ICCS313 - Algorithm Analysis, T1 2019/20

Lecture 5: Graph Algorithms (2)

12 October 2019

Lecturer: Chaya Hiruncharoenvate *Scribe: -*

```
** Any questions from Quiz 1?
** Issues with Quiz 1. Example solutions to Problem 1 & 2.
** Review the proof of bipartite checker from last lecture.
```

1 Directed Graph

A directed graph is commonly used to model directional relationships. For example, a web graph represents a network of web pages. An edge $u \to v$ represents a page u contains link to another page v.

Definition 1.1. A directed graph $G = (V, E)$ consists of a nonempty set of nodes V and a set of directed edges E. Each edge $e \in E$ is specified by an ordered pair of vertices $u, v \in V$. A directed graph is *simple* if it has no loops (that is, edges of the form $u \to u$) and no multiple edges.

2 BFS and DFS revisited

Given a starting node s, what is the running time of BFS and DFS algorithms on a *directed* graph?
BFS: $O(m + n)$
DFS: $O(m + n)$

3 Reachability Problem in Directed Graph

Problem: Given a directed graph G and a node s, find all nodes reachable from s.

Recall that we discussed the reachability problem in undirected graph last time. Does the algorithm for undirected graphs work when G is directed? Yes, the BFS based algorithm still works.

An important application is web page crawling to identify a set of pages reachable from a starting page s.

4 Strong Connectivity

Another interesting thing about directed graphs is that, when v is reachable from u, it is not necessary that u is reachable from v.

[Draw an example]

Definition 4.1. Nodes u and v are *mutually reachable* if there is both a path from u to v and also a path from v to u.

Definition 4.2. A graph is *strongly connected* if every pair of nodes is *mutually reachable*.

Lemma 4.3. *Let s be any node. G is strongly connected if and only if every node is reachable from s, and s is reachable from every node.*

Proof:

($\Rightarrow$) Assume G is strongly connected. By definition, every node is reachable from s and s is reachable from every node.

($\Leftarrow$) Given s, we assume that every node is reachable from s and s is reachable from every node. Consider any pair of nodes (u, v). There is a path from u to v because we have u $\rightarrow$ s, and s $\rightarrow$ v. Similarly, we have a path from v to u. Thus, G is strongly connected. Note it doesn't matter if the path overlaps. ■

Now let's consider a problem where a directed graph G is given and we want to check whether G is strongly connected.

`IsStronglyConnected(G):`

1. Pick ANY node s in graph G
2. Run BFS from s in the graph G
3. Run BFS from s in the graph $G^{reversed}$
4. Return True iff all nodes reached in both BFS

Question: What's the running time? $O(m + n)$

The correctness of `IsStronglyConnected` follows directly from the previous lemma.

Another interesting fact of a directed graph G is that G can be always decomposed into one or more strongly connected components.

[Draw a graph with multiple strongly connected components]

Definition 4.4. A strong component, or strongly connected component, is a maximal subset of mutually reachable nodes

Identifying all strong components in a directed graph can be done in $O(m+n)$. The algorithm was proposed by Robert Tarjan in 1972.

5 Directed Acyclic Graph

A *DAG*, directed acyclic graph, is a directed graph that contains no directed cycle.

Basically, DAG is a special case of directed graphs that has a wide range of applications e.g. course prerequisite graphs, task graphs, etc.

[Draw a DAG and a non-DAG]

A common question about DAG is to find what we call a *topological order*.

Definition 5.1. A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ so that for every edge (v_i, v_j) we have $i < j$

Here's an interesting fact about a topological order and DAG.

