CS 624 - ANALYSIS OF ALGORITHMS

Assignment 5

Aravind Haridas – 02071139

1. Problem 22.1-1 (p592).

Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

solution: **Out-degree Computation:**

- The out-degree of each vertex v in a directed graph can be directly determined by the length of its adjacency list. Each list represents the set of edges leaving vertex v.
- **Computation Time:** The total time to compute the out-degrees for all vertices is based on the sum of the lengths of all adjacency lists. Since you need to access each list to determine its length, and some lists might be empty, the time complexity includes both the number of edges and vertices: $O(|E|+|V|)$. The term $|V|$ is necessary because accessing an empty list (indicating an out-degree of zero) still takes constant time per vertex.

In-degree Computation:

- Calculating the in-degrees is not as straightforward from the adjacency lists since you need to determine how many edges enter each vertex v, which requires examining all adjacency lists to count the occurrences of v.
- **Computation Time:** Like the out-degree, to compute the in-degrees for all vertices, you must traverse all adjacency lists. As each edge needs to be examined and each vertex needs at least to be initialized or checked, the time complexity is $O(|E|+|V|)$.
- The time complexity for computing both out-degrees and in-degrees in a directed graph with adjacency lists is $O(|E|+|V|)O(|E|+|V|)$. This complexity reflects the need to examine every edge and to at least access each vertex once, ensuring a comprehensive assessment of graph connectivity.

CS 624 - ANALYSIS OF ALGORITHMS

2. Problem 22.2-1

Show the d and π values that result from running breadth-first search on the directed graph of Figure 22.2(a), using vertex 3 as the source.

vertex	d(from vertex 3)	π
1	∞	NIL
2	3	4
3	0	NLI
4	2	5
5	1	3
6	1	3

Initial Setup: Thestarts at vertex 3, initializing it with **d[3] = 0** (distance from itself is zero) and **π[3] = NIL** (no predecessor as it is the source).

Propagation of BFS: **Vertex 5 and Vertex 6** both show **d = 1**, indicating they are direct neighbors of the source vertex 3. This implies that the first layer of BFS from vertex 3 directly reaches these vertices. The predecessors of both vertices 5 and 6 are vertex 3, which means they are directly connected to the source vertex without any intermediate vertices.

Further Layers: **Vertex 4** has **d = 2** and **π = 5**. This indicates that vertex 4 is reached from vertex 5, suggesting a path from 3 to 5 to 4.

Vertex 2 has **d = 3** and **π = 4**, suggesting a longer path in the traversal: from vertex 3 to vertex 5, then to vertex 4, and finally reaching vertex 2. This represents a third-layer traversal in BFS, further emphasizing the directionality and flow of paths in the graph.

Unreachable Vertex: **Vertex 1** remains unreachable (**d = ∞** and **π = NIL**), which is crucial to note as it indicates that no path exists from vertex 3 to vertex 1 within the directed graph, highlighting the presence of directed edges that do not facilitate a connection to vertex 1 from vertex 3.

Graph Structure Implications: The distances and predecessors captured in the BFS result table reflect the structural and directional properties of the graph. They are essential for understanding connectivity and reachability within the graph. Such BFS outcomes can help in identifying isolated components, assessing graph connectivity, and planning routes or flows within network structures based on direction and reachability.

CS 624 - ANALYSIS OF ALGORITHMS

3. Problem 22.2-2

Show the d and π values that result from running breadth-first search on the undirected graph of Figure 22.3, using vertex u as the source.

Sol:

vertex	Distance d from vertex u	π
r	4	s
s	3	w
t	1	u
u	0	Nil
v	5	r
w	2	t
x	1	u
y	1	u

BFS Initialization and Execution: BFS starts with vertex 'u' as the source, setting **d[u] = 0** and **π[u] = NIL**. This initialization signifies that 'u' is the root of the BFS tree and has no predecessor. The search proceeds in lexicographic order, exploring neighbors of 'u', which ensures a structured traversal based on the predefined ordering of vertices.

Discovery and Layers: **Vertices t, x, and y** are discovered directly from 'u' with **d = 1**. This indicates their adjacency to the source and positions them as the first layer in the BFS exploration. The equal distance suggests these vertices are direct neighbors of 'u' without any intermediaries. The BFS tree expands outward from these vertices to explore further layers, demonstrated by vertices **w** (**d = 2** from vertex 't') and then **s** (**d = 3** from vertex 'w'), reflecting a cascading effect of discovery through adjacent vertices.

