Lesson 13
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

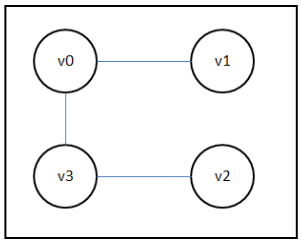
1. In this lesson we are focus on a useful set of operations on a graph and efficient implementation. Efficiency here will depend both on the algorithm and the implementation.

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()

 boolean isBipartite()

int lengthOfShortestPath(Vertex u, Vertex v)
```

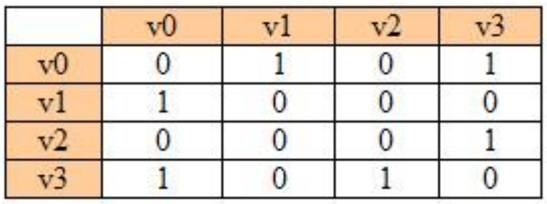
Determining Adjacency

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.



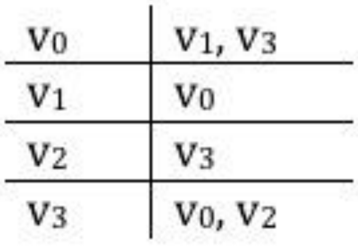
2. *Two ways to represent adjacency.*
   - Adjacency Matrix
   - Adjacency List

3. An *Adjacency Matrix A* is a two-dimensional v x v 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. *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

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
- Erdos number – shortest path tells you where a mathematics researcher is in relation to Erdos
- Problem solving like Sudoku Start with a directed graph; nodes are partially completed games; a directed edge from one vertex to another indicates that after one move, can move from one state of the game to the next
- Connected Components – sample applet

Generic Graph Traversal Algorithm

1. Begin with a <u>connected</u> graph G and a starting vertex s and mark it as visited

2. Build a pool X of vertices that have been visited by adding one new vertex to X at a time

3. while $X \neq V$ do
   - select an edge (v, w) where v is in X and w is not in X
   - mark w as visited
   - add w to X.

<u>Correctness</u>

Fact: The while loop will eventually terminate. Moreover, if there is a path from s to v in G, then the algorithm will find v and mark it as visited.

Proof: If the while loop does not terminate, it means that eventually, the sets X and V – X have the property that V – X is not empty and there is no edge having one endpoint in X and the other in V – X. Let v be any element of V – X. Since G is connected, there is a path p from s to v. Let w be first vertex in the path that belongs to V – X, and let u be its predecessor. Now (u,w) is an edge with u in X, w in V - X. Contradiction! It follows that the while loop terminates. But if the while loop terminates, then the algorithm terminates with X=V; since a vertex is placed in X only if it has been marked as visited, this means that every vertex is eventually found and marked as visited.

Running Time

Decent implementations can arrange so that the number of times any vertex is explored is O(1) and number of times an edge is explored is O(1). Therefore, good implementations of this generic algorithm run in O(n+m). It is not obvious from the generic algorithm shown above how this can be done – with more details for specific algorithms, we will be able to show why this running time can be achieved.

Unconnected Graphs

The algorithm still works when a graph is not connected – it traverses every vertex within any given component. The algorithm can be performed on every component; just need to check, at the completion of the while loop, whether any unvisited vertices remain.

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
```

Correctness. Two points to check:

(1) the while loop terminates
(2) every vertex in G will eventually be marked

Proof of (1) Each pass through the while loop accomplishes one of the following: Either a vertex is popped from the stack (and never again pushed onto the stack), or a previously unvisited vertex is visited. If the while loop should fail to terminate, it would mean there are fewer than |V| pops of the stack, and this means the algorithm continues to find adjacent vertices forever. This is impossible (since the graph is finite). Therefore, the loop will eventually terminate.

