Part III

Complexity

CHAPTER

FOURTEEN

INTRODUCTION TO COMPLEXITY

In the previous unit, we considered which computational problems are solvable by algorithms, without any constraints on the time and memory used by the algorithm. We saw that even with unlimited resources, many problems—indeed, "most" problems—are not solvable by any algorithm.

In this unit on computational complexity, we consider problems that *are* solvable by algorithms, and focus on *how efficiently* they can be solved. Primarily, we will be concerned how much *time* it takes to solve a problem.[36] We mainly focus on *decision* problems—i.e., languages—then later broaden our treatment to *functional*—i.e., search—problems.

A Turing machine's running time, also known as *time complexity*, is the number of steps it takes until it halts; similarly, its *space complexity* is the number of tape cells it uses. We focus primarily on time complexity, and *as previously discussed* (page 3), we are interested in the *asymptotic* complexity with respect to the input size, in the worst case. For any particular asymptotic bound $O(t(n))$, we define the set of languages that are decidable by Turing machines having time complexity $O(t(n))$, where n is the input size:

Definition 129 (DTIME) Define

$$\mathsf{DTIME}(t(n)) = \{L \subseteq \Sigma^* : L \text{ is decided by some Turing machine with time complexity } O(t(n))\} .$$

The set $\mathsf{DTIME}(t(n))$ is an example of a *complexity class*: a set of languages whose complexity in some metric of interest is bounded in some specified way. Concrete examples include $\mathsf{DTIME}(n)$, the class of languages that are decidable by Turing machines that run in (at most) linear $O(n)$ time; and $\mathsf{DTIME}(n^2)$, the class of languages decidable by quadratic-time Turing machines.

When we discussed computability, we defined two classes of languages: the decidable languages (also called *recursive*), denoted by R, and the recognizable languages (also called *recursively enumerable*), denoted by RE. The definitions of these two classes are actually *model-independent*: they contain the same languages, regardless of whether our computational model is Turing machines, lambda calculus, or some other sensible model, as long as it is Turing-equivalent.

Unfortunately, this kind of model independence does not extend to $\mathsf{DTIME}(t(n))$. Consider the concrete language

$$\text{PALINDROME} = \{x : x = x^R, \text{ i.e., } x \text{ equals its reversal}\} .$$

In a computational model with random-access memory, or even in the two-tape Turing-machine model, PALINDROME can be decided in linear $O(n)$ time, where $n = |x|$: just walk pointers inward from both ends of the input, comparing character by character.

However, in the standard one-tape Turing-machine model, it can be proved that there is *no* $O(n)$-time algorithm for PALINDROME; in fact, it requires $\Omega(n^2)$ time to decide. Essentially, the issue is that an algorithm to decide this language would need to compare the first and last symbols of x, which requires moving the head sequentially over the entire string, and do similarly for second and second-to-last symbols of x, and so on. Because $n/4$ of the pairs require

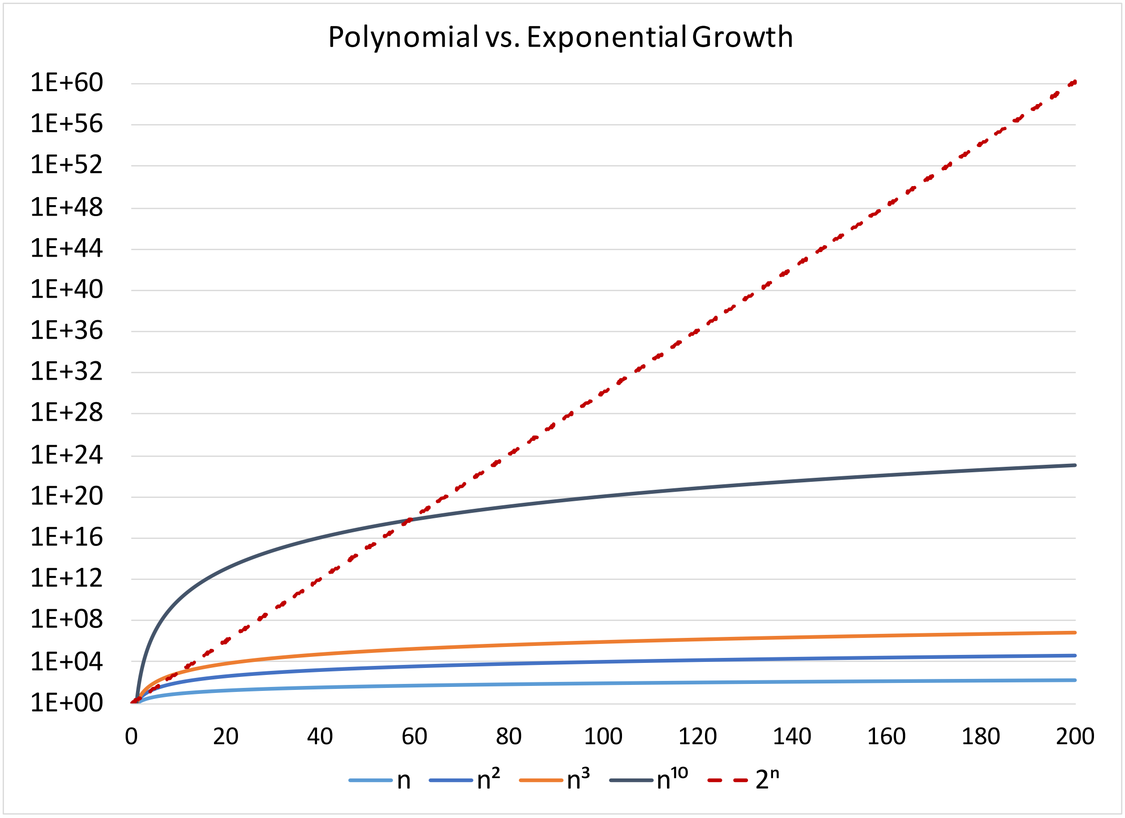
[36] More generally, the field of computational complexity is concerned with quantifying all kinds of different resources, like time, memory, randomness, etc.; and with understanding various kinds of computational models, like Turing machines, branching programs, formulas, nondeterminism, interaction, etc.

moving the head by at least $n/2$ cells each, this results in a total running time of $\Omega(n^2)$.[37] So, PALINDROME $\notin$ $\mathsf{DTIME}(n)$, but in other computational models, PALINDROME can be decided in $O(n)$ time.

A model-dependent complexity class is very inconvenient, because we wish to analyze algorithms at a higher level of abstraction, without worrying about the details of the underlying computational model, or how the algorithm would be implemented on it, like the particulars of data structures (which can affect asymptotic running times).

14.1 Polynomial Time and the Class P

We wish to define a complexity class that captures all problems that can be solved "efficiently", and is also model-independent. As a step toward this goal, we note that there is an enormous qualitative difference between the growth of *polynomial* functions versus *exponential* functions. The following illustrates how several polynomials in n compare to the exponential function 2^n:



The vertical axis in this plot is logarithmic, so that the growth of 2^n appears as a straight line. Observe that even a polynomial with a fairly large exponent, like n^{10}, grows much slower than 2^n (except for small n), with the latter exceeding the former for all $n \geq 60$.

In addition to the dramatic difference in growth between polynomials and exponentials, we also observe that polynomials have nice *closure* properties: if $f(n), g(n)$ are polynomially bounded, i.e., $f(n) = O(n^c)$ and $g(n) = O(n^{c'})$

[37] We caution that the above is *not a proof* that deciding PALINDROME requires $\Omega(n^2)$ time on a standard Turing machine. A correct proof would need to consider *all* Turing machines, even ones that use very clever strategies we might not be able to imagine. Such a proof is known, though it is quite subtle.

for some constants c, c', then

$$\begin{aligned} f(n) + g(n) &= O(n^{\max\{c,c'\}}), \\ f(n) \cdot g(n) &= O(n^{c+c'}), \\ f(g(n)) &= O(n^{c \cdot c'}) \end{aligned}$$

are polynomially bounded as well, because $\max\{c, c'\}$, $c + c'$, and $c \cdot c'$ are constants. These facts can be used to prove that if we compose a polynomial-time algorithm that uses some subroutine with a polynomial-time implementation of that subroutine, then the resulting full algorithm is also polynomial time. This composition property allows us to obtain polynomial-time algorithms in a *modular* way, by designing and analyzing individual components in isolation.

Thanks to the above properties of polynomials, it has been shown that the notion of polynomial-time computation is "robust" across many popular computational models, like the many variants of Turing machines, lambda calculi, etc. That is, any of these models can simulate any other one with only polynomial "slowdown". So, any particular problem is solvable in polynomial time either in *all* such models, or in *none* of them.

The above considerations lead us to define "efficient" to mean "polynomial time (in the input size)". From this we get the following model-independent complexity class of languages that are decidable in polynomial time (across many models of computation).

Definition 130 (Complexity Class P) The complexity class P is defined as

$$\begin{aligned} \mathsf{P} &= \bigcup_{k \geq 1} \mathsf{DTIME}(n^k) \\ &= \{L : L \text{ is decided by some polynomial-time TM } \} \ . \end{aligned}$$

In other words, a language $L \in \mathsf{P}$ if (and only if) it is decided by some Turing machine that runs in time $O(n^k)$ for some constant k.

In brief, P is the class of "efficiently decidable" languages. Based on what we saw in the algorithms unit, this class includes many fundamental problems of interest, like the *decision* versions of the greatest common divisor problem, sorting, the longest increasing subsequence problem, etc. Note that since P is a class of *languages*, or *decision* problems, it technically does not include the *search* versions of these problems—but see below for the close relationship between search and decision.

As a final remark, the *extended Church-Turing thesis* posits that the notion of polynomial-time computation is "robust" across *all* realistic models of computation:

Theorem 131 (Extended Church-Turing Thesis) *A problem is solvable in polynomial time on a Turing machine if and only if it is solvable in polynomial time in* any *"realistic" model of computation.*

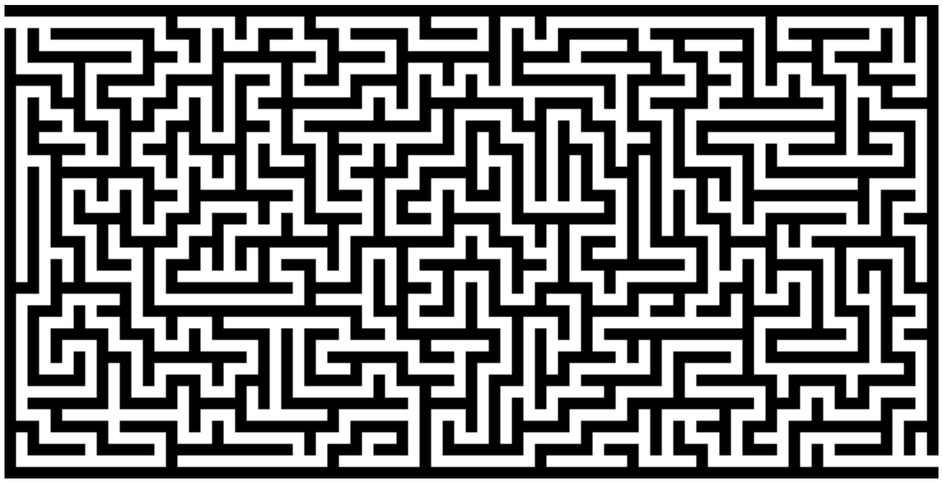
Because "realistic" is not precisely defined, this is just a thesis, not a statement than can be proved. Indeed, this is a major strengthening of the standard Church-Turing thesis, and there is no strong agreement about whether it is even true! In particular, the model of *quantum* computation may pose a serious challenge to this thesis: it is known that polynomial-time quantum algorithms exist for certain problems, like factoring integers into their prime divisors, that we *do not know how to solve in polynomial time on Turing machines*! So, if quantum computation is "realistic", *and* if there really is no polynomial-time factoring algorithm in the TM model, then the extended Church-Turing thesis is false. While both of these hypotheses seem plausible, they are still uncertain at this time: real devices that can implement the full model of quantum computation have not yet been built, and we do not have any *proof* that factoring integers (or any other problem that quantum computers can solve in polynomial time) *requires* more than polynomial time on a Turing machine.

14.2 Examples of Efficient Verification

For some computational problems, it is possible to efficiently *verify* whether a *claimed solution* is actually correct—regardless of whether *computing* a correct solution "from scratch" can be done efficiently. Examples of this phenomenon are abundant in everyday life, and include:

- **Logic and word puzzles** like mazes, Sudoku, or crosswords: these come in various degrees of difficulty to solve, some quite challenging. But given a proposed solution, it is straightforward to check whether it satisfies the "rules" of the puzzle. (See below for a detailed example with mazes.)
- **Homework problems** that ask for correct software code (with justification) or mathematical proofs: producing a good solution might require a lot of effort and creativity to discover the right insights. But given a candidate solution, one can check relatively easily whether it is clear and correct, simply by applying the rules of logic. For example, to verify a claimed mathematical proof, we just need to check whether each step follows logically from the hypotheses and the previous steps, and that the proof reaches the desired conclusion.
- **Music, writing, video, and other media**: creating a high-quality song, book, movie, etc. might require a lot of creativity, effort, and expertise. But given such a piece of media, it is relatively easy for even a non-expert to decide whether it is engaging and worthwhile to them (even though this is a subjective judgment that may vary from person to person). For example, even though the authors of this text could not come close to writing a beautiful symphony or a hit pop song, we easily know one when we hear one.

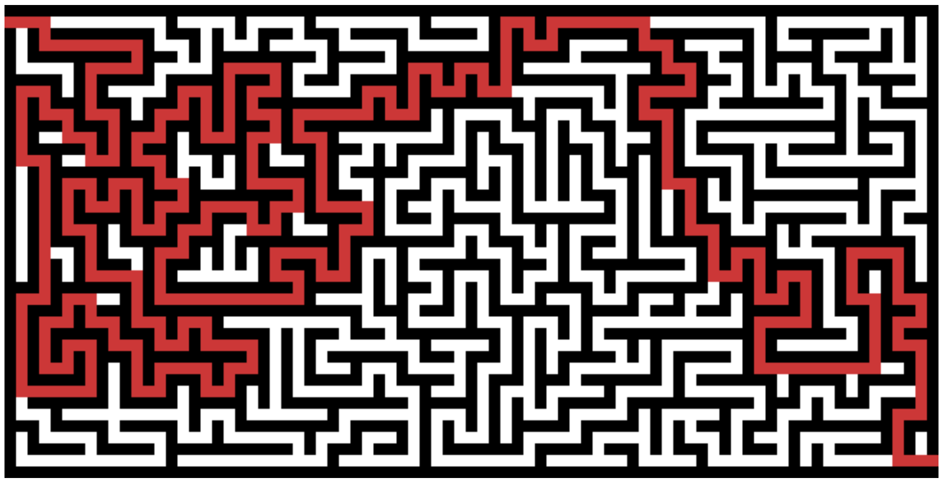
As a detailed example, consider the following maze, courtesy of Kees Meijer's maze generator[38]:



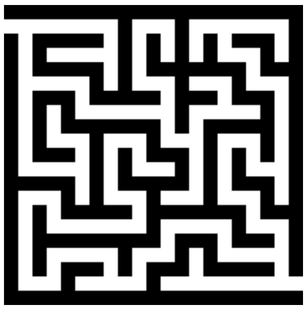
At first glance, it is not clear whether this maze has a solution. However, suppose that someone—perhaps a very good solver of mazes, or the person who constructed the maze—were to claim that this maze is indeed solvable. Is there some way that they could convince us of this fact?

A natural idea is simply to provide us with a path through the maze as "proof" of the claim. It is easy for us to check that this proof is valid, by verifying that the path goes from start to finish without crossing any "wall" of the maze. By definition, any solvable maze has such a path, so it is possible to convince us that a solvable maze really is solvable.

[38] https://keesiemeijer.github.io/maze-generator/



On the other hand, suppose that a given maze *does not* have a solution. Then no matter what *claimed* "proof" someone might give us, the checks we perform will cause us to reject it: because the maze is not solvable, any path must either fail to go from start to finish, or cross a wall somewhere (or both).



In summary: for any given maze, checking that a claimed solution goes from start to finish without crossing any wall is an *efficient verification* procedure:

- The checks can be performed efficiently, i.e., in polynomial time in the size of the maze.
- If the maze is solvable, then *there exists* a "proof"—namely, a path through the maze from start to finish—that the procedure will accept. *How to find* such a proof is not the verifier's concern; all that matters is that it exists.[39]
- If the maze is not solvable, then *no* claimed "proof" will satisfy the procedure.

Observe that we need *both* of the last two conditions in order for the procedure to be considered a correct verifier:

- An "overly skeptical" verifier that cannot be "convinced" by anything, even though the maze is actually solvable, would not be correct.
- Similarly, an "overly credulous" verifier that can be "fooled" into accepting, even when the maze is *not* solvable, would also be incorrect.

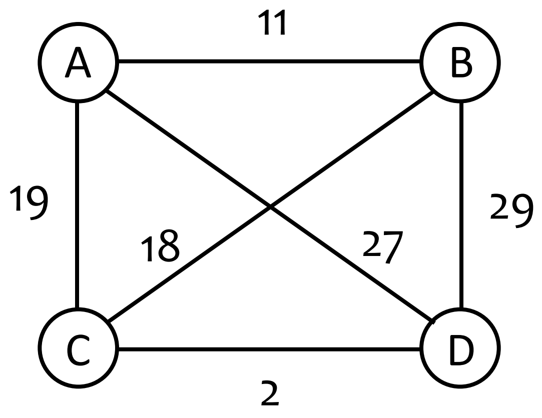
As already argued, our verification procedure has the right balance of "skepticism" and "credulity".

As another example, the *traveling salesperson problem (TSP)* is the task of finding a minimum-weight *tour* of a given weighted graph. In this text, a "tour" of a graph is a *cycle that visits every vertex exactly once*, i.e., it is a path that visits every vertex and then immediately returns to its starting vertex. (Beware that some texts define "tour" differently.) Without loss of generality, for TSP we can assume that the graph is *complete*—i.e., there is an edge between every pair of vertices—by filling in any missing edges with edges of enormous (or infinite) weight. This does not affect the

[39] The reader might have noticed that for any given maze, it is actually possible to use a graph-search algorithm like BFS or DFS to *find* a solution efficiently, when one exists. So, MAZE is even efficiently *decidable*, i.e.,MAZE $\in$ P. This does not contradict anything written above about MAZE also being efficiently *verifiable*. However, it does give us an alternative efficient verifier for MAZE, which just ignores the provided "proof" and determines on its own whether the given maze is solvable. This verifier trivially meets all the efficiency and correctness conditions we have laid out.

solution(s), because any tour that uses any of these new edges will have larger weight than any tour that does not use any of them.

Suppose we are interested in a minimum-weight tour of the following graph:



If someone were to claim that the path $A \to B \to D \to C \to A$ is a minimum-weight tour, could we efficiently verify that this is true? There doesn't seem to be an obvious way to do so, apart from considering all other tours and checking whether any of them have smaller weight. While this particular graph only has three distinct tours (ignoring reversals and different starting points on the same cycle), in general, the number of tours in a graph is exponential in the number of vertices, so this approach would not be efficient. So, it is not clear whether we can efficiently verify that a claimed minimum-weight tour really is one.

However, let's modify the problem to be a *decision* problem, which simply asks whether there is a tour whose total weight is *within some specified "budget"*:

> Given a weighted graph G and a budget k, does G have a tour of weight at most k?

Can the existence of such a tour be proved to an efficient, suitably skeptical verifier? Yes: simply provide such a tour as the proof. The verifier, given a path in G—i.e., a list of vertices—that is *claimed* to be such a tour, would check that all of the following hold:

1. the path starts and ends at the same vertex,
2. the path visits every vertex in the graph exactly once (except for the repeated start vertex at the end), and
3. the sum of the weights of the edges on the path is at most k.

(All of these tests can be performed efficiently; see the formalization in Algorithm 137 below for details.)

For example, consider the following *claimed* "proofs" that the above graph has a tour that is within budget $k = 60$:

- The path $A \to B \to D \to C$ does not start and end at the same vertex, so the verifier's first check would reject it as invalid.
- The path $A \to B \to D \to A$ starts and ends at the same vertex, but it does not visit vertex C, so the verifier's second check would reject it.
- The path $A \to B \to D \to C \to A$ satisfies the first two checks, but it has total weight 61, which is not within the budget, so the verifier's third check would reject it.
- The path $A \to B \to C \to D \to A$ satisfies the first two checks, and its total weight of 58 is within the budget, so the verifier would *accept* it.

These examples illustrate that when the graph has a tour that is within the budget, there are still "proofs" that are "unconvincing" to the verifier—but there will also be a "convincing" proof, which is what matters to us.

On the other hand, it turns out that the above graph does *not* have a tour that is within a budget of $k = 57$. And for this graph and budget, the verifier will reject no matter what claimed "proof" it is given. This is because every tour in the graph—i.e., every path that would pass the verifier's first two checks—has total weight at least 58.

In general, for any given graph and budget, the above-described verification procedure is correct:

- If there is a tour of the graph that is within the budget, then *there exists* a "proof"—namely, such a tour itself—that the procedure will accept. (As before, *how to find* such a proof is not the verifier's concern.)
- If there is no such tour, then *no* claimed "proof" will satisfy the procedure, i.e., it cannot be "fooled" into accepting.

14.3 Efficient Verifiers and the Class NP

We now generalize the above examples to formally define the notion of "efficient verification" for an arbitrary decision problem, i.e., a language. This definition captures the notion of an efficient procedure that can be "convinced", by a suitable "proof", into accepting any string in the language, but cannot be "fooled" into accepting a string that is not in the language.

Definition 132 (Efficient Verifier) An *efficient verifier* for a language L is a Turing machine $V(x, c)$ that takes two inputs, an *instance* x and a *certificate* (or "claimed proof") c, and satisfies the following properties:

- **Efficiency**: $V(x, c)$ runs in time polynomial in the size $|x|$ of the *instance* x, for any x and c.
- **Completeness**: if $x \in L$, then **there exists** some c for which $V(x, c)$ accepts.
- **Soundness**: if $x \notin L$, then **for all** c, $V(x, c)$ rejects.

Alternatively, completeness and soundness together are equivalent to:

- **Correctness**: $x \in L$ *if and only if* there exists some c for which $V(x, c)$ accepts.

We say that a language is *efficiently verifiable* if there is an efficient verifier for it.

The claimed equivalence can be seen by taking the contrapositive of the soundness condition, which is: if there exists some c for which $V(x, c)$ accepts, then $x \in L$. (Recall that when negating a predicate, "for all" becomes "there exists", and vice-versa.) This contrapositive statement and completeness are, respectively, the "if" and "only if" parts of correctness.

We sometimes say that a certificate c is "valid" or "invalid" for a given instance x if $V(x, c)$ accepts or rejects, respectively. Note that the *decision of the verifier* is what determines whether a certificate is "(in)valid"—not the other way around—and that the validity of a certificate depends on both the instance *and the verifier*. There can be many different verifiers for the same language, which in general can make different decisions on the same x, c (though they all must reject when $x \notin L$).

In general, the instance x and certificate c are *arbitrary strings* over the verifier's input alphabet Σ (and they are separated on the tape by some special character that is not in Σ, but is in the tape alphabet Γ). However, for specific languages, x and c will typically *represent* various mathematical objects like integers, arrays, graphs, vertices, etc. As in the computability unit, this is done via appropriate *encodings* of the objects, denoted by $\langle \cdot \rangle$, where without loss of generality *every* string decodes as some object of the desired "type" (e.g., integer, list of vertices, etc.). Therefore, when we write the pseudocode for a verifier, we can treat the instance and certificate as already having the desired types.

14.3.1 Discussion of Completeness and Soundness

We point out some of the important aspects of completeness and soundness (or together, correctness) in Definition 132.

1. Notice some similarities and differences between the notions of *verifier* and *decider* (Definition 54):

    - When the input $x \in L$, both kinds of machine must accept, but a verifier need only accept for *some* value of the certificate c, and may reject for others. By contrast, a decider does not get any other input besides x, and simply must accept it.
    - When the input $x \notin L$, both kinds of machine must reject, and moreover, a verifier must reject for *all* values of c.

2. There is an *asymmetry* between the definitions of completeness and soundness:

    - If a (sound) verifier for language L *accepts* some input x, c, then we can correctly conclude that $x \in L$, by the contrapositive of soundness. (A sound verifier cannot be "fooled" into accepting an instance that is not in the language.)
    - However, if a (complete) verifier for L *rejects* some input x, c, then in general we *cannot reach any conclusion* about whether $x \in L$: it might be that $x \notin L$, or it might be that $x \in L$ but c is just not a valid certificate for x. (In the latter case, by completeness, some other certificate c' is valid for x.)

    In summary, while every string $x \in L$ has a "convincing proof" of its membership in L, a string $x \notin L$ does not necessarily have a proof of its non-membership in L.

14.3.2 Discussion of Efficiency

We also highlight some important but subtle aspects of the notion of efficiency from Definition 132.

