CSE 100 Lecture 18
Graphs
Announcements

- PA3 deadline Monday 8pm
- Midterm 2 next Wednesday
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?
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” $|V|$.

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

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. $|E|^2$

V0: {V1}
V1: {V3, V4}
V2: {V0, V5}
V3: {V2, V5}
V4: {V1, V6}
V5: {}
V6: {V5}
Depth First Search for Graph Traversal

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

Given the graph:

- 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
B. V0, V1, V3, V4, V2
C. V0, V1, V3, V2, V4
D. Other
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
B. V0, V1, V3, V4, V2
C. V0, V1, V3, V2, V4
D. Other
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
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.
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
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. $|V|^2$
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

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Distance (dist)</th>
<th>Previous (prev)</th>
<th>Adjacencies (adj)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0</td>
<td></td>
<td></td>
<td>V1</td>
</tr>
<tr>
<td>V1</td>
<td></td>
<td></td>
<td>V3, V4</td>
</tr>
<tr>
<td>V2</td>
<td></td>
<td></td>
<td>V0, V5</td>
</tr>
<tr>
<td>V3</td>
<td></td>
<td></td>
<td>V2, V5, V6</td>
</tr>
<tr>
<td>V4</td>
<td></td>
<td></td>
<td>V1, V6</td>
</tr>
<tr>
<td>V5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V6</td>
<td></td>
<td></td>
<td>V5</td>
</tr>
</tbody>
</table>

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

**HEAD** [V0] **TAIL**
#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 */
void BFSTraverse( vector<Vertex*> theGraph, int from )
{
    queue<Vertex*> toExplore;
    Vertex* start = theGraph[from];
    // finish the code...
}
/** 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??

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 stack
- While the queue stack is not empty:
  - Dequeue pop the vertex $v$ from the head of the queue top of the stack
  - For each of $v$’s adjacent nodes that has not yet been visited:
    - Mark its distance as 1 + the distance to $v$
    - Enqueue Push it on the queue stack

Stack:

A. BFS  B. DFS  C. Dijkstra’s algorithm  D. Nothing interesting
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
Finding the shortest route from one city to another is a natural application of graph algorithms!

(Of course there are many other examples)
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?

![Graph Diagram]
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 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

- 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)
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
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)

How do you know you’ve found the shortest path to v when you take it off the priority queue?
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)

What is the running time of this algorithm in terms of |V| and |E|? (More than one might be correct—which is tigher?)
A. O(|V|^2)
B. O(|E| + |V|)
C. O(|E| * log(|E|))
D. O(|E| * log(|V|))
E. O(|E|*|V|)
Dijkstra’s Algorithm: Running time

- Each element of each adjacency list can be inserted and deleted from the priority queue; there are $|E|$ such elements.
- An insertion or delete-min in a binary heap implementation of a priority queue is $O(\log N)$; here $N = |E|$ worst-case.
- So, the algorithm has total worst-case time cost $O(|E| \log |E|)$.
- Since $|E| \leq |V|^2$, this is $O(|V|^2 \log |V|)$ and also $O(|E| \log |V|)$.