Minimum Spanning Trees

Problem Definition

Algorithms: Design and Analysis, Part II

Overview

Informal Goal: Connect a bunch of points together as cheaply as possible.

Applications: Clustering (more later), networking.

Blazingly Fast Greedy Algorithms:

- Prim's Algorithm [1957; also Dijkstra 1959, Jarnik 1930]
- Kruskal's algorithm [1956]

$\Rightarrow O(m \log n)$ time (using suitable data structures)

\# of vertices

\# of edges

Problem Definition

vertices   edges

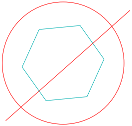
Input: Undirected graph $G = (V, E)$ and a cost c_e for each edge $e \in E$.

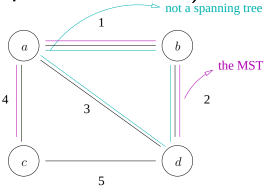
- Assume adjacency list representation (see Part I for details)
- OK if edge costs are negative

Output: minimum cost tree $T \subseteq E$ that spans all vertices.

i.e., sum of edge costs

I.e.: (1) T has no cycles, (2) the subgraph (V, T) is connected (i.e., contains path between each pair of vertices).





Standing Assumptions

Assumption #1: Input graph G is connected.

- Else no spanning trees.
- Easy to check in preprocessing (e.g., depth-first search).

Assumption #2: Edge costs are distinct.

- Prim + Kruskal remain correct with ties (which can be broken arbitrarily).
- Correctness proof a bit more annoying (will skip).

Minimum Spanning Trees

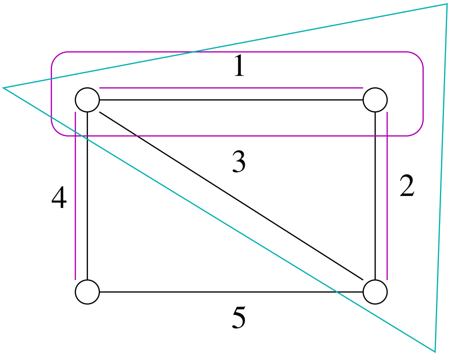
Prim's MST Algorithm

Algorithms: Design and Analysis, Part II

Example

[Purple edges = minimum spanning tree]

(Compare to Dijkstra's shortest-path algorithm)



Prim's MST Algorithm

- Initialize $X = \{s\}$ [$s \in V$ chosen arbitrarily]
- $T = \emptyset$ [invariant: $X =$ vertices spanned by tree-so-far T]
- While $X \neq V$
  - Let $e = (u, v)$ be the cheapest edge of G with $u \in X$, $v \notin X$.
  - Add e to T
  - Add v to X.

While loop: Increase # of spanned vertices in cheapest way possible.

Correctness of Prim's Algorithm

Theorem: Prim's algorithm always computes an MST.

Part I: Computes a spanning tree T^*.
[Will use basic properties of graphs and spanning trees] (Useful also in Kruskal's MST algorithm)

Part II: T^* is an MST.
[Will use the "Cut Property"] (Useful also in Kruskal's MST algorithm)

Later: Fast [$O(m \log n)$] implementation using heaps.

Minimum Spanning Trees

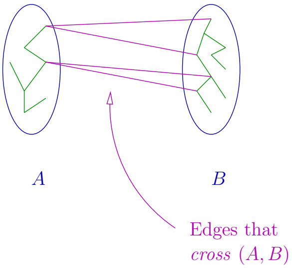
Algorithms: Design and Analysis, Part II

Correctness of Prim's Algorithm (Part I)

Cuts

Claim: Prim's algorithm outputs a spanning tree.

Definition: A cut of a graph $G = (V, E)$ is a partition of V into 2 non-empty sets.



Quiz on Cuts

Question: Roughly how many cuts does a graph with n vertices have?

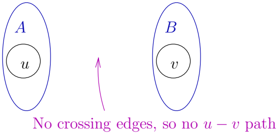
A) n C) 2^n (for each vertex, choose whether in A or in B)

B) n^2 D) n^n

Empty Cut Lemma

Empty Cut Lemma: A graph is not connected $\iff$ $\exists$ cut (A, B) with no crossing edges.

