Computational Solution Techniques for Mathematical Programming Problems

A Brief Survey of Problems and Exact and Heuristic Solution Methods

Nick Bennett

Basic Definitions

- *Optimization*, or *mathematical programming*, is the study and practice of seeking, in a systematic way, the maximum or minimum values of a function (the *objective function*), and the values of the *decision variables* (the inputs to a given function) where the maximum or minimum objecting function values are found. Complicating this in practice is the fact that decision variables are often subject to constraints – and the nature of the objective function may make it difficult to deal with analytically.
- An *algorithm* is a step-by-step problem solving procedure. Algorithms can range from very simple (e.g. cooking recipes, driving directions between two locations on a map, the Sieve of Eratosthenes for enumerating prime numbers) to moderately complicated (e.g. Runge-Kutta methods of numerical integration) to very complicated (e.g. algorithms to maximize potential profits from oil-drilling activities).

Basic Definitions (cont.)

- For our purposes, an *exact algorithm* is one which is mathematically guaranteed (under some stated conditions) to arrive at the solution to a particular type of problem. For some problems, the known exact algorithms are computationally very efficient; for others, all known exact algorithms are computationally expensive – prohibitively so, in some cases.
- A *heuristic algorithm* is one which ignores the question of whether a solution found by the algorithm can be mathematically proven correct. Instead, a heuristic generally incorporates certain rules of thumb, or experience-based rules – which may or may not lead to a solution to a particular problem. In most cases, heuristics require fewer mathematical and logical operations than exact algorithms – though there is no guarantee that the heuristic will arrive at the same solution.

Simple Mathematical Optimization

When there are no constraints on the decision variables, and the objective function is continuously differentiable (i.e. a finite slope exists at every point where the objective function is defined), we can see that if an optimum exists, it must be at a point where the slope is zero – i.e. where no change would lead to an improvement in the objective function. This is true even for multivariate objective functions.

Further, if we have simple constraints on a small number of decision variables (e.g. one is constrained to be within a range, independent of the others), then this approach is still useful.

If we find all the points where the slope of the objective function is zero, then we have found the set of unconstrained candidate points. We exclude from this set those that violate the constraints, and add to it the limit points of the constraints. We now examine this set to see which are true extremum, and which of those is the maximum or minimum (depending on what we want).

Linear Programming via the Simplex Method

When the objective function is linear, all constraints are linear, and the decision variables are continuous, we have a linear programming problem (LP). In 1947, George Dantzig invented the simplex algorithm to solve linear programming problems. In very general terms, this algorithm is as follows:

- Set up the problem as a system of linear equations with more variables than equations (i.e. there may be an infinite number of solutions to this system).
- By a series of pivoting operations, move from one feasible solution to another, where each solution corresponds to an improvement in the original objective function.
- When the objective function cannot be improved further, we have found the optimum.

This process can be viewed as movement from vertex to vertex of an n-dimensional polytope, with each move improving the objective function.

Specialized Linear Programming Variations

Some optimization problems have characteristics that may make them less suited to the simplex method, and/or more suited to others. Examples include:

- The transportation problem: Given n sources, each with a supply s_n of some commodity; m destinations, each with a demand d_m for that commodity; a cost c_{ij} for transporting one unit of the commodity from source i to destination j; find the transportation scheme that satisfies the demand at minimum cost. Solution techniques for this problem exploit the network nature of the problem to solve it more efficiently.
- The assignment problem: Given n resources, and m tasks (where $m \geq n$), where there is a cost c_{ij} associated with assigning resource i to task j, and where each resource must be assigned to exactly one task, and each task may be assigned a maximum of one resource, find the set of assignments that minimizes the total cost. This can be seen as a special case of the transportation problem. However, the additional restrictions lead to more specialized and effective solution techniques; the most well-known of these is called the *Hungarian method*.

Linear Point Solution Techniques

While the simplex method and its variants are usually very good, and are still widely used, there are close-to-worst-case scenarios, with very complicated problems, where the number of steps required to setup and solve the problem increases as an exponential function of the number of variables and constraints. Because of this – and because these very large, very complicated problems are becoming more common – different solution techniques have been developed. The first truly viable alternative to the simplex method for these problems was Karmarkar's algorithm, invented in 1984.

In contrast to the simplex method, which can be viewed as a process of moving from vertex to vertex of a polytope, Karmarkar's algorithm can be viewed as moving through the interior of the polytope, in a relatively small number of steps. For this reason, we refer to this and similar methods as *interior point methods*.

In general, the computational cost of putting LPs into a form that can be solved via interior point methods is very high, but the number of iterations required for solution is low. This trade-off makes these techniques well suited to certain types of very large problems, but poorly suited to others.

Difficult Problems & Heuristic Algorithms

There are many problems for which an exact solution technique is simply too computationally expensive. For these problems, we often resort to heuristic algorithms which give "good enough" solutions, at a much lower computational cost.

One example of this is the traveling salesman problem (TSP): Given n cities, with a distance d_{ij} between cities i and j, find the route through all the cities that minimizes the total distance, and returns to the starting point. This is recognized as a difficult problem, and exact solution techniques are very expensive: once the number of cities moves into the thousands (not that uncommon, when we start seeing such parallels as those between points on a circuit board and cities on a map), the processing time required for exact solution rises into years of CPU time.

One of the best heuristics for the TSP is the Lin-Kernighan algorithm, introduced in 1973. This algorithm works by adaptively swapping pairs of sub-tours to formulate an improved tour at each step.

Another interesting heuristic that has been applied to TSP and similarly difficult problems in the last 15 years is *ant colony optimization* (ACO), which uses many agents (the ants) to traverse a solution space, with pheremone-type signals employed to communicate experience between agents.

