04-630
Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 20

Graphs

- Types of graph
- Adjacency matrix representation
- Adjacency list representation
- Breadth-First Search traversal
- Depth-First Search traversal
- Topological Sorting
- Minimum Spanning Tree
  - Prim’s Algorithm
  - Kruskal’s algorithm
- Shortest Path Algorithms
  - Dijkstra’s algorithm
  - Floyd’s algorithm

Depth-First Search

- This implementation of DFS uses the idea of a traversal time for each vertex

- The clock ticks each time we enter or exit any vertex

- We keep track of the entry and exit time for each vertex

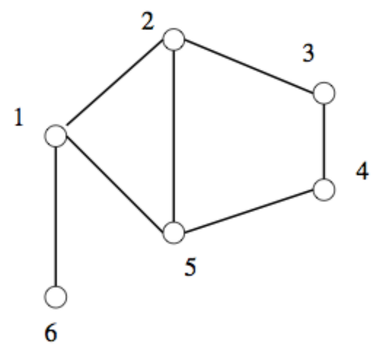
- These entry and exit times are useful in many applications of DFS (e.g. topological sort; see later)

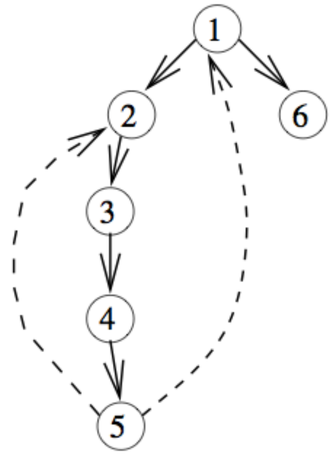
  - `process_vertex_early()` … take action on entry
  - `process_vertex_late()` … take action on exit

- DFS uses a stack but we can avoid using an explicit stack if we use recursion

Depth-First Search

- DFS partitions edges of an undirected graph into exactly two classes
  - Tree edges
  - Back edges





- Tree edges discover new vertices
  - Encoded in the `parent` relation
- Back edges link a vertex to an ancestor of the vertex being expanded

Depth-First Search

DFS(G, u)

$state[u]$ = "discovered"
process vertex u if desired
$entry[u] = time$
$time = time + 1$
for each $v \in Adj[u]$ do
  process edge (u, v) if desired
  if $state[v]$ = "undiscovered" then
    $p[v] = u$
    DFS(G, v)
$state[u]$ = "processed"
$exit[u] = time$
$time = time + 1$

Depth-First Search

```
/* Depth-First Search                                          */

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);
```

Depth-First Search

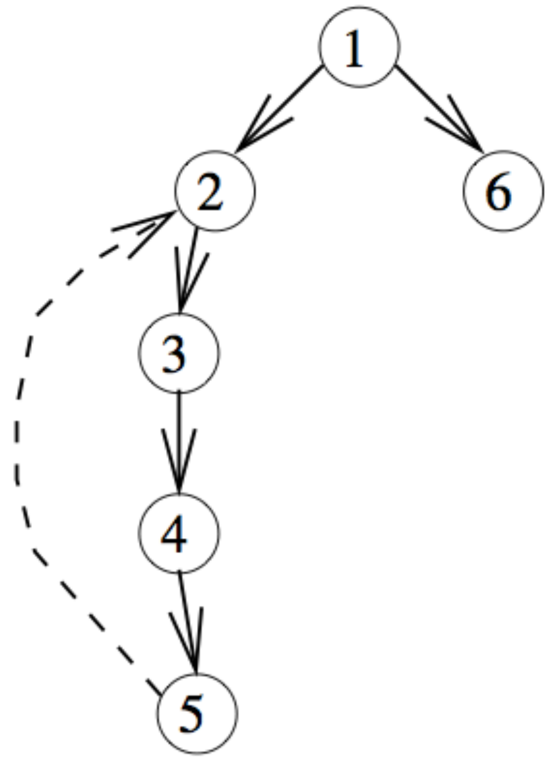
```
    p = g->edges[v];
    while (p != NULL) {
        y = p->y;
        if (discovered[y] == FALSE) {
            parent[y] = v;
            process_edge(v,y);      // not discovered: tree edge
            dfs(g,y);
        }
        else if ((!processed[y])   // discovered but not processed: back edge
                                   // e.g. (5,1) (5,2)
                 || (g->directed)) // discovered, possibly processed, but directed edge
                                   // also a back edge,
            process_edge(v,y);

        if (finished) return;
        p = p->next;
    }

    process_vertex_late(v);
    time = time + 1;
    exit_time[v] = time;

    processed[v] = TRUE;
}
```

Depth-First Search

- Depth-First Search uses essentially the same idea as backtracking
  - 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
- DFS organizes vertices by entry/exit times
- DFS classifies edges as either tree edges or back edges

