Algorithms. Lecture Notes 9

The Notion of $\mathcal{NP}$-Completeness

Recall that reductions can be used to compare the difficulty of problems. Here comes the last central definition:

A decision problem $Y \in \mathcal{NP}$ is said to be **$\mathcal{NP}$-complete** if every (!) problem $X \in \mathcal{NP}$ is polynomial-time reducible to Y. Informally speaking, $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$. We can characterize their hardness as follows: *No $\mathcal{NP}$-complete problem belongs to $\mathcal{P}$, unless $\mathcal{P} = \mathcal{NP}$.* Assume for contradiction that some $Y \in \mathcal{P}$ is $\mathcal{NP}$-complete. Then, by definition, all $X \in \mathcal{NP}$ are polynomial-time reducible to Y. But since $Y \in \mathcal{P}$, this implies $X \in \mathcal{P}$ for all $X \in \mathcal{NP}$.

To summarize what we have shown: It is open whether $\mathcal{P} = \mathcal{NP}$ or not, but if not, then no polynomial-time algorithm can exist for $\mathcal{NP}$-complete problems. Since until now nobody could find a fast algorithm for any such problem despite decades of intensive research, it is generally believed that $\mathcal{P} \neq \mathcal{NP}$, and hence all $\mathcal{NP}$-complete problems are really hard.

$\mathcal{NP}$-completeness of any specific problems can be proved via reductions from other such problems, due to the following theorem: If Y is $\mathcal{NP}$-complete and polynomial-time reducible to $Z \in \mathcal{NP}$, then Z is also $\mathcal{NP}$-complete. This follows immediately from the definition and from transitivity of polynomial-time reducibility.

Some Frequent Misconceptions

To prove $\mathcal{NP}$-completeness of a problem Y, one must give a polynomial-timereduction **from** a known $\mathcal{NP}$-complete problem X, not a reduction from Y **to** X. Remember that Y is harder than X (more precisely: at least as hard) and not easier.

An explanation why the direction of reductions is often confused might be a misconception around the word "reduction". In every-day use, to "re-

duce" something usually means to make it smaller, and this may be misunderstood as "making the complexity smaller", but here it is the other way round! The word "transformation" would avoid this misunderstanding, but "reduction" is the established term.

Furthermore notice that polynomial-time reducibility is not a symmetric relation. If X is reducible to Y, this does in general not imply that Y is also reducible to X. A reduction goes in only one direction, but *inside* a reduction one must show equivalence of the instances, which involves two directions: (1) If x is Yes then $f(x)$ is Yes, and (2) if $f(x)$ is Yes then x is Yes. Moreover, this must hold true for every x, whereas not every instance y of Y is required to be some $f(x)$. In other words, function f is not necessarily surjective. All these aspects are easy to confuse, but this is only a matter of carefully learning the definitions.

Sometimes it is claimed in reports that $\mathcal{NP}$ means "not polynomial", which is complete nonsense. Finally, carefuully distinshuish between "$\mathcal{NP}$-problems" (that is, problems in $\mathcal{NP}$, which also includes $\mathcal{P}$), and "$\mathcal{NP}$-complete problems". Here, sloppy naming produces wrong statements.

Some $\mathcal{NP}$-Complete Problems

Still we have not seen any single NP-complete problem which could be a starting point for reductions. For one problem, $\mathcal{NP}$-completeness must be proved directly by recurring to the definition. Historically, the grandfather (or grandmother?) of all $\mathcal{NP}$-complete problems comes from logic: the Satisfiability (SAT) problem for logical formulae (or circuits). The difficulty of SAT, even when restricted to CNF formulae, may be explained as follows: If we set a variable to 1 in order to satisfy some clause, it may appear in negated form in other clauses, and then we cannot use it anymore to satisfy these other clauses. Any decisions on the truth value of some variable in a clause restrict the possibilities to satisfy other clauses. This may end up in conflicts where some clause is no longer satisfiable. Then we have to try other combinations of values, etc. In fact, nobody knows how to solve SAT in polynomial time.

