CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC11: DEPTH-FIRST SEARCH (169-184)

Lecture slide courtesy of Prof. Steven Skiena

PROBLEM OF THE DAY

- Prove that in a breadth-first search on a undirected graph G, every edge in G is either a tree edge or a cross edge, where a cross edge (x; y) is an edge where x is neither is an ancestor or descendent of y.

BFS VS DFS

- The difference between BFS and DFS results is in the order in which they explore vertices.
- This order depends completely upon the container data structure used to store the *discovered* but not *processed* vertices (The Todo-list),

TODO LIST STRUCTURES

- *Queue* allows the BFS to explore the oldest unexplored vertices first.
  - Thus our explorations radiate out slowly from the starting vertex.
- *Stack* allows the DFS to explore the vertices by lurching along a path, visiting a new neighbor if one is available, and backing up only when we are surrounded by previously discovered vertices.
  - Thus, our explorations quickly wander away from our starting point.

DEPTH-FIRST SEARCH

- DFS has a neat <u>recursive implementation</u> which eliminates the need to explicitly use a stack.
- By maintaining a notion of traversal *time* for each vertex.
  - Our time clock ticks each time we enter or exit any vertex.
  - Keep track of the *entry* and *exit* times for each vertex

DFS PSEUDO CODE

```
DFS(G, u)
    state[u] = "discovered"
    process vertex u if desired
    entry[u] = time                    /* time is global variable */
    time = time + 1
    for each v ∈ Adj[u] do
        process edge (u, v) if desired
        if state[v] = "undiscovered" then
            p[v] = u
            DFS(G, v)
    state[u] = "processed"
    exit[u] = time
    time = time + 1
```

IMPLEMENTATION

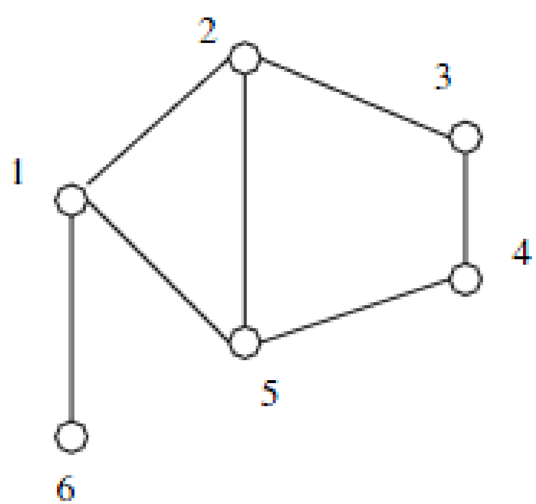
- The beauty of implementing dfs recursively is that recursion eliminates the need to keep an explicit stack:

```
dfs(graph *g, int v)
{
        edgenode *p; /* temporary pointer */
        int y; /* successor vertex */

        if (finished) return; /* allow for search termination */

        discovered[v] = TRUE;
        time = time + 1;
        entry_time[v] = time;

        process_vertex_early(v);
```

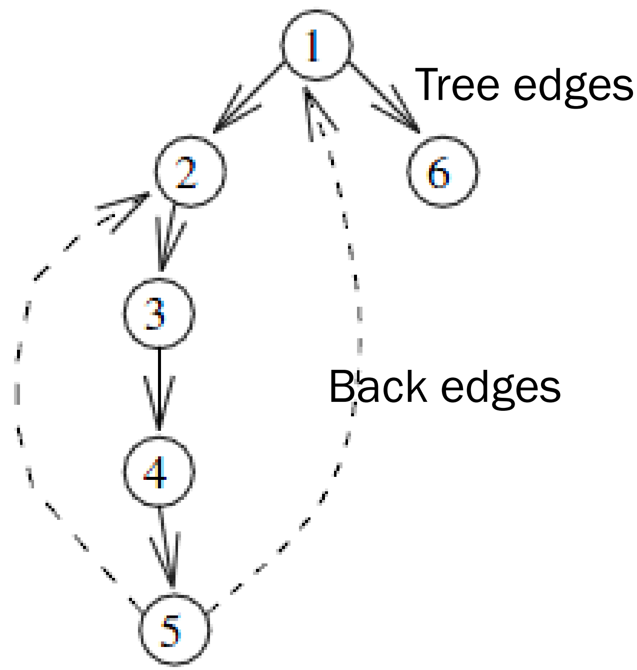
```
        p = g->edges[v];
        while (p != NULL) {
                y = p->y;
                if (discovered[y] == FALSE) {
                        parent[y] = v;
                        process_edge(v,y);
                        dfs(g,y);
                }
                else if ((!processed[y]) || (g->directed))
                        process_edge(v,y);
                if (finished) return;
                p = p->next;
        }
        process_vertex_late(v);
        time = time + 1;
        exit_time[v] = time;
        processed[v] = TRUE;
}
```