Difficult Problems & Heuristic Algorithms (cont.)

Another surprisingly challenging problem to which a variety of heuristic methods have been applied is the *knapsack problem*: Given a set of items, each with a weight and value, find the maximum-value combination of items that together does not exceed a stated weight limit.

The knapsack problem, like the traveling salesman problem, is a *combinatorial optimization problem*, where the solution space is a set of discrete combinations of the inputs – usually growing exponentially larger as the number of inputs increases. For some combinatorial optimization problems (e.g. finding the minimum spanning tree of a network), there are very efficient exact methods for solving even very large problems; for others, like the knapsack problem, a heuristic algorithm is often the best choice.

The binary decision variable nature of the 0-1 knapsack problem ($x_i = 1$ if the i^{th} item is selected, and 0 otherwise) makes it a frequently used illustration of the *genetic algorithm* heuristic technique – even in cases where other heuristics might give a better and/or more efficient solution.

(Of the generally difficult type of problem known *as NP-complete*, the knapsack problem is actually considered one of the easier problems to solve. Efficient methods exist which give acceptable solutions even for very large knapsack problems.)

Simulated Annealing

Simulated annealing (SA) is a heuristic algorithm inspired by the process of annealing in metallurgy. In that context, annealing consists of systematically heating and cooling some material, to increase the size of its crystals – removing impurities or defects, and increasing its strength. Over time, the material is cooled more, with smaller and smaller jumps upward in temperature at each step. With each increase in temperature, some of the atoms of the material may move slightly from their current position; with the successive cooling phase allowing the atoms to settle in a more stable configuration.

SA emulates this strategy by starting at some feasible (but presumably non-optimal) solution point, and moving to a nearby point at each step. This next point is chosen probabilistically, with the likelihood depending on the difference in the objective function values at the two points, and on a global "temperature". Initially, the temperature is high, and movement to a new solution point very random, so that the objective function may often worsen. But as the algorithm proceeds, and the temperature is lowered, the selection of the next solution point becomes more and more biased toward those points that improve the objective function.

Genetic Algorithms

A genetic algorithm (GA) is a heuristic that emulates some of the mechanisms of organic evolution, in an attempt to arrive at a good solution to a problem. A GA starts with a large number of randomly-generated solutions to a problem; each solution is the "genome" of an individual member of the population. In each generation, we follow these steps:

- Each member of the population is evaluated to assess its "fitness".
- A subset of the population is chosen to survive to the next generation. This selection is partly random, but not arbitrary: those individuals with better fitness are more likely to be selected.
- To replace those individuals that will not survive, a subset of the population is chosen to reproduce. Again, this is a random selection, with the most fit being the most likely to be selected.
- The next generation is produced by combining the genomes from pairs of the individuals selected for reproduction. Each pair exchanges some part of the genome to produce a new pair.
- Optionally, an offspring's genomes may be mutated slightly in a random fashion.

Sample Knapsack Problem

- You have ten items, each with a weight as shown on the right.
- You have a knapsack which has a weight limit of 200 pounds; your job is to fill the knapsack with items.
- Working in groups, answer the following questions:
  - Which combination of items gives a total weight closest to 200 pounds, without going over?
  - What is the total weight of these items?
  - How much capacity was wasted?
  - How many possible combinations are there, regardless of total weight? (Remember: one possible combination would include none of the items at all.)
  - How much more difficult do you think the problem would be, if there were 20 items, instead of 10? What about 100 items?
  - What if each item had a value, and you wanted to maximize the total value, while staying within the weight limit?

Item	Weight
1	27
2	26
3	37
4	18
5	33
6	40
7	17
8	32
9	31
10	14

Genetic Algorithm for the Knapsack Problem

- We can represent each combination as a string of 0s and 1s, where a 1 means the corresponding item is included in the knapsack. Using the sample problem, 1010100110 would mean that items 1, 3, 5, 8, and 9 are to be included in the knapsack, for a total weight of (27 + 37 + 33 + 32 + 31) = 160 pounds.
- This string of 0s and 1s is the genome of the individual. To begin, we generate a population with random genomes.
- At each iteration, we compute the weight corresponding to the genome of each individual of the population. Those with the highest weights, which don't exceed 200, will have the highest chance of surviving and reproducing.
- Those selected for reproduction are paired up randomly, and they exchange some portion of their genome to create two new offspring.
- The process is repeated over several generations, with the general aim being to improve the average fitness with each generation.
- Eventually, we hope to arrive at an exact solution, or at least at a solution we consider good enough.

Java Implementation of GA & Knapsack Example

The "**`GA & Sample Knapsack.zip`**" archive includes a number of generic classes (in the **`org.supercomputingchallenge.ga.generic`** package) for solving problems via GA, as well as two classes (in the **`org.supercomputingchallenge.ga.knapsack`** package) which extend and use the generic classes to solve a specific knapsack problem. The knapsack-specific classes are these:

- **`Solution`** – Extends the generic **`Individual`** class with a **`BitSet`** to represent the items included in an individual solution; an array of weights and values to compute the fitness of the individual solution (see the **`getFitness`** method); a simple crossover and mutation scheme to generate offspring from parents (see the **`getOffspring`** method).
- **`SampleKnapsack`** – Serves as the program entry point, with the **`main`** method instantiating a population of **`Solution`** instances, along with a **`RouletteWheel`** instance and a **`Controller`** instance, to put evolutionary pressure on the population through multiple generations.

The files in the archive are packaged with a DrJava project file, "**`Knapsack.xml`**", but they can be used in any Java development environment, or compiled from the command line.