CSCE629 Analysis of Algorithms

Homework 4

Texas A&M U, Fall 2019 *09/20/19*

Lecturer: Fang Song *Due: 10am, 09/27/19*

Instructions.

- Typeset your submission by LaTeX, and submit in PDF format. Your solutions will be graded on *correctness* and *clarity*. You should only submit work that you believe to be correct, and you will get significantly more partial credit if you clearly identify the gap(s) in your solution. You may opt for the "Ill take 15%" option (details in Syllabus).
- You may collaborate with others on this problem set. However, you must **write up your own solutions** and **list your collaborators and any external sources** for each problem. Be ready to explain your solutions orally to a course staff if asked.
- For problems that require you to provide an algorithm, you must give a precise description of the algorithm, together with a proof of correctness and an analysis of its running time. You may use algorithms from class as subroutines. You may also use any facts that we proved in class or from the book.

This assignment contains 4 questions, 11 pages for the total of 65 points and 10 bonus points. A random subset of the problems will be graded.

1. (Component graph) Given a directed graph $G = (V, E)$, we define another graph $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$ called the *component graph* as follows. Suppose that G has strongly connected components $C_1, C_2, \ldots, C_k$. The vertex set V^{SCC} is $\{v_1, \ldots, v_k\}$ where $v_i \in C_i$. There is an edge $(v_i, v_j) \in E^{\text{SCC}}$ if G contains a directed edge $x \to y$ for some $x \in C_i$ and some $y \in C_j$. Alternatively, imagine contracting all edges whose incident vertices are within the same strongly connected component of G, and the resulting graph will be G^{SCC}.
   (a) (10 points) Prove or disapprove that G^{SCC} is a DAG.
   (b) (10 points (bonus)) Give an $O(|V| + |E|)$-time algorithm to compute the component graph of a directed graph $G = (V, E)$.

Solution:

(a) **TRUE** G^{SCC} is a DAG.

<u>**Proof:**</u> We can prove this by contradiction. Assume that G^{SCC} has a cycle. Let C and $C^{'}$ be distinct strongly connected components in a directed graph

$G = (V, E)$, let $u, v \in C$, let $u^{'}, v^{'} \in C^{'}$, and suppose that G contains a path $u \rightsquigarrow u^{'}$ and $v \rightsquigarrow v^{'}$. Then graph G contains paths $u \rightsquigarrow u^{'} \rightsquigarrow v^{'}$ (because $u^{'}, v^{'}$ are part of a strongly connected component and by definition every vertex is mutually reachable in it) and $v^{'} \rightsquigarrow v \rightsquigarrow u$ (similar to previous reasoning). Thus, u and $v^{'}$ are reachable from each other, thereby contradicting the assumption that C and $C^{'}$ are distinct strongly connected components. $\square$

Hence we conclude that,

- G^{SCC} is DAG.
- Let C and $C^{'}$ be distinct strongly connected components in a directed graph $G = (V, E)$, let $u, v \in C$, let $u^{'}, v^{'} \in C^{'}$ and suppose that G contains a path $u \rightsquigarrow u^{'}$. Then G cannot also contain a path $v^{'} \rightsquigarrow v$. This statement is direct implication of the above property. If this was false then we end up having a cycle in the G^{SCC} graph. Or one can say we C and $C^{'}$ are not two distinct and strongly connected components and they can be merged into one.

(b)

```
Algorithm 1 Strongly-Connected-Components(G)
1 - call $DFS(G)$ to compute the finishing times $u.f$ for each vertex $u$
2 - compute $G^T$
3 - call $DFS(G^T)$, but in the main loop of $DFS$, consider the vertices
in order of decreasing $u.f$ (as computed in line 1)
4 - output the vertices of each tree in the depth first forest formed
in line 3 as a separate strongly connected components.
```

Running time: is $O(V + E)$ because line 1-3 takes $\Theta(V + E)$ and line 4 takes $O(V + E)$.

