CS 577: Introduction to Algorithms Fall 2019

Computational Intractability

Instructor: Dieter van Melkebeek

DRAFT

(Last updated: November 13, 2019)

We have so far focused solely on problems which yield efficient solutions.[1][2] We now turn to some of the theory behind problems for which efficient algorithms are unknown. This section will be a bit more abstract than previous sections—problems will no longer have concrete descriptions, instances will not have concrete interpretations, and we will sometimes need to think of efficient algorithms as if they exist, even when they may not. While conceptually more difficult, this point of view grants us the power to consider *arbitrary* algorithms—even ones that we have yet to discover, and even ones that may not exist—and to prove things about them. We will see the power of this abstract way of thinking as we develop the notion of NP-completeness: there is a large suite of natural problems for which the existence of an efficient algorithm for *any single one* of these problems will imply an efficient algorithm for the vast majority of computational problems in computer science, engineering, and mathematics. Moreover, we will interpret the theory as telling us that efficient solutions to these special problems likely don't exist in the first place, and thereby give a sensible (if not entirely rigorous) criterion for when a particular computational problem does not admit an efficient algorithm.

1 The class NP

Recall the fully-general knapsack problem:

Input: A set of n items with nonnegative weights $w_1, w_2, \ldots, w_n$, nonnegative values $v_1, v_2, \ldots, v_n$, and a nonnegative knapsack capacity W.

Output: A subset $I \subseteq \{1, 2, \cdots, n\}$ of the items whose total weight is at most W, and whose total value is maximized.

We gave algorithms for the knapsack which are efficient under certain conditions, but left open the question of whether it has a general efficient algorithm. Indeed, there is no known efficient algorithm for the general knapsack problem.

Another problem that will be of interest to us is the maximum independent set problem (`MIS`). In a graph $G = (V, E)$, we say that a subset $I \subseteq V$ of vertices is *independent* (or *stable*) if no edge contains more than one element of I; *i.e.*, if (u, v) is an edge in G, then either u is not in I, or else v is not in I.

[1] Efficient here will mean "has running time bounded by a polynomial in the bit-length of the input". Most sane encodings of problems can be transformed from one to the other in polynomial time. For instance, transforming between the adjacency-list and adjacency-matrix representations of graphs can be done in time $O(n^2)$, where n is the number of vertices. A notable exception to this is the encoding of integers in unary: while 2^n uses only $n + 1$ bits in the usual encoding of binary, it requires 2^n digits in base-1.

[2] Pseudo-exception: the knapsack problem, for which we gave only a pseudo-polynomial-time algorithm.

Input: A graph $G = (V, E)$.

Output: A subset $I \subseteq V$ of the vertices which is independent and has maximum size.

Like the knapsack problem, this problem also is not known to have an efficient solution.

There are certain aspects to these problems that make them seem feasible:

- Valid solutions can be represented by strings of length polynomial in the input size
- Whether or not a given string is valid can be checked in time polynomial in the input size
- The objective value of a valid candidate solution can be evaluated in time polynomial in the input size

In other words, it is easy to *check* that solutions are valid, and to see how good they are. It's easy to see how the knapsack and independent set problems fit into this categorization: the solution sets I can be encoded as bit strings of length n; checking the validity and evaluating the objective value of a given candidate solution can be done easily in both cases.

Many computational problems from many different disciplines fit into this framework. Given their pervasiveness, it is useful to give problems of this sort a name. Problems with this structure are said to be NP-*optimization*[3] problems.

Formally, we define an NP-optimization problem as follows. The problem is specified by a *solution validity checker* V, a natural number d, and an *objective function* f. Inputs to the problem are encoded as strings x of bits (symbolically: $x \in \{0,1\}^*$), and candidate solutions are encoded as strings y of bits ($y \in \{0,1\}^*$). V determines which choices of y are *valid* solutions for a given input x; *i.e.*, V computes some function $\{0,1\}^* \times \{0,1\}^* \to \{0,1\}$, where we interpret $V(x,y)$ as deciding whether y is a valid solution for the instance x. f is likewise a function $\{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$ such that $f(x,y)$ is the objective value of the solution y on input x.

Corresponding to the three bullet points above, these quantities satisfy the following properties:

- To each input x, there is an associated the *set of valid solutions* (or *witnesses*), S_x. S_x is defined to be the set of strings y that V recognizes y as a valid solution for the instance x. Symbolically
$$S_x \doteq \{y \in \{0,1\}^* : V(x,y) = 1\}$$
We further require that if $y \in S_x$, then $|y| \leq |x|^d$; *i.e.*, , the witnesses can be described by polynomial-length strings.
- $V(x,y)$ can be computed in time bounded by a polynomial in the lengths of x and y. (Note that if $|y| \leq |x|^d$, then, since d is fixed, "bounded by a polynomial in $|x|$ and $|y|$" is equivalent to "bounded by a polynomial in $|x|$".)
- $f(x,y)$ can be computed in time bounded by a polynomial in the lengths of x and y.

[3] The name NP refers to the traditional, formal definition of NP as a class of problems solvable in so-called nondeterministic polynomial time. The term "nondeterministic" refers to the (unrealistic) power for a computer to guess answers, making correct guesses whenever possible, as long as it can efficiently detect when it has made bad guesses. The connection with our definition is that such machines can simply guess the right solution (y), and then run V to check that y is a valid solution for the instance x.

The goal in these NP-optimization problems is, for each instance x, to find a valid solution y^* in S_x so that $f(x, y^*)$ is at least as good as every other valid solution y in S_x. (If the problem is a maximization problem, then we want $f(x, y^*) \geq f(x, y)$ for all y; if the problem is a minimization problem, then we want $f(x, y^*) \leq f(x, y)$ for all y.) If x has no valid solution (S_x is empty), then we require any algorithm solving the problem to declare S_x empty in this case.

As an example, consider the knapsack problem. The strings x encode the full instance: x is some reasonable encoding of the number of items, n, the weights of each item, $w_1, \ldots, w_n$, the values of each item, $v_1, \ldots, v_n$, and the knapsack capacity, W. Solutions are subsets $I \subseteq \{1, \cdots, n\}$, which we can encode as n-bit strings; these are the y's. In any case, $|x| \geq n$, while $|y| = n$, so, by choosing $d = 1$, we get $|y| \leq |x|^d$ for all x and all y of interest. The verifier $V(x, y)$ simply checks that the subset encoded by y has total capacity at most W. Clearly y encodes a valid solution for the instance x if and only if $V(x, y) = 1$. Hence the set S_x is just the collection of n-bit strings y that encode choices of items that fit into the knapsack. Moreover, V clearly runs in time bounded by a polynomial in $|x|$ and $|y|$. Finally, the objective function $f(x, y)$ simply sums the values of the items appearing in the solution encoded by y.

We can also consider the independent set problem. The strings x encode the full instance: x is a reasonable encoding of a graph G, which we'll say has n vertices. Solutions are subsets of the vertices, and hence can be encoded as n-bit strings. The interesting choices of y above are thus n-bit strings, which we think of as the corresponding subsets of the vertices. Clearly, the y we are interested in have $|y| \leq |x|^1$, so we can pick $d = 1$. The verifier $V(x, y)$ simply checks that, for each edge in G, it is not the case that both incident vertices are in y. This clearly runs in polynomial time, and we get the witness set S_x, containing all encodings of independent sets. The objective function $f(x, y)$ simply counts the number of vertices in y, which can be done in polynomial time.

Here are some additional examples of NP-optimization problems:

Traveling Salesman Problem (`TSP`) In the traveling salesman problem, a salesman has a list of n cities he wishes to visit, as well as all pair-wise distances between them. His goal is to visit each city exactly once and return to his starting point, and to travel the minimum total distance in doing so. Formally,

Input: A complete undirected graph $G = (V, E)$ and edge-length function $\ell : E \to \mathbb{R}_{\geq 0}$.

Output: A simple cycle C in G which contains every vertex and which has minimum length.

There is a slight variation on `TSP` which is also worth mentioning, where we drop the requirement that the salesperson returns to his starting point. In this case, we can phrase the goal as to find a simple path P which contains every vertex and which has minimum total length.

Vertex Cover (`VC`) In the vertex cover problem, the input is a graph, and the goal is to find a minimum-size set of vertices (the "cover") so that every edge is incident to at least one vertex in the cover. (Such an edge is *covered* by the cover.) Formally,

Input: An undirected graph $G = (V, E)$.

Output: A minimum-size subset $J \subseteq V$ of the vertices so that, for every edge (u, v) in G, either u is in J, or else v is in J.

There is a relationship between vertex cover and independent set that will become important later. In particular, for any graph G, if I is an independent set in G, then $J \leftarrow V \setminus I$ is a vertex cover of G. This follows straightforwardly from the definitions, but can also be seen intuitively by focusing in on a single particular edge (u, v). We know that either u or v is not in I, so either u or v is in J. Similarly, if J is a vertex cover of G, then $I \leftarrow V \setminus J$ is an independent set in G.

As another note, the vertex cover problem can be solved efficiently on certain kinds of graphs. If G is a tree, then there is a greedy algorithm. If G is bipartite, then there is an algorithm based on network flow. (Exercise: find these algorithms. A hint for the bipartite case: the size of a maximum matching in a bipartite graph is related to the minimum size of a vertex cover.)

Graph Coloring A *k-coloring* of G is a way of coloring the vertices of G using k colors, so that, for each edge, the incident vertices are colored differently. In the graph coloring problem, the input is a graph G, and the goal is to find the smallest k so that G has a k-coloring. Formally,

Input: A graph $G = (V, E)$.

Output: A coloring $c : V \to \{1, 2, \cdots, k\}$ of G so that, for every edge (u, v) in G, $c(u) \neq c(v)$, and so that the choice of k is minimized.

The graph coloring problem has applications to register allocation in compilers.

Like vertex cover, the graph coloring problem has a few easy simple cases. If we restrict ourselves to bipartite graphs G, then we can always find a 2-coloring of G, by coloring one side of the bipartite decomposition one color, and coloring the other side the other color. If we restrict ourselves to planar[4] graphs G, then we can always find a 4-coloring of G using the Four-Color Theorem.

1.1 NP-search and NP-decision

The problems we just saw all naturally fit into the NP-optimization framework. Other problems, however, fit more naturally into slight variations on NP-optimization, called NP-search and NP-decision.

