CSE 100
Graph Search
Dijkstra’s Shortest Path
Q1: Which of the following is NOT one of the two different graph representations discussed in the book?

A. Adjacency List
B. Adjacency Tree
C. Adjacency Matrix

B. Adjacency Tree
Q2: What is an advantage to using an adjacency list over an adjacency matrix to represent a graph?

A. It is more space efficient for sparse graphs (graphs with few edges)
B. It is more space efficient for dense graphs (graphs with lots of edges)
C. Linked lists are considered easier to implement than arrays
Q3: What is a "graph traversal?"

A. Visiting each vertex only one time
B. Finding a tree that includes each vertex in the graph
C. Finding the shortest path between all pairs of vertexes in the graph
Q4: Which of the following is a limitation of Dijkstra’s algorithm?

A. It can only handle graphs with no cycles (acyclic graphs)
B. It can only handle graphs with no negative edges
C. It can only handle graphs with no negative cycles (but can handle negative edges)

B. It can only handle graphs with no negative edges
Breadth First Search

- Explore all the nodes reachable from a given node before moving on to the next node to explore.
Breadth First Search

• Explore all the nodes reachable from a given node before moving on to the next node to explore

Assuming BFS chooses the lower number node to explore first, in what order does BFS visit the nodes in this graph?

A. V0, V1, V2, V3, V4, V5
B. V0, V1, V3, V4, V2, V5
C. V0, V1, V3, V2, V4, V5
D. Other
BFS Traverse: Sketch of Algorithm

The basic idea is a breadth-first search of the graph, starting at source vertex \( s \)

- Initially, mark all vertices as unexplored
- Start at \( s \); mark \( s \) as explored
- Enqueue \( s \) into a queue
- While the queue is not empty:
  - Dequeue the vertex \( v \) from the head of the queue
  - For each of \( v \)’s adjacent nodes, \( w \) that has not yet been visited i.e. not marked as explored:
    - Mark \( w \) as explored
    - Enqueue it in the queue
Shortest Path using BFS

The basic idea is a breadth-first search of the graph, starting at source vertex $s$:

- Initially, give all vertices in the graph a distance of INFINITY.
- Start at $s$; give $s$ distance = 0.
- Enqueue $s$ into a queue.
- While the queue is not empty:
  - Dequeue the vertex $v$ from the head of the queue.
  - For each of $v$'s adjacent nodes that has not yet been visited (i.e. distance to source is infinity):
    - Mark its distance as $1 +$ the distance to $v$.
    - Enqueue it in the queue.
The basic idea is a breadth-first search of the graph, starting at source vertex $s$

- Initially, give all vertices in the graph a distance of INFINITY
- Start at $s$; give $s$ distance = 0
- Enqueue $s$ into a queue
- While the queue is not empty:
  - Dequeue the vertex $v$ from the head of the queue
  - For each of $v$'s adjacent nodes that has not yet been visited:
    - Mark its distance as $1 +$ the distance to $v$
    - Enqueue it in the queue

What is the time complexity (in terms of $|V|$ and $|E|$) of this algorithm?

A. $|V|$
B. $|E|$
C. $|V| \times |E|$
D. $|V| + |E|$
E. $|V|^2$

Note: We are assuming the graph is represented using an adjacency list.

Method 1: Bound the whole loop, bound each substep and multiply by the number of iterations.

Method 2: Unroll the whole loop [this method gives a tight bound].
BFS Traverse: Details

source

V0: dist= 0  prev= -1  adj: V1
V1: dist= 1  prev= V0  adj: V4, V3
V2: dist= 3  prev= V3  adj: V0, V5
V3: dist= 2  prev= V1  adj: V2, V5, V6
V4: dist= 2  prev= V1  adj: V1, V6
V5: dist= 3  prev= V3  adj:
V6: dist= 3  prev= V4  adj: V5

Discuss and fill
Representing the graph with structs

```cpp
#include <iostream>
#include <limits>
#include <vector>
#include <queue>

using namespace std;

struct Vertex
{
    vector<int> adj; // The adjacency list
    int dist;       // The distance from the source
    int index; // The index of this vertex
    int prev;   // The index of the vertex previous in the path
};

vector<Vertex*> createGraph()
{
    ... }
```
**Traverse the graph using BFS */

```cpp
void BFSTraverse(vector<Vertex*> theGraph, int from )
{
    queue<Vertex*> toExplore;
    Vertex* start = theGraph[from];
    start->dist = 0;
    toExplore.push(start);
    // finish the code...
}
```

- Initially, give all vertices in the graph a distance of INFINITY
- Start at \( s \); give \( s \) distance = 0
- Enqueue \( s \) into a queue
- **While the queue is not empty:**
  - Dequeue the vertex \( v \) from the head of the queue
  - For each of \( v \)'s adjacent nodes that has not yet been visited:
    - Mark its distance as \( 1 + \) the distance to \( v \)
    - Enqueue it in the queue
/** Traverse the graph using BFS */
void BFSTraverse( vector<Vertex*> theGraph, int from )
{
    queue<Vertex*> toExplore;
    Vertex* start = theGraph[from];
    start->dist = 0;
    toExplore.push(start);
    while ( !toExplore.empty() ) {
        Vertex* next = toExplore.front();
        toExplore.pop();
        vector<int>::iterator it = next->adj.begin();
        for ( ; it != next->adj.end(); ++it ) {
            Vertex* neighbor = theGraph[*it];
            if (neighbor->dist == numeric_limits<int>::max()) {
                neighbor->dist = next->dist + 1;
                neighbor->prev = next->index;
                toExplore.push(neighbor);
            }
        }
    }
}

While the queue is not empty:
  ▶ Dequeue the vertex \( v \) from the head of the queue
  ▶ For each of \( v \)’s adjacent nodes that has not yet been visited:
    • Mark its distance as 1 + the distance to \( v \)
    • Enqueue it in the queue
Breadth First Search

- Explore all the nodes reachable from a given node before moving on to the next node to explore.

Does BFS always find the shortest path from the source to any node?

A. Yes for unweighted graphs
B. Yes for all graphs
C. No
In a weighted graph, the number of edges no longer corresponds to the length of the path. We need to decouple path length from edges, and explore paths in increasing *path length* (rather than increasing number of edges).

In addition, the first time we encounter a vertex may, we may not have found the shortest path to it, so we need to delay committing to that path.
Shortest path

- For each node in the graph, find the path to source with the shortest cumulative path length

Input: $G = (V, E)$

- $(v, w, lw)$: weight of edge $(v, w)$
- Source: $s$

Output: Shortest path from source to all vertices in the graph $L(v)$, for $v \in V$

Assumptions: No negative edges
Dijkstra’s Algorithm

X: All vertices for which shortest path has been computed
X: \{v_0\} , L(v_0) = 0

Goal: Select the vertex in (V-X) with the shortest path from source. How?

Tentative calculation of distance to the source, also called Dijkstra Score

\[ J(v,w) = L(v) + l_{vw} \]

Iteration 1: \( X = \{ v_0, v_3 \} \) \( L(v_3) = 1 \)

Iteration 2: \( Y = \{ v_0, v_3, v_2 \} \) \( L(v_2) = 2 \)

At each iteration, choose the edge with min. score & add corresponding vertex that was in V-X to X

\[ J(v_3,v_1) = 4 \]
\[ J(v_0,v_1) = 5 \]
\[ J(v_3,v_2) = 2 \]