Proof: $(\Leftarrow)$ Assume the RHS. Pick any $u \in A$ and $v \in B$. Since no edges cross (A, B) there is no u, v path in G. $\Rightarrow$ G not connected.

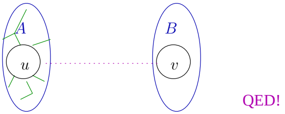


$(\Rightarrow)$ Assume the LHS. Suppose G has no $u - v$ path. Define

$A = \{$Vertices reachable from u in $G\}$ (u's connected component)

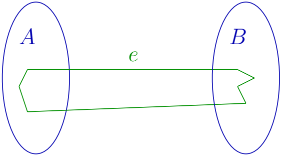
$B = \{$All other vertices$\}$ (all other connected components)

Note: No edges cross cut (A, B) (otherwise A would be bigger!)



Two Easy Facts

Double-Crossing Lemma: Suppose the cycle $C \subseteq E$ has an edge crossing the cut (A, B): then so does some other edge of C.

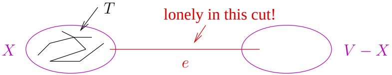


Lonely Cut Corollary: If e is the only edge crossing some cut (A, B), then it is not in any cycle. [If it were in a cycle, some other edge would have to cross the cut!]

Proof of Part I

Claim: Prim's algorithm outputs a spanning tree.
[Not claiming MST yet]

Proof: (1) Algorithm maintains invariant that T spans X
[straightforward induction - you check]



(2) Can't get stuck with $X \neq V$
[otherwise the cut $(X, V - X)$ must be empty; by Empty Cut Lemma input graph G is disconnected]

(3) No cycles ever get created in T. Why? Consider any iteration, with current sets X and T. Suppose e gets added.
Key point: e is the first edge crossing $(X, V - X)$ that gets added to $T \Rightarrow$ its addition can't create a cycle in T (by Lonely Cut Corollary). QED!

Minimum Spanning Trees

Correctness of Prim's Algorithm (Part II)

Algorithms: Design and Analysis, Part II

Correctness of Prim's Algorithm

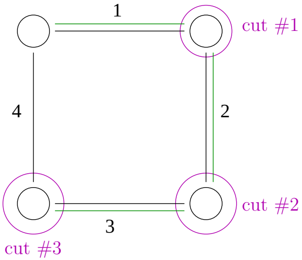
Theorem: Prim's algorithm always outputs a minimum-cost spanning tree.

Key Question: When is it "safe" to include an edge in the tree-so-far?

The Cut Property

CUT PROPERTY: Consider an edge e of G. Suppose there is a cut (A, B) such that e is the cheapest edge of G that crosses it. Then e belongs to the MST of G.

Turns out MST is unique if edge costs are distinct



Cut Property Implies Correctness

Claim: Cut Property $\Rightarrow$ Prim's algorithm is correct.

Proof: By previous video, Prim's algorithm outputs a spanning tree T^*.

Key point: Every edge $e \in T^*$ is explicitly justified by the Cut Property.

$\Rightarrow$ T^* is a subset of the MST

$\Rightarrow$ Since T^* is already a spanning tree, it must be the MST

QED!

Minimum Spanning Trees

Proof of the Cut Property

Algorithms: Design and Analysis, Part II

The Cut Property

Assumption: Distinct edge costs.

CUT PROPERTY: Consider an edge e of G. Suppose there is a cut (A, B) such that e is the cheapest edge of G that crosses it. Then e belongs to the MST of G.

Proof Plan

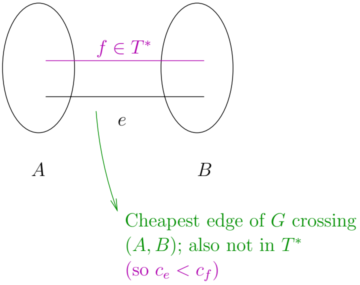
Will argue by contradiction, using an exchange argument.
[Compare to scheduling application]

Suppose there is an edge e that is the cheapest one crossing a cut (A, B), yet e is not in the MST T^*.