In an NP-search problem, the setup is essentially identical to the NP-optimization problem, except that there is no longer an objective function. For example, we might express graph colorability as follows: given a graph G and natural number k, find a coloring of G that uses at most k colors. In this case, we no longer distinguish between two colorings that use a different number of colors, as long as both use at most k colors.

An important aspect to NP-search problems is that it is still important to find some valid solution when they exist. However, we can relax this to get NP-decision problems. Here, the goal is simply to *decide* (yes or no), whether a solution to a search problem exists; it is no longer necessary to produce it.

Some examples of problems that fit neatly into the NP-search framework (and, with minor modification, into the NP-decision framework) are the following:

[4] A graph is planar if it can be drawn in the plane so that every vertex is a point, and the edges are curves connecting their incident points, with the property that no two edges cross each other.

Hamiltonian Path/Circuit The Hamiltonian Path/Circuit problems can be viewed as search versions of the Traveling Salesman Problem. The input is allowed to be not a complete graph, but the goal is then just to decide whether the graph has any simple path/circuit visiting every vertex. The Hamiltonian Circuit problem is the following:

Input: A graph G.

Output: A simple cycle C in G which visits every vertex.

Like `TSP`, the Hamiltonian Circuit problem has the variation in which the goal is just to find simple path P in G that visits every vertex.

Partitioning sums In the partitioning sums problem, the input is a collection of numbers, and the goal is to find a way to partition the numbers into two sets so that both sets have the same sum. Formally,

Input: n integers $a_1, a_2, \ldots, a_n \geq 0$.

Output: A partition $I \subseteq \{1, 2, \cdots, n\}$ so that

$$\sum_{i \in I} a_i = \sum_{i \notin I} a_i$$

Subset sum In the subset sum problem, the input is a collection of numbers and a target sum, and the goal is to find a subset of the numbers summing to the target. Formally,

Input: n integers $a_1, a_2, \ldots, a_n \geq 0$, the target sum T

Output: A selection $I \subseteq \{1, 2, \cdots, n\}$ of the numbers so that

$$\sum_{i \in I} a_i = T$$

3-d perfect matching The 3-d perfect matching problem is a 3-dimensional version of the bipartite perfect matching problem. The input is three disjoint sets and a collection of valid triplets, where each triplet selects one item from each set. The goal is to find a collection of triplets so that each item in each set appears in exactly one triplet. Formally,

Input: Three sets X, Y, and Z, all of size n which are all disjoint. A collection $T \subseteq X \times Y \times Z$ of allowed matchings.

Output: A collection M of n elements of T so that each element of X, Y, and Z appears in exactly one triplet in M.

Satisfiability (`SAT`) In the satisfiability problem, the input is a boolean formula (*e.g.*, $x_1 \vee \overline{x_7}$), and the goal is to find a satisfying assignment to the formula. Formally,

Input: A boolean formula φ on n variables, $x_1, \ldots, x_n$.

Output: An assignment of `True` (1) or `False` (0) to each of the variables x_i so that φ evaluates to `True`.

Circuit Satisfiability (`Circuit-SAT`) The circuit satisfiability problem is similar to the satisfiability problem. The main difference is that the input is a boolean *circuit* instead of a boolean formula. The difference between a formula versus a circuit is in the topology: we can think of formulas by their parse trees—they do in fact form a tree in this way. In a circuit, we allow arbitrary acyclic graphs; in effect, a single expression can be used multiple times throughout a circuit while only being written down once. This difference won't matter much for us, but we'll specifically need circuit satisfiability later. Here is the formal specification for completeness:

Input: A boolean circuit C on n variables $x_1, \ldots, x_n$.

Output: An assignment of `True` (1) or `False` (0) to each of the variables x_i so that C evaluates to `True`.

Conjunctive Normal Form Satisfiability (`CNF-SAT`) Another variation on satisfiability, `CNF-SAT` takes as input a boolean formula φ in so-called *conjunctive normal form* (CNF). φ is in CNF if it is a conjuction (`AND`) of *clauses*, where a clause is a disjunction (`OR`) of *literals*, and a literal is either a variable (x_i) or the negation of a variable ($\overline{x_i}$). Examples are $(x \vee y \vee z) \wedge (x \vee \bar{y} \vee \bar{z})$ and $(x \vee y \vee z \vee w)$, while $(x \wedge y) \vee (z \wedge x)$, and $x \vee (y \wedge (z \vee w))$ are not CNF formulas. The formal specification of `CNF-SAT` is as follows:

Input: A CNF boolean formula φ on n variables, $x_1, \ldots, x_n$.

Output: An assignment of `True` (1) or `False` (0) to each of the variables x_i so that φ evaluates to `True`.

3-CNF Satisfiability (`3-SAT`) Yet another variation on satisfiability, `3-SAT` is just like `CNF-SAT`, except the clauses appearing in the input formula can involve at most 3 variables. The above example of $(x \vee y \vee z \vee w)$ is a CNF formula which is not a 3-CNF formula, while $(x \vee y \vee z) \wedge (x \vee \bar{y} \vee \bar{z})$ is a 3-CNF formula. The formal specification of `3-SAT` is as follows:

Input: A 3-CNF boolean formula φ on n variables, $x_1, \ldots, x_n$.

Output: An assignment of `True` (1) or `False` (0) to each of the variables x_i so that φ evaluates to `True`.

2 P versus NP

A natural question to ask is whether *any* NP-optimization (or -search or -decision) problem can be solved efficiently. This question was first asked in the 1970's[5], and, unfortunately, remains unresolved. In fact, this seemingly innocuous question is the largest open problem in theoretical computer science, known as the P versus NP problem. The notation P refers to problems which have efficient solutions—most of the problems we saw in earlier lectures are in P. It's not too difficult to see that problems with efficient solutions actually meet the definition of NP problems, and hence we say that $\text{P} \subseteq \text{NP}$. The reverse inclusion, $\text{NP} \subseteq \text{P}$, is unknown.

[5] To be more precise, it was first formalized as the P versus NP problem in the 1970's. The idea of algorithmically finding mathematical proofs significantly faster than brute force goes back a bit further, at least to Kurt Gödel.

The notoriety comes from the tremendous number of problems that fit the NP criteria. If indeed NP $\subseteq$ P, then *all* of these problems—in particular the ones for which we don't currently know solutions—have efficient algorithms. While this sounds great, the pervading conjecture is that the answer is *no*, not every NP problem has an efficient algorithm. Indeed, there are thousands of NP problems across science, math, and engineering for which no known efficient algorithms are known. Many researchers from all of these different backgrounds have tried to solve these problems without success; that this sheer breadth of experience has been unable to put so many NP problems in P is taken as the leading evidence that in fact there are problems in NP that do not have efficient algorithms.

While gloomy, there are in fact some benefits to P $\neq$ NP. Foremost is that, if P = NP, then public-key cryptography—the backbone to privacy and security in Internet communications—would be completely broken.

As it turns out, when we consider the NP problems that have arisen in practice and for which we do not know efficient algorithms, the vast majority turn out to be *equivalent*. In fact, they each individually capture the difficulty of *every* NP problem, in the sense that an efficient algorithm for any single one of these problems could be used to construct an efficient algorithm for every NP problem. Such problems are said to be NP-complete.

It just so happens that many naturally-arising NP-complete problems are actually reasonably straightforward to prove NP-complete. The key notion is that of a *reduction*, discussed in the next section. One of the main advantages to the theory of NP-completeness is that the "NP-complete" label can be interpreted to mean "unlikely to have an efficient algorithm". While P versus NP question is still unresolved, the fact that all NP-complete problems are equivalent, and the fact that experts from many disciplines have failed to find efficient algorithms for them, indicate that, even if efficient algorithms do actually exist, they will be extremely difficult to find. For most software engineering applications, this is a suitable stand-in for a mathematical proof of nonexistence.

3 Reductions

The key concept in the theory of NP-completeness is the notion of a reduction. Reductions are a way of solving one problem, A, by constructing instances of another problem, B, so that solutions to the instances of B can be used to construct a solution for A. We have seen a few reductions already, most notably the various applications of network flow. There, B is the original network flow problem, while A was the problem of interest. We took arbitrary instances of A, transformed them into network flow instances, and then dropped into a subroutine to solve the instance.

In general, we will not be so lucky as to already have an efficient algorithm for B. Instead, we imagine ourselves as having the ability to solve instances of B with lightning speed—a single time step in every case. While powerful, it is limited to solving *only* instances of B; if A is a different problem, then we will have to first figure out how to phrase our questions as instances of B in order to use this ability to solve A. We are also limited in that we don't know *how* we solve the instances of B; the ability comes as a blackbox. We refer to the role of B here as an *oracle*, and an algorithm using an oracle for B has oracle access to B.

With the notion of an oracle in hand, a *reduction* from A to B is an efficient (polynomial-time) algorithm for A that has access to an oracle for B. We write $A \leq B$ when such a reduction exists. If $A \leq B$ and $B \leq A$, then A and B are said to be *equivalent*, and write $A \equiv B$.

Our network flow examples fit this nicely: to reduce the edge-disjoint paths problem to the

maximum flow problem, we just add unit capacity to each edge in the input graph, and pass the result to the oracle; the oracle instantly returns to us a maximum flow, from which we saw how to recover a maximum set of edge-disjoint paths.

Taking intuition from the network flow example, we can see how, if B has an efficient algorithm, and A reduces to B, then A will also have an efficient algorithm. The idea is just to replace the uses of the oracle blackbox by the efficient algorithm for B. We lose the oracle's "lightning speed" trait, but replacing it with an efficient algorithm for B still yields an efficient algorithm overall, even if it isn't as efficient as the reduction with the oracle.

We can also replace the oracles in reductions by other reductions. If A reduces to B (via a reduction F), and if B reduces to C (via reduction G), then we can replace F's oracle calls by invocations of G, and get a reduction from A to C.

