# Graph Traversal

June 16, 2020

## 1 Introducition

A graph $ G = (V, E) $ consists of a set of *vertices* $ V $ together with a set $ E $ of vertex pairs or edges

Graphs can be used to represent any relationships. Graph theory provides a language for talking about the properties of relationships

**Figure 1**: Modeling road networks and electronic circuits as graph

## 2 Flavors of Graphs

### 2.1 Undirected vs Directed

A graph $ G = (V, E) $ is undirected if edge $ (x, y)  E $ implies that $ (x, y) $ is also in $ E $. If not, we say that the graph is *directed*.

Most graphs of graph-theoretic intrest are undirected

### 2.2 Weighted vs. Unweighted

Each Edge (or vertex) in a weighted graph $ G $ is assigned a numerical value, or weight. The edge of a road network graph might be weighted with thieir lenght, drive-time, or speed limit, etc. In unweighted graphs, thee is no cost distinction between various edges and vertices

### 2.3 Simple vs. Non-Simple

Certian types of edges complicate the taks of working with graphs. A *self-loop* is an edge (x, x) involving only one vertes. An edge $ (x, y) $ is a multiedge if it occurs more than once in the graph.

Both of the above graph require special care so any graph that avoids them is called a simple graph

### 2.4 Sparse vs. Dense

Graphs are sparce when only a small fraction of the possible vertex pairs ($ n/2 $ for a simple, undirected graph on $ n $ vertices) actually have edges defined between them.

Graphs where a large fraction of the vertex pairs define edges are called *dense*.

Typically dense graphs have a quadratic number of edges, while sparse graphs are linear in size

### 2.5 Cyclic vs. Acyclic

An *acyclic* graph does not contain any cycles. *Trees* are connected, acyclic undirected graphs. Trees are the simplest interesting graph, and are inherently recursive structures because cutting any edge leave two smaller trees

Directed acyclic graphs are called $ DAGs $. They arise naturalyl in scheduling problems, where a directe edge $ (x, y) $ indicates that activity $ x $ must occur before $ y $.

An operation called $ topological sorting $ orders the vertices of a $ DAG $ to respect these precedence constraints. Topological sorting is the first terp of any algorithm on a $ DAG $

### 2.6 Embedded vs. Topological

A graph is *embedded* if the vertices and edges are assigned geometric position. Thus, any drawing of a graph is embedding which may or may not have algorithmic significance

For example, when given a collection of points in the plane, and seek the minimum cost tour visiting all of them (traveling salesman problem), the underlying topology is the *complete graph* connecting each pair of vertices. The weights are typically defined by Euclidean distance between each pair of points

Grids of points are another example of topology from geometry. Many problems on a $ n * m $ grid involve walking between neightborhood points, so the edges re implicitly defined from the geometry

### 2.7 Implicit vs Explicit

Certain graphs are not explicitly constructed and then traversed, but built as we use them.

A good example is the backtrack search. The vertices of this implicit search graph are the state of the search vector, while edges link pairs of that can be directly generated from each other.

Because you do not have to have to store the entire graph, it is often easier to work with an implicit graph then explicitly construct it prior to analysis

### 2.8 Labled vs Unlabled

Each vertex is assigned a unique name or identifier in a *labled* graph to distinquish it from all other vertices. In unlabled graphs, no such distinctions have been made.

Most graph labels naturally emerge such as city names, etc. but unlabled graphs have their value. For instance, a common problem is *isomorphism testing* , which is determining whether the topological structure of two graphs are identical if we ignore any lables. These problems are gnerally solved using backtracking, by trying to assign each vertex in each graph a label such that the structure are identical

**Figure 2**: Important properties / flavors of graphs

## 3 Friendship Graph

**Figure 3**: A portion of the friendship graph

Most graphs that one encounters in real life are sparse. The friendship graph is a good example of that

### 3.1 If i am your friend, dows that mean you are my friend?

This question really asks whether the graph is directed. A graph is undirectt if edges $ (x, y) $ always implies $ (y, x) $. Otherwise its directed. Friendships are undirected because it take a pair to make friends

### 3.2 How close of a fried are you?

In *weighted* graphs, each edge has an associated numerical attribute. We could model the strenght of a friendship by associating each edge with an appropriate value, perhaps from $ -10$ (enemies) to $ 10 $ (blood brothers). The edges of a road network might be weighted with their lenghts, speed limit, etc. A graph is said to be *unweighted* if all edges are assumed to be of equal weight

### 3.3 Am i my own friend?

This question adresses whether the graph is simple, meaning it contains no loops or multipl edges. An edge of the form $ (x, x) $ is said to be a loop. Sometimes people are friends in several different ways. Perhaps $ x $ and $ y $ were collage classmates and now work together at the same company. We can model such relationshipd using *multiedges* - multiple edges $ (x, y) $ perhaps distinguished by different lables.

