Learning Objectives:

- Describe at a high level some interesting computational problems.
- The difference between polynomial and exponential time.
- Examples of techniques for obtaining efficient algorithms
- Examples of how seemingly small differences in problems can make (at least apparent) huge differences in their computational complexity.

13

Efficient computation

> *"The problem of distinguishing prime numbers from composite and of resolving the latter into their prime factors is ... one of the most important and useful in arithmetic ... Nevertheless we must confess that all methods ... are either restricted to very special cases or are so laborious ... they try the patience of even the practiced calculator ... and do not apply at all to larger numbers."*, Carl Friedrich Gauss, 1798

> *"For practical purposes, the difference between algebraic and exponential order is often more crucial than the difference between finite and non-finite."*, Jack Edmunds, "Paths, Trees, and Flowers", 1963

> *"Sad to say, but it will be many more years, if ever before we really understand the Mystical Power of Twoness... 2-SAT is easy, 3-SAT is hard, 2-dimensional matching is easy, 3-dimensional matching is hard. Why? oh, Why?"* Eugene Lawler

So far we were concerned with which functions are computable and which ones are not. But now we return to *quantitative considerations* and study the time that it takes to compute functions mapping strings to strings, as a function of the input length. One of the interesting phenomenona of computing is that there is often a kind of a "thershold phenomenon" or "zero one law" for running time, where for many natural problems, they either can be solved in *polynomial* running time (e.s., something like $O(n^2)$ or $O(n^3)$), or require *exponential* (e.g., at least $2^{\Omega(n)}$ or $2^{\Omega(\sqrt{n})}$) running time. The reasons for this phenomenon are still not fully understood, but some light on this

is shed by the concept of *NP completeness*, which we will encounter later.

In this lecture we will survey some examples of computational problems, for some of which we know efficient (e.g., n^c-time for a small constant c) algorithms, and for others the best known algorithms are exponential. We want to get a feel as to the kinds of problems that lie on each side of this divide and also see how some seemingly minor changes in formulation can make the (known) complexity of a problem "jump" from polynomial to exponential.

In this lecture, we will not formally define the notion of running time, and so use the same notion of an $O(n)$ or $O(n^2)$ time algorithms as the one you've seen in an intro to CS course: "I know it when I see it". In the next lecture, we will define this notion precisely, using our NAND++ and NAND« programming languages. One of the nice things about the theory of computation is that it turns out that, like in the context of computability, the details of the precise computational model or programming language don't matter that much, especially if you mostly care about the distinction between polynomial and exponential time.

13.1 Problems on graphs

We nor present a few examples of computational problems that people are interesting in solving. Many of the problems will involve *graphs*. We have already encountered graphs in the context of Boolean circuits, but let us now quickly recall the basic notation. A graph G consists of a set of *vertices* V and *edges* E where each edge is a pair of vertices. In a *directed* graph, an edge is an ordered pair (u, v), which we sometimes denote as $\overrightarrow{u\,v}$. In an *undirected* graph, an edge is an unordered pair (or simply a set) $\{u, v\}$ which we sometimes denote as $\overline{u\,v}$ or $u \sim v$.[1] We will assume graphs are undirected and *simple* (i.e., containing no parallel edges or self-loops) unless stated otherwise.

[1] An equivalent viewpoint is that an undirected graph is like a directed graph with the property that whenever the edge $\overrightarrow{u\,v}$ is present then so is the edge $\overrightarrow{v\,u}$.

We typically will think of the vertices in a graph as simply the set $[n]$ of the numbers from 0 till $n-1$. Graphs can be represented either in the *adjacency list* representation, which is a list of n lists, with the i^{th} list corresponding to the neighbors of the i^{th} vertex, or the *adjacency matrix* representation, which is an $n \times n$ matrix A with $A_{i,j}$ equalling 1 if the edge $\overrightarrow{u\,v}$ is present and equalling 0 otherwise.[2] We can transform between these two representations using $O(n^2)$ operations, and hence for our purposes we will mostly consider them

[2] In an undirected graph, the adjacency matrix A is *symmetric*, in the sense that it satisfies $A_{i,j} = A_{j,i}$.

as equivalent. We will sometimes consider *labeled* or *weighted* graphs, where we assign a label or a number to the edges or vertices of the graph, but mostly we will try to keep things simple and stick to the basic notion of an unlabeled, unweighted, simple undirected graph.

There is a reason that graphs are so ubiquitous in computer science and other sciences. They can be used to model a great many of the data that we encounter. These are not just the "obvious" networks such as the road network (which can be thought of as a graph of whose vertices are locations with edges corresponding to road segments), or the web (which can be thought of as a graph whose vertices are web pages with edges corresponding to links), or social networks (which can be thought of as a graph whose vertices are people and the edges correspond to friend relation). Graphs can also denote correlations in data (e.g., graph of observations of features with edges corresponding to features that tend to appear together), casual relations (e.g., gene regulatory networks, where a gene is connected to gene products it derives), or the state space of a system (e.g., graph of configurations of a physical system, with edges corresponding to states that can be reached from one another in one step).

