Advanced Algorithms

Class Notes for Friday, October 19, 2012

Bernard Moret

1 Another Optimal Greedy Algorithm

Data compression started quite early in the history of communication—Morse code dates from 1837. Morse code is a variable-length code based on short ("dots") and longs ("dashes") pulses, in which frequent characters are assigned short codes (e.g., the letter E is a single dot) and less common ones are assigned longer codes (for a letter, the longest code is four pulses). You may have seen or heard in a movie the Morse code for the international distress signal, which is three dots, three dashes, and three dots, usually written out as "SOS." Since then, myriads of coding and compression methods have been proposed. We consider a simple formulation: given the alphabet used for the original document and a probability distribution over the alphabet, produce a prefix-free binary code that minimizes the expected length of an encoded message. A code, in this case, is simply a mapping that takes a character from the input alphabet and replaces it by a *glyph*, that is, a string in the output alphabet, here just a binary string. A prefix-free code is one in which no glyph can be the prefix of another. As formulated, such a code is known as a Huffman code, an American mathematician who published his algorithm in 1952 while he was a student at MIT. Huffman codes can be tailored to a language, to a domain of use (e.g., commercial or diplomatic language), or to a particular (long) document by estimating the probability distribution of characters for the domain of application and communicating the code itself prior to establishing transmission. They were in wide use for digital communication in the 1960s and 1970s—modems had built-in encoders and decoders for Huffman codes.

A prefix-free binary code can be viewed as a complete binary tree (a tree in which each node has zero or two children), where the leaves correspond to the glyphs and each glyph can be read as the succession of binary choices (left/right, coded as 0/1) made on the path from the root of the tree to the leaf corresponding to the chosen glyph. The prefix-free property then simply states that no glyph can be on the path to another, restricting glyphs to correspond to leaves. To optimize the code, we expect to see rare characters correspond to leaves at the end of long paths from the root, and common ones at the end of short paths.

Thus we can formally state the problem as follows.

> *Given an alphabet of n characters, $\Sigma = \{a_1, a_2, \ldots, a_n\}$, with associated frequencies of occurrence, $f_1, f_2, \ldots, f_n$, find the binary tree with n leaves, each labelled with a different character, such that the weighted path length, $\sum_{i=1}^{n} f_i \cdot l_i$, is minimized, where l_i is the length of the path from the root to the leaf labelled with a_i.*

Huffman's algorithm is a simple greedy algorithm that builds the code tree bottom-up, by successive pairing of subtrees. We know that the tree has *n* leaves, one for each future

glyph, and we know the frequency of each glyph. Therefore, we start with one subtree of a single node (a leaf) for each glyph. The algorithm proceeds by selecting two subtrees, pairing them by making them children of a new common parent, and iterating until the forest consists of a single tree. Since we start with n subtrees, it will take $n-1$ iterations to produce a single tree. The greediness comes in the choice of the two subtrees: since the construction is bottom-up, the first leaves to be paired will become the leaves most distant from the root of the code trees, and therefore they are chosen to be the leaves with the lowest frequency of occurrence. As we pair leaves into larger subtrees, these subtrees must be assigned a frequency of occurrence of their own in order to be usable in later steps of the algorithm; obviously, the frequency of occurrence of a subtree is just the sum of the frequencies of occurrence of its two component subtrees—or, equivalently, the frequency of occurrence of a subset of characters of the alphabet is just the sum of the individual frequencies of occurrence of each of these characters.

Clearly, optimal codes cannot be unique, since, at each code position, we could invert the choice of bit without affecting the prefix-free property nor the optimality—this inversion of bits corresponds to an exchange of left and right children in the tree. Furthermore, we could swap any two subtrees rooted at the same depth in the tree. Nor is it clear that the greedy algorithm could build all of these trees. Thus we will prove that the tree(s) built by the algorithm are optimal.

The proof is by induction on n, the number of glyphs. When n equals one or two, there is effectively only one tree. Assume then that the algorithm produces an optimal tree for all instances of the problem with no more than $n-1$ glyphs and consider an instance with n glyphs. *Ad absurdum*, assume that the algorithm produces a nonoptimal tree; denote this tree by T_H and let T_{opt} be some tree with minimum weighted path length. Notice first that the glyph with the least frequency of occurrence must be at the greatest depth in T_{opt}—if such were not the case, we could swap this leaf with a lowest leaf in the tree, for a net reduction in weighted path length, thereby contradicting the assumed optimality of T_{opt}. The optimal tree can be transformed so that the glyph with the second least frequency is made the sibling of the node with the least frequency. Observe that this sibling is also a leaf and at the lowest level in the tree, so that, if the glyph with the second least frequency is elsewhere in the tree, it can be swapped with this sibling without increasing the value of the objective function. These two results allow us to assume, without loss of generality, that both T_{opt} and T_H have the two glyphs of least frequency paired as siblings. Consider then the two trees, T'_{opt} and T'_H, obtained by replacing these two siblings and their parent with a single artificial glyph, with a frequency equal to the sum of the frequencies of the two replaced glyphs. Denote the weighted path length of tree T by $w(T)$. Since the new glyph sits just one level higher than the two replaced glyphs, we have $w(T'_{\text{opt}}) = w(T_{\text{opt}}) - f_{l_1} - f_{l_2}$ and $w(T'_H) = w(T_H) - f_{l_1} - f_{l_2}$, where f_{l_1} and f_{l_2} are the frequencies associated with the two replaced glyphs. Now T'_H is a Huffman tree on the alphabet of $n-1$ glyphs formed by replacing a_{l_1} and a_{l_2} with the new glyph. Because we have $w(T_{\text{opt}}) < w(T_H)$, it follows that $w(T'_{\text{opt}}) < w(T'_H)$; but this latter inequality contradicts the optimality of Huffman's algorithm when applied to alphabets of size not exceeding $n-1$.