Lemma 5.2. *If G has a topological order, then G is a DAG.*

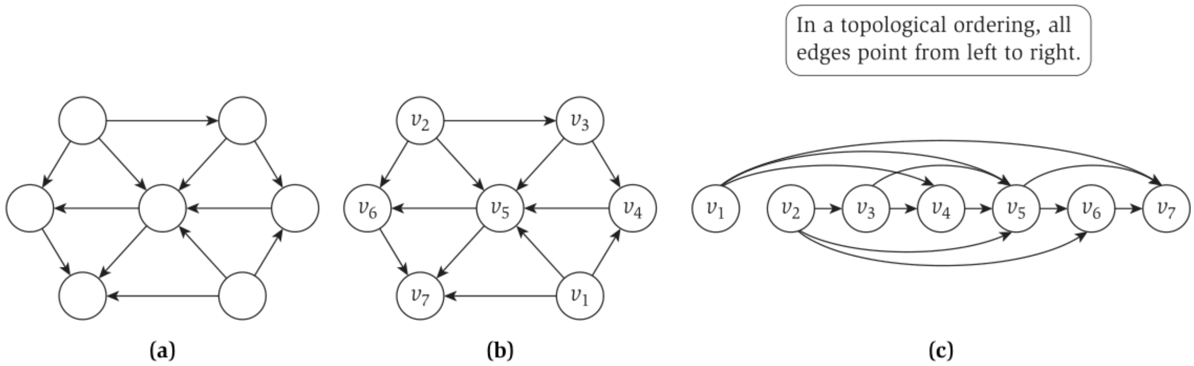

Figure 3.7 (a) A directed acyclic graph. (b) The same DAG with a topological ordering, specified by the labels on each node. (c) A different drawing of the same DAG, arranged so as to emphasize the topological ordering.

Figure 1: Reproduced from KT's Algorithm Design

Proof: Let's prove this by contradiction. Let's assume for the sake of contradiction that G has a directed cycle, call it C. Suppose G has a topological order $v_1, v_2, \ldots, v_n$. Let v_i be the lowest-indexed node in C and v_j be any node before v_i. That means (v_j, v_i) is an edge. By our choice of v_i, $i < j$. On the other hand, since (v_j, v_i) is an edge and $v_1, v_2, \ldots, v_n$ is a topological order, it should be that $j < i$. Hence, a contradiction. ■

Question: Does every DAG have a topological order? If so, how do we find one?

Lemma 5.3. *If G is a DAG, then G has a node with no entering edges.*

Proof: Again, let's prove this by contradiction. Suppose G is a DAG and every node has at least one entering edges. Pick a node v and follow edges backwards from v. We can do this since v has at least one entering edge (u, v). So we can walk backward to u. Again, we can repeat the same argument about u and follow the edge (x, u) back to x. We keep doing this until we visit the same node twice, say w. We know that this will happen because we only have n nodes in total. Let C be the sequence of nodes from $w \to w$. We have that C is a cycle. Thus, G is not a DAG. Hence, a contradiction. ■

Lemma 5.4. *If G is a DAG, then G has a topological order.*

Proof: Let's do this by induction. For the base case, it's trivially true when $n = 1$. Given a DAG with $n > 1$ nodes. First, we find a node v without any entering edges. We know that v must exist from the previous lemma. Now consider $G - \{v\}$. We know that $G - \{v\}$ is still a DAG, since removing a node cannot create cycles. By induction hypothesis, $G - \{v\}$ has a topological order, say $v_1, v_2, \ldots, v_{n-1}$. So, since v has no entering edges, we have that $v, v_1, v_2, \ldots, v_{n-1}$ is a valid topological order of G. ■

`TopologicalSort`(G):

1. Find a node v with no entering edges
2. A = `TopologicalSort`($G - \{v\}$)
3. Return [v] + A

Correctness: Follows directly from the lemma.

Running time: Maintain $count(w)$ the number of entering edges and S a set of node with no incoming edges. Initialization: $O(m + n)$ for scanning the graph. To delete v, we remove v from S, decrease $count(w)$ for all (v, w), add w to S if $count(w)$ becomes 0. An edge is removed only once so $O(m + n)$ total time.