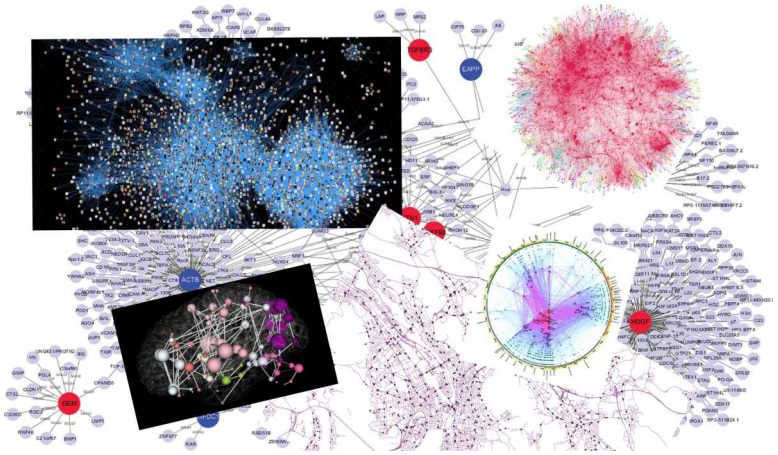

Figure 13.1: Some examples of graphs found on the Internet.

We now give some examples of computational problems on graphs. As mentioned above, to keep things simple, we will restrict attention to undirected simple graphs. In all cases the input graph $G = (V, E)$ will have n vertices and m edges.

13.1.1 Finding the shortest path in a graph

The *shortest path problem* is the task of, given a graph $G = (V, E)$ and two vertices $s, t \in V$, to find the length of the shortest path between s and t (if such a path exists). That is, we want to find the smallest number k such that there are vertices $v_0, v_1, \ldots, v_k$ with $v_0 = s$, $v_k = t$ and for every $i \in \{0, \ldots, k-1\}$ an edge between v_i and v_{i+1}. If each vertex has at least two neighbors then there can be an *exponential* number of paths from s to t, but fortunately we do not have to enumerate them all to find the shortest path. We can do so by performing a breadth first search (BFS), enumerating s's neighbors, and then neighbors' neighbors, etc.. in order. If we maintain the neighbors in a list we can perform a BFS in $O(n^2)$ time, while using a queue we can do this in $O(m)$ time.[3]

[3] A *queue* stores a list of elements in "First In First Out (FIFO)" order and so each "pop" operation removes an element from the queue in the order that they were "pushed" into it; see the Wikipedia page. Since we assume $m \geq n - 1$, $O(m)$ is the same as $O(n + m)$. Dijkstra's algorithm is a well-known generalization of BFS to *weighted* graphs.

More formally, the algorithm for shortest path can be described as follows:

Algorithm *SHORTESTPATH*:

- **Input:** Graph $G = (V, E)$, vertices s, t
- **Goal:** Find the shortest path $v_0, v_1, \ldots, v_k$ such that $v_0 = s$, $v_k = t$ and $\{v_i, v_{i+1}\} \in E$ for every $i \in [k]$, if such a path exists.
- **Operation:**

1. We will maintain a label $L[v]$ for every vertex v. Initially no vertex is labeled except for s that is labeled with "start".
2. We maintain a *queue* Q of vertices, initially Q contains only s.
3. While Q is not empty do the following:

   a. Pop the vertex v from the top of the queue.

   b. If $v = t$ exit output the path which is the reverse order of $v, L[v], L[L[v]], L[L[L[v]]], \ldots, s$.

   c. Otherwise, label all the unlabeled neighbors of v with v and add them to Q
4. Output "no path"

Since we only add to the queue unlabeled vertices, we never push to the queue a vertex more than once, and hence the algorithm takes n "push" and "pop" operations. It returns the correct answer since add the vertices to the queue in the order of their distance from s, and hence we will reach t after we have explored all the vertices that are closer to s than t.

13.1.2 *Finding the longest path in a graph*

The *longest path problem* is the task of, given a graph $G = (V, E)$ and two vertices $s, t \in V$, to find the length of the *longest* simple (i.e., non intersecting) path between s and t. If the graph is a road network, then the longest path might seem less motivated than the shortest path, but of course graphs can be and are used to model a variety of phenomena, and in many such cases the longest path (and some of its variants) are highly moticated. In particular, finding the longest path is a generalization of the famous Hamiltonian path problem which asks for a *maximally long* simple path (i.e., path that visits all n vertices once) between s and t, as well as the notorious traveling salesman problem (TSP) of finding (in a weighted graph) a path visiting all vertices of cost at most w. TSP is a classical optimization problem, with applications ranging from planning and logistics to DNA sequencing and astronomy.

A priori it is not clear that finding the longest path should be harder than finding the shortest path, but this turns out to be the case. While we know how to find the shortest path in $O(n)$ time, for the longest path problem we have not been able to significantly improve upon the trivial brute force algorithm that tries all paths.

Specifically, in a graph of degree at most d, we can enumerate over all paths of length k by going over the (at most d) neighbors of each vertex. This would take about $O(d^k)$ steps, and since the longest simple path can't have length more than the number of vertices, this means that the brute force algorithms runs in $O(d^n)$ time (which we can bound by $O(n^n)$ since the maximum degree is n). The best algorithm for the longest path improves on this, but not by much: it takes $\Omega(c^n)$ time for some constant $c > 1$.[4]

[4] At the moment the best record is $c \sim 1.65$ or so. Even obtaining an $O(2^n)$ time bound is not that simple, see Exercise 13.1.