1. First, efficiency is defined as polynomial time with respect to the *instance x alone*, not the *entire input x, c* together. If we used the latter, then the verifier would be allowed to run in exponential time in $|x|$, simply by giving it an exponentially long certificate c. This does not fit the spirit of "efficiency", and it causes some technical problems as well.[40]

2. Since a verifier must run in time polynomial in the instance size, it can read only polynomially many of the initial symbols of c, and no others. (This is because reading each symbol and moving the head takes a computational step.) So, the rest of the certificate is irrelevant, and can be truncated without affecting the verifier's behavior. There are two immediate consequences:

    - Given an efficient verifier, we can assume without loss of generality that the size of the input certificate is bounded by some polynomial.
    - Conversely, when designing a verifier and analyzing its running time, we can implicitly assume that it reads just enough of the certificate to make its accept/reject decision. More specifically, if every member of the language has a certificate of polynomially bounded size that convinces the verifier, then **we can implicitly assume that the verifier immediately rejects any certificate that is longer than this bound**, without writing it explicitly in the pseudocode. This does not affect completeness or soundness, and in the runtime analysis it allows us to ignore the annoying case where the verifier is given an enormous certificate.

[40] For example, if we measured verifier efficiency in terms of the total input size, then we could "efficiently" prove that there is no tour of a given graph within a certain budget, simply by giving a certificate that lists all the exponentially many tours. The verifier could check all of them in time merely linear in the certificate size. But this is not efficient in the sense we want, because it takes exponential time in the size of the graph.

14.3.3 The Class NP

With the notion of an efficient verifier in hand, analogously to how we defined P as the class of efficiently decidable languages, we define NP as the class of efficiently verifiable languages.

Definition 133 (Complexity Class NP) The complexity class NP is defined as the set of *efficiently verifiable languages*:

$$\mathsf{NP} = \{L : L \text{ is efficiently verifiable}\} .$$

In other words, a language $L \in \mathsf{NP}$ if (and only if) there is an efficient verifier for it, according to Definition 132.[41]

[41] We caution that NP **does not** stand for "not polynomial". It is an abbreviation of the name "*nondeterministic* polynomial," which comes from an equivalent definition of the class based on the computational model of nondeterministic Turing machines. (This model is beyond the scope of this text.) A possible alternative name would be VP, for "verifiable in polynomial time".

Let us now formalize our first example of efficient verification from above: the decision problem of determining whether a given maze has a solution. A maze can be represented as an undirected graph, with a vertex for each "intersection" in the maze (along with the start and end points), and edges between adjacent positions. So, the decision problem is to determine whether, given such a graph, there exists a path between the start vertex s and the end vertex t. This can be represented as the language[42]

$$\text{MAZE} = \{(G, s, t) : G \text{ is an undirected graph that has a path from vertices } s \text{ to } t\} .$$

We define an efficient verifier for MAZE as follows; the precise pseudocode is given in Algorithm 134. An instance is a graph $G = (V, E)$, a start vertex s, and a target vertex t. A certificate is (decoded as) a sequence of up to $|V|$ vertices in the graph. (This limit on the number of vertices in the certificate can be enforced by the decoding, and it simplifies the efficiency analysis by ruling out huge certificates, as explained in the *Discussion of Efficiency* (page 143).) The verifier checks that the sequence describes a valid path in the graph from s to t, i.e., that the first and last vertices in the sequence are s and t (respectively), and that there is an edge between every pair of adjacent vertices in the sequence.

Algorithm 134 (Efficient Verifier for MAZE)

```
Input: instance: a graph and start/end vertices; certificate: a list of up to $|V|$ vertices
Output: whether the certificate represents a path in the graph from the start to the end
  function VerifyMAZE($(G = (V, E), s, t), c = (v_1, \ldots v_m)$)
    if $v_1 \neq s$ or $v_m \neq t$ then reject
    for $i = 1$ to $m - 1$ do
      if $(v_i, v_{i+1}) \notin E$ then reject
    accept
```

Lemma 135 *VerifyMAZE is an efficient verifier for MAZE, so MAZE* $\in \mathsf{NP}$.

Proof 136 We show that VerifyMAZE satisfies Definition 132 by showing that it is efficient and correct.

First, VerifyMAZE runs in time polynomial in the size $|G| \geq |V|$ of the graph: it compares two vertices from the certificate against the start and end vertices in the instance, and checks whether there is an edge between up to $|V| - 1$ pairs of adjacent vertices in the certificate. Each pair of vertices can be checked in polynomial time. The exact polynomial depends on the representation of G and the underlying computational model, but it is polynomial in any reasonable representation (e.g., adjacency lists, adjacency matrix), which is all that matters for our purposes here.

As for correctness:

[42] Henceforth, for notational simplicity we will elide the encoding of inputs (e.g., $\langle G, s, t\rangle$) when defining languages, taking it to be implicit.

- If $(G, s, t) \in$ MAZE, then by definition there is some path $s \to v_2 \to \cdots \to v_{m-1} \to t$ in G that visits $m \leq |V|$ vertices in total, because any cycle in the path can be removed.[43] Then by inspection of the pseudocode, VerifyMAZE$((G, s, t), c = (s, v_2, \ldots, v_{m-1}, t))$ accepts.
- Conversely, if VerifyMAZE$((G, s, t), c = (v_1, \ldots, v_m))$ accepts for some certificate c, then by inspection of the pseudocode, c represents a path between $v_1 = s$ and $v_m = t$ in G, so $(G, s, t) \in$ MAZE by definition.

[43] The bound of $|V|$ on the number of vertices in a *valid* certificate is why we can assume that *any* certificate, valid or not, has at most $|V|$ nodes; see *Discussion of Efficiency* (page 143) for details.

Alternatively, instead of showing the "conversely" part of correctness as above, we could have argued that Verify-MAZE is *sound*, as follows: if $(G, s, t) \notin$ MAZE, then by definition there is no path between s and t in G, so any certificate c will either not start at s, not end at t, or it will have some pair of adjacent vertices with no edge between them. Thus, by inspection of the pseudocode, VerifyMAZE$((G, s, t), c)$ will reject.

This kind of reasoning is correct, but it is more cumbersome and error prone, since it involves more cases and argues about the *non-existence* of certain objects. Usually, and especially for more complex verifiers, it is easier and more natural to directly prove the correctness condition ($x \in L$ if and only if there exists c such that $V(x, c)$ accepts) instead of soundness.

Next, returning to the "limited-budget" TSP example, we define the corresponding language

$$\text{TSP} = \{(G, k) : G \text{ is a weighted graph that has a tour of total weight at most } k\} \; .$$

A certificate for a given instance $(G = (V, E), k)$ is a sequence of up to $|V|$ vertices in the graph. The verifier simply checks that the vertices form a tour whose cost is at most k. The precise pseudocode is given in Algorithm 137.

Algorithm 137 (Efficient Verifier for TSP)

```
Input: instance: weighted graph and weight budget; certificate: a list of up to $|V|$ vertices in the graph
Output: whether the certificate represents a tour within the budget
  function VerifyTSP($(G = (V, E, w), k), c = (v_0, v_1, \ldots, v_m)$)
      if $m \neq |V|$ or $v_0 \neq v_m$ then reject
      if $v_i = v_j$ for some distinct $1 \leq i, j \leq m$ then reject
      $t = 0$
      for $i = 0$ to $m - 1$ do
          $t = t + w(v_i, v_{i+1})$
      if $t > k$ then reject
      accept
```

Lemma 138 *VerifyTSP is an efficient verifier for TSP, so TSP* $\in$ NP.

Proof 139 We show that VerifyTSP satisfies Definition 132 by showing that it is efficient and correct.

First, VerifyTSP runs in time polynomial in the size $|G, k|$ of the instance: checking for duplicate vertices v_i, v_j can be done in polynomial time, e.g., by checking all $O(m^2) = O(|V|^2)$ pairs of distinct $1 \leq i, j \leq m$. Then the algorithm loops over $|V|$ edges, summing their weights. These weights are included in the input instance, so they can be summed in polynomial time in the instance size. Finally, the algorithm compares the sum against k, which can be done in polynomial time.

We now argue correctness:

- If $(G, k) \in$ TSP, then by definition, G has a tour of weight at most k. Letting c be the sequence of vertices in such a tour (starting and ending at the same arbitrary vertex), we have that VerifyTSP$((G, k), c)$ accepts, because all of V's checks are satisfied by this c.

- Conversely, if $\textsc{Verify}\text{TSP}((G,k),c)$ accepts for some $c=(v_0,\ldots,v_m)$, then because all of V's checks are satisfied, this c starts and ends at the same vertex $v_0=v_m$, it visits all $|V|$ vertices exactly once (because there are no duplicate vertices among $v_1,\ldots,v_m$), and the total weight of all m edges between adjacent vertices in c is at most k. Therefore, c is a tour of G having total weight at most k, hence $(G,k)\in\text{TSP}$, as needed.

14.4 P Versus NP

We have defined two complexity classes:

- P is the class of languages that can be decided efficiently.
- NP is the class of languages that can be verified efficiently.

More precisely, a language $L\in\mathsf{P}$ if there exists a polynomial-time algorithm D such that:

- if $x\in L$, then $D(x)$ accepts;
- if $x\notin L$, then $D(x)$ rejects.

Similarly, $L\in\mathsf{NP}$ if there exists a polynomial-time (with respect to its first input) algorithm V such that:

- if $x\in L$, then $V(x,c)$ accepts for at least one certificate c;
- if $x\notin L$, then $V(x,c)$ rejects for all certificates c.

How are these two classes related? First, if a language is efficiently decidable, then it is also efficiently verifiable, trivially: the verifier can just ignore the certificate, and determine on its own whether the input is in the language, using the given efficient decider. (As an exercise, formalize this argument according to the definitions.) This gives us the following result.

Lemma 140 $\mathsf{P}\subseteq\mathsf{NP}$.

The above relationship allows for two possibilities:

- $\mathsf{P}\subsetneq\mathsf{NP}$, i.e., P is a *proper* subset of (hence not equal to) NP; or
- $\mathsf{P}=\mathsf{NP}$.

The latter possibility would mean that every efficiently *verifiable* problem is also efficiently *decidable*. Is that the case? What is the answer to the question

$$\text{Does } \mathsf{P}=\mathsf{NP}\text{ ?}$$

We do not know the answer to this question! Indeed, the "P versus NP" question is perhaps the greatest open problem in Computer Science—and even one of the most important problems in all of Mathematics, as judged by the Clay Mathematics Institute[44], which has offered a $1 million prize for its resolution.

Consider the two example languages from above, MAZE and TSP. We saw that both are in NP, and indeed, they have similar definitions, and their verifiers also have much in common. However, we know that MAZE $\in\mathsf{P}$: it can be decided efficiently simply by checking whether a breadth-first search from vertex s reaches vertex t. On the other hand, we do not know whether TSP is in P: we do not know of any efficient algorithm that decides TSP, and we do not know how to prove that no such algorithm exists. Most experts *believe* that there is no efficient algorithm for TSP, which would imply that $\mathsf{P}\neq\mathsf{NP}$, but the community has no idea how to *prove* this.

How could we hope to resolve the P-versus-NP question? To show that the two classes are not equal, as most experts believe, it would suffice to demonstrate that some *single* language is in NP, but is not in P. However, this seems exceedingly difficult to do: we would need to somehow prove that, of all the infinitely many efficient algorithms,

[44] https://www.claymath.org/millennium-problems/

including many we have not yet discovered and cannot even imagine, none of them decides the language in question.[45] On the other hand, showing that $\mathsf{P} = \mathsf{NP}$ also seems very difficult: we would need to demonstrate that *every* one of the infinitely many languages in NP can be decided efficiently, i.e., by some polynomial-time algorithm.

Fortunately, a rich theory has been developed that will make the resolution of P versus NP (somewhat) simpler. As we will see below, it is possible to prove that some languages in NP are the "*hardest*" ones in that class, in the following sense:

> an efficient algorithm for *any one* of these "hardest" languages would imply an efficient algorithm for *every* language in NP!

So, to prove that $\mathsf{P} = \mathsf{NP}$, it would suffice to prove that *just one* of these "hardest" languages is in P. And in the other direction, the most promising route to prove $\mathsf{P} \neq \mathsf{NP}$ is to show that one of these "hardest" languages is not in P—because if any NP language is not in P, then that is also the case for these "hardest" ones.

In summary, the resolution of the P-versus-NP question lies entirely with the common fate of these "hardest" languages:

- Any *one* of them has an efficient algorithm, if and only if *all* of them do, if and only if $\mathsf{P} = \mathsf{NP}$.
- Conversely, any one of them does *not* have an efficient algorithm, if and only if *none* of them do, if and only if $\mathsf{P} \neq \mathsf{NP}$.

It turns out that there are thousands of known "hardest" languages in NP. In fact, as we will see, TSP is one of them! We will prove this via a series of results, starting with the historically first language that was shown to be one of the "hardest" in NP, next.

[45] Recall that in the computability unit, we proved the existence of *undecidable* languages via techniques like diagonalization. Clearly, an undecidable language cannot be decided efficiently, so it is not in P. However, the challenge here is that we seek a language that is not in P, but also *is* in NP. It can be shown that every language in NP is decidable, so an undecidable language will not serve our purposes. More generally, known techniques like diagonalization and several others have been shown to face fundamental obstacles for resolving P versus NP.

CHAPTER

FIFTEEN

SATISFIABILITY AND THE COOK-LEVIN THEOREM

Our first example of a "hardest" problem in NP is the *satisfiability* problem for *Boolean formulas*. Given as input a Boolean formula like

$$\phi = (y \vee (\neg x \wedge z) \vee \neg z) \wedge (\neg x \vee z) \ ,$$

we wish to determine whether there is a true/false *assignment* to its *variables* that makes the formula *evaluate* to true. Let us define the relevant terms:

- a *(Boolean) variable* like x or y or x_{42} can be *assigned* the value "true" (often represented as 1) or "false" (represented as 0);
- a *literal* is either a variable or its negation, e.g., x or $\neg y$;
- an *operator* is either conjunction (AND), represented as $\wedge$; disjunction (OR), represented as $\vee$; or negation (NOT), represented either as $\neg$ or with an overline, like $\neg(x \wedge y)$ or, equivalently, $\overline{x \wedge y}$.

A Boolean formula is a well-formed mathematical expression involving literals combined with operators, and following the usual rules of parentheses for grouping.

An initial observation is that using the rules of logic like De Morgan's laws[46], we can eliminate any double-negations, and we can iteratively move all negations "inward" to the literals, e.g., $\neg(x \wedge \neg y) = (\neg x \vee y)$. So, from now on we assume without loss of generality that the negation operator appears only in literals.

We define the *size* of a formula to be the number of literals it has, counting duplicate appearances of the same literal.[47] For example, the formula ϕ above has size 6. Note that the size of a formula is at least the number of distinct variables that appear in the formula.

An *assignment* is a mapping of the variables in a formula to truth values. We can represent an assignment over n variables (with some fixed order) as an n-tuple, such as $(a_1, \ldots, a_n)$. For the formula above, the assignment $(0, 1, 0)$ maps x to the value 0 (false), y to the value 1 (true), and z to the value 0 (false). The notation $\phi(a_1, \ldots, a_n)$ denotes the value of ϕ when *evaluated* on the assignment $(a_1, \ldots, a_n)$, i.e., with each variable substituted by its assigned value.

A *satisfying assignment* for a formula is an assignment that makes the formula evaluate to true. In the example above, $(0, 1, 0)$ is a satisfying assignment:

$$\begin{aligned} \phi(0,1,0) &= (1 \vee (\neg 0 \wedge 0) \vee \neg 0) \wedge (\neg 0 \vee 0) \\ &= (1 \vee (1 \wedge 0) \vee 1) \wedge (1 \vee 0) \\ &= 1 \wedge 1 \\ &= 1 \ . \end{aligned}$$

[46] https://en.wikipedia.org/wiki/De_Morgan%27s_laws

[47] Defining size as the number of literals is more robust and convenient than other definitions we might consider, like the total number of symbols in the formula (including operators and parentheses), while still being *proportional* to other reasonable notions. For example, "moving negations inward" does not change the number of literals, but it can change the number of symbols.

On the other hand, $(1,0,1)$ is not a satisfying assignment:

$$\begin{aligned}\phi(1,0,1) &= (0 \vee (\neg 1 \wedge 1) \vee \neg 1) \wedge (\neg 1 \vee 1)\\ &= (0 \vee (0 \wedge 1) \vee 0) \wedge (0 \vee 1)\\ &= 0 \wedge 1\\ &= 0 \ .\end{aligned}$$

A formula ϕ is *satisfiable* if it has at least one satisfying assignment. The decision problem of determining whether a given formula is satisfiable corresponds to the following language.

Definition 141 (Satisfiability Language) The (Boolean) satisfiability language is defined as

$$\text{SAT} = \{\phi : \phi \text{ is a satisfiable Boolean formula}\} \ .$$

A first observation is that SAT is decidable. A formula ϕ of size n has at most n variables, so there are at most 2^n possible assignments. Therefore, we can decide SAT using a brute-force algorithm that simply iterates over all of the assignments of its input formula, accepting if any of them satisfies the formula, and rejecting otherwise. Although this is *not efficient*, it is apparent that it does decide SAT.

We also observe that SAT has an efficient *verifier*.

Lemma 142 *SAT* $\in \mathsf{NP}$.

Proof 143 A certificate, or "claimed proof", for a formula ϕ is just an assignment for its variables. The following efficient verifier simply evaluates the given formula on the given assignment and accepts if the value is true, otherwise it rejects.

```
Input: instance: a Boolean formula; certificate: an assignment for its variables
Output: whether the assignment satisfies the formula
  function $V_{\text{SAT}}(\phi, \alpha)$
      if $\phi(\alpha) = 1$ then accept
      reject
```

This verifier runs in linear time in the size of its input formula ϕ, because evaluating each AND/OR operator in the formula reduces the number of terms in the expression by one.

For correctness, by the definitions of SAT and V_{SAT},

$$\begin{aligned}\phi \in \text{SAT} &\iff \text{exists } \alpha \text{ s.t. } \phi(\alpha) = 1\\ &\iff \text{exists } \alpha \text{ s.t. } V_{\text{SAT}}(\phi, \alpha) \text{ accepts} \ ,\end{aligned}$$

so by Definition 132, V_{SAT} is indeed an efficient verifier for SAT, as claimed. ☐

Is SAT *efficiently* decidable?—i.e., is SAT $\in \mathsf{P}$? The decider we described above is not efficient: it takes time *exponential* in the number of variables in the formula, and the number of variables may be as large as the *size* of the formula (i.e., the number of literals in it), so in the worst case the algorithm runs in exponential time in its input size.

However, the above does not prove that SAT $\notin \mathsf{P}$; it just shows that *one specific* (and naïve) algorithm for SAT is inefficient. Conceivably, there could be a more sophisticated *efficient* algorithm that cleverly analyzes an arbitrary input formula in some way to determine whether it is satisfiable. Indeed, there are regular conferences[48] and competitions[49] to which researchers submit their best ideas, algorithms, and software for solving SAT. Although many algorithms have been developed that perform very impressively on large SAT instances of interest, none of them is believed to run in polynomial time and to be correct on all instances.

[48] http://satisfiability.org/
[49] http://www.satcompetition.org/

To date, we actually *do not know* whether SAT is efficiently decidable—we do not know an efficient algorithm for it, and we do not know how to prove that no such algorithm exists. Yet although the question of whether SAT $\in \mathsf{P}$ is unresolved, the *Cook-Levin Theorem* says that SAT is a "hardest" problem in NP.[50]

Theorem 144 (Cook-Levin) *If SAT* $\in \mathsf{P}$*, then every* NP *language is in* P*, i.e.,* $\mathsf{P} = \mathsf{NP}$.

The full proof of the Cook-Levin Theorem is ingenious and rather intricate, but the main idea is fairly easy to describe. Let L be an arbitrary language in NP; this means there is an efficient *verifier* V for L. Using the hypothesis that SAT $\in \mathsf{P}$, we will construct an efficient *decider* for L, which implies that $L \in \mathsf{P}$, as claimed.

The key idea behind the efficient decider for L is that, using just the fact that V is an efficient verifier for L, we can efficiently *transform* any instance of L into an instance of SAT that has the same "yes/no answer": either both instances are in their respective languages, or neither is. More precisely, there is an efficient procedure that maps any instance x of L to a corresponding Boolean formula $\phi_{V,x}$, such that

$$x \in L \iff \phi_{V,x} \in \text{SAT} .$$

In fact, the above equivalence follows from a stronger property: by the careful design of $\phi_{V,x}$, there is a two-way correspondence between the valid certificate(s) c for x (if any), and the satisfying assignment(s) for $\phi_{V,x}$ (if any). That is, any certificate c that makes $V(x, c)$ accept directly yields a corresponding satisfying assignment for $\phi_{V,x}$, and any satisfying assignment for $\phi_{V,x}$ yields a corresponding certificate c that makes $V(x, c)$ accept.

By the hypothesis that SAT $\in \mathsf{P}$, there is an efficient decider D_{SAT} for SAT, so from all this we get the following efficient decider for L:

```
function $D_L(x)$
    construct $\phi_{V,x}$ as described below
    return $D_{\text{SAT}}(\phi_{V,x})$
```

Since $\phi_{V,x}$ can be constructed in time polynomial in the size of x, and D_{SAT} runs in time polynomial in the size of $\phi_{V,x}$, by composition, D_L as a whole runs in time polynomial in the size of x. And the fact that D_L is correct (i.e., it decides L) follows directly from the correctness of D_{SAT} and the above-stated relationship between x and $\phi_{V,x}$.

This completes the high-level description of the proof strategy. All that remains is to show how to efficiently construct $\phi_{V,x}$ from x so that they satisfy the above-stated relationship, which we do in what follows.

15.1 Configurations and Tableaus

In order to describe the construction of $\phi_{V,x}$, we fist need to recall the notion of a *configuration* of a Turing machine and its representation as a sequence, which we previously discussed in *Wang Tiling* (page 112). A configuration encodes a "snapshot" of a Turing machine's execution: the contents of the machine's tape, the active state $q \in Q$, and the position of the head. We represent these as an infinite sequence over the alphabet $\Gamma \cup Q$ (the union of the finite tape alphabet and the TM's finite set of states), which is simply:

- the contents of the tape, in order from the leftmost cell,
- with the active state $q \in Q$ inserted directly to the left of the symbol corresponding to the cell on which the head is positioned.

For example, if the input is $x_1 x_2 \cdots x_n$ and the initial state is q_0, then the following sequence represents the machine's starting configuration:

$$q_0 x_1 x_2 \cdots x_n \perp \perp \ldots$$

[50] The Cook-Levin Theorem is named after its discoverers, Stephen Cook and Leonid Levin, who found it in the early 1970s. Interestingly, they discovered this theorem *independently* of each other, with Cook working in the USA (and later Canada), and Levin working in Russia. Due to the "Cold War" of the time, the two countries' research communities did not have much interaction.

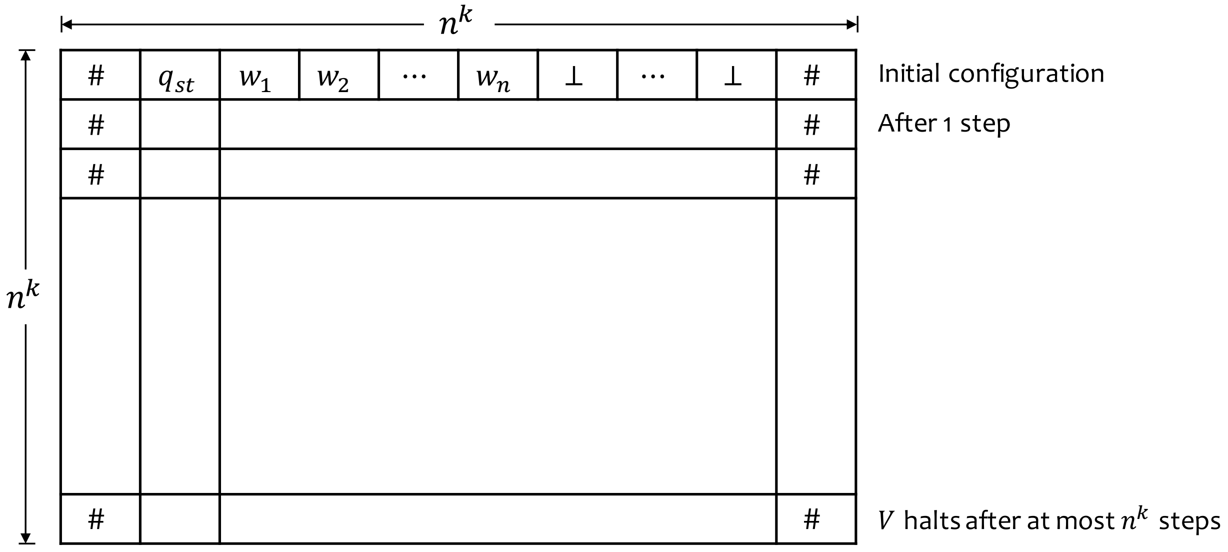
Since the head is at the leftmost cell and the state is q_0, the string has q_0 inserted to the left of the leftmost tape symbol. If the transition function gives $\delta(q_0, x_1) = (q', x', R)$, then the next configuration is represented by

$$x' q' x_2 \cdots x_n \perp\perp \dots$$

The first cell has been modified to x', the machine is in state q', and the head is at the second cell, represented here by writing q' to the left of that cell's symbol.

