Spanning Tree

Design and Analysis of Algorithms
Andrei Bulatov

The Minimum Spanning Tree Problem

Let G = (V,E) be a connected undirected graph

A subset $T \subseteq E$ is called a spanning tree of G if (V,T) is a tree

If every edge of G has a weight (positive) c_e then every spanning tree also has associated weight $\sum_{e \in T} c_e$

The Minimum Spanning Tree Problem

Instance

Graph G with edge weights

Objective

Find a spanning tree of minimum weight

Kruskal's Algorithm

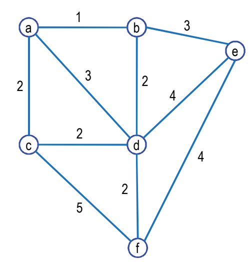
Input: `graph G with weights` c_e

Output: `a minimum spanning tree of G`

Method:

```
T:=∅
while |T|<|V|-1 do
  pick an edge e with minimum weight such that
    it is not from T and
    T∪{e} does not contain cycles
  set T:=T∪{e}
endwhile
```


Example



Kruskal's Algorithm: Soundness

Lemma (the Cut Property)

Assume that all edge weights are different. Let S be a nonempty subset of vertices, $S \neq V$, and let e be the minimum weight edge connecting S and V – S. Then every minimum spanning tree contains e

Use the exchange argument

Proof

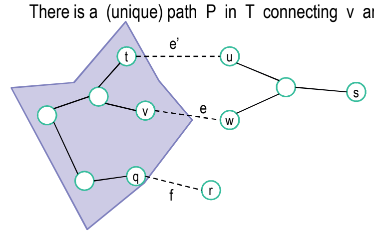
Let T be a spanning tree that does not contain e

We find an edge e' in T such that replacing e' with e we obtain another spanning tree that has smaller weight

Kruskal's Algorithm: Soundness (cntd)

Let e = (v,w)

There is a (unique) path P in T connecting v and w



Let u be the first vertex on this path not in S, and let e' = tu be the edge connecting S and V – S.

Kruskal's Algorithm: Soundness (cntd)

Replace in T edge e' with e

T' = (T – {e'}) ∪ {e}

T' remains a spanning tree

but lighter

QED

Kruskal's Algorithm: Soundness (cntd)

Theorem
Kruskal's algorithm produces a minimum spanning tree

Proof
T is a spanning tree
It contains no cycle
If (V,T) is not connected then there is an edge e such that T ∪ {e} contains no cycle.
The algorithm must add the lightest such edge

Kruskal's Algorithm: Soundness (cntd)

Proof (cntd)
T has minimum weight
We show that every edge added by Kruskal's algorithm must belong to every minimum spanning tree
Consider edge e = (v,w) added by the algorithm at some point, and let S be the set of vertices reachable from v in (V,T), where T is the set generated at the moment
Clearly $v \in S$, but $w \notin S$
Edge (v,w) is the lightest edge connecting S and V – S
Indeed if there is a lighter one, say, e', then it is not in T, and should be added instead

QED

Kruskal's Algorithm: Running Time

Suppose G has n vertices and m edges
Straightforward:
We need to add n – 1 edges, and every time we have to find the lightest edge that doesn't form a cycle
This takes n · m · (m + n), that is $O(m^2 n)$

Using a good data structure that stores connected components of the tree being constructed we can do it in O(m log n) time

Prim's Algorithm

Input: graph G with weights c_e
Output: a minimum spanning tree of G
Method:

```
choose a vertex s
set S:={s}, T:=∅
while S≠V do
  pick a node v not from S such that the value
```
$$\min_{e=(u,v),u\in S} c_e$$
```
  is minimal
  set S:=S∪{v} and T:=T∪{e}
endwhile
```


Prim's Algorithm: Soundness (cntd)

Theorem
Prim's algorithm produces a minimum spanning tree

Proof: DIY

Clustering

The k-Clustering Problem

Instance

A set U of n objects, $p_1, \ldots, p_n$ and a distance function $d(p_i, p_j)$ with natural properties

Objective

Find a partition (clustering) of U into k non-empty subsets such that the spacing (the minimal distance between points in different clusters) is maximal

Clustering vs. Spanning Tree

Let us run Kruskal's algorithm on the complete graph with vertices $p_1, \ldots, p_n$ and weights of edges determined by $d(p_i, p_j)$

Implement the data structure storing connected components of the growing graph

We terminate the algorithm once the number of connected components equals k

Theorem

The components constructed by the algorithm above constitute a k-clustering of maximum spacing

Optimal Caching

Caching

Memory Hierarchy

Eviction, Cache miss, Cache schedule

The Optimal Caching Problem

Instance

A data stream, and a cache size k

Objective

Find a cache schedule with fewest cache misses

Farthest-in-Future Principle

The following greedy algorithm provides an optimal cache schedule

```
when d needs to be brought into the cache,
   evict the item that is needed the farthest into the
   future
```


Least-Recently-Used Principle

The Farthest-in-future principle is not very practical

The reverse principle is normally used

```
when d needs to be brought into the cache,
   evict the item that referenced longest ago
```

It is not optimal anymore

For analysis see

Sleator, Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:2, 1985, 202-208