Advanced Data Structures and Algorithms

CS 361 – Fall 2013

Lec. #11: Graphs

Tamer Nadeem
Dept. of Computer Science
Class Objective/Overview

• Familiarize with *Graph terminologies and applications*

• Understand *Graph Implementations*

• Familiarize with *Graph ADT*

• Understand *Graph Traverse Methods (DFS & BFS)*

• Understand *Partial Orders and Topological Sort*

• Understand *Shortest Path and Shortest Weighted Path*

• Understand *Minimum Spanning Trees*
Graphs
Graph Overview

• A graph $G$ is a set $V(G)$ of vertices (nodes) and a set $E(G)$ of edges which are pairs of vertices.

• Example: Graph $G = (V, E)$ where
  - $V = \{a, b, c, d, e, f, g, h, i, j, k, l\}$
  - $E = \{(a, b), (a, e), (b, e), (b, f), (c, j), (c, g), (c, h), (d, h), (e, j), (g, k), (g, l), (g, h), (i, j)\}$

• How a graph $G$ is different from a tree $T$?
  - A Tree is just a restricted form of a Graph.
  - Trees are graphs with the restriction that a child can only have one parent.
  - Hence, trees don't contain cycles.

• A subgraph consists of a subset of a graph’s vertices and a subset of its edges
Graph Application

Graphs describe *relationships*, very useful to model many (real-life) problems.

- The Internet
- Streets / Highways (Roadmaps)
- Molecules
- Flow Charts
- Social Networks
- Geometric Surfaces (CAD)
- Circuits
- Parts

... Yoko  John
Graph Terminologies

- A **path** in G is a sequence of vertices \([w_1, w_2, \ldots, w_n]\) such that \((w_i, w_{i+1})\) in E for all \(i = 1 \ldots n-1\).

- A path is **simple path** if no two of its vertices, except possibly the first and last, are the same.

- A **cycle** is a path of length 1 or more for which \(w_1 = w_n\).

- A **simple cycle** has no repeated vertices.

Non-simple path: \([a, b, e, f, g, b, g, l]\)

Simple path: \([a, e, k, p, l, q, m, h, d, c, g]\)

Non-simple cycle: \([k, j, n, k, p, o, k]\)
Graph Connectivity

- A graph is **connected** if each pair of vertices have a path between them. Otherwise, it is **disconnected**.

- If $G$ is not connected, the maximal connected subgraphs are the **connected components** of $G$.

- A **complete graph** is a connected graph in which each pair of vertices are linked by an edge.
Directed Graph (Digraph)

- Edges in previous graph examples do not indicate a direction → **undirected graphs**
- Graph with directed edges are called **directed graphs** or **digraphs**
- **Edges** in diagraph are **ordered pairs**
  - \( V = \{ a, b, c, d, e, f \} \)
  - \( E = \{ (b, a), (b, d), (a, c), (c, b), (d, b), (d, e), (e, b), (e, f), (f, e) \} \)

- A directed graph is **acyclic** if it contains no cycles.
  - Sometimes called a **DAG** for "**directed acyclic graph**".

- **Trees** are **DAGs** with the restriction that a child can only have **one parent**.
**Connectedness of Digraph**

- **Strongly connected** if there is a path from any vertex to any other vertex.
- **Weakly connected** if, for each pair of vertices $v_i$ and $v_j$, there is either a path $P(v_i, v_j)$ or a path $P(v_i, v_j)$.

Not Strongly or Weakly Connected (No path E to D or D to E)  

(a)

Strongly Connected

(b)

Weakly Connected (No path from D to a vertex)

(c)
Edges & Degrees

• A node w is **adjacent** to v in G if there exists an edge (u,v) in E.
• The **degree** of a vertex counts the number of edges.
• When edges have **labels** represent numeric values, graph is called a **weighted graph**.

• Let $G = (V, E)$ be an undirected graph.
  • If $G$ is connected, then $|E| \geq |V| - 1$.
  • If $G$ is a tree, then $|E| = |V| - 1$.
  • $\sum \text{deg}(v) = 2 |E|$, $v \in V$

• If $G$ is directed:
  \[ \sum \text{indeg}(v) = \sum \text{outdeg}(v) = |E| \]
  $v \in V$, $v \in V$
A **strongly connected component** of a graph $G$ is a **maximal set of vertices** in $G$ that are mutually accessible.

The **transpose graph** $G^T$ has the same set of vertices $V$ as graph $G$ but a new edge set $E^T$ consisting of the edges of $G$ but with the **opposite direction**.
Graph Implementation

- One common representation is **Adjacency Matrix**
  - 2-D array of vertices
  - \( M[i][j] = 1 \) if there is an edge between vertex \( i \) and \( j \), 0 otherwise
  - For weighted graph, \( M[i][j] \) is the weight of the edges in this case, -1 is used instead of 0 to indicate no edge.

- The advantage of adjacency matrix is that we can determine adjacency in \( O(1) \) time.
Graph Implementation

• A major disadvantage is the need for $O(|V|^2)$ storage size.
  • This is particularly annoying when the graph is **sparse** (small number of edges $\rightarrow$ small fraction of the adjacency matrix elements are 1).