13.1.3 *Finding the minimum cut in a graph*

Given a graph $G = (V, E)$, a *cut* is a subset S of V such that S is neither empty nor is it all of V. The edges cut by S are those edges where one of their endpoints is in S and the other is in $\overline{S} = V \setminus S$. We denote this set of edges by $E(S, \overline{S})$. If $s, t \in V$ then an s, t *cut* is a cut such that $s \in S$ and $t \in \overline{S}$. (See Fig. 13.3.) The *minimum* s, t *cut problem* is the task of finding, given s and t, the minimum number k such that there is an s, t cut cutting k edges (the problem is also sometimes phrased as finding the set that achieves this minimum).[5]

[5] One can also define the problem of finding the *global minimum cut* (i.e., the non-empty and non-everything set S that minimizes the number of edges cut). One can verify that a polynomial time algorithm for the minimum s, t cut can be used to solve the global cut in polynomial time as well (can you see why?).

The minimum s, t cut problem appears in many applications. Minimum cuts often correspond to *bottlenecks*. For example, in a com-

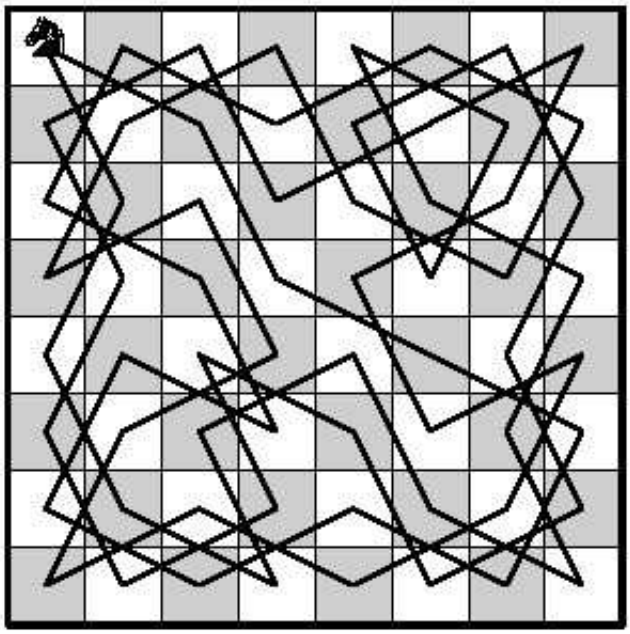

Figure 13.2: A *knight's tour* can be thought of as a maximally long path on the graph corresponding to a chessboard where we put an edge between any two squares that can be reached by one step via a legal knight move.

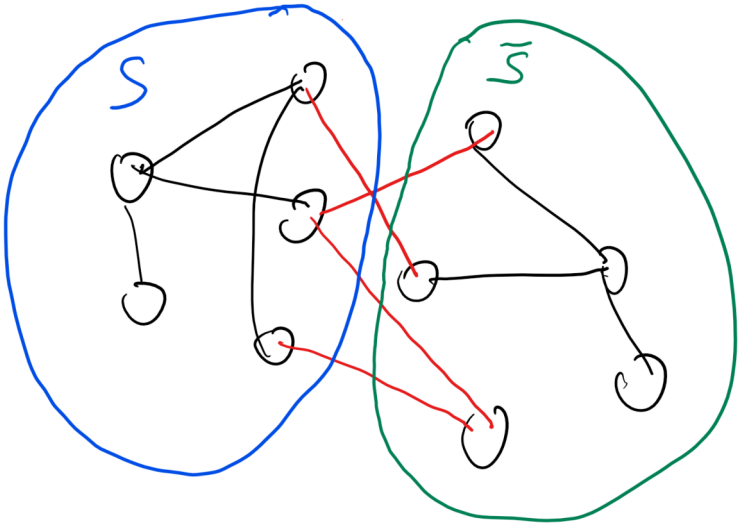

Figure 13.3: A *cut* in a graph $G = (V, E)$ is simply a subset S of its vertices. The edges that are *cut* by S are all those whose one endpoint is in S and the other one is in $\overline{S} = V \setminus S$. The cut edges are colored red in this figure.

munication network the minimum cut between s and t corresponds to the smallest number of edges that, if dropped, will disconnect s from t. Similar applications arise in scheduling and planning. In the setting of image segmentation, one can define a graph whose vertices are pixels and whose edges correspond to neighboring pixels of distinct colors. If we want to separate the foreground from the background then we can pick (or guess) a foreground pixel s and background pixel t and ask for a minimum cut between them.

Solving the minimum cut problem: Here is an algorithm to solve the minimum cut problem:

Algorithm MINCUTNAIVE:

- **Input:** Graph $G = (V, E)$ and two distinct vertices $s, t \in V$
- **Goal:** Return S s.t. $s \in S$ and $t \notin S$ that minimizes $|E(S, \overline{S})|$.
- **Operation:**
  1. If $V = \{s, t\}$ then return $\{s\}$.
  2. Otherwise choose $v \in V \setminus \{s, t\}$, and define G', G'' to be two graphs with vertex set $V \setminus \{v\}$, where in G' the edge set E' is obtained by making all of v's neighbors be neighbors of s, and in G'' the edge set E'' is obtained by making all of v's neighbors be neighbors of t. That is, E' has the same edges as E except that in edges involving v we replace v with s, and E'' has the same edges as E except that in edges involving v we replace v with t.
  3. Compute recursively $S' = MINCUTNAIVE(G', s, t)$ and $S'' = MINCUTNAIVE(G'', s, t)$.
  4. If $S' \cup \{v\}$ cuts fewer edges in G than S'' then return $S' \cup \{v\}$. Otherwise return S''.

It is an excellent exercise for you to pause at this point and verify: **(i)** that you understand what this algorithm does, **(2)** that you understand why this algorithm will in fact return the minimum cut in the graph, and **(3)** that you can analyze the running time of this algorithm.

