

Algorithm Design VIII

Greedy Algorithms

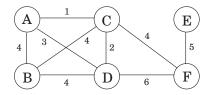
Guoqiang Li School of Software



Minimum Spanning Trees

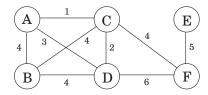
最小生成树





Suppose you are asked to network a collection of computers by linking selected pairs of them.



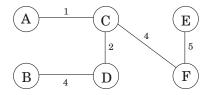


Suppose you are asked to network a collection of computers by linking selected pairs of them.

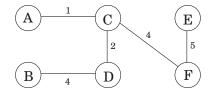
This translates into a graph problem in which

- nodes are computers,
- undirected edges are potential links, each with a maintenance cost.





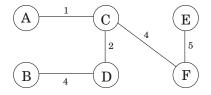




The goal is to

- pick enough of these edges that the nodes are connected,
- the total maintenance cost is minimum.





The goal is to

- pick enough of these edges that the nodes are connected,
- the total maintenance cost is minimum.

One immediate observation is that the optimal set of edges cannot contain a cycle.



Lemma (1)

Removing a cycle edge cannot disconnect a graph.



Lemma (1)

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called trees.



Lemma (1)

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called trees.

A tree with minimum total weight, is a minimum spanning tree, MST.



Lemma (1)

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called trees.

A tree with minimum total weight, is a minimum spanning tree, MST.

Input: An undirected graph G = (V, E); edge weights w_e



Properties of the Optimal Solutions 对于边u->v,如成环,则一定存在v->...->u的一个路径,而这时若移除一条环边 u->v,则整个图中u与v仍然连接(无向图条件下),所以Iemma1成立。

Lemma (1)

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called trees.

A tree with minimum total weight, is a minimum spanning tree, MST.

Input: An undirected graph G = (V, E); edge weights w_e

Output: A tree T = (V, E') with $E' \subseteq E$ that minimizes

$$\operatorname{weight}(T) = \sum_{e \in E'} w_e$$



Lemma (2)

A tree on n nodes has n-1 edges.



Lemma (2)

A tree on n nodes has n-1 edges.

To build the tree one edge at a time, starting from an empty graph.



Lemma (2)

A tree on n nodes has n-1 edges.

To build the tree one edge at a time, starting from an empty graph.

Each of the n nodes is disconnected from the others, in a connected component by itself.



Lemma (2)

A tree on n nodes has n-1 edges.

To build the tree one edge at a time, starting from an empty graph.

Each of the n nodes is disconnected from the others, in a connected component by itself.

As edges are added, these components merge. Since each edge unites two different components, exactly n-1 edges are added by the time the tree is fully formed.

另外一种证明思路:使用数学归纳法证明。直接将整个tree分成多个SCC。之后,从任意一个SCC中挑选一个叶子结点将其先移出这个SCC,之后这时只剩下n-1个节点,根据数学归纳金的之前的假设,可以知道一定存在n-2条边构成这个子树。之后将之前移出去的节点在添加回来,由于是tree,一定没有环路,所以多了一个新的节点,只能多一条边,所以n个节点的tree就有n-1条边。

Lemma (2)

A tree on n nodes has n-1 edges.

To build the tree one edge at a time, starting from an empty graph.

Each of the n nodes is disconnected from the others, in a connected component by itself.

As edges are added, these components merge. Since each edge unites two different components, exactly n-1 edges are added by the time the tree is fully formed.

When a particular edge (u, v) comes up, we can be sure that u and v lie in separate connected components, for otherwise there would already be a path between them and this edge would create a cycle.



Lemma (3)

Any connected, undirected graph G = (V, E) with |E| = |V| - 1 is a tree.



Lemma (3)

Any connected, undirected graph G = (V, E) with |E| = |V| - 1 is a tree.

It is the converse of Lemma (2).



Lemma (3)

Any connected, undirected graph G = (V, E) with |E| = |V| - 1 is a tree.

It is the converse of Lemma (2). We just need to show that G is acyclic.



Lemma (3)

Any connected, undirected graph G = (V, E) with |E| = |V| - 1 is a tree.

It is the converse of Lemma (2). We just need to show that G is acyclic.

While the graph contains a cycle, remove one edge from this cycle.