We combine these last two observations in the following proposition:

Proposition 1. *(Basic properties of reductions)*
The following properties hold for reductions:

1. *Let $A \leq B$. If $B \in \mathrm{P}$, then $A \in \mathrm{P}$.*
2. *Let $A \leq B$ and $B \leq C$. Then $A \leq C$.*

Proof. Both properties follow essentially the same argument.[6] We focus on the first statement. Let F be the reduction from A to B, and let G be the efficient algorithm solving B. We construct an algorithm for A by simply running F, except that, whenever an oracle query is made, G is run as a subroutine to resolve the query. Since G solves B, this algorithm is clearly correct.

To analyze its efficiency, let $T_F(n)$ and $T_G(m)$ be the running times of F and G on inputs of size n and m respectively. On inputs of size n, we know the running time of F is bounded by $T_F(n)$, so we just have to account for the extra time spent answering oracle queries. For each query made by F, we run G on the query string. This takes time $T_G(m)$, where m is the size of the query. Since F has to spend time writing down the query in the first place, we can bound m by $T_F(n)$. Moreover, F makes at most $T_F(n)$ queries. Hence the total time spent resolving queries is bounded by

$$T_f(n) \cdot T_G(T_F(n))$$

and thus the overall running time is

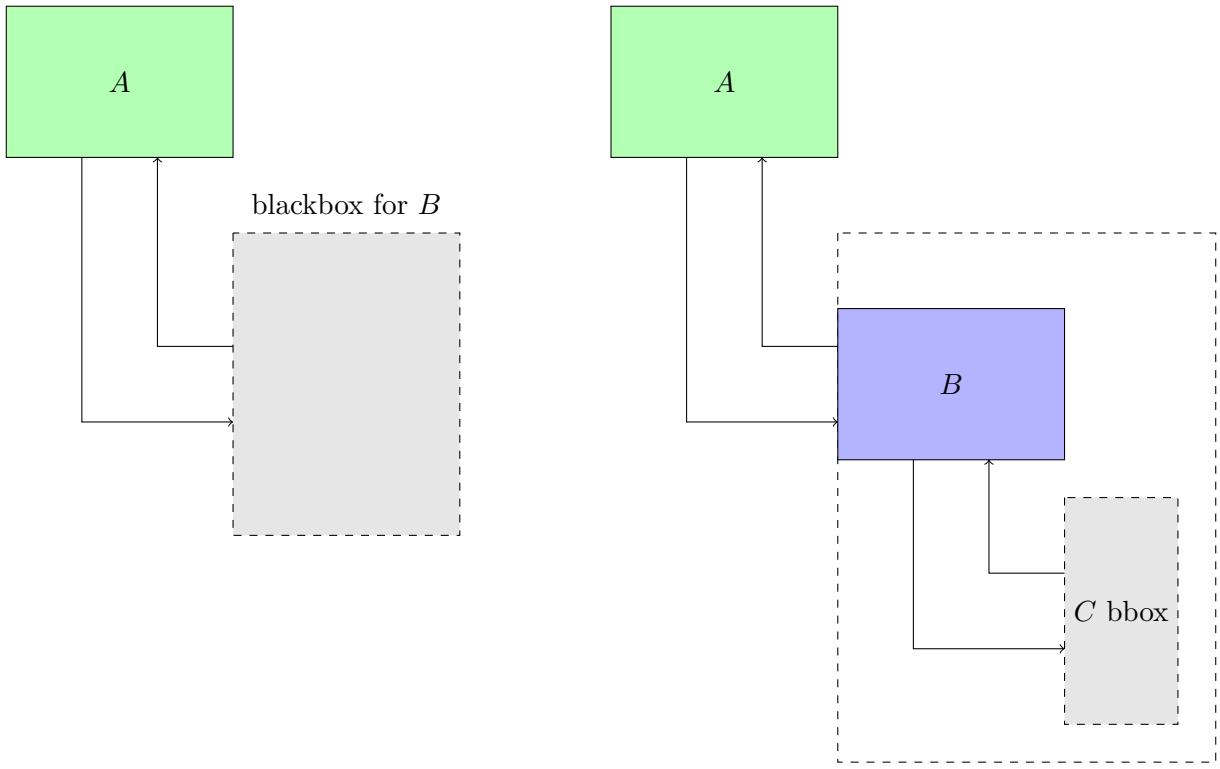
$$T_F(n) + T_F(n) \cdot T_G(T_F(n))$$

When both T_F and T_G are polynomials, this is a polynomial, and hence we can conclude $A \in \mathrm{P}$.

To handle the second statement, the same process is used. The main difference is that the algorithm G makes calls to an oracle for C, so the combined algorithm is a reduction from A to C. □

[6] In fact, the second property implies the first. Think of the decision problem in which the answer for every instance is "no solution". Let this be C. What does it mean for a problem to reduce to C?

Figure 1: Diagrammatic representation of the idea in Proposition 1



3.1 Equivalence of optimization, search, and decision

As our first new application of reductions, we will see that there is a sense in which optimization, search, and decision variants of problems are all essentially equivalent.

We focus on the independent set problem (`MIS`) as a guiding example. We gave the initial formulation of the problem as an optimization problem, `MIS`-Optimization: on input G, find a maximum-size subset I of the vertices so that I is independent.

The corresponding search version, `MIS`-Search, includes an additional parameter, k, and the question is whether there is an independent set I of size at least k. At first glance this seems like an unnatural search variant—why not just ask for some independent set, where the size is now irrelevant in the search problem? The issue is that the empty set is always an independent set; *i.e.*, solving this "more natural" independent set problem is trivial, and for reasons that completely eschew the difficulty inherent in the optimization problem. By instead phrasing the search problem as "find a solution with value at least k", we capture the difficulty of the original optimization problem; we will see this more precisely when we reduce `MIS`-Optimization to `MIS`-Search.[7]

Having fixed this search variant, the decision problem `MIS`-Decision is then straightforward: given a graph G and parameter k, simply decide whether there is an independent set in G of size at least k.

[7] In general, when given a problem that is "naturally" an NP-optimization problem, introducing an extra parameter like k above will be necessary. However, the exact conversion is generally problem-specific. The important relationship that should hold is that the optimization, search, and decision variants should all reduce to each other, in a way similar to what we are showing in this section.

We'll start with the simpler reductions.

Reducing Search to Optimization In general, optimization problems are harder than search problems. The reason is that the search problem just has to find *some* element of S_x, whereas the optimization version has to find a *particular* element of S_x. Thus we should expect that a reduction from search to optimization should be straightforward, and, indeed, it generally is.

We do `MIS` as an example. To reduce `MIS`-Search to `MIS`-Optimization, we assume an oracle for the optimization variant, and need to give an efficient algorithm for the search variant. Thus, suppose our input is G, k. We need to find an independent set of size at least k, or else report that none exist. Given our oracle, a natural thing to do is simply to ask for an independent set of maximum size; suppose it returns a size I. Now consider the following cases: either $|I| \geq k$, or $|I| < k$. In the first case, we can just return I, since it's an independent set of size at most k. In the other case, we can actually conclude that there are no independent sets of size at least k: if such a set existed, then our oracle would be wrong, which it isn't. This completes the reduction.

Reducing Decision to Search Similar to reducing search to optimization, reducing decision to search is quite easy. The search variant has to construct an element of S_x, while the decision variant merely has to decide whether or not S_x is nonempty.

We give an example reducing `MIS`-Decision to `MIS`-Search. We're given as input a graph G and parameter k, and have to decide whether there exists an independent set in G of size at least k. We have an oracle which, given a graph G and parameter k, either indicates that there is no independent set of size at least k, or else returns an independent set of size at least k. Thus we simply pass our input (G, k) to the oracle. If it says that there is no independent set of size at least k, then clearly there is no independent set of size at least k. If instead the oracle returns an independent set of size at least k, then clearly there is an independent set of size at least k. This completes the reduction.

A unifying theme in the above two reductions is that we were reducing from "easy" to "hard". This made the reductions quite easy to devise. The reductions going the other way are a little trickier, but still tame.

Reducing Optimization to Search Here, we need to find an optimal solution, given only the ability to recover *some* solution. This is where the choice in crafting the search variant of an optimization problem is important. We demonstrate this by continuing the `MIS` example.

The input is a graph G, for which we need to find a maximum-size independent set. Our oracle takes as input (G', k), and either gives an independent set in G' of size at least k, or else indicates that none exist. If we specialize $G' \leftarrow G$ in all of our queries, we can view the oracle as saying, for a particular value of k, whether G has an independent set of size at least k (and giving us one if it exists). Thus, we're interested in the largest choice of k so that the oracle returns an independent set of size at least k. We can find this with binary search. Once we find the optimal value, we can ask the oracle to provide us a set of size at least k; since k is largest subject to the existence of an independent set of size at least k, it has to return an independent set I of size k. Hence, by returning I, we return an independent set of maximum size.

Reducing Search to Decision Here, we need to *construct* a solution given only the power to test whether or not a solution exists. In general, this is a fairly problem-specific process, but there is a general technique which works for most problems. The basic idea is to build up a partial solution one piece at a time, using the decision oracle to help make choices along the way. We'll exemplify this with `MIS`.

To reduce `MIS`-Search to `MIS`-Decision, we have an oracle which, given a graph G' and parameter k', indicates whether G' has an independent set of size at least k'. We need to find, given a graph G and parameter k, and independent set I in G with size at least k, or else indicate that none exists. A good first query is just to ask the oracle with $G' \leftarrow G$ and $k' \leftarrow k$. If the oracle says "no", then there is no independent set in G with size at least k, and hence our search procedure can indicate that no solution exists.

Otherwise, we have the guarantee that G has an independent set of size at least k, and the search procedure has to find one such set. The idea is to iteratively build the solution I, starting from $I \leftarrow \varnothing$, while preserving the invariant I extends to an independent set in G of size at least k. We can do this by choosing an arbitrary vertex, v, in G, and trying to decide whether v should go into I or not. The main difficulty in making this choice is in preserving our invariant that I can extend to a solution. It's possible that every independent set of G with size at least k does not include v, and it's possible that every independent set of G with size at least k does include v. Thus we can't just "pick one and go with it", and expect it to work.

