1 Introduction

Computer science is an interesting meeting point of mathematics and engineering. It is a technology field that is also flush with deep mathematical and even philosophical questions such as:

1. Are there fundamental limits of computational efficiency?
2. What is the landscape of computational complexity?
3. What, if any, are the tradeoffs between computational resources? (time-space, time-number-of-processors)
4. Is there a polynomial vs. exponential boundary? Why is it significant?

With its strong technology personality, Computer Science also offers a rich 2-way street between “experimental” and “theoretical” sides. Some notable examples of theory and algorithms influencing, even seeding, technology include:

1. Google (Yahoo, Alta Vista) spawned by decades of information retrieval and search research.
2. Distributed computing and routing leading to the Internet.
3. Sequence and string analysis research to Genome decoding and computational biology.
4. Explosion of Big Data, Data Mining, Machine Learning, Robotics industries founded on decades of academic work.

Conversely, many “experimental” systems succeeded so well that theory/academic work has to do a lot of catching up to understand their design, operations, and behavior. Some notable examples:

1. Internet has spawned whole new areas of research on Decentralized computing; from design, to routing, to self-organizing, to selfish behavior theories.
2. Facebook like social graphs have spawned new areas, models, theories about structure of large social networks, including Small World Phenomenon, information diffusion, contagion etc.
3. Global telecommunication systems pose fundamental algorithmic problems of network design (capacity, routing, robustness, vulnerability).
4. Practical successes of machine learning and data mining poses fundamental questions of classification, data clustering, and even “Learnability.”
The Goal of this course is to teach a core set of algorithmic techniques. I will do this through a small set of clean, stylized problems (mostly on graphs and networks, but some on general data and optimization). The problems reflect my own taste, but also form a core in almost all advanced algorithms courses; are clean and stylized so we can develop an elegant theory, often with some surprises along the way, yet abstract and general enough to apply to a broad spectrum of applications and areas.

Engineers use the principle that “you study systems at small scale to learn something about the behavior, and then use those lessons to scale up.” Our stylized problems play a similar role: by getting rid of many of the messy, application-specific assumptions or constraints, we can understand the “dominant” complexity of the problem, and then apply the design/algorithms to the general problem, perhaps with some small adaptations.

2 Algorithms

The science of computing deals with discovering algorithms to solve computational problems. One of the dominant “schools of research” is centered around “combinatorial” methods, as opposed to continuous (calculus) mathematics. Scientific computing, control theory etc can be examples of the latter. The main edifice of Theory of Computing is built around these combinatorial (discrete) techniques, of which graphs and networks are the pillars.

Indeed, over the last 10-15 years, graphs have reached an almost cult-like status: from social networks to “systems biology” to “networked science” of diseases and epidemics, banking and finance, physical and communication infrastructure (power grid, road networks, airlines, shipping) etc etc. So it may seem surprising that “graphs” and “graph theory” have been around for more than 250 years! Starting with a short but influential paper of Euler.

The problems we consider are set in an abstract framework: a graph whose nodes and edges are abstract objects; the nodes as well the edges are often associated with costs and/or capacities. This formulation is general enough to encompass most of the optimization problems that arise in networking, whether it be computer networks or any other network (e.g. transportation, power, social networks, program flow etc). We will explore the issues of reachability, connectivity, routing, flows, matching or assignments. These explorations will naturally suggest the optimization problems such as the shortest paths, minimum spanning trees, network flows, matching etc.

These optimization problems (and their solutions) have come to occupy a fundamental place in computer science algorithms, in part because they arise in so many different applications. But perhaps their importance owes more to the deep insights uncovered by their study. We use these problems to illustrate how non-trivial algorithms are designed and analyzed, and to show an interplay between data structures and algorithms. A rough outline (and superset) of the syllabus is as follows:

1. (Network Flows)
   - Applications of network flows.
   - Augmentation, Ford-Fulkerson.
   - Edmonds-Karp heuristics.
   - Preflow-Push algorithms.

2. (Shortest Paths)
   - Label-Setting Algorithms, Dijkstra.

2
• How to detect negative cycles?
• Applications of shortest path problem.

