CS 374: Algorithms & Models of Computation

Chandra Chekuri   Lenny Pitt

University of Illinois, Urbana-Champaign

Spring 2015

Graphs, Representation, Search, DFS

Lecture 8

February 12, 2015

Part I

Graph Basics

Why Graphs?

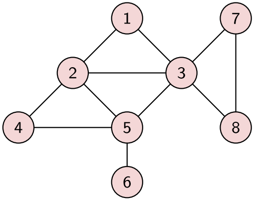
1. Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links), and many problems that don't even look like graph problems.
2. Fundamental objects in Computer Science, Optimization, Combinatorics
3. Many important and useful optimization problems are graph problems
4. Graph theory: elegant, fun and deep mathematics

Graph

Definition

An undirected (simple) graph $G = (V, E)$ is a 2-tuple:

1. V is a set of vertices (also referred to as nodes/points)
2. E is a set of edges where each edge $e \in E$ is a set of the form $\{u, v\}$ with $u, v \in V$ and $u \neq v$.



Example

In figure, $G = (V, E)$ where $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$.

Example: Modeling Problems as Search

State Space Search

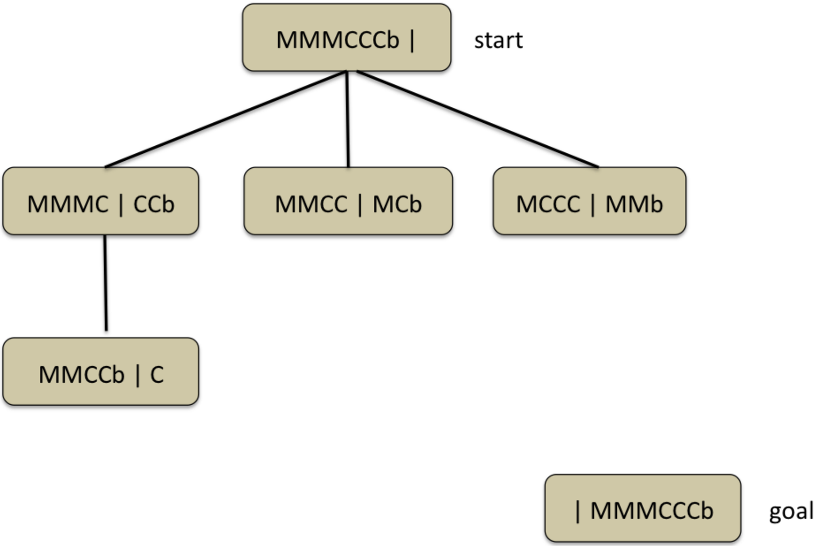
Many search problems can be modeled as search on a graph.
The trick is figuring out what the vertices and edges are.

Missionaries and Cannibals

- Three missionaries, three cannibals, one boat, one river
- Boat carries two people, must have at least one person
- Must all get across
- At no time can cannibals outnumber missionaries

How is this a graph search problem?
What are the vertices?
What are the edges?

Example: Missionaries and Cannibals Graph



Notation and Convention

Notation

An edge in an undirected graphs is an *unordered* pair of nodes and hence it is a set. Conventionally we use $(\mathbf{u}, \mathbf{v})$ for $\{\mathbf{u}, \mathbf{v}\}$ when it is clear from the context that the graph is undirected.

1. $\mathbf{u}$ and $\mathbf{v}$ are the end points of an edge $\{\mathbf{u}, \mathbf{v}\}$
2. Multi-graphs allow
   1. *loops* which are edges with the same node appearing as both end points
   2. *multi-edges*: different edges between same pairs of nodes
3. In this class we will assume that a graph is a simple graph unless explicitly stated otherwise.

