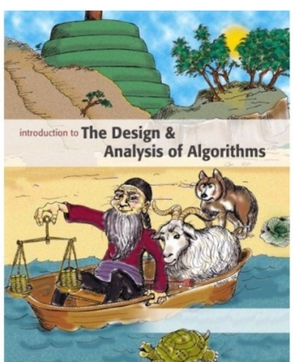
Introduction to

Algorithm Design and Analysis

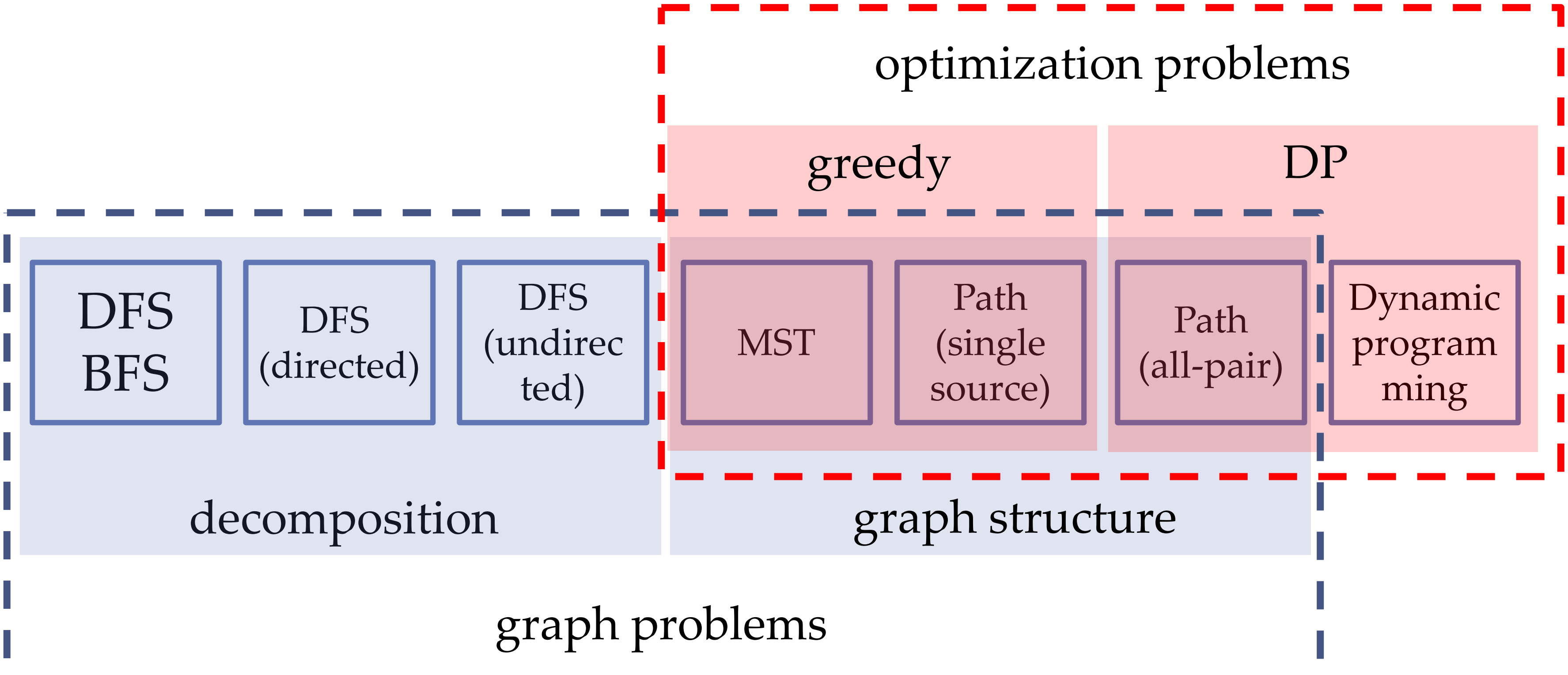
[11] Graph Traversal



Yu Huang

http://cs.nju.edu.cn/yuhuang
Institute of Computer Software
Nanjing University

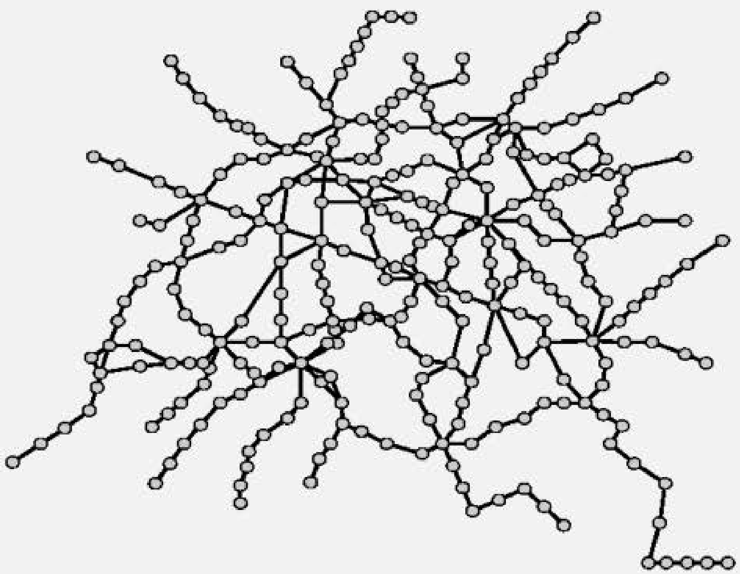
Course Contents

