COMP4500/7500 Advanced Algorithms & Data Structures
Sample Solution to Tutorial Exercise 9 (2019/2)[*]

School of Information Technology and Electrical Engineering, University of Queensland

October 10, 2019

1. (CLRS 34.1-1 p1060 [3rd]) Define the problem **LONGEST-PATH-LENGTH** as the relation that associates each instance of an undirected graph G and two vertices u and v with the number of edges k in a longest simple (no duplicates) path between the two vertices. Define the related decision problem **LONGEST-PATH** as

   $$\textbf{LONGEST-PATH} = \{\langle G, u, v, k\rangle : \begin{array}{l} G \text{ is an undirected graph, } u, v \in G.V, k \geq 0 \text{ is an integer and} \\ \text{there exists a simple path from } u \text{ to } v \text{ in } G \text{ with at least } k \text{ vertices}\} \end{array}$$

   Recall that a decision problem just returns either 0 or 1. Show that the optimisation problem **LONGEST-PATH-LENGTH** can be solved in polynomial time if and only if **LONGEST-PATH** $\in$ P.

   Sample solution. We consider the two directions of the "if and only if" separately. First assume we have a polynomial time algorithm SOLVE-LONGEST-PATH-LENGTH(G, u, v) that implements **LONGEST-PATH-LENGTH**, then the following algorithm implements **LONGEST-PATH**.

```
SOLVE-LONGEST-PATH(G, u, v, k)
1  if SOLVE-LONGEST-PATH-LENGTH(G, u, v) ≥ k
2      return 1
3  else
4      return 0
```

   Because SOLVE-LONGEST-PATH-LENGTH is polynomial time and SOLVE-LONGEST-PATH calls it once, SOLVE-LONGEST-PATH is also polynomial time.

   Second, assume **LONGEST-PATH** $\in$ P. Hence there exists a polynomial-time algorithm SOLVE-LONGEST-PATH(G, u, v, k) —not the above algorithm— that implements **LONGEST-PATH**. The following algorithm implements **LONGEST-PATH-LENGTH**. We use a result of -1 to indicate that there is no path between u and v; note that if u equals v there may be a path of length 0.

```
SOLVE-LONGEST-PATH-LENGTH(G, u, v)
1  k = −1
2  // Invariant: k ≠ −1 implies there exists a simple path in G from u to v of length at least k
3  while SOLVE-LONGEST-PATH(G, u, v, k + 1) == 1
4      k = k + 1
5  return k
```

   Because SOLVE-LONGEST-PATH is polynomial time and SOLVE-LONGEST-PATH-LENGTH calls it at most k times, SOLVE-LONGEST-PATH-LENGTH is also polynomial time. Note that because the path must be simple (no duplicates) the longest path length is bounded by the number of vertices in the graph and hence the number of iterations of the above loop is also.

2. (CLRS 34.1-3 p1060 [3rd]) Give a formal encoding of directed graphs as binary strings using an adjacency-matrix representation giving the complexity of the encoding's size in terms of the number of vertices in the graph. Do the same using an adjacency-list representation. Argue that the two representations are polynomial related.

[*]Copyright © reserved October 10, 2019

Sample solution. For a graph $G = (V, E)$, the adjacency matrix can be encoded by a single bit for each entry and hence is of size $\Theta(|V|^2)$. (One would also need to define the number of vertices, the encoding of which which would be of size $\Theta(\lg |V|)$.) Hence the size of the representation overall is $\Theta(|V|^2)$.

Each adjacency list can be represented as a sequence of at most $|V|$ vertex numbers of the adjacent vertices, where the representation of each vertex number is of size $\Theta(\lg |V|)$. Hence a single adjacency list is of size at $\Theta(|V| \lg |V|)$. The whole graph requires $|V|$ adjacency lists and hence the whole representation is of size $\Theta(|V|^2 \lg |V|)$.

To show that the two encodings are polynomially related one needs to argue that each encoding can be transformed to the other by a polynomial-time algorithm. Assuming G_L is encoded as an adjacency list, the following algorithm constructs the corresponding adjacency matrix encoding G_M.

```
ADJACENCY-LIST-TO-MATRIX($\langle G_L \rangle$)
1  $G_M.matrix$ = a new matrix of size $|G_L.V|^2$
2  for $u = 1$ to $|G_L.V|$
3      for $v = 1$ to $|G_L.V|$
4          $G_M.matrix[u][v] = 0$
5      for $v \in G_L.adjacent[u]$
6          $G_M.matrix[u][v] = 1$
7  return $G_M$
```

The above algorithm executes the innermost statements in the loop $|G_L.V|^2$ times, and the complexity of the individual statements is polynomial in the number of vertices. (The individual statements make use of indexing operations which can be implemented in terms of addition which can be performed in order linear time with respect to the number of bits in the numbers.) Hence the algorithm is polynomial in the number of vertices and hence is polynomial in the size of the input (which is also polynomial in the number of vertices).

Assuming G_M is encoded as an adjacency matrix, the following algorithm constructs the corresponding adjacency list encoding.

```
ADJACENCY-MATRIX-TO-LIST($\langle G_M \rangle$)
1  $G_L.adjacency$ = a vector of $|G_M.V|$ adjacency lists
2  for $u = 1$ to $|G_M.V|$
3      $G_L.adjacency[u]$ = a new empty list
4      for $v = 1$ to $|G_M.V|$
5          if $G_M.matrix[u][v] == 1$
6              $G_L.adjacency[u].add(v)$ // add $v$ to the adjacency list for $u$
7  return $G_L$
```

The above algorithm executes the innermost statements in the loop $|G_L.V|^2$ times, and the complexity of the individual statements is polynomial in the number of vertices. Hence the algorithm is polynomial in the number of vertices and hence is polynomial in the size of the input (which is also polynomial in the number of vertices). Technically the encodings are mapping binary strings and should also map non-instances to non-instances; we have only considered mapping valid instances above.

