Illinois Institute of Technology
Department of Computer Science

Solutions to Homework Assignment 6

CS 430 Introduction to Algorithms
Spring Semester, 2016

Solution:

1. Let's first start with the simple case where we have only 3 jugs, and see how to trivially extend it to allow for infinitely many jugs in theory.

   For any vertex (i, j, k), we can create exactly 6 outgoing edges because we have 3 options for the jug from where we are pouring the water, and 2 options for the jug to where we are pouring the water (rule of product, $3 \times 2 = 6$). For the first edge, which determines the next state after we pour the water from the 1st jug to the 2nd one, we can determine its destination vertex as follows:

   - If $i = 0$: this edge is a self loop pointing to the same vertex (i, j, k).
   - If $i + j > c_2$: this edge points to the vertex $(i - (c_2 - j), c_2, k)$ because only the marginal amount of water $(c_2 - j)$ will be poured to the 2nd jug.
   - Otherwise: the edge points to the vertex $(0, i + j, k)$.

   The destination vertices of the other 5 edges can be achieved in the exactly same way with slight change in the parameters.

   Then, given any starting state (*i.e.,* vertex), we can lively explore the graph by finding the destination vertices of 6 edges for each vertex. Sometimes, a vertex may have more than one self loop, which is natural and accepted. During the exploration, we maintain the color and the predecessor (π) of each vertex, and if we see an already visited vertex (gray or black), we simply skip it just like the BFS only explores the white unvisited vertices. If we find a vertex containing the number t, we use the predecessors list to find out the trace from the starting state to this state, and this trace shows how to pour waters to get the target amount t. If we did not see any vertex having t but the exploration terminated because no new vertex is found, we report error saying "It is impossible to get t".

2. All vertices that are reachable from the starting vertex correspond to all possibly reachable states from the initial state of 3 jugs. Since our exploration process ensembles the BFS, our exploration will reach all nodes that are reachable from the starting vertex. Therefore, our exploration will find out the vertex containing t if it is there, and will report error if it is not in the implicit graph.

3. If c_1, c_2, c_3 and the parameters in the initial vertex are all integers, there are at most $(c_1+1)(c_2+1)(c_3+1)$ possible vertices because only integer differences will be applied. We also know there are 6 edges for each node, so $|V| = O(c_1c_2c_3)$ and $|E| = 6|V| = O(c_1c_2c_3)$. Since the the complexity of our exploration is same as the BFS, the complexity of our exploration is $O(|V| + |E|) = O(c_1c_2c_3)$.

   If they are fractional numbers, suppose the finest precision among the parameters is 0.0001, then at most there are $1/0.0001 \times c_1 + 1$ possible values for the amount of water at the 1st jug (all values within the interval [0,c_1]. Then, there are $(10000c_1 + 1)(10000c_2 + 1)(10000c_3 + 1)$ vertices at most in the implicit graph, and our complexity is still $O(c_1c_2c_3)$ because the precision is a constant. Then, the complexity is still $O(c_1c_2c_3)$ regardless of the precision.

4. In fact, the above algorithm can be applied to arbitrarily many jugs. If there are n jugs, each vertex will have n values indicating the amount of water in each jug, and there are $n!$ outgoing edges from each vertex. The rest is same. For each out-going edge, we can determine its destination vertex as above with 3 branches (if-if-otherwise).

5. (a) An Euler cycle is a single cycle that traverses each edge of G exactly once, but it might not be a simple cycle. An Euler cycle can be decomposed into a set of edge-disjoint simple cycles, however. If G has an Euler cycle, therefore, we can look at the simple cycles that, together, form the Euler cycle. In each simple cycle, each vertex in the cycle has one entering edge and one leaving edge. In each simple cycle, therefore, each vertex v has in-degree(v) = out-degree(v), where the degrees are either 1 (if v is on the simple cycle) or 0 (if v is not on the simple cycle). Adding the in- and out-degrees over all edges proves that if G has an Euler cycle, then in-degree(v) = out-degree(v) for all vertices v.

   We can prove the converse – that if in-degree(v) = out-degree(v) for all vertices v, then G has an Euler cycle – using a constructive proof. Let us start at a vertex u and, via random traversal of edges, create a cycle. We know that once we take any edge entering a vertex $v \neq u$, we can find an edge leaving v that we have not yet taken. Eventually, we get back to vertex u, and if there are still edges leaving u that we have not taken, we can continue the cycle. Eventually, we get back to vertex u and there are no untaken edges leaving u. If we have visited every edge in the graph G, we are done. Otherwise, since G is connected, there must be some unvisited edge leaving a vertex, say v, on the cycle. We can traverse a new cycle starting at v, visiting only previously unvisited edges, and we can splice this cycle into the cycle we already know. That is, if the original cycle is $\langle u, \cdots, v, w, \cdots, u\rangle$, and the new cycle is $\langle v, x, \cdots, v\rangle$, then we can create the cycle $\langle u, \cdots, v, x, \cdots, v, w, \cdots, u\rangle$. We continue this process of finding a vertex with an unvisited leaving edge on a visited cycle, visiting a cycle starting and ending at this vertex, and splicing in the newly visited cycle, until we have visited every edge.

   Suppose a graph G contains an Euler path P. Then, for every vertex v, P must enter and leave v the same number of times, except when it is either the starting vertex or the final vertex of P. When the starting and final vertices are distinct, there are precisely 2 odd degree vertices. When these two vertices coincide, there is no odd degree vertex.

   Conversely, suppose G contains 2 odd degree vertex u and v. Then, we temporarily add a dummy edge (u, v) to G. Now the modified graph contains no odd degree vertex. By the above proof of Euler cycle, this graph contains an Euler cycle that also contains (u, v). Remove (u, v) from the cycle, and now we have an Euler path where u and v serve as initial and final vertices.

   (b) We can use Hierholzer's algorithm to find a Eulaer cycle or path of which the idea is the same as the constructive proof. If the graph contains two odd degree vertices, we temporarily add a dummy edge between them and apply Hierholzer's algorithm to find the Euler cycle, and then remove the dummy edge.

   - Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
   - As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.

   By using a data structure such as a doubly linked list to maintain the set of unused edges incident to each vertex, to maintain the list of vertices on the current tour that have unused edges, and to

maintain the tour itself, the individual operations of the algorithm (finding unused edges exiting each vertex, finding a new starting vertex for a tour, and connecting two tours that share a vertex) may be performed in constant time each, so the overall algorithm takes linear time.