Lecture 10: Graph Data Structures

Steven Skiena

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

http://www.cs.sunysb.edu/~skiena

Introduction

- \( G=(V, E) \)
  - \( V \) = vertex set (nodes)
  - \( E \) = edge set (arcs)
- Graph representation
  - Adjacency list
  - Adjacency matrix
- Graph search
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
    - Topological sort
    - Strongly connected components

Road Networks

In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges.
Electronic Circuits

In an electronic circuit, with junctions as vertices and components as edges.

A Simple Metabolic Pathway

Metabolic Regulation - Methionine Biosynthesis in E. coli

Evolutionary relationship among organisms based on similarity of their CYTOCHROME c proteins.

Green arrows - upregulation
Red arrows - downregulation
Thickness of arrow represents certainty of direction (up/down)
**A complete graph**

**Filter**
- choose a list of genes (MATING, marked in red)
- filter for these genes plus neighbouring genes from the graph

**Mutation network** \( \Delta_{\text{phi4}} \)

**Mutation network** \( \Delta_{\text{phi2}} \)

**Probabilistic network** (\( \Delta_{\text{phi3}} \) and \( \Delta_{\text{phi4}} \)) underlines a green tree of genes which are more interconnected. The genes are colored according to orientation in W14 (light red). The genes which are more interconnected are involved in the same cellular processes like mating (behaviour, pink), aminoacid metabolism (aam, red), osx-gene family (osx, light blue), inositol-phosphate (mitochondrion, dark blue), glucose (blue, purple) and a group of genes of unknown function (unknown, grey).
Huge search spaces – graphs

• Almost any combinatorial search/optimisation problem can be described as a problem over a search space, described as an implicit graph

Graphs

- Set of nodes \( |V| = n \)
- Set of edges \( |E| = m \)
  - Undirected edges/graph: pairs of nodes \((v,w)\)
  - Directed edges/graph: pairs of nodes \((v,w)\)
- Set of neighbors of \(v\): set of nodes connected by an edge with \(v\) (directed: in-neighbors, out-neighbors)
- Degree of a node: number of its neighbors (indegree, outdegree)
- Path: a sequence of nodes such that every two consecutive nodes constitute an edge
- Length of a path: number of nodes minus 1
- Distance between two nodes: the length of the shortest path between these nodes
- Diameter: the longest distance in the graph
Choose

- The boss wants to produce programs to solve the following two problems
  - Euler circuit problem: given a graph $G$, find a way to go through each edge exactly once.
  - Hamilton circuit problem: given a graph $G$, find a way to go through each vertex exactly once.
- The two problems seem to be very similar.
- Person A takes the first problem and person B takes the second.
- Outcome: Person A quickly completes the program, whereas person B works 24 hours per day and is fired after a few months.

Euler Circuit: The original Konigsberg bridge 1735

“Can one walk across the seven bridges and never cross the same one twice?”
Euler circuit

• Every vertex of this graph has an even degree, therefore there exists an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

Hamilton Circuit

A joke (continued):

• **Why?** no body in the company has taken Algorithmics class.
• **Explanation:**
  - Euler circuit problem can be easily solved in polynomial time.
  - Hamilton circuit problem is proved to be NP-hard.
  - So far, no body in the world can give a polynomial time algorithm for a NP-hard problem.
  - Conjecture: there does not exist polynomial time algorithm for this problem.
Algorithmics (6EAP)
Graphs I - terminology

Jaak Vilo
2020 Fall

Flavors of Graphs
The first step in any graph problem is determining which flavor of graph you are dealing with.
Learning to talk the talk is an important part of walking the walk.
The flavor of graph has a big impact on which algorithms are appropriate and efficient.

Directed vs. Undirected Graphs
A graph $G = (V, E)$ is undirected if edge $(x, y) \in E$ implies that $(y, x)$ is also in $E$.