3. (CLRS 34.1-4 but for the subset-sum problem) Is the dynamic programming algorithm for the **SUBSET-SUM** problem for tutorial exercises 6 a polynomial time algorithm? You may assume all of the sizes within s are at most the total capacity C. The dynamic programming solution from the early tutorial exercise follows, except here the "sizes" parameter s has been made explicit.

```
SUBSETSUM($n, C, s$)
for $ci = 0$ to $C$
    $table[0, ci] = 0$
for $i = 1$ to $n$
    for $ci = 0$ to $C$
        if $s[i] > ci$    // Cannot include item $i$
            $table[i, ci] = table[i - 1, ci]$
        else    // $s[i] \leq ci$ — can include item $i$
            $with = table[i - 1, ci - s[i]] + s[i]$
            $without = table[i - 1, ci]$
            $table[i, ci] = $ MAX($with, without$)
return $table[n, C]$
```

Sample solution. The first loop is executed C times, while the second loop is executed $n.C$ times, and hence the latter dominates the complexity. The basic statements in the program involve array indexing, addition and subtraction all of which are polynomial time in the sizes of the numbers involved.

In answering this question we need to express the complexity in terms of the size of the input encoded as a binary string. The parameter n can be encoded in binary representation of size $\Theta(\lg n)$; similarly C requires size $\Theta(\lg C)$. There are n sizes in s, each of which has maximum value C and hence the complete array can be encoded as a sequence of numbers of size $\Theta(n \lg C)$. Hence the size m of the encoding of the parameters is $\Theta(n \lg C)$.

In general, C can be large with respect to n. For example, if we assume $n \in O(\lg C)$ then the size m of the parameters is $O((\lg C)^2)$, which implies $\sqrt{m} \in O(\lg C)$, which implies $2^{\sqrt{m}} \in O(C)$. The complexity of the program is $\Omega(n.C.(\lg C)^k)$ for some constant k and hence it is $\Omega(C)$ and hence it is also $\Omega(2^{\sqrt{m}})$ which is super-polynomial in the input size m. Hence the algorithm is not polynomial.

If we can assume $C \in O(n^k)$ for some constant k, then the size m of the parameters is $\Theta(n \lg n^k) = \Theta(nk \lg n) = \Theta(n \lg n)$ as k is a constant. The complexity of the program is $O(nC(\lg n)^b)$ for some constant b and hence also $O(nn^k(\lg n)^b) \subseteq O(n^d(\lg n)^d)$, where d is the maximum of the constants $k + 1$ and b. Hence the algorithm's complexity is $O((n \lg n)^d)$, which is polynomial in the size of the input.

4. (CLRS 34.2-1) Two graphs $G = (V, E)$ and $G' = (V', E')$ are ***isomorphic*** if there exists a bijection (i.e., a total, onto, one-to-one mapping) $f \in V \to V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. Consider the decision problem

$$\textbf{GRAPH-ISOMORPHISM} = \{\langle G, G' \rangle : G \text{ and } G' \text{ are isomorphic graphs } \}$$

Prove that **GRAPH-ISOMORPHISM** $\in$ NP by describing a polynomial-time algorithm to verify a solution to the decision problem.

Sample solution. For the purposes of this question we'll assume the graphs are encoded as adjacency matrices. For a certificate we make use of a mapping f from vertices V of G to vertices V' of G', which maps a vertex v in G to its corresponding vertex in G' (according to an isomorphism). The mapping can be represented by an array with $|V|$ elements, each of which is a vertex number in G'. The two graphs are isomorphic if they have the same number of vertices, there is an edge $(u, v) \in E$ if and only if there is an edge $(f[u], f[v]) \in E'$, and the mapping f is one-to-one.

```
VERIFY-GRAPH-ISOMORPHISM($G, G', f$)
if $|G.V| \neq |G'.V|$ or $f.length \neq |G.V|$
    return 0
for $u = 1$ to $|G.V|$
    // Check the vertex corresponding to $u$ is valid
    if $f[u] < 1$ or $|G.V| < f[u]$
        return 0
    // Check that $f$ is one-to-one (i.e., it has no duplicates)
    for $v = u + 1$ to $|G.V|$
        if $f[u] == f[v]$
            return 0
// Check that the edges in the two graphs correspond
for $u = 1$ to $|G.V|$
    for $v = 1$ to $|G.V|$
        if $G.matrix[u][v] \neq G'.matrix[f[u]][f[v]]$
            return 0
return 1
```

Note that as the representation of f is an array, we need to check that it only contains vertex numbers and has no duplicates. If duplicates were allowed the following two graphs could be mistakenly viewed as isomorphic using the mapping with $f[1] = 1$ and $f[2] = 1$.

$$\begin{aligned} G &= (\{1,2\}, \{(1,2),(2,1)\}) \\ G' &= (\{1,2\}, \{(1,1)\}) \end{aligned}$$

Technically the inputs are binary strings and one needs to check that they are valid representations of a pair of graphs and an array; such checking is straightforward but very messy and hence has been omitted here.