Advanced Algorithms/Analysis

Greedy algorithm
(Read Ch. 5 from reference book)
CS 721
Fal 2019

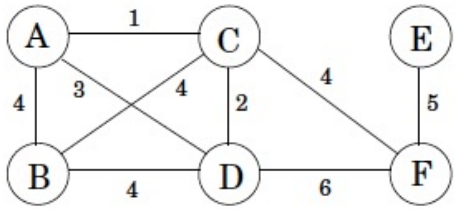
September 16, 2019

Greedy algorithms

- Greedy algorithms build up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.
- Although such an approach can be disastrous for some computational tasks, there are many for which it is optimal.
- We will study three examples of greedy algorithms
  - Minimum spanning tree
  - Huffman code
  - Set cover

Motivation for Minimum Spanning Tree (MST)

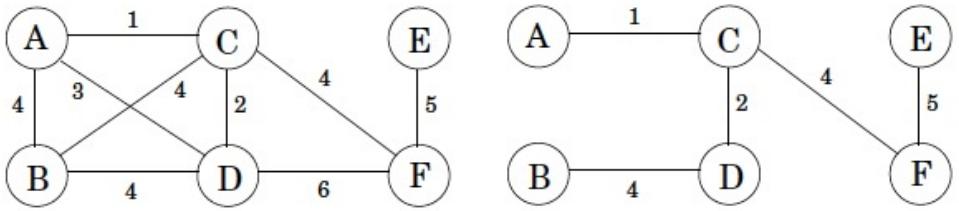
- Suppose you are asked to network a collection of computers by linking selected pairs of them.
- This translates into a graph problem in which nodes are computers, undirected edges are potential links, and the goal is to pick enough of these edges that the nodes are connected.
- But this is not all; each link also has a maintenance cost, reflected in that edge's weight.



- What is the cheapest possible network?

Motivation for Minimum Spanning Tree (MST)

- One immediate observation is that the optimal set of edges cannot contain a cycle, because removing an edge from this cycle would reduce the cost without compromising connectivity.
- **Property 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.
- The particular tree we want is the one with minimum total weight, known as the minimum spanning tree.



- In the example above mst has cost 16.

MST problem

- Formal definition of MST problem is as follows:
  - **Input:** An undirected graph $G = (V, E)$ with edge weights w_e for any edge $e \in E$.
  - **Output:** A tree $T = (V, E')$ with $E' \subseteq E$ that minimizes $weight(T) = \sum_{e \in E'} w_e$.
- We will use different greedy strategies to solve this problem, giving rise to two different algorithms:
  - Kruskal's algorithm
  - Prim's algorithm

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

- In other words, it constructs the tree edge by edge and, apart from taking care to avoid cycles,
- Simply picks whichever edge is cheapest at the moment.
- This is a greedy algorithm: every decision it makes is the one with the most obvious immediate advantage.

A greedy approach contd

- Figure 5.1 shows an example of this greedy approach.
- We start with an empty graph and then attempt to add edges in increasing order of weight (ties are broken arbitrarily):

  B-C; C-D; B-D; C-F; D-F; E-F; A-D; A-B; C-E; A-C
- The first two succeed, but the third, B-D, would produce a cycle if added, so we ignore it and move along.
- The final result is a tree with cost 14, the minimum possible.

Figure 5.1 The minimum spanning tree found by Kruskal's algorithm.

- The correctness of Kruskal's method follows from a certain cut property, which is general enough to also justify a whole slew of other minimum spanning tree algorithms.

