Lesson 12
Implementing Graphs and Graph Algorithms:
Knowledge Has Organizing Power

Wholeness of the Lesson

The body of knowledge in the field of Graph Theory becomes accessible in a practical way through the Graph Abstract Data Type, which specifies the computations that a Graph software object should support. The Graph Data Type is analogous to the human physiology: The abstract intelligence that underlies life relies on the concrete physiology to find expression in thinking, feeling, and behavior in the physical world.

Operations on Graphs and Efficient Implementation

1. In this lesson we are focus on a useful set of operations on a graph and efficient implementation. Efficiency here will depend both on the algorithm and the implementation.

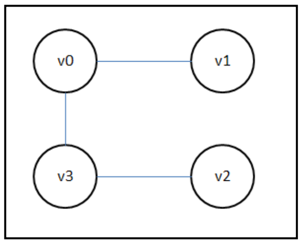
2. *The Graph ADT.* There is no standard set of operations for a graph. We emphasize several that are useful in applications of graphs.

```
boolean areAdjacent(Vertex u, Vertex v)
List getListOfAdjacentVerts (Vertex u)
Graph getSpanningTree()
List getConnectedComponents()
boolean isConnected()
boolean hasPathBetween(Vertex u, Vertex v)
boolean containsCycle()
boolean isTree()
int lengthOfShortestPath(Vertex u, Vertex v)
```

3. Demo: The Graph class.

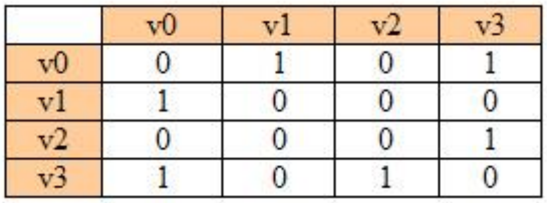
Determining Adjacency

1. Most graph algorithms rely heavily on determining whether two vertices are adjacent and on finding all vertices adjacent to a given vertex. These operations should be as efficient as possible.



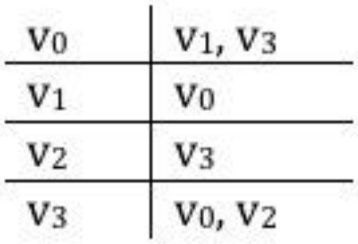
2. *Two ways to represent adjacency.*
   - Adjacency Matrix
   - Adjacency List

3. An *Adjacency Matrix A* is a two-dimensional n x n array consisting of 1's and 0's. A 1 at position *A*[i][j] means that vertices v_i and v_j are adjacent; 0 indicates that the vertices are not adjacent.



	v0	v1	v2	v3
v0	0	1	0	1
v1	1	0	0	0
v2	0	0	0	1
v3	1	0	1	0

4. An *Adjacency List* is a table that associates to each vertex *u* the set of all vertices in the graph that are adjacent to *u*.



V_0	V_1, V_3
V_1	V_0
V_2	V_3
V_3	V_0, V_2

5. Demo: Adjacency in the Graph class.

5. *Advantages and Disadvantages:*

- If there are relatively few edges, an adjacency matrix uses too much space (Example: use a graph to model traffic flow in New York where intersections are represented by vertices and streets are represented by edges.) Best to use adjacency matrix when there are many edges.

- If there are many edges, determining whether two vertices are adjacent becomes costly for an adjacency list. Best to use adjacency list when number of edges is relatively small.

6. Sparse Graphs vs Dense Graphs

- Recall the maximum number of edges in a graph is $n(n - 1)/2$, where n is the number of vertices.

- A graph is said to be *dense* if it has $\theta(n^2)$ edges. It is said to be *sparse* if it has $O(n)$ edges.

- *Strategy*. Use adjacency lists for sparse graphs and adjacency matrices for dense graphs

- For purposes of implementation, in this class we will use adjacency lists. But for purposes of determining optimal running time, we will assume we are using whichever implementation gives the best running time.

Graph Traversal Algorithms

Motivation. Graph traversal algorithms provide efficient procedures for visiting every vertex in a graph. There are many practical reasons for doing this.

