CMSC 380

Graph Traversals and Search
Graph Traversals

- Graphs can be traversed breadth-first, depth-first, or by path length

- We need to specifically guard against cycles
  - Mark each vertex as “closed” when we encounter it and do not consider closed vertices again
Queuing Function

- Used to maintain a ranked list of nodes that are candidates for expansion

- Substituting different queuing functions yields different traversals/searches:
  - FIFO Queue: breadth first traversal
  - LIFO Stack: depth first traversal
  - Priority Queue: Dijkstra’s algorithm / uniform cost
Bookkeeping Structures

- Typical node structure includes:
  - vertex ID
  - predecessor node
  - path length
  - cost of the path

- Problem includes:
  - graph
  - starting vertex
  - goalTest(Node n) – tests if node is a goal state (can be omitted for full graph traversals)
General Graph Search / Traversal

// problem describes the graph, start vertex, and goal test
// queueingFn is a comparator function that ranks two states
// graphSearch returns either a goal node or failure

graphSearch(problem, queuingFn) {
    open = {}, closed = {}

    queuingFn(open, new Node(problem.startvertex)) //initialize

    loop {
        if empty(open) then return FAILURE  //no nodes remain

        curr = removeFront(open)    //get current node

        if problem.goalTest(curr.vertex)  //optional goaltest
            return curr  //for search

        if curr.vertex is not in closed {  //avoid duplicates
            add curr.vertex to closed

            for each Vertex w adjacent to curr.vertex  // expand node
                queuingFn(open, new Node(w, curr));
        }
    }
}
Unweighted Shortest Path Problem

- Unweighted shortest-path problem: Given an unweighted graph $G = (V, E)$ and a starting vertex $s$, find the shortest unweighted path from $s$ to every other vertex in $G$.

- Breadth first search
  - Use FIFO queue
  - Finds shortest path if edges are unweighted (or equal cost)
  - Recover path by backtracking through nodes
Breadth-First Example: Queue

BFS Traversal

v1 v2 v3 v4
DFS Example: Stack

DFS Traversal: v1, v3, v2, v4
Traversing Performance

What is the performance of DF and BF traversal?

Each vertex appears in the stack or queue exactly once in the worst case. Therefore, the traversals are at least $O(|V|)$. However, at each vertex, we must find the adjacent vertices. Therefore, df- and bf-traversal performance depends on the performance of the `getAdjacent` operation.
GetAdjacent

- **Method 1:** Look at every vertex (except u), asking “are you adjacent to u?”

```c++
List<Vertex> L;
for each Vertex v except u
    if (v.isAdjacentTo(u))
        L.push_back(v);
```

- **Assuming O(1) performance for isAdjacentTo, then getAdjacent has O( |V| ) performance and traversal performance is O( |V^2| )**
GetAdjacent (2)

- Method 2: Look only at the edges which impinge on u. Therefore, at each vertex, the number of vertices to be looked at is \( \text{deg}(u) \), the degree of the vertex.

- For this approach where \( \text{getAdjacent} \) is \( O(\text{deg}(u)) \). The traversal performance is

\[
O \left( \sum_{i=1}^{|V|} \text{deg}(v_i) \right) = O(|E|)
\]

since \( \text{getAdjacent} \) is done \( O(|V|) \) times.

- However, in a disconnected graph, we must still look at every vertex, so the performance is \( O(|V| + |E|) \).
Weighted Shortest Path Problem

Single-source shortest-path problem:
Given as input a weighted graph, \( G = ( V, E ) \), and a distinguished starting vertex, \( s \), find the shortest weighted path from \( s \) to every other vertex in \( G \).

Dijkstra’s algorithm (also called uniform cost search)
- Use a priority queue in general search/traversal
- Keep tentative distance for each vertex giving shortest path length using vertices visited so far.
- Record vertex visited before this vertex (to allow printing of path).
- At each step choose the vertex with smallest distance among the unvisited vertices (greedy algorithm).
Example Network
Dijkstra’s Algorithm

- The pseudo code for Dijkstra’s algorithm assumes the following structure for a Vertex object

```java
class Vertex {
    public List adj;      //Adjacency list
    public boolean known;
    public DisType dist;  //DistType is probably int
    public Vertex path;
    //Other fields and methods as needed
}
```
Dijkstra’s Algorithm

void dijkstra(Vertex start)
{
    for each Vertex v in V {
        v.dist = Integer.MAX_VALUE;
        v.known = false;
        v.path = null;
    }

    start.distance = 0;

    while there are unknown vertices {
        v = unknown vertex with smallest distance
        v.known = true;
        for each Vertex w adjacent to v
            if (!w.known)
                if (v.dist + weight(v, w) < w.distance){
                    decrease(w.dist to v.dist+weight(v, w))
                    w.path = v;
                }
    }
}
Correctness of Dijkstra’s Algorithm

