CSCB63 Winter 2018

Week 5 Lecture 2 - Minimum Cost Spanning Trees

Anna Bretscher and Albert Lai

February 23, 2019

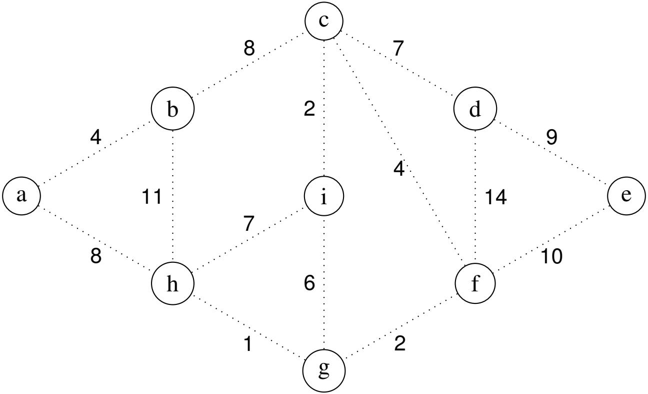
TODAY

Kruskals Algorithm

Prims Algorithm

Dijkstra's Algorithm

Introduction: (Edge-)Weighted Graphs



These are computers and costs of direct connections. What is a cheapest way to network them?

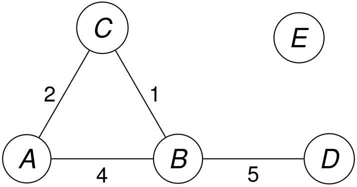
(Edge-)Weighted Graph

- Many useful graphs have *numbers* or *weights* assigned to *edges*.
- Think of each edge e having a *price tag* $w(e)$.
- Usually $w(e) \geq 0$. Some cases have $w(e) < 0$.

A weighted (edge-weighted) graph consists of:

- a set of vertices V
- a set of edges E
- *weights*: a map from edges to numbers $w : E \to \mathbb{R}$
  - *undirected graphs*: $\{u, v\} = \{v, u\}$, *same* weight
  - *directed graphs*: (u, v) and (v, u) may have *different* weights
- **Notation**: $w(u, v)$ or $w(e)$ or $weight(u, v)$ etc.

Storing a Weighted Graph



Adjacency matrix:

	A	B	C	D	E
A	0	4	2	∞	∞
B	4	0	1	5	∞
C	2	1	0	∞	∞
D	∞	5	∞	0	∞
E	∞	∞	∞	∞	0

Adjacency lists:

	adjacency list
A	$(B,4)$, $(C,2)$
B	$(A,4)$, $(C,1)$, $(D,5)$
C	$(A,2)$, $(B,1)$
D	$(B,5)$
E	

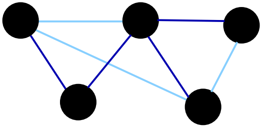
Common Task #1 on Weighted Graphs - Minimum Cost Spanning Trees

Let $G = (V, E)$ be a *connected, undirected* graph with *edge weights* $w(e)$ for each edge $e \in E$.

A *spanning tree* is a tree A such that *every vertex* $v \in V$ is an *endpoint* of at least *one edge* in A.

Q. *Which algorithms have we seen to construct a spanning tree?*

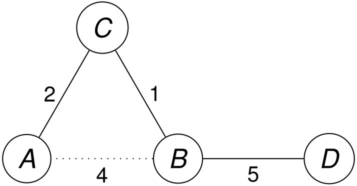
A. DFS, BFS



A *minimum cost spanning tree* (**MST**) is a spanning tree A such that the *sum of the weights is minimum* for all possible spanning trees B.

$$w(A) = \sum_{e \in A} w(e) \leq w(B)$$

Example

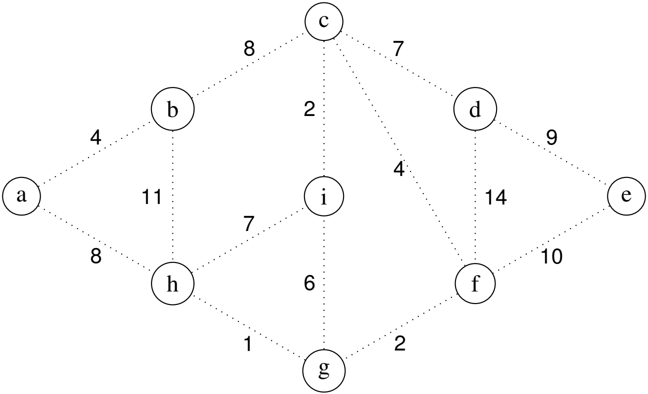


Usually just for *undirected, connected graphs*.

