Optimization Theory (DS2) Lecture #14
Pareto Optimality, Randomness and Sex, and a Wrap Up of the Semester

July 19, 2017

Abstract

Last lecture! The last three technical topics are Pareto optimality, the importance of randomness in algorithms, and sex. Oh, and I slip in a few words about quantum computing!

Lest you think this semester has been a purely academic set of exercises, on Jan. 10, Arjun Narayan noted on Twitter that "lots of people get paid hundreds of thousands of dollars to do...linearization, which requires understanding all the maths." Want to work on Wall Street? What we've studied here is valuable stuff!

1 Our Last New Topics

1.1 Pareto Optimality

This is the point where I confess that much of what we've talked about all semester is just an approximation of reality!

Example: Which is more important in a hamburger, cheapness or deliciousness? If there is a hamburger that is *both* more expensive and less delicious than some other option that is available, will you eat it? (Answer: maybe, if it's closer and more convenient – now we have third axis in our space and on our Pareto front!)

Example 2: When selecting an apartment, how do you balance the distance to school and to someplace fun, such as your preferred karaoke joint?

The idea of having multiple points that can all be considered "optimal" because there are multiple aspects of the problem you care about is due to Vilfredo Damaso Pareto (1848–1923), who is sometimes credited with turning economics into a mathematical science. (He's most famous for the 80/20 rule, but that's not what we're talking about here.)

Pareto originally formulated what is now called *Pareto optimality* or *Pareto efficiency* in terms of optimizing over a set of people: if no individual can be made "better off" (remember, he's an economist) without making someone else "worse off," then your current solution or economic setup is Pareto optimal.

For two objective variables, such as "cheapness" and "deliciousness," as in our hamburger example, it's pretty easy to place every hamburger stand on a two-dimensional plot. A point which has no other point both above *and* to the right of it is Pareto optimal. The set of such points is a Pareto-efficient frontier.

bf n.b.: Pareto efficient has *nothing at all* to do with *fair*. Taking a pie and dividing it among three people by giving half to two people and nothing to the third is still Pareto efficient, because the third person's share can only be increased by *decreasing* the share of another person. A Pareto inefficient solution, on the other hand, would be to waste some of the pie. (Not all Pareto problems are zero-sum, but this example is.)

1.2 The Importance of Randomness: a Very Brief History

Livnat and Papadimitriou say, "One of the most central and striking themes of algorithms research in the past few decades has been the surprising power of randomization." But randomness has a long history; here are just a few of the points that occur to me when thinking about our understanding of its importance and the tension with the idea that everything can be predicted if we just know enough about the current state of things:

1. Democritus (c. 460 B.C. – c. 370 B.C.), who is generally credited with the idea of atoms, also believed that the universe is deterministic, according to Wikipedia and its sources. If we were arguing about quantum mechanics, his approach would be what we call a *hidden variable theory*.

2. Formal concepts of randomness, including what we call Pascal's Triangle, may have been developed in China at the time of the *I Ching*, 1150 B.C.

3. Francis Bacon (1561-1626): often called the Father of the Scientific Method. A forceful proponent of the idea that the universe can be explained by empirical experiment and analysis.

4. René Descartes (1596-1650): believed in the mechanical behavior of people.

5. I hope you're familiar with the basic idea of *Bayesian inference*. Originally developed by Thomas Bayes (1702–1761) and extended by Pierre-Simon Laplace (1749–1827), it's the idea that you can assign a probability to something that you only partially understand, then update that probability based on further additional knowledge (see below).

6. Pierre-Simon Laplace stated explicitly, in 1814, his belief that the universe is deterministic, a philosophical position known as *causal* or *physical* or *scientific determinism*, or by philosophers as *nomological* determinism. It states that the future is determined entirely by the past, with no possible deviation.