Graph Representation I

Adjacency Matrix

Represent $G = (V, E)$ with n vertices and m edges using a $n \times n$ adjacency matrix A where

1. $A[i,j] = A[j,i] = 1$ if $\{i,j\} \in E$ and $A[i,j] = A[j,i] = 0$ if $\{i,j\} \notin E$.
2. Advantage: can check if $\{i,j\} \in E$ in $O(1)$ time
3. Disadvantage: needs $\Omega(n^2)$ space even when $m \ll n^2$

Graph Representation II

Adjacency Lists

Represent $\mathbf{G = (V, E)}$ with $\mathbf{n}$ vertices and $\mathbf{m}$ edges using adjacency lists:

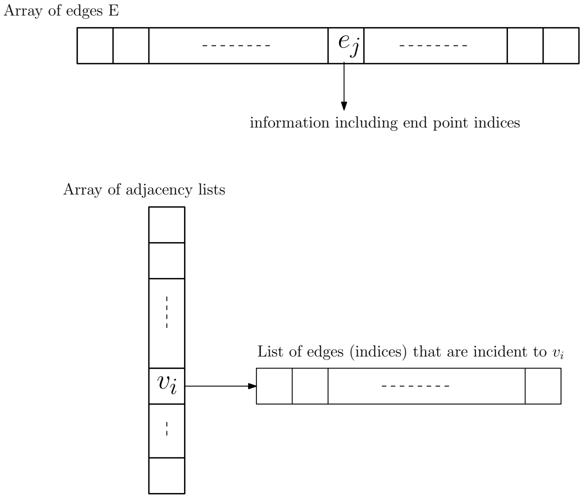
1. For each $\mathbf{u \in V}$, $\mathrm{Adj}(\mathbf{u}) = \{\mathbf{v} \mid \{\mathbf{u}, \mathbf{v}\} \in \mathbf{E}\}$, that is neighbors of $\mathbf{u}$. Sometimes $\mathrm{Adj}(\mathbf{u})$ is the list of edges incident to $\mathbf{u}$.
2. Advantage: space is $\mathbf{O(m + n)}$
3. Disadvantage: cannot "easily" determine in $\mathbf{O(1)}$ time whether $\{\mathbf{i}, \mathbf{j}\} \in \mathbf{E}$
   1. By sorting each list, one can achieve $\mathbf{O(\log n)}$ time
   2. By hashing "appropriately", one can achieve $\mathbf{O(1)}$ time

Note: In this class we will assume that by default, graphs are represented using plain vanilla (unsorted) adjacency lists.

A Concrete Representation

- Assume vertices are numbered arbitrarily as $\{1, 2, \ldots, n\}$.
- Edges are numbered arbitrarily as $\{1, 2, \ldots, m\}$.
- Edges stored in an array/list of size **m**. **E[j]** is **j**'th edge with info on end points which are integers in range **1** to **n**.
- Array **Adj** of size **n** for adjacency lists. **Adj[i]** points to adjacency list of vertex **i**. **Adj[i]** is a list of edge indices in range **1** to **m**.

A Concrete Representation



A Concrete Representation: Advantages

- Edges are explicitly represented/numbered. Scanning/processing all edges easy to do.
- Representation easily supports multigraphs including self-loops.
- Explicit numbering of vertices and edges allows use of arrays: $O(1)$-time operations are easy to understand.
- Can also implement via pointer based lists for certain dynamic graph settings.

Connectivity

Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ (the number of edges in the path) and the path is from v_1 to v_k. Note: a single vertex u is a path of length 0.

Connectivity

Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ (the number of edges in the path) and the path is from v_1 to v_k. Note: a single vertex u is a path of length 0.
2. A cycle is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$ and $\{v_1, v_k\} \in E$. Single vertex not a cycle according to this definition.
   Caveat: Some times people use the term cycle to also allow vertices to be repeated; we will use the term tour.

Connectivity

Given a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ (the number of edges in the path) and the path is from v_1 to v_k. Note: a single vertex u is a path of length 0.
2. A cycle is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k - 1$ and $\{v_1, v_k\} \in E$. Single vertex not a cycle according to this definition.
   Caveat: Some times people use the term cycle to also allow vertices to be repeated; we will use the term tour.
3. A vertex u is connected to v if there is a path from u to v.