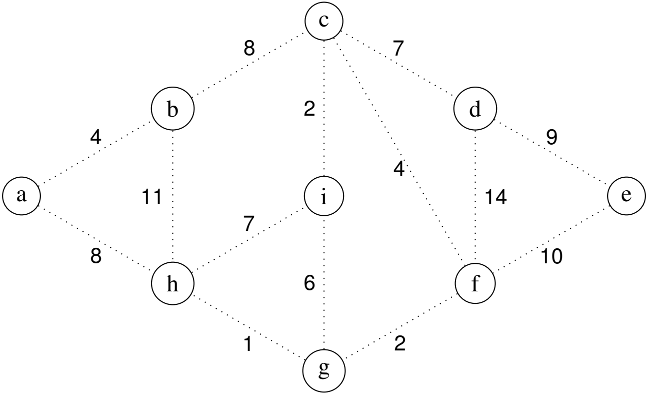
Q. How might we find a *minimum spanning tree*?

A. Let's brain storm - there are several *greedy edge selection techniques* that can work.

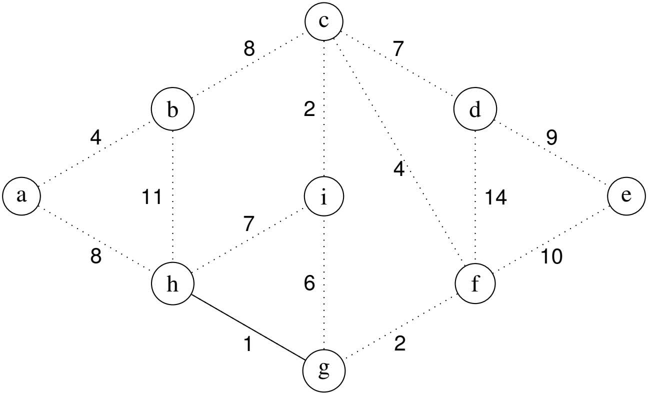
Sample Graph



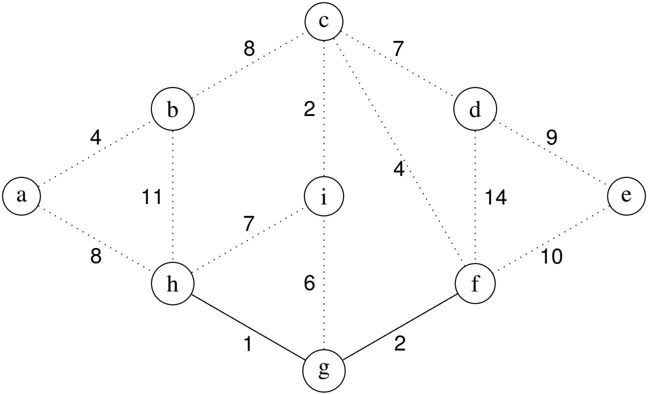
Sample Graph



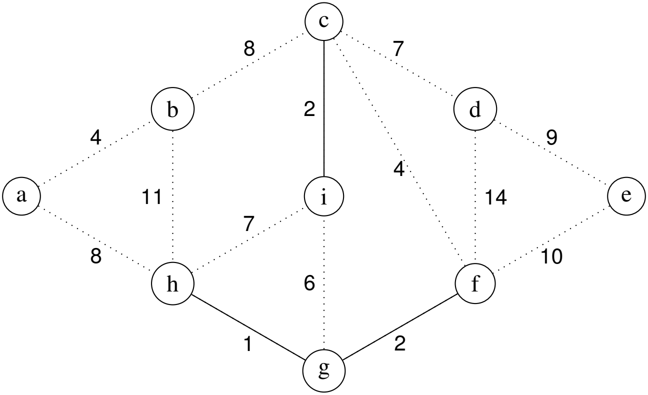
Sample Graph



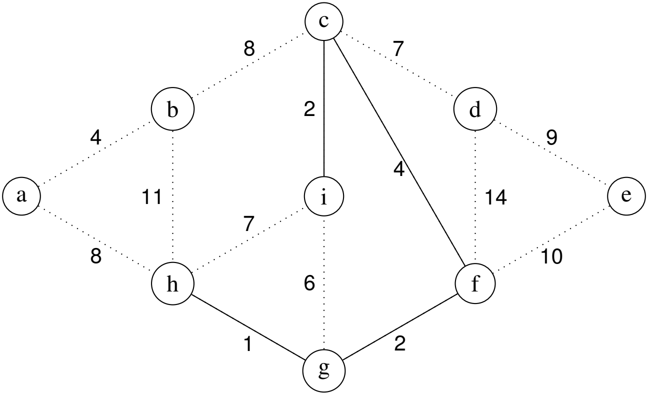