As stated above in the proof overview, for any language $L \in \mathsf{NP}$ there exists an efficient verifier V that runs in time polynomial in the size of its first argument. In other words, there is some constant k such that $V(x, c)$ runs for at most $|x|^k$ steps before halting, for any x, c.[51] Since the head starts at the leftmost cell and can move only one position in each step, this implies that $V(x, c)$ can read or write only the first $|x|^k$ cells of the tape. Thus, instead of using an infinite sequence, we can represent any configuration during the execution of $V(x, c)$ using just a *finite* string of length about $|x|^k$, ignoring the rest since it cannot affect the execution.

For an instance x of size $n = |x|$, we can represent the sequence of *all* the configurations of the machine, over its (at most) n^k computational steps, as an n^k-by-n^k *tableau*, with one configuration per row; for convenience later on, we also place a special $\#$ symbol at the start and end of each row.[52] So, each cell of the tableau contains a symbol from the set $S = \Gamma \cup Q \cup \{\#\}$, where Γ is the finite tape alphabet of V; Q is the finite set of states in V; and $\# \notin \Gamma \cup Q$ is a special extra symbol.



Observe that, since V is deterministic, the contents of the *first* row of the tableau *completely determine* the contents of *all* the rows, via the "code" of V. Moreover, adjacent rows represent a single computational step of the machine, and hence are identical to each other, except in the vicinity of the symbols representing the active states before and after the transition. Finally, $V(x, c)$ accepts if and only if the accept-state symbol $q_{\text{acc}} \in Q$ appears somewhere in its tableau. These are important feature of tableaus that are exploited in the construction of the formula $\phi_{V,x}$, which we describe next.

[51] To be completely accurate, there are constants C, k such that $V(x, c)$ runs for at most $C|x|^k$ steps before halting, for all *sufficiently large* $|x|$ (and all c). There are only finitely many x that are not "sufficiently large," so the yes/no answer of whether $x \in L$ for each of these x can be "hard-coded" into the decider for L, hence we can restrict our attention to deciding L for all sufficiently large x. As for the constant C, for notational simplicity we simply elide it from the expressions.

[52] To be completely accurate, we need $n^k + 1$ rows and $n^k + 3$ columns to represent the tableau: t computational steps require $t + 1$ rows, and we need n^k columns for the tape cells, plus one for the active state and two for the $\#$ symbols. These constant terms are not important to the proof, so for notational simplicity we ignore them. In addition, if the machine halts in fewer than n^k steps, we "pad" the tableau to have exactly $n^k + 1$ rows by appending copies of the final row (corresponding to the halting configuration).

15.2 Constructing the Formula

With the notion of a computational tableau in hand, we now describe the structure of the formula $\phi_{V,x}$ that is constructed from the instance x (also using the fixed code of V).

- The *variables* of the formula represent the *contents* of all the cells in a potential tableau for V. That is, assigning Boolean values to all the variables fully specifies the contents of a *claimed* tableau, including the value of a certificate c in the first row.

  Conceptually, we can think of an assignment to the variables, and the potential tableau that they represent, as a *claimed execution transcript* for V.

- The formula $\phi_{V,x}$ is carefully designed to evaluate whether the claimed tableau (as specified by the variables' values) meets two conditions:

  1. it is the *actual execution tableau* of $V(x, c)$, for the *specific given* instance x, and whatever certificate c appears in the first row, and
  2. it is an *accepting* tableau, i.e., $V(x, c)$ accepts.

  Conceptually, we can think of $\phi_{V,x}$ as *checking* whether the claimed execution transcript for V is genuine, and results in V accepting the specific instance x.

In summary, the formula $\phi_{V,x}$ evaluates to true *if and only if* its variables are set so that they specify the full and accepting execution tableau of $V(x, c)$, for the certificate c specified by the variables. Therefore, as needed,

$$\begin{aligned} \phi_{V,x} \in \text{SAT} &\iff \phi_{V,x} \text{ is satisfiable} \\ &\iff \text{there exists } c \text{ s.t. } V(x, c) \text{ accepts} \\ &\iff x \in L \ , \end{aligned}$$

where the last equivalence holds by Definition 132.

We now give the details. The variables of the formula are as follows:

For each cell position i, j of the tableau, and each symbol $s \in S$, there is a Boolean variable $t_{i,j,s}$.

So, there are $|S| \cdot n^{2k} = O(n^{2k})$ variables in total, which is polynomial in $n = |x|$ because $|S|$ and $2k$ are constants. (Recall that S is a fixed finite alphabet that does not vary with n.)

Assigning Boolean values to these variables specifies the contents of a *claimed* tableau, as follows:

Assigning $t_{i,j,s} =$ true specifies that cell i, j of the claimed tableau has symbol s in it.

Observe that the variables can be assigned *arbitrary* Boolean values, so for the same cell location i, j, we could potentially assign $t_{i,j,s} =$ true for *multiple* different values of s, or none at all. This would make the contents of cell i, j undefined. The formula $\phi_{V,x}$ is designed to "check" for this and evaluate to false in such a case, and also to check for all the other needed properties on the claimed tableau, as we now describe.

The formula $\phi_{V,x}$ is defined as the conjunction (AND) of four subformulas:

$$\phi_{V,x} = \phi_{\text{cell}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{start},x} \wedge \phi_{\text{move},V} \ .$$

Each subformula "checks" (or "enforces") a certain condition on the variables and the claimed tableau they specify, evaluating to true if the condition holds, and false if it does not. The subformulas check the following conditions:

- ϕ_{cell} checks that each cell of the claimed tableau is well defined, i.e., it contains exactly one symbol;
- ϕ_{accept} checks that the claimed tableau is an accepting tableau, i.e., that q_{acc} appears somewhere in it;
- $\phi_{\text{start},x}$ checks that the first row of the claimed tableau is valid, i.e., it represents the initial configuration of the verifier running on the *given instance* x and some certificate c (as specified by the variables);

- $\phi_{\text{move},V}$ checks that each non-starting row of the claimed tableau follows from the previous one, according to the transition function of V.

Observe that, as needed above, all these conditions hold *if and only if* the claimed tableau is indeed the *actual, accepting* execution tableau of $V(x, c)$, for the certificate c that appears in the first row. So, it just remains to show how to design the subformulas to correctly check their respective conditions, which we do next.

15.2.1 Cell Consistency

To check that a given cell i, j has a well-defined symbol, we need *exactly one* of the variables $t_{i,j,s}$ to be true, over all $s \in S$. The formula

$$\bigvee_{s \in S} t_{i,j,s}$$

checks that *at least one* of the variables is true, and the formula

$$\neg \bigvee_{\text{distinct } s,s' \in S} (t_{i,j,s} \wedge t_{i,j,s'}) = \bigwedge_{\text{distinct } s,s' \in S} (\neg t_{i,j,s} \vee \neg t_{i,j,s'})$$

checks that *no more than one* of the variables is true (where the equality holds by De Morgan's laws[53]).

Putting these together over all cells i, j, the subformula

$$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\bigvee_{s \in S} t_{i,j,s} \wedge \bigwedge_{\text{distinct } s,s' \in S} (\neg t_{i,j,s} \vee \neg t_{i,j,s'}) \right] .$$

checks that for all i, j, exactly one of $t_{i,j,s}$ is true, as needed.

The formula has $O(|S|^2) = O(1)$ literals for each cell (because S is a fixed alphabet that does not vary with $n = |x|$), so ϕ_{cell} has size $O(n^{2k})$, which is polynomial in $n = |x|$ because $2k$ is a constant.

15.2.2 Accepting Tableau

To check that the claimed tableau is an accepting one, we just need to check that at least one cell has q_{acc} as its symbol, which is done by the following subformula:

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} t_{i,j,q_{\text{acc}}} .$$

This has one literal per cell, for a total size of $O(n^{2k})$, which again is polynomial in $n = |x|$.

15.2.3 Starting Configuration

For the top row of the tableau, which represents the starting configuration, we suppose that the encoding of an input pair (x, c) separates x from c on the tape using a special symbol $\$ \in \Gamma \setminus \Sigma$ that is in the tape alphabet Γ but not in the input alphabet Σ. (Recall that x and c are strings over Σ, so \$ unambiguously separates them.) Letting $m = |c|$, the top row of the tableau is therefore as follows:

#	q_{st}	x_1	$\cdots$	x_n	\$	c_1	$\cdots$	c_m	$\perp$	$\cdots$	#

[53] https://en.wikipedia.org/wiki/De_Morgan%27s_laws

The row starts with the $\#$ symbol. Since the machine is in active state q_{start}, and the head is at the leftmost cell of the tape, q_{start} is the second symbol. The contents of the tape follow, which are the symbols of the input x, then the $\$$ separator, then the symbols of the certificate c, then blanks until we have n^k symbols, except for the last symbol, which is $\#$.

We now describe the subformula $\phi_{\text{start},x}$ that checks that the first row of the claimed tableau has the above form. We require the row to have the *specific, given* instance x in its proper position of the starting configuration. This is checked by the formula

$$\phi_{\text{input}} = t_{1,3,x_1} \wedge t_{1,4,x_2} \wedge \cdots \wedge t_{1,n+2,x_n} \ .$$

For the tableau cells corresponding to the certificate, **our construction does not know what certificate(s) c, if any, would cause $V(x,c)$ to accept**, or even what their sizes are (other than that they are less than n^k). Indeed, a main idea of this construction is that **any satisfying assignment for $\phi_{V,x}$, if one exists, specifies a valid certificate** c for x (and vice versa).

So, our formula allows *any* symbol $s \in \Sigma \cup \{\perp\}$ from the input alphabet, as well as blank, to appear in the positions after the separator $\$$. We just need to ensure that any blanks appear only *after* the certificate string, i.e., not before any symbol from Σ. Overall, the formula that checks that the portion of the first row dedicated to the certificate is well-formed is

$$\phi_{\text{cert}} = \bigwedge_{j=n+4}^{n^k-1} \left(t_{1,j,\perp} \vee \left(\bigvee_{s\in\Sigma} t_{1,j,s} \wedge \neg t_{1,j-1,\perp} \right) \right) \ .$$

In words, this says that each cell in the zone dedicated to the certificate either has a blank, or has an input-alphabet symbol *and* the preceding symbol is not blank.

Putting these pieces together with the few other fixed cell values, the subformula that checks the first row of the claimed tableau is:

$$\phi_{\text{start},x} = t_{1,1,\#} \wedge t_{1,2,q_{\text{start}}} \wedge \phi_{\text{input}} \wedge t_{1,n+3,\$} \wedge \phi_{\text{cert}} \wedge t_{1,n^k,\#} \ .$$

This subformula has one literal for each symbol in the instance x, the start state, and the special $\#$ and $\$$ symbols. It has $O(|\Sigma|) = O(1)$ literals for each cell in the zone dedicated to the certificate (recall that the tape alphabet Γ is fixed and does not vary with $n = |x|$). Thus, this subformula has size $O(n^k)$, which again is polynomial in $n = |x|$.

15.2.4 Transitions

Finally, we describe the subformula that checks whether every non-starting row in the claimed tableau follows from the previous one. This is the most intricate part of the proof, and the only one that relies on the transition function, or "code", of V.

As a warmup first attempt, recall that any syntactically legal configuration string r (regardless of whether V can actually reach that configuration) determines the next configuration string r', according to the code of V. So, letting $i, i+1$ denote a pair of adjacent row indices, and r denote a legal configuration string, we could write:

1. the formula that checks whether row i equals r *and* row $i+1$ equals the corresponding r', namely,

$$\phi_{i,r} = \bigwedge_{j=1}^{n^k} (t_{i,j,r_j} \wedge t_{i+1,j,r'_j}) \ ;$$

2. the formula that checks whether row $i+1$ follows from row i, namely,

$$\phi_i = \bigvee_{\text{legal } r} \phi_{i,r} \ ;$$

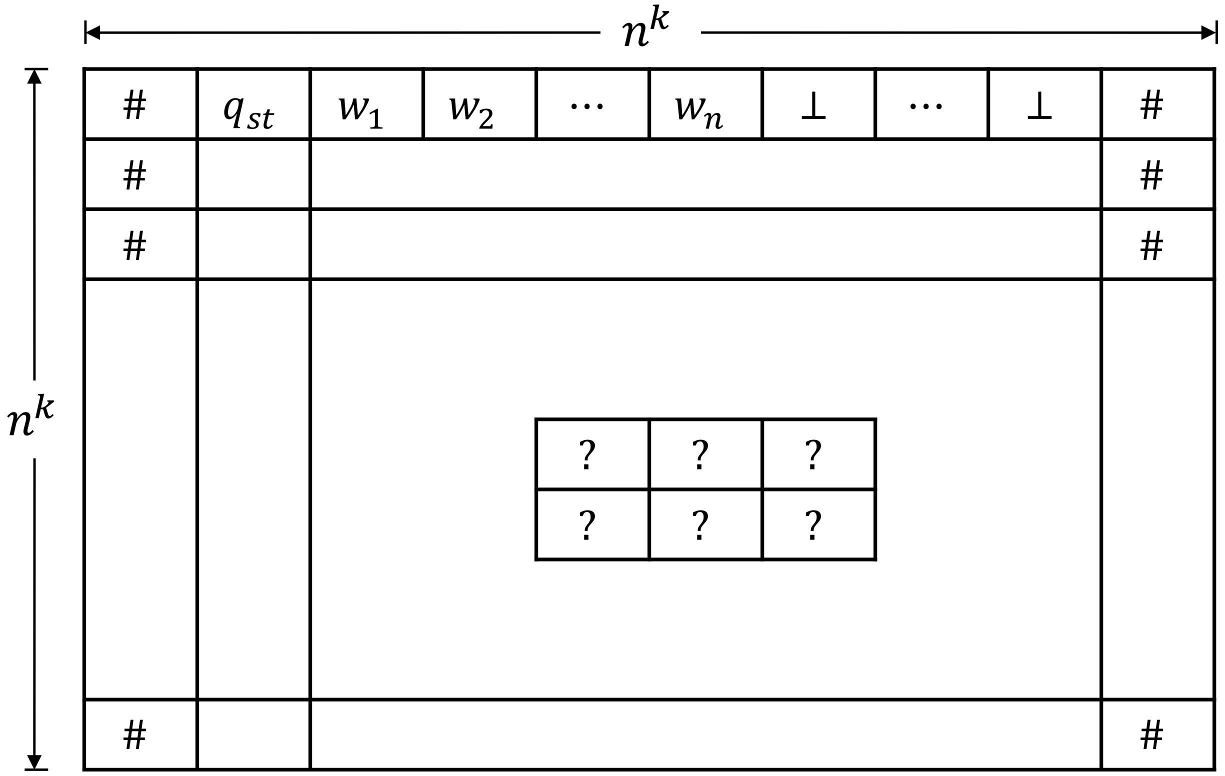
3. the formula that checks whether *every* non-starting row follows from the previous one, namely,

$$\phi_{\text{move},V} = \bigwedge_{i=1}^{n^k-1} \phi_i \ .$$

Because $\phi_{\text{start},x}$ checks that the first row of the claimed tableau is a valid starting configuration, $\phi_{\text{start},x} \wedge \phi_{\text{move},V}$ does indeed check that the entire claimed tableau is valid.

This approach gives a *correct* formula, but unfortunately, it is not *efficient*—the formula does not have size polynomial in $n = |x|$. The only problem is in the second step: because there are multiple valid choices for each symbol of a legal r, and r has length n^k, there are *exponentially many* such r. So, taking the OR over all of them results in a formula of exponential size.

To overcome this efficiency problem, we proceed similarly as above, but using a finer-grained breakdown of adjacent rows than in $\phi_{i,r}$ above. The main idea is to consider the *2-by-3 "windows"* (subrectangles) of adjacent rows, and check that every window is "valid" according to the code of V. It can be proved that if all the overlapping 2-by-3 windows of a claimed pair of rows are valid, then the pair *as a whole* is valid. This is essentially because valid adjacent rows are almost identical (except around the cells with the active states), and the windows overlap sufficiently to guarantee that any invalidity in the claimed rows will be exposed in some 2-by-3 window.[54]



There are $|S|^6 = O(1)$ distinct possibilities for a 2-by-3 window, but not all of them are valid. Below we describe all the valid windows, and based on this we construct the subformula $\phi_{\text{move},V}$ as follows.

For any particular valid window w, let $s_1, \dots, s_6$ be its six symbols, going left-to-right across its first then second row. Similarly to $\phi_{i,r}$ above, the following simple formula checks whether the window of the claimed tableau with top-left corner at i, j matches w:

$$\phi_{i,j,w} = (t_{i,j,s_1} \wedge t_{i,j+1,s_2} \wedge t_{i,j+2,s_3} \wedge t_{i+1,j,s_4} \wedge t_{i+1,j+1,s_5} \wedge t_{i+1,j+2,s_6}) \ .$$

[54] Interestingly, the same does not hold for 2-by-2 windows: it is possible to construct a pair of rows in which every 2-by-2 window is valid, but the pair of rows as a whole does *not* represent a correct transition according to the code of V. So, we must use windows of width at least 3, and 3 suffices.

Now, letting W be the set of all valid windows, similarly to ϕ_i above, the following formula checks whether the window at i, j of the claimed tableau is valid:

$$\phi_{i,j} = \bigvee_{w \in W} \phi_{i,j,w} \ .$$

Finally, we get the subformula that checks whether *every* window in the tableau is valid:

$$\phi_{\text{move},V} = \bigwedge_{\substack{1 \le i \le n^k - 1 \\ 1 \le j \le n^k - 2}} \phi_{i,j} \ .$$

The set W of valid windows has size $|W| \le |S|^6 = O(1)$, so each $\phi_{i,j}$ has $O(1)$ literals. Because there are $O(n^{2k})$ windows to check, the subformula $\phi_{\text{move},V}$ has size $O(n^{2k})$, which again is polynomial in $n = |x|$.

We now describe all the valid windows. The precise details of this are not so essential, because all we used above were the facts that the valid windows are well defined, and that there are $O(1)$ of them. We use the following notation in the descriptions and diagrams:

- γ for an arbitrary element of Γ,
- q for an arbitrary element of Q,
- ρ for an arbitrary element of $\Gamma \cup Q$,
- σ for an arbitrary element of $\Gamma \cup \{\#\}$.

First, a window having the $\#$ symbol somewhere in it is valid only if $\#$ is in the first position of both rows, or is in the third position of both rows. So, every valid window has one of the following forms, where the symbols $\rho_1, \ldots, \rho_6$ must additionally be valid according to the rules below.

#	ρ_1	ρ_2
#	ρ_3	ρ_4

ρ_1	ρ_2	ρ_3
ρ_4	ρ_5	ρ_6

ρ_1	ρ_2	#
ρ_3	ρ_4	#

If a window is not in the vicinity of a state symbol $q \in Q$, then the machine's transition does not affect the portion of the configuration corresponding to the window, so the top and bottom rows match:

σ_1	γ	σ_2
σ_1	γ	σ_2

To reason about what happens in a window that is in the vicinity of a state symbol, we consider how the configuration changes as a result of a right-moving transition $\delta(q, \gamma) = (q', \gamma', R)$:

γ_2	γ_3	γ_4	γ_5	q	γ	γ_6	γ_7	γ_8
γ_2	γ_3	γ_4	γ_5	γ'	q'	γ_6	γ_7	γ_8
		1	2	3	4			

The head moves to the right, so q' appears in the bottom row at the column to the right of where q appears in the top row. The symbol γ is replaced by γ', though its position moves to the left to compensate for the rightward movement of the state symbol. There are four windows affected by the transition, labeled above by their leftmost columns. We thus have the following four kinds of valid windows corresponding to a rightward transition $\delta(q, \gamma) = (q', \gamma', R)$:

σ	γ_2	q
σ	γ_2	γ'

σ	q	γ
σ	γ'	q'

q	γ	σ
γ'	q'	σ

γ	γ_2	σ
q'	γ_2	σ

Note that we have used σ for the edges of some windows, as there can be either a tape symbol or the tableau-edge marker $\#$ in these positions.

We now consider how the configuration changes as a result of a left-moving transition $\delta(q, \gamma) = (q', \gamma', L)$:

γ_2	γ_3	γ_4	γ_5	q	γ	γ_6	γ_7	γ_8
γ_2	γ_3	γ_4	q'	γ_5	γ'	γ_6	γ_7	γ_8
	1	2	3	4	5			

The head moves to the left, so q' appears in the bottom row at the column to the left of where q appears in the top row. As with a rightward transition, we replace the old symbol γ with the new symbol γ'. This time, however, the symbol to the left of the original position of the head moves right to compensate for the leftward movement of the head. So here we have the following five kinds of valid windows corresponding to the leftward transition $\delta(q, \gamma) = (q', \gamma', R)$:

σ	γ_3	γ_2
σ	γ_3	q'

σ	γ_2	q
σ	q'	γ_2

γ_2	q	γ
q'	γ_2	γ'

q	γ	σ
γ_2	γ'	σ

γ	γ_2	σ
γ'	γ_2	σ

We also need to account for the case where the head is at the leftmost cell on the tape, in which case the head does not move:

#	q	γ	γ_2	γ_3	γ_4
#	q'	γ'	γ_2	γ_3	γ_4
1	2	3			

Here we have three kinds of valid windows, but the last one actually has the same structure as the last window in the normal case for a leftward transition. Thus, we have only two more kinds of valid windows:

#	q	γ
#	q'	γ'

q	γ	γ_2
q'	γ'	γ_2

Finally, we need one last set of valid windows to account for the machine reaching the accept state before the last row of the tableau. As stated above, the valid tableau is "padded" with copies of the final configuration, so the valid windows for this case look like:

q_{acc}	γ	σ
q_{acc}	γ	σ

σ_1	q_{acc}	σ_2
σ_1	q_{acc}	σ_2

σ	γ	q_{acc}
σ	γ	q_{acc}

15.3 Conclusion

We summarize the main claims about what we have just done.

Using an efficient verifier V for an arbitrary language $L \in \mathsf{NP}$, we have demonstrated how to efficiently construct a corresponding formula $\phi_{V,x}$ from a given instance x of L. The formula can be constructed in time polynomial in $|x|$, and in particular the total size of the formula is $O(|x|^{2k})$ literals.

Then, we (almost entirely) proved that $x \in L \iff \phi_{V,x} \in \text{SAT}$. In the $\Longrightarrow$ direction: by correctness of V, any $x \in L$ has at least one certificate c that makes $V(x, c)$ accept, so the computation tableau for $V(x, c)$ is accepting. In turn, by design of the formula $\phi_{V,x}$, this tableau defines a satisfying assignment for the formula (i.e., an assignment to its variables that makes $\phi_{V,x}$ evaluate to true). So, $x \in L$ implies that $\phi_{V,x} \in \text{SAT}$.

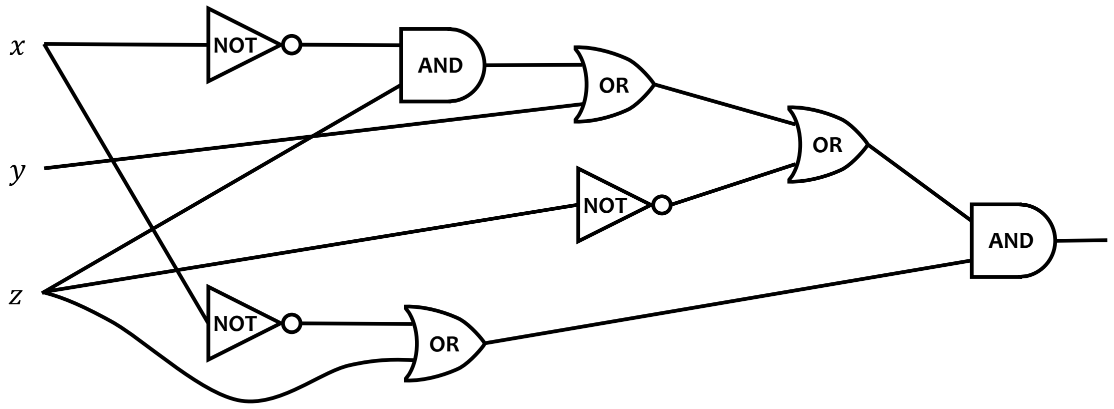
In the $\Longleftarrow$ direction: if the constructed formula $\phi_{V,x} \in \text{SAT}$, then by definition it has a satisfying assignment. Again by the special design of the formula, any satisfying assignment defines the contents of an actual, accepting computation tableau for $V(x, c)$, for the c appearing in the top row of the tableau. Since $V(x, c)$ accepts for this c, we conclude that $x \in L$ by the correctness of the verifier. So, $\phi_{V,x} \in \text{SAT}$ implies that $x \in L$.

Finally, if $\text{SAT} \in \mathsf{P}$, then an efficient decider D_{SAT} for SAT exists, and we can use it to decide L efficiently: given an instance x, simply construct $\phi_{V,x}$ and output $D_{\text{SAT}}(\phi_{V,x})$; by the above property, this correctly determines whether $x \in L$. As a result, $\text{SAT} \in \mathsf{P}$ implies that $\mathsf{NP} \subseteq \mathsf{P}$ and hence $\mathsf{P} = \mathsf{NP}$, as claimed.