Connectivity

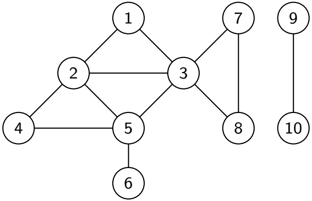
Given a graph $G = (V, E)$:

1. A path is a sequence of *distinct* vertices $v_1, v_2, \dots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ (the number of edges in the path) and the path is from v_1 to v_k. Note: a single vertex u is a path of length 0.
2. A cycle is a sequence of *distinct* vertices $v_1, v_2, \dots, v_k$ such that $\{v_i, v_{i+1}\} \in E$ for $1 \leq i \leq k-1$ and $\{v_1, v_k\} \in E$. Single vertex not a cycle according to this definition.
   Caveat: Some times people use the term cycle to also allow vertices to be repeated; we will use the term tour.
3. A vertex u is connected to v if there is a path from u to v.
4. The connected component of u, $\mathrm{con}(u)$, is the set of all vertices connected to u. Is $u \in \mathrm{con}(u)$?

Connectivity contd

Define a relation $\mathbf{C}$ on $\mathbf{V} \times \mathbf{V}$ as $\mathbf{uCv}$ if $\mathbf{u}$ is connected to $\mathbf{v}$

1. In undirected graphs, connectivity is a reflexive, symmetric, and transitive relation. Connected components are the equivalence classes.
2. Graph is connected if only one connected component.



Connectivity Problems

Algorithmic Problems

1. Given graph **G** and nodes **u** and **v**, is **u** *connected* to **v**?
2. Given **G** and node **u**, find all nodes that are connected to **u**.
3. Find all connected components of **G**.

Connectivity Problems

Algorithmic Problems

1. Given graph $\mathbf{G}$ and nodes $\mathbf{u}$ and $\mathbf{v}$, is $\mathbf{u}$ *connected* to $\mathbf{v}$?
2. Given $\mathbf{G}$ and node $\mathbf{u}$, find all nodes that are connected to $\mathbf{u}$.
3. Find all connected components of $\mathbf{G}$.

Can be accomplished in $\mathbf{O(m + n)}$ time using **BFS** or **DFS**.

Basic Graph Search

Given $G = (V, E)$ and vertex $u \in V$:

```
Explore(u):
    Initialize S = {u}
    while there is an edge (x, y) with x ∈ S and y ∉ S do
        add y to S
```

Basic Graph Search

Given $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ and vertex $\mathbf{u} \in \mathbf{V}$:

```
Explore(u):
    Initialize S = {u}
    while there is an edge (x, y) with x ∈ S and y ∉ S do
        add y to S
```

Proposition

Explore$(\mathbf{u})$ *terminates with* $\mathbf{S} = \mathit{con}(\mathbf{u})$.

Basic Graph Search

Given $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ and vertex $\mathbf{u} \in \mathbf{V}$:

```
Explore(u):
    Initialize S = {u}
    while there is an edge (x, y) with x ∈ S and y ∉ S do
        add y to S
```

Proposition

Explore(u) *terminates with* $\mathbf{S} = \mathit{con}(\mathbf{u})$.

Running time: depends on implementation

1. Naive: $\mathbf{O(mn)}$ with $\mathbf{O(m)}$ time for each scan.
2. Breadth First Search (**BFS**): use queue data structure
3. Depth First Search (**DFS**): use stack data structure
4. DFS/BFS run in $\mathbf{O(m + n)}$ time. Review CS 225 material!

Part II

DFS

Depth First Search

DFS is a very versatile graph exploration strategy. Hopcroft and Tarjan (Turing Award winners) demonstrated the power of **DFS** to understand graph structure. **DFS** can be used to obtain linear time ($\mathbf{O(m+n)}$) algorithms for

1. Finding cut-edges and cut-vertices of undirected graphs
2. Finding strong connected components of directed graphs
3. Linear time algorithm for testing whether a graph is planar

DFS in Undirected Graphs

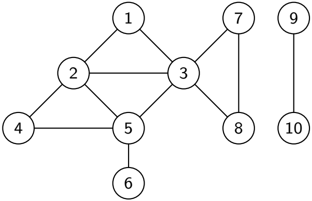
Recursive version.

```
DFS(G)
    Mark all nodes as unvisited
    while there is an unvisited node u do
        DFS(u)
```

```
DFS(u)
    Mark u as visited
    for each edge (u,v) in Adj(u) do
        if v is not marked
            DFS(v)
```

Implemented using a global array Mark for all recursive calls.

