NP-Completeness

Richard Cole

November 20, 2008

1 Polynomial Time

Definition 1 A program or algorithm has worst case running time \( T(n) \) if for all inputs of size \( n \) it completes its computation in at most \( T(n) \) steps. (Strictly, in addition, on some input of size \( n \), the full \( T(n) \) steps are performed.)

By a step we intend a basic operation such as an addition, a comparison, a branch (due to a goto, in a while loop, in an if-statement, to perform a procedure call, etc.). We do not allow more complex steps, such as \( B \leftarrow 0 \) where \( B \) is an \( n \times n \) array, to count as a single step. Finally, we also limit the word size to \( O(\log n) \) bits, so repeated squaring of a number would not be a legitimate series of steps.

Definition 2 An algorithm runs in polynomial time if its running time \( T(n) \) is bounded by a polynomial function of \( n \).

Definition 3 A problem or language is in \( P \) (polynomial time) if there is an algorithm solving the problem or deciding the language that runs in polynomial time.

We view polynomial time as corresponding to the class of problems having efficient algorithms. Of course, if the best algorithm for an algorithm ran in \( n^{100} \) time this would not be efficient in any practical sense. However, in practice, the polynomial time algorithms we find tend to have modest running times such as \( O(n), O(n \log n), O(n^2), O(n^3) \).

One reason for defining \( P \) as the class of efficient algorithms is that there is no principled way of partitioning among polynomial time algorithms. Is \( \theta(n^3) \) OK but \( \theta(n^{3.5}) \) too slow? Then what about \( \theta(n^{3.1}) \), etc.? A second reason is that the class \( P \) is closed under composition: if the output of one polynomial time algorithm is the input to a second polynomial time algorithm, then the overall combined algorithm also runs in time polynomial in the size of the original input.

Another interesting way of viewing polynomial time algorithms is that a moderate increase in resources suffices to double the size of the problems that can be solved.
For example, given that algorithm $A$ can solve a problem of size $m$ in one hour on a particular computer, if $A$ runs in linear time, then given a new computer that runs twice as fast, in the one hour one could solve twice as large a problem with algorithm $A$. By running twice as fast I mean that it performs twice as many operations in the given time. While if $A$ runs in quadratic time ($\theta(n^2)$), then give a new computer that runs four times as fast, in one hour one could solve a problem that is twice as large; again, if $A$ runs in cubic time ($\theta(n^3)$), then given a new computer that runs eight times as fast, in one hour one could solve twice as large a problem; and so forth.

This contrasts with the effect of an exponential running time ($\theta(2^n)$, for instance). Here doubling the speed of the computer increases the size of the problem that can be solved in one hour by one. As a result, exponential time algorithms are generally considered to be infeasible.

## 2 The Class NP

This class of languages is characterized by being “verifiable” in polynomial time. We begin with some examples.

**Example 4** Hamiltonian Circuit.

**Input:** A directed graph $G = (V, E)$.

**Question:** Does $G$ have a Hamiltonian Circuit, that is a cycle that goes through each vertex in $V$ exactly once?

**Output:** “Yes” or “no” as appropriate.

The “Yes” (or “Recognize”) answer is polynomial time verifiable in the following sense. Given a sequence of $n = |V|$ vertices which is claimed to form a Hamiltonian Circuit this is readily checked in linear time (it suffices to check that each vertex appears exactly once in the list and that if the list is the sequence $v_1, v_2, \cdots, v_n$, then $(v_i, v_{(i+1) \mod n}) \in E$ for each $i, 1 \leq i \leq n$).

If this sequence of vertices forms a Hamiltonian Circuit, as claimed, it is called a certificate.

