15-451: Algorithms

Lecture 20: Tractable Problems (notes KS)

1 Tractability and Polynomial Time

Recall the definition of worst case running time of an algorithm:

$$T_{\mathcal{A}}(n) = \max \# \text{ steps algorithm } \mathcal{A} \text{ takes on any input of size } n.$$

Let us say that a problem is *tractable* iff there is some algorithm $\mathcal{A}$ with polynomial running time that solves the problem:

$$T_{\mathcal{A}}(n) = O(n^k)$$

Note that this is purely an upper bound, so any algorithm with running time, say, $O(n \lg n)$ will be fine.

There are several natural objections to this definition of tractability.

- Asymptotic analysis. It might be that that the constants in hidden by O are so large as to make the algorithm useless. This seems to be very rare in practice.
- Degree problem. It might be that the exponent k is so large as to make the algorithm useless. Again, it seems that any real algorithm that has polynomial running already has a fairly small exponent, say, less than 10.

1.1 Robustness and Closure

Another, potentially more severe problem comes from the fact that we do not have a precise definition of $T_{\mathcal{A}}$. In order to define what exactly is meant by the number of steps in a computation we have to fix a *model of computation* and then count steps with respect to that particular model. There are several reasonable choices:

- Random access machines (RAMs)
- Turing machines
- Boolean circuit families
- Markov algorithms
- Rewrite systems

Given the proper definitions, it turns out that polynomial time computable means the same for all these models of computation. E.g., one can simulate a RAM on a (more primitive) Turing machine with only a polynomial increase in running time.

Another crucial property of our definition has to do with the fact that polynomials are closed under substitution. Now suppose we feed the output of a polynomial time algorithm $\mathcal{A}$ as input to another polynomial time algorithm $\mathcal{B}$: $y = \mathcal{A}(x)$ and $z = \mathcal{B}(y)$. Then $|y| \leq p(|x|)$ for some polynomial p. Hence the computation of $\mathcal{B}$ is also polynomial in $|x|$. Thus, we can compose polynomial time algorithms to obtain another polynomial time algorithm.

2 Classification of Problems

Recal the classification from the first lecture.

- Decision Problems

  Here we are really asking a question, and the output is YES or NO. Typically we want to know whether the input has some special property or not.

- Function Problems

  These are perhaps the most important ones in applications. We are given some problem instance x and have to compute a uniquely determined solution z.

- Search Problems

  In a search problem, a given instance is associated with a set of solutions. The goal is to find one of these solutions, or to return NO if the set of solutions is empty.

Often, there are variants of a problem of all three types, some of them more natural than others.

Example 2.1 Consider the Shortest Path Problem (SPP for short). The natural version is a search problem: given the network G, a souce s and a target t, return a shortest path from s to t.

However, we can also consider this as a function problem: compute the distance $d(s, t)$.

Lastly, and more artificially, suppose we are given an additional bound B. We can ask whether there is a path from s to t of cost at most B.

It is intuitively clear that the search version is at least as hard as the function version, which is in turn at least as hard as the decision version: a fast algorithm to solve one immediately produces a fast algorithm to solve the other.

More suprisingly, we can also go into the opposite direction. Given an algorithm for $\mathrm{SPP}_{\mathrm{dv}}$ we can construct an algorithm for $\mathrm{SPP}_{\mathrm{fv}}$: just run binary search on the interval $[0, n\alpha]$ where α is the maximum cost of any edges, and n the number of nodes in the network.

From $\mathrm{SPP}_{\mathrm{fv}}$ we can obtain an algorithm for $\mathrm{SPP}_{\mathrm{sv}}$: First, use the function version to compute the distance $d = d(s, t)$. Then, for all neighbors x of s, again use the function version to determine the distance $d(x, t)$. Find x such that $c(s, x) + d(x, t) = d$ and choose x as the first vertex on the shortest path. Continue until the whole path is constructed.

In either case, the running time is polynomial in the running time of the given algorithm. In other words, one version of SPP is tractable iff the others are. Of course, we know already that SPP is tractable, but we will soon encounter similar problems where tractability is not known.

2.1 Booean Formulae

Several important combinatorial problems arise in connection with Boolean formulae.

Given a Boolean formula $\varphi(x_1, x_2, \ldots, x_n)$ with variables as indicated, a truth assignment is a map from the variables to $\{0, 1\}$. Given a truth assignment α we can inductively define the truth value of φ under α, in symbols $\varphi[\alpha]$.

A formula φ is a *tautology* iff $\varphi[\alpha] = 1$ for all α, a *contradiction* iff $\varphi[\alpha] = 0$ for all α, and *satisfiable* iff $\varphi[\alpha] = 1$ for some α.

