CS 577: Introduction to Algorithms Fall 2015

Computational Intractability

Instructor: Dieter van Melkebeek Scribe: Andrew Morgan

DRAFT

(Last updated: December 9, 2015)

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 not 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 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 function 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.

It's quite easy to come up with more problems that fit this structure, so, given their pervasiveness, it is useful to give them 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 y is a valid solution for the instance x iff $V(x,y) = 1$. f is likewise a function $\{0,1\}^* \times \{0,1\}^* \to \mathbb{R}$ so 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 of length at most $|x|^d$ so that V recognizes y as a valid solution for the instance x. Symbolically

$$S_x \doteq \{y \in \{0,1\}^* : |y| \leq |x|^d, V(x,y) = 1\}$$

  (The fact that every y in S_x has length at most $|x|^d$ corresponds to the first bullet point above.)

- $V(x,y)$ 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. 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 are a few variations on `TSP`. The first is that we might drop the requirement that the salesperson return to his starting point. Thus the goal is to find a simple path P which contains every vertex and minimum total length. We can further specify the starting and/or ending points, and require the path to start/end at the specified points.

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 a few variations. The goal could be to find a simple path P that visits every vertex; this is the Hamiltonian Path problem. Similar to `TSP`, we can allow the starting and/or ending points of P to be specified as part of the input. Yet another variation that was inconsequential in `TSP` but which does make some different for Hamiltonicity is to allow G to be directed instead of undirected.

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 (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 and only be 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, 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 P $\subseteq$ NP. The reverse inclusion, NP $\subseteq$ P, is unknown.

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 show NP $\subseteq$ 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)*

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.[5] 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

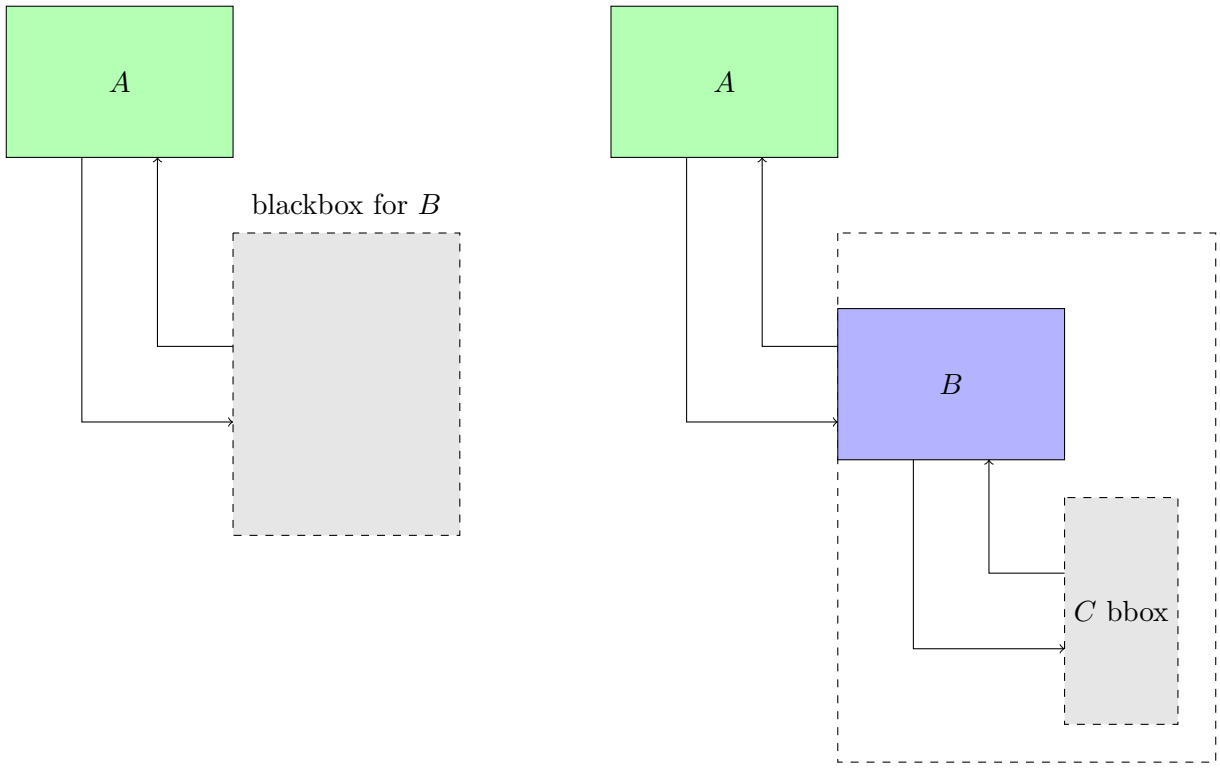
$$T_F(n) + T_F(n) \cdot T_G(T_F(n))$$

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. □

[5] 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.[6]

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.

[6] 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 going from "hard" to "easy". 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 k, and it's possible that every independent set of G with size at least k does include k. 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

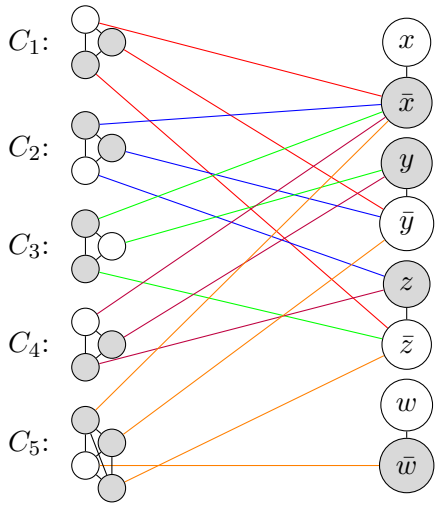
telling us that $C_j(a) = \overline{x_i} \leftarrow \texttt{True}$. Thus we should rule out these choices of I. We can do this by introducing an edge between $v(C_j, a)$ and $v(\overline{C_j(a)})$.

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`

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

(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 unshaded vertices, corresponding to the satisfying assignment $w, x, y, z \leftarrow 1, 0, 1, 1$.

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

(Forthcoming)

4 NP-completeness: universal problems for NP