Lemma (3)

Any connected, undirected graph G = (V, E) with |E| = |V| - 1 is a tree.

It is the converse of Lemma (2). We just need to show that G is acyclic.

While the graph contains a cycle, remove one edge from this cycle.

The process terminates with some graph $G' = (V, E'), E' \subseteq E$, which is acyclic and, by Lemma (1), is also connected.



Lemma (3)

Any connected, undirected graph G = (V, E) with |E| = |V| - 1 is a tree.

It is the converse of Lemma (2). We just need to show that G is acyclic.

While the graph contains a cycle, remove one edge from this cycle.

The process terminates with some graph $G' = (V, E'), E' \subseteq E$, which is acyclic and, by Lemma (1), is also connected.

Therefore G' is a tree, whereupon |E'| = |V| - 1 by Lemma (2). So E' = E, no edges were removed, and G was acyclic to start with.



Lemma (4)

An undirected graph is a tree if and only if there is a unique path between any pair of nodes.



Lemma (4)

An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

In a tree, any two nodes can only have one path between them; for if there were two paths, the union of these paths would contain a cycle.



Lemma (4)

An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

In a tree, any two nodes can only have one path between them; for if there were two paths, the union of these paths would contain a cycle.

On the other hand, if a graph has a path between any two nodes, then it is connected. If these paths are unique, then the graph is also acyclic.

(注意这个I emma的前提是无向图)

A Greedy Approach



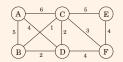
Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from *E* according to the following rule.

Repeatedly add the next lightest edge that doesn't produce a cycle.

Example

Starting with an empty graph and then attempt to add edges in increasing order of weight

$$B-C; C-D; B-D; C-F; D-F; E-F; A-D; A-B; C-E; A-C$$





The Cut Property



Lemma

Suppose edges X are part of a MST of G = (V, E). Pick any subset of nodes S for which X does not cross between S and $V \setminus S$, and let e be the lightest edge across this partition. Then

 $X \cup \{e\}$

is part of some MST.

- 1. 注意同一个图的最小生成树可能有多个!
- 2. 这个I emma没有说X U {e}和之前的X一定要属于同一个MST

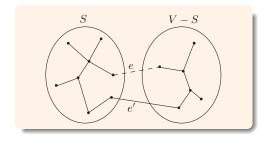
The Cut Property



A cut is any partition of the vertices into two groups, S and $V \setminus S$.

It is safe to add the lightest edge across any cut, provided X has no edges across the cut.

若X可以across两个cut,则就会成环,这时候就已经不是一棵树了。





Proof:



Proof:

Edges X are part of some MST T;



Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.



Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T.



Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.



Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T.



Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle.



Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle.

This cycle must also have some other edge e' across the cut $(S, V \setminus S)$.



Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle.

This cycle must also have some other edge e' across the cut $(S, V \setminus S)$. If we now remove e'

$$T' = T \cup \{e\} \backslash \{e'\}$$

which we will show to be a tree.



Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle.

This cycle must also have some other edge e' across the cut $(S, V \setminus S)$. If we now remove e'

$$T' = T \cup \{e\} \backslash \{e'\}$$

which we will show to be a tree.

T' is connected by Lemma (1), since e' is a cycle edge.



Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle.

This cycle must also have some other edge e' across the cut $(S, V \setminus S)$. If we now remove e'

$$T' = T \cup \{e\} \backslash \{e'\}$$

which we will show to be a tree.

T' is connected by Lemma (1), since e' is a cycle edge. And it has the same number of edges as T; so by Lemma (2) and Lemma (3), it is also a tree.



Proof:



Proof:

T' is a minimum spanning tree, since



Proof:

T' is a minimum spanning tree, since

$$weight(T') = weight(T) + w(e) - w(e')$$



Proof:

T' is a minimum spanning tree, since

$$weight(T') = weight(T) + w(e) - w(e')$$

Both e and e' cross between S and $V \setminus S$, and e is the lightest edge of this type.



