CSE 101 Homework 1 Solutions

Fall 2019

Question 1 (Bottle Game, 35 points). You want to collect exactly \( t \) ounces of water. To accomplish this, you have four bottles which store \( n_1, n_2, n_3, n_4 \) ounces of water respectively for some positive integers \( n_i \). You can attempt to reach your goal by applying some sequence of the following operations:

- Filling a bottle from the sink.
- Emptying a bottle onto the ground.
- Pouring one bottle into another until either the first empties or the second fills.

All four bottles start empty. Give an algorithm to determine whether or not it is possible to apply some sequence of the above operations in order to end up with some bottle containing exactly \( t \) ounces of water. For full credit, your algorithm should run in time \( O((n_1 + n_2 + n_3 + n_4)^3) \) or better.

Solution. We can model the above problem as a graph. The vertices (states) will represent how much liquid is present in each of the bottles. The edges will represent operations that take us from one state to the other. The question then would be, can we reach from the initial state (all bottles empty) to a state where exactly one bottle has \( t \) ounces of liquid.

The solution is divided into 3 parts. First, we come up with a slower algorithm that uses above construction. Second, we show that many states are unreachable when starting from \( (0,0,0,0) \) and performing the given bottle operations. Third, we improve our algorithm to reach required big O.

Slower Algorithm:

Graph Construction:

Formally, the graph \( G = (V,E) \) is constructed as follows:

- Every \( v \in V \) is of the form \( (b_1,b_2,b_3,b_4) \) and represents that bottle \( i \) contains \( b_i \) ounces of liquid. For example: The initial state is \( (0,0,0,0) \) and the state where exactly one bottle has \( t \) ounces can be \( (a,b,c,t) \), \( (a,b,t,c) \), \( (a,t,b,c) \) or \( (t,a,b,c) \) where \( a, b, c \) are integers. Let’s refer to these states as the final states.

- \( (v_i, v_j) \in E \) if we can reach state \( v_j \) from state \( v_i \) by performing exactly one of the available operations. For example: Given \( n_1 = 3, n_2 = 4, n_3 = 5 \) and \( n_4 = 6 \), there is an edge from \( (0,0,0,0) \) to \( (0,0,5,0) \). Similarly, there is an edge from \( (0,0,5,0) \) to \( (0,4,5,0) \) and so on.

Algorithm:

Run \( explore \) from the initial vertex \( (0,0,0,0) \). A state with one bottle having exactly \( t \) ounces is possible if and only if either of the states \( (a,b,c,t) \), \( (a,b,t,c) \), \( (a,t,b,c) \) or \( (t,a,b,c) \) have been visited after \( explore \) has been run (here \( a, b \) and \( c \) are integers). Pseudo-code follows:

1. Define \( V \) to contain all \( (n_1+1) \times (n_2+1) \times (n_3+1) \times (n_4+1) \) states.
2. Mark all vertices in \( V \) as unvisited.
3. Define \( E \) as follows:
   - For each \( v \) in \( V \):
     - For every \( u \) reachable from \( v \) in a single operation:
Let’s try to come up with a loose upper bound for explore. A sequence of operations on the initial state (0,0,0,0) will lead us to one of the final states. Also, if one of the final states is reachable from (0,0,0,0), then a sequence of operations exist that will lead us to one of the final states.

Correctness:
Vertex u is visited after running explore on v if and only if u is reachable from v. Hence, if a sequence of operations on the initial state (0,0,0,0) leads us to one of the final states, it implies that on running explore from (0,0,0,0), we will be able to reach one of the final states. Also, if one of the final states is reachable from (0,0,0,0), then a sequence of operations exist that will lead us to one of the final states.

Runtime Analysis:
Let’s try to come up with a loose upper bound for explore. Bottle i can contain a minimum of 0 and a maximum of \(n_i\) ounces of liquid. In other words, the amount of liquid in bottle i can have \(n_i + 1\) distinct values. Hence, the number of possible states are \((n_1 + 1)(n_2 + 1)(n_3 + 1)(n_4 + 1)\). Thus, \(|V| = (n_1 + 1)(n_2 + 1)(n_3 + 1)(n_4 + 1)\).

Also, at each state, we can perform at most 20 operations - fill up either of the 4 bottles, empty either of the 4 bottles or transfer liquid between two bottles (6 ways to select 2 bottles and 2 possible orderings for each selection). Hence, \(|E| \leq 20|V|\). Note that all the operations may not be possible at each state, this is an upper bound. So, explore routine runs in \(O(|V| + 20|V|) = O(21|V|) = O(|V|) = O((n_1 + 1)(n_2 + 1)(n_3 + 1)(n_4 + 1))\).

