CS 810: Complexity Theory 1/26/2010

Lecture 3: NP-Completeness

Instructor: Dieter van Melkebeek Scribe: Andrew Bolanowski

DRAFT

1 Nondeterminism

Note well: the nondeterministic model of computation is not intuitive, and for good reason. *Nondeterminism is not a physically realizable model of computation.* The utility of the model lies not in modeling actual computation, but in exactly capturing the complexity of an important class of problems.

1.1 Model

The standard model for nondeterministic computation is the nondeterministic Turing machine (NTM). An NTM has a definition very similar to a normal, deterministic Turing machine, except that the transition function δ in a TM may be a relation in an NTM. Thus, for any combination of internal state and read-head state, δ may provide multiple next-state instructions. This leads to multiple valid computation paths and possibly multiple paths that lead to an accepting state.

NTM are typically used for decision problems, so the machine will either accept or reject an input string, depending on whether the string is in the language. The language of an NTM M, $L(M)$ is the set of strings x such that there exists a valid computation of M on input x that halts and accepts. That is, we say that M accepts x if some possible computation path of M accepts x.

There are several fruitful interpretations of NTMs. We can view an NTM as a machine allowed to make guesses, and these guesses are controlled by some force that wants the machine to accept. For a particular input x, if any permissible series of guesses will lead the machine to its accepting state then the machine will make those guesses and accept x. If there is no such series of guesses, then the machine cannot accept x and is forced to reject x.

We can also think of an NTM as a massively parallel computer. Every time it makes a choice, it spawns a copy of itself for every possible branch of that choice, and all of these copies continue processing. If any child machine accepts an input, then the machine as a whole accepts that input.

1.2 Time and Space

In a NTM, it is possible that for some inputs x, some acceptance/rejection paths are very short and use little space, while other paths are much longer. NTM efficiency is stated in terms of worst-case behavior. So, "M runs in time t" means that *every* branch of the execution of M halts within t steps. Similarly, "M runs in space s" means that every branch of the execution of M uses only s cells.

The nondeterministic complexity classes NTIME(t) and NSPACE(s) are precisely analogous to their deterministic counterparts DTIME(t) and DSPACE(s): a problem is in NTIME(t) if there exists an NTM M that solves that problem and runs in time $O(t(n))$ for inputs of size n. A problem is in NSPACE(s) if there exists an NTM M that solves that problem and runs in space $O(s(n))$ for inputs of size n.

The following complexity classes are analogous to their deterministic counterparts:

- $\mathrm{NP} = \cup_{c>0} \mathrm{NTIME}(n^c)$
- $\mathrm{NE} = \cup_{c>0} \mathrm{NTIME}(2^{cn})$
- $\mathrm{NEXP} = \cup_{c>0} \mathrm{NTIME}(2^{n^c})$
- $\mathrm{NL} = \mathrm{NSPACE}(\log(n))$
- $\mathrm{NPSPACE} = \cup_{c>0} \mathrm{NSPACE}(n^c)$

We won't much talk about NPSPACE, because it turns out that PSPACE = NPSPACE, a result we'll prove later.

1.3 Universal Machines

Just as we constructed universal deterministic Turing machines, we can construct universal nondeterministic Turing machines. First notice that we can use the deterministic universal machine construction given in the first lecture here as well. This gives a single universal nondeterministic machine with logarithmic overhead in time and constant overhead in space.

For the task of simulating nondeterministic machines time-efficiently, we can do better than in the deterministic setting. If given as input the running time of the machine we wish to simulate, we demonstrate a machine that simulates the given machine with only a constant factor overhead in time.

Theorem 1. *There exists a NTM U_{NTIME} such that $t_{U_{\mathrm{NTIME}}}(\langle M, x, 0^t \rangle) = O(\mathrm{poly}(|M|) \cdot t)$ and for all $t \geq t_M(x)$, $U_{\mathrm{NTIME}}(\langle M, x, 0^t \rangle) = M(x)$.*

Proof. U_{NTIME} assumes that $M(x)$ runs in time at most t and begins by guessing a computation record for $M(x)$ and writing it down on one of its work tape. For each time step, the computation record includes a guess for each of the following: motion (either L or R) for each of M's tape heads, new cell contents under each of M's tape heads, and new internal state. The length of this computation record is $O(\mathrm{poly}(|M|) \cdot t)$.

