COMP3610 Problem Set 2

Enoch Lau

Question A1

Assumption

The decision problem associated with factorisation is:

$$\textit{HAS-FACTOR} = \{\langle n, k\rangle \mid n \text{ has prime factors less than } k\ \}$$

Under the *P = NP* assumption, *HAS-FACTOR* can be decided in polynomial time.

Algorithm

We present an algorithm *FACTORIZE*(n), which, given an integer n, will return a set of integers representing the prime decomposition of n. It utilises *HAS-FACTOR* to perform a binary search on k to find the smallest prime factor of n. Once this is found, the algorithm divides n by this factor, and repeats the process on the quotient, until there are no more factors to find. *FACTORIZE* uses a subroutine, *BINARY-SEARCH-PRIME*$(n, \textit{left}, \textit{right})$, which will examine the range $\textit{left} \leq n < \textit{right}$; note that this is a half-open interval, and that the subroutine assumes that a prime exists in that range and that there are no primes smaller than *left*.

FACTORIZE(n):

1. *factors* $\leftarrow \{\}$
2. while $n \neq 1$:
   (a) $k \leftarrow$ *BINARY-SEARCH-PRIME*$(n, 1, n+1)$
   (b) *factors* $\leftarrow$ *factors* $\cup \{k\}$
   (c) $n \leftarrow n/k$
3. return *factors*

BINARY-SEARCH-PRIME$(n, \textit{left}, \textit{right})$:

1. if $\textit{right} - \textit{left} \leq 1$:
   (a) return *left*

2. $mid \leftarrow \left\lfloor \frac{left + right}{2} \right\rfloor$
3. if *HAS-PRIME*$(\langle n, mid \rangle)$, return *BINARY-SEARCH-PRIME*$(n, left, mid)$
4. Otherwise, return *BINARY-SEARCH-PRIME*$(n, mid, right)$

Analysis of algorithm

Firstly, note that $d = \Omega(\log n)$ where d is the number of digits in the binary representation of n. From the assumption, suppose that *HAS-PRIME* runs in $O(P(d))$ time for some polynomial $P(d)$. Then the call to *BINARY-SEARCH-PRIME* from *FACTORIZE* runs in $O(\log n\, P(d)) = O(d\, P(d))$ time, because this is a binary search on k.

Let us now consider the number of times that the loop in step 2 of the algorithm executes. In the worst case, a large number of prime factors of n (counting multiplicities) appears when the prime factors are small, say 2; if there are factors larger than 2, this will only reduce the number of factors. Hence, the number of prime factors of n, and the number of times the loop executes, is $O(\log n) = O(d)$.

The other operations are insignificant in terms of time complexity, and thus we conclude that *FACTORIZE* runs in $O(d^2\, P(d))$ time, which is polynomial.

Question A2

PUZZLE is in *NP*

The certificate to be verified is the orientation of the n cards in the collection. We can verify this certificate in polynomial time:

1. For each card, orient the card as in the certificate. Cross off the squares that are covered by the card.
2. Check to see if all squares have been crossed off.

This is clearly polynomial in the number of cards in the collection.

SAT $\leq_p$ *PUZZLE*

We can use a solution of *SAT* to find a solution to *PUZZLE*

For each card c_i, we identify one side with x_i being true and the other side with x_i being false.

1. Each row of squares corresponds to two OR expressions, which are constructed as follows:

(a) Check to see if any of the cards have both squares in the row filled in. If so, skip this row as this row will always be covered regardless of the configuration of the cards.

(b) Check to see if all of the cards have both squares in the row punched out. If so, reject this collection of cards now, as this row will never be covered regardless of the configuration of the cards.

(c) For each card c_i with just one square in the row (out of the two) punched out, contribute:

- (if the left square is unpunctured) x_i to the first of the two OR expressions, and $\overline{x_i}$ to the second of the two OR expressions; and
- (if the right square is unpunctured) $\overline{x_i}$ to the first of the two OR expressions, and x_i to the second of the two OR expressions.