Tree

- A tree is an undirected graph that is connected and acyclic. Much of what makes trees so useful is the simplicity of their structure, for example it satisfies the following three properties:
- **Property 2:** A tree on n nodes has $n-1$ edges.
- **Property 3:** Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.
- **Property 4:** An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

Cut property

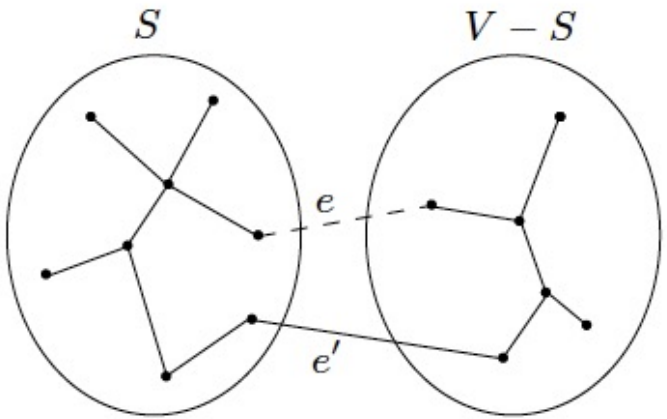
- **Cut property:** Suppose edges X are part of a minimum spanning tree of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V \setminus S$, and let e be the lightest (minimum weight) edge across this partition. Then $X \cup \{e\}$ is part of some MST.
- A cut is any partition of the vertices into two groups, S and $V \setminus S$.
- What this property says is that it is always safe to add the lightest edge across any cut (that is, between a vertex in S and one in $V \setminus S$), provided X has no edges across the cut.
- Why does cut property hold in general?

Cut property contd.

- 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 this edge, we are left with $T' = T \cup \{e\} \setminus \{e'\}$, which we will show to be a tree. How?
  - T' is connected by Property 1, since e' is a cycle edge. And it has the same number of edges as T; so by Properties 2 and 3, it is also a tree.

Cut property contd.

Figure 5.2 $T \cup \{e\}$. The addition of e (dotted) to T (solid lines) produces a cycle. This cycle must contain at least one other edge, shown here as e', across the cut $(S, V - S)$.



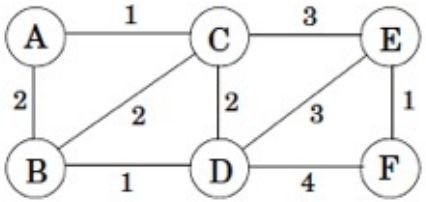
Cut property contd.

- Next, we will show that T' is also an MST
- First note that $weight(T') = weight(T) + w(e) - w(e')$
- Both e and e' cross between S and $V \setminus S$, and e is specifically the lightest edge of this type.
- Therefore $w(e) \leq w(e')$, and $weight(T') \leq weight(T)$.
- Since T is an MST, it must be the case that $weight(T') = weight(T)$ and that T' is also an MST.

Cut property example

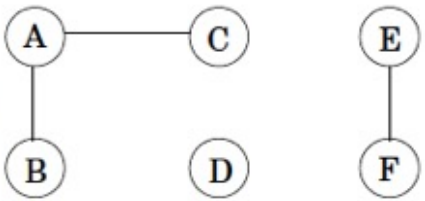
Figure 5.3 The cut property at work. (a) An undirected graph. (b) Set X has three edges, and is part of the MST T on the right. (c) If $S = \{A, B, C, D\}$, then one of the minimum-weight edges across the cut $(S, V - S)$ is $e = \{D, E\}$. $X \cup \{e\}$ is part of MST T', shown on the right.

(a)

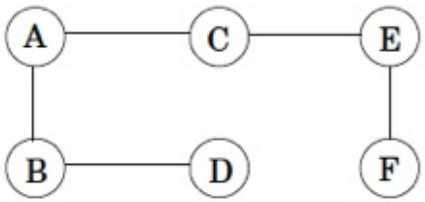


(b)

Edges X:

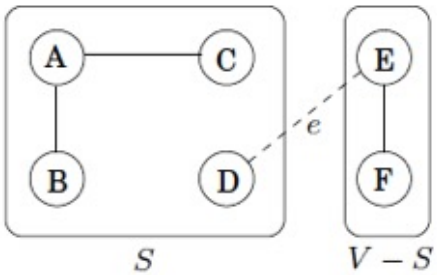


MST T:

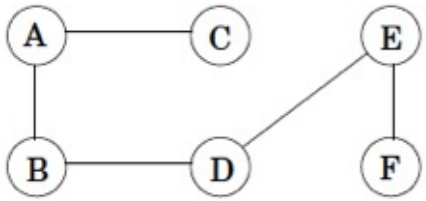


(c)