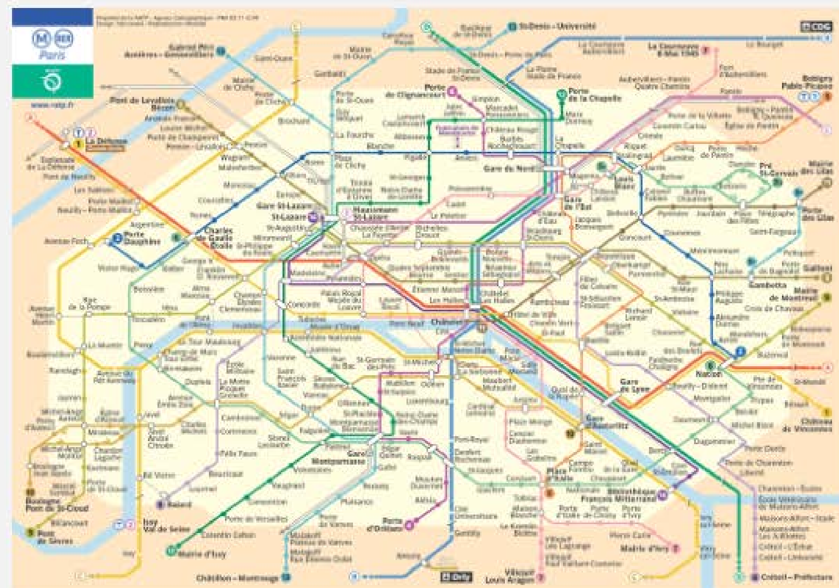


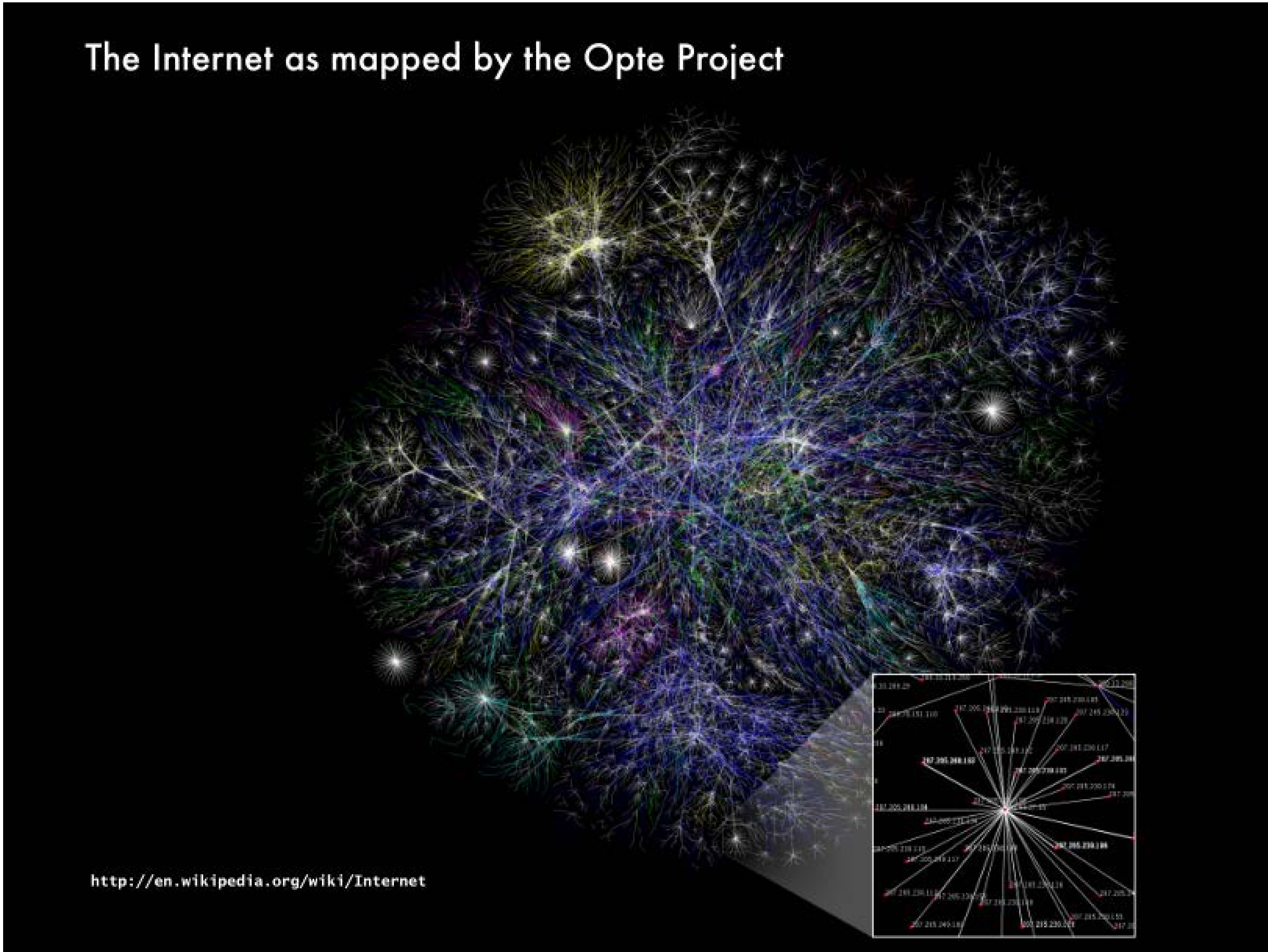
In the Last Class…

- **Dynamic Equivalence Relation**
- **Implementing *disjoint set* by Union-Find**
  - o Straight Union-Find
  - o Making Shorter Tree by Weighted Union
  - o Compressing Path by Compressing Find
    - Amortized analysis of *wUnion-cFind*

