CSCB63 Winter 2021

Week 5 Lecture 2 - Minimum Cost Spanning Trees

Anna Bretscher

February 8, 2021

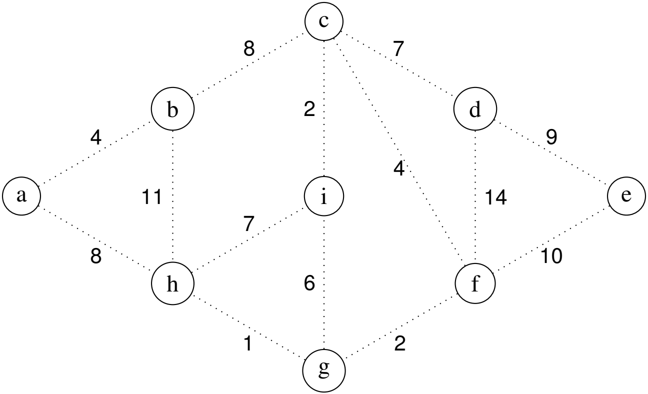
TODAY

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*.

(Edge-)Weighted Graph

- Many useful graphs have *numbers* or *weights* assigned to *edges*.
- Think of each edge e having a *price tag* $w(e)$.

(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 $w(e) < 0$.

(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 $w(e) < 0$.

(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

(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

(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 $w : E \to \mathbb{R}$

(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

(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 $w : E \rightarrow \mathbb{R}$
  - *undirected graphs*: $\{u, v\} = \{v, u\}$, *same* weight
  - *directed graphs*: (u, v) and (v, u) may have *different* weights

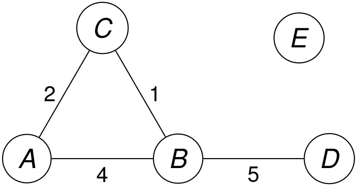
(Edge-)Weighted Graph

- 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 $e \in E$.

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?*

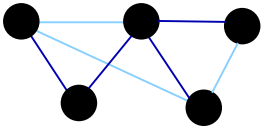
A.

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?*



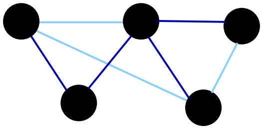
A.

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?*

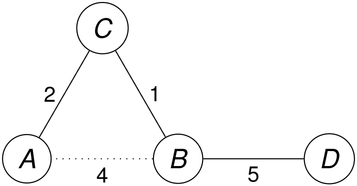


A.

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

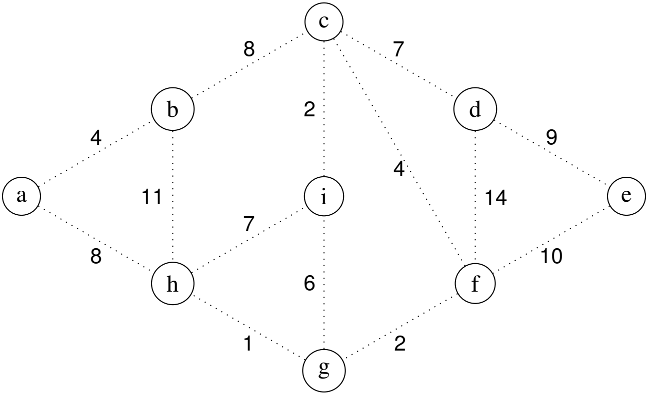


Usually just for *undirected, connected graphs*.

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

A.

Sample Graph

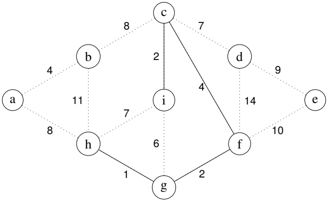


Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm finds an *MST* by

Proof by Contradiction.

-

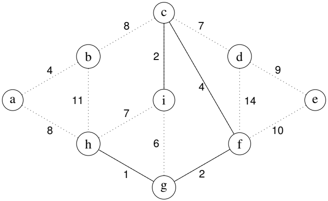


KRUSKAL'S ALGORITHM: PROOF OF CORRECTNESS

Kruskal's algorithm finds an *MST* by

Proof by Contradiction.