In fact, it is easy to check that the greedy algorithm cannot always produce all optimal trees: there can be equivalent pairings (subtrees at the same depth in the code tree) that it cannot produce because it is constrained to select the two subtrees of lowest frequency.

2 Greedy Algorithms as Heuristics

In spite of the previous algorithms we studied, the main use of greedy algorithms is as heuristics for difficult optimization problems. Such problems include both NP-hard problems, for which we do not expect to be able to design efficient optimal algorithms, and problems for which we have polynomial-time optimal algorithms that are rather slow (such as cubic or quartic time algorithms). Much of the time, the quality of greedy solutions cannot be bounded through formal derivations, but is proves very good in practice; occasionally, however, it is possible to produce formal bounds on the quality of the solution. In standard terminology, a heuristic is an algorithm that offers no proven guarantee on the quality of the solution (neither optimality, nor any fixed guarantee on the distance to optimal, such as "within 20%"), whereas an approximation algorithm is one that does not ensure optimality, but does offer a proven guarantee on the quality of the solution. Greedy algorithms are the basis of most good practical heuristics, but only rarely the basis for good approximation algorithms.

We shall explore a few NP-hard examples and one polynomial-time one. The NP-hard problems we shall look at are all well known: the TSP, the knapsack, and vertex cover; the polynomial-time example is one of the most fundamental optimization problems, matching in graphs.

2.1 Knapsack

The *Knapsack* problem is classic NP-hard problem. In many ways, however, it is an "easy" NP-hard problem, in that its complexity derives not so much from its structures, but from arithmetic questions. (We shall develop this statement when discussing dynamic programming.) An instance of the knapsack problem is given by a set of items, each with a size and a value (both positive integers), and by the size of the knapsack. The problem is to select a subset of items whose total does not exceed the size of the knapsack and whose total value is maximum among all such subsets. The most obvious heuristic is to order the objects according to their value "density," that is, the ratio of their value to their size. In fact, if we could use fractions of items instead of being constrained to use or not use entire items are once, this heuristic would immediately provide an optimal solutions: as we reach the first item that cannot fit completely within the remaining capacity of the knapsack, we use whatever fraction of that item does fit within that remaining capacity. (To see that this solution is optimal, imagine the objects as being divided into pieces of unit size. Replacing one small piece with another of larger value density improves any solution, so that the optimal packing includes all of the pieces of highest density that fit, plus as many pieces of the next highest density as necessary to fill the knapsack exactly. This approach using fractional amounts of objects is characteristic of an approach known as *linear programming*.)

Let $G_d(I)$ be the value produced by the greedy method applied to instance I of the knapsack problem (the subscript d is a reminder that objects are sorted by density), let $G_l(I)$ denote the perfect solution obtained by using fractional items (the l is for linear programming), and let $OPT(I)$ be the value of the optimal filling. Obviously, we have $G_dI \leq OPT(I) \leq G_l(I)$. We are interested in the least upper bound for the ratio

$$\frac{OPT(I)}{G_d(I)},$$

as I ranges over all instances. We are using the ratio of OPT to G_d, as opposed to their difference, as the difference can easily be made arbitrarily large by the simple artifice of multiplying the values (but not the weights) by a suitable constant. We use the least upper bound because we want to know the very worst that this ratio can become—though it is possible that the ratio is achieved only as a limit and no instance actually achieves it. For our heuristic as stated, this ratio is unbounded, i.e.,

$$\text{lub}\left\{\frac{OPT(I)}{G_d(I)}\right\} = \infty,$$

as demonstrated by the following example.

> Consider a knapsack of capacity N and two objects, the first of weight 2 and value 3, the second of weight and value both equal to $N-1$. We see that $G_d(I)$ equals 3 since, once the first object is selected, there is not enough room for the second. Thus we have
>
> $$\frac{OPT(I)}{G_d(I)} = \frac{N-1}{3},$$
>
> a value which grows arbitrarily large as N grows.

In order to get a better result, we need to modify the method slightly. Consider these two potential fillings: the result of G_d and the filling consisting of the single most valuable object; select whichever of these two has the larger value. We denote the result of the modified heuristic by $G_{md}(I)$. Note that the modification does not affect the overall running time of the algorithm, but does eliminate pathological cases such as that above. We now claim that this new algorithm is in fact an approximation algorithm, that is, it produces solutions that are no worse than some fixed fraction of the optimal—here no worse than half of value of the optimal solution.