The cut:



MST T':



Kruskal's algorithm

- Kruskal's algorithm works as follows:
  - At any given moment, the edges it has already chosen form a partial solution, a collection of connected components each of which has a tree structure.
  - The next edge e to be added connects two of these components; call them T_1 and T_2. Since e is the lightest edge that doesn't produce a cycle, it is certain to be the lightest edge between T_1 and $V \setminus T_1$ and therefore satisfies the cut property.
- Now we fill in some implementation details.
  - At each stage, the algorithm chooses an edge to add to its current partial solution.
  - To do so, it needs to test each candidate edge (u, v) to see whether the endpoints u and v lie in different components; otherwise the edge produces a cycle.
  - And once an edge is chosen, the corresponding components need to be merged.
  - What kind of data structure supports such operations?

Kruskal's algorithm contd.

- We will model the algorithm's state as a collection of disjoint sets, each of which contains the nodes of a particular component.
- Initially each node is in a component by itself:
  - **makeset(x):** create a singleton set containing just x.
- We repeatedly test pairs of nodes to see if they belong to the same set.
  - **find(x):** to which set does x belong?
- And whenever we add an edge, we are merging two components.
  - **union(x; y):** merge the sets containing x and y.

Kruskal's algorithm contd.

Figure 5.4 Kruskal's minimum spanning tree algorithm.

```
procedure kruskal(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the edges X

for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

- Kruskal's algorithm uses $|V|$ *makeset*, $2|E|$ *find*, and $|V| - 1$ *union* operations.

A data structure for disjoint sets

- One way to store a set is as a directed tree (Figure 5.5).
- 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.
- We will consider three operations:
  - *makeset*
  - *find*
  - *union*

Makeset and Find and Union

```
procedure makeset(x)
π(x) = x
rank(x) = 0
```

- *makeset* is a constant time operation

```
function find(x)
while x ≠ π(x): x = π(x)
return x
```

- *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.

Makeset and Find and Union contd.

```
procedure union(x, y)
r_x = find(x)
r_y = find(y)
if r_x = r_y:  return
if rank(r_x) > rank(r_y):
    π(r_y) = r_x
else:
    π(r_x) = r_y
    if rank(r_x) = rank(r_y):  rank(r_y) = rank(r_y) + 1
```

- Merging two sets is done by *union* operation: make the root of one point to the root of the other.
- Since tree height is the main impediment to computational efficiency, a good strategy is to make the root of the shorter tree point to the root of the taller tree.
- This way, the overall height increases only if the two trees being merged are equally tall.
- Instead of explicitly computing heights of trees, we will use the rank numbers of their root nodes-which is why this scheme is called union by rank.

Properties of union by rank scheme

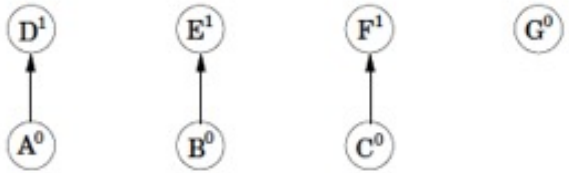
- By design, the rank of a node is exactly the height of the subtree rooted at that node. This means, for instance, that as you move up a path toward a root node, the rank values along the way are strictly increasing.
- **Property 1:** For any x (other than the root node) $rank(x) < rank(\pi(x))$.
- **Property 2:** Any root node of rank k has at least 2^k nodes in its tree.
- **Property 3:** If there are n elements overall, there can be at most $n/2^k$ nodes of rank k.
- This last observation implies, crucially, that the maximum rank is $\log n$.
- Therefore, all the trees have height $\leq \log n$, and this is an upper bound on the running time of *find* and *union* operation.

A sequence of disjoint set data operations example

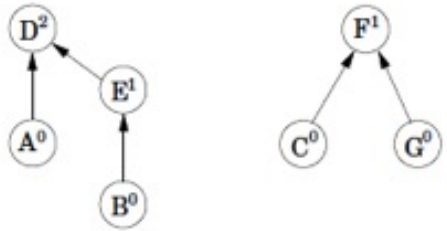
Figure 5.6 A sequence of disjoint-set operations. Superscripts denote rank.

After makeset(A), makeset(B), ..., makeset(G):

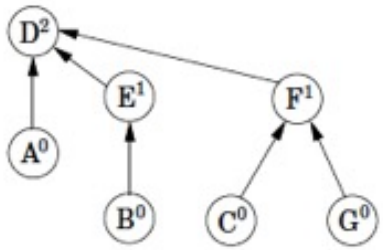
After union(A, D), union(B, E), union(C, F):



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



After union(B, G):



Running time and path compression

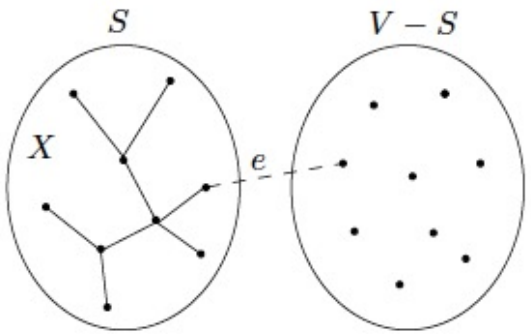
- Sorting $|E|$ edges take $O(|E| \log |V|)$ time.
- There are total $|V|$ *makeset*, $2|E|$ *find*, and $|V| - 1$ *union* operations. They take total $O(|E| \log |V|)$ time.
- Thus running time of Kruskal's algorithm is $O(|E| \log |V|)$.
- But how about all the edges are already sorted or the edge weights are small (say $O(|E|)$), so that sorting can be done in linear time using, say counting sort?
  - In that case data structure operations becomes the main bottleneck.
  - Can we possibly do better by modifying the data structure?
  - The answer is yes by doing something called "path compression", but runtime analysis will require "Amortized analysis" technique and we will revisit this again when we study amortized analysis.

Prim's algorithm

- 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, namely, the lightest edge between a vertex in S and a vertex outside S

