Graph Exploration

- Elucidating graph properties provides a powerful tool for understanding networks and their emergent properties.
- Some properties of interest include: degree distribution, community structure, node centrality, distances, clustering coefficients,…
- Quantifying these properties provides an alternative to “inspection by visualization.”
Graph Exploration

- Furthermore, when the graph is large, exhaustive analysis of the entire graph is infeasible.
- The alternative: Explore a region (or, regions) of the graph, and report the results based on the explored part(s).
Graph Exploration

- Graph exploration can be done
  - deterministically: using, for example, breadth-first search (BFS) or depth-first search (DFS)
  - nondeterministically: using random walks
- In this part, we will focus on BFS.
Breadth-first Search (BFS)

- The main idea behind BFS is very simple:
  - Starting from some pre-specific source node, visit the node and its neighbors; then, for each neighbor, visit its neighbors; and so on until no more nodes can be explored.
Breadth-first Search (BFS)

❖ The question is: How do we ensure that all the neighbors of a given node are visited before any of their neighbors are?
❖ The answer: By using an appropriate data structure!
Queues

- A queue is a data structure that implements a first-in first-out (FIFO) data access model.
- Think of how a queue works at a cafeteria: The first person to enter the queue would be the first person served and the first person to leave the queue.
- Contrast this with a last-in first-out (LIFO) model: Think of the trays at the cafeteria; the last tray added to the stack would be the first one picked up for use.
Queues

Enter the queue (last)

Leave the queue (first)
Two main operations are defined on queue $Q$:

- $enqueue(Q,x)$: add element $x$ to the end of the queue
- $dequeue(Q)$: remove the first element that was enqueued into $Q$ and return it
BFS and Queues

- When a node $u$ is visited during the exploration of the graph, all of its neighbors are enqueued.
- When done visiting all of $u$’s siblings, $u$’s neighbors are visited by getting them out of the queue.
- So, queues are used in BFS to ensure a proper implementation of the algorithm.
Algorithm 1: BFS.

Input: Graph $g = (V, E)$; source node $i \in V$.
Output: $v_j \in \{\text{True, False}\} \ \forall j \in V$.

1. foreach $j \in V$ do
2. \hspace{1em} $v_j \leftarrow \text{False}$; \hspace{3em} // Node $j$ has not been visited yet
3. \hspace{1em} $v_i \leftarrow \text{True}$; \hspace{3em} // Start by visiting the source node $i$
4. Initialize $Q$ to an empty queue;
5. $\text{enqueue}(Q, i)$;
6. while $Q$ is not empty do
7. \hspace{1em} $j \leftarrow \text{dequeue}(Q)$;
8. \hspace{1em} foreach neighbor $h$ of $j$ do
9. \hspace{2em} if $v_h = \text{False}$ then
10. \hspace{3em} $v_h \leftarrow \text{True}$;
11. \hspace{3em} $\text{enqueue}(Q, h)$;
12. return $v$;
Algorithm 1: BFS.

Input: Graph $g = (V, E)$; source node $i \in V$.
Output: $v_j \in \{\text{True}, \text{False}\}$ $\forall j \in V$.

1. $\forall j \in V$ do
2. $v_j \leftarrow \text{False}$; \hfill // Node $j$ has not been visited yet
3. $v_i \leftarrow \text{True}$; \hfill // Start by visiting the source node $i$
4. Initialize $Q$ to an empty queue;
5. $\text{enqueue}(Q, i)$;
6. while $Q$ is not empty do
7. $j \leftarrow \text{dequeue}(Q)$;
8. $\forall$ neighbor $h$ of $j$ do
9. \hfill if $v_h = \text{False}$ then
10. \hfill $v_h \leftarrow \text{True}$;
11. \hfill $\text{enqueue}(Q, h)$;
12. return $v$;

Algorithm 2: Compute Distances.

Input: Graph $g = (V, E)$; source node $i \in V$.
Output: $d_j \in (N[\{1\}]$ $\forall j \in V$.