After writing down the guessed computation record, U_{NTIME} verifies that the guessed computation record corresponds to a valid computation according to M's transition function, and that the computation ends in an accepting state. This is achieved by checking the validity of the computation for each of M's work tapes in turn. (This allows for M to have arbitarily many tapes while the universal machine as a fixed constant number of tapes. Effectively U_{NTIME} uses 1 tape for the guess and 1 tape for verification. These single tapes are reused for each tape of M.) The important point is that if there is an accepting computation path for $M(x)$ of length at most t, then at least one of U_{NTIME}'s guessed computation records causes it to accept as well. We leave it to the reader to verify the time efficiency of the simulation. $\square$

It is critical in the above proof that U_{NTIME} takes as input the amount of time to simulate M for. The construction can be modified if this information cannot be given. Namely, we run the construction above by first passing in 1 as the maximum time. If no accepting computation is found, the construction is repeated with double the amount of maximum time passed in. This is continued until an accepting computation is found. Note that if $M(x) = 1$, then this construction is correct and runs in time $O(\text{poly}(|M|)t_M(x))$. However, if there is no accepting computation path for $M(x)$, this construction never halts.

As for deterministic machines, we use the universal machines for nondeterministic machines to define a complete language - now for NP. The proof of the following is similar to the proof of Proposition 2 from last lecture.

Proposition 1. *Let K_N be the language of tuples $\langle M, x, 0^t \rangle$ such that the NTM M accepts on input x in time t. K_N is complete for NP under $\leq_m^{\log}$.*

Hierarchy results in the nondeterministic model are more difficult to achieve than in the deterministic model, as complementation is more difficult. To perform the same diagonalization for NTMs that we did on DTMs, we would need a simple way to accept an input when a simulated machine rejects that input, and vice versa. However, an NTM accepts when any of its possible computation branches accept, and rejects only when all of its possible computation branches reject. This asymmetry seems to make complementation inefficient. For example, if NTM M on input x had a path leading to an accepting state and a path leading to a rejecting state, M would accept x. Complimenting the accepting and rejecting states to make NTM N would still accept x. So N doesn't do the opposite of M. Whether complementation on NTMs can be efficiently computed is unknown; this is the NP vs. coNP problem.

Last time we introduced the model of a NTM, which we mentioned is not a realistic model of computation but nevertheless exactly captures the complexity of some important classes of problems. In particular, we started to turn our attention to the complexity class NP and briefly mentioned that this complexity class exactly characterizes those languages for which there exists an efficient[1] verification mechanism. In this lecture we continue our discussion of NP and its connection to efficient verifiability. We present the striking relationship between the complexity of almost all the problems known to be in NP. To formalize our results we use the tools we developed in the previous class, namely reducibility and completeness.

2 Efficient Verification

Definition 1. *We say that a language L has an efficient verifier if there exist $c > 0$ and $V \in P$ s.t. $x \in L \iff (\exists y \in \Sigma^{\leq |x|^c})\langle x, y\rangle \in V$.*

V refers to a verification procedure that runs in polynomial time, and y refers to a short proof (i.e, a certificate, or a witness) for the membership of x in L.

It turns out that many problems of practical interest have this property of efficient verifiability. Before giving examples, we establish the connection between these class of problems and the class NP alluded to earlier.

Theorem 2. $L \in \text{NP}$ *iff L has an efficient verifier.*

Proof. $\Leftarrow$) Call L's verifier V, and call V's TM M_V. We describe a NTM N for L that runs in polytime. By definition, there is a constant c such that whenever x is in L there is a string y of

[1] Recall that we capture efficiency in a time-bounded setting by the class P.

length at most $|x|^c$ such that $\langle x, y\rangle$ is accepted by M_V. N simply "guesses" that string y and then (with polynomial overhead) simulates M_V.

$\Rightarrow$) Call L's NTM N. By definition, a string x is in L iff N accepts x within $|x|^c$ steps on some computation path, c being a constant. Then the description of any of those accepting paths is a certificate for x. Formally, $V = \{\langle x, y\rangle |$ *y is the sequence of nondeterministic choices that N makes on an accepting branch of computation* $\}$. We see that V is in P: the TM for V simulates N on x by simulating its nondeterministic transitions in accordance with y, and accepts iff N accepts. $\square$

Now we give some examples of problems that are efficiently verifiable (i.e., that are in NP):

- A boolean formula is in "conjunctive normal form" (CNF) if it is a conjunction of disjunctions of literals, where a literal is a boolean variable or its negation. Our first example of an efficiently verifiable language is SAT = $\{$*boolean formulas ϕ in CNF* $\mid \exists$ *an assignment that satisfies ϕ*$\}$. Given a satisfiable ϕ, the certificate for its membership in SAT is any satisfying assignment, which is clearly of polynomial (in fact linear) size in the length of ϕ.

