ECS 120 Lesson 24 – The Class $\mathcal{NP}$, $\mathcal{NP}$-complete Problems

Oliver Kreylos

Friday, May 25th, 2001

Last time, we defined the class $\mathcal{P}$ as the class of all problems that can be decided by deterministic Turing Machines in polynomial time. We also discovered that nondeterministic Turing Machines are more powerful than deterministic ones – they recognize the same class of languages, but converting an NTM into a DTM increases the running-time exponentially. Thus, we suspect that the class of languages recognized by NTMs in polynomial time contains more complex problems than found in $\mathcal{P}$.

1 The Class $\mathcal{NP}$

We define the class $\mathcal{NP}$ analogously to $\mathcal{P}$:

Definition 1 (The Class $\mathcal{NP}$) $\mathcal{NP} := \left\{ L \subset \Sigma^* \;\middle|\; \exists M, k : L \textit{ is decided by NTM } M \textit{ in time } O(n^k) \right\}$ *is the class of all languages that are decided by* nondeterministic *Turing Machines in polynomial time.*

To show that a problem is in $\mathcal{NP}$, we have to give a nondeterministic algorithm that decides it, and we have to show that the algorithm runs in polynomial time $O(n^k)$. From the conversion algorithm NTM $\rightarrow$ DTM we know that there exists a DTM that decides the same language in exponential time $2^{O(n^k)}$. Since we cannot build nondeterministic Turing Machines (yet), the only practical way to decide $\mathcal{NP}$ problems we know of now is using the exponential algorithm constructed by the conversion. These algorithms are unusable but for the smallest problem sizes, which leads to our notion that problems in $\mathcal{NP} \setminus \mathcal{P}$ are *intractable*. There is strong evidence for this belief, but it has not been proved yet.

2 An Example Problem in $\mathcal{NP}$

To see how $\mathcal{NP}$ problems are typically solved, let us look at an example that is important in operations research: The Traveling Salesman Problem (TSP). Its task is as following: Given a directed weighted graph G, is there a round-trip in the graph such that the sum of weights on the edges traversed is not more than a given k?

To be more precise, a weighted graph G is a triple $G = (V, E, w)$, where V is a (finite) set of nodes, $E \subset V \times V$ is a set of directed edges, and $w\colon E \to \mathbf{R}^+$ is a function returning a (positive) weight for each edge in E. The definition of E is that two nodes $v_1, v_2 \in V$ are connected by an edge from v_1 to v_2, if and only if the pair $(v_1, v_2) \in E$ is an element of the set of edges. A round-trip in a graph with n nodes is a sequence of $n+1$ nodes $(v_0, v_1, \ldots, v_n) \in V^{n+1}$ such that the first node is identical to the last one, $v_0 = v_n$, that all nodes in V appear in the sequence, $\{v_0, v_1, \ldots, v_n\} = V$, and that all adjacent nodes in the sequence are connected by edges, $\forall\, i = 1, 2, \ldots, n : (v_{i-1}, v_i) \in E$. Less formally, a round-trip starts at some node, visits all other nodes in the graph exactly once and then comes back to the start node, while only following edges in the graph.

The original statement of the problem (and the reason for its name) was the following interpretation of G: The nodes are cities, the edges are highways/transportation routes/air routes between the cities, and the edge weights are the cost of travelling along that edge. The premise then is that a salesman has to visit all cities in a single round-trip, and to find an order to visit all cities such that the total travelling cost is not more than[1] some threshold k, see Figure 1.

Expressed as a language, the problem TSP is $\mathrm{TSP} = \left\{ \langle G, k\rangle \;\middle|\; G \text{ is a directed weighted graph with a round-trip of total weight} \leq k \right\}$. Here is a nondeterministic algorithm to decide TSP:

Algorithm TSP On input $\langle G, k\rangle$,

1. Create all possible sequences of $n+1$ nodes from V, where $n = |V|$.
2. For each sequence $(v_0, v_1, \ldots, v_n)$:

[1] Another, even more difficult, problem is to find the round-trip that has *minimal* total cost.