TREE EDGES AND *BACK EDGES*.

- In a DFS of an undirected graph, we assign a direction to each edge, from the vertex which discover it:



undirected graph



depth-first search tree

EDGE CLASSIFICATION FOR DFS

- Every edge is either:

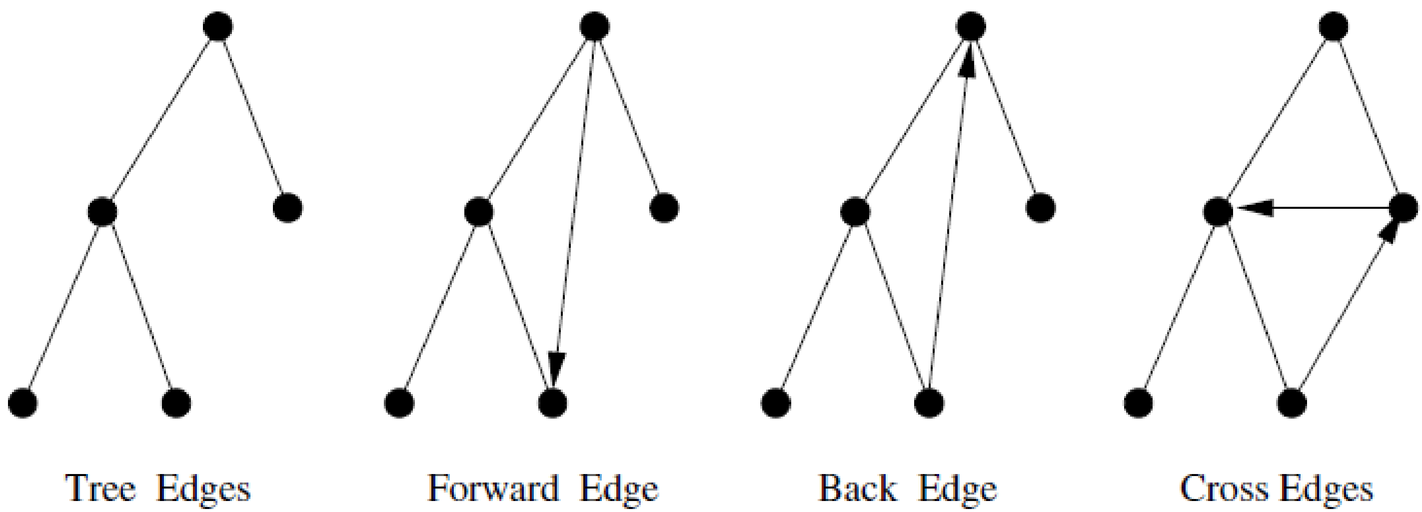


Figure 5.14: Possible edge cases for BFS/DFS traversal

- On any particular DFS or BFS of a directed or undirected graph, each edge gets classified as one of the above.

