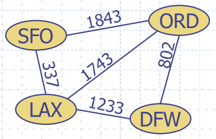
Lecture 15: Graphs and Graph Traversal

Principle of Transcending



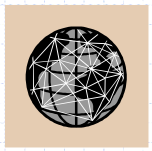
Wholeness Statement

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. *Science of Consciousness:* 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; contacting this field of wholeness of individual and cosmic intelligence benefits individual and society.

Graphs Outline and Reading

- Graphs
  - Definition
  - Applications
  - Terminology
  - Properties
  - ADT
- Data structures for graphs
  - Edge list structure
  - Adjacency list structure
  - Adjacency matrix structure

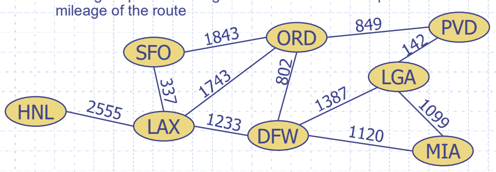
Depth-First Search Outline and Reading



- Definitions
  - Subgraph
  - Connectivity
  - Spanning trees and forests
- Depth-first search
  - Algorithm
  - Example
  - Properties
  - Analysis
- Applications of DFS
  - Path finding
  - Cycle finding

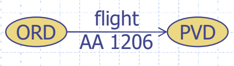
Graph

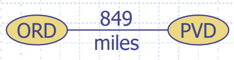
- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight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



Edge Types

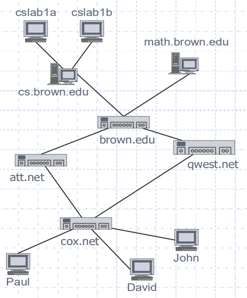
- *Directed edge*
  - ordered pair of vertices (u,v)
  - first vertex u is the *origin*
  - second vertex v is the *destination*
  - e.g., a flight
- *Undirected edge*
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- *Directed graph*
  - all the edges are directed
  - e.g., flight network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





Applications

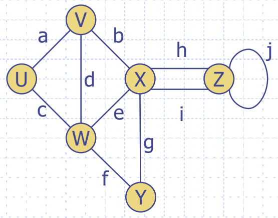
- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



7

Terminology

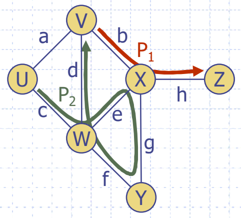
- *End vertices* (or *endpoints*)
  - the 2 vertices joined by an edge
  - U and V are the endpoints of **a**
- *Vertices are adjacent*
  - if they are endpoints of the same edge
  - U and V are adjacent
- Edge is *incident* on a vertex
  - if the vertex is one the edge's endpoints,
  - **a**, **d**, 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



8

Terminology (cont.)

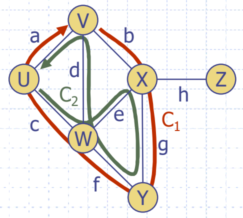
- *Path*
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- *Simple path*
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V)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



9

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



10

List of Terms

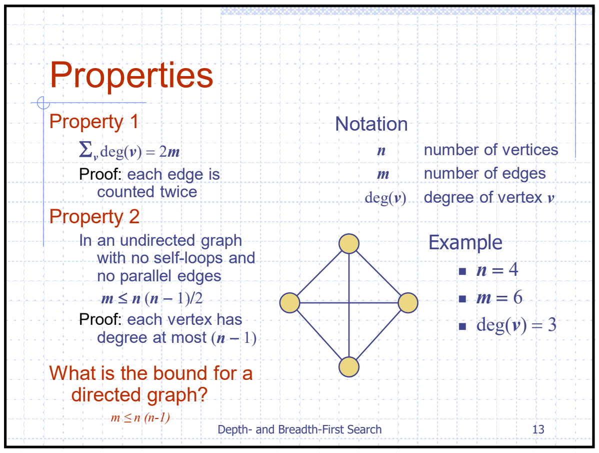
- Graph
- Vertex, vertices
- End vertices
- Adjacent vertices
- Degree of a vertex
- Edges
- Incident edges
- Directed edge, undirected edge
- Directed graph, undirected graph, mixed graph
- Path, simple path
- Cycle, simple cycle

11

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.
   Science of Consciousness: The path to enlightenment is simple: regular practice of the TM technique and a balanced daily routine to stabilize the gains during meditation.