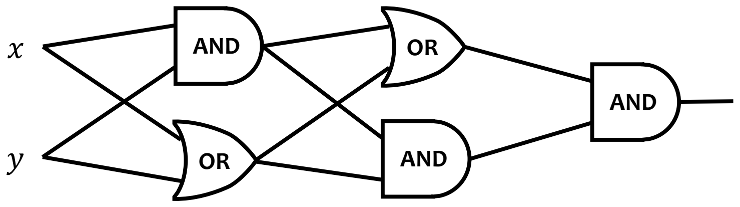
Circuit Satisfiability

A problem that is equivalent to formula satisfiability is that of *circuit satisfiability*. Given a circuit C composed of m logic gates over n binary inputs and with a single binary output, is there a set of input values such that the output is 1? In other words, is there an assignment to the inputs that satisfies the circuit?

Without loss of generality, assume that each logic gate is either a unary NOT gate, a binary AND gate, or a binary OR gate. (This is a *universal gate set*, i.e., any Boolean function can be expressed in terms of these gates.) Since these gates correspond to negation, conjunction, and disjunction, we can mechanically convert a Boolean formula into an equivalent circuit whose size is linear in the size of the formula. The following is a circuit corresponding to the formula $(y \vee (\neg x \wedge z) \vee \neg z) \wedge (\neg x \vee z)$:



What about the other direction, of converting a circuit into a formula? In general, the gates in a circuit may have multiple *fan-out*, meaning that the output of a gate may be used as the inputs to multiple other gates. The following is an example:

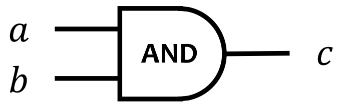


A naïve translation can result in an exponential increase in size: wherever there is multiple fan-out, the subformula corresponding to the intermediate result would be repeated. For the circuit above, such a translation would be

$$((x \wedge y) \vee (x \vee y)) \wedge ((x \wedge y) \wedge (x \vee y)) \ .$$

There are two copies each of the subformulas $x \wedge y$ and $x \vee y$. For larger circuits, a subformula may be repeated many times.

Rather than naïvely repeating, for the output of each gate we can introduce a new variable. For instance, the following is an AND gate with existing variables a and b as inputs, and a new variable c as output:



The relationship between the inputs a and b and the output c is $c = a \wedge b$. A formula cannot directly express equality, but observe that variables x and y are equal exactly when either both of them are true, or both are false. Thus, $x = y$ is logically equivalent to $(x \wedge y) \vee (\neg x \wedge \neg y)$. For $c = a \wedge b$, this idea translates to

$$\begin{aligned} c = a \wedge b &\equiv (c \wedge (a \wedge b)) \vee (\neg c \wedge \neg(a \wedge b)) \\ &\equiv (c \wedge (a \wedge b)) \vee (\neg c \wedge (\neg a \vee \neg b)) \ . \end{aligned}$$

In the second step, we applied De Morgan's laws[55] to move negation inwards, so that only variables (and not subformulas) are negated.

We can similarly express an OR gate with inputs a and b and output c:

$$\begin{aligned} c = a \vee b &\equiv (c \wedge (a \vee b)) \vee (\neg c \wedge \neg(a \vee b)) \\ &\equiv (c \wedge (a \vee b)) \vee (\neg c \wedge (\neg a \wedge \neg b)) \ . \end{aligned}$$

For a NOT gate with input a, we can simply express the output as $\neg a$ without introducing a new variable.

Now that we have subformulas for each gate, the formula for the full circuit is just the conjunction of the subformulas for each gate. Since each gate introduces at most one variable and a subformula of at most six literals, the size of the resulting formula is linear in the size of the circuit.

We have demonstrated that the formula-satisfiability problem SAT is equivalent to the circuit-satisfiability problem CSAT. Since the former is a "hardest" problem in NP, so is the latter. Next, we generalize and formalize the notion of efficient transformations between problems by introducing the notion of *polynomial-time mapping reductions*.

[55] https://en.wikipedia.org/wiki/De_Morgan%27s_laws

CHAPTER

SIXTEEN

NP-COMPLETENESS

With the example of SAT and the Cook-Levin theorem in hand, we now formally define what it means to be a "hardest" problem in NP, and demonstrate several examples of such problems.

16.1 Polynomial-Time Mapping Reductions

The heart of the Cook-Levin theorem is an efficient algorithm for converting an instance of an (arbitrary) language $L \in \mathsf{NP}$ to an instance $\phi_{V,x}$ of SAT, such that

$$x \in L \iff \phi_{V,x} \in \text{SAT} .$$

Thus, any (hypothetical) efficient decider for SAT yields an efficient decider for L, which simply converts its input x to $\phi_{V,x}$ and invokes the decider for SAT on it.

This kind of efficient transformation, from an arbitrary instance of one problem to a corresponding instance of another problem having the same "yes/no answer", is known as a *polynomial-time mapping reduction*, which we now formally define.

Definition 145 (Polynomial-Time Mapping Reduction) A *polynomial-time mapping reduction*—also known as a *Karp reduction* for short—from a language A to a language B is a function $f \colon \Sigma^* \to \Sigma^*$ having the following properties:[56]

1. **Efficiency**: f is *polynomial-time computable*: there is an algorithm that, given any input $x \in \Sigma^*$, outputs $f(x)$ in time polynomial in $|x|$.
2. **Correctness**: for all $x \in \Sigma^*$,

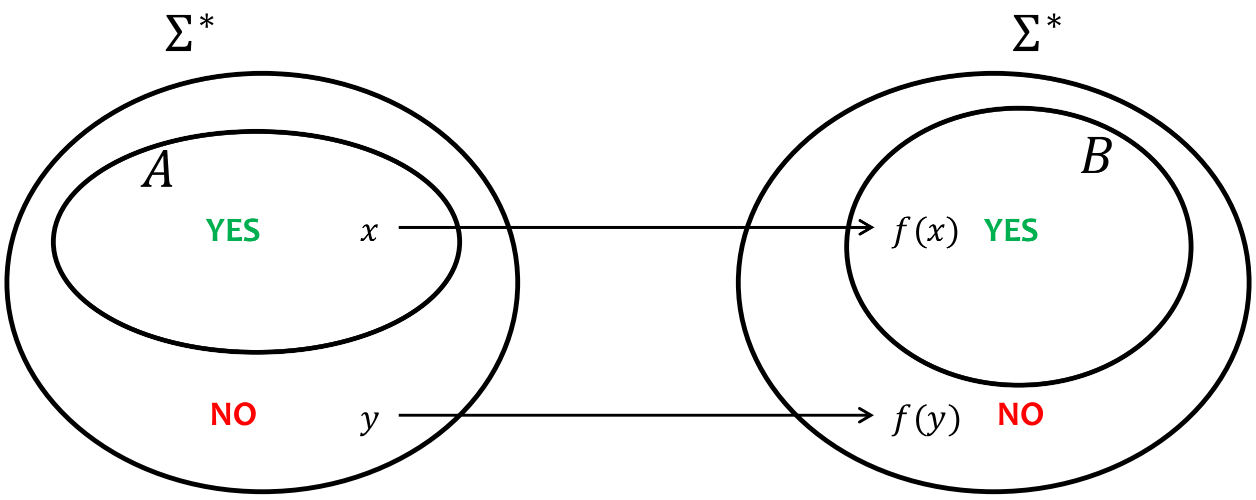
$$x \in A \iff f(x) \in B .$$

Equivalently, if $x \in A$ then $f(x) \in B$, and if $x \notin A$ then $f(x) \notin B$.[57]

If such a reduction exists, we say that A *polynomial-time mapping-reduces* (or *Karp-reduces*) to B, and write $A \leq_p B$.

[56] It is meaningful to drop the efficiency property and consider just (correct) mapping reductions, but these are not useful in the context of P and NP, so in this text we always require mapping reductions to be efficient.

[57] The claimed equivalence follows just by taking the contrapositive of the direction $x \in A \impliedby f(x) \in B$.



We first observe that a polynomial-time mapping reduction is essentially a special case of (or more precisely, implies the existence of) a *Turing reduction* (page 105): we can decide A by converting the input instance of A to an instance of B using the efficient transformation function, then querying an oracle that decides B. The following makes this precise.

Lemma 146 *If $A \le_p B$, then $A \le_T B$.*

Proof 147 Because $A \le_p B$, there is an efficiently computable function f such that $x \in A \iff f(x) \in B$. To show that $A \le_T B$, we give a Turing machine M_A that decides A using an oracle M_B that decides B:

```
function $M_A(x)$
    compute $y = f(x)$
    return $M_B(y)$
```

Clearly, M_A halts on any input, because f is polynomial-time computable and M_B halts on any input. And by the correctness of M_B and the code of M_A,

$$\begin{aligned} x \in A &\iff y = f(x) \in B \\ &\iff M_B(y) \text{ accepts} \\ &\iff M_A(x) \text{ accepts} . \end{aligned}$$

Therefore, M_A decides A, by Definition 54. □

Observe that the Turing reduction given in Proof 147 simply:

1. applies an efficient transformation to its input,
2. invokes its oracle *once*, on the resulting value, and
3. immediately outputs the oracle's answer.

In fact, several (but not all!) of the Turing reductions we have seen for proving undecidability (e.g., for the *halts-on-empty language* (page 107) and *other undecidable languages* (page 109)) are effectively polynomial-time mapping reductions, because they have this exact form. (To be precise, the *efficient transformation* in such a Turing reduction is the mapping reduction, and its use of its oracle is just fixed "boilerplate".)

Although a polynomial-time mapping reduction is effectively a Turing reduction, the reverse does not hold in general, due to some important differences:

- A Turing reduction has no efficiency constraints, apart from the requirement that it halts: the reduction may run for arbitrary finite time, both before and after its oracle call(s).

By contrast, in a polynomial-time mapping reduction, the output of the conversion function must be computable in time polynomial in the input size.

- A Turing reduction is a Turing machine that decides one language given an oracle ("black box") that decides another language, which it may use arbitrarily. In particular, it may invoke the oracle multiple times (or not at all), and perform arbitrary "post-processing" on the results, e.g., negate the oracle's answer.

  By contrast, a polynomial-time mapping reduction does not involve any explicit oracle; it is merely a conversion function that must "preserve the yes/no answer". Therefore, there is no way for it to "make multiple oracle calls," nor for it to "post-process" (e.g., negate) the yes/no answer for its constructed instance. Implicitly, a polynomial-time mapping reduction corresponds to making just one oracle call and then immediately outputting the oracle's answer.[58]

In Lemma 91 we saw that if $A \le_T B$ and B decidable, then A is also decidable. Denoting the class of decidable languages by R, this result can be restated as:

If $A \le_T B$ and $B \in \mathsf{R}$, then $A \in \mathsf{R}$.

Polynomial-time mapping reductions give us an analogous result with respect to membership in the class P.

Lemma 148 *If $A \le_p B$ and $B \in \mathsf{P}$, then $A \in \mathsf{P}$.*

Proof 149 Because $B \in \mathsf{P}$, there is a polynomial-time Turing machine M_B that decides B. And because $A \le_p B$, the Turing machine M_A defined in Proof 147, with the machine M_B as its "oracle", decides A.

In addition, $M_A(x)$ runs in time polynomial in its input size $|x|$, by composition of polynomial-time algorithms. Specifically, by the hypothesis $A \le_p B$, computing $f(x)$ runs in time polynomial in $|x|$, and as already noted, M_B runs in time polynomial in its input length, so the composed algorithm M_A runs in polynomial time.

Since M_A is a polynomial-time Turing machine that decides A, we conclude that $A \in \mathsf{P}$, as claimed. $\square$

Analogously to how Lemma 93 immediately follows from Lemma 91, the following corollary is merely the contrapositive of Lemma 148.

Lemma 150 *If $A \le_p B$ and $A \notin \mathsf{P}$, then $B \notin \mathsf{P}$.*

16.2 NP-Hardness and NP-Completeness

With the notion of a polynomial-time mapping (or Karp) reduction in hand, the heart of the Cook-Levin theorem can restated as saying that

$A \le_p \text{SAT}$ for *every* language $A \in \mathsf{NP}$.

Combining this with Lemma 148, as an immediate corollary we get the statement of Theorem 144: if SAT $\in \mathsf{P}$, then *every* NP language is in P, i.e., $\mathsf{P} = \mathsf{NP}$.

Informally, the above says that SAT is "at least as hard as" every language in NP (under Karp reductions). This is a very important property that turns out to be shared by many languages, so we define a special name for it.

Definition 151 (NP-Hard) A language L is NP-*hard* if $A \le_p L$ for *every* language $A \in \mathsf{NP}$.

We define $\mathsf{NPH} = \{L : L \text{ is NP-hard}\}$ to be the class of all such languages.

[58] Alternatively, we could consider a relaxed definition that allows for multiple oracle calls and arbitrary polynomial-time pre- and post-processing. In other words, this would be a *polynomial-time Turing reduction*, also known as a *Cook reduction*. Such reductions are also very useful, but they do not allow us to make distinctions between complexity classes like NP and coNP, hence the focus on efficient mapping reductions.

We stress that a language need not be in NP, or even be decidable, to be NP-hard. For example, it can be shown that the undecidable language L_{ACC} is NP-hard (see Exercise 160).

With the notion of NP-hardness in hand, the core of the Cook-Levin theorem is as follows.

Theorem 152 (Cook-Levin core) *SAT is* NP-*hard.*

Previously, we mentioned the existence of languages that are, informally, the "hardest" ones in NP, and that SAT was one of them. We now formally define this notion.

Definition 153 (NP-Complete) A language L is NP-*complete* if:

1. $L \in \mathsf{NP}$, and
2. L is NP-hard.

We define $\mathsf{NPC} = \{L : L \text{ is NP-complete}\}$ to be the class of all such languages.

Since SAT $\in$ NP (by Lemma 142) and SAT is NP-hard (by Cook-Levin), SAT is indeed NP-complete.

To show that some language L of interest is NP-hard, do we need to repeat and adapt all the work of the Cook-Levin theorem, with L in place of SAT? Thankfully, we do not! Analogously to Lemma 93—which lets us prove *undecidability* by giving a *Turing* reduction from a known-undecidable language—the following lemma shows that we can establish NP-*hardness* by giving a *Karp* reduction from a known-NP-hard language. (Below we do this for several concrete problems of interest, in *More NP-complete Problems* (page 168).)

Lemma 154 *If $A \leq_p B$ and A is* NP-*hard, then B is* NP-*hard.*

Proof 155 This follows from the fact that $\leq_p$ is a *transitive* relation; see Exercise 158 below. By the two hypotheses and Definition 151, $L \leq_p A$ for every $L \in \mathsf{NP}$, and $A \leq_p B$, so $L \leq_p B$ by transitivity. Since this holds for every $L \in \mathsf{NP}$, by definition we have that B is NP-hard, as claimed. □

Combining Lemma 148, Lemma 150, and Lemma 154, the fact that $A \leq_p B$ has the following implications in various scenarios.

Hypothesis	**Implies**
$A \in \mathsf{P}$	nothing
$A \notin \mathsf{P}$	$B \notin \mathsf{P}$
A is NP-hard	B is NP-hard
$B \in \mathsf{P}$	$A \in \mathsf{P}$
$B \notin \mathsf{P}$	nothing
B is NP-hard	nothing

16.3 Resolving P versus NP

The concepts of NP-hardness and NP-completeness are powerful tools for making sense of, and potentially resolving, the P-versus-NP question. As the following theorem shows, the two possibilities $\mathsf{P} = \mathsf{NP}$ and $\mathsf{P} \neq \mathsf{NP}$ each come with a variety of equivalent "syntactically weaker" conditions. So, resolving P versus NP comes down to establishing *any one* of these conditions (which is still a very challenging task!). In addition, the theorem establishes strict relationships between the various classes of problems we have defined, under each of the two possibilities $\mathsf{P} = \mathsf{NP}$ and $\mathsf{P} \neq \mathsf{NP}$.

Theorem 156 (P versus NP) *The following statements are* equivalent, *i.e., if any* one *of them holds, then* all *of them hold.*

1. Some NP-*hard language is in* P.

   (In set notation: $\mathsf{NPH} \cap \mathsf{P} \neq \emptyset$.*)*

2. Every NP-*complete language is in* P.

   (In set notation: $\mathsf{NPC} \subseteq \mathsf{P}$.*)*

3. $\mathsf{P} = \mathsf{NP}$.

   (In words: every language in NP *is also in* P*, and vice-versa.)*

4. Every *language in* NP*, except the "trivial" languages* Σ^* *and* $\emptyset$*, is* NP-*hard (and hence* NP-*complete).*[59]

   (In set notation: $\mathsf{NP} \setminus \{\Sigma^*, \emptyset\} \subseteq \mathsf{NPH}$.*)*

It follows that $\mathsf{P} \neq \mathsf{NP}$ *if and only if* no NP-*hard language is in* P *(i.e.,* $\mathsf{NPH} \cap \mathsf{P} = \emptyset$*), which holds if and only if* some *nontrivial language in* NP *is* not NP-*hard.*

[59] We remark that Statement 4 *must* exclude the two trivial languages Σ^* and $\emptyset$, because they are *not* NP-hard, regardless of whether $\mathsf{P} = \mathsf{NP}$; see Exercise 159.

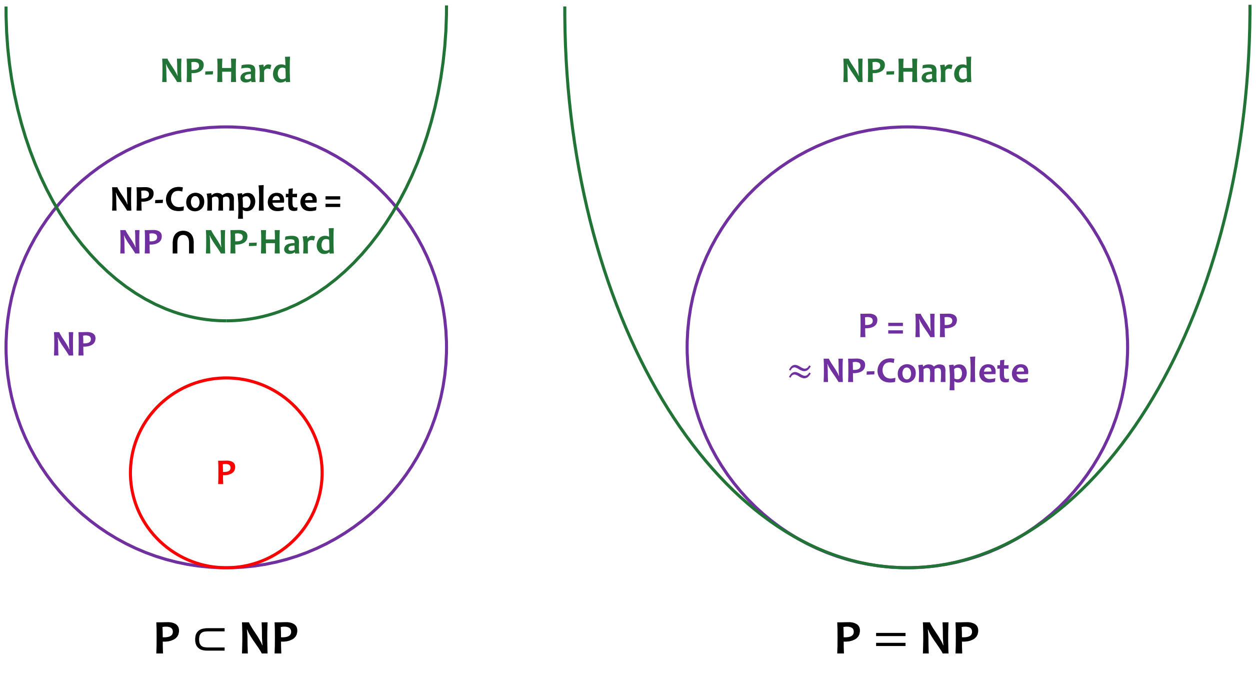
Before giving the proof of Theorem 156, we discuss its main consequences for attacking the P-versus-NP question. First, by the equivalence of Statements 1 and 2, **all NP-complete problems have the same "status"** : either *all* of them are efficiently solvable—in which case $\mathsf{P} = \mathsf{NP}$, by the equivalence with Statement 3—or *none* of them is, in which case $\mathsf{P} \neq \mathsf{NP}$.

So, to prove that $\mathsf{P} = \mathsf{NP}$, it would be sufficient (and necessary) to have an efficient algorithm for *any single* NP-complete (or even just NP-hard) problem. This is by the equivalence of Statements 1 and 3.

To prove that $\mathsf{P} \neq \mathsf{NP}$, it would trivially suffice to prove that *any single* problem in NP is *not* in P. For this we might as well focus our efforts on showing this for some NP-*complete* problem, because by the equivalence of Statements 3 and 1, *some* NP problem lacks an efficient algorithm if and only if *every* NP-complete problem does.

Alternatively, to prove that $\mathsf{P} \neq \mathsf{NP}$, by the equivalence of Statements 3 and 4 it would suffice to show that some nontrivial language in NP is *not* NP-hard. For this we might as well focus our efforts on showing this for some language in P, because by the equivalence of Statements 4 and 1, if *any* nontrivial language in NP is not NP-hard, then *every* nontrivial language in P is not NP-hard.

Based on Theorem 156, the following Venn diagram shows the necessary and sufficient relationships between the classes P, NP, NP-hard (NPH), and NP-complete (NPC), for each of the two possibilities $\mathsf{P} \neq \mathsf{NP}$ and $\mathsf{P} = \mathsf{NP}$.



Proof 157 We refer to Statement 1 as "S1", and similarly for the others. The following implications hold immediately by the definitions (and set operations):

- S2 $\implies$ S1 because an NP-complete, and hence NP-hard, language exists (e.g., SAT).

  (In set notation: if $\text{NPH} \cap \text{NP} = \text{NPC} \subseteq \text{P}$, then by intersecting with P, we get that $\text{NPH} \cap \text{P} = \text{NPC} \neq \emptyset$.)

- S3 $\implies$ S2 because by definition, every NP-complete language is in NP.

  (In set notation: if $\text{P} = \text{NP}$, then $\text{NPC} \subseteq \text{NP} \subseteq \text{P}$.)

- S4 $\implies$ S1 because there is a nontrivial language $L \in \text{P} \subseteq \text{NP}$ (e.g., $L = \text{MAZE}$), and by S4, L is NP-complete and hence NP-hard.

  (In set notation: $\emptyset \neq \text{P} \setminus \{\Sigma^*, \emptyset\} \subseteq \text{NP} \setminus \{\Sigma^*, \emptyset\} \subseteq \text{NPH}$, where the last inclusion is by S4, so by intersecting with P, we get that $\emptyset \neq \text{P} \setminus \{\Sigma^*, \emptyset\} \subseteq \text{NPH} \cap \text{P}$ and hence $\text{NPH} \cap \text{P} \neq \emptyset$.)

So, to prove that each statement implies every other one, it suffices to show that S1 $\implies$ S3 $\implies$ S4.

For S1 $\implies$ S3, suppose that some NP-hard language L is in P. By definition, $A \leq_p L$ for all $A \in \text{NP}$, so by Lemma 148, $A \in \text{P}$. Therefore, $\text{NP} \subseteq \text{P}$, and this is an equality because $\text{P} \subseteq \text{NP}$, as we have already seen.

For S3 $\implies$ S4, suppose that $\text{P} = \text{NP}$ and let $L \in \text{NP}$ arbitrary but nontrivial; we show that L is NP-hard. By nontriviality, there exist some *fixed, distinct* strings $y \in L$ and $z \notin L$. Let $A \in \text{NP} = \text{P}$ be arbitrary, so there is an efficient algorithm M_A that decides A. We show that $A \leq_p L$ via the function f computed by the following pseudocode:

```
function $F(x)$
    if $M_A(x)$ accepts then return $y$
    return $z$
```

It is apparent that F is efficient, because M_A is. And for correctness, by the correctness of M_A, the properties of

$y \neq z$, and the code of F,

$$\begin{aligned} x \in A &\iff M_A(x) \text{ accepts} \\ &\iff f(x) = y \\ &\iff f(x) \in L\ . \end{aligned}$$

Since $A \leq_p L$ for all $A \in \mathsf{NP}$, we have shown that L is NP-hard, as needed. □

Exercise 158 Show that polynomial-time mapping reductions are *transitive*. That is, if $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$.

Exercise 159 Show that *no* language Karp-reduces to Σ^* or to $\emptyset$. In particular, these languages are *not* NP-hard, regardless of whether $\mathsf{P} = \mathsf{NP}$.

Exercise 160 Show that L_{ACC} is NP-hard.

CHAPTER

SEVENTEEN

MORE NP-COMPLETE PROBLEMS

The Cook-Levin Theorem (Theorem 152, showing that SAT is NP-hard), together with the concept of polynomial-time mapping reductions and Lemma 154, are powerful tools for establishing the NP-hardness of a wide variety of other natural problems. We do so for several such problems next.