USEFUL PROPERTIES OF TIME INTERVALS

- *Who is an ancestor?*
  - Suppose that *x* is an ancestor of *y* in the DFS tree.
  - Time interval of *y* must be properly nested within ancestor *x*.
    - we must enter *x* before *y*,
    - we must exit *y* before we exit *x*
- *How many descendants?*
  - Half the time difference between the exit and entry times for *v* tells us how many descendants *v* has in the DFS tree.
    - Clock ticks in entering & exiting (2 times)

SUMMARY OF DFS PROPERTIES

- Entry and Exit times in several applications of BFS,
  - EX> topological sorting & biconnected/strongly-connected components.
- DFS partitions the edges of an undirected graph into exactly two classes: *tree edges* and *back edges*.
  - Tree edges discover new vertices, and are those encoded in the parent relation.
  - Back edges are those whose other endpoint is an ancestor of the vertex being expanded, so they point back into the tree.

APPLICATIONS OF DEPTH-FIRST SEARCH

- Application of DFS is surprisingly *subtle*, however meaning that its correctness requires getting details right.
- The correctness of a DFS-based algorithm depends upon specifics of exactly *when* we process the edges and vertices.
  - We can process vertex *v* either before we have traversed any of the outgoing edges from *v* (*process_vertex_early*()) or
  - After we have finished processing all of them (*process_vertex_late*()).

PROCESSING UNDIRECTED GRAPHS (173)

- In undirected graphs, each edge (*x*, *y*) sits in the adjacency lists of vertex *x* and *y*.
  - there are two potential times to process each edge (*x*, *y*), namely when exploring *x* and when exploring *y*.
- Edge-specific processing happened the first time and take different action the second time we see an edge.
  - EX> The labeling of edges (as tree edges or back edges) occurs during the first time the edge is explored

EDGE LABELING IN UNDIRECTED GRAPH

- How can we tell if we have previously traversed the edge from *y*?
  - If vertex *y* is undiscovered: (*x*, *y*) becomes a tree edge so this must be the first time visiting the edge (x,y)
  - If *y* has not been completely processed: we explored the edge (y,x) when we explored *y* this must be the second time visiting (x,y).
  - If *y* is an ancestor of *x* thus in a discovered state: this must be our first traversal *unless* *y* is the immediate ancestor of *x*—i.e. , (*y*, *x*) is a tree edge.
    - testing if y == parent[x].

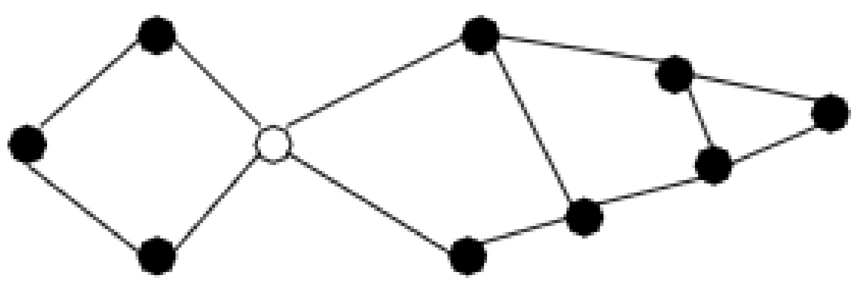
DFS APPLICATION 1: FINDING CYCLES

- Back edges are the key to finding a cycle in an undirected graph.
- Any back edge going from x to an ancestor y creates a cycle with the path in the tree from y to x.

```
process_edge(int x, int y)
{
        if (parent[x] != y) {          /* found back edge! */
                printf("Cycle from %d to %d:",y,x);
                find_path(y,x,parent);
                printf("\n\n");
                finished = TRUE; /* so that we finish after first cycle */
        }
}
```

