November 6, 2016

Graphs 1: Representation, Search, and Traverse

This notes chapter begins with a review of graph and digraph concepts and terminology and introduces the *adjacency matrix* and *adjacency list* representation systems. Then we study the depth-first and breadth-first search processes and elaborations on them, including DFS and BFS traversals and the ultimate Survey algorithms.

Outside the Box

Best Lecture Slides	PU	
Best Video	UCB	
Demos	DFSurvey (undirected)	`LIB/area51/fdfs_ug.x`
	DFSurvey (directed)	`LIB/area51/fdfs_dg.x`
	BFSurvey (undirected)	`LIB/area51/fbfs_ug.x`
	BFSurvey (directed)	`LIB/area51/fbfs_dg.x`

Textbook deconstruction. This notes chapter corresponds roughly to Chapter 22 in [Cormen et al 3e], which covers graph (and digraph) concepts and representations and the basic depth- and breadth-first search processes. The notation $G = (V, E)$ is common to the text, the video linked here, and these notes.

1 Definitions, Notation, and Representations

Graphs and directed graphs are studied extensively in the pre-requisit math sequence Discrete Mathematics I, II [FSU::MAD 2104 and FSU::MAD 3105], so these notes will not dwell on details. We will however mention key concepts in order to establish notation and recall familiarity. Where there may be slight variations in terminology, we side with the textbook (see [Cormen::Appendix B.4]). Assume that $G = (V, E)$ is a graph (directed or undirected).

1.1 Undirected Graphs - Theory and Terminology

- Undirected Graph: aka Ungraph, Bidirectional Graph, Bigraph; Graph
- Vertices, vSize $= |V| =$ the number of vertices of G
- Edges, eSize $= |E| =$ the number of edges of G.
- Adjacency: Vertices x and y are said to be *adjacent* iff $[x, y] \in E$.
- Path from v to w: a set $\{x_0, \ldots, x_k\}$ of vertices such that $x_0 = v$, $x_k = w$, and $[x_{i-1}, x_i]$ is an edge for $i = 1, \ldots, k$. The edges $[x_{i-1}, x_i]$ are called the *edges* of the path and k is the *length* of the path. A path is *simple* if the vertices

defining it are distinct - that is, not repeated anywhere along the path. If p is a path from v to w we say w is *reachable* from v.

- Connected Graph: for all vertices $v, w \in V$, w is reachable from v.
- Cycle: A path of length at least 3 from a vertex to itself.
- Acyclic Graph: Graph with no cycles
- Degree of a vertex v: $\deg(v)$ = the number of edges with one end at v.

1.2 Directed Graphs - Theory and Terminology

- Directed Graph: aka Digraph
- Vertices, vSize = $|V|$ = the number of vertices of G
- Edges, eSize = $|E|$ = the number of (directed) edges of G.
- Adjacency: Vertex x is *adjacent to* vertex y, and y is *adjacent from* x, iff $(x, y) \in E$.
- Path from v to w: a set $\{x_0, \ldots, x_k\}$ of vertices such that $x_0 = v$, $x_k = w$, and (x_{i-1}, x_i) is an edge for $i = 1, \ldots, k$. The edges (x_{i-1}, x_i) are called the *edges* of the path and k is the *length* of the path. A path is *simple* if the vertices defining it are distinct - that is, not repeated anywhere along the path. If p is a path from v to w we say w is *reachable* from v. Note: that these edges are all directed and must orient in the forward direction.
- Strongly Connected Digraph: for all vertices $v, w \in V$, w is reachable from v. Note this means the existence of *two* directed paths - one from v to w and the other from w to v. Also note that putting the two paths together creates a (directed) cycle containing both v and w.
- (Directed) Cycle: A path of length at least 1 from a vertex to itself. Note this is distinct from the undirected case, allowing paths of length 1 (so-called self-loops) and length 2 (essentially and out-and-back path, equivalent to a two-way edge).
- Acyclic Digraph: Digraph with no cycles; DAG
- inDegree of a vertex v: inDeg(v) = the number of edges to v.
- outDegree of a vertex v: outDeg(v) = the number of edges from v.

There are subtle distinctions in the way graphs and digraphs are treated in definitions. There are also "conversions" from graph to digraph and digraph to graph.

- Undirected graph edges have no preferred direction and are represented using the notation $[x, y]$ to mean the *un-ordered pair* in which $[x, y] = [y, x]$. Directed graph edges have a preferred direction and are represented using the notation (x, y) to mean the *ordered pair* in which $(x, y) \neq (y, x)$. (In practice, the notation $[x, y]$ is often dropped when context makes it clear whether the edge is directed or undirected or when both directed and undirected graphs are considered in the same context.)

- In an undirected or directed graph, vertex y is said to be *a neighbor of* the vertex x iff $(x, y) \in E$. In undirected graphs, being neighbors and being adjacent are equivalent. In digraphs, y is a neighbor of x iff y is adjacent *from* x. (Note this terminology differs from that in [Cormen et al 3e].)
- If $D = (V, E)$ is a digraph, the *undirected version* of D is the graph $D' = (V, E')$ where $[x, y] \in E'$ iff $x \neq y$ and $(x, y) \in E$. Note this replaces, in effect, a 1-way edge with a 2-way edge (or two opposite 1-way edges in the representations), except self-loops are eliminated.
- If $G = (V, E)$ is a graph, the *directed version* of G is the graph $G' = (V, E')$ where $(x, y) \in E'$ iff $[x, y] \in E$. Note this replaces each undirected edge $[x, y]$ with two opposing directed edges (x, y) and (y, x).

Theorem 1u. In an undirected graph, $\sum_{v \in V} \deg(v) = 2 \times |E|$.
Theorem 1d. In a directed graph, $\sum_{v \in V} \text{inDeg}(v) = \sum_{v \in V} \text{outDeg}(v) = |E|$.

The proof of Theorem 1 uses aggregate analysis. First show the result for directed graphs, where edges are in 1-1 correspondence with their initiating vertices. Then apply 1d to the directed version of an undirected graph and adjust for the double counting.

1.3 Graph and Digraph Representations

There are two fundamental families of representation methods for graphs - connectivity matrices and adjacency sets. Two archetypical examples apply to graphs and digraphs $G = (V, E)$ with the assumption that the vertices are numbered starting with 0 (in C fashion): $V = \{v_0, v_1, v_2, \ldots, v_{n-1}\}$.