• Adjacency matrix is preferred when the graph is **small** or **dense**
  • Another common representation is **Adjacency List**

• Adjacency List
  • An array of $V$ linked lists
  • List $i$ represents the vertices connected to vertex $i$
  • For weighted graph, the list stores both neighboring vertices and their edge costs

• The total length of all the adjacency lists:
  • If $G$ is directed $\rightarrow |E|$.
  • If $G$ is undirected $\rightarrow 2|E|$.
Graph Implementation

• The adjacency list
  • is more flexible in terms of storage use, but
  • requires $O(|V|)$ time testing to see if $v_1$ is adjacent to $v_2$
  • makes it easier to iterate over all vertices or all edges
  • Space requirement: $O(|V| + |E|)$.

• In C++, you can implement an adjacency list as an array or vector of std::list.
  • other ways include use of a multimap:
  • or a combination of a map and a set:

```cpp
class Node {
  ...
};

// Use of combination of a map and a set
typedef std::map<Node, std::set<Node> > Graph;
Graph g;

g[node1].insert(node3);
g[node1].insert(node4);
g[node1].insert(node5);
g[node2].insert(node6);
g[node3].insert(node6);
...
```

```cpp
// Use of multimap
typedef std::multimap<Node,Node> Graph;

Graph g;

g.insert(Graph::value_type (node1 , node3));
g.insert(Graph::value_type (node1 , node4));
g.insert(Graph::value_type (node1 , node5));
g.insert(Graph::value_type (node2 , node6));
g.insert(Graph::value_type (node3 , node6));
...
```
Graph Implementation

- Vertex Map and Vector vInfo (text book version)
  - Use a `map<T, int>` container, called `vtxMap`, where a `vertex name` is the `key` of type `T`. The `int field` of a map object is an `index` into a `vector of vertexInfo` objects, called `vInfo`. 

```
vertexInfo object

vtxMapLoc
edges
inDegree
occupied
color
dataValue
distance

vtxMap
vertex
index

vInfo
index

mIter (iterator location)

dest   weight

Adjacency Set

Used to build and use a graph
```
Graph ADT
A Graph ADT

• Your text does not present a general purpose graph ADT. Instead, it works directly with adjacency matrices or adjacency lists.

• Various graph algorithms will be a lot easier to write and understand if we start with a graph ADT that supports the basic style of programming that we use with the C++ std library.

• The ADT for graphs is designed to take advantage of iterators and the iterator-based style of programming that we have seen in dealing with the standard library.

• We won’t worry too much about a specific implementation of this interface, but it can be implemented using either adjacency matrices or adjacency lists, depending on the performance characteristics that we want.
A Graph ADT

- **Vertex**
  - Constructors create a vertex that isn’t part of any graph
  - Compare vertices to see if they are equal (or via a < if we want to place vertices in sets or maps)
  - Get an integer id() that uniquely identifies each node within a given graph.
  - This ID can be used to produce a perfect hash function or even used as indices into an array or vector.

```cpp
#include <iostream>

class DiGraph;

class Vertex{
private:
    const DiGraph* graph;
    unsigned vID;
    Vertex(const DiGraph* g, unsigned theID): graph(g), vID(theID){}
friend class DiGraph ;
friend class AllVertices;
friend class AllEdges;
friend class AllOutgoingEdges;
friend class AllIncidentEdges;
friend class AllIncomingEdges;
friend class Edge ;
public:
    Vertex( ): graph(0), vID(0) { }
    unsigned id() const { return vID; }
    bool operator<( const Vertex& v ) const { return vID < v.vID ; }
    bool operator==( const Vertex& v) const
        { return(graph==v.graph) && (vID == v.vID); }
    bool operator!=( const Vertex& v) const
        { return (graph!=v.graph)|| (vID!=v.vID); }
};
struct VertexHash{
    unsigned operator() (Vertex v) const {return v.id(); }
};
#endif
```
A Graph ADT

- **Edge**
  - Compare edge to other edges or ask for its id().
  - Get the vertex that the edge points from (the source()) and the vertex that it points to (dest(), the destination).

```cpp
#include "vertex.h"

class DiGraph;

class Edge {

private:
    const DiGraph* graph;
    unsigned eID;
    unsigned sourceNode;
    unsigned destNode;

    Edge(const DiGraph* g, unsigned theID, unsigned source, unsigned dest):
        graph(g), eID(theID), sourceNode(source), destNode(dest) {}

    friend class DiGraph;
    friend class Graph;
    friend class AllEdges;
    friend class AllIncidentEdges;
    friend class AllIncomingEdges;
    friend class AllOutgoingEdges;

public:
    Edge() {};
    const Vertex source() const {return Vertex(graph, sourceNode);}
    const Vertex dest() const {return Vertex(graph, destNode);}
    unsigned id() const {return eID;}
    bool operator< (const Edge& e) const {return eID < e.eID; }
    bool operator== (const Edge& e) const
        {return (graph == e.graph) && (eID == e.eID);}
    bool operator != (const Edge& e) const
        {return (graph != e.graph) || (eID != e.eID);}

    struct EdgeHash
        {unsigned operator() (Edge e) const { return e.id(); }}
};
```
A Graph ADT

