22c:31 Algorithms

Ch5: Graph Traversal

Hantao Zhang

Computer Science Department

http://www.cs.uiowa.edu/~hzhang/c31
Graphs

Graphs are one of the unifying themes of computer science. A graph $G = (V, E)$ is defined by a set of vertices $V$, and a set of edges $E$ consisting of ordered or unordered pairs of vertices from $V$. 
In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges.
Flavors of Graphs

The first step in any graph problem is determining which flavor of graph you are dealing with. Learning to talk the talk is an important part of walking the walk. The flavor of graph has a big impact on which algorithms are appropriate and efficient.
Directed vs. Undirected Graphs

A graph $G = (V, E)$ is undirected if edge $(x, y) \in E$ implies that $(y, x)$ is also in $E$.

Road networks *between* cities are typically undirected. Street networks *within* cities are almost always directed because of one-way streets. Most graphs of graph-theoretic interest are undirected.
Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of $G$ is assigned a numerical value, or weight.

The edges of a road network graph might be weighted with their length, drive-time or speed limit. In *unweighted* graphs, there is no cost distinction between various edges and vertices.
Simple vs. Non-simple Graphs

Certain types of edges complicate the task of working with graphs. A *self-loop* is an edge \((x, x)\) involving only one vertex.

An edge \((x, y)\) is a *multi-edge* if it occurs more than once in the graph.

Any graph which avoids these structures is called *simple*. 
Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.

Graphs are usually sparse due to application-specific constraints. Road networks must be sparse because of road junctions. Typically dense graphs have a quadratic number of edges while sparse graphs are linear in size.
Cyclic vs. Acyclic Graphs

An \textit{acyclic} graph does not contain any cycles. \textit{Trees} are connected acyclic \textit{undirected} graphs.

Directed acyclic graphs are called \textit{DAGs}. They arise naturally in scheduling problems, where a directed edge $(x, y)$ indicates that $x$ must occur before $y$. 
Implicit vs. Explicit Graphs

Many graphs are not explicitly constructed and then traversed, but built as we use them.

A good example arises in backtrack search.
Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique name or identifier to distinguish it from all other vertices.

An important graph problem is *isomorphism testing*, determining whether the topological structure of two graphs are in fact identical if we ignore any labels.
The Friendship Graph

Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.

This graph is well-defined on any set of people: SUNY SB, New York, or the world.

What questions might we ask about the friendship graph?
If I am your friend, does that mean you are my friend?

A graph is undirected if \((x, y)\) implies \((y, x)\). Otherwise the graph is directed. The “heard-of” graph is directed since countless famous people have never heard of me!
Am I my own friend?

An edge of the form \((x, x)\) is said to be a *loop*. If \(x\) is \(y\)’s friend several times over, that could be modeled using *multiedges*, multiple edges between the same pair of vertices. A graph is said to be *simple* if it contains no loops and multiple edges.
Am I linked by some chain of friends to the President?

A path is a sequence of edges connecting two vertices. Since *Mel Brooks* is my father’s-sister’s-husband’s cousin, there is a path between me and him!

```
Steve                  Dad              Aunt Eve        Uncle Lenny       Cousin Mel
```
How close is my link to the President?

If I were trying to impress you with how tight I am with Mel Brooks, I would be much better off saying that Uncle Lenny knows him than to go into the details of how connected I am to Uncle Lenny. Thus we are often interested in the shortest path between two nodes.
Is there a path of friends between any two people?

A graph is *connected* if there is a path between any two vertices.
A directed graph is *strongly connected* if there is a directed path between any two vertices.
**Who has the most friends?**

The *degree* of a vertex is the number of edges adjacent to it.
Data Structures for Graphs: Adjacency Matrix

There are two main data structures used to represent graphs. We assume the graph $G = (V, E)$ contains $n$ vertices and $m$ edges.

We can represent $G$ using an $n \times n$ matrix $M$, where element $M[i, j]$ is, say, 1, if $(i, j)$ is an edge of $G$, and 0 if it isn’t. It may use excessive space for graphs with many vertices and relatively few edges, however. Can we save space if (1) the graph is undirected? (2) if the graph is sparse?
Adjacency Lists

An adjacency list consists of a $N \times 1$ array of pointers, where the $i$th element points to a linked list of the edges incident on vertex $i$.

