Graph Traversal Algorithms

Many important problems in the field of computer science have solutions that are best modeled by graph traversal. When considering traversal of graphs, we need to consider some sort of systematic procedure for "visiting" each vertex in the graph and generating a solution to the problem based on these traversals.

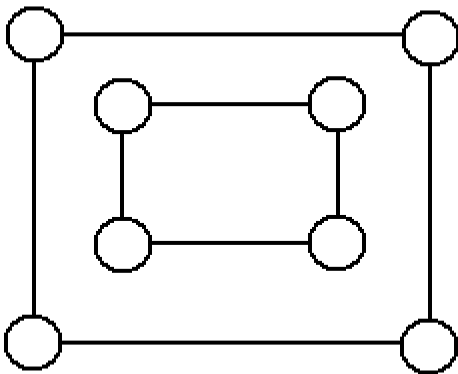
The two main graph traversal algorithms are called **Depth-First Search (DFS**) and **Breadth-First Search (BFS)**. In these algorithms, we base the traversal on the adjacencies found in the graph. Other search algorithms, such as **branch-and-bound** and a number of other algorithms found in the study of Artificial Intelligence, use more of the graph structure. We begin with the "simple" algorithms.

We begin this discussion by recalling the effects of **adjacency** and **connectivity** upon graph traversal. Let G = (V, E) be a graph with vertex set V and edge set E. Two vertices u and v are said to be said to be adjacent in G if $(u, v) \in E$; i.e., (u, v) is an edge in the graph. A graph traversal algorithm moves from one vertex to another along edges, thus one can move from vertex u to vertex v if and only if $(u, v) \in E$. A path from vertex u to vertex v in a graph G can be defined as a sequence of adjacent vertices that starts with u and ends with v. We may present a recursive definition of the existence of a path from u to v as follows.

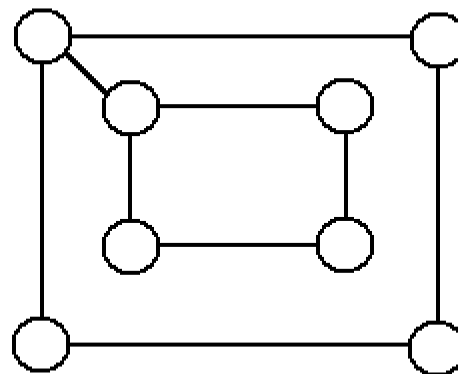
In a graph G = (V, E), there is a path from vertex u to vertex v if and only if either

1) $(u, v) \in E$, or
2) there is a vertex $w \in V$, with $(u, w) \in E$, such that there is a path from w to v.

A graph is said to be **connected** if and only if there is a path from u to v for every pair of vertices u and v. If a graph is not connected, it will be seen to comprise two or more **connected components**. Formally a connected component is a maximal subgraph of a given graph, meaning that the subgraph cannot be expanded by addition of extra vertices that are adjacent to vertices already included in the component.



Graph with
Two Connected Components



Graph with
One Connected Component
(A Connected Graph)

Copyright © by Edward Bosworth

Graph traversals are defined in terms of connected components. A traversal of a single connected component of a graph (or the graph itself, it the graph is connected) produces a tree structure indicating the order in which the vertices were visited. A traversal of a graph with two or more components produces two or more trees, collectively called a **forest**.

Depth-First Search

We show the DFS algorithm as a pair of algorithms, one called DFS and one called dfs. The algorithm presented uses two arrays, called Mark and Back, to manage the search and help in generation of the search forest.

```
Algorithm DFS (G)  // DFS on a graph G = (V, E)
//  The graph G may be connected or unconnected.
//  This operates by marking each vertex.
//  This uses two arrays: Mark and Back.
//
   count = 0
   For each vertex v ∈ V Do     // The primary purpose of
      Mark[v] = 0               // DFS is to initialize these
      Back[v] = 0               // arrays and call dfs.
   End For
   For each vertex v ∈ V(G) Do
      If (0 == Mark[v]) then
           //
           // Vertex v is in a new component, not connected
           // to any vertex already visited by algorithm dfs.
           dfs(v)
      End If
   End Do
```

If G is a connected graph, then dfs(v) will be called exactly once in DFS(G), as every vertex in G will be marked by the first call to dfs(v). Recall that the DFS produces a rooted tree structure corresponding to the traversal of the graph. For a connected graph G, the root vertex of the search tree will be the first vertex used in a call to the dfs algorithm.

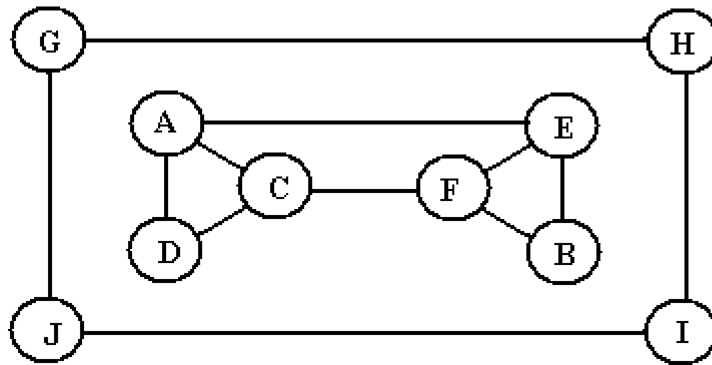
If G is not a connected graph, then dfs(v) will be called once for each component, producing a search tree for each component. The result of DFS(G) will be a search forest, with one search tree for each of the connected components.

Each search tree in the forest corresponds to a connected component in the graph. Each search tree is rooted at that vertex in the connected component that was first selected by the top-level algorithm DFS.

Copyright © by Edward Bosworth

```
Algorithm dfs(v)     // v is a vertex in the graph G
//
   count = count + 1      // This is a global variable
   Mark[v] = count        // Explicit array here
   For each vertex w in V adjacent to v Do
      If (0 == Mark[w]) Then
            Back[w] = v    // Remember where we "came from".
            dfs(w)
      End If
   End Do
```

The best way to proceed here is to solve a specific instance of the DFS problem. We examine the graph shown in the figure below. Note that the graph, as drawn, clearly is not connected, having exactly two connected components with vertex sets {a, b, c, d, e, f} and {g, h, i, j},



In order to illustrate the execution of the algorithm, we must work from the computer representation of the graph and introduce the auxiliary data structures required for DFS. The graph may be represented by an adjacency matrix, with the 0's not shown.