By definition, a Hamiltonian graph has a certificate, namely the list of vertices forming (one of) its Hamiltonian Circuit(s). On the other hand, if the graph is not Hamiltonian, i.e. it does not have a Hamiltonian Circuit, then no attempted certificate will check out, for no proposed sequence of vertices will form a Hamiltonian Circuit. However, it might be that you are given a proposed certificate for a Hamiltonian graph which fails because it is not a Hamiltonian Circuit. This failure does not show anything regarding whether the graph is Hamiltonian; it only shows that the sequence of vertices did not form a Hamiltonian Circuit for this particular graph. You cannot conclude whether or not the graph is Hamiltonian from this.

**Example 5** Clique.
2 THE CLASS NP

Input: \((G, k)\) where \(G\) is an undirected graph and \(k\) an integer.

Question: Does \(G\) have a clique of size \(k\)? A clique is a subset of vertices such that every pair of vertices in the subset is joined by an edge.

Output: “Yes” or “no” as appropriate.

Again the “Yes” (or “Recognize”) answer is polynomial time verifiable, given the following additional information, or certificate: a list of \(k\) vertices which are claimed to form a clique. To verify this it suffices to check that the list comprises \(k\) distinct vertices and that each pair of vertices is joined by an edge. This is readily done in \(O(k^2n)\) time, and indeed in \(O(n^3)\) time (for if \(k > n = |V|\), then the graph does not have a \(k\)-clique).

Again, if the graph does not have a \(k\)-clique, then any set of \(k\) vertices will fail to be fully connected; there will be some pair of vertices with no edge between them. So any proposed certificate will fail to demonstrate that the graph has a \(k\)-clique.

However, just because you are given a set of \(k\)-vertices that does not happen to form a \(k\)-clique, you cannot then conclude that the graph has no \(k\)-clique.

Example 6 Satisfiability.

Input: \(F\), a CNF Boolean formula.

\(F\) is in CNF form if \(F = C_1 \land C_2 \land \cdots \land C_m\), where each clause \(C_i, 1 \leq i \leq m\), is an ‘or’ of literals:

\[ C_i = l_{i1} \lor l_{i2} \lor \cdots \lor l_{ij_i} \]

where each \(l_{ik}\) is a Boolean variable or its complement (negation).

e.g. \(F_1 = (x_1 \lor x_2) \land (\overline{x}_1 \lor x_2 \lor x_3) \land (x_3 \lor x_2); F_2 = x; F_3 = x \land \overline{x}\).

Question: Is \(F\) satisfiable? That is, is there an assignment of truth values to \(F\)’s variables that causes \(F\) to evaluate to True?

e.g. For \(F_1, x_1 = \text{True}, x_2 = \text{False}, x_3 = \text{True}, F_1\) evaluates to True; For \(F_2, x = \text{True}\) causes it to evaluate to True; for \(F_3\), no setting of the variables causes it to evaluate to True.

Here, the additional information, or certificate, is the assignment of truth values to the formula’s variables that cause the formula to evaluate to True. Notice that if the formula never evaluates to True, then any proposed truth assignment for the boolean variables will cause the formula to evaluate to False. In other words, any proposed certificate fails.

Again, just because you are given a truth assignment that cause the formula to evaluate to False, you cannot then conclude that the formula is not satisfiable. All you know is that this particular collection of truth assignments to the Boolean variables did not work (i.e. they caused the Formula to evaluate to False in this case).

Definition 7 A language \(L \in \textbf{NP}\) if there are polynomials \(p, q\), an algorithm \(V\) called the verifier, and there is a certificate \(C(x)\) of length at most \(p(|x|)\) such that
V(x,C(x)) outputs “Yes”, while if x \notin L, for every string s of length at most p(|x|), V(x,s) outputs “No”; and finally, V runs in time q(|x| + |C(x)|) = O(q(p(n)))

Comment If x \notin L there is no certificate for x; in other words, any string claiming to be a certificate is readily exposed as a non-certificate.

However, a non-certificate does not determine whether x \in L or x \notin L.