Lets define a graph $G = (V, E)$ and its transpose will be $G^T = (V, E^T)$ where $E^T = (u, v) : (v, u) \in E$. That is, E^T consists of edges of G with their direction reversed. Given an adjacency list representation of G, the time to create G^T will be $O(V + E)$. Also note that G and G^T will have same strongly connected components. We will also extend the notion for discovery and finishing times to set of vertices. If $U \subseteq V$, then we define $d(U) = min_{u \in U}\{u.d\}$ and $f(U) = max_{u \in U}\{u.f\}$. That is $d(U)$ and $f(U)$ are the earliest discovery time and latest finishing time, respectively of any vertex in U. Reason behind by the algorithm works relies on following lemmas,

- **Lemma 1:** Let C and $C^{'}$ be distinct strongly connected components in directed graph $G = (V, E)$. Suppose there is a edge $(u, v) \in E$, where $u \in C$ and and $v \in C^{'}$. Then $f(C) > f(C^{'})$.

- **Lemma 2:** Let C and $C^{'}$ be distinct strongly connected components in directed graph $G = (V, E)$. Suppose there is a edge $(u, v) \in E^T$, where $u \in C$ and and $v \in C^{'}$. Then $f(C) < f(C^{'})$.

Lemma 2 tell us that each edge in G^T that goes between different strongly connected components goes from a component with an earlier finishing time (in the first depth-first search) to a component with a later finishing time.

From above, lemma 2 provides the key to understanding why the strongly connected components algorithm works. Let us examine what happens when we perform the second depth-first search, which is on G^T. We start with the strongly connected component C whose finishing time $f(C)$ is maximum. The search starts from some vertex $x \in C$, and it visits all vertices in C. By lemma 2, G^T contains no edges from C to any other strongly connected component, and so the search from x will not visit vertices in any other component. Thus, the tree rooted at x contains exactly the vertices of C. Having completed visiting all vertices in C, the search in line 3 (of the pseudo code) selects a vertex as a root from some other strongly connected component $C^{'}$ whose finishing time $f(C^{'})$ is maximum over all components other than C. Again, the search will visit all vertices in $C^{'}$, but by lemma 2, the only edges in G^T from $C^{'}$ to any other component must be to C, which we have already visited. In general, when the depth-first search of G^T in line 3 visits any strongly connected component, any edges out of that component must be to components that the search already visited. Each depth-first tree, therefore, will be exactly one strongly connected component.

To get the component graph we will execute the $Strongly - Connected - Components(G)$ as described above, and then assign each node a value in $[1, k]$, if we the generate k strongly connected components. Then Traversing each node i, for each node j of $Adj[i]$, if $k[i]$ and $k[j]$ have no edges before, then we will add an edge. This process will take constant time hence entire process will take $O(V + E)$ runtime.

2. (10 points) (Singly connected graph) A directed graph $G = (V, E)$ is *singly* connected if G contains at most one simple (i.e. no vertex repeated) path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether or not a directed graph is singly connected.

Solution: To solve this problem we need to notice that if we DFS on a directed graph then graph will be singly connected if and only if all edges are tree edges or back edges. We will fix a vertex u and do DFS on $G = (V, E)$ starting from

u will generate a tree T rooted at x, each vertex of the tree is reachable from u in the graph G. Other vertices that are not in the tree are not reachable from u in the graph G. An edge $(u, v) \in E \setminus E(T)$, $E(T)$ are edges of the tree, where $\{u, v\} \subseteq V(T)$, $V(T)$ are vertices of the tree, belongs to one of the following cases,

- A back edge (which is not a problem)
- A forward edge $\implies$ Graph is not singly connected (since there are at least two paths from u to v)
- A cross edge $\implies$ Graph is not singly connected (since there are at least two paths from u to v).