	A	B	C	D	E	F	G	H	I	J
A			1	1	1					
B					1	1				
C	1			1		1				
D	1		1							
E	1	1				1				
F		1	1		1					
G								1		1
H							1		1	
I								1		1
J							1		1	

Copyright © by Edward Bosworth

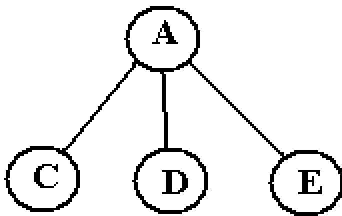
Vertex	A	B	C	D	E	F	G	H	I	J
Mark	0	0	0	0	0	0	0	0	0	0
Back	0	0	0	0	0	0	0	0	0	0

We now consider the algorithm DFS(G), arbitrarily deciding that the statement `For each vertex v ∈ V Do` is interpreted as scanning the above array. The first vertex to be the root of a search tree is v = A, which is the first vertex marked with a 0. Note that we could have started the search at any vertex; I choose A for no good reason.

The first effect of calling dfs(v) with v = A is to set the mark of A to 1, so we have the following for the mark array.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	0	0	0	0	0	0	0	0
Back	0	0	0	0	0	0	0	0	0	0

Before continuing with the search, we should note an artifact of the way in which the algorithm is often presented – we can see the entire graph and search it mentally with great facility. This presentation will focus on only those parts of the graph that are visible to the algorithm at the time a decision is made. When we have processed A, the situation is as follows.



Here we show only vertex A and the vertices adjacent to it. The rest of the graph is "invisible" at this point. The algorithm proceeds recursively, implicitly using a call stack. As this is the first call, the call stack might be viewed as
STACK => A

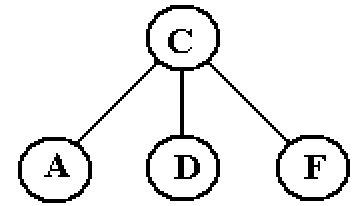
Consider now the statement `For Each vertex w in V adjacent to v Do`
There are many ways to implement this in a programming language. One way would be as follows: `For w = A to J Do If Adjacency[w, A] = 1 Then`

The requirement of the algorithm is that each vertex adjacent to A be explored. The order of exploration is not important and depends on the data structure used to represent the graph. In these notes, we follow the books suggestion and process vertices in alphabetical order, thus we next call algorithm dfs on vertex C. After this call, we have the following.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	0	0	0	0	0	0	0
Back	0	0	A	0	0	0	0	0	0	0

Copyright © by Edward Bosworth

At this point the stack status is given by STACK => A => C. Vertex C has been marked with the number 2, denoting its position in the traversal order. Again, all we see is those three vertices that are adjacent to vertex C. The algorithm calls for us to process each of those three vertices, but we see that vertex A has already been marked. For this reason, vertex D is next.

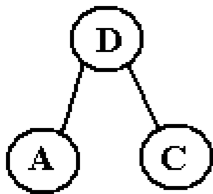


It is at this point in the algorithm that we first see two types of edges in the graph. There are three edges incident on vertex C: (C, A) – incident on a vertex already visited and two edges (C, D) and (C, F) incident on vertices that have yet to be visited. The DFS algorithm has names for these types of edges: tree edge and back edge.

A **tree edge** is an edge incident on the vertex being processed that is also incident on an unmarked vertex. A **back edge** is an edge incident on the vertex being processed that is also incident on a marked vertex. The origin of this latter name should be obvious.

We now see the use of the Back array; it identifies back edges. The algorithm now calls dfs (D), after which call we have the following.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	3	0	0	0	0	0	0
Back	0	0	A	C	0	0	0	0	0	0



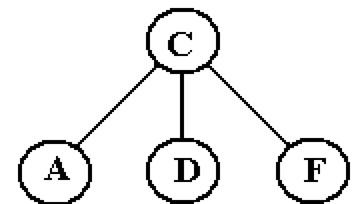
At this point the status of the stack is STACK => A => C => D.

There are two vertices adjacent to D: A and C. Both have been marked, so we remove D from the stack and return to C.

The situation after D is popped off the call stack by the return from the recursive call dfs(D) is shown below.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	3	0	0	0	0	0	0
Back	0	0	A	C	0	0	0	0	0	0

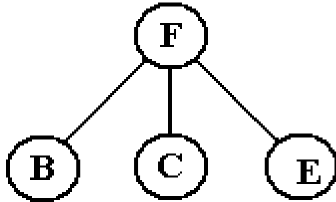
The stack status is given by STACK => A => C.
There are three vertices adjacent to C (just as there was when we last visited the vertex), but now two of them (A and D) have been marked. The only vertex that is both adjacent to vertex C and unmarked is vertex F, so we visit that one.



Copyright © by Edward Bosworth

The situation after vertex F is visited is shown below.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	3	0	4	0	0	0	0
Back	0	0	A	C	0	C	0	0	0	0

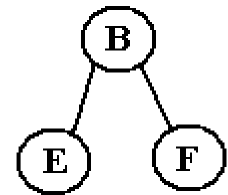


The status of the stack is given by STACK => A => C => F.
There are three vertices adjacent to F, we attempt to visit B first and note that it is marked with 0. So the next step in the algorithm is to process dfs(B).

The situation after vertex B is visited is shown below.

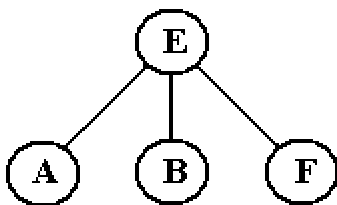
Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	5	2	3	0	4	0	0	0	0
Back	0	F	A	C	0	C	0	0	0	0

The stack is given by STACK => A => C => F => B.
There are two vertices adjacent to B: E and F. We attempt to visit E first and note that it is unmarked, so we process dfs(E).



The situation after vertex E is visited is shown below.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	5	2	3	6	4	0	0	0	0
Back	0	F	A	C	B	C	0	0	0	0



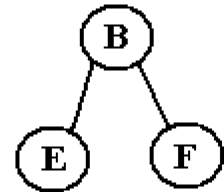
The stack is given by STACK => A => C => F => B => E.
There are three vertices adjacent to E, but each has been marked with a positive number. For this reason, we return and move up the call stack. We called dfs(E) from visiting B, so it is there we return.

Copyright © by Edward Bosworth

The situation after E is popped off the stack by the return from the recursive call is shown below.

