CS/ECE 374 B — Fall 2019

Homework 7

Due Tuesday, November 12 at 8 p.m.

---

1. (a) Suppose that you have a graph $G = (V, E)$ where each node is labeled with a digit; i.e., there is a function $\mathsf{label} : V \to \{0, \ldots, 9\}$. Design an efficient algorithm that, given two nodes, $s, t \in V$, decides whether there is a path from s to t in which the node labels follow the pattern 374374…. I.e., if the path is $s = v_0 \to v_1 \to \cdots \to v_n = t$, then the label of v_i is 3 if $i \bmod 3 \equiv 0$, 7 if $i \bmod 3 \equiv 1$, and 4 if $i \bmod 3 \equiv 2$.

   Solution: We can simply use a modified DFS algorithm to decide whether there is a path with the pattern 374374...

```
1:  function PATTERN(G, Visited, v, t, prev)
2:      Visited[v] = True
3:      if v = t then
4:          return True
5:      if prev = 3 then
6:          next = 7
7:      else if prev = 7 then
8:          next = 4
9:      else if prev = 4 then
10:         next = 3
        temp = False
11:     for each edge vw do
12:         if label(w) = next and Visited[w] = False then
13:             temp = temp or Pattern(G, Visited, w, t, next)
        return temp
14: function MAIN(G)
15:     Visited = [False] * (len(G.nodes))
16:     return Pattern(G, Visited, s, t, 4)
```

Since the algorithm traverses the nodes and edges only once, the runtime of this algorithm is $O(|V| + |E|)$

Another option is to transform G into a directed graph $G' = (V', E')$, where $v \to u \in E'$ if $\{v, u\} \in E$ and $(\ell(v), \ell(u)) \in \{(3,7), (7,4), (4,3)\}$. It is easy to see that any path in G that has the label structure 374374… corresponds to a path in G'. To complete the problem, we need to verify that s has label 3; if yes, it runs DFS on G' to see if t is reachable, otherwise it returns false.

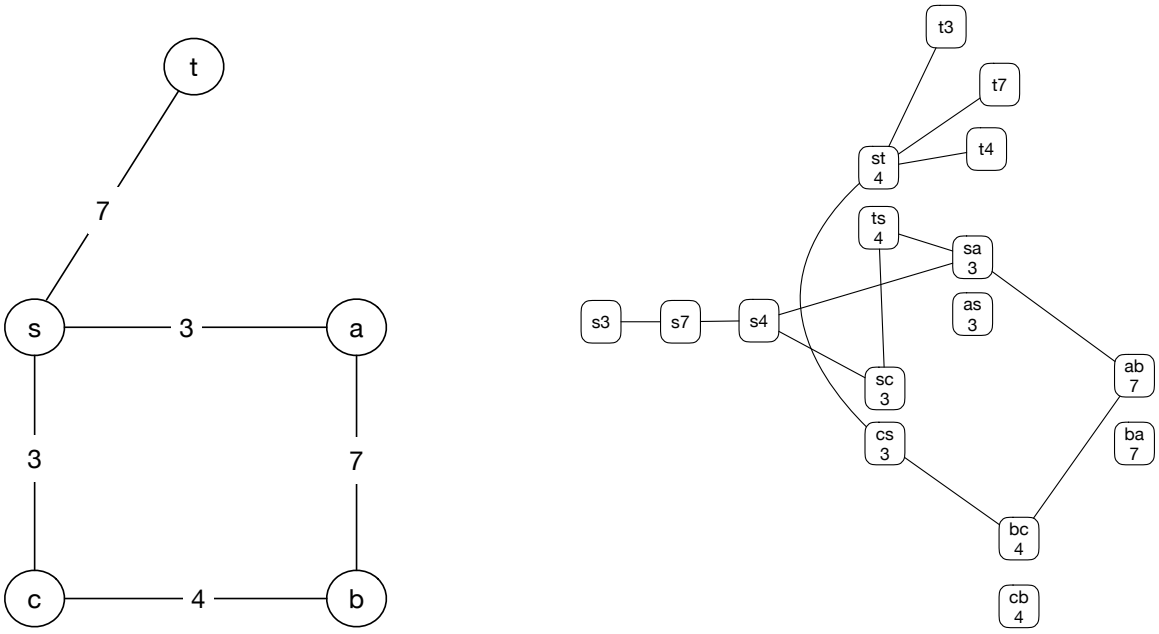
The transformation takes $O(V + E)$ time and the DFS also takes $O(V + E)$, so the overall complexity is $O(V + E)$. ■

(b) Suppose now that the labels are on edges instead of vertices; i.e., $\mathsf{label} : E \to \{0, \ldots, 9\}$. Design an algorithm that reduces this problem to the previous part, i.e., calls the previous part as a subroutine.