This idea gives us an algorithm to do a DFS on every every vertex and check if for any cross edges and forward edges. This algorithm will take $O(V(V+E))$, since generally we have more edges than vertices running time will become $O(EV)$.

Prof. Samir Khuller in 1999 proposed an improvement in algorithm. Based on the following theorem.
Theorem: Let H be a strongly connected graph. H is not singly connected if and only if at least one of the following conditions holds. The DFS search in H either yields a cross edge, or a forward edge, or a vertex v such that from the subtree rooted at v, there are at least two back edges to proper ancestors of v.

Based on this theorem we can generate an algorithm. For given directed graph G create a component graph G^{SCC} then, if we pick any two vertices such that it is in one of the strongly connected components then we can design an algorithm with run time $O(E)$ such that it checks for all the conditions. If the two vertices lie in two distinct strongly connected components then we can do DFS on the component graph from each vertex and this take $O(V)$ for each vertex and for all vertices it will take at most $O(V^2)$. Hence total runtime of the this algorithm will be $O((V+E)+(E)+(V^2))$, because linear time for generating the component graph plus for checking the conditions on the strongly connected components plus for vertices in two unique strongly connected components. Hence total runtime will be dominated by the $O(V^2)$ term.

3. (Semi-connected graphs) A directed graph G is *semi-connected* if, for every pair of vertices u and v, either u is reachable from v or v is reachable from u (or both).
   (a) (5 points) Give an example of a DAG with a unique source (a source is a vertex with no entering edges) that is **not** semi-connected.
   (b) (10 points) Describe and analyze an algorithm to determine whether a given DAG is semi-connected.
   (c) (10 points) Describe and analyze an algorithm to determine whether an arbitrary directed graph is semi-connected.

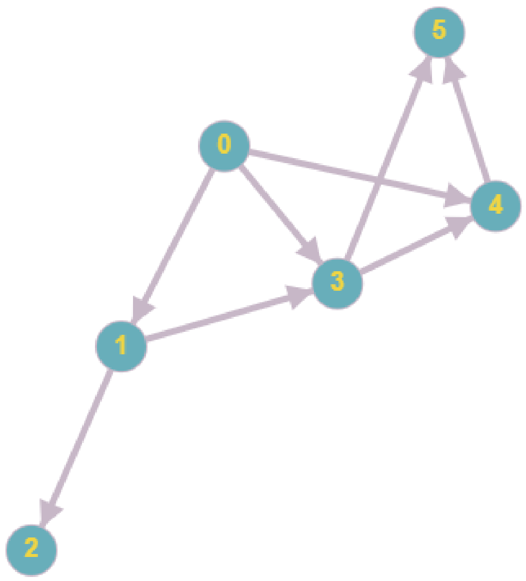

Figure 1: Example of a DAG that is not semi-connected graph

Solution:

(a) We can see a example for DAG that is not semi-connected in **Fig 1**. In that figure we have one unique source vertex which is vertex 0 (it has no entering edges) and we have sinks (no outgoing edges) vertex 2 and vertex 5. We notice that there is no connection between vertices 2 and 3, 0 and 5. But according to the definition of semi-connected graph, each pair of vertices must have path between them but here we have vertex pairs which don't have edge joining them. Hence this graph is not semi-connected.

(b) **Definition:** a ***Hamiltonian path*** (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once. A ***Hamiltonian cycle*** (or Hamiltonian circuit) is a Hamiltonian path that is a cycle. Determining whether such paths and cycles exist in graphs is the Hamiltonian path problem, which is NP-complete.

Now we will define a necessary and sufficient condition for DAG to be semi-connected.

Theorem: A DAG is semi-connected if and only if the topological sort of the DAG has a single path that goes through all the vertices i.e., there exists a Hamiltonian path. This is equivalent to saying that after we do topological sort on DAG and arrange them in linear order then there must exit an edge between all the consecutive pairs.