12



Properties

Property 1

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

Proof: each edge is counted twice

Property 2

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

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

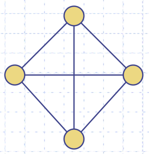
Proof: each vertex has degree at most $(n-1)$

What is the bound for a directed graph?

$m \leq n(n-1)$

Notation

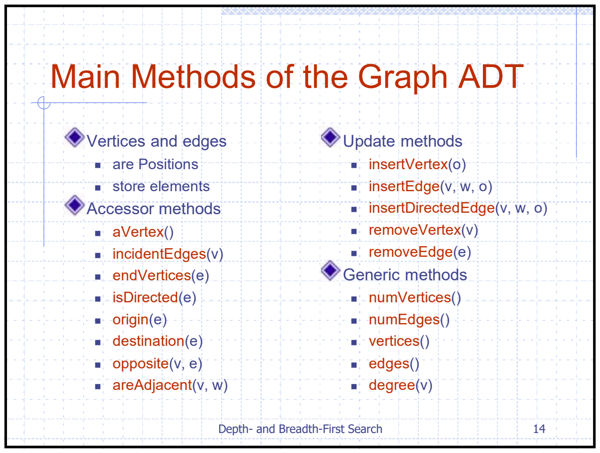
n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

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

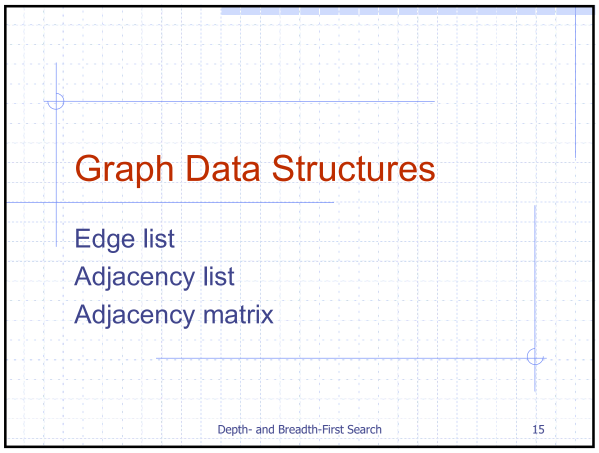
13



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)