What we can do though, is try one—say leaving v out of I—and see if I still extends to an independent set of size at least k. The trick then is to rephrase the question of whether G has an independent set of size at least k that does not include v as whether some graph G' has an independent set of size at least some k'. Indeed, we can do this by using $G' \leftarrow G - v$ and $k' \leftarrow k$: clearly, G' has an independent set of size at least k' if and only if G has an independent set of size at least k that does not include v. Thus by asking the oracle about (G', k'), we learn what we need to know. There are then two cases: either the oracle says "yes" or "no". If the oracle says "yes", then there is an independent set in G of size at least k that does not include v, and so we can commit to not including v in G. If the oracle says "no", then v must go into I, so we can commit to that choice; moreover, we can commit to never putting the neighbors of v into I, since otherwise I would not be an independent set.

In either case, we commit to include or not include some of the vertices in G. What remains is to build up the rest of the independent set. We can do this recursively, by simply removing the committed vertices from G, and updating k accordingly. In particular, we'll set $G' \leftarrow G$ minus all committed vertices, and $k' \leftarrow k - a$, where we committed to including a vertices in I. Recursing on (G', k'), we get an independent set I' of size at least $k - a$. We then return the independent set consisting of all the elements of I', as well as all the vertices from G which were committed to being in I. Since every vertex from G which was committed to being in I had all its neighbors committed not to be in I, the end result is an independent set in G of size at least k, as desired.

3.2 Mapping reductions: A reduction of `CNF-SAT` to `MIS`

We now move on to a reduction between two of the problems we saw above, namely reducing `CNF-SAT` to `MIS`. We'll focus on reducing the search version of `CNF-SAT` to the search version of `MIS`. Thus the goal will be, on input a CNF formula φ, to find a satisfying assignment to φ in polynomial time taking advantage of a blackbox for `MIS`-Search.

The difficulty here is in translating the question of satisfiability of φ into a question about

maximum independent set sizes in graphs. As we will see, we can actually do this remarkably well: we can translate φ into a single graph G and compute a parameter k so that G has an independent set of size at least k *if and only if* φ is satisfiable. Moreover, there will be a simple way to translate independent sets with size at least k in G into satisfying assignments for φ.

Hence the full reduction of `CNF-SAT` to `MIS` starts with the input φ, applies the above transformation to get G and k, queries the `MIS` oracle with (G, k), getting "no solution" or an independent set I in G with size at least k. If "no solution" is returned, then we can conclude that φ is unsatisfiable; otherwise, we transform I into a satisfying assignment to φ and return that. Given the transformation above, this is a correct and efficient reduction of `CNF-SAT` to `MIS`, as desired. All we have to do now is give the transformation.

To do this, it will be helpful to have the following notations:

- Write the input as $\varphi = \bigwedge_{j=1}^{m} C_j$, where C_j denotes the j-th clause of φ, and m is the number of clauses in φ.
- Let w_j be the number of literals in C_j. (Also known as the *width* of C_j.)
- Let the variables be $x_1, x_2, \ldots, x_n$, collectively referred to as x.
- Let $C_j(a)$ for $a = 1, \ldots, w_j$ denote the a-th literal of the clause C_j.

The specific way we will do the transformation is to construct G and k so that maximal independent sets have size either k or less, and so that the independent sets of size k can be thought of as satisfying assignments to φ.

The basic principle we will exploit to achieve this is the following: let K_ℓ be the complete graph on ℓ vertices (an "ℓ-clique"). Then the maximum-size independent sets in K_ℓ have size one, and every set of vertices of size one is an independent set. Moreover, if we take a disjoint union of multiple cliques, the maximum size of an independent set is just the number of cliques in the union, and each maximum-size independent set picks exactly one vertex from each clique in the union. Maximum-size independent sets in a disjoint union of cliques then correspond to *choices*, ways to select a single vertex from each clique. Moreover, by adding in additional edges, we *invalidate* choices that pick the two endpoints of the new edge.

We use this principle in our reduction by making the following cliques:

- One 2-clique for each variable x_i, representing the choices of $x_i \leftarrow$ `True` and $x_i \leftarrow$ `False`. We call the vertices of the clique for x_i $v(x_i)$ and $v(\overline{x_i})$.
- One w_j-clique for each clause C_j, representing the choices of $a = 1, \ldots, w_j$. We call the vertices of the clique for C_j $v(C_j, a)$ for $a = 1, \ldots, w_j$.

These will be all the vertices in the graph. The maximum-size independent sets I in this graph have size $n+m$, and select $v(x_i)$ or $v(\overline{x_i})$ for each variable x_i, and select $v(C_j, a)$ for some a for each clause C_j. We will interpret a particular choice of I as selecting an *assignment* to the variables x_i, where $x_i \leftarrow$ `True` if $v(x_i) \in I$, or else $x_i \leftarrow$ `False` when $v(\overline{x_i}) \in I$. We also interpret I as selecting literals from clauses, and in particular those that *satisfy* the clause. That is, when $v(C_j, a) \in I$, we think of this as I saying that "the a-th literal of C_j is set to `True`".

Unsurprisingly, there are some conflicts in some choices of I in terms of this interpretation. If $v(x_i) \in I$, $v(C_j, a) \in I$, and $C_j(a) = \overline{x_i}$, then I is telling us first that $x_i \leftarrow$ `True`, but I is also

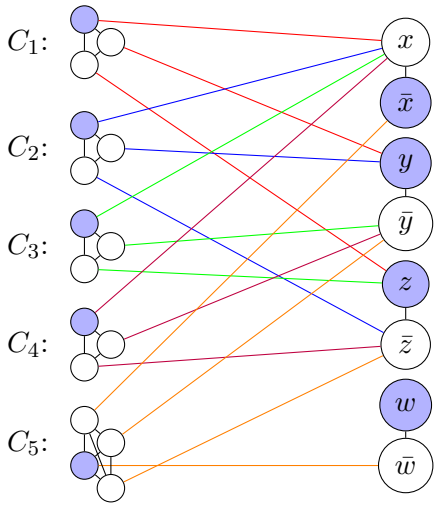
telling us that $C_j(a) = \overline{x_i} \leftarrow \texttt{True}$. Thus we should rule out these choices of I. We can do this by introducing an edge between $v(C_j, a)$ and $v(\overline{C_j(a)})$.

Adding in these edges essentially completes the transformation we want: we let G be the graph whose vertices are all those above, together with the edges introduced in the previous paragraph. (See Figure 2 for an example.) We set $k \leftarrow n + m$. Then independent sets in G of size at least k correspond with satisfying assignments to φ.

Figure 2: Reduction from `CNF-SAT` to `MIS`

$$
\begin{array}{lcll}
 & (\bar{x} \vee \bar{y} \vee \bar{z}) & \rightarrow & C_1 \\
\wedge & (\bar{x} \vee \bar{y} \vee z) & \rightarrow & C_2 \\
\wedge & (\bar{x} \vee y \vee \bar{z}) & \rightarrow & C_3 \\
\wedge & (\bar{x} \vee y \vee z) & \rightarrow & C_4 \\
\wedge & (x \vee y \vee w \vee z) & \rightarrow & C_5
\end{array}
$$

(a) A CNF formula with three variables and four clauses. It has many satisfying assignments; one is $w, x, y, z \leftarrow 1, 0, 1, 1$.



(b) The corresponding independent set instance, where $k = 5 + 4 = 9$. An assignment of size 9 is indicated by the shaded vertices, corresponding to the satisfying assignment $x, y, z, w \leftarrow 0, 1, 1, 1$.

Let's quickly prove the desired properties of this transformation from scratch:

- The first property is that, if G has an independent set of size k, then φ has a satisfying assignment. Let I be such an independent set. Then since G is a union of k cliques, I has to pick exactly one vertex from each clique. Define the assignment by $x_i \leftarrow \texttt{True}$ if $v(x_i) \in I$, and $x_i \leftarrow \texttt{False}$ if $v(\overline{x_i})$ otherwise. I has to pick one of these two vertices, so x_i gets some value. This assignment satisfies every clause C_j of φ essentially by construction: I has to pick some a so that $v(C_j, a) \in I$, and so $v(\overline{C_j(a)}) \notin I$, and hence $v(C_j(a)) \in I$, and therefore $C_j(a) \leftarrow \texttt{True}$.

- The second property is that, if φ has a satisfying assignment, then G has an independent set of size at least k. Fix a satisfying assignment to φ. Each clause C_j is satisfied, so for each $j = 1, \ldots, m$, there is a choice of a_j from $1, \ldots, w_j$ so that the a_j-th literal of C_j is set to true ($C_j(a_j) \leftarrow \texttt{True}$) in the assignment. Then construct I according to $v(x_i) \in I$ if $x_i \leftarrow \texttt{True}$, or else $v(\overline{x_i}) \in I$, and put $v(C_j, a_j) \in I$ for each $j = 1, \ldots, m$. Then I has size at least $n+m = k$. I is also an independent set: it only chooses one vertex from each of the cliques in G, and never chooses both endpoints of the edges $(v(C_j, a), v(\overline{C_j(a)}))$ by construction.

- The third property is that computing G from φ can be done in polynomial time. This clearly holds.

- The fourth and final property is that extracting a satisfying assignment to φ from an independent set I in G with $|I| \geq k$ can be done efficiently. This also clearly holds: simply follow the first of these bullet points.

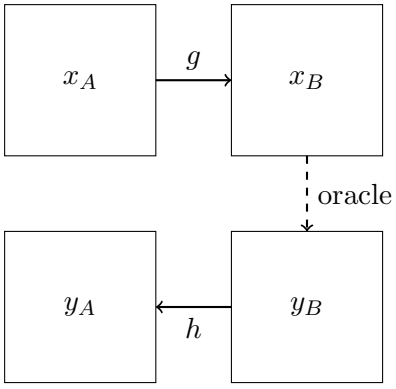
Mapping reductions The above reduction from `CNF-SAT` to `MIS` had a nice structure to it. It consisted of translating each instance of `CNF-SAT` into a single instance of `MIS`, such that the witnesses for `MIS` could be pulled back to witnesses for `CNF-SAT`. These reductions have nice properties that make them useful in theoretical computer science; we will refer to them as *mapping reductions*, and define them formally as follows:

A *mapping reduction* of the search problem A to the search problem B (written $A \leq_m B$) is the following:

- A polynomial-time computable mapping $g : x_A \mapsto x_B$ that sends instances x_A of A to instances x_B of B, so that S_{x_A} is nonempty iff S_{x_B} is nonempty. (*i.e.*, x_A has a witness iff x_B has a witness.)
- A polynomial-time computable mapping $h : x_A, y_B \mapsto y_A$ that transforms the witness y_B for x_B into a witness y_A for x_A.

