TP3 - NP Properties and Polynomial Reductions

Here is Lucas Sauvage's ACT TP3 assignment.

Answers

PART 1

Q1

Definition of a NP algorithm L is said to be NP if there exists a polynomial Q and a polynomial-time algorithm A with two inputs and Boolean output such that :

$ L = {u | ∃c, A(c, u) = True, |c| ≤ Q(|u|)} $

Certification A certificate is a series of choices that serves as a proof (in polynomial time) that an instance `u` is true, i.e. belongs to the language `L`.

In this TP, the certification is an assignement function which specifies for each object i,the bag j it goes into.

How to implement certification ? Given an array `arr` and a set of bags `j`. `arr[i]` indicates which bag item at index `i` go in.

To check if the array's a validation of the algorithm, we can go through the array and sum all the items weight.

If the sum exceed the total capacity, the verification's false, otherwise it's true.

Certificate size As the certificate corresponds to the affectation of all items into bags, the array size is `n` : $ aff : [1..n] → [1..k] $

Each entry in the array indicates the bag number for the corresponding item. To encode a bag number, we need approximately $log_2(k)$ bits.

Therefore, the total size of the certificate is $ |c| = n * log_2(k) $

Is the size of certificates polynomially bounded ? First, we need to approximately calculate |u| size (input size). As x_i is the weight of the item `i`, `c` the capacity of the bag and `k` the number of bags, the size of the inputs is (to encode n , we need $log_2(n)$ size):

$ |u| = \sum_{i=1}^n log_2(x_i)+log_2(c) +log_2(k) $

As $|c| = n * log_2(k)$, $|u|$ contains *at least* n (because there is n inputs). Moreover, $|u|$ contains also $log_2(k)$ (for the size of the bags), so there's always a polynom Q such that : $ |c| ≥Q(|u|) $

Verification algorithm Determine wether this algorithm runs in $O(n^k)$ for $k \in \mathbb{N}^*$.

We iterate j from `1` to `k+1` which runs in $O(k)$. The sum runs in $O(len(x)) \Leftrightarrow O(n)$.

The total complexity of this algorithm is $O(k) * O(n) ⇔O(kn) $ which is polynomial.

Q2

Q2.1 – Randomly generating a certificate Since each object is assigned independently to a random bag between 1 and k, the probability of generating any particular certificate is $(1/k)^n$.

Therefore, the algorithm generates certificates uniformly.

Q2.2 – Outline of a nondeterministic polynomial algorithm for the problem To create the outline of the problem, we need to decompose it into steps : - With an input of an instance u, we generate a certificate - We check if the certificate is correct - We returns True or False regarding the result of the previous step

With A as the polynomial-time algorithm with two inputs and Boolean output, an outline would be :

[Instance *u*] => [Certificate *c* generated randomly (for now I guess)] => [Verify $A(u, c)$] => [Returns True or False]

Q3 – $NP \subset EXPTIME$

EXPTIME Definition In computational complexity theory, the complexity class EXPTIME (sometimes called EXP or DEXPTIME) is the set of all decision problems that are solvable by a deterministic Turing machine in exponential time, i.e., in O(2p(n)) time, where p(n) is a polynomial function of n.

See Wikipedia

Q3.1 – Cerificate values for n and k fixed For k and n fixed, the certificate can take k^n values. Indeed, for each value of `aff[i]`, there's k possibilites and the length of aff is n. That's $k * k * ... * k$ n times so k^n.

Q3.2 – Enumeration of all certificates To define an order, I suggest to start from the lowest one, i.e. $[1, 1, ..., 1], n$ times.

Then we add 1 to the first digit, until reaching k. Then we reset the digit and move to the second and so on and so forth.

The maximum certificate will be reached at the end of the algorithm, and it's $[k, k, ..., k], n$ times.

THis allows a enumeration of all possible certificates without forgetting one.

Q3.3 – The British Museum algorithm The British Museum algorithm is a general approach to find a solution by checking all possibilities one by one, beginning with the smallest.

An algorithm would be :

- Generate minimal certificate
- While not $[k, k, ..., k], n$ times :
  - Verify is the certificate is a solution
  - If not : Next certificate
- If none certificate is a solution, returns False

Q4

For this question, I didn't understand what the teacher meant by ''. I assume it's a **txt** file formatted as follows :

```
10 # number of items
4 5 8 9 2 1 2 4 3 10 # list of item's weight
7 # the capacity
```

PART 2

Q1

Partition Input :

n : items number

$x_1, ..., x_n$: the items

Output :

Yes if

$$\exists J \subset [1..n] \mid \sum_{i \in J} x_i = (\sum_{i=1}^{n} x_i)/2$$

No otherwise

To reduce *Partition* into *BinPack*, we need to first compute the sum of all items in the list. Let's call this sum S.

Then we need to set the number of bags (k) to 2, and the capacity for each bag $S/2$.