Example



DFS Tree/Forest

```
DFS(G)
    Mark all nodes unvisited
    Set T to be empty
    while ∃ unvisited node u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    for uv in Adj(u) do
        if v is not marked
            add uv to T
            DFS(v)
```

DFS Tree/Forest

```
DFS(G)
    Mark all nodes unvisited
    Set T to be empty
    while ∃ unvisited node u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    for uv in Adj(u) do
        if v is not marked
            add uv to T
            DFS(v)
```

Edges classified into two types: $uv \in E$ is a

1. tree edge: belongs to **T**
2. non-tree edge: does not belong to **T**

Properties of DFS tree

Proposition

1. $\mathbf{T}$ *is a forest*
2. *connected components of* $\mathbf{T}$ *are same as those of* $\mathbf{G}$.
3. *If* $\mathbf{uv} \in \mathbf{E}$ *is a non-tree edge then, in* $\mathbf{T}$, *either:*
   1. $\mathbf{u}$ *is an ancestor of* $\mathbf{v}$, *or*
   2. $\mathbf{v}$ *is an ancestor of* $\mathbf{u}$.

Question: Why are there no *cross-edges*?

DFS with Predecessors

Keep track of predecessors.

```
DFS(G)
    for all u ∈ V(G) do
        Mark u as unvisited
        Set pred(u) to null
    T is set to ∅
    while ∃unvisited u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    for each uv in Out(u) do
        if v is not marked then
            add edge uv to T
            set pred(v) to u
            DFS(v)
```

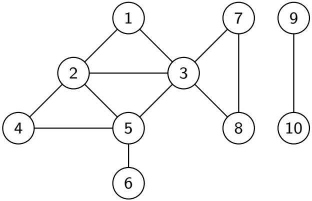
DFS with Visit Times

Keep track of when nodes are visited.

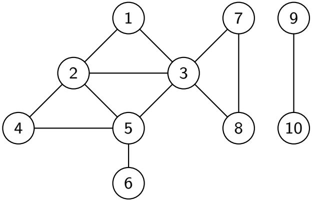
```
DFS(G)
    for all u ∈ V(G) do
        Mark u as unvisited
    T is set to ∅
    time = 0
    while ∃unvisited u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each uv in Out(u) do
        if v is not marked then
            add edge uv to T
            DFS(v)
    post(u) = ++time
```

Example



Example



pre and post numbers

Node **u** is **active** in time interval $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$

Proposition

For any two nodes **u** *and* **v**, *the two intervals* $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$ *and* $[\mathrm{pre}(\mathbf{v}), \mathrm{post}(\mathbf{v})]$ *are disjoint or one is contained in the other.*

pre and post numbers

Node **u** is **active** in time interval $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$

Proposition

For any two nodes **u** *and* **v**, *the two intervals* $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$ *and* $[\mathrm{pre}(\mathbf{v}), \mathrm{post}(\mathbf{v})]$ *are disjoint or one is contained in the other.*

Proof.

pre and post numbers

Node $\mathbf{u}$ is **active** in time interval $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$

Proposition

For any two nodes $\mathbf{u}$ *and* $\mathbf{v}$*, the two intervals* $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$ *and* $[\mathrm{pre}(\mathbf{v}), \mathrm{post}(\mathbf{v})]$ *are disjoint or one is contained in the other.*

Proof.

- Assume without loss of generality that $\mathrm{pre}(\mathbf{u}) < \mathrm{pre}(\mathbf{v})$. Then $\mathbf{v}$ visited after $\mathbf{u}$.

pre and post numbers

Node **u** is **active** in time interval $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$

Proposition

For any two nodes **u** *and* **v**, *the two intervals* $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$ *and* $[\mathrm{pre}(\mathbf{v}), \mathrm{post}(\mathbf{v})]$ *are disjoint or one is contained in the other.*

Proof.