14



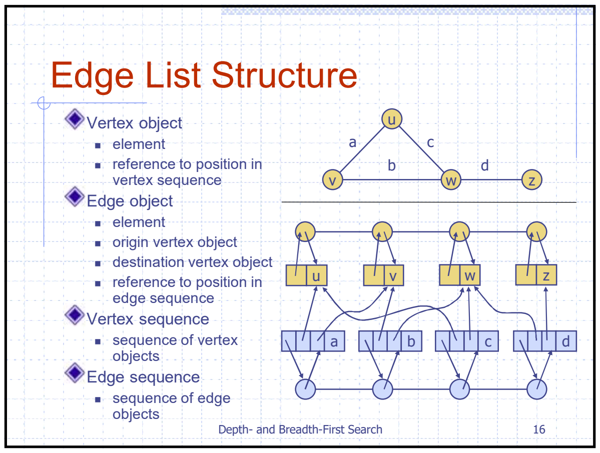
Graph Data Structures

Edge list

Adjacency list

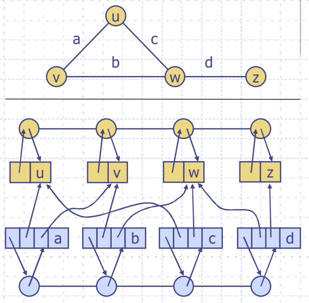
Adjacency matrix

15

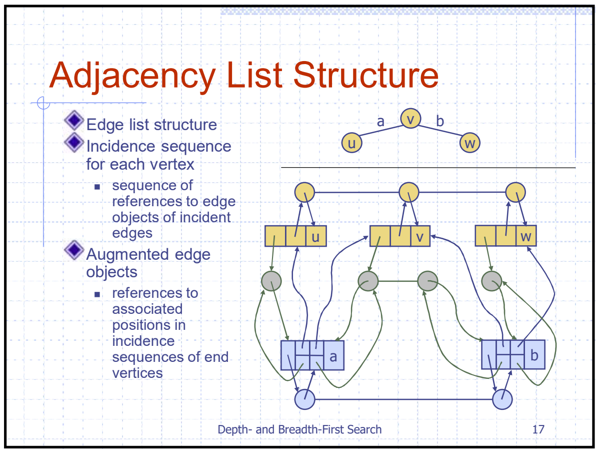


Edge List Structure

- 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

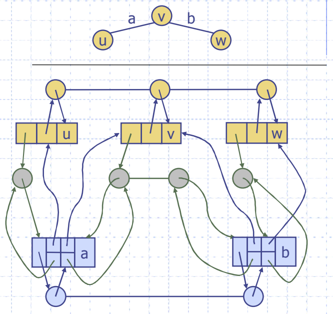


16

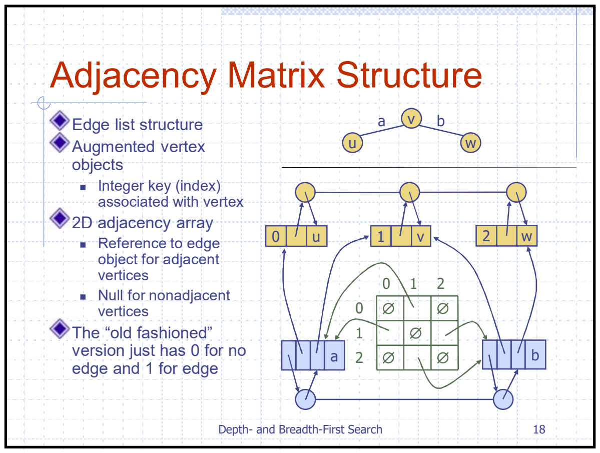


Adjacency List Structure

- 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

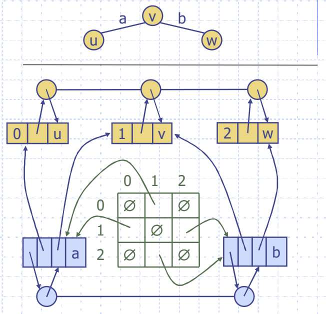


17



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



18

Asymptotic Performance

◆ n vertices, m edges ◆ no parallel edges ◆ no self-loops ◆ Bounds are "big-Oh"	Edge List	Adjacency List	Adjacency Matrix
Space			
incidentEdges(v)			
areAdjacent(v, w)			
insertVertex(o)			
insertEdge(v, w, o)			
removeVertex(v)			
removeEdge(e)			

19

Asymptotic Performance

◆ n vertices, m edges ◆ no parallel edges ◆ no self-loops ◆ Bounds are "big-Oh"	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$		
incidentEdges(v)	m		
areAdjacent(v, w)	m		
insertVertex(o)	1		
insertEdge(v, w, o)	1		
removeVertex(v)	m		
removeEdge(e)	1		

20

Asymptotic Performance

◆ n vertices, m edges ◆ no parallel edges ◆ no self-loops ◆ Bounds are "big-Oh"	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	
incidentEdges(v)	m	$\deg(v)$	
areAdjacent(v, w)	m	$\min(\deg(v), \deg(w))$	
insertVertex(o)	1	1	
insertEdge(v, w, o)	1	1	
removeVertex(v)	m	$\deg(v)$	
removeEdge(e)	1	1	

21

Asymptotic Performance

◆ n vertices, m edges ◆ no parallel edges ◆ no self-loops ◆ Bounds are "big-Oh"	Edge List	Adjacency List	Adjacency Matrix
aVertex()			
edges()			
vertices()			
endVertices(e)			
opposite(v, e)			
degree(v)			
numEdges()			

22

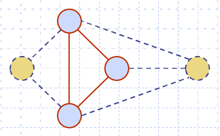
Asymptotic Performance

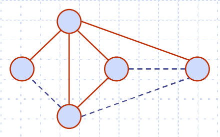
◆ n vertices, m edges ◆ no parallel edges ◆ no self-loops ◆ Bounds are "big-Oh"	Edge List	Adjacency List	Adjacency Matrix
aVertex()	1	1	
edges()	m	m	
vertices()	n	n	
endVertices(e)	1	1	
opposite(v, e)	1	1	
degree(v)	m	1	
numEdges()	1	1	