1.3.1 Adjacency Matrix Representation

In the *adjacency matrix* representation we define an $n \times n$ matrix M by

$$M(i, j) = \begin{cases} 1 & \text{if } G \text{ has an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

M is the adjacency matrix of G.

Consider the graph `G1` depicted here:

```
                  0 --- 1     2 --- 6 --- 7
Graph G1          |           |           |
                  |           |           |
                  3 --- 4 --- 5 --- 8 --- 9
```

The adjacency matrix for `G1` is:

$$M = \begin{bmatrix}
- & 1 & - & 1 & - & - & - & - & - & - \\
1 & - & - & - & - & - & - & - & - & - \\
- & - & - & - & - & 1 & 1 & - & - & - \\
1 & - & - & - & 1 & - & - & - & - & - \\
- & - & - & 1 & - & 1 & - & - & - & - \\
- & - & 1 & - & 1 & - & - & - & 1 & - \\
- & - & 1 & - & - & - & - & 1 & - & - \\
- & - & - & - & - & - & 1 & - & - & 1 \\
- & - & - & - & - & 1 & - & - & - & 1 \\
- & - & - & - & - & - & - & 1 & 1 & -
\end{bmatrix}$$

(where we have used "$-$" instead of "0" for readability). Adjacency matrix representations are very handy for some graph contexts. They are easy to define, low in overhead, and there is not a lot of data structure programming involved. And some questions have very quick answers:

1. Using an adjacency matrix representation, the question "Is there an edge from v_i to v_j ?" is answered by interpreting $M(i, j)$ as boolean, and hence in time $\Theta(1)$.

Adjacency matrix representations suffer some disadvantages as well, related to the fact that an $n \times n$ matrix has $\Theta(n \times n)$ places to store and visit:

2. An adjacency matrix representation requires $\Theta(n^2)$ storage.
3. A loop touching all edges of a graph (or digraph) represented with an adjacency matrix requires $\Omega(n^2)$ time.

Particularly when graphs are sparse - a somewhat loose term implying that the number of edges is much smaller than it might be - these basic storage and traversal tasks should not be so expensive.

1.3.2 Sparse Graphs

A graph (or digraph) could in principle have an edge from any vertex to any other.[1] In other words the adjacency matrix representation might have 1's almost anywhere, out of a total of $\Theta(n^2)$ possibilities. In practice, most large graphs encountered in

[1]We usually don't allow self-edges in undirected graphs without specifically warning the context that self-edges are allowed.

real life are quite sparse, with vertex degree significantly limited, so that the number of edges is $O(n)$ and the degree of each vertex may even be $O(1)$.[2] Observe:

4. Suppose d is an upper bound on the degree of the vertices of a graph (or digraph). Then $d \times n$ is an upper bound on the number of edges in the graph.

So, a graph with, say, 10,000 vertices, and vertex degree bounded by 100, would have no more than 1,000,000 edges, whereas the adjacency matrix would have storage for 100,000,000, with 99,000,000 of these representing "no edge".[3]

1.3.3 Adjacency List Representation

The *adjacency list* representation is designed to require storage to represent edges only when they actually exist in the graph. There is per-edge overhead required, but for sparse graphs this overhead is negligible compared to the space used by adjacency matrices to store "no info".

The adjacency list uses a vector v of lists reminiscent of the supporting structure for hash tables. The vector is indexed $0, \ldots, n-1$, and the list $v[i]$ consists of the subscripts of vertices that are adjacent from vertex v_i.

For example, the adjacency list representation of the graph G1 illustrated above is:

```
v[0]: 1 , 3
v[1]: 0                  0 --- 1     2 --- 6 --- 7
v[2]: 5 , 6              |           |           |
v[3]: 0 , 4              |           |           |
v[4]: 3 , 5              3 --- 4 --- 5 --- 8 --- 9
v[5]: 2 , 4 , 8                Graph G1
v[6]: 2 , 7
v[7]: 6 , 9
v[8]: 5 , 9
v[9]: 7 , 8
```

As with the matrix representation above, time and space requirements for adjacency list representations are based on what needs to be traversed and stored:

[2]For example, Euler's Formula on planar graphs implies that if G is planar then $|E| \leq 3|V| - 6$, hence $|E| = O(|V|)$.

[3]The human brain can be modeled loosely as a directed graph with vertices representing neuronal cells and edges representing direct communication from one neuron to another. This digraph is estimated to have about 10^{10} vertices with average degree about 10^3.

5. An adjacency list representation requires $\Theta(|V| + |E|)$ storage
6. A loop touching all edges of a graph (or digraph) represented with an adjacency list representation requires $\Omega(|V| + |E|)$ time.

Proofs of the last two factoids use aggregate analysis.

Note that when G has many edges, for example when $|E| = \Theta(|V|^2)$, the estimates are the same as for matrix representations. But for sparse graphs, in particular when $|E| = O(|V|)$, the estimates are dramatically better - linear in the number of vertices.

1.3.4 Undirected v. Directed Graphs

Both the adjacency matrix and adjacency list can represent either directed or undirected graphs. And in both systems, an undirected edge has a somewhat redundant representation.

An adjacency matrix M represents an undirected graph when it is symmetric about the main diagonal:

$$M(i,j) = M(j,i) \quad \text{for all } i,j$$

One could think of this constraint as a test whether the underlying graph is directed or not.

An adjacency list $v[]$ represents an undirected graph when all edges (x, y) appear twice in the lists - once making y adjacent from x and once making x adjacent from y:

```
...
v[x]:  ... , y , ...
...
v[y]:  ... , x , ...
...
```

1.3.5 Variations on Adjacency Lists

Many graph algorithms require examining all vertices adjacent from a given vertex. For such algorithms, traversals of the adjacency lists will be required. These traversals are inherently Ω(list.size), and replacing list with a faster access time set structure might even slow down the traversal (although not affecting its asymptotic runtime).

On the other hand, unlike the situation with hash tables where we could choose the size of the vector to hold the average list size to approximately 2, we have no such luxury in representing graphs.