Sample Graph



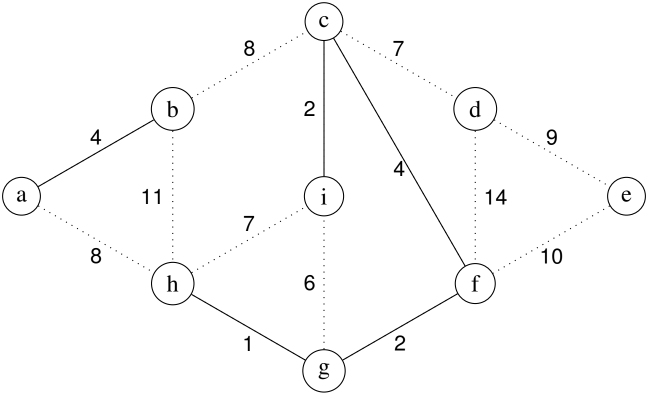
Sample Graph

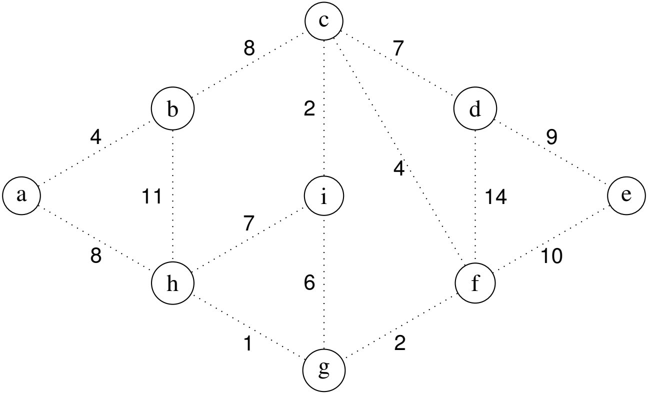


SAMPLE GRAPH

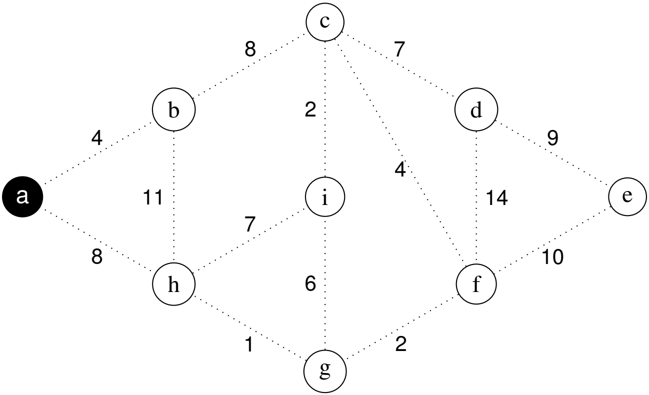


Sample Graph

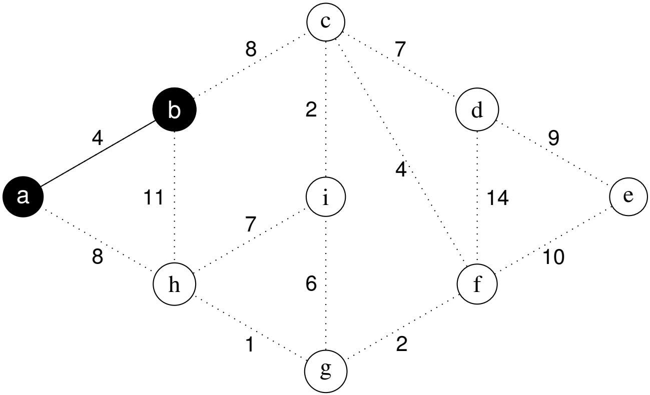




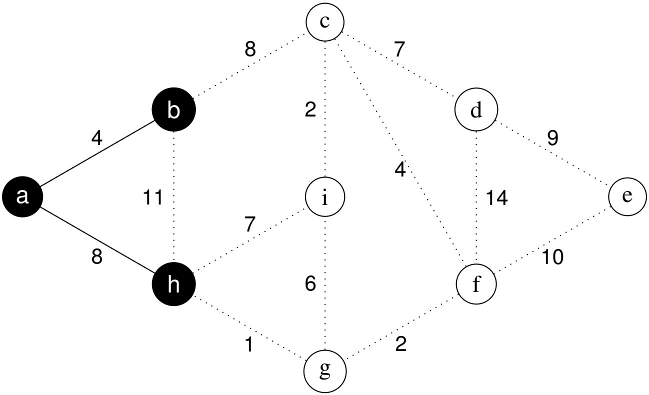
vertex	a	b	c	d	e	f	g	h	i
priority	0	∞	∞	∞	∞	∞	∞	∞	∞
pred									



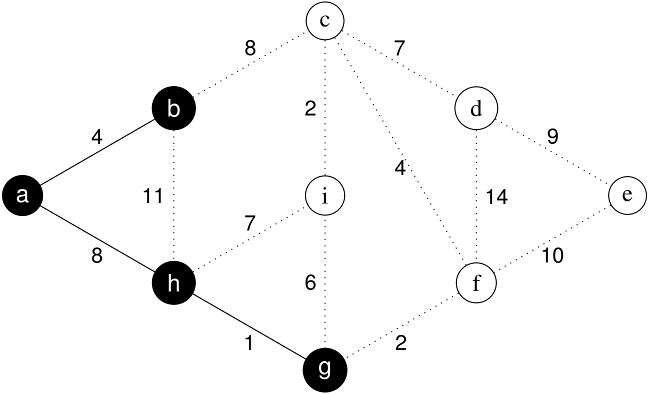
vertex	b	h	c	d	e	f	g	i