• **DiGraph**
  • The DiGraph class serves as a container of vertices and edges.
  • Add vertices and edges
  • Remove vertices and edges.
  • Ask how many vertices (numVertices()) and edges (numEdges()) are in the graph.
  • Ask the indegree/outdegree of a vertex v (indegree(v)/outdegree(v)).
  • Ask if v is adjacent to w via isAdjacent(v,w)

```cpp
class DiGraph
{
    public :
        Vertex addVertex();
        void removeVertex(const Vertex& v);
        virtual Edge addEdge(const Vertex& source, const Vertex& dest);
        virtual void removeEdge(const Edge& e);
        unsigned int numVertices() const;
        unsigned int numEdges() const;
        unsigned indegree (Vertex) const;
        unsigned outdegree (Vertex) const;
        virtual bool isAdjacent (const Vertex& v, const Vertex& w) const;
        // Fetch existing edge. Return Edge() if no edge from v to w in graph
        Edge getEdge(const Vertex& v, const Vertex& w) const;
        void clear();
    // iterators
        AllVertices vbegin() const;
        AllVertices vend() const;
        AllEdges ebegin() const;
        AllEdges eend() const;
        AllOutgoingEdges outbegin (Vertex source) const;
        AllOutgoingEdges outend (Vertex source) const;
        AllIncomingEdges inbegin (Vertex dest) const;
        AllIncomingEdges inend (Vertex dest) const;
    protected:
        ...
};
```
A Graph ADT

- **Digraph Iterators**

<table>
<thead>
<tr>
<th>class</th>
<th>starts at</th>
<th>ends at</th>
<th>Iterates over</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllVertices</td>
<td>dg.vbegin()</td>
<td>dg.vend()</td>
<td>all vertices in the graph dg</td>
</tr>
<tr>
<td>AllEdges</td>
<td>dg.ebegin()</td>
<td>dg.eend()</td>
<td>all edges in the graph dg</td>
</tr>
<tr>
<td>AllIncomingEdges</td>
<td>dg.inbegin(v)</td>
<td>dg.inend(v)</td>
<td>all edges in the graph dg that have v as their destination</td>
</tr>
<tr>
<td>AllOutgoingEdges</td>
<td>dg.outbegin(v)</td>
<td>dg.outend(v)</td>
<td>all edges in the graph dg that have v as their source</td>
</tr>
</tbody>
</table>

- For example, to visit every vertex in a graph dg, we would write:
  ```
  for (AllVertices p = dg.vbegin(); p != dg.vend(); ++p)
  {
    Vertex v = *p;
    doSomethingTo(v);
  }
  ```

- To visit every edge emerging from a vertex v, we would write:
  ```
  for (AllOutgoingEdges p = dg.outbegin(v); p != dg.outend(v); ++p)
  {
    Edge e = *p;
    doSomethingElseTo(e);
  }
  ```
## A Graph ADT

### Graph

- A Graph is a DiGraph in which all edges go both ways.
- Its interface is identical to that of a DiGraph. In fact, it is a subclass of DiGraph, so any function that works on DiGraph can be applied to Graph as well.
- However, in Graph when you add an edge \((v,w)\), you automatically get an edge \((w,v)\) as well.
  - easiest way to accomplish this requirement is to modify the addEdge() function to add a pair of directed edges
  - Similarly, removeEdge() need to be modified as well

```cpp
#ifndef GRAPH_H
#define GRAPH_H
#include "digraph.h"

class Graph : public DiGraph
{
    public :
        virtual Edge addEdge(const Vertex& source, const Vertex& dest);
        virtual void removeEdge(const Edge&);
    }

#endif
```

```cpp
Edge Graph::addEdge(const Vertex& source, const Vertex& dest)
{
    DiGraph::addEdge(dest, source);
    return DiGraph::addEdge(source, dest);
}
```
Traversing a Graph
Spanning Trees

• Many problems require us to traverse all the vertices of a graph, or to search for vertices with some desired property.

• We adapt some of the traversal techniques that we learned for trees. This will yield the “depth-first” and “breadth-first” traversals for graphs.

• Every tree is a graph, but not every graph is a tree. However, we can find useful trees embedded within graphs.

• A spanning tree for a connected graph G=(V,E) is a graph G´=(V,E´) such that E´ is subset of E and each edge of E is a directed version of an edge in E, and G´ is a tree.
  
  • The spanning tree is a tree that is "embedded" in the graph.

• A minimum spanning tree (MST) for a weighted undirected graph is a spanning tree with minimum weight.
Graph Search/Traverse

• Consider the problem of searching a general tree for a given node.
  • In a **depth-first search (traversal) - DFS**, we investigate one child’s descendants before exploring its right siblings.
    • Once a possible path is found, continue the search until the end of the path
  • In a **breadth-first search (traversal) - BFS**, we explore all nodes at the same depth before moving on to any deeper nodes.
    • Start several paths at a time, and advance in each one step at a time

• Issues to consider
  • There is no root – **need a start vertex**
  • Be careful and do not enter a repetitive cycle – **mark each vertex as it is processed**
Graph Traversal - Depth-First Search (DFS)

- Initialize the \textit{visited} set
- Identify the initial vertex \(v\) of the Digraph \(dg\)
- Call \textit{depthFirst}(\(dg, v, \text{visited}\))
  - Update visited with \(v\)
  - For each of unvisited neighbor \(w\) of \(v\)
    - \textit{depthFirst}(\(dg, w, \text{visited}\))

```cpp
void depthFirst(Digraph& dg, Vertex v, set<Vertex>& visited) {
    visited.insert(v);
    for (AllOutgoingEdges e = dg.outbegin(v); e != dg.outend(v); ++e) {
        Vertex w = (*e).dest();
        if (visited.count(w) == 0)
            depthFirst(dg, w, visited);
    }
}
```
Graph Traversal - Depth-First Search (DFS)

- 2 is never visited
- DFS search/spanning tree
- Non-recursive version? (using a stack)
Graph Traversal - Depth-First Search (DFS)

• The use of the set will slow this traversal a little bit, though we know that the std::set operations used here are only $O(\log |V|)$.
  