We can prove by induction that Algorithm $MINCUTNAIVE$ does indeed return the minimum cut. Indeed, it definitely does so for graphs of two vertices. Now we assume by induction that $MINCUTNAIVE$ solves the minimum cut problem for graphs of at most $n - 1$ vertices, and we will prove that it does so for graphs of n

vertices. Indeed, under the inductive hypothesis, our recursive calls in step 3 return the minimum cuts S' and S'' of the $n-1$ vertex graphs G' and G'' respectively. But if v is the vertex we choose in step 2, we can think of G' as simply a graph where we "merged" s and v to be a single vertex with the neighbors of both s and v, and G'' as the graph where we merged t and v. So the s,t cuts in G' correspond to s,t cuts in G that don't separate s and v, while s,t cuts in G'' correspond to s,t cuts in G that don't separate t and v. Since in the minimum cut S, either s or t will be in the same side as v, the best one out of the minimal cuts from G' and G'' will be the minimum cut in G.

The running time $T(n)$ of $MINCUTNAIVE$ on n vertex graphs can be described by the recursive equation $T(n) = 2T(n-1) + f(n)$ where $f(n)$ is the time to execute the non recursive steps 1,2 and 4. In an algorithms course we might want to worry about the exact data structures we use to implement these steps, but for this course it is enough that we can do these steps in polynomial time (which is not hard to see). Nevertheless, this running time bound is terrible: even if $f(n)$ was equal to 1 we would still get that $T(n) \geq 2^n$! Indeed, this recursive algorithm is nothing but a fancy description of the trivial algorithm that enumerates over all the roughly 2^n (in fact, precisely 2^{n-2}) sets S that contain s and don't contain t.

Since minimum cut is a problem we want to solve, this seems like bad news. Luckily however we are able to find much faster algorithms that run in *polynomial time* (which, as mentioned in the mathematical background lecture, we denote by $poly(n)$) for this problem. There are several algorithms to do so, but many of them rely on the Max Flow Min Cut Theorem that says that the minimum cut between s and t equals the maximum amount of *flow* we can send from s to t, if every edge has unit capacity.[6] For example, this directly implies that the value of the minimum cut problem is the solution for the following linear program:[7]

$$\begin{aligned} \max_{x \in \mathbb{R}^m} & F_s(x) - F_t(x) \text{s.t.} \\ & \forall_{u \notin \{s,t\}} F_u(x) = 0 \end{aligned} \tag{13.1}$$

where for every vertex u and $x \in \mathbb{R}^m$, $F_u(x) = \sum_{e \in E \text{s.t.} u \in e} x_e$.

Since there is a polynomial-time algorithm for linear programming, the minimum cut (or, equivalently, maximum flow) problem can be solved in polynomial time. In fact, there are much better algorithms for this problem, with currently the record standing at $O(\min\{m^{10/7}, m\sqrt{n}\})$.[8]

[6] A *flow* of capacity c from a *source* s to a *sink* t in a graph can be thought of as describing how one would send some quantity from s to t in the graph (e.g., sending c liters of water (or any other matter one can partition arbitrarily), on pipes described by the edges). Mathematically, a flow is captured by assigning numbers to edges, and requiring that on any vertex apart from s and t, the amount flowing it is equal to the amount flowing out, while in s there are c units flowing out and in t there are c units flowing in.

[7] A *linear program* is the task of maximizing or minimizing a linear function of n real variables $x_0, \ldots, x_{n-1}$ subject to certain linear equalities and inequalities on the variables.

[8] TODO: add references in biliographical notes: Madry, Lee-Sidford

13.1.4 *Finding the maximum cut in a graph*

We can also define the *maximum cut* problem of finding, given a graph $G = (V, E)$ the subset $S \subseteq V$ that *maximizes* the number of edges cut by S.[9] Like its cousin the minimum cut problem, the maximum cut problem is also very well motivated. For example, it arises in VLSI design, and also has some surprising relation to analyzing the Ising model in statistical physics.

[9] We can also consider the variant where one is given s, t and looks for the s, t-cut that maximizes the number of edges cut. The two variants are equivalent up to $O(n^2)$ factors in the running time, but the we use the global max cut forumlation since it is more common in the literature.

Once again, a priori it might not be clear that the maximum cut problem should be harder than minimum cut but this turns out to be the case. We do not know of an algorithm that solves this problem much faster than the trivial "brute force" algorithm that tries all 2^n possibilities for the set S.

13.1.5 *A note on convexity*

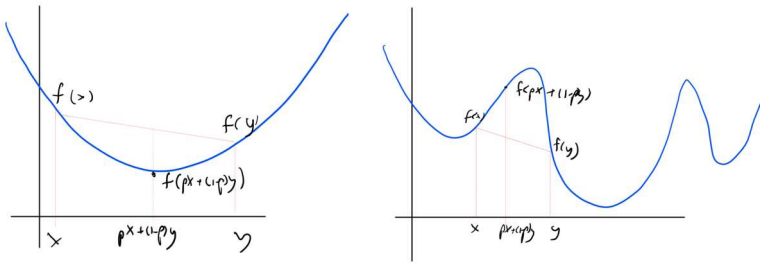

Figure 13.4: In a *convex* function f (left figure), for every x and y and $p \in [0,1]$ it holds that $f(px + (1-p)y) \leq p \cdot f(x) + (1-p) \cdot f(y)$. In particular this means that every *local minimum* of f is also a *global minimum*. In contrast in a *non convex* function there can be many local minima.