- Telephone network – check whether there is a break in the network
- Driving directions – some graph traversal algorithms tell you the shortest path from one vertex to another

Depth-First Search

1. Depth-first search is an example of a graph traversal algorithm. At each vertex, it is possible to do additional processing, but the basic algorithm just shows how to traverse the graph.

2. The basic DFS strategy: Pick a starting vertex and visit an adjacent vertex, and then one adjacent to that one, until a vertex is reached that has no further unvisited adjacent vertices. Then backtrack to one that does have unvisited adjacent vertices, and follow that path. Continue in this way till all vertices have been visited.

Algorithm: Depth First Search (DFS)
Input: A simple connected undirected graph G = (V,E)
Output: G, with all vertices marked as visited.

```
Initialize a stack S  //supports backtracking
Pick a starting vertex s and mark it as visited
S.push(s)
while S ≠ ∅ do
    v ← S.peek()
    if some vertex adjacent to v not yet visited then
        w ← next unvisited vertex adjacent to v
        mark w
        push w onto S
    else //if can't find such a w, backtrack
        S.pop()  //logically, add a vertex w to X, the "pool" of explored vertices
```

Some Implementation Issues

1. It’s useful to represent DFS as a class. Then, other algorithms that make use of the DFS strategy can be reprsented as subclasses. To this end, insert “process” options at various breaks in the code to allow subclasses to perform necessary processing of vertices and/or edges. (This is an application of the *template design pattern*.)

2. Need to decide how to represent the notion that a vertex “has been visited”. Often this is done by creating a special bit field for this purpose in the Vertex class.

   An alternative is to view visiting as being conducted by DFS, so DFS is responsible for tracking visited vertices. Can do this with a hashtable – insert (u,u) whenever u has been visited. Checking whether a vertex has been visited can then be done in O(1) time. (We use this approach.)

3. Code for Efficient Access of Adjacency List

```
  public Vertex nextUnvisitedAdjacent(Vertex v) {
      List<Vertex> listOfAdjacent = adjacencyList.get(v);
      Iterator<Vertex> it = listOfAdjacent.iterator();
      Vertex retVert = null;
      //this loop will visit each element matched with v in the adjacency list
      //ONLY ONCE, since whenever a list element is encountered,
      //it is removed after processing
      while(it.hasNext()) {
         Vertex u = it.next();
         if(visitedVertices.containsKey(u)) {
            it.remove();
         }
        if(!visitedVertices.containsKey(u)) {
            retVert = u;
            it.remove();
            return retVert;
        }
     }
     //unvisited adjacent vertex not found
     return retVert;  //returning null
}
```

4. The adjacency list that is used during DFS will be only a copy of the original.

```
public HashMap<Vertex,LinkedList<Vertex>> getAdjacencyList() {
     HashMap<Vertex,LinkedList<Vertex>> copy = new
                    HashMap<Vertex,LinkedList<Vertex>>();
      for(Vertex v : adjList.keySet()) {
               copy.put(v, getListOfAdjacentVerts(v));

      }
      return copy;
 }
```

5. Demo: DFS class

Running time. It seems at first that the best bound we can claim is O(nm) since, while exploring each vertex, we seem to be searching potentially all edges. Looking more closely, for each v, only the edges incident to v are searched. With this observation, under certain implementation assumptions, we can improve the bound to O(n+m).

- Every vertex eventually is marked and pushed onto the stack, and then is eventually popped from the stack, and each of these occurs only once. Therefore, each vertex undergoes O(1) steps of processing.
- In addition, each vertex v will experience a peek operation, at which time the algorithm will search for an unvisited vertex adjacent to v. This peek step, together with the search, will take place repeatedly until every vertex adjacent to v has been visited – in other words, deg(v) times.

Therefore, for each v, O(1) + O(deg(v)) steps are executed. The sum over all v in V is

$$\mathbf{O(\textstyle\sum_v (1 + deg(v))) = O(n + 2m) = O(n+m)}$$

Disconnected Graphs**.** If a graph is not connected, with components G_1, G_2, . . ., G_k, so that each component G_i has n_i vertices and m_i edges, then running DFS on each component requires $O(n_i + m_i)$ time; summing over all components gives the same result as above: $O(n + m)$ running time. To make sure that, after completing a round of DFS on one component, locating the next unvisited vertex is not too costly, we maintain an iterator for the list of vertices; this ensures that each vertex is accessed only once.

