The Computational Complexity of Linear Programming and Integer Linear Programming

CSCI 4113/6101
Instructor: Norbert Zeh
September 9, 2025

Now that we know what linear programs and integer linear programs are, and we got a glimpse of how to model optimization problems as LPs or ILPs, the next question is how we find optimal solutions for LPs and ILPs.

1 Linear Programming

We will soon discuss the Simplex Algorithm as a classical algorithm for solving LPs. The Simplex Algorithm is the most popular and one of the fastest algorithms for solving linear programs in practice. For some pathological inputs, however, the Simplex Algorithm may take exponential time and thus does not find a solution efficiently. Remarkably, any such pathological input has an almost identical input "nearby" that takes the Simplex Algorithm polynomial time to solve (Spielman and Teng 2004). Thus, we have to try really, really hard if we want to make the Simplex Algorithm take exponential time.

Still, in theory, the Simplex Algorithm is not a polynomial-time algorithm. The Ellipsoid Algorithm (Khachiyan 1979) and Karmarkar's Algorithm (Karmarkar 1984), not discussed in this course, *can* solve every linear program in polynomial time. The Ellipsoid Algorithm can do that even for LPs with an exponential number of constrains, provided these constraints are represented appropriately. The Ellipsoid Algorithm relies on a "separation oracle" to drive its search for an optimal solution. A separation oracle is an algorithm that decides whether a solution of the LP is feasible and, if not, produces a constraint of the LP violated by the solution. Even if the LP has an exponential number of constraints, it may be possible to implement a separation oracle for the LP that runs in polynomial time and space. If that's the case, then the Ellipsoid Algorithm solves the LP in polynomial time. In practice, the Ellipsoid Algorithm suffers from numerical instability and is much slower than both the Simplex Algorithm and Karmarkar's Algorithm.

Even though we do not discuss the Ellipsoid Algorithm or Karmarkar's algorithm in this course, the main take-away from this short section is that

Theorem 1. *Any linear program can be solved in polynomial time in its size.*

As a short example of how it is possible to implement a polynomial-time separation oracle for an LP with exponentially many constraints, consider one of the ILP formulations of the MST problem discussed in the previous topic:

$$\begin{aligned} &\text{Minimize } \sum_{e\in E} w_e x_e \\ \text{s.t. } &\sum_{e\in E} x_e = n-1 \\ &\sum_{e \text{ crosses } S} x_e \geq 1 \qquad \forall \text{cut } S \text{ in } G \\ &x_e \in \{0,1\} \quad \forall e \in E. \end{aligned} \tag{1}$$

The Ellipsoid algorithm cannot solve *integer* linear programs, so let us instead consider the following linear program:

$$\begin{aligned} &\text{Minimize } \sum_{e\in E} w_e x_e \\ \text{s.t. } &\sum_{e\in E} x_e = n-1 \\ &\sum_{e \text{ crosses } S} x_e \geq 1 \qquad \forall \text{cut } S \text{ in } G \\ &x_e \leq 1 \qquad \forall e \in E \\ &x_e \geq 0 \qquad \forall e \in E. \end{aligned} \tag{2}$$

As we will discuss in the next topic, this LP is called the "LP relaxation" of (1).

Since there are $2^n - 2$ cuts in an n-vertex graph, this LP has exponentially many constraints. However, they do not need to be stored explicitly. All we have to do is to store the graph G itself. A separation oracle for (1) would be very easy to design. A separation oracle for the LP relaxation (2) takes a little more effort.

Given a solution $\hat{x}$ of this ILP, the oracle can test in linear time whether $\sum_{e\in T} x_e = n-1$. If not, then it reports the first constraint as a constraint violated by this solution. To test whether there exists a cut S whose corresponding constraint in (2) is violated, we can find a cut S that minimizes $\sum_{e \text{ crosses } S} \hat{x}_e$. Such a cut is called a **minimum cut**. If this cut satisfies $\sum_{e \text{ crosses } S} \hat{x}_e \geq 1$, then all cuts do, that is $\hat{x}$ is a feasible solution. If it doesn't, then we report the constraint corresponding to this cut as a violated constraint.

But how do we find such a cut? To discuss this, we need to understand network flows, which we will discuss after our discussion of linear programs. A maximum flow in a network can be computed in polynomial time, and one of the exercises in the part of the course discussing maximum flows will ask you to prove that this implies that a minumum cut can also be found in polynomial time.

2 Integer Linear Programming

So we can solve linear programs in polynomial time. What about *integer linear programs?* In this section, we prove that solving integer linear programs is NP-hard, even if they have only a polynomial number of constraints.