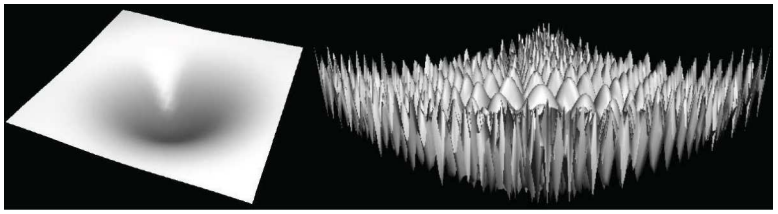

Figure 13.5: In the high dimensional case, if f is a *convex* function (left figure) the global minimum is the only local minimum, and we can find it by a local-search algorithm which can be thought of as dropping a marble and letting it "slide down" until it reaches the global minimum. In contrast, a non-convex function (right figure) might have an exponential number of local minima in which any local-search algorithm could get stuck.

There is an underlying reason for the sometimes radical difference between the difficulty of maximizing and minimizing a function over a domain. If $D \subseteq \mathbb{R}^n$, then a function $f : D \to R$ is *convex* if for every $x, y \in D$ and $p \in [0,1]$ $f(px + (1-p)y) \leq pf(x) + (1-$

$p)f(y)$. That is, f applied to the p-weighted midpoint between x and y is smaller than the p-weighted average value of f. If D itself is convex (which means that if x, y are in D then so is the line segment between them), then this means that if x is a *local minimum* of f then it is also a *global minimum*. The reason is that if $f(y) < f(x)$ then every point $z = px + (1-p)y$ on the line segment between x and y will satisfy $f(z) \leq pf(x) + (1-p)f(y) < f(x)$ and hence in particular x cannot be a local minimum. Intuitively, local minima of functions are much easier to find than global ones: after all, any "local search" algorithm that keeps finding a nearby point on which the value is lower, will eventually arrive at a local minima.[10] Indeed, under certain technical conditions, we can often efficiently find the minimum of convex functions, and this underlies the reason problems such as minimum cut and shortest path are easy to solve. On the other hand, *maximizing* a convex function (or equivalently, minimizing a *concave* function) can often be a hard computational task. A *linear* function is both convex and concave, which is the reason both the maximization and minimization problems for linear functions can be done efficiently.

[10] One example of such a local search algorithm is gradient descent which takes a small step in the direction that would reduce the value by the most amount based on the current derivative. There are also algorithms that take advantage of the *second derivative* (hence are known as *second order methods*) to potentially converge faster.

The minimum cut problem is not a priori a convex minimization task, because the set of potential cuts is *discrete*. However, it turns out that we can embed it in a continuous and convex set via the (linear) maximum flow problem. The "max flow min cut" theorem ensuring that this embedding is "tight" in the sense that the minimum "fractional cut" that we obtain through the maximum-flow linear program will be the same as the true minimum cut. Unfortunately, we don't know of such a tight embedding in the setting of the *maximum* cut problem.

The issue of convexity arises time and again in the context of computation. For example, one of the basic tasks in machine learning is *empirical risk minimization*. That is, given a set of labeled examples $(x_1, y_1), \ldots, (x_m, y_m)$, where each $x_i \in \{0,1\}^n$ and $y_i \in \{0,1\}$, we want to find the function $h : \{0,1\}^n \to \{0,1\}$ from some class H that minimizes the *error* in the sense of minimizing the number of i's such that $h(x_i) \neq y_i$. Like in the minimum cut problem, to make this a better behaved computational problem, we often embed it in a continuous domain, including functions that could output a real number and replacing the condition $h(x_i) \neq y_i$ with minimizing some continuous *loss function* $\ell(h(x_i), y_i)$.[11] When this embedding is *convex* then we are guaranteed that the global minimizer is unique and can be found in polynomial time. When the embedding is *non convex*, we have no such guarantee and in general there can be many global or local minima. That said, even if we don't find the global (or

[11] We also sometimes replace or enhance the condition that h is in the class H by adding a *regularizing term* of the form $R(h)$ to the minimization problem, where $R : H \to \mathbb{R}$ is some measure of the "complexity" of h. As a general rule, the larger or more "complex" functions h we allow, the easier it is to fit the data, but the more danger we have of "overfitting".

even a local) minima, this continuous embedding can still help us. In particular, when running a local improvement algorithm such as Gradient Descent, we might still find a function h that is "useful" in the sense of having a small error on future examples from the same distribution.[12]

[12] In machine learning parlance, this task is known as *supervised learning*. The set of examples $(x_1, y_1), \ldots, (x_m, y_m)$ is known as the *training set*, and the error on additional samples from the same distribution is known as the *generalization error*, and can be measured by checking h against a *test set* that was not used in training it.

13.2 Beyond graphs

Not all computational problems arise from graphs. We now list some other examples of computational problems that of great interest.

13.2.1 The 2SAT problem

A *propositional formula* φ involves n variables $x_1, \ldots, x_n$ and the logical operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$, also denoted as $\bar{\cdot}$). We say that such a formula is in *conjunctive normal form* (CNF for short) if it is an AND of ORs of variables or their negations (we call a term of the form x_i or $\overline{x}_i$ a *literal*). For example, this is a CNF formula

$$(x_7 \vee \overline{x}_{22} \vee x_{15}) \wedge (x_{37} \vee x_{22}) \wedge (x_{55} \vee \overline{x}_7) \tag{13.2}$$

We say that a formula is a k-CNF it is an AND of ORs where each OR involves exactly k literals. The 2SAT problem is to find out, given a 2-CNF formula φ, whether there is an assignment $x \in \{0,1\}^n$ that *satisfies* φ, in the sense that it makes it evaluate to 1 or "True".