Proof:

T' is a minimum spanning tree, since

$$weight(T') = weight(T) + w(e) - w(e')$$

Both e and e' cross between S and $V \setminus S$, and e is the lightest edge of this type. Therefore $w(e) \leq w(e')$, and

$$weight(T') \le weight(T)$$



Proof:

T' is a minimum spanning tree, since

$$weight(T') = weight(T) + w(e) - w(e')$$

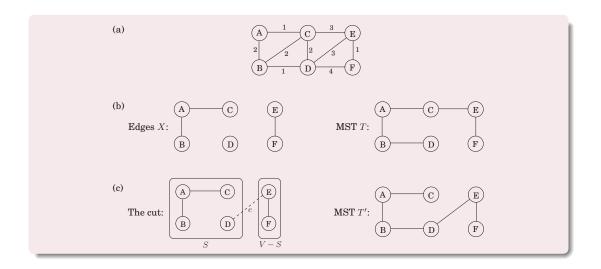
Both e and e' cross between S and $V \setminus S$, and e is the lightest edge of this type. Therefore $w(e) \leq w(e')$, and

$$weight(T') \le weight(T)$$

Since T is an MST, it must be the case that weight(T') = weight(T) and that T' is also an MST.

An Example of Cut Property





Kruskal's Algorithm



```
KRUSKAL (G, w)
input: A connected undirected graph G = (V, E), with edge weight w_e
output: A minimum spanning tree defined by the edges X
                                kruskal 算法在添加边的同时还需要保证添加的新边不会成环。
for all u \in V do
                                这里使用了**并查集**的方法来解决。即对于每一个cut过的划分,使用一个set来记录其中包括的点,当添加一条新的边的时候,若边的两端都同时位于一个并查集合中,则这条边一定成环(因为已经位于同一个集合中的点一定是之间有联通路径的),所以就不
   makeset (u);
end
X = \{ \};
                                能加入。
Sort the edges E by weight;
for all (u, v) \in E in increasing order of weight do
   if find (u) \neq \text{find } (v) then
       add (u, v) to X;
       union (u,v)
   end
end
```

Kruskal's Algorithm



```
\begin{array}{ll} \operatorname{makeset}(x) & \operatorname{create\ a\ singleton\ set\ containing\ } x & |V| \\ \operatorname{find}(x) & \operatorname{find\ the\ set\ that\ } x \ \operatorname{belong\ to} & 2\cdot|E| \\ \operatorname{union}(x,y) & \operatorname{merge\ the\ sets\ containing\ } x \ \operatorname{and\ } y & |V|-1 \end{array}
```



Union by rank

We store a set is by a directed tree.



Union by rank

We store a set is by a directed tree. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree.



Union by rank

We store a set is by a directed tree. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree.

This root element is a convenient representative, or name, for the set.



Union by rank

We store a set is by a directed tree. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree.

This root element is a convenient representative, or name, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.



Union by rank

We store a set is by a directed tree. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree.

This root element is a convenient representative, or name, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

In addition to a parent pointer π , each node also has a rank that, for the time being, should be interpreted as the height of the subtree hanging from that node.



MAKESET (x)

$$\pi(x) = x;$$

rank(x)=0;



```
MAKESET (x)
\pi(x) = x;
rank (x)=0;
```

```
FIND (x)

while x \neq \pi(x) do

x = \pi(x);

end

return (x);
```



MAKESET (
$$x$$
)
$$\pi(x) = x;$$
 rank (x)=0;

MAKESET is a constant-time operation.

```
FIND (x)

while x \neq \pi(x) do

x = \pi(x);

end

return (x);
```



```
MAKESET (x)
\pi(x) = x;
rank (x)=0;
```

```
FIND (x)

while x \neq \pi(x) do
x = \pi(x);
end
return (x);
```

MAKESET is a constant-time operation.

FIND follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree.



```
MAKESET (x)
\pi(x) = x;
rank (x)=0;
```

```
FIND (x)

while x \neq \pi(x) do

x = \pi(x);

end

return (x);
```

MAKESET is a constant-time operation.

FIND follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree.

The tree actually gets built via the third operation, UNION, and so we must make sure that this procedure keeps trees shallow.