Proof of (2) <u>Observation</u>: If the algorithm marks vertex v, it will also eventually mark every vertex adjacent to v. <u>Proof:</u> When a vertex v is first marked, it is pushed on the stack. A peek operation will repeatedly reveal v on the stack as long as there are edges (v,w) for which w is unvisited. Therefore, for each vertex v that is marked and each (v,w) in in E, w will also eventually be marked.

Suppose at the end of the algorithm, some vertex w has not been marked. Let $s – v_1 – v_2 – \ . \ . \ . – v_k = w$ be a path to w from s. Let i be least such that v_i is unmarked; therefore v_{i-1} has been marked. By the Observation, v_i will eventually be marked. Contradiction.

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. *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), to be described next.

Implementation Techniques

1. Ensure O(1) access time to determine whether a vertex has been visited (we use a hashtable for this)

2. Use an adjacency list to keep track of adjacent elements.

3. *Running Time.* We show the running time is O(n+m). Since we have already proven correctness, we know that 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.

   It is conceivable that repeatedly searching the entire list of vertices adjacent to v (obtained from the adjacency list) could be costly, but we arrange it so that each vertex in the list of vertices adjacent to v is accessed exactly one time during the entire component loop. We do this by removing from the list each vertex w that is found in the search immediately after it is processed (sometimes it will be marked as visited; other times, it will already have been marked because it was adjacent to an earlier processed vertex). Note that since visited vertices are never re-used, removing w from the adjacency list can be done safely. Therefore, for each vertex v, there will be deg(v) accesses of its adjacent vertices, with O(1) processing on each. (Note: the adjacency list that is used during DFS will be only a copy of the original.)

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

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

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. *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 connected 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()
```

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:

$$O(1) + O(1) + . . . + O(m_1 + n_1) + O(1) + . . . + O(1) + O(m_i + n_i) + . . . + O(m_k+n_k)$$
$$= O(n) + O(m + n) = O(m+n).$$

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. *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}

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

3. *Running Time.* The algorithm for computing connected components proceeds in two steps:

   A. Partition V into component sets of vertices $V_0, V_1, \ldots, V_k$. [This is accomplished when DFS is run on the graph and we have recorded the component number of each vertex, as described above.]

   B. Partition E into components $E_0, E_1, \ldots, E_k$ so that, for all $e \in E_i$, the endpoints of e belong to V_i. [This extra step is for convenience, so that every component can be represented as a graph.]

   <u>Running time for A</u>: The extra steps to partition the vertices into component sets require only constant time (since all that is done is to add a value to a hashtable and add a vertex to the list that corresponds to its component number), so this part of the algorithm runs in O(n+m), same as DFS.

   <u>Running time for B</u>: The partition $E_0, E_1, \ldots, E_k$ is obtained by reading the adjacency list of each component. In particular, for each component V_i and each vertex v in V_i, we form edge objects for each vertex adjacent to v; the result is collections E_i containing edges having endpoints in V_i. For each i, the running time is

   $$O(\sum_v \deg(v)) = O(m_i)$$

   where the sum is taken over all $v \in V_i$. Summing over $i \in \{1, 2, \ldots, k\}$ yields a running time of O(m) for step B.

   Therefore total running time for obtaining connected components is O(n+m).

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.)

3. *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.

   Running Time After component search has completed, we have lists of vertices and edges by component. Therefore, for each i, determining the values m_i and n_i requires O(1) time. If there are k components, then running time for determining whether G has a cycle (once the connected components are known) is O(k). Therefore,

   1. If the graph components have been identified, running time to determine whether G has a cycle is O(k) (k = # components) which is O(n)
   2. If graph components have not been identified, running time is O(m+n).

4. *Optional: An Alternative Approach.* If you have already computed the *connected components*, the approach given above tells how to determine if the graph contains a cycle.

   Instead, if you have already computed a *spanning forest* for the graph, you can use this information instead to determine whether the graph has a cycle, in the following way: Let F be the spanning forest for the graph G.

   - If $\varepsilon_F = \varepsilon_G$, the graph has no cycle
   - If $\varepsilon_F \neq \varepsilon_G$, then the graph contains a cycle

   Running Time. A spanning forest is presented either as a collection of edges, discovered during Depth First Search, or as the subgraph induced by those edges. Either way, the value of ε_F can be read in O(1). Likewise, ε_G can be read in O(1). Therefore, running time to determine if there is a cycle is O(1) if a spanning forest is already known, O(m+n) if not.

   Optional Exercise. Verify the correctness of this algorithm for detecting existence of a cycle by proving the following theorem:

   Theorem. Let F be a spanning forest for G and let ε_F, ε_G denote the number of edges in F and G, respectively. Then G contains a cycle if and only if $\varepsilon_F \neq \varepsilon_G$.

   Hint: Prove the following two claims:

   Claim 1. Suppose a graph G with n vertices and m edges satisfies $m < n - 1$. Then G is not connected. (Hint: If G is connected, think about what you get when you run the spanning tree algorithm.)

   Claim 2. Suppose a connected graph G with n vertices and m edges satisfies $m \geq n$. Then G contains a cycle. (Hint: Assume G is connected. If G were acyclic, then $m = n$ -1.)

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)
```