If a graph process requires many direct queries of the form "is y adjacent from x" (as distinct from traversing the entire adjacency list), a question that requires $\Omega(d)$ time to answer using sequential search in a list of size d, it can be advantageous to replace list with a faster access set structure, which would allow the question to be answered in time $O(\log d)$. (In the graph context, d is the size of the adjacency list and the outDegree of the vertex whose adjacency list is being searched.)

1.3.6 Dealing with Graph Data

Often applications require maintaining data associated with vertices and/or edges. Vertex data is easily maintained using auxilliary vectors. Edge data in an adjacency matrix representation is also straightforward to maintain in another matrix.

Edge data is slightly more difficuly to maintain in an adjacency list representation, in essence because the edges are only implied by the representation. This problem can be handled in two ways. Adjacency lists can be replaced with edge lists - instead of listing adjacent vertices, list pointers to the edges that connect the adjacent pairs. Then an edge can be as elaborate as needed. It must at minimum know what its two vertex ends are, of course, something like this:

```
template <class T>
struct Edge
{
  T data_; // whatever needs to be stored
  unsigned from_, to_;  // vertices of edge
}
```

Or a hash table can be set up to store edge data based on keys of the form `(x,y)`, where `(x,y)` is the edge implied by finding `y` in the list `v[x]`.

1.4 Graph Classes

We now have four distinct representations to enshrine in code: adjacency matrix and adjacency list representations for both undirected and directed graphs. The following class hierarchy provides a plan for defining these four representations while enforcing terminology uniformity and re-using code by elevating to a parent class where appropriate:

```
Graph              // abstract base class
  GraphMatrixBase  // base class for adj matrix reps
    UnGraphMatrix  // undirected adj matrix representation
    DiGraphMatrix  // directed adj matrix representation
  GraphListBase    // base class for adj list reps
    UnGraphList    // undirected adj list representation
    DiGraphList    // directed adj list representation
```

The following pseudo-code provides a plan for implementation:

```
class Graph  // abstract base class for all representations
{
  typedef unsigned Vertex;
public:
  virtual void     SetVrtxSize (unsigned n) = 0;
  virtual void     AddEdge     (Vertex from, Vertex to) = 0;
  virtual bool     HasEdge     ()           const;
  virtual unsigned VrtxSize    ()           const;
  virtual unsigned EdgeSize    ()           const;
  virtual unsigned OutDegree   (Vertex v)   const = 0;
  virtual unsigned InDegree    (Vertex v)   const = 0;
  ...
};

class GraphMatrixBase : public Graph
{
...
  AdjIterator   Begin (Vertex x) const;
  AdjIterator   End   (Vertex x) const;
...
  fsu::Matrix m_;
};

class GraphListBase   : public Graph;
{
...
  AdjIterator   Begin (Vertex x) const;
  AdjIterator   End   (Vertex x) const;
...
  fsu::Vector < fsu::List < Vertex > > v_;
};
```

The `AdjIterator` type needs to be defined for both matrix and list representations. This is a forward iterator traversing the collection of vertices adjacent from `v`. For the adjacency list representation `AdjIterator` is a list ConstIterator. `Begin` can be defined as follows:

```
AdjIterator GraphListBase::Begin (Vertex x)
{ return v_[x].Begin(); }
```

The following implementations should completely clarify the way edges are represented in all four situations:

```
DiGraphMatrix::AddEdge(Vertex x, Vertex y)
{
  M[x][y] = 1;
}

UnGraphMatrix::AddEdge(Vertex x, Vertex y)
{
  M[x][y] = 1;
  M[y][x] = 1;
}

DiGraphList::AddEdge(Vertex x, Vertex y)
{
  v[x].Insert(y);
}

UnGraphList::AddEdge(Vertex x, Vertex y)
{
  v[x].Insert(y);
  v[y].Insert(x);
}
```

2 Search

One of the first things we want to do in a graph or digraph is find our way around. There are two widely used and famous processes to perform a search in a graph, both of which have been introduced and used in other contexts: depth-first and breadth-first search. In trees, for example, preorder and postorder traversals follow the depth-first search process, and levelorder traversal follows the breadth-first process. And solutions to maze problems typically use one or the other to construct a solution path from start to goal. Trees and mazes are representable as special kinds of graphs (or digraphs), and that context provides the ultimate generality to study these fundamental algorithms.

We will assume throughout the remainder of this chapter that $G = (V, E)$ is a graph, directed or undirected, with $|V|$ vertices and $|E|$ edges. We also assume that the graph is presented to the search algorithms using the adjacency list representation.

2.1 Breadth-First Search

The *Breadth-First Search* [BFS] process begins at a vertex of G and explores the graph from that vertex. At any stage in the search, BFS considers all vertices adjacent from the current vertex before proceeding deeper into the graph.

```
Breadth First Search (v)
Uses: double ended control queue conQ
      vector of bool             visited

for each i, visited[i] = false;
conQ.PushBack(v);
visited[v] = true;
// PreProcess(0,v);
while (!conQ.Empty())
{
  f = conQ.Front();
  if (n = unvisited adjacent from f)
  {
    conQ.PushBack(n); // PushFIFO
    visited[n] = true;
    // PreProcess(f,n);
  }
  else
  {
    conQ.PopFront();
    // PostProcess(f);
  }
}
```

This statement of the algorithm shows the control structure only, which uses the control queue and visited flags, with other activities gathered under Pre- and Post-processing of vertices and edges. The PreProcess and PostProcess calls may be modified to suit the target purpose of the search. If a particular goal vertex `g` is sought, PostProcess can check whether `n = g` and return if true. It is usually also desireable to return a solution path, in which case PreProcess can record the information that `f` is the predecessor of `n` in the search.