2. Join all of these OR expressions together with AND operators to make a Boolean formula in conjunctive normal form, which we then solve in *SAT*. We interpret the solution from *SAT* as follows:

   (a) If *SAT* finds a solution, then there is a solution to *PUZZLE*:

   - If a card has a corresponding variable in the solution, orient the card according to the assignment of true or false.
   - Otherwise, orient the card with an arbitrary side facing up.

   (b) If *SAT* does not find a solution, there is no solution for the *PUZZLE* problem either.

   This works because if *all* the OR expressions are true in the cnf formula, then that implies that *all* the squares have been covered by at least one card.

The running time of the mapping is clearly polynomial in the size of the input, namely the number of cards, as before calling *SAT*, it simply scans through the list of cards (albeit multiple times); after *SAT*, it is a linear operation to map the state of the variables back to the orientation of the cards.

We can use a solution of *PUZZLE* to find a solution to *SAT*

Suppose now that we have an arbitrary Boolean formula in conjunctive normal form.

1. We create a card c_i for each variable x_i, and as above, we associate one face with true and the other with false. For each OR expression in the cnf formula:

   (a) Create a new row of squares for this expression, and by default, all the squares in the row have been punched out.

   (b) If a variable x_i is included in the row, fill in the left hand square on card c_i.

   (c) If a variable $\overline{x_i}$ is included in the row, fill in the right hand square on card c_i.

This is designed such that the value of x_i induces c_i to cover up the left hand square when it is oriented according to the value of x_i. Note that the cases where, for some j, neither x_j or $\overline{x_j}$, or both x_j and $\overline{x_j}$ are included are handled implicitly.

2. Add a card that has all the left hand squares punched out, and the right hand squares filled in. This is because we lack the symmetry in the pairs of OR expressions that we had when we constructed the Boolean formula from the cards in the puzzle previously. By creating this card, we ensure that either the left hand or right hand column of squares is covered regardless of the orientation of the cards, because we do not have a corresponding OR expression for that column.

3. Solve using *PUZZLE*.

4. If *PUZZLE* finds a solution, the orientations of the card c_i dictates whether x_i is true or false depending on which face is up. If the additional card has been flipped around so the filled squares are on the left hand side instead of the right, flip all of the values around, because we can find an equivalent solution by substituting all x_i for $\overline{x_i}$.

5. If there is no solution to *PUZZLE*, there is no solution to *SAT* either.

This mapping is clearly polynomial in the size of the input, namely the length of the formula, as we consider each variable as written in the formula once only before calling *PUZZLE*; after calling *PUZZLE*, it is a linear operation to map the card orientations to the values of variables.

Together, these two parts prove that there is a polynomial time mapping from *SAT* to *PUZZLE*. Since *PUZZLE* is in *NP*, we conclude that *PUZZLE* is *NP*-complete.

Question A3

We will not change the greedy 2-approximation algorithm as given in the lectures, but simply provide a tighter bound on the approximation.

Firstly, we note that the bound in the proof $T_i - t_j \leq \frac{1}{m}\sum t_i$ is not tight because machine i, before it is assigned job j, is the machine with the least load, and if the least load is $\frac{1}{m}\sum t_i$, then the m machines together have completed all of the jobs. Instead, we note that the length of all the jobs prior to job j is $\sum t_i - t_j$, and this implies that $T_i - t_j \leq \frac{1}{m}\left(\sum t_i - t_j\right)$, otherwise the m machines will all have finished processing all the jobs (excluding j) already before T_i starts on job j.

Hence:

$$\begin{aligned} T_i - t_j &\leq \frac{1}{m}\left(\sum t_i - t_j\right) \\ &\leq OPT - \frac{t_j}{m} \\ T_i &\leq OPT - \frac{t_j}{m} + t_j \\ &\leq OPT - \frac{OPT}{m} + OPT \qquad \text{since } t_j \leq OPT \\ &= OPT\left(2 - \frac{1}{m}\right) \end{aligned}$$

Question A4

Example where suggested greedy algorithm fails

Suppose we have the following items:

Weight	Value
10	2
2	1
2	1
2	1
2	1
2	1

with a knapsack of size 10. The greedy algorithm returns only the item of weight 10 and value 2, whereas the optimal solution consists of the five items of weight 2 and value 1, giving a total value of 5. It differs by a factor greater than 2, and thus this greedy algorithm is not a 2-approximation algorithm.