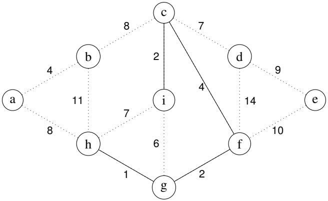
-
-



Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm finds an *MST* by

Proof by Contradiction.

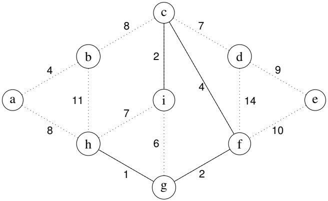


Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm finds an *MST* by

Proof by Contradiction.

-
-
-
-

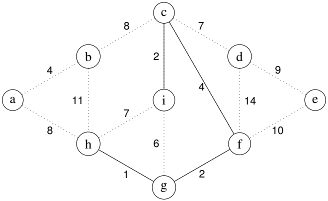


Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm finds an *MST* by

Proof by Contradiction.

-
-
-
-
-



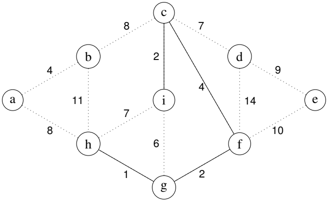
Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm finds an *MST* by

Proof by Contradiction.

-
-
-
-
-

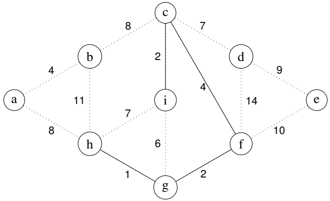
→



KRUSKAL'S ALGORITHM: PROOF OF CORRECTNESS

Kruskal's algorithm finds an *MST* by

Proof by Contradiction.

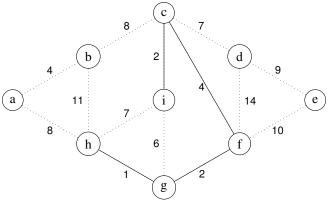


Kruskal's Algorithm: Proof of Correctness

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

Proof by Contradiction.

-

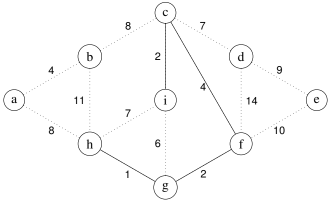


KRUSKAL'S ALGORITHM: PROOF OF CORRECTNESS

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

Proof by Contradiction.

-
-



Kruskal's Algorithm: Proof of Correctness

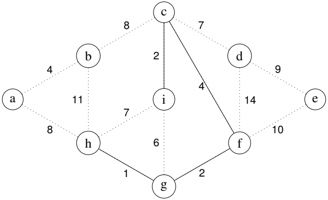
Kruskal's algorithm finds an *MST* by repeatedly adding the *least weight edge* that does *not induce* a *cycle*.

Proof by Contradiction.

-

-

Case 1. $w(e') = w_i$.



Kruskal's Algorithm: Proof of Correctness

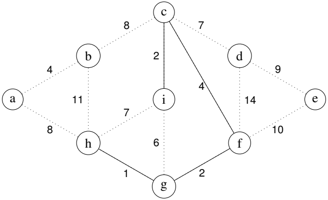
Kruskal's algorithm finds an *MST* by repeatedly adding the *least weight edge* that does *not induce* a *cycle*.

Proof by Contradiction.

-
-

Case 1. $w(e') = w_i$.

Case 2. $w(e') > w_i$.



Kruskal's Algorithm

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

A.

Kruskal's Algorithm

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

A.

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

A.

KRUSKAL'S ALGORITHM

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

A.