  This fundamental problem is a stripped down version of its more natural form, SEARCH-SAT, a commonly occurring problem in many fields such as artificial intelligence. SEARCH-SAT further asks, given ϕ, a satisfying assignment if one exists. Even though at first sight SAT might appear to be too simple compared to SEARCH-SAT, a closer look reveals that they are equally difficult in the sense that, given ϕ, if we know a solution for either problem then we can efficiently compute the solution for the other. So SAT exactly captures the inherent complexity of the SEARCH-SAT problem.

  We can argue this by using the notion of polynomial time oracle reductions: Clearly, SAT $\leq_o^{\mathrm{P}}$ SEARCH-SAT. We show that SEARCH-SAT $\leq_o^{\mathrm{P}}$ SAT by providing a procedure that, given ϕ, uses a SAT-oracle to compute a solution for SEARCH-SAT as follows. Query the oracle with ϕ, and if the answer is negative then ϕ is not satisfiable. Else pick any variable in ϕ, say x_o, and set it to any value, say true. Now query the SAT-oracle to see if ϕ with the value true substituted for x_o is satisfiable. If not, then set x_o to false. Proceed with the remaining variables and gradually build up a satisfying assignment for ϕ. The total number of oracle-queries is linear in the number of variables of ϕ. So SAT $\leq_o^{\mathrm{P}}$ SEARCH-SAT and SEARCH-SAT $\leq_o^{\mathrm{P}}$ SAT, which is abbreviated as SAT $\equiv_o^{\mathrm{P}}$ SEARCH-SAT.

- It is often more convenient to work with a different version of satisfiability with formulas that have more structure. For this purpose, we define the language 3SAT = $\{\phi \mid \phi$ is a satisfiable boolean formula in 3-CNF form $\}$. A formula is in 3-CNF form if it is in CNF form and each clause contains exactly three literals. We can also define SEARCH-3SAT, which will have the similar relation. 3SAT $\equiv_o^{\mathrm{P}}$ SEARCH-3SAT

- A "vertex cover" in a graph G is a subset of vertices such that every edge in G is incident to at least one vertex from that subset. The language VC = $\{\langle G, k\rangle | \exists$ *vertex cover for G of size* $\leq k\}$ is efficiently verifiable: the certificate for membership is the purported vertex cover.

  Similar to the relationship between SAT and SEARCH-SAT, VC is the decision-version of its more naturally occurring form, namely the SEARCH-VC problem, which asks, given G, a vertex cover of minimal size. There are intermediate versions of this problem as well. One can develop the problem that just asks for a vertex cover of size $\leq k\}$ for inputted k. One can develop the problemm that asks for the minimum number of vertices needed to form a vertex cover. All verstions are equivalent under polynomial reductions.

To show that VC $\equiv_o^{\mathrm{P}}$ SEARCH-VC, it is helpful to first consider an intermediate problem, namely the VC optimization problem, which asks, given G, the size of the minimal vertex cover. Using a binary search, we query the VC-oracle to find the minimum number of vertices needed. Once we find the minimum size of a vertex cover, we can find it step by step using a similar method to the SAT problem. We pick a vertex and redraw the graph as if it were in the vertex cover. Then we query and ask if there is a vertex cover of size $\leq k-1\}$ on this new graph. If yes, we keep this vertex in the cover and continue. Otherwise we restore the graph to continue. We traverse the graph until we successfully remove the k vertices that result in the empty graph. This will be the vertex cover. So, SEARCH-VC is no harder than the optimization problem.

- SUBSET-SUM = $\{\langle y_1, ..., y_n, t\rangle \mid \exists\ I \subseteq \{1, ..., n\}\ s.t.\ \sum_{i\in I} y_i = t\}$ Given a bunch of integers and a target value, is there a subset of those integers that adds up to the target?

3 The P vs NP question

The three problems above are examples of literally thousands of problems that are encountered in everyday science and engineering, all of which have equivalent (in the sense of polytime Turing reductions) decision-versions that are in NP, and none of which have a known efficient solution. This is referred to as the P vs NP question: clearly P $\subseteq$ NP, as any deterministic TM is also a NTM; but is P = NP? This is one of the most important problems of contemporary mathematics due to its philosophical and practical significance. A philosophical interpretation of this question is whether being able to efficiently verify proofs implies the ability to efficiently come up with those proofs. When stated this way its answer seems obvious, nevertheless this question remains open. On the practical side, a positive answer to this question would have utopic consequences[2] for science and engineering, but would also bring the demise of public-key cryptography, as we will see later in the semester.

4 The NP vs co-NP question

Given a complexity class $\mathcal{C}$, we define co-$\mathcal{C}$=$\{\bar{L} | L \in \mathcal{C}\}$. The NP vs co-NP question relates to the question of whether nondeterministic computations can be easily complemented. Recall from last lecture the asymmetry between a nondeterministic computation that yields a "yes" answer and one that yields a "no" answer: on a given input, an NTM rejects only if every possible computation path rejects that input. Equivalently, a co-nondeterministic computation accepts an input iff every possible computation path accepts it.

