CS/ECE 374 Sec A ✧ Spring 2023

Homework 8

Due Wednesday, March 29, 2023 at 10am

---

0. **Not to submit:** Several problems can be solved via graph modeling. GPS 8 gives you some examples. Solve the following from Jeff Erickson's chapter on basic graph algorithms.

   - Problem 18.
   - Problem 20.
   - Problem 26, part (a).

1. Let $G = (V, E)$ be *directed* graph. Each node $v \in V$ is given a color from $\{0, 1, 2, \ldots, k\}$; let $c(v)$ denote the color of v. Let X and Y be two non-empty sets of nodes where $X \cap Y = \emptyset$.

   - Describe an efficient algorithm that given G, the colors $c(v), v \in V$, and X, Y, checks whether there is a path from some node $u \in X$ to some node $v \in Y$ such that the path contains at most three nodes with colors that are not 0. 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. That is, now each edge $e \in E$ has a color $c(e)$ where $c(e) \in \{0, 1, 2, \ldots, k\}$ (the nodes do not have colors). Now the goal is to check whether there is a path from some node in X to some node in Y such that the path contains at most three edges with colors that are not 0. Reduce the problem to the one in the previous part.

2. For both problems below use the paradigm of solving the problem on a DAG and using the insights to solve it on a general directed graph via the meta-graph. For both problems assume that the input consists of a directed graph $G = (V, E)$ with n nodes and m edges.

   - Call a node u *happy* if every node $v \in V$ can reach u. Describe a linear-time algorithm to check whether G contains at least k happy nodes where k is a given input parameter.
   - Let $G = (V, E)$ be a directed graph. We have seen a linear-time algorithm that checks whether G is strongly connected. Suppose G is not strongly connected. Describe a linear time algorithm to check if one can add at most *two* edges to G to make it strongly connected. Argue about the correctness of your algorithm. *Hints:* First solve the problem where adding one edge is necessary and sufficient. Think of an example DAG such that two additional edges are necessary and sufficient to make it strongly connected. Think of an minimal example of a DAG such that two edges are insufficient to make it strongly connected.

3. **Not to submit:** This question is about cycles in graphs.

- Describe a linear time algorithm that given a *directed* graph $G = (V, E)$ and a node $s \in V$ outputs a directed cycle containing s if there is at least one, or correctly states that there is no directed cycle containing s.
- Describe a linear time algorithm that given an *undirected* graph $G = (V, E)$ and a node $s \in V$ outputs a cycle containing s if there is at least one, or correctly states that there is no cycle containing s.
- Describe a linear-time algorithm that given a *directed* graph outputs all the nodes in G that are contained in some cycle. More formally you want to output

$$S = \{v \in V \mid \text{there is some cycle in } G \text{ that contains v}\}.$$

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

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