Solution: Since now the labels are on edges instead of vertices, we have to reconstruct a graph that has labels on the vertices. The straightforward mapping would be to create a vertex in the new graph for each edge; however, this presents a slight problem: suppose ab, bc, and bd are edges in G, then $a \to b \to c$ and $c \to b \to d$ are both valid paths. This would suggest that the vertex representing bc should have an edge with both ab and bd, but this creates the possibility of the path in G' $ab \to bc \to bd$, which does not correspond to a valid path in G. Instead, we a vertex for each direction of an edge: $V' = \{(u,v) | uv \in E\}$. Now we can have an edge between (u,v) and (v,w) but we have make sure that it's only taken in one direction. Fortunately, the pattern of the paths accepted in part (a) helps us ensure that; if the label on uv is 3 and the label on vw is 7, we add a G' edge between (u,v) and (v,w); on the other hand, if the label on uv is 7 the label on vw is 3, we do not add the edge.

Now a path that in G' that satisfies the requirements of part (a) corresponds to a *walk* in G. (Note that vertices and even edges can be repeated.) We can now use part (a) to find a path from an edge of the form (s,x) to an edge of the form (y,t). The last challenge is that there can be as many as $O(v)$ such edges, so the naive strategy of calling part (a) for all pairs of such edges is not efficient. Instead we modify the graph by adding special nodes t_3, t_7, t_4 and s_3, s_7, s_4 with the labels corresponding to their index. t_3, t_7, and t_4 are connected to each node of the form (y,t). s_4 is connected to all edges of the form (s,x) with label 3, and we add edges $s_3 s_7$ and $s_7 s_4$ to set up a "runway". Now we simply run the algorithm of part (a) 3 times to see if there is a path from s_3 to one of t_3, t_7 or t_4.

Below is a figure showing an original graph G and the modified graph G'.



Note that $|V'| = O(|E|)$ and $|E'| = O(|V|^3)$. (Each edge in E' between (u,v) and (v,w) corresponds to a triplet of vertices in V). The transformation is linear in G', and the execution of part (a) takes $O(|V'| + |E'|) = O(|E| + |V|^3) = O(|V|^3)$ ■

(c) Going back to vertex labels, design an algorithm to see if there is a path from s to t where the sequence of vertex labels matches a regular expression r (over the alphabet

$\{0, \ldots, 9\}$.

Solution: Since r is a regular expression, it can be represented as a DFA M. Let Q be a set of states of the DFA M. We can reconstruct a graph $G' = (V', E')$ such that

- for each vertex $u \in V$, $(u, q) \in V'$, where $q \in Q$,and $label((u, q)) = label(u)$
- for each edge $u \to v \in E$, $(u, q) \to (v, \delta(q, c)) \in E'$ where $c = label(v)$ and $q \in Q$.

Now we can just run DFS to find a path from (s, s') to (t, a) where s' is a starting state in DFA and $a \in A$. Since $|Q|$ is a constant, the runtime of the algorithm is $O(|V| + |E|)$.

Note that, as in the previous part, a path in G' could correspond to a *walk* in G since there are multiple vertices in G' that correspond to a vertex in G.

■

2. Solve problem 11 in Chapter 6 of the textbook (both parts)

(a) **Solution:** We can reduce this problem to a cycle detection problem in a directed graph G = (V, E) as follows:

- The vertices of G corresponds to variables in the parallel assignment. (eg. a, b, x, y).
- The edges of G corresponds to dependencies of the parameters. For each sub assignment in the parallel assignment, we construct edges pointing from the right side variables to the left side variable. For example, if a = b + c, we create two edges b->a and c->a.
- We do not need to associate additional values with the vertices or edges.
- We need to do cycle detection in this directed graph.
- We can solve this problem using Depth-First search.
- The algorithm runs in O(|V| + |E|) time.

The following algorithm IsAcyclic will return true if the directed graph has a cycle, meaning we need to use an additional temporary variables to serialize the given parallel assignment, or false if the directed graph does not have a cycle, meaning we need no additional temporary variables.

```
1: function IsAcyclic(G)
2:     for all vertices v do
3:         v.status = NEW
4:     for all vertices v do
5:         if v.status = NEW then
6:             if IsAcyclicDFS(v) == False then
7:                 return False
       return True
8:
9: function IsAcyclicDFS(v)
10:    v.status = ACTIVE
11:    for each edge v-> w do
12:        if w.status == ACTIVE then
13:            return False
14:        else if w.status == NEW then
15:            if IsAcyclicDFS(w) == False then
16:                return False
17:    v.status = FINISHED
18:    return True
```

■

(b) **Solution:** We can test if a given parallel assignment can be serialized with exactly one additional temporary variable by applying the following method. First, in order for the assignment to need at least one temporary variable, the graph must have a strongly connected component of size larger than one. Then, for all strongly connected components in the graph, remove each node and its corresponding edges in that strongly connected component, and test if the remaining graph is a DAG. If there exists a node in each SCC such that after removing the node, the remaining graph is a DAG, we can say the given parallel assignment can be serialized with exactly one temporary variable. We construct a directed graph G = (V, E) as follows:

- The vertices of G corresponds to variables in the parallel assignment. (eg. a, b, x, y).
- The edges of G corresponds to dependencies of the parameters. For each sub assignment in the parallel assignment, we construct edges pointing from the right side variables to the left side variable. For example, if a = b + c, we create two edges b->a and c->a.
- We do not need to associate additional values with the vertices or edges.
- We need to first computes strongly connected components in G using Kosaraju and Sharir's Algorithm in Jeff's note. Then, for each strongly connected component, we loop through all its node and test if the remaining graph is a DAG after removing that node and its corresponding edges.
- Kosaraju and Sharir's Algorithm and IsAcyclic run in O(|V|+|E|) time. Since we call IsAcyclic for each vertex, the whole algorithm runs in O(|V|*(|V|+|E|)) time.