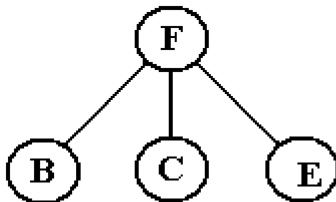
Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	5	2	3	6	4	0	0	0	0
Back	0	F	A	C	B	C	0	0	0	0

The call stack is given by STACK => A => C => F => B.
Again, there are two vertices adjacent to B: E and F. Both of these vertices have been marked with positive integers, so we again return up the call chain.



The situation after B is popped off the call stack by the return is shown below.

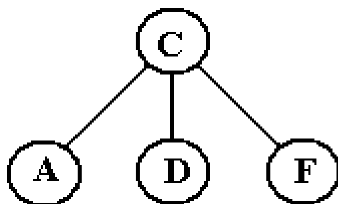
Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	5	2	3	6	4	0	0	0	0
Back	0	F	A	C	B	C	0	0	0	0



The call stack is given by STACK => A => C => F.
There are three vertices adjacent to vertex F: A, C, and E. Each of these has been marked with a positive integer, so again we return up the call chain.

The situation after F has been removed from the call stack by return from the recursive call is shown below.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	5	2	3	6	4	0	0	0	0
Back	0	F	A	C	B	C	0	0	0	0

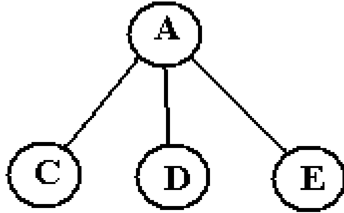


The call stack is given by STACK => A => C. There are three vertices adjacent to vertex C: A, D, and F. Just as before, each of these vertices has been marked with a positive integer, so again we execute a return from the recursive call and move up the call stack.

Copyright © by Edward Bosworth

The situation after C has been removed from the call stack is shown below.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	5	2	3	6	4	0	0	0	0
Back	0	F	A	C	B	C	0	0	0	0

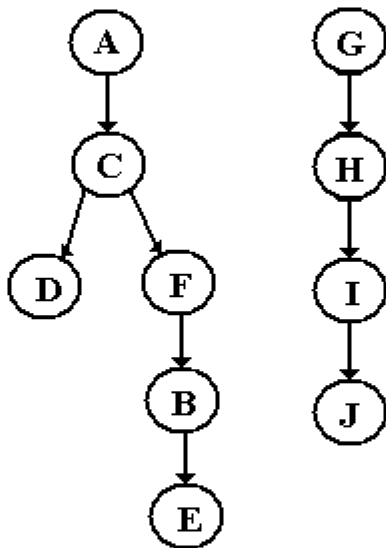


The call stack status is given by STACK => A. There are three vertices adjacent to A, all of which have been marked with positive integers, so we return from the call to dfs(A). At this point we have returned to the top-level algorithm DFS(G).

The top level algorithm then scans the mark array for the next vertex after A to be marked with a 0. The next vertex to have this property is G. The student is invited to show that the algorithm visits vertices G, H, I, and J in that order, giving rise to the following situation when the top-level algorithm DFS exits.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	5	2	3	6	4	7	8	9	10
Back	0	F	A	C	B	C	0	G	H	I

These two arrays now contain the results of the depth first search and serve as a basis for the generation of the two search trees found in the DFS forest for this graph.



The search trees for this graph are obtained from the array Back as follows. There are two vertices in this graph that are roots of the DFS search trees when the algorithm is executed as above: A and G. These are identified by the fact that the Back entry for each is 0.

The DFS tree rooted at A is constructed by reading the Back array. What vertices have Back[v] = A? C is the only vertex. What vertices have Back[v] = C? There are two such vertices – D and F. The process proceeds as expected to produce the search trees. Note that every edge placed in the search tree is an edge that the algorithm would identify as a **tree edge**. The name comes from the fact that the edge is a part of the search tree – big surprise.

Copyright © by Edward Bosworth

Breadth-First Search

We now examine the other major graph traversal algorithm – BFS (Breadth-First Search). Again, BFS is presented in the textbook as a pair of algorithms, to allow for search of each of the connected components of a graph.

The BFS algorithm uses a data structure called a **queue** – a first-in first-out data structure, We shall model the queue in our example as a list, adding to the "back" of the queue and removing from the "front" of the queue. In our adaptation of the algorithm we have the following operations on the queue, which we shall denote as Q.

Initialize(Q)	this sets up the data structure and initializes the queue to empty
IsEmpty(Q)	this returns True if and only if the queue is empty
Add(Q, *v*)	add vertex v to the queue
Remove(Q, *v*)	remove a vertex from the queue and return it as *v*.

As before with DFS, we shall use a number of arrays, including a "Back" array not mentioned in the textbook. The queue may be implemented either as an array or a linked list; the details of its implementation are not of interest at present.

The top-level algorithm, BFS, is applied to the entire graph.

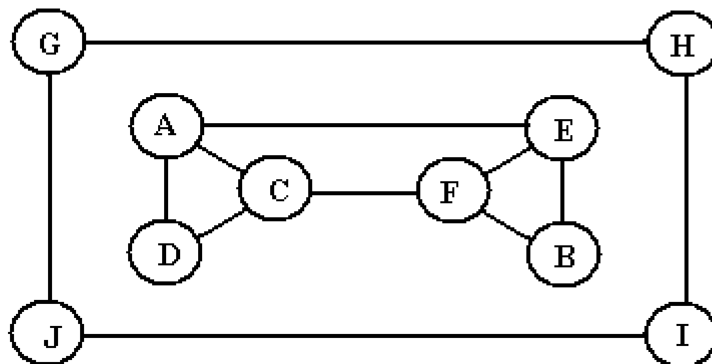
```
Algorithm BFS(G)
//  Implements a breadth-first search traversal for a graph G.
//  The graph can have one or more connected components.
//
// This pair of algorithms uses four global variables.
// Count - the order of the vertex in the traversal
// Q     - the queue used by bfs to order the search.
// Mark  - the "mark array" used to mark each vertex
// Back  - the "back array" used to construct the search tree.
//
   Count = 0         // Initialize the global variables
   Initialize(Q)
//
   For each vertex v in V Do
      Mark[v] = 0
      Back[v] = 0
   End Do

// Now do the search
//
   For each vertex v in V Do
      If Mark[v] = 0 Then bfs(v)
   End Do
```


Copyright © by Edward Bosworth

Here is the "low-level" algorithm bfs(v) – note that it is not recursive. We have labeled three points in the algorithm as 1, 2, and 3 to facilitate talking about it.

```
Algorithm bfs(u)
//
// Visits all unvisited vertices adjacent to vertex u
// and assigns them a number in the order they are visited.
// This also allows the search tree to be built.
//
   count = count + 1   // Increment the global variable count
   mark [u] = count
   Add(Q, u)           // Add this vertex to the queue
// Point 1
   While ( Not IsEmpty(Q)) Do
      Remove (Q, v)    // Remove the front vertex from queue
//    Point 2
      For each vertex w in V adjacent to v Do
//          Point 3
            If (Mark[w] = 0) Then
                 Count = Count + 1
                 Mark[w] = Count
                 Back[w] = v
                 Add(Q, w)
            End If
      End For Each
//
//    Note that I remove the vertex at the top of the loop
//
   End While
```

Here again is our favorite sample graph with two connected components. As before, we expect the BFS algorithm to yield a search forest, with one search tree for each of the two connected components in the graph.



