CS2040S: Data Structures and Algorithms

Tutorial Problems for Week 10: Graphs and Traversal

For: 18 Oct 2021, Tutorial 8

Solution: Secret! Shhhh... This is the solutions sheet.

Problem 1. Bipartite Graph Detection

A *bipartite graph* is a graph whose vertices can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V; but there is no edge between vertices in U and also no edge between vertices in V.

Given an undirected graph with n vertices and m edges, we wish to check if it is bipartite.

Describe the most efficient algorithm you can think of to check whether a graph is bipartite. What is the running time of your algorithm?

Solution: A bipartite graph has the following two properties:

- 2-colourable: it is possible to assign a colour to every vertex in the graph such that every vertex is coloured one of two colours (say red or blue), such that no two adjacent vertices are coloured with the same colour.
- No odd-length cycles: every cycle in the graph contains an even number of edges.

A graph possessing any one of the above properties will also possess the other two properties, i.e. (graph is bipartite) $\Leftrightarrow$ (graph is 2-colourable) $\Leftrightarrow$ (graph has no odd-length cycles).

Therefore, to check if a graph is bipartite, we can simply check if it is 2-colourable or it has no odd-length cycle. The algorithm for checking if a graph is 2-colourable is presented below. We run DFS on the graph, colouring vertices with alternating colours. During the DFS, if we discover that the neighbour of the current vertex is already assigned a colour, and the colour assigned is the same as the colour of the current vertex, we report that the graph is not 2-colourable and hence not bipartite. Otherwise, if the DFS manages to assign a colour to every vertex, we report that the graph is bipartite.

The running time of this algorithm is $O(n+m)$. A sample implementation of the DFS algorithm to check if a graph is bipartite in pseudocode can be found in Algorithm 1. Simply call DFS(v,blue) on all vertices v which have not been colored (i.e color[v] is white). If isBipartite is true at the end of the algorithm then it is a bipartite graph otherwise it is not a bipartite graph.

Algorithm 1 DFS for Bipartite Graph Detection

```
colour[1...n] ← WHITE                          ▷ Unvisited nodes are white
isBipartite ← true                              ▷ Flag to indicate if graph is bipartite
procedure DFS(u, c)                             ▷ u is the current vertex, c is the colour to be assigned to u
    if colour[u] ≠ white then                   ▷ This node has been visited before
        if colour[u] ≠ c then                   ▷ Colour of this node is different from colour to be assigned
            isBipartite ← false
        end if
        return
    end if
    colour[u] ← c
    for each neighbour v of u do
        if c = BLUE then
            DFS(v, RED)
        else
            DFS(v, BLUE)
        end if
    end for
end procedure
```

Problem 2. Cycle Detection

Given a graph with n vertices and m edges, we wish to check if the graph contains a cycle.

Problem 2.a. First, consider the case of an undirected graph. Describe an algorithm to check if the graph contains a cycle.

Solution: There are a few ways of doing this. Two ways are described below.

Method 1. If the graph does not contain cycles, then it must be a forest: a set of connected components where every connected component is a tree. One property of trees is: the number of vertices v in the tree and the number of edges e in the tree is related by $e = v - 1$. Therefore, for each component, we can perform a DFS to count the number of vertices v and edges e in that component, and check if $e = v - 1$.

Method 2. The DFS algorithm can be modified to detect cycles in an undirected graph by searching for *back-edges*, an edge that goes from the current vertex in the DFS to a vertex, other than the parent vertex, that has already been visited before. The presence of *back-edges* indicates the presence of a cycle, since it means we are able to visit the same vertex more than once in the algorithm.

The running time of the algorithms for both methods is $O(n + m)$. A sample implementation in pseudocode for the second method can be found in Algorithm 2.