Idea: Exchange e with another edge in T^* to make it even cheaper (contradiction).

Question: Which edge to exchange e with?

Attempted Exchange



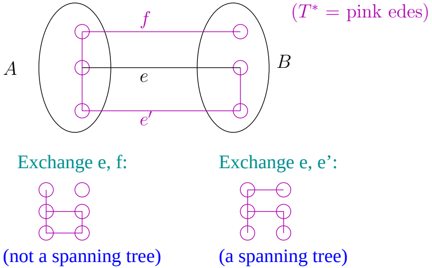
Note: Since T^* is connected, must construct an edge $f (\neq e)$ crossing (A, B).

Idea: Exchange e and f to get a spanning tree cheaper than T^* (contradiction).

Exchanging Edges

Question: Let T^* be a spanning tree of G, $e \notin T^*$, $f \in T^*$. Is $T^* \cup \{e\} - \{f\}$ a spanning tree of G?

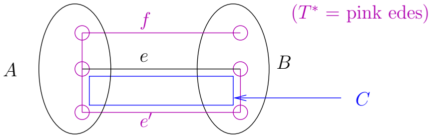
A) Yes always
B) No never
C) If e is the cheapest edge crossing some cut, then yes
D) Maybe, maybe not (depending on the choice of e and f)



Smart Exchanges

Hope: Can always find suitable edge e' so that exchange yields bona fide spanning tree of G.

How? Let C = cycle created by adding e to T^*.



By the Double-Crossing Lemma: Some other edge e' of C [with $e' \neq e$ and $e' \in T^*$] crosses (A, B).

You check: $T = T^* \cup \{e\} - \{e'\}$ is also a spanning tree.

Since $c_e < c_{e'}$, T cheaper than purported MST T^*, contradiction.

Minimum Spanning Trees

Fast Implementation of Prim's Algorithm

Algorithms: Design and Analysis, Part II

Running Time of Prim's Algorithm

- Initialize $X = \{s\}$ [$s \in V$ chosen arbitrarily]
- $T = \emptyset$ [invariant: X = vertices spanned by tree-so-far T]
- While $X \neq V$
  - Let $e = (u, v)$ be the cheapest edge of G with $u \in X$, $v \notin X$.
  - Add e to T, add v to X.

Running time of straightforward implementation:
- $O(n)$ iterations [where n = # of vertices]
- $O(m)$ time per iteration [where m = # of edges]

$\Rightarrow O(mn)$ time

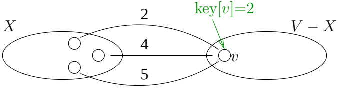
BUT CAN WE DO BETTER?

Prim's Algorithm with Heaps

[Compare to fast implementation of Dijkstra's algorithm]

Invariant #1: Elements in heap = vertices of $V - X$.

Invariant #2: For $v \in V - X$, key[v] = cheapest edge (u, v) with $i \in X$ (or $+\infty$ if no such edges exist).



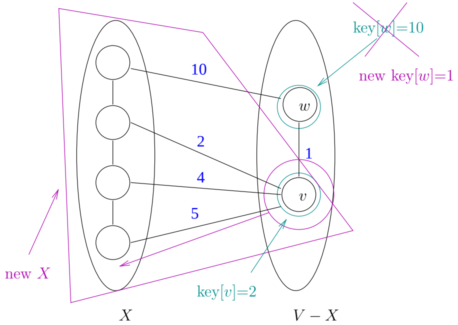
Check: Can initialize heap with $O(m + n \log n) = O(m \log n)$ preprocessing.

To compare keys | $n - 1$ Inserts | $m \geq n - 1$ since G connected

Note: Given invariants, Extract-Min yields next vertex $v \notin X$ and edge (u, v) crossing $(X, V - X)$ to add to X and T, respectively.