Determining the satisfiability of Boolean formulas arises in many applications and in particular in software and hardware verification, as well as scheduling problems. The trivial, brute-force, algorithm for 2SAT will enumerate all the 2^n assignments $x \in \{0,1\}^n$ but fortunately we can do much better.

The key is that we can think of every constraint of the form $\ell_i \vee \ell_j$ (where ℓ_i, ℓ_j are *literals*, corresponding to variables or their negations) as an *implication* $\overline{\ell}_i \Rightarrow \ell_j$, since it corresponds to the constraints that if the literal $\ell'_i = \overline{\ell}_i$ is true then it must be the case that ℓ_j is true as well. Hence we can think of φ as a directed graph between the $2n$ literals, with an edge from ℓ_i to ℓ_j corresponding to an implication from the former to the latter. It can be shown that φ is unsatisfiable if and only if there is a variable x_i such that there is a directed path from x_i to $\overline{x}_i$ as well as a directed path from $\overline{x}_i$ to x_i (see Exercise 13.2). This reduces 2SAT to the (efficiently solvable) problem of determining connectivity in directed graphs.

13.2.2 *The 3SAT problem*

The 3SAT problem is the task of determining satisfiability for 3CNFs. One might think that changing from two to three would not make that much of a difference for complexity. One would be wrong. Despite much effort, do not know of a significantly better than brute force algorithm for 3SAT (the best known algorithms take roughy 1.3^n steps).

Interestingly, a similar issue arises time and again in computation, where the difference between two and three often corresponds to the difference between tractable and intractable. As Lawler's quote alludes to, we do not fully understand the reasons for this phenomenon, though the notions of **NP** completeness we will see later does offer a partial explanation. It may be related to the fact that optimzing a polynomial often amounts to equations on its derivative. The derivative of a a quadratic polynomial is linear, while the derivative of a cubic is quadratic, and, as we will see, the difference between solving linear and quadratic equations can be quite profound.

13.2.3 *Solving linear equations*

One of the most useful problems that people have been solving time and again is solving n linear equations in n variables. That is, solve equations of the form

$$\begin{aligned} a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} &= b_0 \\ a_{1,0}x_0 + a_{1,1}x_1 + \cdots + a_{1,n-1}x_{n-1} &= b_1 \\ \vdots + \vdots + \vdots &= \vdots \\ a_{n-1,0}x_0 + a_{n-1,1}x_1 + \cdots + a_{n-1,n-1}x_{n-1} &= b_{n-1} \end{aligned} \tag{13.3}$$

where $\{a_{i,j}\}_{i,j\in[n]}$ and $\{b_i\}_{i\in[n]}$ are real (or rational) numbers. More compactly, we can write this as the equations $Ax = b$ where A is an $n \times n$ matrix, and we think of x, b are column vectors in $\mathbb{R}^n$.

The standard Gaussian elimination algorithm can be used to solve such equations in polynomial time (i.e., determine if they have a solution, and if so, to find it).[13] As we discussed above, if we are willing to allow some loss in precision, we even have algorithms that handle linear *inequalities*, also known as linear programming. In contrast, if we insist on *integer* solutions, the task of solving for linear

[13] To analyze this fully we need to ensure that the bit complexity of the numbers involved does not grow too much, but fortunately we can indeed ensure this using . Also, as is usually the case when talking about real numbers, we do not care much for the distinction between solving equations exactly and solving them to arbitrarily good precision.

equalities or inequalities is known as integer programming, and the best known algorithms are exponential time in the worst case.

R **Bit complexity of numbers** Whenever we discuss problems whose inputs correspond to numbers, the input length corresponds to how many bits are needed to describe the number (or, as is equivalent up to a constant factor, the number of digits in base 10, 16 or any other constant). The difference between the length of the input and the magnitude of the number itself can be of course quite profound. For example, most people would agree that there is a huge difference between having a billion (i.e. 10^9) dollars and having nine dollars. Similarly there is a huge difference between an algorithm that takes n steps on an n-bit number and an algorithm that takes 2^n steps.

One example, is the problem (discussed below) of finding the prime factors of a given integer N. The natural algorithm is to search for such a factor by trying all numbers from 1 to N, but that would take N steps which is *exponential* in the input length, which is number of bits needed to describe N. [14] It is an important and long open question whether there is such an algorithm that runs in time polynomial in the input length (i.e., polynomial in $\log N$).

[14] The running time of this algorithm can be easily improved to roughly $\sqrt{N}$, but this is still exponential (i.e., $2^{n/2}$) in the number n of bits to describe N.

13.2.4 *Solving quadratic equations*

Suppose that we want to solve not just *linear* but also equations involving *quadratic* terms of the form $a_{i,j,k}x_jx_k$. That is, suppose that we are given a set of quadratic polynomials $p_1, \ldots, p_m$ and consider the equations $\{p_i(x) = 0\}$. To avoid issues with bit representations, we will always assume that the equations contain the constraints $\{x_i^2 - x_i = 0\}_{i\in[n]}$. Since only 0 and 1 satisfy the equation $a^2 - a$, this assumption means that we can restrict attention to solutions in $\{0,1\}^n$. Solving quadratic equations in several variable is a classical and extremely well motivated problem. This is the generalization of the classical case of single-variable quadratic equations that generations of high school students grapple with. It also generalizes the quadratic assignment problem, introduced in the 1950's as a way to optimize assignment of economic activities. Once again, we do not know a much better algorithm for this problem than the one that enumerates over all the 2^n possiblities.