The runtime of BFS is straightforward to estimate, ignoring the cost of pre and post processing. The aggregate runtime cost breaks down into three categories:

$$\begin{aligned}\text{Total Cost} = \quad & \text{cost of initializing the visited flags} \\ + \quad & \text{cost of the calls to queue push/pop operations} \\ + \quad & \text{cost of finding the unvisited adjacents.}\end{aligned}$$

Initializing the visited flags costs $\Theta(|V|)$. There is one push and one pop operation for each edge in the graph that is accessible from `v`, so that the number of queue operations is no greater than $2|E|$. The third term is dependent on the graph representation, which we assume is the adjacency list. Finding the next unvisited vertex adjacent from `f` requires sequential search of the adjacency list. This search is accomplished using an adjacency iterator that pauses when the next unvisited adjacent is found. The iterator is re-started where it is paused, but never re-initialized. Moreover, a given adjacency list is traversed only one time, when its vertex is at the front of the queue. Therefore the cost of finding unvisited adjacents is the aggregate cost of traversing all of the adjacency lists one time. The aggregate size of adjacency lists is $2|E|$ for graphs and $|E|$ for digraphs, but also all (reachable) vertices must be tested, so the cost of finding all of the next unvisited adjacents is $O(|V| + |E|)$. Therefore the runtime is bounded above by $\Theta(|V|) + O(|E|) + O(|V| + |E|)$:

Theorem 2. The runtime of BFS is $O(|V| + |E|)$.

We cannot conclude the estimate is tight only because not all edges are accessible from a given vertex.

As an example, performing BFS(5) on the graph `G1` encounters the vertices as follows:

```
adj list rep        Graph G1                        BFS::conQ
-------------       ---------                       <---------
v[0]: 1 , 3                                         null          ...
v[1]: 0             0 --- 1       2 --- 6 --- 7     5             6 3 9
v[2]: 5 , 6         |             |           |     5 2           6 3 9 7
v[3]: 0 , 4         |             |           |     5 2 4         3 9 7
v[4]: 3 , 5         3 --- 4 --- 5 --- 8 --- 9     5 2 4 8       3 9 7 0
v[5]: 2 , 4 , 8                                     2 4 8         9 7 0
v[6]: 2 , 7                                         2 4 8 6       7 0
v[7]: 6 , 9                                         4 8 6         0
v[8]: 5 , 9                                         4 8 6 3       0 1
v[9]: 7 , 8                                         8 6 3         1
                                                    8 6 3 9       null
                                                    ...
Vertex discovery order: 5 2 4 8 6 3 9 7 0 1
    grouped by distance: [ (5) (2 4 8) (6 3 9) (7 0) (1) ]
```

2.2 Depth-First Search

Like BFS, the *Depth-First Search* [DFS] process also begins at a vertex of G and explores the graph from that vertex. In contrast to BFS, which considers all adjacent vertices before proceeding deeper into the graph, DFS follows as deep as possible into the graph before backtacking to an unexplored possibility.

```
Depth First Search (v)
Uses: double ended control queue conQ
      vector of bool              visited

for each i, visited[i] = false;
conQ.PushFront(v);
visited[v] = true;
// PreProcess(0,v)};
while (!conQ.Empty())
{
  f = conQ.Front();
  if (n = unvisited adjacent from f)
  {
    conQ.PushFront(n); // PushLIFO
    visited[n] = true;
    // PreProcess(f,n);
  }
  else
  {
    conQ.PopFront();
    // PostProcess(f);
  }
}
```

It is remarkable that the only difference between DFS and BFS is in the way unvisited adjacents are place on the control queue `conQ`. In DFS, `conQ` has LIFO behavior, functioning as a control stack. In BFS, `conQ` has FIFO behavior, functioning as a control queue. The effect is that in DFS, the front of `conQ` is the previously pushed unvisited adjacent vertex, whereas in BFS, the front of `conQ` remains the same after the unvisited adjacent vertex has been pushed.

THEOREM 3. The runtime of DFS is $O(|V| + |E|)$.

The argument is a repeat of that for Theorem 2. Again we cannot conclude the estimate is tight only because not all edges are accessible from a given vertex.

Continuing with the example `G1`, here is a trace of DFS(5). (The orientation in the way we show conQueue is reversed for DFS.)

```
adj list rep        Graph G1                         DFS::conQ
------------        --------                         --------->
v[0]: 1 , 3                                          null          ...
v[1]: 0             0 --- 1      2 --- 6 --- 7       5             5
v[2]: 5 , 6         |            |           |       5 2           5 4
v[3]: 0 , 4         |            |           |       5 2 6         5 4 3
v[4]: 3 , 5         3 --- 4 --- 5 --- 8 --- 9        5 2 6 7       5 4 3 0
v[5]: 2 , 4 , 8                                      5 2 6 7 9     5 4 3 0 1
v[6]: 2 , 7                                          5 2 6 7 9 8   5 4 3 0
v[7]: 6 , 9                                          5 2 6 7 9     5 4 3
v[8]: 5 , 9                                          5 2 6 7       5 4
v[9]: 7 , 8                                          5 2 6         5
                                                     5 2           null
                                                     ...
DFS discovery order: 5 2 6 7 9 8 4 3 0 1  (push order) ``preorder''
DFS finishing order: 8 9 7 6 2 1 0 3 4 5  (pop order) ``postorder''
```

Note that the conQ is illustrated with the front (where push/pop is occurring) to the right, the way a stack is typically illustrated.

2.3 Remarks

It is worth noting the memory use patterns for BFS and DFS. In either case, memory use is $\Theta(|V|)$ plus the maximum size of the set of gray vertices (those currently in the control queue). In the case of BFS, the gray vertices form a "search frontier" roughly a fixed distance away from the starting vertex `v`, growing in diameter (and typically size) as it moves away from `v`. (See Lemma 5 of Section 4.2.) In the case of DFS, the gray vertices represent the current path from `v` to the most recently discovered vertex.

Of course, the sizes of these collections are dependent on the structure of the graph being searched. But typically, the search frontier of BFS can be significantly larger than the longitudinal search path of DFS. This is made quite clear when searching from the root of a binary tree: the BFS frontiers are cross-sections in the tree, approaching $|V|$ in size, whereas the DFS paths are downward paths in the tree, limited to $\log|V|$ for a complete tree.

One can think of DFS as the strategy a single person might use to find a goal in a maze, armed with a way to mark locations that have been visited. The person proceeds to an adjacent unvisited vertex as long as there is one. If at some point there is no unvisited adjacent vertex, the person backtracks along the marked path to a vertex that has an unvisited adjacent and then proceeds. Similarly, BFS might be a strategy employed by a search party of many people. At each vertex, the party splits into sub-groups, one for each unvisited adjacent, and the subgroup proceeds to that adjacent.

This analogy illustrates two important differences between DFS and BFS: (1) BFS discovers a most efficient path to any goal, because the various sub-search-parties operate concurrently and proceed away from the starting vertex. The first party to reach the goal sounds the "found" signal. (2) BFS is more expensive in its use of memory, because there are multiple searches in process concurrently.

