CSE 101 Class Notes

Greedy Algorithms: Kruskal's Minimum Spanning Tree

November 13, 2006

Problem Description

Instance: An undirected graph $G = (V, E)$, $|V| = n$, $|E| = m$ with edge weights $w(e) \geq 0$.

Solution: A subset $T \in E$.

Constraints: All vertices $v \in V$ connected by some edge in T.

Objective: Minimize the total weight $w(T) = \sum_{e \in E} w(e)$.

For example, a minimum spanning tree might define the fewest roads necessary to connect a set of cities, or the shortest wiring connecting elements on a chip. Note that, being a tree, the solution will always have $n - 1$ edges.

Greedy Algorithm (Kruskal)

Our strategy is to build a forest $F \in E$, starting with the empty forest and finishing with an MST. Considering the edges e in order of increasing weight, we add e to F only if it does not create a cycle.

Proving the algorithm's correctness is complicated by the fact that the solutions do not have the self-similarity property we usually exploit – at each point, a choice is made in the context of an existing forest F. We can regain self-similarity by redefining the problem. Instead of finding the MST of a graph, the problem is to extend a (possibly empty) forest $F \in E$ to an MST.

Since this kind of redefinition can be generalized to other stateful greedy algorithms, we state it as a lemma:

> **Lemma (stateful "modify-the-solution"):** Let P be a partial greedy solution. Let d be the next decision point in the greedy algorithm, and o the algorithm's decision. Let S' be a complete solution extending P that does not choose o. Then $\exists S$, another complete solution that extends P, chooses o for d, and is at least as good as S'.

In other words, we show that at any point, following the greedy strategy from that point onward is optimal.

Applying this to the MST problem, we have the following:

Lemma: Let F be a forest, and let e be the smallest-weight edge so that $F \cup \{e\}$ has no cycles. Let T' be a spanning tree so that $F \subset T'$ and $e \notin T'$. We must show that there exists a greedy solution such that $F \cup \{e\} \subset T$ and $w(T) \leq w(T')$.

Proof: Let $e = (u, v)$ be the greedy choice given forest F, and let T' be a non-greedy spanning tree extending F. We must find a greedy solution T that (1) is a tree, (2) has a smaller weight, and (3) spans G.

First, $T' \cup \{e\}$ creates a cycle C, since u and v are connected in T'. Since $F \cup \{e\}$ has no cycles, there is an edge $e' \in C - F$ contained in T'. Let $T = T' \cup \{e\} - \{e'\}$. Since $F \cup \{e'\} \subset T'$, and since we removed e' when adding e, T has no cycles, and is therefore a tree.

Second, since $F \cup \{e'\} \in T'$, e' does not create a cycle in F. However, since e is the greedy choice given F, $w(e) \leq w(e')$. Therefore $w(T) = w(T') + w(e) - w(e') \leq w(T')$.

Third, let $p' = x \rightsquigarrow u' \rightarrow_{e'} v' \rightsquigarrow y$ be a path from x to y in T' using $e' = (u', v')$. We must show that there exists a path p in T from x to y using $e = (u, v)$. (Note that if $e' \notin p$, we are done, since we can just use T''s path in T.) Since T' is a tree, removing e' must separate the graph, and since $T = T' - \{e'\} \cup \{e\}$ is also a tree, $\{e\}$ must connect these two components – if it did not, it would create a cycle in one of them. Therefore, since the two components are connected, there exists a path $x \rightsquigarrow u \rightarrow v \rightsquigarrow y$ or $y \rightsquigarrow u \rightarrow v \rightsquigarrow x$ in T.

Lemma: Let G be a connected graph, and F_t be the forest of size t that we create after t iterations of Kruskal's algorithm. Then there is an MST T of G such that $F_t \in T$.

Proof: (by induction on t). Base case: $F_0 = \phi$, so any MST contains F_0.

Inductive case: Assume there is an MST $T' \supset F_t$. Let $F_{t+1} = F_t \cup \{e\}$. If $e \in T'$, we're done. Otherwise, we can apply the MST lemma above to get a spanning tree T with $F_t \cup e \in T$ and $w(T) \leq w(T')$. Therefore T is an MST and $F_{t+1} = F_t \cup \{e\} \in T$.