Graph Everywhere



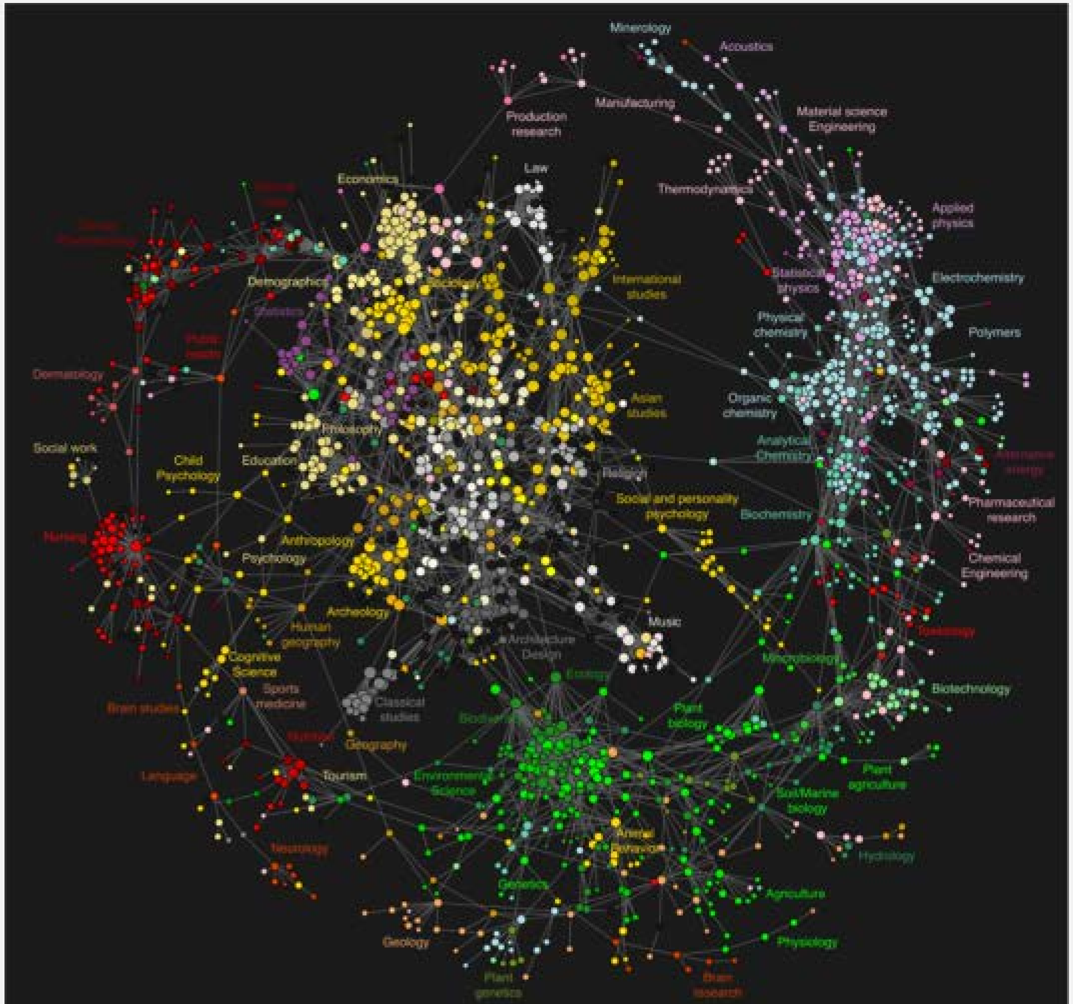



The Internet as mapped by the Opte Project
http://en.wikipedia.org/wiki/Internet

10 million Facebook friends

"Visualizing Friendships" by Paul Butler

Map of science clickstreams



http://www.plosone.org/article/info:doi/10.1371/journal.pone.0004803

Graph Basics

- **Node**
  - Entities of interest
  - $V(G)$
- **Edge**
  - Relations of interest
  - $E(G) \subseteq V \times V$

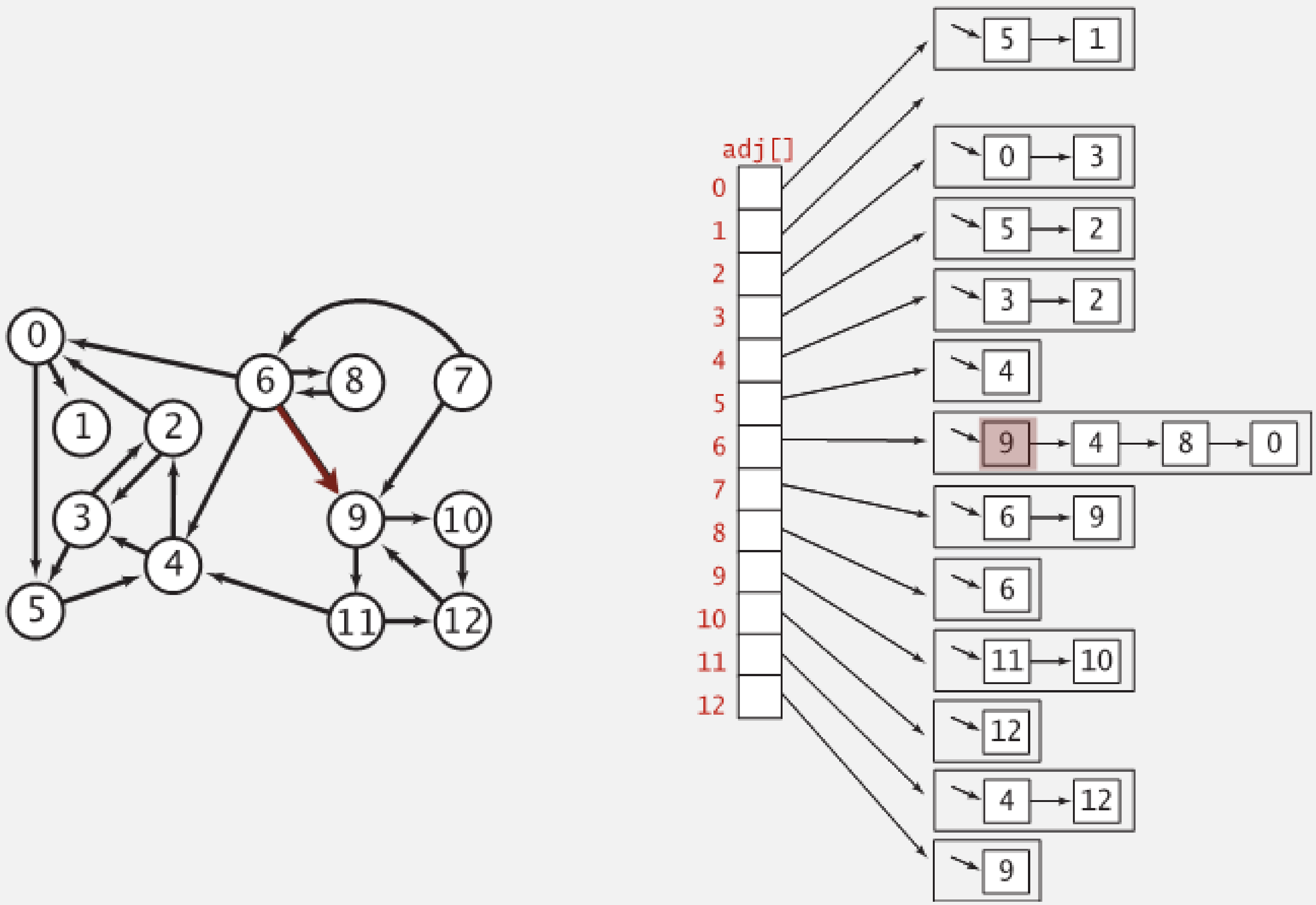
Graph Traversals

- Depth-First and Breadth-First Search
- Finding Connected Components

- General DFS/BFS Skeleton
- Depth-First Search Trace

Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.