Propagation Through the Graph: **Vertex r** is reached from 's' at a distance of 4, showing the sequential layering effect in BFS as the search radiates outward from 'u'. **Vertex v** is the most distal in this BFS tree, reached from 'r' at a distance of 5, representing the farthest point from 'u' based on the given traversal order and connectivity.

Graph Connectivity and Structure: The distances (**d**) and predecessors (**π**) effectively map out the structure of the graph from the perspective of 'u', illustrating how each vertex is connected and the path to reach it from 'u'.

This BFS result highlights the importance of traversal order (lexicographic in this case), which can influence the structure of the BFS tree and the discovery times of each vertex.

Analysis and Implications: The BFS tree's structure, as revealed by **π** values, can be crucial for understanding the graph's topology, identifying clusters or communities, and analyzing the shortest paths within the graph. The results also serve practical purposes in networking, routing, and planning applications, where the shortest paths and connectivity patterns are critical.

CS 624 - ANALYSIS OF ALGORITHMS

4. Problem 22.2-3 (p602) — Note, corrected in 3rd printing.

Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if line 18 were removed.

Sol: Given Pseudo code is

BFS(G, s)

```
1.  for each vertex u ∈ G.V - {s}
2.    u.color = WHITE
3.    u.d = ∞
4.    u.π = NIL
5.  s.color = GRAY
6.  s.d = 0
7.  s.π = NIL
8.  Q = Φ
9.    ENQUEUE(Q, s)
10.     while Q ≠ Φ
11.     u = DEQUEUE(Q)
12.     for each v ∈ G.Adj[u]
13.     if v.color == WHITE
14.         v.color = GRAY
15.         v.d = u.d + 1
16.         v.π = u
17.         ENQUEUE(Q, v)
18.   u.color = BLACK
```

In BFS, vertices are traditionally colored WHITE, GRAY, or BLACK to track their discovery and exploration status. The suggestion is to remove the use of BLACK (line 18), which marks full exploration, and argue whether using only WHITE and GRAY suffices.

Key Analysis:

Vertex Processing in BFS:

WHITE: Vertex has not been discovered.

GRAY: Vertex is discovered and enqueued, pending full exploration.

BLACK: Vertex has been fully explored, and all its adjacent vertices have been discovered.

By removing the transition to BLACK, vertices will remain GRAY after processing. However, since vertices are only enqueued when they are WHITE and are never re-enqueued once they turn GRAY, the BFS process remains efficient and correct.

Effectiveness of Two Colors:

The BFS queue mechanism ensures that each vertex is processed once. Vertices are dequeued, their adjacencies checked, and then they are effectively considered fully processed without needing to be marked BLACK.

The removal of BLACK does not affect the algorithm's ability to traverse the graph correctly, as GRAY already indicates active processing and WHITE ensures no vertex is revisited.

Conclusion:

Removing line 18 from the BFS algorithm (the line that changes vertex color from GRAY to BLACK) does not affect the correctness or outcome of BFS. The primary operational mechanism preventing vertices from being revisited is their inclusion in the queue as GRAY and their subsequent exclusion once processed. Therefore, using only WHITE and GRAY is sufficient to ensure proper BFS functionality, reducing memory usage by allowing vertex color to be stored with a single bit.

This modification simplifies the BFS algorithm without compromising its ability to accurately and efficiently traverse a graph, demonstrating a practical approach to algorithm optimization in terms of space efficiency.

CS 624 - ANALYSIS OF ALGORITHMS

5. Problem 22.2-4 (p602)

What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

The BFS implementation using an adjacency matrix representation requires $O(V^2)$ time, where V is the number of vertices in the graph. This is due to the necessity to check every possible vertex pair to ascertain connectivity, which is inherent to the matrix representation. This contrasts with an adjacency list representation where the complexity is dependent on the number of edges and vertices, typically $O(V+E)$, which can be significantly less than $O(V^2)$ especially in sparse graphs.

CS 624 - ANALYSIS OF ALGORITHMS

6. Problem 22.2-7 (p602)

There are two types of professional wrestlers: “babyfaces” (“good guys”) and “heels” (“bad guys”).Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries.

Give an O(n + r)-time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

Sol: Graph Representation: Model the wrestlers and their rivalries as a bipartite graph where vertices represent wrestlers and edges represent rivalries.

Two-Coloring Technique: Use a two-colouring method (e.g., colours representing "babyface" and "heel") to determine if the graph can be coloured such that no two adjacent vertices (representing rival wrestlers) have the same color.