3. (Min Spanning Trees)
• Applications.
• Cut and cycle conditions of optimality.
• Prim, Kruskal, and Sollin’s algorithms.
• Spanner networks

4. (Matching)
• Applications of assignment.
• Bipartite matching using network flow.
• Hungarian method for assignment.
• Stable matching and its applications.

5. (Small World Models)
• Small world phenomenon.
• Milgrom experiment.
• Kleinberg model.
• Analysis of the Kleinberg model.

6. (Linear and Integer Programming)
• Formulation.
• Simplex method.
• LP Duality.
• Integer Programming.

7. (A Theory of Learning)
• Valiant’s PAC model.
• VC dimension, sample complexity.

8. (Some Misc. Topics)
• Voting and Arrow’s Impossibility Theorem.

Besides developing efficient algorithms for important problems, the research on combinatorial optimization problems was also the key force in laying the foundation of Complexity Theory and NP-Completeness. Indeed, Jack Edmonds’ characterizationn of computational problems as good and bad—the former being those that admit a polynomial time solution, the latter being the ones that do not have a polynomial time solution—came about from his research on the “matching problem”, for which he was able to develop a provably polynomial time algorithm and the traveling salesman problem, for which none seemed possible. Soon afterwards, Steven Cook proved his well-known theorem of NP-Completeness, which was quickly followed by the influential work of Richard Karp who showed that twelve familiar combinatorial problems (including traveling salesman and graph coloring) were NP-Complete.
An Aside (C. Papadimitriou): A keyword search in Melvyl (UC Berkeley online library) revealed that about 6000 papers each year have the term “NP-Completeness” in their title, abstract or list of key words. This is more than each of the terms “compilers,” “database,” “expert,” “operating system.” The range is impressive: from statistics and AI to automatic control and nuclear engineering.

3 Graphs

A graph $G$ is a pair $(V, E)$ where $V$ is a finite set and $E$ is a binary relation on $V$. The graphs are a versatile model for organizing data as well as formulating a wide spectrum of problems. Example problems include computing distances, finding circularities in relationships, determining connectivity, determining least cost communication networks, generating efficient assembly codes for evaluating expression, and measuring reliability of networks.

In one sense, graph is nothing more than a binary relation. However, it derives its real power from the visualization as a set of points (vertices) interconnected by edges. In that regard, it generalizes trees, and has proved to be a powerful abstraction or model.

3.1 Example Uses of Graphs

In many situations, graphs are natural models, and their use is obvious. The following examples are from “Graph theory and its applications to problems of society” by Fred Roberts.

1. Nodes are locations (traffic intersections) in a city; edges are streets. This is a directed graph. We may be interested in re-directing edges to optimize traffic flow.

2. Suppose certain streets need to be cleaned on certain days. Plan an optimal cleaning route.

3. Let the nodes be locations in a power plant. Draw a edge from location $x$ to $y$ if a watchman located at $x$ can see a warning light at $y$. How to determine the minimum number of watchmen to see all the warning lights.

4. Node are fragments of DNA or RNA sequence. Put an edge between two if the two fragments overlap. The resulting graph is used for reconstructing the DNA sequence.

5. Nodes are CS classes. Put an edge between two nodes if the corresponding classes have or likely to have common students. A coloring of this graph can be used to schedule the classes or exams without creating conflicts.

6. Frequency allocation in mobile phones. Put a node for each zone, and an edge between two if they conflict (that is, if the phones in the two zones interfere and so the frequencies assigned to two neighboring zones should be disjoint). The problem is to assign to each node a band $B_i$ such that each band is at least $L$ long, and $B_i \cap B_j = \emptyset$ whenever $(i, j)$ is an edge in the graph.

Similarly, problems involving facility location, shortest paths, Hamiltonian cycles, flows arise naturally in many network situations. Many of these problems arise in Internet context too—cache placement, open shortest path first routing, MPLS routing for setting up tunnels etc.

