Advanced Algorithms

Class Notes for Monday, October 22, 2012

Bernard Moret

1 Greedy Algorithms as Heuristics

1.1 The Travelling Salesperson Problem (TSP)

The TSP problem is perhaps the best studied of all hard optimization problems. Entire books and thousands of papers have been written about it, proposing and analyzing hundreds of heuristics and many optimal search methods. It is so well studied (and, given how hard it is, so well solved) that a standard approach to the solution of many NP-hard optimization problems is to begin by checking whether it would be possible to rephrase these problems as versions of the TSP and use some of the existing algorithms for the TSP. We will not look at the more involved heuristics nor at the attempts to derive reasonably efficient optimal algorithms, but just look at two categories of simple greedy heuristics. (We will return to the TSP, however, when studying iterative improvement algorithms, as well as when studying dynamic programming.)

Recall that an instance of the TSP is given by a set of n cities and an $n \times n$ pairwise distance matrix (we shall assume that the distance measure is symmetric); the goal is to find a shortest tour, that is a simple cycle that spans all n cities and that has the smallest possible total length. Since we can start anywhere on a cycle, perhaps the simplest possible greedy heuristic can be phrased as follows:

> Starting at any city, always move to the nearest unvisited city, until no such city is left, then return to your starting point.

For obvious reasons, this is known as the "nearest-neighbor" heuristic. Note its close resemblance to Prim's algorithm: like Prim's, it adds one vertex (city) at a time to a connected subgraph (a tree for Prim's, a path for this algorithm). This is not a very good heuristic in practice, although it can be somewhat improved by noting that different starting cities will typically yield different tours, so that one can run n separate calls to the NN algorithm, one from each possible starting point, and keep the best of the 50 tours thus obtained.

A somewhat better heuristic relaxes the connectivity constraint. Instead of thinking of a tour as a sequence of edges that must be constructed in increasing subscript order (i.e., selecting the first edge of the tour first, the second edge of the tour second, etc.), it is useful to think of the tour as an unordered set of edges, which when drawn on a map of the cities happens to form a spanning simple cycle. Within this framework, a feasible partial solution is any subset of edges that forms a collection of disjoint simple paths (including isolated vertices, which are simple paths of length 0), since such a collection can always be extended to a tour. The initial collection is empty; the solution is a simple path through all n vertices, which is then completed, as in the NN algorithm, by a last "return" edge. It is the equivalent of Kruskal's algorithm and can be phrased as follows:

> While there remain disconnected components, select the shortest remaining edge that neither induces a cycle nor creates a vertex of degree 3.

Just as in Kruskal's algorithm, this heuristic is "maximally greedy," always adding the cheapest possible edge remaining in the entire set; and, similarly again, the choice must be filtered to avoid cycles; finally, this heuristic, like Kruskal's algorithm, returns a unique answer when all edge costs are distinct (although for the TSP, of course, the answer may not be optimal).

Neither of these two heuristics performs well in practice. As both are methods that build a tour one edge at a time, it makes sense to ask whether methods that build a tour one vertex at a time might do better. These methods start with a trivial cycle and expand it into a tour by incorporating a new vertex at each step. Call the new vertex x; in order to maintain a cycle, the algorithm replaces a single edge of the cycle, say (u,v), with the two edges (u,x) and (x,v). In order to minimize the increase in the value of the objective function, the edge to be replaced is chosen so as to minimize $d(u,x)+d(x,v)-d(u,v)$, which is nonnegative when the triangle inequality holds. These replacements neither remove vertices nor alter their relative order in the cycle: they simply determine where in the relative order the next vertex should be placed. Selection of the next vertex to include can be random, or based on additional distance considerations, such as nearest insertion (choose the vertex closest to the group of already included vertices), farther insertion (choose the vertex farthest from the group of already included vertices), or cheapest insertion (a double minimization: choose the vertex that will minimize the min value of $d(u,x)+d(x,v)-d(u,v)$). In practice, the cheapest insertion is a poor (and expensive) choice and nearest insertion is worse than random; farthest insertion tends to fare best, presumably because it builds an "outline" of the full tour quickly, then works on resolving the details.