13.3 More advanced examples

We now list a few more examples of interesting problems that are a little more advanced but are of significant interest in areas such as physics, economics, number theory, and cryptography.

13.3.1 The permanent (mod 2) problem

Given an $n \times n$ matrix A, the *permanent* of A is the sum over all permutations π (i.e., π is a member of the set S_n of one-to-one and onto functions from $[n]$ to $[n]$) of the product $\prod_{i=0}^{n-1} A_{i,\pi(i)}$. The permanent of a matrix is a natural quantity, and has been studied in several contexts including combinatorics and graph theory. It also arises in physics where it can be used to describe the quantum state of multiple boson particles (see here and here).

If the entries of A are integers, then we can also define a *Boolean* function $perm_2(A)$ which will output the result of the permanent modulo 2. A priori computing this would seem to require enumerating over all $n!$ possiblities. However, it turns out we can compute $perm_2(A)$ in polynomial time! The key is that modulo 2, $-x$ and $+x$ are the same quantity and hence the permanent moulo 2 is the same as taking the following quantity modulo 2:

$$\sum_{\pi \in S_n} sign(\pi) \prod_{i=0}^{n-1} A_{i,\pi(i)} \tag{13.4}$$

where the *sign* of a permutation π is a number in $\{+1, -1\}$ which can be defined in several ways, one of which is that $sign(\pi)$ equals $+1$ if the number of swaps that "Bubble" sort performs starting an array sorted according to π is even, and it equals -1 if this number is odd.[15]

From a first look, Eq. (13.4) does not seem like it makes much progress. After all, all we did is replace one formula involving a sum over $n!$ terms with an even more complicated formula involving a sum over $n!$ tersm. But fortunately Eq. (13.4) also has an alternative description: it is simply the determinant of the matrix A, which can be computed using a process similar to Gaussian elimination.

[15] It turns out that this definition is independent of the sorting algorithm, and for example if $sign(\pi) = -1$ then one cannot sort an array ordered according to π using an even number of swaps.

13.3.2 *The permanent (mod 3) problem*

Emboldened by our good fortune above, we might hope to be able to compute the permanent modulo any prime p and perhaps in full generality. Alas, we have no such luck. In a similar "two to three" type of a phenomenon, we do not know of a much better than brute force algorithm to even compute the permanent modulo 3.

13.3.3 *Finding a zero-sum equilibrium*

A *zero sum game* is a game between two players where the payoff for one is the same as the penalty for the other. That is, whatever the first player gains, the second player loses. As much as we want to avoid them, zero sum games do arise in life, and the one good thing about them is that at least we can compute the optimal strategy.

A zero sum game can be specified by an $n \times n$ matrix A, where if player 1 chooses action i and player 2 chooses action j then player one gets $A_{i,j}$ and player 2 loses the same amount. The famous Min Max Theorem by John von Neumann states that if we allow probabilistic or "mixed" strategies (where a player does not choose a single action but rather a *distribution* over actions) then it does not matter who plays first and the end result will be the same. Mathematically the min max theorem is that if we let Δ_n be the set of probability distributions over $[n]$ (i.e., non-negative columns vectors in $\mathbb{R}^n$ whose entries sum to 1) then

$$\max_{p \in \Delta_n} \min_{q \in \Delta_n} p^\top A q = \min_{q \in \Delta_n} \max_{p \in \Delta_n} p^\top A q \tag{13.5}$$

The min-max theorem turns out to be a corollary of linear programming duality, and indeed the value of Eq. (13.5) can be computed efficiently by a linear program.

13.3.4 *Finding a Nash equilibrium*

Fortunately, not all real-world games are zero sum, and we do have more general games, where the payoff of one player does not necessarily qual the loss of the other. John Nash won the Nobel prize for showing that there is a notion of *equilibrium* for such games as well. In many economic texts it is taken as an article of faith that when actual agents are involved in such a game then they reach a Nash equilibrium. However, unlike zero sum games, we do not know of an

efficient algorithm for finding a Nash equilibrium given the description of a general (non zero sum) game. In particular this means that, despite economists' intuitions, there are games for which natural stategies will take exponential number of steps to converge to an equilibrium.

13.3.5 *Primality testing*

Another classical computational problem, that has been of interest since the ancient greeks, is to determine whether a given number N is prime or composite. Clearly we can do so by trying to divide it with all the numbers in $2, \ldots, N-1$, but this would take at least N steps which is *exponential* in its bit complexity $n = \log N$. We can reduce these N steps to $\sqrt{N}$ by observing that if N is a composite of the form $N = PQ$ then either P or Q is smaller than $\sqrt{N}$. But this is still quite terrible. If N is a 1024 bit integer, $\sqrt{N}$ is about 2^{512}, and so running this algorithm on such an input would take much more than the lifetime of the universe.

Luckily, it turns out we can do radically better. In the 1970's, Rabin and Miller gave *probabilistic* algorithms to determine whether a given number N is prime or composite in time $poly(n)$ for $n = \log N$. We will discuss the probabilistic model of computation later in this course. In 2002, Agrawal, Kayal, and Saxena found a deterministic $poly(n)$ time algorithm for this problem. This is surely a development that mathematicians from Archimedes till Gauss would have found exciting.

