# HW 7: Extra problems

Instructor: Sariel Har-Peled

**CS/ECE 374: Algorithms & Models of Computation, Fall 2017** Version: **1.0**

**1** 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.

**2** 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.

**3** Let $G = (V, E)$ be directed graph. A subset of edges are colored red and a subset are colored blue and the rest are not colored. Let $R \subset E$ be the set of red edges and $B \subset E$ be the set of blue edges. 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 edge and at most one blue edge. Ideally your algorithm should run in $O(n + m)$ time where $n = |V|$ and $m = |E|$.

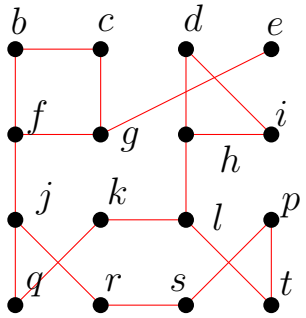
**4** The police department in the city of Shampoo-Banana has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. The city needs an algorithm to check whether the mayor's contention is indeed true.

- Formulate this problem graph-theoretically, and describe an efficent algorithm for it.
- Suppose it turns out that the mayor's original claim is false. Call an intersection $u$ *good* if any intersection $v$ that you can reach from $u$ has the property that $u$ can be reached from $v$. Now the mayor claims that over 95% of the intersections are good. Describe an efficient algorithm to verify her claim. Your algorithm should basically be able to find all the good intersections.

Ideally your algorithms for both parts should run in linear time. You will receive partial credit for a polynomial-time algorithm.

**5** 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 a spanning tree of $G$ rooted at $r$. 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 $T_v$.
- Use the property in the preceding part to design a linear-time algorithm that outputs all the cut-edges of $G$. You don't have to prove the correctness of the algorithm but you should point out how your algorithm ensures the desired property. *Hint:* Consider a DFS tree $T$ and some additional information you can compute during DFS. You may want to run DFS on the example graph with the cut edges identified.

**6** 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:

1. Fill a jar with water from the lake until the jar is full.
2. Empty a jar of water by pouring water into the lake.
3. 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 $12(A+1)(B+1)(C+1) = O(ABC)$ 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
  - 1/2 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

**7** You've been hired to store a sequence of $n$ books on shelves in a library. The order of the books is fixed by the cataloging system and cannot be changed; each shelf must store a contiguous interval of the given sequence of books. You are given two arrays $H[1 \,..\, n]$ and $T[1 \,..\, n]$, where $H[i]$ and $T[i]$ are respectively the height and thickness of the $i$th book in the sequence. All shelves in this library have the same length $L$; the total thickness of all books on any single shelf cannot exceed $L$.

1. Suppose all the books have the same height $h$ (that is, $H[i] = h$ for all $i$) and the shelves have height larger than $h$, so each book fits on every shelf. Describe and analyze a greedy algorithm to store the books in as few shelves as possible. (**Hint:** The algorithm is obvious, but why is it correct?)
2. That was a nice warmup, but now here's the real problem. In fact the books have different heights, but you can adjust the height of each shelf to match the tallest book on that shelf. (In particular, you can change the height of any empty shelf to zero.) Now your task is to store the books so that the sum of the heights of the shelves is as small as possible. Show that your greedy algorithm from part (a) does *not* always give the best solution to this problem.
3. Describe and analyze an algorithm to find the best assignment of books to shelves as described in part (b).

**8** Consider a directed graph $G$, where each edge is colored either red, white, or blue. A walk[1] 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 $v_0 \to v_1 \to \cdots \to v_k$ is a French flag walk if, for every integer $i$, the edge $v_i \to v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

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.

**9** ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[2] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

Each car has a *position* and a *velocity*, both with integer $x$- and $y$-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*. The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0, 0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race.[3] The race ends when the first car reaches a position inside the finishing area.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the "starting area" is the first column, and the "finishing area" is the last column.

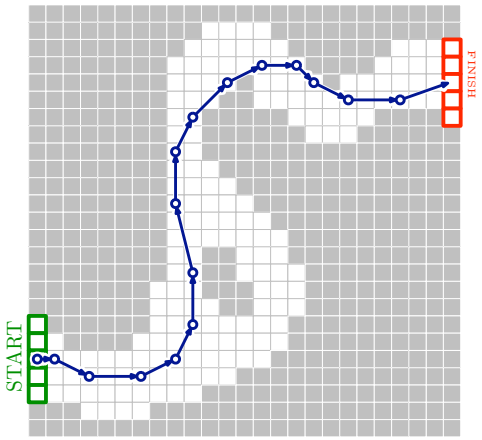
Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack. (**Hint:** Build a graph. No, not that graph, a different one. What are the vertices? What are the edges? What problem is this?)

[1] Recall that a ***walk*** in a directed graph $G$ is a sequence of vertices $v_0 \to v_1 \to \cdots \to v_k$, such that $v_{i-1} \to v_i$ is an edge in $G$ for every index $i$. A ***path*** is a walk in which no vertex appears more than once.

[2] The actual game is a bit more complicated than the version described here. See `http://harmmade.com/vectorracer/` for an excellent online version.

[3] However, it is not necessary for the line between the old position and the new position to lie entirely within the track. Sometimes Speed Racer has to push the A button.

| velocity | position |
|---|---|
| $(0, 0)$ | $(1, 5)$ |
| $(1, 0)$ | $(2, 5)$ |
| $(2, -1)$ | $(4, 4)$ |
| $(3, 0)$ | $(7, 4)$ |
| $(2, 1)$ | $(9, 5)$ |
| $(1, 2)$ | $(10, 7)$ |
| $(0, 3)$ | $(10, 10)$ |
| $(-1, 4)$ | $(9, 14)$ |
| $(0, 3)$ | $(9, 17)$ |
| $(1, 2)$ | $(10, 19)$ |
| $(2, 2)$ | $(12, 21)$ |
| $(2, 1)$ | $(14, 22)$ |
| $(2, 0)$ | $(16, 22)$ |
| $(1, -1)$ | $(17, 21)$ |
| $(2, -1)$ | $(19, 20)$ |
| $(3, 0)$ | $(22, 20)$ |
| $(3, 1)$ | $(25, 21)$ |




A 16-step Racetrack run, on a $25 \times 25$ track. This is *not* the shortest run on this track.