To test if edge $(i, j)$ is in the graph, we search the $i$th list for $j$, which takes $O(d_i)$, where $d_i$ is the degree of the $i$th vertex. Note that $d_i$ can be much less than $n$ when the graph is sparse. If necessary, the two copies of each edge can be linked by a pointer to facilitate deletions.
public class Vertex implements Comparable<Vertex> {
    public String name; // label
    // the next four variables for search
    public boolean processed;
    public boolean discovered;
    public Vertex predecessor; // previous vertex
    public int distance; // distance from source to it

    public Vertex(String v) { name = v; distance = -1; }

    public int hashCode() { return name.hashCode(); }

    public String toString() { return name; }
}
Class Vertex

/*
   Compare on the basis of distance from source first and then lexicographically
*/

public int compareTo(Vertex other) {
    int diff = distance - other.distance;
    if (diff != 0) return diff;
    else return name.compareTo(other.name);
}
public class Graph {
    // from name to vertex, using HashMap
    private HashMap<String, Vertex> myVertices;

    /* Adjacent list is a set of vertices represented by a
     * search tree: TreeSet<Vertex>
     * the class Edge is not needed.
     * from vertex to adjacent list, using HashMap */
    private HashMap<Vertex, TreeSet<Vertex>> myAdjList;

    private int myNumVertices;
    private int myNumEdges;

    ...
public Vertex addVertex(String name) {
    Vertex v;
    v = myVertices.get(name);
    if (v == null) {
        v = new Vertex(name);
        myVertices.put(name, v);
        myAdjList.put(v, new TreeSet<Vertex>());
        myNumVertices += 1;
    }
    return v;
}
public Vertex getVertex(String name) {
    return myVertices.get(name);
}

public boolean hasVertex(String name) {
    return myVertices.containsKey(name);
}
public boolean hasEdge (String from, String to) {
    Vertex v = myVertices.get(from);
    Vertex w = myVertices.get(to);
    if (v == null || w == null) return false;
    return myAdjList.get(v).contains(w);
}

public void addEdge(String from, String to) {
    if (hasEdge(from, to)) return;  // no duplicate edges
    myNumEdges += 1;
    Vertex v = addVertex(from);
    Vertex w = addVertex(to);
    myAdjList.get(v).add(w);
    myAdjList.get(w).add(v);  // undirected graph only
}
class Graph

public Iterable<Vertex> adjacentTo(Vertex v) {
    if (!myAdjList.containsKey(v)) return EMPTY_SET;
    return myAdjList.get(v);
}

public Iterable<Vertex> getVertices() {
    return myVertices.values();
}

public int numVertices() { return myNumVertices; }

public int numEdges() { return myNumEdges; }
Tradeoffs Between Adjacency Lists and Adjacency Matrices

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Winner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faster to test if ((x, y)) exists?</td>
<td>matrices</td>
</tr>
<tr>
<td>Faster to find vertex degree?</td>
<td>lists</td>
</tr>
<tr>
<td>Less memory on small graphs?</td>
<td>lists ((m + n)) vs. (n^2)</td>
</tr>
<tr>
<td>Less memory on big graphs?</td>
<td>matrices (small win)</td>
</tr>
<tr>
<td>Edge insertion or deletion?</td>
<td>matrices (O(1))</td>
</tr>
<tr>
<td>Faster to traverse the graph?</td>
<td>lists (m + n) vs. (n^2)</td>
</tr>
<tr>
<td>Better for most problems?</td>
<td>lists</td>
</tr>
</tbody>
</table>

Both representations are very useful and have different properties, although adjacency lists are probably better for most problems.
Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. For *efficiency*, we must make sure we visit each edge at most twice. For *correctness*, we must do the traversal in a systematic way so that we don’t miss anything. Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze.
Marking Vertices

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Each vertex will always be in one of the following three states:

- **undiscovered** – the vertex in its initial, virgin state.
- **discovered** – the vertex after we have encountered it, but before we have checked out all its incident edges.
- **processed** – the vertex after we have visited all its incident edges.
Obviously, a vertex cannot be *processed* before we discover it, so over the course of the traversal the state of each vertex progresses from *undiscovered* to *discovered* to *processed*.
To Do List

We must also maintain a structure containing all the vertices we have discovered but not yet completely explored. Initially, only a single start vertex is considered to be discovered.