17.1 3SAT

As a second example of an NP-complete problem, we consider satisfiability of Boolean formulas with a specific structure. First, we introduce a few more terms that apply to Boolean formulas:

- A *clause* is a disjunction of literals, such as $(a \vee b \vee \neg c \vee d)$.
- A Boolean formula is in *conjunctive normal form (CNF)* if it consists of a conjunction of clauses. An example of a CNF formula is:

$$\phi = (a \vee b \vee \neg c \vee d) \wedge (\neg a) \wedge (a \vee \neg b) \wedge (c \vee d)$$

- A *3CNF formula* is a Boolean formula in conjunctive normal form where each clause has **exactly** three literals, such as the following:

$$\phi = (a \vee b \vee \neg c) \wedge (\neg a \vee \neg a \vee d) \wedge (a \vee \neg b \vee d) \wedge (c \vee d \vee d)$$

We then define the language 3SAT as the set of satisfiable 3CNF formulas:

$$\text{3SAT} = \{\phi : \phi \text{ is a satisfiable 3CNF formula}\}$$

Observe that 3SAT contains strictly fewer formulas than SAT, i.e. 3SAT $\subset$ SAT. Despite this, we will demonstrate that 3SAT is NP-complete.

First, we must show that 3SAT $\in$ NP. The verifier is the same as the one for SAT, except that it is specialized to 3CNF formulas:

```
Input: instance: a 3CNF formula; certificate: assignment to its variables
Output: whether the assignment satisfies the formula
  function $V(\phi, \alpha)$
    if $\phi(\alpha)$ = true then accept
    reject
```

The value of $\phi(\alpha)$ can be computed in linear time, and we can also check that ϕ is in 3CNF and that α is a valid assignment in linear time, so V is efficient in $|\phi|$. Furthermore, if ϕ is in 3CNF and is satisfiable, there is some certificate α such that $V(\phi, \alpha)$ accepts, but if ϕ is not in 3CNF or is unsatisfiable, $V(\phi, \alpha)$ rejects for all α. Thus, V efficiently verifies 3SAT.

We now show that 3SAT is NP-hard with the reduction SAT $\le_p$ 3SAT. Given a Boolean formula ϕ, we need to efficiently compute a new formula $f(\phi)$ such that $\phi \in$ SAT $\iff f(\phi) \in$ 3SAT. We can do so as follows:

1. First, convert ϕ into a formula ϕ' in conjunctive normal form. We can do so efficiently, *as described in detail below* (page 169).
2. Second, convert the CNF formula ϕ' into the 3CNF formula $f(\phi)$. We can do so as follows:
   a) For each clause consisting of fewer than three literals, repeat the first literal to produce a clause with three literals. For example, (a) becomes $(a \vee a \vee a)$, and $(\neg b \vee c)$ becomes $(\neg b \vee \neg b \vee c)$.
   b) For each clause with more than three literals, subdivide it into two clauses by introducing a new dummy variable. In particular, convert the clause

   $$(l_1 \vee l_2 \vee \cdots \vee l_m)$$

   into

   $$(z \vee l_1 \vee \cdots \vee l_{\lfloor m/2 \rfloor}) \wedge (\neg z \vee l_{\lfloor m/2 \rfloor + 1} \vee \cdots \vee l_m)$$

   Observe that if all l_i are false, then the original clause is not satisfied. The transformed subformula is not satisfied for any value of z, since it would require $z \wedge \neg z$, which is impossible. On the other hand, if at least one l_i is true, then the original clause is satisfied. The transformed subformula can be satisfied by choosing z to be true if $i > \lfloor m/2 \rfloor$ or z to be false if $i \le \lfloor m/2 \rfloor$. Thus, this transformation preserves the satisfiability of the original clause.

   We repeat the process on the clauses of the transformed subformula, until all clauses have size exactly three. Each transformation introduces $O(1)$ literals, giving us the (approximate) recurrence

   $$T(m) = 2T(\frac{m}{2}) + O(1)$$

   By the Master Theorem, we conclude that $T(m) = O(m)$, so the recursive transformation increases the size of the formula by only a linear amount.

Applying both transformations gives us a new formula $f(\phi)$ that is polynomial in size with respect to $|\phi|$, and the transformations can be done in polynomial time. The result $f(\phi)$ is in 3CNF, and it is satisfiable exactly when ϕ is. Thus, we have $\phi \in$ SAT $\iff f(\phi) \in$ 3SAT, and f is computable in polynomial time, so we have SAT $\le_p$ 3SAT. Since SAT is NP-hard, so is 3SAT.

Having shown that 3SAT $\in \mathsf{NP}$ and 3SAT is NP-hard, we conclude that 3SAT is NP-complete.

Converting a formula to CNF

If a formula ϕ is not in conjunctive normal form, then its structure is as follows:

$$\begin{aligned}\phi = &(X_{1,1} \wedge \cdots \wedge X_{1,m_1}) \vee \\ &(X_{2,1} \wedge \cdots \wedge X_{2,m_2}) \vee \\ &\cdots \vee \\ &(X_{n,1} \wedge \cdots \wedge X_{n,m_n})\end{aligned}$$

Here, $X_{i,k}$ is an arbitrary subformula. We can transform ϕ to the following:

$$\begin{aligned}\phi' = &(z_1 \vee z_2 \vee \cdots \vee z_n) \wedge \\ &(\neg z_1 \vee X_{1,1}) \wedge \cdots \wedge (\neg z_1 \vee X_{1,m_1}) \wedge \\ &(\neg z_2 \vee X_{2,1}) \wedge \cdots \wedge (\neg z_2 \vee X_{2,m_2}) \wedge \\ &\cdots \wedge \\ &(\neg z_n \vee X_{n,1}) \wedge \cdots \wedge (\neg z_n \vee X_{n,m_n})\end{aligned}$$

This transformation preserves satisfiability. To see why this is so, we consider two cases for the original formula ϕ:

- **Case 1:** There is an assignment α such that all of $X_{i,1}, \dots, X_{i,m_i}$ are true in $\phi(\alpha)$ for some i. Then the original formula $\phi(\alpha)$ is satisfied by the ith term in its disjunction. The new formula ϕ' is satisfied by the same assignment for the original variables, but with the addition of $z_i = 1$ and $z_{j \neq i} = 0$. Specifically, $z_i = 1$ causes the first clause to be satisfied, $z_{j \neq i} = 0$ result in any clause containing an $X_{j \neq i,k}$ to be satisfied, and $X_{i,k} = 1$ result in all clauses containing $X_{i,k}$ to be satisfied. Thus, all clauses in ϕ' are satisfied.
- **Case 2:** For all assignments α, at least one of $X_{i,1}, \dots, X_{i,m_i}$ is false in $\phi(\alpha)$ for all i. Then all terms of the disjunction in the original formula are false, so $\phi(\alpha)$ as a whole is unsatisfied. Consider an assignment α' to the new formula ϕ' such that α and α' assign the same values to the original variables in ϕ. Let X_{i,k_i} be a false term in $\phi(\alpha)$ for each i. Then the clause $(\neg z_i \vee X_{i,k_i})$ in $\phi'(\alpha')$ can only be satisfied by setting $z_i = 0$. However, if all $z_i = 0$, then the first clause in $\phi'(\alpha')$ is unsatisfied, so the entire formula is unsatisfied.

We see that if ϕ is satisfiable by an assignment α, then the new formula ϕ' is satisfiable by a modified assignment α' as described above. However, if ϕ is unsatisfiable, then so is ϕ'. Thus, this transformation preserves the satisfiability of the original formula.

The transformation only increases the size of the formula by a linear amount – the new formula ϕ' contains an additional literal $\neg z_i$ for each subformula $X_{i,k}$, plus another n literals for its first clause, where n is the number of terms in the outer disjunction of ϕ. The number of literals in ϕ' is at most triple the number in ϕ.

We can repeat this transformation on the subformulas $X_{i,k}$ to convert them into CNF formulas $X'_{i,k}$, followed by an application of the distributive law

$$a \vee (b_1 \wedge \cdots \wedge b_n) = (a \vee b_1) \wedge \cdots \wedge (a \vee b_n)$$

to turn $(\neg z_i \vee X'_{i,k})$ into CNF. The latter doubles the size of the formula. Overall, we do at most a linear number of transformations, each of which increases the size of the formula by a linear amount, so the total increase is at most quadratic.

Modifying the proof of Cook-Levin to show that 3SAT is NP-hard

We can modify our proof of the Cook-Levin theorem to directly show that 3SAT is NP-hard. In particular, we demonstrate how to construct $\phi_{V,x}$ so that it is in conjunctive normal form. We can then apply the transformation discussed above to turn a CNF formula into a 3CNF one.

We first observe that a CNF formula can be constructed recursively from smaller subformulas:

- A formula $\phi = x$ with a single variable is in CNF – it has the single clause (x).
- A formula $\phi = \phi_1 \wedge \phi_2$ is in CNF if ϕ_1 and ϕ_2 are in CNF – it consists of the combination of the clauses in ϕ_1 and ϕ_2.
- A formula $\phi = \phi_1 \vee \phi_2$ is only in CNF if ϕ_1 and ϕ_2 each have a single clause – ϕ is then just a single clause that combines the literals in ϕ_1 and ϕ_2.

Examining each piece of $\phi_{V,x}$, we see:

- The cell-consistency subformula

  $$\phi_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} [\bigvee_{s \in S} t_{i,j,s} \wedge \bigwedge_{\text{distinct } s,s' \in S} (\neg t_{i,j,s} \vee \neg t_{i,j,s'})]$$

  is in CNF. The inner conjunction is a conjunction of disjunctions, so it is clearly in CNF. The inner disjunction is a single clause, so it is in CNF. We combine the two with a conjunction, and the result is also in CNF since the two subformulas are in CNF. Finally, the outer conjunction also produces a CNF formula since the individual pieces are in CNF.

- The acceptance subformula

$$\phi_{\text{accept}} = \bigvee_{1\le i,j\le n^k} t_{i,j,q_{\text{acc}}}$$

  is a single clause in CNF.

- The starting-configuration subformula

$$\phi_{\text{start},x} = t_{1,1,\#} \wedge t_{1,2,q_{\text{start}}} \wedge \phi_{\text{input}} \wedge t_{1,n+3,\$} \wedge \phi_{\text{cert}} \wedge t_{1,n^k,\#}$$

  is in CNF – it consists of n^k clauses, each with either a single literal (e.g., $(t_{1,1,\#})$) or $|\Gamma|$ literals (for each clause in ϕ_{cert}).

- The transition formula for a window with upper-left corner at i, j

$$\phi_{i,j} = \bigvee_{w\in W} \phi_{i,j,w}$$

  is **not** in CNF. The individual $\phi_{i,j,w}$ consist of a sequence of conjunctions, so $\phi_{i,j}$ is a disjunction of conjunctions, rather than a conjunction of disjunctions required for CNF. However, we can convert $\phi_{i,j}$ to CNF by applying the distributive law for $\vee$ and $\wedge$. In particular, we have

$$(\bigwedge_i a_i) \vee (\bigwedge_j b_j) = \bigwedge_{i,j}(a_i \vee b_j)\ .$$

  We can see that this law holds if we consider two cases:

  - If all a_i are true, then the left-hand side is true. The right-hand side is also true – each clause $(a_i \vee b_j)$ contains an a_i, so each clause is satisfied, which makes the formula as a whole true.

    This same reasoning applies for the case when all b_j are true.

  - If at least one a_i is false and at least one b_j is false, then the left-hand side is false. The right-hand side is also false, since there is a clause $(a_i \vee b_j)$ where both a_i and b_j are false.

  How does the size of the right-hand formula compare to the left-hand one? If the two conjunctions on the left contain m and n literals, respectively, then the left-hand side has $m + n$ total literals. The right-hand side has $m \cdot n$ clauses, each with two literals, for a total size of $2mn$. When $m = n$, we go from $2m$ literals to a size of $2m^2$.

  Suppose we have c conjunctions on the left-hand side, each with m literals. Then repeated application of the distributive law takes us from a formula with cm literals to one with cm^c. For $\phi_{i,j}$, we go from a size of $6|W|$ in its original form to a size of $|W| \cdot 6^{|W|}$ in CNF. This is exponential in the number of distinct valid windows $|W|$, but this number is a characteristic of the verifier V, not the input x. So even though it is a larger constant than before, it is still a constant.

  Once we have $\phi_{i,j}$ in CNF, then

$$\phi_{\text{move},V} = \bigwedge_{\substack{1\le i\le n^k-1\\ 1\le j\le n^k-2}} \phi_{i,j}$$

  is also in CNF. And while its size is larger than previously, it is only by a constant factor, and its total size is still $O(n^{2k})$ literals.

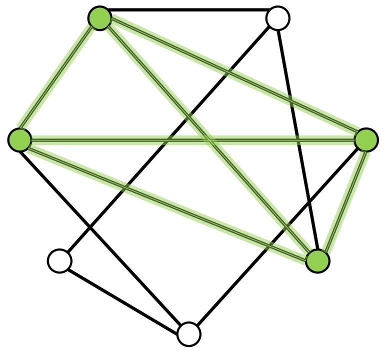
Since $\phi_{V,x}$ is just a conjunction of the individual pieces above, it is in CNF if those pieces are each in CNF. We can then apply the CNF-to-3CNF transformation to obtain a formula in 3CNF. Thus, if we can decide 3SAT, we can decide an arbitrary language in NP, and 3SAT is NP-hard.

17.2 Clique

NP-completeness is not exclusive to Boolean satisfiability problems. Rather, there is a wide range of problems across all fields that are NP-complete. As an example, we take a look at the clique problem.

Given an undirected graph $G = (V, E)$, a *clique* is a subset of the vertices $C \subseteq V$ such that there is an edge in G between every pair of vertices in the clique. Equivalently, the subgraph *induced* by the clique C, meaning the vertices in C and all edges between them, is a complete graph.

The following is an example of a clique of size four, with the highlighted vertices in the clique:



We are often interested in finding a clique of maximum size in a graph, called a *maximum clique* for short. For instance, if a graph represents a group of people and their individual friend relationships, we might want to find the largest set of mutual friends (hence the name "clique"). However, to obtain a decision problem, we introduce a limited budget as we did for the *traveling salesperson problem* (page 139):

$$\text{CLIQUE} = \{(G, k) : G \text{ is an undirected graph that has a clique of size } k\}$$

Here, we have defined CLIQUE with an exact budget k rather than a lower bound – if a graph $G = (V, E)$ has a clique C of size m, then it has one for any size $0 \le k \le m$ by removing arbitrary vertices from C. In other words, a subset of a clique is always itself a clique.

To show that CLIQUE is NP-complete, we start by demonstrating that it is in NP. A verifier for CLIQUE takes a graph G and a budget k, and a set of vertices as the certificate, and check that the vertices in the set indeed comprise a clique of size k. We define the verifier as follows. The input instance is a graph and a non-negative integer k. The certificate is a set of exactly k vertices in the graph. The verifier simply checks that these vertices form a clique, i.e., that there is an edge between each pair of (distinct) vertices in the certificate.

```
Input: instance: an undirected graph and non-negative integer $k$; certificate: a set of $k$ vertices in the graph
Output: whether the vertices form a clique in the graph
  function VerifyCLIQUE($(G = (V, E), k), c = \{v_1, \dots, v_k\}$)
      for all distinct $i, j \in \{1, \dots, k\}$ do
          if $(v_i, v_j) \notin E$ then reject
      accept
```

We first analyze the runtime of the verifier. The first step can be done efficiently – checking whether c is a subset of V can be done naïvely in quadratic time via nested loops. (Furthermore, by checking that $k \le |V|$ and that $|c| = k$, we limit the loop bounds to the size of the input graph.) The loop in the second step does a quadratic number of iterations, each of which can be done efficiently, so the total time is still polynomial.

We now analyze correctness. If $(G, k) \in$ CLIQUE, then there is some clique $C = \{v_1, \dots, v_k\} \subseteq V$ of size $0 \le k \le |V|$. If we pass the set C as the certificate to the verifier, we have $|C| = k$ and $C \subseteq V$, so the certificate is not rejected in the first step. Then since C is a clique, there is an edge between every pair of vertices in C, so the loop in the second step does not reject. Thus, the verifier accepts $(G, k), C$.

On the other hand, if $(G = (V, E), k) \notin$ CLIQUE, then there is no set of k vertices C such that C comprises a clique of G. If k is not in the range $[0, |V|]$, the verifier rejects immediately. In addition, for any certificate that is not a subset of V whose size is k, the verifier rejects in the first step. If the verifier gets to the second step, then the certificate c must be a set of vertices $c = \{v_1, \dots, v_k\} \subseteq V$. By assumption, G does not have a clique of size k, so any set of k vertices from G must have a missing edge between some pair of vertices within that set. Since the verifier checks for an edge between all pairs of vertices from the given set, it will find such a missing edge and reject. Thus, when $(G, k) \notin$ CLIQUE, the verifier rejects all certificates.

Since the verifier is correct and efficient, we have demonstrated that CLIQUE is efficiently verifiable and thus in NP.

We now show that CLIQUE is NP-hard with a polytime reduction from the known NP-hard 3SAT, i.e. 3SAT $\le_p$ CLIQUE. To do so, we must define a polynomial-time computable function f that converts a 3CNF formula ϕ into an instance (G, k) of the clique problem. More specifically, we require:

- $\phi \in \text{3SAT} \implies f(\phi) \in \text{CLIQUE}$
- $\phi \notin \text{3SAT} \implies f(\phi) \notin \text{CLIQUE}$

Before considering how we can map formulas to graphs, let's review the definition of 3SAT. It is the set of satisfiable 3CNF formulas, where a formula has m clauses of three literals each:

$$\phi = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge (\ell_4 \vee \ell_5 \vee \ell_6) \wedge \dots \wedge (\ell_{3m-2} \vee \ell_{3m-1} \vee \ell_{3m})$$

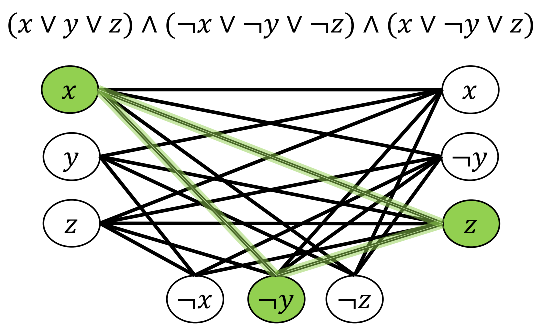
Each literal is either a variable itself (e.g., v) or its negation (e.g., $\neg v$).

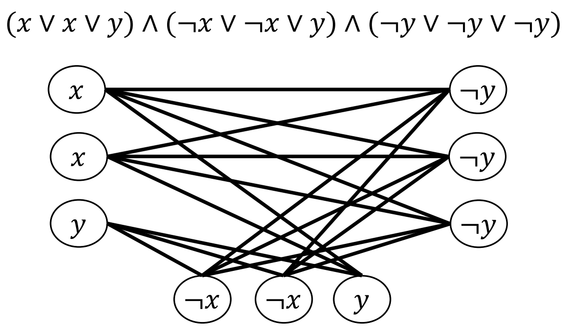
A satisfying assignment simultaneously satisfies each clause, so that each clause has at least one literal that is true. For a formula with m clauses to be satisfiable, there must be some set of m literals, one from each clause, that can simultaneously be true. If no such set exists, the formula is unsatisfiable.

We can now relate 3CNF-satisfiability to the clique problem. Given a formula ϕ of m clauses, we can construct a graph G such that:

- Each literal $\ell_i \in phi$ has a corresponding vertex v_i in G.
- A pair of vertices v_i and v_j have an edge in G if the corresponding literals ℓ_i and ℓ_j are in different clauses of ϕ, and the two literals can be simultaneously true. This is exactly when the two literals are not negations of each other (e.g., one is x and the other is $\neg x$) – either they are the same (e.g., they are both x or are both $\neg y$), or they refer to different variables (e.g., one is x and the other is $\neg y$).

The following illustrates the graph constructed for two different formulas with three clauses each, with one formula satisfiable and the other unsatisfiable:


$(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$
x
y
z
$\neg x$
$\neg y$
$\neg z$
x
$\neg y$
z


$(x \vee x \vee y) \wedge (\neg x \vee \neg x \vee y) \wedge (\neg y \vee \neg y \vee \neg y)$
x
x
y
$\neg x$
$\neg x$
y
$\neg y$
$\neg y$
$\neg y$

Observe that the satisfiable formula on the left does have a clique of size three (one such clique is highlighted), while the unsatisfiable formula on the right does not have a clique of size three.

The formal conversion algorithm that computes the mapping reduction is as follows:

Input: a 3CNF formula ϕ
Output: a graph G and integer m, such that $\phi \in \text{3SAT} \iff (G, m) \in \text{CLIQUE}$

```
function 3SATtoClique($\phi$)
    $m$ = number of clauses in $\phi$
    $G$ = an empty graph
    for all literals $\ell_i$ in $\phi$ do
        add a vertex $v_i$ to $G$
    for all pairs of literals $\ell_i, \ell_j$ in different clauses of $\phi$ do
        if $\ell_i \neq \neg\ell_j$ then
            add an edge $(v_i, v_j)$ to $G$
    return $(G, m)$
```

We first analyze the efficiency of this algorithm. The first loop takes linear time with respect to the size of the formula (i.e., the number of literals in the formula), and the second loop takes quadratic time. Thus, the entire conversion takes polynomial time with respect to the input size.

We now analyze correctness. First, we show that $\phi \in$ 3SAT $\implies f(\phi) \in$ CLIQUE. Since $\phi \in$ 3SAT, it is satisfiable, which means there is some satisfying assignment α such that each clause in ϕ has at least one literal that is true. Let $S = \{s_1, s_2, \dots, s_m\}$ refer to a set comprised of one true literal from each clause. Then:

- Each pair of literals $s_i, s_j \in S$ can be true simultaneously, since they are all true under the assignment α.
- For each $s_i, s_j \in S$, there is an edge in G between their corresponding vertices, since the literals are from different clauses and can be simultaneously true.
- Since there is an edge in G between the vertices corresponding to all pairs $s_i, s_j \in S$, those vertices constitute a clique in G. Since $|S| = m$, G has a clique of size m.

We now show the converse, that $f(\phi) \in$ CLIQUE $\implies \phi \in$ 3SAT. It is important to note that we are **not** inverting the function f here. Rather, this says that if the graph/budget pair produced **as the output of** f are in CLIQUE, then the input formula must be in 3SAT. Since $f(\phi) = (G, m) \in$ CLIQUE, the output graph G has a clique of size m. Denote the set of vertices comprising the clique as $C = \{c_1, \dots, c_m\}$. Then:

- Since C is a clique, there is an edge between any pair of vertices c_i, c_j in C.
- The algorithm f only places an edge between vertices if the corresponding literals are from different clauses. Thus, the m vertices in C must come from the m different clauses of the input formula ϕ.
- The algorithm f only places an edge between vertices if their corresponding literals can be true simultaneously. Since there is an edge between every pair of vertices in C, all of their corresponding literals can be true at the same time.
- We can define an assignment α of ϕ's variables by setting them such that each literal corresponding to the vertices in C becomes true. Since all the literals can be true simultaneously, there is no conflict when defining this assignment. (Any variables that remain unset after this process can be set arbitrarily.)
- Then α is a satisfying assignment – each literal corresponding to the vertices in C is from a different clause, and they are all true, so each of the m clauses has a true literal and is therefore satisfied. Thus, ϕ is satisfiable.

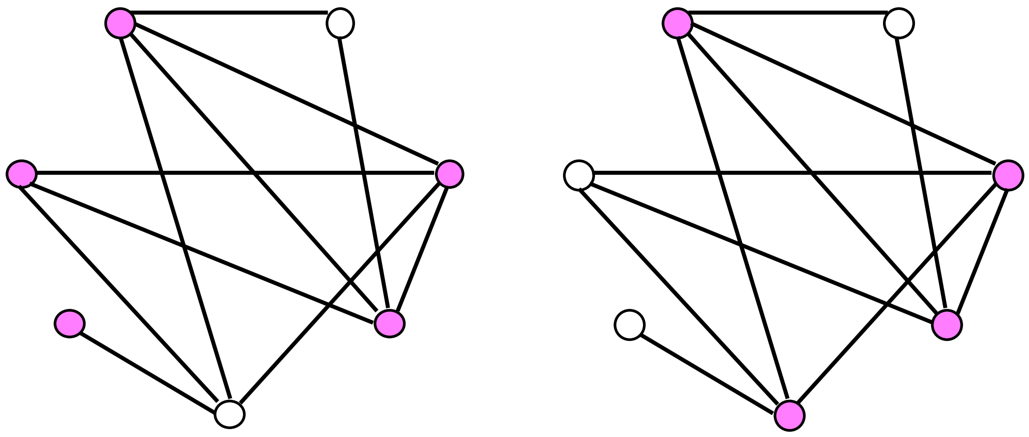
Since $\phi \in$ 3SAT $\iff f(\phi) \in$ CLIQUE and f can be computed efficiently, we have demonstrated that 3SAT $\leq_p$ CLIQUE. Since 3SAT is NP-hard, we conclude that CLIQUE is NP-hard, and since CLIQUE $\in$ NP, it is also NP-complete.

