Handout 11B: Complexity review - Solutions

William Pires

Exercise 1 (True False)**.**

1. All of $n * \log(n)$, $n^{100} + 3n^2$, $n^{0.001n}$, $2^{\sqrt{n}}$ are polynomial.

   Answer : $n * \log(n)$, $n^{100} + 3n^2$ are polynomial, $n^{0.001n}$, $2^{\sqrt{n}}$ aren't.

2. If you show a verifier $V(x, c)$ for a language L, where V runs in time polynomial in $|x| + |c|$, then L must be in NP.
   Answer : False, for L to be in NP you need a verifier with runtime polynomial in $|x|$.[1] If you allow c to be of length exponential in $|x|$, and you allow V to run in time polynomial in $|x| + |c|$, then you have effectively allowed V to run in time exponential in $|x|$.

3. All NP Complete problems are polytime reducible to each other.

   Answer : True. Here's a proof.

   Let A, B be NP-Complete, we want to show $A \leq_{\mathrm{P}} B$.

   We know B is NP complete, and thus it's NP hard. So for any $L \in \mathrm{NP}$, we have $L \leq_{\mathrm{P}} B$.

   Since A is NP complete, it means $A \in \mathrm{NP}$.

   Combining the two (taking the language $L = A$), we have $A \leq_{\mathrm{P}} B$.

4. Say the input of a TM is an integer N given in binary, and M runs in time $O(N^2)$, then M runs in polynomial time.

   Answer : False. N is given as input in binary, so the input size is $O(\log_2(N))$. But $N^2 = 2^{2\log_2(N)}$, using $N = 2^{\log_2(N)}$. So the runtime is exponential in the input length.

Exercise 2. Show that NP is closed under union. That is if $L_1, L_2 \in \mathrm{NP}$ then $L_1 \cup L_2 \in \mathrm{NP}$.

Let $L_1, L_2 \in \mathrm{NP}$. Then by definition, there exists a polynomial time verifier V_1, such that :

$$x \in L_1 \Leftrightarrow \exists c_1 \text{ such that } V_1(x, c_1) \text{ accepts.}$$

Similarly, we have a polynomial time verifier V_2 for L_2 such that :

$$x \in L_2 \Leftrightarrow \exists c_2 \text{ such that } V_2(x, c_2) \text{ accepts.}$$

We now want to build a polynomial time verifier V for $L_1 \cup L_2$. The idea is that given c, we first run V_1 to check if it accepts, and then we run V_2. We accept if either accepts, and reject if they both reject.

[1] But this means that if $x \in L$ there is a c of length polynomial in $|x|$ such that $V(x, c)$ accepts. Otherwise, V wouldn't have the time to read c in polynomial time.

Algorithm 1 A verifier for $L_1 \cup L_2$

Input: x, c

Run V_1 on x, c ▷ V_1 runs in polynomial time in $|x|$
if V_1 accepts **then**
  Accept x
end if

Run V_2 on x, c ▷ V_2 runs in polynomial time in $|x|$
if V_2 accepts **then**
  Accept x
end if
Reject.

First, it's clear that V runs in polynomial time in $|x|$, as all it does is run V_1, V_2 on x, c. [2]

So it remains to show

$$x \in L_1 \cup L_2 \iff \exists\, c \text{ such that } V(x,c) \text{ accepts.}$$

Assume $x \in L_1 \cup L_2$, we want to show there exists a c such that $V(x,c)$ accepts. If $x \in L_1$, we know there exists a c_1 such that $V_1(x, c_1)$ accepts. So taking $c = c_1$ leads to V accepting.
Else if $x \in L_2$, we know there exists a c_2 such that $V_2(x, c_2)$ accepts. So taking $c = c_2$ leads to V accepting.

Thus

$$x \in L_1 \cup L_2 \Rightarrow \exists\, c \text{ such that } V(x,c) \text{ accepts}$$

We now show the other direction. Assume there exists a c such that $V(x,c)$ accepts. Then in the pseudocode either $V_1(x,c)$ or $V_2(x,c)$ must have accepted. If V_1 accepted, then by definition

$$V_1(x,c) \text{ accepts } \Rightarrow x \in L_1$$

Else if V_2 accepted then by definition

$$V_2(x,c) \text{ accepts } \Rightarrow x \in L_2$$

Anyway, it must be that $x \in L_1 \cup L_2$. So $V(x,c)$ accepts $\Rightarrow x \in L_1 \cup L_2$.

