Luis Gascon, Ethan Webb, Femi Dosumu
COSC 336
May 9, 2023

Assignment 8

Exercise 1.

Describe in plain English (a short paragraph with at most 5-6 lines should be enough) an algorithm for the following task:

Input: A directed graph $G = (V, E)$, and a node $u \in V$.

Goal: Output 1 if there is a path from every $v \in G$ to u (so if u is reachable from any other node), and output 0 otherwise.

Your algorithm should have runtime $O(n + m)$. (Hint: Use an idea that we have seen for similar connectivity problems in directed graphs.)

Our algorithm starts off by creating an array that stores the node representation, so we'll call it `nodeArr`. We then reverse the direction of the edges in our graph, which should take at most $O(1)$ time complexity. We can either traverse the graph breadth-first or depth-first, but for our algorithm, we'll implement the traversal using DFS. We'll create another array the size of `nodeArr`, called `rArr` to store the values of the traversed nodes. If the contents of `nodeArr` is equal to that of `rArr` then we return 1, else we return 0. This algorithm should have a time complexity of $O(n + m)$ because the graph traversal is our slowest operation.

Exercise 2.

We have seen that Dijkstra's algorithm can be implemented in two ways: Variant (a) uses an array to store the *dist*[] values of the unknown nodes, and Variant (b) uses a MIN-HEAP to store these values.

(a) Suppose in your application $m \leq 3n$. Which variant gives a faster runtime? Justify your answer.

Variant (b) would give us a faster runtime since our application uses a sparse graph.

$$m \leq 3n$$
$$m = O(n)$$

A min-heap provides us a run time of $\Theta(E \log V)$ since we perform at most $E \times$min heap operations, where each operation has a run time of $O(\log n)$.

(b) Suppose in your application $m \geq n^2/3$. Which variant gives a faster runtime? Justify your answer.

Variant (a) would give us a faster runtime since our application uses a dense graph.

$$m \geq \frac{n^2}{3}$$
$$m = \Omega(n^2)$$

Using an array to store the `dist` values is actually faster as it results in a run time of $O(V^2)$, regardless of how dense the graph is, while a min-heap results in a run time of $O(V^2 \log V)$.

Our algorithm starts off by initializing two integer arrays `dist` and `npath`. I've set the undiscovered nodes to have a distance of infinity, and to represent that, I've set the values to the the maximum integer value that Java supports. Using BFS to traverse through our graph helps us identify the shortest path.

```
int[] dist = new int[g.n];
int[] npath = new int[g.n];

for (int i = 0; i < g.n; i++) {
    dist[i] = Integer.MAX_VALUE;
    npath[i] = 0;
}
```

A queue is used for our BFS traversal where we dequeue the current node and store that value to `p`. We then traverse the neighboring nodes of `p` to check if that neighboring node has been visited.

If the neighboring node hasn't been visited, we set the distance of the neighboring node to the distance that the predecessor node is currently in, plus 1. The `npath` value is set to the `npath` value that the predecessor node has.

```
while (!queue.isEmpty()) {
  int p = queue.poll();
  for (int v : neighbors) {
      if (dist[v] == Integer.MAX_VALUE) {
          dist[v] = dist[p] + 1;
          npath[v] = npath[p];
          queue.add(v);
      }
          ...
}
```

If the neighboring node has been visited, and it's distance is equal to that of the proceeding node, plus 1, then we add the npath value of the proceeding node since it's a path from predecessor to neighbor.

```
  else if (dist[v] == dist[p] + 1)
    npath[v] += npath[p];
```

Lastly, we just output the values of arrays `dist` & `npath`.

```
for (int index = 1; index < g.n; index++)
System.out.printf("dist[%d] = %d \t npath[%d] = %d%n",
        index, dist[index], index, npath[index]);
```

The algorithm in its entirety.

```
static void getShortestPath(Adj_List_Graph g, int s) {
  int[] dist = new int[g.n];
  int[] npath = new int[g.n];

  for (int i = 0; i < g.n; i++) {
      dist[i] = Integer.MAX_VALUE;
      npath[i] = 0;
  }

  Queue<Integer> queue = new LinkedList<Integer>();

  dist[s] = 0;
  npath[s] = 1;
  queue.add(s);
  while (!queue.isEmpty()) {
      int p = queue.poll();
      List<Integer> neighbors = g.adj.get(p);
      for (int v : neighbors) {
          if (dist[v] == Integer.MAX_VALUE) {
              dist[v] = dist[p] + 1;
              npath[v] = npath[p];
              queue.add(v);
          }

          else if (dist[v] == dist[p] + 1)
              npath[v] += npath[p];

      }
  }

  for (int index = 1; index < g.n; index++)
      System.out.printf("dist[%d] = %d \t npath[%d] = %d%n",
              index, dist[index], index, npath[index]);

}
```

Program output can be found on the next page.

```
G1 results:

dist[2] = 1    npath[2] = 1
dist[3] = 1    npath[3] = 1
dist[4] = 1    npath[4] = 1
dist[5] = 2    npath[5] = 2
dist[6] = 2    npath[6] = 1
dist[7] = 3    npath[7] = 3

G2 results:

dist[2] = 1    npath[2] = 1
dist[3] = 1    npath[3] = 1
dist[4] = 1    npath[4] = 1
dist[5] = 1    npath[5] = 1
dist[6] = 1    npath[6] = 1
dist[7] = 2    npath[7] = 5
dist[8] = 3    npath[8] = 5
dist[9] = 3    npath[9] = 5
```