Road networks between cities are typically undirected.
Street networks within cities are almost always directed because of one-way streets.
Most graphs of graph-theoretic interest are undirected.

Weighted vs. Unweighted Graphs
In weighted graphs, each edge (or vertex) of $G$ is assigned a numerical value, or weight.

The edges of a road network graph might be weighted with their length, drive-time or speed limit.
In unweighted graphs, there is no cost distinction between various edges and vertices.

Simple vs. Non-simple Graphs
Certain types of edges complicate the task of working with graphs. A self-loop is an edge $(x, x)$ involving only one vertex.
An edge $(x, y)$ is a multi-edge if it occurs more than once in the graph.

Any graph which avoids these structures is called simple.

Sparse vs. Dense Graphs
Graphs are sparse when only a small fraction of the possible number of vertex pairs actually have edges defined between them.

Graphs are usually sparse due to application-specific constraints. Road networks must be sparse because of road junctions.
Typically dense graphs have a quadratic number of edges while sparse graphs are linear in size.
**Cyclic vs. Acyclic Graphs**

An acyclic graph does not contain any cycles. Trees are connected acyclic undirected graphs.

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

**Implicit vs. Explicit Graphs**

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

A good example arises in backtrack search.

**Embedded vs. Topological Graphs**

A graph is embedded if the vertices and edges have been assigned geometric positions.

Example: TSP or Shortest path on points in the plane.
Example: Grid graphs.
Example: Planar graphs.

**Labeled vs. Unlabeled Graphs**

In labeled graphs, each vertex is assigned a unique name or identifier to distinguish it from all other vertices.

An important graph problem is isomorphism testing, determining whether the topological structure of two graphs are in fact identical if we ignore any labels.

**The Friendship Graph**

Consider a graph where the vertices are people, and there is an edge between two people if and only if they are friends.

This graph is well-defined on any set of people: SUNY SB, New York, or the world.
What questions might we ask about the friendship graph?

**If I am your friend, does that mean you are my friend?**

A graph is undirected if $(x, y)$ implies $(y, x)$. Otherwise the graph is directed.

The “heard-of” graph is directed since countless famous people have never heard of me!

The “had-sex-with” graph is presumably undirected, since it requires a partner.
Perception of/and experience

• Simple example of 5 entities (persons) and their relationships
• Who would you prefer to be?
• Who wouldn’t you want to be?
• And what if the relationship means “company A sells to company B”?
• What if relationship means “love”?

Am I linked by some chain of friends to the President?

A path is a sequence of edges connecting two vertices. Since Mel Brooks is my father’s-sister’s-husband’s cousin, there is a path between me and him:

How close is my link to the President?

If I were trying to impress you with how tight I am with Mel Brooks, I would be much better off saying that Uncle Lenny knows him than to go into the details of how connected I am to Uncle Lenny. Thus we are often interested in the shortest path between two nodes.

Is there a path of friends between any two people?

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

Who has the most friends?

The degree of a vertex is the number of edges adjacent to it.
Complete graph

• Every node is connected to every other node

• Clique – fully connected subgraph of a graph

Enumerate all possible graphs with 5 nodes

Q: How many?

• 5 nodes => 5*4/2 different possible undirected edges without self-loops
  • Consider the possibility of each edge being present or not...
  • Directed: ->, <->, none (4 options)

  4 ** 10 different graphs with 5 vertices

Subgraph

• Subset of vertices (m) \( V' \) is a subset of \( V \)
• Subset of edges (n) \( E' \) is a subset of \( E \), s.t. \( \{u,v\} \) in \( E' \) if \( u,v \) both in \( V' \)

• \( 2^m \) different subsets of vertices (= many!)

