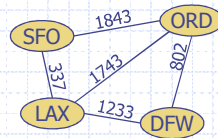


Lecture 12: Graphs and Graph Traversal

Principle of
Transcending



Depth- and Breadth-First Search

1

Wholeness Statement

Graphs have many useful applications in different areas of computer science. However, to be useful we have to be able to traverse them. There are two primary ways that graphs are systematically explored, either using depth-first or breadth-first search. The TM technique provides a simple, effortless way to systematically explore the different levels of the conscious mind until the process of thinking is transcended and unbounded silence is experienced; this is the field of wholeness of individual and cosmic intelligence.

Depth- and Breadth-First Search

2

Graphs Outline and Reading

- ◆ Graphs (§6.1)
 - Definition
 - Applications
 - Terminology
 - Properties
 - ADT
- ◆ Data structures for graphs (§6.2)
 - Edge list structure
 - Adjacency list structure
 - Adjacency matrix structure

Depth- and Breadth-First Search

3

Depth-First Search Outline and Reading

- ◆ Definitions (§6.1)
 - Subgraph
 - Connectivity
 - Spanning trees and forests
- ◆ Depth-first search (§6.3.1)
 - Algorithm
 - Example
 - Properties
 - Analysis
- ◆ Applications of DFS (§6.5)
 - Path finding
 - Cycle finding



Depth- and Breadth-First Search

4

Breadth-First Search Outline and Reading

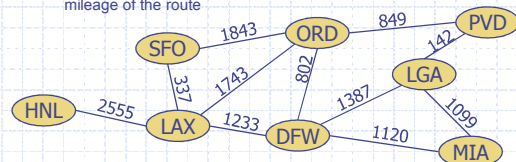
- ◆ Breadth-first search (§6.3.3)
 - Algorithm
 - Example
 - Properties
 - Analysis
 - Applications
- ◆ DFS vs. BFS (§6.3.3)
 - Comparison of applications
 - Comparison of edge labels

Depth- and Breadth-First Search

5

Graph

- ◆ A graph is a pair (V, E) , where
 - V is a set of nodes, called **vertices**
 - E is a collection of pairs of vertices, called **edges**
 - Vertices and edges are positions and store elements
- ◆ Example:
 - A vertex represents an airport and stores the three-letter airport code
 - An edge represents a flight route between two airports and stores the mileage of the route



Depth- and Breadth-First Search

6

Edge Types

- ◆ **Directed edge**
 - ordered pair of vertices (u,v)
 - first vertex u is the *origin*
 - second vertex v is the *destination*
 - e.g., a flight
- ◆ **Undirected edge**
 - unordered pair of vertices (u,v)
 - e.g., a flight route
- ◆ **Directed graph**
 - all the edges are directed
 - e.g., flight network
- ◆ **Undirected graph**
 - all the edges are undirected
 - e.g., route network

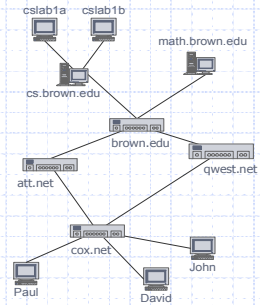


Depth- and Breadth-First Search

7

Applications

- ◆ Electronic circuits
 - Printed circuit board
 - Integrated circuit
- ◆ Transportation networks
 - Highway network
 - Flight network
- ◆ Computer networks
 - Local area network
 - Internet
 - Web
- ◆ Databases
 - Entity-relationship diagram

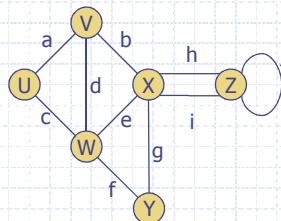


Depth- and Breadth-First Search

8

Terminology

- ◆ **End vertices (or endpoints)**
 - the 2 vertices joined by an edge
 - U and V are the endpoints of a
- ◆ **Vertices are adjacent**
 - if they are endpoints of the same edge
 - U and V are adjacent
- ◆ **Edge is incident on a vertex**
 - if the vertex is one of the edge's endpoints
 - a, d, and b are incident on V
- ◆ **Degree of a vertex**
 - number of incident edges
 - X has degree 5
- ◆ h and i are parallel edges
- ◆ j is a self-loop
- ◆ A **Simple Graph** has no parallel edges or self-loops
 - We will assume graphs are simple

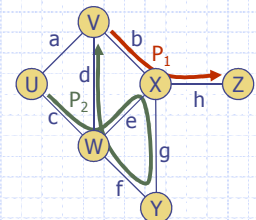


Depth- and Breadth-First Search

9

Terminology (cont.)