2.0.1 Reductions among NP Problems

We begin with a number of reductions among pairs of problems in NP. The basic form is the following. Given a polynomial time decision or membership algorithm for language A we use it as a subroutine to give a polynomial time algorithm for problem B. Recall that a decision algorithm reports “Recognize” (“Yes”) or “Reject” (“No”) as appropriate.

Example 8 Independent Set.

Input: Undirected Graph G, integer k.
Question: Does G have an independent set of size k, that is a subset of k vertices with no edges between them?

Claim 9 Given a polynomial time algorithm for Clique there is a polynomial time algorithm for Independent Set.

Proof: Let G = (V,E) be the input to the Independent Set problem. Consider the graph \overline{G} = (\overline{V}, \overline{E}). We note that S \subseteq V is an independent set in G if and only if it is a clique in \overline{G}. (For if a subset of k vertices have no edges among themselves in graph G, then in graph \overline{G} all possible edges between them are present, i.e. the k vertices form a k-clique in \overline{G}. The converse is true also: a set of k vertices forming a k-clique in \overline{G}, also form an independent set in G.

Thus the Independent Set algorithm is simply to run the Clique algorithm on the pair (\overline{G}, k), and report its result.

Notation. We will use the name of a problem to indicate the set of objects satisfying the problem condition. Thus Indpt-Set = \{(G,k) | G has an independent set of size k\}.

The algorithm A_{IS}, for Independent Set, performs 3 steps.

1. On input (G,k), A_{IS} computes an input for the Clique algorithm, A_{Clique}; this is the input (\overline{G}, k).

2. It runs A_{Clique}(\overline{G}, k).

3. It uses the answer from Step 2 to determine its own answer (the same answer is this case).

Demonstrating the algorithm is correct also entails showing:
4. $A_{IS}$’s answer is correct. In this case that means showing that

$$(G, k) \in \text{Indpt-Set} \iff (\overline{G}, k) \in \text{Clique}$$

5. Assuming that $A_{Clique}$ runs in polynomial time, then showing that $A_{IS}$ also runs in polynomial time.

Often, we will argue Step 4 right after Step 1 in order to explain why the constructed input makes sense.

The next claim is similar and is left to the reader.

**Claim 10** Given a polynomial time algorithm for Independent Set there is a polynomial time algorithm for Clique.

**Example 11** Hamiltonian Path (HP).

Input: $(G, s, t)$, where $G = (V, E)$ is a directed graph and $s, t \in V$.

Question: Does $G$ have a Hamiltonian Path from $s$ to $t$, that is a path that goes through each vertex exactly once?

**Claim 12** Given a polynomial time algorithm for Hamiltonian Circuit (HC) there is a polynomial time algorithm for Hamiltonian Path.

**Proof:** Let $(G, s, t)$ be the input to the Hamiltonian Path Problem. $A_{HP}$, the algorithm for the Hamiltonian Path problem, proceeds as follows.

1. It builds a graph $H$ with the property that $H$ has a Hamiltonian Circuit exactly if $G$ has a Hamiltonian Path from $s$ to $t$.

2. It runs $A_{HC}(H)$.

3. It reports the answer given by $A_{HC}(H)$.

$H$ consists of $G$ plus one new vertex, $z$ say, together with new edges $(t, z), (z, s)$.

4. We argue that $G$ has a Hamiltonian Path from $s$ to $t$ exactly if $H$ has a Hamiltonian Circuit. For if $H$ has a Hamiltonian Circuit it includes edges $(z, s), (t, z)$ as these are the only edges incident on $z$; we write $s = v_1$ and $t = v_n$. So the circuit has the form $z, v_1, v_2, \cdots, v_n$, and then $v_1, v_2, \cdots, v_n$ is the corresponding Hamiltonian Path in $G$. Conversely, if $G$ has Hamiltonian Path $v'_1, v'_2, \cdots, v'_n$, where $s = v'_1$ and $t = v'_n$, then $H$ has Hamiltonian Circuit $z, v'_1, v'_2, \cdots, v'_n$.