Since simple graphs are oftern eaiser to work with, we might be better off declaring that no one is their own friend

### 3.4 Who has the most friends

The *degree* of a vertex is the number of edges adjacent to it. The most popular person defines the vertex of highest degree in the friendship graph. Remote hermits are associated wit degree-zero vertices

In dense graphs, most vertices have high degrees, as opposed to *sparse* graphs with relatively few edges. In a *regular graph* , each vertex has exactly the same degree.

### 3.5 Do my friends live near me?

Social networds are not divorced from geography. Many of your friends are your friends only because they happen to live near you or uded to live near you.

Thus, a full understanding of social networks requires and *embedded graph* where each vertex is asoociated with the point on this world where they live. The geographic information may not be explicitly encoded, but the fact that the graph is inherently embedded in the plane shapes our interpretation of any analysis

### 3.6 Oh, you also know her?

Social networks are build on the premise of explicitly defining the links between members and their member-friends. Such graphs consist of directed edges from person/vertex $ x $ professing his friendship to person/vertex $ y $

That said, the complete friendhsip graph of the world is represented implicitly. Each person knows who their friends are, but cannot find out about the other people's friendships except by asking them.

The six degree of separation theory argues that there is a short path linking every two people in the world but offers us no help in actually finding the path

### 3.7 Are you truely individual, or just one of the faceless crowd?

Does each vertex have a name/label which reflects its identity, and is this label important for our analysis?

Alot of the time, we just lable the vertex with an index. A person studying infectious disease may label each vertex with whether the person is healthy or sick but the name of the person is irrelevant

## 4 Data Structures for Graphs

The basic choices are **adjacency matrices** and **adjacency lists**. We assume $ G = (V, E) $ contains $ n $ vertices and $ m $ edges

**Figure 4**: The adjacency matrix and adjacency list of a given graph

**Figure 5**: Relative advantages of adjacency lists and matrices

**Adjacency lists are the right data strucutre for most applications of graphs**

### 4.1 Adjacency Matrix

We can represent $ G $ using an $ n * n $ matrix $ M $, where element $ M[i, j] = 1 $ if $ (i, j)$ is an edge of $ G $, and $ 0 $ if it isn't.

This allows us to answer fastly, "is $(i, j)$ in G?"

It allows for rapid updates for edge insertion and deletion

It may use excessive space for graphs with many verties and relatively few edges.

Adjacency Matrix's are good because they are simple to represent but inherent problem is that they are quadratic space on sparse graphs

### 4.2 Adjacency Lists

Make it harder to verify whether a given edge $ (i, j) $ is in $ G $, since we must search through the appropriate list to find the edge.

Howerver, it is surprisingly easy to design graph algorithms that avoid any need for such queries

Typically, we sweep through all the edges of the graph in one pass via a **BFS** or **DFS** traversal and update the implications of the current edge as we visit it

Below is code for an Adjacency list; We represent edges using an array of linked lists

```
#define MAXV 1000 /* maximum number of vertices */

typedef struct {
```

```
    int y; /* adjacency info */
    int weight; /* edge weight, if any */
    struct edgenode *next; /* next edge in list */
} edgenode;

typedef struct {
    edgenode *edges[MAXV+1]; /* adjacency info */
    int degree[MAXV+1]; /* outdegree of each vertex */
    int nvertices; /* number of vertices in graph */
    int nedges; /* number of edges in graph */
    bool directed; /* is the graph directed? */
} graph;
```

We represent directed edge $ (x, y)$ by an edgenode $ y $ in $ x's $ adjacency list. the degree feild of the `graph` counts the number of meaninful entries for the given vertex.