- ◆ **Path**
 - sequence of alternating vertices and edges
 - begins with a vertex
 - ends with a vertex
 - each edge is preceded and followed by its endpoints
- ◆ **Simple path**
 - path such that all its vertices and edges are distinct
- ◆ **Examples**
 - $P_1 = (V, b, X, h, Z)$ is a simple path
 - $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ is a path that is not simple

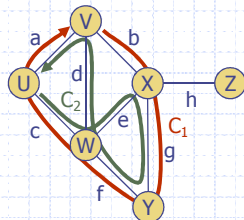


Depth- and Breadth-First Search

10

Terminology (cont.)

- ◆ **Cycle**
 - circular sequence of alternating vertices and edges
 - i.e., a path with the same start and end vertices
- ◆ **Simple cycle**
 - cycle such that all its vertices and edges are distinct
 - i.e., the path is simple
- ◆ **Examples**
 - $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ is a simple cycle
 - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ is a cycle that is not simple



Depth- and Breadth-First Search

11

List of Terms

- ◆ Graph
 - Vertex, vertices
 - End vertices
 - Adjacent vertices
 - Degree of a vertex
- ◆ Edges
 - Incident edges
 - Directed edge, undirected edge
 - Directed graph, undirected graph, mixed graph
- ◆ Path, simple path
- ◆ Cycle, simple cycle

Depth- and Breadth-First Search

12

Main Point

1. A path in a graph is a sequence of alternating vertices and edges, starting with a vertex and ending with a vertex. A path is simple if all its vertices and edges are distinct.
The path to enlightenment is simple: regular practice of the TM technique and a balanced daily routine to stabilize the gains during meditation.

Depth- and Breadth-First Search

13

Properties

Property 1

$$\sum_v \deg(v) = 2m$$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no parallel edges

$$m \leq n(n-1)/2$$

Proof: each vertex has degree at most $(n-1)$

What is the bound for a directed graph?

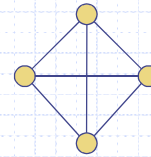
$$m \leq n(n-1)$$

Notation

n number of vertices
 m number of edges
 $\deg(v)$ degree of vertex v

Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$



Depth- and Breadth-First Search

14

Main Methods of the Graph ADT

- ◆ Vertices and edges
 - are Positions
 - store elements
- ◆ Accessor methods
 - `aVertex()`
 - `incidentEdges(v)`
 - `endVertices(e)`
 - `isDirected(e)`
 - `origin(e)`
 - `destination(e)`
 - `opposite(v, e)`
 - `areAdjacent(v, w)`
- ◆ Update methods
 - `insertVertex(o)`
 - `insertEdge(v, w, o)`
 - `insertDirectedEdge(v, w, o)`
 - `removeVertex(v)`
 - `removeEdge(e)`
- ◆ Generic methods
 - `numVertices()`
 - `numEdges()`
 - `vertices()`
 - `edges()`
 - `degree(v)`

Depth- and Breadth-First Search

15

Graph Data Structures

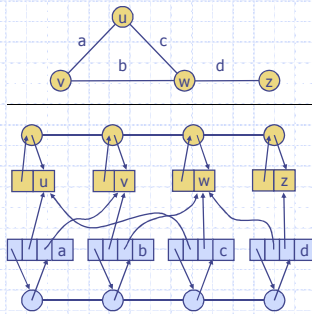
Edge list
Adjacency list
Adjacency matrix

Depth- and Breadth-First Search

16

Edge List Structure

- ◆ Vertex object
 - element
 - reference to position in vertex sequence
- ◆ Edge object
 - element
 - origin vertex object
 - destination vertex object
 - reference to position in edge sequence
- ◆ Vertex sequence
 - sequence of vertex objects
- ◆ Edge sequence
 - sequence of edge objects