Line graph

Given a graph \( G \), its line graph \( L(G) \) is a graph such that
• each vertex of \( L(G) \) represents an edge of \( G \); and
• two vertices of \( L(G) \) are adjacent if and only if their corresponding edges share a common endpoint ("are adjacent") in \( G \).
Algorithmics (6EAP)
Graphs I – representation and traversal

Jaak Vilo
2020 Fall

Representation of Graphs

- Adjacency list: $O(V+E)$
  - Preferred for sparse graph
  - $|E| \ll |V|^2$
  - $\text{Adj}[u]$ contains all the vertices $v$ such that there is an edge $(u, v) \in E$
  - Weighted graph: $w(u, v)$ is stored with vertex $v$ in $\text{Adj}[u]$
  - No quick way to determine if a given edge is present in the graph

- Adjacency matrix: $O(V^2)$
  - Preferred for dense graph
  - Symmetry for undirected graph
  - Weighted graph: store $w(u, v)$ in the $(u, v)$ entry
  - Easy to determine if a given edge is present in the graph

Tradeoffs Between Adjacency Lists and Adjacency Matrices

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Adjacency Lists</th>
<th>Adjacency Matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space to store $(u, v)$ matrix</td>
<td>$V$</td>
<td>$V^2$</td>
</tr>
<tr>
<td>Space to store $(u, v, w)$ array</td>
<td>$V+E$</td>
<td>$V^2$</td>
</tr>
<tr>
<td>Space to find value $(u, v)$ in matrix</td>
<td>$V$</td>
<td>$V$</td>
</tr>
<tr>
<td>Space to find value $(u, v, w)$ in matrix</td>
<td>$V$</td>
<td>$V$</td>
</tr>
<tr>
<td>Less memory on small graphs?</td>
<td>$\text{Adj}[u]$</td>
<td>$V^2$</td>
</tr>
<tr>
<td>Less memory on big graphs?</td>
<td>$\text{Adj}[u]$</td>
<td>$V^2$</td>
</tr>
<tr>
<td>Edge insertion in dense graph?</td>
<td>$V^2$</td>
<td>$V^2$</td>
</tr>
<tr>
<td>Edge deletion in dense graph?</td>
<td>$V^2$</td>
<td>$V^2$</td>
</tr>
<tr>
<td>Faster to traverse the graph?</td>
<td>$V + E$</td>
<td>$V^2$</td>
</tr>
<tr>
<td>Faster to solve most problems?</td>
<td>$V + E$</td>
<td>$V^2$</td>
</tr>
</tbody>
</table>

Both representations are very useful and have different properties, although adjacency lists are probably better for most problems.
Edge list representation (dense)

```
A: 1  
B: 3  D: 5  E: 7  
C: 3  
D: 1  E: 1  
E: 7  
```

Traversing a Graph

One of the most fundamental graph problems is to traverse every edge and vertex in a graph. For efficiency, we must make sure we visit each edge at most twice. For correctness, we must do the traversal in a systematic way so that we don’t miss anything. Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze.

Marking Vertices

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored. Each vertex will always be in one of the following three states:

- **undiscovered** – the vertex in its initial, virgin state.
- **discovered** – the vertex after we have encountered it, but before we have checked out all its incident edges.
- **processed** – the vertex after we have visited all its incident edges.

Breadth-First Search (BFS)

- **Graph search**: given a source vertex $s$, explores the edges of $G$ to discover every vertex that is reachable from $s$
  - Compute the distance (smallest number of edges) from $s$ to each reachable vertex
  - Produce a breadth-first tree with root $s$ that contains all reachable vertices
  - Compute the shortest path from $s$ to each reachable vertex
- **BFS discovers all vertices at distance $k$ from $s$ before discovering any vertices at distance $k+1$**

Simple BFS from $n$

```
enqueue( Q, n )  
while Q not empty  
u = dequeue( Q )  
process u  
for each v in Adjacency( u )  // neighbours  
  if v not yet discovered  
    then enqueue( Q, v )
```

