1 Lecture Notes, Fundamental Algorithms: Greedy Algorithms

The greedy approach to algorithm design is to build a solution via a series of simple decisions which are never revisited. This often leads to simple, intuitive algorithms. The downside is that there may be no guarantee of optimality.

However, the algorithms for two famous problems, Huffman Coding and Minimum Spanning Trees, do produce optimal results. For other problems, such as set cover, we do not achieve optimality.

1.1 Set Cover

We begin with a motivating example: Where should Amazon locate its warehouses so that each town is within 2 hours of a warehouse, while minimizing the number of warehouses? (In reality, time is not the only issue.)

Each possible warehouse covers (is within 2 hours) of some subset of towns. So the task amounts to identifying a minimum sized set of warehouses that cover all the towns.

More abstractly, we define set cover as follows.
Input: A universe $U = \{e_1, e_2, \ldots, e_n\}$ of n items, and sets $S_1, S_2, \ldots, S_m \subset U$.
Output: A small number of sets, ideally the minimum number, $S_{i_1}, S_{i_2}, \ldots, S_{i_r}$, such that $\cup_{j=1}^{r} S_{i_j} = U$. We say these sets cover U.

Here is a greedy algorithm for this problem: Repeatedly choose a set that includes the largest number of uncovered items.

Lemma 1. *If the optimum solution has size k, then the greedy algorithm finds a solution using at most $n \ln k$ sets.*

Proof. We focus on one optimal solution, $S_{o_1}, S_{o_2}, \ldots, S_{o_k}$. The largest set in this optimal solution, S_{o_1} say, has size at least n/k. Hence the first set chosen by the greedy algorithm has size at least n/k (since one of the sets it considers is S_{o_1}).

Now the greedy solution has some $n_2 \leq n(1 - 1/k)$ items to cover. One set in the optimal solution contains at least n_2/k of these items, and hence so does the solution chosen by the greedy algorithm.

Now, at most $n_2(1 - 1/k) \leq n(1 - 1/k)^2$ items are uncovered.

Iterating, after the greedy algorithm chooses i sets, at most $n(1-1/k)^i$ items remain uncovered.

Recall that $(1 - 1/k)^k < 1/e$, and therefore $(1 - 1/k)^{k \ln n} < 1/n$. It follows that after $k \ln n$ iterations, fewer than $n/n = 1$ items are uncovered, i.e. all the items are covered. □

One can show that up to a constant factor, this is the best possible bound on the performance of the greedy algorithm.

1.2 Minimum Spanning Tree (MST) Algorithms

Input: A weighted, undirected, connected graph $G = (V, E)$. The weights are real values, but each weight can be positive, negative, or zero.
Output: A minimum weight spanning tree of G. A spanning tree is a tree that connects all the edges of G using edges in E. The weight of a spanning tree is the sum of the weights of the edges it contains.

The key result guiding all minimum spanning tree algorithms is the following cut lemma.

Lemma 2. *Let $G = (V, E)$ be an undirected graph and let $F \subset E$ be a cycle-free set of edges. Let V_a and V_b be a partition of V, meaning that $V_a \cup V_b = V$ and $V_a \cap V_b = \phi$, with the property that all the edges in F either have both endpoints in V_a or both endpoints in V_b; formally: $F \subset V_a \times V_a \cup V_b \times V_b$.*

Suppose that that there is a minimum spanning tree T that contains all the edges in F.

Let e be a minimum weight edge between V_a and V_b. Then there is a minimum spanning tree that contains all the edges in $F \cup \{e\}$.

Proof. Suppose e is not an edge in T (otherwise we are done). Then, when added to T, $e = (u, v)$ would form a cycle, namely the path P in T from u to v plus the edge (u, v). There must be at least one edge e' on P from V_a to V_b, for otherwise T could not connect all the vertices in V. Note that $e' \notin F$. So $F \subset T \setminus \{e'\}$.

By assumption, $\text{wt}(e) \leq \text{wt}(e')$. $(T \setminus \{e'\}) \cup \{e\}$ is also a spanning tree of G, and necessarily $\text{wt}(T') \leq \text{wt}(T)$. So T' must also be an MST which contains both F and e. □

This leads to two distinct algorithms. Prim's algorithm builds a single tree, always adding a least weight edge with one endpoint in the tree. As we will see, it's structure closely resembles that of Dijkstra's algorithm, and as a consequence it has the same runtime performance. Kruskal's algorithm builds a forest starting from n 1-node trees; it goes through the edges in increasing weight order, adding an edge if it joins two distinct trees, which are thereby combined into a single tree. As it turns out, in the worst case, the dominant cost for Kruskal's algorithm is the cost for sorting the edges. If their weights are integers in a small range, $[1, n^c - 1]$ for some constant c, then the sorting can be done by radix sort and the runtime reduces to almost linear time.