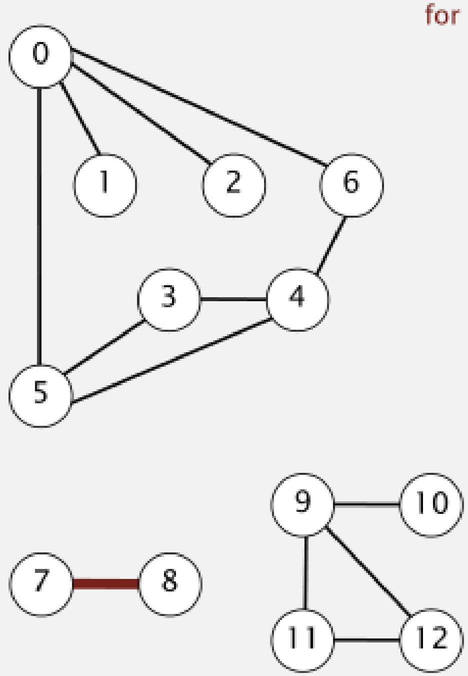


Adjacency-matrix graph representation

Maintain a two-dimensional V-by-V boolean array;

for each edge $v–w$ in graph: `adj[v][w] = adj[w][v] = true.`

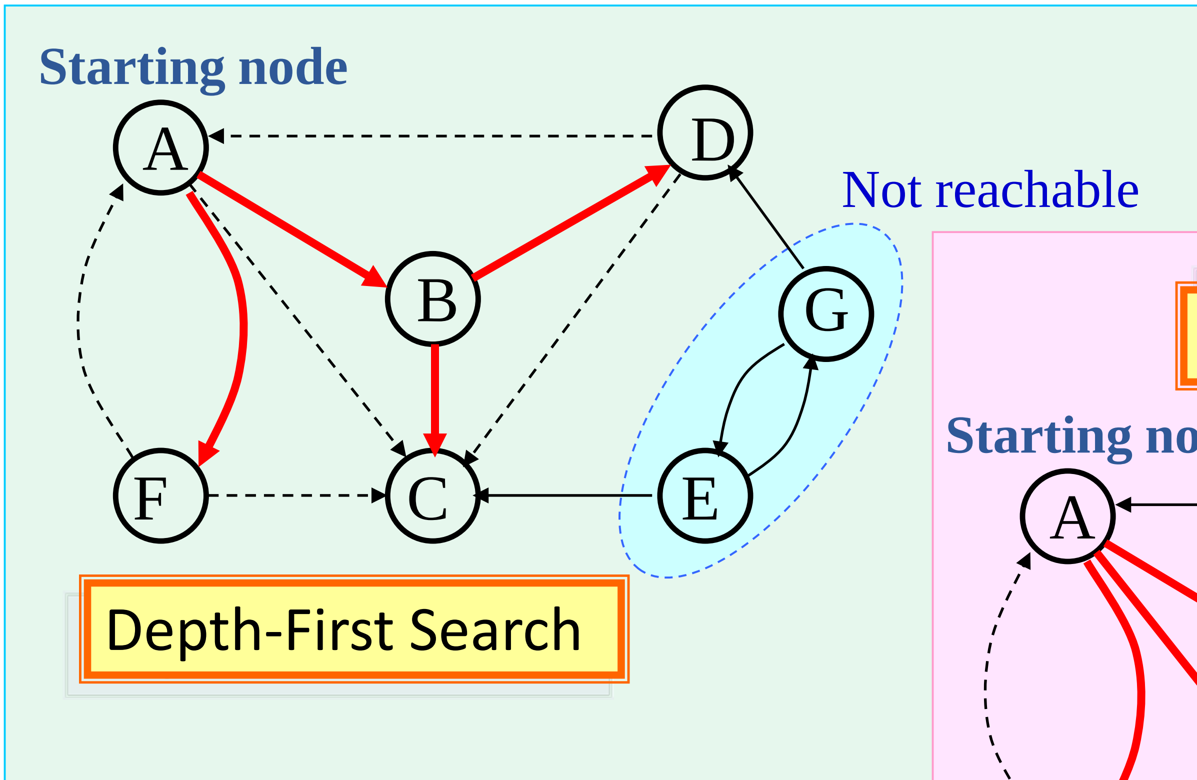
Directed vs. **Undirected** graphs

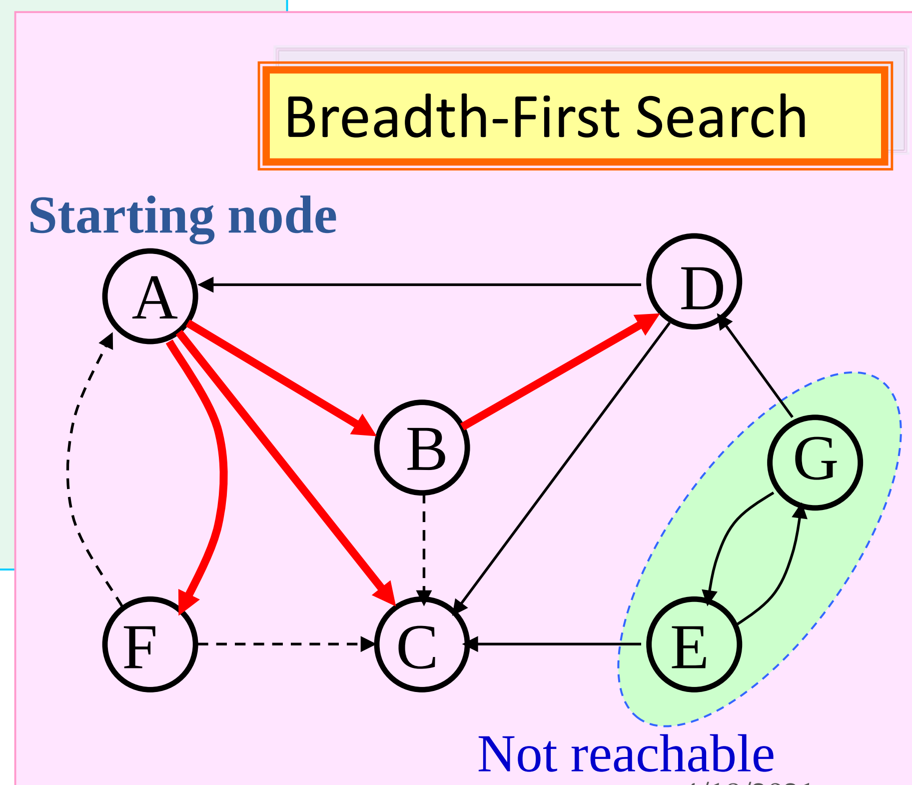


two entries for each edge

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

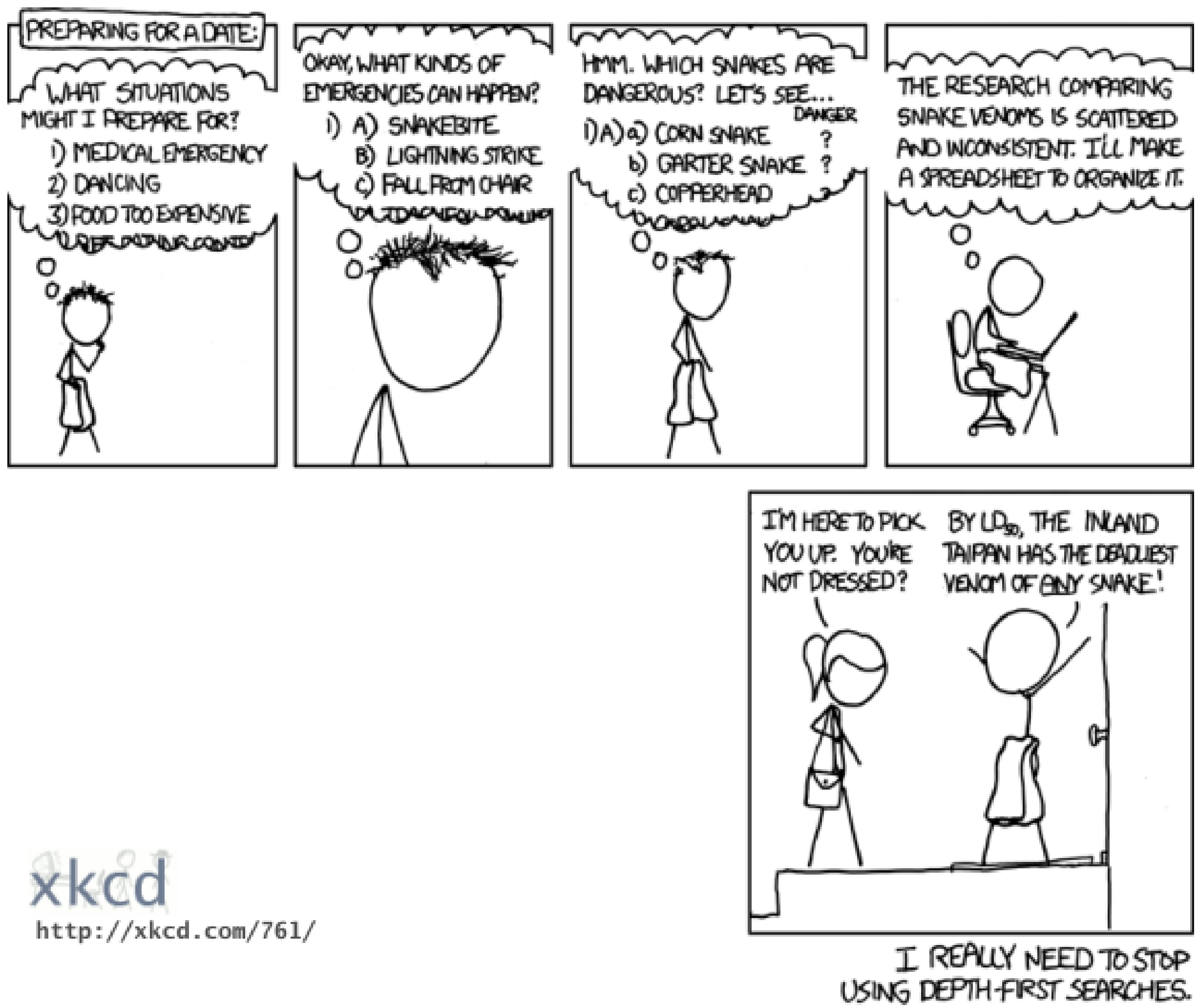
Graph Traversal





- - - - - - - - - *Edges only "checked"*

Depth-first search application: preparing for a date



Outline of Depth-First Search

- dfs(G,v)
- Mark v as "discovered".
- For each vertex w that edge vw is in G:
- If w is undiscovered:
- dfs(G,w)
- Otherwise:
- "Check" vw without visiting w.
- Mark v as "finished".

A vertex must be exact one of three different status:

- undiscovered
- discovered but not finished
- finished

That is: exploring vw, visiting w, exploring from there as much as possible, and backtrack from w to v.

Outline of Breadth-First Search

- Bfs(G,*s*)
- Mark *s* as "discovered";
- **enqueue**(pending,*s*);

- while (pending is nonempty)
- **dequeue**(pending, *v*);
- For each vertex *w* that edge *vw* is in G:
- If *w* is "undiscovered"
- Mark *w* as "discovered" and **enqueue**(pending, *w*)
- Mark *v* as "finished";

Finding Connected Components

- **Input: a symmetric digraph G, with n nodes and $2m$ edges(interpreted as an undirected graph), implemented as a array *adjVertices*[1,…n] of adjacency lists.**