However, all of these methods remain heuristics, with no performance guarantee; and none of them dominates the others, in the sense that it is always possible to find instances of the TSP on which any one of these heuristics will beat another.

1.2 Bipartite matching

As a last, quick example of greedy heuristics, let us look at the problem of maximum matching in bipartite graphs. As we saw, a matching in a graph is a subset of edges that do not share any endpoint; a maximum matching is a matching of maximum cardinality. A graph is said to be bipartite if its vertices can be partitioned into two sets in such a way that all edges of the graph will have one endpoint in one set and the other endpoint in the other set—the vertices of each set form an independent set. Matching in bipartite graphs is one of the fundamental optimization problems, as it is used for *assignment* problems, that is, problems where one wants to find the optimal way to assign, say works crews to jobs—problems to be solved everyday on construction sites, in factories, in airlines and railways, as well as in job scheduling for computing systems.

A simple greedy algorithm selects the unmatched vertex of lowest degree and matches it with its neighbor or, if it has multiple neighbors, with a neighbor of lowest degree; then the two matched vertices and all incident edges are removed and the process iterates. This algorithm is simple and efficient, but not optimal; in fact, it is not an approximation algorithm, as its error cannot be bounded by a constant factor. However, it is surprisingly hard to design a counterexample—a single bipartite graph on which the greedy algorithm (with possibly different starts if there are multiple unmatched vertices of lowest degree) cannot return the optimal solution.

2 Iterative Improvement Algorithms

As we have seen, greedy algorithms only rarely guarantee an optimal solution; and most of the time, we cannot get an approximation guarantee either. The heuristic may be excellent in practice, but how can we improve a solution returned by the greedy algorithm, especially when that solution may be poor? One good approach is to design an iterative improvement algorithm: an algorithm that refines the current solution through local changes, using an approach that can be repeated many times, each time gaining in the quality of the solution. Such improvements may not guarantee an optimal solution as a final result, but they usually provide substantial improvements over the initial solution; and, in a few very important cases, they do provide an optimal solution. Almost all machine learning algorithms are iterative improvement algorithms and a few of these guarantee an optimal solution—for instance, the perceptron algorithm, one of the oldest algorithms in the area, guarantees finding a linear separator between two populations if one exists.

2.1 The TSP again

Let us start by revisiting the TSP. Given a current tour that you suspect of being suboptimal, what can you do to improve it? If the change is to be local and inexpensive, it should involve just a few edges at a time. The minimum number of edges involved is two (removing one leaves no choice but to put it right back). After removing two nonadjacent edges, we get two simple paths with four “cut ends;” there are three ways of reconnecting these cut ends, but of these one is the original and the other is illegal, as it closes each path into a cycle and thus produces two cycles rather than one. This leaves one alternative reconnection pattern, whose cost can be compared to the cost of the original connection (is the sum of the lengths of the two new edges smaller than the sum of the lengths of the two original edges?), keeping the better pair. This step can be attempted with every possible pair of nonadjacent edges; moreover, after several changes, new nonadjacent pairs have been created that can also be tested. Thus one can devise a routine that will keep testing until no pair yields an improvement. At this stage, the tour is said to be *2-optimal*, i.e., optimal with respect to 2-edges swaps. The routine is relatively inexpensive (at least for an NP-hard problem) and produces quite good tours, although it remains a heuristic, as the quality of the solution produced cannot be bounded, Once 2-exchanges have been exhausted, one could then proceed to 3-exchanges: remove 3 nonadjacent edges, thereby cutting the tour into three simple paths and leaving 6 cut ends, and reconnect. There are now a cubic number of candidate triples and, for each triple, 8 reconnection patterns, so the computational demands increase rapidly. Theoretically, one could use k-exchanges for larger and larger values of k, but the work required quickly outweighs any benefits, since even large values of k cannot ensure optimality. (Obviously, optimality can be guaranteed by setting k to the number of cities, which simply undoes the entire work to date and requires solving the problem from scratch—not a solution at all! And there exists a proof that, even with k set to a quarter of the number of cities, the solution returned can be twice the length of the optimal solution.) The time is better spent exploring 2-opt solutions from various starting tours, for instance; the famous Lin-Kernighan heuristic for the TSP, which seems to get consistently within 5% of optimal, mixes full 2-opt searches with partial 3-opt searches and dynamic programming.