The NP vs co-NP question asks whether having an efficient NTM for a language implies the existence of an efficient NTM for the complement of that language. Similar to the P vs NP question, the answer intuitively seems to be negative: it doesn't look like we can take the question of membership in an NP-language and efficiently translate it into a question of membership in a co-NP language and vice versa. Nevertheless this problem is also open.

[2]One might question the utility of being able to solve a problem in time, say $O(n^{1000})$, but historical evidence suggests that once a problem is pulled in to the class P, it is quickly trimmed down to friendlier exponents.

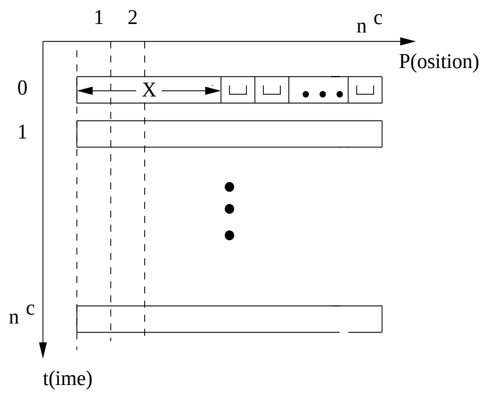

Figure 1: The computation tableau of M on input x

For example, with SAT, if there is a satisfying assignment to ϕ, it is easy to verify. If there is no satisfying assignment, we don't know of an efficent proof. The negation of an unsatisfyable statement is a tautology. So an equivalent question is whether there exists efficent proofs of tautologies.

The NP vs co-NP question relates to the topic of proof complexity, which can be summed up as the question of whether tautologies have short witnesses. Similar to the way that the satisfiability problem captures the complexity of the class NP as we discuss below, it follows that the problem of whether a formula is a tautology captures the complexity of the class co-NP. We will cover proof complexity at a later lecture.

Notice in passing that P = NP implies NP = co-NP but not necessarily the other way around.

5 NP-Completeness

It turns out nearly all the problems for which there is an efficient verification procedure but for which we don't know of an efficient solution, are different manifestations of the very same problem. That is, the inherent complexity behind them is so tightly connected that if one of them can be efficiently solved then so can all the rest.

The next theorem states the connection of the language SAT to the entire class NP under polynomial time mapping reductions. For didactic reasons, we show this result for a simplified model of computation, namely a single-tape TM with sequential access. The result extends to the random access model since we can simulate a random access machine on a sequential machine with a quadratic blowup in the running time of the former.

Theorem 3. *SAT is complete for* NP *under* $\leq_m^{\mathrm{P}}$. *Further, each bit of the mapping reduction is computable in logarithmic space and polylogarithmic time.*

Proof. (Sketch) We already showed above that SAT $\in$ NP. We now need to show that every problem in NP polynomial time mapping reduces to SAT. For this, fix a language L $\in$ NP. Let M be an NTM that decides L and that runs in time n^c on an input of length n, where c is a constant. In the rest of the proof we discuss how, for any string x, the question of x's membership in L can be efficiently translated into a question of satisfiability of a CNF formula, ϕ_x. That is, given $x \in \Sigma^n$ we construct a CNF formula ϕ_x in polytime in n such that $x \in \mathrm{L(M)} \iff \phi_x \in SAT$. We do this by considering the computation tableau of M on input x.

A *computation tableau* of M on input x depicts the contents of the tape of M in successive steps during a computation of M on x. Clearly, we need only consider the tape positions up to n^c, and time steps up to n^c.

ϕ_x comprises variables and clauses. We first describe its variables. For each time step, we have variables that describe the configuration of M at that time step. Therefore for all $0 \leq t \leq n^c$, and for all $q \in Q$, we have a boolean variable $y_{t,q}^{(state)}$ which is true iff M is in state q at time step t while computing on x. In addition, for all $0 \leq t \leq n^c$, and for all $1 \leq p \leq n^c$, we have $y_{t,p}^{(tapehead)}$ which is true iff N's tape head is located at the p^{th} position of its tape at time step t while computing on x. Also, for all $a \in \Sigma$ we have $y_{t,p,a}^{(tapehead)}$, which is true iff N's tape contains the symbol a at the pth position of its tape at time step t while computing on x.

