Algorithms and Data Structures

Graph Traversal

Dr. Bernhard Anrig

HS 2012/13

Outline

Definitions

DFS Depth First Search

Applications of DFS

BFS Breath First Search

Applications of BFS

DFS vs. BFS

Decorator Pattern for Graphs

Outline

Definitions

DFS Depth First Search

Applications of DFS

BFS Breath First Search

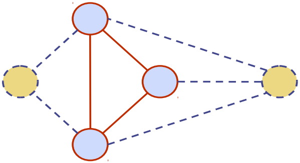
Applications of BFS

DFS vs. BFS

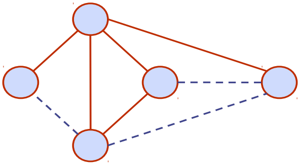
Decorator Pattern for Graphs

Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G



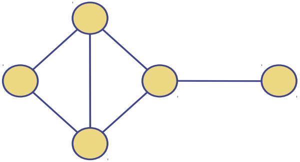
Subgraph



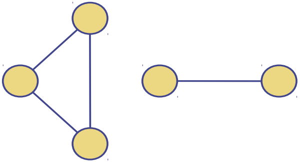
Spanning subgraph

Connectivity

- 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

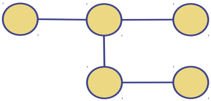


Non connected graph with two connected components

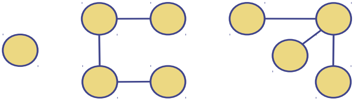
Trees and Forests

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

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



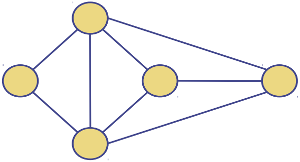
Tree



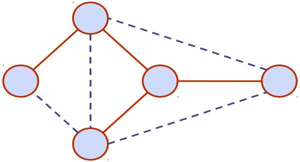
Forest

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

Outline

Definitions

DFS Depth First Search

Applications of DFS

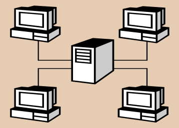
BFS Breath First Search

Applications of BFS

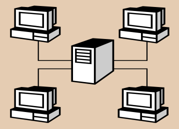
DFS vs. BFS

Decorator Pattern for Graphs

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
- 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 Euler tour is to binary trees



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 $u \in G.vertices()$
  $setLabel(u, UNEXPLORED)$
for all $e \in G.edges()$
  $setLabel(e, UNEXPLORED)$
for all $v \in G.vertices()$
  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 $e \in G.incidentEdges(v)$
  if $getLabel(e) = UNEXPLORED$
    $w \leftarrow G.opposite(v,e)$
    if $getLabel(w) = UNEXPLORED$
      $setLabel(e, DISCOVERY)$
      $DFS(G, w)$
    else
      $setLabel(e, BACK)$