Theorem 2. *Integer linear programming is NP-hard.*

Proof. Recall the basic strategy for proving that a problem Π is NP-hard. It suffices to provide a **polynomial-time reduction** from an NP-hard problem Π' to Π. Specifically, we need to provide a

polynomial-time algorithm $\mathcal{A}$ that takes any instance I' of Π' and computes from it an instance I of Π such that I is a yes-instance of Π if and only if I' is a yes-instance of Π'. If you have not seen any NP-hardness proofs yet or you need a refresher on how they work, the appendix provides a (not so) brief introduction to the topic.

Here, we choose Π to be ILP and Π' to be the satisfiability problem (SAT). A SAT instance is a Boolean formula in CNF over some set of Boolean variables $x_1, \ldots, x_n$:

$$\begin{aligned} F &= C_1 \wedge \cdots \wedge C_m \\ C_i &= \lambda_{i1} \vee \cdots \vee \lambda_{ik_i} && \forall 1 \leq i \leq m \\ \lambda_{ij} &\in \{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\} && \forall 1 \leq i \leq m, 1 \leq j \leq k_i. \end{aligned}$$

$\bar{x}_i$ denotes the negation of x_i.

We translate F into an ILP over variables $x_1, \ldots, x_n \in \{0, 1\}$. Each literal λ_{ij} is translated into a linear function $\phi_{ij}(x_1, \ldots, x_n)$ defined as

$$\phi_{ij}(x_1, \ldots, x_n) = \begin{cases} x_k & \text{if } \lambda_{ij} = x_k \\ 1 - x_k & \text{if } \lambda_{ij} = \bar{x}_k \end{cases}.$$

Each clause C_i is translated into the constraint

$$\Phi_i(x_1, \ldots, x_n) \geq 1,$$

where

$$\Phi_i(x_1, \ldots, x_n) = \sum_{j=1}^{k_i} \phi_{ij}(x_1, \ldots, x_n).$$

It turns out that even deciding whether this ILP has a *feasible* solution is NP-hard, so we can choose any objective function, such as $f(x_1, \ldots, x_n) = 0$, to complete the definition of the ILP.

To prove that deciding whether this ILP has a feasible solution is NP-hard, we prove that F is satisfiable if and only if the ILP has a feasible solution:

First assume that F is satisfiable. Then there exists an assignment $\hat{x} = (\hat{x}_1, \ldots, \hat{x}_n)$ of truth values to the variables $x_1, \ldots, x_n$ in F such that every clause C_i contains a true literal λ_{ij}. For this literal, $\phi_{ij}(\hat{x}_1, \ldots, \hat{x}_n) = 1$. Thus, since $\phi_{ij'}(\hat{x}_1, \ldots, \hat{x}_n) \geq 0$ for all $1 \leq j' \leq k_i$, $\Phi_i(\hat{x}_1, \ldots, \hat{x}_n) = \sum_{j'=1}^{k_i} \phi_{ij'}(\hat{x}_1, \ldots, \hat{x}_n) \geq 1$, that is, $\hat{x}$ satisfies the constraint corresponding to C_i in the ILP. Since $\hat{x}$ satisfies every clause of F, this shows that it satisfies every constraint of the ILP, so $\hat{x}$ is a feasible solution of the ILP.

Now assume that the ILP is feasible. Then there exists a feasible solution $\hat{x} = (\hat{x}_1, \ldots, \hat{x}_n)$ of the ILP. Since $\hat{x}_i \in \{0, 1\}$ for all $1 \leq i \leq n$, we can interpret this solution as an assignment of truth values to the variables $x_1, \ldots, x_n$ in F. Since $\hat{x}$ is a feasible solution of the ILP, we have $\Phi_i(\hat{x}_1, \ldots, \hat{x}_n) = \sum_{j'=1}^{k_i} \phi_{ij'}(\hat{x}_1, \ldots, \hat{x}_n) \geq 1$, for every clause C_i. Thus, $\phi_{ij}(\hat{x}_1, \ldots, \hat{x}_n) = 1$ for at least one literal $\lambda_{ij} \in C_i$, that is, λ_{ij} is made true by the solution $\hat{x} = (\hat{x}_1, \ldots, \hat{x}_n)$ and, thus, C_i is satisfied by $\hat{x}$. Since $\hat{x}$ satisfies every constraint in the ILP, it also satisfies every clause C_i in F, that is, it is a satisfying truth assignment of F.

We have shown that we can, in polynomial time, convert any Boolean formula F into an ILP that is feasible if and only if F is satisfiable. This conversion is a polynomial-time reduction from SAT to ILP.

Since SAT is NP-hard, this proves that ILP is NP-hard. □

A NP-HARDNESS

Designing algorithms is (to a large degree) about finding the fastest possible way to solve a given problem Π. To verify that we have found the fastest possible algorithm, we need to prove that there is no faster algorithm to solve Π than the one we have found. For example, we can prove that $\Omega(n \lg n)$ is the best possible running time achievable for sorting a sequence of n elements if we use only comparisons to decide in which order the input elements should be arranged. Merge Sort, Quicksort, and Heap Sort all run in $O(n \lg n)$ time and use only comparisons to determine the correct order of the input elements. In that sense, they are optimal.