In other situations, however, graphs are not directly suggested by the problem itself, but nevertheless they can be invaluable tool. Scheduling of lecture halls, exam time slots, for instance, can be be modeled as a graph coloring problem. Such indirect uses of minimum spanning tree, for instance, include optimal broadcast in an unreliable medium, optimal data storage in two-dimensional arrays, min-max path problems, and cluster analysis.
3.2 Course Philosophy

The philosophy of CS 231 will be “in depth” coverage of a few fundamental combinatorial problems and algorithms. Unlike CS 130a or CS 130b, we will not be simply content with learning a new algorithm. Instead, we will aim for a deeper understanding of the structure of the problem. This bottom-up approach of algorithm design is in sharp contrast with the top-down “hit and miss” approach, where one tries the standard techniques, such as greedy or dynamic programming.

The bottom-up approach often leads to a unification of algorithms as well. For instance, all algorithms for the min spanning tree can be characterized as greedy, with two abstract rules—cut rule and cycle rule. Similarly, most of the shortest path algorithms are based on the technique of relaxation; the maxflow algorithms tend to fall into two fundamentally different classes—augmentation, and preflow-push.

By gaining insight into the problem by uncovering some fundamental properties and structures, we will see that often good algorithms will suggest themselves. We will observe this phenomena throughout the course. In each case, we also find that converting this structural understanding into a highly efficient algorithm requires development of non-trivial data structures. Several important data structures, such as Fibonacci heaps and splay trees, were motivated by minimum spanning trees and max flow algorithms.

3.3 $P$ or $NP$

Graphs are a powerful model for many real life problems. But simply formulating a problem as a graph problem is not an end—it’s just a beginning. While many graph problems admit efficient (and well known) solutions, just as many (and more) do not. In fact, the catalog of NP-Completeness problem is littered with hard graph problems. It is important to understand the fine details of graph algorithms, so we can tell whether the same algorithm will work for a slightly modified problem or not. Often similar sounding problems have very different complexity. Examples:

- **[Spanning Subgraphs.]** Given is an edge-weighted graph $G$.
  1. [MST.] Compute its cheapest spanning subgraph.
  2. [Steiner MST.] Given a subset of vertices $S \subseteq V$, called terminals, compute the cheapest subgraph spanning $S$.

- **[Simple Tours.]** Given is a connected unweighted undirected graph.
  1. [Euler.] Is there a simple tour of all edges?
  2. [Hamiltonian.] Is there a simple tour of all vertices?

- **[Min Spanning Trees.]** Given is an edge-weighted graph $G$.
  1. [MST.] Compute its cheapest spanning subgraph.
  2. [k-MST.] Compute the cheapest tree spanning at least $k$ nodes; $k$ is part of the input.

- **[Matching.]**
  1. [Bipartite.] Given two sets $X$ and $Y$, $|X| = |Y| = n$, and a compatibility relation between them, does there exist a perfect matching? (Think of $X$ as women, $Y$ as men, and compatibility as “willingness to marry each other”. Then, this is the marriage problem.)
  2. [3-dimensional Matching.] Given 3 sets $X$, $Y$, and $Z$, $|X| = |Y| = |Z| = n$, and a set of acceptable “triples”, does there exist a perfect 3-way matching? (Well, you can think about 3-way marriages if you like, but the problem is real in forming business teams; e.g. teams of legal experts in a law firm, investment experts in financial markets etc.)
• **[Coloring.]** Given a planar graph $G$.
  1. **[2-Coloring.]** Checking if $G$ is bipartite.
  2. **[4-Coloring.]** Consequence of 4-color theorem.
  3. **[3-Coloring.]** NP-Complete.
  4. By contrast, the coloring problem for general non-planar graph is inapproximable. It is known that if an $n^c$ approximation algorithm exists, then $P = NP$.

4 **Administrative**

• My web page is [www.cs.ucsb.edu/~suri](http://www.cs.ucsb.edu/~suri). Email address is suri@cs.ucsb.edu. The course material will be posted at [www.cs.ucsb.edu/~suri/cs231/231.html](http://www.cs.ucsb.edu/~suri/cs231/231.html)

• Professor’s office hours: Tues 11-12 AM

• No TA for the course.

• There is not required textbook for the course. My lecture notes should be sufficient. However, there are several excellent textbooks on algorithms or network flows if you need to consult. For instance, *Introduction to Algorithms*, by Cormen, Leiserson, Rivest and Stein, *Network Flows: Theory, Algorithms, and Applications* by Ahuja, Magnanti, Orlin, and *Algorithm Design*, by Kleinberg and Tardos.