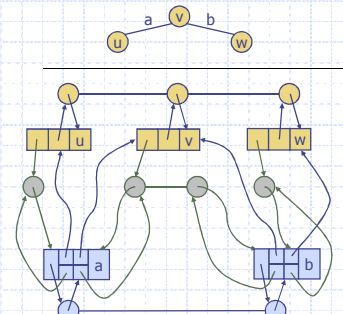


Depth- and Breadth-First Search

17

Adjacency List Structure

- ◆ Edge list structure
- ◆ Incidence sequence for each vertex
 - sequence of references to edge objects of incident edges
- ◆ Augmented edge objects
 - references to associated positions in incidence sequences of end vertices

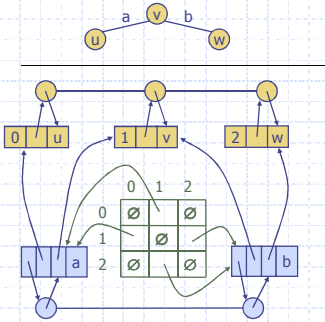


Depth- and Breadth-First Search

18

Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
 - Integer key (index) associated with vertex
- 2D adjacency array
 - Reference to edge object for adjacent vertices
 - Null for nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge



Depth- and Breadth-First Search

19

Asymptotic Performance

- n vertices, m edges
- no parallel edges
- no self-loops
- Bounds are "big-Oh"

Space

`incidentEdges(v)`

`areAdjacent(v, w)`

`insertVertex(o)`

`insertEdge(v, w, o)`

`removeVertex(v)`

`removeEdge(e)`

Edge
List

Adjacency
List

Adjacency
Matrix

Depth- and Breadth-First Search

20

Asymptotic Performance

- n vertices, m edges
- no parallel edges
- no self-loops
- Bounds are "big-Oh"

	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$		
<code>incidentEdges(v)</code>	m		
<code>areAdjacent(v, w)</code>	m		
<code>insertVertex(o)</code>	1		
<code>insertEdge(v, w, o)</code>	1		
<code>removeVertex(v)</code>	m		
<code>removeEdge(e)</code>	1		

Depth- and Breadth-First Search

21

Asymptotic Performance

- n vertices, m edges
- no parallel edges
- no self-loops
- Bounds are "big-Oh"

	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	
<code>incidentEdges(v)</code>	m	$\deg(v)$	
<code>areAdjacent(v, w)</code>	m	$\min(\deg(v), \deg(w))$	
<code>insertVertex(o)</code>	1	1	
<code>insertEdge(v, w, o)</code>	1	1	
<code>removeVertex(v)</code>	m	$\deg(v)$	
<code>removeEdge(e)</code>	1	1	

Depth- and Breadth-First Search

22

Asymptotic Performance

- n vertices, m edges
- no parallel edges
- no self-loops
- Bounds are "big-Oh"

	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
<code>incidentEdges(v)</code>	m	$\deg(v)$	n
<code>areAdjacent(v, w)</code>	m	$\min(\deg(v), \deg(w))$	1
<code>insertVertex(o)</code>	1	1	n^2
<code>insertEdge(v, w, o)</code>	1	1	1
<code>removeVertex(v)</code>	m	$\deg(v)$	n^2
<code>removeEdge(e)</code>	1	1	1

Depth- and Breadth-First Search

23

Asymptotic Performance

- n vertices, m edges
- no parallel edges
- no self-loops
- Bounds are "big-Oh"

	Edge List	Adjacency List	Adjacency Matrix
<code>aVertex()</code>			
<code>edges()</code>			
<code>vertices()</code>			
<code>endVertices(e)</code>			
<code>opposite(v, e)</code>			
<code>degree(v)</code>			
<code>numEdges()</code>			

Depth- and Breadth-First Search

24

Asymptotic Performance

- ◆ n vertices, m edges
- ◆ no parallel edges
- ◆ no self-loops
- ◆ Bounds are "big-Oh"

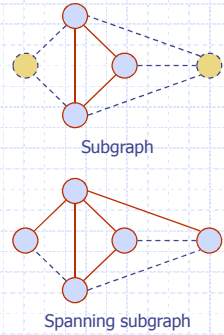
	Edge List	Adjacency List	Adjacency Matrix
<code>aVertex()</code>	1	1	1
<code>edges()</code>	m	m	m
<code>vertices()</code>	n	n	n
<code>endVertices(e)</code>	1	1	1
<code>opposite(v, e)</code>	1	1	1
<code>degree(v)</code>	m	1	n
<code>numEdges()</code>	1	1	1