Finally, since $F_i \in T$ and $|T| = n - 1$, $F_{n-1} = T$ is an MST, so our algorithm works.

Implementation

Here is a direct implementation of the above algorithm:

```
MST(G = (V, E))
    T <- NIL
    for i = 1 .. n - 1
        wmin <- Infiniti
        min <- NIL
        for e = (u,v) in E
            DFS(T, u)
```

```
8                 if visited(v)
9                     E <- E - e
10                else if wmin > w(e)
11                    wmin <- w(e)
12                    min <- e
13            T <- T + min
14            E <- E - e
15        return T
```

This is really bad – the loop at line 3 executes n times, that on line 6 $O(m)$ times, and `DFS` takes time $O(n + m)$, so the entire algorithm is $O(nm^2)$, which is $O(n^5)$ on a complete graph (*nothing* is this slow).

However, we can improve this basic approach through datastructures, preprocessing, and restructuring to yield an efficient implementation of the algorithm. First, we can preprocess by sorting the edges in order of increasing weight. Second, we can restructure to consider edges in order of increasing weight, so we only have to look at each one once.

```
1     MST(G = (V, E))
2         T <- NIL
3         L <- sort(E)
4         for e = (u,v) in E
5             if not connected(T, u, v)
6                 T <- T + e
7         return T
```

Note that we replaced the depth-first search in the original with a more abstract `find_cycle`. If we do this with depth-first search, this gives an $O(m \log m + mn) = O(mn)$ algorithm, which is better but still not that great. However, here we can do better by applying a clever data structure. If we think of each connected graph component as a set of nodes, we need a data structure that supports the following operations:

- Find whether two nodes a and b are in the same set (line 5).
- Merge two of these sets (line 6).

These operations are supported efficiently by a "union-find" data structure in time $O(n \log^* n)$ for n operations (where $\log^* n$ is the number of times you can take the logarithm of n, e.g. $\log^* 65,536 = \log^* 2^{2^{2^2}} = 4$). Therefore the algorithm's final running time is $O(m \log m + m \log^* m) = O(m \log m)$, with the sort being the most expensive operation.

Disjoint Subset Implementations

While the best union-find data structures give the times above, we'll present a couple of simpler data structures for this that still make the time reasonably fast.

Last time we saw that Kruskal's algorithm needs a fast implementation for disjoint subsets. The simplest way to do this would be to represent the set as an array indexed by node-name, whose elements are the representative elements of a subset. To merge two subsets, we then do the following:

```
merge(A[1..n], x, y)
    for i = 1 .. n
        if A[i] = x
            A[i] = y
```

For this approach, `find` is $O(1)$, but `merge` is $O(n)$. Since `MST` requires $n-1$ merges and $2m$ finds, using this data structure, it will take time $O(n^2 + m + m \log m) = O(n^2 + m \log n)$. This is a win over the initial $O(mn)$, with the sorting operation dominating for dense graphs ($m = O(n^2)$).

However, for sparse graphs, the $O(n^2)$ term still dominates. To improve this, we want to make merges cheaper, perhaps at the cost of more expensive finds. A straightforward tactic would be to represent each set as a list, and merge them by adjusting pointers:

```
List[x] <- { linked list of just x for each x }
Leader[x] <- { true for each node x }
merge(x, y)
    merge_lists(List[x], List[y])
    Leader[find(x)] <- false

find(x)
    y <- List[x]
    while not Leader[head(y)]
        y <- tail(y)
    return head(y)
```

But since `find` is now $O(n)$, we are back to $O(nm)$ for `MST`.

We can improve this by relabeling the smaller list in `merge`, and then avoiding the search in `find`. While this makes `merge` $O(n)$ in the worst case, we claim the total cost of k merges will be less than $O(kn)$. To show this, we look at the sum of the sizes of all merged partitions. This equals the number of times each node is relabeled, summed over all nodes (this approach is called *amortized analysis* using the *accounting method*). If a node u is relabeled in a merge then, since it was a member of the smaller partition, after the merge u's set has grown by at least a factor of 2. Since no set can be larger than n nodes, u cannot be merged more than $\log n$ times, so the total cost can be no more than $O(k \log n)$. This is not the fastest implementation, but it is fast enough for `MST`, since the $O(m \log n)$ sort now dominates for both dense and sparse graphs.