The DFS algorithm for handling possibly disconnected graphs is as follows:

```
Algorithm: Depth First Search (DFS)
Input: A simple undirected graph G = (V,E)
Output: G, with all vertices marked as visited.
  while there are more vertices do   //never reads a vertex twice
      s ← next vertex
      if s is unvisited then
          mark s as visited
          initialize a stack S
          S.push(s)
          while S ≠ ∅ do
              v ← S.peek()
              if some vertex adjacent to v not yet visited then
                    w ← next unvisited vertex adjacent to v
                    mark w
                    push w onto S
              else
                S.pop()
```

Running time. It may seem that since the outer while loop executes n times, the correct calculation for total running time is something like n * (m + n) but in fact, many of the iterations of this outer loop involve only O(1) processing. For each vertex s that is read, if s is unvisited, then it belongs to one of the connected components, say G_i, that has m_i edges and n_i vertices. So total processing that occurs when that vertex is read is $O(m_i + n_i)$. However if s has already been visited, the condition of the if statement fails and processing is O(1). This leads to the following computation of running time:

$$\begin{aligned} &O(1) + O(1) + \ldots + O(m_1 + n_1) + O(1) + \ldots + O(1) + O(m_i + n_i) + \ldots + O(m_k+n_k) \\ &\quad = O(n) + O(m + n) = O(m+n). \end{aligned}$$

Finding a Spanning Tree (Forest) with DFS

1. A spanning tree for a connected graph can be found by using DFS and recording each new edge as it is discovered. Since every vertex is visited, a subgraph is created, in this way, that includes every vertex. No cycle is created because we do not record edges when encountering already visited vertices. If the graph is not connected, the algorithm can be done in each component to create a *spanning forest.* (Example)

2. *Correctness.* Call the collection of recorded edges T. After the initial vertex, each time an unvisited vertex is found, another edge is added to T. This means n-1 edges are added. The only way a cycle could be introduced is if the current vertex is joined to an already visited vertex, completing the cycle. Since this type of step is never performed, a cycle does not arise. Therefore T is acyclic, with n-1 edges, so T is a tree. Each visited vertex is an endpoint of an edge in T (the first edge has two new vertices, each of the others has just one, reaching in this way all n vertices) so T is a spanning tree.

3. *Implementation.* Create a subclass SpanningTree of the DFS class, and implement the “process edge” option so that, as DFS runs, edges are inserted into a list, which is eventually returned.

4. To return the collection of edges as a Graph object (as in the specification), it is necessary to provide a constructor in Graph that transforms a collection of edges into a Graph object.

5. Demo: `FindSpanningTree` class

6. Running Time. The extra steps added to keep track of discovered edges is constant, in each pass through the “componentLoop” (see the code). Therefore, like DFS, running time is O(n + m).

Finding Connected Components

1. Each time DFS completes a round, it completes a search on a single connected component. Therefore, we can track the rounds of DFS and assign numbers to vertices according to the round in which they are discovered. Round 0 vertices belong to component #0, round 1 vertices to component #1, etc.

2. *Implementation.* Create another subclass of the DFS class to handle connected components. During the in-between rounds, increment the counter. Record the component number for each vertex using a hashtable.

   When DFS has completed, the vertices and edges can be organized into graphs and a list of graphs can be returned.

   In practice, it is helpful to insert in the Graph object not only the list of connected components, but also the hashtable that indicates which component each vertex belongs to.

Array of Components	
0	v_1, v_2, v_3, v_4
1	v_5, v_7, v_9
2	v_6, v_8, v_{11}
3	v_{10}

Component-Map

Vertex → Component Number Map	
v_1	0
v_2	0
v_3	0
v_4	0
v_5	1
v_6	2
. . .	. . .

Vertex-Component-Map

Answering Questions About Connectedness and Cycles

1. Once we have obtained the connected components, we can provide answers to questions that may be asked of the graph:

    - Is it connected?
    - Is there a path from given vertices u and v?
    - Does the graph contain a cycle?

2. *Connected.* The graph is connected if and only if the number of connected components is 1. (Running time: O(1), once connected components are known.)

3. *Path.* There is a path from u to v if and only if u and v belong to the same component. (Running time: O(1), once connected components are known.)