Depth- and Breadth-First Search

25

Subgraphs

- ◆ A subgraph S of a graph G is a graph such that
 - $\text{vertices}(S) \subseteq \text{vertices}(G)$
 - $\text{edges}(S) \subseteq \text{edges}(G)$
- ◆ A spanning subgraph of G is a subgraph that contains all the vertices of G , i.e., $\text{vertices}(S) = \text{vertices}(G)$

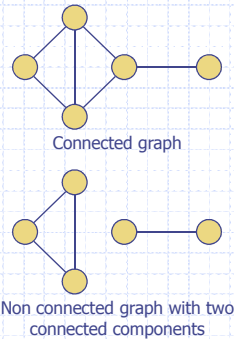


Depth- and Breadth-First Search

26

Connectivity

- ◆ Two vertices are *connected* if there is a path between them
- ◆ A graph is *connected* if there is a path between every pair of vertices
- ◆ A *connected component* of a graph G is a maximal connected subgraph of G



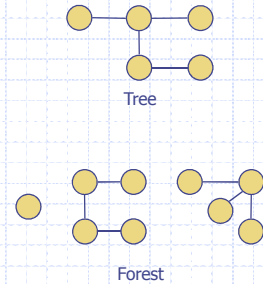
Depth- and Breadth-First Search

27

Trees and Forests

- ◆ A (free) *tree* is an **undirected** graph T such that
 - T is **connected**
 - T has no **cycles**

This definition is different from the definition of a rooted tree
- ◆ A *forest* is an undirected graph without cycles
- ◆ The connected components of a forest are trees

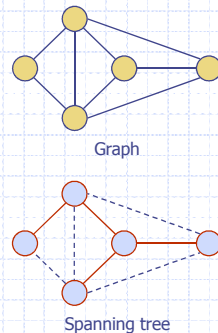


Depth- and Breadth-First Search

28

Spanning Trees and Forests

- ◆ A spanning tree of a connected graph is a spanning subgraph that is a tree
- ◆ A spanning tree is not unique unless the graph is a tree
- ◆ Spanning trees have applications to the design of communication networks
- ◆ A spanning forest of a graph is a spanning subgraph that is a forest



Depth- and Breadth-First Search

29

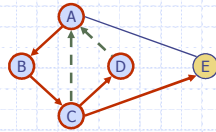
More Terms

- ◆ Subgraph
- ◆ Connectivity
 - Connected Vertices (path between them)
 - Connected Graph (all vertices are connected)
 - Connected Component (maximal connected subgraph)
- ◆ Tree (connected, no cycles)
- ◆ Forest (one or more trees)
- ◆ Spanning Tree and Spanning Forest

Depth- and Breadth-First Search

30

Depth-First Search



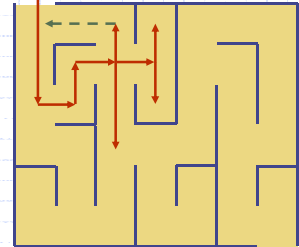
Depth- and Breadth-First Search

31

DFS and Maze Traversal

✦ The DFS algorithm is similar to a classic strategy for exploring a maze

- We mark each intersection, corner and dead end (vertex) visited
- We mark each corridor (edge) traversed
- We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)

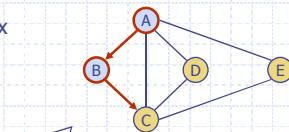
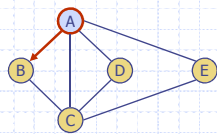


Depth- and Breadth-First Search

32

Depth-First Search Example

- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- - - back edge

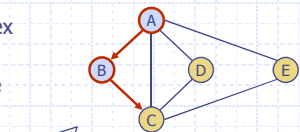
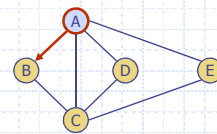


Depth- and Breadth-First Search

33

Example

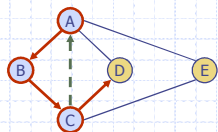
- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- - - back edge



Depth- and Breadth-First Search

34

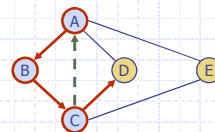
Example (cont.)



Depth- and Breadth-First Search

35

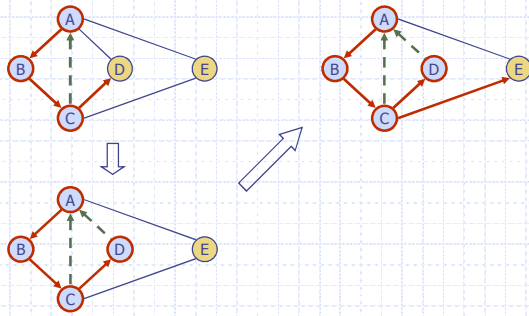
Example (cont.)