Hence, there are a number of combinatorial problems associated with Boolean formulae. For example:
Tautology (TAUT)

- Instance: A Boolean formula φ.
- Yes-instance: A tautology.

Satisfiability (SAT, decision version)

- Instance: A Boolean formula φ.
- Yes-instance: A satisfiable formula.

Satisfiability (SAT, search version)

- Instance: A Boolean formula φ.
- Solution: A satisfying truth assignment if it exists, "No" otherwise.

All these problems yield to a brute force attack: we can compute $\varphi[\alpha]$ in polynomial time, so we only need to perform an explicit search over all 2^n possible truth assignments. Of course, this approach does not establish tractability. In fact we will see that these problems most likely are intractable.

Exercise 2.1 Show how to solve $\mathsf{SAT}_{\mathrm{sv}}$ given an algorithm for $\mathsf{SAT}_{\mathrm{dv}}$.

Exercise 2.2 Show how to solve TAUT given an algorithm for SAT.

We will focus on decision problems in what follows; as we have seen this is enough as far as basic tractability is concerned (at least in many cases). Note that we can always think of a problem instance as a string over some alphabet Σ, and indeed the alphabet $\{0, 1\}$. Hence decision problem A can be associated with

- a set of words $I_A \subseteq \Sigma^*$, the set of all instances of A, and
- a set of words $Y_A \subseteq I_A$, the set of all Yes-instances of A.

We will insist that membership in I_A can always be decided in polynomial time, so we can basically ignore this part. The whole decision problem comes down to the set Y_A.

Definition 2.1 *The class of all tractable decision problems is written* P.

Formally, we can think of P as a class of languages, namely the class of all the sets Y_A.

Note that the encoding has to be natural, we cannot allow tricks such as the following. Encode a natural number n as $bin(n)\#0 \in \{0, 1, \#\}^*$ if n is composite, and as $bin(n)\#1$ otherwise. In this encoding primality testing becomes trivial: just check the last bit. The analogous trick trivializes any decision problem (even undecidable ones).

3 Reductions

To make the idea of using a given algorithm as a subroutine inside another algorithm more precise, we will define two notions of reductions from one combinatorial problem to another. In the design of algorithms, this technique is quite standard: instead of constructing an algorithm from scratch, use an existing one, or perhaps several (whose implementation may already be found in a library). For example, the construction of a minimal spanning tree can be reduced to the union-find problem, plus sorting.

From the point of view of complexity theory, reductions can be used to produce upper bounds: The idea is that if we can somehow reduce problem A to problem B, the complexity of A cannot be any larger that the complexity of B, plus the complexity of the reduction. Thus, to obtain good bounds we need to make sure that the reduction itself is not terribly complicated.

3.1 Turing Reductions

In the first, more general type of reduction we assume that calls to some algorithm $\mathcal{B}$ can be performed at a cost of 1 step: we generate the input, call the subroutine, and then get back the output (for decision problems just a single bit). In other words, we are allowed to ask membership queries "Is $z \in Y_B$?" at a charge of one step. We can think of having an *oracle* that answer these questions (truthfully, and reliably, unlike in reality).

Definition 3.1 *Decision problem A is* polynomial time Turing reducible *to decision problem B iff there is an algorithm $\mathcal{A}$ for A that uses an oracle $\mathcal{B}$ for B, and has polynomial running time. If B is also polynomial time Turing reducible to A we say the problems are* polynomial time Turing equivalent.

In symbols, $A \leq_p^T B$ and $A \equiv_p^T B$.

Example 3.1 $\mathsf{SAT} \leq_p^T \mathsf{TAUT}$ and $\mathsf{TAUT} \leq_p^T \mathsf{SAT}$. Thus, SAT and TAUT are polynomial time Turing equivalent.

Exercise 3.1 Show that $\leq_p^T$ is a transitive relation.

Exercise 3.2 Show that $A \leq_p^T B$ and $B \in \mathrm{P}$ implies $A \in \mathrm{P}$.

3.2 Many-One Reductions

A more restricted type of reduction is obtained by translating an instance of problem A directly into an instance of problem B, using a polynomial time computation to do so. In other words, we need a polynomial time computable function f such that x is a yes instance of A iff $f(x)$ is a yes instance of B. In this case A is said to be *polynomial time reducible* to B. A and B are *polynomial time equivalent* if they are mutually reducible.

Notation: $A \leq_p B$ and $A \equiv_p B$

Exercise 3.3 Show that $\leq_p$ is transitive.

Exercise 3.4 Define $A \leq_{\text{lin}} B$ to mean that there is a linear time reduction from A to B. Show that this is a transitive relation.

