Advanced algorithms
NP Completeness Notes

Søren Dahlgaard

June 15, 2011

1 Disposition

1. Formal notion. Relation Q from $I \to S$. We want $S = \{0, 1\}$
2. Decision problems - easier argumentation.
3. Encodings. Input length vs size.
4. Languages. Problem is defined by $L = \{x \in \Sigma^* : Q(x) = 1\}$.
5. Accept, decide. P
6. Polynomial-time verification, NP, $co - NP$.
7. Definition of NP-Complete. How do we show this?
8. Reductions and proofs
9. Sketch of circuit-sat
10. Clique example.

2 Formal notion

We first want a formal notion of what a "problem is". An abstract problem is a relation Q between a set of instances I and a set of solutions S. The relation is not injective, as eg. an instance of the SHORTEST-PATH problem might have several solutions.

We only concentrate on decision problems. Where $S = \{0, 1\}$. A such problem for SHORTEST-PATH would take an additional integer k and will return whether there is a path shorter than k length.

When we have an optimization we can usually make a decision problem by saying "is there a solution at least as good as k" for some k (not necessarily an integer).

2.1 Encodings

We want our problems on computers, so we want our problem instances encoded as binary strings. That is, we want an encoding $e : I \to \{0,1\}^*$. A problem for which $I = \{0,1\}^*$ is called a concrete problem. Note that some instances might not make sense.

We talk about running time in terms of the size of encodings. Eg. if an algorithm takes an integer k and runs in $\Theta(k)$. If the encoding is unary (a string of 1's) then the running time is $O(n) = |i|$. If i is encoded in binary the running time is $O(2^n) = 2^{|i|}$. This is running time as a function of *input length* as opposed to *input size* which could be eg. amount of vertices. Note that input size and input length with reasonable encodings are often polynomially related, so we will not talk more about this.

We call two encodings polynomially related if we can compute one from the other in polynomial time. For instance we can compute a base 3 integer from a base 2 integer in polynomial time.

If some e_1, e_2 are polynomially related, then $e_1(Q) \in P \Leftrightarrow e_2(Q) \in P$. We can compute $e_1(i)$ from $e_2(i)$ in polynomial time $O(n^c)$. Then if $|e_2(i)| = n$ we must have $|e_1(i)| \leq n^c$ as the output can't be bigger than the running time. If $e_1(Q)$ is polynomially solvable we can solve $e_1(i)$ in $|e_1(i)|^k$, so we get $O(n^{ck})$. □

We assume problem instances are encoding in reasonable fashion (eg. not unary). We denote a standard encoding with angle brackets. For a graph G the standard encoding is therefore $\langle G \rangle$.

2.2 Languages

We use the alphabet $\Sigma = \{0,1\}$ as well as the empty string ε. The set of all binary strings is denoted

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$$

We concentrate on languages $L \subseteq \Sigma^*$. We can do various operations on these languages:

- $L_1 \cup L_2 = \{x : x \in L_1 \vee x \in L_2\}$
- $L_1 \cap L_2 = \{x : x \in L_1 \wedge x \in L_2\}$
- $\overline{L} = \Sigma^* \setminus L$
- $L_1 L_2 = \{x_1 x_2 : x_1 \in L_1 \wedge x_2 \in L_2\}$
- $L^* = \{\varepsilon\} \cup L_1 \cup L_2 \cup \ldots$

If we look at a decision problem Q we can view it as a language:

$$L = \{x \in \Sigma^* : Q(x) = 1\}$$

For instance the language for PATH would be $\langle G, u, v, k \rangle$ for all graphs G where there exists a path from $u \rightsquigarrow v$ of at most k length.

An algorithm A accepts a language $L = \{x \in \Sigma^* : A(x) = 1\}$. And rejects a string if $A(x) = 0$. Note that it might not necessarily reject all strings $x \notin L$

(it might loop forever). An algorithm decides a language L if for any $x \in L$ it correctly decides whether $A(x) = 1$. It must therefor accept or reject any string in L. We also talk about polynomial time accept/decision.

