \subseteq T$.

$T \cup \{e\}$ is connected. (since T is), and so is $(T \cup \{e\}) \setminus \{f\} = T''$. T'' and T have the same number of vertices (say, n) and the

same number of edges $(n-1)$, so T'' is a tree. T'' is a spanning tree of G. By our assumption (*), T'' is not a minimum spanning tree.

$w(T'') = w(T) + w(e) - w(f) \leq$

$\leq w(T)$, since $w(e) \leq w(f)$ and

$w(e) - w(f) \leq 0$.

So, $w(T'') \leq w(T)$.

This contradicts a consequence of our assumption that T'' is not a minimum spanning tree.

Contradiction. □

Performance of Kruskal algorithm

Use min-heap to quickly find next minimum-weight edge, and Find-Union to merge components.

1. To create a heap H (Make Heap):

   $\Theta(m)$ time.

2. To execute H.Remove() up to m times

   $\Theta(m \lg m)$ time in the worst case.

   Since $n - 1 \le m < n^2$, $\lg(n-1) \le \lg m < 2 \lg n$,

   so $\Theta(\lg m) = \Theta(\lg n)$.

Therefore the time needed to perform this step is

$$\Theta(m \lg n)$$

If only first $O(n)$ lightest edges are used, this time reduces to

$$O(n \lg n).$$

3. Initialization of Find-Union (creation of n singleton sets):

$\Theta(n)$ time

4. Performing $n-1$ unions (one for each of $n-1$ edges added to the Kruskal forest F)

$\Theta(n)$ time

(each union takes $O(1)$ time).

5. Performing up to m Finds

$O(m \lg^* n)$ time

(each find takes amortized $\Theta(\lg^* n)$ time, where $\lg^* n$ is the inverse to $2^{2^{\cdot^{\cdot^{2}}}}$).

If only first $O(n)$ lightest edges are used, this step takes only

$O(n \lg^* n)$ time.

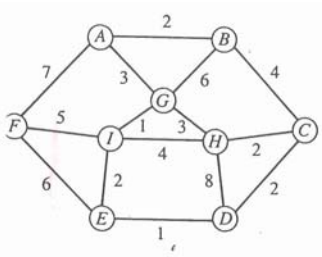
Total: $\Theta(m + m \lg n + n + n + n \lg^* n) =$

$= \Theta(m \lg n)$.

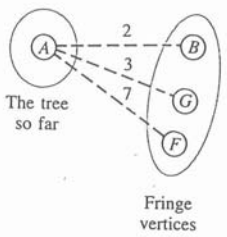
If only first $O(n)$ lightest edges are used, this time reduces to $\Theta(m + n \lg n + n + n + n \lg^* n) =$

$= \Theta(m + n \lg n)$.

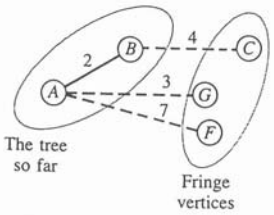
Dijkstra – Prim Algorithm



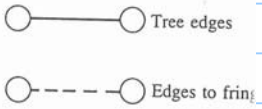
(a) A weighted graph.



(b) The tree and fringe after the starting vertex is selected.



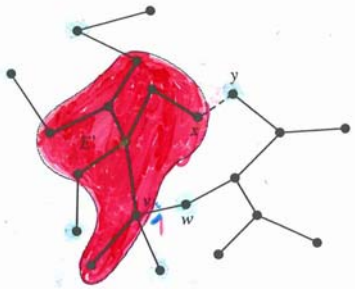
(c) After selecting an edge and vertex. *BG* is not shown because *AG* is a better choice (has lower weight) to reach *G*.



The Dijkstra/Prim algorithm e - # of edges
n - # of vertices

Tree vertices: in the tree constructed so far.
Fringe vertices: not in the tree, but adjacent to some vertex in the tree.

Select an arbitrary vertex to start the tree;
while there are fringe vertices **do**
    select an edge of minimum weight between a tree vertex and a fringe vertex;
    add the selected edge and the fringe vertex to the tree
end



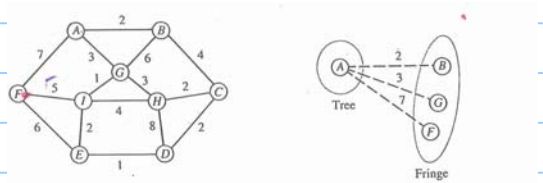
Details of proof in file
Minimum spanning tree (Prim).pdf

Algorithm 4.1 Minimum spanning tree (Dijkstra/Prim)

Input: $G = (V, E, W)$, a weighted graph.