13.3.6 *Integer factoring*

Given that we can efficiently determine whether a number N is prime or composite, we could expect that in the latter case we could also efficiently *find* the factorization of N. Alas, no such algorithm is known. In a surprising and exciting turn of events, the *non existence* of such an algorithm has been used as a basis for encryptions, and indeed it underlies much of the security of the world wide web. We will return to the factoring problem later in this course. We remark that we do know much better than brute force algorithms for this problem. While the brute force algorithms would require $2^{\Omega(n)}$ time to factor an n-bit integer, there are known algorithms running in time roughly $2^{O(\sqrt{n})}$ and also algorithms that are widely believed (though not fully rigorously analyzed) to run in time roughly $2^{O(n^{1/3})}$.[16]

[16] The "roughly" adjective above refers to neglecting factors that are polylogarithmic in n.

13.4 Our current knowledge

Figure 13.6: The current computational status of several interesting problems. For all of them we either know a polynomial-time algorithm or the known algorithms require at least 2^{n^c} for some $c > 0$. In fact for all except the *factoring* problem, we either know an $O(n^3)$ time algorithm or the best known algorithm require at least $2^{\Omega(n)}$ time where n is a natural parameter such that there is a brute force algorithm taking roughly 2^n or $n!$ time. Whether this "cliff" between the easy and hard problem is a real phenomenon or a reflection of our ignorane is still an open question.

The difference between an exponential and polynomial time algorithm might seem merely "quantiative" but it is in fact extremely significant. As we've already seen, the brute force exponential time algorithm runs out of steam very very fast, and as Edmonds says, in practice there might not be much difference between a problem where the best algorithm is exponential and a problem that is not solvable at all. Thus the efficient algorithms we mention above are widely used and power many computer science applications. Moreover, a polynomial-time algorithm often arises out of significant insight to the problem at hand, whether it is the "max-flow min-cut" result, the solvability of the determinant, or the group theoretic structure that enables primality testing. Such insight can be useful regardless of its computational implications.

At the moment we do not know whether the "hard" problems are truly hard, or whether it is merely because we haven't yet found the right algorithms for them. However, we will now see that there are problems that do *inherently require* exponential time. We just don't know if any of the examples above fall into that category.

13.5 Lecture summary

- There are many natural problems that have polynomial-time algorithms, and other natural problems that we'd love to solve, but for which the best known algorithms are exponential.
- Often a polynomial time algorithm relies on discovering some hidden structure in the problem, or finding a surprising equivalent formulation for it.
- There are many interesting problems where there is an *exponential gap* between the best known algorithm and the best algorithm that we can rule out. Closing this gap is one of the main open questions of theoretical computer science.

13.6 Exercises

Exercise 13.1 — exponential time algorithm for longest path. Give a $poly(n)2^n$ time algorithm for the longest path problem in n vertex graphs.[17] ■

Exercise 13.2 — 2SAT algorithm. For every 2CNF φ, define the graph G_φ on $2n$ vertices corresponding to the literals $x_1, \ldots, x_n, \overline{x}_1, \ldots, \overline{x}_n$, such that there is an edge $\overrightarrow{\ell_i \ell_j}$ iff the constraint $\overline{\ell}_i \vee \ell_j$ is in φ. Prove that φ is unsatisfiable if and only if there is some i such that there is a path from x_i to $\overline{x}_i$ and from $\overline{x}_i$ to x_i in G_φ. Show how to use this to solve 2SAT in polynomial time. ■

Exercise 13.3 — Regular expressions. A *regular* expression over the binary alphabet is a string consisting of the symbols $\{0, 1, \emptyset, (,), *, |\}$. An expression exp corresponds to a function mapping $\{0,1\}^* \to \{0,1\}$, where the `0` and `1` expressions correspond to the functions that map only output 1 on the strings `0` and `1` respectively, and if exp, exp' are expressions corresponnding to the functions, f' then $(exp)|(exp')$ corresponds to the function $f \vee f'$, $(exp)(exp')$ corresponds to the function g such that for $x \in \{0,1\}^n$, $g(x) = 1$ if there is some $i \in [n]$ such that $f(x_0, \ldots, x_i) = 1$ and $f(x_{i+1}, \ldots, x_{n-1}) = 1$, and $(exp)*$ corresponds to the function g such that $g(x) = 1$ if either $x = \emptyset$ or there are strings $x_1, \ldots, x_k$ such that x is the concatenation of $x_1, \ldots, x_k$ and $f(x_i) = 1$ for every $i \in [k]$.

Prove that for every regular expression exp, the function corresponding to exp is computable in polynomial time. Can you show that it is computable in $O(n)$ time? ■

[17] **Hint:** Use dynamic programming to compute for every $s, t \in [n]$ and $S \subseteq [n]$ the value $P(s, t, S)$ which equals to 1 if there is a simple path from s to t that uses exactly the vertices in S. Do this iteratively for S's of growing sizes.

13.7 *Bibliographical notes*

Eugene Lawler's quote on the "mystical power of twoness" was taken from the wonderful book "The Nature of Computation" by Moore and Mertens. See also this memorial essay on Lawler by Lenstra.

18

[18] TODO: add reference to best algorithm for longest path - probably the Bjorklund algorithm

13.8 *Further explorations*

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

13.9 *Acknowledgements*