priority	4	8	∞	∞	∞	∞	∞	∞
pred	a	a						



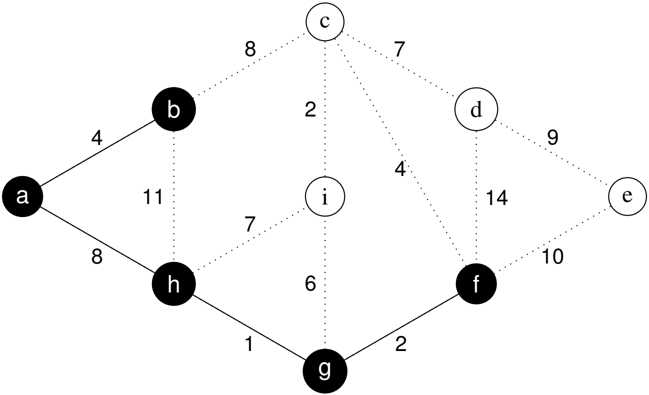
vertex	h	c	d	e	f	g	i
priority	8	8	∞	∞	∞	∞	∞
pred	a	b					



vertex	g	i	c	d	e	f
priority	1	7	8	∞	∞	∞
pred	h	h	b			



vertex	f	i	c	d	e
priority	2	6	8	∞	∞
pred	g	g	b		



vertex	c	i	e	d
priority	4	6	10	14
pred	f	g	f	f

Greedy Algorithms

Kruskal's Pick the *least weight edge* that doesn't *induce a cycle*.