Quiz: Issue with Invariant #2

Question: What is: (i) current value of key[v] (ii) current value of key[w] (iii) value of key[w] after one more iteration of Prim's algorithm?



A) 11, 10, 4

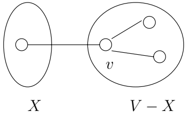
B) 2, 10, 10

C) 2, 10, 1

D) 2, 10, 2

Maintaining Invariant #2

Issue: Might need to recompute some keys to maintain Invariant #2 after each Extract-Min.



Pseudocode: When v added to X:
- For each edge $(v, w) \in E$:
  - If $w \in V - X$ $\rightarrow$ The only whose key might have changed (Update key if needed:)
    - Delete w from heap
    - Recompute key[w]:=min{key[w],c_{vw}}
    - Re-Insert into heap

Subtle point/exercise:

Think through book-keeping needed to pull this off

Running Time with Heaps

- Dominated by time required for heap operations

- $(n-1)$ Inserts during preprocessing

- $(n-1)$ Extract-Mins (one per iteration of while loop)

- Each edge (v, w) triggers one Delete/Insert combo
[When its first endpoint is sucked into X]

$\Rightarrow O(m)$ heap operations [Recall $m \geq n-1$ since G connected]

$\Rightarrow O(m \log n)$ time [As fast as sorting!]

Minimum Spanning Trees

Fast Implementation of Prim's Algorithm

Algorithms: Design and Analysis, Part II

Running Time of Prim's Algorithm

- Initialize $X = \{s\}$ [$s \in V$ chosen arbitrarily]
- $T = \emptyset$ [invariant: X = vertices spanned by tree-so-far T]
- While $X \neq V$
  - Let $e = (u, v)$ be the cheapest edge of G with $u \in X$, $v \notin X$.
  - Add e to T, add v to X.

Running time of straightforward implementation:
- $O(n)$ iterations [where n = # of vertices]
- $O(m)$ time per iteration [where m = # of edges]

$\Rightarrow O(mn)$ time

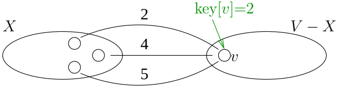
BUT CAN WE DO BETTER?

Prim's Algorithm with Heaps

[Compare to fast implementation of Dijkstra's algorithm]

Invariant #1: Elements in heap = vertices of $V - X$.

Invariant #2: For $v \in V - X$, key[v] = cheapest edge (u, v) with $i \in X$ (or $+\infty$ if no such edges exist).



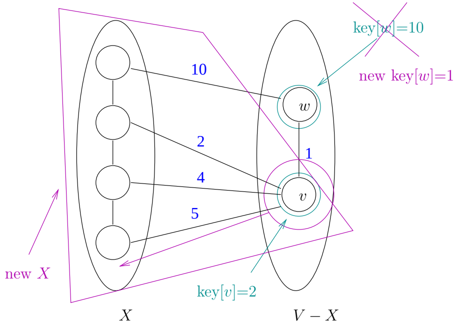
Check: Can initialize heap with $O(m + n \log n) = O(m \log n)$ preprocessing.

To compare keys | $n-1$ Inserts | $m \geq n-1$ since G connected

Note: Given invariants, Extract-Min yields next vertex $v \notin X$ and edge (u, v) crossing $(X, V - X)$ to add to X and T, respectively.

Quiz: Issue with Invariant #2

Question: What is: (i) current value of key[v] (ii) current value of key[w] (iii) value of key[w] after one more iteration of Prim's algorithm?

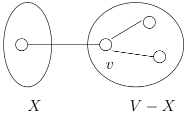


A) 11, 10, 4 C) 2, 10, 1

B) 2, 10, 10 D) 2, 10, 2

Maintaining Invariant #2

Issue: Might need to recompute some keys to maintain Invariant #2 after each Extract-Min.



