CS124	Lecture 20

Approximation algorithms for NP-complete problems are often great when they exist, but sometimes the requirement of a guarantee on the approximation ratio between our solution and the actual optimal solution (a guarantee valid for every input) is too restrictive: there may be no practical approximation algorithm with a practically useful approximation ratio.

What Can We Do?

Fortunately, there is a great deal we can do. Here are just a few possibilities:

- Restrict the inputs.

  NP-completeness refers to the worst case inputs for a problem. Often, inputs are not as bad as those that arise in **NP**-completeness proofs. For example, although the general SAT problem is hard, we have seen that the cases of 2SAT and Horn formulae have simple polynomial times algorithms.

- Develop heuristics.

  Sometimes we might not be able to make absolute guarantees, but we can develop algorithms that seem to work well in practice, and have arguments suggesting why they should work well. For example, the simplex algorithm for linear programming is exponential in the worst case, but in practice it's generally the right tool for solving linear programming problems.

- Use randomness.

  So far, all our algorithms have been *deterministic*; they always run the same way on the same input. Perhaps if we let our algorithm do some things randomly, we can avoid the **NP**-completeness problem?

  Actually, the question of whether one can use randomness to solve an **NP**-complete problem is still open, though it appears unlikely. (As is, of course, the problem of whether one can solve an **NP**-complete problem in polynomial time!) However, randomness proves a useful tool when we try to come up with approximation algorithms and heuristics. Also, if one can assume the input comes from a suitable "random distribution", then often one can develop an algorithm that works well on average.

To begin, we will look at heuristic methods. The amount we can prove about these methods is (as yet) very

limited. However, these techniques have had some success in practice, and there are arguments in favor of why they are reasonable thing to try for some problems.

Local Search

"Local search" is meant to represent a large class of similar techniques that can be used to find a good solution for a problem. The idea is to think of the solution space as being represented by an undirected graph. That is, each possible solution is a node in the graph. An edge in the graph represents a possible move we can make between solutions.

For example, consider the Number Partition problem for the homework assignment. Each possible solution, or division of the set of numbers into two groups, would be a vertex in the graph of all possible solutions. For our possible moves, we could move between solutions by changing the sign associated with a number. So in this case, our graph of all possible solutions, we have an edge between any two possible solutions that differ in only one sign. Of course this graph of all possible solutions is huge; there are 2^n possible solutions when there are n numbers in the original problem! We could never hope to even write this graph down. The idea of local search is that we never actually try to write the whole graph down; we just move from one possible solution to a "nearby" possible solution, either for as long as we like, or until we happen to find an optimal solution.

To set up a local search algorithm, we need to have the following:

1. A set of possible solutions, which will be the vertices in our local search graph.
2. A notion of what the *neighbors* of each vertex in the graph are. For each vertex x, we will call the set of adjacent vertices $N(x)$. The neighbors must satisfy several properties: $N(x)$ must be easy to compute from x (since if we try to move from x we will need to compute the neighbors), if $y \in N(x)$ then $x \in N(y)$ (so it makes sense to represent neighbors as undirected edges), and $N(x)$ cannot be too big, or more than polynomial in the input size (so that the neighbors of a node are easy to search through).
3. A cost function, from possible solutions to the real numbers.

The most basic local search algorithm (say to minimize the cost function) is easily described:

1. Pick a starting point x.
2. While there is a neighbor y of x with $f(y) < f(x)$, move to it; that is, set x to y and continue.

3. Return the final solution.

The Reasoning Behind Local Search

The idea behind local search is clear; if keep getting better and better solutions, we should end up with a good one. Pictorially, if we "project" the state space down to a two dimensional graph, we are hoping that the picture has a sink, or *global optimum*, and that we will quickly move toward it. See Figure 20.1.

