CS/ECE 374 ✧ Fall 2018

Homework 7

Due Wednesday, October 31, 2018 at 10am

Groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

The following unnumbered problems are not for submission or grading. No solutions for them will be provided but you can discuss them on Piazza.

- Consider a directed graph G, where each edge is colored either red, white, or blue. A walk in G is called a *French flag walk* if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \to v_1 \to \ldots \to v_k$ is a French flag walk if, for every integer i, the edge $v_i \to v_{i+1}$ is red if $i \mod 3 = 0$, white if $i \mod 3 = 1$, and blue if $i \mod 3 = 2$. Describe an efficient algorithm to find all vertices in a given edge-colored directed graph G that can be reached from a given vertex v through a French flag walk.
- Describe a linear time algorithm that given a directed graph $G = (V, E)$ and a node $s \in V$ decides whether there is a cycle containing s. Do the same when G is undirected.

0. **For your information.** You have seen in the lab several problems that illustrate the power of graphs and graph search to model a variety of puzzles and games. There are several interesting problems of this type at the end of Jeff's notes. See also Spring'18 home work on this topic: https://courses.engr.illinois.edu/cs374/sp2018/A/homework/hw7.pdf. You should be able to quickly see how to model the configurations via vertices and how to enforce the move rules via edges in the graph.

1. Let $G = (V, E)$ be *directed* graph. A subset of vertices are colored red and a subset are colored blue and the rest are not colored. Let $R \subset V$ be the set of red vertices and $B \subset V$ be the set of blue vertices.

   - Describe an efficient algorithm that given G and two nodes $s, t \in V$ checks whether there is an s-t path in G that contains at most one red vertex and at most one blue vertex. For simplicity assume that s, t do not have colors. Ideally your algorithm should run in $O(n + m)$ time where $n = |V|$ and $m = |E|$. Do not try to invent a new algorithm. Come up with a way to create a new graph G' and use a standard algorithm on G'.
   - Here is a small variation where edges are colored instead of vertices. Some of the edges in G are colored red and some are colored blue and the rest are not colored. Let $R \subset E$ be the red edges and $B \subset E$ be the blue edges. Describe an efficient algorithm that given G and two nodes s, t checks whether there is an s-t path that contains at most one red edge and at most one blue edge. Reduce the problem to the one in the previous part.

No proofs necessary but your algorithms should be clear.

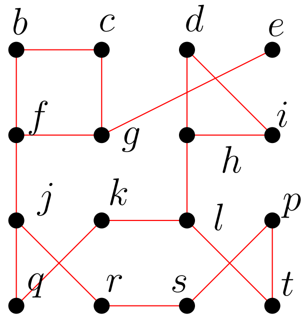
2. Let $G = (V, E)$ be a directed graph.

   - Describe a linear-time algorithm that outputs all the nodes in G that are contained in some cycle. More formally you want to output
   $$S = \{v \in V \mid \text{there is some cycle in } G \text{ that contains v}\}.$$
   - Describe a linear time algorithm to check whether there is a node $v \in V$ such that v can reach every node in V. First solve the problem when G is a DAG and then generalize it via the meta-graph construction.

   No proofs necessary but your algorithm should be clear. Use known algorithms as black boxes rather. In particular the linear-time algorithm to compute the meta-graph is useful here.

3. Given an *undirected* connected graph $G = (V, E)$ an edge (u, v) is called a cut edge or a bridge if removing it from G results in two connected components (which means that u is in one component and v in the other). The goal in this problem is to design an efficient algorithm to find *all* the cut-edges of a graph.

   - What are the cut-edges in the graph shown in the figure?



   - Given G and edge $e = (u, v)$ describe a linear-time algorithm that checks whether e is a cut-edge or not. What is the running time to find all cut-edges by trying your algorithm for each edge? No proofs necessary for this part.
   - Consider *any* spanning tree T for G. Prove that every cut-edge must belong to T. Conclude that there can be at most $(n - 1)$ cut-edges in a given graph. How does this information improve the algorithm to find all cut-edges from the one in the previous step?
   - Suppose T is any spanning tree of G. Root it at some arbitrary node. Prove that an edge (u, v) in T where u is the parent of v is a cut-edge iff there is no edge in G, other than (u, v), with one end point in T_v (sub-tree of T rooted at v) and one end point outside T_v.
   - Now consider the DFS tree T. Use the property in the preceding part to design a linear-time algorithm that outputs all the cut-edges of G. What additional information can you maintain while running DFS? Recall that there are no cross-edges in a DFS tree T. You don't have to prove the correctness of the algorithm.

Solved Problem

4. Professor McClane takes you out to a lake and hands you three empty jars. Each jar holds a positive integer number of gallons; the capacities of the three jars may or may not be different. The professor then demands that you put exactly k gallons of water into one of the jars (which one doesn't matter), for some integer k, using only the following operations:

   (a) Fill a jar with water from the lake until the jar is full.

   (b) Empty a jar of water by pouring water into the lake.