  • We could get even faster average time by using hashing.
  
  • It’s also possible to use the vertex’s id() to index into the vector

• Thus achieving true $O(1)$ time for lookups and amortized $O(1)$ for insertions.

• Hence, the total time complexity for DFS is:
  
  • $O(|V| + |E|)$ when using adjacency list
  
  • $O(|V|^2)$ when using adjacency matrix

• The set of edges at the end of a DFS traversal form a spanning tree of the portion of the graph reachable from the start vertex.

  • The set of edges is called a depth-first spanning tree.
Graph Traversal - Breadth-First Search (BFS)

• Identify the initial vertex \textit{start} of the Digraph \textit{dg}

• \textbf{breadthFirst}(\textit{dg}, \textit{start})
  • Initialize the set \textit{visited} and queue \textit{q}
  • Add \textit{start} to the empty queue \textit{q}
  • Add \textit{start} to the empty set \textit{visited}
  • While(!\textit{q}.empty())
    • \textit{v} = \textit{q}.dequeue() //top+pop/get_front
    • For each unvisited neighbor \textit{w} of \textit{v}
      • If (\textit{w} is not visited)
        • \textit{q}.enqueue(\textit{w}) //push
        • \textit{Visited}.insert(\textit{w})

```cpp
void breadthFirst (const DiGraph& g,
  const Vertex& start) {
  queue<Vertex> q;
  set<Vertex, less<Vertex> > visited;
  q.push (start);
  visited.insert (start);
  while (!q.empty()) {
    Vertex v = q.front();
    q.pop();
    for (AllOutgoingEdges e = g.outbegin(v);
      e != g.outend(v); ++e) {
      Vertex w = (*e).dest();
      if (visited.count(w) == 0) {
        q.push (w);
        visited.insert (w);
      }
    }
  }
}
```
Graph Traversal - Breadth-First Search (BFS)
Graph Traversal - Breadth-First Search (BFS)
Graph Traversal - Breadth-First Search (BFS)

0,1,4,3,5,6

BFS Tree/Spanning Tree
Graph Traversal - Breadth-First Search (BFS)

- The use of the set will slow this traversal a little bit, though we know that the std::set operations used here are only $O(\log |V|)$.
  - We could get even faster average time by using hashing.
  - It’s also possible to use the vertex’s id() to index into the vector.
- Queue operations is in order $O(1)$.
- Thus achieving true $O(1)$ time for lookups and amortized $O(1)$ for insertions.
- Hence, the total time complexity for BFS is:
  - $O(|V| + |E|)$ when using adjacency list
  - $O(|V|^2)$ when using adjacency matrix.
- The set of edges at the end of a BFS traversal form a spanning tree of the portion of the graph reachable from the start vertex.
  - The set of edges is called a breadth-first spanning tree.
Almost *every graph algorithm* is based upon either depth-first or breadth-first search.

*Depth-first* is *easier to program*, does not require an additional ADT (the queue)

*Breadth-first* (or depth-first using an explicit stack) is *slightly faster*

Real choice often *depends upon the nature of the search* and what you are trying to accomplish with your particular algorithm.
Graph Algorithms
Partial Orders and the Topological Sort
Partial Order

• Sometimes we need to arrange things according to a *partial order* for any two elements (e.g., \(a \& b\)):
  - \(a < b\)
  - \(a == b\)
  - \(a > b\)
  - \(a\) is incomparable to \(b\)

• This last possibility distinguishes partial order operations from the more familiar *total order* operations, such as \(<\) on integers,

• The possibility that some pairs of elements may be incomparable to one another makes sorting via a partial order *very different* from conventional sorting.
Partial Order - Example

• Consider the relation among college courses defined as:
  • \( a < b \) if course \( a \) is listed as a prerequisite for \( b \)
  • \( a == b \) if \( a \) and \( b \) are the same course

• For example, in the ODU CS dept.:
  • cs150<cs250, cs170<cs270, cs150<cs381, cs250<cs361, cs361<cs350, cs250<cs355, cs381<cs355, cs381<cs390, cs270<cs471, cs361<cs471

• We can represent this as the graph

• Prerequisite is transitive operation

• When we keep adding items according to a rule until we can not add more, we call that taking the closure of the rule on that set of items

• Transitive closures of orderings are often quite useful.
  • transitive closure of the prerequisite can be used to determine if a student’s planned sequence for taking CS courses is “legal”.

Consider the set of formulas that could be entered into a spreadsheet:

- Spreadsheets store formulas in “cells”.
- Each cell is identified by its column (using letters) and row (using numbers).

A practical problem faced by all spreadsheet implementers is **what order to process the formulas** in.

Given x and y be any two formulas, we define a *partial order* as follows:

- $x < y$ if the left-hand side variable of $x$ appears on the right-hand side of $y$
- $x = y$ if they are the same formula

The graph here captures that partial order. The *transitive closure* of this order defines a “must-be-evaluated-before” relation.