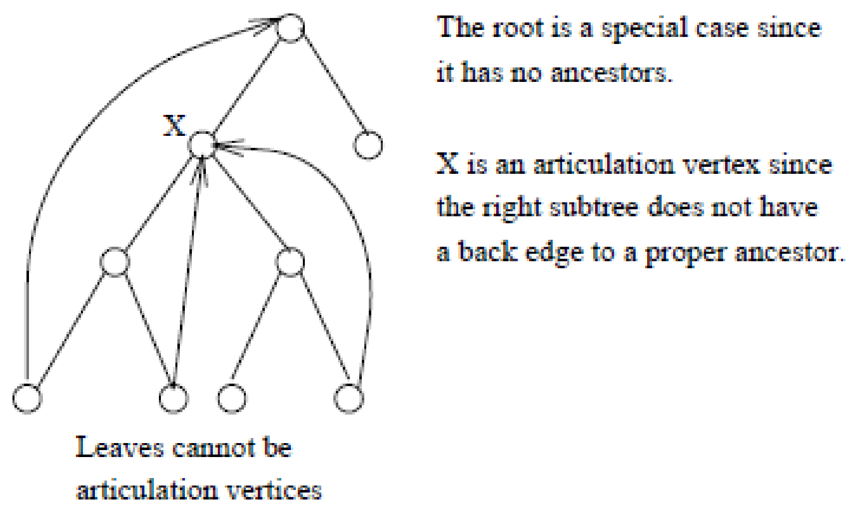
DFS APPLICATION 2: ARTICULATION VERTICES

- *articulation vertex* or *cut-node* : a single vertex whose deletion disconnects a connected component of the graph
- Any graph that contains an articulation vertex is inherently fragile, because deleting that single vertex causes a loss of connectivity between other nodes.



FINDING ARTICULATION VERTICES

- Brute force method O(n(m+n)) :
  - just delete each vertex to do a DFS or BFS on the remaining graph to see if it is connected.
- Better method O(n+m):
  - In a DFS tree, a vertex v (other than the root) is an articulation vertex iff v is not a leaf and some subtree of v has no back edge incident until a proper ancestor of v.



DEPTH-FIRST SEARCH ON DIRECTED GRAPHS

- When traversing undirected graphs, every edge is either in the depth-first search tree or a back edge to an ancestor in the tree.
  - Suppose we encountered a "forward edge" (x, y) directed toward a descendant vertex. In this case, we would have discovered (x, y) while exploring y, making it a back edge.
  - Suppose we encounter a "cross edge" (x, y), linking two unrelated vertices.
- For directed graphs, depth-first search labelings can take all four labels

EDGE CLASSIFICATION IMPLEMENTATION

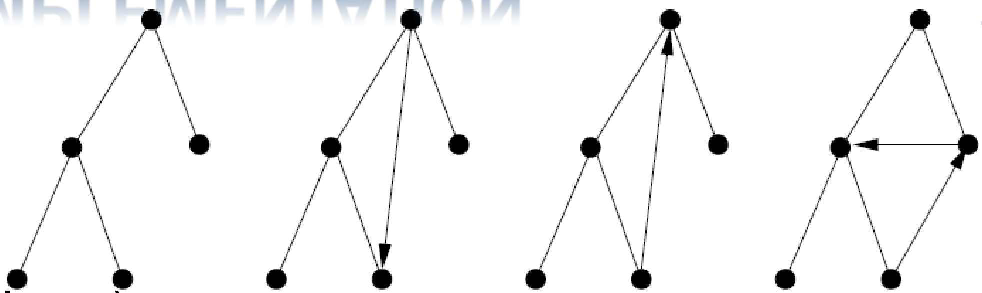
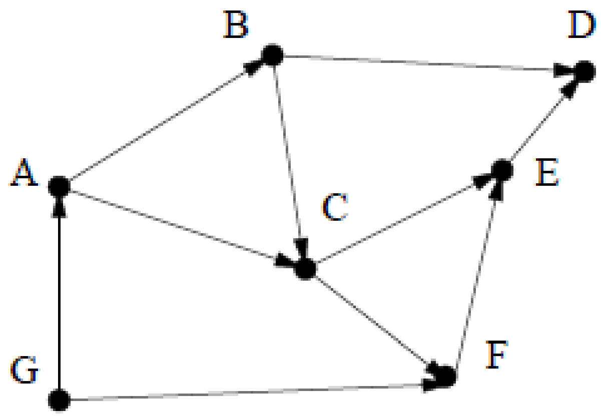


Figure 5.14: Possible edge cases for BFS/DFS traversal

```
int edge_classification(int x, int y)
{
        if (parent[y] == x)
                return(TREE);
        if (discovered[y] && !processed[y])
                return(BACK);
        if (processed[y] && (entry time[y]<entry time[x]))
                return(FORWARD);
        if (processed[y] && (entry time[y]>entry time[x]))
                return(CROSS);
        printf(”Warning: unclassified edge (%d,%d)”,x,y);
}
```

APPLICATION : TOPOLOGICAL SORTING

- A directed acyclic graph (DAG) has no directed cycles.