Correctness. Two points to check:

(1) the while loop eventually exits

(2) after the while loop exits, every vertex in G has been marked

Proof of (1) In each pass in the while loop, one marked vertex is removed, and no vertex that has been removed is ever added back to the queue.

Proof of (2) Suppose at the end of the algorithm, some vertex w has not been marked. Let $s – v_1 – v_2$ - . . . – $v_k = w$ be a path to w from s. Let i be least such that v_i is unmarked; therefore v_{i-1} has been marked, and so was added to Q. Eventually, (v_{i-1}, v_i) must be examined. If v_i has not yet been marked, it will be marked at that time. Contradiction.

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.

2. 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*) is called the *BFS rooted spanning tree.*

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

4. The length of a shortest path from a starting vertex s (which could be any vertex in the graph) to any vertex v can be found by just reading the level to which v belongs. This is established by the following Theorem:

Theorem. In a BFS rooted spanning tree with root s, the length of the shortest path from s to a vertex v is equal to the level to which v belongs.

Exercise: 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 .

(Optional) **Proof of Theorem**. Proceed by induction on shortest path lengths. Clear for 0. Suppose the result holds for shortest path lengths < i – that is, if a shortest path from s to w has length j < i, then the level of w is j. Suppose v is such that some shortest path p: s – a – b – . . . – u – v from s to v has length i. Then s – a – … – u is a shortest path from s to u of length i – 1. It follows by the induction hypothesis that u belongs to level i - 1. Since u is adjacent to v, there are three possibilities for the level of v: i - 2, i -1, or i. But by the BFS algorithm, we already know that there is a path of length i - 2 (i - 1) from s to any vertex at level i - 2 (i - 1), so level of v could not be either i - 2 or i - 1 (since p is a shortest path). It follows that the level of v is precisely i, as required.

5. *Algorithm for computing shortest path* from s to a vertex v:

(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

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

Determining Whether *G* Is Bipartite

1. Recall that *G* is bipartite iff *G* does not contain an odd cycle.

2. Therefore, we want to obtain an algorithm for determining whether *G* has an odd cycle.

3. **Theorem.** (*Odd Cycle Theorem*). If *G* is connected and has an odd cycle, then during the building of the rooted tree, BFS will encounter a vertex that is adjacent to and at the same level as another already visited vertex. Conversely, if such a vertex is encountered, there is necessarily an odd cycle in *G*.

4. Therefore, the algorithm for determining bipartite-ness is to keep track of levels of vertices as the spanning tree in each component is being built, and if a vertex is encountered that is both adjacent to and at the same level as a previously visited vertex within the current component, return FALSE (since the graph has an odd cycle); if this never happens, return TRUE.

