CSE 100: GRAPH ALGORITHMS
Graphs: Example

A directed graph

\[ V = \{ \} \]

\[ |V| = \]

\[ E = \{ \} \]

\[ |E| \]

Path:
A graph $G = (V,E)$ consists of a set of vertices $V$ and a set of edges $E$

- Each edge in $E$ is a pair $(v,w)$ such that $v$ and $w$ are in $V$.
- If $G$ is an undirected graph, $(v,w)$ in $E$ means vertices $v$ and $w$ are connected by an edge in $G$. This $(v,w)$ is an unordered pair.
- If $G$ is a directed graph, $(v,w)$ in $E$ means there is an edge going from vertex $v$ to vertex $w$ in $G$. This $(v,w)$ is an ordered pair; there may or may not also be an edge $(w,v)$ in $E$.
- In a weighted graph, each edge also has a “weight” or “cost” $c$, and an edge in $E$ is a triple $(v,w,c)$.
- When talking about the size of a problem involving a graph, the number of vertices $|V|$ and the number of edges $|E|$ will be relevant.
Connected, disconnected and fully connected graphs

- Connected graphs:

- Disconnected graphs:

- Fully connected (complete graphs):
Q: What are the minimum and maximum number of edges in a undirected connected graph $G(V,E)$ with no self loops, where $N=|V|$?

A. $0, N^2$
B. $N, N^2$
C. $N-1, N(N-1)/2$
Sparse vs. Dense Graphs

A dense graph is one where $|E|$ is “close to” $|V|^2$. A sparse graph is one where $|E|$ is “closer to” $|V|$. 
A 2D array where each entry $[i][j]$ encodes connectivity information between $i$ and $j$

- For an unweighted graph, the entry is 1 if there is an edge from $i$ to $j$, 0 otherwise
- For a weighted graph, the entry is the weight of the edge from $i$ to $j$, or “infinity” if there is no edge
- Note an undirected graph’s adjacency matrix will be symmetrical
Representing Graphs: Adjacency Matrix

How big is an adjacency matrix in terms of the number of nodes and edges (BigO, tightest bound)?
A. \(|V|\)
B. \(|V| + |E|\)
C. \(|V|^2\)
D. \(|E|^2\)
E. Other

When is that OK? When is it a problem?
A dense graph is one where $|E|$ is “close to” $|V|^2$.
A sparse graph is one where $|E|$ is “closer to” $|V|$.

Adjacency matrices are **space inefficient** for sparse graphs
Representing Graphs: Adjacency Lists

- Vertices and edges stored as lists
- Each vertex points to all its edges
- Each edge points to the two vertices that it connects
- If the graph is directed: edge nodes differentiate between the head and tail of the connection
- If the graph is weighted edge nodes also contain weights
Each vertex has a list with the vertices adjacent to it. In a weighted graph this list will include weights.

How much storage does this representation need? (BigO, tightest bound)
A. $|V|$
B. $|E|$
C. $|V| + |E|$
D. $|V|^2$
E. $|E|^2$
Searching a graph

- Find if a path exists between any two nodes
- Find the shortest path between any two nodes
- Find all nodes reachable from a given node

Generic Goals:
- Find everything that can be explored
- Don’t explore anything twice
Generic approach to graph search
Depth First Search for Graph Traversal

- Search as far down a single path as possible before backtracking
Depth First Search for Graph Traversal

• Search as far down a single path as possible before backtracking

Assuming DFS chooses the lower number node to explore first, in what order does DFS 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. V0, V1, V2, V4, V5, V3
Unweighted Shortest Path

- Input: an unweighted directed graph $G = (V, E)$ and a source vertex $s$ in $V$
- Output: for each vertex $v$ in $V$, a representation of the shortest path in $G$ that starts in $s$ and ends at $v$ and consists of the minimum number of edges compared to any other path from $s$ to $v$
Depth First Search for Graph Traversal

• Search as far down a single path as possible before backtracking

Does DFS always find the shortest path between nodes?
A. Yes
B. No
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: Idea**

- Input: an unweighted directed graph $G = (V, E)$ and a source vertex $s$ in $V$
- Output: for each vertex $v$ in $V$, a representation of the shortest path in $G$ that starts in $s$ and ends at $v$

Start at $s$. It has distance 0 from itself.
Consider nodes adjacent to $s$. They have distance 1. Mark them as visited.
Then consider nodes that have not yet been visited
    - adjacent to those at distance 1. They have distance 2. Mark them as visited.
Etc. etc. until all nodes are visited.
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
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
BFS Traverse: Sketch of Algorithm

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

Questions:
- What data do you need to keep track of for each node?
- How can you tell if a vertex has been visited yet?
- This algorithm finds the length of the shortest path from s to all nodes. How can you also find the path itself?
BFS Traverse: Details

source

\[ \begin{align*}
V_0: \text{dist} &= \quad \text{prev} = \quad \text{adj: } V_1 \\
V_1: \text{dist} &= \quad \text{prev} = \quad \text{adj: } V_3, V_4 \\
V_2: \text{dist} &= \quad \text{prev} = \quad \text{adj: } V_0, V_5 \\
V_3: \text{dist} &= \quad \text{prev} = \quad \text{adj: } V_2, V_5, V_6 \\
V_4: \text{dist} &= \quad \text{prev} = \quad \text{adj: } V_1, V_6 \\
V_5: \text{dist} &= \quad \text{prev} = \quad \text{adj: } \\
V_6: \text{dist} &= \quad \text{prev} = \quad \text{adj: } V_5
\end{align*} \]

The queue (give source vertex dist=0 and prev=-1 and enqueue to start):

\[ \text{HEAD} \quad V_0 \quad \text{TAIL} \]

![Graph diagram with vertices and edges]
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()
{
    ... 
}
Unweighted Shortest Path: C++ code

/** Traverse the graph using BFS */
void BFSTraverse( vector<Vertex*> theGraph, int from )
{
    queue<Vertex*> toExplore;
    Vertex* start = theGraph[from];
    // finish the code…


struct Vertex
{
    vector<int> adj;
    int dist;
    int index;
    int prev;
};
/** 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);
            }
        }
    }
}
What is this algorithm??