Proof: Let there be k vertices of the DAG G arranged in the topological order $v_1, v_2, \ldots, v_k$.

$\Longrightarrow$: Given the DAG arranged in topological order is semi-connected then for any i, there cannot be a path from v_{i+1} to v_i because we have assumed that the vertices are arranged in topological order. The only possible path from v_i and v_{i+1} is a direct edge between them and this edge must be present between them for DAG to be semi-connected. Since there are edges between all consecutive pairs of vertices in the topological order, there is a path through all vertices. Hence there must be a Hamiltonian path.

$\Longleftarrow$: Given DAG arranged in topological order has a Hamiltonian path then there is a path that goes through all vertices. That means that there is path between any pair of vertices v_i, v_j by simply following the relevant part of this global path. Hence the DAG must be semi-connected.

Observation: If a DAG is semi-connected then the graph must have one source in topological sort. If it has two source then we have two vertices with no path between which implies the graph is not semi-connected.

Let's try to understand by theorem using an example, assume we have a DAG with 5 vertices arranged in topological order. If this DAG is semi-connected then for each pair of vertex v_i, v_j there must be edge. But since they have been arranged in topological order and there can be edges from left to right only. Which means that there can be edge from v_i to v_j where $i < j$ but there can't be an edge from v_j to v_i. If the graph is semi-connected then v_1 must have edges going to $v_2, \ldots, v_5$ (to satisfy the property of semi-connectedness), v_2 must have edges $v_3, \ldots, v_5$ (to satisfy the property of semi-connectedness) and so on. This leads to existence of edge between all consecutive vertices. If we can find such kind of a path then we have path which goes from all vertices, which is nothing but a Hamiltonian path.

Above theorem gives an skeleton for an algorithm. First we will do topological sorting on DAG and then we will check if there are edges between each consecutive pairs. If yes, then DAG is semi-connected, if no, then DAG is not semi-connected.

Algorithm 2 *Semi-connected(G)*
```
Do topological sort on graph $G$
if there exist a edge between all consecutive vertices $v_i, v_j$
  return Graph is semi-connected
return Graph is not semi-connected
```

Time complexity of this algorithm is $O(V + E)$ because time taken for topological sorting on the DAG is linear time and checking the existence of an edge between all the pairs can be done in constant time.

(c) If we are given an arbitrary directed graph, then to check if the given graph is semi-connected we can convert the original graph into a component graph. We are doing this because as proved in problem 1, a component graph of a directed graph is a DAG so, we can apply above theorem (and algorithm) on the component graph to check the semi-connectedness of the given directed graph.

Intuition behind why this works is, if the component graph is semi-connected then it implies that there exist a path between all distinct strongly connected component. So there will be path for any vertex from one strongly connected component to another (from the definition of semi-connectedness there is no necessity on existence of path in backwards direction; we just need a path in one direction) and vertices within a strongly connected component are mutually reachable hence the original graph is semi-connected.

Algorithm 3 *Semi-connected(G)*

```
Compute the component graph of $G$, call it $G^{\mathrm{SCC}}$
Perform topological sort on $G^{\mathrm{SCC}}$ to get the ordering of its vertices
$v_1, v_2, \ldots, v_k$
for $i = 1, \ldots, k-1$
  if there is no edge from $v_i$ to $v_{i+1}$
     return G is not semi-connected
return G is semi-connected
```

Time complexity of this algorithm will be of order $O(V+E)$ because to compute topological ordering and component graph both take linear time in the order of $O(V+E)$.

4. An *Euler tour* of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each *edge* of G exactly once, although it may visit a vertex more than once.

   (a) (10 points) Show that G has an Euler tour if and only if in-degree(v) = out-degree(v) for each vertex $v \in V$.

   (b) (10 points) Describe an $O(|E|)$-time algorithm to find an Euler tour of G if one exists.