Depth-First Search

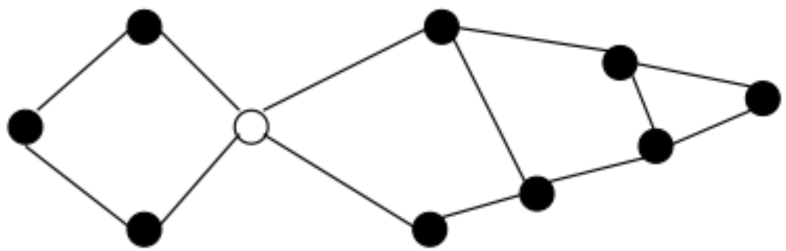
- Applications of Depth-First Search
  - Finding *Cycles*
    - If there are no back edges, then all edges are tree edges and no cycles exist
    - Finding a back edge identifies a cycle



Depth-First Search

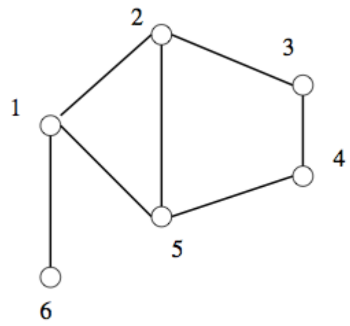
- Applications of Depth-First Search

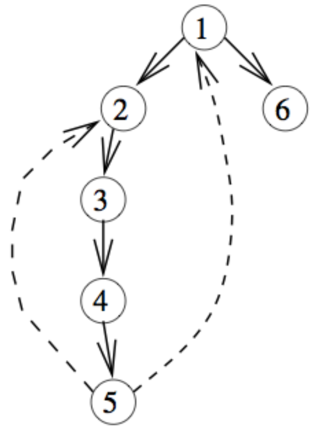
  – Finding *Articulation Vertices* (also known as a *cut node*): weakest points in a graph/network



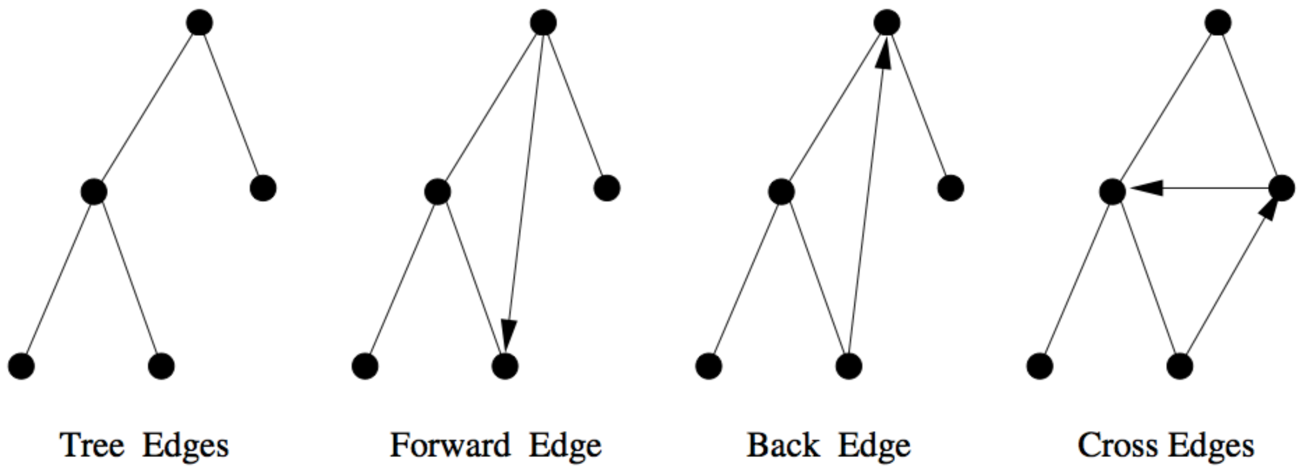
Depth-First Search

- Depth-First Search on Directed Graphs
  - When traversing undirected graphs, every edge is either in the depth-first search tree or it is a back edge to an ancestor in the tree