4 Nondeterministic Polynomial Time

Recall that it easy to convince someone that a large number is composite: we can provide two factors, and the test of our assertion is to multiply out the alleged factor and check their product is the given number. Likewise, it is easy to provide proof for SAT: hand over a satisfying truth assignment. The truth assignment here is the *certificate* or the *witness* for the satisfiability of the formula.

Another way of thinking about this is in terms of a protocol. A *verifier* is trying to assertain whether a formula is satisfiable. He sends the formula to an *oracle*, who returns a truth assignment. The verifier then checks that the assignment really satisfies the formula. If so, the verifier accepts the formula, otherwise he rejects it.

This motivates the following definition.

Definition 4.1 *A decision problem A is solvable in* nondeterministic polynomial time *iff there is a polynomial algorithm $\mathcal{A}$ that check whether a an alleged certificate is indeed a certificate, and if for every Yes-instance there is a certificate whose size is polynomial in the size of the instance.*

In other words, x is a Yes-instance iff

$$\exists\, y \left(|y| \leq p(|x|) \text{ and } \mathcal{A}(x, y) = 1 \right)$$

where p is some polynomial.

The class of all such decision problems is written NP.

Thus, in order to turn this into an honest deterministic algorithm we have to search over all potential certificates, and there are exponentially many possibilities.

Another way of thinking about this is that the nondeterministic algorithm guesses the certificate, and then verifies in polynomial that the guess was correct. The nondeterministic algorithm is considered correct if it has at least one proper guess for each yes instance, but none for any no instance. Note that this produces an asymmetry between yes and no answers that is not present in classical, deterministic algorithms.

Instead of guessing the certificate, we might also query an oracle. After receiving the alleged certificate, we perform a polynomial time computation to verify correctness. In terms of a protocol, this looks like so:

Verifier	Oracle
send instance x	
	send back certificate y
verify y is certificate,	
return Yes/No accordingly	

Lastly, we could interpret NP in terms of very weak probabilistic algorithms. Let A be a problem in NP. Given a polynomial time algorithm $\mathcal{A}$ that tests whether a certificate is correct, we can build an algorithm $\mathcal{B}$ as follows.

- On input x, generate a random bit string y of length $p(|x|)$.
- Return $\mathcal{A}(x, y)$.

In this setting we have

$$\begin{array}{lll} x \in Y_A & \text{implies} & \Pr[\mathcal{B}(x) = 1] > 0 \\ x \notin Y_A & \text{implies} & \Pr[\mathcal{B}(x) = 1] = 0 \end{array}$$

Note that this does not produce practical algorithms, we need something like $\Pr[\mathcal{B}(x) = 1] > 1/2$.

From the definitions, $\mathrm{P} \subseteq \mathrm{NP}$. However, the following is arguably the main open problem in TCS, and has been open for 30 years.

Open Problem: Is $\mathrm{P} = \mathrm{NP}$?

It appears that the two classes are distinct, but at present it is not even clear what line of attack might yield a proof.

Example 4.1 SAT is in NP. Note, though, that TAUT does not appear to be in NP.

Here are some more examples of problems in NP. Let $G = \langle V, E \rangle$ be an undirected graph.

A *vertex (to edge) cover* of G is a set $C \subseteq V$ such that $\forall e \in E\ \exists v \in C(v \text{ lies on } e)$. A *clique* of G is a set $D \subseteq V$ such that $\forall v, u \in D(\{v, u\} \in E)$. An *independent set* of G is a set $I \subseteq V$ such that $\forall v, u \in I(\{v, u\} \notin E)$.

By adding an artificial bound k on the size of any such set we can turn the existence of a vertex cover/clique/independent set into a decision problem.

Example 4.2 VC, Clique, IS are all in NP.

Since there are $\binom{n}{k}$ subsets of V to consider (the potential certificates) brute force will be exponential time. Note that intuitively one might think that there are reasonable ways to speed up the search. For example

Exercise 4.1 Show that VC, Clique, IS are all polynomial time equivalent.

Exercise 4.2 Show that $VC_{\mathrm{fv}} \leq_p^T \mathsf{VC}_{\mathrm{dv}}$. Show that $\mathsf{VC}_{\mathrm{sv}} \leq_p^T \mathsf{VC}_{\mathrm{dv}}$.

4.1 Negation

In the classical setting, there is essentially no difference between a decision problem and its negation: $I_{\overline{A}} = I_A$ and $Y_{\overline{A}} = I_A - Y_A$. In particular, A is in P iff $\overline{A}$ is.

But for NP the situation changes: having a witness for A has nothing to do with having a witness for $\neg A$.

For example, to show that a formula is satisfiable the witness is a truth assignment. But it is not clear how the property "not satisfiable", i.e., being a contradiction, could be verified by some witness.

Likewise, being composite is clearly a property in NP: just guess the factors and perform the multiplication. But for the negation, being a prime, the situation is much more complicated. It turns out that primality is in fact in NP (V. Pratt), but the proof requires some clever number theory.