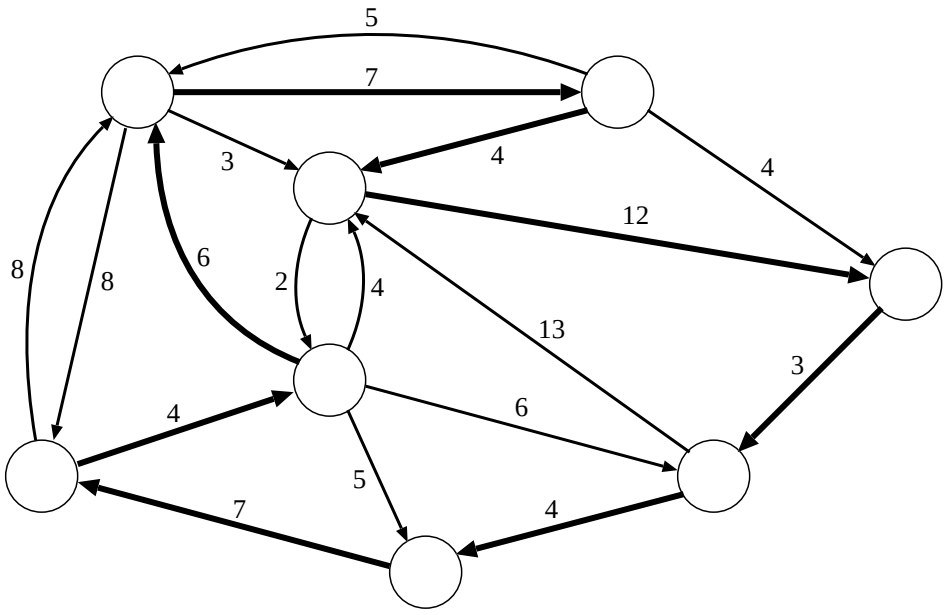

Figure 1: A graph for the Traveling Salesman Problem. The bold arrows define a round-trip of total weight 47.

3. Check whether it is a valid round-trip (using the three conditions from above):
   (a) Check that $v_0 = v_n$.
   (b) Check that $\{v_0, v_1, \ldots, v_n\} = V$.
   (c) Check that for all $i = 1, 2, \ldots, n : (v_{i-1}, v_i) \in E$.
4. Check that it has total weight not more than k: $\sum_{i=1}^{n} w(v_{i-1}, v_i) \leq k$.
5. If all conditions are met, accept; otherwise, reject.

This algorithm is correct: If there is any round-trip of total weight not more than k, it will be found because the algorithm considers all possible round-trips. To show that TSP $\in \mathcal{NP}$ we now have to show that its complexity is polynomial. It is easy to see that steps 3, 4 and 5 are polynomial; they merely involve checking some simple conditions. The problematic one is step 1: It involves creating *all possible* sequences of $n+1$ nodes, and there can be exponentially many, n^{n+1} to be precise. But a nondeterministic Turing Machine can generate all of them in polynomial time: It will perform $n+1$ selection steps, branching n ways in each step and selecting a different node from V as the next node in the sequence in every branch. After $n+1$ steps, the total number of active computation chains will be n^{n+1}, and each chain will have generated a different node sequence, see Figure 2.

3 $\mathcal{NP}$ as the Set of Verifiable Languages

The structure of the algorithm for TSP is typical for problems in $\mathcal{NP}$: They almost always consist of two steps:

1. "Guess" a solution to the problem, and
2. verify that the guessed solution is correct.

The "guessing" step is typically implemented as generating all possible potential solutions in parallel, and the verification step is typically deterministic and simple. It is noteworthy that the verification step cannot be omitted: Since a nondeterministic Turing Machine cannot really only guess the correct answer, all the wrong answers it guesses must be rejected by the verification