  - For directed graphs, there are 4 depth-first search labellings



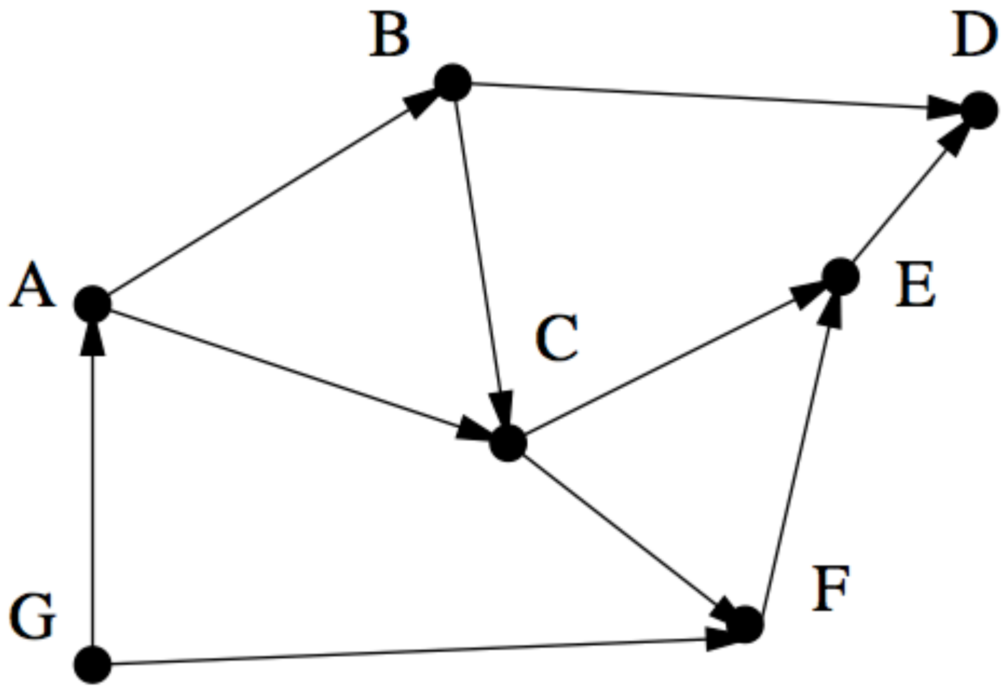
Depth-First Search

```
int edge_classification(int x, int y){

   /* if x is the parent of y, it's a tree edge                        */
   if (parent[y] == x) return(TREE);

   /* if y is discovered but not processed, this means we've           */
   /* already encountered on the traversal so it's a back edge         */
   if (discovered[y] && !processed[y]) return(BACK);

   /* if y has been processed, and its entry time is greater than x's */
   /* then it's a forward edge                                         */
   if (processed[y] && (entry_time[y]>entry_time[x])) return(FORWARD);

   /* if y has been processed, and its entry time is less than x's    */
   /* then it's a cross edge                                           */
   if (processed[y] && (entry_time[y]<entry_time[x])) return(CROSS);

   /* otherwise we have an invalid condition and it's unclassified.   */
   printf("Warning: unclassified edge (%d,%d)\n",x,y);
}
```

Topological Sorting

- The most important operation on directed acyclic graphs (DAGs)
- It orders the vertices on a line such that all directed edges go from left to right
  - Not possible if the graph contains a directed cycle
  - It provides a ordering to process each vertex before any of its successors
  - E.g. edges represent precedence constraints, such that the edge (x, y) means job x must be done before job y
  - Any topological sort defines a valid schedule
- Each DAG has at least one topological sort

Topological Sorting



A DAG with only one topological sort (G, A, B, C, F, E, D)

Topological Sorting

- Topological sorting can be performed using DFS
- A directed graph is a DAG iff there are no back edges
- Labelling the vertices in the reverse order in which they are *processed* (completed) finds the topological sort of a DAG

Topological Sorting

- Why? Consider what happens to each directed edge (x, y) as we encounter it exploring vertex x
  - If y is currently undiscovered, then we start a DFS of y before we can continue with x. Thus y is marked processed/completed before x is, and x appears before y in the topological order
  - If y is discovered but not processed/completed, then (x, y) is a back edge, which is forbidden in a DAG
  - If y is processed/completed, then it will have been so labeled before x. Therefore, x appears before y in the topological order

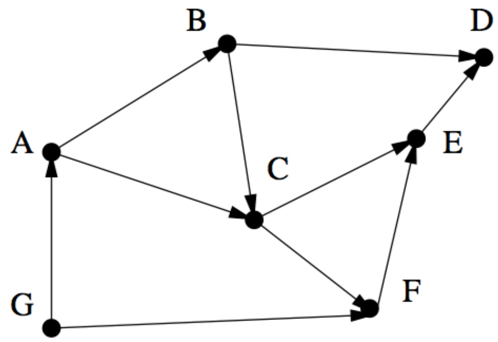
Topological Sorting

```
process_vertex_late(int v){
        push(&sorted,v);    // explicit stack for the sorted vertices
}

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");
}
```

Topological Sorting

```
/* Perform topological sort by doing a DFS on the graph,            */
/* pushing each vertex on a stack as soon as we have evaluated      */
/* all outgoing edges.                                              */
/* The top vertex on the stack always as no incoming edges from any */
/* vertex on the stack.                                             */
/* After the DFS, repeatedly popping the vertices from the stack    */
/* yields a topological ordering                                    */

topsort(graph *g) {

   int i;

   init_stack(&sorted);

   for (i=1; i<=g->nvertices; i++)
      if (discovered[i] == FALSE)
         dfs(g,i);  // push(&sorted,i) when processed

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

Topological Sorting



A DAG with only one topological sort (G, A, B, C, F, E, D)

```
DFS(g,A)-> DFS(g,B)  -> DFS(g,C)  -> DFS(g,E)  -> DFS(g,D)  -> Push(D)
                                                  Push(E)
                                    -> DFS(g,F)    Push(F)
                                    Push(C)
                        Push(B)
            Push(A)
DFS(g,G)-> Push(G)

Order of discovery:  A, B, C, E, D, F, G
Order of processing: D, E, F, C, B, A, G
Stack:     G
           A
           B
           C
           F
           E
           D
Topological Sort:    G, A, B, C, F, E, D
```