```
X = { } (edges picked so far)
repeat until |X| = |V| − 1:
  pick a set S ⊂ V for which X has no edges between S and V − S
  let e ∈ E be the minimum-weight edge between S and V − S
  X = X ∪ {e}
```

Figure 5.8 Prim's algorithm: the edges X form a tree, and S consists of its vertices.



Prim's algorithm pseudocode

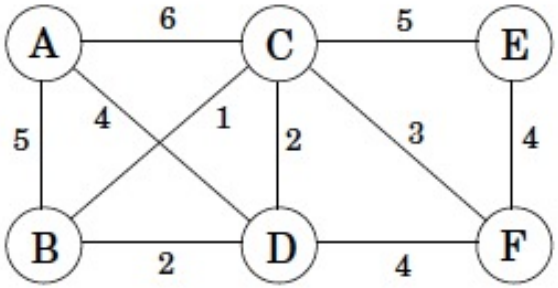
Figure 5.9 *Top:* Prim's minimum spanning tree algorithm. *Below:* An illustration of Prim's algorithm, starting at node A. Also shown are a table of cost/prev values, and the final MST.

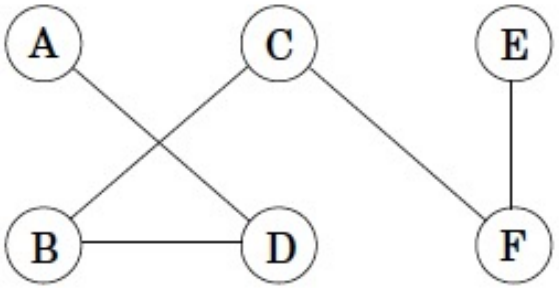
```
procedure prim(G, w)
Input:   A connected undirected graph G = (V, E) with edge weights w_e
Output:  A minimum spanning tree defined by the array prev

for all u ∈ V:
    cost(u) = ∞
    prev(u) = nil
Pick any initial node u_0
cost(u_0) = 0