We can define P as $P = \{L \subseteq \{0,1\}^* : \text{ There exists an algorithm that decides } L \text{ in polynomial time.}$

It is also works if we replace "decides" with "accepts". The idea is that if we have an algorithm A that accepts x in $O(n^k)$, we can construct an algorithm that runs A for $O(n^k)$ time and if it hasn't returned halts it and return 0. Note that we might not be able to construct this algorithm A' very easily.

3 Polynomial-time verification

We want to define the complexity class NP as the class of languages for which we can verify that a solution is correct in polynomial time.

Imagine that along with an instance $\langle G, u, v, k\rangle$ of PATH, we're also given a path P from $u \rightsquigarrow v$. It is easy to verify if this path satisfies the constraints of the problem. For each $e \in P$ check that $e \in G.E$ and check that $\sum_{e=(u,v)\in P} w(u,v) \leq k$.

We define such a verification algorithm $A : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ as an algorithm that takes a problem instance x and a syndicate y and outputs whether y is a solution to x or not. A verification algorithm A verifies a language:

$$L = \{x \in \{0,1\}^* : \text{ there exists } y \in \{0,1\}^* \text{ such that } A(x,y) = 1\}$$

We say that a language L belongs to NP if there exists a polynomial-time verification algorithm A and a constant c, such that:

$$L = \{x \in \{0,1\}^* : A(x,y) = 1 \text{ for some } y \text{ with } |y| = O(|x|^c)\}$$

It is obvious that $P \subseteq NP$. We also have a complexity class co-NP that consists of $L : \overline{L} \in NP$. We also have $P \in$ co-NP.

4 NP-Complete and reductions

We use reductions to reduce a problem instance to an instance of another probem. We say that a problem Q can be reduced to another problem Q' if any instance can be easily rephrased. For example a linear equation $ax + b = 0$ can be rephrased as a quadratic equating easily by adding $0x^2$: $0x^2 + ax + b = 0$.

If a language L_1 can be reduced to another language L_2 in polynomial-time we write $L_1 \leq_p L_2$. More formally we have a function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in L_1 \Leftrightarrow f(x) \in L_2$. Because we only discuss decision problems we can easily produce an answer to L_1 from an answer to L_2. It is obvious that if $L_2 \in P$ and $L_1 \leq_p L_2$ then $L_1 \in P$.

We say that a language L is NP-Complete if:

1. $L \in NP$
2. $\forall L' \in NP : L' \leq_p L$

If (2) is satisfied for a language L but not necessarily (1), we say the language is NP-Hard. It is obvious that if any NP-Complete problem is in P then $P = NP$.

4.1 Circuit satisfiability

We will show that this problem is NP-Complete and then use reductions to show that other problems are.

The decision problem we look at is:

$$\text{CIRCUIT-SAT} = \{\langle C \rangle : C \text{ is satisfiable boolean combinational circuit}\}$$

We say that the size of such a problem is the amount of AND, NOT and OR gates as well as the amount of wires.

We have that the language belongs to NP. Simply compute the gates' output values according to the syndicate provided. This can be done in linear time. We must also note that $|y| = O(|x|^c)$ which is clear because $|y|$ only provides the value for some wire, but the amount of wires is strictly less than $|x|$.

Proof. $L \in NP$ so we must have a verification algorithm A that runs in polynomial time.

The idea is now to create a combinational circuit that "simulates" A.

1. Let $n = |x|$ and let $T(n)$ be the running time of A.
2. Let c_i be the configuration ith configuration of A. That is the program counter, x, y, the state, etc.
3. Let M be the the circuit that implements the computer hardware maps each configuration c_i to the next c_{i+1}.
4. If A runs in $T(n)$ steps then its output must be somewhere in $c_{T(n)}$.
5. Create a combinational circuit consisting of $T(n)$ instances of M. Hardwire the input x and leave space for input y. Ignore all output but the one bit that specifies the answer of A.

Suppose that there exists a y with $|y| = O(n^k)$. Then $C(y) = A(x, y) = 1$. If there is input y such that $C(y) = 1$ then we must also have $A(x, y) = 1$.