- **Output: an array *cc*[1..n] of component number for each node v_i**

- void connectedComponents(Intlist[ ] *adjVertices*, int *n*,
  int[ ] *cc*) // *This is a wrapper procedure*
-     int[ ] color=new int[*n*+1];
-     int *v*;
-     <Initialize color array to white for all vertices>
-     for (*v*=1; *v*≤*n*; *v*++)
-         if (color[*v*]==white)
-             ccDFS(*adjVertices*, *color*, *v*, *v*, *cc*);
-     return

Depth-first search

ccDFS: the procedure

- void ccDFS(IntList[ ] *adjVertices*, int[ ] *color*, int *v*, int *ccNum*, int [ ] *cc*)//*v* as the code of current connected component
-     int *w*;
-     IntList remAdj;

-     color[v]=gray;
-     cc[v]=ccNum;
-     remAdj=adjVertices[v];
-     while (remAdj≠nil)
-         w=first(remAdj);
-         if (color[w]==white)
-             ccDFS(adjVertices, color, w, ccNum, cc);
-         remAdj=rest(remAdj);
-     color[v]=black;
-     return

The elements of *remAdj* are neighbors of *v*

Processing the next neighbor, if existing, another depth-first search to be incurred

v finished

Analysis of CC Algorithm

- **connectedComponents, the wrapper**
  - Linear in *n* (color array initialization+for loop on *adjVertices*)
- **ccDFS, the depth-first searcher**
  - In one execution of ccDFS on *v*, the number of instructions(rest(*remAdj*)) executed is proportional to the size of *adjVertices*[*v*].
  - Note: Σ(size of *adjVertices*[*v*]) is 2*m*, and the adjacency lists are traveresed **only once**.