Copyright © by Edward Bosworth

The adjacency matrix for the sample graph is shown below.

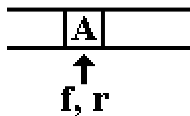
	A	B	C	D	E	F	G	H	I	J
A			1	1	1					
B					1	1				
C	1			1		1				
D	1		1							
E	1	1				1				
F		1	1		1					
G								1		1
H							1		1	
I								1		1
J							1		1	

The work arrays for this search are shown below. As before, we index each array by the vertex name; each element of the array back contains either a 0 or a vertex name. Just after BFS is called and just before the call to bfs(A), the work arrays are as follows.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	0	0	0	0	0	0	0	0	0	0
Back	0	0	0	0	0	0	0	0	0	0

We now call bfs(v) with v = A. After point 1 in the bfs algorithm, we have the following.

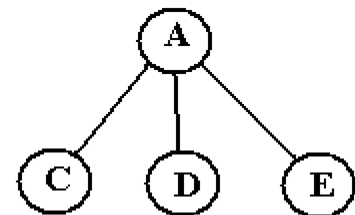
Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	0	0	0	0	0	0	0	0
Back	0	0	0	0	0	0	0	0	0	0



The status of the queue is shown at left. At this point, it has only one vertex. We shall add and remove by inspection.

The loop is entered since the queue is not empty. We first remove the front vertex from the queue (a slight variation of the book's algorithm) and examine its adjacency list.

We again look for vertices that are adjacent to the first vertex A. We find that there are three such vertices: C, D, and E. Again, the order of the search will depend on the order in which we select the vertices from the adjacency list, which could be written as {C, D, E}. We again follow the book's convention of taking the vertices in alphabetical order.



Copyright © by Edward Bosworth

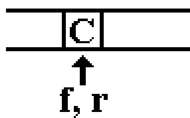
At this point, the algorithm starts to add vertices to the queue and remove them when they are to be visited. To make things a bit clearer, we look at what is happening as a result of the statement `For each vertex w in V adjacent to v` where $v = \text{A}$.

$v = \text{A}$, $\text{Adj}(v) = \{\text{C, D, E}\}$, $w = \text{C}$

Mark vertex C with the count, update the back array, and add it to the queue.

The arrays at this point are:

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	0	0	0	0	0	0	0
Back	0	0	A	0	0	0	0	0	0	0



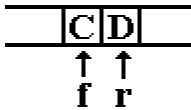
The status of the queue is shown at left. Note that vertex A has been removed from the front of the queue, leaving it temporarily empty (at a time we don't check it) and then re-inserting vertex C.

$v = \text{A}$, $\text{Adj}(v) = \{\text{C, D, E}\}$, $w = \text{D}$

Mark vertex D with the count, update the back array, and add it to the queue.

The arrays at this point are:

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	3	0	0	0	0	0	0
Back	0	0	A	A	0	0	0	0	0	0



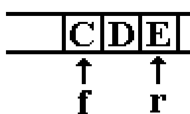
The status of the queue after vertex D has been visited is shown at left. Note that the algorithm will not remove any vertex from the queue until all vertices adjacent to vertex A have been examined and enqueued.

$v = \text{A}$, $\text{Adj}(v) = \{\text{C, D, E}\}$, $w = \text{E}$

Mark vertex E with the count, update the back array, and add it to the queue.

The arrays at this point are:

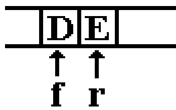
Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	3	4	0	0	0	0	0
Back	0	0	A	A	A	0	0	0	0	0



The status of the queue at this point is shown at left. We have placed on the queue all three vertices that are adjacent to vertex A, which is the vertex with which we originally invoked the algorithm bfs. We now are at the bottom of the `For each vertex` loop.

At point 1 in the bfs algorithm, the queue is found not to be empty, so the next vertex is removed from the queue. We now have the following situation.

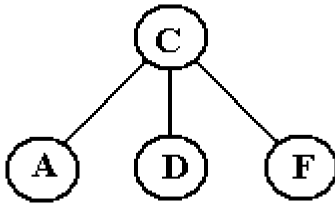
Copyright © by Edward Bosworth



After vertex C has been removed from the queue, we have the situation at left. We next examine each vertex adjacent to vertex C and determine it status with regard to the marking.

The arrays at this point have not been changed, but we repeat them for clarity.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	3	4	0	0	0	0	0
Back	0	0	A	A	A	0	0	0	0	0



At this point in the algorithm, we are looking for unmarked vertices adjacent to vertex C. The picture at left shows the situation. We have Adj(C) = {A, D, F}.

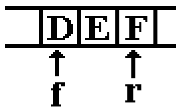
v = C, Adj(v) = {A, D, F}, w = A
Note that vertex A is marked, so we do not process it.

v = C, Adj(v) = {A, D, F}, w = D
Again, note that vertex D is marked, so we do not process it.

v = C, Adj(v) = {A, D, F}, w = F
Mark vertex F with a 5, update the back array, and add vertex F to the queue.

The arrays at this point have not been changed, but we repeat them for clarity.

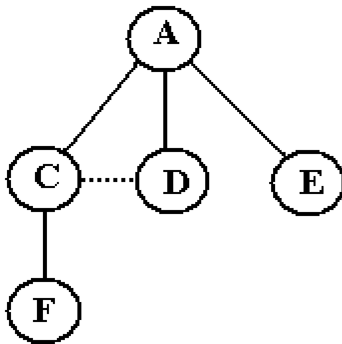
Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	3	4	5	0	0	0	0
Back	0	0	A	A	A	C	0	0	0	0



The status of the queue is again shown at left. We have finished with the vertices that are adjacent to vertex C and are ready to remove another vertex from the queue. Before we do this, we should stop and look at the partial search tree that has been generated at this point.

