Today we will continue our discussion about local search. We will also discuss one of the most popular local search algorithms (used everywhere in machine learning) known as gradient descent. We then started talking about graph algorithms, specifically the shortest path problem.

Disclaimer: These lecture notes are informal in nature and are not thoroughly proofread. In case you find a serious error, please send email to the instructor pointing it out.

Local search

All the paradigms we have seen so far — divide and conquer, dynamic programming and greedy — give ways to build a solution to some combinatorial/optimization problem. We have seen examples and limitations of each paradigm. We now see a rather different approach: can we start with some solution to a problem, and constantly “improve” it? This is the main idea behind local search. Let us illustrate with an example we have seen before, the matching problem.

Recap: matching problem. We have $n$ children and $n$ gifts, and the goal is to assign precisely one gift to each child, so as to maximize the total happiness. Our input consists of happiness values, i.e., child $i$’s happiness if he/she is given the gift $j$. We denoted this by $H_{i,j}$.

Now, given any solution to this problem, i.e., any assignment of gifts to children, how can we try to improve it? A natural idea is to see if for a pair of children, swapping their gifts would improve their total happiness. In other words, if child $i$ is currently assigned the gift $p_i$ and child $j$ is assigned the gift $p_j$, we see if

$$H_{i,p_i} + H_{j,p_j} > H_{i,p_j} + H_{j,p_i}.$$ (1)

If so, we swap the gifts, and this yields us a solution of strictly larger value of the total happiness. Suppose we now keep repeating this procedure until no such swap results in an increased value. Such an algorithm is called a local search procedure. Abstractly, it starts with one solution and it repeatedly checks if it can be improved by performing some simple “local” operations (in this case, by swapping the gifts of two children). The procedure ends if no such improvement is possible.

We can ask if local search always ends up finding the optimum assignment. It turns out that this is not the case (because we are only seeing if pairwise swaps improve the solution; it could be that to get to the optimal solution, we need to change a whole bunch of assignments). It’s an exercise (see HW 2) to construct an explicit example in which the local search algorithm does not always (i.e., for some starting solution) return the optimum solution.

Not optimal, but not too bad

For the matching problem, we prove that the simple local search procedure still does something interesting:

**Theorem 1.** Suppose we run the local search algorithm described formally below, and end up with an assignment of gift $p_i$ to child $i$, for all $i \in [n]$. Suppose the cost of the optimal solution is denoted as $\text{opt}$. Then we have

$$\sum_i H_{i,p_i} \geq \frac{\text{opt}}{2}.$$ 

$[n]$ is standard notation for the set $\{1, 2, \ldots, n\}$. 

Local search continued on next page...
Remark. Note that we have no idea about the optimal solution— it could be equal to the assignment \( p \), or it could be very far. In any case, the theorem says that the cost cannot be too far from the locally optimal solution \( p \).

Remark 2. There could in general be many locally optimal solutions (corresponding to different initial assignments). The global optimum is always a local optimum as well (trivially so, because if there’s a swap that improves it, it cannot be the global optimum).

Formal description of the algorithm. For completeness, let us recap the algorithm described above:

1. Start with any assignment \( p \) (i.e., child \( i \) gets gift \( p_i \)).
2. Check if there exist indices \( i, j \) such that (1) holds. If so, swap \( p_i \) and \( p_j \).
3. Repeat step 2 until there are no such \( i, j \).

Note. The algorithm always terminates, because the cost of the solution is strictly decreasing (and there are only finitely many possible assignments). The running time of the procedure is somewhat tricky to bound. If done in a bad order, the swaps could end up taking a lot of steps. However, we do not care about this for now.

Let us now prove the theorem.

Proof of Theorem 1. Let \( q \) denote the optimal solution. I.e., in the optimal solution child \( i \) is given the gift \( q_i \). Thus what we want to show is that

\[
\sum_i H_{i,p} \geq \frac{1}{2} \sum_i H_{i,q}.
\]

A rough intuition about the proof is the following: when can the solution \( p \) be really bad compared to \( q \)? This could happen if there is an edge \( i, q_i \) that has a much larger happiness value compared to \( i, p_i \). But in this case, because \( p \) also assigned every gift to some child, the gift \( q_i \) must have been assigned to some child, say \( j \). (I.e., there is some child \( j \) such that \( p_j = q_i \).) Because the solution is locally optimum, we can see that \( H_{j,q_i} \) must be quite large (because otherwise, swapping the gifts for children \( i, j \) would result in a larger value, and so \( p \) would not be locally optimal.)

Formally, local optimality implies that we have \( H_{i,p} + H_{j,p} \geq H_{i,p} + H_{j,p} \). Using \( p_j = q_i \) and the fact that each happiness value is \( \geq 0 \), this implies that

\[
H_{i,p} + H_{j,q_i} \geq H_{i,q_i} + H_{j,p} \geq H_{i,q_i}.
\]

Now, let us sum over \( i \). The only question is, what is the value of \( \sum_i H_{j,q_i} \)? This is a sum in which for each \( q_i \), we are looking at the \( j \) such that \( p_j = q_i \) and summing over these happiness values. As the \( q_i \) is simply a permutation of all the gifts, this summation is precisely a sum over all \( j \) of \( H_{j,p_j} \), which is simply the cost of the locally optimum solution.

Using this fact and dividing by two, we obtain (2), which completes the proof of the theorem. \( \square \)

In our HWs we will see that if we consider more complicated swaps, specifically if we consider every triple of children and see if reassigning their gifts improves the solution, then we can guarantee that the locally optimum solution is at least a factor \( 2/3 \) times the optimum. But note that in this case the time to find a potential improvement is roughly \( n^3 \) (we need to check all triples). Thus the search step is more expensive, while the solution is guaranteed to be better. This is a common tradeoff in local search. (We will see in a few lectures that the happiness problem can indeed be solved exactly in polynomial time.)
Gradient descent

The next, and perhaps most common example of local search is the so-called \textit{gradient descent} algorithm. The general problem here is the following:

\begin{center}
\textbf{Gradient descent.} Given a convex function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ defined over a convex domain $D$, find the $x \in D$ that minimizes $f$.
\end{center}

Note that we are talking about real-valued functions whose domain is a subset of $\mathbb{R}^n$. The domain itself is a convex set (i.e., for any $x, y \in D$, the entire line segment $xy$ is contained in $D$). A function is called convex if for all $x, y$ in the domain, we have

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y) \quad \text{for all } \lambda \in (0, 1).$$

Plugging in $\lambda = 1/2$, we get $f((x + y)/2) \leq [f(x) + f(y)]/2$. This is an equivalent definition of convexity for continuous functions. For single variable functions, i.e., when the dimension $n = 1$, you may have seen the definition $f''(x) \geq 0$. This turns out to be equivalent to the definition above, when the function is twice differentiable.

As this is a pretty vast topic, I will skip a detailed discussion of gradient descent. A good starting point are my notes from another class: http://www.cs.utah.edu/~bhaskara/courses/theoryml/scribes/lecture7.pdf

You can also find plenty of material online about gradient descent. The treatment in a book of Shai Shalev-Schwartz and Shai Ben-David (available at http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/understanding-machine-learning-theory-algorithms.pdf) is very thorough.