Prim's Start with a *minimum tree* or *set* consisting of a *single vertex*

Add a *least weight edge* that "grows" the *tree* without creating a cycle.

Often think of this as a set of *vertices and edges* in a set S (the tree) and *adding* edge (v, z) to S where $v \in V - S$ and $z \in S$ where $w(v, z)$ is minimum for all such edges.

Q. How can we convince ourselves that our algorithms are correct?

A. We can prove using a *contradiction argument*.

Kruskal's Algorithm: Proof of Correctness

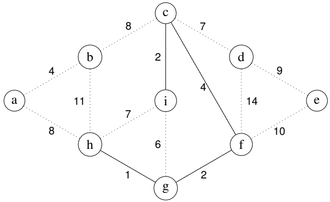
Kruskal's algorithm finds an *MST* by repeatedly adding the *least weight edge* that does *not induce* a *cycle*.

Proof by Contradiction.

- Order edges in *non-decreasing order* of *weight*, i.e. such that $w_1 \leq w_2 \leq \ldots \leq w_n$ where $w(e_i) = w_i$.
- Let K be the spanning tree returned by *Kruskals* algorithm.
- Suppose that O is an *optimal MST*, such that weight of O is less than weight of K. K is not *optimal*.
- Let $e_i = (u, v)$ be the *first* edge in our *ordering* that is not in both K and O.
- Can $e_i \in O$ but $e_i \notin K$?

→ no, because K only *omits* edges if they create a *cycle*.

- Therefore, $e_i \in K$ but $e_i \notin O$.



KRUSKAL'S ALGORITHM: PROOF OF CORRECTNESS

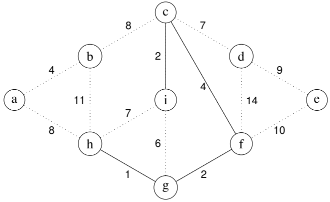
Kruskal's algorithm finds an *MST* by repeatedly adding the *least weight edge* that does *not induce* a *cycle*.

Proof by Contradiction.

- Since O is connected, there must exist a *unique path* p from u to v and an edge e' on p that is not in K.
- Since K did not select e' (but had the option to), $w(e') \geq w_i$.

  Case 1. $w(e') = w_i$. Then we can simply switch e_i and e' and now O has the same weight as before but is more *similar* to K. Repeat the same argument until either **Case 2** or the two trees are the same and K is *optimal*.

  Case 2. $w(e') > w_i$. Now consider a new tree O' constructed by removing e' from O and adding e_i. Now O' has weight less than O contradicting that K and O differ. Therefore K must be *optimal*.



Kruskal's Algorithm

Q. How should we store the *edges* sorted by *non-decreasing weight*?

A. *MIN priority queue*!

Q. How can we *add edges* and make sure that *no cycle* is *induced*.

A. Think of *joining* together *clusters* (subtrees) of *connected vertices*. ← Will learn *efficient* way to do this soon...but one way is to use *linked lists* (not super efficient).

```
Kruskal(E, V)
    S := new container() for chosen edges
    PQ := min priority queue of edges and weights
    for each vertex v:
        v.cluster := {v}
    while not PQ.is_empty():
        {u,v} = PQ.extract_min():
        if u.cluster ≠ v.cluster:
            S.add({u,v})
            union(u.cluster, v.cluster)
    return S
```

Storing Clusters: Easy Way - Linked Lists

Idea.