Next we describe the clauses of ϕ_x. The clauses basically force the variables of ϕ_x to have their intended meaning. That is, we set-up these clauses so that they are satisfiable iff there is a setting of the variables that describes a valid accepting computation of M on input x. To achieve this, ϕ_x contains: i) clauses that capture the initial configuration of M, ii) clauses that express the valid transitions of M (valid tape head movements and valid evolution of tape contents at each time step, in accordance with the transition relation δ of M), and iii) clauses that express the final accepting configuration of M at time step n^c. There are many details involved in setting up these clauses, we briefly touch on some here and leave the rest to the reader.

For example, since the 0th row in the tableau corresponds to the initial configuration of M on x, we have the unit clauses $(y_{0,q_0}^{(state)} \wedge \bigwedge_{i \neq 0} \overline{y}_{0,q_i}^{(state)})$ to demand that M be at its initial state before the first step of the computation. Also associated with the initial configuration we have the unit clauses $(y_{0,0}^{(tapehead)} \wedge \bigwedge_{0<p} \overline{y}_{0,p}^{(tapehead)})$ to specify the tape head position of M. In addition, we have $(\bigwedge_{p>n} y_{0,p,\sqcup}^{(contents)} \wedge \bigwedge_{p>n, a \neq \sqcup, a \in \Sigma} \overline{y}_{0,p,a}^{(contents)})$ to require that M's tape only contain blanks beyond the input x. Further, denoting the ith symbol of x as x_i, we have $(\bigwedge_{p \leq n} y_{0,p,x_p}^{(contents)} \wedge \bigwedge_{p \leq n, a \neq x_p, a \in \Sigma} \overline{y}_{0,p,a}^{(contents)})$ to specify that the input is x.

Since the contents of a cell cannot change if the tape head was not on that cell the step before, for each $1 \leq p \leq n$ and $0 \leq t \leq n^c$ we have the clause $(y_{t,p}^{(tapehead)} \vee \bigwedge_{a \in \Sigma} (y_{t,p,a}^{(contents)} \leftrightarrow y_{t+1,p,a}^{(contents)}))$ (we don't write it in CNF form here for readability.)

For the cell on which the tape head is positioned at time (i.e, row) t, there may be multiple possibilities for the next position of the tape head and the contents of the cell depending on the transition relation δ of M. For example, if $\{(q_2, b, R), (q_3, c, L)\} \subset \delta(q_1, a)$, then for each $1 \leq p \leq n$ and $0 \leq t \leq n^c$ we have the clauses

$$\begin{aligned}&(y_{t,p}^{(tapehead)} \wedge y_{t,q_1}^{(state)} \wedge y_{t,p,a}^{(contents)}) \rightarrow \\ &(y_{t+1,p+1}^{(tapehead)} \wedge y_{t+1,q_2}^{(state)} \wedge y_{t+1,p,b}^{(contents)} \wedge \psi_R) \vee (y_{t+1,p-1}^{(tapehead)} \wedge y_{t+1,q_3}^{(state)} \wedge y_{t+1,p,c}^{(contents)} \wedge \psi_L)\ ,\end{aligned}$$

where ψ_R is $(\bigwedge_{i \neq p+1, 1 \leq i \leq n^c} \overline{y}_{t+1,i}^{(tapehead)} \wedge \bigwedge_{j \neq q_2, j \in Q} \overline{y}_{t+1,j}^{(state)} \wedge \bigwedge_{s \neq b, s \in \Sigma} \overline{y}_{t+1,p,s}^{(contents)})$ and ψ_L is left to the reader.

In the end, the formula ϕ_x contains $O(n^{2c})$ variables and is constructible in time polynomial in the length of x. The freedom of setting the variables of ϕ_x corresponds to M's freedom of making nondeterministic choices while computing on x, and thus ϕ_x is satisfiable iff there is a valid computation path of M that accepts x.

The resulting formula ϕ_x has a very simple structure. Due to this structure the reader can verify that each individual bit of the formula can be computed very efficiently, namely in time polynomial in the length of the position(address) of the bit to be computed, or equivalently, in time polylogarithmic in the size of the input x.

This establishes the mapping reduction, and the proof is complete.

□

5.1 Completeness for NQLIN

Next we present a theorem that shows a much tighter result on the complexity of reductions that connect certain problems in NP to the SAT problem. Namely, SAT is also complete for those problems in NP that can be solved in quasi-linear time, under quasi-linear time mapping reductions. Moreover, in proving the hardness part of the theorem, we do not make any simplifying assumptions on the model of computation like we did in the previous theorem.

We start by defining the complexity classes related to quasi-linear time in the obvious way.

- $\text{QLIN} = \cup_{c>0} \text{DTIME}(n(\log^c(n)))$
- $\text{NQLIN} = \cup_{c>0} \text{NTIME}(n(\log^c(n)))$

Theorem 4. *SAT is complete for NQLIN under $\leq_m^{QLIN}$. Further, each bit of the mapping reduction is computable in logarithmic space and polylogarithmic time.*

Proof. As in Theorem 3, we first show that SAT $\in$ NQLIN, then discuss how any language in NQLIN can be reduced in quasi-linear time to SAT.