23

Subgraphs

- A subgraph S of a graph G is a graph such that
  - vertices(S) ⊆ vertices(G)
  - edges(S) ⊆ edges(G)
- A spanning subgraph of G is a subgraph that contains all the vertices of G, i.e., vertices(S) = vertices(G)

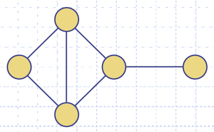

Subgraph

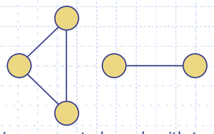

Spanning subgraph

24

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

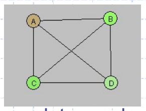

Connected graph


Non connected graph with two connected components

25

Complete Graphs

- A graph G is complete if for every pair of vertices (u,v), there is an edge (u,v) in G.
- This is the complete graph on 4 vertices, denoted K_4.



- In general, the complete graph on n vertices is denoted K_n.
- For a complete undirected graph G,

  $m = n(n-1)/2$

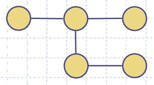
  that is to say, K_n has exactly $n(n-1)/2$ edges.

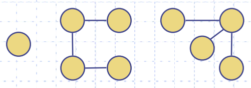
26

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

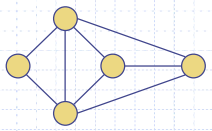

Tree

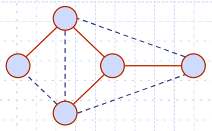

Forest

27

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


Graph


Spanning tree

28

More Terms

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

29

Main Point

2. A spanning tree connects all vertices of a graph without any cycles. A spanning forest is a subgraph in which each connected component is a spanning tree of the vertices in that component.
   Science of Consciousness: The pure field of consciousness connects everything in creation and governs everything through laws (algorithms). Contact with pure consciousness brings out the qualities of this field into our mind and body for the benefit of everyone.

30

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

31

More Terms

- Subgraph
- Connectivity
  - Connected Vertices
  - Connected Graph
  - Connected Component
  - Complete Component (or Graph)
- Tree
- Forest
- Spanning Tree and Spanning Forest

32

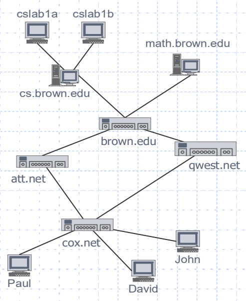
Depth-First Search Outline and Reading

- Definitions (Review)
  - Subgraph
  - Connectivity
  - Spanning trees and forests
- Depth-first search
  - Algorithm
  - Example
  - Properties
  - Analysis
- Applications of DFS
  - Path finding
  - Cycle finding

33

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



34

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

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

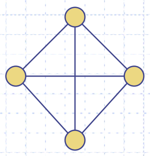
Proof: each vertex has degree at most $(n-1)$

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

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

35

Main Methods of the Undirected Graph ADT

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

36

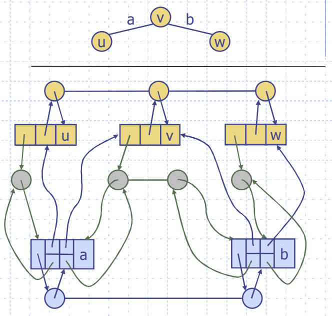
Graph Data Structures

Adjacency list implementation is the one we will assume

37

Recall: Adjacency List Structure

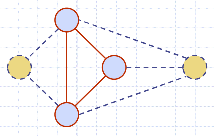
- Edge list structure
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices

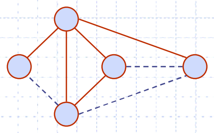


38

Subgraphs

- A subgraph S of a graph G is a graph such that
  - vertices(S) ⊆ vertices(G)
  - edges(S) ⊆ edges(G)
- A spanning subgraph of G is a subgraph that contains all the vertices of G, i.e., vertices(S) = vertices(G)