Prim's Algorithm The algorithm builds a tree T rooted at an arbitrary starting vertex s. After i phases it will have added i edges and vertices to T; for every other vertex v, it will maintain the cost of the least weight edge from T to v; this value is ∞ if there is no such edge. Among these edges, it adds the lightest one to T, an edge (u, w) say, with $u \in T$ and $w \notin T$. Once w is added to T, the algorithm updates the least weight edge to the neighbors x of w, completing the phase. By assumption, G is connected, so the algorithm ends only when a spanning tree of all of G has been built.

Pseudo-code follows. The algorithm uses a priority queue Q and two arrays Dist$[1 : n]$ and Pred$[1 : n]$. For each vertex v, Pred$[v]$ records the other endpoint of the least weight edge joining v to the current T, or if v is already in T, the other endpoint of the edge which joined it to T; Dist$[v]$ records the weight of this edge.

```
procedure PRIM(G, s)
    for v = 1 to n do Dist[v] ← ∞; Pred[v] ← nil
    end for
    Dist[s] ← 0;
    T ← φ;
    EnQueue(Q, V, Dist);
    while Q not empty do
        u ← DeleteMin(Q); T ← T ∪ {u}
        for each edge (u, v) with v ∉ T do
            if wt(u, v) < Dist[v] then
                Dist[v] ← wt(u, v); Pred[v] ← u;
                ReduceKey(Q, v, Dist[v])
            end if
        end for
```

end while
end procedure

The minimum spanning tree consists of all the edges $(v, \text{Pred}[v])$.

It should be clear, that just like Dijkstra's algorithm, the main cost will be the n DeleteMins and up to m ReduceKey operations. So the running time is $O(m + n \log n)$ if we use a Relaxed Heap, and $O((m + n) \log n)$ if we use a binary heap.

Correctness can be argued as follows. The inductive hypothesis is that at all times, T is contained in some MST of G. Clearly, it is true initially, with $T = \phi$. For the inductive step, consider the cut with the current T on one side, and the remaining vertices on the other side. One of the minimum weight edges crossing this cut is the edge identified by the DeleteMin, and therefore if this edge is added to T, the resulting tree is still contained in an MST.

1.3 Kruskal's Algorithm

This is a two step procedure.

procedure KRUSKAL(G)
  Step 1: Sort the edges in increasing order by weight
  Step 2:
  Create a forest F of n 1-node trees
  for each edge $e = (u, v)$ in turn, in increasing weight order **do**
    if u and v are in two different trees in F **then**
      add (u, v) to F, joining these two trees into one tree
    end if
  end for
end procedure

The main issue in implementing this algorithm is how to test whether u and v are in distinct trees.

To this end, we maintain each tree as a set of vertices. Then, to process an edge (u, v), we identify the sets/trees T_u and T_v to which its endpoints belong. If they are the same set, we discard the edge; otherwise, we combine the two sets (and add the edge to the forest we are building).

Thus we need a data structure that supports the following operations on a set of n items held in $k \leq n$ disjoint sets.

- Find(u): returns the set to which u belongs.
- Union(S, T): forms a new set $S \cup T$ and discards sets S and T.

Initially, all the sets comprise a single item.

Our algorithm will perform $n - 1$ Union operations and up to $2m$ Finds (2 Finds for each edge). Besides the cost of these operations, the main cost is for the sorting in Step 1.

There is a very simple data structure called the Union-Find data structure which supports these operations.

We will name each set using an arbitrary but fixed item in the set. We maintain each set as a tree with parent pointers; each node stores a single item, and the item at the root is used to name the set.

We assume we have pointers from an array of items to their occurrence in the data structure. So to carry out a Find(u), we access the node holding item u, and follow parent pointers to the root of the tree, returning the item at the root.

To do a union of sets S and T, we make the root of one of the trees the parent of the other root. We want the "larger" tree to provide the root. To this end, we maintain ranks. Initially, the roots of the one-item trees have rank 0. When we combine two trees, if they have different ranks, the higher rank tree becomes the root of the combination. It they are equal, either tree provides the root, and its rank is increased by 1. Clearly, performing a union takes $O(1)$ time.

The next lemma shows that a tree with a rank r root is large.