First, observe that SAT can be computed in quasi-linear time by a nondeterministic random access machine that first guesses in linear time a satisfying assignment and then evaluates the formula with the guessed assignment in quasi-linear time.

To show that SAT is NQLIN-hard, we begin by making a key observation. In principle, a quasi-linear-time nondeterministic machine M can access locations on non-index tapes that have addresses of quasi-linear length. We claim that without loss of generality, we can assume that these addresses are at most of logarithmic length. The reason is that we can construct a NTM M' that simulates M with only a constant factor overhead in time and satisfies the above restriction. For each non-index tape τ of M, M' uses an additional non-index tape τ' on which M' stores a list of all (address,value) pairs of cells of τ which M accesses and that have an address value of more than logarithmic length. During the simulation of M, M' uses τ in the same way as M does to store the contents of the cells of τ with small addresses; it uses τ' for the remaining cells of τ accessed by M. M' can keep track of the (address,value) pairs on tape τ' in an efficient way by using an appropriate data structure, e.g., sorted doubly linked lists of all pairs corresponding to addresses of a given length, for all address lengths used. Note that the list of (address,value) pairs is at most quasi-linear in size so the index values M' uses on τ' are at most logarithmic. M' can retrieve a pair, insert one, and perform the necessary updates with a constant factor overhead in time by exploiting the power of nondeterminism to guess the right tape locations[3]. Thus, M' simulates M with a constant factor overhead in time and only accesses cells on its tapes with addresses of at most logarithmic length.

Next, similar to the proof of Theorem 3, with each step in a computation of M' we associate a block of boolean variables. Each block represents the following information at the beginning of a

[3]A deterministic simulation would incur a logarithmic overhead in time, which would be fine for our purposes, but would require a more involved data structure like a balanced search tree.

particular time step: i) the internal state of the machine, ii) the configuration (i.e, the contents and the tape head position) of all index tapes, iii) the tape head positions of all non-index tapes, iv) the contents of each cell that is under a tape head, and v) the transition that the machine is about to take at that step. Notice that this information is all that is needed to advance M''s execution from that particular step to the next.

Each block needs to contain only $O(\log n)$ many variables to be able to capture its intended information: i), iv) and v) can be represented by a constant number of variables, while in light of our earlier observation ii) and iii) can be described by $O(\log n)$ many variables.

We use these blocks of variables to set up clauses in such a way that the clauses are satisfied iff the blocks represent a valid accepting computation of M' on a given input. We achieve this by checking: (i) that the initial block corresponds to a valid transition out of an initial configuration of M', (ii) that all pairs of successive computation steps are consistent in terms of the internal state of M', the contents of the index tapes, and the tape head positions of all tapes that are not indexed, (iii) that the accesses to the indexed non-input tapes are consistent, (iv) that the accesses to the input tape are consistent with the input x, and (v) that the final step leads to acceptance. As in the proof of Theorem 3, conditions (i), (v), and each of the linear number of constituent conditions of (ii) can be expressed by clauses of polylogarithmic size using the above variables and additional auxiliary variables. Each bit of those clauses can be computed in polylogarithmic time and logarithmic space. All that remains is to show that the same can be done for conditions (iii) and (iv).

We check the consistency of the accesses to the indexed non-input tapes for each tape separately. For tape τ, one (inefficient) way to perform the consistency check is to look at all pairs of blocks and verifying that, if they accessed the same cell of τ, and if no other block in between accessed that cell, then the contents of that cell in the second block is as dictated by the transition encoded in the first block. While this approach captures the essence of the formulation, it isn't adequate for us due to the quadratic overhead it introduces.

This construction can be made efficient by first sorting the blocks, for each non-index tape τ, in a stable way[4] on the value of the tape head location in each block. Then we can perform the consistency check for tape τ by looking at all pairs of *consecutive* blocks and verifying that, if they accessed the same cell of τ, the contents of that cell in the second block is as dictated by the transition encoded in the first block, and if they accessed different cells, then the contents of the cell in the second block is blank. These conditions can be expressed in the same way as (ii) above. Note that a stable sort is necessary in order to maintain the order of accesses to the same tape cell.

We now construct boolean clauses that mimic a deterministic sorting procedure. Since we are avoiding quadratic overheads, we focus on any $n \log n$ sorting procedure. At this point, it may look as if we hit a block: in order to efficiently formulate a quasi-linear computation, we need to efficiently formulate another quasi-linear computation. We are not stuck, however; the latter computation is of a very specific type, one which lends itself to quasi-linear formulations, as we see next.