An undirected edge $ (x, y)$ appears twice in any adjacency-based graph structure, once as $ y $ in $ x$'s list and once as $ x $ in $ y$'s list. The boolean flag `directed` identifies whether the given graph is to be interperted as directed or undirected

```
initialize_graph(graph *g, bool directed)
{
    int i; /* counter */
    g -> nvertices = 0;
    g -> nedges = 0;
    g -> directed = directed;

    for (i=1; i<=MAXV; i++) g->degree[i] = 0;
    for (i=1; i<=MAXV; i++) g->edges[i] = NULL;
}
```

Actually reading the graph requires inserting each edge into this structure

```
read_graph(graph *g, bool directed)
{
    int i; /* counter */
    int m; /* number of edges */
    int x, y; /* vertices in edge (x,y) */

    initialize_graph(g, directed);

    scanf("%d %d",&(g->nvertices),&m);

    for (i=1; i<=m; i++) {
    scanf("%d %d",&x,&y);
    insert_edge(g,x,y,directed);
    }
}
```

the critical routine is `insert_edge`. The new `edgenode` is inserted at the begining of the appropriate adjacency list, since the order doesen't matter. We parameterize our insertion with the `directed`

Boolean flag, to identify whether we need to insert two copies of each edge or only one

```
insert_edge(graph *g, int x, int y, bool directed)
{
    edgenode *p; /* temporary pointer */

    p = malloc(sizeof(edgenode)); /* allocate edgenode storage */

    p->weight = NULL;
    p->y = y;
    p->next = g->edges[x];

    g->edges[x] = p; /* insert at head of list */

    g->degree[x] ++;

    if (directed == FALSE)
        insert_edge(g,y,x,TRUE);
    else
        g->nedges ++;
}
```

Printing the associated graph is just a matter of two nested loops, one through vertices, the other through adjacent edges

```
print_graph(graph *g)
{
    int i; /* counter */
    edgenode *p; /* temporary pointer */

    for (i=1; i<=g->nvertices; i++) {
        printf("%d: ",i);
        p = g->edges[i];
        while (p != NULL) {
            printf(" %d",p->y);
            p = p->next;
        }
        printf("\n");
    }
}
```

# 5 Traversing A Graph

we have to have a systematic way of going through a graph while keeping of where we have gone

## 5.1 Idea

The ifea behind graph traversal is to mark each vertex when we first vitit it and keep track of what we have not yet explored.

We realy on Boolean flags (or sets) to keep track of visited

Each vertex will exist in one of the three states - Undiscovered: the vertex is in its initial, virgin state - Discovered: the vertex has been found, but we have not yey checked out all its incident edges - Processed: the vertex after we have visited all its incident edges

### 5.2 Steps

State of vertex goes from `undiscovered` -> `discovered` -> `processed`

Initially, only the single start vertex is considered to be discovered. To completely explore a vertex $ v $, we must evaluate each edge, leaving $ v $.

If an edge goes to an undiscovered vertex $ v $, we must evaluate each edge leaving $ v $. If an edge goes to aan undiscovred vertex $ x $, we mark $ x $ undiscovered and add it to the list of work to do

We ignore an edge that goes to a processed vertex, because further contemplation will tell us nothing new about the graph

We can also ignore any edge going to a discovered but not procesed vertex, because the destination already resides on the list of vertices to process

Each undirected edge will be considered exactly twice, once when each of its endpoints is explored. Directed edges will be considered only once, when exploring the source vertex.

Every edge and vertex in the connected component must eventually be visited.

## 6 Breadth First Search

**Figure 9**: An undirected graph and its breadth-first search tree

In a breadth-first search of an undirected graph, we assign a direction to each edge, from the discovered $ u $ to the discovered $ v $. We thus denote $ u $ to be the parent of $ v $.

Since each node has exactly one parent, except for the root, this defines a tree on the vertices of the graph

In figure 9, the tree illustrates the shortest path from one one to another node

```
BFS(G,s)
    for each vertex u   V [G] - {s} do
        state[u] = "undiscovered"
        p[u] = nil, i.e. no parent is in the BFS tree
    state[s] = "discovered"
    p[s] = nil
    Q = {s}
    while Q !=  do
    u = dequeue[Q]
    process vertex u as desired
    for each v   Adj[u] do
        process edge (u,v) as desired
        if state[v] = "undiscovered" then
```

```
        state[v] = "discovered"
        p[v] = u
        enqueue[Q,v]
    state[u] = "processed"
```

### 6.1 Implementation

Our breath-first search implementation uses two Boolean arrays to maintain our knowladge about each vertex in the graph. A vertex is `discovered` the first time we visit it