Definition 4.2 *The class co-*NP *consists of all decision problems A such that $\overline{A}$ is in* NP.

Note that $\mathrm{P} \subseteq \mathrm{NP} \cap \text{co-NP}$.

Primality is in $\mathrm{NP} \cap \text{co-NP}$.

5 Hardness and Completeness

It seems that $\mathrm{P} \neq \mathrm{NP}$, but no one is able to prove this at present. In order to find a problem that is in NP but fails to be in P it is natural to look for the hardest problem in NP.

Definition 5.1 *A decision problem A is* NP-hard *iff for all problems B in* NP*: $B \leq_p A$.*

A decision problem A is NP-complete *iff A is* NP*-hard and belongs to* NP*.*

Proposition 5.1 *Suppose A is* NP*-hard and $A \in \mathrm{P}$. Then $\mathrm{P} = \mathrm{NP}$.*

This is used in the opposite direction: once a problem is shown to be NP-hard, we may safely assume that there is no polynomial time algorithm for it.

The existence of an NP-hard problem is not surprising: just take the disjoint union of all problems in NP. Fix an enumeration $(L_e)_{e\in\mathbb{N}}$ of all problems in NP and consider

$$\{\, e\#x \;\big|\; x \in L_e \,\} \subseteq \mathbf{2}^*\#\mathbf{2}^*$$

The reduction here is trivial (linear time): $f(x) = e\#x$.

But, this universal set is not itself in NP. To guarantee membership in NP we need to encode running time. To this end, write $\mathcal{A}_t$ for the decision algorithm that is obtained from $\mathcal{A}$ by truncating the computation after at most t steps. If the computation has already terminated, $\mathcal{A}_t$ returns the proper result, otherwise return 0 by default. Now define the universal set

$$\mathcal{U} = \{\, e\#x\#1^t \;\big|\; \exists\, y\, (|y| \leq t \wedge \mathcal{A}_t(x, y) = 1) \,\}$$

where e is a code number for the polynomial time verifier $\mathcal{A}$ associated with problem L_e

Theorem 5.1 *The set $\mathcal{U}$ is* NP*-complete.*

Proof.

For membership in NP, note that one can construct the verifier $\mathcal{A}$ from e in polynomial time. Hence, to check whether $e\#x\#1^t \in \mathcal{U}$ we only have to guess y, and run $\mathcal{A}$ for t steps.

For hardness, consider the reduction

$$f : \mathbf{2}^* \to \{0, 1, \#\}*$$
$$f(x) = e\,\#\,x\,1^{p(|x|)}$$

where p is the polynomial running time bound for the problem L_e. It is clear that f is computable in time $O(p(n))$, and clearly $x \in L_e \iff f(x) \in \mathcal{U}$. $\square$

So, we have at least one NP-complete problem. Unfortunately, $\mathcal{U}$ is too artificial to be of any interest. But, it still useful to show that other problems are also NP-complete. This involves two steps, the first of which is usually the harder one.

- Show that A is in NP, and
- show that $B \leq_p A$ where B is some already known NP-complete problem.

This suffices by transitivity of $\leq_p$.

5.1 Cook's Theorem

NOTE: This section assumes some familiarity with the notion of a Turing machine.

In this section we will show that the class NP contains complete sets with respect to polynomial time reduction.

Theorem 5.2 *Cook's Theorem*

Satisfiability is NP*-complete.*

Proof. Let L be in NP. We may safely assume that $L \subseteq \mathbf{2}^*$, otherwise use coding. Suppose $\mathcal{M}$ is a non-deterministic Turing machine that accepts L and has running time $T_{\mathcal{M}}(n) \leq p(n)$ where p is some polynomial. Again by coding one may assume that the tape alphabet of $\mathcal{M}$ is $\{0,1\}$; the blow-up in running time is only a constant.

Now let $x \in \{0,1\}^n$. We construct a boolean formula Φ depending uniformly on x and $\mathcal{M}$ such that: $x \in L \Leftrightarrow \Phi$ is satisfiable. The satisfying truth assignment – if it exists – codes an accepting computation of $\mathcal{M}$ on input x. Let $N = p(n)$ and let $[m]$ be the states of $\mathcal{M}$. We assume that 1 is the initial state of $\mathcal{M}$, m its accepting state and that $\mathcal{M}$ never leaves state m once this state is reached. Introduce the following boolean variables:

states $S(t,p)$, $0 \leq t \leq N$, $1 \leq p \leq m$,
head position $H(t,i)$, $0 \leq t,i \leq N$,
tape inscription $C(t,i)$, $0 \leq t,i \leq N$.

Hence the total number of variables is $O(p^2(n))$.

