Computability and Complexity - Assignment 1

Mikkel Willén

mw@di.ku.dk

Collaborators: Caroline Amalie Kierkegaard, qlj556 and Jonas Flach-Jensen, sjm233

February 23, 2024

Task 1

If a witness y of length $|y| \leq p(|x|)$ for which $M(x, y) = 1$ gives a language $L \in \mathbf{NP}$, then a witness y_p which is y padded with random garbage such that $|y_p| = p(|x|)$ for which $M(x, y_p) = 1$ gives the same language $L \in \mathbf{NP}$.

Task 2

We first prove L is in $\mathbf{NP}$.

We need to show, that we can verify a result in polynomial time. We construct a Turing Machine as the following:

$$M(n, \langle p, q \rangle) = \begin{cases} 1 & \text{if } n = p \cdot q \\ 0 & \text{otherwise} \end{cases}$$

where we assume p and q are distinct primes.

We know prove L is in $\mathbf{NP}$.

We need to show, that we can verify a result in polynomial time. We construct a Turing Machine as the following:

$$M(n, \langle p, q \rangle) = \begin{cases} 1 & \text{if } n = p \cdot q \\ 0 & \text{otherwise} \end{cases}$$

where we assume p and q are NOT distinct primes, meaning they can be nonprimes, one of them can be prime and the other is not or they can be the same prime number.

Task 3

We prove that `OneNegSat` is in **P** with the following algorithm.

Algorithm 1: OneNegSat

    Data: CNF formula with at most one negated literal in each clause
    Result: Satisfying assignment if it exists or 0 if not

1 **while** *atleast one of $x_1, \ldots, x_i$ is not set and there exists a clause with exactly one literal* **do**

2 | find first clause with 1 literal c_i

3 | **if** *$x_j \in c_i$ is negated and x_j is not set* **then**

4 | | set x_j to 0 and for each clause with x_j remove it, if it is satisfied or remove x_j from it if not

5 | **else if** *x_j is not set* **then**

6 | | set x_j to 1 and for each clause with x_j remove it, if it is satisfied or remove x_j from it if not

7 | **else**

8 | | return 0

9 | **end if**

10 **end while**

11 set all unset x's to 1

This algorithm works since, a clause with more than one literal, will always be satisfiable, since we can just set every x to 1. The only possibility then is, that we have a clause with a single literal, forcing that literal to be set, and then have another clause, with this value negated, making it not satisfiable. The algorithm runs in $O(n^2)$ time, since we remove atleast one clause with each iteration of the while loop, and then in each iteration we check all clauses.

Task 4

a)

Looking at the first while loop, we can see that if currv = t the algorithm is done, and we skip down to the bottom. If currv $\neq$ t, but we are stuck, that means there are no more neighbours to check, and we have found no path. If it evaluates to true, it means we have are not done, and have not check any of the neighbours yet, so we add all the neighbours to a list.

Looking at the second while loop, we can see that, if nbqueue is empty, there are no more neigbours to check, which means there is no path. If we are not stuck we have found a vertex in the nbqueue, for which `DECIDE-PATH` have returned true, meaning there exists a path from the vertex to t, and we then set currv to that vertex, add the vertex to the path and repeat the process from the new vertex.

b)

We could make an algorithm like the following:

Algorithm 2: FACTORING

Data: k: possible factor (initial value is 2), N: number to factor, factors: list of current factors found

Result: List of prime factors

1 if $k = N$ **then**
2   return factors $++ [k]$
3 else if *Factor* $\langle N, k \rangle$ **then**
4   FACTORING $\langle k, \frac{N}{k}, \text{factors} ++ [k] \rangle$
5 else
6   FACTORING $\langle k + 1, N, \text{factors} \rangle$
7 end if

The worst case of this algorithm is if N is a prime number. Then we run the algorithm N times. Since FACTOR has a running time of

$$O\left(N^c \log(N)\right)$$

we have that FACTORING has a running time of

$$O\left(N^{c+1} \log(N)\right)$$

so the algorithm is polynomial in $\log N$.

