3. Traditional Methods — Part 1

... are all alike in their promises.  
It is only in their deeds that they differ.  

Molière, *The Miser*

There are many classic algorithms that are designed to search spaces for an optimum solution. In fact, there are so many algorithms that it’s natural to wonder why there’s such a plethora of choices. The sad answer is that none of these traditional methods is robust. Every time the problem changes you have to change the algorithm. This is one of the primary shortcomings of the well-established optimization techniques. There’s a method for every problem — the problem is that most people only know one method, or maybe a few. So they often get stuck using the wrong tool to attack their problems and consequently generate poor results.

The classic methods of optimization can be very effective when appropriate to the task at hand. It pays for you to know when and when not to use each one. Broadly, they fall into two disjoint classes:

- Algorithms that only evaluate complete solutions.
- Algorithms that require the evaluation of partially constructed or approximate solutions.

This is an important difference. Whenever an algorithm treats complete solutions, you can stop it at any time and you’ll always have at least one potential answer that you can try. In contrast, if you interrupt an algorithm that works with partial solutions, you might not be able to use any of its results at all.

Complete solutions mean just that: all of the decision variables are specified. For example, binary strings of length $n$ constitute complete solutions for an $n$-variable SAT. Permutations of $n$ cities constitute complete solutions for a TSP. Vectors of $n$ floating-point numbers constitute complete solutions for an NLP. We can easily compare two complete solutions using an evaluation function. Many algorithms rely on such comparisons, manipulating one single complete solution at a time. When a new solution is found that has a better evaluation than the previous best solution, it replaces that prior solution. Examples include exhaustive search, local search, hill climbing, and gradient-based numerical optimization methods. Some modern heuristic methods such as simulated annealing, tabu search, and evolutionary algorithms also fall into this class as well (see chapters 5 and 6).
Partial solutions, on the other hand, come in two forms: (1) an incomplete solution to the problem originally posed, and (2) a complete solution to a reduced (i.e., simpler) problem.

Incomplete solutions reside in a subset of the original problem's search space. For example, in an SAT, we might consider all of the binary strings where the first two variables were assigned the value 1 (i.e., TRUE). Or in the TSP, we might consider every permutation of cities that contains the sequence 7 – 11 – 2 – 16. Similarly, for an NLP, we might consider all of the solutions that have \( x_3 = 12.0129972 \). In each case, we fix our attention on a subset of the search space that has a particular property. Hopefully, that property is also shared by the real solution!

Alternatively, we can often decompose the original problem into a set of smaller and simpler problems. The hope is that in solving each of these easier problems, we can eventually combine the partial solutions to get an answer for the original problem. In a TSP, we might only consider \( k \) out of \( n \) cities and try to establish the shortest path from city \( i \) to city \( j \) that passes through all \( k \) of these cities (see section 4.3). Or, in the NLP, we might limit the domains of some variables \( x_i \), thereby reducing the size of the search space significantly, and then search for a complete solution within that restricted domain. Such partial solutions can sometimes serve as building blocks for the solution to the original problem.

Making a complex problem simpler by dividing it up into little pieces that are easy to manage sounds enticing. But algorithms that work on partial solutions pose additional difficulties. You have to: (1) devise a way to organize the subspaces so that they can be searched efficiently, and (2) create a new evaluation function that can assess the quality of partial solutions. This latter task is particularly challenging in real-world problems: what is the value of a remote control channel changer? The answer depends on whether or not there is a television to go along with it! The utility of the remote control device depends heavily on the presence or absence of other items (e.g., a television, batteries, perhaps a cable connection). But when you evaluate a partial solution you intrinsically cannot know what the other external conditions will be like. At best, you can try to estimate them, but you could always be wrong.

The former task of organizing the search space into subsets that can be searched efficiently is also formidable, but many useful ideas have been offered. You might organize the search space into a tree, where complete solutions reside at the leaves of the tree. You could then progress down each branch in the tree, constructing the complete solution in a series of steps. You might be able to eliminate many branches where you know the final solutions will be poor. And there are other ways to accomplish this sort of "branch and bound" procedure. Each depends on how you arrange the solutions in the search space, and that in turn depends on your representation.

Remember, there's never only one way to represent different solutions. We've seen three common representations for the SAT, TSP, and NLP: (1) binary strings, (2) permutations, and (3) floating-point vectors, respectively. But we