Data Structure for BFS

- **Adjacency list**
- **color[u]** for each vertex
  - **WHITE** if $u$ has not been discovered
  - **BLACK** if $u$ and all its adjacent vertices have been discovered
  - **GRAY** if $u$ has been discovered, but has some adjacent white vertices
  - Frontier between discovered and undiscovered vertices
- **d[u]** for the distance from (source) $s$ to $u$
- **p[u]** for predecessor of $u$
- **FIFO queue Q** to manage the set of gray vertices
  - Q stores all the gray vertices
### Example (BFS)

(Courtesy of Prof. Jim Anderson)

**Initialization**

- **Set up s and initialize Q**

```plaintext
BFS(G,s)
1 for each vertex u ∈ V[G] = {s}
2 do color[u] ← WHITE
3 d[u] ← ∞
4 π[u] ← NIL
5 color[s] ← GRAY
6 d[s] ← 0
7 π[s] ← NIL
8 Q ← Ø
9 ENQUEUE(Q, s)
10 while Q ≠ Ø
11 do u ← DEQUEUE(Q)
12 for each v ∈ Adj[u]
13 do if color[v] ← WHITE
14 then color[v] ← GRAY
15 d[v] ← d[u] + 1
16 π[v] ← u
17 ENQUEUE(Q, v)
18 color[u] ← BLACK
```

**Graph Representation**

1. **Initial State**
   - **Q:** s
   - **D:** s
   - **π:** s

2. **First Step**
   - **Q:** w, r
   - **D:** s, w
   - **π:** s, w

3. **Second Step**
   - **Q:** t, x, v
   - **D:** s, w, r
   - **π:** s, w, r

**Notes:**
- BFS explores the graph in a breadth-first manner, starting from the root node `s`.
- At each step, it visits all unvisited neighbors of the current node and marks them as visited.
- The process continues until all nodes are visited.
Analysis of BFS

- \( O(|V|+|E|) = O(n+m) \)
  - Each vertex is enqueued \( O(1) \) at most once \( \Rightarrow O(n) \)
  - No vertex is re-painted white
  - A vertex is inserted into queue and retrieved from there only once
  - Each adjacency list is scanned at most once \( \Rightarrow O(m) \)

Shortest path

- Print out the vertices on a shortest path from \( s \) to \( v \)

```python
PRINT-PATH(G, s, u)
1 if v = s
2 then print s
3 else if \( \pi[v] = NIL \)
4 then print "no path from" s "to" v "exists"
5 else PRINT-PATH(G, s, \( \pi[v] \))
6 print v
```

BFS – tree edges, cross edges

- For undirected graph \( G \), in BFS, every edge will be
  - Tree edge, or
  - Cross-edge (not ancestor, not descendent)

Connected Components

The connected components of an undirected graph are the separate “pieces” of the graph such that there is no connection between the pieces. Many seemingly complicated problems reduce to finding or counting connected components. For example, testing whether a puzzle such as Rubik’s cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configurations is connected. Anything we discover during a BFS must be part of the same connected component. We then repeat the search from any undiscovered vertex (if one exists) to define the next component, until all vertices have been found:
Rubik’s cube

There are many algorithms to solve scrambled Rubik’s Cubes. The maximum number of face turns needed to solve any instance of the Rubik’s Cube is 20,[1] and the maximum number of quarter turns is 26.[2] These numbers are also the diameters of the corresponding Cayley graphs of the Rubik’s Cube group. An algorithm that solves a cube in the minimum number of moves is known as God’s algorithm.

In 2009, Tomas Rokicki proved that 29 moves in the quarter-turn metric is enough to solve any scrambled cube.[19] And in 2014, Tomas Rokicki and Morley Davidson proved that the maximum number of quarter-turns needed to solve the cube is 26. [20]


Two-Coloring Graphs

The vertex coloring problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color.

A graph is bipartite if it can be colored without conflicts while using only two colors. Bipartite graphs are important because they arise naturally in many applications. For example, consider the “had-sex-with” graph in a heterosexual world. Men have sex only with women, and vice versa. Thus gender defines a legal two-coloring.