Copyright © by Edward Bosworth

Here is the search tree after vertices A and C have been fully processed.



We again classify edges in the graph according to how they are encountered in the search process. When considering edges incident on the start vertex, we note that all three of the edges – (A, C), (A, D), and (A, E) – are included in the search tree and are called **tree edges**.

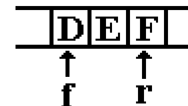
Consider now the edges incident on vertex C, the second vertex visited and marked. There are three edges incident on vertex C: (C, A) = (A, C), (C, D), and (C, F). Each of these edges belongs to a different class.

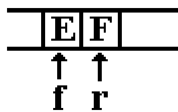
The edge (C, A) is an edge back to a vertex already visited that also is a part of the search path from the root vertex A to the vertex C. In tree terminology A is an **ancestor vertex** of C; in fact it is the **parent vertex** of C. We might call the edge (C, A) a **back edge**, following the terminology used in DFS, although this terminology is not much used in BFS.

The edge (C, D) is an edge to a vertex already visited that is not a part of the search path from the root to C. This type of edge is called a **cross edge**.

When we encountered edge (C, F), we had yet to mark vertex F and add it to the queue, so this edge will become part of the search tree and edge (C, F) is also a **tree edge**.

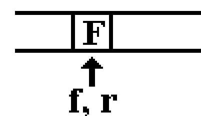
We now return to the top of the loop, having examined all vertices adjacent to vertex C. The status of the queue is shown at left, so vertex D is the next vertex to be removed from the queue.





After vertex D has been removed from the queue, the state of the queue is as shown at left. We consider the adjacency list for vertex D. Noting that Adj(D) = {A, C} and that each of vertices A and C is marked, move on.

Not having found an unmarked vertex adjacent to vertex D, we finish that loop and go back to the top. The queue is not empty, so we remove vertex E from the front of the queue and examine its adjacency list. We have Adj(E) = {A, B, F}. The work matrices are shown below.



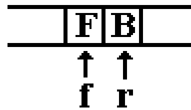
Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	0	2	3	4	5	0	0	0	0
Back	0	0	A	A	A	C	0	0	0	0

Copyright © by Edward Bosworth

v = E, Adj(v) = {A, B, F}, w = A
As vertex A is marked, we do not process it.

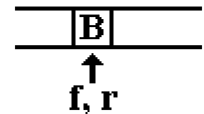
v = E, Adj(v) = {A, B, F}, w = B
As vertex B is not marked, we process it. Mark vertex B with the count, update the back array, and add vertex B to the queue. The situation after this has been done is as follows.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	6	2	3	4	5	0	0	0	0
Back	0	E	A	A	A	C	0	0	0	0



The state of the queue after vertex B has been added is shown at left.

v = E, Adj(v) = {A, B, F}, w = F
As vertex F is marked, we do not process it.



We have completed the work with vertices adjacent to vertex E. The loop begins again with the test showing the queue is not empty. Vertex F is removed from the queue, leaving the situation as at right. We examine the adjacency list of vertex F and note that Adj(F) = {B, C, E}. Each of these three vertices has been marked, so that the loop examining vertices adjacent to F does not mark any vertices or add them to the queue.

Having completed the work with vertices adjacent to vertex F, the loop returns again to the top. The queue is not empty, so the next vertex is removed. It is vertex B. We examine the adjacency list of vertex B and find it to be Adj(B) = {E, F}. As each of these vertices has been marked, neither is marked again or added to the queue, so we end this loop.

Having completed the work with vertices adjacent to vertex B, the loop returns again to the top. The queue is found to be empty, so the call to bfs(A) completes and control returns to the top level program BFS.

After the return from bfs(A), the status of the work arrays is as follows.

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	6	2	3	4	5	0	0	0	0
Back	0	E	A	A	A	C	0	0	0	0

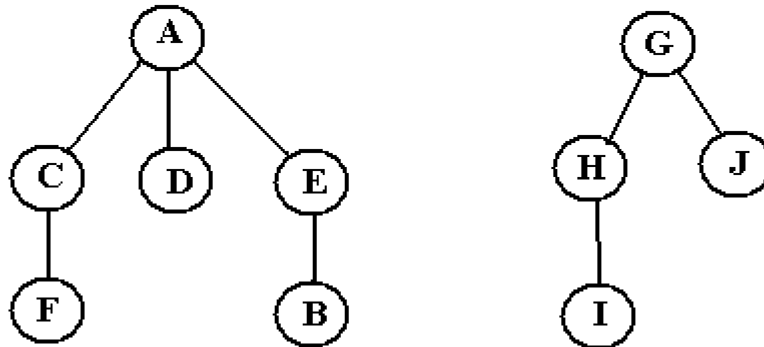
The next unmarked vertex in the list is G, so DFS next calls vertex G. (Terminology problem here – I have also used G for the graph name. I have never used the graph name in the context of algorithm dfs, so the call dfs(G) must be for vertex G.)

Copyright © by Edward Bosworth

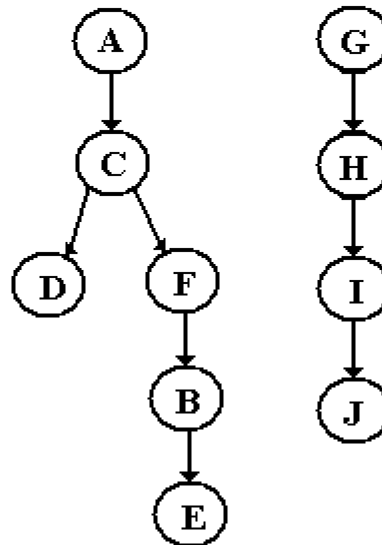
Vertex G has the following adjacency list: Adj(G) = {H, J}. The student should verify that the vertices are visited in the following order: H, J, and I; with vertex I being visited from vertex J. After the second (and last) component has been visited, the matrices are:

Vertex	A	B	C	D	E	F	G	H	I	J
Mark	1	6	2	3	4	5	7	8	10	9
Back	0	E	A	A	A	C	0	G	H	G

The search forest generated by BFS on the graph is shown below.



For comparison, the search forest generated by DFS on the graph is also shown.