   (c) Pour water from one jar to another, until either the first jar is empty or the second jar is full, whichever happens first.

   For example, suppose your jars hold 6, 10, and 15 gallons. Then you can put 13 gallons of water into the third jar in six steps:

   - Fill the third jar from the lake.
   - Fill the first jar from the third jar. (Now the third jar holds 9 gallons.)
   - Empty the first jar into the lake.
   - Fill the second jar from the lake.
   - Fill the first jar from the second jar. (Now the second jar holds 4 gallons.)
   - Empty the second jar into the third jar.

   Describe and analyze an efficient algorithm that either finds the smallest number of operations that leave exactly k gallons in any jar, or reports correctly that obtaining exactly k gallons of water is impossible. Your input consists of the capacities of the three jars and the positive integer k. For example, given the four numbers $6, 10, 15$ and 13 as input, your algorithm should return the number 6 (for the sequence of operations listed above).

Solution: Let A, B, C denote the capacities of the three jars. We reduce the problem to breadth-first search in the following directed graph:

- $V = \{(a,b,c) \mid 0 \le a \le p \text{ and } 0 \le b \le B \text{ and } 0 \le c \le C\}$. Each vertex corresponds to a possible ***configuration*** of water in the three jars. There are $(A+1)(B+1)(C+1) = O(ABC)$ vertices altogether.
- The graph has a directed edge $(a,b,c) \to (a', b'c')$ whenever it is possible to move from the first configuration to the second in one step. Specifically, there is an edge from (a,b,c) to each of the following vertices (except those already equal to (a,b,c)):
  - $(0,b,c)$ and $(a,0,c)$ and $(a,b,0)$ — dumping a jar into the lake
  - (A,b,c) and (a,B,c) and (a,b,C) — filling a jar from the lake
  - $\left\{\begin{array}{ll}(0, a+b, c) & \text{if } a+b \le B \\ (a+b-B, B, c) & \text{if } a+b \ge B\end{array}\right\}$ — pouring from the first jar into the second
  - $\left\{\begin{array}{ll}(0, b, a+c) & \text{if } a+c \le C \\ (a+c-C, b, C) & \text{if } a+c \ge C\end{array}\right\}$ — pouring from the first jar into the third
  - $\left\{\begin{array}{ll}(a+b, 0, c) & \text{if } a+b \le A \\ (A, a+b-A, c) & \text{if } a+b \ge A\end{array}\right\}$ — pouring from the second jar into the first

- $\left\{\begin{matrix}(a,0,b+c) & \text{if } b+c \le C \\ (a,b+c-C,C) & \text{if } b+c \ge C\end{matrix}\right\}$ — pouring from the second jar into the third
- $\left\{\begin{matrix}(a+c,b,0) & \text{if } a+c \le A \\ (A,b,a+c-A) & \text{if } a+c \ge A\end{matrix}\right\}$ — pouring from the third jar into the first
- $\left\{\begin{matrix}(a,b+c,0) & \text{if } b+c \le B \\ (a,B,b+c-B) & \text{if } b+c \ge B\end{matrix}\right\}$ — pouring from the third jar into the second

Since each vertex has at most 12 outgoing edges, there are at most $12(A+1)(B+1)(C+1) = O(ABC)$ edges altogether.

To solve the jars problem, we need to find the ***shortest path*** in G from the start vertex $(0,0,0)$ to any target vertex of the form $(k,\cdot,\cdot)$ or $(\cdot,k,\cdot)$ or $(\cdot,\cdot,k)$. We can compute this shortest path by calling ***breadth-first search*** starting at $(0,0,0)$, and then examining every target vertex by brute force. If BFS does not visit any target vertex, we report that no legal sequence of moves exists. Otherwise, we find the target vertex closest to $(0,0,0)$ and trace its parent pointers back to $(0,0,0)$ to determine the shortest sequence of moves. The resulting algorithm runs in $O(V+E) = \boldsymbol{O(ABC)}$ ***time***.

We can make this algorithm faster by observing that every move either leaves at least one jar empty or leaves at least one jar full. Thus, we only need vertices (a,b,c) where either $a=0$ or $b=0$ or $c=0$ or $a=A$ or $b=B$ or $c=C$; no other vertices are reachable from $(0,0,0)$. The number of non-redundant vertices and edges is $O(AB+BC+AC)$. Thus, if we only construct and search the relevant portion of G, the algorithm runs in $\boldsymbol{O(AB+BC+AC)}$ ***time***. ■

Rubric (for graph reduction problems): 10 points:
- 2 for correct vertices
- 2 for correct edges
  - −½ for forgetting "directed"
- 2 for stating the correct problem (shortest paths)
  - "Breadth-first search" is not a problem; it's an algorithm.
- 2 points for correctly applying the correct algorithm (breadth-first search)
  - −1 for using Dijkstra instead of BFS
- 2 points for time analysis in terms of the input parameters.
- Max 8 points for $O(ABC)$ time; scale partial credit