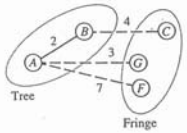
Output: The edges in a minimum spanning tree.

```
1.  { Initialization }
    Let x be an arbitrary vertex.
    V_T := {x}; E_T := ∅;
    stuck := false;
2.  { Main loop; x has just been brought into the tree.
    Update fringe and candidates.  Then add one vertex and edge. }
    while V_T ≠ V and not stuck do
3.  { Replace some candidate edges. }
        for each fringe vertex y adjacent to x do
            if W(xy) < W(the candidate edge e incident with y) then
                xy replaces e as the candidate edge for y;
            end { if }
        end { for };
4.  { Find new fringe vertices and candidate edges. }
        for each unseen y adjacent to x do
            y is now a fringe vertex;
            xy is now a candidate
        end { for };
5.  { Ready to choose next edge. }
        if there are no candidates then stuck := true {no spanning tree};
        else
6.  { Choose next edge. }
            Find a candidate edge, e, with minimum weight;
            x := the fringe vertex incident with e;
            Add x and e to the tree;
            { x and e are no longer fringe and candidate. }
        end { if }
    end { while }
```

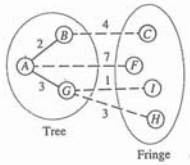


(a) A weighted graph.

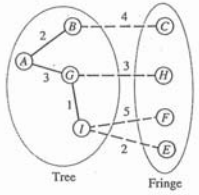
(b) After selection of the starting vertex.



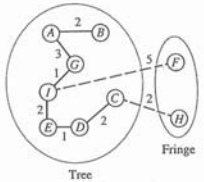
(c) *BG* was considered but did not replace *AG* as a candidate.



(d) After *AG* is selected and fringe and candidates updated.

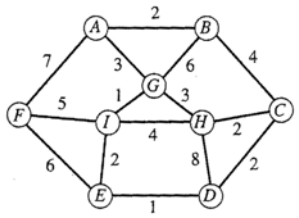


(e) *IF* has replaced *AF* as a candidate.

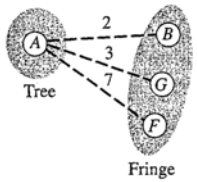


(f) After several more passes. The two candidate edges will be put in the tree.

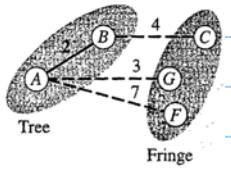
Figure 4.15 An example for the minimum spanning tree algorithm.



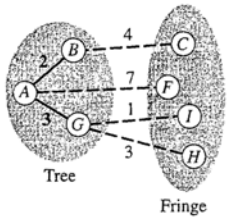
(a) A weighted graph



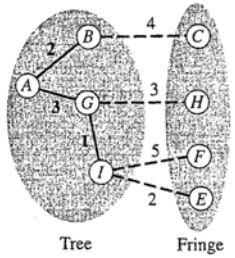
(b) After selection of the starting vertex



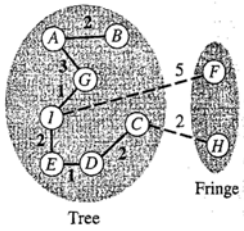
(c) BG was considered but did not replace AG as a candidate.



(d) After AG is selected and fringe and candidates are updated



(e) IF has replaced AF as a candidate.



(f) After several more passes: The two candidate edges will be put in the tree.

Figure 8.4 An example for Prim's minimum spanning tree algorithm.

Exercise 8.9, where we discover that the worst case is worse than $\Theta(n^2)$. Can we do any better?

	fringeLink	*fringeWgt*	*parent*	*status*	*adjacencyList*
A				*intree*	→
B	nil	2	*A*	*intree*	→
C	*F*	4	*B*	*fringe*	→
D				*unseen*	→
E				*unseen*	→
F	nil	7	*A*	*fringe*	→
G	*F*	3	*A*	*intree*	→
H	*C*	3	*G*	*fringe*	→
I	*H*	1	*G*	*fringe*	→

Shaded entries in *fringeLink* are no longer in use. *fringeList* = *I*.

(Adjacency lists not shown. Nodes are assumed to be in alphabetical order within each list.)