For most problems which we don't even know how to solve in *polynomial* time—any polynomial will do—we also *don't* know how to prove that there is no polynomial-time algorithm to solve them. The best we know how to do is to provide evidence that it is, in a formal mathematical sense, highly unlikely that a polynomial-time algorithm solving such a problem exists. NP-hardness is the tool we use to do this.

This appendix gives a brief overview of NP-hardness. We start by defining decision problems and exploring their relationship to optimization problems. Then we briefly touch on models of computation, which are necessary to formalize the computational difficulty of problems. Next we defining the complexity classes Pand NP. Then we define what it means for a problem to be NP-hard. We conclude with a discussion of how to use polynomial-time reductions to prove that a problem is NP-hard.

A.1 DECISION VS OPTIMIZATION

Both P and NP are classes of decision problems. A **decision problem** is a problem with a yes/no answer: Is this graph connected? Is this input sequence sorted? Does this graph have a vertex cover of size at most 20?

An instance I of a decision problem Π is a **yes-instance** if the answer to the question asked by Π on input I is yes. Otherwise, I is a **no-instance**.

While we are normally interested in problems that have more complex answers than simply yes or no, the importance of decision problems as a central tool for studying the difficulty of computational problems stems from two observations.

First, we can translate any optimization problem Π_o into a corresponding decision problem Π_d Specifically, if Π_o is a minimization problem, we can turn it into a decision problem Π_d by pairing every instance I of Π_o with a real number k. Π_d asks then whether I has a solution S with $f(S) \leq k$. For a maximization problem, we'd ask whether I has a solution S with $f(S) \geq k$.

The vertex cover example above is such a decision problem derived from an optimization problem. In the vertex cover problem, we normally look for the smallest possible vertex cover. Here, we only want to know whether the input graph, the input instance I, has a vertex cover of size at most 20. Whether this vertex cover has size 2, 11 or exactly 20 is immaterial, as is the exact set of vertices in such a cover. All we want to know is whether a vertex cover of size at most 20 exists—we don't need to find it. Similarly, we can ask whether a given graph has a spanning tree of weight at most 523 or whether there exists a path between two vertices of length at most 19.1. These are the decision versions of the minimum spanning tree and shortest path problems.

Clearly, any algorithm $\mathcal{A}$ that solves an optimization problem Π_o can be used to also solve the corresponding decision problem Π_d by comparing the objective function value $f(S)$ of the optimal solution S computed by $\mathcal{A}$ with the threshold k given as part of the instance (I, k) of Π_d.

OBSERVATION 3. *If an optimization problem Π_o can be solved in $T(n)$ time, then its corresponding decision problem Π_d can be solved in $O(T(n))$ time.*

Thus, if we can provide evidence that Π_d is hard to solve, then the same must be true for Π_o. In particular, if there exists a polynomial-time algorithm that solves Π_o, then there also exists a polynomial-time algorithm that solves Π_d. We are usually interested in the contrapositive: If there is no polynomial-time algorithm that solves Π_d, then there does not exist a polynomial-time algorithm that solves Π_o either.

The second useful property of decision problems is that they are equivalent to formal languages: Any formal language $\mathcal{L}$ over some alphabet Σ gives rise to a decision problem $\Pi_{\mathcal{L}}$: Given a string $\sigma \in \Sigma^*$, decide whether $\sigma \in \mathcal{L}$. Conversely, any decision problem Π can be translated into a formal language $\mathcal{L}_\Pi$: We choose some encoding of the input instances of the problem Π over some alphabet Σ. For example, if we want to solve the problem using a real computer, the computer needs to store the input I in memory, which means it encodes I as a binary string σ_I. The language $\mathcal{L}_\Pi$ is then the language of all strings σ_I that encode yes-instances.

This equivalence between decision problems and formal languages gives us a clean model to study the computational difficulty of decision problems. For example, recall from CSCI 2115 that a formal language can be decided by a DFA (the corresponding decision problem can be solved by a DFA) if and only if it is regular.

A.2 MODELS OF COMPUTATION

How hard a problem is to solve depends on the operations we are allowed to use in our algorithm. The sorting problem mentioned briefly in the introduction of this appendix is a case in point: There is no algorithm that correctly sorts all inputs in $o(n \lg n)$ time and uses only comparisons to determine the correct order of the input elements. Either the algorithm uses operations other than comparisons or there must exist an input that forces it to take $\Omega(n \lg n)$ time. If we want to sort integers and use the integer values themselves as indices into an array, then Counting Sort allows us to sort integers between 1 and $m = O(n)$ in $O(n)$ time. Radix Sort allows us to extend the range to $m = O(n^c)$ for any constant c. Bucket Sort allows us to sort even arbitrary numbers in expected $O(n)$ time, provided they are distributed uniformly over a given interval. All of these faster algorithms inspect the actual *values* of the numbers to be sorted, something we cannot do using comparisons only.

