Advanced Algorithms

Lecture 1: Course Survey and Approximation Algorithms

Lecturer: Nathan Klein

1 Introduction

Welcome to CS 530! This course serves as a broad overview of modern algorithms, with the goal of showing you a diverse set of mathematical tools, models, and applications.

1.1 Course Overview

While undergraduate algorithms courses typically focus on solving clean problems with polynomial time algorithms, we will explore what happens when we start to introduce more complexity into the problem or the model. In particular, we will study the following areas:

1. **NP-Hard Problems**: NP-Hard problems are not just mathematical curiosities cooked up to show the limits of computation. They appear all the time in the real world in applications like scheduling planes, delivering packages, and designing robust networks. We will study *approximation algorithms*: polynomial time procedures which output approximately optimal solutions to NP-Hard problems. Along the way we will learn about linear programming, an incredible tool both for theory and practice. We will also discuss probability, concentration bounds, and *average-case analysis*, where NP-Hard problems can become easy on average.

2. **Online Algorithms**: Often, decisions have to be made in real-time, and we can't wait to see all the data to decide on a solution. A classic example of this is ride-hailing apps, where the decision of which driver to assign to a rider has to be made in real time. Here we will explore how to evaluate and design algorithms that work in the online setting. We will then talk about a fairly recent field of research called *algorithms with predictions* in which average case and worst-case analysis are merged to obtain algorithms that work well on average and still do not have catastrophic worst-case behavior.

3. **Big Data**: The modern world produces a huge amount of data, on the order of billions of terabytes a year. This part of the course will focus on how to deal with large data with tools like locally sensitive hashing and dimensionality reduction. We will also explore the geometry of high dimensional data and the curse (and blessing) of dimensionality.

4. **Spectral Graph Theory**: Graphs are hugely important objects in practice: they are used to model road networks, social networks, and even the internet. In this part we will explore how linear algebra can help us make sense of graphs in useful and unexpected ways.

5. **Additional Topics**: Finally, we will get a taste of two additional topics: game theory and coding theory. Game theory asks how to build and analyze systems that involve selfish agents, like road networks or the stock market. Coding theory involves encoding data so that you can recover it from only a part of the encoded message. A simple way to do this is to just copy the data twice. It turns out you can do much, much better than this.

Each of these sections will also involve learning some key mathematical and algorithmic ideas. For example, in “NP-hard problems" we’ll learn about linear and semidefinite programming and concentration bounds, in “online algorithms" we’ll learn about duality, and in “big data" we’ll learn important concepts in linear algebra like low rank approximation and SVD.

1.2 How to Enjoy Taking this Course

Math is a language, and – like any other language – you need to memorize vocabulary. To this end, I’ll provide Anki flashcards each week that will help you prepare for the lectures, which you’ll have quizzes on. Knowing the flashcards will make going to class more enjoyable, as it will make sure you’re familiar with the language I’m using. Even more than that, self-testing has been shown to significantly boost learning and is even a strong predictor of a student’s final grade [Rod+21].

Another reason to study the flashcards is that it will make you more confident and able to ask questions when something I say doesn’t make sense. When you feel you’ve put in the work, you’re more likely to feel justified in asking. And asking question is an excellent way to help you stay engaged and make sure you and your classmates understand the material.

On top of that, come to office hours and discussions when you have time. We are more than happy to help out with homework, chat about potential final projects, and provide more explanation for concepts from class.

2 Approximating NP-Hard Problems

Like it or not, NP-Hard problems are everywhere. In undergraduate algorithms, typically we were happy to prove that either a problem had a polynomial time algorithm or that it was NP-Hard (in which case we would give up). In the first part of this unit, we will discuss one way to soldier on: *approximation algorithms*. These are polynomial time algorithms which give approximate solutions to optimization problems.

Approximation Algorithm

An α-approximation for an optimization problem is a polynomial time algorithm which produces a solution of cost within a factor of α of the optimal solution *for every instance.*
A randomized approximation algorithm only needs to output a solution of *expected* cost within a constant factor of the optimal solution.

So, for a minimization problem, the solution S returned by the algorithm must have the property $c(S) \leq \alpha \cdot c(OPT)$, where $c(S)$ is the cost of S and OPT is an optimal solution. For a maximization problem, we will have $c(S) \geq \alpha \cdot c(OPT)$.