A vertex is considered `processed` after we have traversed all outgoing edges from it. Thus each vertex passes from undiscovered to discovered to processed over the course of rh search. This information could have been maintained using one enumerated type variable, but we used two Boolean variables instead

```
bool processed[MAXV+1]; /* which vertices have been processed */
bool discovered[MAXV+1]; /* which vertices have been found */
int parent[MAXV+1]; /* discovery relation */

    Each vertex is initialized as undiscovered:

initialize_search(graph *g)
{
    int i; /* counter */
    for (i=1; i<=g->nvertices; i++) {
        processed[i] = discovered[i] = FALSE;
        parent[i] = -1;
    }
}
```

Onces a vertex is discovered, it is places on a queue. Since we process these vertices in first-in, first-out order, the oldest vertices are expanded first, which are exactly those closest to the root

```
bfs(graph *g, int start)
{
    queue q; /* queue of vertices to visit */
    int v; /* current vertex */
    int y; /* successor vertex */
    edgenode *p; /* temporary pointer */

    init_queue(&q);
    enqueue(&q,start);
    discovered[start] = TRUE;

    while (empty_queue(&q) == FALSE) {
        v = dequeue(&q);
        process_vertex_early(v);
        processed[v] = TRUE;
        p = g->edges[v];
        while (p != NULL) {
```

```
            y = p->y;
            if ((processed[y] == FALSE) || g->directed)
                process_edge(v,y);
            if (discovered[y] == FALSE) {
                enqueue(&q,y);
                discovered[y] = TRUE;
                parent[y] = v;
            }
            p = p->next;
        }
        process_vertex_late(v);
    }
}
```

### 6.2 Finding Paths

Except for the root every node has a parent. The parent defines a tree of discovery with the inital search node as the root of the tree

Because vertices are discovered in order of increasing distance of the root, this tree has a very important property. The unique tree path from the toor of each node $ x  V $ uses the smallest number of edges (or equivalently, intermediate nodes) possible on any root-to-$x$ path in the graph

We can reconstruct the path by following the chain of ancestory from $x$ to the root.

```
find_path(int start, int end, int parents[])
{
    if ((start == end) || (end == -1))
        printf("\n%d",start);
    else {
        find_path(start,parents[end],parents);
        printf(" %d",end);
    }
}
```

Breadth-First Search on figure 9 graph generates the following parent relation

## 7 Application of BFS

BFS runs in $ O (n + m) $ time on both directed or undirected graphs; $ n $-vertex and $ m$-edfes

### 7.1 Connected Components

We say that a graph is connected if there is a path between any two vertices.

A connected component of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices

Simply put, all vertices (nodes) need to have an edge going to them

Testing whether a puzzle such as a Rubik's cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configurations is connected

This is a good BFS problem because the vertex order does not matter. We take a node and check its connections, and continue with all other nodes/components

```
connected_components(graph *g)
{
    int c; /* component number */
    int i; /* counter */

    initialize_search(g);

    c = 0;
    for (i=1; i<=g->nvertices; i++)
        if (discovered[i] == FALSE) {
            c = c+1;
            printf("Component %d:",c);
            bfs(g,i);
            printf("\n");
    }
}

process_vertex_early(int v)
{
    printf(" %d",v);
}
    process_edge(int x, int y)
{
}
```

Important to note that the counter $ c $ is labeling all nodes/vertices that are the same with the same number

### 7.2 Two-Coloring Graphs

The vertex-coloring problem seeks to assign a label to each vertex of a graph such that no edge links any two vertices of the same color. We could avoid this by assigning each vertex a unique color

Howerver, the goal is to use a few colors as possible. Vertex coloring problems ofttern arise in scheduling applications such as register allocation in compilers

A graph is bipartite___ if it can be colored without conflicts while using two colors. Bipartite graphs aise in “had-sex” graph in heterosexual world.

