Homework No. 6 (Answers)

CSCI 4470/6470 Algorithms, CS@UGA, Fall 2017
Due Monday December 4, 2017

There are 150 points in total (130 points for undergraduates)

1. (25 points) Problem Min Vertex Cover is defined as
   Input: a graph $G = (V, E)$;
   Output: a subset $C \subseteq V$, called vertex cover, such that
   (1) $\forall (u, v) \in E$, either $u \in C$ or $v \in C$, and
   (2) $|C|$ is the minimum.

   Design an exhaustive search algorithm to solve problem Min Vertex Cover such that it runs in time $O(1.619^n)$ or better.

   Answers:

   We use an idea similar to the one to solve the Max Independent Set problem (see Lecture Note 5). The process picks an arbitrary vertex $v$ that is associated with at least one edge and branches into two different search paths. One is to include the vertex into the desired solution vertex cover set, and another to exclude it from the vertex set.

   The former case results in a subgraph $G_1$ to be recursively searched without the vertex $v$, while the latter case, all neighbors of $v$ have to be included in $C$, resulting in a subgraph $G_2$ to be recursively searched without the vertex $v$ nor its neighbors $\mathcal{N}(v)$. The algorithm terminates when the given graph is empty.
Algorithm pseudocode:

Algorithm \texttt{MinVertexCover} \((G)\)

1. \textbf{if} \(G = \emptyset\) \textbf{return} \((\emptyset)\)
2. \textbf{else} pick an arbitrary vertex \(v\) associated with at least one edge in \(G\)
3. let \(G_1\) be \(G\) with \(v\) removed
4. \(C_1 = \{v\} \cup \text{MinVertexCover} \((G_1)\)\)
5. let \(G_2\) be \(G\) with \(v\) and all its neighbors removed
6. \(C_2 = \mathcal{N}(v) \cup \text{MinVertexCover} \((G_2)\)\)
7. \textbf{if} \(|C_1| \leq |C_2|\) \textbf{return} \((C_1)\)
8. \textbf{else return} \((C_2)\)

Assume that the running time for \texttt{MinVertexCover} \((G)\) is \(T(n)\), where \(n = |G|\). Then the running time has the following recurrence

\[
T(n) = T(n - 1) + T(n - 1 - m) + O(n^2)
\]

where \(O(n^2)\) is the time to process graph after \(v\) is identified and \(m\) is the minimum number of neighbors that \(v\) can have.

Because \(v\) is associated with at least one edge, \(m \geq 1\). So

\[
T(n) = T(n - 1) + T(n - 2) + O(n^2)
\]

According to the lecture note 5, \(T(n) = O(1.6181^n) = O(1.619^n)\).
2. (25 points) Optimization problem 0-1 Knapsack is defined as:

**Input**: a set of $n$ items of values $v_i$ and sizes $s_i$, $i = 1, 2, \ldots, n$ and knapsack size $B$;

**Output**: a subset $A \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in A} s_i \leq B \text{ and } \sum_{i \in A} v_i \text{ is the maximum}$$

We formulate a decision problem 0-1 Knapsack Decision as follows:

**Input**: a set of $n$ items of values $v_i$ and sizes $s_i$, $i = 1, 2, \ldots, n$, bag size $B$, and value $k$;

**Output**: “yes” if and only if there is a packing of items into the bag with total value $\geq k$.

Prove that if there is a polynomial time algorithm for 0-1 Knapsack Decision, then there is a polynomial time algorithm for 0-1 Knapsack.

**Proof**

Let $V = \{v_1, \ldots, v_n\}$ and $S = \{s_1, \ldots, s_n\}$.

Assume we have a polynomial time algorithm $P$ for the 0-1 Knapsack Decision problem such that $P(S, V, B, k) = 1$ if and only if there is a subset $A \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in A} s_i \leq B \text{ and } \sum_{i \in A} v_i \geq k$$

We now use $P$ to construct another algorithm $Q$ to solve problem 0-1 Knapsack. The idea is to first use $P$ to determine the maximum value of $k$, called $k_{\text{max}}$, such that $P(S, V, B, k_{\text{max}}) = 1$. Then it uses $P$ to test if removing an item $i$ would make the packing still possible to reach the value $k_{\text{max}}$. If so, discard the item, otherwise, keep it. The process iterates through all items and the remaining items should all the optimal packing set.
Let $MaxValue = \sum_{i=1}^{n} v_i$.

Assume $P$ runs in time $p(n)$.

$Q$ works as:

1. binary search for a value $k_{\text{max}}$ in the range $[1..MaxValue]$, such that
   
   $P(S, V, B, k_{\text{max}}) = 1$ and
   
   $P(S, V, B, k_{\text{max}} + 1) = 0$;

2. let $A = \emptyset$;

3. for $i = 1, \ldots, n$ do
   
   let $V = V - \{v_i\}; S = S - \{s_i\}$;
   
   if $P(S, V, B, k_{\text{max}}) = 0$
   
   $A = A \cup \{i\}$;
   
   $V = V \cup \{v_i\}; S = S \cup \{s_i\}$;

4. return ($A$)

Now we calculate the running time of $Q$.

Because $MaxValue = O(2^n)$, step 1 binary search takes time $p(n) \times O(\log_2 MaxValue)$ is a polynomial. Steps 2-4 takes $O(n \times p(n))$ time. So $Q$ runs in polynomial time.
3. (10 points) Will the claim in Q2 still hold if the inequality “≥ k” in the definition of problem 0-1 Knapsack Decision is replaced with inequality “≤ k”? How about being replaced with equality “= k”?