- each *cluster* is a *linked list*
- *v.cluster* is pointer to *v*'s own *linked list*
- *u.cluster* $\neq$ *v.cluster* is pointer equality, $\Theta(1)$ *time*
- *merging* two clusters is *merging* two linked lists, BUT:

  $\rightarrow$ a lot of *vertices* need their *cluster pointers* updated

Luckily, if you move the *smaller list* to the *larger one*, then:

- whenever *v.cluster* needs *update*, cluster size roughly *doubles*
- If cluster *size doubles*, at most *how many* cluster updates can we do?
- each *v.cluster* is updated at most $\lg n$ times

We will see a *faster* way *later* in this course.

Kruskal's Algorithm Time Complexity

Complexity

- *Building* PQ and *removing* edges: $\Theta(m \lg m)$.
- v.cluster *updates*: $O(\lg n)$ per vertex $\to O(n \lg n)$
- the rest is $\Theta(1)$ per vertex or edge

Total $O(n \lg n + m \lg m)$ time worst case.

Q. What do we know about $\lg m$ and $\lg n$?

A. $\lg m \in O(\lg n)$.

Therefore,

$O((n+m)\lg n)$ time.

Faster if faster cluster implementation.

Prim's Algorithm Again

Prim's algorithm finds an *MST* by something similar to *breadth-first search*, but with a twist:

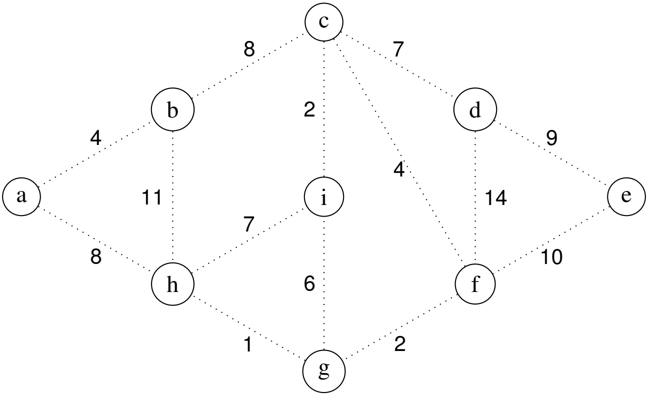
The *queue* is changed to a *min priority queue*.

The algorithm *grows a tree T* one edge at a time.

Priority of vertex v = *smallest edge weight* between v and T so far. (∞ if no such edge.)

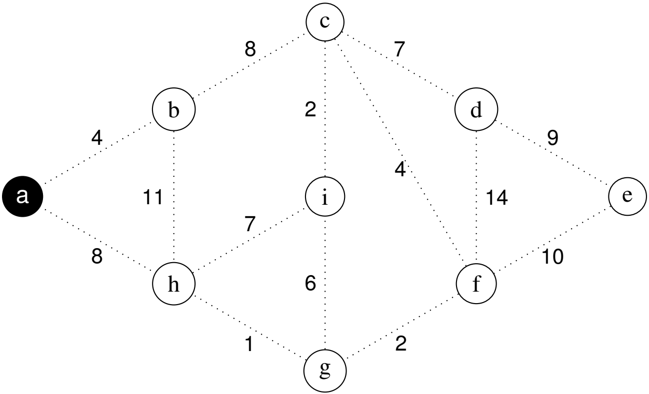
Let's step through the example again...

PRIM'S ALGORITHM: A FEW EXAMPLE STEPS



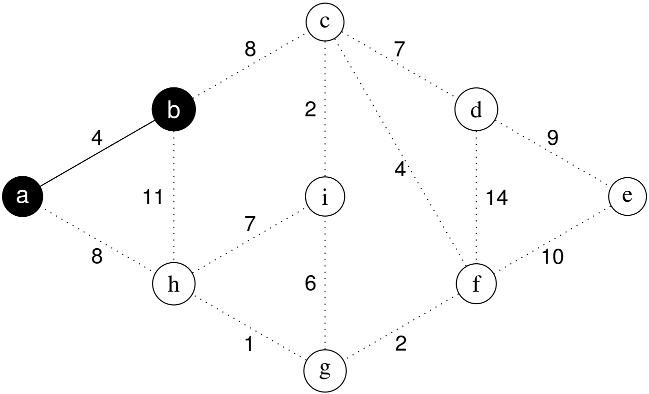
vertex	a	b	c	d	e	f	g	h	i
priority	0	∞	∞	∞	∞	∞	∞	∞	∞
pred									