H = makequeue(V)      (priority queue, using cost-values as keys)
while H is not empty:
    v = deletemin(H)
    for each {v, z} ∈ E:
        if cost(z) > w(v, z):
            cost(z) = w(v, z)
            prev(z) = v
            decreasekey(H, z)
```

An illustration of Prim's algorithm





Set S	A	B	C	D	E	F
$\{\}$	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$5/A$	$6/A$	$4/A$	∞/nil	∞/nil
A, D		$2/D$	$2/D$		∞/nil	$4/D$
A, D, B			$1/B$		∞/nil	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

Relation to Dijkstra's algorithm and running time

- While explaining the basic idea of Prim's algorithm, we can equivalently think of S as growing to include the vertex $v \notin S$ of smallest cost, that is $\boxed{cost(v) = \min_{u \in S} w(u, v)}$
- This is strongly reminiscent of Dijkstra's algorithm, and in fact the pseudocode is almost identical.
- The only difference is in the key values by which the priority queue is ordered.
  - In Prim's algorithm, the value of a node is the weight of the lightest incoming edge from set S.
  - Whereas in Dijkstra's it is the length of an entire path to that node from the starting point.
- Nonetheless, the two algorithms are similar enough that they have the same running time, which depends on the particular priority queue implementation.

Relation to Dijkstra's algorithm and running time contd.

- Therefore, using binary heap implementation of priority queue, running time of Prim's algorithm is $O(|V| + |E|) \log |V|)$ or equivalently $O(|E| \log |V|)$.
- In this sense, Dijkstra's algorithm can also be thought of as a greedy algorithm.

Hoffman encoding: a practical example

In the MP3 audio compression scheme, a sound signal is encoded in three steps.

1. It is digitized by sampling at regular intervals, yielding a sequence of real numbers $s_1, s_2, \ldots, s_T$. For instance, at a rate of 44,100 samples per second, a 50-minute symphony would correspond to $T = 50 \times 60 \times 44,100 \approx 130$ million measurements.
2. Each real-valued sample s_t is quantized: approximated by a nearby number from a finite set Γ. This set is carefully chosen to exploit human perceptual limitations, with the intention that the approximating sequence is indistinguishable from $s_1, s_2, \ldots, s_T$ by the human ear.
3. The resulting string of length T over alphabet Γ is encoded in binary.

It is in the last step that Huffman encoding is used.

Hoffman encoding: an example first

To understand its role, let's look at a toy example in which T is 130 million and the alphabet Γ consists of just four values, denoted by the symbols A;B;C;D.

- What is the most economical way to write this long string in binary?
- The obvious choice is to use 2 bits per symbol, say codeword 00 for A, 01 for B, 10 for C, and 11 for D, in which case 260 megabits are needed in total.
- Can there possibly be a better encoding than this?

Variable length encoding

Suppose each symbol has different frequency:

Symbol	Frequency
A	70 million
B	3 million
C	20 million
D	37 million

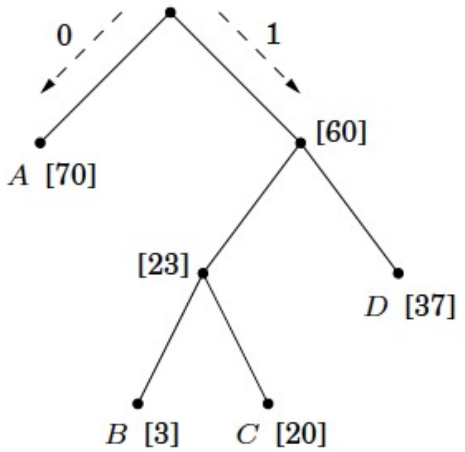