Pseudocode: When v added to X:

- For each edge $(v, w) \in E$:
  - If $w \in V - X \rightarrow$ The only whose key might have changed (Update key if needed:)
    - Delete w from heap
    - Recompute key[w]:=min{key[w],c_{vw}}
    - Re-Insert into heap

Subtle point/exercise:

Think through book-keeping needed to pull this off

Running Time with Heaps

- Dominated by time required for heap operations

- $(n-1)$ Inserts during preprocessing

- $(n-1)$ Extract-Mins (one per iteration of while loop)

- Each edge (v,w) triggers one Delete/Insert combo
 [When its first endpoint is sucked into X]

$\Rightarrow O(m)$ heap operations [Recall $m \geq n-1$ since G connected]

$\Rightarrow O(m \log n)$ time [As fast as sorting!]

Minimum Spanning Trees

Kruskal's MST Algorithm

Algorithms: Design and Analysis, Part II

MST Review

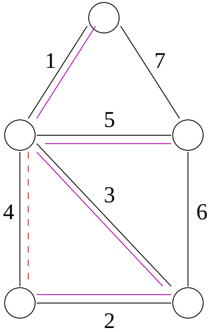
Input: Undirected graph $G = (V, E)$, edge costs c_e.

Output: Min-cost spanning tree (no cycles, connected).

Assumptions: G is connected, distinct edge costs.

Cut Property: If e is the cheapest edge crossing some cut (A, B), then e belongs to the MST.

Example



Kruskal's MST Algorithm

- Sort edges in order of increasing cost

[Rename edges $1, 2, \ldots, m$ so that $c_1 < c_2 < \ldots < c_m$]

- $T = \emptyset$
- For $i = 1$ to m
  - If $T \cup \{i\}$ has no cycles
    - Add i to T
- Return T

Minimum Spanning Trees

Correctness of Kruskal's Algorithm

Algorithms: Design and Analysis, Part II

Correctness of Kruskal (Part I)

Theorem: Kruskal's algorithm is correct.

Proof: Let $T^* =$ output of Kruskal's algorithm on input graph G.

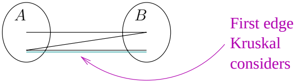
(1) Clearly T^* has no cycles.

(2) T^* is connected. Why?

(2a) By Empty Cut Lemma, only need to show that T^* crosses every cut.

(2b) Fix a cut (A, B). Since G connected at least one of its edges crosses (A, B).

Key point: Kruskal will include first edge crossing (A, B) that it sees [by Lonely Cut Corollary, cannot create a cycle]



Correctness of Kruskal (Part II)

(3) Every edge of T^* satisfied by the Cut Property. (Implies T^* is the MST)

Reason for (3): Consider iteration where edge (u, v) added to current set T. Since $T \cup \{(u, v)\}$ has no cycle, T has no $u - v$ path.

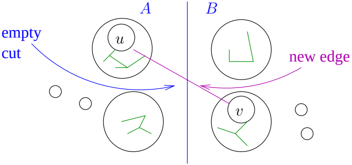
$\Rightarrow \exists$ empty cut (A, B) separating u and v. (As in proof of Empty Cut Lemma)

$\Rightarrow$ By (2b), no edges crossing (A, B) were previously considered by Kruskal's algorithm.

$\Rightarrow (u, v)$ is the first (+ hence the cheapest!) edge crossing (A, B).

$\Rightarrow (u, v)$ justified by the Cut Property. QED

Current State:



Minimum Spanning Trees

Implementing Kruskal's Algorithm via Union-Find

Algorithms: Design and Analysis, Part II

Kruskal's MST Algorithm

- Sort edges in order of increasing cost. ($O(m \log n)$, recall $m = O(n^2)$ assuming nonparallel edges)
- $T = \emptyset$
  - For $i = 1$ to m ($O(m)$ iterations)
    - If $T \cup \{i\}$ has no cycles ($O(n)$ time to check for cycle [Use BFS or DFS in the graph (V, T) which contains $\leq n - 1$ edges])
      - Add i to T
- Return T

Running time of straightforward implementation: ($m = \#$ of edges, $n = \#$ of vertices) $O(m \log n) + O(mn) = O(mn)$

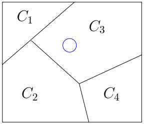
Plan: Data structure for $O(1)$-time cycle checks $\Rightarrow O(m \log n)$ time.