7. More generally, advances in physics during the 18th and 19th centuries were primarily about mechanics, electricity and magnetism, and optics; there was a strong belief that things were deterministic. The big exception was thermodynamics, where it was understood (e.g., Bernoulli, 1700-1782) that gases were composed of molecules bouncing randomly around.

8. Gregor Mendel (1822-1884): now revered as the father of modern genetics, though his work was fairly obscure in his own lifetime; recognized that inheritance includes random choice of characteristics.

9. Early 20th century: recognition of the importance of randomness in quantum mechanics. Erwin Schrödinger's (1887–1961) equation describes the evolution of a quantum system in terms of waves described by a complex amplitude; these amplitudes determine the phase and interference properties. Max Born (1882–1970) recognized that the square of the absolute value of the amplitude represents the probability of finding the system in that state, hence they are known as probability amplitudes.

10. 1936: Introduction of the Turing Machine (deterministic computing theory at its height).

11. 1948: Claude Shannon recognizes both the impact of random noise on the amount of information you can push through a channel, and that information involves the entropy of information, which is how hard it is to predict.

12. 1930s to 1950s: Beginnings of Monte Carlo (random) methods in algorithms and experiments; contributors include Enrico Fermi, Nicholas Metropolis (after whom an important form is named), Stanislaw Ulam, John von Neumann, Edward Teller, Augusta Teller, Arianna Rosenbluth, Marshall Rosenbluth, and others.

13. 1963-1972: Edward Lorenz discovers the Lorenz attractor and does various followup work, establishing the foundations of chaos theory, the idea that not everything can be predicted exactly: *sensitive dependence on initial conditions*. Not, strictly speaking, randomness, but the genesis of the idea that nonrandom doesn't mean fully predictable.

14. 1968 saw the publication of the first volume of Donald Knuth's epic *The Art of Computer Programming* (TAOCP). Includes some discussion of how to *generate* random numbers, and how to deal with random data. This is the true first pinnacle in the study of algorithms.

15. Genetic algorithms, introduced as early as 1975, by John Holland.

16. Simulated annealing was introduced by Armen G. Khachaturyan, Svetlana V. Semenovskaya, Boris K. Vainshtein (1979); and by Armen G. Khachaturyan, Svetlana V. Semenovskaya, Boris K. Vainshtein (1981). (via Wikipedia)

17. Neural networks and their successor, deep learning, sometimes involve randomness, or are combined with random subroutines for certain functions. More importantly, perhaps, is that their nonlinear behavior makes direct prediction of their outcomes hard.

1.2.1 Bayesian Inference

It's a bit of an aside, but you should know Bayes' Theorem:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}, \tag{1}$$

where A and B are events with a certain probability, and $A|B$ is read "A given B," that is, the probability of even A when we know that B is true.

Let's see if we can come up with an example in class...

1.3 Sex as an Algorithm

Sex is optimizing for population, not a single winner. This has significant implications for fitness, according to Adi Livnat and Christos Papadimitriou, writing in "Sex as an Algorithm," in the Nov. 2016 *Communications of the ACM*. (Note that I'm not wild about the title; not only is it a bit lurid for no particularly good reason, but really the focus is about the effects of sexual reproduction, with nothing to do with the act itself.)

A good quote:

> Recent research at the interface of evolution and CS has revealed that evolution under sex possesses a surprising and multifaceted computational nature: It can be seen as a coordination game between genes played according to the powerful Multiplicative Weights Update Algorithm; or as a randomized algorithm for deciding whether genetic variants perform well across all possible genetic combinations; it allows mutation to process and transmit information from transient genetic combinations to future generations; and much more.

There is a great question:

> What exactly is evolution optimizing, if anything?

They found that the genes are competing with each other, and at each generation, each gene picks a "strategy," consisting of the relative balance of *alleles* (versions of a gene for something particular) of itself. However, the organism has one fitness function that determines the probability of each individual surviving to reproduce.

