Lecture 10: Local search

Plan/outline

We completed the analysis of Prim's algorithm for MST (see the notes for the previous lecture). Then, we started looking at local search algorithms.

Local search

Suppose we have a constrained optimization problem in which we have no real idea of how to proceed. One naive solution is to start with any feasible solution, and try to "perturb it" slightly, so as to improve the objective value, while maintaining feasibility. How we define a perturbation depends on the problem at hand.

It is natural to ask if/when local search is optimum. After all, since the objective value monotonically improves, we are guaranteed that the algorithm converges to *some solution* (assuming the feasible space is "compact"/bounded/finite).

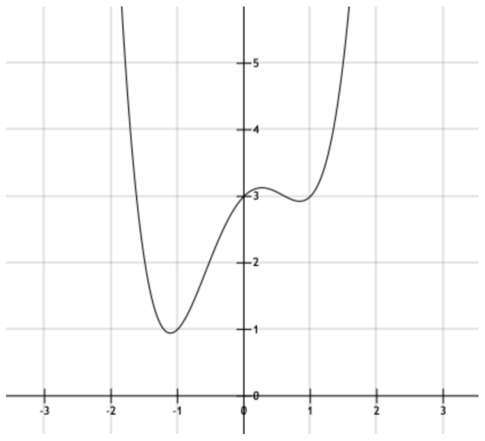
One of the most popular applications of local search is multi-variate optimization. Suppose we have a real valued function $f : \mathbb{R}^n \mapsto \mathbb{R}$, and given a domain D, we wish to find $\text{argmin}_{x \in D} f(x)$.

Then, we can start with some feasible point x, and look in a small enough ball around x (suppose the ball has radius η, and denote it by $B(x, \eta)$, and move to a point $y \in B(x, \eta) \cap D$ such that $f(y) < f(x)$. At first sight, it is not clear how to find such a y. But by standard calculus, assuming f is differentiable at x, we know that for small enough η and some vector z, we have $f(x + \eta z) \approx f(x) + \eta \langle \nabla f_{|x}, z \rangle$. Thus if we set z to be the unit vector in the direction of $-\nabla f_{|x}$, then up to a second order correction, the value of f drops (i.e., becomes $f(x) - \eta \|\nabla f_{|x}\|^2$).

This general idea is known as **gradient descent** and it is perhaps the most ubiquitous algorithm in modern machine learning. Note that there are many logistical issues here: what if going along ∇f takes us out of D? How should we choose the "step size" η, and so on. This leads us into the rabbit hole of optimization, which we do not propose to get into in this course.

Zooming out slightly, we still have not answered the question of when local search is optimal. From the above reasoning, we obtain that gradient descent always converges to a *locally optimum* solution (i.e., a solution in which x is the best point in $B(x, \eta)$). Does this mean that there are no points with a smaller optimum value?

In general, there can be multiple local optima that are considerably worse than global optima (in fact, one can even consider polynomial minimization on the real line to see examples), e.g., consider the function $f(x) = x^4 - 2x^2 + x + 3$. The plot looks as follows:



We have a local minimum at $x \approx 1$, where the function value is considerably larger than the global minimum (at $x \approx -1$.

Convex optimization. That said, there is an important sub-class of optimization problems for which any local minimum is also the global minimum. This is the class of *convex optimization* problems, defined as the problem of minimizing a "convex function" over a "convex domain" D.

Let us define the terms in parentheses. For univariate functions, convexity is often defined via the second derivative. However, there is a much nicer definition which does not even assume the differentiability of f everywhere. We say that a function f is convex if for all $0 \leq \lambda \leq 1$ and for every $x, y \in D$, we have $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$.

For intuition, if we set $\lambda = 1/2$, this is saying that $f((x+y)/2) \leq \frac{f(x)+f(y)}{2}$. The definition is clearly well defined for multi-variate functions as well. The only issue can be that $x, y \in D$ but $\lambda x + (1-\lambda)y \notin D$. This is ruled out by the assumption that D is a convex set. Indeed, a convex set is precisely a set D such that for all $x, y \in D$, the point $\lambda x + (1-\lambda) y \in D$ for all $\lambda \in [0,1]$.

Local and global minima of a convex function. Why is every local minimum a global minimum in convex optimization? Let us prove it by assuming the contradiction. Let x be a local minimum -- i.e.,

(++) for some $\eta > 0$, we have that $f(x) \leq f(y)$ for all $y \in B(x, \eta)$.

Furthermore, suppose that z is the global minimum, and that $f(z) < f(x)$ (strictly smaller value). Now, if $z \in B(x, \eta)$, this is clearly a contradiction to (++). Otherwise, consider moving from x to z along the straight line, and consider the first time we exit the ball $B(x, \eta)$. This point, like all points on the segment xz, can be expressed as $\lambda x + (1-\lambda) z$, for some $1 > \lambda > 0$. By convexity, we have $f(\lambda x + (1-\lambda) z) \leq \lambda f(x) + (1-\lambda) f(z) < \lambda f(x) + (1-\lambda) f(x) = f(x)$.

Thus we have demonstrated a point in $B(x, \eta)$ whose $f()$ value is strictly smaller than $f(x)$ -- this contradicts (++), and completes the proof.