For the running time. A has constant size independent of $|x|$. Also the size needed to represent c_i is polynomial in n (otherwise A wouldn't be polynomial-time). The

Assuming that constructing M takes polynomial time we can construct the entire ciruit in polynomial time because we need to construct $O(n^k)$ copies of M. □

5 NP-Completeness proofs

If a language $L' \in NPC$ exists so $L' \leq_p L$ then L is NP-Hard. This is easy to show.

In order to prove that a language L is NP-Complete we therefore need only to show:

1. $L \in NP$
2. Pick a language L' that is NP-Complete and prove that $L \leq_p L'$. Ie:

3. show a function f that maps instances of L' to instances of L.

4. Show for f that $x \in L' \Leftrightarrow f(x) \in L$.

5. Show that f runs in polynomial time.

5.1 SAT

This problem is much like Circuit-Sat. We use AND, NOT, OR, implication and bi-implication.

To show that SAT $\in NP$ we simply make an algorithm A that assigns the values and checks the result.

In order to produce a formula from a circuit we could recursively express the formula of the final gate by the values of the input wires. This is however exponential. Instead we create a variable for each wire. If we have an AND-gate with x_1, x_2 as inputs we define variable x_3 and create the formula $(x_3 \Leftrightarrow (x_1 \wedge x_2))$. We do this for each gate and connect then with $\wedge$. We lastly add the final output wire to the series of $\wedge$.

See example p. 1081. It is easy to see that this formula is satisfiable iff the circuit is.

5.2 3-CNF

3-CNF is a formula which is a series of AND clauses each of which is an OR clause of exactly three distinct variables.

Proof. The following three step algorithm changes any formula to a 3-CNF:

1. Create a binary parse tree of the formula. Note that $(x_1 \vee x_2 \vee x_3)$ is the same as $(x_1 \vee (x_2 \vee x_3))$. Therefore we can create a binary tree.

2. View the tree as combinational circuit and create a formula like before. Note that this formula will have at most 3 literals in each clause. This finishes step 1.

3. For each clause in the formula from step 1 ϕ_i we create a formula equal to $\neg\phi_i$. We do this by writing the truth table. It will have at most 8 lines so we will get a formula of most $O(1)$ clauses for each ϕ_i. It will also give clauses of the form $y_1 \vee y_2 \vee y_3$ where each y_j is an or clause with at most three variables.

4. Apply demorgans laws to get 3-CNF. If a clause has only 2 variables just add a variable p: $(x_1 \vee x_2 \vee p) \wedge (x_1 \vee x_2 \vee \neg p)$. If there's only one variable add an extra var q.

Because each step preserves satisfiasbility it is clear that SAT $\Leftrightarrow$ 3-CNF.

We already know that step 1 is done in polynomial time. We saw that step 2 is at most 8 times as big, so it is also in polynomial time. The last step can be done in linear time. □

6 More proofs zzz

In this section we show that some problems can be reduced to others, blabla. I have not given verification algorithms, but these are easy.

6.1 Clique problem

Given a graph G find the largest k such that a subgraph $V' \subseteq V$ has edges for all $u, v \in V'$. The related decision problem is given a k to return if a clique of size k exists.

Proof. We wish to show 3-CNF-SAT $\leq_p$ CLIQUE.

For each ϕ_i in the formula ϕ we create three vertices in a graph G. We put an edge between two vertices u, v if the following two holds:

- u, v are in different triples (clauses in the 3-CNF).
- u is not the negation of v in the 3-CNF.

This graph can easily be built on polynomial time.

We want to show that G has a clique of size k (the amount of clauses in ϕ) iff ϕ is satisfiable: If ϕ is satisfiable there is a variable from each clause such that these variables are consistent. If we pick them as a clique they must therefore have edges between all of them. If there is a clique of size k we know that there can be at most one vertex from each triplet. Therefor we must have a vertex from each triplet and they make up a solution. □

6.2 Vertex cover problem

Given a graph $G = (V, E)$ we wish to find a set $V' \subseteq V$ such that for any edge $(u, v) \in E$ either u or v is in V' minimizing the size of V'. The decision problem is to check if a V' of size k exists.

