On the Existence of Dense or Sparse NP-Hard Languages

Ville Salo

2011

1 Introduction

A solution to the inclusion problem for an NP-hard language implies a deterministic polynomial time solution to every language in NP. This is what makes the existence of NP-hard, and in particular NP-complete languages interesting. The known concrete examples of NP-complete languages are too many to count, implying most puzzle types can in fact be used to implement computation of almost any kind.

One of the oldest questions in complexity theory is about the possible 'sizes' of NP-complete or NP-hard languages. In particular, for a long time it was open whether such languages can be sparse. Formally, by sparse we mean the language contains only polynomially many elements of a given size. This question can be asked for different types of reductions, and for many-one reductions, bounded-truth table reductions and Turing reductions, rather interesting results are known. In particular, Mahaney's theorem settles the case of bounded-truth table reductions and many-one reductions rather completely: with respect to these reductions, sparse languages can only be NP-hard or NP-complete if $\mathrm{P} = \mathrm{NP}$.

In this essay, we will give a proof of Mahaney's theorem for many-one reductions. The case of bounded truth-table reductions does not really require any essentially new ideas. Our proof goes somewhat along the lines of the proof for bounded truth-table reductions found in [1], although ours is considerably simpler because of the restriction to many-one reductions, and perhaps better illustrates the magic of the left set.

Before proving Mahaney's theorem, we warm up by proving the corresponding theorem for sparse co-NP-hard languages, illustrating some of the basic ideas involved. In the case of co-NP, a direct reduction from the complement of SAT to a sparse language is enough for solving SAT in deterministic polynomial time. In the proof of Mahaney's theorem, we take an arbitrary language L in NP (for instance SAT), and then construct a more easily searchable version of L, called its left set. By reducing the left set of L (which is also in NP) to a sparse language, we can then implement a deterministic polynomial time algorithm for solving L by using a kind of 'binary search in the space of all proofs'. Here, unlike in the co-NP hard case, the structure of L is not used in any way. In fact, the proof would work even if no NP-hard languages were previously known.

2 Definitions

Let $\Sigma = \{0, 1\}$.

Definition 2.1. Words are defined in the usual way, and we denote by $u \leq v$ the usual lexicographic order. We will only need to compare words of equal length. If $u \leq v$, we also say u is to the left of v.

Definition 2.2. We write Σ^n for the set of words over Σ of length n. We write $\Sigma^* = \bigcup_n \Sigma^n$, and say L is a language if $L \subset \Sigma^*$. For a language L, the *complement* of L is $L^c = \{w \in \Sigma^* : w \notin L\}$.

Definition 2.3. For $S \subset \Sigma^*$, we write $S^{\leq n}$ for the set $\{w \in S : |w| \leq n\}$.

Definition 2.4. A language L is called *sparse* if there exists a polynomial p such that $\forall n : S^{\leq n} \leq p(n)$. A language is called *dense* if it is the complement of a sparse language.

Definition 2.5. For a Turing machine A computing a partial function, we denote $A(w)$ for the word output by A when given input w.

Definition 2.6. P is the set of languages accepted by deterministic polynomial time Turing machines, and NP is the set of languages accepted by nondeterministic polynomial time Turing machines. co-NP is the set of complements of languages in NP.

Definition 2.7. We say the Turing machine A many-one reduces L to L' if

$$\forall w \in \Sigma^* : w \in L \iff A(w) \in L'.$$

If some deterministic polynomial time Turing machine A many-one reduces L to L', we write $L \leq L'$.

As stated in the following, the only reductions we consider are the deterministic polynomial time many-one reductions.

Definition 2.8. A language L' is called NP-hard if $\forall L \in \text{NP} : L \leq_m^p L'$. L is called NP-complete if it is NP-hard and is itself in NP. Co-NP-hardness and co-NP-completeness are defined similarly.

Definition 2.9. SAT is the usual satisfiability problem (as a binary language). Some natural binary encoding is used with the natural properties that assigning a value to a variable in an SAT instance ϕ cannot increase the size of ϕ, and that an SAT instance (with encoding) of length n cannot contain more than n variables. We write $\phi[x = \text{true}]$ for ϕ with x assigned the value true.

Lemma 2.10. *SAT is NP-complete.*

The following is easy to see.

Lemma 2.11. *Let L be in NP. Then there exists a language L' in P and a polynomial p such that*

$$x \in L \iff \exists w \in \Sigma^{p(|x|)} : (x, w) \in L'$$

where pairs are encoded in some suitable fashion.

Definition 2.12. Let L and L' be as in Theorem 2.11. For $x \in L$, we say w is a witness for x if $(x, w) \in L'$. Of course, the set of witnesses depends on the choice of L'.

Definition 2.13. If $L \in$ NP and L' is as given by Theorem 2.11, we say L is a *prover* for L.