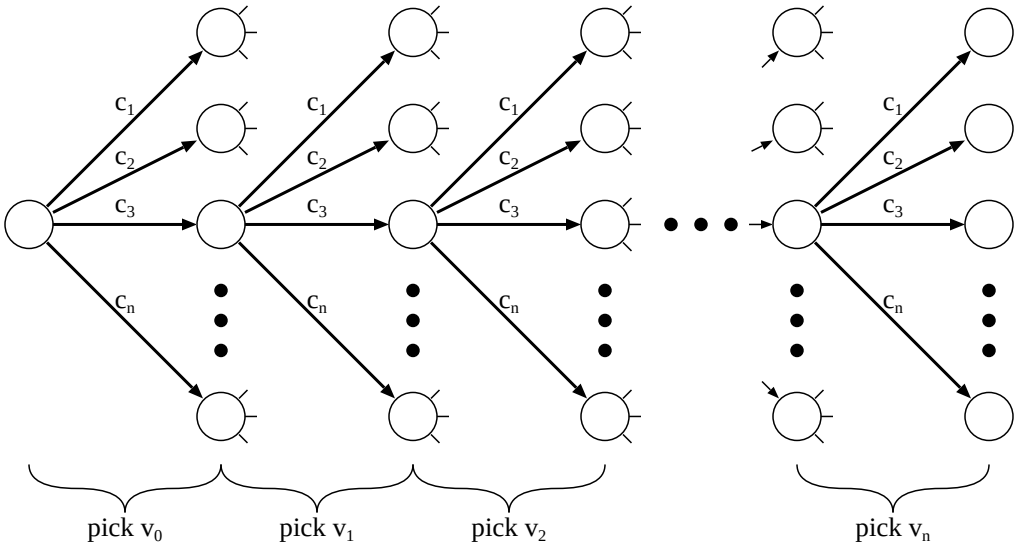

Figure 2: Computation tree of an NTM generating all possible sequences $(v_0, v_1, \ldots, v_n)$ of $n+1$ nodes in $V = \{c_1, c_2, \ldots, c_n\}$.

step. Only then does the algorithm follow our definition of nondeterministic computation. This typical two-step structure of $\mathcal{NP}$ algorithms led to another definition of $\mathcal{NP}$:

Definition 2 (Alternative Definition of $\mathcal{NP}$) *The class $\mathcal{NP}$ is the set of all languages that can be* verified *in polynomial time by a* deterministic *Turing Machine. A language L can be* verified*, if there exists some DTM M such that $L = \{ w \mid M \text{ accepts } \langle w, c \rangle \text{ for some } c \}$. The string c that is given as an additional "hint" to the deciding Turing Machine is called a* certificate*. A language can be verified in polynomial time, if machine M runs in time $O(n^k)$, where $n = |w|$ is the length of word w.*

In the Traveling Salesman Problem, a certificate would be a sequence of nodes that is a round-trip of total weight not more than k; if such a sequence exists, the deterministic verifier can check its validity in polynomial time. Note that the verifier must run in time polynomial in the length of the original input word w, not in the length of w and c combined. Therefore, the length of c must be polynomial in the length of w itself; otherwise, M could not even read its complete input word $\langle w, c \rangle$ in the time allowed.

4 $\mathcal{P}$ Versus $\mathcal{NP}$

We have seen that there is an exponential algorithm to solve the Traveling Salesman Problem – the one returned by the NTM-to-DTM conversion algorithm – but this does not prove that there is no other, more clever, algorithm that solves TSP in polynomial time. As it turns out, this question is still unsolved. No polynomial algorithm has been found yet, but the non-existence of a polynomial algorithm has not been proved yet. On a more fundamental level, it is unknown whether there are *any* problems in $\mathcal{NP}$ that do not have polynomial algorithms; in other words, it is unknown whether $\mathcal{NP} \setminus \mathcal{P} = \emptyset$, or equivalently, whether $\mathcal{P} = \mathcal{NP}$ are in fact equal sets[2] ($\mathcal{P} \subset \mathcal{NP}$ is trivially true). Currently, researchers strongly believe that $\mathcal{P} \neq \mathcal{NP}$, but the final word is not spoken yet.

Another related interesting question is whether $\mathcal{NP}$ is closed under complement, i. e., whether the complement of a language in $\mathcal{NP}$ is again in $\mathcal{NP}$. It seems that deciding a word is *in* a language is less complex than proving that a word is *not* in a language, but this has not been proved yet. We define the class co-$\mathcal{NP}$ as all languages whose complements are in $\mathcal{NP}$: co-$\mathcal{NP} = \left\{ \overline{L} \;\middle|\; L \in \mathcal{NP} \right\}$.

5 $\mathcal{NP}$-complete Problems

If $\mathcal{P}$ is really a proper subset of $\mathcal{NP}$, then there must be "hard" problems that are in $\mathcal{NP}$ but not in $\mathcal{P}$. A class of very good candidates for those are $\mathcal{NP}$-complete problems. These form the most difficult problems in $\mathcal{NP}$, in the meaning that if they can be solved in polynomial time, all other problems in $\mathcal{NP}$ can be. Before we can define these problems, we need to revisit the technique of mapping reduction.

5.1 Polynomial Reducibility

We have used mapping reductions before to show the decidability/undecidability of certain problems. We will now refine the notion of reducibility even more to be able to argue about the time complexity of problems as well. In mapping reducibility, we required the existence of a computable function f that maps instances of one problem to instances of another one. At that time,

[2]The "proof" that if $N = 1$ then $NP = P$ does not help much either.

we did not restrict the running-time of a Turing Machine computing f. As a specialization, we now consider computable functions that can be computed in a reasonable amount of time, defined as usual.