- A topological sort of a graph is an ordering on the vertices so that all edges go from left to right. DAGs (and only DAGs) has at least one topological sort (here G; A;B; C; F;E;D).

APPLICATIONS OF TOPOLOGICAL SORTING

- Topological sorting is often useful in scheduling jobs in their proper sequence.
- In general, we can use it to order things given precedence constraints.
- Example: Dressing schedule from CLR.

TOPOLOGICAL SORTING VIA DFS

- A directed graph is a DAG if and only if no back edges are encountered during a depth-first search.
- Labeling each of the vertices in the reverse order that they are marked *processed* finds a topological sort of a DAG.
- Why?
- Consider what happens to each directed edge {x,y} as we encounter it during the exploration of vertex x:

CASE ANALYSIS

- If y is currently *undiscovered*, then we then start a DFS of y before we can continue with x. Thus y is marked *processed* before x is, and x appears before y in the topological order, as it must.

- If y is *discovered* but not *processed*, then {x,y} is a back edge, which is forbidden in a DAG.

- If y is *processed*, then it will have been so labeled before x. Therefore, x appears before y in the topological order, as it must.

TOPOLOGICAL SORTING IMPLEMENTATION

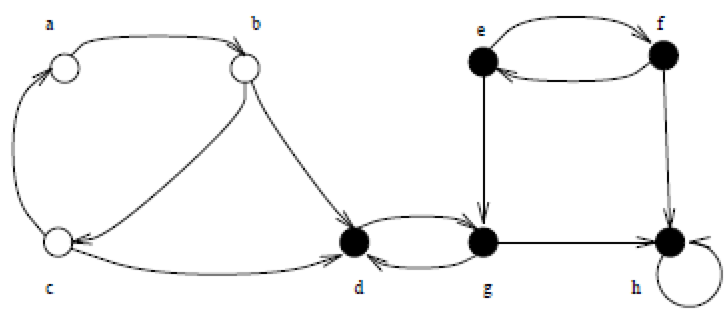
```
process_vertex_late(int v)
{
        push(&sorted,v);
}

process_edge(int x, int y)
{
        int class; /* edge class */
        class = edge_classification(x,y);
        if (class == BACK)
                printf("Warning: directed cycle found, not a DAG\n");
}
```

```
topsort(graph *g)
{
    int i;                        /* counter */
    init_stack(&sorted);

    for (i=1; i<=g->nvertices; i++)
        if (discovered[i] == FALSE)
                dfs(g,i);

    print_stack(&sorted);     /* report topological order */
}
```

We push each vertex on a stack as soon as we have evaluated all outgoing edges. The top vertex on the stack always has no incoming edges from any vertex on the stack. Repeatedly popping them off yields a topological ordering.