A 2-approximation algorithm

We describe a greedy algorithm that sorts the items based on $\frac{\text{value}}{\text{weight}}$. Given prices P, weights A and the size of the knapsack b:

KNAPSACK-APPROX(P, A, b):

1. Discard any items with a weight greater than b, because they will never fit into the knapsack anyway.
2. If the sum of the weights of all the items available is less than or equal to b, just return the set of all the items, because that is the optimal solution.
3. Arrange the items such that $\frac{p_1}{a_1} \geq \frac{p_2}{a_2} \geq \cdots \geq \frac{p_n}{a_n}$.
4. Find the first k such that $\sum_{i=1}^{k} a_i > b$:

(a) *sum_size* $\leftarrow 0$, *sum_value* $\leftarrow 0$

(b) for $k = 1$ to n:

i. *sum_size* $\leftarrow$ *sum_size* $+ a_k$

ii. if *sum_size* $> b$, then break out of the loop.

iii. otherwise, *sum_value* $\leftarrow$ *sum_value* $+ p_k$

5. Compare the values of the sets $\{1, \ldots, k-1\}$ and $\{k\}$:

(a) if $p_k >$ *sum_value*, return $\{k\}$.

(b) otherwise return $\{1, \ldots, k-1\}$.

Analysis of algorithm

First, we prove that a greedy algorithm where items are ordered by $\frac{\text{value}}{\text{weight}}$ gives the optimal solution for the fractional knapsack problem (as we will refer to this in the analysis below):

- We have optimal substructure: if we remove the last item placed into the knapsack, say j, the value of the items chosen for the weight $b - a_j$ must be optimal otherwise we can substitute in a more optimal solution.
- We have the greedy choice property: the greedy choice holds because if we replaced j with another object, the value of the sack will stay the same or decrease, because the remaining items are sorted in decreasing order of $\frac{\text{value}}{\text{weight}}$.

We then prove that the above greedy algorithm is a 2-approximation algorithm for the 0-1 knapsack problem:

- From the algorithm, $a_k + \sum_{i=1}^{k-1} a_i > b$; the k-th item overflows the knapsack.
- Clearly, when we overflow the knapsack, we will put more value into the knapsack than the optimal solution to the fractional knapsack problem will:

$$\sum_{i=1}^{k} > OPT_f \geq OPT$$

where OPT_f and OPT are the values of the optimal solutions to the fractional and 0-1 knapsack problems respectively. $OPT_f \geq OPT$ necessarily because we do not include the final item if it does not fit wholly within the knapsack.

- Obviously:

$$2 \max \left\{ p_k, \sum_{i=1}^{k-1} p_i \right\} \geq p_k + \sum_{i=1}^{k-1} p_i$$

- Hence:

$$\max\left\{p_k, \sum_{i=1}^{k-1} p_i\right\} > \frac{OPT}{2}$$

The algorithm runs in $O(n \log n)$ time, which is polynomial because:

- The first two steps take $O(n)$ time.
- The sorting stage takes $O(n \log n)$ time.
- Finding the value of k and returning the result takes $O(n)$ time.

Question A5

Algorithm

We turn the problem of finding the representative set into one of finding a set cover.

REPRESENTATIVE-SET-APPROX(P, Δ):

1. Create a set for every protein, that contains the protein itself as well as any other proteins that are within Δ distance. That is, for every $p_i \in P$, set $R_i \leftarrow \{q \in P \mid d(p_i, q) \leq \Delta\}$.
2. Assign each set R_i a weight of 1, and insert the set of sets $\{R_i\}$ into the *SET-COVER-APPROX* algorithm as described in the lectures.
3. Return the set of proteins that correspond to the sets chosen by the set cover approximation algorithm.

Analysis of algorithm

SET-COVER-APPROX returns a covering whose weight is at most $H(n)$ times more than the optimal weight. Since we set all the weights to be 1, this means that we have at most $H(n)$ times more proteins than the optimal number selected in the representative set problem. Hence, we have an $H(n) = O(\log n)$ approximation to the representative set problem.