**Lesson 12, Part I**

# Implementing Graphs and Graph Algorithms – Undirected Graphs:

*Knowledge Has Organizing Power*

# Operations on Graphs and Efficient Implementation

**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.

# 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.

1. *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()

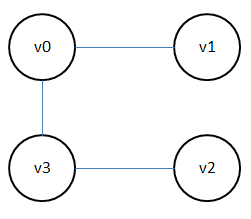
boolean isTree()

boolean isBipartite()

int lengthOfShortestPath(Vertex u, Vertex v)

**Determining Adjacency**

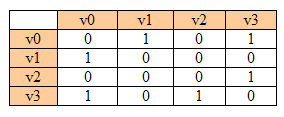
1. Most graph algorithms rely heavily on determining whether two vertices are adjacent and on finding all vertices adjacent to a given vertex. These operations should be as efficient as possible.



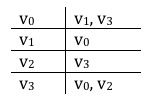
1. *Two ways to represent adjacency.*

* Adjacency Matrix
* Adjacency List

1. An *Adjacency Matrix* *A* is a two-dimensional ν x ν array consisting of 1’s and 0’s. A 1 at position *A*[i][j] means that vertices vi and vj are adjacent; 0 indicates that the vertices are not adjacent.



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.*



*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 θ(*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
* 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 connected 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 ≠ 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.

***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)** Observation: If the algorithm marks vertex v, it will also eventually mark every vertex adjacent to v. Proof: 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 – v1 – v2 – . . . – vk = w be a path to w from s. Leti be least such that vi is unmarked; therefore vi-1 has been marked. By the Observation, vi 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.

If access of the list of vertices adjacent to v (obtained from the adjacency list) is not done in the right way, it could be costly when this list is large. Our approach is to examine, at each step, the *first* element still remaining in the adjacency list for a vertex *v*; whether the element found is processed further or passed over, we do only O(1) work on each adjacent element, and then discard the element, so that at the next access, we can again examine the first element. In this way, we never examine an element in the list of vertices adjacent to *v* more than once. 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, so the original Graph is not affected by these removals..)Therefore, for each v, O(1) + O(deg(v)) steps are executed. The sum over all v in V is

**O(∑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 G1,

G2, . . ., Gk, so that each component Gi has ni vertices and mi edges, then running DFS on each component requires O(ni + mi) 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 Gi, that has mi edges and ni vertices. So total processing that occurs when that vertex is read is O(mi + ni). 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(m1 + n1) + O(1) + . . . + O(1) + O(mi + ni) + . . . + O(mk+nk)

= 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”. 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** | |  | **Vertex Component Number**  **Map** | |
| 0 | v1, v2, v3, v4 |  | v1 | 0 |
| 1 | v5, v7, v9 |  | v2 | 0 |
| 2 | v6, v8, v11 |  | v3 | 0 |
| 3 | v10 |  | v4 | 0 |
|  |  |  | v5 | 1 |
|  | //increment each round |  | v6 | 2 |
|  | componentNum = 0; |  | . . . | . . . |

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

A. Partition V into component sets of vertices V0, V1, . . ., Vk. [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 E0, E1, . . ., Ek so that, for all e ∈ Ei, the endpoints of e belong to Vi. [This extra step is for convenience, so that every component can be represented as a graph.]

Running time for A: 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.

Running time for B: The partition E0, E1, . . ., Ek is obtained by reading the adjacency list of each component. In particular, for each component Vi and each vertex v in Vi, we form edge objects for each vertex adjacent to v (do this by consulting the adjacency list: for each v, scan the list of adjacent vertices and for each w in that list, if w also belongs to Vi, then add the edge (v,w) to Ei). The result is collections Ei containing edges having endpoints in Vi. For each i, the running time is

O(∑v deg(v)) = O(mi)

where the sum is taken over all v ∈ Vi. Summing over i ∈ {1, 2, …, 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.)

1. *Cycle.* Each component is connected. Suppose the components are C0, C1, C2,…, C*k*. For each *i*, let *ni* be the number of vertices and *mi* the number of edges. If for each *i* we have that *mi = ni* –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, mi* is not equal to *ni* – 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 *mi = ni* - 1. Contradiction!

Therefore, the criterion for existence of a cycle is:

*G has a cycle if and only if there is a connected component Ci of G such that the number ni of vertices of Ci satisfies mi ≠ ni –1, where mi is the number of edges in Ci.*

*Running Time* After component search has completed, we have lists of vertices and edges by component. Therefore, for each i, determining the values mi and ni 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).

**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 ≠ ∅ do

v ← 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 – v1 – v2 - . . . – vk = w be a path to w from s. Leti be least such that vi is unmarked; therefore vi-1 has been marked, and so was added to Q. Eventually, (vi-1, vi) must be examined. If vi 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(∑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 Ci in time O(ni+mi) 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: v0 – v1 – v2 – … – vn 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.

