

Lecture 12: Depth-First Search

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794-4400

<http://www.cs.stonybrook.edu/~skiena>

Topic: Problem of the Day

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 .

Questions?

Topic: Depth-First Search

Depth-First Search

DFS has a neat recursive implementation which eliminates the need to explicitly use a stack.

Discovery and final times are a convenience to maintain.

```
void 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]) {
```

```

        parent[y] = v;
        process_edge(v, y);
        dfs(g, y);
    } else if ((!processed[y]) && (parent[v] != 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;
}

```

Backtracking and Depth-First Search

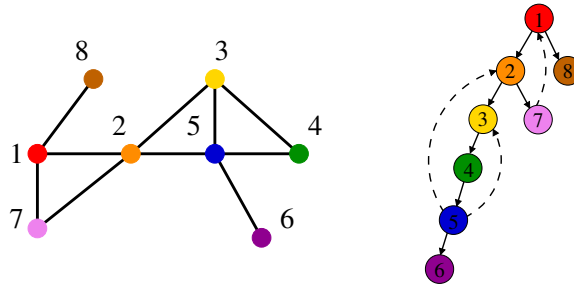
Depth-first search uses essentially the same idea as backtracking.

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.

The *Key Idea* with DFS

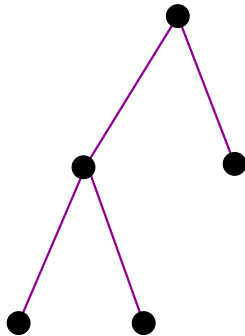
A depth-first search of a graph organizes the edges of the graph in a precise way.

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

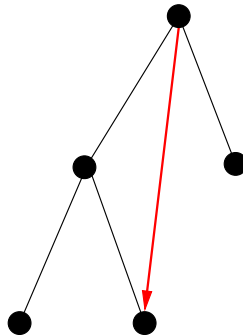


Edge Classification for DFS

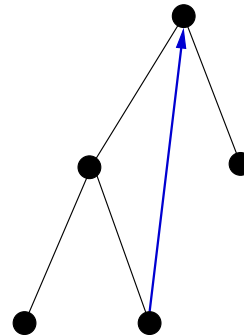
Every edge is either:



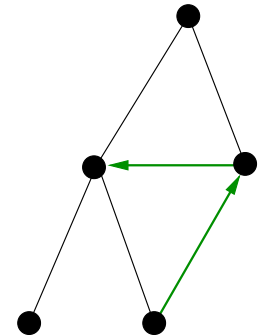
Tree Edges



Forward Edge



Back Edge



Cross Edges

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

Edge Classification Implementation

```
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: self loop (%d,%d)\n", x, y);

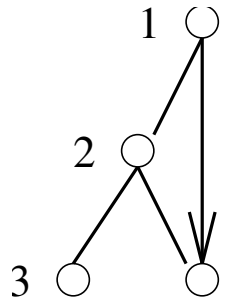
    return -1;
}
```

DFS: Tree Edges and Back Edges Only

The reason DFS is so important is that it defines a very nice ordering to the edges of the graph.

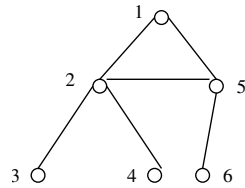
In a DFS of an undirected graph, every edge is either a tree edge or a back edge.

Why? Suppose we have a forward edge. We would have encountered $(4, 1)$ when expanding 4, so this is a back edge.

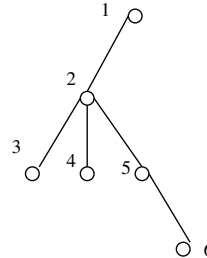


No Cross Edges in DFS

Suppose we have a cross-edge



When expanding 2, we would discover 5, so the tree would look like:



Questions?

Topic: Applications of DFS: Finding Cycles and Articulation Vertices

DFS Application: Finding Cycles

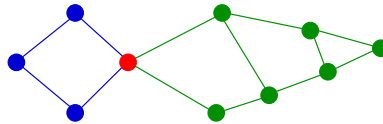
Back edges are the key to finding a cycle in an undirected graph.

Any back edge going from x to an ancestor y creates a cycle with the path in the tree from y to x .

```
void process_edge(int x, int y) {
    if (parent[y] != x) { /* found back edge! */
        printf("Cycle from %d to %d:", y, x);
        find_path(y, x, parent);
        finished = true;
    }
}
```


Articulation Vertices

Suppose you are a terrorist, seeking to disrupt the telephone network. Which station do you blow up?



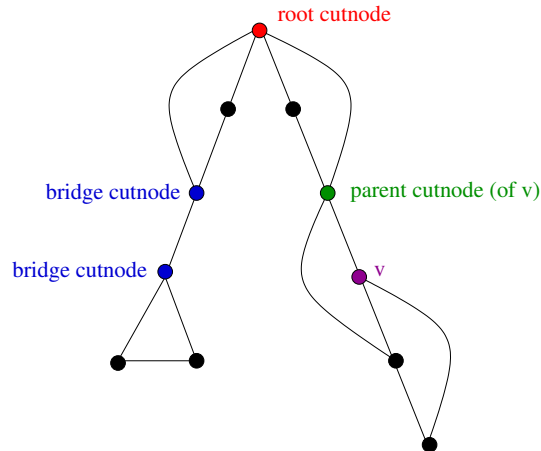
An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph.

Clearly connectivity is an important concern in the design of any network.

Articulation vertices can be found in $O(n(m+n))$ – just delete each vertex to do a DFS on the remaining graph to see if it is connected.

