CS2800 Design and Analysis of Algorithms
Problem Set #4 February 13, 2023

1. Give a linear-time algorithm to find an odd cycle in a digraph.

Solution: A digraph has a cycle iff there is a backedge. While doing a DFS, add an additional parity information for each vertex. For the starting vertex s, the parity is set to 0. If DFS(u) calls DFS(v) directly, then parity of v will be the opposite of parity of u. If there is a back-edge from a vertex v with parity b to a vertex u with parity $1-b$, then that is an odd-length cycle. The algorithm runs in $O(|V|+|E|)$-time.

2. Fake News Now wants to spread a rumor among the people of Gullible Town which has n inhabitants. If a member of Gullible Town hears a rumor, he/she will spread it to all his/her friends. By snooping around on social media, Fake News Now knows the friends of each of the inhabitants of Gullible Town. Since you are in the R& D wing of Fake News Now, you are tasked with finding the smallest number of people to whom you should say the rumor so that finally at some point, everyone in Gullible Town will know it. Design and analyze an algorithm to perform this task.

Solution: Construct a graph G where the vertices correspond to the people in the town. There is an edge between u and v if they are friends (assume that friendship is a symmetric relation). Now, if you spread the rumour to one person in a connected component, everyone will know it eventually. So the minimum number of people to be informed is the number of connected components.

It takes $O(|V|+|E|)$ to construct the graph, and another $O(|V|+|E|)$ to count the number of connected components, which is the answer.

3. Given an undirected graph G, design an algorithm to check if it is possible to orient the edges of the graph such that the constructed digraph becomes strongly connected.

Hint: You might want to recollect the concept of bridges that you learnt in your graph theory course.

Solution: An orientation of an undirected graph that makes the digraph strongly connected is called a strong orientation.

Consider the following algorithm: Perform the DFS on G. Every tree edge is oriented from the parent to the child. Every back edge is oriented from the descendant to an ancestor. Since the graph is undirected, there are only tree edges and back edges.

Once, we get this orientation, we run the Kosaraju-Sharir algorithm to find SCC(G). If there is only one strongly connected component, we say that the graph has a strong orientation, otherwise we say that it has none.

Let us prove the following lemma to prove the correctness of the algorithm. The proof will contain the required details for the correctness of the algorithm.

Lemma. *If a graph G has no bridges, then G has a strong orientation*

Proof. We start with the observation that if G has no bridges, then every edge (u, v) must be part of a cycle. If not, we can delete (u, v) and there will be no path from u to v. This would mean (u, v) is a bridge.

Let C be any cycle in the graph. Orient all the edges in C in a cycle. Orient any chord within the cycle arbitrarily. If there is some vertex not covered by the cycle, find the edge (u, v) where $u \in C$ and v is outside. Now, there must exists a cycle C' containing (u, v). Orient this cycle. Continue until all the vertices are covered by some cycle.

At any point of this process, the subgraph that is oriented is strongly connected since we are orienting cycles. We can do this till all vertices are covered since every edge is part of a cycle and we will get a strong orientation. □

4. Suppose that we are given a DAG G with a unique source s and a unique sink t. A vertex $v \notin \{s, t\}$ is said to be a *cut vertex* if every path from s to t in G goes via v.

   (a) Design an $O(|V| + |E|)$-time algorithm to find all cut vertices in G.

   For this, start by doing a topological sort of G. Now, show that a vertex v is a cut vertex iff there is no edge $(u, w) \in G$ such that u occurs before v in the topological order and w occurs after v in the topological order.

   First, use the observation above (without proof) to design you algorithm. Next, prove the statement above.

   Solution: First let's state and prove the following.
   Lemma. *A vertex v is a cut vertex iff there is no edge $(u, w) \in G$ such that u occurs before v in the topological order and w occurs after v in the topological order.*

   Proof. First, let us observe that if G has a unique source s, and unique sink t, then every u is reachable from s, and t is reachable from u. You can prove this by taking the topological order and inductively proving it for vertices in that order.

   Now, let v be a cut vertex and suppose that $u \prec v \prec w$ in the topological ordering. If there is an edge (u, w), then we have a path from s to u to w and then to t contradicting that v was a cut vertex.