Partial Order - Example

```
<table>
<thead>
<tr>
<th>Variable</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>$10$</td>
</tr>
<tr>
<td>a2</td>
<td>$20$</td>
</tr>
<tr>
<td>a3</td>
<td>$a1 + a2$</td>
</tr>
<tr>
<td>b1</td>
<td>$2*a1$</td>
</tr>
<tr>
<td>b2</td>
<td>$2*a2$</td>
</tr>
<tr>
<td>b3</td>
<td>$b1 + b2 + c3$</td>
</tr>
<tr>
<td>c3</td>
<td>$a3 / a1 / a2$</td>
</tr>
</tbody>
</table>
```
Topological Sort

- A **topological sort** is an **ordered list** of the **vertices** in a **directed acyclic graph** such that:
  - if there is a **path from v to w** in the graph, then **v appears before w** in the list.

- **Example of topological sorts:**
  - Possible sequence in which a student might take classes.
  - An order in which the formulas could be evaluated.

- **TopologicalSort(dg, sorted)**
  - A node of **indegree=0** can be placed at the **start** of the sorted order
    - clearly nothing that must precede it.
    - There must be at least one vertex of indegree = 0.
      - If not, the graph has a cycle and no topological sort exists
  - After placing all nodes of indegree=0 in the list, we then add all nodes whose indegree become zero **without** the edges from the nodes already placed.
  - Repeating this process yields a topological sort.
// A topological sort of a directed graph is any listing of the
// vertices in g such that v1 precedes v2 in the listing only if
// there exists no path from v2 to v1.

// The following routine attempts a topological sort of g. If the
// sort is successful, the return value is true and the ordered
// listing of vertices is placed in sorted. If no topological sort is
// possible (because the graph contains acycle), false is
// returned and sorted will be empty.

bool topologicalSort(const DiGraph& g, list<Vertex>& sorted) {
    // Step 1: get the indegrees of all vertices. Place vertices with
    // indegree 0 into a queue.
    hash_map<Vertex, unsigned, VertexHash> inDegree;
    queue<Vertex> q;
    for (AllVertices v = g.vbegin(); v != g.vend(); ++v) {
        unsigned indeg = g.indegree(*v);
        inDegree[*v] = indeg;
        if (indeg == 0)
            q.push(*v);
    }
    // Step 2. Take vertices from the q, one at a time, and add to
    // sorted. As we do, pretend that we have deleted these vertices
    // from the graph, decreasing the indegree of all adjacent
    // nodes. If any nodes attain an indegree of 0 because of this,
    // add them to the queue.
    while (!q.empty()) {
        Vertex v = q.front();
        q.pop();
        sorted.push_back(v);
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) {
            Vertex adjacent = (*e).dest();
            inDegree[adjacent] = inDegree[adjacent] - 1;
            if (inDegree[adjacent] == 0)
                q.push(adjacent);
        }
    }
    // Step 3: Did we finish the entire graph?
    if (sorted.size() == g.numVertices())
        return true;
    else {
        sorted.clear();
        return false;
    }
}
Topological Sort - Example

q = {B, A}
sorted = {}        q = {A, C}
sorted = {B}        q = {C, D}
sorted = {B, A}

q = {E}
sorted = {B, A, C, D, F}        q = {F, E}
sorted = {B, A, C, D}        q = {D}
sorted = {B, A, C}
Topological Sort - Analysis

• In analyzing this algorithm, we will assume that:
  • The graph is implementing using adjacency lists
  • The indegree map is implemented using a vector-like structure.

• The loop itself goes around once for every vertex in the graph
  • The number of iterations of this loop is $|V|$
  • The entire step 1 loop is $O(|V|)$.

A topological sort of a directed graph is any listing of the vertices in $g$ such that $v_1$ precedes $v_2$ in the listing only if there exists no path from $v_2$ to $v_1$.

The following routine attempts a topological sort of $g$. If the sort is successful, the return value is true and the ordered listing of vertices is placed in sorted. If no topological sort is possible (because the graph contains acycle), false is returned and sorted will be empty.

```cpp
bool topologicalSort(const DiGraph& g, list<Vertex>& sorted) {
  // Step 1: get the indegrees of all vertices. Place vertices with indegree 0 into a queue.
  hash_map<Vertex, unsigned, VertexHash> inDegree;
  queue<Vertex> q;
  for (AllVertices v = g.vbegin(); v != g.vend(); ++v) { // O(|V|)
    unsigned indeg = g.indegree(*v); // O(1)
    inDegree[*v] = indeg; // O(1)
    if (indeg == 0)
      q.push(*v); // O(1)
  }
  // Step 2. Take vertices from the q, one at a time, and add to sorted. As we do, pretend that we have deleted these vertices from the graph, decreasing the indegree of all adjacent nodes. If any nodes attain an indegree of 0 because of this, add them to the queue.
  
  // O(|V|)
  return;
}
```
Topological Sort - Analysis

• What about the number of iterations of inner and outer loops?
  • Each vertex goes into the queue at most once
  • So, in a successful sort, the outer loop will execute $|V|$ times.
  • The inner loop simply visits the edges emanating from the vertex being visited by the outer loop.
  • So, the inner loop will visit every edge in the graph exactly once and hence, executed $|E|$ times.
  • The total cost is $O(|V| + |E|)$.

```c++
// Step 2. Take vertices from the q, one at a time, and add to sorted.
while (!q.empty()) {
    Vertex v = q.front(); // O(1)
    q.pop(); // O(1)
    sorted.push_back(v); // O(1)
    for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) {
        Vertex adjacent = (*e).dest(); // O(1)
        inDegree[adjacent] = inDegree[adjacent] - 1;
        if (inDegree[adjacent] == 0)
            q.push(adjacent); // O(1)
    }
}

// Step 3: Did we finish the entire graph?
if (sorted.size() == g.numVertices())
    return true;
else {
    sorted.clear();
    return false;
}
```
Topological Sort - Analysis

- The only non-trivial operations is clearing the sorted list.
- Since this list is actually a list of vertices that we have successfully sorted, it contains at most $|V|$ elements, and so the clear operation is $O(|V|)$.