[2] Here's a technical point you can ignore. We should first check that c isn't too large. By that, I mean $|c| = O(n^k)$ where we have V_1, V_2 both run in time $O(n^k)$. I.e, if c is bigger than the runtime of V_1 and V_2, we should reject. This to avoid the case where c is so large that giving it as input to V_1 would take too long. And if c is bigger than the runtime of V_1, V_2 we know they would never accept (x, c).

Exercise 3.

- Fix some constant k. The problem k-Clique is defined as follow : You are given as input a graph G, G is a graph on n nodes. You can think of G as being described by a $n \times n$ binary matrix A where $A_{i,j} = 1$ iff there's an edge (i, j) in G (so the input size is n^2). You must accept G if and only there is a subset of k vertices V^* of G, such that these vertices form a complete graph (any two nodes have an edge between them). Why is this problem in P ?

  We can give a polynomial time algorithm for the problem as follow :

Algorithm 2 An algorithm for k-Clique

Input: $\langle G \rangle$ where G is a graph

for all subset S of k vertices from G **do**
  Check that for each pairs (u, v), with $u, v \in S$ we have that (u, v) is an edge in G.
  ▷ This means the vertices in S form a complete graph.
  If for all pairs, the edge (u, v) is in G, accept.
end for
Reject.

  This algorithm basically looks at all the subsets of k vertices in G and checks they form a complete graph. Clearly if $\langle G \rangle \in$ k-Clique, then we will accept, and otherwise we will rejects.

  In the algorithm, we look at all the $\binom{n}{k}$ subsets of k vertices of G. For each subset, we need to check $O(k^2)$ edges are present, one for all pair. Since k is a constant, we can assume this takes constant time. So the runtime of the algorithm is $O(n^k)$. Since k is a constant, this is polynomial time.

- The NP complete problem Clique is defined as follow : You are given as input a graph G, G is a graph on n nodes. **But now k is given to you as input in decimal**. (So the input size is roughly $(n^2 + \log_{10}(k))$). Again, You must accept G if and only there is a subset of k vertices V^* of G, such that these vertices form a complete graph. Why doesn't the previous proof work to show this problem is in P ?

  If we had to adapt the above algorithm for this proof it would look like that :

Algorithm 3 An algorithm for Clique

Input: $\langle G, k \rangle$ ▷ k is now part of the input

for all subset S of k vertices from G **do**
  Check that for each pairs $(u, v) \in S$ (u, v) is an edge in G.
  if for all pairs, the edge (u, v) is in G **then**
    Accept.
  end if
end for
Reject.

Imagine the input is $\left(G, k = \frac{n}{2}\right)$. Then, the algorithm will have to look at $\binom{n}{\frac{n}{2}}$ subsets, but that is roughly 2^n many subsets. However, the input size is only $n^2 + \log_{10}(n)$, so the runtime of the algorithm is exponential. [3]

Exercise 4. Problem 7.18 in the book. Show that if P = NP,then every language $A \in \text{P}$, except $A = \emptyset$ and $A = \Sigma^*$, is NP-Complete. (Hint : think about the definition of $L \leq_{\text{P}} A$, knowing that $L \in \text{NP}$ implies $L \in \text{P}$.)

This exercise is great to test how comfortable you feel with the definition of NP completeness.

Assume P = NP and let $A \in \text{P}$, be any language except $\emptyset$ or Σ^*. That means there exists two strings y, z such that $y \in A$ and $z \notin A$.

Now consider the NP complete problem 3-SAT, since P = NP by assumption, we have 3-SAT $\in P$. So there exists a polynomial time decider M for 3-SAT.

We want to show A is NP-hard. We will show 3-SAT $\leq_{\text{P}} A$. So, we claim the following is a mapping reduction from 3-SAT to A.

Algorithm 4 A mapping reduction f from 3-SAT to A

```
Input: ⟨φ⟩                                                        ▷ φ is a CNF
  Run M on ⟨φ⟩      ▷ This takes polynomial time      ▷ This tells us if ⟨φ⟩ ∈ 3-SAT or not
  if M accepts then
      Return y                                        ▷ ⟨φ⟩ ∈ 3-SAT, so we return y ∈ A
  else
      Return z                                        ▷ ⟨φ⟩ ∉ 3-SAT, so we return z ∉ A
end if
```

It's clear the above mapping f is computed in polynomial time. Besides it's also clear that if $\langle\phi\rangle \in$ 3-SAT, we have $f(\langle\phi\rangle) = y \in A$. And if $\langle\phi\rangle \notin$ 3-SAT, then $f(\langle\phi\rangle) = z \notin A$. So $f(\langle\phi\rangle) \in A \iff \langle\phi\rangle \in$ 3-SAT.