As another example, mentioned above, only regular languages can be decided by a DFA. If we want to decide a non-regular language at all, let alone efficiently, we need a machine more powerful than a DFA. The bottom line is that to study whether a problem can be solved at all or whether it can be solved efficiently, we need to agree on what operations our algorithm is allowed to use.

The two most commonly considered models of computation when studying the computational complexity of problems are the **Turing Machine** and the **Random Access Machine (RAM)**. In the interest of keeping this appendix short, I will not discuss these models in detail here. All you need to know is that:

- The Random Access Machine captures fairly closely the capabilities of actual computers. It supports comparisons, basic arithmetic operations, random access into memory, etc. There are subtle differences that lead to the distinction between different types of Random Access Machines. For example, the exact set of arithmetic operations that are allowed—do we have a square-root function and floor and ceiling functions or not—has a significant impact on the time it takes to solve certain geometric problems. If we assume that the objects the RAM can manipulate in constant time are real numbers (not just floating point numbers of limited precision), then we have to be very careful not to abuse this ability because we can pack an unlimited amount of information into a single real number; by manipulating real numbers in constant time then, we can circumvent many lower bounds.

- A Turing Machine is a finite automaton equipped with a **tape** that extends infinitely in both directions and acts as the machine's memory. The Turing Machine accesses this tape using a **read-write head** that it can move across the cells (memory locations) of the tape. The input of the Turing Machine is stored on the tape and the read-write head is initially positioned at the beginning of the input. In each step, the Turing Machine reads the cell of the tape where the read-write head is currently located. Based on the content of this cell and on its current state, it replaces the cell content with a new symbol in its alphabet, leaves the read-write head where it is or moves it one position to the left or right, and transitions to a new state. It may also to decide to stop its computation. When it does, it accepts or rejects the input based on whether the current state is accepting or non-accepting.

- The Random Access Machine is clearly more powerful than the Turing Machine. If we want to access a certain memory location, we can do this in constant time on a Random Access Machine, while it may take quite some time to move the read-write head of the Turing Machine to this memory location. In spite of this difference in memory access cost, we have the following important equivalence between Turing Machines and Random Access Machines:

  THEOREM 4. *A language can be decided in polynomial time by a Random Access Machine if and only if it can be decided in polynomial time by a Turing Machine.*

  Thus, when studying which problems can be solved in polynomial time, we can focus on Turing Machines. This is useful because Turing Machines are more primitive than Random Access Machines; they support fewer operations in constant time. This makes them easier to reason about formally.

A.3 P

Now that we know what a Turing Machine is, the complexity class P is easy to define.

DEFINITION 5. P is the class of all formal languages that can be decided in polynomial time by a deterministic Turing machine.

Given Theorem 4, P is also exactly the class of all formal languages that can be decided by a RAM in polynomial time. Given the equivalence between formal languages and decision problems, we often simply say that P is the class of all decision problems that can be solved in polynomial time (without the

specific reference to a deterministic Turing machine or RAM and without expressing decision problems as formal languages).

A.4 NP

NP is a class of formal languages that is a superset of P. It can be defined in at least two equivalent ways.

A.4.1 Non-Deterministic Polynomial Time

One way to define it is as the class of all formal languages that can be decided in polynomial time by a *non-deterministic* Turing machine. Indeed, NP stands for "**N**on-deterministic **P**olynomial time". You should have learned about non-deterministic Turing machines in CSCI 2115. In contrast to a deterministic Turing machine, whose behaviour in each step is completely determined by the current state of the Turing machine and the character under the read-write head, the transition function of a non-deterministic Turing machine associates only a *set* of possible actions to be taken with each such state-character pair. (An action is characterized by the new character to write in the current tape cell, the new state to transition to, and the direction into which to move the read-write head). The Turing machine can then choose an arbitrary action from this set. This is the non-deterministic part of its behaviour.

A consequence of this definition is that the exact computation carried out by the Turing machine depends on the actions the machine chooses to take whenever there is more than one action to choose from. Some set of these choices may lead the Turing machine to accept the input. Another set of choices may lead the Turing machine to reject the very same input. We say that a non-deterministic Turing machine decides a language $\mathcal{L}$ if a string $\sigma \in \Sigma^*$ belongs to $\mathcal{L}$ if and only if there exists a set of choices the Turing machine can make that lead it to accept σ. Conversely, a string σ does not belong to $\mathcal{L}$ only if no matter the choices the Turing machine makes, it always rejects σ.

