EECS 336: Introduction to Algorithms	**Lecture 5**
Greedy by Value	Kruskal, Matroids

Reading: 4.5-4.6, MIT notes on matroids.

Last Time:

- greedy-by-value
- MST

Today:

- MST / matroid (cont.)
- dynamic greedy
- shortest paths, MSTs

Algorithm: Greedy-by-Value

1. $S = \emptyset$
2. Sort elts by decreasing value.
3. For each elt e (in sorted order):

    if $\{e\} \cup S$ is feasible

        add e to S

    else discard e.

Example 2: minimum spanning tree

input:

- graph $G = (V, E)$
- costs $c(e)$ on edges $e \in E$

output: spanning tree with minimum total cost.

Structural Observations about Forests

Def: $G' = (V, E')$ is a **subgraph** of $G = (V, E)$ if $E' \subseteq E$.

Def: An acyclic undirected graph is a **forest**

Fact 1: an MST on n vertices has $n - 1$ edges.

Lemma 1: If $G = (V, F)$ is a forest with m edges then it has $n - m$ connected components.

Proof: Induction (on number of edges)

base case: 0 edges, n CCs.

IH: assume true for m.

IS: show true for $m + 1$

- IH $\Rightarrow n - m$ CCs
- add new edge.
- must not create cycle

$\Rightarrow$ connects two connected components.

$\Rightarrow$ these 2 CCs become 1 CC.

$\Rightarrow$ $n - m - 1$ CCs.

QED

Lemma 2: (Augmentation Lemma) If $I, J \subset E$ are forests and $|I| < |J|$ then exists $e \in J \setminus I$ such that $I \cup \{e\}$ is a forest.

Proof:

Lemma 1

$\Rightarrow$ # CCs of $(V, I) >$ # CCs of $(V, J) \geq$ # CCs of $(V, I \cup J)$

$\Rightarrow$ add elements $e \in J$ to I until # CCs change.

[PICTURE]

$\Rightarrow$ $(V, I \cup \{e\})$ is acyclic.

Fact 2: subgraphs of acyclic graphs are acyclic

Correctness

"output is tree and has minimum cost"

Goal: understand why greedy-by-value works.

Lemma 1: Greedy outputs a forest.

Proof: Induction.

Lemma 2: if G is connected, Greedy outputs a tree.

Proof: (by contradiction)

Theorem: Greedy-by-Value is optimal for MSTs

Approach: "greedy stays ahead"

Proof: (by contradiction of first mistake)

- Greedy and OPT have $n-1$ edges (Fact 1)
- Let $I = \{i_1, \ldots, i_{n-1}\}$ be elt's of Greedy.

  (in order)
- Let $J = \{j_1, \ldots, j_{n-1}\}$ be elt's of OPT.

  (in order)
- Assume for contradiction: $c(I) > c(J)$
- Let r be first index with $c(j_r) < c(i_r)$
- Let $I_{r-1} = \{i_1, \ldots, i_{r-1}\}$
- Let $J_r = \{j_1, \ldots, j_r\}$
- $|I_{r-1}| < |J_r|$ & Augmentation Lemma

  $\Rightarrow$ exists $j \in J_r \setminus I_{r-1}$

  such that $I_{r-1} \cup \{j\}$ is acyclic.
- Suppose j considered after i_k $(k \leq r-1)$
- $I_k \subseteq I_{r-1}$

  $\Rightarrow$ $I_k \cup \{j\} \subseteq I_{r-1} \cup \{j\}$
- $I_{r-1} \cup \{j\}$ acyclic & Fact 2

  $\Rightarrow$ all subsets are acyclic

  $\Rightarrow$ $I_k \cup \{j\}$ acyclic

  $\Rightarrow$ j should have been added.

$\rightarrow\leftarrow$

Matroids

Def: A **set system** $M = (E, \mathcal{I})$ where

- E is **ground set**.
- $\mathcal{I} \subseteq 2^E$ is set of **compatible** subsets of E.

Question: When does greedy-by-value algorithm work?

Question: What properties of MSTs were necessary for greedy-by-value to work?