5. Clearly $A_{HP}$ runs in polynomial time if $A_{HC}$ runs in polynomial time.

**Example 13** Degree $d$ Bounded Spanning Tree (DBST).
Input: \((G, d)\), where \(G\) is an undirected graph \(G\) and \(d\) is an integer.

Question: Does \(G\) have a spanning tree \(T\) such that in \(T\) each vertex has degree at most \(d\)?

**Example 14** Undirected Hamiltonian Path (UHC).

Input: \((G, s, t)\), where \(G = (V, E)\) is an undirected graph, and \(s, t \in V\).

Question: Does \(G\) have a Hamiltonian Path from \(s\) to \(t\), that is a path going through each vertex exactly once?

**Claim 15** Given a polynomial time algorithm for Degree \(d\) Bounded Spanning Tree there is a polynomial time algorithm for Undirected Hamiltonian Path.

**Proof:** Let \((G, s, t)\) be the input to the Hamiltonian Path Problem. \(\mathcal{A}_{\text{UHP}}\), the algorithm for the Hamiltonian Path problem, proceeds as follows.

1. \(\mathcal{A}_{\text{UHP}}\) builds a graph \(H\) with the property that \(H\) has a degree 3 bounded spanning tree exactly if \(G\) has a Hamiltonian Path from \(s\) to \(t\).

2. It runs \(\mathcal{A}_{\text{DBST}}(H)\).

3. It reports the answer given by \(\mathcal{A}_{\text{DBST}}(H)\).

\(H\) is the following graph: \(G\) plus vertices \(v'\) for each \(v \in V\), plus vertices \(s'', t''\), together with edges \((v', v)\) for each \(v \in V\) and edges \((s'', s), (t'', t)\).

4. We argue that \(G\) has a Hamiltonian Path from \(s\) to \(t\) exactly if \(H\) has a spanning tree with degree bound 3. Note that all the new vertices in \(H\) have degree 1 and consequently all the new edges must be in any spanning tree \(T\) of \(H\). This means that the remaining edges in \(T\) form a Hamiltonian Path in \(G\) from \(s\) to \(t\), as each of \(s\) and \(t\) can have at most one more incident edge, and edge \(v \neq s, t\) can have at most two more incident edges; further, to achieve connectivity, these additional edges must be present.

5. Clearly \(\mathcal{A}_{\text{HP}}\) runs in polynomial time if \(\mathcal{A}_{\text{DBST}}\) runs in polynomial time.

**Reductions** We observe that all these constructions have the following form: given a polynomial time algorithm \(\mathcal{A}_B\) for membership in \(B\) we construct a polynomial time algorithm \(\mathcal{A}_A\) for membership in \(A\) as follows.

Input to \(\mathcal{A}_A\): \(I\).

1. Construct \(f_{A \leq B}(I)\) in polynomial time.

2. Run \(\mathcal{A}_B(f_{A \leq B}(I))\).
3. Report the answer from Step 2.

We denote this construction by $A \leq_p B$, or $A \leq B$ for short. It is read as: $A$ can be reduced to $B$ in polynomial time.

In order for Step 3 to be correct we need that:

$$I \in A \iff f_{A\leq B}(I) \in B.$$ 

As $f_{A\leq B}(I)$ is computed in polynomial time it produces a polynomial sized output. Thus $A_B(f_{A\leq B}(I))$ runs in time polynomial in $|I|$, as $A_B$ runs in polynomial time. This uses the fact that if $p$ and $q$ are bounded degree polynomials, then so is $p(q(\cdot))$.

This is called a polynomial time reduction of problem $A$ to problem $B$. It shows that if there is a polynomial time membership (decision) algorithm for $B$ then there is also a polynomial time membership algorithm for $A$. The converse is true also: if there is no polynomial time membership algorithm for $A$ then there is not one for $B$ either.