Advanced Algorithms

Class Notes for Monday, October 15, 2012

Bernard Moret

1 Algorithm Design

Now that we have covered worst-case, amortized, competitive, and randomized analysis, we are ready to start studying algorithmic design. As stated at the beginning of the course, the focus is on efficient algorithms—algorithms that run in low polynomial time. Most such algorithms fall into one of five categories: greedy, iterative improvement, divide-and-conquer, dynamic programming, and randomized—although, strictly speaking, the last is not separate: one can combine randomization with any of the first four. We will take up each in turn over the rest of the semester.

2 Greedy Algorithms

These are everyone's favorite algorithms: they are simple and intuitive, as they simply seek to optimize the next choice based on just local information. This approach makes them easy to design, easy to code, and usually very efficient, but it also prevents most of them from reaching the global optimum. As a result, greedy algorithms are usually heuristics, used to get quick (but not all that good) solutions, or as part of a more complex optimization algorithm. However, some of the most famous algorithms are in fact optimal greedy algorithms, among them Kruskal's and Prim's algorithms for constructing a minimum spanning tree (MST), Dijkstra's algorithm to compute a tree of shortest paths, and Huffman's algorithm to produce an optimal prefix-free code.

I assume that you are familiar with at least one of the two MST algorithms, but I will rephrase both in a single, more general context, and proceed to prove that this general algorithm indeed returns a minimum spanning tree. In the process, we will learn something about the MST problem, but also about what may be required of a problem for it to admit an optimal greedy solution.

2.1 Greedy algorithms for the MST problem

Greedy algorithms use only a fraction of the information available about the problem. Bottom-up greedy methods build solutions piece by piece (starting from the empty set) by selecting, among the remaining pieces, that which optimizes the value of the partial solution. Thus the idea is to produce the largest immediate gain while maintaining feasibility. Top-down greedy methods (much less common) build solutions by selecting a piece whose *removal* optimizes the value of the remaining collection while ensuring that this collection continues to contain feasible solutions.

In the case of the MST problem, we are given an undirected graph $G = (V,E)$, and a length (distance/weight/etc.) for each edge, $d\colon E \longrightarrow \mathcal{R}$. A bottom-up method starts from an empty set and adds one piece at a time (a vertex or an edge, although the distinction is somewhat artificial), subject to not creating cycles, until a tree is built; a top-down method starts from the entire graph and removes one edge at a time, subject to not disconnecting the graph, until a tree is obtained. In each case, the choice is made on the basis of the contribution made by the chosen edge (or vertex) to the current collection, not to a complete spanning tree.

We can view the bottom-up methods as proceeding by coalescing equivalence classes. Here an equivalence class consists of a set of vertices with an associated set of edges that form a minimum spanning tree for the vertices in the class. Initially, each vertex is the sole element of its equivalence class, and the associated set of edges is empty. When the algorithm terminates, only one equivalence class remains and the associated set of edges is a minimum spanning tree. At each step of the algorithm, we select an edge with an endpoint in each of two equivalence classes and coalesce these classes, thereby combining two trees into one larger tree. Edges, both endpoints of which lie in the same equivalence class, are permanently excluded, since their selection would lead to a cycle. In order to minimize the increase in the value of the objective function, the greedy method dictates that the allowable edge of least cost be chosen next. This choice can be made with or without additional constraints. At one extreme we can apply no additional constraint and always choose the shortest edge that combines two spanning trees into a larger spanning tree. At the other extreme we can designate a special equivalence class which must be involved in any merge operation. The first approach can be viewed as selecting edges; it is known as Kruskal's algorithm, after J. B. Kruskal, who first presented it in 1956. The second, at least for programming purposes, is best viewed as selecting vertices (adding them one by one to a single partial spanning tree) and is known as Prim's algorithm, after R. C. Prim, who presented it in 1957. Thus these are among the oldest algorithms formally defined in Computer Science.

2.2 A proof of correctness for the generalized algorithm

We can prove the correctness of both algorithms (Kruskal's and Prim's) at once by proving the correctness of the more general algorithm.

Theorem 1. *If, at each step of the algorithm, an arbitrary equivalence class, T, is selected, but the edge selected for inclusion is the smallest that has exactly one endpoint in T, then the final tree that results is a minimum spanning tree.*

Proof. As G is connected, there is always at least one allowable edge at each iteration. As each iteration can proceed, irrespective of our choice of T, and as an iteration decreases the number of equivalence classes by one, the algorithm terminates.