Matching problem -- a combinatorial example

Let us now illustrate local search in a context of the so-called "matching" problem (in which the optimization is over a discrete set).

Matching problem. Suppose we have n children and n gifts. Child i has a *happiness value* V_{ij} if he/she is given gift j, and suppose the V_{ij} values are all given (and are ≥ 0).The goal is to come up with an assignment of the gifts to children (each child must get precisely one gift), so as to maximize the *total happiness*. If child i is given gift $\sigma(i)$, then the objective is to maximize $\sum_{i=1}^{n} V_{i,\sigma(i)}$.

The brute force solution of checking all possibilities takes time $n!$, because every assignment σ (as above) is a permutation, and there are $n!$ permutations.

Greedy algorithm. Another natural procedure is the greedy algorithm: consider the children in some order, say $1, 2, \dots, n$. To child i, we give the gift with the largest V_{ij} value that is still available. This procedure can, in general, be really bad. Suppose we have $n = 2$ and we have the values $V_{11} = 1, V_{12} = 2, V_{21} = 1, V_{22} = 100$. In this case, starting with child 1 is really bad, because we end up assigning gift 2 to child 1, thereby getting a total happiness value of only 3 (while the optimum is 101).

However, in this case, we can see easily that swapping the gifts really improves the solution. This suggests the following *local search* procedure.

Local search. Start with any feasible assignment. Go over all pairs i, j of children, and see if swapping the gifts results in a higher value of the total happiness. If so, perform the swap, and repeat.

Running time. In principle, this algorithm can take a really long time. The running time is $n^2 \times$ (number of times we repeat the process). The latter is not easy to bound, but it is always $< n!$, because the cost of solution strictly improves, and there are only $n!$ solutions. But let us not worry too much about the running time for now.

Does the algorithm always find an optimal solution? It turns out that the answer is NO. Consider the following instance, with $n = 3$.

Suppose the black edges have weight 1 and the blue edges are weight 2. Suppose any non-edges (say between child 2 on the left and gift 1 on the right) are of weight 0. Then, it is easy to see that the set of black edges form a locally optimal solution. There is no single swap that improves the value of the solution, however the objective value of the solution is 3, while the optimum happines is 6 (obtained by picking the blue edges).

Interestingly, we can prove that this is essentially the worst possible *gap*.

Theorem. Let σ be any locally optimal assignment, and let γ be the optimal solution (i.e., the best assignment to child i is $\gamma(i)$). Then the total happiness value of the assignment σ is at least $(1/2)$ the total happiness of γ.

Proof. The intuitive idea behind the proof is the following. Suppose we have some V_{ij} that is really large (as in the example in which greedy fails). Then in the assignment σ, there must be two children the *sum* of whose happiness values is $\geq V_{ij}$. To see this, suppose ℓ is the index such that $\sigma(\ell) = j$ (in other words, $\ell = \sigma^{-1}(j)$). If $\ell = i$, there is nothing to prove, since child i has happiness value V_{ij}. Otherwise, we claim that $V_{i,\sigma(i)} + V_{\ell,j} \geq V_{ij}$. This is because otherwise, we can swap the gifts, and get a value of $V_{ij} + V_{\ell,\sigma(i)} \geq V_{ij}$. (As all the happiness values are non-negative.)

The argument above says that the optimum solution does not "badly miss" any high-weight edges. We use this idea to now complete the proof.

For any edge $i, \gamma(i)$ in the optimal solution, we must have $V_{i,\sigma(i)} + V_{\sigma^{-1}(\gamma(i)),\gamma(i)} \geq V_{i,\gamma(i)}$.

Now, suppose we sum this quantity over all i. We get

$$\sum_i V_{i,\sigma(i)} + \sum_i V_{\sigma^{-1}(\gamma(i)),\gamma(i)} \geq \sum_i V_{i,\gamma(i)}.$$

The first term on the LHS is the total happiness value of the solution σ, and the RHS is the total happiness value of the solution γ. Now, what is the second term on the LHS? As i goes over $1, 2, \dots, n$, $\gamma(i)$ also goes over $1, 2, \dots, n$ (it is a permutation). Thus we can re-write it as $\sum_{j=1}^n V_{\sigma^{-1}(j),j}$. But this is precisely the same as $\sum_i V_{i,\sigma(i)}$, and thus the LHS is simply $2 \sum_i V_{i,\sigma(i)}$.

Dividing by 2, we obtain the theorem.

Comparing with the continuous problems. When we defined local search for continuous optimization problems, we considered a ball of a certain radius around the given points. In the discrete problem above, the "ball" around one solution σ can be viewed as the set of all permutations that can be obtained by swapping two elements of σ.

In fact, by considering all possible re-assignments of *triples* of vertices, one can prove a theorem similar to the above, but with a better constant --- 2/3 instead of 1/2. Intuitively, this corresponds to searching in a "larger ball" around the given solution σ. The search now takes longer --- n^3 instead of n^2, but we have a better guarantee on the solution obtained.

Comments on the matching problem. While local search methods give a reasonable solution for the matching problem, it turns out that there are other techniques that give polynomial time algorithms that find the *optimal* matching. One classic algorithm here, known as the "four Hungarians" procedure, turns out to be simple and optimal. It's related to algorithms for *max flow*, that we study briefly in the course.