Solution:

(a) **<u>Definition:</u>** A ***path*** or ***walk*** in a graph is a sequence of adjacent edges, such that consecutive edges meet at shared vertices. A path that begins and ends on the same vertex is called a ***cycle*** or ***circuit***. Hence a path in a graph is a sequence of vertices such that every vertex in the sequence is adjacent to vertices before and after in the sequence.

Note: A cycle is a simple cycle in which the only repeated vertices are the first and last vertices.

Definition: A graph is said to be ***connected*** if any two of its vertices are joined by a path. A graph that is not connected is a ***disconnected*** graph. A disconnected graph is made up of connected subgraphs that are called components.

Definition: An ***Euler path*** in a graph G is a path that includes *every* edge in G; an ***Euler cycle*** is a cycle that includes every edge. Euler cycle is also know with different names like *Euler tour* and *Euler circuit*. Here the starting and ending vertices are the same, so if we trace along every edge exactly once and end up where we started, it becomes a Euler cycle.

Note: If a graph is not connected, there is no hope of finding such a path or circuit.

Definition: In graph theory, the ***degree*** or ***valency*** of a vertex of a graph is the number of edges that are incident to the vertex, and in a multigraph, loops are counted twice. The degree of a vertex v is denoted $deg(v)$. For a directed graph degree of vertex will be two kinds. First one in-degree i.e., the total no. of the incoming edges and second one out-degree i.e., the total no. of the outgoing edges.

Handshaking lemma: The degree sum formula states that, given a graph $G = (E, V)$ (for a graph with vertex set V and edge set E),

$$\sum_{v \in V} deg(v) = 2|E|.$$

Note: Since $|E|$ has to be an integer that implies any graph G always has even number of vertices with odd degree.

Theorem: A connected undirected graph $G = (E, V)$ has Euler cycle if and only if for all vertex $v \in V$ has even degree. For directed graph this theorem will be modified as follows, a connected directed graph $G = (E, V)$ has Euler cycle if and only if for all vertex $v \in V$ has in-degree(v) = out-degree(v).

This theorem, with its "if and only if" clause, makes two statements. One statement is that if every vertex of a connected graph has an even degree then it contains an Euler cycle. It also makes the statement that only such graphs can have an Euler cycle. In other words, if some vertices have odd degree, the the graph cannot have an Euler cycle. Notice that this statement is about Euler cycles and not Euler paths.

Proof: $\implies$: If G is Eulerian then there is an Euler Cycle, P, in G. Every time a vertex is listed, that accounts for two edges adjacent to that vertex, the one before it in the list and the one after it in the list. This circuit uses every edge exactly once. So every edge is accounted for and there are no repetition of edges while counting. But vertices can be repeated, so if we count the degree for each vertex then, always we will have an incoming edge and outgoing edge, and for points in the extreme there are joined by an edge (since there are no repetition of edges there will be no duplicates that we need worry while computing the degree of a vertex). Thus every degree must be even for undirected graph and for directed graph we will have in-degree(v) = out-degree(v).

This can be thought as P is either simple cycle (does not intersect itself), or not. If P is a simple cycle, each vertex in a simple cycle has in-degree(v) = out-degree(v), so the claim is true. If P is a cycle but not a simple cycle, then it must contain a simple cycle; remove it from G and from P; the remaining P is still an Euler cycle for the remaining G. Repeat removing (simple) cycles until no edges left. When removing a cycle, an in-edge and out-edge of the vertices on the cycle are removed. After a cycle deletion, the in-degree and out-degree of a node on the cycle decrease by exactly 1. At the end, when no edges are left, all in-degrees and out-degrees are 0. So all vertices v must have started with in-degree(v) = out-degree(v).