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

A.

```
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*

Storing Clusters: Easy Way - Linked Lists

Idea.

- each *cluster* is a *linked list*
- v.*cluster* is pointer to v's own *linked list*

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*

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

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

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:

-

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:

-
- If cluster *size doubles*, at most *how many* cluster updates can we do?

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:

-
- If cluster *size doubles*, at most *how many* cluster updates can we do?
-

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:

-
- If cluster *size doubles*, at most *how many* cluster updates can we do?
-

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:

-
- If cluster *size doubles*, at most *how many* cluster updates can we do?
-

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

Complexity

Complexity

- *Building* PQ and *removing* edges:

KRUSKAL'S ALGORITHM TIME COMPLEXITY

Complexity

- *Building* PQ and *removing* edges:
- v.cluster *updates*:

Complexity

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

KRUSKAL'S ALGORITHM TIME COMPLEXITY

Complexity

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

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

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

A.

Complexity

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

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

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

A.

Therefore,

Kruskal's Algorithm Time Complexity

Complexity

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

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

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

A.

Therefore,

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:

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*.

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.

Prim's Algorithm Again

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

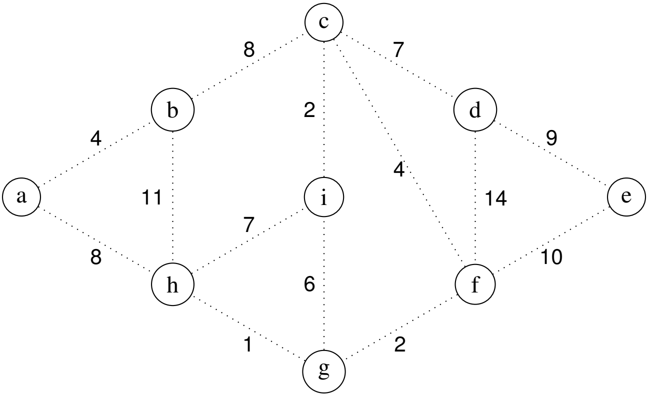
The *queue* is changed to a *min priority queue*.

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

Priority of vertex v =

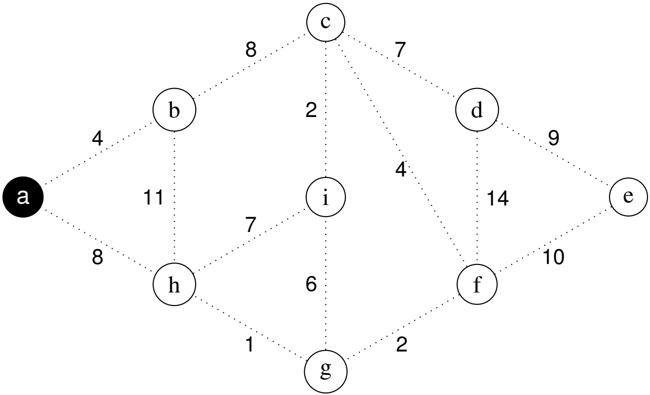
Let's step through the example again...

Prim's Algorithm: A Few Example Steps



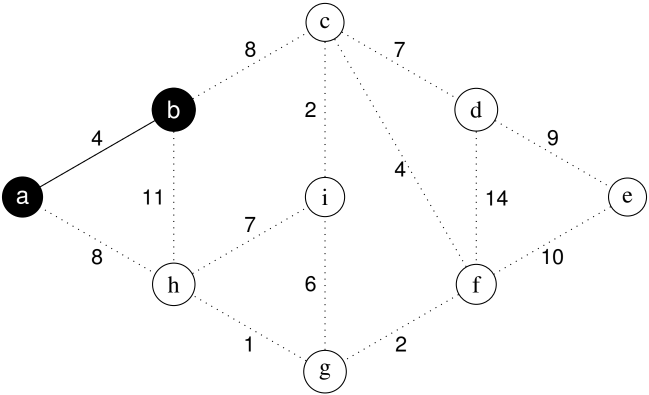
vertex	a	b	c	d	e	f	g	h	i
priority	0	∞	∞	∞	∞	∞	∞	∞	∞
pred									

Prim's Algorithm: A Few Example Steps



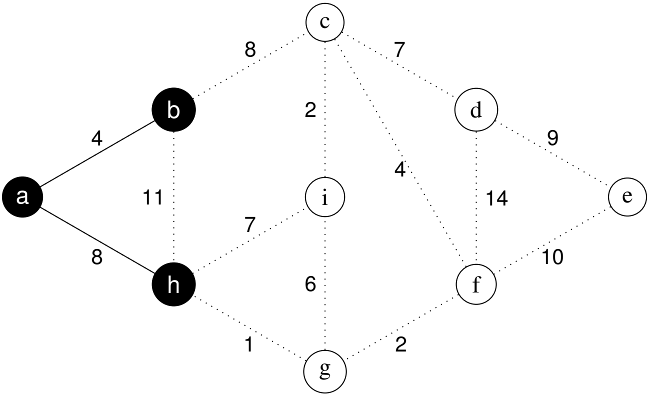
vertex	b	h	c	d	e	f	g	i
priority	4	8	∞	∞	∞	∞	∞	∞
pred	a	a						

Prim's Algorithm: A Few Example Steps



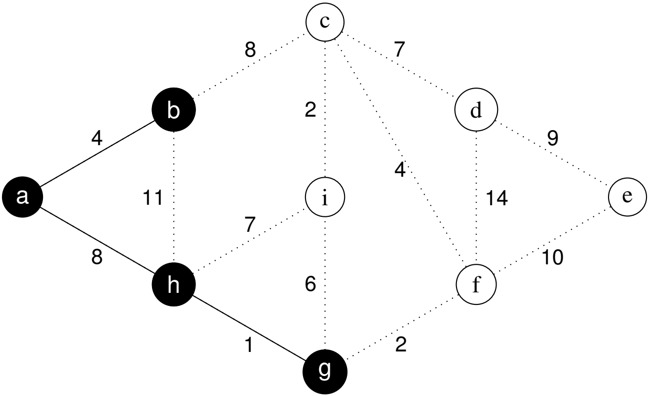
vertex	h	c	d	e	f	g	i
priority	8	8	∞	∞	∞	∞	∞