To completely explore a vertex, we look at each edge going out of it. For each edge which goes to an undiscovered vertex, we mark it *discovered* and add it to the list of work to do. Note that regardless of what order we fetch the next vertex to explore, each edge is considered exactly twice, when each of its endpoints are explored.
Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.
Suppose not, i.e. there exists a vertex which was unvisited whose neighbor was visited. This neighbor will eventually be explored so we would visit it:
Breadth-First Traversal

The basic operation in most graph algorithms is completely and systematically traversing the graph. We want to visit every vertex and every edge exactly once in some well-defined order. Breadth-first search is appropriate if we are interested in shortest paths on unweighted graphs.
Data Structures for BFS

We use two Boolean arrays to maintain our knowledge about each vertex in the graph. A vertex is discovered the first time we visit it. A vertex is considered processed after we have traversed all outgoing edges from it. Once a vertex is discovered, it is placed on a FIFO queue. Thus the oldest vertices / closest to the root are expanded first.

bool processed[MAXV];
bool discovered[MAXV];
int parent[MAXV];
public void BFS(Vertex s) {
    this.initSearch();
    s.distance = 0;  s.discovered = true;

    Queue<Vertex> q = new LinkedList<Vertex>();
    q.add(s);
    while (!q.isEmpty()) {
        Vertex v = q.remove();
        System.out.println("visit " + v);
        for (Vertex w : this.adjacentTo(v)) if (!w.discovered) {
            w.distance = v.distance+1;
            w.discovered = true;
            w.predecessor = v;
            q.add(w);
        }
        v.processed = true;
    }
}
BFS Example
Shortest Paths and BFS

In BFS vertices are discovered in order of increasing distance from the root, so this tree has a very important property. The unique tree path from the root to any node $x \in V$ uses the smallest number of edges (or equivalently, intermediate nodes) possible on any root-to-$x$ path in the graph.
Recursion and Path Finding

We can reconstruct this path by following the chain of ancestors from $x$ to the root. Note that we have to work backward. We cannot find the path from the root to $x$, since that does not follow the direction of the parent pointers. Instead, we must find the path from $x$ to the root.

```c
fin _path(int start, int end, int parents[]) {
    if ((start == end) || (end == -1))
        printf("%d",start);
    else {
        fin _path(start,parents[end],parents);
        printf(" %d",end);
    }
}
```
Connected Components

The connected components of an undirected graph are the separate “pieces” of the graph such that there is no connection between the pieces.

Many seemingly complicated problems reduce to finding or counting connected components. For example, testing whether a puzzle such as Rubik’s cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configuration is connected.

Anything we discover during a BFS must be part of the same connected component. We then repeat the search from any undiscovered vertex (if one exists) to define the next component, until all vertices have been found:
Two-Coloring Graphs

The *vertex coloring* problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color.

A graph is *bipartite* if it can be colored without conflict while using only two colors. Bipartite graphs are important because they arise naturally in many applications.

For example, consider the “dating” graph in a heterosexual world. Men date only with women, and vice versa. Thus gender defines a legal two-coloring.
Problem of the Day

Prove that in a breadth-first search on a undirected graph $G$, every edge in $G$ is either a tree edge or a cross edge, where a cross edge $(x, y)$ is an edge where $x$ is neither is an ancestor or descendent of $y$. 
The *Key Idea with DFS*

A depth-first search of a graph organizes the edges of the graph in a precise way.

In a DFS of an undirected graph, we assign a direction to each edge, from the vertex which discover it:
public void DFS(Vertex s) {
    this.initSearch();
    s.distance = 0;  s.discovered = true;
    recDFS(s);
}

public void recDFS(Vertex v) {
    System.out.println("visit "+ v);
    for (Vertex w : this.adjacentTo(v)) if (!w.discovered) {
        w.distance = v.distance+1;
        w.discovered = true;
        w.predecessor = v;
        recDFS(w);
    }
    v.processed = true;
}
Edge Classificatio for DFS

Every edge is either:

1. A Tree Edge

2. A Back Edge to an ancestor

3. A Forward Edge to a descendant

4. A Cross Edge to a different node

On any particular DFS or BFS of a directed or undirected graph, each edge gets classified as one of the above.
DFS: Tree Edges and Back Edges Only

The reason DFS is so important is that it defines a very nice ordering to the edges of the graph. In a DFS of an undirected graph, every edge is either a tree edge or a back edge.