PRIM'S ALGORITHM: A FEW EXAMPLE STEPS



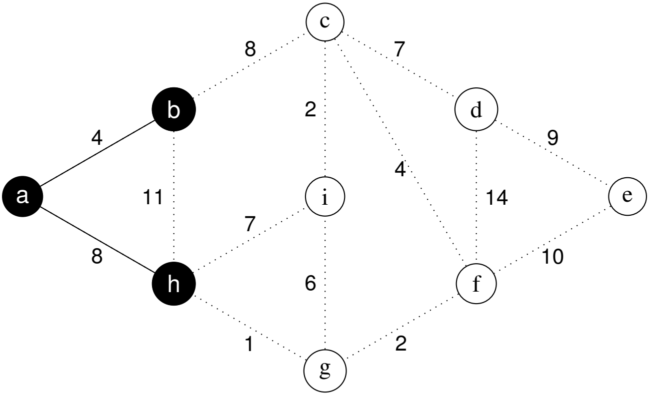
vertex	b	h	c	d	e	f	g	i
priority	4	8	∞	∞	∞	∞	∞	∞
pred	a	a						

PRIM'S ALGORITHM: A FEW EXAMPLE STEPS



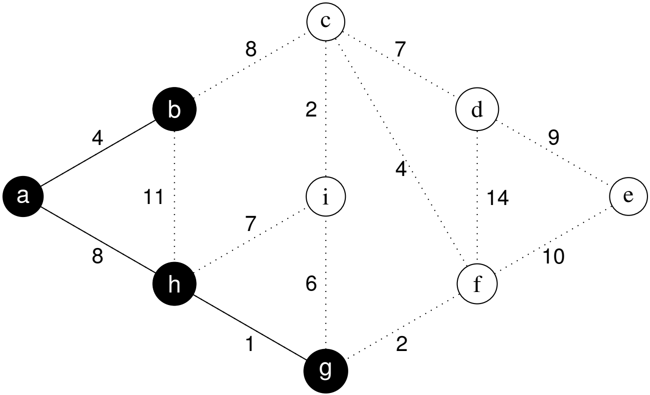
vertex	h	c	d	e	f	g	i
priority	8	8	∞	∞	∞	∞	∞
pred	a	b					

Prim's Algorithm: A Few Example Steps



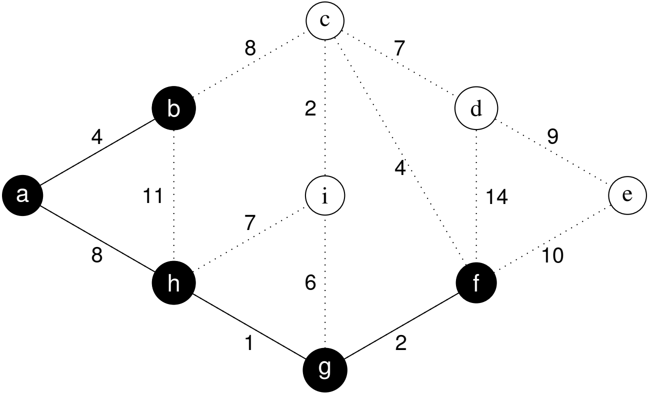
vertex	g	i	c	d	e	f
priority	1	7	8	∞	∞	∞
pred	h	h	b			