Definition 2.14. Let L' be a prover for L. Then the left set K of L' is the language

$$\{(x, y) : \exists w \in \Sigma^{p(|x|)} : y \leq w \wedge (x, w) \in L'\}.$$

We also say such a K is a left set for L. The *maximal witness* w_{max} for x is the lexicographically largest w such that $(x, w) \in K$.

3 Dense Languages Cannot be NP-Hard

...unless P = NP.

The existence of an NP-hard language whose complement is sparse is of course equivalent to the existence of a co-NP-hard language that is sparse. For such a language, we have the following theorem.

Theorem 3.1. *Let S be a sparse co-NP-hard language. Then $P = NP$.*

Proof. Let q be such that $\forall m : S^{\leq m} \leq q(m)$ (definition of sparseness). Since SAT is in NP, SAT^c is in co-NP. Therefore there exists a deterministic polynomial time Turing machine A reducing SAT^c to S. Assume A operates in time $h(m)$ on inputs of length m, where h is a polynomial. Write $r(m) = q(h(m))$. Then r is a polynomial such that for each m, there are at most $r(m)$ words z in S such that A may reduce a word x of length m to z.

Given an SAT instance ϕ of length n with $m \leq n$ variables $x_1, \ldots, x_m$, the algorithm keeps track of a set U of formulas obtained by partial assignments of values to the x_i. By definition, ϕ is satisfiable if and only if one complete assignment of values gives the constant true formula. Unfortunately, this leads to an exponential amount of cases to check. However, using the reduction A we can make sure the size of U stays at size at most $r(n)$.

We say a set U of formulas obtained from ϕ by partial assignments of values to the variables x_i is *nice* if

$$\phi \in \text{SAT} \implies U \cap SAT \neq \emptyset.$$

We construct an algorithm that keeps track of a nice set of polynomial size for a linear amount of steps, while still checking all combinations of values for the variables x_i.

$U = \{\phi\}$
for $i = 1 \to m$ **do**
  $U \leftarrow \{\psi[x_i = \text{true}], \psi[x_i = \text{false}] : \psi \in U\}$ (this of course keeps U nice).
  Remove duplicates from U using NUB, keeping U nice.
  If $|U| > r(n)$, accept ϕ (stopping execution).
end for
Accept ϕ if $U \cap \text{SAT} \neq \emptyset$.

Note that the last step can easily be done in deterministic polynomial time as all variables have been assigned values to, and there are only polynomially many formulas in U.

NUB simply iterates through all pairs (ϕ_1, ϕ_2) of formulas in U, and removes either one of any two formulas ϕ_1 and ϕ_2 such that $A(\phi_1) = A(\phi_2)$. This keeps U nice because

$$\begin{aligned} A(\phi_1) = A(\phi_2) &\iff (\phi_1 \in \mathrm{SAT}^c \iff \phi_2 \in \mathrm{SAT}^c) \\ &\iff (\phi_1 \in \mathrm{SAT} \iff \phi_2 \in \mathrm{SAT}) \end{aligned}$$

by the definition of A.

Now it is clear that the algorithm runs in deterministic polynomial time, and that if $|U|$ never exceeds $r(n)$, the invariant that U stays nice makes sure the last step of the algorithm correctly determines whether ϕ is in SAT. We only need to make sure it is safe to accept ϕ if $|U| > r(n)$. But this is also clear: since NUB made sure U does not contain duplicates, A maps the instances in U to different strings. But there are only $r(n)$ words in S that A can map inputs to, and therefore at least one instance in U maps to a word outside S. But this means that for some $\psi \in U$,

$$A(\psi) \notin S \implies \psi \notin \mathrm{SAT}^c \implies \psi \in \mathrm{SAT}.$$

This concludes the construction. Since the algorithm solves SAT in deterministic polynomial time, it follows from the NP-completeness of SAT that P = NP. □

4 Sparse Languages Cannot be NP-Hard

...unless P = NP.

The approach of the previous section will not work, since the crucial step of cutting the search when U gets too large cannot be executed: at this point, given a reduction to a sparse NP-hard language, we would only know that some $\psi \in U$ is *not* satisfiable.

The problem is solved by taking an arbitrary language L in NP, and reducing its left set to a sparse NP-hard language. Then we can, in deterministic polynomial time, find a witness for a given $x \in L$, or find proof that $x \notin L$, by using the reduction from the left set to a sparse language. This is because the special form of the left set allows us to implement a kind of binary search for the maximal witness w_{max} of x.

To justify the reduction from a left set to a sparse NP-hard language, we will of course need the following.

Lemma 4.1. *Let $L \in NP$ and let K be a left set for it. Then $K \in NP$.*

Proof. Let L' be the prover for L whose left set K is. Given (x, y), the algorithm guesses a word $w \in \Sigma^{p(|x|)}$, and accepts if and only if $y \leq w$ and $(x, w) \in L'$. Such an algorithm has accepting runs for (x, y) if and only if there is a witness for x greater than of equal to y if and only if $(x, y) \in K$. Clearly it can be made to work in nondeterministic polynomial time. □

Definition 4.2. If $u, v \in \Sigma^n$, we write $[u, v]$ for the set $\{w \in \Sigma^n : u \leq w \leq v\}$.