- So the total cost of the topological sort is $O(|V| + |E|)$.

- Does the topological sort faster than conventional sort?
  - No, the number of edges can be as high as $|V|^2$.

```c++
// Step 2. Take vertices from the q, one at a time, and add to sorted.
while (!q.empty()) {
    // O(|E| + |V|)
    Vertex v = q.front();
    q.pop();
    sorted.push_back(v);
    for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) {
        Vertex adjacent = (*e).dest();
        inDegree[adjacent] = inDegree[adjacent] - 1;
        if (inDegree[adjacent] == 0)
            q.push(adjacent);
    }
}

// Step 3: Did we finish the entire graph?
if (sorted.size() == g.numVertices()) // O(1)
    return true;
else {
    sorted.clear(); // O(|V|)
    return false;
}
```
Path Finding
Shortest Path

• In the shortest path, the start vertex is 0 steps from the start.

• Given a list of vertices at distance \( k \) steps from the start, build a list of vertices that are \( k+1 \) steps from the start.

• Repeat until the finish vertex is found.

• The shortest-path algorithm includes a queue that indirectly stores the vertices. Each iterative step removes a vertex from the queue and searches its adjacency set to locate all of the unvisited neighbors and add them to the queue.
typedef hash_map<Vertex, unsigned, VertexHash> VDistMap;

void findShortestPath(DiGraph& g, list<Vertex>& path,
                      Vertex start, Vertex finish) {
    hash_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;
    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) {
        dist[*vi] = g.numVertices();
    }
    dist[start] = 0;
    queue<Vertex> visitQueue;
    visitQueue.push(start);

    // From each vertex in queue, update distances of adjacent vertices
    while (!visitQueue.empty() && (dist[finish] != g.numVertices())) {
        Vertex v = visitQueue.front();
        int d = dist[v];
        visitQueue.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) {
            Vertex w = (*e).dest();
            if (dist[w] > d+1) {
                dist[w] = d + 1;
                visitQueue.push(w);
                cameFrom[w] = v;
            }
        }
    }

    // Extract path
    if (dist[finish] != g.numVertices()) {
        Vertex v = finish;
        while (!(v == start)) {
            path.push_front(v);
            path.push_front(start);
        }
    }
}
Shortest Path- Algorithm

• The first loop assigns large distance value (larger than any legitimate path) to each vertex. The exception is that the start vertex is given a distance 0.

• Then we create a queue visitQueue, placing the start vertex into it.

• The main loop removes vertices, one at a time, from the queue. The shortest known distance to that node is saved in d.

• Then each adjacent vertex is examined. If its distance is bigger than \( d+1 \), we now know that we can get there “faster” thru current vertex.
  • We set its distance to \( d+1 \) and place it on the queue

• We keep repeating this until we have computed a new, shortest distance for the finish vertex or until the queue is empty (there’s no way to get from the start vertex to the finish).

• The final loop traces back from the finish to start.
Shortest Path- Example

• Find the shortest path for this graph from F to C.
Shortest Path - Analysis

- Assume an adjacency list implementation of the graph
- Start by labeling everything that is obviously $O(1)$
- Looking at the first loop, we can see that it clearly goes around once for each vertex.
- The entire loop is $|V|*O(1) = O(|V|)$
- Looking at the second loop
  - Same pattern as in the last analysis.
- The inner loop eventually visits every edge in the graph once
  - executed a total $|E|$ times.
- The outer loop is executed once per vertex
  - performed $|V|$ times.
- The entire cost of the two nested loops is $O(|V|+|E|)$. 

```cpp
typedef hash_map<Vertex, unsigned, VertexHash> VDistMap;
void findShortestPath(DiGraph& g, list<Vertex>& path,
                      Vertex start, Vertex finish) {
    hash_map<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;
    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) {
        dist[*vi] = g.numVertices();
    }
    dist[start] = 0;
    queue<Vertex> visitQueue;
    visitQueue.push(start);
    while (!visitQueue.empty() &&
           (dist[finish] != g.numVertices())) {
        Vertex v = visitQueue.front();
        int d = dist[v];
        visitQueue.pop();
        for (AllOutgoingEdges e = g.outbegin(v);
             e != g.outend(v); ++e) {
            Vertex w = (*e).dest();
            if (dist[w] > d+1) {
                dist[w] = d + 1;
                visitQueue.push(w);
                cameFrom[w] = v;
            }
        }
    }
}
```
Shortest Path- Analysis

• The final loop goes around once for every vertex in the shortest path that we have found.
• So this loop may execute as many as $|V|$ times.
• So the entire loop is $O(|V|)$.

Then, the entire algorithm is $O(|V| + |E|)$. 
Weighted Shortest Path

• Now, let’s consider a more general form of the same problem. Attach to each edge a weight indicating the cost of traversing that edge.