Prim's Algorithm: A Few Example Steps



vertex	f	i	c	d	e
priority	2	6	8	∞	∞
pred	g	g	b		

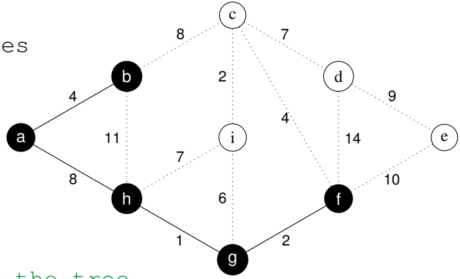
Prim's Algorithm: A Few Example Steps



vertex	c	i	e	d
priority	4	6	10	14
pred	f	g	f	f

Prim's Algorithm

```
Prim(V, E)
   S := new container() for edges
   PQ := new min-heap()
   start := pick a vertex
   PQ.insert(start, 0)
    for each vertex v ≠ start:
       # initialize pq
       PQ.insert(v, ∞)
    while not PQ.is_empty():
       # add least edge to grow the tree
        u := PQ.extract_min()
        S.add({u.pred, u})
    for each z in u's adjacency list:
       # update priorities based on u now in S
       if z in PQ && weight(u,z) < priority of z:
            PQ.decrease_priority(z, weight(u,z))
            z.pred := u
   return S
```



Prim's Algorithm Time Complexity

Q. How many times does a *vertex* enter/leave the *min-heap*?

A. Every *vertex* enters and leaves min-heap *once*: $\Theta(\lg n)$ per vertex, totalling $\Theta(n \lg n)$

Q. How many times can a *vertex's priority* decrease?

A. Every edge may trigger a change of priority: so $\forall v \in V, O(deg(v))$ which is $O(m)$ and takes $O(\lg n)$ for a total of $O(mlogn)$.

- Everything else, can be done in $\Theta(1)$ per *vertex* or per *edge*
- Total $O((n+m)\lg n)$ time worst case.

PRIM'S CORRECTNESS PROOF

To begin with we will first prove a useful property:

Cut Property: Let S be a nontrivial subset of V in G (i.e. $S \neq \emptyset$ and $S \neq V$). If (u, v) is the *lowest-cost edge* crossing $(S, V - S)$, then (u, v) is in *every MST* of G.

Proof.

- Suppose there exists an *MST* T that does not contain (u, v).
- Consider the sets S and $V - S$.
- There must exist a *path* from u to v.
- On this path, there must exist an *edge* e that *crosses* between $V - S$ into S.
- Since (u, v) is the least weight edge crossing between V and $S - V$, swapping (u, v) with e will reduce the weight of T.
- Therefore, T is not an *MST*.

Correctness of Prim's Algorithm

The correctness of *Prim's* Algorithm follows...from the **Cut Property**.

Q. How would the argument go?

A.

- Consider *optimal* MST O and *Prim's Algorithm* tree T.
- Order edges of T according to order they are selected.
- Consider the *first edge* $e = (u, v)$ in the ordering that is in T but not in O.
- At the stage of *Prim's* when e was added there was a set S of vertices such that $u \in S$, $v \in V - S$.
- If the edge weights are *unique*, by the **Cut Property**, e must belong to O. Therefore consider when *edge weights* are *not unique*.
- Since $e \notin O$, there exists a path p from u to v such that an edge $e' = (x, y)$ exists on p and $x \in S$ and $y \in V - S$.
- If $w(e') = w(e)$ then we can swap e' with e and the tree will still span tree G and be minimal.
- Must be that $w(e') \nless w(e)$ since then *Prim's* algorithm would have chosen it.
- If $w(e') > w(e)$ then swapping e' with e reduces the weight of O, which is a contradiction.

Common Theme for Greedy Correctness Proofs

- Let R be our *greedy rule* for selecting edges.
- Consider our *edge set* sorted according to R.
- Let O be an *optimal solution* that differs from our algorithm solution T.
- Consider our first edge $\{u,v\}$ in the edge set ordering that differs between O and T.
- Show that by definition of R, $\{u,v\} \in T$.
- Consider the set of selected vertices $S \subset V(T)$ when $\{u,v\}$ is chosen. By construction, $u \in S$ and $v \in V - S$.
- Consider the *path* p from u to v in O and the edge $e \in p$ that crosses from S to $V - S$.
- Show that swapping e and $\{u,v\}$ in O maintains the *MST properties* of O either *improves* or *maintains* the optimality of O.

★ You may find it helpful to know that many *greedy algorithm proofs* (for other types of problems) follow a similar template.

★ L02's notes have a different but similar template - another perspective.