One important point is that with different alleles, sex optimizes not for the individual best allele, but for *the allele that plays reasonably well in combination with the most other alleles of other genes*. Again, a quote:

> There is a mismatch between heuristics and evolution. Heuristics should strive to create populations that contain outstanding individuals. In contrast, evolution under sex seems to excel at something markedly different: at creating a "good population."

I'll note that although the ideas here are interesting, they have generated some skepticism among biologists.

1.4 Quantum Computing and Optimization

Back in Lecture 9, when we talked about minimum weight matching, I mentioned that the classical algorithm is useful for matching errors in a quantum computer. But it turns out that, yes, a quantum computer is also useful for solving optimization problems!

While we might hope naively that a quantum computer would give us an exponential speedup, in fact, the best we get on *arbitrary, unstructured* problems is a quadratic speedup: $O(N) \rightarrow O(\sqrt{N})$ (i.e., if the original problem is $O(2^n)$, with Grover you can get it down to $O(2^{n/2})$), using an algorithm known as Grover's algorithm and its generalization, amplitude amplification.

For specific problems with some structure, bigger speedups are possible. There has been a lot of work recently on quantum machine learning, mostly the linear algebra side of it. I'm dubious about the practicality of some of the work, but it's potentially valuable.

A recent paper by Moylett, Linden and Montanaro (`arXiv:1612.06203`) specifically discusses quantum solutions to the travelling salesman problem.

2 Core Ideas from the Semester

Now that we are done with the semester, we can see a little more clearly the complete picture. It is important to distinguish the **problem**, the **objective function**, and the **algorithmic strategy**.

2.1 The Problem Itself

With respect to the problem, there are many aspects that affect our approach to solving it:

1. whether the problem is single or multiple objective (Pareto);
2. the extent to which the problem is understood (whether the objective function is known and fixed – it's possible even that it varies over time), determining whether we must divide our work into exploration and exploitation phases;
3. whether an exact solution is required or an approximate solution is allowed;
4. whether certificates of infeasibility & optimality exist (affecting whether the problem can be NP-complete);
5. whether a mapping to another problem is known to exist (also obviously affecting NP-completeness);
6. whether we are addressing the primal or dual form (noting that we have a certain amount of freedom to choose);
7. whether it is a maximization or minimization problem (again, something we can choose by modifying the problem, e.g. by switching between the primal and dual forms); and

8. the relative value of quick, approximate solutions versus slower, more exact ones (keeping in mind that "slower" in some cases might be age-of-the-univese kinds of numbers); and even the use of optimization in online (real time) situations, from stock trading to adjusting a flight path as wind conditions vary to life-critical real-time optimization such as control of the flight surfaces on a plane.

2.2 The Objective Function

With respect to the objective function, the most important things are:

1. whether the objective function space is continuous or discrete;
2. whether the input variables are continuous or discrete (linear or integer problem);
3. whether the function is linear or nonlinear in the input variables;
4. whether it's differentiable (once or twice);
5. whether it's deterministic or random from some distribution (known or unknown), which is tied to whether we need to perform *exploration* before settling in to *exploit* our solution;
6. whether it's single-valued (Pareto).

3 Classes of Optimization Problems We Addressed

3.1 Linear Programming/Problems

Recall that we spent the early part of the semester focusing on linear programming, in which the objective function is smooth, a simple linear combination of the input variables (of which there might be many). In the n-dimensional space (for n variables), then, obviously the value of the objective function increases in some direction, so the focus of solving the problem becomes moving in a direction that helps us until we slam into one of the constraints. The constraints collectively define a convex *polytope* in n-space, and our optimal point will be at a vertex of the polytope (or possibly an edge or face).

Recall that we used the *simplex* algorithm to solve our problems, which first required us to put the problem into *standard equality form*, where

1. it is a maximization problem;
2. other than the non-negativity constraints, all constraints are *equalities*; and
3. every variable has a non-negativity constraint.