A simple fact is that if $A \leq_m B$, then $A \leq B$, hence the use of "reduction" in the term mapping reduction. This is because we can solve A with oracle access to B by, on input x_A, using g to compute x_B, using the oracle to get y_B, and then using h to get a solution y_A for x_A. Diagrammatically:

Figure 3: Mapping reduction diagram



We can similarly define mapping reductions for decision problems by only insisting on the first property, and define mapping reductions for optimization problems by imposing a reasonable condition on how the objective value of y_B relates to the objective value of y_A.

Mapping reductions satisfy similar properties as those stated in Proposition 1:

Proposition 2. *(Basic properties of mapping reductions)*
The following properties hold for mapping reductions:

1. *If $A \leq_m B$, then $A \leq B$.*
2. *If $A \leq_m B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.*

3. *If* $A \leq_m B$ *and* $B \leq_m C$*, then* $A \leq_m C$*.*

The arguments are essentially identical to before.

4 NP-completeness: universal problems for NP

4.1 NP-completeness, definition and motivation

Proving that `CNF-SAT` $\leq_m$ `MIS` tells us that `CNF-SAT` is no harder than `VC`. We can prove the other direction as well, but there is in fact a much more general phenomenon at play here. As it turns out, `CNF-SAT` has the property that *every* problem in NP—even the ones we have yet to formulate!—reduce to it. That `MIS` $\leq$ `CNF-SAT` follows as a direct corollary of this. Formally, we will prove the following theorem, which will be the starting point of our presentation of the theory of NP-completeness:

Theorem 1. *(Existence of* NP*-hard problems)*
For every NP*-search problem* A*,* $A \leq_m$ `Circuit-SAT`*.*

Proof. Let's now prove Theorem 1. The goal is to show that for every NP-search problem A, $A \leq_m$ `Circuit-SAT`. So we fix an NP-search problem A. This means we get the verifier V and a natural number d so that each input x has a set of witnesses S_x, where each witness y has length $|y| \leq |x|^d$. The goal is to transform each input x into a boolean circuit C so that C has a satisfying assignment if and only if S_x is nonempty.

Our high-level strategy for achieving this is to, on input x, create C from V so that C has $|x|^d$ input bits, and accepts a string y as `True` if and only if y is a valid witness for x (*i.e.*, , $y \in S_x$). In other words, we'll think of $C(y)$ (the behavior of C when given input y) as just determining whether or not $V(x, y)$ accepts.

In this way we will get a mapping reduction from A to `Circuit-SAT`: the mapping g sends the input x to the circuit C, and the mapping h pulls satisfying assignments y to C directly back to the witness y for x. h is clearly polynomial-time computable (it just copies its input), so we really only need to specify g.

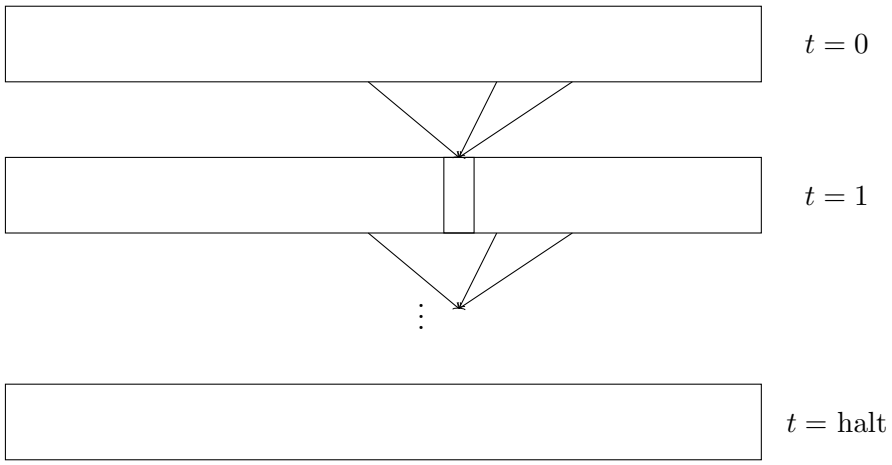
To do this, we will appeal to the following lemma:

Lemma 1. *(Circuits can simulate algorithms)*
Given an input length n*, we can construct from* V *in time polynomial in* n *a circuit* $C_n(x, y)$ *on* $n + n^d$ *inputs such that for every* $x \in \{0,1\}^n$ *and every* $y \in \{0,1\}^{n^d}$*,* $C_n(x, y) = V(x, y)$*.*

Using this lemma, we can specify g as follows. On input x, compute the circuit $C_{|x|}$ as determined by the lemma. Then hardcode the first $|x|$ many bits of $C_{|x|}$ to equal x, and obtain the circuit C that we desire. This works: to evaluate $C(y)$, we see this is equal to computing $C_{|x|}(x, y)$, which, by Lemma 1, equals $V(x, y)$, exactly as we need.

Modulo the proof of Lemma 1, we have proven Theorem 1. Unfortunately, we won't formally prove Lemma 1; it requires more carefully formalizing the notion of an algorithm, using, *e.g.*, , Turing machines. That being said, we can sketch a rough idea for the proof. Intuitively, every algorithm is a fixed list of simple instructions. On a given input, these instructions, with some auxiliary memory, make some step-by-step changes to the memory, and eventually halt. The key insight behind Lemma 1 is to represent the execution of a program on its input as a sequence of snapshots, one for each step of the algorithm. Crucially, each bit of a snapshot depends on only

Figure 4: Pictorial proof of Lemma 1



the *previous* snapshot, and no other information. Moreover, since each step of the algorithm is a simple instruction. Thus we can compute each bit of a snapshot as just a boolean circuit with inputs from the preceding snapshot. By piecing all these together, we get a boolean circuit which computes the same function as the algorithm we are simulating. The size of this circuit is roughly the running time squared, which is still polynomial in the input size, since V runs in polynomial time. □

Problems K for which Theorem 1 holds (*i.e.*, , for every A in NP, $A \leq K$) are said to be NP-*hard*. If K is in NP and is NP-hard, then we say K is NP-*complete*. Hence we can rephrase the above theorem as stating that `Circuit-SAT` is NP-hard. Since `Circuit-SAT` is in NP, we can further say that `Circuit-SAT` is NP-complete.

Using Proposition 1, we can make the following observations:

- NP-complete problems are, in a certain very precise sense, the *hardest* problems in NP. The specific sense is this: there exists an NP-complete problem with an efficient algorithm if and only if every problem in NP has an efficient algorithm (*i.e.*, , P = NP).

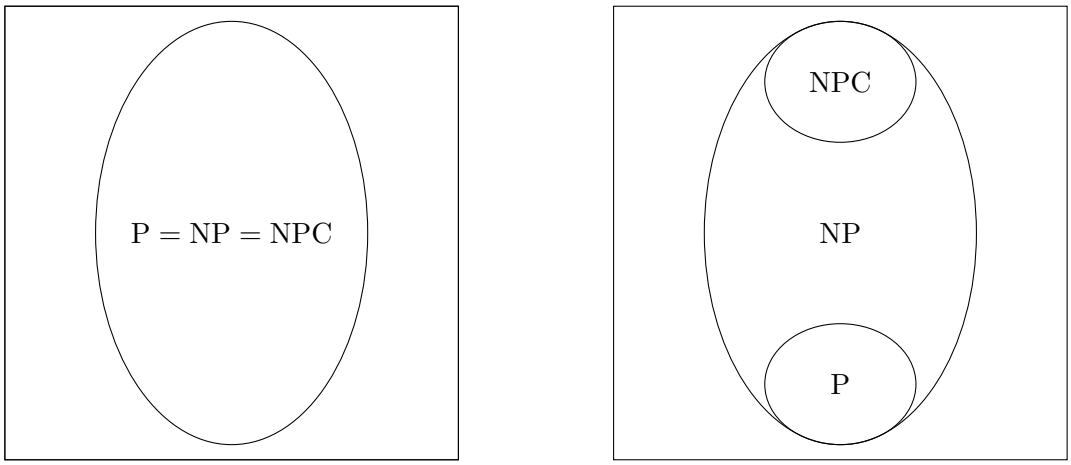
  This follows from the first point in Proposition 1: letting K be an NP-complete problem with an efficient algorithm, it follows from the proposition that if A is an NP problem, then $A \leq K$, and so A has an efficient algorithm.

  A notable special case of this observation is that if any single NP-complete problem has an efficient algorithm, then so too does *every* NP-complete problem. *i.e.*, , either they are all efficiently solvable, or else none are efficiently solvable. We can thus represent the two possible worlds as follows:

- Following the first bullet point, we have that P = NP if and only if there exists an efficient algorithm for some NP-complete problem; taking contrapositives, we have P ≠ NP if and only if every NP-complete problem has no efficient algorithm.

  Since it is widely conjectured in the theoretical computer science community that P ≠ NP, we can view NP-hardness as a sufficient condition that a problem has no efficient algorithm.

Figure 5: Two possibilities: P = NP, or P ≠ NP



(a) P = NP (b) P ≠ NP

The fact that there are many natural NP-complete problems, and that many researchers from many diverse backgrounds have been unable to find efficient algorithms for any single one of them, is the primary evidence that P ≠ NP.

- Continuing the second observation, it becomes desirable to show that certain seemingly-hard problems are NP-hard. Re-proving Theorem 1 for each particular problem would be a major pain. However, by considering Proposition 1, we see that if a particular problem B has $K \leq B$ where K is NP-hard, then B is also NP-hard.

  While a superficially simple observation, this carries with it a lot of power. If we want to show that some other problem is NP-hard, then we no longer have to reduce from every problem in NP; instead, we just have to give a single reduction from `Circuit-SAT`.

  This strategy turns out to be quite effective: there are thousands of natural NP-hard problems, and nearly all of them have been proven NP-hard via a sequence of reductions from `Circuit-SAT`. In fact, most of these reductions are actually mapping reductions.

  In a bit, we will prove the following fact: `Circuit-SAT` $\leq_m$ `CNF-SAT`. This proves that `CNF-SAT` is NP-hard. It additionally implies that `MIS` is NP-hard, since we have already shown that `CNF-SAT` $\leq_m$ `MIS`. Note how much more concrete and attainable this is than reducing from an arbitrary NP-hard problem as in Theorem 1.