We care about approximation algorithms for three reasons:

1. We can use approximation algorithms to get solutions of reasonable quality to NP-Hard problems. If we want algorithms that work in the worst case and the size of our input is large, this is the only option unless P=NP. Thus, these are useful algorithms in practice.

2. Working on approximation algorithms helps us understand and categorize NP-Hard problems. A problem with a 1.01 approximation such as knapsack is clearly different from a problem with no $n^{0.99}$ approximation (unless P=NP) such as max clique.

3. While not the focus of this course, approximation algorithms can be used for problems that are not NP-Hard. A 2-approximation running in time $O(n)$ may be preferable to an exact algorithm running in time $O(n^2)$ when a massive amount of data is involved.

Plus, the field is tied to many areas of theoretical computer science and math, some of which we will see in this course.

While I sing the field's praises (it's the main area I work in, so it's hard for me not to), I should also mention a drawback of approximation algorithms. By focusing on worst-case complexity, we may be chasing ghosts! Do these worst-case instances ever really come up? For some problems, it seems the answer is yes. But for others, not so much. Later on we will briefly look at average case analysis.

2.1 Complexity Classes

For each optimization problem, we would like to obtain an approximation algorithm with ratio α and prove that it is NP-Hard to approximate better than α. For many problems we are far from achieving this goal. However, for many problems we know at least whether they are in PTAS, APX, or in neither.

PTAS, FPTAS, and APX

1. A PTAS (Polynomial Time Approximation Scheme) is an approximation algorithm with ratio $1+\epsilon$ for any $\epsilon > 0$, i.e. running time polynomial for every fixed $\epsilon > 0$ (for a maximization problem, the ratio would be $1-\epsilon$). For example, a PTAS may have running time $n^{2^{1/\epsilon}}$. A problem with a PTAS is in the complexity class PTAS.

2. A FTPAS (Fully Polynomial Time Approximation Scheme) is a PTAS with running time polynomial in the input and $\frac{1}{\epsilon}$. For example, an FPTAS may have running time $\frac{n^3}{\epsilon^2}$.

3. A problem is in APX (Approximable) if there is an approximation algorithm with ratio $\alpha \in O(1)$.

A problem is called APX-Hard if there is a PTAS preserving reduction from all problems in APX to that problem: in other words, if a PTAS for the given problem would imply a PTAS for all others. Many problems we work with will be APX-Hard. Such a problem has no PTAS unless P=NP.

There are also problems with approximation ratios like $\log(n), n^\epsilon$, and so on. Even though these problems may not be in APX, finding approximation algorithms of this form is also of great interest.

2.2 The Traveling Salesperson Problem

One of the most famous optimization problems is the *traveling salesperson problem*, or TSP for short. Here you are given a list of n cities and the distances between them, which we assume are the same in both directions. I like to think about this as a complete graph $G = (V, E)$ with costs $c : E \to \mathbb{R}_{\geq 0}$ on the edges. Now, you want to find the cheapest Hamiltonian cycle of G. It turns out this problem is NP-Hard, and is not in APX (not even close).

Lemma 2.1. *There is no α approximation for TSP on n cities for any $\alpha \geq 1$ unless P=NP. Even an $O(2^n)$ approximation algorithm would imply P=NP.*

Proof. Suppose there was an α approximation for TSP for any α. Then, we can solve the Hamiltonian cycle problem, which asks if a given graph contains a Hamiltonian cycle as follows.

Given an instance of Hamiltonian cycle $G = (V, E)$, we create an instance of TSP on $G' = (V, E')$ (the vertex set is the same) by setting $c_e = 1$ if $e \in E$ and $c_e = \alpha n + 1$ otherwise.

If G has a Hamiltonian cycle, OPT is n, and so an α approximation for TSP run on G' will return a Hamiltonian cycle of cost at most αn. But that means it cannot use any edges not in G (as they have cost greater than αn, so it must return a Hamiltonian cycle, this implying P=NP. □

However, a very natural variant of TSP called *metric* TSP is in APX. Formally:

Definition 2.2 (Metric TSP)**.** *Given a metric space with elements V and metric $c : V \times V \to \mathbb{R}_{\geq 0}$, find the minimum cost Hamiltonian cycle.*

Now, since c is a metric, we have the triangle inequality: $c(u, w) \leq c(u, v) + c(v, w)$ for all $u, v, w \in [n]$. It turns out this allows us to get a 2-approximation, showing that it is in APX. We first notice the following, where $G = (V, E)$ is the complete graph with weights $w_e = c(u, v)$ for $e = \{u, v\}$ on the edges. For $F \subseteq E$, let $c(F) = \sum_{e \in F} c_e$.