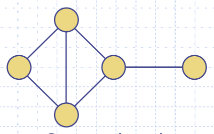

Subgraph

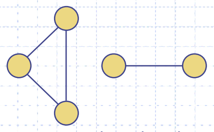

Spanning subgraph

39

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


Connected graph

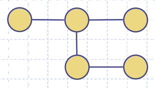

Non connected graph with two connected components

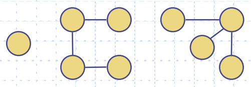
40

Trees and Forests

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

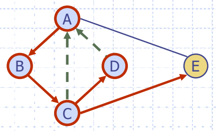
  This definition is different from the definition of a rooted tree
- A *forest* is an undirected graph without cycles
- The connected components of a forest are trees


Tree

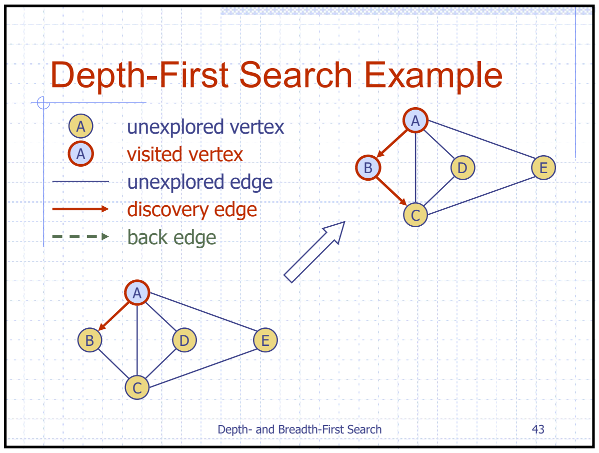

Forest

41

Depth-First Search



42

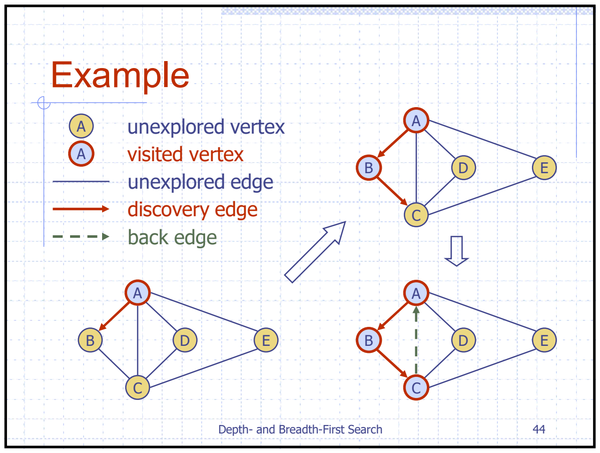


Depth-First Search Example

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

Depth- and Breadth-First Search 43

43

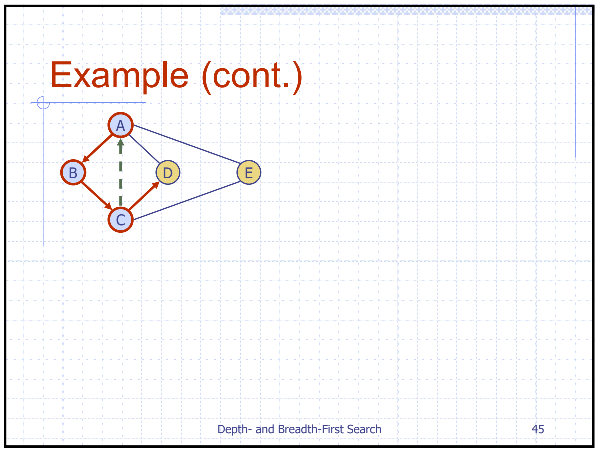


Example

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

Depth- and Breadth-First Search 44

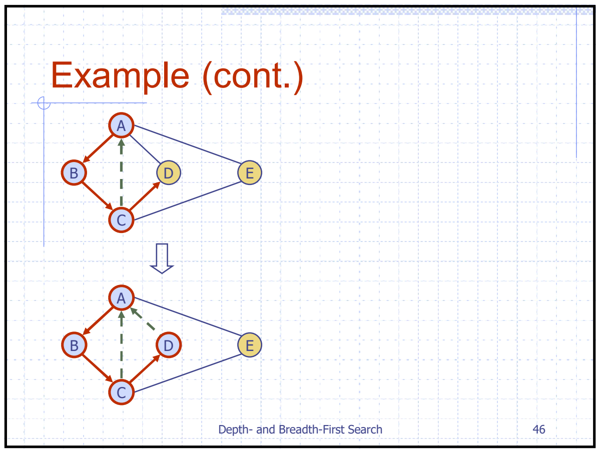
44



Example (cont.)

