Basic Algorithms, Fall 2024
CSCI-UA.0310-001

Homework 8

Instructor: Rotem Oshman
Name: Ishan Pranav

References

Peer-reviewed with Crystal Huang.

Question 1: Breadth-first search

Recall the BFS algorithm from the lecture that runs on graph G starting from root node $s \in V$:

```
BFS(G, s)
    s.color = gray, s.dist = 0.
    for u ∈ V \ {s}
        u.color = white, u.dist = ∞
    Q = ∅
    Q.enqueue(s)
    while Q ≠ ∅
        u = Q.dequeue()
        for v ∈ Adj[u]
            if v.color = white
                v.color = gray
                v.dist = u.dist + 1
                Q.enqueue(v)
        u.color = black
```

Homework 8

1. Consider the undirected graph shown in Figure 1. If we begin a BFS traversal starting at node A, in what order are the nodes visited? Assume that the adjacency list is ordered alphabetically, i.e., we visit earlier letters alphabetically first, so from A we will visit B before C, and so on.

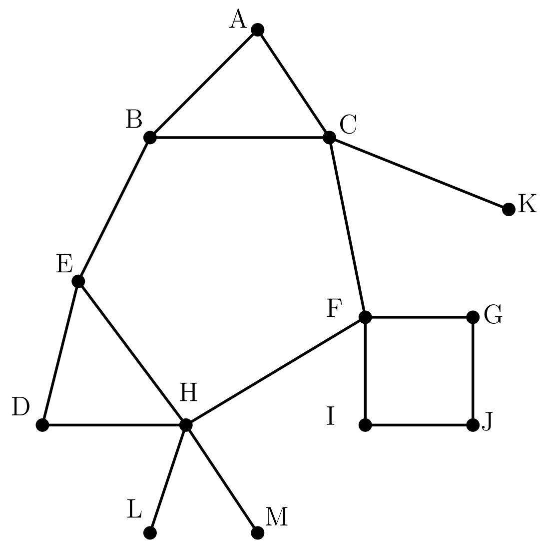

Figure 1: Graph to traverse

Solution: We can step through the breadth-first search algorithm to determine that the order of the vertices visited is A, B, C, E, F, K, D, H, G, I, L, M, and J.

First. We enqueue A.

Second. We dequeue A and enqueue B and C.

Third. We dequeue B and enqueue E.

Fourth. We dequeue C and enqueue F and K.

Fifth. We dequeue E and enqueue D and H.

Sixth. We dequeue F. We skip H. We enqueue G and I.

Seventh. We dequeue K.

Eighth. We dequeue H and enqueue L and M.

Ninth. We dequeue G and enqueue J.

Tenth. We dequeue I. We skip J.

Eleventh, twelfth, and thirteenth. We dequeue L, M, and J.

The vertices are visited as enqueued: A, B, C, E, F, K, D, H, G, I, L, M, and J.

2. Explain what happens if we don't check whether a node is white before enqueuing it.

Solution: Suppose we did not check whether a node is white before enqueuing. Then the breadth-first search algorithm may not terminate.

For example, suppose we perform this variant of breadth-first search on graph $G = (V, E)$ with vertices $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ and edges $E = \{\{\mathrm{A}, \mathrm{B}\}\}$. Assume, without loss of generality, that the initial vertex is A and that other vertices are visited in alphabetical order.

We can step through the algorithm to demonstrate the flaw.

First. We enqueue A.

Second. We dequeue A and enqueue B.

Third. We dequeue B and enqueue A. Note that we enqueued A in this situation because we did not check its color and thus ignored the fact that it had been previously visited.

This brings us back to the second step (dequeue A and enqueue B), meaning that the algorithm never terminates, even after all vertices have been visited.

3. Assume we replace the (FIFO) queue with a stack and therefore in line 7 obtain the node *last* added. Does $v.dist$ still represent the shortest distance from s to v?

Solution: No, using this implementation of breadth-first search, the distance assigned to a vertex v may not represent the shortest distance from the starting vertex s to v.

Consider the undirected graph $G = (V, E)$ with vertices $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$ and edges $E = \{\{\mathrm{A}, \mathrm{B}\}, \{\mathrm{A}, \mathrm{C}\}, \{\mathrm{B}, \mathrm{E}\}, \{\mathrm{C}, \mathrm{D}\}, \{\mathrm{D}, \mathrm{E}\}\}$.

Assume, without los of generality, that the initial vertex is A and that other vertices are visited in alphabetical order.

We can see that the shortest path from A to E is $(\mathrm{A}, \mathrm{B}, \mathrm{E})$, so the shortest distance from A to E is 2.