Breadth-First Search (BFS): Implement BFS from any uncolored vertex, alternating colors between levels of the BFS tree. If any vertex requires a color that conflicts with a previously assigned color, the graph is not bipartite, and thus, the wrestlers cannot be split as required without conflicting rivalries.

Check and Output: If the BFS completes without conflicts, output the coloring as the division of wrestlers. If conflicts arise, indicate that a suitable division is not possible.

Time Complexity: The algorithm runs in *O(n+r)* time, where *n* is the number of wrestlers and r is the number of rivalries, due to the linear nature of BFS in graph traversal.

7. Problem 22.3-8 (p611) Give a counterexample to the conjecture that if a directed graph G contains a path from u to v and if v.d < u.d in in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.

Sol: To provide a counterexample to the conjecture that if a directed graph G contains a path from vertex u to vertex v, and if $u.d<v.d$ during a depth-first search (DFS) of G, then v must be a descendant of u in the depth-first forest produced, consider the following graph structure and DFS traversal scenario:

Graph Structure:

Consider a graph with vertices u,v,w,*u*,*v*,*w*, and x*x* and directed edges as follows:

- $u \rightarrow w$
- $w \rightarrow v$
- $v \rightarrow x$
- $u \rightarrow x$

```
u ----> w ----> v
|               |
|               |
+--------------> x
```

Depth-First Search Traversal:

1. Start at Vertex u:
   - Discover u ($u.d=1$).
   - Traverse to w.
2. Traverse from w to v:
   - Discover w ($w.d=2$).
   - Move to v, discover v ($v.d=3$).
3. Continue from v to x:
   - Discover x ($x.d=4$).
   - Complete exploration of x, mark x finished, backtrack to v, mark v finished.
4. Backtrack to w and then to u:
   - Finish exploring w, mark w finished.
   - Return to u and directly go to x (since x is already discovered and finished, no changes.
   - Finally, finish u.

Analysis of Traversal:

- The direct path from u to v via w leads to discovery and finish times such that $u.d$=1 and $v.d$=3.
- Although u is discovered before v and there is a path from u to v, v is not a descendant of u in the DFS tree. This is because after discovering v, the DFS continues to x and completes it before backtracking past v and w to u, which then directly accesses x.

Conclusion:

This traversal scenario demonstrates that despite $u.d<v.d$ and a direct path from u to v, the DFS structure does not result in v being a descendant of u. This counterexample effectively disproves the conjecture, highlighting that discovery and finish times alone do not determine descendant relationships in the DFS tree, particularly when non-linear paths and multiple branches are involved.

CS 624 - ANALYSIS OF ALGORITHMS

8. Problem 22.3-9 (p612) Give a counterexample to the conjecture that if a directed graph G contains a path from u to v then any depth-first search must result in v.d ≤ u.f.

Sol: **Graph Structure:** Consider a directed graph with three vertices {1, 2, 3} and directed edges:

- (1,2)
- (1,3)
- (2,1)

DFS Execution: Start DFS at vertex 1:

- Discover 1 ($1.d$=1).
- Move to vertex 2 ($2.d$=2).
- Return to vertex 1 and then discover vertex 3 ($3.d$=3).
- Finish processing vertex 3 ($3.f$=4).
- Finish processing vertex 2 ($2.f$=5).
- Finally, finish processing vertex 1 ($1.f$=6).

Analysis:
In this traversal, although there is a direct path from vertex 2 to vertex 3, the DFS starts from vertex 1 and processes vertex 2 before it processes vertex 3.

The critical point is that when vertex 2 is revisited after the traversal returns from vertex 3, the finish time for vertex 2 is only set after vertex 3 has been completely processed and finished.

This leads to the finish time of vertex 2 ($2.f$=5) being greater than the discovery time of vertex 3 ($3.d$=3), thus $v.d \leq u.f$ does not hold in this scenario where v is vertex 3 and u is vertex 2.

Counterexample Validation: This specific sequence shows that even if there's a path from vertex 2 to vertex 3, depending on the order of processing and the graph's structure, it's not guaranteed that every DFS will result in $v.d \leq u.f$ In this case, $3.d > 2.f$ disproves the conjecture effectively.

Conclusion : The counterexample effectively demonstrates that the conjecture that if a directed graph G contains a path from u to v, then any depth-first search must result in $v.d \leq u.f$ is not universally true. The example highlights the importance of the order of traversal and the graph's structure in determining the discovery and finishing times in DFS, which can vary significantly with different starting points and processing orders.

