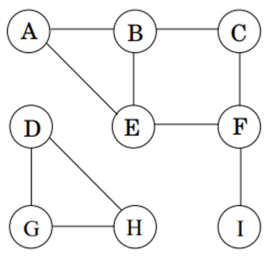
CSCI 2300 — Introduction to Algorithms
Homework 2 (document version 1.0) — DUE February 7, 2020
Graph Algorithms

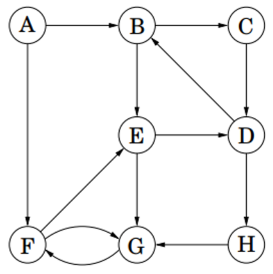
- This homework is due by 11:59PM on the due date above and must be submitted in Submitty as a single PDF called `upload.pdf`.
- Just like the labs, please avoid using Google to find suggestions or solutions. You are allowed to use the textbooks and class notes as references. We strongly encourage you to use the office hours, labs/recitations, and Discussion Forum to successfully complete this assignment.
- Also, you are allowed to consult with your classmates, collaborating in line with the policies set forth in the syllabus. You must write up your own separate solution; do not copy-and-paste anything.
- Remember the goal is to use your own brain to work these problems out, which will help you develop the skills to do well on the exams and, more importantly, become a substantially better computer scientist.
- For your solutions and proofs, please write clearly and concisely, and use rigorous formal arguments and pseudocode that mimics the style used in the Dasgupta textbook. Be consistent.
- In general, if you are asked to provide an algorithm, the best thing to do is to provide a clear description in English (e.g., "Use BFS, but with the following change..." or "Run BFS two times as follows...").
- Be as concise as you can in your answers. Long answers are difficult to grade.

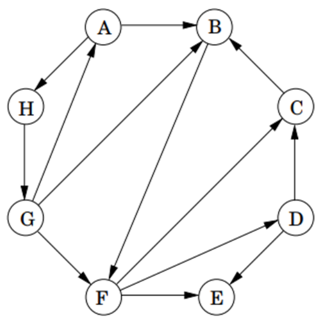
Warm-Up Problems (do not hand in for grading)

Work out the problems below as practice problems before going on to the next section. Note that warm-up problems might sometimes show up on an exam!

1. DPV Problem 3.3 (all parts)
2. Walk through the DFS algorithm starting from node A in the graphs below. Repeat but start from node G.
3. DPV Problem 3.4 (all parts)







Required Problems (hand in for grading)

Work out the problems below and submit for grading before the deadline.

1. DPV Problem 3.9 – Write your algorithm's runtime using Big $O()$ notation. Be sure to explain why your algorithm is a linear-time algorithm.

   SOLUTIONS AND GRADING GUIDELINES:

   In brief, two key steps: (1) calculate the degree of each node; then (2) systematically traverse each edge and accumulate degree sums in `twodegree[]` array positions corresponding to each endpoint node. Be sure solutions do not double-count or miscount degree values. A more detailed solution follows.

   $\Longrightarrow$ 2 points for counting the degree of each node
   First, the degree of each node can be determined by simply counting the number of elements in its adjacency list. More specifically, we can populate array `degree[]` with such values.

   $\Longrightarrow$ 2 points for traversing each edge and adding to `twodegree[]`
   Next, we populate the `twodegree[]` array first by initializing all entries to zero, then by using a modified `explore()` procedure that adds the degree of each neighbor of given node `v` to `twodegree[v]`, as in:

```
# see Figure 3.3
procedure explore(G,v)
Input:   G=(V,E) is a graph; v in V
Output:  visited(u) is set to true for all nodes u reachable from v

visited(v) = true
previsit(v)
for each edge (v,u) in E:
    twodegree[v] = twodegree[v] + degree[u]  # <====
    if not visited(u):
        explore(u)
postvisit(v)
```

   Alternatively, we can populate the `twodegree[]` array first by initializing all entries to zero, then by simply iterating through each edge of each node's adjacency list, as in:

```
for each edge (a,b) in E:
    twodegree[a] = twodegree[a] + degree[b]
```

   $\Longrightarrow$ 1 points for explanation of Big $O()$ runtime
   Runtime of calculating the degree of each node is $O(|V|)$. Runtime of calculating values for `twodegree[]` array is $O(|E|)$. Therefore, overall runtime is $O(|V| + |E|)$.