Men have sen only with women, and vice versa. Thus gender defines a legal two-coloring int the model

For the model, we can augment breadth-first search so that whenever we discover a new vertex, we color it the oppsite of its parent. We check whether any nondiscovery edge links two vertices of the same color

Such conflict means that the graph cannot be-two colored. Otherwise, we will have constructed a

proper two-coloring whenever we terminate without conflict

```
twocolor(graph *g)
{
    int i; /* counter */
    for (i=1; i<=(g->nvertices); i++)
        color[i] = UNCOLORED;

    bipartite = TRUE;

    initialize_search(&g);

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

process_edge(int x, int y)
{
    if (color[x] == color[y]) {
        bipartite = FALSE;
    printf("Warning: not bipartite due to (%d,%d)\n",x,y);
    }
    color[y] = complement(color[x]);
}

complement(int color)
{
    if (color == WHITE) return(BLACK);
    if (color == BLACK) return(WHITE);
    return(UNCOLORED);
}
```

BFS can seprate the men from the women, but we can't tell them apart just by using the graph structure

# 8 Depth First Search

DSF and BFS differe on the datastucture to store the discovered vertexes but not processes

Queue: > By storing the vertices in a first-in, first-out (FIFO) queue, we explore the oldest unexplored vertices first. Thus our explorations radiate out slowly from the starting vertex, defining a BFS

Stack: > By storing the vertices in the last-in, first-out stack, we explore the vertices by lurching along a path, visiting a new neighbor if one is available, and backing up only whne we are surrounded by previously discovere vertices. Thus, out explorations quickly wander away from our starting

point, defining a DFS

Our implementation of dfs maintains a notion of traversal time for each vertex. Our time clock ticks each time we enter or exit any vertex. We keep track of the entry and exit times for each vertex. We keep track of the entry and exit times for each vertex

## 8.1 DFS recursive implementation

```
DFS(G,u)
    state[u] = "discovered"
    process vertex u if desired
    entry[u] = time
    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
```

The time intervals have intersting and useful properties with respect to depth-first search

Time Properties: - Who is an ancestor: - Suppose that $ x $ is an ancestor of $ y $ in the DFS tree. That implies that we must enter $ x $ before $ y $ since there is no eay we can be born before our own father or grandfather! We also must exit $ y $ before we exit $ x $, because the mechanics of $ DFS $ ensures we cannot exit $ x $ untill after we have backed up from the search of all its descendants. Thus the time interval of $ y $ must be properly nested within ancestor $ x $ - How mant Descendants: - The difference between the exit and entry times for $ v $ tells us how many descendents $ v $ has in the DFS tree. The clock gets incremented on each vertex entry and vertex exit, so half the time difference denotes the number of descendents of $ v $

These entry/exit times are important in several applications particularly topological sorting and biconnected/strongly-connected components

DFS partitions the edges of an undirected graph into two classes: *tree edges* and *back edges*. The tree edges and discover new vertices and are those encoded in the parent relation. Back edges are those whos other endpoint is an ancestor of the vertex being expanded, so they point back into the tree.

All edges fall into those two classes in DFS. This edge classification proves fundamental to the corectness of DFS-based algorithms

**Figure 5.10**: An undirected graph and its depth-first search tree

## 8.2 Implementation

A depth-first search can be though of as a breadth-first search with a stack instead of a queue. 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;
}
```

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

In undirected graphs, each edge $(x, y)$ stis in the adjacency list of vertex $ x $ and $ y $. So there are two potiental times we see an edge. The first time we see an edge, we may want to either process it or take a different action

To konw if we have seen an edge before or not is easy, just see if the vertex that make the edges $(x, y)$ are seen or not

### 8.3 Finding Cycles

Back edges are key in finding a cycle in a undirected graph. If there are no back edges, all edges are tree edges, and no cycle exists in a tree.

```
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;
    }
}
```

### 8.4 Articulation Vertices

**Figure 11**: An articulation vertex is the weakest point in the graph

If you were a vandal and you wanted to distrupt the telephone network, which vertex would you destroy

You would delete an *articulation vertex* or *cut node*.

The connectivity of a graph is the smallest number of vertices whos deletion will disconnect the graph

Brute force tests are easy. Simpyl delete each vertex $ v $ and then do a BFS or DFS traversal of the remaining graph to see if its connected or not, but time is $ O (n(n + m ))$>

**Figure 12**: DFS tree of a graph contains two articulation vertices (namely $ 1 $ and $ 2$). Back edge $(5, 2)$. keeps vertices $ 3 $ and $ 4 $ from being cut-nodes. Vertices $ 5 $ and $ 6 $ escape as leaves of the DFS tree

The better more clever linear-time algo tests all the vertices of a connected graph using a single depth-first search

The DFS tree tells us that this tree connects all the vertices of the graph. If the $ DFS $ tree represented the entirety of the graph, all internal (nonleaf) nodes would be articulation vertices, since deleting any one of them would seprate a leaf from the root. Blowing up a leaf $ 2 $ and $ 6 $ cannot disconnect the tree, since it connects no one but itself to the main trunk

The root of the search tree is a special case because if it has only once child, it functions as a leaf. Else it is an articulation vertex

Finding articulation vertices requires maintaining the extent to which back edges (i.e., security cables) link chunks of the DFS tree back to ancestor nodes

let `reachable_ancestor[v]` denote the earliest reachable ancestor of vertex $ v $, meaning the oldest ancestor of $ v $ that we can reach by a combination of tree edges and back edges. Initially, `reachable_ancestor[v] = v`:

```
int reachable_ancestor[MAXV+1]; /* earliest reachable ancestor of v */
int tree_out_degree[MAXV+1]; /* DFS tree outdegree of v */
```

```
process_vertex_early(int v)
{
reachable_ancestor[v] = v;
}
```

We update `reachable_ancestor[v]` whenever we encounter a back edge that takes us to an earlier ancestor than we have previously seen. The relative age/rank of our ancestor can be determined from their `entry_time's`.

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

    int class; /* edge class */

    class = edge_classification(x,y);

    if (class == TREE)
        tree_out_degree[x] = tree_out_degree[x] + 1;

    if ((class == BACK) && (parent[x] != y)) {
        if (entry_time[y] < entry_time[ reachable_ancestor[x] ] )
            reachable_ancestor[x] = y;
    }
}
```

The key issue is determining how the reachability rlation impacts whether vertex $ v $ is an articulation vertex.

**Figure 13**: The three cases of articulation vertices: root, bridge, and parent cut-nodes

**Cases**: - Root cut-nodes - If the root of the DFS tree has two or more children, it must be an articulation vertex. No edges from the subtree of the second child can possibly connect to the subtree of the first child - Bridge cut-nodes - If the earliest reachable vertex is from $ v $ is $ v $, then deleting the single edge $(parent[v], v) $ disconnects the graph. Clearly $ parent[v] $ must be an articulation vertex since it cuts $ v $ from the graph. Vertex $ v $ is also an articulation vertex unless it is a leaf of the DFS tree. For any leaf, nothing falls off when you cut it. - Parent cut nodes - If the earliest reachable vertex from $ v $ is the parent of $ v $ then deleting the parent must serve $ v $ from the tree unless the parent it the root

The following routin below systematicall evaluates each of the three conditions as we back up from the vertex after traversing all outgoing edges. We use `entry_time[v]` to represent the age of the vertex $ v $. The reachability time `time_v` calculated below denotes the oldest vertex that can be reached using back edges.

```
process_vertex_late(int v)
{
    bool root; /* is the vertex the root of the DFS tree? */
    int time_v; /* earliest reachable time for v */
    int time_parent; /* earliest reachable time for parent[v] */
```

```
    if (parent[v] < 1) { /* test if v is the root */
        if (tree_out_degree[v] > 1)
            printf("root articulation vertex: %d \n",v);
        return;
    }

    root = (parent[parent[v]] < 1); /* is parent[v] the root? */
    if ((reachable_ancestor[v] == parent[v]) && (!root))
        printf("parent articulation vertex: %d \n",parent[v]);

    if (reachable_ancestor[v] == v) {
        printf("bridge articulation vertex: %d \n",parent[v]);
        if (tree_out_degree[v] > 0) /* test if v is not a leaf */
            printf("bridge articulation vertex: %d \n",v);
    }

    time_v = entry_time[reachable_ancestor[v]];
    time_parent = entry_time[ reachable_ancestor[parent[v]] ];

    if (time_v < time_parent)
        reachable_ancestor[parent[v]] = reachable_ancestor[v];
}
```

**Figure 14**: Possible edge cases for **BFS/DFS** traversal

We talked mostly about vertex disruption. But we could have easily talked about edge disruption. A single edge whos deletion disconnects the graph is called a *bridge* ; any graph without such an edge is said to be *edge-biconencted*.

Identifying wheter a given edge $(x, y)$ is a bridge is easily done in linear time by deleting the edge and testing whether the resulting graph is connected.

All bridges can be identified in the same $ O (n + m ) $ time. Edge $ (x, y )$ is a bridge if (1) it is a tree edge and $ (2) $ no back edge connects from $ y $ or below to $ x $ or above.

# 9 Depth-First Search on Directed Graphs

For directed graphs, DFS labeling can take on a wider range of possibilities. Indeed, all four of the cases in **Figure 5.14** can occur in traversing directed graphs.