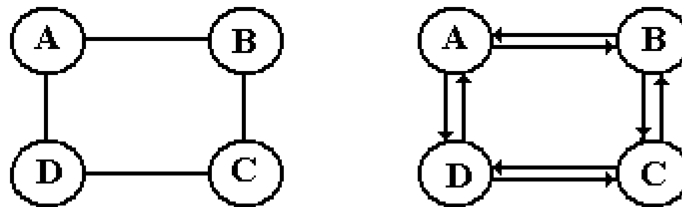


Copyright © by Edward Bosworth

Directed Graphs

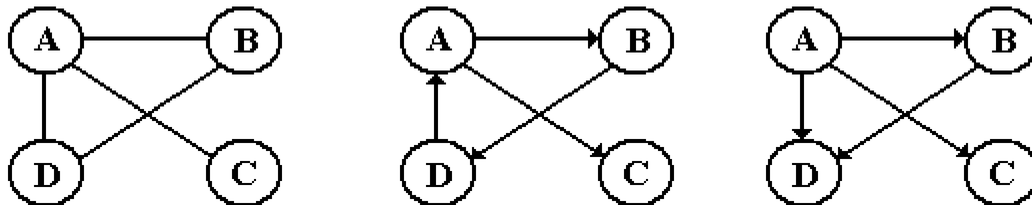
The next topic for discussion is **topological sort** (see below). Topological sort is a process applied to **directed graphs**, which we now discuss. Briefly stated, a directed graph is a graph in which the edges have direction – they should be viewed as "one-way streets".

We should first point out that every undirected graph may be represented as a directed graph. The following figure shows an undirected graph on four vertices (C_4 – the cycle on four vertices) and its equivalent as a directed graph. Note that each "street" is "two-way".



We have defined a **cycle** in a graph loosely as a path that begins at a given vertex, goes to at least one other vertex, and returns to the start vertex. A cycle that involves all other vertices is called a **Hamiltonian Cycle** – the subject of a lot of interesting research. Directed graphs can also contain cycles, but the paths have to "go with" the direction of the edges.

The next figure shows two similar directed graphs and the undirected graph equivalent.



Note that the graph in the middle has a single directed cycle, which could be denoted as (A, B, D). The graph on the right does not contain a cycle, as the edge (A, D) is reversed. The graph on the right is called a **directed acyclic graph** (or **dag** for short) as it is a directed graph without cycles. The graph on the left, being undirected, also contains a cycle.

The graph algorithms we have studied, including BFS and DFS, operate not directly on a graph but on its adjacency structure – either an adjacency matrix or an adjacency list. All of these algorithms can easily be applied to directed graphs.

	A	B	C	D
A	0	1	1	0
B	0	0	0	1
C	0	0	0	0
D	1	0	0	0

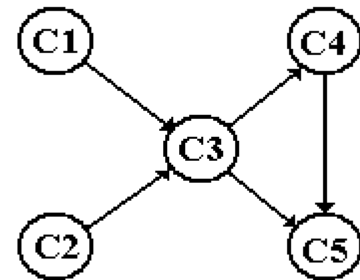
The adjacency matrix for the middle graph is shown at left. Note that the matrix is not symmetric, as are the adjacency matrices for undirected graphs. One uses this matrix in the expected way to apply graph algorithms, such as BFS and DFS.

Copyright © by Edward Bosworth

Topological Sort

We now turn to a process that can be applied only to **directed acyclic graphs**. This process, called **topological sorting**, is a presentation of the vertices of the graph in a way that preserves the ordering implied by the directions of the graph edges. The best example of topological sorting is a listing of a set of college courses, one per term, in a way that does not violate the prerequisite requirements.

Consider the set of course prerequisites represented by the graph below.

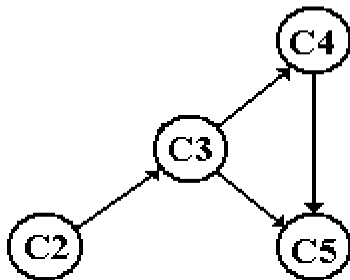


C1 and C2 are both prerequisites for C3.
C3 is a prerequisite for both C4 and C5.
C4 is a prerequisite for C5.

One way to view this problem is to consider **implicit prerequisites** – the implied prerequisites for course C5 are all of C1, C2, C3, and C4 – so course C5 must be taken last. By inspection, we see that the two possible orderings for these courses are

C1, C2, C3, C4, C5, and
C2, C1, C3, C4, C5.

We use topological sort to produce these results. There are two algorithms commonly used to do topological sorting – DFS and Source Removal. We present the Source Removal as it is simpler. First we define a **source vertex** as a vertex with no "incoming arrows". The algorithm functions by removing a source vertex and all edges incident to it, and placing the vertex in a list. In case of more than one source vertex, one chooses arbitrarily.

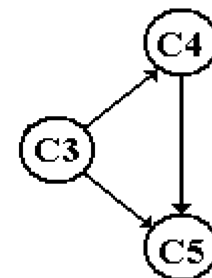


Applying the algorithm to the example, we note that we have two source vertices – C1 and C2. Here I arbitrarily pick vertex C1 and remove it, giving rise to the graph at left.

Our list at the moment contains one vertex (C1).

We look now and see that there is exactly one source vertex, C2. We remove it to get the next graph.

The list now is (C1, C2) and after removing these two vertices we have the graph at right. It should be obvious that there is only one source vertex – C3. We remove this to get the list (C1, C2, C3) and the algorithm proceeds to produce the list (C1, C2, C3, C4, C5). The reader should verify that, had we chosen to remove vertex C2 first, the list would have been (C2, C1, C3, C4, C5) as expected.