Task 5

a)

We start by writing down the table.

x_1	x_2	x_3	NEQ	clause
0	0	0	0	$x_1 \vee x_2 \vee x_3$
0	0	1	1	
0	1	0	1	
0	1	1	1	
1	0	0	1	
1	0	1	1	
1	1	0	1	
1	1	1	0	$\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$

Figure 1: CNF representation of NEQ

This gives the CNF formula

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

b)

We start by writing down the table.

x_1	x_2	x_3	MAJ	clause
0	0	0	0	$x_1 \vee x_2 \vee x_3$
0	0	1	0	$x_1 \vee x_2 \vee \overline{x_3}$
0	1	0	0	$x_1 \vee \overline{x_2} \vee x_3$
0	1	1	1	
1	0	0	0	$\overline{x_1} \vee x_2 \vee x_3$
1	0	1	1	
1	1	0	1	
1	1	1	1	

Figure 2: CNF representation of MAJ

This gives the CNF formula

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

Task 6

a)

$\boxed{\tilde{F} \Rightarrow F}$

We have two cases. The first cases is the clauses with at most 3 literals. These are the same in both F and $\tilde{F}$ and are trivially true.

The second case is the clauses with more than 3 literals. We prove the contrapositive by contradiction. If we assume $\tilde{F}$ is true, but F is not true. For a given clause in $\tilde{F}$ to be true, it means that at least one a_i is true. If not then $y_0 = 1$ which means $y_1 = 1$ and so on until we reach $y_k = 1$, which then evaluates to false, so this is not possible. If one a_i is true then F must also be true, since one of the literals evaluates to true, and we have a contradiction. This means $\tilde{F} \Rightarrow F$.

$\boxed{F \Rightarrow \tilde{F}}$

We have two cases. The first cases is the clauses with at most 3 literals. These are the same in both F and $\tilde{F}$ and are trivially true.

The second case is the clauses with more than 3 literals. We prove the contrapositie by contradiction. We assume F is true but $\tilde{F}$ is not. Since F is true, we have that at least one a_i in each clause is true. For the corresponding clause in $\tilde{F}$, we have that the same a_i is true, and we can construct y such that $y_{i-1} = 1$ and $y_i = 0$. $y_{i-1} = 1$ means that $y_{i-2} = 1$ and so on until we reach $y_0 = 1$. $y_i = 0$ means that $y_{i+1} = 0$ and so on until we reach $y_k = 0$, which makes $\tilde{F}$ satisfiable, meaning we have a contradiction, so $F \Rightarrow \tilde{F}$.

b)

$\boxed{\tilde{F} \Rightarrow F}$ We have two cases. The first cases is the clauses with at most 3 literals. These are the same in both F and $\tilde{F}$ and are trivially true.

For any clause c_j in F, which we remove such that F is satisfiable, we can remove any clause from the corresponding set of clauses in $\tilde{F}$ such that $\tilde{F}$ is satisfiable. This follows since if we remove the clause with y_0, we can set all other $y_j = 0$, making the set satisfiable. If we remove the clause with y_k we can set all other clauses to $y_j = 1$, making the set satisfiable. Lastly, if we

remove any clause with a_i we can set $y_{i-1} = 1$ and $y_i = 0$, which makes the set satisfiable by the same logic as in a).

$\boxed{F \Rightarrow \tilde{F}}$ We have two cases. The first cases is the clauses with at most 3 literals. These are the same in both F and $\tilde{F}$ and are trivially true.

For any clause c_i in $\tilde{F}$, if we remove it, then $\tilde{F}$ is satisfiable. If we then remove the clause from F from which the clause c_i was created, then F is satisfiable, since all other clauses has a corresponding clause or set of clauses in $\tilde{F}$.

c)

Task 7

a)

(i) $\boxed{Reducibility}$
For all x_i we create A_i^T and A_i^F of size $(n + m)$. For all c_j we create B_j^1 and B_j^2 of size $(n + m)$. We create target T of size $(n + m)$. In total this gives us

$$O(2n(n+m) + 2m(n+m) + (n+m) = O(n^2 + m^2)$$

which shows we can reduce in polynomial time.

(ii) $\boxed{\text{yes} \mapsto \text{yes}}$
For all A_i we must choose either A_i^T or A_i^F for a legal assignment. By construction of the target and the x_i indeces of A_i we reach the target T for the given index, if we choose one or the other, but not both.
To satisfy a clause c_j the sum of the index c_j must be greater than 0, else we have choosen falsefying assignments for all literals of the clause c_j. Per construction of B we can reach T.

(iii) $\boxed{\text{no} \mapsto \text{no}}$
If we have to choose both A_i^T and A_i^F or neither of them we have a falsefying assignment. By construction of the target and the x_i indeces of A_i we cannot reach the target T for the given index, if we choose both or neither of them.
For c_j not to be satisfied, the sum of the indeces j must be 0, else we have chosen assignments for the literals of c_j such that it evaluates to true. Per construction of B we cannot reach T in this case.

b)

This from of satisfying assignment is that only one literal must evaluate to true. The clauses c_j is going to look like the following:

$$(x_1 \text{ xor } x_2 \text{ xor}, \dots, \text{xor } x_i)$$

c)

First we prove that it is in **NP**.
To verify an assignment of literals $x_1, \dots, x_i$, we just insert the witness in the formula and evaluate. This can be done in linear time.