Depth- and Breadth-First Search

36

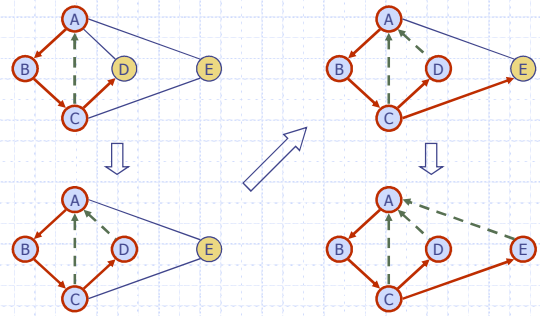
Example (cont.)



Depth- and Breadth-First Search

37

Example (cont.)



Depth- and Breadth-First Search

38

DFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm $DFS(G)$
Input graph G
Output labeling of the edges of G as discovery edges and back edges
for all $u \in G.vertices()$ **do**
 $setLabel(u, UNEXPLORED)$
for all $e \in G.edges()$ **do**
 $setLabel(e, UNEXPLORED)$
for all $v \in G.vertices()$ **do**
 if $getLabel(v) = UNEXPLORED$
 $DFS(G, v)$

Algorithm $DFS(G, v)$
Input graph G and a start vertex v of G
Output labeling of the edges of G in the connected component of v as discovery edges and back edges
 $setLabel(v, VISITED)$
for all $e \in G.incidentEdges(v)$ **do**
 if $getLabel(e) = UNEXPLORED$
 $w \leftarrow G.opposite(v, e)$
 if $getLabel(w) = UNEXPLORED$
 $setLabel(e, DISCOVERY)$
 $DFS(G, w)$
 else
 $setLabel(e, BACK)$

Depth- and Breadth-First Search

39

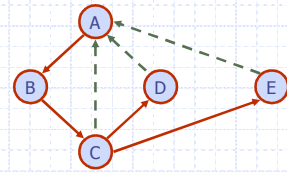
Properties of DFS

Property 1

$DFS(G, v)$ visits all the vertices and edges in the connected component of v

Property 2

The discovery edges labeled by $DFS(G, v)$ form a spanning tree of the connected component of v



Depth- and Breadth-First Search

40

Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
 - once as UNEXPLORED
 - once as VISITED
- Each edge is labeled twice
 - once as UNEXPLORED
 - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
 - Recall that $\sum_v \deg(v) = 2m$

Depth- and Breadth-First Search

41

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
 - Visits all the vertices and edges of G
 - Determines whether G is connected
 - Computes the connected components of G
 - Computes a spanning forest of G

Depth- and Breadth-First Search

42

Depth-First Search

- ◆ DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- ◆ DFS can be further extended to solve other graph problems
 - Find and report a path between two given vertices
 - Find a cycle in the graph
- ◆ Depth-first search is to graphs what the Euler tour is to binary trees

Path Finding

- ◆ We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- ◆ We call $DFS(G, u)$ with u as the start vertex
- ◆ We use a stack S to keep track of the path between the start vertex and the current vertex
- ◆ As soon as destination vertex z is encountered, we return the path as the contents of the stack

Algorithm $pathDFS(G, v, z)$
 $setLabel(v, VISITED)$
 $S.push(v)$
if $v = z$
 $path \leftarrow S.elements()$
for all $e \in G.incidentEdges(v)$ **do**
 if $getLabel(e) = UNEXPLORED$
 $w \leftarrow opposite(v, e)$
 if $getLabel(w) = UNEXPLORED$
 $setLabel(e, DISCOVERY)$
 $S.push(e)$
 $pathDFS(G, w, z)$
 $S.pop()$ { e gets popped }
 else
 $setLabel(e, BACK)$
 $S.pop()$ { v gets popped }

Cycle Finding

- ◆ We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- ◆ We use a stack S to keep track of the path between the start vertex and the current vertex
- ◆ As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

