CS2040 Tutorial 9 Suggested Solution

Week 11, starting 24 Oct 2022

Q1 Graph Modelling and Searching

You have a graph **G** of **N** persons and **M** *friendships* between two people:

- If person **A** is a *friend* of person **B**, then **B** is also a *friend* of **A**
- If **A** is a *friend* of **B** and **B** is a *friend* of **C**, it does **NOT** mean that **A** and **C** must be *friends*
- If **X** and **Y** are *friends*, or **X** is *friend* of one *in a relationship* with **Y**, then **X** and **Y** have a *relationship*
- It is possible that for 2 people **X** and **Y**, there is no (direct/indirect) *relationship* between **X** and **Y**

Find the:

- best data structure / representation for **G** to be in
- the algorithm to solve each problem

and

- its time complexity

to solve each of these parts:

(a) Ali wants to find out if two distinct given people are *friends*
This query may be run **repeatedly Q** times

(b) Bob wants to find out, just once, if two distinct given people are in a *relationship*

(c) Bob wants to find out if 2 distinct given people are in a *relationship*
This query may be run **repeatedly Q** times

Answer

Each bullet point gives some information about **G**, *friendship*, and *relationship*:

- 1st shows that **G** is undirected
- 2nd shows that **A** and **B** being *friends* actually means that **A** and **B** are **adjacent**
- 3rd is a recursive relationship, showing that **A** and **B** being in a *relationship* means that they are in the same connected component
- 4th shows that **G** is **unconnected**

(a) It would be best for **G** to be an adjacency matrix, because we can then find whether the 2 people are adjacent (aka *friends*) in O(1) time. **Q** queries will take a total of O(**Q**) time.

(b) It would be best for **G** to be an adjacency list, so that breadth/depth first search can be efficiently used from one person as source to find the second person. If the second person is found, the answer is true. If no such person is found and all reachable vertices are visited, return false.

This takes O(**N** + **M**) time

(c) A brute force algorithm would repeat **(b)** Q times, resulting in O(**Q**(**N** + **M**)) a time algorithm over **Q** operations. However, we can just run one O(**N** + **M**) algorithm ONCE, and then perform **Q** queries in O(1) time each, resulting in an O(**N** + **M** + **Q**) algorithm over **Q** queries.

It would be best for **G** to be an adjacency list, so that breadth/depth first traversal can be efficiently used to label connected components:

> Using any possible *unvisited* vertex as the starting point for breadth-first or depth-first traversal, mark each reachable node with a *unique* component ID, so that you never miss out any node. Share the visited array across passes so that you never traverse nodes in the same connected component twice. This labels each vertex with a component ID

After labelling, each query can be performed in O(1) time by returning whether the 2 people have the same component ID.

Q2 Cycle Detection

An *undirected* graph **G** with **V** vertices and **E** edges is a tree if any one of the following properties hold:

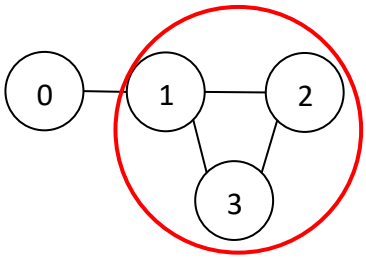
1. **G** is connected and **E** = **V** - 1
2. **G** has no cycles (is acyclic) and **E** = **V** - 1
3. There exists a unique path between every pair of vertices in **G**

You are quite interested in property #2 and want to explore further. **Implement efficient algorithms** to find if a non-empty graph **G**, represented using an **adjacency matrix**, contains a cycle when **G** is:

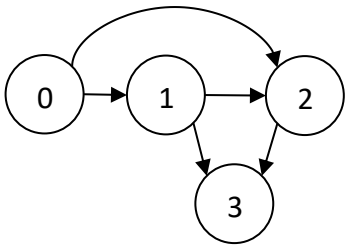
(a) undirected (remember that **G** may not be connected)

(b) directed (remember that **G** may not be connected - even if it is, it may not be strongly connected)

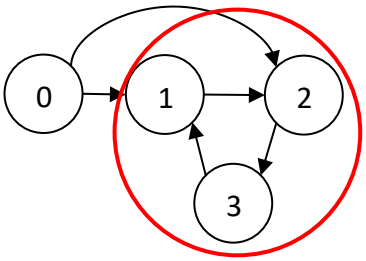
Determine the time complexity of your algorithm that works on the adjacency matrix, as well as what the time complexity would be if an adjacency list was used instead.



cycle in **undirected** graph



NO cycle in **directed** graph



cycle in **directed** graph

Answer

In an **undirected** graph, when attempting to generate a sequence of edges that form a cycle, we cannot move back to the parent vertex or that will create a false positive. E.g. 1->2->1 is not an undirected cycle. If we guard against this, then if searching any unvisited node leads to a visited node while recursing, we have found a cycle.