Example

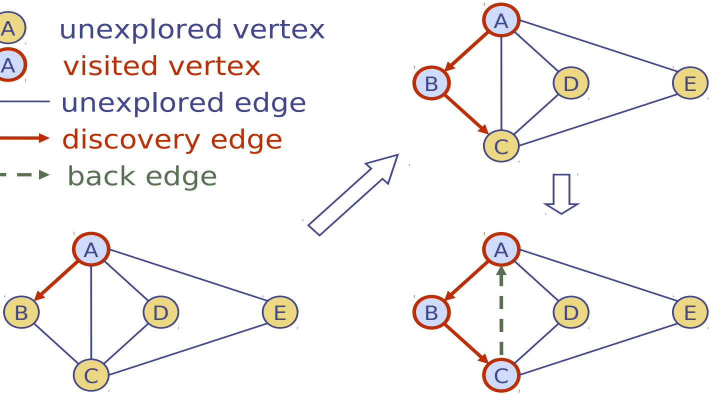
A unexplored vertex

A visited vertex

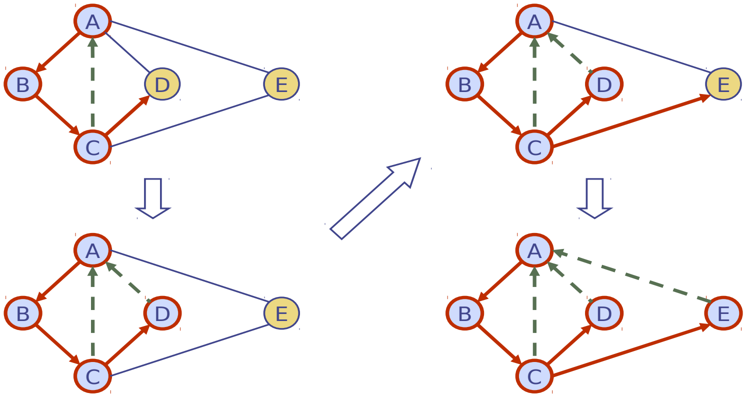
unexplored edge

discovery edge

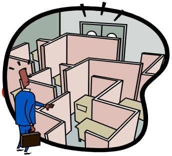
back edge



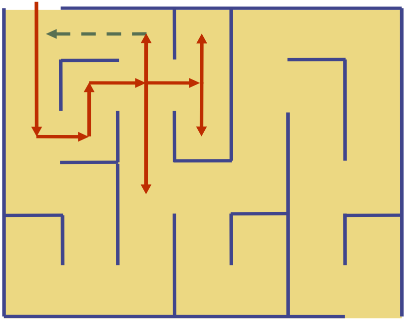
Example (cont.)



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)



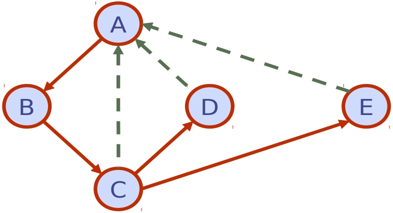
Properties of DFS

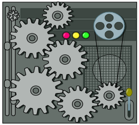
Property 1

$DFS(G, v)$ 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





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

Outline

Definitions

DFS Depth First Search

Applications of DFS

BFS Breath First Search

Applications of BFS

DFS vs. BFS

Decorator Pattern for Graphs

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
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    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 }
```

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, z)
  setLabel(v, VISITED)
  S.push(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v,e)
        S.push(e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            pathDFS(G, w, z)
            S.pop()
        else
            C ← new empty stack
            repeat
                o ← S.pop()
                C.push(o)
            until o = w
            return C.elements()
  S.pop()
```

Outline

Definitions

DFS Depth First Search

Applications of DFS

BFS Breath First Search

Applications of BFS

DFS vs. BFS

Decorator Pattern for Graphs

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
- 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