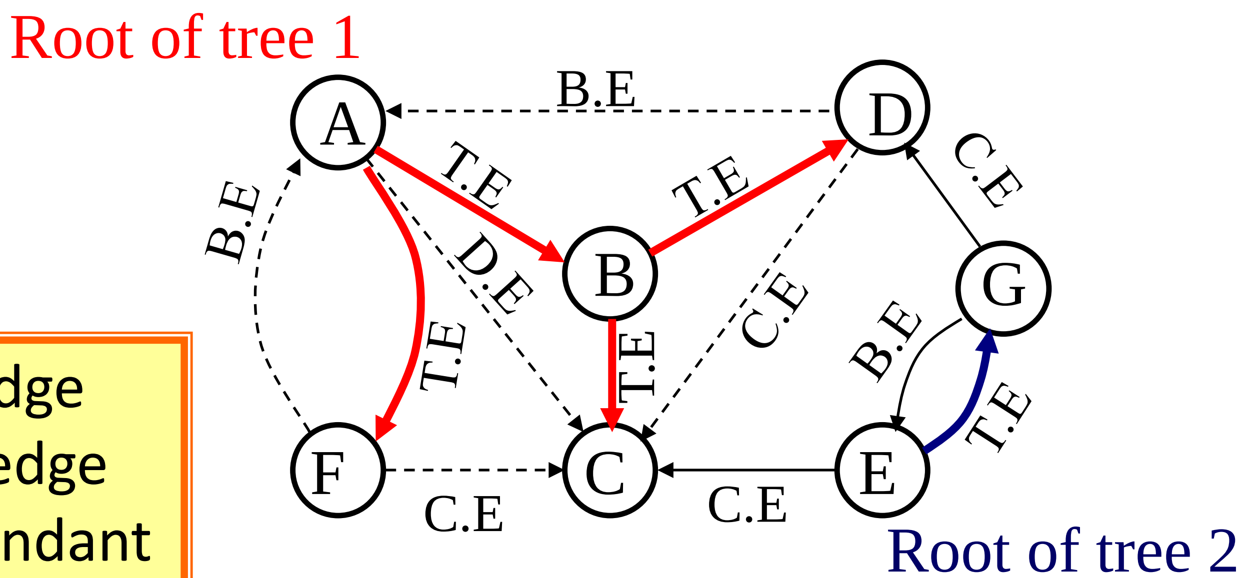
- **So, the time complexity is in Θ(m+n)**
  - Extra space requirements:
    - Color array
    - Activation frame stack for recursion

Visits On a Vertex

- **Classification for the visits on a vertex**
  - First visit(exploring): status: white→gray
  - (Possibly) multi-visits by backtracking to: status keeps gray
  - Last visit(no more branch-finished): status: gray→black
- **Different operations can be done, during the different visits on a specific vertex**
  - On the vertex
  - On (selected) incident edges

Depth-first Search Trees

DFS forest={(DFS tree1), (DFS tree2)}



T.E: tree edge
B.E: back edge
D.E: descendant edge
C.E: cross edge

A finished vertex is never revisited, such as C

Depth-First Search – Generalized Skeleton

- Input: Array *adjVertices* for graph G
- Output: Return value depends on application.
- int dfsSweep(IntList[] *adjVertices*,int n, …)
-     int ans;
-     <Allocate color array and initialize to white>
-     For each vertex *v* of G, in some order
-         if (color[v]==white)
-             int vAns=dfs(*adjVertices*, color, v, …);
-             <Process vAns>
-         // Continue loop
-     return ans;

Depth-First Search – Generalized Skeleton

- int dfs(IntList[] *adjVertices*, int[] color, int v, …)
- int w;
- IntList remAdj;
- int ans;
- color[v]=gray;
- <Preorder processing of vertex v>
- remAdj=*adjVertices*[v];
- while (remAdj≠nil)
- w=first(remAdj);
- if (color[w]==white)
- <Exploratory processing for tree edge vw>
- int wAns=dfs(*adjVertices*, color, w, …);
- < Backtrack processing for tree edge vw , using wAns>
- else
- <Checking for nontree edge vw>
- remAdj=rest(remAdj);
- <Postorder processing of vertex v, including final computation of ans>
- color[v]=black;
- return ans;

If partial search is used for a application, tests for termination may be inserted here.

Specialized for connected components:
- parameter added
- preorder processing inserted – cc[v]=ccNum

Breadth-First Search - Skeleton

- Input: Array *adjVertices* for graph G
- Output: Return value depends on application.
- void bfsSweep(IntList[] *adjVertices*,int n, …)
- int ans;
- <Allocate color array and initialize to white>
- For each vertex *v* of G, in some order
- if (color[v]==white)
- void bfs(*adjVertices*, color, v, …);
- // Continue loop
- return;

Breadth-First Search - Skeleton

- void bfs(IntList[] *adjVertices*, int[] color, int v, …)
- int w; IntList remAdj; Queue pending;
- color[v]=gray; enqueue(pending, *v*);
- while (pending is nonempty)
- *w*=dequeue(pending); remAdj=*adjVertices*[*w*];
- while (remAdj≠nil)
- *x*=first(remAdj);
- if (color[*x*]==white)
- color[*x*]=gray; enqueue(pending, *x*);
- remAdj=rest(remAdj);
- <processing of vertex w>
- color[*w*]=black;
- return ;

can be further generalized

DFS vs. BFS Search

- **Processing opportunities for a node**
  - Depth-first: 2
    - At discovering
    - At finishing
  - Breadth-first: only 1, when de-queued
  - At the second processing opportunity for the DFS, the algorithm can make use of information about the descendants of the current node.