For variables without a non-negativity constraint, we achieved this by substitution of pairs of variables such as x_3^+ and x_3^- and setting $x_3 = x_3^+ - x_3^-$, then adding constraints $x_3^+, x_3^- \geq 0$. To turn inequalities into equalities, we added *slack variables*.

Once that's done, we take a basic solution, pick a variable, and increase it until we hit a stop. We then pick a different basis and reset the problem into *canonical form* for the new basis, which will allow us to increase a *different* variable without accidentally undoing some of the good work we have already done.

3.2 Integer Problems

Integer problems have the additional constraint that at least some of the variables must be integers in the solution, rather than continuous; e.g., if you're packing a transport ship, you can't ship half a car! The only technique we really addressed here was the *linear relaxation* of the integer problem, where under certain circumstances we can be sure that our solution using our linear techniques will ultimately result in an integer solution.

Important additional integer programming techniques we didn't cover include *branch and bound* and *dynamic programming*.

3.3 Graph Problems

To me, the most interesting and intuitive problems in this field are those on graphs. We looked at three, though there are others.

3.3.1 Minimum Cost Spanning Tree

A *minimum cost spanning tree* is a subset of the links in a weighted graph that creates exactly one path between every pair of nodes. This naturally results in a tree, a graph with no loops. If every link has a distinct weight, the minimum spanning tree is unique, and all algorithms for MST will find it; if some links have the same cost, it is possible that there are multiple trees that have the same cost, and different algorithms or different starting points for the same algorithm may find different trees.

We talked about the following algorithms:

1. Boruvka's algorithm (1926) builds a "forest" of trees, adding the lowest-cost edge to each tree and iterating until they merge into a single tree. Requires that all edges have distinct weights.

2. Prim's algorithm (1930) builds a single tree, starting from some point.

3. Kruskal's algorithm is similar to Boruvka's, but rather than iterating through the nodes (or components) to add the lowest-cost link from each component, it stores all the edges in a global list and adds the lowest-cost edge that connects things.

One important thing we learned from addressing this problem is that many algorithms can solve the same problem in different ways.

3.3.2 Shortest Path (Pair-wise or Tree)

The above minimized the global cost (e.g., dollars to build the network). In the shortest path problem, instead we are concerned with creating the cheapest path between each

pair of nodes (e.g., highest bandwidth or lowest latency). This problem has two forms: the pair-wise form, in which we are given a source and destination and are only looking for the shortest path between that one pair; and the spanning tree form, in which we are creating a spanning tree of the shortest paths from a given source to all destinations in the graph:

1. The most famous solution, of course, is Dijkstra's algorithm, which can be used for either.
2. We also looked at Radia Perlman's distributed approach to building the shortest path spanning tree, which is used in Ethernet networks; this was our first distributed algorithm discussed in class.
3. We followed up on the above with a homework set proving that the distributed form of Dijkstra, known as Open Shortest Path First, guarantees consistency in routing even with distributed solutions to the spanning tree problem.
4. We also looked at the A* algorithm, which can be used in more open environments where we don't necessarily have a fixed graph we are operating on. It works by augmenting our expected cost with a minimum estimate of the cost from each point on the frontier to the destination.

3.3.3 Minimum Weight Bipartite Matching

It turns out that many "matching" problems, where we have to e.g. match people to jobs or to houses or what have you, can be represented as minimum weight bipartite matching problems on graphs (remember what a bipartite graph is?).

The algorithm we studied in detail is the Hungarian algorithm (created by two Americans). Recall that we took the matrix representing the table of costs, essentially figured out how to factor out the minimum cost so that some of the entries became zero, then see if we can create a minimal matching. Various forms of voodoo were invoked to create a subgraph and reduce the problem, including covering the zeros.

3.4 Non-linear Problems

Perhaps the most fun we had (or at least, the most fun I had) this semester was working on non-linear problems. In this case, our objective function is non-linear, and so doesn't uniformly increase in some fixed direction throughout the polytope of possible answers. Instead, there may be "hills" and "valleys" in our function. Our problem definition may be to find the global minimum, or may be to find only a local minimum. In general, unless we know quite a bit about the objective function, finding the global minimum can be hard; even NP-hard, if we have a discrete space of possible solutions.