1. $\forall j \in V$ do
2. $d_j \leftarrow 1$; \hfill // All nodes are initially at distance 1 from the source node $i$
3. $d_i \leftarrow 0$; \hfill // Initialize the distance to the source node $i$
4. Initialize $Q$ to an empty queue;
5. $\text{enqueue}(Q, i)$;
6. while $Q$ is not empty do
7. $j \leftarrow \text{dequeue}(Q)$;
8. $\forall$ neighbor $h$ of $j$ do
9. \hfill if $d_h = 1$ then
10. \hfill $d_h \leftarrow d_j + 1$; \hfill // The distance to node $h$ is updated
11. \hfill $\text{enqueue}(Q, h)$;
12. return $d$;

B. \[15 \text{ pts}\]

Give the pseudo-code of Algorithm IsBipartite that, given a graph $g = (V, E)$, determines whether the graph is bipartite and takes on the order of $m+n$ operations.

Hint: Think of slightly modifying Algorithm BFS.

C. \[15 \text{ pts}\]

Give the pseudo-code of Algorithm ComputeLargestCCSize that, given a graph $g = (V, E)$, computes the size (in terms of the number of nodes) of the largest connected component of $g$. Your algorithm must perform on the order of $m+n$ operations in the worst case on a graph with $n$ nodes and $m$ edges and you need to discuss why that is the case. If you decide to use sets and set operations, then the times reported in Problem 1 must be assumed.

The pseudo-code of Algorithm ComputeLargestCCSize is given in Algorithm 3. The loop on Line 4 visits each node once. However, if the graph has $k$ connected components, the condition of the if statement on Line 5 would be true exactly $k$ times. Therefore, the block between Lines 6 and 17 is executed once for each connected component in a manner identical to how BFS explores a graph.
Breadth-First Search (BFS)

Algorithm 1: BFS.

Input: Graph \( g = (V, E) \); source node \( i \in V \).
Output: \( v_j \in \{ \text{True}, \text{False} \} \) \( \forall j \in V \).

1. \( \text{foreach} \ j \in V \ do \)
2. \( \quad v_j \leftarrow \text{False}; \)  // Node \( j \) has not been visited yet
3. \( \quad v_i \leftarrow \text{True}; \)  // Start by visiting the source node \( i \)
4. Initialize \( Q \) to an empty queue;
5. \( \text{enqueue}(Q, i); \)
6. \( \text{while} \ Q \text{ is not empty} \ do \)
7. \( \quad j \leftarrow \text{dequeue}(Q); \)
8. \( \quad \text{foreach neighbor} \ h \ of \ j \ do \)
9. \( \quad \quad \text{if} \ v_h = \text{False} \ then \)
10. \( \quad \quad \quad v_h \leftarrow \text{True}; \)
11. \( \quad \quad \quad \text{enqueue}(Q, h); \)
12. \( \text{return} \ v; \)

Algorithm 2: ComputeDistances.

Input: Graph \( g = (V, E) \); source node \( i \in V \).
Output: \( d_j \in \{ 1 \} \) \( \forall j \in V \).

1. \( \text{foreach} \ j \in V \ do \)
2. \( \quad d_j \leftarrow 1; \)  // All nodes are initially at distance 1 from the source node
3. \( \quad d_i \leftarrow 0; \)  // Initialize the distance to the source node
4. Initialize \( Q \) to an empty queue;
5. \( \text{enqueue}(Q, i); \)
6. \( \text{while} \ Q \text{ is not empty} \ do \)
7. \( \quad j \leftarrow \text{dequeue}(Q); \)
8. \( \quad \text{foreach neighbor} \ h \ of \ j \ do \)
9. \( \quad \quad \text{if} \ d_h = 1 \ then \)
10. \( \quad \quad \quad d_h \leftarrow d_j + 1; \)  // The distance to node \( h \) is updated
11. \( \quad \quad \text{enqueue}(Q, h); \)
12. \( \text{return} \ d; \)
BFS and Queues

❖ If the input graph $g$ has $n$ nodes and $m$ edges, how many operations does BFS perform?
❖ Key idea: Think of how many times each edge is traversed during the execution of the algorithm.
Questions?