DFS can be used to find cycles efficiently, and may be easier to implement than using BFS for cycle detection as we can backtrack easily.

However, in a **directed** graph, while recursing to find cycles, landing at a visited node need not indicate the presence of a cycle. E.g. when DFS makes a path 0->1->2->3 and then 1->3 in the middle example, 3 will already have been visited on the second path, but it does NOT indicate the presence of a cycle.

FYI: Such edges like 1->3 are known as forward-edges/cross-edges (forward vs cross edges are different)

Once again, since the graph is not connected/strongly-connected, we have to start from every possible vertex as the source, or we may miss out cycles in other connected components / strongly connected components. As before, share the visited array to maintain efficiency.

Finding **undirected** cycle:

```
boolean hasUndirCycle(boolean[][] adjMat) {
   boolean[] visited = new boolean[adjMat.length];
   for (int idx = 0; idx < visited.length; idx++)
      if (!visited[idx] && // skip this vertex if previously visited
            hasUndirCycle(adjMat, visited, idx, -1)) // no parent
         return true; // 1 match is enough
   return false;
}

private boolean hasUndirCycle(boolean[][] adjMat, boolean[] visited,
        int curr, int parent) {
   if (visited[curr]) return true; // been here before, has cycle
   visited[curr] = true; // if we ever reach this vertex again, cycle
   for (int nb = 0; nb < adjMat.length; nb++) { // each neighbour
      if (!adjMat[curr][nb] || nb == parent) continue; //no edge or uturn
      if (hasUndirCycle(adjMat, visited, nb, curr)) return true;
   }
   return false;
}
```

Time complexity is O($\mathbf{N}^2$) as each of the **N** vertices will be visited exactly once. Each visit may require finding of up to **N** neighbours. With an adjacency matrix, we need to search all cells in a row to find all neighbours.

If adjacency list was used instead, neighbours of a vertex can be enumerated more quickly, resulting in O(**N** + **M**) time.

FYI: in terms of **N** only, **M** can be between 0 and choose(**N**, 2) so O(**N** + **M**) time will be in between O(**N**) and O($\mathbf{N}^2$) inclusive, depending on how sparse/dense the graph is.

Finding directed cycle

```
// {0,1,2} -> {unvisited, still visiting, confirmed safe}

boolean hasDirCycle(boolean[][] adjMat) {
   int[] visited = new int[adjMat.length];
   for (int idx = 0; idx < visited.length; idx++)
      if (visited[idx] != 2 && // skip this vertex if previously visited
            hasDirCycle(adjMat, visited, idx))
         return true; // 1 match is enough
   return false;
}

private boolean hasDirCycle(boolean[][] adjMat, int[] visited,
        int curr) {
   if (visited[curr] == 2) return false; // no out edges, confirmed safe
   if (visited[curr] == 1) return true; // been here but not safe, cycle
   visited[curr] = 1; // has cycle if we visit this vertex in this state
   for (int nb = 0; nb < adjMat.length; nb++) { // each neighbour
      if (!adjMat[curr][nb]) continue; //no edge
      if (hasDirCycle(adjMat, visited, nb)) return true;
   }
   visited[curr] = 2; // no cycle found, no more out edges, must be safe
   return false;
}
```

To differentiate between edges that lead to cycles (back-edges) and forward/cross-edges, there is a cycle if we are still recursing and hit a vertex which we have not returned from, while there is NO cycle involving a node that has already visited all outgoing edges but not managed to find a cycle.

This has the same time complexity as when finding undirected cycle (whether using adjacency matrix or adjacency list) as both perform **DFS** on each unvisited vertex, sharing visited array.

Question 3 (Online Discussion) – We're Under Attack!

In a computer game, there are **P** players. Each player is identified by an *id* in 0..(**P**-1), and has a *race* (in the game), either Orc or Human. Orcs are **supposed** to *attack* Humans only, and Humans are **supposed** to *attack* Orcs only. However, **some joker** will sometimes choose to *attack* someone of his own *race*.

Given the positive integer **P**, as well as a list of **A** distinct *attacks*, each of the form (attacking player id, victim player id), determine and output if:

- We're Under Attack! – some player is DEFINITELY *attacking* another of the same *race*
- Are we Under Attack? – it is possible that some player is *attacking* another of the same *race*
- Attack? What Attack? – it is DEFINITELY true that no player is *attacking* another of the same *race*

Unfortunately, you have NO information about the *race* of any of the **P** players =(

Assume you know the races first and draw some small test case. How do you know for sure that 2 players of the same race are fighting? Then work from there with the reality that you don't know anyone's race