Clearly, SAT belongs to $\mathcal{NP}$: If someone gives us a satisfying truth assignment, we can confirm in linear time (in the size of the formula) that it really satisfies the formula. But SAT is probably not in $\mathcal{P}$: A theorem due to S. Cook says that SAT is $\mathcal{NP}$-complete, even for CNF as input. The proof is long and very technical, but one can give the rough idea in a few sentences:

We have to show that any decision problem $X \in \mathcal{NP}$ is polynomial-time reducible to SAT. Since $X \in \mathcal{NP}$, there exists a polynomial-time algorithm that checks the validity of a given solution to an instance x of X. Like every algorithm, it can run on a machine that performs only extremely simple steps, so simple that the internal state of the machine at any time (contents of memory cells etc.) can be described by Boolean variables. A Boolean formula in CNF, of size polynomial in $|x|$, describes the steps of computation. This CNF is built in such a way that a satisfying truth assignment corresponds to the successful verification of a solution to x. Hence, the whole construction reduces X to SAT in polynomial time. (We stress that this was only a brief sketch of the proof. Don't worry if you find it cryptic. We only need the statement of Cook's theorem, but not its proof.)

Surprisingly, some further restriction does not take away the hardness of the SAT problem: A kCNF is a CNF with at most k literals in every clause. kSAT is the SAT problem for kCNF formulae. We show that 3SAT is still $\mathcal{NP}$-complete, by a polynomial-time reduction from the (more general!) SAT problem for arbitrary CNF. The reduction is best described by an iterative algorithm for the transformation. Given a CNF, we do the following as long as a clause C with more than 3 literals exist: We split the set of literals of C in two shorter clauses A, B, append a fresh variable u to A and $\bar{u}$ to B. That is, u must not occur in any other clause. It is easy to see that $(A \vee u) \wedge (B \vee \bar{u})$ is satisfiable if and only if $C = A \vee B$ is satisfiable. Hence we have got an equivalent instance of the problem. After a polynomial number of steps we are down to a 3CNF. (The hardest part is to see that the number of steps is really bounded by a polynomial. Note that the new variables blow up the formula. One needs a good argument, for example: The sum of cubes of lengths of all clauses decreases strictly.)

Note that this reduction cannot produce an equivalent 2CNF. Actually, 2SAT is in $\mathcal{P}$. It can even be solved in linear time, through a rather nontrivial graph algorithm that we cannot show here.

3SAT is an excellent starting point for further NP-completeness proofs. The limitation to three literals per clause makes it nice to handle. Next we reduce 3SAT to Independent Set, thus proving in one go the NP-completeness of Vertex Cover, Independent Set, and Clique. Let us be given an instance of 3SAT, that is, a 3CNF with n variables and m clauses. The reduction constructs a graph as follows.

(1) For each variable we create a pair of nodes (for the negated and unnegated variable), joined by an edge.

(2) For each clause we create a triangle, with the literals as nodes.

These pairs and triangles have together $2n + 3m$ nodes.
(3) An edge is also inserted between any node in a pair (1) and any node in a triangle (2) which are labeled with identical literals.

One can show that the problem instances are equivalent: The 3CNF formula is satisfiable if and only if this graph has an independent set of $k = m + n$ nodes. (This needs some thinking, but the proof steps are straightforward.)

The $\mathcal{NP}$-completeness of many problems has been established by chains of such reductions, among them the famous Traveling Salesman problem, several partioning, packing and covering problems, numerical problems like Subset Sum and (hence) Knapsack, various scheduling problems, and much more. $\mathcal{NP}$-complete problems appear in all branches of combinatorics and optimization.

$\mathcal{NP}$-Completeness: Wrap-Up

What should you (at least) have learned about $\mathcal{NP}$-completeness in a basic algorithms course? You should:

- have understood the definitions on a technical level, not only vaguely,
- have understood their relevance,
- be able to carry out *simple* reductions (doing complicated reductions is clearly something for specialized scientists in the field),
- know some representative $\mathcal{NP}$-complete problems,
- know where to find more material.

If, in practice, a computational problem is encountered that apparently does not admit a fast algorithm, it is a good idea to look up existing lists of $\mathcal{NP}$-complete problems. Maybe the decision version Y of the problem at hand is close enough to some problem X in a list, and a polynomial-time reduction from X to Y can be established. Then it is clear that Y must be treated with heuristics, with suboptimal but fast approximation algorithms, or with exact but slow algorithms.

A classical reference with hundreds of $\mathcal{NP}$-complete problems is:

Garey, Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco 1979.

Problem: Shortest Paths

Given: an undirected or directed graph $G = (V, E)$, the lengths $l(u, v)$ of all edges $(u, v) \in E$, and a start ("source") node $s \in V$.

Goal: For all nodes $x \in V$, compute the distance $d(s, x)$ from s to x, and a (directed) path of length $d(s, x)$ from s to x.