A.4.2 Polynomial-Time Verification

Most students do not feel very comfortable with the concept of non-determinism. So you may like the following completely equivalent definition of NPin terms of *deterministic* Turing machines better.

Consider a language $\mathcal{L} \subseteq \Sigma^*$ and let $\mathcal{L}' \subseteq \Sigma^*$ be a language such that $\sigma \in \mathcal{L}$ if and only if there exists a string $\tau \in \Sigma^*$ such that $\sigma\tau \in \mathcal{L}'$. You can think about τ as a "proof" that $\sigma \in \mathcal{L}$ and about $\mathcal{L}'$ as the language of all pairs (σ, τ) such that τ is a valid proof of σ's membership in $\mathcal{L}$. Correspondingly, we say that any Turing Machine M that decides $\mathcal{L}'$ **verifies** $\mathcal{L}$: given a pair (σ, τ), it verifies whether τ is a valid proof of σ's membership in $\mathcal{L}$. It may be that M rejects the input (σ, τ) even though $\sigma \in \mathcal{L}$. This happens when $(\sigma, \tau) \notin \mathcal{L}'$. This only means that τ fails to prove that $\sigma \in \mathcal{L}$, not that $\sigma \notin \mathcal{L}$. In particular, there may exist another string $\tau' \in \Sigma^*$ such that $(\sigma, \tau') \in \mathcal{L}'$.

As an example, we can choose $\mathcal{L}$ to be the language of all (encodings of) graph-integer pairs (G, k) such that G has a vertex cover of size k. For every string $\sigma \in \mathcal{L}$ encoding a yes-instance (G, k), choose a set of at most k vertices that cover all the edges of G, and let τ be an appropriate encoding of the identities of these vertices. Then $\mathcal{L}'$ is the language of all such pairs (σ, τ). It is easy to construct a deterministic Turing Machine that takes such a pair of strings (σ, τ) and decides in polynomial time whether τ encodes a set of at most k vertices and whether these k vertices cover all edges in the graph

G encoded by σ. In contrast, we don't know whether there exists a deterministic Turing Machine that decides in polynomial time whether (G, k) is a yes-instance. In fact, we have strong reasons to believe that this is highly unlikely.

The following is a definition of NP in terms of polynomial-time verification of formal languages:

DEFINITION 6. NP is the class of all formal languages that can be verified in polynomial time by a deterministic Turing Machine.

This notion of polynomial-time verification of a language $\mathcal{L}$ takes some care to define. We know that for a deterministic Turing Machine M to verify $\mathcal{L}$, M must decide a language $\mathcal{L}'$ such that $\sigma \in \mathcal{L}$ if and only if $(\sigma, \tau) \in \mathcal{L}'$ for some $\tau \in \Sigma^*$. For this to be a polynomial-time verification, we need to stipulate that M runs in polynomial time in its input size, but this is not enough: We want that deciding whether $(\sigma, \tau) \in \mathcal{L}'$ takes time polynomial in $|\sigma|$, not only time polynomial in $|\sigma\tau|$! If the length of τ is exponential in $|\sigma|$, then even if M takes polynomial time in $|\sigma\tau|$, it may take exponential time in $|\sigma|$ to decide whether $(\sigma, \tau) \in \mathcal{L}'$, simply because τ has exponential length. So what we need is that there exists a "short" proof that $\sigma \in \mathcal{L}$: $\sigma \in \mathcal{L}$ if and only if there exists a string τ whose length is only polynomial in $|\sigma|$ and such that $(\sigma, \tau) \in \mathcal{L}'$.

To summarize, we say that a language $\mathcal{L} \subseteq \Sigma^*$ can be **verified in polynomial time** by a deterministic Turing Machine if there exist constants c and d, a language $\mathcal{L}' \subseteq \Sigma^*$, and a deterministic Turing Machine M such that

- A string $\sigma \in \Sigma^*$ belongs to $\mathcal{L}$ if and only if there exists a string τ of length $|\tau| \leq |\sigma|^c$ such that $(\sigma, \tau) \in \mathcal{L}'$, and
- Given any string $\rho \in \Sigma^*$, M takes $O(|\rho|^d)$ time to decide whether $\rho \in \mathcal{L}'$.