- Assume without loss of generality that $\mathrm{pre}(\mathbf{u}) < \mathrm{pre}(\mathbf{v})$. Then **v** visited after **u**.
- If **DFS**(**v**) invoked before **DFS**(**u**) finished, $\mathrm{post}(\mathbf{v}) < \mathrm{post}(\mathbf{u})$.

pre and post numbers

Node **u** is **active** in time interval $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$

Proposition

For any two nodes **u** *and* **v**, *the two intervals* $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$ *and* $[\mathrm{pre}(\mathbf{v}), \mathrm{post}(\mathbf{v})]$ *are disjoint or one is contained in the other.*

Proof.

- Assume without loss of generality that $\mathrm{pre}(\mathbf{u}) < \mathrm{pre}(\mathbf{v})$. Then **v** visited after **u**.
- If **DFS**$(\mathbf{v})$ invoked before **DFS**$(\mathbf{u})$ finished, $\mathrm{post}(\mathbf{v}) < \mathrm{post}(\mathbf{u})$.
- If **DFS**$(\mathbf{v})$ invoked after **DFS**$(\mathbf{u})$ finished, $\mathrm{pre}(\mathbf{v}) > \mathrm{post}(\mathbf{u})$. □

pre and post numbers

Node **u** is **active** in time interval $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$

Proposition

For any two nodes **u** *and* **v***, the two intervals* $[\mathrm{pre}(\mathbf{u}), \mathrm{post}(\mathbf{u})]$ *and* $[\mathrm{pre}(\mathbf{v}), \mathrm{post}(\mathbf{v})]$ *are disjoint or one is contained in the other.*

Proof.

- Assume without loss of generality that $\mathrm{pre}(\mathbf{u}) < \mathrm{pre}(\mathbf{v})$. Then **v** visited after **u**.
- If **DFS**$(\mathbf{v})$ invoked before **DFS**$(\mathbf{u})$ finished, $\mathrm{post}(\mathbf{v}) < \mathrm{post}(\mathbf{u})$.
- If **DFS**$(\mathbf{v})$ invoked after **DFS**$(\mathbf{u})$ finished, $\mathrm{pre}(\mathbf{v}) > \mathrm{post}(\mathbf{u})$. □

pre and post numbers useful in several applications of **DFS**- soon!

Part III

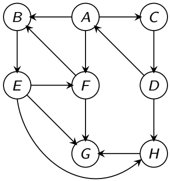
Directed Graphs and Decomposition

Directed Graphs

Definition

A directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ consists of

1. set of vertices/nodes $\mathbf{V}$ and
2. a set of edges/arcs $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$.



An edge is an *ordered* pair of vertices. $(\mathbf{u}, \mathbf{v})$ different from $(\mathbf{v}, \mathbf{u})$.

Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:

1. Road networks with one-way streets.
2. Web-link graph: vertices are web-pages and there is an edge from page $\mathbf{p}$ to page $\mathbf{p'}$ if $\mathbf{p}$ has a link to $\mathbf{p'}$. Web graphs used by Google with PageRank algorithm to rank pages.
3. Dependency graphs in variety of applications: link from $\mathbf{x}$ to $\mathbf{y}$ if $\mathbf{y}$ depends on $\mathbf{x}$. Make files for compiling programs.
4. Program Analysis: functions/procedures are vertices and there is an edge from $\mathbf{x}$ to $\mathbf{y}$ if $\mathbf{x}$ calls $\mathbf{y}$.

Representation

Graph $\mathbf{G = (V, E)}$ with $\mathbf{n}$ vertices and $\mathbf{m}$ edges:

1. **Adjacency Matrix**: $\mathbf{n \times n}$ *asymmetric* matrix $\mathbf{A}$. $\mathbf{A[u, v] = 1}$ if $\mathbf{(u, v) \in E}$ and $\mathbf{A[u, v] = 0}$ if $\mathbf{(u, v) \not\in E}$. $\mathbf{A[u, v]}$ is not same as $\mathbf{A[v, u]}$.
2. **Adjacency Lists**: for each node $\mathbf{u}$, $\mathbf{Out(u)}$ (also referred to as $\mathbf{Adj(u)}$) and $\mathbf{In(u)}$ store out-going edges and in-coming edges from $\mathbf{u}$.