Theorem 4.3. *Let S be a sparse NP-hard language. Then $P = NP$.*

Proof. Let L be in NP, let L' be a prover for it and let K be the left set of L'n (so $L \in \mathrm{P}, K \in \mathrm{NP}$). Let q be such that $\forall m : S^{\leq m} \leq q(m)$. By Lemma 4.1, K is in NP, and therefore by the NP-hardness of S, there exists a deterministic polynomial time Turing machine A many-one reducing K to S. Assume A operates in time $h(m)$ on inputs of length m, where h is a polynomial. Write $r(m) = q(h(m))$. Then r is a polynomial such that for each m, there are at most $r(m)$ words z in S such that A may reduce a word x of length m to z.

We describe an algorithm that, given x, determines whether $x \in L$ in deterministic polynomial time. Since L is an arbitrary language in NP, this proves P = NP. Let $|x| = n$. We say a set of disjoint intervals $\mathcal{I}$ is *nice for* x, or simply *nice*, if

$$x \in L \implies \exists I \in \mathcal{I} : w_{max} \in I,$$

where w_{max} is the maximal witness for x (which exists when $x \in L$). The algorithm keeps track of a nice set of disjoint intervals $\mathcal{I}$. The set of intervals is kept at a polynomial size for a polynomial amount of steps, and at the end of the algorithm, all the intervals will be singletons, so it will be easy to check whether w_{max} exists by simply looking through all of them. This follows from the fact that L' is in P.

The main algorithm SEARCH goes as follows.

$\mathcal{I} = \{[0^{p(n)}, 1^{p(n)}\}$
for $i = 1 \to p(n)$ **do**
  Halve every interval of $\mathcal{I}$ (which of course keeps $\mathcal{I}$ nice).
  Remove duplicates from $\mathcal{I}$ by calling NUB, keeping it nice.
  If $|\mathcal{I}| > r(n)$, reduce the size of $\mathcal{I}$ to be at most $r(n)$ using CULL, keeping $\mathcal{I}$ nice.
end for
Check whether $(x, y) \in L'$ for all $[y, y] \in \mathcal{I}$, and accept x if such y is found.

Removing duplicates means we remove redundant intervals by using the sparseness of S. This is done as follows.

Claim. *Let $[u_1, v_1]$ and $[u_2, v_2]$ be intervals in $\mathcal{I}$ such that $A((x, u_1)) = A((x, u_2))$, and $v_1 < u_2$. Then, if $\mathcal{I}$ is nice, $\mathcal{I} - \{[u_1, v_1]\}$ is nice as well.*

Proof. If $\mathcal{I}$ is nice, but $\mathcal{I} - \{[u_1, v_1]\}$ is not nice, then clearly w is defined, and $w \in [u_1, v_1]$. In particular, this means $u_1 \leq w$. But

$$A((x, u_1)) = A((x, u_2)) \implies (u_1 \leq w \iff u_2 \leq w) \implies u_2 \leq w.$$

Thus

$$v_1 < u_2 \leq w \implies w \notin [u_1, v_1],$$

a contradiction. □

Now, NUB simply goes through the list $\mathcal{I}$ in descending order, removing each interval if an interval with the same left endpoint image has been seen already. By the Claim, $\mathcal{I}$ stays nice after this.

The implementation of CULL is even simpler: After calling NUB, we take the first $r(n)$ (or all) intervals of $\mathcal{I}$ as the new value of $\mathcal{I}$. Among the first $r(n) + 1$ intervals, one of the (now unique) A-images of the left endpoints must

be outside of S, which means that w must be to the left of the left endpoint of the $(r(n)+1)$th interval. Therefore, $\mathcal{I}$ stays nice even if just its first $r(n)$ intervals are taken.

It is clear that the algorithm runs in deterministic polynomial time: There are $p(n)$ steps in the loop, and during each step, the set $\mathcal{I}$ is of size at most $2r(n)$, and the operations we execute on it are very simple. The last step of the algorithm can also be performed in polynomial time, because $\mathcal{I}$ has size at most $r(n)$, and L' is in P.

If x is in L, then it has a maximal witness w such that $(x,w) \in L'$. Since niceness is a loop invariant, after executing the loop of SEARCH, w will be contained in one of the $r(n)$ intervals in $\mathcal{I}$, which are now singleton sets. In particular, the algorithm will find that $(x,w) \in L'$ (or possibly some other witness). On the other hand, the algorithm can only accept x if a witness is found. Therefore, we have described an algorithm for deciding L in deterministic polynomial time. Since L was arbitrary, P = NP. □

References

[1] L. A. Hemaspaandra, M. Ogihara, The Complexity Theory Companion, 2002