We formulate the sorting procedure by making use of *sorting networks*. Sorting networks are a specific type of circuit with n inputs and n outputs which, given n input values, each of length $\log n$, outputs those inputs in stable-sorted order. The circuit consists of a single type of element, called comparator element, which is basically a stable-sorting box for two elements. See Figure 2.

Without going into further detail we use the fact that there exist easily computable sorting

[4]A stable sort is one that exchanges the order of two elements only when it has to.

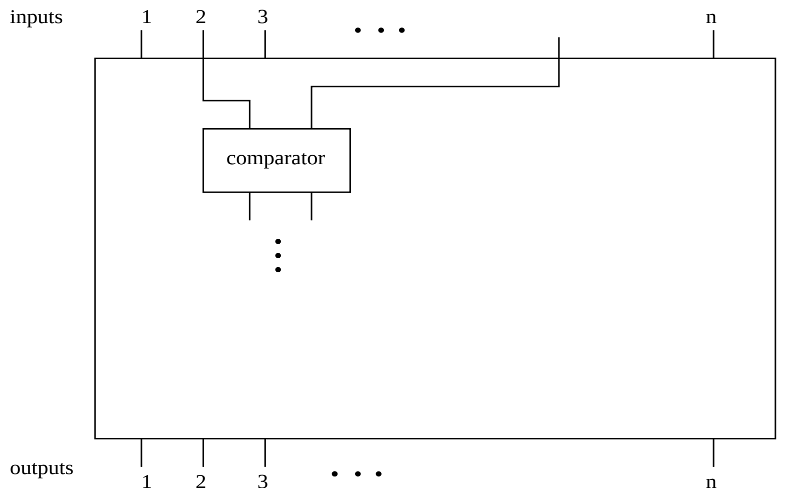

Figure 2: A simple diagram of a sorting network

networks of size $O(n \log^2 n)$. There are a number of constructions that yield this result, among which Batcher's networks are worth mentioning due to their simplicity. These are built using the merge-sort divide-and-conquer strategy, where each (so-called odd-even) merger network is constructed using another level of divide-and-conquer. We refer the reader to the algorithms book CLRS for more on sorting networks.

We associate a block of boolean variables with each connection in the network and include clauses that enforce the correct operation of each of the comparator elements of the network. The latter conditions can be expressed in a similar way as condition (ii) above. The size and constructibility properties of the network guarantee that the resulting Boolean formula is of quasi-linear size and such that each bit can be computed in polylogarithmic time and logarithmic space.

The consistency of the input tape accesses with the actual input x can be checked in a similar way as condition (iii). The only difference is that before running the stable sorting for the input tape, we prepend n dummy blocks, the ith of which has the input tape head set to location i. The approach for handling condition (iii) then enforces that all input accesses are consistent with the values encoded in the dummy blocks. Since we know explicitly the variable that encodes the ith input bit in the dummy blocks, we can include simple clauses that force that variable to agree with the ith bit of x.

So we have established a quasi-linear mapping reduction from any problem in NQLIN to the SAT problem, and the proof is complete.

□

Strikingly, it turns out that all of the known naturally occurring NP-complete problems are actually NQLIN-complete. Hence these problems in NP are really connected in a very tight sense. Ignoring polylogarithmic factors, they can all be solved in the same amount of (nondeterministic) time.

Because we have already shown that SAT is NQLIN-complete and QLIN reductions are transitive, proving NQLIN-completeness for subsequent languages is an easier task: to show L is NQLIN-complete, we must only show that SAT $\leq_m^{QLIN}$ L. In fact, each time a new problem is proved NQLIN-complete, this gives us another possible tool for proving subsequent problems NQLIN-complete. To give a flavor for these reductions, we prove NQLIN-completeness of two additional problems: 3SAT and VC.

Theorem 5. *3SAT is complete for NQLIN under $\leq_m^{QLIN}$.*

Proof. We give a quasi-linear time reduction from SAT to 3SAT. Given a formula ϕ in CNF form, all that needs to be done is to output an equivalent formula ϕ' in 3-CNF form. Clauses containing three literals in ϕ are transferred to ϕ' without modification. A clause containing only one or two literals is converted into a clause with three literals by repeating one or two of the literals already contained in the clause.