Default representation is adjacency lists. Concrete representation discussed previously for undirected graphs easily extends to directed graphs.

Directed Connectivity

Given a graph $G = (V, E)$:

1. A **(directed) path** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from v_1 to v_k. By convention, a single node u is a path of length 0.

Directed Connectivity

Given a graph $G = (V, E)$:

1. A **(directed) path** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. The length of the path is $k - 1$ and the path is from v_1 to v_k. By convention, a single node u is a path of length 0.
2. A **cycle** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$ and $(v_k, v_1) \in E$. By convention, a single node u is not a cycle.

Directed Connectivity

Given a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$:

1. A **(directed) path** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ and the path is from v_1 to v_k. By convention, a single node u is a path of length 0.
2. A **cycle** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$. By convention, a single node u is not a cycle.
3. A vertex u can reach v if there is a path from u to v. Alternatively v can be reached from u

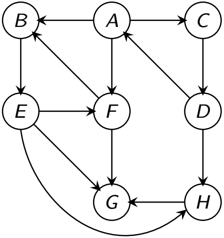
Directed Connectivity

Given a graph $\mathbf{G = (V, E)}$:

1. A **(directed) path** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ and the path is from v_1 to v_k. By convention, a single node u is a path of length 0.
2. A **cycle** is a sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$. By convention, a single node u is not a cycle.
3. A vertex u can reach v if there is a path from u to v. Alternatively v can be reached from u
4. Let $rch(u)$ be the set of all vertices reachable from u.

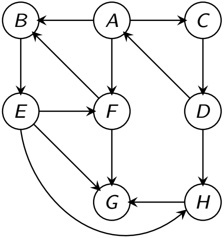
Connectivity contd

Asymmetricity: **D** can reach **B** but **B** cannot reach **D**



Connectivity contd

Asymmetricity: **D** can reach **B** but **B** cannot reach **D**



Questions:

1. Is there a notion of connected components?
2. How do we understand connectivity in directed graphs?

Connectivity and Strong Connected Components

Definition

Given a directed graph $\mathbf{G}$, $\mathbf{u}$ is strongly connected to $\mathbf{v}$ if $\mathbf{u}$ can reach $\mathbf{v}$ *and* $\mathbf{v}$ can reach $\mathbf{u}$. In other words $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$ and $\mathbf{u} \in \mathrm{rch}(\mathbf{v})$.

Connectivity and Strong Connected Components

Definition

Given a directed graph $\mathbf{G}$, $\mathbf{u}$ is strongly connected to $\mathbf{v}$ if $\mathbf{u}$ can reach $\mathbf{v}$ *and* $\mathbf{v}$ can reach $\mathbf{u}$. In other words $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$ and $\mathbf{u} \in \mathrm{rch}(\mathbf{v})$.

Define relation $\mathbf{C}$ where $\mathbf{uCv}$ if $\mathbf{u}$ is (strongly) connected to $\mathbf{v}$.

Connectivity and Strong Connected Components

Definition

Given a directed graph **G**, **u** is strongly connected to **v** if **u** can reach **v** *and* **v** can reach **u**. In other words $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$ and $\mathbf{u} \in \mathrm{rch}(\mathbf{v})$.

Define relation **C** where **uCv** if **u** is (strongly) connected to **v**.

Proposition

C *is an equivalence relation, that is reflexive, symmetric and transitive.*

Connectivity and Strong Connected Components

Definition

Given a directed graph $\mathbf{G}$, $\mathbf{u}$ is strongly connected to $\mathbf{v}$ if $\mathbf{u}$ can reach $\mathbf{v}$ *and* $\mathbf{v}$ can reach $\mathbf{u}$. In other words $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$ and $\mathbf{u} \in \mathrm{rch}(\mathbf{v})$.

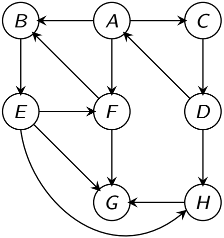
Define relation $\mathbf{C}$ where $\mathbf{uCv}$ if $\mathbf{u}$ is (strongly) connected to $\mathbf{v}$.