Time Relation on Changing Color

- **Keeping the order in which vertices are encountered for the first or last time**
    - A global integer time: 0 as the initial value, incremented with each color changing for *any* vertex, and the final value is $2n$
    - Array *discoverTime*: the i th element records the time vertex v_i turns into gray
    - Array *finishTime*: the i th element records the time vertex v_i turns into black
    - The active interval for vertex v, denoted as $active(v)$, is the duration while v is gray, that is:

$$discoverTime[v], \ldots, finishTime[v]$$

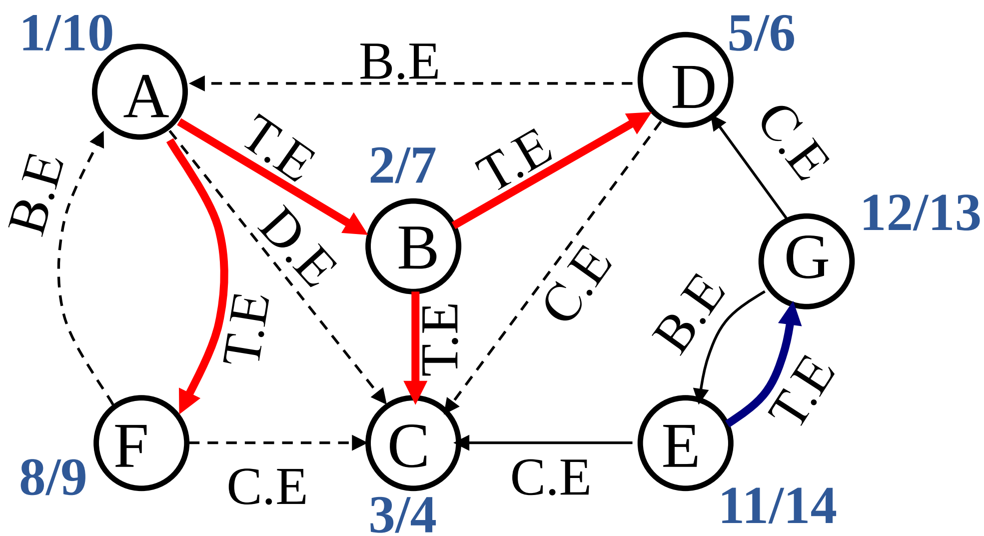
Depth-First Search Trace

- **General DFS skeleton modified to compute discovery and finishing times and "construct" the depth-first search forest.**
- **int dfsTraceSweep(IntList[ ] *adjVertices*,int n, int[ ] *discoverTime*, int[ ] *finishTime*, int[ ] *parent*)**
- **int ans; int *time*=0**
- **<Allocate color array and initialize to white>**
- **For each vertex *v* of G, in some order**
- **if (color[v]==white)**
- **parent[v]=-1**
- **int vAns=dfsTrace(*adnVertices*, color, v, *discoverTime*, *finishTime*, *parent*, *time*);**
- **// Continue loop**
- **return ans;**

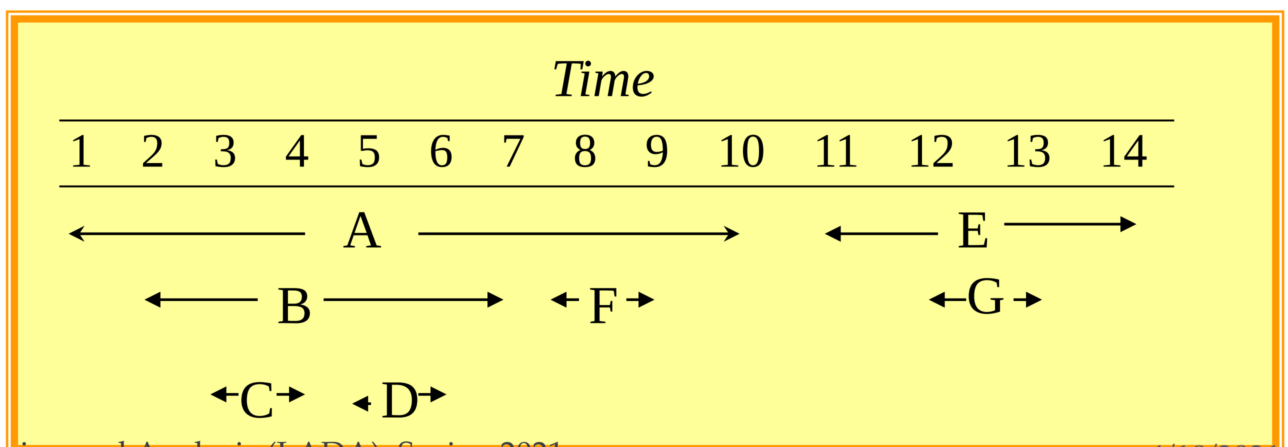
Depth-First Search Trace

- int dfsTrace(intList[ ] *adjVertices*, int[ ] color, int v, int[ ] *discoverTime*,
- int[ ] *finishTime*, int[ ] *parent* int *time*)
- int w; IntList remAdj; int ans;
- color[v]=gray; time++; *discoverTime*[v]=time;
- remAdj=*adjVertices*[v];
- while (remAdj≠nil)
- w=first(remAdj);
- if (color[w]==white)
- parent[w]=v;
- int wAns=dfsTrace(*adjVertices*, color, w, *discoverTime*, *finishTime*, *parent*, *time*);
- else <Checking for nontree edge vw>
- remAdj=rest(remAdj);
- time++; *finishTime*[v]=time; color[v]=black;
- return ans;

Active Interval



The relations are summarized in the next frame



Properties of Active Intervals(1)

- **If w is a descendant of v in the DFS forest, then $active(w) \subseteq active(v)$, and the inclusion is proper if $w \neq v$.**
- **Proof:**
    - Define a partial order <: w<v iff. w is a proper descendants of v in its DFS tree. The proof is by induction on <.
    - If v is minimal. The only descendant of v is itself. Trivial.
    - Assume that for all x<v, if w is a descendant of x, then active(w)⊆active(x).
    - Let w be any proper descendant of v in the DFS tree, there must be some x such that vx is a tree edge on the tree path to w, so w is a descendant of x. According to dfsTrace, we have $active(x) \subset active(v)$, by inductive hypothesis, $active(w) \subset active(v)$.

Properties of Active Intervals(2)

- **If $active(w) \subseteq active(v)$, then w is a descendant of v. And if $active(w) \subset active(v)$, then w is a proper descendant of v.**

  That is: *w* is discovered while *v* is active.

- **Proof:**
  - If w is **not** a descendant of v, there are two cases:
    - v is a proper descendant of w, then $active(v) \subset active(w)$, so, it is impossible that $active(w) \subseteq active(v)$, contradiction.
    - There is no ancestor/descendant relationship between v and w, then $active(w)$ and $active(v)$ are disjoint, contradiction.