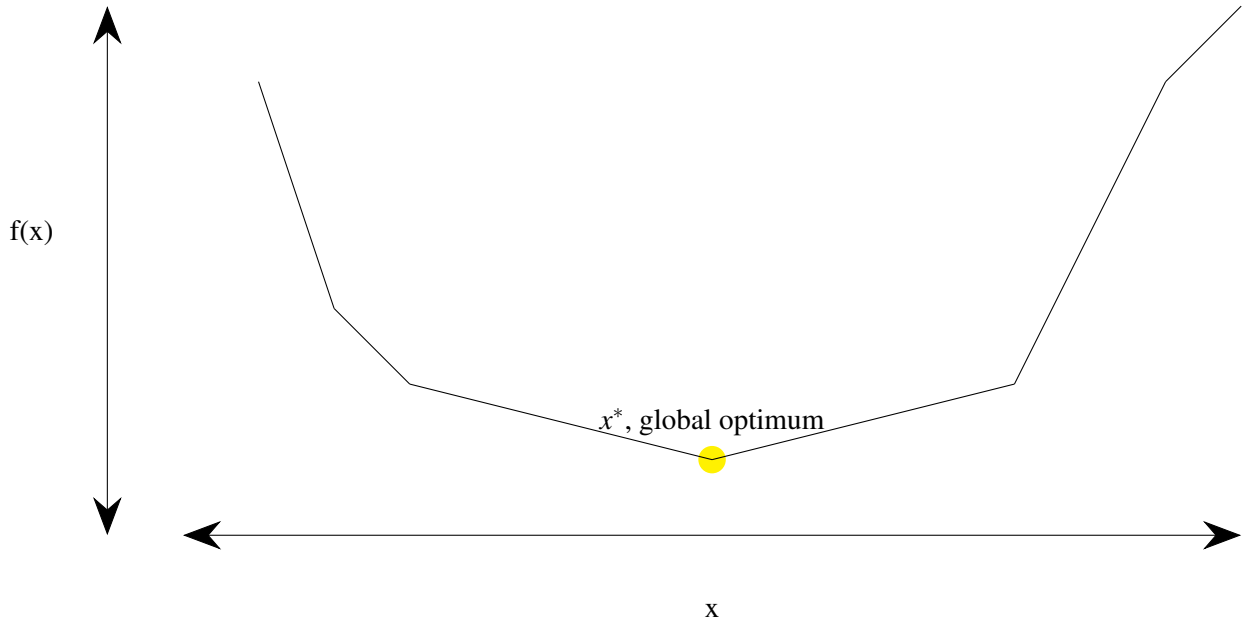

Figure 20.1: A very nice state space.

There are two possible problems with this line of thinking. First, even if the space does look this way, we might not move quickly enough toward the right solution. For example, for the number partition problem from the homework, it might be that each move improves our solution, but only by improving the residue by 1 each time. If we start with a bad solution, it will take a lot of moves to reach the minimum. Generally, however, this is not much of a problem, as long as the cost function is reasonably simple.

The more important problem is that the solution space might not look this way at all. For example, our cost function might not change smoothly when we move from a state to it neighbor. Also, it may be that there are several *local optima*, in which case our local search algorithm will hone in on a local optimum and get stuck. See Figure 20.2.

This second problem, that the solution space might not "look nice", is crucial, and it underscores *the importance of setting up the problem*. When we choose the possible moves between solutions – that is, when we construct the