Bipartite graphs

- people and groups
- men-women
- Stable marriage
  - find matching that will not be “broken” by inevitable divorces
- Apples and Oranges

Finding a Two-Coloring

We can augment breadth-first search so that whenever we discover a new vertex, we color it the opposite of its parent.

```cpp
// two_color(graph g)
{ 
    int i;
    for (i = 0; i < g.size(); i++)
        color[i] = UNCOLORED;
    bipartite = TRUE;
    initial_search(g);
    for (i = 0; i < g.size(); i++)
        if (is_root(i) == FALSE)
            if (color[i] == WHITE) 
                color[i] = BLACK;
                bipartite = FALSE;
}
```

Depth-First Search (DFS)

- DFS: search deeper in the graph whenever possible
  - Edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it
  - When all of v’s edges have been explored (finished), the search backtracks to explore edges leaving the vertex from which v was discovered
  - This process continues until we have discovered all the vertices that are reachable from the original source vertex
  - If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source
  - The entire process is repeated until all vertices are discovered
- DFS will create a forest of DFS-trees

We can assign the first vertex in any connected component to be whatever color/sex we wish.
Recursive:

DFS(u):
  process u
  for each v in Adjacency(u)
    DFS(v) if v undiscovered

Data Structure for DFS

- Adjacency list
- color[u] for each vertex
  - WHITE if u has not been discovered
  - GRAY if u is discovered but not finished
  - BLACK if u is finished
- Timestamps: \(1 \leq d[u] < f[u] \leq 2|V|\)
  - \(d[u]\) records when u is first discovered (and grayed)
  - \(f[u]\) records when the search finishes examining u's adjacency list (and blacken u)
- \(\pi[u]\) for predecessor of u

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:

DFS: initialise and visit all yet unexplored vertices

DFS-visit – visit all reachable nodes

DFS-Visit(u)
1. color[u] ← GRAY \(\triangleright\) White vertex u has just been discovered.
2. time ← time + 1
3. d[u] ← time
4. for each v ∈ Adj(u) \(\triangleright\) Explore edge \((u, v)\).
5. do if color[v] = WHITE
6. then \(\pi[v] ← u\)
7. DFS-Visit(v)
8. color[u] ← BLACK \(\triangleright\) Blacken u; it is finished.
9. f[u] ← time \(\triangleright\) time + 1
Properties of DFS

- Time complexity: $\Theta(V + E)$
  - Loops on lines 1-3 and 5-7 of DFS: $\Theta(V)$
  - DFS-VISIT
    - Called exactly once for each vertex
    - Loops on lines 4-7 for a vertex $v$: $\left| \text{Adj}[v] \right|$
    - Total time $= \sum_{v \in V} \left| \text{Adj}[v] \right| = O(V + E)$
- DFS results in a forest of trees
- Discovery and finishing times have parenthesis structure

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.

```c
DFS(graph *g, int v)
{
    charmark *P; (* temporary pointer *)
    int p; (* recursion return *)
    if (discovered[v] == TRUE)
        return;
    time = time + 1;
    start.time[v] = time;
    process(v, &discovered);
    p = g->adj(v);
    while (p != NULL) {
        if (start.time[v] < start.time[p] {*
            *p = g->adj(v);
            DFS(g, p);
            *p = g->adj(u);
        }
    }
    finish.time[v] = time;
}
```
Parenthesis Theorem

**Theorem 22.7**

For all $u, v$, exactly one of the following holds:

2. $d[u] < d[v] < f[v] < f[u]$ and $v$ is a descendant of $u$.

- Like parentheses:
  - OK: $() [][]()$ or $[]()$ ( )
  - Not OK: $([])$

**Corollary**

$v$ is a proper descendant of $u$ if and only if $d[u] < d[v] < f[u] < f[v]$.