Answer:

- MSTs are same size (Fact 1)
- augmentation property (Lemma 2)
- downward closure (Fact 2)

Note: augmentation property implies Fact 1.

Def: A **matroid** is a **set system** $M = (E, \mathcal{I})$ satisfying:

M1 "subset property"
if $I \in \mathcal{I}$, all subsets of I are in $\mathcal{I}$.

M2 "augmentation property"
if $I, J \in \mathcal{I}$ and $|I| < |J|$, then exists $e \in J \setminus I$ such that $I \cup \{e\} \in \mathcal{I}$.

(compatible sets also called **independent sets**).

Corollary: acyclic subgraphs are a matroid.

Theorem: greedy algorithm is optimal iff feasible outputs are a **matroid**.

Proof:

- ($\Rightarrow$) same as for Theorem 1.
- ($\Leftarrow$) homework.

Conclusion: to see if greedy-by-value works, check matroid properties.

Dynamic Greedy Algorithms

"adjust ordering dynamically as greedy algorithm proceeds"

Template: Repeat:

- Process minimal element by metric.
- Adjust metric on remaining elements.

Note: priority queues useful for dynamic greedy algs.

Def: priority queue data structure

Operations:

- insert(v,k): adds elt v to queue with key k (priority)
- decreasekey(v,k): decreases the key of v to k

  (if key is less than k, leave it the same)
- deletemin: returns elt with minimum key.

Runtimes:

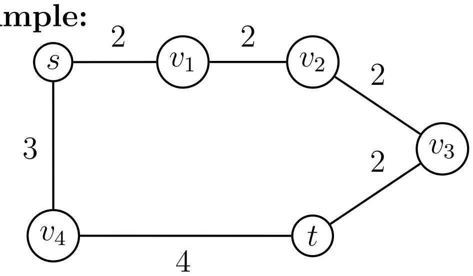
- can implement all operations in $O(\log n)$

Shortest Paths

"find short path from vertex s to t in graph"

E.g., driving directions, Internet routing.

Example:



Idea: given known distance to closest $S \subset V$, then distance of closest neighbor of S to s can be found. Then, induction.

Metric: shortest one-hop distance from vertices with known distances.

Update: (after processing vertex v)

- v's distance is known.
- update metric on unknown vertices if one-hop path from v is shorter.

Algorithm: Dijstra's Shortest Path Alg (w. Priority Q)

1. initialize
   (a) for all v, insert(v,∞)
   (b) deceasekey(v,0)
2. while queue not empty
   (a) (v,d) = deletemin()
   (b) if v = t, return d.
   (c) for each neighbor u of v:

   decreasekey(u,$d + c(v, u)$)

Runtime: $T(n, m) = m \log n$.

Correctness

Theorem: Dijkstra is optimal

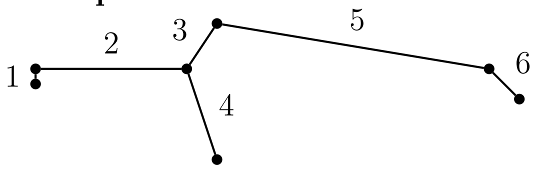
Proof: (by induction on known vertices, see text)

MSTs, revisited

Idea: grow tree from s by adding cheapest new vertex.

Note: as we add vertices, must reevaluate cost of vertices.

Example:



Idea: grow tree from start vertex adding closest vertex to any vertex in tree

Metric: minimum one-hop distance to any vertex in current tree.

Update: (after processing vertex v)

- add v to tree.
- update metric on non-tree vertices if one-hop distance to v is shorter.

Algorithm: Prim's MST Alg

1. initialize
   (a) for all v, insert(v,∞)
   (b) decreasekey(v,0)
2. while queue not empty
   (a) (v,d) = deletemin()
   (b) for each neighbor u of v:

   decreasekey(u,$c(v,u)$)

Runtime: $T(n,m) = O(n \log m)$

Correctness

Lemma: (cut lemma) For any (A,B)-cut and $e' = (u,v)$ the min cost edge crossing cut, e' is in every MST.

Proof: (contradiction)

Conclusion: each edge Prim adds is minimum edge on cut, therefore Prim never adds wrong edge.