Theorem 1. *For any instance of the knapsack problem, we have*

$$1 \leq \frac{OPT(I)}{G_{md}(I)} < 2.$$

The lower bound is obviously tight. The upper bound is tight as well: consider a family of instances made of three items; the knapsack has capacity $2N$, two of the objects have size and value equal to N, and the third object has size $N+1$ and value $N+2$. That third

object has the highest value density and is also the single most valuable object, so we have $G_{md} = N + 2$; but clearly the optimal solution is to pack the first two objects, for a value of $OPT = 2N$, so that the ratio is $2 - \frac{4}{N+2}$, which converges to 2 as N grows large.

Now we turn to a proof of our theorem. We first modify our heuristic again. Instead of running the density-ordered greedy heuristic all the way to completion, we terminate it as soon as an object fails to fit. Call this truncated version G_{td} and call the modification of G_{md} which uses G_{td} as its first alternative G_{mtd}. This modification never causes G_{mtd} to perform better than G_{md}, so that it will be sufficient to prove that the ratio of the optimal to G_{mtd} is less than two. The reason for defining this truncated version is to be able to relate it to the linear programming solution G_l; we have

$$G_l(I) = G_{td}(I) + x_j v_j$$

where j denotes the first object that fails to fit and where we have $0 < x_j < 1$, the fraction of the object j that fills the knapsack exactly,

Now, for any instance I, the following inequalities hold:

$$G_{td}(I) \leq G_{mtd}(I) \leq G_{md}(I) \leq OPT(I) \leq G_l(I).$$

We have two cases, depending on whether or not the value of the first object that failed to fit, v_j, is larger than the value of the truncated greedy filling, $G_{td}(I)$.

Case 1: $v_j \leq G_{td}(I)$. It follows that

$$OPT(I) \leq G_l(I) = G_{td}(I) + x_j v_j < G_{td}(I) + v_j \leq 2G_{td}(I) \leq 2G_{mtd}(I).$$

The strict inequality comes from the fact that x_j is less than 1, since object j did not fit completely into the knapsack.

Case 2: $v_j > G_{td}(I)$. It follows that $G_{mtd}(I) \geq v_j$, as the second alternative of the algorithm will be selected (although the jth object may not be the most valuable, it is more valuable than the filling achieved by the truncated greedy procedure). Hence we have

$$OPT(I) \leq G_l(I) = G_{td}(I) + x_j v_j < G_{td}(I) + v_j < 2v_j \leq 2G_{mtd}(I).$$

Thus the knapsack problem can be approximated reasonably well by a combination of two very simple procedures. How close the approximation really is can best be judged by using the ratio $G_l(I)/G_{md}(I)$ (in which both terms are easily computed) as a conservative estimate for $OPT(I)/G_{md}(I)$ (in which, of course, it is NP-hard to compute $OPT(I)$).

2.2 Vertex Cover

The *Vertex Cover (VC)* problem is very simply stated: given an undirected graph, find a minimum subset of vertices such that every edge of the graph has at least one endpoint in that subset. (An edge is "covered" by either of its endpoints.) It is also an NP-hard

problem, and remains hard even for very restricted graph structures (e.g., it remains hard for planar graphs with vertices of degree not exceeding 3). The obvious greedy algorithm is to select next that vertex which is an endpoint of the largest number of as-yet-uncovered edges. This greedy algorithm does well in practice, but it offers no guarantees: in the worst-case, it may produce a cover that is $(\log |V|)$ times larger than the optimal one. In contrast, it is an easy matter to design an approximation algorithm that guarantees a worst-case ratio of 2—even though the algorithm appears rather strange at first. This algorithm proceeds as follows: select any as-yet-uncovered edge; put *both* of its endpoints in the set cover; remove the edge, its two endpoints, and any edges covered by these two endpoints from the graph; and repeat until no uncovered edge remains. The strange, but crucial part, is the selection of *both* endpoints: after all, a single one would have covered the edge. The selection of both, however, ensures that every new edge selected by the algorithm shares no endpoint with any previously selected edge: the collection of selected edges forms what is called a *matching* (something we will study in detail later). Say this matching has k edges; then our algorithm placed exactly $2k$ vertices in the cover. But now note that the optimal solution must have at least one of the two endpoints of each of the k edges in the matching, since an endpoint of one edge in the matching cannot cover any other edge in the matching; thus the optimal solution must have at least k vertices, and our solution cannot be worse than twice the optimal. The contrast between the pure greedy heuristic (greatest degree vertex), which does well in practice but very poorly in the worst case, and the approximation algorithm, which has a decent guarantee, but (as it turns out) does not do much better in practice, is a common finding in greedy heuristics: the best ones often cannot be bounded. (Of course, we can run *both* algorithms and retain the smaller cover, thereby doing well in practice *and* guaranteeing a performance no worse than twice the optimal.)