Algorithm 2 DFS for Cycle Detection in Undirected Graphs

```
1: $visited[1 \dots v] \leftarrow false$
2: $hasCycle \leftarrow false$
3: procedure DFS($u$, $p$)    ▷ $u$ is the current vertex, $p$ is the predecessor of $u$ in the DFS tree
4:     $visited[u] \leftarrow true$
5:     for each neighbour $v$ of $u$ do
6:         if $v \neq p$ and $visited[v]$ then
7:             $hasCycle \leftarrow true$
8:         end if
9:         DFS($v, u$)
10:    end for
11: end procedure
```

Problem 2.b. Next, consider the case of a directed graph. Describe an algorithm to check if the graph contains a cycle.

Solution: There are a few ways of doing this as well. Three ways are described below.

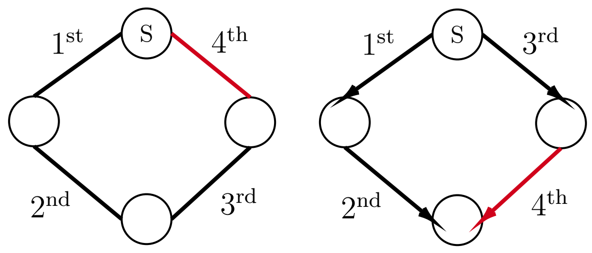
Method 1. A directed graph that contains a cycle is not a Directed Acyclic Graph (DAG). Therefore, such a graph does not have a topological ordering. Hence, we can run any topological sorting algorithm on the graph, and check if a valid topological ordering can be found.

Method 2. Notice that Algorithm 2 no longer works for the case of a directed graph, as directed graphs contain *cross edges.* A *cross edge* is an edge that connects a vertex to a previously visited vertex that is not the parent vertex of the current vertex, and that is not in the same branch of recursion used by DFS to visit the current vertex (you might want to check out 'DFS spanning tree'). Cross edges do not generate cycles, and we can modify Algorithm 2 to avoid reporting a cycle when a cross edge is detected. The modified algorithm is shown below in Algorithm 3.

Algorithm 3 DFS for Cycle Detection in Directed Graphs

```
1: $status[1 \dots v] \leftarrow NOT\_VISITED$
2: $hasCycle \leftarrow false$
3: procedure DFS($u$, $p$)    ▷ $u$ is the current vertex, $p$ is the predecessor of $u$ in the DFS tree
4:     $status[u] \leftarrow VISITING$
5:     for each neighbour $v$ of $u$ do
6:         if $v \neq p$ and $status[v] = VISITING$ then
7:             $hasCycle \leftarrow true$
8:         end if
9:         DFS($v, u$)
10:    end for
11:    $status[u] \leftarrow VISITED$
12: end procedure
```

The figure below shows examples of a *back edge* for 2a and *cross edges* for 2b.

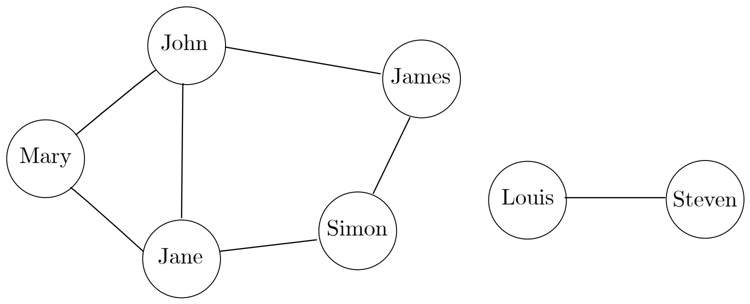


Back Edge Cross Edge

Method 3. The third way to solve this is to run Kosaraju's algorithm on the directed graph. If the number of SCCs we get is equal to the number of vertices then there is no cycle in the directed graph, otherwise it must mean there is at least one SCC with more than 1 vertex. In that/those SCC(s), there must be a cycle so that each vertex in the SCC can visit every other vertex in the SCC.

Problem 3. Friends Network (CS2010 AY17/18 Sem 1 Final Exam)

Peter is a very friendly person who has a lot of friends. In fact, he can construct a graph with n vertices and m edges, where the vertices represent his friends and the edges represent friends who know each other directly. An example is given below.



First, Peter wants to find out if a given pair of friends X and Y know each other directly (e.g John and Jane in the example).

