HW7 Solution, CS430, Spring 2017

April 9, 2017

1 Water pouring problem

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.

(a)

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".

(b)

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.

(c)

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.

(d)

When implementing your algorithm, notice that 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).

2 Ethnographer's problem

If we have the graph indicating the causal relations ($A \to B$ indicates A must precede B), we can simply find whether there is a cycle to see if there is any inconsistency. To create such a causal relation graph (similar to PERT graph), we do the followings.

First, for any person P_i, we create two nodes B_i, D_i to represent the events P_i's birth and death respectively. Then, we create the following edges:

1. $B_i \to D_i$ because a person must be born before he dies.
2. $D_i \to B_j$ if P_i died before P_j was born (fact (a)).
3. $B_i \to D_j$ and $B_j \to D_i$ if P_i and P_j were both alive at some time. The fact that they were alive at the same time means either must have been born before the other dies.

After this causal relation graph is constructed, we can run DFS to find the cycle in the graph as instructed in the recitation (*i.e.*, there is a cycle if you encounter a gray node throughout the DFS). If there is a cycle, these facts are not inconsistent to each other, otherwise we can do the topological sorting after the DFS terminates to find out the linear order of the facts.

3 Problem 23-4 page 641

Algorithm a.

Proof : The algorithm terminates when no more edges can be removed, which must result in a spanning tree. Let's assume the spanning tree we achieved from this algorithm is not a minimum spanning tree. We can try replacing any edge with the already removed edge to get another spanning tree, hoping to get a spanning tree with smaller weight. However, all removed edges' weights are greater than existing edges in the spanning tree because we chose the edges to be deleted in the non-increasing order by their weights. Therefore, it is not possible to get a different spanning tree with smaller weight. This concludes the tree is the MST.

Most efficient implementation : Sorting in line 1 can be done in $O(|E| \lg |E|)$ using either merge sort or heap sort. The connectivity check at line 4 can be done by running the DFS on the graph, which is $O(|V| + |E|)$ at each check, leading to $O(|E|(|V| + |E|))$ in total. Then, the final complexity of this implementation is $O(|E|^2 + |E||V|)$.

Algorithm b.

Disproof : The algorithm also returns a spanning tree, but this is clearly not a MST because we may have always chosen the minimum-weight edge in the random selection in 2.
Most efficient implementation : Union in line 3 and 4 can be implemented via disjoint-set operation ($O(\alpha(|V|))$), and the cycle check can be done via DFS ($O(|V| + |E|)$). The total complexity of this implementation is $O(|E|(\alpha(|V|) + |V| + |E|)) = O(|E|(\alpha(|V|) + |E|))$.

Algorithm c.

Proof : This algorithm also produces a spanning tree. To see whether it is a MST, we can construct a loop invariant first. Suppose after the i-th iteration, E^i is the set of edges that have been considered till the i-th iteration in the loop (*i.e.,* $E^i = \{e_1, e_2, e_3, \cdots, e_{i-1}, e_i\}$), and V^i is the set of vertices connected to at least one edge in E^i. Then, T is always a forest throughout the loop, and G^i is always the a sub-graph of $G = (V, E)$ (*i.e.,* $V^i \subseteq V$ and $E^i \subseteq E$).

Loop invariant: T is always a minimum spanning forest (MSF, an analogue of the MST) within the sub-graph G^i after the i-th iteration.

If the above loop invariant holds, the T after the last iteration is a MSF, and T is also a spanning tree of G, therefore T would be a MST. The next step is to prove that this loop invariant is true, which is done by the following proof by induction.

Proof.
Base case: After the first iteration, $E_1 = \{e_1\}$, and V^1 is the set of two nodes incident to e_1, and obviously T is a MST within $G_1 = (V_1, E_1)$ because there is only one edge in this sub-graph, and it is also a MSF.
Inductive Hypothesis: After the k-th iteration, T is a MSF within the sub-graph G^k.
Inductive Step & Proof: Before the $(k + 1)$-th iteration (which is also after the k-th iteration), according to the hypothesis, T is a MSF. During the $(k+1)$-th iteration, we add a random edge (denoted as e_{k+1}) to T first, and we have two cases as follows.

$T' = T \cup e_{k+1}$ `has a cycle:` We remove the maximum-weight edge e' on the cycle, therefore $T' - e'$ must have a smaller weight than the previous tree $T = T' - e_{k+1}$. Therefore, T' is a MSF within the sub-graph G^{k+1} after the $(k + 1)$-th iteration.
$T' = T \cup e_{k+1}$ `has no cycle:` T used to be the MSF, and we added an edge e_{k+1} to two trees (or nodes) to connect them. e_{k+1} at this point is the only edge connecting these two trees (or nodes) in the sub-graph G^{k+1}, therefore $T' = T \cup e_{k+1}$ is still a MSF.

In either case, T' is still a MSF.
Conclusion: Since T after the 1st iteration was a MSF, and we also have proved that T continues to be a MSF after an iteration if it was the MSF before it, we can conclude that T is constantly a MSF after any iteration of the loop in Algorithm c. □

Therefore, the aforementioned loop invariant holds for this loop, and therefore T is a MST within the graph G.
Most efficient implementation : As mentioned above, union in line 3 can be done in $O(\alpha(|V|))$, the cycle check in line 4 can be done in $O(|V| + |E|)$, the maximum-weight edge finding in line 5 can be done in $O(1)$ if we have maintained the max-heap on the edges, and line 6 can be done via extract-max in $O(\lg |E|)$. Therefore, the total complexity of this implementation is $O(|E|(\alpha(|V|) + |E|))$.