Why? Suppose we have a forward edge. We would have encountered \((4, 1)\) when expanding 4, so this is a back edge.
No Cross Edges in DFS

Suppose we have a cross-edge (2, 5)

When expanding 2, we would discover 5, so the tree would look like:
DFS Application: Finding Cycles

Back edges are the key to finding a cycle in an undirected graph. Any back edge going from $x$ to an ancestor $y$ creates a cycle with the path in the tree from $y$ to $x$.

```c
process-edge(int x, int y)
if (parent[x] != y) { (* found back edge! *)
    printf("Cycle from %d to %d:",y,x);
    path(y,x,parent);
    finishe = TRUE;
}
```
Articulation Vertices

Suppose you are a terrorist, seeking to disrupt the telephone network. Which station do you blow up?

An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph. Clearly connectivity is an important concern in the design of any network. Articulation vertices can be found in $O(n(m+n))$ – just delete each vertex to do a DFS on the remaining graph to see if it is connected.
A Faster $O(n + m)$ DFS Algorithm

In a DFS tree, a vertex $v$ (other than the root) is an articulation vertex iff $v$ is not a leaf and some subtree of $v$ has no back edge incident until a proper ancestor of $v$.

The root is a special case since it has no ancestors.

X is an articulation vertex since the right subtree does not have a back edge to a proper ancestor.

Leaves cannot be articulation vertices.
Topological Sorting

A directed, acyclic graph has no directed cycles.

A topological sort of a graph is an ordering on the vertices so that all edges go from left to right. DAGs (and only DAGs) has at least one topological sort (here $G, A, B, C, F, E, D$).
Topological sorting is often useful in scheduling jobs in their proper sequence. In general, we can use it to order things given precedence constraints.
Example: Building a house.
Example: Identifying errors in DNA fragment assembly

Certain fragments are constrained to be to the left or right of other fragments, unless there are errors.

```
A B R A C
A C A D A
A D A B R
D A B R A
R A C A D
```

```
A B R A C A D A B R A
A B R A C
R A C A D
A C A D A
A D A B R
D A B R A
```

Solution – build a DAG representing all the left-right constraints. Any topological sort of this DAG is a consistent ordering. If there are cycles, there must be errors.
Topological Sorting via DFS

A directed graph is a DAG if and only if no back edges are encountered during a depth-first search.
Labeling each of the vertices in the reverse order that they are marked *processed* find a topological sort of a DAG.
Why? Consider what happens to each directed edge \( \{x, y\} \) as we encounter it during the exploration of vertex \( x \):
Case Analysis

- If $y$ is currently *undiscovered*, then we then start a DFS of $y$ before we can continue with $x$. Thus $y$ is marked *completed* before $x$ is, and $x$ appears before $y$ in the topological order, as it must.

- If $y$ is *discovered* but not *completed*, then $\{x, y\}$ is a back edge, which is forbidden in a DAG.

- If $y$ is *completed*, then it will have been so labeled before $x$. Therefore, $x$ appears before $y$ in the topological order, as it must.
Strongly Connected Components

A directed graph is strongly connected iff there is a directed path between any two vertices. The strongly connected components of a graph is a partition of the vertices into subsets (maximal) such that each subset is strongly connected.

Observe that no vertex can be in two maximal components, so it is a partition.
There is an elegant, linear time algorithm to find the strongly connected components of a directed graph using DFS which is similar to the algorithm for biconnected components.
Backtracking and Depth-First Search

Depth-first search uses essentially the same idea as backtracking. Both involve exhaustively searching all possibilities by advancing if it is possible, and backing up as soon as there is no unexplored possibility for further advancement. Both are most easily understood as recursive algorithms.
2-51. [7] Six pirates must divide $300 dollars among themselves. The division is to proceed as follows. The senior pirate proposes a way to divide the money. Then the pirates vote. If the senior pirate gets at least half the votes he wins, and that division remains. If he doesn’t, he is killed and then the next senior-most pirate gets a chance to do the division. Now you have to tell what will happen and why (i.e., how many pirates survive and how the division is done)? All the pirates are intelligent and the first priority is to stay alive and the next priority is to get as much money as possible.

2-52. [7] Reconsider the pirate problem above, where only one indivisible dollar is to be divided. Who gets the dollar and how many are killed?