3 Survey

The information that can be extracted during one of our basic search algorithms can be very useful. On the other hand, when one has a specific goal such as discovering a path between two vertices, there may be no need for all that information. If you need to hang a picture on a wall, you don't need a professional assessment/survey of your property, only a simple tape measure. But when you want a total evaluation, a survey is called for. For graphs and digraphs, we have two such surveys available: breadth-first and depth-first. Breadth-first survey concentrates on *distance* and depth-first survey concentrates on *time.*

3.1 Breadth-First Survey

It may be past time that we succomb to the temptation to package algorithms as classes. We do that now for the graph surveys. Here is pseudo code for a BFSurvey class:

```
class BFSurvey
{
public:
         BFSurvey ( const Graph& g );
  void   Search   ( );
  void   Search   ( Vertex v );
  void   Reset    ( );
  Vector < int >     distance;  // distance from origin
  Vector < Vertex >  parent;    // for BFS tree
  Vector < Color >   color;     // bookkeeping
private:
  const Graph&      g_;
  Vector < bool >   visited_ ;
  Deque  < Vertex > conQ_   ;
};
```

The class contains a reference to a graph object on which the survey is performed, private data used in the algorithm control, and public variables to house three results of the survey - `distance`, `parent`, and `color` for each vertex in the graph. (These could be privatized with accessors and other trimmings for data security.) These data are instantiated by the survey and have the following interpretation when the survey is completed:

code	**math**	**english**
`distance[x]`	$d(x)$	the number of edges travelled to get from `v` to `x`
`parent[x]`	$p(x)$	the vertex from which `x` was discovered
`color[x]`	$x.color$	either black or white, depending on whether `x` was reachable from `v`

During the course of `Search`, when a vertex is pushed onto the control queue in FIFO order, it is colored gray and assigned distance one more than its parent at the front of the queue. The vertex is colored black when popped from the queue. At any given time during `Search`, the gray vertices are precisely those in the FIFO control queue.

The 1-argument constructor initializes the Graph reference and sets all the various data to the initial/reset state:

```
BFSurvey::BFSurvey (const Graph& g)
  : distance(g.vSize, g.eSize + 1), parent(g.vSize, null),
    color(g.vSize, white), visited_(g.vSize, false),
    g_(g)
{}
```

The Search(v) method is essentially BFSearch(v) with the pre- and post-processing functions defined to maintain the survey data. However, the visited flags and parent information are not automatically unset at the start, so that the method can be called more than once to continue the survey in any parts of the graph that were not reachable from `v`.

```
void BFSurvey::Search( Vertex v )
{
  conQ_.Push(v);
  visited_[v] = true;
  distance[v] = 0;
  color[v]    = grey;
  while (!conQ_.Empty())
  {
    f = conQ_.Front();
    if (n = unvisited adjacent from f in g_)
    {
      conQ_.PushBack(n);  // PushFIFO
      visited_[n] = true;
      distance[n] = distance[f] + 1;
      parent[n]   = f;
      color[n]    = grey;
    }
    else
    {
      conQ_.PopFront();
      color[f] = black;
    }
  }
}
```

The no-argument Search method repeatedly calls Search(v), thus ensuring that the survey considers the entire graph. Often there are relatively few vertices not reached on the first call, but nevertheless Search() perseveres until every vertex has been discovered.

```
void BFSurvey::Search()
{
  Reset();
  for (each vertex v of g_)
  {
    if (color[v] == white) Search(v);
  }
}
void BFSurvey::Reset()
{
  for (each vertex v of g_)
  {
    visited_[v] = 0;
    distance[v] = g_.eSize + 1; // impossibly large
    parent[v] = null;
    color[v] = white;
  }
}
```

We have shown in Theorem 2 of Section 2.1 that BFSearch(v) has runtime $O(|V|+|E|)$, and this bound carries over to BFSurvey::Search(). Note that BFSurvey::Search() touches every edge and vertex in the graph. Therefore we can apply Factoid 6 in Section 1.3.3 to conclude that the bound is tight:

Theorem 4. BFSurvey::Search() has runtime $\Theta(|V|+|E|)$.

3.2 Depth-First Survey

The similarities between depth- and breadth-first search/survey algorithms are far more numerous than the differences. Yet the differences are critical to understanding and using the two. So as tedious as it may be, it is important to concentrate by finding the differences and understanding their consequences. Here is pseudo code for a DFSurvey class:

```
class DFSurvey
{
public:
         DFSurvey ( const Graph& g );
  void   Search   ( );
  void   Search   ( Vertex v );
  void   Reset    ( );
  Vector < unsigned >  dtime;  // discovery time
  Vector < unsigned >  ftime;  // finishing time
  Vector < Vertex >    parent; // for DFS tree
  Vector < Color >     color;  // various uses
private:
  unsigned          time_;
  const Graph&      g_;
  Vector < bool >   visited_ ;
  Deque  < Vertex > conQ_   ;
};
```

The class structure is identical to that of `BFSurvey`, with these exceptions:

(1) The two `DFSurvey::Search` algorithms use `DFS [LIFO]` rather than `BFS [FIFO]`
(2) `DFSurvey` maintains global time used to time-stamp *discovery time* and *finishing time* for each vertex, rather than the distance data of `BFSurvey`.

These time, parent, and color data are instantiated by the survey and have the following interpretation when the survey is completed:

code	**math**	**english**
`dtime[x]`	$t_d(x)$	*discovery time* = time x is first discovered
`ftime[x]`	$t_f(x)$	*finishing time* = time x is released from further investigation
`parent[x]`	$p(x)$	the vertex from which x was discovered
`color[x]`	$x.color$	black or white, depending on whether x was reachable from v

The 1-argument constructor initializes the Graph reference and sets all the various data to the initial/reset state.

During the course of `Search`, vertices are colored gray at discovery time and pushed onto the control queue in LIFO order, and colored black at finishing time and popped from the queue. At any given time during `Search`, the gray vertices are precisely those in the LIFO control queue.