**Algorithm *ShortestPath(G, s, v)***

**Input**: A connected graph G with vertices s and v  
 **Output**: The shortest length of a path in G from s to v

* 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
* Return the level of v in T

**Running Time** The extra work to keep track of levels as the BST rooted tree is built is O(1), so running time is O(m + n).

Note: It might seem that it is a waste to compute shortest paths between s and every other vertex, but in fact there is no faster way to compute shortest path between just two vertices. See <https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/lec20/lec20.htm>

**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 v1 – v2 – . . . – vk – vk+1 is a shortest path from v1 to vk+1, then whenever 1 ≤ i ≤ k, the path v1 – v2 – . . . – vi is a shortest path from   
v1 to vi .  
  
*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. We show v is in   
Level i. 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.

*Optional:* Algorithm for computing shortest path from s to any vertex v:

The previously described algorithm provides an efficient procedure for determining the shortest path from a given vertex to all other vertices in the graph, but does not provide a *shortest path* in each case (only shortest path length). Here we compute a *shortest path* from s to v, for each v.

Output of the algorithm is a map P with the property that for each vertex v, P[v] stores a shortest path from s to v.  
  
*Algorithm:* Perform BFS, with start vertex 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 path P[w] from s to w.

Initially, let P[s] = ∅. When considering edge (v,w), where v has just been removed from the queue, if w is unvisited, set

P[w] = P[v] U (v,w) . After BFS completes, return the map P.

[Note: It needs to be checked that when we attempt to obtain P[w], a value for P[v] already exists – this is part of the correctness proof.]

*Observations.*

1. *Proof of Correctness.* Note that P[s] is a shortest path from s to itself. Notice that during BFS, as vertices are pulled off the queue, the vertex at level 0 is pulled off first, then all vertices at level 1, then all vertices at level 2, and so forth (once we begin pulling off vertices of level k, no vertices of an earlier level occur in the queue).  
     
   We proceed by induction on levels to show that when a vertex w is first pulled off the queue
   1. the value P[w] has already been computed, and
   2. for each v adjacent to w, the length of the path P[v] = P[w] U {(w,v)} is equal to the level of the vertex v

If w has level 0, the result is obvious. Assume this holds true for vertices at levels below k, where k > 0; we prove for those at level k. Suppose lev(w) = k, and w is pulled from the queue. Vertex w was placed on the queue because it was adjacent to a vertex u at level k – 1 and P[w] was correctly computed (by induction hypothesis) to be a path from s to w of length is level(w). For each v adjacent to w, the algorithm will form P[v] by computing P[v] = P[w] U {(v,w)}; in each case, the length of P[v] is clearly level(w) + 1 = level(v), as required. This completes the induction.

1. *Running Time.* The algorithm is a simple variant of BFS; as each node v is encountered, there is an extra O(1) processing time to obtain the value of P[v] and insert into the hashtable. Therefore, running time is, like BFS, O(n+m).
2. *Example of Dynamic Programming.* Suppose we have a BFS rooted tree T with root s and v, w are two vertices (different from s). The shortest path from s to v and from s to w will, in many cases, include a common subpath; as we move from s to each of v, w, the path will be the same until a splitting point t is encountered. This means that P[t] is a subgraph of both P[v] and P[w]. We can view computation of P[t] as a common “subproblem” for P[v] and P[w]. If we were to compute each P[u] (for u in V) individually, we would repeat many computations of subpaths starting at s. The algorithm given here *memoizes* paths computed so far so that each is computed only once. This is therefore an example of a *dynamic programming solution*.
3. *Extra space.* This algorithm uses at least nlog n extra space – n2 in the worst case – to store all the paths P[u]. [Best case: tree has height   
   log n; worst case: tree is has only one branch of length n] If the objective is to find a shortest path from s to just one particular vertex v, there is an algorithm that is just as fast, but that requires only O(n) extra storage – see the Lab.

**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.

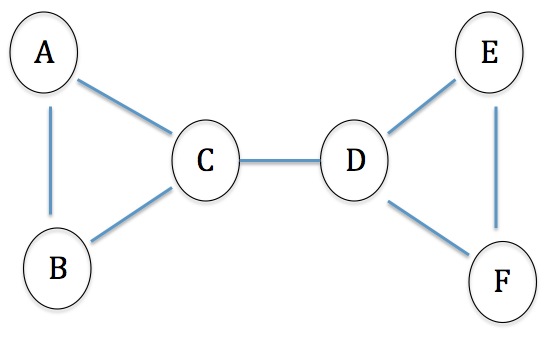
5. *Implementation.* Implement the search for an odd cycle as a subclass of the BFS class.

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).

**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 *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

4. In Part II of this Lesson, we explore special features of DFS when it is applied to *directed graphs.*

**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. 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. |  |