We should probably have started with *hill climbing*. In hill climbing, you pick one of the variables and move along it until it makes the objective function worse rather than better, then pick a different one of the variables and repeat until all of the variables are optimized. Done properly, this will take you to a local minimum (or maximum, depending on how the problem is defined). It is possible, though, that you'll actually hit a saddle point and stay there, so you really ought to check for that.

A more mathematical solution is *gradient descent*, in which we take the first derivative (using the partial derivatives) and find the vector in the entire n-space that gives us the greatest bang for our buck in terms of change, and jump out in that direction. A key problem with this is that we make finite-size jumps, and if our jumps are too small we make very slow progress, but if they are too large then we might be jumping over valleys or slamming back and forth along the walls, either being inefficient or possibly failing to reach a nearby minimum at all.

Newton's method takes the second derivative and attempts to make a better guess at each jump. It seems computationally intensive in each step, but in low-dimensional spaces it's reasonable and can get you much faster convergence.

Of course, all of those are just to find local minima. If you want a global minimum, or something close to it, without iterating through the entire space, you can apply heuristics such as simulated annealing, Metropolis and other Monte Carlo optimization techniques, particle swarm computing, ant colonies, or other techniques. You can also apply neural networks or deep learning to this, but be careful of any claims that they solve NP-complete problems in an efficient fashion. I am not aware of any credible claim for any technique that will solve NP-complete problems in polynomial time.

3.5 Coda: Lecture by Lecture

- Lec. 1: Introduction
- Basics of Linear Programming
  - Lec. 2: Examples of Linear, Integer and Graph Problems (formulating formal descriptions of problems)
  - Lec. 3: Three Kinds of Certificates, Standard Equality Form, and a Simplex Iteration
  - Lec. 4: The Heart of the Simplex Algorithm (bases and canonical forms) *In my opinion, this is the hardest week of the semester!*
  - Lec. 5: Sticking a Stake in the Heart of the Simplex Algorithm
- Graph Algorithms
  - Lec. 6: Basic Ideas of Graphs, Dijkstra's Shortest Path First Algorithm
  - Lec. 7: Several Minimum Spanning Tree Algorithms
  - Lec. 8: A^* Algorithm, Duality, and Minimum Spanning Tree Via Linear Algebra
  - Lec. 9: Minimum Weight Bipartite Matching
  - (Sadly, we are going to skip some important problems such as set cover, and min cut max flow. Semester projects, perhaps?)
- Integer Problems
  - Lec. 10: The Knapsack Problem

  - (Sadly, we are going to skip *many* important problems and solutions here!)

- Computational Complexity
  - Lec. 11: Basic Ideas of Computational Complexity

- Non-Linear Programming (NLP) and Advanced Techniques
  Warning: These lectures require a little basic calculus! But don't worry too much if you're not up on your derivatives; there will be no required homework on these lectures.
  - Lec. 12: Basics: Finding a Min/Max of a Function (Derivatives, Gradient Descent, Newton's Method)
  - Lec. 13: Deterministic Exact, Deterministic Heuristic, and Random Algorithms (AI-like methods, more on computational complexity)
  - Lec. 14: Pareto Optimality and Wrapping Up the Semester

4 Where To Next?

Suggestions for classes and independent study...

Classes:

- **Linear Algebra**: Ideally, you would have already taken LA before taking this class, but I know many (most?) of you haven't. Go take it, learn about eigenvectors and eigenvalues, etc.

- **Probability**: An absolute must for everyone, in my opinion. The probability class at SFC covers only the basics of discrete probability, you should consider following up on this with a class in continuous probability at Yagami or online.

- Where do you get Bayesian inference and statistics here?

- Differential and partial differential equations are necessary for many of the advanced techniques covered in the last three weeks.