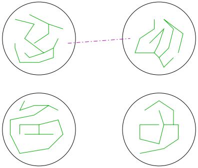
The Union-Find Data Structure

Raison d'être of union-find data structure: Maintain partition of a set of objects.

FIND(X): Return name of group that X belongs to.

UNION(C_i, C_j): Fuse groups C_i, C_j into a single one.





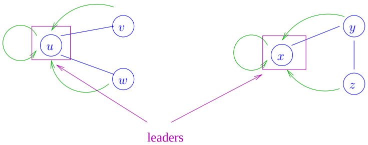
Why useful for Kruskal's algorithm: Objects = vertices

- Groups = Connected components w.r.t. chosen edges T.
- Adding new edge (u, v) to T $\iff$ Fusing connected components of u, v.

Union-Find Basics

Motivation: $O(1)$-time cycle checks in Kruskal's algorithm.

Idea #1: - Maintain one linked structure per connected component of (V, T).
- Each component has an arbitrary leader vertex.



Invariant: Each vertex points to the leader of its component ["name" of a component inherited from leader vertex]

Key point: Given edge (u, v), can check if u & v already in same component in $O(1)$ time. [if and only if leader pointers of u, v match, i.e., FIND(u)=FIND(v)] $\Rightarrow$ $O(1)$-time cycle checks!

Maintaining the Invariant

Note: When new edge (u, v) added to T, connected components of u & v merge.

Question: How many leader pointer updates are needed to restore the invariant in the worst case?

A) $\Theta(1)$

B) $\Theta(\log n)$

C) $\boxed{\Theta(n)}$ (e.g., when merging two components with $n/2$ vertices each)

D) $\Theta(m)$

Maintaining the Invariant (con'd)

Idea #2: When two components merge, have smaller one inherit the leader of the larger one. [Easy to maintain a size field in each component to facilitate this]

Question: How many leader pointer updates are now required to restore the invariant in the worst case?

A) $\Theta(1)$

B) $\Theta(\log n)$

C) $\Theta(n)$ (for same reason as before, i.e., when merging two components with $n/2$ vertices each)

D) $\Theta(m)$

Updating Leader Pointers

But: How many times does a single vertex v have its leader pointer updated over the course of Kruskal's algorithm?

A) $\Theta(1)$

B) $\Theta(\log n)$

C) $\Theta(n)$

D) $\Theta(m)$

Reason: Every time v's leader pointer gets updated, population of its component at least doubles $\Rightarrow$ Can only happen $\leq \log_2 n$ times.

Running Time of Fast Implementation

Scorecard:

$O(m \log n)$ time for sorting

$O(m)$ times for cycle checks [$O(1)$ per iteration]

$O(n \log n)$ time overall for leader pointer updates

$O(m \log n)$ total (Matching Prim's algorithm)

Minimum Spanning Trees

State-of-the-Art and Open Questions

Algorithms: Design and Analysis, Part II

State-of-the-Art MST Algorithms

Question: Can we do better than $O(m \log n)$? (Running time of Prim/Kruskal.)

Answer: Yes!

$O(m)$ randomized algorithm [Karger-Klein-Tarjan JACM 1995]

$O(m\,\alpha(n))$ deterministic [Chazelle JACM 2000]

"Inverse Ackerman Function": In particular, grows much slower than $\log^* n :=$ # of times you can apply log to n until result drops below 1 (inverse of "tower function" $2^{2^{2^{\cdots^{2}}}}$)

Open Questions

Weirdest of all: [Pettie/Ramachandran JACM 2002] Optimal deterministic MST algorithm, but precise asymptotic running time is unknown! [Between $\Theta(m)$ and $\Theta(m\alpha(n))$, but don't know where]

Open Questions:
- Simple randomized $O(m)$-time algorithm for MST [Sufficient: Do this just for the "MST verification" problem]
- Is there a deterministic $O(m)$-time algorithm?

Further reading: [Eisner 97]