Union



```
UNION (x, y)
r_x = \text{find}(x);
r_y = \text{find}(y);
if r_x = r_y then return;
if rank (r_x) > rank (r_y) then
   \pi(r_y) = r_x;
end
else
   \pi(r_x) = r_y;
   if rank (r_x) = rank (r_y) then rank (r_y) = rank (r_y) +1;
end
```

An Example



After $makeset(A), makeset(B), \dots, makeset(G)$:















After union(A, D), union(B, E), union(C, F):





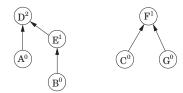




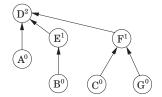
An Example



After union(C, G), union(E, A):



After union(B, G):





Lemma (1)

For any non-root x, $rank(x) < rank(\pi(x))$.



Lemma (1)

For any non-root x, $rank(x) < rank(\pi(x))$.

Proof Sketch:



Lemma (1)

For any non-root x, $rank(x) < rank(\pi(x))$.

Proof Sketch:

By design, the rank of a node is exactly the height of the subtree rooted at that node. This means, for instance, that as you move up a path toward a root node, the rank values along the way are strictly increasing.



Lemma (2)

Any root node of $\operatorname{rank} k$ has least 2^k nodes in its tree.



Lemma (2)

Any root node of rank k has least 2^k nodes in its tree.

Proof Sketch:



Lemma (2)

Any root node of rank k has least 2^k nodes in its tree.

Proof Sketch:

A root node with rank k is created by the merger of two trees with roots of rank k-1. By induction to get the results.



Lemma (3)

If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.



Lemma (3)

If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.

Proof Sketch:



Lemma (3)

If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.

Proof Sketch:

A node of rank k has at least 2^k descendants.



Lemma (3)

If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.

Proof Sketch:

A node of rank k has at least 2^k descendants.

Any internal node was once a root, and neither its rank nor its set of descendants has changed since then.

Properties



Lemma (3)

If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.

Proof Sketch:

A node of rank k has at least 2^k descendants.

Any internal node was once a root, and neither its rank nor its set of descendants has changed since then.

Different rank-k nodes cannot have common descendants. Any element has at most one ancestor of rank k.



With the data structure as presented so far, the total time for Kruskal's algorithm becomes



With the data structure as presented so far, the total time for Kruskal's algorithm becomes

• $O(|E| \log |V|)$ for sorting the edges $(\log |V| \approx \log |E|)$,



With the data structure as presented so far, the total time for Kruskal's algorithm becomes

- $O(|E|\log |V|)$ for sorting the edges $(\log |V| \approx \log |E|)$,
- $O(|E|\log |V|)$ for the union and find operations that dominate the rest of the algorithm.



With the data structure as presented so far, the total time for Kruskal's algorithm becomes

- $O(|E| \log |V|)$ for sorting the edges $(\log |V| \approx \log |E|)$,
- $O(|E|\log|V|)$ for the union and find operations that dominate the rest of the algorithm.

But what if the edges are given to us sorted? Or if the weights are small (say, O(|E|)) so that sorting can be done in linear time?



With the data structure as presented so far, the total time for Kruskal's algorithm becomes

- $O(|E|\log |V|)$ for sorting the edges $(\log |V| \approx \log |E|)$,
- $O(|E|\log|V|)$ for the union and find operations that dominate the rest of the algorithm.

But what if the edges are given to us sorted? Or if the weights are small (say, O(|E|)) so that sorting can be done in linear time?

THEN THE DATA STRUCTURE PART BECOMES THE BOTTLENECK!



With the data structure as presented so far, the total time for Kruskal's algorithm becomes

- $O(|E|\log |V|)$ for sorting the edges $(\log |V| \approx \log |E|)$,
- $O(|E|\log |V|)$ for the union and find operations that dominate the rest of the algorithm.

But what if the edges are given to us sorted? Or if the weights are small (say, O(|E|)) so that sorting can be done in linear time?

THEN THE DATA STRUCTURE PART BECOMES THE BOTTLENECK!

The main question:



With the data structure as presented so far, the total time for Kruskal's algorithm becomes

- $O(|E|\log |V|)$ for sorting the edges $(\log |V| \approx \log |E|)$,
- $O(|E|\log |V|)$ for the union and find operations that dominate the rest of the algorithm.

But what if the edges are given to us sorted? Or if the weights are small (say, O(|E|)) so that sorting can be done in linear time?

THEN THE DATA STRUCTURE PART BECOMES THE BOTTLENECK!