Copyright © by Edward Bosworth

<u>More Graph Examples: "Word Trail"</u>

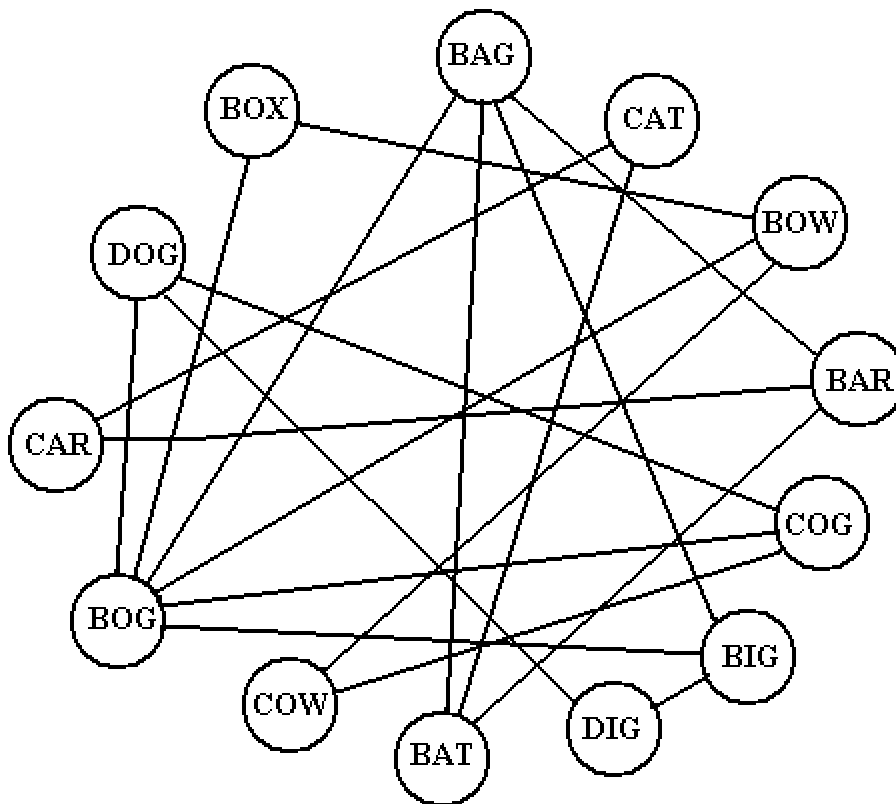
The word trail game is based on a common dictionary of words. One is supposed to move from one word to another in a finite number of steps, with each step representing the change of a single letter. A sample word trail would be DOG → DOT→ COT → CAT.

The common constraint is that the words in the trail be found in a usual dictionary. One excellent source of words would be the Scrabble™ dictionary. Under this usual constraint, the word trail DOG → DOT→ DAT → CAT would not be acceptable, as the word "DAT" is not found in most dictionaries.

We consider the word trail problem as an example of a graph search problem. In order to make the problem manageable, we restrict the dictionary of words to be used in the search to the following list of arbitrarily chosen three-letter words.

BAG BAT BOG BOX CAT COW DOG
BAR BIG BOW CAR COG DIG

We convert this list to a graph by assigning each word to a vertex, labeled by that word, and connecting two vertices by an edge if and only if the words differ by exactly one character.



The only issue with drawing this graph is to arrange the vertices in a manner that minimizes the confusion in the drawing. The figure above is this author's best try.

Copyright © by Edward Bosworth

One can use either DFS or BFS to discover the word trail. BFS turns out to be the better choice, as this search strategy will find the shortest path from one word to the other. DFS will always find a path, but the first path found will not necessarily be the best.

In order to apply BFS to this problem, we make a minor modification to mark the vertices with the "distance" to the source vertex – the number of edges traversed. The modified algorithm is shown below.

```
Algorithm bfs_trail(x, y)
//
// Uses breadth-first search to create a "word trail" from
// vertex x to vertex y.
//
  For each vertex v in V(G) Do
    Mark[v] = - 1       // A distance of 0 is a valid mark
    Back[v] =   0       // for a visited vertex, so we must
  End Do                // initialize each to -1.
//
  Add(Q, x)          // Add this vertex to the queue and
  mark [x] = 0       // set its distance to zero.
// Point 1
  While ( Not IsEmpty(Q)) Do
    Remove (Q, v)    // Remove the front vertex from queue
    DistToThis = Mark[v]
//   Point 2
    For each vertex w in V adjacent to v Do
//        Point 3
        If (Mark[w] < 0) Then
             Mark[w] = DistToThis + 1
             Back[w] = v
             Add(Q, w)
        End If

        If (w == y) then exit algorithm    // Done

    End For Each
//
//   Note that I remove the vertex at the top of the loop
//
  End While
```

Note that the algorithm can be easily modified to discover the distance from any source vertex to all other vertices in the connected component.

We now do the BFS starting at vertex DOG and see when we get to vertex CAT.

Copyright © by Edward Bosworth

First we create and initialize the Mark and Back arrays.

	BAG	BAR	BAT	BIG	BOG	BOW	BOX	CAR	CAT	COG	COW	DIG	DOG
Mark	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Back	0	0	0	0	0	0	0	0	0	0	0	0	0

dfs(DOG)
Add DOG to the Queue Q = (DOG)
Mark the array for DOG, setting its distance to 0.

	BAG	BAR	BAT	BIG	BOG	BOW	BOX	CAR	CAT	COG	COW	DIG	DOG
Mark	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0
Back	0	0	0	0	0	0	0	0	0	0	0	0	0

Remove DOG from the queue, so Q = () Dist = Mark[Dog] = 0
Adj(DOG) = (BOG, COG, DIG)
Mark[BOG] < 0, so
Mark[BOG] = 1
Back[BOG] = DOG
Add BOG to the queue Q = (BOG)

Mark[COG] < 0, so
Mark[COG] = 1
Back[COG] = DOG
Add COG to the queue Q = (BOG, COG)

Mark[DIG] < 0, so
Mark[DIG] = 1
Back[DIG] = DOG
Add DIG to the queue Q = (BOG, COG, DIG)
No more adjacent to DOG

The status now is:

	BAG	BAR	BAT	BIG	BOG	BOW	BOX	CAR	CAT	COG	COW	DIG	DOG
Mark	-1	-1	-1	-1	1	-1	-1	-1	-1	1	-1	1	0
Back	0	0	0	0	DOG	0	0	0	0	DOG	0	DOG	0

Q = (BOG, COG, DIG)

Remove BOG from the queue, so Q = (COG, DIG) Dist = Mark[BOG] = 1
Adj(BOG) = (BAG, BIG, BOW, BOX, COG, DOG)
Mark[BAG] < 0, so
Mark[BAG] = 2
Back[BAG] = BOG
Add BAG to the queue Q = (COG, DIG, BAG)

Copyright © by Edward Bosworth

Mark[BIG] < 0, so
Mark[BIG] = 2
Back[BIG] = BOG
Add BIG to the queue, so Q = (COG, DIG, BAG, BIG)

Mark[BOW] < 0, so
Mark[BOW] = 2
Back[BOW] = BOG
Add BOW to the queue, so Q = (COG, DIG, BAG, BIG, BOW)

Mark[BOX] < 0, so
Mark[BOX] = 2
Back[BOX] = BOG
Add BOX to the queue, so Q = (COG, DIG, BAG, BIG, BOW, BOX)

Mark[COG] = 1, so no action.