Definition 3 (Polynomial Reducibility) *Let $A, B \subset \Sigma^*$ be two decidable problems, and let $f\colon \Sigma^* \to \Sigma^*$ be a function that can be computed* in polynomial time. *Then we say that A is* polynomially reducible *to B if and only if for all $w \in \Sigma^* : w \in A \iff f(w) \in B$. In symbols, we write $A \leq_P B$, and we sometimes say that A is* not more difficult *than B.*

Apart from the requirement that f can be computed in time $O(n^k)$ for some constant k, this definition is exactly the same as the original definition of mapping reducibility. But with the additional constraint, we can show the following:

- If $Q \in \mathcal{P}$, and $P \leq_P Q$, then $P \in \mathcal{P}$ as well. To decide P in polynomial time, we compute $f(w)$ for an input word w, and run the decider for Q on $f(w)$. The mapping step takes polynomial time in the length of w, and the second deciding step takes polynomial time in the length of $f(w)$. Since $f(w)$ was written to the tape by the machine computing f, its length must also be polynomial in the length of w, say that $|f(w)| = O\big(|w|^k\big)$. This means that the deciding machine will take time $O\big(|f(w)|^l\big) = O\big(|w|^{kl}\big)$, which is polynomial in $|w|$.
- If $P \notin \mathcal{P}$, and $P \leq_P Q$, then $Q \notin \mathcal{P}$ either. This is just the contraposition of the above statement.

With this definition, we can show the tractability of a problem by polynomially reducing it *to* a known tractable problem, and we can show the intractability of a problem by polynomially reducing it *from* a known intractable problem.

We can now go back and define $\mathcal{NP}$-complete problems as the most difficult ones in $\mathcal{NP}$.

Definition 4 ($\mathcal{NP}$-complete Problems) *A problem $P \in \mathcal{NP}$ is $\mathcal{NP}$-complete, if and only if every other problem $Q \in \mathcal{NP}$ can be polynomially reduced to it: $\forall\, Q \in \mathcal{NP} : Q \leq_P P$. In other words, P is at least as difficult as any other problem in $\mathcal{NP}$.*

From the definition follows that a problem must satisfy two requirements to be $\mathcal{NP}$-complete:

1. It must be in $\mathcal{NP}$ itself, and
2. it must be at least as difficult as any other problem in $\mathcal{NP}$.

If a problem only satisfies the second condition, it is more difficult than any problem in $\mathcal{NP}$. These problems are called *$\mathcal{NP}$-hard*.

We call $\mathcal{NP}$-complete problems the most difficult in $\mathcal{NP}$, because all other problems in $\mathcal{NP}$ can be polynomially reduced to them. This implies, that if we find a polynomial algorithm for *any* $\mathcal{NP}$-complete problem, we automatically found a polynomial algorithm for every other one, showing that $\mathcal{P} = \mathcal{NP}$. This statement can be formalized in a theorem:

Theorem 1 *If P is an $\mathcal{NP}$-complete problem, and $P \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.*

The proof for this theorem follows directly from the definition of polynomial reducibility and $\mathcal{NP}$-complete problems.

This theorem also shows that $\mathcal{NP}$-complete problems are important from a theoretical point of view: If one wanted to solve the $\mathcal{P} = \mathcal{NP}$ question, one can focus on $\mathcal{NP}$-complete problems: If there is any problem in $\mathcal{NP}$ that *requires* exponential running-time, then it must be $\mathcal{NP}$-complete; if, on the other hand, one $\mathcal{NP}$-complete problem has a polynomial algorithm, all others must have one as well.