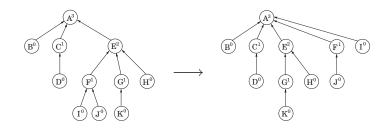
The main question:

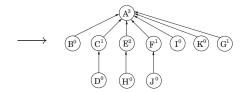
How can we perform union's and find's faster than $\log n$?



```
FIND (x) if x \neq \pi(x) then \pi(x)=FIND (\pi(x)); return (\pi(x));
```









The benefit of this simple alteration is long-term rather than instantaneous and thus necessitates a particular kind of analysis.



The benefit of this simple alteration is long-term rather than instantaneous and thus necessitates a particular kind of analysis.

We need to look at sequences of find and union operations, starting from an empty data structure, and determine the average time per operation.



The benefit of this simple alteration is long-term rather than instantaneous and thus necessitates a particular kind of analysis.

We need to look at sequences of find and union operations, starting from an empty data structure, and determine the average time per operation.

This amortized cost turns out to be just barely more than O(1), down from the earlier $O(\log n)$.

A General Kruskal's Algorithm



```
\begin{split} X &= \{\ \}; \\ \text{repeat until } |X| &= |V|-1; \\ \text{pick a set } S \subset V \text{ for which } X \text{ has no edges between } S \text{ and } V-S; \\ \text{let } e \in E \text{ be the minimum-weight edge between } S \text{ and } V-S; \\ X &= X \cup \{e\}; \end{split}
```



A popular alternative to Kruskal's algorithm is Prim's, in which the intermediate set of edges X always forms a subtree, and S is chosen to be the set of this tree's vertices.



A popular alternative to Kruskal's algorithm is Prim's, in which the intermediate set of edges X always forms a subtree, and S is chosen to be the set of this tree's vertices.

On each iteration, the subtree defined by X grows by one edge.

The lightest edge between a vertex in S and a vertex outside S. We can equivalently think of S as growing to include the vertex $v \notin S$ of smallest cost:



prim算法与Dijktra算法思路基本一致,只不过他只关心当前的最短边,而不是从起点到当前点的最短路径,所以在update的时候无需考虑之前处理过的点

A popular alternative to Kruskal's algorithm is Prim's, in which the intermediate set of edges X always forms a subtree, and S is chosen to be the set of this tree's vertices.

On each iteration, the subtree defined by X grows by one edge.

The lightest edge between a vertex in S and a vertex outside S. We can equivalently think of S as growing to include the vertex $v \notin S$ of smallest cost:

 $cost(v) = \min_{u \in S} w(u, v)$

The Algorithm



```
PRIM(G, w)
input: A connected undirected graph G = (V, E), with edge weights w_e
output: A minimum spanning tree defined by the array prev
for all u \in V do
    cost(u) = \infty;
    prev(u) = nil;
end
pick any initial node u_0;
cost(u_0) = 0;
H = \text{makequeue}(V) \setminus \text{using cost-values as keys};
while H is not empty do
    v = \text{deletemin}(H);
    for each (v, z) \in E do
         if cost(z) > w(v, z) then
             cost(v) = w(v, z); prev(z) = v;
             decreasekey (H,z);
        end
    end
end
```

Dijkstra's Algorithm



```
DIJKSTRA (G, l, s)
input: Graph G = (V, E), directed or undirected; positive edge length \{l_e \mid e \in E\};
        Vertex s \in V
output: For all vertices u reachable from s, dist(u) is the set to the distance from s to
for all u \in V do
    dist(u) = \infty;
    prev(u) = nil;
end
dist(s) = 0;
H = \text{makequeue}(V) \setminus \text{using dist-values as keys};
while H is not empty do
    u = \text{deletemin}(H);
    for all edge (u, v) \in E do
         if dist(v) > dist(u) + l(u, v) then
              dist(v) = dist(u) + l(u, v); prev(v) = u;
              decreasekey (H,v);
         end
end
```

Set Cover



A county is in its early stages of planning and is deciding where to put schools.



A county is in its early stages of planning and is deciding where to put schools.

There are only two constraints:



A county is in its early stages of planning and is deciding where to put schools.

There are only two constraints:

· each school should be in a town,



A county is in its early stages of planning and is deciding where to put schools.

There are only two constraints:

- each school should be in a town,
- and no one should have to travel more than 30 miles to reach one of them.