Mystery(G,v) ( v is the vertex where the search starts )
  Stack S := {}; ( start with an empty stack )
  for each vertex u, set visited[u] := false;
  push S, v;
  while (S is not empty) do
    u := pop S;
    if (u is not visited) then
      visited[u] := true;
      for each unvisited neighbour w of u
        push S, w;
    end if
  end while
END Mystery()

Options:
A. BFS  B. DFS  C. Dijkstra’s algorithm  D. Nothing interesting
Why Did BFS Work for Shortest Path?

- Vertices are explored in order of their distance away from the source. So we are guaranteed that the first time we see a vertex we have found the shortest path to it.
- The queue (FIFO) assures that we will explore from all nodes in one level before moving on to the next.
BFS on weighted graphs?

- Run BFS on this weighted graph. What are the weights of the paths that you find to each vertex from v0? Are these the shortest paths?
BFS on weighted graphs?

• 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 \emph{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.
Dijkstra’s Algorithm

Key ideas:
• Vertices are explored similar to previous graph search algos: start with a source and iteratively grow like a mold
• Vertices maintain a “tentative” measure of their distance from the source called: The Dijkstra Score

Nodes have: prev dist done
Dijkstra’s Algorithm

Key ideas:

- Vertices are explored similar to previous graph search algos: start with a source and iteratively grow like a mold
- Vertices maintain a “tentative” measure of their distance from the source called: The Dijkstra Score
- The scores of all vertices that have been explored so far is maintained in some data structure
- Among a choice of candidate vertices, the vertex with the minimum score is picked.
- If a new shorter route is discovered to a vertex, its score is updated and inserted into the data structure storing all the scores
- How do we make progress? Think about the very first iteration
Towards a fast implementation of Dijkstra’s Algorithm

• When the shortest path to a new vertex is discovered, we need to update the Dijkstra scores of all vertices that are connected to the new vertex
• What should happen in the next iteration, once the shortest path to v1 is discovered?
A. The score for v2 should be updated
B. The score for v3 should be updated
C. The score for both v2 and v3 should be updated
D. None of the above
Towards a fast implementation of Dijkstra’s Algorithm

- Which data structure should be used to maintain the Dijkstra scores?
  A. Linked list
  B. Sorted array
  C. Heap
  D. BST
  E. None of the above
Dijkstra’s Algorithm

- Initialize the graph: Give all vertices a dist of INFINITY, set all “done” flags to false.
- Start at s; give s dist = 0 and set prev field to -1.
- Enqueue (s, 0) into a priority queue. This queue contain pairs (v, cost) where cost is the best cost path found so far from s to v. It will be ordered by cost, with smallest cost at the head.
- While the priority queue is not empty or until all shortest paths are discovered:
  - Dequeue the pair (v, c) from the head of the queue.
  - If v’s “done” is true, continue.
  - Else set v’s “done” to true. We have found the shortest path to v. (It’s prev and dist field are already correct).
  - For each of v’s adjacent nodes, w (whose done flag is not true):
    - Calculate the best path cost (also referred to as score), c, to w via v by adding the edge cost for (v, w) to v’s “dist”.
    - If c is less than w’s “dist”, replace w’s “dist” with c, replace its prev by v and enqueue (w, c).
Dijkstra’s Algorithm: Data Structures

• Maintain a sequence (e.g. an array) of vertex objects, indexed by vertex number
  • Vertex objects contain these 3 fields (and others):
    • “dist”: the cost of the best (least-cost) path discovered so far from the start vertex to this vertex
    • “prev”: the vertex number (index) of the previous node on that best path
    • “done”: a boolean indicating whether the “dist” and “prev” fields contain the final best values for this vertex, or not

• Maintain a priority queue
  • The priority queue will contain (pointer-to-vertex, path cost) pairs
  • Path cost is priority, in the sense that low cost means high priority
  • Note: multiple pairs with the same “pointer-to-vertex” part can exist in the priority queue at the same time. These will usually differ in the “path cost” part
Your Turn

The array of vertices, which include dist, prev, and done fields (initialize dist to ‘INFINITY’ and done to ‘false’):

V0: dist= prev= done= adj: (V1,1), (V2,6), (V3,3)
V1: dist= prev= done= adj: (V2,4)
V2: dist= prev= done= adj:
V3: dist= prev= done= adj: (V2,1)

The priority queue (set start vertex dist=0, prev=-1, and insert it with priority 0 to start)
Dijkstra’s Algorithm: Questions

- Initialize the graph: Give all vertices a dist of INFINITY, set all “done” flags to false
- Start at s; give s dist = 0 and set prev field to -1
- Enqueue (s, 0) into a priority queue. This queue contain pairs (v, cost) where cost is the best cost path found so far from s to v. It will be ordered by cost, with smallest cost at the head.
- While the priority queue is not empty:
  - Dequeue the pair (v, c) from the head of the queue.
  - If v’s “done” is true, continue
  - Else set v’s “done” to true. We have found the shortest path to v. (It’s prev and dist field are already correct).
  - For each of v’s adjacent nodes, w:
    - Calculate the best path cost, c, to w via v by adding the edge cost for (v, w) to v’s “dist”.
    - If c is less than w’s “dist”, replace w’s “dist” c and enqueue (w, c)

When a node comes out of the priority queue, how do you know you’ve found the shortest path to the node?