Lemma 2.3. *Let F be a multi-set of edges of G so that F is connected (contains a spanning tree) and every vertex has even degree. Then, F can be "shortcut" to a Hamiltonian cycle of no greater cost.*

Proof. $G' = (V, F)$ is an Eulerian multi-graph. So (by a standard fact in graph theory) it has an Eulerian tour, a walk that visits every edge exactly once and returns to the starting point. Furthermore, we can compute this tour in polynomial time.[1]

Traverse the sequence of edges $e_1, e_2, \ldots, e_k$ visited by this tour. Whenever you are about to go to a vertex v you've already been to with edge $e = \{a, b\}$, if $e \neq e_k$, let $f = \{b, c\}$ be the next edge in the tour. Delete e and f and replace it with the edge $\{a, c\}$ (if $a = c$, no need to add the self-loop). The cost of the tour does not increase, because of the triangle inequality: $w(a, c) \leq w(a, b) + w(b, c)$. Furthermore, we will only visit the first vertex twice. So, this results in a Hamiltonian cycle H with $c(H) \leq c(F)$. □

Lemma 2.4 (The Double Tree Algorithm)**.** *The algorithm which takes a minimum spanning tree T of the graph and double all of its edges is a 2 approximation for metric TSP.*

[1] The intuition for this is simple, and is called Hierholzer's algorithm. Start at any vertex v and start walking without repeating edges (i.e., take a trail) until you return to v. You cannot possibly get stuck anywhere besides v, because every vertex has even degree, so whenever you enter a vertex there must also be a way to leave it. If all edges have been visited, terminate. Otherwise, by connectivity, there is a vertex v' visited on this walk which has unvisited edges. Starting at v', find another trail ending at v', and repeat until all edges have been visited.

Proof. First, notice that the multi-set of edges F returned by this algorithm forms a connected Eulerian graph. So, applying Lemma 2.3 we only have to argue that $c(R) = 2 \cdot c(T) \leq 2 \cdot c(OPT)$, or equivalently, $c(T) \leq c(OPT)$. But OPT is a tree plus an edge, so the MST is cheaper than OPT, completing the proof of the lemma. □

It turns out you can do even better by using a minimum cost matching using an algorithm discovered by Christofides [Chr76] and, independently, Serdyukov [Ser78].

Algorithm 1 Christofides-Serdyukov Algorithm

1: Find a minimum cost spanning tree T of G
2: Let O be the set of vertices with odd degree in T. Compute the minimum cost perfect matching M on O.
3: Return $T \uplus M$.

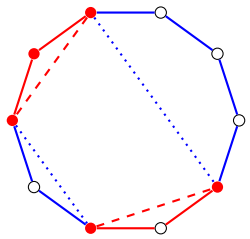

Figure 1: The set of odd vertices of the minimum spanning tree is marked in red (filled) along the optimal Hamiltonian cycle. Notice OPT can be decomposed into two disjoint sets of paths joining the odd vertices, marked in blue and red. Using the triangle inequality, these paths can be shortcut to form two matchings (dashed and dotted). Therefore the cheapest matching must cost at most $OPT/2$.

Theorem 2.5. *The Christofides-Serdyukov algorithm (Algorithm 1) is a $\frac{3}{2}$ approximation for metric TSP.*

Proof. The algorithm is well defined because there are an even number of odd vertices in the tree by the handshake lemma. Furthermore, the result, $T \uplus M$, is connected (because it contains a spanning tree), and Eulerian, because M increases the degree of all odd vertices by 1. So it can be shortcut to a Hamiltonian cycle by Lemma 2.3.