Algorithm $cycleDFS(G, v)$
 $setLabel(v, VISITED)$
 $S.push(v)$
if $cycle \leftarrow null$ **then return**
for all $e \in G.incidentEdges(v)$
 if $getLabel(e) = UNEXPLORED$
 $w \leftarrow opposite(v, e)$
 $S.push(e)$
 if $getLabel(w) = UNEXPLORED$
 $setLabel(e, DISCOVERY)$
 $cycleDFS(G, w)$
 $S.pop()$
 else
 $cycle \leftarrow$ new empty sequence
 $o \leftarrow w$
 repeat
 $cycle.insertLast(o)$
 $o \leftarrow S.pop()$
 until $o = v$
 $setLabel(e, BACK)$
 $S.pop()$

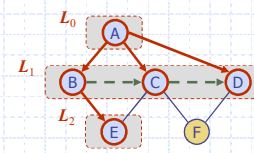
Recursive Programs

- ◆ The call structure can be described as a depth-first search of a rooted tree
 - Each non-root vertex corresponds to a recursive call
 - A tree is a logical construct, not an explicit data structure

Main Point

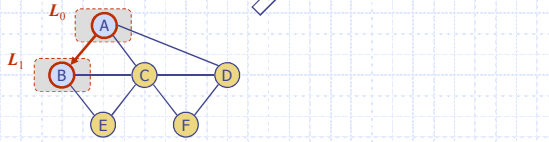
2. During depth-first search of a graph, each path is followed until the end is reached, then it backs up to branch out and explore new edges; all adjacent vertices are visited before backtracking.
 The mind is naturally seeking fields of greater happiness. The TM technique uses the nature of the mind to immediately and effortlessly take the mind to the deepest levels where true happiness and fulfillment can be gained.

Breadth-First Search



Example

- A unexplored vertex
- A visited vertex
- unexplored edge
- discovery edge
- - - cross edge



Depth- and Breadth-First Search

49

Example

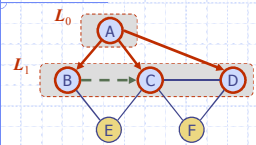
- A unexplored vertex
- A visited vertex
- unexplored edge
- discovery edge
- - - cross edge



Depth- and Breadth-First Search

50

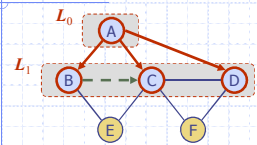
Example (cont.)



Depth- and Breadth-First Search

51

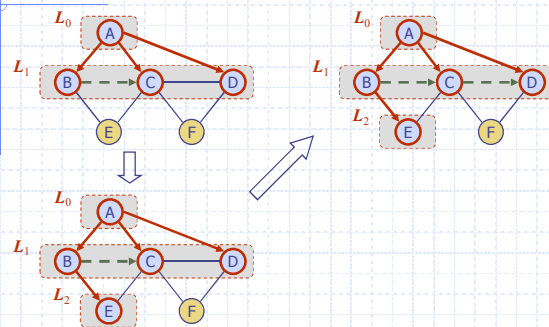
Example (cont.)



Depth- and Breadth-First Search

52

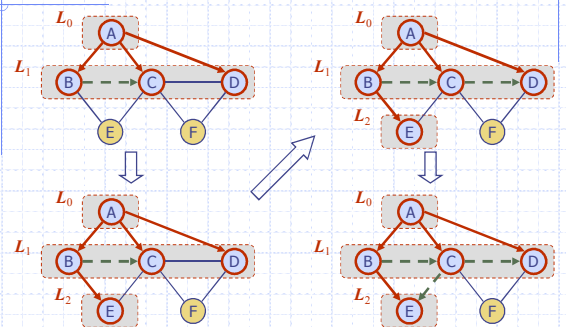
Example (cont.)



Depth- and Breadth-First Search

53

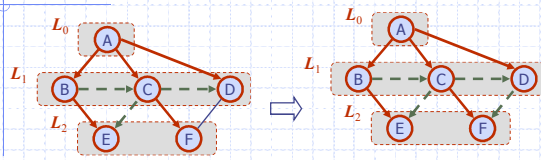
Example (cont.)



Depth- and Breadth-First Search

54

Example (cont.)