A Faster $O(n + m)$ DFS Algorithm

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 .

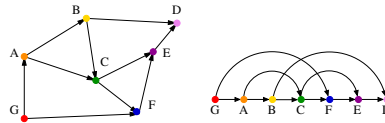


Questions?

Topic: Topological Sorting

Topological Sorting

A directed, acyclic graph has no directed cycles.



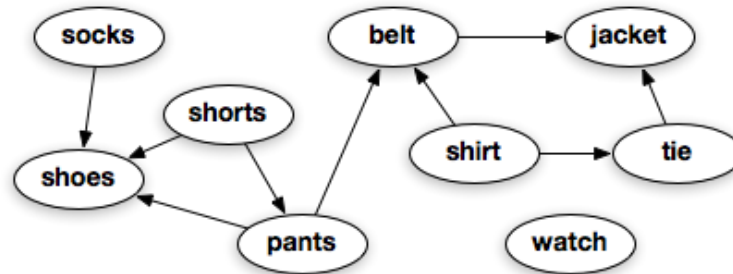
A topological sort of a graph is an ordering on the vertices so that all edges go from left to right.

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 priority schedule



Example: Identifying errors in DNA fragment assembly

Certain fragments are constrained to be to the left or right of other fragments, unless there are errors.

A B R A C A	<u>A B R A C A D A B R A</u>
R A C A D A	A B R A C A
A C A D A B	R A C A D A
C A D A B R	A C A D A B
A D A B R A	C A D A B R
	A D A B R A

Solution – build a DAG representing all the left-right constraints. Any topological sort of this DAG is a consistent ordering. If there are cycles, there must be errors.

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 *completed* before x is, and x appears before y in the topological order, as it must.
- If y is *discovered* but not *completed*, then $\{x, y\}$ is a back edge, which is forbidden in a DAG.
- If y is *completed*, then it will have been so labeled before x . Therefore, x appears before y in the topological order, as it must.

Topological Sorting Implementation

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

```
void 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");  
    }  
}
```

```
void topsort(graph *g) {
    int i;    /* counter */

    init_stack(&sorted);

    for (i = 1; i <= g->nvertices; i++) {
        if (!discovered[i]) {
            dfs(g, i);
        }
    }
    print_stack(&sorted);    /* report topological order */
}
```

We push each vertex on a stack 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.

Topological Sorting by Indegree-0 Node Removal

```
void topsort(graph *g, int sorted[]) {
    int indegree[MAXV+1];    /* indegree of each vertex */
    queue zeroIn;           /* vertices of indegree 0 */
    int x, y;               /* current and next vertex */
    int i, j;               /* counters */
    edgenode *p;            /* temporary pointer */

    compute_indegrees(g, indegree);
    init_queue(&zeroIn);
    for (i = 1; i <= g->nvertices; i++) {
        if (indegree[i] == 0) {
            enqueue(&zeroIn, i);
        }
    }

    j = 0;
    while (!empty_queue(&zeroIn)) {
        j = j + 1;
        x = dequeue(&zeroIn);
        sorted[j] = x;
        p = g->edges[x];
        while (p != NULL) {
```

```
        y = p->y;
        indegree[y]--;
        if (indegree[y] == 0) {
            enqueue(&zeroin, y);
        }
        p = p->next;
    }
}

if (j != g->nvertices) {
    printf("Not a DAG -- only %d vertices found\n", j);
}
}
```

```
void compute_indegrees(graph *g, int in[]) {
    int i;          /* counter */
    edgenode *p;   /* temporary pointer */

    for (i = 1; i <= g->nvertices; i++) {
        in[i] = 0;
    }

    for (i = 1; i <= g->nvertices; i++) {
        p = g->edges[i];
        while (p != NULL) {
            in[p->y]++;
            p = p->next;
        }
    }
}
```

Questions?

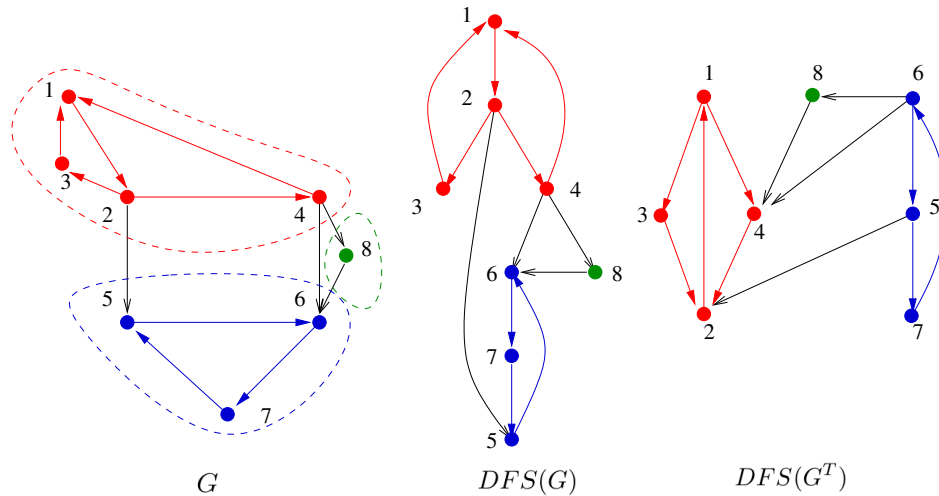
Topic: Applications of DFS: Strongly Connected Components

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 which is similar to the algorithm for biconnected components.

Questions?