Design and Analysis of Algorithms – CSE 101
Greedy Algorithms : Design and Correctness

Search and Optimization Problems The problems solved by greedy algorithms are the same as for backtracking: If possible, find a solution of the following format that satisfies the following constraints; or, find a solution of the following format that satisfies the following constraints and maximizes (minimizes) a certain function. The format should have a large but finite number of possible values, e.g., a small number of possible values in each position of an array means an exponentially large number of total values. It should be easy to verify when a constraint is broken, and to evaluate the objective function, for optimization problems. For example, in the spanning tree problem, the solution format is a subset of the edges. The constraint is that the sub-graph formed by these edges is connected. The objective is to minimize the total weights of elements in the spanning tree.

Basic Design Steps

1. View each solution as making a series of *decisions*. Identify the *options* for each decision. The format for candidate solutions is a big clue as to how to do this. For example, if the format is an array, for each index, we must choose the value of that array element. In the spanning tree example, we can view a solution as deciding the next edge to add $n-1$ times. There are $n-1$ decision points, and for each, the options are the remaining edges that connect two separate components.

2. Pick a single choice, and do a case analysis of the options. What options are possible at each choice? How do the constraints and previous choices limit the options? For example, if we have previously added edges that connect two nodes (indirectly), we will never choose to add an edge between them.

3. Find a conjecture for an "easy" decision, one where the case analysis suggests that one case is better than the other. In the spanning tree example, it always seems better to include the edge of least weight. If we don't include it, we'd have to later include some other edge to connect its endpoints, and that one would cost at least as much.

4. Prove your conjecture works: Method 1. The Modify the Solution Method. Prove the following: MSM Lemma: Let d be the decision the greedy algorithm makes, and $greedy-op$ the option for d that the greedy algorithm chooses. Let $otherop$ be another option for this decision, and let S' be a solution that meets all the constraints and makes the decision $otherop$ for d. Then there is a solution S that makes the decision $greedy-op$ and is at least as good as S'.

   To prove this lemma you need to give: 1. a construction for S from S' and the problem (there might be more than one case). The construction should always have S pick option $greedy-op$. 2. For each case, verify that S does meet the constraints. Use the assumption that S' meets the constraints. You may want to use the defining property of d and $greedy-opt$. 3. For an optimization problem, in each case, verify that $value(S) \geq value(S')$ (for a maximization problem) or $cost(S) \leq cost(S')$ (for a minimization problem). This usually follows from the construction and the defining property of d and $greedy-op$.

   From this lemma, it follows that the greedy algorithm is the same as a back-tracking algorithm that tries the greedy option first. The time spent back-tracking to consider other options was wasted, because from the lemma, the first option beats all the others.

5. Prove your conjecture works: Method 2. The Achieves-the-Bound Method. Identify an obstacle in a problem instance that would prevent an algorithm from doing better. For example, "If there are K events going on at the same time, then we need at least K rooms to hold them in." Here, the obstacle is the K events, at the same time, and the obstacle gives a lower bound of K on the cost of a solution. Prove the following two lemmas: ABM Lemma 1: Let Obs be an obstacle in the problem, that gives bound $Bound(Obs)$. Then for any solution S that meets the constraints, $Cost(S) \geq Bound(Obs)$ (minimization) or $Value(S) \leq Bound(Obs)$ (maximization).

ABM Lemma 2: Let S_{Greedy} be the solution that the greedy algorithm finds. Then from S we can construct an obstacle Obs so that $Bound(Obs) = Cost(S)$ or $Value(S)$.

6. How does making the greedy choice affect the problem? Write a recursive definition of the greedy algorithm as: Find the special decision point d. Make the greedy option for d. Simplify, given this option. Recursively solve the simplified problem, given that option. When all decisions are made, output the solution corresponding to all the greedy options.

   Sometimes, once a choice is made, the problem is similar to the original one, on a smaller instance. Then the recursive calls are to the same problem. If it isn't, you can still use the greedy approach by having a recursive procedure with a partial solution as a parameter. The recursive procedure calls itself with more and more of the solution filled in.

Improving and analyzing greedy algorithms Then think of ways to reorganize the algorithm to make it faster, but do the same steps.

Frequently, it is easy to eliminate the recursion. For example, Kruskal's minimum spanning tree algorithm uses the following recursive strategy. Given a graph and a partial spanning tree P. Find the minimum cost edge that connects nodes that are not in the same component in P. Add that edge to P. Recurse.

Instead, it is equivalent to do the following: Sort the edges by weight. Initialize P to the empty set. Running through the edges from smallest to largest weight , check wheter the edge connects nodes that are not in the same component in P. If it does, add it to P; otherwise, go on to the next edge.

These two do the same work; the second just does part of that work (sorting edges) in advance.

To analyze these algorithms, lets think of how we check whether two nodes are in the same component. Note that we can use a depth-first search, which is $O(n+m)$ to see if the two nodes are connected in P. One observation is that $|P| < n$ always, so the time to do depth-first search *in P* is $O(n + |P|) = O(n)$. The obvious method to do the first, recursive algorithm involves searching for each edge, and for each edge doing the DFS. This would take time $O(mn)$ per recursive stage. There are n recursive stages, so the total time is $O(mn^2)$.

For the second algorithm, we sort the edges in advance. This takes time $O(m \log m)$ using a mergesort or quicksort. Then we go through each edge once, and do a DFS once per edge. That will take time $O(nm)$ total. So the total time of the algorithm is $O(m \log m + mn) = O(mn)$, a factor of n better than the first. The reason mn is bigger than $m \log m$ is that $m < n^2$, so $\log m \leq 2 \log n < n$.

In the last few weeks of class, we'll see that by identifying the right data structures to do the operations, we can often speed up greedy algorithms quite a bit. We'll show how to use data structures for disjoint sets to reduce the time of Kruskal's algorithm to $O(m \log m)$, i.e., make the hardest part sorting the edges! For now, you don't need to worry about coming up with the best version of the greedy strategy.