The Search(v) method is DFSearch(v) with the pre- and post-processing functions defined to maintain the survey data. Note however that the visited flags are not automatically unset at the start, so that the method can be called more than once to continue the survey in any parts of the graph that were not reachable from `v`. Global time is incremented immediately after each use, which ensures that no two time stamps are the same.

```
void DFSurvey::Search( Vertex v )
{
  dtime[v] = time_++;
  conQ_.Push(v);
  visited_[v] = true;
  color[v]    = grey;
  while (!conQ_.Empty())
  {
    f = conQ_.Front();
    if (n = unvisited adjacent from f in g_)
    {
      dtime[n] = time_++;
      conQ_.PushFront(n);  // PushLIFO
      visited_[n] = true;
      parent[n]   = f;
      color[n]    = grey;
    }
    else
    {
      conQ_.PopFront();
      color[f] = black;
      ftime[f] = time_++;
    }
  }
}
```

The no-argument Search method repeatedly calls Search(v), thus ensuring that the survey considers the entire graph.

```
void DFSurvey::Search()
{
  Reset();
  for (each vertex v of g_)
  {
    if (color[v] == white) Search(v);
  }
}
void DFSurvey::Reset()
{
  for (each vertex v of g_)
  {
    visited_[v] = 0;
    parent[v] = null;
    color[v] = white;
    dtime[v] = 2|V|;  // last time stamp is 2|V| -1
    ftime[v] = 2|V|;
    time_ = 0;
  }
}
```

The runtime of DFSurvey::Search() succombs to an argument identical to that for BFSurvey::Search():

THEOREM 5. DFSurvey::Search() has runtime $\Theta(|V| + |E|)$.

4 Interpreting and Applying Survey Data

This section is devoted to theory of BFS and DFS, including verification that the BFS tree is a minimal distance tree and two important analytical results on DFS time stamps. We begin by defining the search trees.

4.1 BFS and DFS Trees

Note that for either BFSurvey or DFSurvey, parent informationis collected during the course of the algorithm and stored in the vector `parent[]`. Assume either DFS or BFS context, and suppose we have run Search() - the complete survey.

If parent$[x] \neq$ null, define $p(x) = *$parent$[x] =$ the vertex from which x is discovered, and:

$T(s) = \{(p(x), x) | x \neq s \text{ and } x \text{ is reachable from } s\}$

$V(s) = \{x | x \text{ is reachable from } s\}$

$F = \{(p(x), x) | \text{parent}[x] \neq \text{null}\}$

Lemma 1 (Tree Lemma). After XFSurvey::Search(s), $(V(s), T(s))$ is a tree with root s. If G is a directed graph, the edges of $T(s)$ are directed away from s.

Proof. First note that s is the unique vertex in $T(s)$ with null parent pointer, by inspection of the algorithm. Also note that $(p(x), x)$ is an edge (directed from $p(x)$ to x) in the graph, again by inspection of the algorithm. Following the parent pointers until a null pointer parent[s] = null is reached defines a (directed) path in $T(s)$ from s to x.

Now count the vertices and edges. For each vertex x other than s, $(p(x), x)$ is an edge distinct from any other $(p(y), y)$ because $x \neq y$. Therefore the edges are in 1-1 correspondence with the vertices other than s. We have a connected graph with $|V(s)| = |E(s)| + 1$, so it must be a tree. □

We call $(V(s), T(s))$ the *search tree* generated by the survey starting at s.

Lemma 2 (Forest Lemma). After XFSurvey::Search(), (V, F) is a forest whose trees are all of the search trees generated during the search:

$$F = \cup_s T(s)$$

where the union is over all vertices s with parent[s] = null.

Proof. By the Tree Lemma, $T(s)$ is a tree for each starting vertex s. Suppose some edge in F connects two of these trees, say $T(s_1)$ and $T(s_2)$. The edge must necessarily be of the form $(p(x), x)$ for some x, where $x \in T(s_1)$ and $p(x) \in T(s_2)$. But then the parent-path from x will pass through $p(x)$ to s_2, which means that s_2 should have been discovered by XFSurvey::Search(s_1). □

We call (V, F) the *search forest* generated by the survey.

4.2 Interpreting BFSurvey

We have alluded to the shortest path property of BFS in previous sections. It is time to come to full contact with a proof, and we devote most of the rest of this section to doing that. We follow the proof in [Cormen et al 3e]. For any pair x, y of vertices in G, define the *shortest-path-distance* from x to y to be $\delta(x, y) =$ the length of the shortest path from x to y in G, or $\delta(x, y) = 1 + |E|$ if y is not reachable from x.

LEMMA 3. Let $G = (V, E)$ be a directed or undirected graph and $x, y, z \in V$. If y is reachable from x and z is reachable from y then

$$\delta(x, z) \leq \delta(x, y) + \delta(y, z).$$

Proof. First note that z is reachable from x by concatenating shortest paths from x to y and y to z. This path from x through y to z has length $\delta(x, y)+\delta(y, z)$. The shortest path from x to z can be no longer than this path. Therefore $\delta(x, z) \leq \delta(x, y)+\delta(y, z)$.

Note in passing that $\delta(y, z) = 1$ if $(y, z) \in E$. $\square$

ASSUMPTIONS. For the remainder of this section, let $G = (V, E)$ be a directed or undirected graph and suppose BFSurvey::Search(s) has been called for some starting vertex $s \in V$. Let $d(x) =$ distance[x] for each $x \in V$.

LEMMA 4. The path in $T(x)$ from s to x has length $d(x)$.

Proof. Use mathematical induction on $d(x)$. $\square$

COROLLARY. For each vertex x, $d(x) \geq \delta(s, x)$.

LEMMA 5. If x and y are both gray vertices (i.e., in the control queue) with x colored gray before y (i.e., x pushed before y), then $d(x) \leq d(y) \leq d(x) + 1$.

LEMMA 6. If x and y are reachable from s and x is discovered before y, then $d(x) \leq d(y)$.

Proof. Examine the code to see that when x is pushed onto the control queue, $d(x) = d(p(x)) + 1$ (and at that time $p(x)$ is the front of the queue). Show by mathematical induction that d values are non-decreasing for all vertices in the queue. Because d values are never changed once a vertex is pushed, if x is pushed before y then $d(x) \leq d(y)$. $\square$

LEMMA 7. $d(x) = \delta(s, x)$ for all reachable x.