4. *Cycle.* Each component is connected. Suppose the components are C_0, C_1, C_2,…, C_k. For each *i*, let n_i be the number of vertices and m_i the number of edges. If for each *i* we have that $m_i = n_i - 1$, then by an earlier theorem, each component is a tree and therefore, the graph contains no cycle.

   On the other hand, if for some *i*, m_i is not equal to $n_i - 1$, then this component must contain a cycle: If not, then this component is connected and acyclic and therefore a tree, and so it must satisfy $m_i = n_i - 1$. Contradiction!

   Therefore, the criterion for existence of a cycle is:

   G has a cycle if and only if there is a connected component C_i of G such that the number n_i of vertices of C_i satisfies $m_i \neq n_i - 1$, where m_i is the number of edges in C_i.

Main Point

To answer questions about the structure of a graph *G*, such as whether it is connected, whether there is a path between two given vertices, and whether the graph contains a cycle, it is sufficient to use DFS to compute the connected components of the graph. Based on this one piece of information, all such questions can be answered efficiently. This phenomenon illustrates the SCI principle of the *Highest First*: Experience the home of all knowledge first, and all particular expressions of knowledge become easily accessible.

Breadth First Search

1. An equally efficient way to traverse the vertices of a graph is to use the *Breadth First Search* (BFS) algorithm.

2. *Idea.* Pick a starting vertex. Visit every adjacent vertex. Then take each of those vertices in turn and visit every one of its adjacent vertices. And so forth.

Algorithm: Breadth First Search (BFS)
Input: A simple connected undirected graph G = (V,E)
Output: G, with all vertices marked as visited.

```
Initialize a queue Q
Pick a starting vertex s and mark s as visited
Q.add(s)
while Q $\neq \emptyset$ do
    v $\leftarrow$ Q.dequeue()   //adds v to X, the “pool” of marked vertices
    for each unvisited w adjacent to v do
        mark w
        Q.add(w)
```

Running Time. Analysis is similar to that for DFS. In the outer loop, 2 steps of processing occur on each vertex v, and then all edges incident to v are examined (some are discarded, others are processed). Therefore, for each v, O(1) + O(deg(v)) steps are executed. The sum is then

$$O(\sum_v (1 + \deg(v))) = O(n + 2m) = O(n+m)$$

Implementation. It is useful to implement as a separate class, like DFS, supporting subclassing by classes that use the BFS template.

Disconnected Graphs. As in the case of DFS, BFS can be run on each component C_i in time $O(n_i+m_i)$ time. Detecting the next unvisited vertex after completion of BFS on each component requires O(1) only, so, as with DFS, total running time to traverse the entire graph is O(n + m).

Finding a Shortest Path

1. A special characteristic of the BFS style of traversing a graph is that, with very little extra processing, it outputs the shortest path between any two vertices in the graph. (There is no straightforward to do this with DFS.)

2. If p: $v_0 – v_1 – v_2 – \ldots – v_n$ is a path in G, recall that its length is n, the number of edges in the path. BFS can be used to compute the shortest path between any two vertices of the graph.

3. As with DFS, the discovered edges during BFS collectively form a spanning tree (assume G is connected, so there is a spanning tree). A tree obtained in this way, starting with a vertex s (the *starting vertex when performing BFS*) is called the *BFS rooted spanning tree.*

4. Recall that a rooted tree can be given the usual *levels.*

5. Given a connected graph G with vertices s and v, Here is the algorithm to return the shortest length of a path in G from s to v

   A. Perform BFS on G starting with s to obtain the BFS rooted tree T with root s, together with a map recording the levels of T
   B. Return the level of v in T

Theorem. For any simple graph G and any vertices s, v in G, the length of the shortest path from s to v is equal to the level to which v belongs in any BFS rooted tree with root s.

Exercise 1. If $v_1 – v_2 – . . . – v_k – v_{k+1}$ is a shortest path from v_1 to v_{k+1}, then whenever $1 \leq i \leq k$, the path $v_1 – v_2 – . . . – v_i$ is a shortest path from v_1 to v_i .

Exercise 2. Suppose v has level i in a BFS rooted tree with root s. Then there is a path from s to v of length i. [Proof is by induction on levels.]