Skip

```
primMST(G, n)  // OUTLINE
    Initialize the priority queue pq as empty.
    Select an arbitrary vertex s to start the tree;
    Set its candidate edge to (−1, s, 0) and call insert(pq, s, 0).
    While pq is not empty:
        v = getMin(pq); deleteMin(pq);
        Add the candidate edge of v to the tree.
        updateFringe(pq, G, v);

updateFringe(pq, G, v)  // OUTLINE
    For all vertices w adjacent to v, letting newWgt = W(v, w):
        If w is unseen:
            Set its candidate edge to (v, w, newWgt).
            insert(pq, w, newWgt);
        Else if newWgt < fringeWgt of w:
            Revise its candidate edge to (v, w, newWgt).
            decreaseKey(pq, w, newWgt);
```

Analysis

Let $n = |V|$ and $m = |E|$.

$\Theta(m + n^2) = \Theta(n^2)$.

Plain version →

Lower Bound

$\Omega(m)$

With Fibonacci heaps implementation (to maintained partially ordered set of candidate edges), the worst-case running time becomes $\Theta(m + n \lg n)$.

Plain version is:
Same or better than Kruskal if $m \in \Theta(n^2)$
Worse than Kruskal if $m \in \Theta(n)$.
- same as Kruskal if $m \in O(n)$ or if only $O(n)$ lightest edges were used by Kruskal.

A Shortest-Path Algorithm

The weight of a path $v_0, v_1, \ldots, v_k$ in a weighted graph $G = (V, E, W)$ is $\sum_{i=0}^{k-1} W(v_i v_{i+1})$

The *distance* from a vertex x to a vertex y, denoted $d(x,y)$, is the weight of a shortest path from x to y.

Tree vertices: in the tree constructed so far.
Fringe vertices: not in the tree, but adjacent to some vertex in the tree
Unseen vertices: all others.

$dist[y] = d(v,y)$ for y in the tree,
$dist[z] = d(v,y)+W(yz)$ for z on the fringe,

where yz is the candidate edge to z.

Plain version

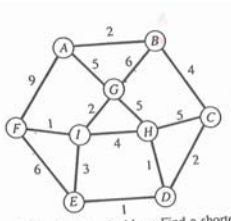
$\Theta(n^2)$

$m \lg n$

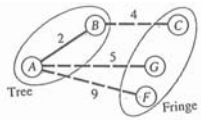
$\Theta(e \log n)$

With Priority Queue (heap)

Dijkstra-Prim can be as fast as
$\Theta(m + n \lg n)$ with Fibonacci heaps
implementation of Priority Queue.

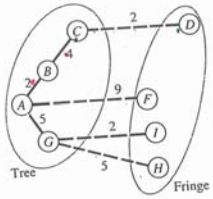


(a) The Graph. Problem: Find a shortest path from A to H.



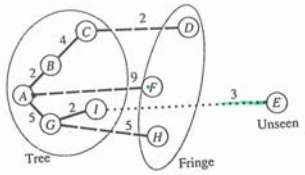
$d(A, B) + W(BC) = 6$
$d(A, A) + W(AG) = 5$
$d(A, A) + W(AF) = 9$
Select AG next.

(b) An intermediate step.

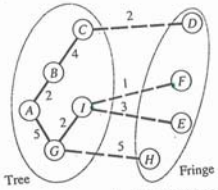


$d(A, C) + W(CD) = 8$
$d(A, A) + W(AF) = 9$
$d(A, G) + W(GI) = 7$
$d(A, G) + W(GH) = 10$
Select GI next.

(c) An intermediate step. (CH was considered but not chosen to replace GH as a candidate.)



GI is selected.



(e) AF was replaced bu IF as a candidate.

Example

Note the difference between this (Dijkstra's SSSP) and Dijkstra-Prim MCST algorithms.

Here, minimization of

distance from the root to vertex x

$d(Ax) + w(xy)$ (SSSP)

weight of a candidate edge xy

rather than just of

$w(xy)$ } weight of a candidate edge xy

takes place.

dijkstraSSSP(G, n) *// OUTLINE*

Initialize all vertices as *unseen*.
Start the tree with the specified source vertex s; reclassify it as *tree*;
define $d(s, s) = 0$.
Reclassify all vertices adjacent to s as *fringe*.
While there are fringe vertices:
    Select an edge between a tree vertex t and a fringe vertex v such that
    $(d(s, t) + W(tv))$ is minimum;
    Reclassify v as *tree*; add edge tv to the tree;
    define $d(s, v) = (d(s, t) + W(tv))$.
    Reclassify all *unseen* vertices adjacent to v as *fringe*.