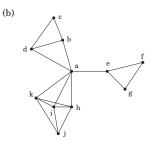
A county is in its early stages of planning and is deciding where to put schools.

There are only two constraints:

- each school should be in a town,
- and no one should have to travel more than 30 miles to reach one of them.

Q: What is the minimum number of schools needed?







This is a typical (cardinality) set cover problem.



This is a typical (cardinality) set cover problem.

• For each town x, let S_x be the set of towns within 30 miles of it.



This is a typical (cardinality) set cover problem.

- For each town x, let S_x be the set of towns within 30 miles of it.
- A school at x will essentially "cover" these other towns.



This is a typical (cardinality) set cover problem.

- For each town x, let S_x be the set of towns within 30 miles of it.
- A school at x will essentially "cover" these other towns.
- ullet The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

Set Cover Problem

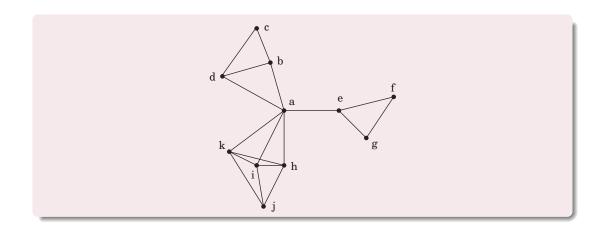


SET COVER

- Input: A set of elements B, sets $S_1, \ldots, S_m \subseteq B$
- Output: A selection of the S_i whose union is B.
- Cost: Number of sets picked.

The Example









Lemma

Suppose B contains n elements and that the optimal cover consists of OPT sets. Then the greedy algorithm will use at most $\ln n \cdot OPT$ sets.



Lemma

Suppose B contains n elements and that the optimal cover consists of OPT sets. Then the greedy algorithm will use at most $\ln n \cdot OPT$ sets.

Proof.



Lemma

Suppose B contains n elements and that the optimal cover consists of OPT sets. Then the greedy algorithm will use at most $\ln n \cdot OPT$ sets.

Proof.

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$).



Lemma

Suppose B contains n elements and that the optimal cover consists of OPT sets. Then the greedy algorithm will use at most $\ln n \cdot OPT$ sets.

Proof.

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$).

Since these remaining elements are covered by the optimal OPT sets, there must be some set with at least n_t/OPT of them.



Lemma

Suppose B contains n elements and that the optimal cover consists of OPT sets. Then the greedy algorithm will use at most $\ln n \cdot OPT$ sets.

Proof.

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$).

Since these remaining elements are covered by the optimal OPT sets, there must be some set with at least n_t/OPT of them.

Therefore, the greedy strategy will ensure that

$$n_{t+1} \le n_t - \frac{n_t}{OPT} = n_t (1 - \frac{1}{OPT})$$



Lemma

Suppose B contains n elements and that the optimal cover consists of OPT sets. Then the greedy algorithm will use at most $\ln n \cdot OPT$ sets.

Proof.

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$).

Since these remaining elements are covered by the optimal OPT sets, there must be some set with at least n_t/OPT of them.

Therefore, the greedy strategy will ensure that

$$n_{t+1} \le n_t - \frac{n_t}{OPT} = n_t \left(1 - \frac{1}{OPT}\right)$$

which by repeated application implies

$$n_t \le n_0 (1 - \frac{1}{OPT})^t$$





A more convenient bound can be obtained from the useful inequality

$$1 - x \le e^{-x} \text{ for all } x$$

with equality if and only if x = 0,



A more convenient bound can be obtained from the useful inequality

$$1 - x \le e^{-x} \text{ for all } x$$

with equality if and only if x = 0,

Thus

$$n_t \le n_0 (1 - \frac{1}{OPT})^t < n_0 (e^{-\frac{1}{OPT}})^t = ne^{-\frac{t}{OPT}}$$



A more convenient bound can be obtained from the useful inequality

$$1-x \le e^{-x}$$
 for all x

with equality if and only if x = 0,

Thus

$$n_t \le n_0 (1 - \frac{1}{OPT})^t < n_0 (e^{-\frac{1}{OPT}})^t = ne^{-\frac{t}{OPT}}$$

At $t = \ln n \cdot OPT$, therefore, n_t is strictly less than $ne^{-\ln n} = 1$, which means no elements remain to be covered.