We know want to show `HACKED-SAT` $\leq_P$ `SUBSET-SUM` to show that `HACKED-SAT` is **NP-complete**.

(i) $\boxed{Reducibility}$

For all x_i we create A_i^T and A_i^F of size $(n + m)$. We create target T of size $(n + m)$. In total this gives us

$$O(2n(n + m) + (n + m) = O(n^2 + nm)$$

which shows we can reduce in polynomial time.

(ii) $\boxed{\text{yes} \mapsto \text{yes}}$
For all A_i we must choose either A_i^T or A_i^F for a legal assignment. By construction of the target and the x_i indeces of A_i we reach the target T for the given index, if we choose one or the other, but not both.
To satisfy a clause c_j the sum of the indeces c_j must be equal to 1, else we have choosen either more than 1 literal to be true or no literals to be true for the clause c_j. By construction of the target T index j is 1.

(iii) $\boxed{\text{no} \mapsto \text{no}}$
If we have to choose both A_i^T and A_i^F or neither of them we have a falsefying assignment. By construction of the target and the x_i indeces of A_i we cannot reach the target T for the given index, if we choose both or neither of them.
For c_j not to be satisfied, the sum of the indeces j must be 0 or greater than 1, else we have chosen assignments for the literals of c_j such that 0 or more than 1 literal evaluates to true. Then per construction of B we cannot reach T in this case.

Task 8

We construct a Turing machine M simulating another Turing machine, that performs the function $g(x)$. If we assume M runs in polynomial time, we have a contradiction, since we know running $g(x)$ is the halting problem.

Task 9

The main idea for the proof is to pad the problem with additional data.
We let $P : \mathbb{N} \to \mathbb{N}$ be a function that is computable in time polynomial in n. We then define $\texttt{SAT}_P$ to be $\texttt{CNF-SAT}$ with all size-n formulas φ padded with $n^{P(n)}$ 1's as the following:

$$\texttt{SAT}_P = \left\{ \varphi 01^{n^{P(n)}} \mid \varphi \in \texttt{CNF-SAT} \text{ and } n = |\varphi| \right\}$$

This means that, given some string x, scan from the back until we reach the first 0. We then let φ be everything before that 0, and we let n be the length of this string. $n = |\varphi|$. The string after 0 is 1^k. We then have that $x \in \texttt{SAT}_P$ if $\varphi \in \texttt{CNF-SAT}$ and $k = n^{P(n)}$.

We now want to choose some padding in some way such that $\texttt{SAT}_P$ is too hard to be in **P** but too easy to be **NP-complete** (assuming **P** $\neq$ **NP**.

We observe two things

- If $P(n) \in O(1)$, then $\texttt{SAT}_P$ is **NP-complete**

- If $P(N) = \Omega\left(\dfrac{n}{\log n}\right)$, then $\texttt{SAT}_P \in$ **P**

We know want to prove these to points.

If $P(n) \in O(1)$, then $\texttt{SAT}_P$ is **NP-complete**, since the amount of added data then is 01^{n^c}, which is a polynomial amount. These means, that verifying a witness y take polynomial time to remove the padding and then polynomial time to verify, since we know just plug in the variables and

evaluate. This gives us polynomial time in total to verify.

We can also reduce $\texttt{SAT}_P \leq_P \texttt{CNF-SAT}$ in polynomial time, since all we have to do is remove the padding, and this can be done in polynomial time.

If $P(N) = \Omega\left(\dfrac{n}{\log n}\right)$, then $\texttt{SAT}_P \in \mathbf{P}$. We have that the input of $\texttt{SAT}_P$ is

$$n_P = n + n^{\dfrac{n}{\log n}}$$

We can rewrite the last part of this with log rules.

$$n^{\dfrac{n}{\log n}} = n^{\dfrac{1}{\log n} n} = n^{\log_n(2)n} = n^{\log_n(2^n)} = 2^n$$

This means the input of $\texttt{SAT}_P \geq 2^n$ and since the $\texttt{CNF-SAT}$ problem can be solved in 2^n time (by all permutations), we have that φ can be solved in polynomial time of the input $\varphi 01^{n^{\frac{n}{\log n}}}$.