This explanation should align well with the principles discussed in your class notes, focusing on the specifics of DFS timings and their implications in proving or disproving conjectures about graph traversal properties.

CS 624 - ANALYSIS OF ALGORITHMS

9. Problem 22.5-1 (p620)

How can the number of strongly connected components of a graph change if a new edge is added?

For each possibility, give an example that illustrates it.

Sol: When adding a new edge to a directed graph, the number of strongly connected components (SCCs) can either decrease or remain unchanged, depending on how the edge affects the connectivity among the vertices:

1. Reduction in the Number of Strongly Connected Components

Explanation**:** Adding an edge can potentially connect two previously separate SCCs into a single SCC, thereby reducing the total number of SCCs in the graph. This occurs if the new edge creates a path that connects two vertices from different SCCs such that each vertex in the resulting union can reach every other vertex.

Example: Consider a graph with vertices {A, B, C} and directed edges {A → B, C → A}. Here, we have two SCCs: {A, B} (where A can reach B and vice versa via a cycle) and {C} (which is isolated except for its connection to A).

- Before addition: SCCs = {{A, B}, {C}}
- Adding an edge: Add an edge B → C.
- After addition: Now, C can reach A and B (via the new edge from B to C and existing edges), and A and B can reach C, thus forming a single SCC.
- Result: SCCs = {{A, B, C}}

2. No Change in the Number of Strongly Connected Components

Explanation: Adding an edge that does not connect vertices from different SCCs in a way that forms a cycle among them will not change the number of SCCs. This typically occurs when the new edge is added within an existing SCC or between vertices that do not contribute to forming a new cycle that includes vertices from multiple SCCs.

Example: Consider the same graph with vertices {A, B, C} and directed edges {A → B, B → A, C → A}.

- Before addition: SCCs = {{A, B}, {C}}
- Adding an edge: Add an edge B → C.
- After addition: Although B → C connects B to C, there is still no path from C back to B or A that completes a cycle including C with A and B. Thus, the structure of SCCs does not change.
- Result: SCCs = {{A, B}, {C}}

CS 624 - ANALYSIS OF ALGORITHMS

Summary:

When adding a new edge to a directed graph, it's essential to analyze whether the addition enables new cycles that can encompass vertices from different existing SCCs, thereby potentially reducing the number of SCCs. If the new edge does not facilitate such cycles, the number of SCCs remains unchanged. Each case depends significantly on the specific structure of the graph and the placement of the new edge. These scenarios align with the principles of graph theory as discussed in typical algorithms courses, emphasizing the impact of connectivity and cycles on the properties of graph components.

CS 624 - ANALYSIS OF ALGORITHMS

10. Problem 22.5-3 (p620)

Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of increasing finishing times. Does this simpler algorithm always produce correct results?

Sol: **Claim:** Professor Bacon suggests that the algorithm for finding strongly connected components (SCCs) could be simplified by using the original graph, not its transpose, and performing the second depth-first search (DFS) in order of increasing finishing times.

Counterexample Explanation:

- Graph Structure: Consider a graph with vertices {1, 2, 3} and edges {(2, 1), (2, 3), (3, 2)}.
- First DFS on Original Graph: Suppose we start the DFS at vertex 2. The DFS explores vertex 3 next, and then vertex 3 completes before returning to 2. Finally, vertex 1 is explored, ending up with finishing times likely in the order $f[3]<f[2]<f[1]$.
- Implication of Finishing Times: With these finishing times, if we were to perform the second DFS on the original graph in order of increasing finishing times (starting at 3), we face an issue:
- Starting at vertex 3, we can reach vertices 2 and then 1, suggesting erroneously that all vertices are part of the same SCC, which conflicts with the actual separate SCCs: {2, 3} and {1}.

Why It Fails:

The approach fails because scanning vertices in order of increasing finishing times in the original graph does not respect the crucial property that the transpose graph leverages: reversing edge directions ensures that the second DFS explores only within the same SCC, preventing cross-component traversal that can occur if the original graph is used.

A DFS on the original graph starting from a vertex with the smallest finish time (in this case, 3) could traverse back into earlier components (like reaching from 3 back to 2 and 1), falsely combining distinct SCCs into a single component.

Conclusion: Using the original graph and scanning vertices in order of increasing finishing times for the second DFS does not always produce correct results for identifying SCCs. The example provided illustrates how such an approach could merge separate SCCs into one, leading to incorrect SCC identification. This counterexample effectively disproves Professor Bacon's claim, highlighting the necessity of using the transpose of the graph and scanning vertices in decreasing order of their finishing times.