- Is there some sort of variable-length encoding, in which just one bit is used for the frequently occurring symbol A, possibly at the expense of needing three or more bits for less common symbols?
- A danger with having codewords of different lengths is that the resulting encoding may not be uniquely decipherable. For instance, if the codewords are $\{0; 01; 11; 001\}$, the decoding of strings like 001 is ambiguous. We will avoid this problem by insisting on the prefix-free property: no codeword can be a prefix of another codeword.

Prefix-free property

- Any prefix-free encoding can be represented by a full binary tree in which every node has either zero or two children - where the symbols are at the leaves, and where each codeword is generated by a path from root to leaf, interpreting left as 0 and right as 1
- Figure 5.10 shows an example of such an encoding for the four symbols A;B;C;D.

Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets.

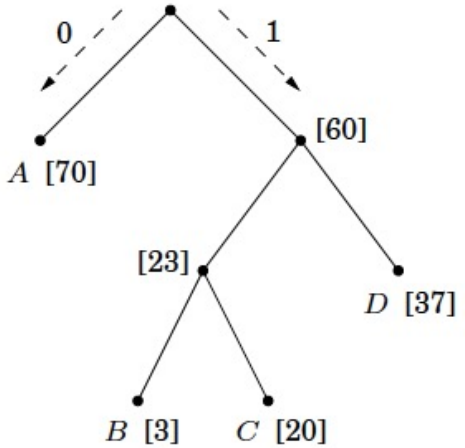
Symbol	Codeword
A	0
B	100
C	101
D	11



Prefix-free property contd.

Figure 5.10 A prefix-free encoding. Frequencies are shown in square brackets.

Symbol	Codeword
A	0
B	100
C	101
D	11



- Decoding is unique: a string of bits is decrypted by starting at the root, reading the string from left to right to move downward, and, whenever a leaf is reached, outputting the corresponding symbol and returning to the root. I
- Using encoding scheme pf Figure 5.10 the total size of the binary string drops to 213 megabits, a 17% improvement.

Prefix-free property contd.

- In general, how do we find the optimal coding tree, given the frequencies $f_1, f_2, \ldots, f_n$ of n symbols?
- To make the problem precise, we want a tree whose leaves each correspond to a symbol and which minimizes the overall length of the encoding:
  cost of tree $= \sum_{i=1}^{n} f_i \cdot$ (depth of i^{th} symbol in tree)
- An alternative formulation is, The cost of tree is the sum of the frequencies of all leaves and interal nodes, except the root.

Greedy algorithm for Hoffmann encoding

- The first formulation of the cost function tells us that the two symbols with the smallest frequencies must be at the bottom of the optimal tree, as children of the lowest internal node (this internal node has two children since the tree is full).
- Otherwise, swapping these two symbols with whatever is lowest in the tree would improve the encoding (this is infact the proof of optamility).
- This suggests that we start constructing the tree greedily:
  - Find the two symbols with the smallest frequencies, say i and j, and make them children of a new node, which then has frequency $f_i + f_j$.
  - To keep the notation simple, let's just assume these are f_1 and f_2.
  - By the second formulation of the cost function, any tree in which f_1 and f_2 are sibling-leaves has cost $f_1 + f_2$ plus the cost for a tree with $n - 1$ leaves of frequencies $(f_1 + f_2), f3, f4, \ldots, f_n$.

Greedy algorithm for Hoffmann encoding

- The latter problem is just a smaller version of the one we started with.
- So we pull f_1 and f_2 off the list of frequencies, insert (f1 + f2), and repeat until the list is empty.
- The resulting algorithm is shown below