17.3 Vertex Cover

Given an undirected graph $G = (V, E)$, a *vertex cover* is a subset of the vertices $C \subseteq V$ such that every edge in E is covered by a vertex in C. An edge $e = (u, v)$ is *covered* by C if at least one of the endpoints u, v is in C. In other words, C is a vertex cover for $G = (V, E)$ if $C \subseteq V$ and

$$\forall e = (u, v) \in E.\, u \in C \lor v \in C$$

The following are two vertex covers for the same graph:



The cover on the left has a size of five vertices, while the cover on the right has a size of four.

As a motivating example of the vertex-cover problem, consider a museum structured as a set of interconnected hallways, with exhibits on the walls. The museum needs to hire guards to ensure the security of the exhibits, but guards are expensive, so it seeks to minimize the number of guards hired while still ensuring that each hallway is covered by a guard. The museum layout can be represented as a graph with the hallways as edges and the vertices as hallway endpoints. Then the museum just needs to figure out what the minimum vertex cover is to determine how many guards to hire (or security cameras/motion detectors to install) and where to place them.

As we did for the *traveling salesperson problem* (page 139), we define a limited-budget, decision version of vertex cover:

$$\text{VERTEX-COVER} = \{(G, k) : G \text{ is an undirected graph that has a vertex cover of size } k\}$$

Here, we have defined VERTEX-COVER with an exact budget k rather than an upper bound – observe that if a graph $G = (V, E)$ has a vertex cover C of size m, then it has one for any size $m \leq k \leq |V|$ by adding arbitrary vertices from $V \setminus C$ into C. Put in museum terms, if the museum's budget allows, it can hire more guards than are strictly necessary to cover all hallways.

We now demonstrate that VERTEX-COVER is NP-hard, leaving the proof that VERTEX-COVER $\in$ NP as an exercise. Combining the two, we will be able to conclude that VERTEX-COVER is NP-complete.

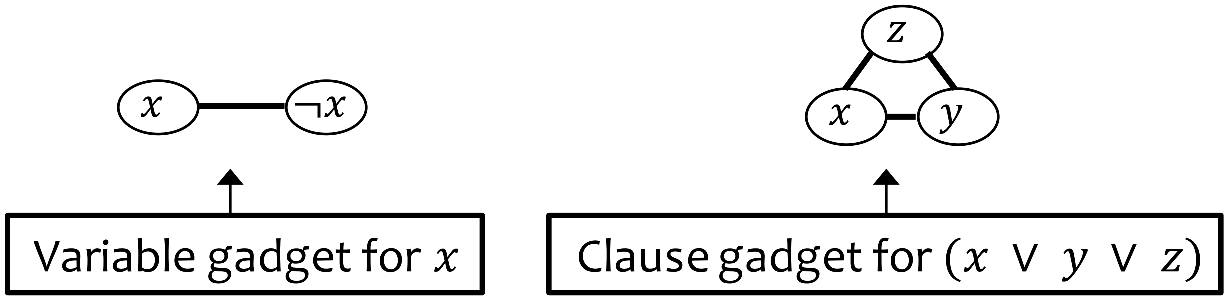
Exercise 161 Show that VERTEX-COVER $\in$ NP.

We show that VERTEX-COVER is NP-hard via the reduction 3SAT $\leq_p$ VERTEX-COVER. To do so, we must define a polynomial-time computable function f that converts a 3CNF formula ϕ into an instance (G, k) of the vertex-cover problem. More specifically, we require:

- $\phi \in$ 3SAT $\implies f(\phi) \in$ VERTEX-COVER
- $\phi \notin$ 3SAT $\implies f(\phi) \notin$ VERTEX-COVER

Given a 3CNF formula ϕ with n variables and m clauses, we construct a graph G corresponding to that formula. The graph is comprised of *gadgets* that are subgraphs, representing individual variables and clauses. We will then connect the gadgets together such that the resulting graph G has a vertex cover of size $n + 2m$ exactly when ϕ is satisfiable.

The following are our gadgets for variables and clauses:

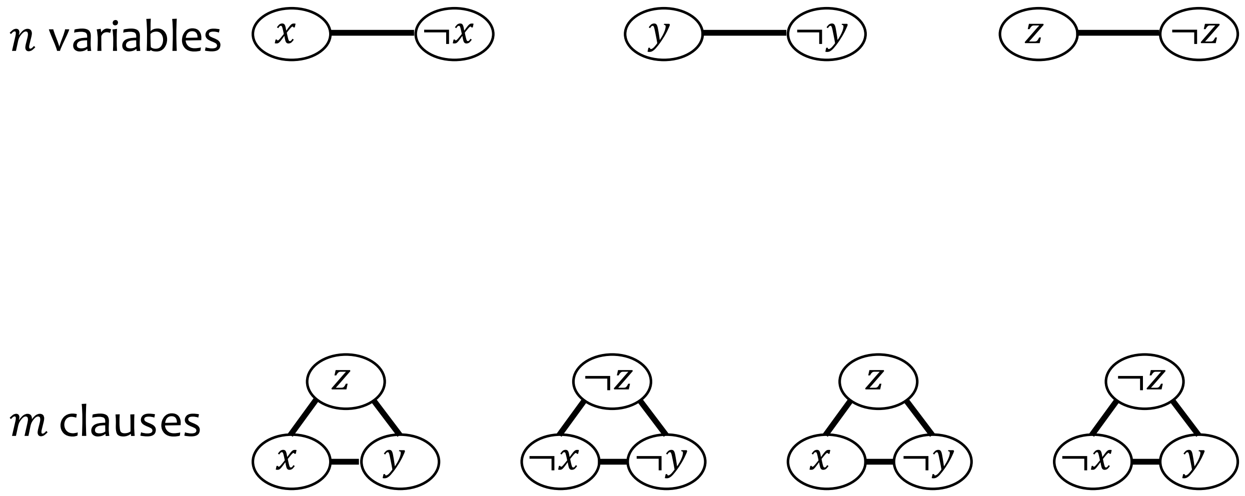


A gadget for variable x is a "barbell," consisting of two vertices labeled by x and $\neg x$ connected by an edge. A gadget for a clause $x \lor y \lor z$ is a triangle, consisting of a vertex for each literal, with the three vertices all connected together.

As a concrete example, consider the formula

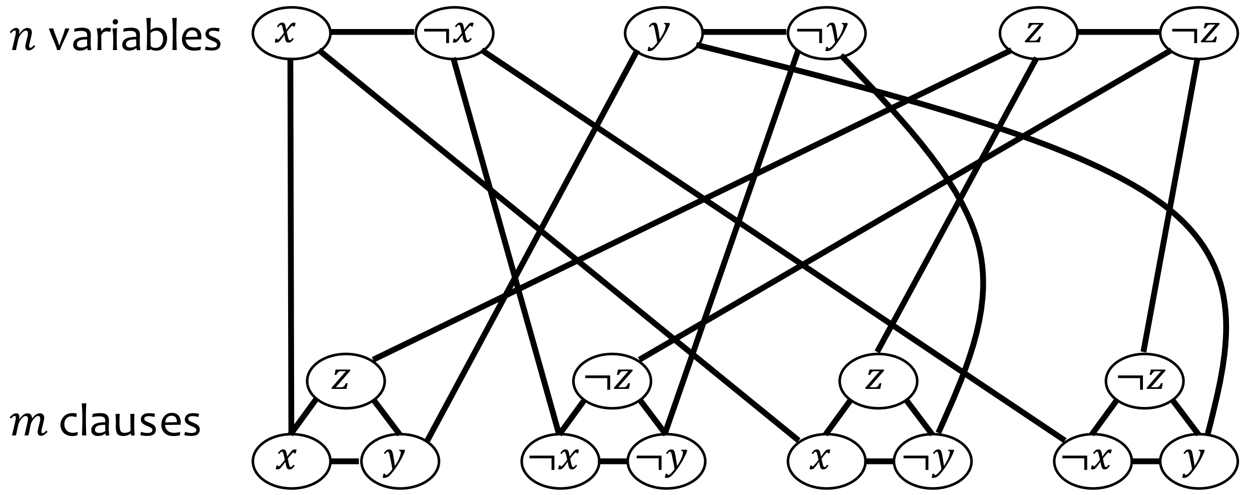
$$\phi = (x \lor y \lor z) \land (\neg x \lor \neg y \lor \neg z) \land (x \lor \neg y \lor z) \land (\neg x \lor y \lor \neg z)$$

We start building the corresponding graph G by incorporating gadgets for each clause and variable:



Here, we have placed the variable gadgets at the top of the figure and the clause gadgets at the bottom.

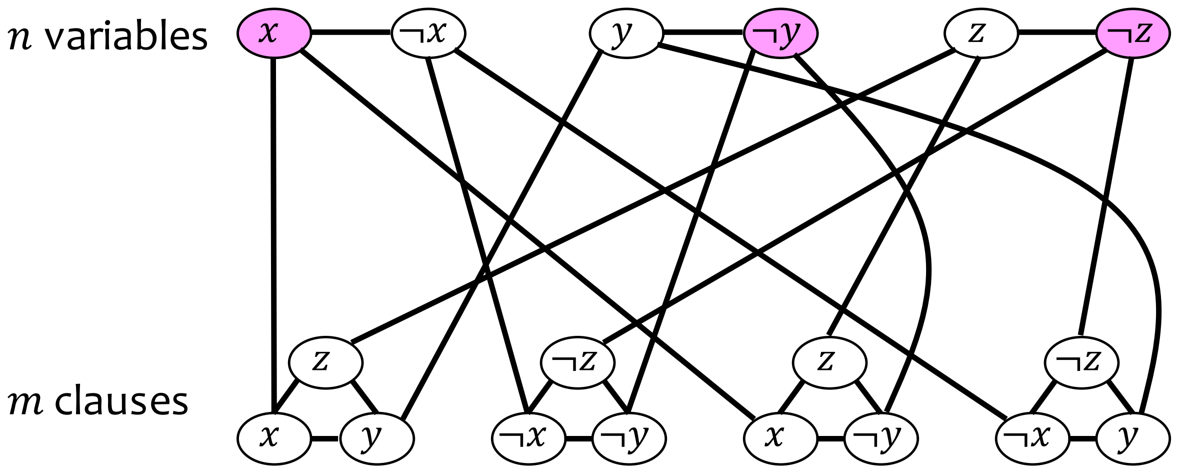
The next step in the construction is to connect each vertex in a variable gadget to the vertices in the clause gadgets that have the same literal as the label. For instance, we connect x in the respective variable gadget to each vertex labeled by x in the clause gadgets, $\neg x$ in the variable gadget to each vertex labeled by $\neg x$ in the clause gadgets, and so on. The result is as follows:



This completes the construction of G, and we have $f(\phi) = (G, n + 2m)$. There are $3m + 2n$ vertices in G, n edges within the variable gadgets, $3m$ edges within the clause gadgets, and $3m$ edges that cross between the clause and variable gadgets, for a total of $O(m + n)$ edges. Thus, G has size $O(n + m)$, and it can be constructed efficiently.

We proceed to show that if $\phi \in$ 3SAT, then $f(\phi) = (G, n + 2m) \in$ VERTEX-COVER. Since $\phi \in$ 3SAT, there is some satisfying assignment α such that $\phi(\alpha)$ is true. Then C is a vertex cover for G, where:

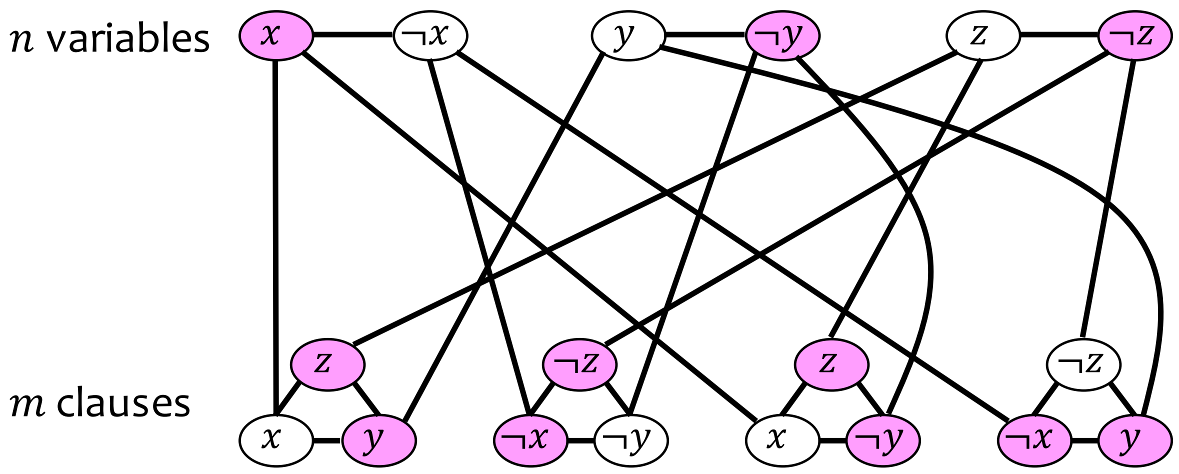
- For the variable gadget corresponding to each variable x, x is in C if $\alpha(x) = 1$, otherwise $\neg x$ is in C (i.e. if $\alpha(x) = 0$).



  Observe that every edge internal to a variable gadget is covered. Furthermore, since α is a satisfying assignment, each clause has at least one literal that is true. Then the edge connecting the vertex in the corresponding clause gadget to the same literal in a variable gadget is already covered – the vertex corresponding to that literal in the variable gadget is in C.

  Thus, every clause gadget has at least one out of its three "crossing" edges covered by a vertex in a variable gadget, and so far, we have n vertices in C.

- For each uncovered crossing edge (u, v), where u is in a variable gadget and v is in a clause gadget, we have $v \in C$. There are at most two such crossing edges per clause. If a clause has only one uncovered crossing edge, then pick another arbitrary vertex in the clause to be in C. Similarly, if a clause has no uncovered crossing edges, then pick two of its vertices arbitrarily to be in C.



Now all crossing edges are covered by a vertex in a variable gadget, or one in a clause gadget, or both. Furthermore, since we pick exactly two vertices out of each clause gadget, the edges internal to each clause gadget are also covered. Thus, all edges are covered, and C is a vertex cover. This second step adds two vertices per clause, or $2m$ vertices. Combined with the previous vertices, C has $n + 2m$ vertices, and we have shown that G has a vertex cover of size $n + 2m$.

We now show that $\phi \notin$ 3SAT $\implies (G, n + 2m) \notin$ VERTEX-COVER. More precisely, we demonstrate the contrapositive $(G, n + 2m) \in$ VERTEX-COVER $\implies \phi \in$ 3SAT. In other words, if G has a vertex cover of size $n + 2m$, then ϕ has a satisfying assignment.

A vertex cover of G of size $n + 2m$ must include the following:

- exactly one vertex from each variable gadget to cover the edge internal to the variable gadget
- exactly two vertices from each clause gadget to cover the edges internal to the clause gadget

This brings us to our total of $n + 2m$ vertices in the cover. While all internal edges are covered, we must also consider the edges that cross between clause and variable gadgets. With the two vertices from each clause gadget, only two of the three crossing edges are covered by those vertices. The remaining crossing edge must be covered by a vertex from a clause gadget. Thus, if G has a vertex cover C of size $n + 2m$, each clause k has a crossing edge (u_l, v_k) where:

- u_l is a variable vertex with label l for some literal l
- v_k is a vertex in the clause gadget with the same label l
- $u_l \in C$, so that the crossing edge is covered

Then the assignment α such that $\alpha(l) = 1$ if $u_l \in C$ is a satisfying assignment – for each clause k, we have the true literal l corresponding to the vertex v_k. Since all clauses are satisfied, ϕ as a whole is satisfied.

This completes our proof that 3SAT $\leq_p$ VERTEX-COVER. We conclude from the facts that 3SAT is NP-hard and VERTEX-COVER $\in$ NP that VERTEX-COVER is NP-complete.

Exercise 162 a) Define the language

$$\text{INDEPENDENT-SET} = \{(G, k) : G \text{ is an undirected graph with an independent set of size } k\}$$

An *independent set* of a graph $G = (V, E)$ is a subset of the vertices $I \subseteq V$ such that no two vertices in I share an edge. Show that VERTEX-COVER $\leq_p$ INDEPENDENT-SET.

Hint: If G has a vertex cover of size m, what can be said about the vertices that are not in the cover?

b) Recall the language

$$\text{CLIQUE} = \{(G, k) : G \text{ is an undirected graph that has a clique of size } k\}$$

Show that INDEPENDENT-SET $\leq_p$ CLIQUE.

17.4 Set Cover

Consider another problem, that of hiring workers for a project. To successfully complete the project, we need workers with some combined set of skills S. For instance, we might need an architect, a general contractor, an engineer, an electrician, a plumber, and so on for a construction project. We have a candidate pool of n workers, where worker i has a set of skills S_i. As an example, there may be a single worker who is proficient in plumbing and electrical work. Our goal is to hire a minimum-size team of workers to *cover* all the required skills.

We formalize this problem as follows. We are given a set of elements S, representing the required skills. We are also given n subsets $S_i \subseteq S$, representing the set of skills that each worker i has. We want to find the smallest collection of S_i's that cover the set S. In other words, we wish to find the smallest set of indices C such that

$$S = \bigcup_{i \in C} S_i$$

As a concrete example, let S and $S_1, \dots, S_6$ be as follows:

$$\begin{aligned} S &= \{1, 2, 3, 4, 5, 6, 7\} \\ S_1 &= \{1, 2, 3\} \\ S_2 &= \{3, 4, 6, 7\} \\ S_3 &= \{1, 4, 7\} \\ S_4 &= \{1, 2, 6\} \\ S_5 &= \{3, 5, 7\} \\ S_6 &= \{4, 5\} \end{aligned}$$

There are no set covers of size two, but there are several with size three. One example is $C = 1, 2, 6$, which gives us

$$\begin{aligned} \bigcup_{i \in C} S_i &= S_1 \cup S_2 \cup S_6 \\ &= \{1, 2, 3\} \cup \{3, 4, 6, 7\} \cup \{4, 5\} \\ &= \{1, 2, 3, 4, 5, 6, 7\} \\ &= S \end{aligned}$$

We proceed to define a language corresponding to the set-cover problem. As with vertex cover, we include a budget k for the size of the cover C.

$$\begin{aligned} \text{SET-COVER} = \{(S, S_1, S_2, \dots, S_n, k) : \; & S_i \subseteq S \text{ for each } i, \text{ and there is a collection} \\ & \text{of } k \text{ sets } S_i \text{ that cover } S\} \end{aligned}$$

It is clear that SET-COVER $\in$ NP – we can use as a certificate a set of indices C, and the verifier need only check that C is a size-k subset of $[1, n]$, and that $S = \cup_{i \in C} S_i$. This can be done efficiently.

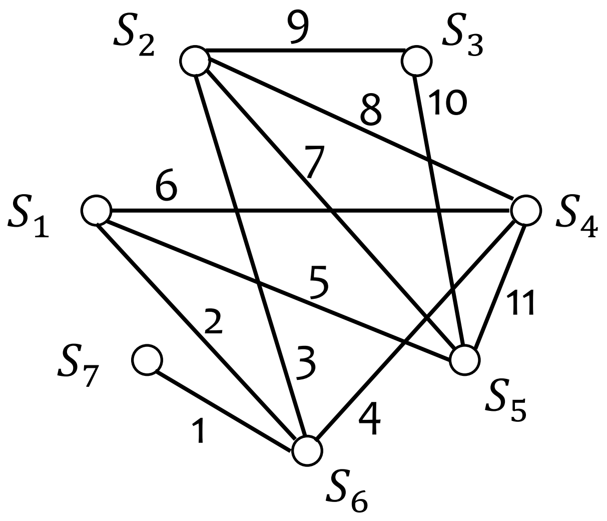
We now perform the reduction VERTEX-COVER $\leq_p$ SET-COVER to demonstrate that SET-COVER is NP-hard. We need to convert an instance $(G = (V, E), k)$ to an instance $(S, S_1, \dots, S_n, k')$ such that G has a vertex cover of size k exactly when S has a set cover of size k'. Observe that an element of a vertex cover is a vertex v, and it covers some subset of the edges in E. Similarly, a set cover consists of sets S_i, each of which cover some subset of the items in S. Thus, an edge corresponds to a skill, while a vertex corresponds to a worker.

Given a graph $G = (V, E)$ and a budget k, we define the function f such it translates the set of edges E to the set of skills S, and each vertex $v_i \in V$ to a set S_i consisting of the edges incident to v_i. Formally, we have $f(G, k) = (S, S_1, \dots, S_{|V|}, k)$, where $S = E$ and:

$$S_i = \{e \in E : e \text{ is incident to } v_i\text{, i.e. } e = (u, v_i) \text{ for some } u \in V\}$$

This transformation can be done efficiently, as it is a direct translation of the graph G.

As an example, suppose that the graph G is as follows:



This graph has a cover of size four consisting of the vertices labeled S_2, S_4, S_5, S_6. The transformation $f(G, k)$ produces the following sets:

$$\begin{aligned}
S &= \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}\\
S_1 &= \{2, 5, 6\}\\
S_2 &= \{3, 7, 8, 9\}\\
S_3 &= \{9, 10\}\\
S_4 &= \{4, 6, 8, 11\}\\
S_5 &= \{5, 7, 10, 11\}\\
S_6 &= \{1, 2, 3, 4\}\\
S_7 &= \{1\}
\end{aligned}$$

Then $C = \{2, 4, 5, 6\}$ is a set cover of size four.

We now demonstrate that in the general case, $(G = (V, E), k) \in$ VERTEX-COVER exactly when $f(G, k) = (S, S_1, \dots, S_{|V|}, k) \in$ SET-COVER.

- If $(G, k) \in$ VERTEX-COVER, then G has a vertex cover $C \subseteq V$ of size k that covers all the edges in E. This means that

  $$E = \bigcup_{v_i \in C} \{e \in E : e \text{ is incident to } v_i\}$$

  Let $C' = \{i : v_i \in C\}$. Then we have

  $$\begin{aligned}
  \bigcup_{i \in C'} S_i &= \bigcup_{i \in C'} \{e \in E : e \text{ is incident to } v_i\}\\
  &= E = S
  \end{aligned}$$

  Thus, C' covers S, and S has a set cover of size k.

- If $(G, k) \notin$ VERTEX-COVER, then G does not have a set cover of size k. Either $k > |V|$, or for any subset of vertices $C \subseteq V$, of size k, we have

  $$E \neq \bigcup_{v_i \in C} \{e \in E : e \text{ is incident to } v_i\}$$

  Then for any subset $C' \subseteq \{S_1, \dots, S_{|V|}\}$ of size k, we also have

  $$\begin{aligned}\bigcup_{i \in C'} S_i &= \bigcup_{i \in C'} \{e \in E : e \text{ is incident to } v_i\} \\ &\neq E = S\end{aligned}$$

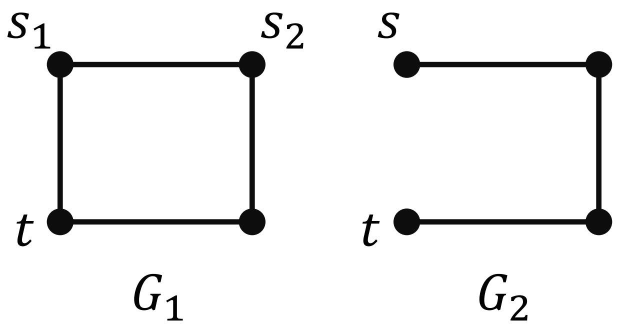
  Thus S does not have a set cover of size k.

This completes our reduction VERTEX-COVER $\leq_p$ SET-COVER.

We conclude our discussion of set cover by observing that a vertex cover is actually a special case of a set cover. When we transform an instance of the vertex-cover problem to one of set cover, each edge in a graph is interpreted as a skill and each vertex as worker, which means that each skill is shared by exactly two workers. This is more restrictive then the general case for set cover, where any number of workers (including none) may have a particular skill. The reduction VERTEX-COVER $\leq_p$ SET-COVER just exemplifies that if we can solve the general case of a problem efficiently, we can also solve special cases efficiently. Conversely, if a special case is "hard," then so is the general case.[60]

17.5 Hamiltonian Cycle

Given a graph G, a *Hamiltonian path* from s to t is a path that starts at s, ends at t, and visits each vertex exactly once. A *Hamiltonian cycle* is a path that starts and ends at the same vertex and visits all other vertices exactly once. For example, G_1 below has a Hamiltonian cycle as well as a Hamiltonian path between s_1 and t, but it does not have a Hamiltonian path between s_2 and t. On the right, G_2 has a Hamiltonian path between s and t, but it does not have a Hamiltonian cycle.