The proof is by induction, though we present it somewhat informally. Let T_A be a spanning tree produced by the above algorithm and let T_M be a minimum spanning tree. We give a procedure for transforming T_M into T_A. We will form a sequence of trees, each slightly different from its predecessor, with the property that the sums of the lengths of the edges in successive trees are the same. Since T_A will be at the far end of the sequence of transformations, it must also be a minimum spanning tree. Label the edges of T_A by the iteration on which they entered the tree. Let e_i be the edge of lowest index that is present in T_A but not in T_M. The addition of e_i to T_M forms a cycle. Note that the length of e_i is greater than or equal to that of every other edge in the cycle; otherwise, T_M would not be a minimum spanning tree, because breaking any edge in the cycle would produce a spanning tree and breaking a longer edge, if one such existed, would reduce the total cost. Now, when e_i was added to the forest of trees that eventually became T_A, it connected an arbitrarily chosen tree, T, to some other tree. Traverse the cycle in T_M starting from the endpoint of e_i that was in T and going in the direction that does not lead to an immediate traversal of e_i. At some point in this traversal, we first encounter an edge with exactly one endpoint in T. It might be the first edge we encounter, or it might be many edges into the traversal, but such an edge must exist since the other endpoint of e_i is not in T. Furthermore, this edge, call it $\hat{e}$, cannot be e_i. Now the length of e_i cannot exceed that of $\hat{e}$, because $\hat{e}$ was an allowable edge, but was not selected; thus the two edges have equal length. We replace $\hat{e}$ with e_i in T_M: the resulting tree has the same total length. Note that $\hat{e}$ may or may not be an edge of T_A, but if it is, then its index is greater than i, so that our new tree now first differs from T_A at some index greater than i. Replacing T_M with this new minimum spanning tree, we continue this process until there are no differences, i.e., until T_M has been transformed into T_A. $\square$

2.3 What makes a greedy algorithm optimal?

Perhaps a better question would be: "What characteristics should a problem have in order to be solvable optimally with a greedy algorithm?" The answer remains incomplete today, but is complete and exact for optimality criteria that include linear objectives (as in the MST problem: a simple sum) and bottleneck objective (minimax and maximin problems). The details are complex and the results themselves not all that useful, since determining whether a problem has the required properties is generally very hard. But the framework is interesting and brings to light important lessons from the MST algorithms. First, since the greedy algorithms produces a succession of sets, each with one element added to the previous set, a good abstract setting for describing problems is a lattice—a partially ordered set of (feasible) subsets of some finite set S. Second, because any subset in that lattice must be constructible by the greedy algorithm (there is always some assignment of values that would make that subset the best of its size), there must be a path in the lattice from the empty set to any lattice element. Together, these two requirements define what is known as an accessible set system:

Definition 1. *Let S be a finite set and $\mathcal{C}$ a nonempty collection of subsets of S; the structure $(S, \mathcal{C})$ is called an* accessible set system *if and only if it obeys the following axiom:*

(accessibility axiom) If $X \in \mathcal{C}$ and $X \neq \emptyset$, then there exists $x \in X$ such that $X - \{x\} \in \mathcal{C}$.

The elements of S are the building blocks, e.g., edges in the MST problem. The elements of $\mathcal{C}$ are partial solutions (subforests in the MST problem) and so are called feasible sets. A maximal feasible set (one that cannot be expanded by the greedy algorithm under any objective function) is called a basis. To set this up for an optimization problem, we add an objective function, which assigns a value to each feasible set and we ask to find a basis with the best possible value. Informally, the greedy algorithm, when run on a set system, builds a solution by beginning with the empty set and successively adding the current best element while maintaining feasibility. We formalize this notion as follows.

Definition 2. *Given an accessible set system, $(S,\mathcal{C})$, and an objective function, $f\colon \mathcal{C} \to \mathcal{R}$, the* best-in greedy algorithm *starts with the empty set; at each step i, it chooses an element $x_i \in S$ such that*

1. $\{x_1, x_2, \ldots, x_i\} \in \mathcal{C}$*; and*
2. $f\big(\{x_1,\ldots,x_i\}\big) = \max\big\{ f\big(\{x_1,\ldots,x_{i-1},y\}\big) \mid \{x_1,\ldots,x_{i-1},y\} \in \mathcal{C}\big\}$*;*

the algorithm terminates when it has constructed a basis, i.e., when it can no longer incorporate another element into its partial solution.

Note that the objective function need not improve at every step: all that the greedy algorithm does is choose the best available extension to the current set.

In the case of the MST problem, our lattice of subsets of edges has much stronger properties, however. First, no basis can contain another—in fact, all have the same size ($|V|-1$ edges in the spanning tree); second, any subset of a feasible set is itself a feasible set, a property called, for obvious reasons, heredity. Third, and most interesting, the step used in the proof of our theorem above shows that we must have a type of exchange property. We used this exchange only on full solutions, but in fact a bit of thinking shows that it can be used at any step in the construction, since we only made used of the current step in the greedy algorithm. We can state this exchange property as follows.

(exchange axiom) If X and Y are members of $\mathcal{C}$ such that $|X| > |Y|$, then there exists $x \in X - Y$ such that $Y \cup \{x\}$ is also in $\mathcal{C}$.

An immediate consequence of this axiom is that all bases must have the same size.

An accessible set system that obeys the exchange axiom is called a *matroid*; matroids have been studied for half a century by mathematicians and CS theoreticians, find applications in numerous domains, and have given rise to a lot of beautiful mathematics. However, they are only a first step: a matroid structure is sufficient for a greedy algorithm to return optimal solutions for problems with linear objectives, but it is not necessary—the exchange axiom is too strong. Putting this another way: the MST problem is perhaps the simplest and most structured version of a problem where greedy algorithms can succeed, but there are more complex problems, with less structure, for which we can design optimal greedy algorithms. In 1993, Helman, Moret, and Shapiro (SIAM J. Discr. Math.) showed that neither matroids nor greedoids (a weaker version, replacing the exchange axiom by a pair of rather less intuitive ones) gave the right characterization, and gave the correct characterization, which works for both linear and bottleneck objectives. Little progress has been made since then for other objective functions.