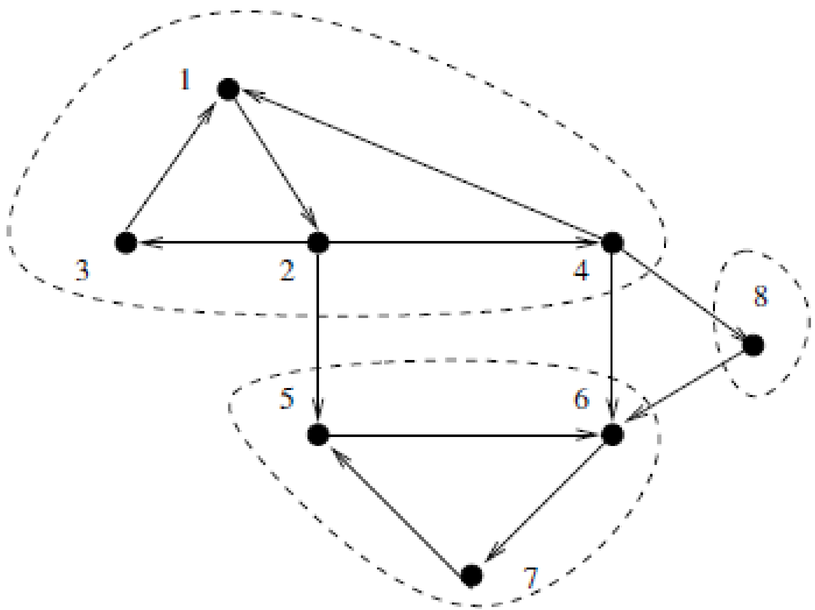
APPLICATION: STRONGLY CONNECTED COMPONENTS

- A directed graph is strongly connected iff there is a directed path between any two vertices.
- The strongly connected components of a graph is a partition of the vertices into subsets (maximal) such that each subset is strongly connected.

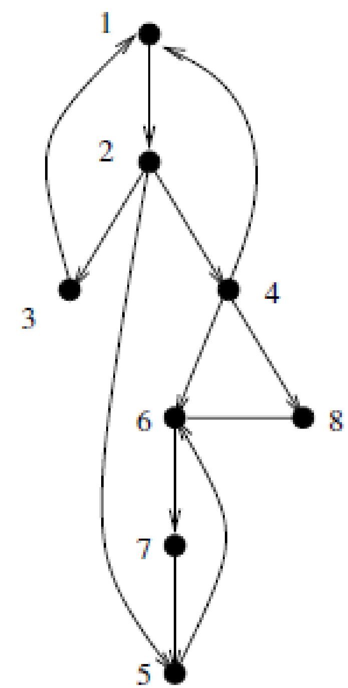


- Observe that no vertex can be in two maximal components, so it is a partition.

- There is an elegant, linear time algorithm to find the strongly connected components of a directed graph using DFS.



strongly-connected components



associated DFS tree

BACKTRACKING AND DEPTH-FIRST SEARCH

- Depth-first search uses essentially the same idea as backtracking.
- Both involve exhaustively searching all possibilities by advancing if it is possible, and backing up as soon as there is no unexplored possibility for further advancement.
- Both are most easily understood as recursive algorithms.

IMPLEMENTATION

```
strong_components(graph *g)
{
    int i;                  /* counter */
    for (i=1; i<=(g->nvertices); i++) {
        low[i] = i;
        scc[i] = -1;
    }
    components_found = 0;
    init_stack(&active);
    initialize_search(&g);

    for (i=1; i<=(g->nvertices); i++)
        if (discovered[i] == FALSE) {
        dfs(g,i);
    }
}
```

```
int low[MAXV+1]; /* oldest vertex surely in component of v */
int scc[MAXV+1]; /* strong component number for each vertex */

process_edge(int x, int y)
{
	int class; /* edge class */
	class = edge_classification(x,y);
	if (class == BACK) {
		if (entry_time[y] < entry_time[ low[x] ] )
			low[x] = y;
	}
	if (class == CROSS) {
		if (scc[y] == -1) /* component not yet assigned */
			if (entry_time[y] < entry_time[ low[x] ] )
				low[x] = y;
	}
}
```

```
process_vertex_early(int v) { push(&active,v); }
process_vertex_late(int v)
{
	if (low[v] == v)		/* edge (parent[v],v) cuts off scc */
		pop_component(v);

	if (entry_time[low[v]] < entry_time[low[parent[v]]])
		low[parent[v]] = low[v];
}
pop_component(int v)
{
	int t;				/* vertex placeholder */
	components_found = components_found + 1;
	scc[ v ] = components_found;
	while ((t = pop(&active)) != v) {
		scc[ t ] = components_found;
	}
}
```