Motivations:

Motivations for the Shortest Path problem as such should be obvious. The problem as formulated above is also known as the **Single Source Shortest Paths** problem. Even if we want to know only a shortest path from s to a particular node t, in general we will have to compute the distances to *all* nodes that might appear as interior nodes on a shortest path, and we do not know these nodes beforehand. Therefore, computing the distances from the source to all nodes should not make the problem more complex.

Problem: Undirected Graph Connectivity

An undirected graph is **connected** if there exists a path between any two nodes. The **connected components** are the maximal connected subgraphs.

Given: an undirected graph $G = (V, E)$.

Goal: Decide whether G is connected. If not, compute the connected components.

Problem: Strong Connectivity in Directed Graphs

A directed graph is **strongly connected** if there exists a *directed* path from every node to every node. The **strongly connected components** are the maximal strongly connected subgraphs.

Given: a directed graph $G = (V, E)$.

Goal: Decide whether G is strongly connected. If not, compute the strongly connected components.

Motivations:

If the graph models states of a system and possible transitions between them, strong connectivity means it is always possible to recover every state, i.e., the system has no irreversible moves. The street map of a city with one-way streets should be strongly connected as well, or the traffic planners made a mistake.

Problem: Articulation Points

An **articulation point** in a connected, undirected graph is a node v such that removal of v and of the edges incident to v makes the graph disconnected.

Given: an undirected graph $G = (V, E)$.

Goal: Find all articulation points.

Motivations:

The problem is concerned with reliability (failure tolerance) of networks, e.g., communication networks. If an articulation point v in the network fails, some nodes cannot communicate with each other any more. Maybe the network was quickly built without careful design. Once we have diagnosed the articulation points, we know where additional edges should be inserted to make the network more robust.

Problem: Graph Coloring

Given a set of k colors, a **k-coloring** of a graph assigns a color to each vertex, so that adjacent vertices get different colors. A graph is **k-colorable** if it admits a k-coloring. The 2-colorable graphs are exactly the bipartite graphs.

Given: an undirected graph $G = (V, E)$ and an integer k.

Goal: Construct some k-coloring of G, or report that G is not k-colorable.

Motivations:

Imagine that a person who is not exactly an expert in botany gets a set of plants, and he is told that they belong to two different species. He does not always see whether two plants belong to the same species or not,

however, *some* pairs of plants are obviously different. Is it possible for him to divide the set correctly and efficiently? This can be translated into the 2-coloring problem: Every species (class, category, etc.) is represented by a "color". The plants (or whatever objects) are nodes of a graph $G = (V, E)$, where any two nodes that are *known* to belong to different classes are joined by an edge. The 2-colorable graphs are also called bipartite graphs.

Various problems dealing with packing, frequency assignment, job assignment, scheduling, partitioning, etc., can be considered as Graph Coloring, where the graph models pairwise conflicts. Note that Interval Partitioning problem is a special case of Graph Coloring, with the goal to minimize the number of colors: Intervals are nodes, two nodes are adjacent if the corresponding intervals overlap, and the "colors" are copies of the resource.

Problem: Detecting Directed Cycles

A **directed cycle** in a directed graph is a cycle that can be traversed respecting the orientation of the edges: $v_1, v_2, v_3, \ldots, v_n, v_1$, where every (v_i, v_{i+1}) and (v_n, v_1) is a directed edge. A *directed acyclic graph* (DAG) is a directed graph without directed cycles. DAGs should not be confused with trees which are connected graphs without *any* cycles (which are in general undirected).

Given: a directed graph $G = (V, E)$.

Goal: Find a directed cycle in G, or report that G is a DAG.

Motivations:

Directed cycles are undesirable in plans of tasks where directed edges (u, v) model pairwise precedence relations (task u must be done before task v). These tasks can be calculations in a program or logic circuit, jobs in a project, steps in a manufacturing process, etc. In such models, directed cycles indicate errors in the design.

Some systems in Artificial Intelligence, so-called partial order planners, automatically create plans to achieve some goal, given a formal description of the goal and of available actions. Part of the construction algorithms are tests for directed cycles. If such cycles are detected in a plan, some actions must be removed, and the corresponding partial goals must be realized in a different way, avoiding new cycles.

Problem: Topological Order

A **topological order** of a directed graph $G = (V, E)$ is an order of all nodes of V so that all directed edges go to the right. In other words, for every directed edge (u, v), node u appears earlier than node v in the order.

Given: a directed graph $G = (V, E)$.

Goal: Construct a topological order of G, or report that G does not admit a topological order.

Motivations:

The nodes are jobs with pairwise dependency constraints, as above. Any topological order is a possible order of executing these jobs without violating the precedence constraints.