A.4.3 EQUIVALENCE OF THE TWO DEFINITIONS OF NP

Why are the two definitions of NP in terms of deciding a language in non-deterministic polynomial time or verifying the language in deterministic polynomial time equivalent? It is easy to prove that a language can be verified in deterministic polynomial time if and only if it can be decided in non-deterministic polynomial time:

From non-deterministic decision to deterministic verification: Assume we have a non-deterministic Turing Machine M that decides the language $\mathcal{L}$. Also assume that M produces an answer in at most $|\sigma|^c$ time for any string $\sigma \in \Sigma^*$ and whenever M makes a non-deterministic choice, it chooses from a constant number a of options. The choices made by M during some computation of length at most $|\sigma|^c$ can be encoded using an a-ary number τ with at most $|\sigma|^c$ digits. We define a language $\mathcal{L}'$ consisting of all pairs (σ, τ) such that M accepts σ if it makes the choices encoded by τ. This guarantees that $\sigma \in \mathcal{L}$ if and only if there exists a string τ such that $(\sigma, \tau) \in \mathcal{L}'$. Moreover, if $\sigma \in \mathcal{L}$, then there exists a string τ of length $|\tau| \leq |\sigma|^c$ such that $(\sigma, \tau) \in \mathcal{L}'$. Thus, any deterministic Turing machine that runs in polynomial time and decides $\mathcal{L}'$ provides a polynomial-time verification of $\mathcal{L}$.

Such a Turing machine M' is easy to construct. Given an input (σ, τ), M' performs the same computation as M performs on σ. Whenever M would make a non-deterministic choice, M' reads the next digit from τ and (deterministically) chooses the corresponding choice from the options available

to M. This guarantees that M' runs in polynomial time. It also guarantees that M' accepts (σ, τ) if and only if the choices encoded by τ lead M to accept σ, that is, if $(\sigma, \tau) \in \mathcal{L}'$—$M'$ decides $\mathcal{L}'$.

From deterministic verification to non-deterministic decision: For the other direction, assume we have constants c and d, a language $\mathcal{L}'$, and a deterministic Turing Machine M such that

- $\sigma \in \mathcal{L}$ if and only if there exists a string τ of length $|\tau| \leq |\sigma|^c$ such that $(\sigma, \tau) \in \mathcal{L}'$,
- M decides $\mathcal{L}'$, and
- Given any string ρ, M produces an answer in at most $|\rho|^d$ time.

We obtain a non-deterministic Turing Machine M' that decides $\mathcal{L}$ in polynomial time as follows:

Given an input $\sigma \in \Sigma^*$, M' starts by generating a string $\tau \in \Sigma^*$ of length $|\tau| \leq |\sigma|^c$. It "guesses" this string using its ability to make non-deterministic choices. It then runs M on the input (σ, τ) and returns the result. This machine clearly runs in polynomial time in $|\sigma|$. Does it decide $\mathcal{L}$?

Given a string $\sigma \in \mathcal{L}$, there exists a string τ of length at most $|\sigma|^c$ such that $(\sigma, \tau) \in \mathcal{L}'$. One possible set of choices that M' is able to make generates exactly this string τ and running M on the resulting string (σ, τ) then leads M' to accept σ.

If $\sigma \notin \mathcal{L}$, then no matter which string τ of length at most $|\sigma|^c$ M' guesses, we have $(\sigma, \tau) \notin \mathcal{L}'$. Thus, running M on (σ, τ) leads M' to reject σ. Therefore, M' rejects σ for any possible set of non-deterministic choices it makes.

A.5 NP-Hardness and NP-Completeness

Given the definitions of P and NP, it is obvious that $\mathsf{P} \subseteq \mathsf{NP}$: every language that can be decided in polynomial time can also be verified in polynomial time. Indeed, given a deterministic Turing Machine M that decides $\mathcal{L}$ in polynomial time, we can use it to verify $\mathcal{L}$ in polynomial time. Given a pair of strings (σ, τ), we simply ignore τ and run M on σ. If $\sigma \in \mathcal{L}$, this procedure accepts σ no matter the choice of τ. If $\sigma \notin \mathcal{L}$, it rejects σ no matter the choice of τ. This meets the requirements of polynomial-time verification of $\mathcal{L}$.

One of the most tantalizing questions in Computer Science is whether this inclusion is strict, whether $\mathsf{P} \neq \mathsf{NP}$. While nobody has a conclusive answer to this question, the evidence is overwhelming that the answer should be yes. I'll comment on this in a little more detail below. For now, let us assume that $\mathsf{P} \neq \mathsf{NP}$. Then the following definition captures the idea that a decision problem cannot be solved in polynomial time.

Definition 7. A formal language $\mathcal{L}$ (decision problem Π) is **NP-hard** if $\mathcal{L} \in \mathsf{P}$ implies that $\mathsf{P} = \mathsf{NP}$. $\mathcal{L}$ is **NP-complete** if $\mathcal{L}$ is NP-hard and $\mathcal{L} \in \mathsf{NP}$.

Stated differently, this says that if we can find a polynomial-time algorithm to solve an NP-hard decision problem Π, then $\mathsf{P} = \mathsf{NP}$. If we believe that $\mathsf{P} \neq \mathsf{NP}$, then no such polynomial-time algorithm to solve Π can exist.