We know $c(T) \leq c(OPT)$. So it only remains to prove that $c(M) \leq \frac{1}{2}c(OPT)$. To see this, notice that the optimum Hamiltonian cycle can be partitioned into two sets of paths that join the odd vertices together. These paths can be shortcut to matchings, M_1 and M_2 as in Fig. 1. But this means that $c(M_1) + c(M_2) \leq c(OPT)$, so the cheapest matching costs at most $\frac{1}{2} \cdot c(OPT)$. □

This is a tight analysis. One nice example showing tightness is below in Fig. 2, which is even tight after shortcutting. Note that it is NP-Hard to find an optimal shortcutting (but even allowing an optimal shortcutting there are examples which show a lower bound of $\frac{3}{2}$ for Christofides [DT09]).

You might ask: can you do better than Christofides? It took a while to beat, but it turns out you can. In 2020, Anna Karlin, Shayan Oveis Gharan and I proved there is an approximation algorithm with ratio $\frac{3}{2} - 10^{-36}$ [KKO21]. It is conjectured that a ratio of $\frac{4}{3}$ is achievable, and showing that is a major open problem in theoretical computer science.

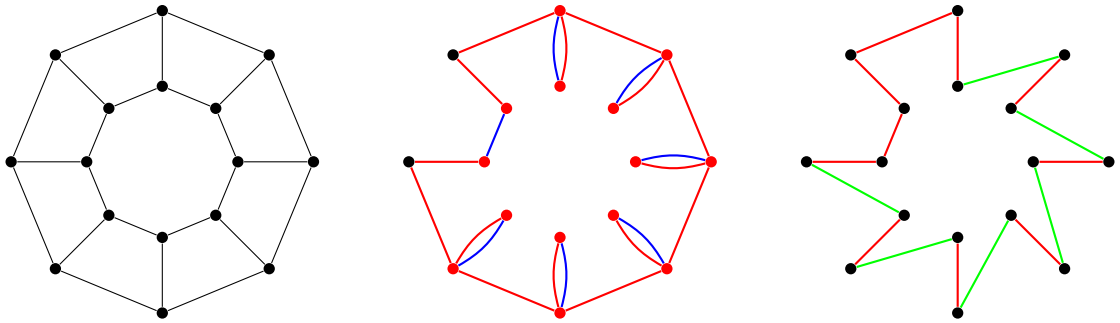

Figure 2: A tight example for Christofides. Let $c_e = 1$ for all edges. In red is the minimum spanning tree, and marked in red are the odd vertices. In blue is a minimum cost O-Join. The cost will be $\frac{3}{2}n - 2$, but the graph is Hamiltonian, so this is a lower bound of $3/2$ as $n \to \infty$. Note we can force Christofides to pick the red edges by making their costs just slightly cheaper than the remaining edges. On the right is a possible shortcutting of the Eulerian graph which also costs $\frac{3}{2}n - 2$, where the red edges have cost 1 and the green edges have cost 2.

References

[Chr76] Nicos Christofides. *Worst Case Analysis of a New Heuristic for the Traveling Salesman Problem*. Report 388. Pittsburgh, PA: Graduate School of Industrial Administration, Carnegie-Mellon University, 1976 (cit. on p. 5).

[DT09] Vladimir Deineko and Alexander Tiskin. "Min-weight double-tree shortcutting for Metric TSP: Bounding the approximation ratio". In: *Electronic Notes in Discrete Mathematics* 32 (2009). DIMAP Workshop on Algorithmic Graph Theory, pp. 19–26. ISSN: 1571-0653. DOI: https://doi.org/10.1016/j.endm.2009.02.004 (cit. on p. 5).

[KKO21] Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. "A (Slightly) Improved Approximation Algorithm for Metric TSP". In: *STOC*. ACM, 2021 (cit. on p. 5).

[Rod+21] Fernando Rodriguez, Sabrina Kataoka, Mariela Janet Rivas, Pavan Kadandale, Amanda Nili, and Mark Warschauer. "Do spacing and self-testing predict learning outcomes?" In: *Active Learning in Higher Education* 22.1 (2021), pp. 77–91. DOI: 10.1177/1469787418774185. eprint: https://doi.org/10.1177/1469787418774185 (cit. on p. 2).

[Ser78] A. I. Serdyukov. "O nekotorykh ekstremal'nykh obkhodakh v grafakh". In: *Upravlyaemye sistemy* 17 (1978), pp. 76–79 (cit. on p. 5).