We define the languages corresponding to the existence of a Hamiltonian path or cycle as:

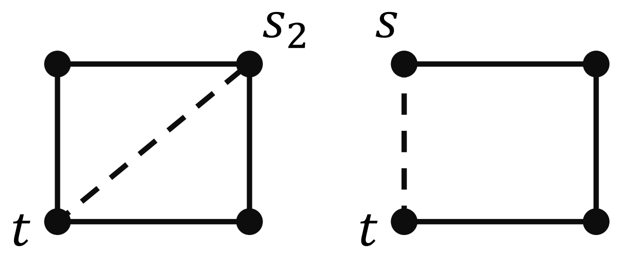
$$\begin{aligned}&\text{HAMPATH} = \{(G, s, t) : G \text{ is an undirected graph with a Hamiltonian path between } s \text{ and } t\} \\ &\text{HAMCYCLE} = \{G : G \text{ is an undirected graph with a Hamiltonian cycle}\}\end{aligned}$$

Both languages are in NP – we take the actual Hamiltonian path or cycle as a certificate, and a verifier need only follow the path to check that it is valid as in the verifier for *TSP* (page 139). We claim without proof that HAMPATH is NP-hard – it is possible to reduce 3SAT to HAMPATH, but the reduction is quite complicated, so we will not consider it here. Instead, we perform the reduction HAMPATH $\leq_p$ HAMCYCLE to show that HAMCYCLE is also NP-hard. We

[60] Since both VERTEX-COVER and SET-COVER are NP-complete, they are actually equally hard, so an instance of either can be converted to an instance of the other. However, the conversion of an instance of set cover to vertex cover is not as straightforward as that of vertex cover to set cover.

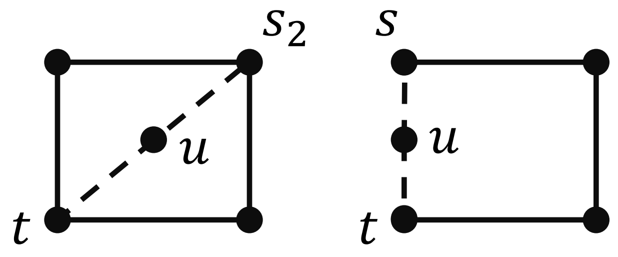
need to demonstrate an efficient function f such that $(G, s, t) \in$ HAMPATH if and only if $f(G, s, t)$ has a Hamiltonian cycle.

Our first attempt is to produce a new graph G' that is the same as G, except that it has an additional edge between s and t. Then if G has a Hamiltonian path from s to t, G' has a Hamiltonian cycle that follows that same path from s to t and returns to s through the additional edge. While this construction works in mapping yes instances of HAMPATH to yes instances of HAMCYCLE, it does not properly map no instances of HAMPATH to no instances of HAMCYCLE. The following illustrates the construction on both a no and a yes instance of HAMPATH, with the dotted line representing the added edge:



In the original graph on the left, there is no Hamiltonian path from s_2 to t, yet the new graph does have a Hamiltonian cycle. Since the function maps a no instance to a yes instance, it does not demonstrate a valid mapping reduction.

The key problem with the transformation above is that it does not force a Hamiltonian cycle to go through the additional edge. If the cycle did go through the additional edge, we could remove that edge from the cycle to obtain a Hamiltonian path in the original graph. To force the cycle through the additions to the graph, we need to add a new vertex, not just an edge. Thus, in place of the single edge, we add two edges with a vertex between them, as in the following:



Now the generated graph on the left does not have a Hamiltonian cycle, but the new graph on the right still does. Thus, this procedure looks like it should work to map yes instances to yes instances and no instances to no instances.

Formally, given $(G = (V, E), s, t)$, we have

$$f(G, s, t) = G' = (V \cup \{u\}, E \cup \{(s, u), (t, u)\})$$

The function is clearly efficient, as it just adds one edge and two vertices to the input graph. We prove that it is also correct:

- Suppose G has a Hamiltonian path from s to t. This path visits all the vertices in G, and it has the structure $(s, v_2, \ldots, v_{n-1}, t)$. The same path exists in G', and it visits all vertices except the added vertex u. The modified path $(s, v_2, \ldots, v_{n-1}, t, u, s)$ with the additions of the edges (t, u) and (u, s) visits all the vertices and returns to s, so it is a Hamiltonian cycle, and $G' \in$ HAMCYCLE.
- Suppose G' has a Hamiltonian cycle. Since it is a cycle, we can start from any vertex in the cycle and return back to it. Consider the specific path that starts at and returns to s. It must also go through u along the path, so it has the structure $(s, \ldots, u, \ldots, s)$. Since the only edges incident to u are (s, u) and (t, u) the path must either enter u from t and exit to s, or it must enter from s and exit to t. We consider the two cases separately.

- **Case 1:** t precedes u. Then the path must have the form $(s, \dots, t, u, s)$. Since the subpath $(s, \dots, t)$ visits all vertices except u and only includes edges from G, it constitutes a Hamiltonian path in G from s to t.
- **Case 2:** t follows u. Then the path has the form $(s, u, t, \dots, s)$. Since the subpath $(t, \dots, s)$ visits all vertices except u and only includes edges from G, it constitutes a Hamiltonian path in G from t to s. The graph is undirected, so following the edges in reverse order gives a Hamiltonian path from s to t.

Thus, G has a Hamiltonian path from s to t.

We have shown that G has a Hamiltonian path from s to t exactly when $G' = f(G, s, t)$ has a Hamiltonian cycle. This demonstrates that f is a valid mapping from instances of HAMPATH to instances of HAMCYCLE, so HAMPATH $\le_p$ HAMCYCLE. We conclude that HAMCYCLE is NP-hard.

Exercise 163 Define the language

$$\text{LONG-PATH} = \{(G, k) : G \text{ is an undirected graph with a simple path of length } k\}$$

A *simple path* is a path without any cycles. Show that LONG-PATH is NP-complete.

Exercise 164 Recall the TSP language:

$$\text{TSP} = \left\{ \begin{array}{l} (G,k) : G \text{ is a weighted, complete graph that has a tour that visits} \\ \qquad\qquad \text{all vertices in } G \text{ and has a total weight at most } k \end{array} \right\}$$

Show that TSP is NP-hard with the reduction HAMCYCLE $\le_p$ TSP.

17.6 Subset Sum

The *subset-sum* problem is the task of finding a subset of set of integers S that add up to a target value k. The decision version of this problem is as follows:

$$\text{SUBSET-SUM} = \left\{ (S,k) : S \text{ is a set of integers that has a subset } S^* \subseteq S \text{ such that } \sum_{s \in S^*} s = k \right\}$$

This language is in NP – we can take the actual subset $S^* \subseteq S$ as a certificate, and a verifier can efficiently check that it is a valid subset of S and that its elements sum to the value k. We show 3SAT $\le_p$ SUBSET-SUM to demonstrate that SUBSET-SUM is also NP-hard: we define a polytime computable function f that takes a 3CNF formula ϕ with n variables and m clauses and produces a set of integers S and target value k such that

$$\phi \in \text{3SAT} \iff f(\phi) = (S, k) \in \text{SUBSET-SUM}$$

A satisfying assignment α of ϕ has two key properties:

- for each variable $x_i \in \phi$, exactly one of $\alpha(x_i)$ or $\alpha(\neg x_i)$ is 1
- for each clause $C_j = (l_{j,1} \vee l_{j,2} \vee l_{j,3}) \in \phi$, at least one of $\alpha(l_{j,1}), \alpha(l_{j,2}), \alpha(l_{j,3})$ is 1

The set of integers S and value k produced by $f(\phi)$ are carefully designed to correspond to these properties. Specifically, the set S consists of decimal (base-10) integers that are $n+m$ digits long – the first n digits correspond to the n variables, while the last m digits correspond to the clauses. For each variable x_i, S contains two numbers v_i and w_i:

- v_i is 1 in digit i and in digit $n + j$ for each clause C_j that contains the literal x_i, 0 in all other digits
- w_i is 1 in digit i and in digit $n + j$ for each clause C_j that contains the literal $\neg x_i$, 0 in all other digits

We also set the ith digit of the target value k to be 1 for $1 \le i \le n$. For a subset S^* to add up to k, it must contain exactly one of v_i or w_i for each $1 \le i \le n$, since only those two numbers contain a 1 in the ith digit. (Since we are using decimal numbers, there aren't enough numbers in S to get a 1 in the ith digit through a carry from less-significant digits.) Thus, a subset S^* that adds up to k corresponds to an actual assignment of values to the variables, enforcing the first property above.

For a clause digit $n+j, 1 \leq j \leq m$, the subset S^* can produce one of the following values corresponding to the clause $C_j = (l_{j,1} \vee l_{j,2} \vee l_{j,3})$:

- 0 if S^* contains none of the numbers corresponding to $l_{j,1}, l_{j,2}, l_{j,3}$ being true (e.g. if $l_{j,1} = x_i$, then S^* does not contain v_i, and if $l_{j,1} = \neg x_i$, then S^* does not contain w_i)
- 1 if S^* contains exactly one number corresponding to $l_{j,1}, l_{j,2}, l_{j,3}$ being true
- 2 if S^* contains exactly two numbers corresponding to $l_{j,1}, l_{j,2}, l_{j,3}$ being true
- 3 if S^* contains all three numbers corresponding to $l_{j,1}, l_{j,2}, l_{j,3}$ being true

For the clause C_j to be satisfied by the assignment corresponding to S^*, all but the first case work. We need to add more numbers to S and set digit $n+j$ of k such that the three latter cases are all viable. To do so, for each clause C_j:

- set the value for digit $n+j$ to be 3 in the target number k
- add two numbers y_j and z_j to S that have a 1 in digit $n+j$ and 0 in all other digits

This completes our construction of S and k. This can be done efficiently – we generate $2n+2m+1$ numbers, each of which is $n+m$ digits long, and we can do so in quadratic time with respect to $|\phi| = O(n+m)$. For a subset $S^* \subseteq S$ to add up to k, it must contain:

- exactly one of v_i or w_i for each variable $x_i, 1 \leq i \leq n$, corresponding to an assignment to the variables
- none, one, or both of y_j and z_j for each clause $C_j, 1 \leq j \leq m$, depending on whether the clause C_j has three, two, or one literal(s) satisfied by the corresponding assignment

As a concrete example, consider the formula

$$\phi(x_1, x_2, x_3, x_4) = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_4)$$

We generate the following numbers, with columns corresponding to digits and unspecified digits implicitly 0:

	x_1	x_2	x_3	x_4	C_1	C_2	C_3
v_1	1				1		
v_2		1			1		1
v_3			1		1		
v_4				1			1
w_1	1						1
w_2		1				1	
w_3			1			1	
w_4				1		1	
y_1					1		
y_2						1	
y_3							1
z_1					1		
z_2						1	
z_3							1
k	1	1	1	1	3	3	3

Consider the satisfying assignment $\alpha(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$. The corresponding subset S^* contains w_1, v_2, w_3, v_4, which add up to 1111113 – clauses C_1 and C_2 are satisfied by one literal each (x_2 and $\neg x_3$, respectively), while clause C_3 is satisfied by all three literals. To add up to $k = 1111333$, S^* also contains y_1, z_1, y_2, z_2.

Now consider the assignment $\alpha'(x_1, x_2, x_3, x_4) = (0, 0, 0, 1)$ that does not satisfy the formula. The corresponding subset S^* contains w_1, w_2, w_3, v_4, which add up to 1111022. The highest value we can get in digit 5 (corresponding to clause C_1) is 2, by adding both y_1 and z_1 to the subset. This does not get us to the target value of 3.

Formally, for an arbitrary formula ϕ with n variables and m clauses and resulting $f(\phi) = (S, k)$, we have:

- If $\phi \in$ 3SAT, then it has an assignment α that satisfies all clauses, meaning that at least one literal in each clause is true. The subset

  $$S_a^* = \{v_i : \alpha(x_i) = 1\} \cup \{w_i : \alpha(x_i) = 0\}$$

  sums to 1 in each of the first n digits, and at least 1 in each of the last m digits – each clause C_j is satisfied by at least one literal, and the digit $n + j$ is set to 1 in the number corresponding to that literal in the subset S_a^*. Thus, the full subset

  $$\begin{aligned} S^* = S_a^* \cup \\ &\{y_j : \phi(\alpha) \text{ sets } \leq 2 \text{ literals of } C_j \text{ to } 1\} \cup \\ &\{z_j : \phi(\alpha) \text{ sets } 1 \text{ literal of } C_j \text{ to } 1\} \end{aligned}$$

  adds up to the target value k, so $f(\phi) = (S, k) \in$ SUBSET-SUM.

- If $f(\phi) = (S, k) \in$ SUBSET-SUM, then there is some subset S^* of the numbers $S = \{v_i\} \cup \{w_i\} \cup \{y_j\} \cup \{z_j\}$ that adds up to k. This subset S^* must contain exactly one of v_i or w_i for each $1 \leq i \leq n$ so that the ith digit in the sum is the target 1. This corresponds to the assignment

  $$\alpha(x_i) = \begin{cases} 1 & \text{if } v_i \in S^* \\ 0 & \text{if } w_i \in S^* \end{cases}$$

  For each digit $n + j, 1 \leq j \leq m$, there must be some v_i or w_i in S^* that has that digit set to 1 – there are only two numbers among the y_j and z_j that have this digit set to 1, so they cannot add up to the target value of 3 on their own. If there is a $v_i \in S^*$ that has the $(n + j)$th digit set to 1, then by construction, the clause C_j contains the literal x_i, so it is satisfied by the assignment α since $\alpha(x_i) = 1$. On the other hand, if there is a $w_{i'} \in S^*$ that has the $(n + j)$th digit set to 1, then the clause C_j contains the literal $\neg x_{i'}$, and it is satisfied by the assignment α since $\alpha(x_{i'}) = 0$. In either case, the clause C_j is satisfied. Since this reasoning applies to all clauses, the assignment α satisfies the formula as a whole, and $\phi \in$ 3SAT.

We have shown that f is efficiently computable and that $\phi \in \text{3SAT} \iff f(\phi) \in \text{SUBSET-SUM}$, so we have 3SAT $\leq_p$ SUBSET-SUM. We can conclude that SUBSET-SUM is NP-hard, and since it is also in NP, that it is NP-complete.

Exercise 165 Define the language KNAPSACK as follows:

$$\text{KNAPSACK} = \left\{ \begin{aligned} (V, W, n, v, C) : |V| = |W| = n \text{ and there exist indices} \\ i \in 1, \ldots, n \text{ such that } \sum_i V[i] \geq v \text{ and} \\ \sum_i W[i] \leq C \end{aligned} \right\}$$

This corresponds to the following problem:

- There are n items.
- Each item i has a value $V(i)$ and a weight $W(i)$, where $V(i)$ and $W(i)$ are integers.
- We have a target value of at least v, where v is an integer.
- We have a weight capacity limit of at most C, where C is an integer.
- Is it possible to select a subset of the items such that the total value of the subset is at least v but

the total weight of the subset is at most C?

Show that KNAPSACK is NP-complete.

17.7 Concluding Remarks

We have explored only the tip of the iceberg when it comes to NP-complete problems. Such problems are everywhere, including constraint satisfaction (SAT, 3SAT), covering problems (vertex cover, set cover), resource allocation (knapsack, subset sum), scheduling, graph colorability, model-checking, social networks (clique, maximum cut), routing (HAMPATH, TSP), games (Sudoku, Battleship, Super Mario Brothers, Pokémon), and so on. An efficient algorithm for any of these problems admits an efficient solution for all of them.

The skills to reason about a problem and determine whether it is NP-hard are critical. Researchers have been working for decades to find an efficient solution for NP-complete problems to no avail. This makes it exceedingly unlikely that we will be able to do so. Instead, when we encounter such a problem, a better path is to focus on approximation algorithms, which we turn to next.

CHAPTER EIGHTEEN

SEARCH AND APPROXIMATION

Thus far, we have focused our attention on decision problems, formalized as languages. We now turn to *functional* or *search* problems, those that do not have a yes/no answer. Restricting ourselves to decision problems does not leave us any room to find an answer that is "close" to correct, but a larger codomain will enable us to consider approximation algorithms for NP-hard problems. We previously encountered *Kolmogorov complexity* (page 261) as an example of an uncomputable functional problem. Here, we consider problems that are computable, with the intent of finding efficient algorithms to solve such problems. Some examples of search problems are:

- Given an array of integers, produce a sorted version of that array.
- Given a Boolean formula, find a satisfiable assignment.
- Given a graph, find a minimum vertex cover.

In general, a search problem may be:

- a *maximization* problem, such as finding a maximum clique in a graph
- a *minimization* problem, such as finding a minimum vertex cover
- an *exact* problem, such as finding a satisfying assignment or a Hamiltonian path

Maximization and minimization problems are also called *optimization problems*.

For each kind of search problem, we can define a corresponding decision problem.[61] For maximization and minimization, we introduce a limited budget. Specifically, does a solution exist that meets the budget? The following are examples:

$$\begin{aligned}\text{CLIQUE} &= \{(G,k) : G \text{ is an undirected graph that has a clique of size } k\}\\ \text{VERTEX-COVER} &= \{(G,k) : G \text{ is an undirected graph that has a vertex cover of size } k\}\end{aligned}$$

For an exact problem, the corresponding decision problem is whether a solution exists that meets the required criteria. The following is an example:

$$\text{SAT} = \{\phi : \phi \text{ is a satisfiable Boolean formula}\}$$

We have seen that the languages CLIQUE, VERTEX-COVER, and SAT are NP-complete. How do the search problems compare in difficulty to the decision problems?

It is clear that given an efficient algorithm to compute the answer to a search problem, we can efficiently decide the corresponding language. For instance, a decider for CLIQUE can invoke the algorithm to find a maximum clique and then check whether the result has size at least k. Thus, the decision problem is no harder than the search problem.

What if we have an efficient decider D for CLIQUE? Can we then efficiently find a clique of maximum size? It turns out that we can. Given a graph $G = (V, E)$, we first find the size of a largest clique by running D on (G, k) for each k from $|V|$ down to 0, stopping at the first k for which D accepts:

[61] We previously saw that a functional problem has an *equivalent formulation as a decision problem* (page 261). However, this correspondence was based on translating a functional problem to a sequence of decision problems on the binary representation of the answer. This is different than what we are considering now, which is between a search problem and a "natural" formulation of a corresponding decision problem.

```
Input: an undirected graph
Output: the size of (number of vertices in) its largest clique
  function $S_1(G = (V, E))$
    for $k = |V|$ down to $0$ do
      if $D(G, k)$ accepts then return $k$
```

Since a clique is a subset of the vertices, its size cannot exceed that of the vertex set, so we query the decider to determine whether there is a clique of this size. If so, we are done; if not, we repeatedly query again with the next-smaller possible size. We stop as soon as we get an affirmative answer. Thus, the algorithm produces the size of a largest clique. The algorithm is also efficient, since it makes as most $|V|$ calls to D, and D is efficient by hypothesis.

Note that in general for other optimization problems, a linear search like this to find the optimal size/weight/etc. may not be efficient; it depends on how the number of possible answers relates to the input size. For the max-clique problem, the size of the search space is linear in the input size, so a linear search suffices. For other problems, the size of the search space might be exponential in the input size, in which case a binary search may be needed.

Once we know the size of a largest clique, we can actually search for a clique of that size. This can be done using either of two common strategies.

The first strategy is a "destructive" one, where we remove pieces of the input until all we have left is an object we're looking for. In the case of the clique problem, we temporarily discard a vertex and all its incident edges, then query the decider to see if the graph still has a clique of the required size. If so, the vertex is unnecessary—there is some largest clique that does not use the vertex—so we remove it permanently. Otherwise, the discarded vertex is part of every largest clique, so we restore it. We repeat this process for all vertices, and at the end, we will be left with just those that make up a largest clique. The algorithm is as follows:

```
Input: an undirected graph $G$ whose largest clique has size $k$
Output: a largest clique in $G$
  function FindMaxClique($G = (V, E), k$)
    for all $v \in V$ do
      let $G'$ be $G$ with $v$ and all its incident edges removed
      if $D(G', k)$ accepts then
        $G = G'$
    return $V(G)$, the set of (remaining) vertices in $G$
```

This algorithm does a single pass over the vertices. Each iteration removes a vertex and its incident edges, which can be done efficiently, and invokes the decider D, which is efficient by hypothesis. Thus, the entire algorithm is efficient.

The second common strategy is a "constructive" one: we build up a solution piece by piece, guessing a piece of the solution and checking the guess by querying the decider. For the clique problem, we can guess that a vertex v is in some clique of size k. If it is, then there must be a clique of size $k - 1$ among just the vertices that are adjacent to v, i.e., its *neighborhood*. This is because a clique of size k is a complete subgraph of size k, and if we ignore one vertex in this subgraph, the rest is a complete subgraph of size $k - 1$. Conversely, if the neighborhood of v has a clique of size $k - 1$, then adding v to it produces a clique of size k, since there is an edge between v and each vertex in its neighborhood. So, our algorithm iterates over each vertex in the graph and checks if its neighborhood has a clique of size one smaller than currently desired. If so, we include v in our final output and continue to search just within the neighborhood for a clique of the smaller size. If not, the vertex is not part of any clique of the desired size, and we just move on to the next candidate vertex. The formal algorithm is as follows:

```
Input: an undirected graph $G$ whose largest clique has size $k$
Output: a largest clique in $G$
  function FindMaxClique($G = (V, E), k$)
    $C = \emptyset$
    for all $v \in V$ do
```

```
        $G' = \text{NEIGHBORHOOD}(G, v)$
        **if** $D(G', k-1)$ accepts **then**
            $C = C \cup \{v\}$
            $G = G'$
            $V = V(G')$
            $k = k - 1$
    **return** $C$
**function** NEIGHBORHOOD($G = (V, E), v$)
    $V' = \{u \in V : u \neq v, (u, v) \in E\}$        ▷ all neighbors of $v$ (other than $v$ itself)
    $E' = \{(u, w) \in E : u, w \in V'\}$        ▷ all edges between $v$'s neighbors
    **return** $(V', E')$
```

This algorithm also does a single pass over the vertices in G. Each iteration computes the neighborhood of a vertex, which can be done efficiently by in turn iterating over all the vertices and edges in the graph. Each iteration of the algorithm also invokes D, which is efficient by hypothesis. Since the algorithm does a linear number of iterations and each is efficient, the algorithm as a whole is also efficient.

Example 166 Suppose we have an efficient decider D for VERTEX-COVER. We can efficiently find a minimum vertex cover as follows. Given a graph $G = (V, E)$, we do the following:

- Find the size of a minimum vertex cover by running D on (G, k) for each k from 1 to $|V|$, stopping on the first k for which D accepts.
- We use a "constructive" strategy to deduce which vertices are part of the cover. Pick a vertex v. If it is in a minimum vertex cover of size k, we can remove it and all adjacent edges, and the resulting graph has a vertex cover of size $k - 1$. If v is not in a minimum vertex cover, the resulting graph will not have a vertex cover of size $k - 1$. Thus, we can use a call to D on the new graph to determine whether or not v is in a minimum cover. If it is, we continue our search on the smaller graph, otherwise we restore the previous graph. We continue the search until we've checked each vertex for whether it is in the minimum cover.

The full search algorithm is as follows:

```
**Input:** an undirected graph
**Output:** a minimum vertex cover of the graph
  **function** MINVERTEXCOVER($G = (V, E)$)
      $k = 0$
      **while** $D(G, k)$ rejects **do**
          $k = k + 1$
      $C = \emptyset$
      **for all** $v \in V$ **do**
          let $G'$ be $G$ with $v$ and all its incident edges removed
          **if** $D(G', k-1)$ accepts **then**
              $C = C \cup \{v\}$
              $G = G'$
              $k = k - 1$
      **return** $C$
```

The first loop invokes D at most $|V|$ times, since a cover is a subset of all the vertices. Since D is efficient, the loop completes in polynomial time. The second loop invokes D at most $|V|$ times, and G' can be constructed efficiently, so the loop as a whole is efficient. Thus, the algorithm runs in polynomial time with respect to $|G|$. We conclude that if we can decide VERTEX-COVER efficiently, we can find an actual minimum vertex cover efficiently.

Exercise 167 Given an efficient decider D for SAT, demonstrate an efficient algorithm that finds a satisfying assignment for a Boolean formula ϕ, if such an assignment exists.

We have shown that the search problems of finding a maximum clique or minimum vertex cover has the same difficulty as the decision problems CLIQUE or VERTEX-COVER, respectively. Since CLIQUE and VERTEX-COVER are NP-complete, this means that an efficient solution to either search problem leads to $\mathsf{P} = \mathsf{NP}$. Informally, we say that the search problems are *NP-Hard*, much as we did for a decision problem for which an efficient solution implies $\mathsf{P} = \mathsf{NP}$. (Note, however, that the search problems are **not** NP-complete, as the class NP consists only of decision problems.)

We claim without proof that an NP-hard search problem has an efficient algorithm if and only if its corresponding decision problem is efficiently solvable. Thus, we are unlikely to construct efficient algorithms to find exact solutions to NP-hard search problems. Instead, we focus our attention on designing efficient algorithms to *approximate* NP-hard optimization problems.