2.2 Bipartite matching

Matching and network flow are the two most important problems for which an iterative improvement method delivers optimal solutions. Thus we begin with the simplest version of these problems, maximum bipartite matching. In improving an existing matching, we must start by identifying unmatched vertices of degree at least 1, and we need at least one such on each side of the bipartite graph. Consider the trivial 4-vertex bipartite graph with vertex set $\{a,b,1,2\}$ and edge set $\{\{a,1\},\{a,2\},\{b,1\}\}$, with current matching $M = \{\{a,1\}\}$. It is clear that there exists a larger matching, namely $M^* = \{\{a,2\},\{b,1\}\}$. Note that, in order to transform M into M^*, the set of matched vertices will simply gain two new members, but the set of matched edges, while larger by one, may have nothing in common with the previous set. In this trivial example, we could have started our search at vertex b, which has just one neighbor, vertex 1; but vertex 1 was already matched, so we had to follow the matched edge back to vertex a, where we found that a had an unmatched neighbor, vertex 2. We thus identified a path of three edges, the first and the last unmatched, the middle one matched; by flipping the status of each edge, from matched to unmatched and vice versa, we replaced a path with one matched edge by the same path, but with two matched edges. Let us formally define what this type of path is.

> Let $G = (V,E)$ be a graph and M be a matching. An *alternating path* with respect to M is a path such that such that every other edge on the path is in M, while the others are in $E - M$. If, in addition, the path is of odd length and the first and last vertices on the path are unmatched, then the alternating path is called an *augmenting path*.

The reason it is called an augmenting path is that we can use is to augment the size of the matching: whereas an alternating path may have the same number of edges in M and in $E - M$, or one more in M, or one more in $E - M$, an augmenting path must have one more edge in M than in $E - M$. Moreover, because of the definition of matchings, it is safe to flip the status of every edge in an augmenting path from matched to unmatched, and vice versa: none of the vertices on the augmenting path can have been the endpoint of a matched edge other than those already on the path.

Augmenting paths are thus the tool we needed to design an iterative improvement algorithm: in general terms, we start with an arbitrary matching (including possibly an empty one), then we search for an augmenting path in the graph; if one is found, we augment the matching by flipping the status of all edges along the augmenting path; if none is found, we stop. The obvious question, at this point, is whether the absence of any augmenting path indicates just a local maximum or a global one. The answer is positive: if G has no augmenting path with respect to M, then M is a maximum matching—it is optimal. We phrase this result positively.

Theorem 1. *Let G be a graph, M^* an optimal matching for G, and M any matching for G such that we have $|M| < |M^*|$. Then G has an augmenting path with respect to M.*

This result is due to French mathematician Claude Berge and so known as Berge's theorem. The proof is deceptively simple, but note that it is nonconstructive.

Proof. Let $M \oplus M^*$ denote the symmetric difference of M and M^*, i.e., $M \oplus M^* = (M \cup M^*) - (M \cap M^*)$, and consider the subgraph $G' = (V, M \oplus M^*)$. All vertices of G' have degree two or less, because they have at most one incident edge from each of M and M^*; moreover, every connected component of G' is one of: (i) a single vertex; (ii) a cycle of even length, with edges drawn alternately from M and M^*; or (iii) a path with edges drawn alternately from M and M^*. As the cardinality of M^* exceeds that of M, there exists at least one path composed of alternating edges from M and M^*, with more edges from M^* than from M. The path must begin and end with edges from M^* and the endpoints are unmatched in M, because the path is a connected component of G'; hence this path is an augmenting path. $\square$

Berge's theorem shows that the use of augmenting paths not only enables us to improve on the quality of an initial solution, it enables us to obtain an optimal solution. Note that the definitions of alternating paths and augmenting paths hold just as well for nonbipartite graphs as for bipartite ones; and Berge's theorem does too. The next step is to develop an algorithm for finding augmenting path and this is where the difference between bipartite and nonbipartite graphs will show.