Depth- and Breadth-First Search

55

BFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm $BFS(G)$

Input graph G

Output labeling of the edges and partition of the vertices of G

```

for all  $u \in G.vertices()$ 
  setLabel( $u$ , UNEXPLORED)
for all  $e \in G.edges()$ 
  setLabel( $e$ , UNEXPLORED)
for all  $v \in G.vertices()$ 
  if getLabel( $v$ ) = UNEXPLORED
    BFS( $G$ ,  $v$ )
  
```

Algorithm $BFS(G, s)$

```

 $L_0 \leftarrow$  new empty sequence
 $L_0.insertLast(s)$ 
setLabel( $s$ , VISITED)
 $i \leftarrow 0$ 
while  $\neg L_i.isEmpty()$ 
   $L_{i+1} \leftarrow$  new empty sequence
  for all  $v \in L_i.elements()$ 
    for all  $e \in G.incidentEdges(v)$ 
      if getLabel( $e$ ) = UNEXPLORED
         $w \leftarrow opposite(v, e)$ 
        if getLabel( $w$ ) = UNEXPLORED
          setLabel( $e$ , DISCOVERY)
          setLabel( $w$ , VISITED)
           $L_{i+1}.insertLast(w)$ 
        else
          setLabel( $e$ , CROSS)
   $i \leftarrow i + 1$ 
  
```

Depth- and Breadth-First Search

56

Properties

Notation

G_s : connected component of s

Property 1

$BFS(G, s)$ visits all the vertices and edges of G_s

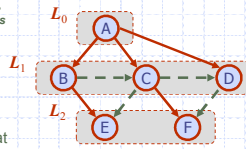
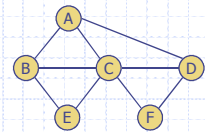
Property 2

The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges



Depth- and Breadth-First Search

57

Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
 - once as UNEXPLORED
 - once as VISITED
- Each edge is labeled twice
 - once as UNEXPLORED
 - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method `incidentEdges` is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
 - Recall that $\sum_v \deg(v) = 2m$

Depth- and Breadth-First Search

58

Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
 - Visits all the vertices and edges of G
 - Determines whether G is connected
 - Computes the connected components of G
 - Computes a spanning forest of G

Depth- and Breadth-First Search

59

Breadth-First Search

- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
 - Find and report a path with the minimum number of edges between two given vertices
 - Find a simple cycle, if there is one

Depth- and Breadth-First Search

60

Applications

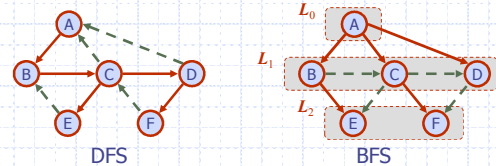
- ◆ Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
 - Compute the connected components of G
 - Compute a spanning forest of G
 - Find a simple cycle in G , or report that G is a forest
 - Given two vertices of G , find a path in G between them with the minimum number of edges, or report that no such path exists

Depth- and Breadth-First Search

61

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	✓	✓
Shortest paths		✓
Biconnected components	✓	



Depth- and Breadth-First Search

62

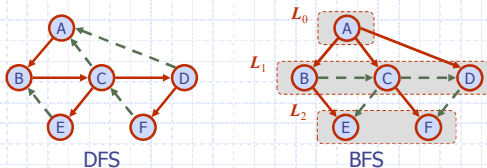
DFS vs. BFS (cont.)

Back edge (v, w)

- w is an ancestor of v in the tree of discovery edges

Cross edge (v, w)

- w is in the same level as v or in the next level in the tree of discovery edges



Depth- and Breadth-First Search

63

Main Point

3. During breadth-first search of a graph, the search repeatedly takes one step in all directions until all vertices and edges are visited. This is a bit like searching for fulfillment in waking state, i.e., floating on the surface of the mind or through one's daily activity. In contrast, Transcendental Meditation takes the mind immediately and effortlessly to the deepest levels where true fulfillment can be gained.

Depth- and Breadth-First Search

64

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. Almost any algorithm for solving a problem on a graph or digraph requires examining or processing each vertex or edge.
2. Depth-first and breadth-first search are two particularly useful and efficient search strategies requiring linear time.

Depth- and Breadth-First Search

65

3. **Transcendental Consciousness** is the goal of all searches, the field of complete fulfillment.
4. **Impulses within Transcendental Consciousness:** The dynamic natural laws within this unbounded field govern all activities and evolution of the universe.
5. **Wholeness moving within itself:** In Unity Consciousness, one experiences that the self-referral activity of the unified field gives rise to the whole breadth and depth of the universe.

Depth- and Breadth-First Search

66