Lemma 3. *A tree with a rank r root holds at least 2^r items.*

Proof. This follows easily by induction on r. For the base case $r = 0$, we have $2^r = 1$, and this case corresponds to the one-node trees. So the claim holds in this case.

For the inductive step, we see that a tree with root of rank $r + 1$ is formed by combining two rank r trees, each of which has at least 2^r nodes by the inductive hypothesis, and therefore the combined tree has at least 2^{r+1} nodes. □

Thus $r \leq \log n$. And therefore every find runs in $O(\log n)$ time.

Theorem 1. *Kruskal's algorithm runs in $O(m \log n + n)$ time.*

Proof. The sort runs in $O(m \log m) = O(m \log n)$ time, and each Find takes $O(\log n)$ time. There are at most 2 finds per edge, for a total cost of $O(m \log n)$. Finally, the $n - 1$ unions take $O(n)$ time in total. □

However, we can sharply improve the Union-Find runtime with the following modification of the algorithm: A Find(u) operation traverses a path P from u to the root s of its tree. We simply reset the parent pointer for every node on P to point to s, so if any of these items are accessed in the future, it takes $O(1)$ time to reach s, rather than the length of the path P. This does not yield an $O(1)$ bound on each subsequent Find(u), as due to further unions s may cease to be the root of u's tree. Still it does result in very good performance.

We will state a bound, but not prove it here. (See the challenge problem on the homework for details).

We define the function Tower(k) as follows.

$$\text{Tower}(k) = \begin{cases} 0 & k = 0 \\ 2^{\text{Tower}(k-1)} & k \geq 1 \end{cases}$$

We define $\log^* n$ to be the least k such that $\text{Tower}(k) \geq n$. We note the following values for the Tower function. $\text{Tower}(0) = 1, \text{Tower}(1) = 2, \text{Tower}(2) = 4, \text{Tower}(3) = 16, \text{Tower}(4) = 2^{16} = 65,536, \text{Tower}(5) = 2^{65,536}$. So in practice, $\log^* n \leq 5$. The $\log^* n$ function is often defined as how many times one needs to take a log in order to reduce n to at most 1 (for example, if $n = 2^{65,536}$, the fifth iterate has $\log \log \log \log \log n = 1$, so $\log^* 2^{65,536} = 5$).

The runtime for n Unions and $2m$ Finds is bounded by $O((n + m) \log^* n)$. In fact, even this bound is an overestimate asymptotically speaking.

1.4 Huffman Codes

Huffman codes are used to encode characters that occur with different frequencies; to minimize the average length of an encoding, we will want shorter codes for the more frequent characters.

Suppose we use a binary encoding. Each character c needs to be represented using a unique binary string, called code(c). In addition, to avoid ambiguity, we cannot allow code(c) to be a

prefix of code(d) for any pair of characters c and d (a prefix is just an initial portion of a string). An example of a set of codes with this property is: 00,01,100,101,11.

Suppose we are given n characters $c_1, c_2, \ldots, c_n$ and they occur with frequencies $f_1 \geq f_2 \geq \ldots \geq f_n$. The task is to devise an encoding that minimizes the average length of an encoded string of characters.

We represent the encoding scheme using an n-leaf binary tree, with edges to left children labeled by 0 and to right children by 1. The leaf nodes are labeled by the n characters. The encoding of a character is simply the sequence of binary labels on the path from the root to the leaf holding the character.

The average cost of the encoding is given by $\frac{1}{n}\sum_{i=1}^{n} f_i \cdot$ (length of path to c_i).

Clearly, two deepest leaves should be occupied by the least frequent characters, c_{n-1} and c_n. Furthermore, they will be siblings. The parent of these two nodes is accessed with frequency $f_{n-1} + f_n$. So if we deleted these two leaves, we would have created a minimum cost tree for characters with frequencies $f_1, f_2, \ldots, f_{n-2}, f_{n-1}+f_n$. This leads to the following greedy algorithm.

The algorithm is as follows. Take the two characters c_{n-1} and c_n with the smallest frequencies and replace them with a meta-character c'_{n-1} with frequency $f_{n-1} + f_n$. Now create a minimum cost encoding tree for the characters $c_1, c_2, \ldots, c_{n-2}, c'_{n-1}$ (they may not be in frequency order any more). Finally, make c_{n-1} and c_n the children of the leaf holding c'_{n-1}.

We can keep the characters in a MinHeap using their frequencies as the keys. We will need to do $2(n-1)$ DeleteMins in total (2 DeleteMins for each internal node that we create) and $2n-1$ Inserts. This takes a total of $O(n \log n)$ time.