So this is a valid polynomial time mapping reduction from 3-SAT to A.

So A is NP-hard, and since $A \in \text{P}$, $A \in \text{NP}$ and thus by definition A is NP complete.

You might wonder why we need that A isn't $\emptyset$ or Σ^*. In a mapping reduction from a language L to A , we need $x \in L \Leftrightarrow f(x) \in A$.

But if we let $A = \emptyset$, then there's no strings in A ! So if $x \in L$, we can never have $f(x) \in A$. So there can't be a mapping reduction from L to A if $L \neq \emptyset$.

[3].It's important to realize we look at the running time in the WORST CASE input. And here, the algorithm is very slow whenever you give $k = n/2$ (or any large enough function of n).

Similarly, if $A = \Sigma^*$, then all strings are in A. So if $x \notin L$, we can never have $f(x) \notin A$. So there can't be a mapping reduction from L to A if $L \neq \Sigma^*$.

Exercise 5. Problem 7.21 in the book. Let G represent an undirected graph. Also let

- SPATH = $\{\langle G, a, b, k \rangle$— G contains a simple path of and length **at most** k from a to $b\}$
- LPATH = $\{\langle G, a, b, k \rangle$— G contains a simple path of and length **at least** k from a to $b\}$

Show that SPATH $\in$ P and LPATH is NP-Complete. (A simple path is a path that doesn't visit the same node twice.)

Recall that a simple path is a path that never visits a node twice (i.e. the path has no cycles).

We first show SPATH $\in P$. To check there is a simple path of length $\leq k$ from a to b it suffices to do the following : Perform a Breath-First Search starting at a in G.

When we perform a Breath first search (BFS), we first see all the nodes at distance 1 from a, then we see the ones at distance 2, etc... So if we reach b before we're at distance $k+1$, we accept, else we reject. Obviously doing a BFS takes times polynomial in the size of the input.

We now show LPATH is NP-Complete.

First, we show LPATH is in NP. Here's a verifier for LPATH :

Algorithm 5 A verifier for LPATH

Input: $\langle G, a, b, k \rangle, p$ $\triangleright$ p is the extra string the verifier takes as input

Check p is a simple path from a to b in G.
Check that p has length at least k.

if Both of the above are true **then**
  Accept.
else
  Reject.
end if

Clearly, the above runs in polynomial time. Also it's clear that if $\langle G, a, b, k \rangle \in$ LPATH, then there must be a simple path of length $\geq k$ between a and b. So just set p to be this path, this will lead the verifier to accept. Obviously, if no such path exists, there's no p that would lead the verifier to accept $\langle G, a, b, k \rangle, p$.

So it remains to show the problem is NP-hard. To do this we will reduce the NP-Complete problem HamPath to LPATH. We need to give a polynomial time mapping reduction from Hampath to LPATH.

Recall that HamPath is the following problem : Given a graph G and two nodes s, t is there an Hamiltonian path from s to t in G.

Given G, s, t as input for Hampath, our function outputs $\langle G, a = s, b = t, k = n\rangle$. I.e. $f(\langle G, s, t\rangle) = \langle G, s, t, n\rangle$.

Clearly this is computable in polynomial time. So it remains to show that $\langle G, s, t\rangle \in$ Hampath if and only if $\langle G, s, t, n\rangle \in$ LPATH.

This isn't hard, once we unpack the definition of Hamiltonian path. By definition there is an Hamiltonian path from s to t, if and only if, there is a simple path between s and t that visits every vertex once.

Since the path starts at s, ends at t and goes through every of the n vertices once, it means it must be of length n [4].

So by definition G has an Hamiltonian path from s to t if and only if there is a simple path between s and t of length n.

Also note that simple paths can't have length more than n, else a vertex would be repeated so it wouldn't be simple.

Thus, we clearly have that there's an Hamiltonian path path from s to t if and only if there is a simple path between s and t of length at least n.

So $\langle G, s, t\rangle \in$ HamPath iff $\langle G, s, t, n\rangle \in$ LPATH. This proves Hampath $\leq_{\mathrm{P}}$ LPATH. So LPATH is NP-hard.

Since LPATH is NP-hard and in NP, it's NP complete.

[4] Here I count the length of a path as number of vertices