To prove that a problem is unlikely to have a polynomial-time solution—unless $\mathsf{P} = \mathsf{NP}$—it suffices to prove that the problem is NP-hard. An NP-hard problem does not even have to be known to be in NP. For a problem to be NP-complete, it also has to be in NP. This captures the idea that NP-complete

problems are the hardest problems in NP: we know how to verify a solution in polynomial time but we cannot find such a solution in polynomial time unless $\mathsf{P} = \mathsf{NP}$.

So what's this evidence that $\mathsf{P} = \mathsf{NP}$ is highly unlikely? Since every NP-complete problem is itself in NP, the conclusion that $\mathsf{P} = \mathsf{NP}$ would imply that we can solve every NP-complete problem in polynomial time. In other words, by coming up with a polynomial-time algorithm for just one NP-hard problem, we obtain for free polynomial-time algorithms for *all* NP-complete problems. There are by now thousands of problems that are known to be NP-complete and which have eluded all attempts by the brightest minds in Computer Science to design polynomial-time algorithms for these problems over the last six decades. Do we really believe that we can obtain polynomial-time algorithms for all of them in one fell swoop, by solving just one of them in polynomial time?

This is not a proof that $\mathsf{P} \neq \mathsf{NP}$. You can choose to believe that $\mathsf{P} = \mathsf{NP}$ or that $\mathsf{P} \neq \mathsf{NP}$. Whatever you choose to believe, however, if a problem is NP-hard, then the stakes are very high when searching for a polynomial-time algorithm for this problem: if you succeed, you prove that $\mathsf{P} = \mathsf{NP}$ and thereby settle one of the most iconic questions in Computer Science. You'd also become a millionaire because there is a $1M prize for answering this question.

A.6 Polynomial-Time Reductions

Now that we know that an NP-hard problem is unlikely to have a polynomial-time algorithm that solves it, how do we prove that a problem is NP-hard?

A problem that is trivially NP-hard is the following:

$$A_{\text{TM}}^{\text{poly}} = \{(M, c, \sigma) \mid M \text{ is a non-deterministic Turing machine that accepts } \sigma \text{ in at most } |\sigma|^c \text{ time}\}.$$

Assume that $\mathcal{A}$ is a deterministic algorithm that decides $A_{\text{TM}}^{\text{poly}}$ and does so in at most $|\tau|^d$ time, for any string $\tau \in \Sigma^*$ that encodes a triple (M, c, σ). Consider any language $\mathcal{L} \in \mathsf{NP}$. This implies that there exists a non-deterministic Turing machine M such that M accepts every string $\sigma \in \mathcal{L}$ in at most $|\sigma|^c$ time, for some constant c, and does not accept any string $\sigma \notin \mathcal{L}$. In particular, $\sigma \in \mathcal{L}$ if and only if $(M, c, \sigma) \in A_{\text{TM}}^{\text{poly}}$. Since the definition of M does not depend on σ, M has a constant-size description, that is, the encoding of the tuple (M, c, σ) has length $|\sigma| + O(1)$, for any string $\sigma \in \Sigma^*$. The algorithm $\mathcal{A}$ takes at most $(|\sigma| + O(1))^d = O(|\sigma|^d)$ time to decide whether $(M, c, \sigma) \in A_{\text{TM}}^{\text{poly}}$ and, thereby, whether $\sigma \in \mathcal{L}$. Thus, we can use $\mathcal{A}$ to decide membership in $\mathcal{L}$ deterministically in polynomial time, that is, $\mathcal{L} \in \mathsf{P}$. Since this is true for every language $\mathcal{L} \in \mathsf{NP}$, this proves that $\mathsf{NP} \subseteq \mathsf{P}$ and, therefore, since $\mathsf{P} \subseteq \mathsf{NP}$, that $\mathsf{P} = \mathsf{NP}$. Therefore, $A_{\text{TM}}^{\text{poly}}$ is NP-hard.

With the first NP-problem in hand, we can now prove that other problems are NP-hard by using polynomial-time reductions.

Definition 8. A **reduction** from some language $\mathcal{L}_1$ to another language $\mathcal{L}_2$ is a deterministic algorithm $\mathcal{R}$ that converts any input string $\sigma \in \Sigma^*$ into a string $\mathcal{R}(\sigma) \in \Sigma^*$ such that $\sigma \in \mathcal{L}_1$ if and only if $\mathcal{R}(\sigma) \in \mathcal{L}_2$. $\mathcal{R}$ is a **polynomial-time reduction** if it runs in $O(|\sigma|^c)$ time, for some constant c.

The usefulness of polynomial-time reductions to prove NP-hardness stems from the following lemma and its corollary.