Depth- and Breadth-First Search 45

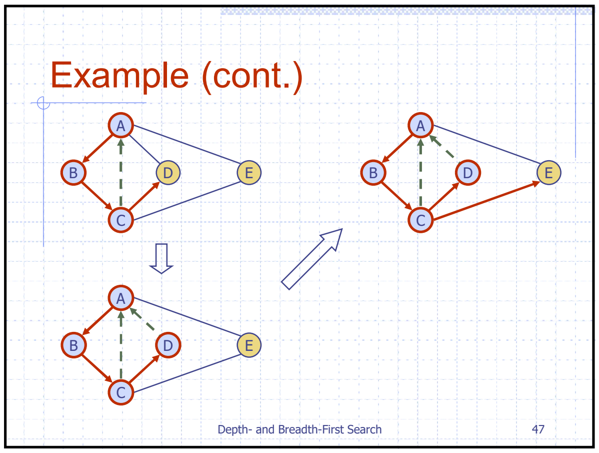
45



Example (cont.)

Depth- and Breadth-First Search 46

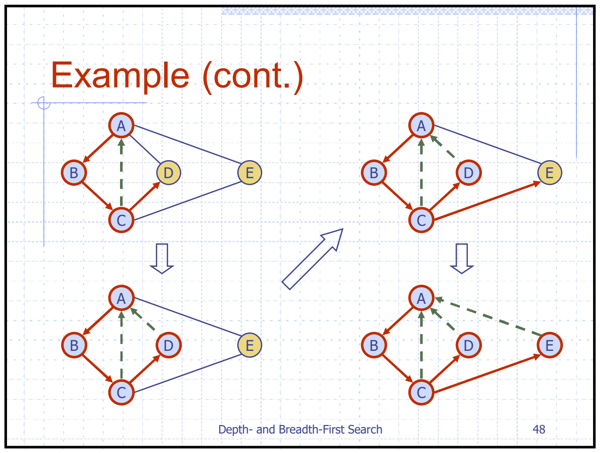
46



Example (cont.)

Depth- and Breadth-First Search 47

47

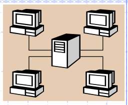


Example (cont.)

Depth- and Breadth-First Search 48

48

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 ∈ G.vertices()  do
    setLabel(u, UNEXPLORED)
  for all  e ∈ G.edges() do
    setLabel(e, UNEXPLORED)
  for all  v ∈ 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 ∈ G.incidentEdges(v) do
    if  getLabel(e) = UNEXPLORED
      w ← G.opposite(v,e)
      if  getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

49

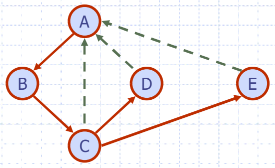
Properties of DFS

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



50

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

51

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

52

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

53

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 ← S.elements()
  for all  e ∈ G.incidentEdges(v) do
    if  getLabel(e) = UNEXPLORED
      w ← 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 }
```

54

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 ¬= null then return
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                cycleDFS(G, w)
                S.pop()
            else
                cycle ← new empty sequence
                o ← w
                repeat
                    cycle.insertLast(o)
                    o ← S.pop()
                until o = w
                setLabel(e, BACK)
    S.pop()
```

55

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

56

Main Point

3. During dept-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.
   Science of Consciousness: 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.

57

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. The edges of a graph connect vertices. Thus connectivity and connected components are important concepts in graph theory.
2. Paths, cycles, spanning trees, and components are important ways that connected vertices can be viewed. Different graph traversal algorithms will systematically compute these ways that vertices can be connected as the basis of specific applications.

58

3. **Transcendental Consciousness** is the underlying basis and connects everything in creation.
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 of the universe.

59