Correctness of reduction Yes for BinPack => Yes for Partition :

Let's consider a set of items whose sum is equal to S (i.e. $S = \sum_{i=1}^{n} x_i$). "Yes" means we can put items in two bags of capacity c then we found a subset whose sum is equal to $\frac{\sum}{2} <=> \frac{S}{2}$. (Each bag represent a subset) Let $k = 2$. An instance of BinPack is : $(x_{i1}, ..., x_{ik})$ goes in bag 1 and $(x_{ik+1}, ..., x_{in})$ goes in bag 2 Let $J = (x_1, ..., x_k)$, the list of items placed in bag 1. Since every item is placed in one of the two bags : $\sum_{i \in J} x_i + \sum_{i \notin J} = \sum_{i=1}^{n} x_i = S$ Because both bags respect the same capacity we have $\sum_{i \in J} x_i \leq S/2$ and $\sum_{i \notin J} x_i \leq S/2$ so necesseraly $\sum_{i \in J} = c/2$ Therefore, J is a valid subset for Partition, so

Yes for Partition => Yes for BinPack :

Let c the capacity for each bag on the BinPack instance. If there exists a subset $J = (x_1, ..., x_k)$ such that $\sum_{i=1}^{k} x_i = S/2$, then we have $\sum_{i \in J} x_i$ in bag 1 and $\sum_{i \notin J} x_i$ in bag 2. Therefore, $\sum_{i \in J} x_i = \sum_{i \notin J} x_i = \frac{S}{2} = c$ which implies that Yes for Partition equals Yes for BinPack. This transformation only requires to compute $S = \sum_{i=1}^{n} x_i$ which is $O(n)$ which is polynomial.

Q1.2

If Partition can be reduced to BinPack, it proves that $Partition \leq_P BinPack$ and it means that each instance of Partition can be transformed into a BinPack instance in polynomial time.

As Partition is NP-Complete, we can assert that BinPack is, at least, as much as harder, which makes it NP-Hard.

Moreover, check a certificate takes polynomial time, so $BinPack \in NP$.

As $BinPack \in NP$ and $BinPack \in NP - Hard$, **BinPack is NP-complete** !

Q1.3

There is a unique case where we can reduce *BinPack* into *Partition*, and it's when $k = 2$. So, theorically there's a reduction, but for only one case so it's not relevant.

Q2

Partition is a special case of Sum, with $c = \frac{\sum x_i}{2}$, i.e. exactly half of the total sum. Therefore, any instance of Partition can be transformed into an instance of Sum, which gives : $Sum \leq_P Partition$

To reduce Sum into Partition (i.e. $([x_1, ..., x_n], c)$) we need to first compute the sum of the items $S = \sum_{i=1}^{n} x_i$. If $S < 2c$, it means we need to "complete" the list to reach 2c. To reach 2c we need to add an item of weight equals to $2c - S$.

The instance $([x_1, .., x_n, x_{n+1}], c)$ is then an instance to Partition.

Sum instance : $([x_1, ..., x_n], c)$ || Partition instance : $([x_1, ..., x_n])$

Yes for Sum => Yes for Partition

Let's assume that $J \subseteq [1..n]$ with $\sum_{i \in J} x_i = c$

In Partition, the total sum is equals to 2c, so half of it is c.

Take subset J as a instance to Partition computes a sum equals to half the total sum.

Yes for Partition => Yes for Sum

Let's assume that $J' \subseteq [1..n]$ with $\sum_{i \in J'} x_i = c$

If $x_{n+1} \in J'$ then the sum of all the others items equals $c - x_{n+1} = c - (2c - S) = S - c$. The other half is then equals to c, and the sum of a subset is equals to c

Q4

As we have $Sum \leq_P Partition$ and $Partition \leq_P BinPack$, we obviously have $Sum \leq_P BinPack$

(reduction in reduction.py)

Q5

To reduce *BinPackDiff* into *BinPack* we need to get the maximum capacity (c_{max}) of the bags for *BinPackDiff*.

Then, we go through all bags (of capacity c_i) and add an artificial item $c_{max} - c_i$.

Therefore, we can assume each bag has the same capacity and then reduce it to *BinPack*.

PART 3

Q1

Let assume that *BinPackOpt1* is P and returns the minimal k_{min} needed.

To compute *BinPack* instance $[(x_1, .., x_n), c, k]$ we compute *BinPackOpt1*$[(x_1, .., x_n), c]$. *BinPack* then computes "yes" or "no", which is polynomial, so it implies that *BinPack* $\in P$. It' the same for *BinPackOpt2*.

But we already know that *BinPack* is $\in NP$ (even in $NP-complete$) so this would implies that $P = NP$ which can't be true. It means that *BinPackOpt* (1 or 2) is at least $NP - Hard$.

Q2

Let's assume that *BinPack* is in P. To show that *BinPackOpt1* is also in P we need to proves there exists a reduction from *BinPackOpt1* to *BinPack*, i.e. $BinPackOpt1 \leq_P BinPack$.

We need to transform an instance of *BinPackOpt1* (i.e. $[(x_1, ..., x_n), c]$) into *BinPack*. That mean we have to found the k_{min} such that $BinPack([x_1, ..., x_n], c, k_{min})$ returns True.

Now that we've found a reduction, if *BinPack* is P, then *BinPackOpt1* is P and $BinPackOpt1 \leq_P BinPack$

Q3

It's the same as above. If we find a reduction from *BinPackOpt2* into *BinPack*, then *BinPackOpt2* is P.

To do so, we need to first find the k_{min} (same as above). Then we need to compute *BinPack* with the k_{min} and find the first instance where *BinPack* returns "Yes".