Mark[DOG] = 0, so no action.

No more adjacent to BOG

The status now is

	BAG	BAR	BAT	BIG	BOG	BOW	BOX	CAR	CAT	COG	COW	DIG	DOG
Mark	2	-1	-1	2	1	2	2	-1	-1	1	-1	1	0
Back	BOG	0	0	BOG	DOG	BOG	BOG	0	0	DOG	0	DOG	0

Q = (COG, DIG, BAG, BIG, BOW, BOX)

Remove COG from the queue, so Q = (DIG, BAG, BIG, BOW, BOX)
Adj(COG) = (BOG, COW, DOG) Dist = Mark[COG] = 1
Mark[BOG] = 1, so no action

Mark[COW] < 0, so
Mark[COW] = 2
Back[COW] = COG
Add COW to the queue, so Q = (DIG, BAG, BIG, BOW, BOX, COW)

Mark[DOG] = 0, so no action.

No more adjacent to COG.

The status now is

	BAG	BAR	BAT	BIG	BOG	BOW	BOX	CAR	CAT	COG	COW	DIG	DOG
Mark	2	-1	-1	2	1	2	2	-1	-1	1	2	1	0
Back	BOG	0	0	BOG	DOG	BOG	BOG	0	0	DOG	COG	DOG	0

Q = (DIG, BAG, BIG, BOW, BOX, COW)

Copyright © by Edward Bosworth

Remove DIG from the queue, so Q = (BAG, BIG, BOW, BOX, COW)
    Adj(DIG) = (BIG, DOG)

        Mark[BIG] = 2, so no action.

        Mark[DOG] = 0, so no action.

    No more adjacent to DIG. No change in status.

Remove BAG from the queue, so Q = (BIG, BOW, BOX, COW)
    Adj(BAG) = (BAR, BAT, BIG, BOG)   Dist = Mark[BAG] = 2

        Mark[BAR] < 0, so
            Mark[BAR] = 3
            Back[BAR] = BAG
            Add BAR to the queue, so Q = (BIG, BOW, BOX, COW, BAR)

        Mark[BAT] < 0, so
            Mark[BAT] = 3
            Back[BAT] = BAG
            Add BAT to the queue, so Q = (BIG, BOW, BOX, COW, BAR, BAT)

        Mark[BIG] = 2, so no action

        Mark[BOG] = 1, so no action.

    No more adjacent to BAG.
At this point, an intelligent search would note that "BAT" differs from "CAT" by exactly one letter and move "BAT" to the front of the queue. BFS does not do this.

The status now is

	BAG	BAR	BAT	BIG	BOG	BOW	BOX	CAR	CAT	COG	COW	DIG	DOG
Mark	2	3	3	2	1	2	2	-1	-1	1	2	1	0
Back	BOG	BAG	BAG	BOG	DOG	BOG	BOG	0	0	DOG	COG	DOG	0

    Q = (BIG, BOW, BOX, COW, BAR, BAT)

Remove BIG from the queue, so Q = (BOW, BOX, COW, BAR, BAT)
    Adj(BIG) = (BAG, BOG, DIG)

        Mark[BAG] = 2, so no action.

        Mark[BOG] = 1, so no action.

        Mark[DIG] = 1, so no action.

    No more adjacent to BIG. No change in status.

Copyright © by Edward Bosworth

Remove BOW from the queue, so Q = (BOX, COW, BAR, BAT)
Adj(BOW) = (BOG, BOX, COW)

Mark[BOG] = 1, so no action

Mark[BOX] = 2, so no action.

Mark[COW] = 2, so no action.

No more adjacent to BOW. No change in status.

Remove BOX from the queue, so Q = (COW, BAR, BAT)
Adj(BOX) = (BOG, BOW)

Mark[BOG] = 1, so no action.

Mark[BOW] = 2, so no action.

No more adjacent to BOX. No change in status.

Remove COW from the queue, so Q = (BAR, BAT)
Adj(COW) = (BOW, COG)

Mark[BOW] = 2, so no action.

Mark[COG] = 1, so no action.

No more adjacent to COW. No change in status.

Remove BAR from the queue, so Q = (BAT)
Adj(BAR) = (BAG, BAT, CAR) Dist = Mark[BAR] = 3

Mark[BAG] = 2, so no action.

Mark[BAT] = 3, so no action

Mark[CAR] < 0, so
Mark[CAR] = 4
Back[CAR] = BAR
Add CAR to the queue, so Q = (BAT, CAR)

The status now is

	BAG	BAR	BAT	BIG	BOG	BOW	BOX	CAR	CAT	COG	COW	DIG	DOG
Mark	2	3	3	2	1	2	2	4	-1	1	2	1	0
Back	BOG	BAG	BAG	BOG	DOG	BOG	BOG	BAR	0	DOG	COG	DOG	0

Q = (BAT, CAR)

Copyright © by Edward Bosworth

Remove BAT from the queue, so Q = (CAR)
Adj(BAT) = (BAG, BAR, CAT) Dist = Mark[BAT] = 3

Mark[BAG] = 2, so no change.

Mark[BAR] = 3, so no change.

Mark[CAT] < 0, so
Mark[CAT] = 4
Back[CAT] = BAT
Add CAT to the queue, so Q = (CAR, CAT)

No more adjacent to BAT.

At this point the word trail is complete.

The status now is

	BAG	BAR	BAT	BIG	BOG	BOW	BOX	CAR	CAT	COG	COW	DIG	DOG
Mark	2	3	3	2	1	2	2	4	4	1	2	1	0
Back	BOG	BAG	BAG	BOG	DOG	BOG	BOG	BAR	BAT	DOG	COG	DOG	0

Q = (CAR, CAT)

The word trail is CAT ← BAT ← BAG ← BOG ← DOG, which can be rewritten in the expected order as DOG → BOG → BAG → BAT → CAT.

Just to be complete, we finish the BFS.

Remove CAR from the queue, so Q = (CAT)
Adj(CAR) = (BAR, CAT)

Mark[BAR] = 3, so no action.

Mark[CAT] = 4, so no action.

No more adjacent to CAR. No change in status.

Remove CAT from the queue, so Q = ()
Adj(CAT) = (BAT, CAR)

Mark[BAT] = 3, so no action.

Mark[CAR] = 4, so no action.

No more adjacent to CAT. No change in status.
The queue is empty, so BFS terminates with the status as shown above.

Copyright © by Edward Bosworth