- Suppose that we have a particular problem B which is in NP, and we believe it to be NP-hard. Then we should find some convenient NP-hard problem A and prove that $A \leq B$.

  Usually, it's wise to choose A to be NP-complete, rather than just any NP-hard problem. This is because NP-complete problems are the easiest NP-hard problems: if a problem is NP-hard, then every NP-complete problem reduces to it. This isn't a major point—most convenient choices of A are already NP-complete—but is still worth mentioning.

  There is still the question of which choice of A to make. In general, different choices of A can make finding a reduction $A \leq B$ considerably easier or harder. Finding a good problem is a

bit of an art, but can be learned with practice. We will soon give a list of problems which are NP-complete and which are common candidates from which to start a reduction.

4.2 NP-complete problems

In this section, we'll list some of the most common starting points for NP-hardness reductions. We summarize them in Theorem 2. We'll give reductions for some of them in later sections. All of these problems are defined above, in Section 1.

Theorem 2. *(List of basic* NP*-complete problems)*
The following problems are NP*-complete:*[8]

- *Satisfiability (*`SAT`*), including* `Circuit-SAT`*,* `CNF-SAT`*, and* `3-SAT`
- *Independent Set (*`MIS`*)*
- *Vertex Cover (*`VC`*)*
- *Hamiltonicity (cycle and path, directed and undirected versions)*
- *Traveling Salesman (*`TSP`*) (cycle and path versions)*
- *Graph 3-Coloring (A variation on graph-coloring, where $k = 3$ on all inputs.)*
- *3-D Perfect Matching*
- *Subset Sum*
- *Partitioning Sums*
- *General Knapsack*

Theorem 1 gives us that `Circuit-SAT` is NP-hard. We'll give a reduction `Circuit-SAT` $\leq_m$ `CNF-SAT` in the next section, and observe that all the clauses resulting in this reduction have width three, and hence that it is really a reduction `Circuit-SAT` $\leq_m$ `3-SAT`. `SAT` contains `3-SAT` as a special case, so this proves the first bullet point. We have already shown `CNF-SAT` $\leq_m$ `MIS`, giving the second point. That `MIS` $\leq_m$ `VC` follows from the connection between independent set and vertex cover discussed after defining `VC`. We won't prove the hardness of Hamiltonicity, but there are reasonable reductions from both `3-SAT` and `VC`. We leave the reduction from Hamiltonicity to `TSP` as a simple exercise. In Section 4.2.2, we will give a reduction from `3-SAT` to graph three coloring. We will not prove the hardness of 3-D perfect matching, but there is a reasonable reduction from `3-SAT`. The hardness of subset sum will follow from a reduction given in Section 4.2.3, which we following with a reduction from subset sum to partitioning sums. We leave as a simple exercise the problem of reducing subset sum or number partitioning to the general knapsack problem.

There are thousands of other problems, each with different variations and restrictions, that are all NP-complete. In fact, the list of naturally-arising NP problems not known to be in P and not known to be NP complete is rather short—there are anywhere from a handful to one or two dozen, depending on how loosely "natural" is interpreted. Here are a couple that make almost every list:

[8] There are many, many other NP-complete problems. The most popular reference is *Computers and Intractability* by Garey and Johnson.

- **Factoring integers**. The problem is to, given an integer n in binary, to present a prime factorization of n. The hypothesis that this problem is not in P is the foundation for the security in the RSA public-key cryptography system.

- **Graph Isomorphism**. The input to the graph isomorphism problem is two graphs, G and H. The problem is to decide whether there is a bijection f between the vertices of G and the vertices of H so that (v, u) is an edge of G if and only if $(f(v), f(u))$ is an edge in H. (Such a bijection is known as an isomorphism between G and H.)

4.2.1 NP-completeness of `CNF-SAT` and `3-SAT`

In this section, we give a reduction from `Circuit-SAT` to `3-SAT`, establishing the NP-hardness of `3-SAT`. All of our later reductions will start from `3-SAT` or a problem that started from `3-SAT`, so this is a crucial technical step. To show the reduction, we start with a boolean circuit C on n variables $x_1, \ldots, x_n$, and need to give a 3-CNF formula φ which is satisfiable if and only if C is satisfiable.

We will do this by making φ as follows: for each logic gate g in C, create a variable y_g for φ. We think of an assignment to the y_g's as representing the entire evaluation of C on some input, where the gate g in C outputs the value stored in y_g.

Not all of these assignments are valid: for instance, suppose g_1 and g_2 are two gates feeding into $g_{\texttt{out}}$, which is an `AND` gate. The assignment with $y_{g_1} \leftarrow 0$, $y_{g_2}, \leftarrow 0$, and $y_{g_{\texttt{out}}} \leftarrow 1$ does not correspond to a valid evaluation of C. We can eliminate this with the clause $(y_{g_1} \vee y_{g_2} \vee \overline{y_{g_{\texttt{out}}}})$.

In general, we can eliminate invalid assignments in a gate-by-gate fashion by introducing a few clauses that remove assignments that aren't consistent with each gate. To be specific, for each gate g of C, we introduce the following clauses:

- If g is an input gate, we introduce no new clauses. y_g is allowed to take on any value since it represents the value of an input to C, which could be anything.

- If g is a `NOT` gate with input g', we need to force that $y_g = \neg y_{g'}$, and so we introduce the clauses $(y_g \vee y_{g'})$ and $(\overline{y_g} \vee \overline{y_{g'}})$.

- If g is an `AND` gate with inputs g_1 and g_2, we need to force $y_g = y_{g_1} \wedge y_{g_2}$. Hence we introduce the clauses

$$(y_g \vee \overline{y_{g_1}} \vee \overline{y_{g_2}}) \qquad (\overline{y_g} \vee y_{g_1} \vee y_{g_2})$$
$$(\overline{y_g} \vee y_{g_1} \vee \overline{y_{g_2}}) \qquad (\overline{y_g} \vee \overline{y_{g_1}} \vee y_{g_2})$$

- If g is an `OR` gate with inputs g_1 and g_2, we need to enforce $y_g = y_{g_1} \vee y_{g_2}$. Hence we introduce the clauses

$$(\overline{y_g} \vee y_{g_1} \vee y_{g_2}) \qquad (y_g \vee \overline{y_{g_1}} \vee \overline{y_{g_2}})$$
$$(y_g \vee \overline{y_{g_1}} \vee y_{g_2}) \qquad (y_g \vee y_{g_1} \vee \overline{y_{g_2}})$$

By introducing clauses of this form for every gate in C, we enforce that the only satisfying assignments to ϕ are those which correspond to a valid way of plugging some assignment into C and evaluating C.

To complete the reduction we need to do one more step: make sure that ϕ only accepts *satisfying* assignments to C. Let $g_{\texttt{out}}$ be the output gate of C. Then we want to eliminate assignments to the y_g's which set $y_{g_{\texttt{out}}} \leftarrow 0$. We can do this by introducing a final clause, $(y_{g_{\texttt{out}}})$.

The end result of this reduction is a conjunction ϕ of CNF clauses of width at most three. ϕ is satisfiable if and only if C is satisfiable. Moreover ϕ can be computed from C in polynomial time, by simply iterating over the gates in C and writing down the corresponding clauses. Thus we have a mapping reduction from `Circuit-SAT` to `3-SAT`.

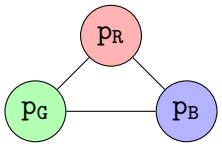
4.2.2 NP-completeness of graph 3-coloring

In this section, we give a reduction from `3-SAT` to graph 3-coloring, proving the NP-hardness (and thus NP-completeness) of the latter. The approach is structurally similar to the reduction from `CNF-SAT` to `VC`, in that we construct some "gadgets" that help us encode the starting CNF formula in a piece-by-piece fashion.

We'll start with a gadget that helps to simplify some of our later arguments. For any solution to a graph 3-coloring instance, we can permute the colors (*e.g.*, , swap all occurrences of red and blue) and get another valid 3-coloring. Breaking this symmetry will help make the later arguments more concrete.

One way to do this is to note that the complete graph on three vertices, K_3, can only be 3-colored by coloring each of the vertices a different color. Thus we can include a copy of K_3, and think of it as giving us a color palette in the following way: Suppose the vertices of K_3 are $\{\texttt{p}_\texttt{R}, \texttt{p}_\texttt{B}, \texttt{p}_\texttt{G}\}$. Then we will call "`Red`" whatever is the color of vertex $\texttt{p}_\texttt{R}$, "`Blue`" whatever is the color of vertex $\texttt{p}_2$, and "`Green`" whatever is the color of vertex $\texttt{p}_\texttt{G}$.

Figure 6: The color palette



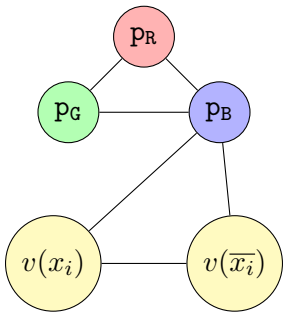
Having decided this, we can use the color palette to force another vertex v to "not be `Red`" or "be `Blue`" by adding the right edges between the palette and v. For example, to force v to "not be `Red`", we would just include an edge between vertex $\texttt{p}_\texttt{R}$ of the palette and v; to force v to "be `Blue`", we would just include an edge between vertices $\texttt{p}_\texttt{R}$ and v and between vertices $\texttt{p}_\texttt{G}$ and v. Since adjacent vertices can't have the same color, this means v can't be colored `Red` in the first case, and can't be colored `Red` or `Green` in the second case (and hence must be colored `Blue`).

Let's now move on to the variable gadgets. Here, the goal is to make a simple graph that can be colored in essentially only two ways, each corresponding to a different assignment of some boolean variable. For the sake of notation, suppose x_i is the boolean variable for which we are building the gadget. A natural way of representing x_i in the 3-coloring problem to just to have a vertex, say $v(x_i)$, and to force it to only have two colors, where each color corresponds to an assignment to `True` or `False` to x_i. Using the color palette, we can say that `Red` is `False`, `Green` is `True`,

and `Blue` will not ever be the color given to a variable. We can enforce this last condition by just adding an edge between $\mathtt{p_B}$ and $v(x_i)$.