5. *Implementation.* Implement the search for an odd cycle as a subclass of the BFS class.

6. *Running Time.* The extra processing steps to test whether an odd cycle has been found run in O(1). Therefore, running time to determine whether a graph is bipartite is O(n + m).

Optional: **Outline of Proof of Odd Cycle Theorem**

Let G = (V, E) be a simple graph.

Claim 1. If G has an odd simple cycle, then as any BFS rooted tree for G is built, some vertex v will be found that is adjacent to another vertex in the tree at the same level as v.

Note: Since every odd cycle contains an odd simple cycle, it is enough to detect the presence of odd simple cycles.

Proof. Pick any vertex *s* in G, and start building the BFS rooted spanning tree T starting at *s*. There are two cases:

1. Case 1. *s belongs to an odd simple cycle C in G*. We will show that C will be discovered as T is being built. Assume C has 2n+1 edges. Notice s must have two children both of which are vertices in C. And each of these must have a child that belongs to C. When level n is reached, two vertices in each level > 0 have been found that belong to C, and so all 2n+1 vertices in C have been discovered. The two vertices in level n must therefore be adjacent.

2. Case 2. *s does not belong to an odd simple cycle in G*. Since s does not belong to an odd simple cycle, let u_0 be the first vertex encountered during the building of T that does belong to an odd simple cycle C; say u_0 is discovered at level k and that C has 2n+1 edges. Now we can reason as in Case 1 to conclude that at level k + n, we will have found the last two vertices in C and they must be adjacent.

Claim 2. Suppose that, as a BFS rooted tree for G is being built, a vertex u is found at the same level as an already visited vertex v and u, v are adjacent in G. Then u, v belong to an odd cycle in G.

Proof. We follow a path from each vertex, p_u, p_v, through successive parents, grandparents, etc, one level at a time, until the two paths meet for the first time at vertex w. We are guaranteed that there is such a meeting point since the paths must eventually meet at the root. Let q_v be the path from w back to v, obtained by following p_v in reverse order. Then

$$p_u \cup q_v \cup \{(v,u)\}$$

is a simple odd cycle containing u and v.

Main Point

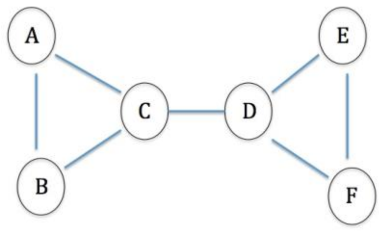
The BFS and DFS algorithms are procedures for visiting every vertex in a graph. BFS proceeds “horizontally”, examining every vertex adjacent to the current vertex before going deeper into the graph. DFS proceeds “vertically”, following the deepest possible path from the starting point and, after reaching 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 and determine if a graph is bipartite.

3. DFS can be used to compute the *biconnected components* – we will not explore this topic in this class.



G is <u>*biconnected*</u> if removal of an edge does not disconnect G.

A <u>*biconnected component*</u> 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

4. Both BFS and DFS work (essentially unchanged) for *directed* graphs. One of the specialties of DFS on directed graphs is that if the graph has no directed cycles, it scan be used to produce a *topological ordering.* We conclude with this final topic

Traversals of Directed Graphs: Review of Digraphs

- A **digraph** is a graph whose edges are all directed
  - Represent $x \to y$ by the ordered pair (x,y)