---

Parenthesis theorem

In any depth-first search of a (directed or undirected) graph $G = (V, E)$, for any two vertices $u$ and $v$, exactly one of the following three conditions holds:

1. the intervals $[d[u], f[u])$ and $[d[v], f[v])$ are entirely disjoint, and neither $u$ nor $v$ is a descendant of the other in the depth-first forest,
2. the interval $[d[u], f[u])$ is contained entirely within the interval $[d[v], f[v])$, and $u$ is a descendant of $v$ in a depth-first tree, or
3. the interval $[d[v], f[v])$ is contained entirely within the interval $[d[u], f[u])$, and $v$ is a descendant of $u$ in a depth-first tree.

---

Example (Parenthesis Theorem)
Proof

We begin with the case in which \( d[u] < d[v] \).

- There are two subcases to consider, according to whether \( d[v] < f[u] \) or not.
- The first subcase occurs when \( d[v] < f[u] \), so \( v \) was discovered while \( u \) was still gray. This implies that \( v \) is a descendant of \( u \). Moreover, since \( v \) was discovered more recently than \( u \), all of its outgoing edges are explored, and \( v \) is finished, before the search returns to and finishes \( u \). In this case, therefore, the interval \([d[v], f[v]]\) is entirely contained within the interval \([d[u], f[u]]\).
- In the other subcase, \( f[u] < d[v] \), and inequality (22.2) implies that the intervals \([d[u], f[u]]\) and \([d[v], f[v]]\) are disjoint.
- Because the intervals are disjoint, neither vertex was discovered while the other was gray, and so neither vertex is a descendant of the other.
- The case in which \( d[v] < d[u] \) is similar, with the roles of \( u \) and \( v \) reversed in the above argument.

Depth-First Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is \( G_p = (V, E_p) \) where \( E_p = \{ (\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL} \} \).
  - How does it differ from that of BFS?
  - The predecessor subgraph \( G_p \) forms a \textit{depth-first forest} composed of several depth-first trees. The edges in \( E_p \) are called \textit{tree edges}.

White-path Theorem

Theorem 22.9

\( v \) is a descendant of \( u \) if and only if at time \( d[u] \), there is a path \( u \rightarrow v \) consisting of only white vertices. (Except for \( u \), which was just colored gray.)

Classification of Edges

- **Tree edge:** in the depth-first forest. Found by exploring \((u, v), \quad -- v \text{ was white}\)
- **Back edge:** \((u, v), \text{ where } u \text{ is a descendant of } v \text{ (in the depth-first tree)}, \quad -- v \text{ was gray}\)
- **Forward edge:** \((u, v), \text{ where } v \text{ is a descendant of } u, \text{ but not a tree edge.} \quad -- v \text{ was black and } d[u] < d[v]\)
- **Cross edge:** any other edge. Can go between vertices in same depth-first tree or in different depth-first trees. \quad -- v was black and \( d[u] > d[v] \)

Types of edges in DFS

- **Tree edge**
- **Back edge**
- **Forward edge**
- **Cross edge**

Types of edges in DFS

- **Tree edge**
- **Back edge**
- **Forward edge**
- **Cross edge**
**Edge Classification for DFS**

Every edge is either:

1. A Tree Edge
2. A Back Edge
3. A Forward Edge
4. A Cross Edge

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

**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.

(for undirected graphs)

**No Cross Edges in DFS**

Suppose we have a cross-edge

**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 \( x \).

```
process edge (r, s, t) {
  if (parent[r] = s) /* found back edge */
    print("Found edge from \( r \) to \( s \) (\( r \), \( s \)).
  find path(c, parent)"; false)

  failed = TRUE
```