We will also need to have the complementary literal $\overline{x_i}$ represented, which we can do by introducing another vertex denoted as $v(\overline{x_i})$, and including an edge between it and $\mathtt{p_B}$. We will want to enforce that $\overline{x_i}$ gets a different value than x_i, which we can again do by adding an edge between $v(x_i)$ and $v(\overline{x_i})$. Altogether, our vertex gadgets have the following structure:

Figure 7: A gadget for x_i and $\overline{x_i}$



The color yellow is used to indicate "`Red` or `Green`" (*i.e.*, "not `Blue`"). Not all vertices colored yellow have to be the same color. In fact, in this picture, they have to be different colors.

By making n copies of this gadget (using the same palette for each of them), we get vertices $v(x_i)$ and $v(\overline{x_i})$ for $i = 1, \cdots, n$ so that every valid 3-coloring of the graph-so-far has $v(x_i)$ and $v(\overline{x_i})$ colored either `Red` or `Green` (and they are opposite each other), for every $i = 1, \cdots, n$. We are interpreting a color which assigns $v(x_i)$ the color `Green` to correspond to the assignment which sets $x_i \leftarrow$ `True`, and the opposite for `Red`.

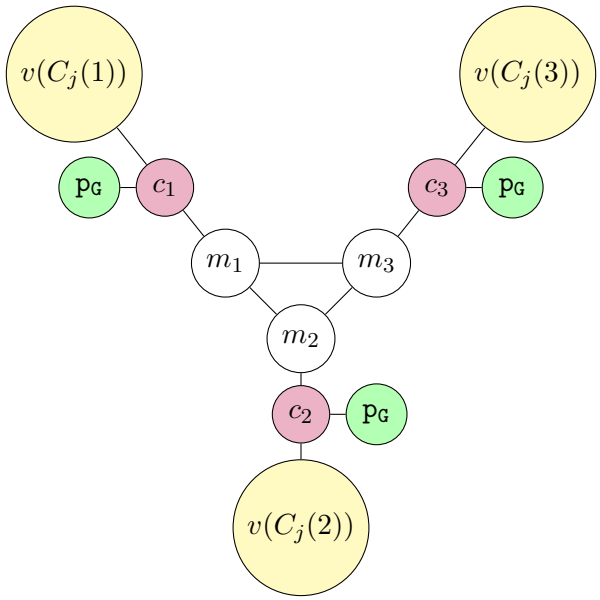
This suffices for the variable gadgets; now we just need to put restrictions on which colorings will be allowed according to which assignments satisfy the clauses. Let $C_1, C_2, \ldots, C_m$ be the clauses of the input 3-CNF formula, and let $C_j(a)$ refer to the a-th literal of C_j, for $a = 1, 2, 3$. (This is as in the reduction from `CNF-SAT` to `VC`.) We will focus on handling a single clause C_j at a time.

In making the clause gadget for C_j, we need to rule out exactly the assignments for which $C_j(1), C_j(2), C_j(3) \leftarrow$ `False`. Translating this to our 3-coloring construction, we need to rule out exactly the 3-colorings of our graph-so-far which have $v(C_j(1)), v(C_j(2)), v(C_j(3)) \leftarrow$ `Red`. There are a number of ways of doing this; Figure 8 gives one such way. The text in the figure explains how precisely the gadgets goes about doing this, but the important point is that a 3-coloring of our palette and variable gadgets corresponds to an assignment that satisfies C_j if and only if the 3-coloring can be extended to a 3-coloring of the clause gadget.

Thus by adding in all the gadgets for each of the clauses, we eliminate exactly the colorings which correspond to unsatisfying assignments. Since every coloring pre-clause-gadgets corresponds to some assignment, it follows that the post-clause-gadgets graph is 3-colorable if and only if the original 3-CNF formula is satisfiable.

It is easy to verify that this construction can be done in polynomial time, and that the valid 3-colorings of the constructed graphs can be efficiently pulled back to satisfying assignments to the original 3-CNF formula. Hence we have a mapping reduction of `3-SAT` to graph 3-coloring.

Figure 8: Clause gadgets



Purple represents "`Red` or `Blue`", yellow represents "`Red` or `Green`", and white represents "any color". The occurrences of $v(C_j(a))$ for $a = 1, 2, 3$ and of $\texttt{p}_\texttt{G}$ are the corresponding pre-existing vertices, whereas $m_1, m_2, m_3, c_1, c_2, c_3$ are all fresh vertices for this clause gadget. If every yellow vertex is colored `Red`, then each purple vertex must be colored `Blue`, and so none of the white vertices can be colored `Blue`, which means no 3-coloring can exist. On the other hand, if at least one of the yellow vertices is `Green` (say $v(C_j(a))$), then c_a can be colored `Red`, the remaining purple vertices may be colored `Blue`, m_a may be colored `Blue`, and the remaining white vertices may be colored either `Red`,`Green` or `Green`,`Red`.

4.2.3 NP-completeness of subset sum and number partitioning

To show the NP-completeness of subset sum, we will give a mapping reduction from `3-SAT`. As before, we will do the reduction in steps, first representing all satisfiable assignments, and then pruning out the unsatisfying ones.

The first order of business is representing all possible assignments. Let the variables in our given `3-SAT` instance be $x_1, \ldots, x_n$. We will introduce $2n$ numbers, $a_1, a_2, \ldots, a_{2n}$ and set a target sum T with the following property: For every set $I \subseteq \{1, \cdots, 2n\}$ with $\sum_{k \in I} a_k = T$, and every $i = 1, \cdots, n$, precisely one of $2i-1$ or $2i$ is in I. We can use this to represent assignments by saying $x_i \leftarrow \texttt{True}$ if $2i - 1 \in I$, or else $x_i \leftarrow \texttt{False}$ if $2i \in I$.

To pick $a_1, \ldots, a_{2n}$ and T, it will be helpful to think of them not as integers written in base 10, but rather as integers written in base 2. In particular, we will set a_{2i-1} and a_{2i} to be the integer whose binary expansion is all zeroes, except for being `1` in the i-th position. We will choose T to be the integer whose first n bits are set to `1`, with the rest set to zero. Visually:

$$\begin{array}{rcccc} a_1 & : & 0 & \cdots & 0 & 1 \\ a_2 & : & 0 & \cdots & 0 & 1 \\ a_3 & : & 0 & \cdots & 1 & 0 \\ a_4 & : & 0 & \cdots & 1 & 0 \\ & & & \vdots & & \\ a_{2n-1} & : & 1 & \cdots & 0 & 0 \\ a_{2n} & : & 1 & \cdots & 0 & 0 \\ \hline T & : & 1 & \cdots & 1 & 1 \end{array}$$

This construction has the desired property we want. To prove this, let $I \subseteq \{1, \cdots, 2n\}$ with $\sum_{i \in I} a_i = T$, and consider the first bit. $a_3, a_4, \ldots, a_{2n}$ all have this set to zero, so they have no affect on the first bit of the sum. Since T has a 1 as its first bit, an odd number of a_1 or a_2 has to be selected by I, and hence exactly one of them is selected. Continuing on to the second bit, we know that exactly one of a_1 and a_2 is selected by I, and so they cannot influence the second bit in the sum $\sum_{i \in I} a_i$. It follows from the same reasoning as above that then exactly one of a_3 or a_4 is selected by I. Continuing in this fashion, we get that for each $i = 1, 2, \ldots, n$, exactly one of a_{2i-1} or a_{2i} is selected by I.

Now it remains to eliminate choices of I corresponding to unsatisfying assignments. As before, we will do this one clause at a time. Let's focus on the first clause, C_1. C_1 involves three variables, x_{i_1}, x_{i_2}, and x_{i_3}. Let k_1 be either $2i_1 - 1$ or $2i$, according to whether x_{i_1} appears as itself or appears as its negation in C_1. The idea is that k_1 is in I if and only if I corresponds to an assignment for which the occurrence of x_{i_1} in C_1 (as itself or its negation) satisfies C_1. Concretely, if $C_1 = (x_1 \vee x_2 \vee x_3)$, then $k_1 = 2 \cdot 1 - 1 = 1$, but if $C_1 = (\overline{x_1} \vee x_2 \vee x_3)$, then $k_1 = 2 \cdot 1 = 2$. We can similarly define k_2 and k_3 with respect to x_{i_2} and x_{i_3} respectively. The goal is to eliminate choices of I which have none of k_1, k_2, or k_3.

We can do this by extending T and a_{k_1}, a_{k_2}, and a_{k_3}, as well as adding two new numbers, a_{2n+1} and a_{2n+2} as follows. a_{k_1}, a_{k_2}, and a_{k_3} get their $(n+1)$-th bits set. a_{2n+1} and a_{2n+2} have only one bit set, specifically the $(n+1)$-th bit. Finally, T is extended to have its $(n+1)$-th and $(n+2)$-th bits set. Extending our previous table, we get the following:

$$\begin{array}{rcccc} a_{k_1} & : & 0 & 1 & \cdots \\ a_{k_2} & : & 0 & 1 & \cdots \\ a_{k_3} & : & 0 & 1 & \cdots \\ a_{2n+1} & : & 0 & 1 & \cdots \\ a_{2n+2} & : & 0 & 1 & \cdots \\ \hline T & : & 1 & 1 & \cdots \end{array}$$

where the $\cdots$ includes all of the bits that were set before (or zeros in the case of a_{2n+1} and a_{2n+2}). Arguing as we did before, one can still show that the only way to choose $I \subseteq \{1, \cdots, 2n+2\}$ so that $\sum_{i \in I} a_i$ agrees with T on the first n bits is for I to choose precisely one of $2i - 1$ or $2i$, for every $i = 1, \ldots, n$, and possibly choose $2n+1$ and/or $2n+2$. This shows that we haven't broken the structure we built up previously, but also shows that the only choices influencing the $(n+1)$-th and $(n+2)$-th bits of $\sum_{i \in I} a_i$ are whether or not each of $k_1, k_2, k_3, 2n+1, 2n+2$ is in I. So now note that the only way to get $\sum_{i \in I} a_i$ to agree with T on the first $n+2$ bits is to choose precisely three of $k_1, k_2, k_3, 2n+1, 2n+2$ to be in I. The only way to do this is to pick at least one of k_1, k_2, k_3, and then enough of $2n+1, 2n+2$ to get up to three choices. Since we are interpreting $k_a \in I$ to mean the a-th literal of C_1 is set to `True`, it follows that the only way to pick $I \subseteq \{1, \cdots, 2n+2\}$

so that $\sum_{i \in I} a_i = T$ is to choose an assignment to $x_1, \ldots, x_n$ that satisfies C_1. Thus we have taken care of C_1.