The correct labeling of each edge can be readily determined from the state, discovery time, and parent of each vertex as encoded in the following function

```
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)\n",x,y);
```

```
}
```

Both BFS/DFS algortithms will traverse all edges in the same connected component as the starting point. Since we need to start with a vertex in each component to traverse a disconencted graph, we must start from any vertex remaining undiscovered after a component search

```
DFS-graph(G)
    for each vertex u  V [G] do
        state[u] = "undiscovered"
            for each vertex u  V [G] do
        if state[u] = "undiscovered" then
            initialize new component, if desired
            DFS(G,u)
```

## 9.1 Topological Sorting

**Figure 15**: A DAG with only one topological sort $ (G, A, B, C, F, E, D) $

Topological sorting is the most important operation on directed acyclic graphs $ (DAGs)$.

Such a graph of ordering cannot exist if there is a cycle

Topological sorting can help us finding the shortest path in a $ DAG $.

Topological sorting can be performed efficently using DFS searching. A directed graph is a DAG if and only if no back edges are encountered.

> Labeling the vertices in the reverse order that they are marked *processed* finds a topological sort of a $ DAG $.

The statement above is true because of the following

Consider that happens to each directed edge $ {x, y} $ s we encoutner it exploring vertex $ x $

- If $ y $ is currently discovered, then we start a DFS of $ y $ before we 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 copleted, then $x, y$ is a back edge, whic is forbidden in a $ DAG $
- If $ y $ processed, then it will have been so labeled before $ x$. Therefore, $ x $ appears before $ y $ in the topological order, as it must

```
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 the 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 yeilds a topological ordering

## 9.2 Strongle Connected Components

**Figure 16**: The strongly-conencted components of a graph, with the associated DFS tree

strongly connected components are components formed after partitioning a graph into chunks such that directed paths exist between all pairs of vertices within a given chunk.

A directed path is strongly connected if there is a directed path between any vertices.

It is stragithforward to use graph traversal to test whether a graph $ G = (V, E) $ is strongly conencted in linear time.

- first do a traversal from some arbitrary vertex $ v $. Every vertex in the graph had better be reachable from $ v $ (and hence discovered on the $ BFS $ or $ DFS $ starting from $ v $, other wise $ G $ cannot possibly be strongly connected.
- next construct $ G' = (V, E')$ with the same vertex and edge set as $ G $, but with all edges reversed- i.e, directed edge $ (x,y)  E $ iff $ (y,x)  E $. Thus, any path from $ v $ to $ z $ in $ G' $ corresponds to a path from $ z $ to $ v $ in $ G $. By doing a DFS from $ v $ in $ G'$, we find all vertices with path to $ v $ in $ G $. The graph is strongly connected iif all vertices in $ G $ can $ (1) $ reach $ v $ and $ (2)$ are reachable from $ v $

Graphs that are not strongly connected can be partitioned into strongly connected components

The set of such components and the weakly-connected edges that link them together can be determined using $ DFS $. The algorithm is based on the observarion that is is easy to find a directed cycle using a depth-first search

All vertices in this cycle must be in the same strongly connected component. This we can shrink (contract) the vertices on this cycle down to a single vertex representing the component and then

repeat. The process terminates when no directed cycles remains, and each vertex represents a different strongly conencted component

### 9.2.1 Implementation

we update our notion of the oldest-reachable vertex in respose to $ (1) $ nontree edges and (2) backing up from a vertex. Due to the directed nature, we must also contend with fordward edges (from a vertex to a descendant) and cross edges (from a vertex back to an nonancestor but previously discovered vertex)

Our algorithm will peal one storng component off the tree at a time, and assign each of its vertices the number of component it is in

```
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);
        }
}
```

Define `low[v]` to be the oldest vertex know to be in the strongly conencted component as $ v $. This vertex is not necessarily an ancestor, buy may be a distant cousin of $ v $ becauseo of cross edges. Cross edges that point vertices from *prviosuly* strong connected compoennts of the graph cannot help us, because there can be no way back from them to $ v $, but otherwise cross edges are fair game. Forward edges have no impack on reachability over the depth-first three edges, and hence can be disregarded

```
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;
    }
}
```

A new strongly conencted compoennt is found whenever the lowest reachable vertex from $ v $ is $ v $. If so, we can clear the stack of this component. Otherwise, we give our parent the benefit of the oldest ancestor we can reach and backtrack

```
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;
    }
}
```