Lemma 9. *If $\mathcal{L}_2 \in \mathsf{P}$ and there exists a polynomial-time reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$, then $\mathcal{L}_1 \in \mathsf{P}$.*

Proof. Assume that $\mathcal{R}$ is a polynomial-time reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$. Let c be a constant such that $\mathcal{R}$ runs in at most $|\sigma|^c$ time on any input σ. Since just writing down $\mathcal{R}(\sigma)$ takes $|\mathcal{R}(\sigma)|$ time, we have $|\mathcal{R}(\sigma)| \leq |\sigma|^c$.

Next assume that $\mathcal{L}_2 \in \mathsf{P}$. Then there exists an algorithm $\mathcal{D}$ that takes $|\sigma|^d$ time to decide whether any given string $\sigma \in \Sigma^*$ belongs to $\mathcal{L}_2$.

We can use $\mathcal{R}$ and $\mathcal{D}$ to decide whether a string $\sigma \in \Sigma^*$ belongs to $\mathcal{L}_1$: We use $\mathcal{R}$ to compute $\mathcal{R}(\sigma)$ from σ and then apply $\mathcal{D}$ to $\mathcal{R}(\sigma)$ to decide whether $\mathcal{R}(\sigma) \in \mathcal{L}_2$. Since $\mathcal{R}$ is a reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$, we have $\sigma \in \mathcal{L}_1$ if and only if $\mathcal{R}(\sigma) \in \mathcal{L}_2$, that is, the answer given by $\mathcal{D}$ for the input $\mathcal{R}(\sigma)$ decides whether $\sigma \in \mathcal{L}_1$. We thus have a correct decision algorithm for $\mathcal{L}_1$.

The running time of this algorithm is polynomial in $|\sigma|$: Running $\mathcal{R}$ on σ takes at most $|\sigma|^c$ time. Running $\mathcal{D}$ on $\mathcal{R}(\sigma)$ takes at most $|\mathcal{R}(\sigma)|^d$ time. Since $|\mathcal{R}(\sigma)| \leq |\sigma|^c$, running $\mathcal{D}$ on $\mathcal{R}(\sigma)$ thus takes at most $|\sigma|^{cd}$ time. The total running time of the algorithm is therefore $|\sigma|^c + |\sigma|^{cd}$, which is polynomial in $|\sigma|$. □

COROLLARY 10. *If $\mathcal{L}_1$ is NP-hard and there exists a polynomial-time reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$, then $\mathcal{L}_2$ is also NP-hard.*

Proof. By Lemma 9, the existence of a polynomial-time reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$ means that $\mathcal{L}_1 \in \mathsf{P}$ if $\mathcal{L}_2 \in \mathsf{P}$. Since $\mathcal{L}_1$ is NP-hard, $\mathcal{L}_1 \in \mathsf{P}$ implies that $\mathsf{P} = \mathsf{NP}$. By combining these two implications, we obtain that $\mathcal{L}_2 \in \mathsf{P}$ implies that $\mathsf{P} = \mathsf{NP}$, that is, $\mathcal{L}_2$ is NP-hard. □

Polynomial-time reductions come in many flavours. Some are straightforward; others, really intricate and clever. The reduction from SAT to ILP that shows that ILP is NP-hard, shown in section 2, is essentially as simple as they come. The reduction that shows that SAT is NP-hard is one of the more intricate ones. SAT was the first every problem after $A_{\text{TM}}^{\text{poly}}$ to be shown to be NP-hard, so the only way to do so was to describe a polynomial-time reduction from $A_{\text{TM}}^{\text{poly}}$ to SAT. A full description of the reduction is beyond the scope of this course (even though I keep hoping that I will get organized enough to have time to discuss it in detail in CSCI 2115 one day). Here is a very rough sketch of the reduction:

Given a triple (M, c, σ) for which we want to decide whether $(M, c, \sigma) \in A_{\text{TM}}^{\text{poly}}$, we can construct a Boolean formula F in polynomial time that models the behaviour of M on the input σ. In particular, F is satisfiable if and only if there exists a set of non-deterministic choices (reflected by an appropriate assignment of Boolean values to the variables in F) that lead M to accept σ in at most $|\sigma|^c$ steps. Thus, F is satisfiable if and only if $(M, c, \sigma) \in A_{\text{TM}}^{\text{poly}}$, that is, the construction of F from (M, c, σ) constitutes a polynomial-time reduction from $A_{\text{TM}}^{\text{poly}}$ to SAT. Since $A_{\text{TM}}^{\text{poly}}$ is NP-hard, so is SAT, by Cor. 10.

REFERENCES

Karmarkar, Narendra (1984). "A New Polynomial Time Algorithm for Linear Programming". In: *Combinatorica* 4.4, pp. 373–395.

Khachiyan, Leonid G. (1979). "A polynomial algorithm for linear programming". In: *Doklady Akademiia Nauk USSR* 244, pp. 1093–1096.

Spielman, Daniel A. and Shang-Hua Teng (2004). "Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time". In: *Journal of the ACM* 51.3, pp. 385–463.