```
procedure Huffman(f)
Input:  An array f[1···n] of frequencies
Output:  An encoding tree with n leaves

let H be a priority queue of integers, ordered by f
for i = 1 to n:  insert(H, i)
for k = n+1 to 2n−1:
  i = deletemin(H),  j = deletemin(H)
  create a node numbered k with children i, j
  f[k] = f[i] + f[j]
  insert(H, k)
```

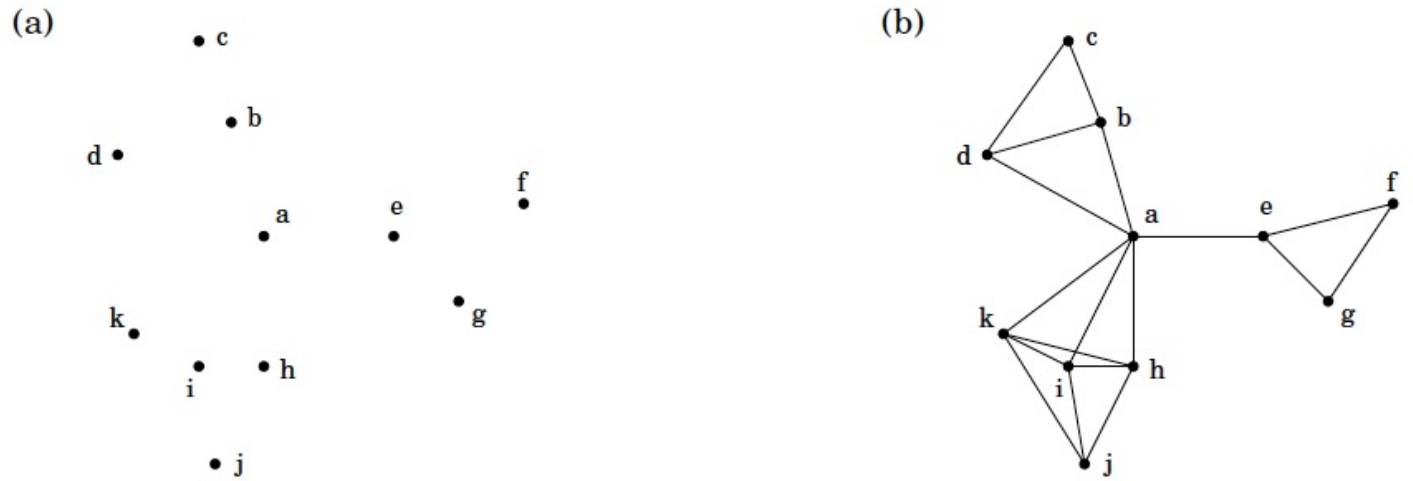
- Using binary heap implementation ofpriority queue the running of this algorithm is $O(n \log n)$.

Greedy algorithm: caveat

- Not all greedy algorithms are optimal.
- So far we have seen two greedy algorithms which are optimal.
- Next we will see an example of greedy algorithm which is not optimal (but close to optimal)

Set cover problem: a practical example

Figure 5.11 (a) Eleven towns. (b) Towns that are within 30 miles of each other.



- The dots in Figure 5.11 represent a collection of towns.
- This 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.
- What is the minimum number of schools needed?

Set cover problem:

This is a typical 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.
- The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

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.

- In our example, the elements of B are the towns.

Greedy algorithm for set cover problem:

- This problem lends itself immediately to a greedy solution:

  Repeat until all elements of B are covered:

  Pick the set S_i with the largest number of uncovered elements.

- This is extremely natural and intuitive.
- Let's see what it would do on our earlier example:
  - It would first place a school at town a, since this covers the largest number of other towns.
  - Thereafter, it would choose three more schools c, j, and either f or g for a total of four.
  - However, there exists a solution with just three schools, at b, e, and i.
  - The greedy scheme is not optimal!

But luckily, it isn't too far from optimal.

Greedy algorithm for set cover problem

Claim: Suppose B contains n elements and that the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \ln n$ 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 k sets, there must be some set with at least n_t/k of them.
- Therefore, the greedy strategy will ensure that: $n_{t+1} \leq n_t - \frac{n_t}{k} = n_t(1 - 1/k)$.
- Suppose greedy algorithm runs for T iterations (so it will output T sets). Applying this rule repeteadly we get $n_T \leq n_0(1 - 1/k)^T$.

Greedy algorithm for set cover problem contd.

- Applying the inequality $1 - x \leq e^{-x}$ which works for all real values of x, we can write
  $n_T \leq n_0(1 - 1/k)^T \leq n_0(e^{-1/k})^T = n_0 e^{-T/k} = ne^{-T/k}$
- Setting $ne^{-T/k} = 1$, we get $T = k \ln n$. □
- The ratio between the greedy algorithm's solution and the optimal solution varies from input to input but is always less than lnn.
- We call this maximum ratio the approximation factor of the greedy algorithm.
- Later we will show that "Set cover" is an NP-complete problem meaning that it does not have any effcient (polynomial time) solution, however, using greedy algorithm we can get an $\ln n$ approximate solution.