We can continue this process on to C_2. Note that in the construction in the previous paragraph, the only way to choose I so that $\sum_{i \in I} a_i$ agrees with T on the first $2n+2$ bits is for it to also have `0` as the higher bits. Thus we still don't have to worry about carries, and the process goes through: we add two more `1`'s to T, add a_{2n+3} and a_{2n+4} to have a single `1` in the $(n+3)$-th position, and to add a `1` in the $(n+3)$-th position to each $a_{k_1}, a_{k_2}, a_{k_3}$, where now k_1, k_2, k_3 are defined according to C_2. The end result is the same as before, where now solutions I with $\sum_{i \in I} a_i = T$ correspond exactly with assignments to $x_1, \ldots, x_n$ that satisfy C_1 and C_2. We can continue this process for each of the clauses $C_3, C_4, \ldots, C_m$, at which point the construction of all the clause gadgets is complete.

The end result is a collection of $2n + 2m$ numbers and target sum T with the property that every subset $I \subseteq \{1, \cdots, 2n + 2m\}$ for which $\sum_{i \in I} a_i = T$ corresponds to an assignment satisfying every clause $C_1, \ldots, C_m$. The construction of $a_1, \ldots, a_{2n+2m}$ and T can be easily done in time polynomial in n and m when we represent these numbers in binary. Moreover, the process of recovering satisfying assignments from solution sets I can be done efficiently as well, and therefore we have a mapping reduction from `3-SAT` to subset sum.

It's worth pointing out the importance of our use of binary as a means of encoding the input numbers. While more complicated, we could just as well use ternary, or other bases larger than one. On the other hand, a unary representation would lead to an exponential-time algorithm—note that T is just a sequence of $2n + 2m$ `1`'s in binary, and hence its value is $2^{2n+2m} - 1$. In some sense we should expect this, though: we saw earlier a pseudopolynomial-time algorithm for the knapsack problem which also works (with slight modification) for subset sum. If we use a unary encoding of the input numbers, this pseudopolynomial algorithm is an algorithm which runs in time polynomial in the bit-length of the input. Thus, if there were a polynomial-time reduction from `3-SAT` to a unary-encoded subset sum, then in fact we would be able to solve `3-SAT` in polynomial time and prove P = NP!

Reducing subset sum to number partitioning Having just shown the NP-completeness of subset sum, we give a mapping reduction from subset sum to number partitioning to show that the latter is NP-complete.

Let's first take stock of what we have to do. In subset sum, we have $a_1, \ldots, a_n$ and T as above, and the goal is to find a set $I \subseteq \{1, 2, \cdots, n\}$ so that $\sum_{i \in I} a_i = T$. In the number partitioning problem, we have $b_1, \ldots, b_m$, and the goal is to find a set $J \subseteq \{1, 2, \cdots, m\}$ so that $\sum_{j \in J} b_j = \sum_{j \notin J} b_j$. We can rewrite the latter as requiring $\sum_{j \in J} b_j = \frac{1}{2} \sum_j b_j$.

Note that proving the reverse reduction is quite easy. Given an instance of the number partitioning problem, we construct a new instance of subset sum in which $a_i \leftarrow b_i$, and $T \leftarrow \frac{1}{2} \sum_j b_j$. It's easy to see that the solution sets I in the constructed subset sum instance are then identical to the solution sets J in the number partitioning problem. To prove the NP-hardness of the number partitioning problem, however, the reduction has to go the other way, even though it's a bit trickier.

The trick isn't too difficult though. At a basic level, we want to use the same numbers as in the subset sum instance, so we set $b_i \leftarrow a_i$ for $i = 1, 2, \cdots, n$. We also want to include some additional numbers $b_{n+1}, b_{n+2}, \ldots, b_m$ for some m so that subsets of the a_i's that sum to T can be extended to subsets of all the b_i's that sum to half the total sum of the b_i's, and vice-versa. Intuitively, the extra b_j's we add pad the underlying subset sum solutions to meet the stricter structure of the

number partitioning problem.

Let's try to get away with just two b_j's, whose values we refer to as X and Y. The idea is that if I is a solution to the original subset sum instance, then I together with X will be a solution to the constructed number partitioning problem. To realize this, suppose we can set X and Y so that

1. the sum of all the numbers including X and Y is equal to $2(T + X)$, and
2. X and Y cannot both be on the same side of a valid solution to the constructed number partitioning problem.

Then simply setting $b_{n+1} \leftarrow X$ and $b_{n+2} \leftarrow Y$ gives us the reduction from subset sum to number partitioning. Here is the proof: Suppose that $I \subseteq \{1, 2, \ldots, n\}$ is a solution to the original subset sum instance, *i.e.*, $\sum_{i \in I} a_i = T$. Then $J \leftarrow I \cup \{n + 1\}$ has

$$\sum_{j \in J} b_j = T + X = \frac{1}{2} \sum_j b_j$$

Conversely, suppose $J \subseteq \{1, 2, \cdots, n + 2\}$ is a solution to the constructed number partitioning instance, *i.e.*, $\sum_{j \in J} b_j = \sum_{j \notin J} b_j$. By possibly exchanging J with its complement, we can assume that $n + 1 \in J$. We know that then $n + 2$ is not in J, since X and Y cannot be on the same side of a valid solution. We also know that $\sum_{j \in J} b_j = T + X$, so, subtracting X from both sides and writing $I \leftarrow J \setminus \{n + 1\} \subseteq \{1, 2, \cdots, n\}$, we get

$$T = \sum_{i \in I} b_i = \sum_{i \in I} a_i$$

as desired. So all we have to do now is find values of X and Y that work.

One way to do this is to set $X = Z - T$ and $Y = (Z - S) + T$, where $S = \sum_i a_i$. The resulting number partitioning instance has total sum $2 \cdot Z$, and so each partition in a valid solution has to sum to Z. It follows that any choice of Z satisfies (1), leaving us the freedom to choose Z to satisfy (2). In order to satisfy (2), it suffices to have $X + Y > Z$, and so $2Z - S > Z$, and so $Z > S$. Thus we can choose $Z = S + 1$ to satisfy (2).

Thus the whole reduction from subset sum to number partitioning is to map the instance $a_1, \cdots, a_n$, T to $b_1, \ldots, b_{n+2}$, where $b_i = a_i$ for $i = 1, \ldots, n$, $b_{n+1} = S + 1 - T$, and $b_{n+2} = 1 + T$. We know that solutions exist on one side of this map if and only if they exist on the other, and, morever, that solutions to the number partitioning instance can be efficiently turned into solutions to the original subset sum instance, by just removing the occurrence of $n + 1$ from the side of the partition that contains it.

5 Dealing with NP-hard problems

The general consensus among the relevant experts is that P and NP are not equal. In fact, it's believed that `SAT` has no algorithm which substantially improves on the naïve brute force approach. So unless all these experts are wrong, there is not much hope for efficient solutions to any NP-hard problem.

However, we can sometimes change the problem to make it easier. The following sections illustrate a few different ways to approach this.

5.1 Taking advantage of additional structure

When facing a computational problem, we often start by abstracting out unnecessary details to keep things simple. For example, we might be given the floor plan of a large building, and asked to find placement of security cameras so that the cameras cover every portion of the building. We can model this problem as a vertex cover problem—a simple variant is to think of walls of rooms as vertices, with edges between walls that can 'see' each other. Of course, we know that vertex cover is hard, so now the temptation is to say that our original problem is hopelessly difficult.

But suppose that we know that we can model this problem as an instance of vertex cover in which the graph is a tree. Then we can actually solve it quite efficiently—there is a simple greedy algorithm! Even if our input graph is not quite a tree, but is close to one in a certain precise sense,[9] then we can still find efficient algorithms to decide vertex cover.

In general, most problems in the 'real world' have so-called "extra structure" that can be exploited. Often this extra structure lies in the details that we have abstracted away when we simplified our original problem. Thus a wise algorithms designer, upon abstracting his problem into an NP-hard problem, seeks to find this additional structure, rather than give up.

5.2 Approximation algorithms

In some situations, the real-world problem we want to solve is inherently NP-hard. In these cases, no amount additional structure will save us, so we have to do something else.

One common technique is the use of *approximation algorithms*. An example of this is for vertex cover: suppose we seek to find, instead of a perfectly minimal vertex cover, a vertex cover which is not too much larger than the smallest vertex cover. This is clearly no harder than our original problem, and actually seems a bit easier.

An example approximation algorithm for vertex cover first finds a maximal matching in the underlying graph[10], and then chooses both endpoints of the edges in this matching. It is easy to show that this is indeed a vertex cover. Since every vertex cover has to include at least one of the vertices of the matching, we can also deduce that our vertex cover is no more than double the size of the smallest vertex cover. Thus if it suffices for our purposes to find a vertex cover which is no more than twice as big as it needs to be, then this algorithm will work perfectly.

Many other computational problems have good known approximations.

However, there is sometimes a limit to approximation quality. For instance, the vertex cover problem can be approximated to within a factor of 2, but it is also known that an efficient algorithm approximating the minimum vertex cover to within a factor of 1.3 would imply that P = NP. Similar hardness results hold for many other NP-hard optimization problems.

5.3 Heuristics

It sometimes happens that we are interested in solving a problem which is inherently NP-hard, and for which we cannot afford the best-known approximation guarantees. In these cases, the theory of algorithms seems to be insufficient to be of much help. However, real-world problems rarely need the complete guarantee that mathematical rigor can provide. An algorithms designer can thus fall back on heuristic approaches, and hope that they happen to work well enough for their problem.

[9] The precise sense is that the graph has *bounded tree-width*.

[10] We covered an algorithm for finding a maximum matching in *bipartite* graphs. There is an algorithm for finding maximum matchings in general graphs, but it is considerably more complicated.