To understand the upper bound for graph construction (line 1-3 of pseudo code), note that line 1 can be accomplished by nested for loop in \(O(|V|)\) i.e. enumerating all possible states. Line 3 performs \(\sum(\text{out-degree})\) operations which is \(O(|E|) = O(20|V|) = O(|V|)\).

So the proposed algorithm runs in \(O(|V|) = O(n_1 + 1)(n_2 + 1)(n_3 + 1)(n_4 + 1) = O(n_1n_2n_3n_4)\).

Not all vertices are reachable using bottle operations from (0,0,0,0):
For operation (1) and operation (2), we must either empty or fill one of the bottles. Also, for operation (3), we transfer contents from one of the bottles to the other till either one gets empty or is filled up. Hence, for each reachable state at-least one bottle is either empty or at-least one bottle is filled. In other words, we cannot have a state where for every \(i\), \(b_i\) contains 1 to \(n_i - 1\) ounces of liquid as in these states none of the bottles are empty nor filled.

Hence, the number of potentially reachable states can be counted as follows. Number of potentially reachable states is bounded above by:

- (Number of states where \(b_1\) is empty) + (Number of states where \(b_1\) is full)
- (Number of states where \(b_2\) is empty) + (Number of states where \(b_2\) is full)
- (Number of states where \(b_3\) is empty) + (Number of states where \(b_3\) is full)
- (Number of states where \(b_4\) is empty) + (Number of states where \(b_4\) is full)

\[= 2(n_2 + 1)(n_3 + 1)(n_4 + 1) + 2(n_1 + 1)(n_3 + 1)(n_4 + 1) + 2(n_1 + 1)(n_2 + 1)(n_4 + 1) + 2(n_1 + 1)(n_2 + 1)(n_3 + 1)\]

which is clearly cubic and not bi-quadratic. Let us call this states list of cubic size as reduced states. On expanding, it can be shown algebraically that this is less than \(k(n_1 + n_2 + n_3 + n_4)^3\) for some constant \(k\) and sufficiently large \(n\).

Observe that listing all the reduced states takes \(O((n_1 + n_2 + n_3 + n_4)^3)\) by using nested for loops.

Faster Algorithm:
Now, we know that only reduced states, i.e. \(O((n_1 + n_2 + n_3 + n_4)^3)\) vertices can be reached from the initial state of (0,0,0,0). However, we cannot use the pseudo-code described above to achieve this run-time. This is because we are initializing \(n_1n_2n_3n_4\) vertices and marking them as un-visited in step 1 and 2 of the algorithm. This causes the overall run-time to become \(O(n_1n_2n_3n_4)\) even though explore from (0,0,0,0) never visits the vertices that are not in reduced states. So we can initialize \(V\) to only contain reduced states.

Pseudo-code follows:

1. Define \(V\) to contain only reduced states
2. Mark all vertices in \( V \) as un-visited.
3. Define \( E \) as follows:
   For each \( v \) in \( V \):
   - For every \( u \) reachable from \( v \) in a single operation:
     Insert \( u \) in the adjacency list of \( v \)
4. Run \( \text{explore}((0,0,0,0)) \)
5. if any \((t,a,b,c),(a,t,b,c),(a,b,t,c)\) or \((a,b,c,t)\) is in visited:
   return True (we can keep track of this easily by using a global variable)
6. else return False

So the number of vertices in our graph is of size of reduced states which is bounded by \( O((n_1 + n_2 + n_3 + n_4)^3) \). The number edges is still bounded by \( 20|V| \). Initializing the graph (line 1-3 of pseudo code) takes \( O((n_1 + n_2 + n_3 + n_4)^3) \) as we are just enumerating \( O(|V| + |E|) \) items. Hence, the algorithm runs in \( O((n_1 + n_2 + n_3 + n_4)^3) \).

Note that the correctness result from slow algorithm still holds because we have only removed vertices and edges that are un-visited by slow algorithm.

**Question 2** (Vacation Planning, 35 points). Sylvester is planning a trip to Graphania. He has a road map, showing how cities are connected by (two-way) roads. He also has a list of prices for the cheapest flight flying into each city and the cheapest flight flying out of each city. He wants to plan a trip whereby he flies into one city, drives to another city (assume that this costs nothing) and flies out for as little total money as possible.