Proof. Let $\overline{G}$ define the complement of G, which is $\overline{E} = \{(u, v) : u, v \in V, u \neq v, (u, v) \notin E\}$ and the same V.

The claim is that for an instance $\langle G, k\rangle$ of CLIQUE, the instance $\langle \overline{G}, |V| - k\rangle$ of VERTEX-COVER is a reduction. Obviously this can be done in polynomial time.

If V' is a clique of size k in $\overline{G}$ then $V \setminus V'$ is a vertex-cover of G. For any edge $(u, v) \in \overline{E}$ we have that either u or v is in $V \setminus V'$, so all edges in $\overline{E}$ must be covered. If not we would have $u, v \in V'$ but not $u, v \in E$.

Conversely if $V \setminus V'$ is a vertex cover. If $u, v \in V$ and $u, v \notin V'$ then we must have $(u, v) \notin \overline{E}$ and therefore $(u, v) \in E$. □

6.3 Hamilton Cycle

The problem is whether a simple path exists that visits each vertex exactly once. We wish to show VERTEX-COVER $\leq_p$ HAM-CYCLE

The idea is to create a graph consiting of $|E$ widgets (see fig. p. 1092). Each of these widgets has exactly simple three paths through it. For each vertex we create a path between $(u, u^{(i)}, 6), (u, u^{(i+1)}, 1)$ where $u^{(i)}$ is the ith vertex connected to u. This way, if we select u to be in the vertex cover we can make

a path through all of widgets corresponding to edges u are connected to. If u covers all edges we can thus make a hamilton path through all the vertices by starting in $(u, u^{(1)}, 1)$. We also add k selector vertices and connect each one to $(u, u^{(1)}, 1)$ and $(u, u^{(\text{degree}(u))}, 6)$.

Proof not added as it is not pensum.

6.4 TSP

Decision problem is a tour of length at most k.

Given any instance of hamilton cycle we create the graph $c(i,j) = 0, (i,j) \in E$ and $c(i,j) = 1$ otherwise. If there is a TSP tour of length 0 there is a hamilton cycle.

6.5 Subset-Sum

Given a set of integer S and an integer $t > 0$. Is there a subset $S' \subseteq S$ which elements sum to t?

We want to show 3-CNF-SAT $\leq_p$ SUBSET-SUM. Assume no clause contains both x_i and $\neg x_i$. Also each x_i is in at least one clause.

Proof. Create an instance of SUBSET-SUM as follows:

1. For each $x_i \in \phi$ create two base 10 numbers with $n + k$ digits like so:
   - Set the n most significant digits to 0 except for digit i. Set this to one.
   - Let each of the k least significant digits correspond to a clause in ϕ. If $x_i \in C_k$ then set the least significant digit to 1, if $x_i \in C_{k-1}$ set the second least significant digit to 1, etc.
     For the other variable if $\neg x_i \in C_k$ set the least significant digit to 1, etc.
     Set all other digits to 0.
   - Call these vars v_i and v_i'
2. For each clause C_j create two variables with all digits to zero except the $k - j$th least significant digit. One variable has a 1 here, one has a 2. Let these vars be s_1 and s_1' (s for slack).

These are unique, which is very important because S is a set!

Not that if we sum all these numbers no digit will be greater than 6 (3 variables per clause plus the 3 from the two slack variables). We can therefore just use any base strictly greater than 6 and we won't get carries when adding.

The claim is now, that if we set t to be n 1's followed by k 4's then there will be $S' \subseteq S$ that sums to t iff ϕ is satisfiable.

Assume ϕ has a satisfying assignment. If $x_i = 1$ in this, include v_i in S' otherwise include v_i'. If we now sum up S' we will have 1's in the first n digits (obviously). Because all clauses are satisfied there must be at least 1 in the last k digits as well. We can now just add slack variables untill we get 4 in each of those digits.

Suppose that there is $S' \subseteq S$ that sums to t. There must be exactly one of v_i or v_i' in S' for each i. Also because the slack variables sum to 3 each clause is satisfied. □