- **Artificial Intelligence**: In Suwa-sensei's class, you'll learn more about the philosophy than the existing technologies, but he knows it all. Likewise, in Takefuji-sensei's classes, you'll find some AI-related optimization techniques and even non-silicon computing, such as using amoebae to optimize some functions.

Further topics for self-study:

- We have discussed algorithms here and executed them by hand, but to complete your understanding you should implement some of them. To do the "kin-tore" (muscle building) exercises for this, take any of the programming classes at SFC. *Especially, if you are a freshman and have just taken FIT1, plan to take FIT2 in the fall!*

- As noted above, probably the most important algorithmic techniques we didn't cover in detail are branch-and-bound, and dynamic programming.

- Besides algorithms and complexity theory, the other major component of solid theoretical computer science with a big influence on practical programming is *data structures*. How your program stores and accesses data elements can have a huge impact on the efficiency of your algorithm. We no longer have a specific class in this topic at SFC, but much of the knowledge has been scattered about into different classes. In the years when I teach Fundamentals of Systems Programming, using the programming language C, I introduce some of the key data structures, leading up to red-black trees; when Mitsugi-san teaches it, he emphasizes other basic structures.

- We did very little with the concept of duality during the semester, but people who work in this field consider it to be a central concept with broad practical implications.

- For some fun, and to think about what it means for a physical system to compute, go look at Scott Aaronson's "soap bubble optimization".

- Advancing toward classifiers and machine learning: supervised, unsupervised, reinforcement learning

Books, besides the ones listed below:

- Diestel (Japanese translation of 2nd edition is by Yagami's Ohta-sensei, but in English it's now up to 5th edition, available in Kindle now and hardback on Jan. 1, 2017)

5 Sources for the Semester

Some text and examples throughout the semester have been adapted from the paperback edition of A Gentle Introduction to Optimization*, B. Guerin* et al.

1. I use this book for simplex and background math in the first few weeks of the semester, and the Hungarian algorithm when we talk about minimum weight perfect matching on a bipartite graph:
   B. Guenin, J. Könemann, L. Tuncel,
   A Gentle Introduction to Optimization Theory.

2. And of course this book (the standard text on algorithms) for some of the graph algorithms:
   Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein,
   Introduction to Algorithms, Third Edition

3. And Russell and Norvig on AI, for the A* algorithm:
   Stuart Russell and Peter Norvig,
   Artificial Intelligence: A Modern Approach, 3e

(n.b.: There are several versions of this edition floating around, one purporting to be the international edition, but it seems the U.S. edition is the best.)

4. This book is a good reference I've used while preparing some material, but a little too heavy for an SFC class. I highly recommend it for further study for those interested in the topic, though (and it's cheap!):
   Christos H. Papadimitriou and Kenneth Steiglitz,
   Combinatorial Optimization: Algorithms and Complexity

5. One of these next two is the book that the prior instructor used when teaching this in Japanese. I think it's this one:
   Mikio Kubo,
   Kumiawase saitekika to arugorizumu
   but it might have been this one:
   Mikio Kubo and Tomomi Matsui,
   Kumiawase saitekika [tanpenshuu]

6. Wikipedia has been helpful, but the quality is very uneven.

7. For gradient descent, these sites have been helpful:
   Rudel, `http://sebastianruder.com/optimizing-gradient-descent/`
   Supstat, `http://vis.supstat.com/2013/03/gradient-descent-algorithm-with-r/`
   (with animation in R)
   Lukaszkujawa, `https://lukaszkujawa.github.io/gradient-descent.html`
   Also, there is an ad hoc list of test functions for global optimizers:
   There is a convenient summary at Wikipedia:
   `https://en.wikipedia.org/wiki/Test_functions_for_optimization`
   This comes from Rody Oldenhuis in Matlab by way of Gilberto A. Ortiz, also in Matlab on MathWorks.

6 Final Thoughts

Yes, have some.