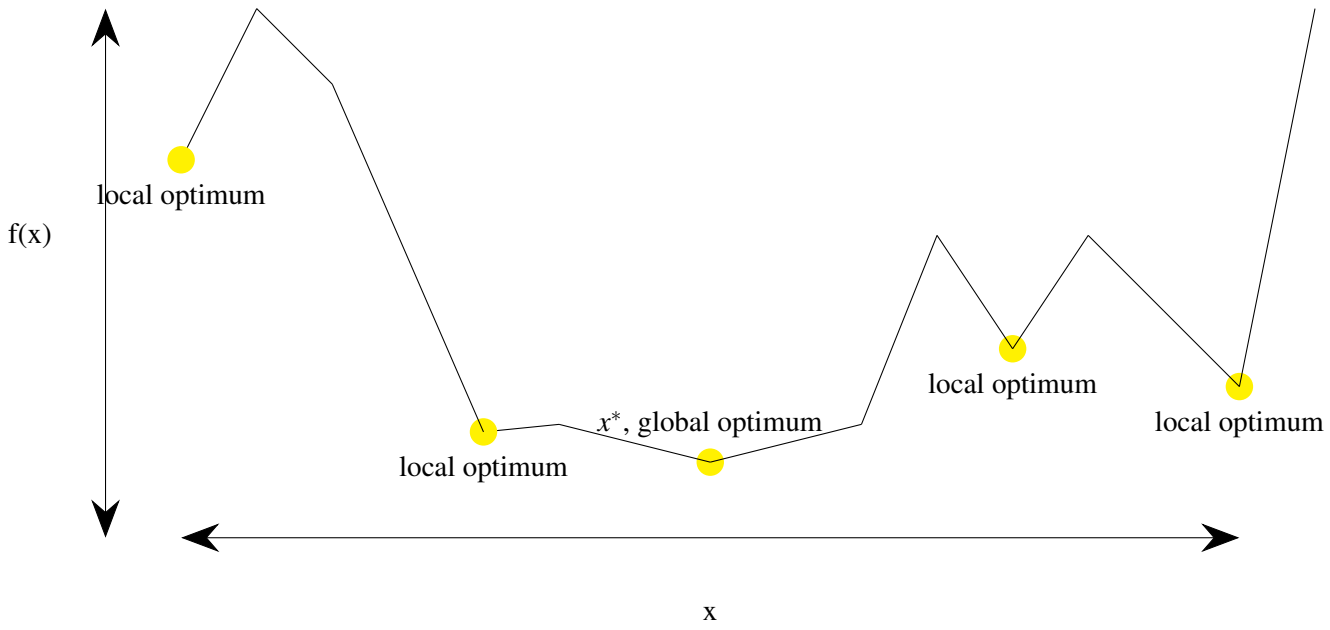

Figure 20.2: A state space with many local optima; it will be hard to find the best solution.

mapping that gives us the neighborhood of each node– we are setting up how local search will behave, including how the cost function will change between neighbors, and how many local optima there are. How well local search will work depends tremendously on how smart one is in setting up the right neighborhoods, so that the solution space really does look the way we would like it to.

Examples of Neighborhoods

We have already seen an example of a neighborhood for the homework problem. Here are possible neighborhoods for other problems:

- MAX3SAT: A possible neighborhood structure is two truth assignments are neighbors if they differ in only one variables. A more extensive neighborhood could make two truth assignments neighbors if they differ in at most two variables; this trades increased flexibility for increase size in the neighborhood.
- Traveling Salesperson: The k-opt neighborhood of x is given by all tours that differ in at most k edges from x. In practice, using the 3-opt neighborhood seems to perform better than the 2-opt neighborhood, and using 4-opt or larger increases the neighborhood size to a point where it is inefficient.

Lots of Room to Experiment

There are several aspects of local search algorithms that we can vary, and all can have an impact on performance. For example:

1. What are the neighborhoods $N(x)$?
2. How do we choose an initial starting point?
3. How do we choose a neighbor y to move to? (Do we take the first one we find, a random neighbor that improves f, the neighbor that improves f the most, or do we use other criteria?)
4. What if there are ties?

There are other practical considerations to keep in mind. Can we re-run the algorithm several times? Can we try several of the algorithms on different machines? Issues like these can have a big impact on actual performance. However, perhaps the most important issue is to think of the right neighborhood structure to begin with; if this is right, then other issues are generally secondary, and if this is wrong, you are likely to fail no matter what you do.

Local Search Variations

There are many variations on the local search technique (below, assume the goal is to minimize the cost function):

- Hill-climbing – this is the name for the basic variation, where one moves to a vertex of lower (or possibly equal) cost.
- Metropolis rule – pick a random neighbor, and if the cost is lower, move there. If the cost is higher, move there with some probability (that is usually set to depend on the cost differential). The idea is that possibly moving to a worse state helps avoid getting trapped at local minima.
- Simulated annealing – this method is similar to the Metropolis rule, except that the probability of going to a higher cost neighbor varies with time. This is analogous to a physical system (such as a chemical polymer) being cooled down over time.
- Tabu search – this adds some memory to hill climbing. Like with the Metropolis rule and simulated annealing, you can go to worse solutions. A penalty function is added to the cost function to try to prevent cycling and promote searching new areas of the search space.

- Parallel search ("go with the winners")– do multiple searches in parallel, occasionally killing off searches that appear less successful and replacing them with copies of searches that appear to be doing better.
- Genetic algorithms – this trendy area is actually quite related to local search. An important difference is that instead instead of keeping one solution at a time, a group of them (called a population) is kept, and the population changes at each step.

It is still quite unclear what exactly each of these techniques adds to the pot. For example, some people swear that genetic algorithms lead to better solutions more quickly than other methods, while others claim that by choosing the right neighborhood function one can do as well with hill climbing. In the years to come, hopefully more will become understood about all of these methods.