2. DPV Problem 3.15 (all parts)

SOLUTIONS AND GRADING GUIDELINES:

(a) $\Longrightarrow$ 2 points for correct modeling and claim

By modeling the intersections as nodes of a graph with streets as directed edges (since they are one-way streets), the claim that there is a way to drive legally from any intersection to any other intersection is equivalent to claiming that the graph is strongly connected.

$\Longrightarrow$ 2 points for correct solution (only one SCC)

A graph is strongly connected if and only if the graph has only one strongly connected component (SCC).

$\Longrightarrow$ 1 point for confirming algorithm runtime is linear

This can be checked in linear time (as shown in Section 3.4.2).

(The above details are sufficient for full credit on this part of the problem.)

(b) $\Longrightarrow$ 2 points for correct modeling and claim

The key to this problem is formulating the claim to state that starting from the town hall, one cannot get to any other SCC in the graph. This is equivalent to stating that the SCC containing the town hall node is a sink component.

$\Longrightarrow$ 2 points for correct solution

We can determine this in linear time by first identifying the components, then running DFS again from the town hall node to check if any edges leave the given component. (This two-step process can also be combined; a component found by the algorithm is a sink if and only if there are no edges leaving the component into any component found before it.)

Alternatively, we can run DFS (or BFS) starting from the town hall node, marking all reachable nodes as "reachable from town hall" in the process. Then, run DFS (or BFS) on the same graph but with all edges reversed (which can be shown to be a linear time algorithm), marking each node as "able to reach town hall."

If any node is marked as "reachable from town hall" but is not marked as "able to reach town hall," we know the answer is false, i.e., the SCC containing the town hall node is not a sink component.

$\Longrightarrow$ 1 point for confirming algorithm runtime is linear

In either case above, runtime is linear time. This must be stated somewhere in the solution given.

3. DPV Problem 3.22

   SOLUTIONS AND GRADING GUIDELINES:

   $\Longrightarrow$ 2 points for describing that graph must have only one source SCC
   A node from which all other nodes are reachable is called a *vista* node. If a graph has a vista node, then it must have only one source strongly connected component (SCC) since, by definition, nodes from two (or more) source SCCs are not reachable from one another. This one source SCC must contain the vista node. (Further, every node in the source SCC will be a vista node.)

   $\Longrightarrow$ 3 points for correct solution
   Given the above, the algorithm is to compute all SCCs of the given graph, verifying that there is only one source SCC.

   Alternatively, the algorithm is to run DFS from any node to identify a node in a source SCC, which will be the node with the highest `post` value. This node must be in a source SCC, but we're not done yet. We must also then run DFS (or BFS) from this node to check whether we can reach all other nodes of the given graph.

   $\Longrightarrow$ 1 point for confirming algorithm runtime is linear
   In either solution above, runtime is linear time. This must be stated somewhere in the solution given.

4. DPV Problem 3.23

   SOLUTIONS AND GRADING GUIDELINES:

   $\Longrightarrow$ 2 points for using linearization
   The given graph is a directed acyclic graph (DAG). Start by linearizing the DAG. Any path from s to t must pass through nodes between s and t in the linearized order; therefore, we can ignore the other nodes of G.

   $\Longrightarrow$ 3 points for clearly describing algorithm
   Let $s = v_0, v_1, \ldots, v_{k-1}, v_k = t$ be the nodes from s to t in the linearized order. For each i, let n_i be the number of paths from s to v_i. Since each path to a node i plus an edge (i, j) gives a path to node j, we have

   $$n_j = \sum_{(i,j) \in E} n_i$$

   Since $i < j$ for all $(i, j) \in E$, we can compute n_j values in increasing order of j.

   The answer then is the final calculated value n_k.

   $\Longrightarrow$ 1 point for confirming algorithm runtime is linear
   Linearization runs in linear time. Calculating n_j values in linearized order also runs in linear time. This must be stated somewhere in the solution given.

5. DPV Problem 3.24

   SOLUTIONS AND GRADING GUIDELINES:

   $\Longrightarrow$ 3 points for describing algorithm
   Start by linearizing the DAG. Since the edges can only go in the increasing direction of the linearized order and the required path must visit all nodes, the algorithm is to check if the DAG has edge $(i, i+1)$ for every pair of consecutive nodes i and $i+1$ in the linearized order.

   $\Longrightarrow$ 1 point for confirming algorithm runtime is linear
   Both linearization and the checking of outgoing edges from each node take linear time; therefore, the algorithm is linear. This must be stated somewhere in the solution given.