• Find a path between designated start and finish nodes that minimizes the sum of the weights of the edges in the path.

• Example: What’s the cheapest airline route way to get from D.C. to N.Y?

• To deal with this problem, we adapt the unweighted shortest path algorithm.
  • Keep nodes in a collection (queue).
  • From the collection, we extracted a node closest to the start.
  • From that node we considered the smallest possible step, updating the distances of the adjacent nodes accordingly.
Dijkstra’s Algorithm

• With weighted graphs, we do the same, but the **step size is determined by the weights**.

• We use a **priority queue** to keep track of the nodes closest to the start.

• We will use a **map** (VDistMap) to associate with each vertex the shortest distance known so far from the start to that vertex.
  
  • This value starts impossibly high, so that any path we find to that vertex will look good by comparison.

• The **CompareVerticesByDistance** structure provides a **comparison operator** based upon this **shortest-distance-known-so-far**.
  
  • We’ll use that operator to maintain a priority queue of the vertices based upon the distance.

• The **edge weights** are passed to our algorithm as another **map**, this one **keyed on edges** instead of vertices.
Dijkstra’s Algorithm

- The algorithm we will develop is called Dijkstra’s algorithm.
- We begin by initializing distance map and priority queue.
- Repeatedly remove from the priority queue the vertex, \( v \), closest to the start.
- We look in the distance map to get that vertex’s minimum distance from the start, \( d \).
- Then we look at each outgoing edge of \( v \) with adjacent vertex \( w \).
- We know we can reach \( w \) with distance \( d+\text{weight}[*e] \).

```cpp
void findWeightedShortestPath (DiGraph& g, list<Vertex>& path, Vertex start, Vertex finish, map_hash<Edge, int, EdgeHash>& weight{
    Map_hash<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;
    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;
    priority_queue <Vertex, VDistMap, CompareVerticesByDistance> pq(g.vbegin(), g.vend(), CompareVerticesByDistance (dist));
    while (!pq.empty()) {
        Vertex v = pq.top();
        int d = dist[v];
        pq.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e ){
            Vertex w = (*e).dest();
            if (dist[w] > d+weight [*e]) {
                dist[w] = d + weight [*e];
                pq.adjust(w);
                cameFrom[w] = v ;
            }
        }
    }
    Vertex v = finish; // Extract path
    if (dist[v] != INT_MAX){
        while (!(v == start)) {
            path.push_front(v);  v = cameFrom[v];
        }
        path.push_front (start);
    }
}
```
Dijkstra’s Algorithm

• It’s possible that exist a shorter way to get to w. So we compare the value d+weight[*e] to the shortest known distance already recorded for w.

• If we already know a shorter way to get to w, we leave it alone.

• If, however, this new distance is shorter, we update the shortest known distance for w
  • adjust its position in the priority queue
  • record that the shortest known way to reach w is via v

• This continues until we have emptied out the priority queue.

```cpp
void findWeightedShortestPath (DiGraph& g, list<Vertex>& path, Vertex start, Vertex finish, map_hash<Edge, int, EdgeHash>& weight{
    Map_hash<Vertex, Vertex, VertexHash> cameFrom;
    VDistMap dist;
    for (AllVertices vi = g.vbegin(); vi != g.vend(); ++vi) {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;
    priority_queue <Vertex, VDistMap, CompareVerticesByDistance> pq(g.vbegin(), g.vend(), CompareVerticesByDistance (dist));
    while (!pq.empty()) {
        Vertex v = pq.top();
        int d = dist[v];
        pq.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e ){
            Vertex w = (*e).dest();
            if (dist[w] > d+weight[*e]) {
                dist[w] = d + weight[*e];
                pq.adjust(w);
                cameFrom[w] = v ;
            }
        }
    }
    Vertex v = finish; // Extract path
    if (dist[v] != INT_MAX){
        while (!(v == start)) {
            path.push_front(v);
            v = cameFrom[v];
        }
        path.push_front (start);
    }
}
```
Dijkstra’s Algorithm - Example

• Find the shortest path for this graph from A to D.

```plaintext
minInfo(B,4)  minInfo(C,11)  minInfo(E,4)
```

Priority queue

```plaintext
minInfo(C,10)  minInfo(C,11)  minInfo(E,4)  minInfo(D,12)
```

Priority queue

```plaintext
minInfo(C,10)  minInfo(C,11)  minInfo(D,12)
```

Priority queue

```plaintext
minInfo(D,12)
```

Priority queue
Dijkstra’s Algorithm - Analysis

- Dijkstra’s algorithm is similar to the unweighted shortest path algorithm.
- The introduction of the priority queue makes some of the steps more expensive than in the unweighted case.
- The first and final loops are obvious of $O(|V|)$.
- The declaration an initialization of the priority queue is $O(|V|)$.
- Pop and adjust call on the priority queue are of $O(\log |V|)$.
Dijkstra’s Algorithm - Analysis

- The **inner loop** executes a total of $|E|$ times.
- The **outer loop** executes $|V|$ times.
- So the total cost of adjusting is $O(|E| \cdot \log |V|)$, and the cost of the rest of the two loops is $O(|E|+|V|)$.
- The total is $O(|E|+|V|+|E| \cdot \log |V|)$.
- But $|E| \cdot \log |V| > |E|$
  - $\rightarrow$ this simplifies to $O(|V|+|E| \cdot \log |V|)$.
- The total cost of Dijkstra’s algorithm is $O(|V|+|E| \cdot \log |V|)$.
Minimum Spanning Trees
Minimum Spanning Tree