- Applications
  - one-way streets
  - flights
  - task scheduling

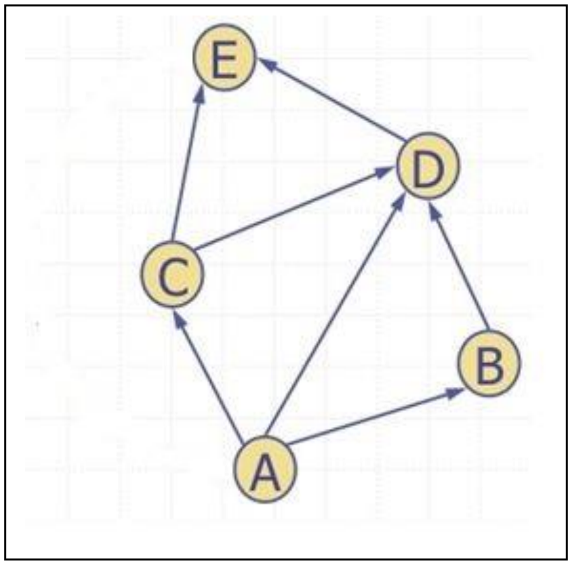
- Still use notation $G = (V, E)$ but now
  - Each edge goes in one direction:
    - Edge (a,b) goes from a to b, but not b to a.

- The *underlying graph* of G is the undirected graph obtained by ignoring the direction of each edge.

- If G is simple, $m \leq n (n - 1)$.

- *Implementation Adjustment:* We keep in-edges and out-edges in separate adjacency lists. [See the code]

- In a directed graph, for any $v \in V$,
  - $\text{indeg}(v) = |\{w \mid (w,v) \in E\}|$
  - $\text{outdeg}(v) = |\{w \mid (v,w) \in E\}|$
  - $\sum_v \text{indeg}(v) = \sum_v \text{outdeg}(v) = |E|$



- In a digraph, a *directed path* from vertex v to vertex w is a path from v to w in which all edges are directed forward.
  Example:
  C-D-E is a directed path from C to E

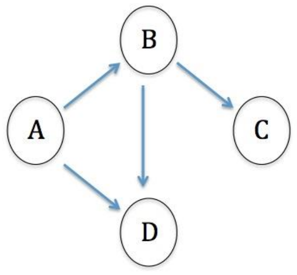
- Vertex w is *reachable* from vertex v if there is a directed path from v to w. Example: E is reachable from C but A is not reachable from C.

  Note: Any vertex is reachable from itself (via the empty path).

Questions to Answer About Digraphs

1. DFS algorithm works the same as in the undirected case, except that the task of examining unvisited adjacent vertices for a vertex v is replaced by examining unvisited *out vertices* for a vertex v (the out vertices are the vertices w for which (v,w) belongs to E).

2. Also, DFS in the directed case marks all vertices that are *reachable* from the starting vertex. Example:



Here, if starting vertex is A, DFS marks all vertices.

If starting vertex is B, DFS marks only B, D, C (an outer loop would need to search for next unvisited vertex in order to mark the vertex A)

3. DFS and minor extensions of it can be used to determine the following in linear (O(n + m)) time

    a. All vertices reachable from a given starting vertex
    b. Whether vertex v can be reached from vertex w, for any v, w.
    c. Whether G is *strongly connected;* that is, whether, for any vertices v, w, there is a directed path from v to w (this is stronger than *connectedness*, which is a property concerning the underlying undirected graph)
    d. A topological ordering of the graph, *provided G has no directed cycles.* (A directed cycle is a directed path with no repeated edges for which the first and last vertex in the path are the same.)

Topological Orderings

Definition. Suppose G = (V,E) is a directed graph. A topological ordering of G is a function f: V → N satisfying: for all (v,w) in E, f(v) < f(w).

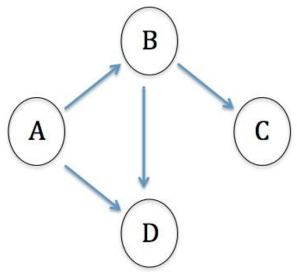
Idea. A topological ordering is a way of organizing the vertices of G into a list in such a way that whenever (v,w) is a directed edge of G, v precedes w in the list.

Uses.

- Organize a sequence of tasks so that whenever task_1 needs to be done before task_2, the output sequence ensures that this requirement is met. Tasks are represented as vertices and v → w only if task v must be executed before task w.