Problem 3.a. What is the most appropriate graph data structure to store his friends graph in this scenario?

Solution: Adjacency Matrix

Problem 3.b. How would he answer his query using the graph data structure you have proposed in Problem 3a?

Solution: Check if the the entry in row X, column Y of the adjacency matrix is set to true. This can be done in $O(1)$ time.

Next, Peter wants to know if a given pair of friends X and Y are related to each other indirectly, that is, there is no edge from X to Y but there is at least one path from X to Y with more than 1 edge (e.g. Mary is related indirectly to Simon through Jane among other possibilities in the example).

Problem 3.c. What is the most appropriate graph data structure to store his friends graph in this scenario?

Solution: Adjacency List

Problem 3.d. Describe an algorithm that answers his query using the graph data structure you have proposed in Problem 3c. What is the running time of your algorithm?

Solution: Scan through the neighbours of X in the adjacency list and check if Y is connected directly to X. If so, we report that X is not indirectly connected to Y. This scan takes $O(n)$ time, since X can be connected to at most $n-1$ other vertices.

Otherwise, we perform a DFS starting from X to check if Y is in the same connected component as X. This takes $O(n+m)$ time.

Overall, our algorithm takes $O(n+m)$ time.

Finally, Peter wants to answer k queries of whether two given friends X and Y are related to each other *indirectly*.

Problem 3.e. Describe the most efficient algorithm you can think of to answer the k queries. What is the running time of *each* query?

Solution: By extending the idea in 3d, we perform a DFS on the adjacency list to label each connected component in the graph. Each vertex is labeled with the component number of the component it belongs to, and we can store the component numbers in an array. This preprocessing step takes $O(n+m)$ time.

To answer each query of whether two given friends X and Y are indirectly related, we first check if X and Y are directly related in $O(1)$ time using the adjacency matrix as done in 3b. If they are directly related, then they cannot be indirectly related. Otherwise, we can check if the component numbers of X and Y are the same in $O(1)$ time. If their component numbers are the same, they are in the same connected component and thus indirectly related, else they are not related at all. Thus, each query will take $O(1)$ time.

Problem 4. Skyscrapers

There are n skyscrapers in a city, numbered from 1 to n. You would like to order the skyscrapers by height, from the tallest to the shortest. Unfortunately, you do not know the exact heights of the skyscrapers.

Problem 4.a. Suppose that you have m pieces of information about the skyscrapers. Each piece of information tells you that skyscraper x is taller than skyscraper y for some pair of skyscrapers x and y. **Describe the most efficient algorithm you can think of to output any one possible ordering of the buildings by height which is consistent with the m pieces of information given. What is the running time of your algorithm?**

Solution: We can model the graph by representing the n skyscrapers as vertices, and the m pieces of information as directed edges. If the piece of information tells you that skyscraper x is taller than skyscraper y, then we represent that as a directed edge from x to y. Specifically, the graph will be a Directed Acyclic Graph (DAG), as the graph cannot have cycles (the heights cannot have a circular relation). Getting a possible ordering of heights consistent with the information given is thus equivalent to finding a topological ordering of the vertices. We can run the Topological Sort algorithm which has time complexity $O(n + m)$.

Problem 4.b. Suppose that you have m pieces of information about the skyscrapers. Each piece of information tells you one of the following regarding two skyscapers x and y:

- x is taller than y
- x has the same height as y

Describe the most efficient algorithm you can think of to output any one possible ordering of the buildings by height. What is the running time of your algorithm?

Solution: Since there can be skyscrapers with the same height, we cannot directly apply the Topological Sort algorithm in 4a since we have two different types of relations. However, using a UFDS we can "combine" the skyscrapers with the same height as one single vertex by reading in all the "equality" relations and and doing a unionSet operation on each pair of equal skyscrapers to create our combined "vertex". Thus, each "vertex" will represent one or more skyscrapers with the same height. This allows us to again obtain a DAG, which we can now apply the Topological Sort algorithm with $O(n + m)$ time complexity.