Φ_1: $\bigwedge_{0 \leq t \leq N} EO_m(S(t,1),\ldots,S(t,m))$

Φ_2: $\bigwedge_{0 \leq t \leq N} EO_{N+1}(H(t,0),\ldots,H(t,N))$.

Φ_2: $S(0,1) \wedge S(N,m) \wedge H(0,0) \wedge \bar{C}(0,0)\ wedge \bigwedge_{1 \leq i \leq n} C(0,i) = x_i \wedge \bigwedge_{n<i\ leqN} \bar{C}(0,i)$.

Φ_4: $\bigwedge_{0 \leq t,i \leq N} (\bar{H}(t,i) \Rightarrow C(t,i) = C(t+1,i))$.

Φ_5: $\bigwedge_{0 \leq t,i \leq N, p \in [m]} (S(t,p) \wedge H(t,i) \Rightarrow \bigvee_{(q,a,\Delta) \in \delta(p,C(t,i))} S(t+1,q) \wedge H(t+1,i+\Delta) \wedge C(t+1,i) = a$.

Lastly define $\Phi = \Phi_1 \wedge \ldots \wedge \Phi_5$. □

Conjunctive Normal Form

Let $U = \{u_1,\ldots,u_m\}$ be a collection of boolean variables. A literal over U is a boolean formula of the form u_i or $\bar{u}_i$.

Φ is in conjunctive normal form (CNF) iff $\Phi = \Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_n$ where $\Phi_i = z_{i,1} \vee z_{i,2} \vee \ldots \vee z_{i,k(i)}$, z_i a literal.

Φ is in k-conjunctive normal form (k-CNF) iff $k(i) = k$ for all $i = 1, \ldots, n$. Φ_1 and Φ_2 are equivalent ($\Phi_1 \Leftrightarrow \Phi_2$) iff $\forall \alpha$truth-assignment$(\Phi_1[\alpha] = \Phi_2[\alpha])$.

Lemma 5.1 *For every boolean formula Φ there exists an equivalent boolean formula $CNF(\Phi)$ in conjunctive normal form.*

Proof. Proof is by induction on build-up of Φ. The cases "Φ is a literal" and "$\Phi = \Phi_0 \wedge \Phi_1$" are both trivial. For $\Phi = \Phi_0 \vee \Phi_1$ we can use the distributivity of $\vee$ and $\wedge$.

The hard case is $\Phi = \neg\Psi$:

$$\neg\mathsf{CNF}(\Psi) \Leftrightarrow \neg(\Psi_1 \wedge \Psi_2 \wedge \ldots \wedge \Psi_n) \Leftrightarrow \neg\Psi_1 \vee \ldots \vee \neg\Psi_n \Leftrightarrow \bigwedge_{\substack{f:[n]\to N \\ 1 \le f(i) \le k(i)}} \bigvee_{j \in [n]} \bar{z}(j, f(j)).$$

$\square$

There are other strategies to obtain CNF. A rather elegant one is to first push all negations inside until only variables occur negated. Then one can construct the CNF and DNF in parallel, again using the distributivity of $\vee$ and $\wedge$.

Note, though, that regardless of the method used there is no polynomial bound on the size of the CNF of a boolean formula. Hence the map $\Phi \mapsto CNF(\Phi)$ is not computable in polynomial time.

We now show that SAT remains hard even when the formulae under consideration are required to be in CNF.

Problem: Conjunctive

Instance: N

Question: o

rmal Form Satisfiability (CNF-SAT) A boolean formula Φ in CNF. Is Φ satisfiable ?

The decision problem 3SAT is defined similarly (Φ is required to be in 3CNF).

Corollary 5.1 *CNF-SAT is* NP*-complete.*

Proof. Note – the last lemma does not provide this result! $\mathsf{CNF}(\Phi)$ is in general of size exponential in Φ. Hence the map $\Phi \mapsto \mathsf{CNF}(\Phi)$ cannot be computed in polynomial time. However, the proof of Cook's theorem produces essentially a formula in CNF, done. $\square$

Corollary 5.2 *3SAT is* NP*-complete.*

Proof. It suffices to show that CNF-SAT $\le_p$ 3SAT.

Let $\Phi = \Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_n$ where $\Phi_i = z_{i,1} \vee z_{i,2} \vee \ldots \vee z_{i,k(i)}$, z_i a literal. Several cases arise:
$k(i) = 1$: add new dummy variables y_1, y_2 and replace Φ_i by $\Phi'_i = (z_{i,1} \vee y_1 \vee y_2) \wedge (z_{i,1} \vee \bar{y}_1 \vee y_2) \wedge (z_{i,1} \vee y_1 \vee \bar{y}_2) \wedge (z_{i,1} \vee \bar{y}_1 \vee \bar{y}_2)$.
$k(i) = 2$: add new dummy variable y and replace Φ_i by $\Phi'_i = (z_{i,1} \vee z_{i,2} \vee y) \wedge (z_{i,1} \vee z_{i,2} \vee \bar{y})$.
$k(i) \ge 4$: exercise. $\square$

Several remarks are in order.