Consider a clause containing four literals: $(\ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4)$. This clause can be converted into an equivalent 3-CNF formula by introducing new variables: $(\ell_1 \vee \ell_2 \vee z) \wedge (\overline{z} \vee \ell_3 \vee \ell_4)$. This new formula is satisfiable if and only if the original clause was satisfiable. In general, a clause of the form $(\ell_1 \vee \ell_2 \vee \ell_3 \vee ... \vee \ell_k)$ is converted into the formula

$$(\ell_1 \vee \ell_2 \vee z_1) \wedge (\overline{z_1} \vee \ell_3 \vee z_2) \wedge (\overline{z_2} \vee \ell_4 \vee z_3) \wedge ... \wedge (\overline{z_{k-3}} \vee \ell_{k-1} \vee \ell_k).$$

We leave it to the reader to verify this reduction takes quasi-linear time. □

The reduction from SAT to 3SAT is quite simple as the two problems are very closely related. The following proof gives a reduction that is more typical of those required to proof NP-completeness.

Theorem 6. *VC is complete for NQLIN under $\leq_m^{QLIN}$.*

Proof. We prove VC is NQLIN-complete by reducing from SAT. Given a CNF formula ϕ, we show how to convert it into a graph G and integer k so that $\phi(x)$ is satisfiable iff the minimum size of a vertex cover in G is k.

Let ϕ have m clauses $c_1, c_2, ..., c_m$ and n variables $x_1, x_2, ..., x_n$. As with many reductions from 3SAT or SAT, the basic idea is to create a set of gadgets C for each clause and a set of gadgets X for each variable and connect them in an appropriate fashion. The gadget C_j for clause c_j is a complete graph over as many vertices as there are literals in c_j, where each vertex represents a literal in the clause. The gadget X_i for variable x_i is a two vertex complete graph, where one vertex represents x_i and the other represents $\overline{x_i}$. The gadgets are connected in the following way. Let v be a vertex representing variable x_i in a clause c_j. Then an edge is inserted into the graph connecting v with the gadget X_i. If x_i appears positively in c_j, the connection is made to the portion of the gadget representing x_i, otherwise it is made to the portion representing $\overline{x_i}$. An example of the construction is given in Figure 3.

Now consider the minimum size of a possible vertex cover. For the gadgets representing variables, at least one of the vertices must be chosen. For a gadget representing clause c_j, at least $|c_j| - 1$ of the vertices must be chosen. Then a vertex cover in the graph must have size at least $n + \sum_{j=1}^{m}(|c_j| - 1)$. In fact, there is a vertex cover of this size if and only if ϕ is satisfiable, and our reduction outputs the graph as described and $n + \sum_{j=1}^{m}(|c_j| - 1)$ as the target vertex cover size. We demonstrate the reverse implication, and leave the other direction as an exercise. Given a satisfying assignment to ϕ, we first take one vertex from each of the variable gadgets according to the assignment. Because the assignment satisfies every clause, this choice of variables already covers at least one edge going out of each clause gadget. Choosing the other $|c_j| - 1$ vertices from each clause gadget completes the vertex cover. We leave it as an exercise to verify that the graph can be generated in adjacency list form in quasi-linear time. □

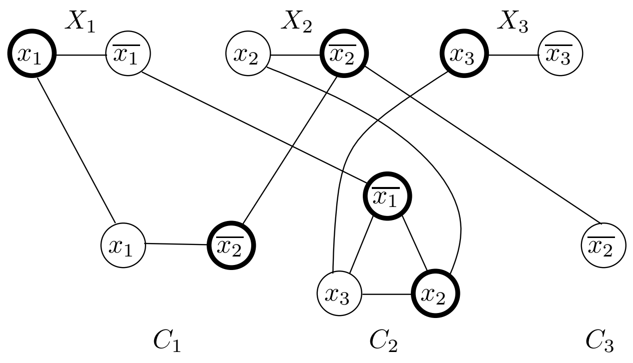

Figure 3: The graph generated by the reduction from SAT to VC for the formula $\phi = (x_1 \vee \overline{x_2}) \wedge (x_3 \vee x_2 \vee \overline{x_1}) \wedge (\overline{x_2})$. The assignment $x_1 = true, x_2 = false, x_3 = true$ satisfies ϕ, and the induced vertex cover is highlighted in the graph.

6 Next lecture

The fact that all naturally occurring NP-complete problems are also NQLIN-complete brings two questions: 1) Are there any complete problems in NP that are not in NQLIN? If there is a sufficiently strong hierarchy for nondeterministic time, then clearly there are. Recall that due to the asymmetry involved, obtaining hierarchy results for nondeterministic computation is more difficult. We will address this issue next time and obtain some results.

2) Assuming that P $\neq$ NP, are there problems in NP\P that are not NP-complete, namely are there any NP-intermediate problems? We will see that there do exist NP-intermediate problems, although they might not be natural. In fact, the general belief is that the current natural NP problems for which we do not yet know an efficient solution, and which we cannot show to be NP-complete either, such as graph isomorphism or factoring, are actually either NP-complete or efficiently solvable but we haven't been clever enough so far.