   If v was not a cut vertex, then there is a path from s to t in $G - \{v\}$ and this must include an edge from a vertex $u \prec v$ to a vertex $w \succ v$ in the topological order. □

Now, the algorithm will first topologically order the DAG, and will do the following in the topological order: take a vertex v, find the vertex u with the smallest end-time (farthest in the topological order) in its adjacency list and remove all the vertices from v to u from the list of potential cut vertices. The list of potential cut vertices can be an array ordered based on the topological ordering. Keep doing this until the only vertices that remain are the cut vertices.

(b) Does you algorithm work if there are multiple sources or sinks? Which part of the argument will break? Can you construct a counter-example illustrating this?

Solution: Finding cut vertices between s and t when there are more than one source and sink will not be possible by the above algorithm. You can construct a counter-example to the lemma stated in the previous part when there are more than one source or sink. Try to find the counter-example.

5. The 2-SAT problem is defined as follows. Consider a propositional formula ϕ in 2-CNF defined as follows:

$$\phi = (l_{1,1} \vee l_{2,2}) \wedge (l_{2,1} \vee l_{2,2}) \wedge \cdots \wedge (l_{m,1} \vee l_{m,2}),$$

where each $l_{i,j}$ is either a boolean variable x_k or its negation $\overline{x}_k$. You want to check if there is a truth assignment to the variables $x_1, x_2, \ldots, x_n$ such that the propositional formula ϕ is satisfiable. For example, the formula $\phi = (x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (x_2 \vee x_3)$ is satisfiable with the truth assignment $x_1 = 1, x_2 = 1$, and $x_3 = 0$.

Let us design a linear-time algorithm for checking the satisfiability as follows. First define a graph G_ϕ on $2n$ vertices as follows: for each i, there is a vertex corresponding to x_i and one of $\overline{x}_i$. Now, for each clause $(l_i \vee l_j)$, add a directed edge from $\overline{l}_i$ to l_j and one from $\overline{l}_j$ to j_i.

(a) Show that if G_ϕ has a strongly connected component containing x_i and $\overline{x}_i$ for some i, then ϕ is not satisfiable.

Hint: Recall that $(x \vee y)$ is logically equivalent to $\overline{x} \Rightarrow y$, and to $\overline{y} \Rightarrow x$

Solution: Let $L = \{l_1, l_2, \ldots, l_k\}$ be the literals that appear in a strongly connected component. Since there is an edge (l_i, l_j) when $l_i \Rightarrow l_j$, we can say that in any satisfying assignment all the literals $l_1, l_2, \ldots, l_k$ should receive the same truth value. Thus, if x_i and $\overline{x}_i$ appear in a strongly connected component, then every satisfying assignment must set both x_i and $\overline{x}_i$ to the same truth value, which is impossible.

(b) Now, show that if none of the strongly connected components contains a variable and its negation, then the formula ϕ is satisfiable.

For this, start by considering a sink component in $\mathrm{SCC}(G_\phi)$, and assign all the literals in it to be true, and all their negations to be false. Show that you can extend this idea suitably to get a satisfying assignment for ϕ.

Solution: Let C be a sink component of SCC(G_ϕ). First observe that if l_i and l_j are in C, then $\bar{l}_i$ and $\bar{l}_j$ must also lie in a same(but different from C) strongly connected component. This is because if l_i, l_j in C, then $l_i \Leftrightarrow l_j$ and therefore $\bar{l}_i \Leftrightarrow \bar{l}_j$.

Let $\overline{C}$ denote the strongly connected component of G_ϕ containing the complements of the literals in C. We can show that if C is a sink component in SCC(G_ϕ) iff $\overline{C}$ is a source component of SCC(G_ϕ). To see one direction, if C is a sink, but $\overline{C}$ is not a source then there is some other component C' such that for every literal $l' \in C'$ and $\bar{l} \in C$, we have $l' \Rightarrow \bar{l}$. But, this would mean $l \Rightarrow \overline{l'}$ contradicting the fact that C was a sink.