**Answers:**

If the inequality “≥ k” in the definition of problem 0-1 Knapsack Decision is replaced with inequality “≤ k”, the claim would not hold. This is because without placing any item in the knapsack yields 0 value, which is ≤ k for any non-negative value and trivial question to answer. It would not characterize the difficult of the original 0-1 Knapsack problem.

If “≤ k” is replaced with equality “= k”, the claim may not be true. This is because the binary search (as outlined for Q2), may not necessarily correctly identify the value k_{max}.

For example, P may answer “no” to k = \frac{MaxValue}{2} and to k = maxValue but there may be a packing at the exact value \frac{3}{4}MaxValue, misleading Q to believe that there is no packing of value ≥ \frac{MaxValue}{2}.
4. (20 points) Prove that problem 0-1 Knapsack Decision is in the class NP.

**Proof:**

It suffices to construct the following deterministic polynomial time algorithm that can verify if a given certificate witnesses a packing of items results in a desired value.

The certificate for a packing is encoded as a binary string $b$ of length $n$, such that $b_i = 1$ indicates item $i$ is chosen to pack, $i = 1, \ldots, n$.

Let $V = \{v_1, \ldots, v_n\}$ and $S = \{s_1, \ldots, s_n\}$.

The verification algorithm on input: $V, S, B, k, b$

1. check if $b$ consists of exactly $n$ binary bits;
2. let $A = \{i : b_i = 1, 1 \leq i \leq n\}$;
3. check if $\sum_{i \in A} s_i \leq B$;
4. check if $\sum_{i \in A} v_i \geq k$;
5. return ("yes") if and only if steps 1, 3, and 4 all yields true.

The algorithm is deterministic and takes polynomial time to do all the 5 steps.
5. (10 points) Give a simple reason to explain that if a reduction function $f$ holds for $L_1 \leq L_2$, $f^{-1}$ usually does not hold for $L_2 \leq L_1$, where $f^{-1}$ is the inverse function of $f$.

**Answers:**

$f$ is a mapping function and it may map multiple objects to the same image. So it may not guarantee the existence of $f^{-1}$. 
6. (20 points, Graduate students only, bonus for undergraduates) Prove that if $L_1 \leq_p L_2$ and $L_2$ admits an algorithm that runs in time $O(n^{O(\log^2 n)})$, then $L_1$ can be solved in time $O(n^{O(\log n)})$ as well.

**Proof:**

Assume algorithm $F$ computes transformation function $f$, and algorithm $A_2$ solves for $L_2$ in time $O(n^{O(\log^2 n)})$.

We construct an algorithm to determine $x \in L_1$ for any string $x$, as follows.

1. It first transforms $x$ to $f(x)$;
2. then it calls $A_2(f(x))$; and
   
   return ("yes") if and only if $A_2(f(x)) = 1$.

Total time is the sum of time for $F$ and time for $A_2$.

$$O(|x|^c) + O(|f(x)|^{O(\log^2 |f(x)|)})$$

where $|f(x)|$ is actually $O(|x|^c)$ because $F$ runs in time $O(|x|^c)$.

So

$$O(|x|^c) + O(|f(x)|^{O(\log |f(x)|)}) = O(|x|^c + O((|x|^c)^{O(\log_2 |x|^c)})) = O(n^{O(\log^2 n)})$$
7. (20 points) Assume two languages \( L \) and \( L' \) can be polynomially reduced to each other, i.e., \( L \leq_p L' \) and \( L' \leq_p L \). Prove that, if \( L \) is NP-complete, so is \( L' \).

**Proof:**

Logically argue that \( L' \) is NP-hard and \( L' \) is in NP.

1. Because \( L \) is NP-complete, so \( L \) is NP-hard;
2. Because \( L \) is NP-hard and \( L \leq_p L' \), \( L' \) is also NP-hard;
3. Because \( L' \leq_p L \), there is a transformation \( f \) such that
   \[
   x \in L' \iff f(x) \in L
   \]
4. Because \( L \) is in NP, there is a polynomial time verifier \( A_L \) for \( L \) such that
   \[
   z \in L \iff \exists y A_L(z, y) = 1
   \]
5. Combine 3 and 4,
   \[
   x \in L' \iff \exists y A_L(f(x), y) = 1
   \]
6. Rename \( A_L(f(x), y) \) to be \( A_{L'}(x, y) \). Therefore,
   \[
   x \in L' \iff \exists y A_{L'}(x, y) = 1
   \]
   where \( A_{L'} \) is polynomial time algorithm.
7. From 6, \( L' \) is in NP.
8. We conclude that \( L' \) is NP-complete.
8. **(20 points)** Prove that if some NP-hard problem can be solved in deterministic polynomial time, then $\mathcal{P} = \mathcal{NP}$.

**Proof:**

Let $L$ be a NP-hard problem and its can be solved in deterministic polynomial time. Because $L$ is NP-hard, every problem $A \in \mathcal{NP}$, $A \leq_P L$.

If $L$ can be solved in deterministic polynomial time, so can $A$.

So all problems in $\mathcal{NP}$ are polynomial time solvable. Thus $\mathcal{P} = \mathcal{NP}$. 