Claim: For a graph G if it has in-degree(v) = out-degree(v) (or even degree) for every vertex, then for any vertex v there must be a path starting from v that comes back to v.
Proof for any vertex v, there must be a cycle that contains v. Start from v, and chose any outgoing edge of v, say (v, u). Since in-degree(v) = out-degree(v) we can pick some outgoing edge of u and continue visiting edges. Each time we pick an edge, we can remove it from further consideration. At each vertex other than v, at the time we visit an entering edge, there must be an outgoing edge left unvisited, since in-degree(v) = out-degree(v) for all vertices. The only vertex for which there may not be an unvisited entering edge is v because we started the cycle by visiting one of v's outgoing edges. Since there's always a leaving edge we can visit for any vertex other than v, eventually the cycle must return to v, thus proving the claim. In conclusion, we know that we will return to v eventually because every time we encounter a vertex other than v we are listing one edge adjacent to it. There are an even number of edges adjacent to every vertex, so there will always be a suitable unused edge to list next. So this process will always lead us back to v. $\square$

$\impliedby$: Suppose every degree is even, then we have to prove that the graph has Euler Cycle. We will show that there is an Euler cycle by induction on the number of edges in the graph. The base case is for a graph G with two vertices with two edges between them. This graph is obviously Eulerian.

Now suppose we have a graph G on $m > 2$ edges. We assume for total no. of edges less than m, we have Euler cycle. We start at an arbitrary vertex v and follow edges, arbitrarily selecting one after another until we return to v. Call this trail W. Let E_W be edges of W. We make new graph $G^{'} = (V, E \setminus E_W)$. Assume, new graph $G^{'}$ has components $C_1, C_2, \ldots, C_k$. These components satisfy the induction hypothesis i.e., they are connected, have edges less than m, every vertex has in-degree(v) = out-degree(v) (even degree for undirected graph). We know that every vertex has in-degree(v) = out-degree(v) in the new graph $G^{'}$ because when we removed W, we removed an even number of edges from the vertices listed in that cycle. By induction each component has an Euler cycle, lets call them $P_1, P_2, \ldots, P_k$. By this process we get k edge disjoint cycles with at least one common vertex between each of them. Since G is connected which means there is a vertex a_i in each component C_i, which is present in both W and P_i. With loss of generality, assume that as we follow W, the vertices $a_1, a_2, \ldots, a_k$ are encountered in that order. We describe an Euler cycle G by starting at v follow W until reaching a_1, follow the entire P_1 ending back at a_1, follow W until reaching a_2, follow the entire P_2 ending back at a_2 and so on. End by following W, until reaching a_k follow the entire P_k ending back at a_k, then finish off W, ending at v. $\square$

Corollary: A connected undirected graph contains an Euler Path (not Euler Cycle) if and only if it contains only two vertices with odd degrees. For a connected directed graph, it will contain an Euler path if and only if it contains only two vertices with $|deg_{in}(v) - deg_{out}(v)| = 1$.

(b) As mentioned in the proof we will use that to construct an algorithm which merges the edge disjoint cycles. This is known as Hierholzer's algorithm.

- Choose any starting vertex v, and follow a trail of edges from that vertex until returning to v. It is not possible to get stuck at any vertex other than v, because the even degree of all vertices ensures that, when the trail enters another vertex w there must be an unused edge leaving w. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph.
- As long as there exists a vertex u that belongs to the current tour but that has adjacent edges not part of the tour, start another trail from u, following unused edges until returning to u, and join the tour formed in this way to the previous tour.

Also remember we need to check if for all vertices we have in-degree(v) = out-degree(v). If this condition is violated then the graph has no Euler Cycles.

If the condition is true then, pick a vertex v and perform DFS from it until finding a back edge that links back to v. Once you find this cycle, traverse all edges of the cycle, and delete the corresponding edge in the adjacency list of G; to delete edges of G quickly we assume that we have modified DFS so that for each edge that we traverse we store a pointer to the corresponding edge in the adjacency list of G. With this information, deletion of an edge can be done in constant time, basically because we dont need to search for the edge in G. Then we repeat the process. Overall this takes $O(|E|)$ time.