Thus, we can set all the literals in C to 1 and C' to 0 since any clause $l \Rightarrow l'$ is satisfied if $l = 0$ (irrespective of the value of l') and by $l' = 1$ (irrespective of the value of l).

(c) Verify that this idea gives a linear-time algorithm to check the satisfiability (and find a satisfying assignment, if one exists) for a 2-CNF formula ϕ.

Solution: The algorithm is clear from Parts (a) and (b).

6. The *transitive closure* of a directed graph G is a graph G^* such that $(u, v) \in G^*$ iff there is a path from u to v in G. In this question, we will see how to find the transitive closure using boolean matrix multiplication. Given two boolean matrices A and B, the boolean matrix product C is defined as: $C[i, j] = \bigvee_k (A[i, k] \wedge B[k, j])$. It is known that there is an algorithm to perform boolean matrix multiplication in time $O(n^\omega)$ where $2 < \omega \leq 2.37$.

(a) Let A be the adjacency matrix of G. Show that the adjacency matrix of the G^* is $(A + I)^n$ where you perform boolean matrix multiplication. (Use induction)

Solution: From the definition of multiplication $(A + I)^2[i, j]$ is 1 precisely when there is a path of length at most 2 in G. This is because $(A + I)^2[i, j]$ is 1 iff there exists a k (maybe equal to i or j) such that $(A + I)[i, k] = 1$ and $(A + I)[k, j] = 1$ which means there is an edge from i to k and from k to j. Use standard induction now.

(b) Use the part above to design an $O(n^\omega \log n)$-time algorithm to compute G^*.

Solution: Keep squaring $A + I$ until you reach $(A + I)^n$. The total time will be $\Theta(n^\omega \log n)$.

(c) Suppose that G is a *directed acyclic graph*. Let $v_1, v_2, \ldots, v_n$ be a topological ordering of the vertices of G. Let G_1 be the graph on the vertices $v_1, \ldots, v_{n/2}$ and G_2 be the graph on the vertices $v_{n/2}, \ldots, v_n$, and let G_3 be the graph $G \setminus \{G_1 \cup G_2\}$. Let A_1 and A_2 be the $n/2 \times n/2$-adjacency matrices of G_1 and G_2, and let A_3 be the $n/2 \times n/2$ bipartite adjacency matrix of G_3 defined as follows: $A_3[i, j] = 1$ (where $i \leq n/2$ and $j > n/2$) iff $(i, j) \in G$.

Show that the adjacency matrix of G^* is given by $\begin{pmatrix} (A_1+I)^{n/2} & (A_1+I)^{n/2}A_3(A_2+I)^{n/2} \\ 0 & (A_2+I)^{n/2} \end{pmatrix}$.

Solution: Every path in the DAG either

1. lies completely inside G_1 and is of length at most $n/2$, or
2. lies completely inside G_2 and is of length at most $n/2$, or
3. has a part of length at most $n/2$ in G_1, then crosses to G_2 via an edge and has a part of length at most $n/2$ in G_2.

The first path is covered by $(A_1 + I)^{n/2}$, the second by $(A_2 + I)^{n/2}$ and the third by $(A_1 + I)^{n/2}A_3(A_2 + I)^{n/2}$.

(d) Use the part above in a divide-and-conquer algorithm to obtain an $O(n^\omega)$-time algorithm to compute the transitive closure of directed acyclic graphs.

Solution: The part above gives the divide and conquer algorithm where $(A_1 + I)^{n/2}$ and $(A_2 + I)^{n/2}$ are computer recursively. Then $(A_1 + I)^{n/2}A_3(A_2 + I)^{n/2}$ is computed in $O(n^\omega)$ time. The running time of the algorithm is given by

$$T(n) = 2T(n/2) + \Theta(n^\omega).$$

(e) Show that if G is strongly connected, then G^* is the graph such that for every $u, v \in V$, (u, v) and (v, u) are edges.

Solution: This is easy to verify.

(f) Use parts above to design an $O(n^\omega)$-time algorithm to compute the transitive closure of a directed graph G.

Solution: First compute $\text{SCC}(G)$ and perform Part (c) to obtain the transitive closure of $\text{SCC}(G)$. Then use Part (e) with it to obtain the transitive closure of the whole graph.