Proof. Suppose that the result fails. Let δ be the smallest shortest-path-distance for which the result fails, and let y be a vertex for which $d(y) > \delta(s, y) = \delta$. Let x be the next-to-last vertex on a shortest path from s to y. Then $\delta(s, y) = 1 + \delta(s, x)$

and, because of the minimality of $\delta = \delta(s,y)$, $d(x) = \delta(s,x)$. We summarize what we know so far:

$$d(y) > \delta(s,y) = 1 + \delta(s,x) = 1 + d(x)$$

Now consider the three possible colors of y at the times x is at the front of the control queue. If y is white, then y will be pushed onto conQ while x is at the front, making $d(y) = d(x) + 1$, a contradiction. If y is black, it has been popped and $d(y) \le d(x)$ by Lemma 6, again a contradiction. If y is gray, then $d(y) \le d(x) + 1$ by Lemma 4, a contradiction yet again. Therefore under all possibilities our original assumption is false. □

Putting these facts together we have:

Theorem 6 (Breadth-First Tree Theorem). Suppose BFSurvey::Search(s) has been called for the graph or digraph $G = (V, E)$. Then

(1) For each reachable vertex $x \in V$, $d(x)$ is the shortest-path-distance from s to x; and
(2) The breadth-first tree contains a shortest path from s to x.

4.3 Interpreting DFSurvey

The time stamps on vertices during a DFSurvey::Search() provide a way to codify the effects of LIFO order in the control system for DFS. We have already observed that time stamps are unique, and one stamp is used for each change of color of a vertex. Let us define $t_d(x) = \text{dtime}[x]$ and $t_f(x) = \text{ftime}[x]$ for each vertex x. Inspection of the algorithm shows that discovery occurs before finishing:

Lemma 8. For each vertex x, $t_d(x) < t_f(x)$.

Therefore the interval $[t_d(x), t_f(x)]$ represents the time values for which x is in the control LIFO, that is, the times when x has color gray. Prior to $t_d(x)$, x is white, and after $t_f(x)$, x is black.

Theorem 7 (Parenthesis Theorem). Assume $G = (V, E)$ is a (directed or undirected) graph and that DFSurvey::Search() has been run on G. Then for two vertices x and y, exactly one of the following three conditions holds:

(1) The time intervals $[t_d(x), t_f(x)]$ and $[t_d(y), t_f(y)]$ are disjoint, and x and y belong to different trees in the DFS forest.
(2) $[t_d(x), t_f(x)]$ is a subset of $[t_d(y), t_f(y)]$, and x is a descendant of y in the forest.
(3) $[t_d(x), t_f(x)]$ is a superset of $[t_d(y), t_f(y)]$, and x is an ancester of y in the forest.

Proof. First suppose x and y belong to different trees in the forest. Then x is discovered during one call Search(v) and y is discovered during a different call Search(w) where $v \neq w$. Then x is colored gray and then black during Search(v), and y is colored gray and then black during Search(w). Clearly these two processes do not overlap in time, and condition (1) holds.

Suppose on the other hand that x and y are in the same tree in the search forest. Without loss of generality we assume y is a descendant of x. Then, by inspection of the algorithm, x must be colored gray before y. Hence, $t_d(x) < t_d(y)$. But due to the LIFO order of processing, this means that y is colored black before x. Therefore $t_f(y) < t_f(x)$. That is, $[t_d(y), t_f(y)]$ is a subset of $[t_d(x), t_f(x)]$, and condition (3) holds. A symmetric argument completes the proof. $\Box$

THEOREM 8 (WHITE PATH THEOREM). In a depth-first forest of a directed or undirected graph $G = (V, E)$, vertex y is a descendant of vertex x iff at the discovery time $t_d(x)$ there is a path from x to y consisting entirely of white vertices.

Proof. First note that discovery times $t_d(x) =$ `dtime[`x`]` are stamped prior to any processing of x in the DFSurvey::Search algorithm.

First suppose z is a descendant of x. If $z = x$ then $\{x\}$ is a white path. If $z \neq x$ then $t_d(x) < t_d(z)$ by the Parenthesis Theorem, so z is white at time $t_d(x)$. Applying the observation to any z in the DFS tree path from x to y shows that the DFS tree path from x to y consists of white vertices.

Conversely, suppose at time $t_d(x)$ there is a path from x to y consisting entirely of white vertices. If some vertex in this path is not a descendant of x, let v be the one closest to x with this property. Then the predecessor u on the path is a descendant of x. At time $t_d(u)$, v is white and an unvisited adjacent of u, so v will be discovered and $p(v) = u$. That is, v is a descendant of u, and hence of x, contradicting the assumption that v is not a descendant of x. Therefore every vertex on the white path is a descendant of x. $\Box$

4.4 Classification of Edges

The surveys can be used to classify edges of a graph or directed graph. We will use DFSSurvey for this purpose. Given an edge, there are four possibilities: (1) it is in the DFS Forest; it goes from x to another vertex in the same tree, either (2) an ancester or (3) a descendant; or (4) it goes to a vertex that is neither ancester nor descendant, whether in the same or a different tree.

1. ***Tree edges*** are edges in the depth-first forest.
2. ***Back edges*** are edges (x, y) connecting a vertex x in the DFS forest to an ancester y in the same tree of the forest.
3. ***Forward edges*** are edges (x, y) connecting a vertex x in the DFS forest to a descendant y in the same tree of the forest.
4. ***Cross edges*** are any other edges. These might go to another vertex in the same tree or a vertex in a different tree.

For an undirected graph, this classification is based on the first encounter of the edge in the DFSurvey.

Note these observations relating the color of the terminal vertex of an edge to the edge classification. Suppose $e = (x, y)$ is an edge of G, and consider the moment in (algorithm) time when e is explored. Then:

1. If y is **white** then e is a tree edge.
2. If y is **gray** then e is a back edge.
3. If y is **black** then e is a forward or cross edge.

THEOREM 9. In a depth-first survey of an undirected graph G, every edge is either a tree edge or a back edge.

Proof. Let $e = (x, y)$ be an edge of G. Since G is undirected, $e = (y, x)$ as well, so we can assume that x is discovered before y. At time $t_d(x)$, y is white. Suppose e is first explored from x. Then y is white at the time, and hence e becomes a tree edge. If e is first explored from y, then x is gray at the time, and e is a back edge. □

THEOREM 10. A directed graph D contains no directed cycles iff a depth-first search of D yields no back edges.