However, if we step through the modified breadth-first search algorithm (a hybrid of breadth-first search and depth first search), we see that the algorithm handles the immediate children of the source node correctly, but mishandles its other descendent vertices.

First. We push A with distance 0.

Second. We pop A, noting edges $\{\mathrm{A}, \mathrm{B}\}$ and $\{\mathrm{A}, \mathrm{C}\}$. We push B with distance $0 + 1 = 1$ and C with distance $0 + 1 = 1$.

Third. We pop C, noting edge $\{\mathrm{C}, \mathrm{D}\}$. We push D with distance $1 + 1 = 2$.

Fourth. We pop D, noting edge $\{\mathrm{D}, \mathrm{E}\}$. We push E with distance $2 + 1 = 3$. Already, this result is incorrect. We can continue stepping through the algorithm to show that it remains incorrect at termination.

Fifth. We pop E.

Sixth. We pop B, noting edge $\{\mathrm{B}, \mathrm{E}\}$. We skip E.

The algorithm incorrectly assigns a distance of 3 to E, although the shortest distance from A to E is 2. We conclude that, in the modified algorithm, the distance assigned to v is not necessarily the shortest distance from s to v.

Homework 8

Instructor: Rotem Oshman
Name: Ishan Pranav

4. Consider graphs for which edges have assigned weights. Assume we modify line 11 of BFS to add the weight of the edge from u to v instead of adding 1. Does the modified algorithm compute the shortest weighted distance correctly? If so, prove the correctness; otherwise, provide a counterexample for which the algorithm fails.

Solution: No, the modified algorithm is incorrect.

For example, suppose we perform this variant of the algorithm on graph $G = (V, E)$ with vertices $V = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$ and edges E. For each edge $e \in E$, note $e = (u, v, w)$, where u is the source vertex, v is the target vertex, and w is the weight of the edge. Suppose

$$E = \{(\mathrm{A}, \mathrm{B}, 10), (\mathrm{A}, \mathrm{C}, 1), (\mathrm{B}, \mathrm{D}, 20), (\mathrm{C}, \mathrm{D}, 2)\}.$$

Assume, without loss of generality, that G is directed, that the initial vertex is A, and that other vertices are visited in alphabetical order.

We can step through the algorithm to demonstrate the flaw.

First. We enqueue A with distance 0.

Second. We dequeue A, noting edges $(\mathrm{A}, \mathrm{B}, 10)$ and $(\mathrm{A}, \mathrm{C}, 20)$. We enqueue B with distance $0 + 10 = 10$ and C with distance $0 + 1 = 1$.

Third. We dequeue B, noting edge $(\mathrm{B}, \mathrm{D}, 20)$. We enqueue D with distance $10 + 20 = 30$.

Fourth. We dequeue C, noting edge $(\mathrm{C}, \mathrm{D}, 2)$. We skip D.

This gives 30 for the distance from A to D by following path $(\mathrm{A}, \mathrm{B}, \mathrm{D})$. However, there exists a path $(\mathrm{A}, \mathrm{C}, \mathrm{D})$ whereby the distance from A to D is 3. Of course, $3 < 30$. Thus, the modified algorithm fails.

Question 2: Universal sink

Recall that the *adjacency matrix* representation of a *directed* graph $G = (V, E)$ is the $|V| \times |V|$ matrix M_G where $M_G[i][j]$ is 1 if there is an edge from vertex i to vertex j, and 0 otherwise. You may assume no vertex in the graph has an edge going into itself (i.e., no self-loops). We want to determine whether there is any vertex in G that has edges coming to it from *all other* vertices but no edges going out from it (this is known as a **universal sink**).

1. Write an $O(|V|^2)$ algorithm that, given M_G, checks if there is a universal sink.

Solution:

Algorithm I. IsSink(M_G, v) with a $|V| \times |V|$ adjacency matrix M_G representing directed graph $G = (V, E)$ with no self-loops and a vertex $v \in V$; returns `true` if v is a universal sink; otherwise, `false`:

For $u \in (V \setminus \{v\})$, if $M_G[v, u] = 1$ or $M_G[u, v] = 0$, then return `false`.

Return `true`.

Algorithm II. QuadraticSink(M_G) with a $|V| \times |V|$ adjacency matrix M_G representing directed graph $G = (V, E)$ with no self-loops; returns `true` if there is a universal sink in G; otherwise, `false`:

For $v \in V$, if IsSink(M_G, v) is `true`, then return `true`.

Return `false`.

Proposition I. *Claim.* Algorithm I determines if v is a universal sink in G in running time $O(|V|)$.