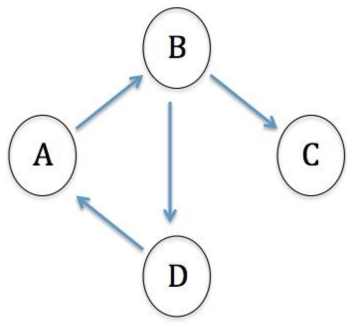
- Devise a schedule of classes subject to the constraint that certain classes have pre-requisites. Classes are represented as vertices and v → w if and only if class v is a pre-requisite for class w.

Examples



1. A topological ordering:
   f(A) = 1, f(B) = 2, f(D) = 3, f(C) = 4
   - gives the ordering A, B, D, C

2. An assignment that is not a topological ordering:
   f(A) = 1, f(D) = 2, f(B) = 3, f(C) = 4
   - now, (B,D) is an edge, but f(B) > f(D)



No topological ordering is possible here because of the directed cycle A →B → C → D: We must have f(A) < f(B) < f(D) < f(A), so f(A) < f(A).

Topological Sorting with DFS

Theorem. A digraph admits a topological ordering if and only if it contains no directed cycle (iff it is a *directed acyclic graph (DAG)*).

Algorithm: TopSort

Input: A DAG G = (V, E), starting vertex s (having no in-vertices)
Output: An arrangement of the *n* elements of V reachable from s, in topological order

$n \leftarrow |V|$

- run DFS till a vertex v is encountered that has no unvisited out vertices (this is precisely where the stack is popped) and label v with n

- continue running DFS till another v is encountered, with no unvisited out vertices, and label this vertex with n-1

- continue until all vertices have been labeled; if all vertices that are reachable from start vertex are used up, continue on to the next unvisited vertex

NOTE: In case a starting vertex is not provided as input, a starting vertex can be found by scanning in the in-list of the graph for a vertex that has zero in-vertices. Any such vertex can serve as the starting vertex. (This requires only O(n) extra work.)

Correctness: First, note that if we do an initial scan of the in-list to find a starting vertex, we will always be able to find a suitable choice: Because the graph contains no directed cycles, there must be a vertex that has no in-vertices.

We will perform an induction on DAGs that contain a particular vertex *s* having no in-vertices in the graph. We proceed by induction on the number *k* of vertices that are reachable from *s* to show that the algorithm outputs the reachable vertices in topologically sorted order. The result holds for $k = 1$. Assume the result whenever the number of reachable vertices in a DAG containing *s* is < k.

Consider a DAG $G=(V, E)$ containing *s* with no in-vertices, for which the number of vertices reachable from *s* is *k*. Proceeding according to the algorithm, DFS runs until it first encounters a vertex *v* with no unvisited out vertices, and labels it *k*. DFS will certainly find such a vertex since *G* has no directed cycles. Let *u* be the in-vertex of *v* that TopSort visited immediately before visiting *v* (such a *u* exists since $k > 1$).

Consider the graph *G'* obtained by removing *v* from *G* (together with edges incident to *v*). *G'* also contains *s*, is a DAG, and contains exactly k -1 vertices that are reachable from *s*. By the induction hypothesis, we can carry out TopSort successfully to assign natural numbers 1 .. $k – 1$. We now number the vertices in the graph *G* with the same values as assigned in *G'*. Since *v* had no out-vertices, assigning the largest number *k* to *v* is consistent with the assignments of 1 .. $k – 1$ to the other vertices. Therefore, the *k* vertices in *G* that are reachable from *s* are now topologically sorted. This completes the induction.

Running Time: The only change from the usual DFS algorithm is that we assign numeric values to each vertex. Therefore, running time is O(n + m).

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.

https://www.youtube.com/watch?v=bIA8HEEUxZI

https://www.softwaretestinghelp.com/java-graph-tutorial/#Java_Graph_Data_Structure