1. 2SAT by way of comparison is in P.

   To see this consider a boolean formula Φ in 2-CNF with variables $x_1, \ldots, x_n$. Define the following directed graph $G_\Phi = \langle V_\Phi, E_\Phi \rangle$. G_Φ has vertices $x_1, \ldots, x_n$ and $\bar{x}_1, \ldots, \bar{x}_n$; there are edges $(\overline{z}, z')$ and $(\overline{z'}, z)$ for every clause of the form $z \vee z'$ in Φ. One can show that: Φ is satisfiable iff there is no cycle in G_Φ that contains both x_i and $\bar{x}_i$ for any $i = 1, \ldots, n$.

2. MONOTONE 3SAT is also NP-complete (in each clause either all variables negated or all unnegated).

3. !3SAT is also NP-complete (here the truth assignment is required to satisfy exactly one literal per clause).

4. SAT is quite powerful; e.g. Hamiltonian Path can be directly reduced to SAT as follows.

As a little exercise in the use of Boolean formulae to encode combinatorial problems, let us show how Hamiltonian cycles can be translated into a satisfiability problem.

Proposition 5.2 *Hamiltonian Cyle* $\leq_p$ SAT.

Proof. Let $G = \langle [n], E \rangle$ be the graph in question.

Set $m = |E|$ and enumerate the edges of G as $e_1, \ldots, e_m$. Now introduce boolean variables $U = \{ u_i \mid i \in [m] \} \cup \{ v_{i,t} \mid i \in [n], t \in [0, n] \}$.

We will construct a boolean formula Φ_G on U that is satisfiable iff G is Hamiltonian. The crucial idea is to have $\alpha(u_i) = \text{true}$ iff edge e_i lies on the Hamiltonian cycle. Let us write $E2T_r(x_1, \ldots, x_r)$ for the boolean formula that is true iff exactly two of the variables $x_1, \ldots, x_r$ are true. We have $\Phi_G = \Phi_1 \wedge \Phi_2$ where the intended meaning of the components is as follows. Φ_1: every vertex is incident upon exactly two chosen edges. Φ_2: the subgraph induce by the chosen edges is connected.

$\Phi_1 = \bigwedge_{i \in [n]} E2T_r(e_{j_1}, \ldots, e_{j_r})$.

$\Phi_2 = v_{1,0} \wedge \bigwedge_{i>0} \bar{v}_{i,0} \wedge \bigwedge_{t \in [n]} (v_{i,t} \Leftrightarrow v_{i,t-1} \vee \bigvee (v_{j,t-1} \wedge u_{i,j}) \wedge \bigwedge_{i \in [n]} v_{i,n}$.

Φ_G can be constructed from G in polynomial time since $\langle \Phi_G \rangle = O(n^3 \log n)$. $\square$

Note that component Φ_1 alone is not sufficient; it only expresses the statement "G has a two-factor" (i.e., it is the disjoint union of cycles). Coincidentally, the latter property can be tested in polynomial time.

6 Some NP-complete Problems

Problem:	**Vertex Cover (VC)**
Instance:	A graph $G = \langle V, E \rangle$, a natural number k, $1 \leq k \leq \|V\|$.
Question:	Does G have a vertex cover of size k ?

Similarly define decision problems Independent Set (IS) and Clique.

Problem:	**Longest Path**
Instance:	A graph $G = \langle V, E \rangle$, a natural number k, $1 \leq k \leq \|V\|$.
Question:	Is there a simple path of length k in G ?

Problem:	**Hamiltonian Path**
Instance:	A graph $G = \langle V, E \rangle$, a natural number k, $1 \leq k \leq \lvert V \rvert$.
Question:	Is there a Hamiltonian Path in G ?

Problem:	**Hamiltonian Cycle (HC)**
Instance:	A graph $G = \langle V, E \rangle$.
Question:	Is there a Hamiltonian Cycle (cycle of length $\lvert V \rvert$) in G ?

Problem:	**Travelling Salesman (TSP)**
Instance:	A distance function $d : C \times C \rightarrow \mathbb{N}$, a bound $B \in \mathbb{N}$.
Question:	Is there a tour of all cities of length at most B ?

Problem:	**Three-Dimensional Matching (3DM)**
Instance:	Three set X, Y, Z of cardinality n, $M \subseteq X \times Y \times Z$.
Question:	Is there a matching $M' \subseteq M$ of size n?