Proof. By definition, v is a universal sink in $G = (V, E)$ if, for all $u \in V$ where $u \neq v$, we have $(u, v) \in E$ but $(v, u) \notin E$. The IsSink algorithm compares v to all other vertices u. It returns `true` only if v has edges coming to it from all other vertices u and no edge going out from it to any vertex u. Otherwise, IsSink returns `false`. Thus, Algorithm I correctly determines if v is a universal sink.

This implementation of Algorithm I visits every element in V and performs constant-time operations within the loop body, so it has running time $O(|V| \times 1) = O(|V|)$. □

Proposition II. *Claim.* Algorithm II determines if there exists a universal sink in G in running time $O(|V|^2)$.

Proof. For $G = (V, E)$, The QuadraticSink algorithm checks every vertex $v \in V$, using Algorithm I to determine if v is a universal sink. From Proposition I, Algorithm I correctly determines if v is a universal sink. Since Algorithm II returns `true` if Algorithm I determines that any vertex is a universal sink (and `false` if none satisfies this test), Algorithm II correctly determines if there exists a universal sink in G.

This implementation of Algorithm II visits every element in V and performs IsSink on each ($O(|V|)$ by Proposition I) within the loop body, so it has running time $O(|V| \times |V|) = O(|V|^2)$. □

2. Write an $O(|V|)$ algorithm for the problem. Justify why it is correct, as well as why it satisfies this run-time bound.

Solution:

Algorithm III. LinearSink(M_G) with a $|V| \times |V|$ adjacency matrix M_G representing directed graph $G = (V, E)$ with no self-loops; returns `true` if there is a universal sink in G; otherwise, `false`:

If $V = \emptyset$, then return `false`.

Let $v^* \leftarrow v \in V$, choosing v arbitrarily.

Let $U \leftarrow V \setminus \{v^*\}$.

Let $W \leftarrow \emptyset$.

For $u \in U$:

- if $M_G[v^*][u] = 1$ then:
  - assign $W \leftarrow W \cup \{v^*\}$;
  - assign $v^* \leftarrow u$;
- otherwise:
  - assign $W \leftarrow W \cup \{u\}$.

Return IsSink(M_G, v^*).

Invariant I. *Claim.* For every step of the loop, no vertex $w \in W$ is a universal sink.

Proof. We can prove this invariant by structural induction on W.

Basis. Consider $W = \emptyset$. Then the invariant holds vacuously at initialization.

Hypothesis. Consider $W = X$ where $X \neq \emptyset$. Assume there exists no universal sink $w \in W$.

Inductive step. Consider $W = X \cup \{v\}$ for $v \in V$. From Algorithm III, there are two cases:

- Suppose $M_G[v^*][u] = 1$. Then $v = v^*$ (before the reassignment of v^*). Thus $M_G[v][u] = 1$, indicating that there is an outgoing edge from v, or $(v, u) \in E$. So v is not a universal sink.
- Suppose instead $M_G[v^*][u] = 0$. Then $v = u$. Thus $M_G[v^*][v] = 0$, indicating that there is no incoming edge to v, or $(v^*, v) \notin E$. So v is not a universal sink.

In all cases, v is not a universal sink.

For all $x \in X$, the invariant holds by the inductive hypothesis. Since the invariant holds for v, it must therefore hold for all vertices $w \in (X \cup \{v\})$.

Hence, by structural induction, there exists no universal sink $w \in W$. This invariant holds at initialization, maintenance, and termination.

Proposition III. *Claim.* Algorithm III determines if there exists a universal sink in G in running time $O(|V|)$.

Proof. Suppose $V = \emptyset$. Then there is no universal sink in G, and Algorithm III returns `false`, which is correct.

Suppose instead $V \neq \emptyset$. Then there exists an arbitrary $v^* \in V$.

The loop executes for each of the $(|V| - 1)$ elements in $U = V \setminus \{v^*\}$.

From Algorithm III, at initialization we have $|W| = 0$. Also, for each iteration, exactly one element is inserted into W in all cases. This implies that $|W| = (|V| - 1)$ at termination. From Invariant I, at termination, we have for all $w \in W$ that w is not a universal sink. Of course $v^* \notin W$, but $v^* \in V$. In fact, v^* is the only vertex that has not been eliminated.

If v^* is a universal sink, then there is a universal sink in G; otherwise, there is none.

Algorithm III returns $\textsc{IsSink}(M_G, v^*)$, which, by Proposition I, correctly determines if v^* is a universal sink. Thus, Algorithm III correctly determines if there exists a universal sink in G.

The loop body performs constant-time operations in all cases, so the running time of the loop is $O((|V|-1) \times 1) = O(|V|-1)$. The running time of $\textsc{IsSink}(M_G, v^*)$ is $O(|V|)$ from Proposition I. Thus, the running time of $\textsc{LinearSink}$ is $O(|V| - 1) + O(|V|) = O(|V|)$. $\square$