Given an input x, an optimization problem seeks to find an output y such that value(x, y) is optimized for some metric $value$. For a maximization problem, the goal is to find an optimal value y^* such that:

$$\forall y.\ \text{value}(x, y^*) \geq \text{value}(x, y)$$

That is, value(x, y^*) is at least as large as value(x, y) for all other outputs y, so that y^* maximizes the metric for a given input x. Similarly, in a minimization problem, the goal is to find y^* such that:

$$\forall y.\ \text{value}(x, y^*) \leq \text{value}(x, y)$$

Here, y^* produces the minimum value for the metric value(x, y) over all y. In both cases, we define $OPT(x) =$ value(x, y^*) as the optimal value of the metric for an input x. We then define an approximation as follows:

Definition 168 (α-approximation) Given an input x, a solution y is an *α-approximation* if:

- For a maximization problem,

  $$\alpha \cdot OPT(x) \leq \text{value}(x, y) \leq OPT(x)$$

  where $\alpha < 1$.

- For a minimization problem,

  $$OPT(x) \leq \text{value}(x, y) \leq \alpha \cdot OPT(x)$$

  where $\alpha > 1$.

In both cases, α is the *approximation ratio*.

The closer α is to one, the better the approximation.

Definition 169 (α-approximation Algorithm) An *α-approximation algorithm* is an algorithm A such that for all inputs x, the output $A(x)$ is an α-approximation.

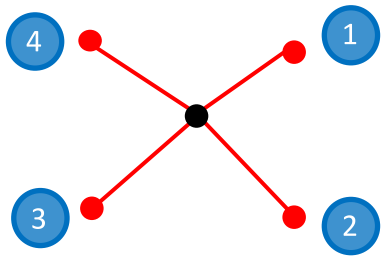
18.1 Approximation for Minimum Vertex Cover

Let us return to the problem of finding a minimum vertex cover. Can we design an efficient algorithm to approximate a minimum cover with a good approximation ratio α? Our first attempt is the *cover-and-remove* algorithm, which repeatedly finds an edge, removes one of its endpoints, and removes all edges incident to that endpoint:

```
Input: an undirected graph
Output: a vertex cover of the graph
  function CoverAndRemove($G$)
      $C = \emptyset$
```

```
    while $G$ has at least one edge do
        choose an arbitrary edge $e = (u, v) \in E$
        remove $v$ and all its incident edges from $G$
        $C = C \cup \{v\}$
    return $C$
```

How well does this algorithm perform? Consider the following star-shaped graph:

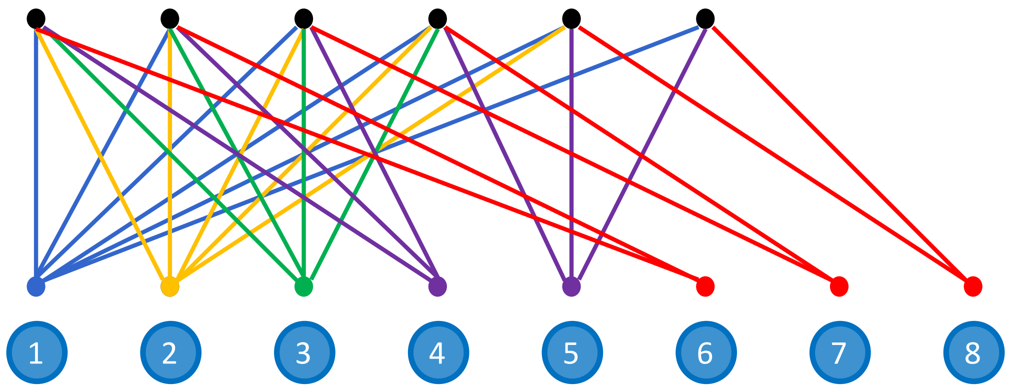


On this graph, it is possible for the algorithm to pick the vertices on the outside of the star to remove rather than the one in the middle. The result would be a cover of size four, while the optimal cover has size one (just the vertex in the middle), giving an approximation ratio of 4 for this graph. However, larger star-shaped graphs can give rise to arbitrarily larger approximation ratios – there is no constant α such that the cover-and-remove algorithm achieves a ratio of α on all possible input graphs.

Observe that in the star-shaped graph, the vertex in the center has the largest *degree*, the number of edges that are incident to it. If we pick that vertex first, we cover all edges, achieving the optimal cover of just a single vertex for the star-shaped graph. This motivates a greedy strategy that repeatedly adds and removes a vertex with the *largest* remaining degree to the cover:

```
Input: an undirected graph
Output: a vertex cover of the graph
  function GreedyCover($G$)
    $C = \emptyset$
    while $G$ has at least one edge do
        choose a vertex $v \in V$ having the most remaining incident edges
        remove $v$ and all its incident edges from $G$
        $C = C \cup \{v\}$
    return $C$
```

While this algorithm works well for a star-shaped graph, there are other graphs for which it does not achieve an approximation within a constant α. Consider the following bipartite graph:



The optimal cover consists of the six vertices at the top, while the algorithm instead chooses the eight vertices at the bottom. The vertex at the bottom left has a degree of six, compared to the maximum degree of five among the top vertices, so the bottom-left vertex is picked first. Then the second vertex on the bottom has degree five, while the top vertices now have degree no more than four. Once the second vertex is removed, the vertex with highest degree is the third on the bottom, and so on until all the bottom vertices have been chosen for the cover. While the algorithm achieves a ratio of $4/3$ on this graph, larger graphs can be constructed with a similar structure, with k vertices at the top and ~$k \log k$ at the bottom. The algorithm produces a cover with ratio $\log k$ compared to the optimal, and that ratio can be made arbitrarily large by increasing k.

The problem with both the cover-and-remove and greedy algorithms is the absence of a *benchmark*, a way of comparing the algorithm against the optimal result. Before we proceed to design another algorithm, we establish a measure for a minimum vertex cover that we can benchmark against, using a set of pair-wise disjoint edges from the input graph.

Given a graph $G = (V, E)$, a subset of edges $S \subseteq E$ is *pair-wise disjoint* if no two edges in S share a common vertex. Given any pair-wise disjoint set of edges S from G, any vertex cover C must have at least $|S|$ vertices – since no two edges in S share a common vertex, we must choose a distinct vertex to cover each edge in S. This gives us a way of benchmarking an approximation algorithm – ensure that the algorithm picks no more than a constant number of vertices for each edge in some pair-wise disjoint set of edges S.

In fact, only a small modification to the cover-and-remove algorithm is necessary to meet this benchmark. Rather than selecting just one of the two endpoints of each chosen edge, we include *both* of them in the cover, and remove them and all their incident edges. The remaining edges do not share a vertex with the edge we chose, so they are each disjoint with respect to that edge. By induction, the full set of edges chosen by this *double-cover* algorithm is pairwise disjoint. The algorithm is as follows:

```
Input: an undirected graph
Output: a 2-approximate minimum vertex cover of the graph
  function DoubleCover($G$)
      $C = \emptyset$
      while $G$ has at least one edge do
          choose an arbitrary edge $e = (u, v) \in E$
          remove $u, v$ and all their incident edges from $G$
          $C = C \cup \{u, v\}$
      return $C$
```

Claim 170 *The edges chosen by the double-cover algorithm are pairwise disjoint for any input graph.*

Proof 171 Let S_i be the set of edges chosen by the double-cover algorithm in the first i iterations on an arbitrary graph G. We show by induction that S_i is pair-wise disjoint.

- **Base case:** S_1 contains only a single edge, so it is trivially pair-wise disjoint.
- **Inductive step:** Assume S_i is pair-wise disjoint. When the algorithm chooses an edge (u, v), it removes both u and v and all edges incident to either u or v. Thus, the graph that remains after iteration i does not contain any of the vertices u, v for each edge $(u, v) \in S_i$. The edges that remain in the graph all have endpoints that are distinct from those that appear in S_i, so all remaining edges are disjoint from any edge in S_i. Then no matter what edge e the algorithm chooses next, it will be disjoint from any edge in S_i. Since S_i itself is pair-wise disjoint, $S_{i+1} = S_i \cup \{e\}$ is also pair-wise disjoint.

How good is this algorithm? Let S be the full set of edges chosen by the algorithm on an input graph G. Since S is pair-wise disjoint, the minimum vertex cover C^* of G must have at least $|S|$ vertices. The cover C produced by the double-cover algorithm has $2|S|$ vertices. We have:

$$\text{value}(G, C) = 2|S| \leq 2 \cdot \text{value}(G, C^*)$$

Thus, the resulting cover is a 2-approximation, and double-cover is a 2-approximation algorithm.

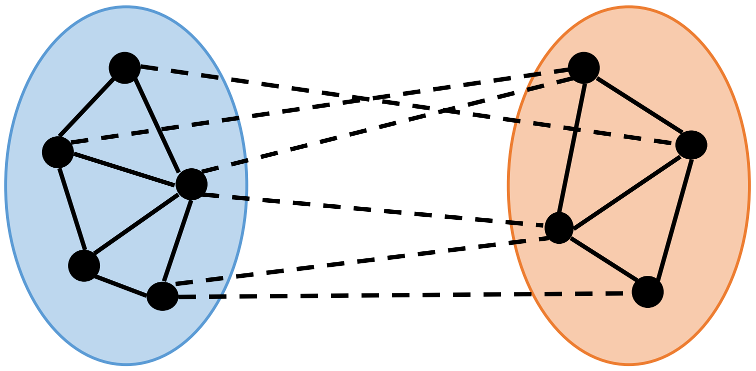
18.2 Maximum Cut

As another example, we design an approximation algorithm for the NP-hard problem of finding a *maximum cut* in an undirected graph. A *cut* in a graph $G = (V, E)$ is just a subset $S \subseteq V$ of the vertices; this partitions V into two disjoint sets, S and its complement $\overline{S} = V \setminus S$. The *size* of a cut S is the number of edges that cross between S and $\overline{S}$. More formally, we define the set of *crossing* edges of a cut S:

$$C(S) = \{(u, v) \in E : u \in S \text{ and } v \in T\} .$$

Then the size of the cut S is $|C(S)|$. Observe that for any cut S, its complement cut $\overline{S}$ has the same crossing edges and hence the same size, because the edges (u, v) and (v, u) are the same.

The following is an example of a cut S, with the crossing edges $C(S)$ represented as dashed lines:



A *maximum cut* of a graph (or max-cut, for short) is a cut that has maximum size among all cuts in the graph. Maximum cuts have important applications to circuit design, combinatorial optimization, and statistical physics.

The decision version of the maximum-cut problem is defined as:

$$\text{MAX-CUT} = \{(G, k) : G \text{ has a cut of size } k\} .$$

This is an NP-complete problem, though we do not prove it here. This implies that the search problem of finding a maximum cut is NP-hard. So, rather than trying to find an exact solution to the search problem, we design an approximation algorithm. We start by identifying an appropriate benchmark.

Given a graph $G = (V, E)$, we define the set $I_v \subseteq E$ to be the edges incident to vertex $v \in V$:

$$I_v = \{(u, v) \in E : u \in V\} .$$

In other words, I_v is the set of edges that have v as an endpoint. We claim the following:

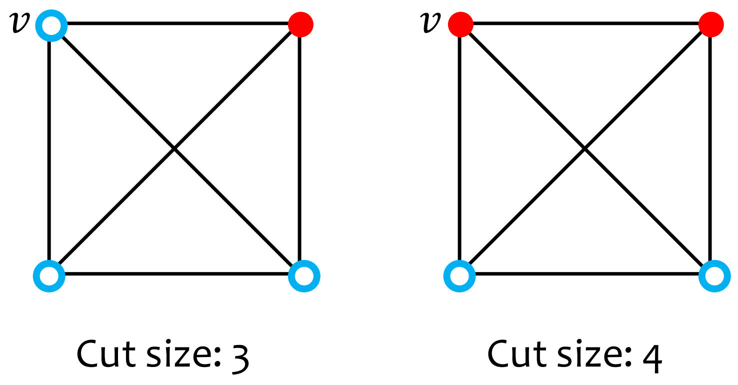
Claim 172 *Let S^* be a maximum cut in a graph $G = (V, E)$. For every vertex $v \in V$, at least half the edges of I_v are in the crossing set $C(S^*)$.*

Proof 173 We prove the contrapositive: if for some vertex $v \in V$, fewer than half the edges of I_v are in $C(S^*)$, then S^* is not a maximum cut. Without loss of generality, assume that $v \in S^*$ (otherwise, we can replace S^* with its complement). Then the edges I_v can be partitioned into two subsets:

$$I_{v,c} = \{(u, v) \in I_v : u \notin S^*\}$$
$$I_{v,nc} = \{(u, v) \in I_v : u \in S^*\}$$

In words, $I_{v,c} \subseteq C(S^*)$ consists of the edges incident to v that cross the cut, while $I_{v,nc}$ consists of those that do not (both endpoints are in S^*). By assumption, $|I_{v,c}| < |I_{v,nc}|$.

If we remove v from S^*, then the edges in $I_{v,nc}$ become crossing edges, while those in $I_{v,c}$ are no longer crossing edges, as illustrated below.



Thus, removing v from S^* increases the number of crossing edges by $|I_{v,nc}| - |I_{v,c}| > 0$, i.e., $|C(S^* \setminus \{v\})| > |C(S^*)|$. Because there is a cut with more crossing edges than those of S^*, we conclude that S^* is not a maximum cut, as claimed. □

Corollary 174 *The size of any maximum cut S^* in a graph $G = (V, E)$ is at least $|E|/2$, i.e., $|C(S^*)| \geq |E|/2$.*

Proof 175 We first observe that

$$|E| = \frac{1}{2}\sum_{v \in V} |I_v|$$

since each edge appears in exactly two sets I_v. Similarly,

$$|C(S^*)| = \frac{1}{2}\sum_{v \in V} |I_{v,c}|$$

where $I_{v,c}$ is the set of crossing edges incident to v. By Claim 172, $|I_{v,c}| \geq |I_v|/2$, which gives us:

$$\begin{aligned} |C(S^*)| &= \frac{1}{2}\sum_{v \in V} |I_{v,c}| \\ &\geq \frac{1}{2}\sum_{v \in V} |I_v|/2 \\ &= |E|/2\ . \end{aligned}$$

We have proved a lower bound on the number of edges in any maximum cut. A trivial upper bound is the total number of edges $|E|$, because the crossing edges are a subset of E. In summary, for any maximum cut S^*,

$$|E|/2 \leq |C(S^*)| \leq |E|\ .$$

Having established bounds on the size of a maximum cut, we now design an algorithm that outputs a cut of size at least $|E|/2$, which is a 1/2-approximation of the optimal. We take inspiration from the argument in Proof 173: the algorithm repeatedly finds a vertex for which moving it to the opposite side of the cut increases the cut size, until no such vertex exists. We call this the *local-search* algorithm for max-cut.

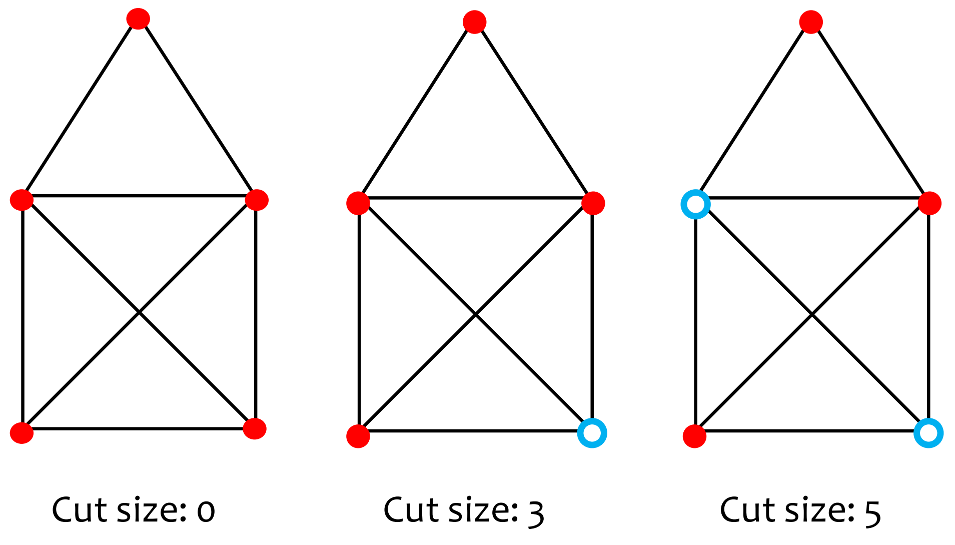
```
Input: an undirected graph
Output: a $1/2$-approximate maximum cut in the graph
  function LocalSearchCut($G = (V, E)$)
```

```
$S = \emptyset$
**repeat**
    **for all** $v \in S$ **do**
        **if** $|C(S \setminus \{v\})| > |C(S)|$ **then**
            $S = S \setminus \{v\}$
    **for all** $u \notin S$ **do**
        **if** $|C(S \cup \{u\})| > |C(S)|$ **then**
            $S = S \cup \{v\}$
**until** no change was made to $S$ in the most recent iteration **return** $S$
```

The algorithm halts when there is no vertex for which moving it to the opposite side of the cut S increases the cut size. By the reasoning in Proof 173, this means that for every vertex v, at least half the edges in I_v are crossing edges of S. Thus, $C(S)$ has at least half the edges in E, i.e., $|C(S)| \geq |E|/2 \geq |C(S^*)|/2$ for any cut S^*. Thus, S is a 1/2-approximation to a maximum cut in the graph.

Since the algorithm makes at least one change in each iteration (except the last one) of the outer loop, and each move increases the size of the cut, we can argue by the *potential method* (page 7) that the algorithm halts within $|E| + 1$ iterations (because S starts with zero crossing edges, and can have no more than $|E|$ crossing edges). Each iteration loops over all the vertices and tests whether moving each one to the other side of the cut increases the cut size, which can be done efficiently.

The following illustrates the execution of the local-search algorithm on a sample graph:



Initially, all of the vertices are in $\overline{S}$, illustrated above as solid circles. The algorithm chooses the bottom-right vertex to move to the other side of the cut, since doing so increases the cut size from zero to three. We represent its membership in S by depicting the vertex as an open circle. Then, the algorithm happens to choose the vertex at the middle left, because doing so increases the cut size to five. At this point, there is no single vertex that can be moved to the other side of the cut to increase the cut size, so the algorithm terminates. (Incidentally, the maximum cut for this graph has six edges; can you find it?)

18.3 Knapsack

The *knapsack problem* involves selecting a subset of items that maximizes their total value, subject to a weight constraint. More specifically, we have n items, and each item i has a value v_i and a weight w_i. We have a weight capacity C, and we wish to find a subset S of the items that maximizes the total value

$$\text{value}(S) = \sum_{i \in S} v_i$$

without exceeding the weight limit:

$$\sum_{i \in S} w_i \le C$$

In this problem formulation, we are not permitted to select the same item more than once; for each item, we either select it to be in S or we do not. Thus, this formulation is also known as the *0-1 knapsack problem*. For simplicity, we assume that each item's weight is no more than C – otherwise, we can immediately remove this item from consideration.

A natural dynamic-programming algorithm to compute an optimal set of items does $O(nC)$ operations, given n items and a weight limit C. Assuming that the weights and values of each item are numbers that are similar in size to (or smaller than) C, the total input size is $O(n \log C + \log C) = O(n \log C)$, with all numbers encoded in a non-unary base.[62] Thus, the dynamic-programming solution is not efficient with respect to the input size. In fact, this problem is known to be NP-hard, so we are unlikely to find an efficient solution to solve it exactly.

Instead, we attempt to approximate the optimal solution to knapsack with a greedy algorithm. Our first attempt is the *relatively greedy algorithm*, in which we select items by decreasing *relative* value, until we cannot take any more. In other words, we choose items in decreasing order by the ratio v_i/w_i.

```
Input: arrays of item values and weights, and a knapsack capacity
Output: a valid selection of items
  function RelativelyGreedy($V[1,\dots,n], W[1,\dots,n], C$)
    let $D[1,\dots,n]$ be indices $1,\dots,n$, sorted so that $V[D[i]]/W[D[i]]$ are non-increasing
    $S = \emptyset$, weight $= 0$
    for $i = 1$ to $n$ do
      if weight $+ W[D[i]] \le C$ then
        $S = S \cup \{D[i]\}$
        weight $=$ weight $+ W[D[i]]$
    return $S$
```

This algorithm runs in $O(n \log n \log C)$ time, where $n = |V|$; the time is dominated by sorting, which does $O(n \log n)$ comparisons (e.g. using *merge sort* (page 13)), and each comparison takes $O(\log C)$ time for numbers that are similar in size to C. Thus, this algorithm is efficient with respect to the input size.

Unfortunately, the relatively greedy algorithm is not guaranteed to achieve any approximation ratio α. Consider what happens when we have two items, the first with value $v_1 = 1$ and weight $w_1 = 1$, and the second with value $v_2 = 100$ and weight $w_2 = 200$. Then item 1 has a value-to-weight ratio of 1, while item 2 has a value-to-weight ratio of 0.5. This means that the relatively greedy algorithm will always try to pick item 1 before item 2. If the weight capacity is $C = 200$, the algorithm produces $S = \{1\}$, with a total value of $\text{value}(S) = 1$. However, the optimal solution is $S^* = \{2\}$, with a total value of $\text{value}(S^*) = 100$. This gives us an approximation ratio of 0.01, but we can easily find other examples with arbitrarily smaller ratios.

The main problem in the example above is that the relatively greedy algorithm ignores the most valuable item in favor of one with a better value-to-weight ratio. A *"single-greedy" algorithm* that takes just one item that has the largest value actually produces an optimal result for the above example. This algorithm is as follows:

[62] An algorithm is said to run in *pseudopolynomial time* if its runtime complexity is polynomial in the *value* of the integers in the input, as opposed to the *size* of the integers. Another way of defining pseudopolynomial time is for an algorithm to run in polynomial time with respect to the input when all numbers in the input are encoded in unary. The dynamic-programming algorithm for knapsack takes pseudopolynomial time.

```
function SingleGreedy($V, W$) return an index $i$ that maximizes $V[i]$
```

This algorithm runs in time $O(n \log C)$, because it make a single pass over the array of values, keeping track of the largest value (and its index) found so far. While it achieves the optimal result for our previous example, it is straightforward to construct examples where the single-greedy algorithm fails to achieve any approximation ratio α. Suppose we have 201 items, where all but the last item have value 1 and weight 1, while the last item has value 2 and weight 200. In other words, we have

$$\begin{aligned} v_1 = v_2 = \cdots = v_{200} = 1 \\ w_1 = w_2 = \cdots = w_{200} = 1 \\ v_{201} = 2 \\ w_{201} = 200 \end{aligned}$$

With a weight limit $C = 200$, the optimal solution is to pick items $1, \dots, 200$ for a total value of 200, but the single-greedy algorithm picks the single item 201, for a value of 2. Thus, it achieves a ratio of just 0.01 on this instance of the problem. Moreover, the example can be modified to make the ratio arbitrarily small.

Observe that for this second example, the relatively greedy algorithm actually would produce the optimal solution. It seems like the single-greedy algorithm does well where the relatively greedy algorithm fails, and the relatively greedy algorithm performs well when the single-greedy algorithm does not. This motivates us to construct a *combined-greedy algorithm* that runs both the relatively greedy and single-greedy algorithm and picks the better of the two solutions. It can be shown that the combined-greedy algorithm 1/2-approximates the optimal solution for any instance of the knapsack problem.

```
function CombinedGreedy($V, W, C$)
    $S_1$ = RelativelyGreedy($V, W, C$)
    $S_2$ = SingleGreedy($V, W$)
    if value($S_1$) > value($S_2$) then
        return $S_1$
    else
        return $S_2$
```

Exercise 176 In this exercise, we will prove that the combined-greedy algorithm is a 1/2-approximation algorithm for the 0-1 knapsack problem.

a) Define the *fractional knapsack problem* as a variant that allows taking any partial amount of each item. (For example, we can take $3/7$ of item i, for a value of $(3/7)v_i$, at a weight of $(3/7)w_i$.) Prove that for this fractional variant, the optimal value is no smaller than for the original 0-1 variant (for the same weights, values, and knapsack capacity).

b) Prove that the sum of the values obtained by the relatively greedy and single-greedy algorithm is at least the optimal value for the fractional knapsack problem.

c) Using the previous parts, prove that the combined-greedy algorithm is a 1/2-approximation algorithm for the 0-1 knapsack problem.

18.4 Other Approaches to NP-Hard Problems

While we do not know how to efficiently find exact solutions to NP-hard problems, we can apply heuristics to these problems. There are two general kinds of heuristics:

- Those that are guaranteed to get the right answer, but may take a long time on some inputs. However, they may be efficient enough for inputs that we care about. Examples include the dynamic-programming algorithm for 0-1 knapsack where the weight capacity C is small compared to the number of items n, as well as SAT solvers for Boolean formulas that emerge from software verification.
- Those that are guaranteed to run fast, but may produce an incorrect or suboptimal answer. The α-approximation algorithms we considered above are examples of this, as well as greedy algorithms for the traveling salesperson problem, which has no α-approximation algorithm for general instances of the problem.

Some algorithms are only applicable to special cases of a problem. For instance, there are efficient algorithms for finding a maximum cut on *planar* graphs, which can be depicted on a plane without any crossing edges.

Another general approach is randomized algorithms, which may be able to find good solutions with high probability. We will consider such algorithms in the next unit.