Problem:	**Partition**
Instance:	A set A of cardinality n, a size function $s : A \rightarrow \mathbb{N}^+$.
Question:	Is there a partition $A = A_1 \cup A_2$ such that $\sum_{a \in A_1} s(a) = \sum_{a \in A_2} s(a)$?

Problem:	**Subset Sum**
Instance:	A set A of cardinality n, a size function $s : A \rightarrow \mathbb{N}^+$ and a number B.
Question:	Is there a subseteq $A_0 \subseteq A$ such that $\sum_{a \in A_0} s(a) = B$?

And some proofs.

Theorem 6.1 *Vertex Cover is* NP*-complete.*

Proof. It suffices to show that 3SAT $\leq_p$ VC.

Let $U = \{u_1, \ldots, u_m\}$ be a collection of boolean variables and $\Phi = \Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_n$ where $\Phi_i = z_{i,1} \vee z_{i,2} \vee z_{i,3}$ be a boolean formula over U in 3CNF. Construct a graph $G = G(\Phi)$ as follows: Vertices: truth-setting x_i and x'_i, $i \in [m]$, testing $v_{i,j}$, $i \in [n]$, $j \in [3]$.

Edges: $\{x_i, x'_i\}$, triangles $\{v_{i,j}, v_{i,k}\}$, $j \neq k$, $\{x_k, v_{i,j}\}$ iff $z_{i,j} = u_k$ and $\{x'_k, v_{i,j}\}$ iff $z_{i,j} = \bar{u}_k$.

Lastly set $B = m + 2n$. The following claims are easily verified.

Claim: G has vertex cover of size B iff Φ is satisfiable.

Claim: G and B can be constructed in polynomial time.

Thus 3SAT $\leq_p$ VC and we're through. □

Corollary 6.1 *Independent Set and* Clique *are* NP*-complete.*

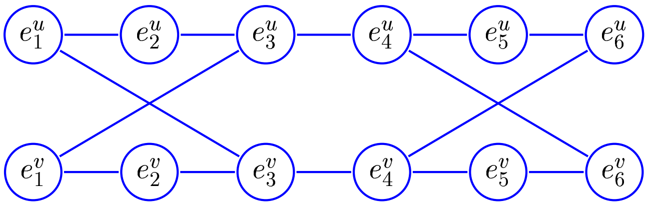
Theorem 6.2 *Hamiltonian Cycle is* NP*-complete.*

Proof. Reduction from Vertex Cover. Let $G = \langle V, E\rangle$ and B be an instance of vertex cover, $1 \le B \le n = |V|$. We may assume wlog that for all v in V $\deg(v) \ge 2$. Fix an enumeration $v(i)$, $i \in [\deg(v)]$, of all neighbors of v.

Define a new graph H as follows:
Vertices: $a_1, \dots, a_b$ and e_i^v for all $v \in e \in E$, $i \in [6]$.

Edges come in several types.

Box-edges: for edge $e = \{u, v\}$ as indicated in the figure.



Chain-edges: consider all edges $e = \{v, v(i)\}$, $e' = \{v, v(i+1)\}$ incident upon v, and add edges $\left\{e_6^{v(i)}, {e'}_1^{v(i+1)}\right\}$

Link-edges: lastly, connect the nodes in the independent set to the edge-boxes via $\{a_j, e_6^1\}$ and $\{a_j, e_6^{d(v)}\}$ for all $j \in [B]$.

With the edges linking the boxes we obtain a v-chain.

Note that H has $B + 12 \cdot |E| = O(n^2)$ vertices and can thus be constructed from G in poynomial time.

Claim 1 Let P be a Hamiltonian cycle in H. Then P must enter and exit each box at the same side, say, v. P can pass through the e-box in exactly one of two ways: type full or type half.

Claim 2 Let P be a Hamiltonian cycle in H. Then P consists of blocks of the form $a_i, e_1^v, \dots, e_6^{v(\deg(v)}, a_j$ where $\dots$ contains the whole v-chain.

Claim 3 Let P be a Hamiltonian cycle in H. Then G has a vertex cover of size B.
To see this define $C = \{\, v \in V \mid P \text{ uses the } v\text{-chain } \}$. Now let $e = \{u, v\} \in E$. As P passes through the e-box it must use the u-chain or v-chain. Thus C covers e.

Claim 4 Suppose G has a vertex cover of size B. Then H has a Hamiltonian cycle.

Construct P as follows: P has B blocks $a_i, e_1^v, \dots, e_6^{v(\deg(v)}, a_{i+1}$ where v is in C. The way P passes through the e-box on the v-chain is determined by whether both $u, v \in C$ type half or just one of u, v lies in C type full.