Proposition

$\mathbf{C}$ *is an equivalence relation, that is reflexive, symmetric and transitive.*

Equivalence classes of $\mathbf{C}$: *strong connected components* of $\mathbf{G}$.
They *partition* the vertices of $\mathbf{G}$.
$\mathrm{SCC}(\mathbf{u})$: strongly connected component containing $\mathbf{u}$.

Strongly Connected Components: Example



Directed Graph Connectivity Problems

1. Given $\mathbf{G}$ and nodes $\mathbf{u}$ and $\mathbf{v}$, can $\mathbf{u}$ reach $\mathbf{v}$?
2. Given $\mathbf{G}$ and $\mathbf{u}$, compute $\mathsf{rch}(\mathbf{u})$.
3. Given $\mathbf{G}$ and $\mathbf{u}$, compute all $\mathbf{v}$ that can reach $\mathbf{u}$, that is all $\mathbf{v}$ such that $\mathbf{u} \in \mathsf{rch}(\mathbf{v})$.
4. Find the strongly connected component containing node $\mathbf{u}$, that is $\mathrm{SCC}(\mathbf{u})$.
5. Is $\mathbf{G}$ strongly connected (a single strong component)?
6. Compute *all* strongly connected components of $\mathbf{G}$.

Directed Graph Connectivity Problems

1. Given $\mathbf{G}$ and nodes $\mathbf{u}$ and $\mathbf{v}$, can $\mathbf{u}$ reach $\mathbf{v}$?
2. Given $\mathbf{G}$ and $\mathbf{u}$, compute $\mathsf{rch}(\mathbf{u})$.
3. Given $\mathbf{G}$ and $\mathbf{u}$, compute all $\mathbf{v}$ that can reach $\mathbf{u}$, that is all $\mathbf{v}$ such that $\mathbf{u} \in \mathsf{rch}(\mathbf{v})$.
4. Find the strongly connected component containing node $\mathbf{u}$, that is $\mathrm{SCC}(\mathbf{u})$.
5. Is $\mathbf{G}$ strongly connected (a single strong component)?
6. Compute *all* strongly connected components of $\mathbf{G}$.

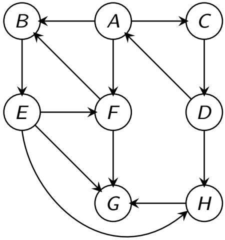
First five problems can be solved in $\mathbf{O(n + m)}$ time by adapting **BFS**/**DFS** to directed graphs. The last one requires a clever **DFS** based algorithm.

DFS in Directed Graphs

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to ∅
    time = 0
    while there is an unvisited node u do
        DFS(u)
    Output T
```