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

```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.insertLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.insertLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

Example

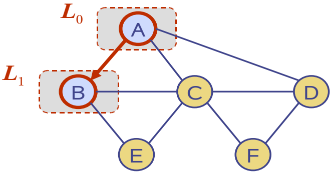
unexplored vertex

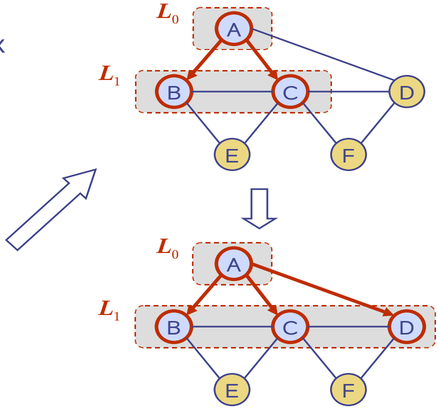
visited vertex

unexplored edge

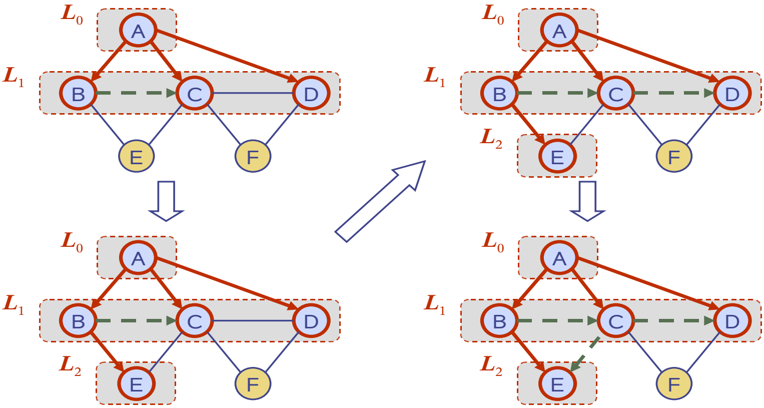
discovery edge

cross edge

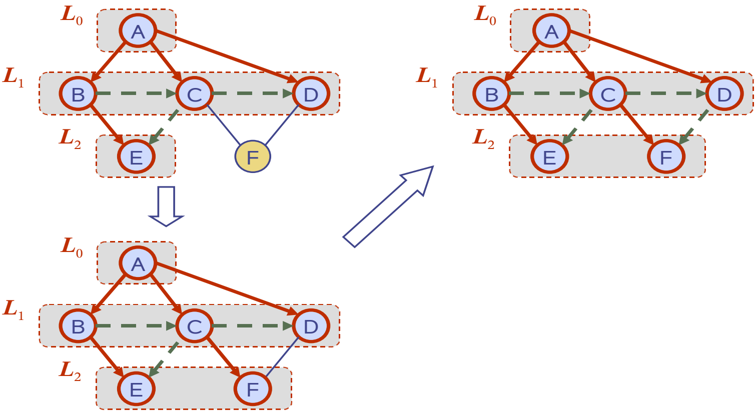




Example (cont.)



Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

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

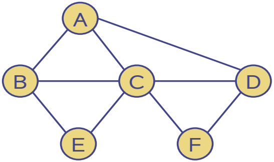
Property 2

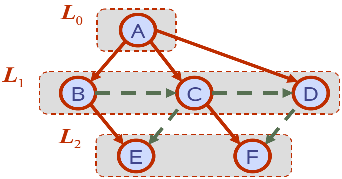
The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

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





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

Outline

Definitions

DFS Depth First Search

Applications of DFS

BFS Breath First Search

Applications of BFS

DFS vs. BFS

Decorator Pattern for Graphs

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

Outline

Definitions

DFS Depth First Search

Applications of DFS

BFS Breath First Search

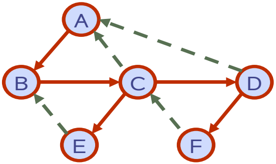
Applications of BFS

DFS vs. BFS

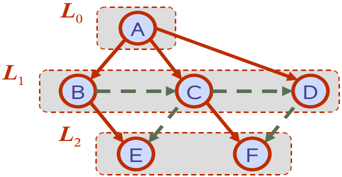
Decorator Pattern for Graphs

DFS vs. BFS

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



DFS



BFS

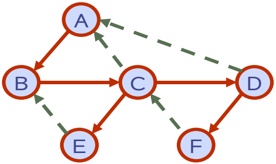
DFS vs. BFS (cont.)

Back edge (v,w)

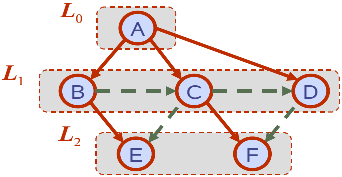
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)

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



DFS



BFS

Outline

Definitions

DFS Depth First Search

Applications of DFS

BFS Breath First Search

Applications of BFS

DFS vs. BFS

Decorator Pattern for Graphs

Decorations

What are decorations of vertices and edges??

- We want to "Mark" visited vertices.
- We want to save the number of unvisited in-incident edges.
- We want to mark in any way vertices or edges.

see also Chapter 12, page 602 (3rd edition)
Chapter 13, page 597 (4th edition)

Decorations

How to do this ?

- Add "attributes" or decorations to existing objects
- Each attribute is identified by a specific key (its name for instance).
- We allow this attribute to take different values for different objects
- One object may have more than one attribute

Note: the way explained below does not exactly match the notion of the "Decorator Pattern" you (will) see in Software Engineering!

DecorablePosition ADT

- `element()` Returns the element stored at this position
- `put(k,x)` Sets to `x` the value of attribute `k`.
  Returns the old value or null if this is a new attribute
- `get(k)` Returns the value of attribute `k` or null is this attribute has no value.
- `remove(k)` Remove the attribute `k`
  Returns the old value of `k` or null if there is no value
- `keySet()` Returns all keys for this position
- `values()` Returns all values for this position

A DecorablePosition interface

```
public interface DecorablePosition<E>
    extends Position<E>, Map<Object,Object> {
}
```

No new methods added!

Map allows for any Object to be used as keys and values, but not known in advance.

A GraphPosition abstract class

```
public abstract class GraphPosition<E>
   implements DecorablePosition<E>{

    private Map<Object,Object> map;
    private E element;

    public GraphPosition(E e){
        map = new HashMap<Object,Object>();
        element = e;
    }
    public E element(){
        return element;
    }
```

```
    public Object get(Object attribute){
        return map.get(attribute);
    }
    public Object put(Object attribute, Object value){
        return map.put(attribute,value);
    }
    public Object remove(Object attribute){
        return map.remove(attribute);
    }
    public Collection<Object> values(){
        return map.values();
    }
    public Set<Objects> keySet(){
        return map.keys();
    }
    ...
}
```