**Lemma – DAG acyclicity**

- **DAG is acyclic if and only if DFS of G yields no back edges**
  - Suppose that there is a back edge \( (u, v) \). Then vertex \( v \) is an ancestor of vertex \( u \) in the depth-first forest. There is thus a path from \( v \) to \( u \) in \( G \), and the back edge \( (u, v) \) completes a cycle.
  - Suppose that \( G \) contains a cycle \( c \). We show that a DFS of \( G \) yields a back edge. Let \( v \) be the first vertex to be discovered in \( c \), and let \( (u, v) \) be the preceding edge in \( c \). At time \( d[v] \), the vertices of \( c \) form a path of white vertices from \( v \) to \( u \). By the white-path theorem (Theorem 22.9), vertex \( u \) becomes a descendant of \( v \) in the depth-first forest. Therefore, \( (u, v) \) is a back edge.

**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 $v$.

The root is a special case since it has no ancestors.

$X$ is an articulation vertex since its child vertex does not have a back edge to a proper ancestor.

**Recursive:**

$$\text{DFS}(u):$$
- process $u$
- for each $v$ in Adjacency($u$)
  - DFS($v$) if $v$ undiscovered

**Simple DFS from n (non-recursive)**

push $(Q, n)$

while $u = \text{pop}(Q)$
- process $u$
- for each $v$ in reverse Adjacency($u$)
  - push $(Q, v)$

**DFS**
- Stack

**BFS**
- Queue
Randomised search

Use:
Randomized Queue

Algorithmics (6EAP)
Graphs I – topological sorting, strongly connected components
Jaak Vilo
2020 Fall

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).
Topological Sort

- A topological sort of a directed acyclic graph (DAG) is a linear order of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.
  - If the graph contains cycles, no linear ordering is possible.
  - A topological sort can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.
- DAG are used in many applications to indicate precedence among events.

Example

(Courtesy of Prof. Jim Anderson)

Linked List:

Example

Linked List:
Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E

Example

Linked List:

A → B → D → C → E
Theorem: Correctness of Topological sort

- TOPOLOGICAL-SORT(G) produces a topological sort of a directed acyclic graph G
  - Suppose that DFS is run on a given DAG G to determine finishing times for its vertices. It suffices to show that for any pair of distinct vertices u, v, if there is an edge in G from u to v, then \( f(v) < f(u) \).
  - The linear ordering is corresponding to finishing time ordering
    - Consider any edge \((u, v)\) explored by DFS(G). When this edge is explored, \( v \) cannot be gray (otherwise, \((u, v)\) will be a back edge). Therefore \( v \) must be either white or black
      - If \( v \) is white, \( v \) becomes a descendant of \( u \), \( f(v) < f(u) \) (ex. pants & shoes)
      - If \( v \) is black, it has already been finished, so that \( f(v) \) has already been set \( f(v) < f(u) \) (ex. belt & jacket)

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.
Strongly Connected Components

- $G$ is strongly connected if every pair $(u, v)$ of vertices in $G$ is reachable from one another.
- A **strongly connected component (SCC)** of $G$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \rightarrow v$ and $v \rightarrow u$ exist.

Component Graph

- $G^{\text{SCC}} = (V^{\text{SCC}}, E^{\text{SCC}})$.
- $V^{\text{SCC}}$ has one vertex for each SCC in $G$.
- $E^{\text{SCC}}$ has an edge if there’s an edge between the corresponding SCC’s in $G$.
- $G^{\text{SCC}}$ for the example considered:

```
   a -- b
   |   |
   v   v
   c -- d
```

$G^{\text{SCC}}$ is a DAG

**Lemma 22.13**

Let $C$ and $C'$ be distinct SCC’s in $G$, let $u, v \in C, u', v' \in C'$, and suppose there is a path $u \rightarrow u'$ in $G$. Then there cannot also be a path $v' \rightarrow v$ in $G$.

**Proof:**

- Suppose there is a path $v' \rightarrow v$ in $G$.
- Then there are paths $u \rightarrow u' \rightarrow v'$ and $v' \rightarrow v \rightarrow u$ in $G$.
- Therefore, $u$ and $v'$ are reachable from each other, so they are not in separate SCC’s.