Proof. If DFS produces a back edge (x, y), adding that edge to the DFS tree path from x to y creates a cycle.

If D has a (directed) cycle C, let y be the first vertex discovered in C, and let (x, y) be the preceding edge in C. At time $t_d(y)$, the vertices of C form a white path from y to x. By the white path theorem, x is a descendant of y, so (x, y) is a back edge. □

5 Spinoffs from BFS and DFS

If theorems have corollaries, do algorithms have cororithms? Maybe, but that is difficult to speak. "Spinoff" is very informal term meaning an extra outcome or simple modification of the algorithm that requires little or no extra verification or anaylsis.

5.1 Components of a Graph

Suppose $G = (V, E)$ is an undirected graph. G is called *connected* iff for every pair $x, y \in V$ of vertices there is a path in G from x to y. A *component* of G is a graph C such that

(1) C is a subgraph of G,
(2) C is connected, and
(3) C is maximal with respect to (1) and (2)

The technology developed in Section 4.3 shows that the following instantiation of the DFS algorithm produces a vector `component<unsigned>` such that `component`[x] is the component containing x for each vertex x of G. All that is needed is to declare the `components` vector and define PostProcess and make a small adjustment to DFSurvey::Search():

```
void DFSurvey::Search()
{
  unsigned components = 0;
  for (each vertex v of g_)
    if (color[v] == white)
    {
      components +=1;
      Search(v);
    }
}
PostProcess(f)
{
  component[f] = components;
}
```

Recall that we know the DFS forest is a collection of trees, each tree generated by a call to `Search(v)`. The DFS trees are in 1-1 correspondence to the components of G.

The algorithm above counts the components and assigns each vertex its component number as it is processed.

This is an algorithm that runs in time $\Theta(|V| + |E|)$ and computes a vector that, subsequently, looks up the component of any vertex in constant time. The algorithm doesn't need vertex color (equate color white with unvisited), only the minimum control structure variables.

5.2 Topological Sort

A directed graph is *acyclic* if it has no (directed) cycles. A directed acyclic graph is called a *DAG* for short. DAGs occur naturally in various places, such as:

vertices	**directed edge**
cells in a spreadsheet	dependancy from another cell
targets in a makefile	dependency on another target
courses in a curriculum	course pre-requisit

In these and other models, it is important to know what order to consider the vertices. For example, courses need to be taken respecting the pre-requisit structure, **make** needs to build the targets in dependency order, and a spreadsheet cell should be calculated only after the cells on which it depends have been calculated. A *topological sort* of a DAG is an ordering of its vertices in such a way that all edges go from lower to higher vertices in the ordering.

DFS can be used to extract a topological sort from a DAG, as follows:

```
TopSort
Modifies DFSurvey
Uses double-ended queue outQ

Run DFSurvey(D)
As a vertex x is finished, outQ.PushFront(x) // LIFO - reverses order

Then outQ [front to back] is a topological sort of D
```

Theorem 10 does the heavy lifting to show the correctness of TopSort.

Exercises

1. Conversions between directed and undirected graphs.
   (a) Consider an undirected graph $G = (V, E)$ represented by either an adjacency matrix or an adjacency list. What changes to the representations are made when G is converted to a directed graph? Explain.
   (b) Consider a directed graph $D = (V, E)$ represented by either an adjacency matrix or an adjacency list. What changes to the representations are made when D is converted to an undirected graph? Explain.
2. Find the appropriate places in the Graph hierarchy to re-define each of the virtual methods named in the Graph base class, and provide the implementations.
3. The way vertices are stored in adjacency lists has an arbitrary effect on the order in which they are processed by DFS and BFS.
   (a) Explain these effects.
   (b) How might the graph edge insertion operations be modified to enforce encountering vertices in numerical order?
4. Prove: During BFSurvey::Search() on a graph G, if x and y are both gray vertices with x colored gray before y, then $d(x) \leq d(y) \leq d(x) + 1$. (This is Lemma 5 above.)
5. Describe 3 other ways to find the components of a graph (other than the algorithm in Section 5.1): (a) Directly from BFS or DFS survey data, (b) Using a BFS or DFS forest, and (c) using a traversal of the graph edge set and Partition / Union-Find.
6. Consider an alternative topological sort algorithm offered first by Donald Knuth:

```
TopSort2
Operates on: Directed Graph D = (V,E)
Uses: vector <unsigned> id indexed on vertices
      FIFO queue conQ
      double-ended queue outQ

for each vertex x, id[x] = inDegree(x);
for each vertex x
  if id[x] = 0
    conQ.Push(x);

While (!conQ.Empty())
{
   f = conQ.Front();
   conQ.Pop();
   for every neighbor n of f
   {
      --id[f];
      if (id[f] == 0)
        conQ.Push(n); // front or back?
   }
   outQ.Push (t);     // front or back?
}
if (outQ.Size() == |V|)
  export outQ as a topological sort of D (front to back)
else
  export ``D has a cycle''
```

(a) Show that TopSort2 produces a topological sort iff D is acyclic.
(b) The two push operations were not specified as FIFO (push back) or LIFO (push front). Which choices will ensure that TopSort2 produces the same topological sort as TopSort?
(c) Use aggregate analysis to derive and verify the worst case runtime for TopSort2.

Software Engineering Projects

7. Develop the graph class hierarchy as outlined in Section 1.4 above. Be sure to provide adjacency iterators facilitating BFS and DFS implementations.
8. Implement BFSurvey and DFSurvey operating on graphs and digraphs via the API provided by the hierarchy above.
9. Develop two classes BFSIterator and DFSIterator that may be used by UnGraphList and DiGraphList. The goal is that these traversal loops are defined for the graph/digraph g:

```
for (BFSIterator i.Initialize(g,v); !i.Finished(); ++i)
{
  std::cout << *i;
}
for (DFSIterator i.Initialize(g,v); !i.Finished(); ++i)
{
  std::cout << *i;
}
```

   and accomplish BFSurvey::Search(v) and DFSurvey::Search(v), respectively, and output the vertex number in discovery order. Of course, the traversals defined with iterators may be stopped, or paused and restarted, in the client program. The iterators should provide access to all of the public survey information.