Please refer to Jeff's note for the detailed implementation of Kosaraju and Sharir's Algorithm. For the following algorithm, we can assume we have already computed all

strongly connected components. If the algorithm returns true, then the given parallel assignment can be serialized with exactly one additional temporary variable, else no. G-v demote the graph obtained from G by deleting vertex v and its related edges. Function IsAcyclic is from problem 2a.

```
1:  function isOneVariable(G)
2:      atleastOne = false
3:      for each strongly connected component do
4:          if size of the SCC > 1 then
5:              atleastOne = true
6:          dag = false
7:          for each node in SCC do                                    ▷
8:              if IsAcyclic(G-v) then
9:                  dag = true
10:                 break
11:         if !dag then
12:             return false
13:     return true and atleastOne
```

■

3. Solve problem 15 in Chapter 6 of the textbook.

Solution (1): This is a solution using dynamic programming based on a 2D array.
In order to simplify the later steps and transformation to Dynamic Programming, assume the list X is sorted increasingly and $(X[i], Y[i])$ still represents the same set of points as the unsorted input.
Define a recursive function **LongestPath(X,Y,i)** which calculates the length of the longest monotonically increasing path in the given list of points, X, Y starting from the points $(X[i], Y[i])$. The function is recursively defined as follow

$$LongestPath(X, Y, i) = max\{LongestPath(X, Y, j) + Length(X[i], Y[i], X[j], Y[j]) \\ |for j \in [i, n], Y[j] > Y[i]\}$$

The idea behind this recursion is that the longest monotonically increasing path starting from a certain point $(X[i], Y[i])$ is the longest among all paths starting from points on the top right of it plus the path to them. The final result will be the maximum among all longest paths from every point.
Now we need to transfer this backtracking algorithm into a DP solution. We can observe that each recursive function call at $(X[i], Y[i])$ only depends the j that is greater than i, so we can use an 1D array to store the return value for the recursive **LongestPath(X,Y,i)**.

```
1: function LongestPathDP(X, Y)
2:     Initialize L[n] ← 0
3:     for i ← [n, 1] do
4:         for j ← [n, i] do
5:             if Y[j] > Y[i] then
6:                 L[i] = max(L[i], L[j] + Length(X[i], Y[i], X[j], Y[j]))
7:     return max(L)
8:
```

This DP algorithms have two nested loop each looping $O(n)$ times. If we assume **Length()** have a constant run time, then the run time for the overall algorithm is $O(n^2)$. The assumption we made about X being sorted can be accomplished within $O(n^2)$ time as well so it does not affect the overall run time complexity. ■

Solution (2): This is a solution using DAG.
We first convert the two arrays given X, Y into a DAG that contains information about the relative position among the points.
We build a Graph $G = (V, E)$, where each $v \in V$ represents a point given with identifier $v.i$ representing their place in the input arrays and two attributes $(v.X, v.Y)$ representing its coordinates. Add two extra vertices s, t with identifier 0 and $n+1$ and corresponding coordinates $(-\infty, -\infty)$ and (∞, ∞) as the start and end point for all paths.
For two vertices $u, v \in V$, there exists an edge $e = (u, v)$ if $u.X < v.X, u.Y < u$ and its weight is assigned with $Length(v.X, v.Y, u.X, u.Y)$. Let $w(u, v)$ denotes the weight of $e = (u, v) \in E$ and any edge from or to s or t has zero weight.
The graph is acyclic because each path is contains strictly increasing vertices identifier, i.e., coordinates in a 2D space.
Then the problem is transferred into a longest path problem in a DAG which can be solve using textbook 6.4. The longest path we are looking for starts from s and ends at t. We can define a recursive function LLP(v) which denotes the length of the longest path in F from v to t. It satisfies the following recurrence

$$LLP(v) = \begin{cases} 0 & if\ v = t \\ maxw(v, w) + LLP(w) | (v, w) \in E & else \end{cases}$$

The first function call will be $LLP(s)$.

Obviously, $LLP(v)$ only depends on the vertices behind v in the topological order, so we can transform the above recurrence into a dynamic programming solution.

```
1: function LongestPathGraphDP(V, E)
2:     initializa LLP[n] ← −∞
3:     V_sorted ← topological sorted V
4:     for v ∈ V_sorted backward do
5:         for (v, w) ∈ E, v, w ∈ V do
6:             LLP[v] = maxLLP[v], w(v, w) + LLP[w]
7:     return LLP[0]
8:
```

The algorithm runs in $O(|E| + |V|)$ time, where $|E| = O(n^2), |v| = O(n)$, so the overall run time is also $O(n^2)$. Two solutions are equivalent in terms of efficiency.

■

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