Properties of Active Intervals(3)

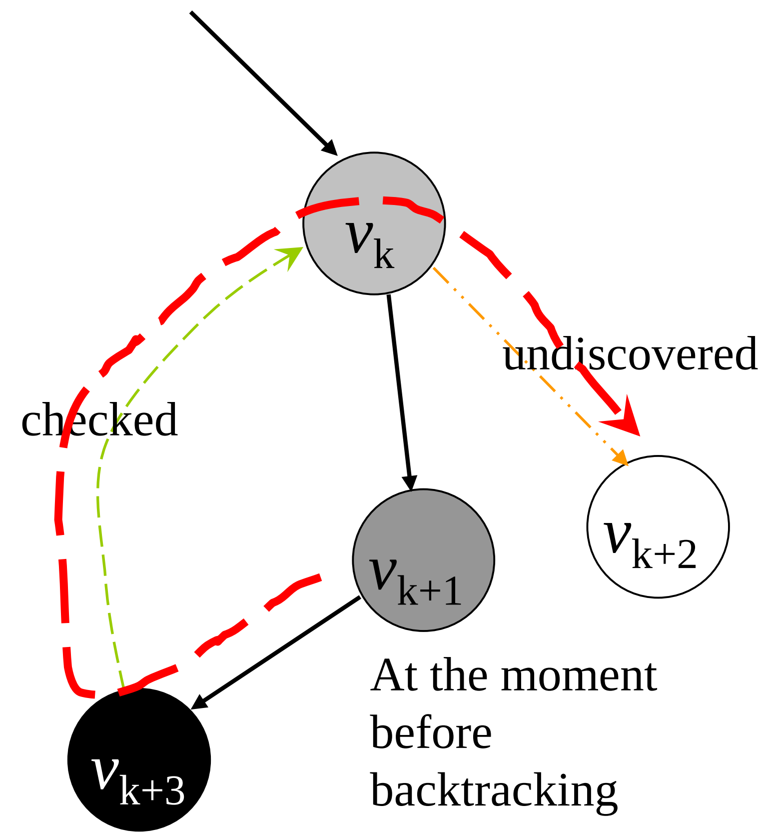
- **If v and w have no ancestor/descendant relationship in the DFS forest, then their active intervals are disjoint.**
- **Proof:**
  - o If v and w are in different DFS tree, it is trivially true, since the trees are processed one by one.
  - o Otherwise, there must be a vertex c, satisfying that there are tree paths c to v, and c to w, without edges in common. Let the leading edges of the two tree path are cy, cz, respectively. According to dfsTrace, $active(y)$ and $active(z)$ are disjoint.
  - o We have $active(v) \subseteq active(y)$, $active(w) \subseteq active(z)$. So, $active(v)$ and *active*(w) are disjoint.

Properties of Active Intervals(4)

- **If edge vw$\in E_G$, then**
  - vw is a **cross edge** iff. *active*(w) entirely precedes *active*(v).
  - vw is a **descendant edge** iff. there is some third vertex x, such that *active*(w)$\subset$*active*(x)$\subset$*active*(v),
  - vw is a **tree edge** iff. *active*(w)$\subset$*active*(v), and there is no third vertex x, such that *active*(w)$\subset$*active*(x) $\subset$*active*(v),
  - vw is a **back edge** iff. *active*(v)$\subset$*active*(w),

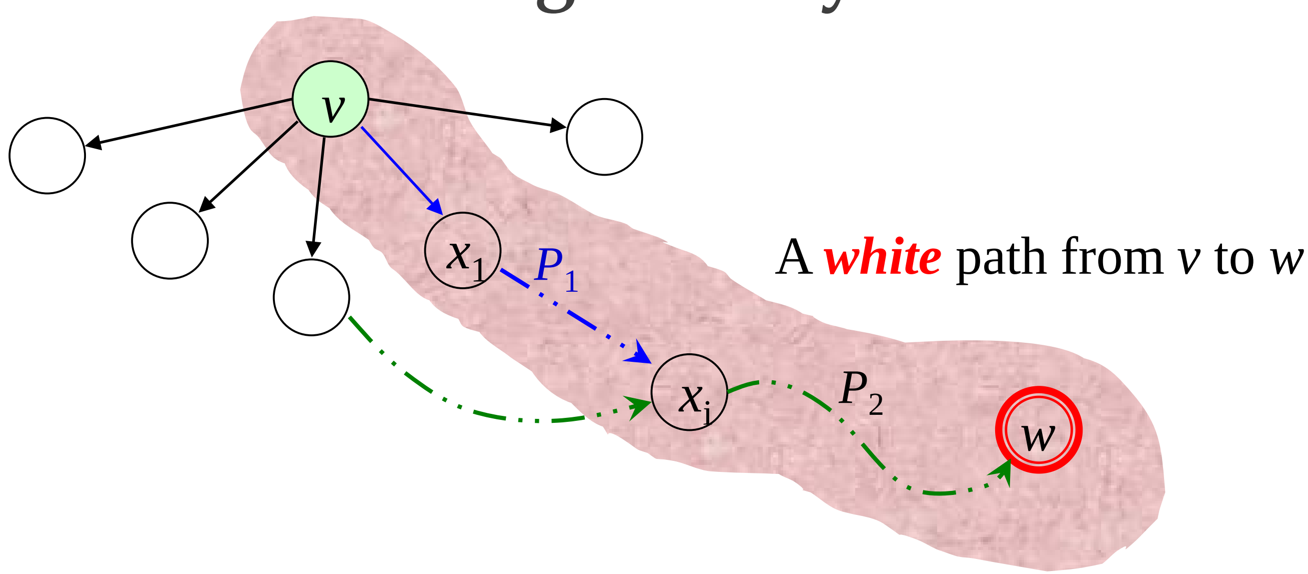
Ancestor and Descendant

- That w is a descendant of v in the DFS forest means that there is a direct path from v to w in some DFS tree.
- The path is also a path in G.
- **However, if there is a direct path from v to w in G, is w necessarily a descendant of v in *the* DFS forest?**



DFS Tree Path

- [**White Path Theorem**] **w is a descendant of v in a DFS tree iff. at the time v is discovered(just to be changing color into gray), there is a path in G from v to w consisting entirely of white vertices.**



Proof of White Path Theorem

- **Proof**
  - o $\Rightarrow$ All the vertices in the path are descendants of v.
  - o $\Leftarrow$ by induction on the length k of a white path from v to w.
    - When k=0, v=w.
    - For k>0, let P=(v, x_1,x_2,…x_k=w). There must be some vertex on P which is discovered during the active interval of v, e.g. x_1, Let x_i is earliest discovered among them. Divide P into P_1 from v to x_i, and P_2 from x_i to w. P_2 is a white path with length less than k, so, by inductive hypothesis, w is a descendant of x_i. Note: $active(x_i) \subseteq active(v)$, so x_i is a descendant of v. By transitivity, w is a descendant of v.

Thank you!

Q & A

Yu Huang
http://cs.nju.edu.cn/yuhuang