- The algorithm is correct because of a property of shortest paths:
- If $P_k = v_1, v_2, \ldots, v_j, v_k$, is a shortest path from $v_1$ to $v_k$, then $P_j = v_1, v_2, \ldots, v_j$, must be a shortest path from $v_1$ to $v_j$. Otherwise $P_k$ would not be as short as possible since $P_k$ extends $P_j$ by just one edge (from $v_j$ to $v_k$).
- $P_j$ must be shorter than $P_k$ (assuming that all edges have positive weights). So the algorithm must have found $P_j$ on an earlier iteration than when it found $P_k$.
- i.e. Shortest paths can be found by extending earlier known shortest paths by single edges, which is what the algorithm does.
Running Time of Dijkstra’s Algorithm

- The running time depends on how the vertices are manipulated.
- The main ‘while’ loop runs $O(|V|)$ time (once per vertex).
- Finding the “unknown vertex with smallest distance” (inside the while loop) can be a simple linear scan of the vertices and so is also $O(|V|)$. With this method the total running time is $O(|V|^2)$. This is acceptable (and perhaps optimal) if the graph is dense ($|E| = O(|V|^2)$) since it runs in linear time on the number of edges.
- If the graph is sparse, ($|E| = O(|V|)$), we can use a priority queue to select the unknown vertex with smallest distance, using the deleteMin operation ($O(\lg|V|)$). We must also decrease the path lengths of some unknown vertices, which is also $O(\lg|V|)$. The deleteMin operation is performed for every vertex, and the “decrease path length” is performed for every edge, so the running time is $O(|E|\lg|V| + |V|\lg|V|) = O((|V|+|E|)\lg|V|) = O(|E|\lg|V|)$ if all vertices are reachable from the starting vertex.
Dijkstra and Negative Edges

- Note in the previous discussion, we made the assumption that all edges have positive weight. If any edge has a negative weight, then Dijkstra’s algorithm fails. Why is this so?

- Suppose a vertex, u, is marked as “known”. This means that the shortest path from the starting vertex, s, to u has been found.

- However, it’s possible that there is negatively weighted edge from an unknown vertex, v, back to u. In that case, taking the path from s to v to u is actually shorter than the path from s to u without going through v.

- Other algorithms exist that handle edges with negative weights for weighted shortest-path problem.
Directed Acyclic Graphs

- A *directed acyclic graph* is a directed graph with no cycles.

- A *strict partial order* $R$ on a set $S$ is a binary relation such that
  - for all $a \in S$, $aRa$ is false (irreflexive property)
  - for all $a, b, c \in S$, if $aRb$ and $bRc$ then $aRc$ is true (transitive property)

- To represent a partial order with a DAG:
  - represent each member of $S$ as a vertex
  - for each pair of vertices $(a, b)$, insert an edge from $a$ to $b$ if and only if $a \mathrel{R} b$
More Definitions

- Vertex i is a **predecessor** of vertex j if and only if there is a path from i to j.
- Vertex i is an **immediate predecessor** of vertex j if and only if (i, j) is an edge in the graph.
- Vertex j is a **successor** of vertex i if and only if there is a path from i to j.
- Vertex j is an **immediate successor** of vertex i if and only if (i, j) is an edge in the graph.
Topological Ordering

A topological ordering of the vertices of a DAG $G = (V,E)$ is a linear ordering such that, for vertices $i, j \in V$, if $i$ is a predecessor of $j$, then $i$ precedes $j$ in the linear order, i.e. if there is a path from $v_i$ to $v_j$, then $v_i$ comes before $v_j$ in the linear order.
Topological Sort

```java
void topsort( ) throws CycleFoundException
{
    Queue<Vertex> q = new Queue<Vertex>( );
    int counter = 0;

    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter; // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw new CycleFoundException( );
}
```
TopSort Example

1 -> 2 -> 3 -> 4 -> 5
6 -> 7 -> 8 -> 9 -> 10
Running Time of TopSort

1. At most, each vertex is enqueued just once, so there are $O(|V|)$ constant time queue operations.
2. The body of the for loop is executed at most once per edges = $O(|E|)$
3. The initialization is proportional to the size of the graph if adjacency lists are used = $O(|E| + |V|)$
4. The total running time is therefore $O(|E| + |V|)$