- We want to find the subgraph of the original graph that:
  - is connected
  - spans (i.e., has every vertex and some edges from) the original graph
  - Involves the edges leading the minimum possible sum of weights
- There will only be one path from any vertex to any other vertex
  - if there were more than one → a redundant connection somewhere.
  - Any connected graph with that property is a tree known as the “minimum spanning tree”

Network of Hubs

Minimum amount of cable = 241
Prim’s Algorithm

- One solution, **Prim’s algorithm**, turns out to be almost identical to Dijkstra’s.
- The main difference is that, **instead of** keeping track of the total **distance to each vertex from the start**, we base our priority queue on the **minimum distance to the vertex from any vertex** that has already been processed.
- Collect edges in **cameFrom** instead of vertices.
- The “**distance**” associated with each vertex is simply the **smallest edge weight** seen so far.

```cpp
void findMinSpanTree (DiGraph& g, set<Edge>& spanTree, Vertex start, hash_map<Edge, int, EdgeHash>& weight) {
    VDistMap dist;
    hash_map<Vertex, Edge, VertexHash> cameFrom;
    for (AllVertices vi = g.vbegin; vi != g.vend; ++vi) {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;
    priority_queue<Vertex, VDistMap, CompareVerticesByDistance>
    pq (g.vbegin(), g.vend(), CompareVerticesByDistance(dist));
    // Collect edges of span tree
    while (!pq.empty()) {
        Vertex v = pq.top();
        int d = dist[v];
        pq.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e){
            Vertex w = (*e).dest();
            if (dist[w] > weight[*e]) {
                dist[w] = weight[*e];
                pq.adjust (w);
                cameFrom[w] = *e;
            }
        }
    }
    for (hash_map<Vertex, Edge, VertexHash>::iterator p =
        cameFrom.begin(); p != cameFrom.end(); ++p)
        spanTree.insert ((*p).second);
}
```
Prim’s Algorithm - Example

• At each stage, the algorithm adds a new vertex and an edge that connects the new vertex to the ones in the tree
• The algorithm starts with any initial vertex

![Graph](image)

(a) Spanning tree with vertices A, B
minSpanTreeSize = 2, minTreeWeight = 2

(b) Spanning tree with vertices A, B, D
minSpanTreeSize = 3, minTreeWeight = 7

(a) Spanning tree with vertices A, B, D, C
minSpanTreeSize = 4, minTreeWeight = 14
Prim’s Algorithm - Analysis

• The analysis of Prim’s algorithm is **identical to Dijkstra’s**.

• The total cost of Prim’s algorithm is $O(|V|+|E| \log |V|)$.

```cpp
void findMinSpanTree (DiGraph& g, set<Edge>& spanTree, Vertex start, hash_map<Edge, int, EdgeHash>& weight) {
    VDistMap dist;
    hash_map<Vertex, Edge, VertexHash> cameFrom;
    for (AllVertices vi = g.vbegin; vi != g.vend; ++vi) {
        dist[*vi] = INT_MAX;
    }
    dist[start] = 0;
    priority_queue<Vertex, VDistMap, CompareVerticesByDistance> pq (g.vbegin, g.vend, CompareVerticesByDistance(dist));
    // Collect edges of span tree
    while (!pq.empty()) {
        Vertex v = pq.top();
        int d = dist[v];
        pq.pop();
        for (AllOutgoingEdges e = g.outbegin(v); e != g.outend(v); ++e) {
            Vertex w = (*e).dest();
            if (dist[w] > weight[*e]) {
                dist[w] = weight[*e];
                pq.adjust (w);
                cameFrom[w] = *e;
            }
        }
    }
    for (hash_map<Vertex, Edge, VertexHash>::iterator p = cameFrom.begin(); p != cameFrom.end(); ++p)
        spanTree.insert ((*p).second);
}
```
Questions?
Assignment #8

• Due Thursday Dec 5rd, 11:59pm

• Written Assignment
  • Ford & Topp, Chapter #16:
    - Q.13(a-d) P.1010
    - Q.14 P.1011
    - Q.16 P.1012

• Programming Assignment
  • Ford & Topp, Chapter #16:
    - Q.42 P.1019

In this question, the adjacency set implementation of the graph class is given in file “d_graph.h”. Some of the graph algorithms require the priority queue class (d_pqueue.h) that utilizes the heap class (d_heap.h)
Assignment #8

• Due Thursday Dec 5rd, 11:59pm
• Submission Format:
  • Written Assignment
    • Create single PDF file with name: `cs361_assignment_8_<firstName>_<lastName>`
    • Have a cover page with your name and your email
    • Submit through Blackboard.
  • Programming Assignment
    • Make sure your two programs compile and execute using `g++` on Dept’s Linux machines.
    • Create a “Readme.txt” file for each program that list how to compile and execute the program. Include your name and your email.
    • Your main file should be named “main.cpp”.
    • You should have a Makefile.
    • Zip all files (.h, .c, Makefile, etc.) for each program and name them `assign8_program` respectively.
    • Submit through Blackboard.
  • Final Submission Materials (2 Files):
    • One PDF for written assignment.
    • One ZIP file for the program