pred	a	b					

PRIM'S ALGORITHM: A FEW EXAMPLE STEPS



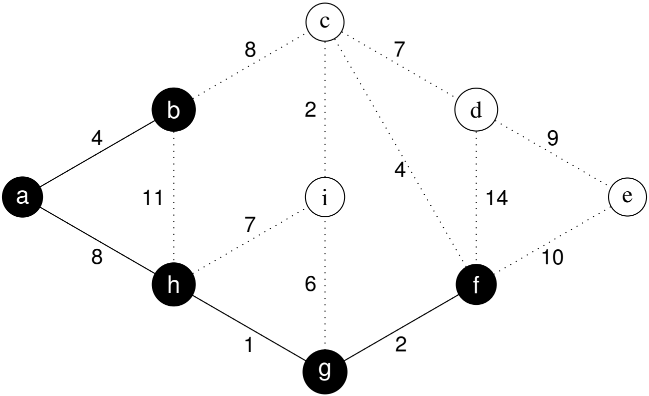
vertex	g	i	c	d	e	f
priority	1	7	8	∞	∞	∞
pred	h	h	b			

Prim's Algorithm: A Few Example Steps



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

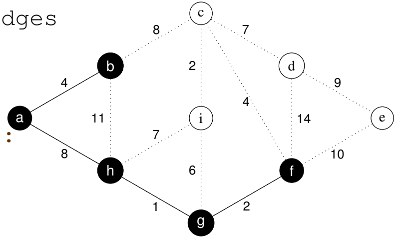
Prim's Algorithm: A Few Example Steps



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*?

Prim's Algorithm Time Complexity

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

A.

PRIM'S ALGORITHM TIME COMPLEXITY

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

A.

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

Prim's Algorithm Time Complexity

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

A.

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

A.

Prim's Algorithm Time Complexity

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

A.

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

A.

- Everything else, can be done in $\Theta(1)$ per *vertex* or per *edge*

Prim's Algorithm Time Complexity

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

A.

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

A.

- 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).

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 $V - S$.

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 $V - S$.
-

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 $V - S$.
-
-

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 $V - S$.
-
-
-

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 $V - S$.
-
-
-
- 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.

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.

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.

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.

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.
-

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.
-
-

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.
-
-
-

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.
-
-
-
-

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.
-
-
-
-
-

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.
-
-
-
-
-
-

Common Theme for Greedy Correctness Proofs

- Let R be our *greedy rule* for selecting edges.

Common Theme for Greedy Correctness Proofs

- Let *R* be our *greedy rule* for selecting edges.
- Consider our *edge set* sorted according to *R*.

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.

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.

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, $\{u, v\} \in T$.

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 $v \in V - S$.

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 $V - S$.

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.

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.

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.