Thus G has a VC of size B iff H is Hamiltonian. Hence VC is polynomial time reducible to Hamiltonian cycle. □

Corollary 6.2 *Traveling Salesman is* NP*-complete.*

Corollary 6.3 *Hamiltonian Path is* NP*-complete.*

Corollary 6.4 *Longest Path is* NP*-complete.*

There are several important variations of the TSP.

In the triangle TSP (ΔTSP) the distances are reqired to be symmetric and conform to the triangle inequality: $d(i,j) = d(j,i)$, $d(i,j) \leq d(i,k) + d(k,j)$.

An instance of the Geometric TSP is a finite set S of points in the plane with integer coordinates and a bound B. The question is now: Is there a tour of cost $\leq B$ where $d(x,y) = \lceil d_{euc}(x,y) \rceil$? Both ΔTSP and Geometric TSP are NP-complete (membership in NP is slightly non-trivial for Geometric TSP). Notice that ΔTSP is actually covered by our hardness argument: the instances of TSP involved in the reduction all have the triangle properties.

Theorem 6.3 *3DM is* NP*-complete.*

Proof. Direct imbedding of 3SAT. Assume $\varphi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ is a Boolean formula in 3CNF over variables $x_1, \ldots, x_n$. To construct an instance of 3DM, we use m many incarnations of all literals.

- Truth setting triples: (a_{ij}, x_{ij}, b_{ij}) and $(a_{ij}, \overline{x}_{ij}, b_{i,j+1})$
- Clause triples: (s_j, u, t_j) where u is a literal in C_j.
- Garbage collection: (α, u, β) where u is a literal.

The sets X, Y and Z are just the projections of these triples and all have cardinality $2mn$. There are $2nm$ truth setting triples, $3m$ clause triples, and garbage collection is used to fill up the sets to the proper cardinality.

Now suppose $M' \subseteq M$ is a matching.

Claim: M' contains exactly all triples (a_{ij}, x_{ij}, b_{ij}) or all triples $(a_{ij}, \overline{x}_{ij}, b_{i,j+1})$, for all i.

To see this, note that since $|M'| = 2nm$ there must be a triple $(a_{ij}, -, -)$ in M' for all i, j. Hence M' must contain at least m many of the truth setting triples for each variable. But an easy induction shows that it can only contain one kind (either triples for x_i or triples for $\overline{x}_i$, but never both).

Thus, we can translate the matching M' into a truth assignment:

$$\alpha(x_i) = \begin{cases} 0 & \text{if } M' \text{ contains only } x_i \text{ triples,} \\ 1 & \text{otherwise.} \end{cases}$$

Since M' must contain one triple of the form $(s_j, -, -)$ for all j, it is not hard to see that α is a satisfying truth assignment. Conversely, every such assignment is easily translated into a corresponding matching. $\square$

Theorem 6.4 *Partition is* NP*-complete.*

Proof. Imbedding of 3DM.

Consider an instance $M \subseteq X \times Y \times Z$ of 3DM. Let $m = |M|$ and $n = |X| = |Y| = |Z|$. We may safely assume that $[n] = X = Y = Z$, and that M is enumerated by $(f(i), g(i), h(i))$ for three functions $f, g, h : [m] \to [n]$.

We will define the index set A as $[n] \cup \{\alpha, \beta\}$. To describe the size function $s : A \to \mathbb{N}$ let $p = \lceil \lg m \rceil + 1$ and set

$$s(i) = 2^{p(3n-f(i))} + 2^{p(2n-g(i))} + 2^{p(n-h(i))}$$

for $s \in [n]$. Thus, $s(i)$ is a number whose binary expansion has exactly three 1 bits, all of whom appear in positions a multiple of p.

As usual, write $s(I)$ for $\sum_{x \in I} s(x)$ whenever $I \subseteq A$. Set $S = s([n])$.

Claim: Matchings correspond exactly to subsets $I \subseteq [m]$ such that $s(I) = S$.

Lastly, set

$$s(\alpha) = 2S - B, \qquad s(\beta) = S + B.$$

Now suppose we have a partition $A = A_1 \cup A_2$ such that $s(A_1) = s(A_2)$. Clearly, we cannot have both α and β on the same side of the partition. So suppose without loss of generality $\alpha \in A_1$, $\beta \in A_2$. Then $s(A_1 - \{\alpha\}) = S$, hence we have a matching by the claim.

The opposite direction is similar. □

Problem: **KNAPSACK**

Instance: A finite set A, a size function $s : A \to N^+$ and a value function $v : A \to N^+$, two bounds B_s and B_v.

Question: Is there a subseteq A_0 of A such that $s[A_0] \leq B_s$ and $v[A_0] \geq B_v$?

Corollary 6.5 *Knapsack is* NP*-complete.*

Proof. Knapsack is a special case of Partition: set $v = s$, $B_s = B_v = B/2$. □