(a) Give a linear time algorithm for solving this problem. [25 points]

(b) Rock slides close some of the lanes of some roads, turning them into one-way roads. Does the above algorithm still work? Why or why not? [10 points]

**Solution.**

(a) We can model the above problem by constructing an undirected graph \( G \), whose vertices are cities and whose edges are roads (for any two cities \( v_i \) and \( v_j \), there exists an edge \((v_i, v_j)\) if there is a road connecting them). Our algorithm will then use DFS to find all connected components of \( G \), calculate the minimum flight cost for each component, and return the smallest overall flight cost among all components. The reasoning is that if two vertices are in the same connected component, it should be possible for Sylvester to drive between the two cities.

**Algorithm.** Run DFS on \( G \) and find all connected components. For each connected component:

- Loop through all vertices and find the vertex with the cheapest flight in, and the vertex with the cheapest flight out.
- Compute the sum of (cheapest flight in) + (cheapest flight out)

Then return the smallest sum of (cheapest flight in) + (cheapest flight out) over all connected components.

**Correctness.** DFS will find all the connected components of \( G \). Given a vertex \( v \) in \( G \), the number of vertices in \( v \)’s connected component is precisely the number of vertices that are reachable from \( v \). We only have edges between vertices if there exists a road connecting them, so it is possible to drive between all of the cities in a connected component (no more and no fewer). Therefore, to find the cheapest trip within a connected component, we can just find the vertex with the cheapest flight in and the vertex with the cheapest flight out, and we know they will be reachable from each other. Since we cannot travel between connected components, the overall cheapest trip would be the minimum of the cheapest trips for each connected component.

The optimal answer to the problem (overall cheapest trip) needs to have the cheapest flight in and flight out be between two cities that are connected by roads, meaning that they would be in the same connected component. Since our algorithm finds the cheapest trip among all connected components, we conclude that our algorithm outputs the correct answer.
Runtime analysis. Let $|V|$ be the number of vertices (cities) in $G$, and $|E|$ be the number of edges (roads). DFS can find connected components in $O(|V| + |E|)$ time, since each vertex only needs to be visited once. Computing the cheapest flight for each connected component can be done in $O(|V|)$ time, by iterating through vertices and updating the cheapest flight in / flight out found for each component. Finding the overall cheapest sum can also be done in $O(|V|)$ time if we know the cheapest flight in and flight out for each component, since we just need to calculate each sum, then find the minimum among all sums we calculated. Therefore, the overall runtime of this algorithm is $O(|V| + |E|)$.

(b) No, the above algorithm will not work. Our algorithm uses DFS to find connected components and assumes that two vertices in the same connected component are reachable from each other. This works if $G$ is undirected, because in an undirected graph a vertex is reachable from another vertex if and only if they are in the same connected component. This means that we can just work within connected components. However, for directed graphs, it is no longer the case that vertices that are reachable from each other must be in the same strongly connected component – it is possible to have vertices in one strongly connected component that are reachable from vertices in other strongly connected components.

(Note: there still exists a linear time algorithm to solve this problem, but it is much more complicated.)

Question 3 (Cycles and Orderings, 30 points). Let $G$ be a directed graph and $C$ a cycle of its vertices. Assume that we run DFS on $G$ computing pre- and post-order numbers. For the following statements, either prove them or provide a counterexample.

(a) The vertex of $C$ with the smallest preorder number has the largest postorder number. [15 points]

(b) The vertex of $C$ with the largest preorder number has the smallest postorder number. [15 points]

Solution.

(a) Let $v_1, v_2, ..., v_n$ be the vertices of the cycle $C$, and let $v_1$ be the vertex of $C$ with the smallest preorder number. We will prove that $v_1$ also has the largest postorder number.

Since $v_1$ has the smallest preorder number, $v_1$ is the vertex of $C$ that will be explored first. Since $v_2, ..., v_n$ are also vertices in $C$, they must all be reachable from $v_1$. Therefore, we will have discovered all of the vertices $v_2, ..., v_n$ before we are finished exploring $v_1$. But since $v_1$ has the smallest preorder number, this means we must discover all of $v_2, ..., v_n$ while we are in the process of exploring $v_1$. We conclude that the vertex $v_1$ must also have the largest postorder number.

(b) This is not necessarily true. Below is a counterexample with 4 vertices, where DFS starts at vertex $v_1$. The cycle $C$ consisting of vertices $v_1, v_3$ and $v_4$ has $v_4$ with the largest preorder number, but $v_3$ with the smallest postorder number.

```
Preorder: 1
Postorder: 8

Preorder: 6
Postorder: 7

Preorder: 2
Postorder: 5

Preorder: 3
Postorder: 4
```