Transpose of a Directed Graph

- $G^T = \text{transpose of directed } G$.
  - $G^T = (V, E^T), E^T = \{(u, v) : (v, u) \in E\}$.
  - $G^T$ is $G$ with all edges reversed.
- Can create $G^T$ in $\Theta(V + E)$ time if using adjacency lists.
- $G$ and $G^T$ have the *same* SCC’s, $(u$ and $v$ are reachable from each other in $G$ if and only if reachable from each other in $G^T$).

Algorithm to determine SCCs

**SCC($G$)**

1. call DFS($G$) to compute finishing times $f[u]$ for all $u$
2. compute $G^T$
3. call DFS($G^T$), but in the main loop, consider vertices in order of decreasing $f[u]$ (as computed in first DFS)
4. output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

**Time:** $\Theta(V + E)$.

**Example:** On board.

Kosaraju’s algorithm, 1978

Tarjan – 1972

Gabow – 1999 (Cheriyan, Melhorn 1995)
**Lemma 22.14**

Let $C$ and $C'$ be distinct SCC’s in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.

**Proof:**

- Case 1: $d(C) < d(C')$
  - Let $s$ be the first vertex discovered in $C$.
  - At time $d(s)$, all vertices in $C$ and $C'$ are white. Thus, there exists a path of white vertices from $s$ to all vertices in $C$ and $C'$.
  - By the white-path theorem, all vertices in $C$ and $C'$ are descendants of $s$ in depth-first tree.
  - By the parenthesis theorem, $f(x) = f(C) > f(C')$. 

- Case 2: $d(C) > d(C')$
  - Let $y$ be the first vertex discovered in $C'$.
  - At time $d(y)$, all vertices in $C'$ are white and there is a white path from $y$ to each vertex in $C'$. All vertices in $C'$ become descendants of $y$. Again, $f(C) = f(C')$.
  - At time $d(y)$, all vertices in $C'$ are also white.
  - By earlier lemma, since there is an edge $(u, v)$, we cannot have a path from $C$ to $C'$.
  - So no vertex in $C'$ is reachable from $y$.
  - Therefore, at time $f(y)$, all vertices in $C$ are still white.
  - Therefore, for all $w \in C, f(w) > f(y)$, which implies that $f(C) > f(C')$.
SCCs and DFS finishing times

**Corollary 22.15**
Let $C$ and $C'$ be distinct SCC's in $G = (V, E)$. Suppose there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

**Proof:**
\* $(u, v) \in E^T \Rightarrow (v, u) \in E$.
\* Since SCC's of $G$ and $G^T$ are the same, $f(C') > f(C)$, by Lemma 22.14.

Correctness of SCC

\* When we do the second DFS, on $G^T$, start with SCC $C$ such that $f(C)$ is maximum.
  \* The second DFS starts from some $x \in C$, and it visits all vertices in $C$.
  \* Corollary 22.15 says that since $f(C) > f(C')$ for all $C \neq C'$, there are no edges from $C$ to $C'$ in $G^T$.
  \* Therefore, DFS will visit only vertices in $C$.
\* Which means that the depth-first tree rooted at $x$ contains exactly the vertices of $C$.

Correctness of SCC

\* The next root chosen in the second DFS is in SCC $C'$ such that $f(C')$ is maximum over all SCC’s other than $C$.
  \* DFS visits all vertices in $C'$, but the only edges out of $C'$ go to $C$, which we’ve already visited.
  \* Therefore, the only tree edges will be to vertices in $C'$.
\* We can continue the process.
\* Each time we choose a root for the second DFS, it can reach only
  \* vertices in its SCC—get tree edges to these,
  \* vertices in SCC’s already visited in second DFS—get no tree
e dges to these.

Strongly Connected Components

Example

Why does strongly connected component method work?

\* See CLRS (2-3 pages)