(Optional) **Proof of Theorem**. First build a BFS rooted tree with root s. Proceed by induction to prove the following statement φ(i), for all i:

For every vertex v in G, the shortest path from s to v has length i if and only if the level of v is i.

For i = 0, the only vertex that has shortest path length 0 from s is s itself, which has level 0. Therefore, φ(0) holds.

Assume φ(j) is true for all j < i and let v be a vertex in G. For one direction, assume v has level i. Then there is (by Exercise 2) a path from s to v of length i. Suppose there is also a shorter path, of length j < i, from s to v. By induction hypothesis, this would imply that v is in level j, and this is not the case.

For the other direction, assume there is a shortest path from s to v of length i – denote this path p: s – a – b – . . . – u – v. By Exercise 1, s – a – … – u is a shortest path from s to u, and this path clearly has length i - 1. By the induction hypothesis, lev(u) = i - 1. Since (u,v) is an edge in the tree, there are just two possibilities for the level of v: i – 2 or i (recall that in a BFS rooted tree, there is never an edge between two vertices at the same level). If lev(v) = i – 2, then by the induction hypothesis, there is a shortest path from s to v of length i - 2; but we already know that the shortest path length from s to v is i, so this possibility must be ruled out. It follows therefore that lev(v) = i, as required.

By induction, we have shown that, for every i, φ(i) holds, and this establishes the theorem.

6. <u>*Algorithm for computing shortest path* from s to any vertex v</u>:

(Above, we described an algorithm for obtaining shortest *path length* from s to a vertex v – simply read off the level of v. Here we compute a *shortest path* from s to v)

Perform BFS, beginning with s. Keep track of the levels of vertices as the spanning tree for this component is being built. Also, keep a hashtable associating with each vertex w a shortest path P[w] from s. Initially, P[s] = ∅. When considering edge (v,w), if w is unvisited, set

$$P[w] = P[v] \cup (v,w).$$

Main Point

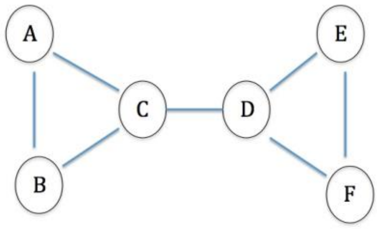
The BFS and DFS algorithms are procedures for visiting every vertex in a graph. BFS proceeds “horizontally”, examining every vertex adjacent to the current vertex before going deeper into the graph. DFS proceeds “vertically”, following the deepest possible path from the starting point and, after reaching the end, backtracks to follow another path to the end starting from some earlier point on the first path, and continues till all vertices have been reached. These approaches to graph traversal are analogous to the horizontal and vertical means of gaining knowledge, as described in SCI: The horizontal approach focuses on a breadth of connections at a more superficial level, and reaches deeper levels of knowledge more slowly. The vertical approach dives deeply to the source right from the beginning; having fathomed the depths, subsequent gain of knowledge in the horizontal direction enjoys the influence of the depths of knowledge already gained.

Special Uses of DFS and BFS

1. Both DFS and BFS can be used to compute connected components and a spanning tree

2. BFS can be used to compute shortest paths.

3. (Optional)DFS can be used to compute the *biconnected components.*



G is *biconnected* if removal of an edge does not disconnect G.

A *biconnected component* is either a separating edge or is a subgraph that is maximal with respect to being biconnected

Biconnected components: A-B-C, C-D, D-E-F

Connecting the Parts of Knowledge With the Wholeness of Knowledge

DETECTING BIPARTITE GRAPHS

1. The BFS algorithm provides an efficient procedure for traversing all vertices in a given graph.

2. By tracking edges and levels during execution of the BFS algorithm, it is possible to detect the presence of an odd cycle (this occurs if an already visited vertex is found which is at the same level as the current vertex and is also adjacent to it). This allows us to determine whether the graph is bipartite.

---

3. *Transcendental Consciousness* is the field of all possibilities, located at the source of thought by an effortless procedure of transcending.

4. *Impulses within the Transcendental field*: The entire structure of the universe is designed in seed form within the transcendental field, all in an effortless manner.

5. *Wholeness moving within itself*: In Unity Consciousness, each expression of the universe is seen as the effortless creation of one's own unbounded nature.