From a more practical point of view, $\mathcal{NP}$-complete problems are useful to reason about the intractability of unknown problems: If a given problem P is polynomially reducible from a known $\mathcal{NP}$-complete problem, then this is very strong evidence that P does not have an efficient algorithm and is intractable. It is no proof, however; the $\mathcal{P} = \mathcal{NP}$ question is still unsolved. Still, one could consider it a waste of time to try and find a polynomial algorithm for a problem that can be polynomially reduced from an $\mathcal{NP}$-complete one. The following theorem formalizes this by giving an alternative definition of $\mathcal{NP}$-completeness:

Theorem 2 *A problem $P \in \mathcal{NP}$ is $\mathcal{NP}$-complete, if and only if some other $\mathcal{NP}$-complete problem Q can be polynomially reduced to it: $Q \leq_P P$.*

This theorem makes reasoning about the $\mathcal{NP}$-completeness of problems much easier; it is comparable in power to the reduction technique for undecidable problems. Back then, we had to prove one problem, namely A_{TM}, to be undecidable the "hard way," by using diagonalization; from that point on, we could show other problems undecidable by reducing them from A_{TM}. Over time, as the toolbox of undecidable problems grows, proving more problems undecidable becomes easier and easier. The same situation exists here: We only have to show one problem to be $\mathcal{NP}$-complete by reducing *every other problem in $\mathcal{NP}$ to it*; afterwards, it is sufficient to reduce *one* known $\mathcal{NP}$-complete problem to an unknown one to prove its intractability. The initial problem referred to will be the satisfiability problem for boolean formulas, and we will prove it $\mathcal{NP}$-complete next time.

To wrap up the discussion of $\mathcal{NP}$, let us recall the classes of decidable languages seen so far and give an overview of the fine structure of the decidable languages in Figure 3:

- $\mathcal{P}$ is the class of problems that can be *decided* by DTMs in polynomial time. These are also (roughly) the problems that can be solved by efficient algorithms in a reasonable amount of time.

- $\mathcal{NP}$ is the class of problems that can be *verified* by DTMs in polynomial time, or that can be decided by NTMs in polynomial time. If $\mathcal{P} \neq \mathcal{NP}$, there are problems in $\mathcal{NP}$ that do not have efficient algorithms. We define that all problems in $\mathcal{NP} \setminus \mathcal{P}$ are intractable.

- co-$\mathcal{NP}$ is the set of complements of languages that are in $\mathcal{NP}$. Since $\mathcal{P}$ is closed under intersection, $\mathcal{P} \subset \mathcal{NP} \cap \text{co-}\mathcal{NP}$, but the question whether $\mathcal{NP} = \text{co-}\mathcal{NP}$ has not been answered yet.

- If $\mathcal{P} \neq \mathcal{NP}$, as is widely believed, then there exists a class of "most difficult" problems in $\mathcal{NP}$, the $\mathcal{NP}$-complete problems. In this case, the complements of $\mathcal{NP}$-complete problems are the most difficult problems in co-$\mathcal{NP}$.

- The problems in $\text{DL} \setminus \mathcal{NP}$ cannot even be verified in polynomial time; they are the most complex decidable problems and are all intractable. We call these problems $\mathcal{NP}$-hard.

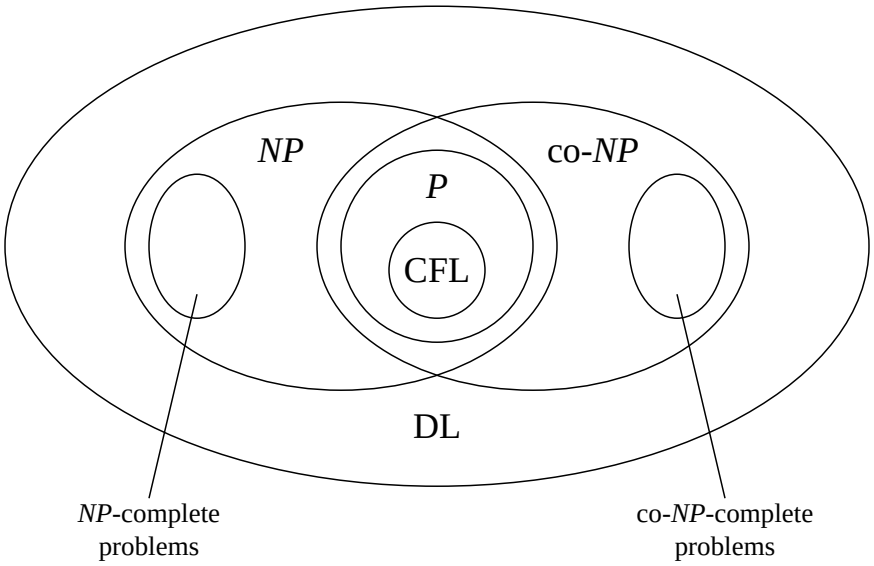

Figure 3: Fine structure of the class of decidable languages. Since all context-free languages are in $\mathcal{P}$, the lower levels of the Chomsky Hierarchy are embedded in the innermost ring.