```
DFS(u)
    Mark u as visited
    pre(u) = ++time
    for each edge (u, v) in Out(u) do
        if v is not marked
            add edge (u, v) to T
            DFS(v)
    post(u) = ++time
```

Example



DFS Properties

Generalizing ideas from undirected graphs:

1. **DFS(u)** outputs a directed out-tree **T** rooted at **u**

DFS Properties

Generalizing ideas from undirected graphs:

1. **DFS(u)** outputs a directed out-tree **T** rooted at **u**
2. A vertex **v** is in **T** if and only if $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$

DFS Properties

Generalizing ideas from undirected graphs:

1. $\mathbf{DFS(u)}$ outputs a directed out-tree $\mathbf{T}$ rooted at $\mathbf{u}$
2. A vertex $\mathbf{v}$ is in $\mathbf{T}$ if and only if $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$
3. For any two vertices $\mathbf{x}, \mathbf{y}$ the intervals $[\mathrm{pre}(\mathbf{x}), \mathrm{post}(\mathbf{x})]$ and $[\mathrm{pre}(\mathbf{y}), \mathrm{post}(\mathbf{y})]$ are either disjoint or one is contained in the other.

DFS Properties

Generalizing ideas from undirected graphs:

1. $\mathbf{DFS(u)}$ outputs a directed out-tree $\mathbf{T}$ rooted at $\mathbf{u}$
2. A vertex $\mathbf{v}$ is in $\mathbf{T}$ if and only if $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$
3. For any two vertices $\mathbf{x}, \mathbf{y}$ the intervals $[\mathrm{pre}(\mathbf{x}), \mathrm{post}(\mathbf{x})]$ and $[\mathrm{pre}(\mathbf{y}), \mathrm{post}(\mathbf{y})]$ are either disjoint or one is contained in the other.
4. After initialization of Mark array and data structures: the running time of $\mathbf{DFS(u)}$ is $\mathbf{O(k)}$ where $\mathbf{k} = \sum_{\mathbf{v} \in \mathrm{rch}(\mathbf{u})} |\mathbf{Adj(v)}|$.

DFS Properties

Generalizing ideas from undirected graphs:

1. $\mathbf{DFS(u)}$ outputs a directed out-tree $\mathbf{T}$ rooted at $\mathbf{u}$
2. A vertex $\mathbf{v}$ is in $\mathbf{T}$ if and only if $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$
3. For any two vertices $\mathbf{x}, \mathbf{y}$ the intervals $[\mathrm{pre}(\mathbf{x}), \mathrm{post}(\mathbf{x})]$ and $[\mathrm{pre}(\mathbf{y}), \mathrm{post}(\mathbf{y})]$ are either disjoint or one is contained in the other.
4. After initialization of Mark array and data structures: the running time of $\mathbf{DFS(u)}$ is $\mathbf{O(k)}$ where $\mathbf{k} = \sum_{\mathbf{v} \in \mathrm{rch}(\mathbf{u})} |\mathbf{Adj(v)}|$.
5. $\mathbf{DFS(G)}$ takes $\mathbf{O(m+n)}$ time.

DFS Properties

Generalizing ideas from undirected graphs:

1. $\mathbf{DFS(u)}$ outputs a directed out-tree $\mathbf{T}$ rooted at $\mathbf{u}$
2. A vertex $\mathbf{v}$ is in $\mathbf{T}$ if and only if $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$
3. For any two vertices $\mathbf{x}, \mathbf{y}$ the intervals $[\mathrm{pre}(\mathbf{x}), \mathrm{post}(\mathbf{x})]$ and $[\mathrm{pre}(\mathbf{y}), \mathrm{post}(\mathbf{y})]$ are either disjoint or one is contained in the other.
4. After initialization of Mark array and data structures: the running time of $\mathbf{DFS(u)}$ is $\mathbf{O(k)}$ where $\mathbf{k} = \sum_{\mathbf{v} \in \mathrm{rch}(\mathbf{u})} |\mathbf{Adj(v)}|$.
5. $\mathbf{DFS(G)}$ takes $\mathbf{O(m + n)}$ time.
6. Edges added form a *branching*: a forest of out-trees. Output of $\mathbf{DFS(G)}$ depends on the order in which vertices are considered.

DFS Properties

Generalizing ideas from undirected graphs:

1. $\mathbf{DFS(u)}$ outputs a directed out-tree $\mathbf{T}$ rooted at $\mathbf{u}$
2. A vertex $\mathbf{v}$ is in $\mathbf{T}$ if and only if $\mathbf{v} \in \mathsf{rch}(\mathbf{u})$
3. For any two vertices $\mathbf{x, y}$ the intervals $[\mathrm{pre}(\mathbf{x}), \mathrm{post}(\mathbf{x})]$ and $[\mathrm{pre}(\mathbf{y}), \mathrm{post}(\mathbf{y})]$ are either disjoint or one is contained in the other.
4. After initialization of Mark array and data structures: the running time of $\mathbf{DFS(u)}$ is $\mathbf{O(k)}$ where $\mathbf{k} = \sum_{\mathbf{v} \in \mathsf{rch}(\mathbf{u})} |\mathbf{Adj(v)}|$.
5. $\mathbf{DFS(G)}$ takes $\mathbf{O(m + n)}$ time.
6. Edges added form a *branching*: a forest of out-trees. Output of $\mathbf{DFS(G)}$ depends on the order in which vertices are considered.

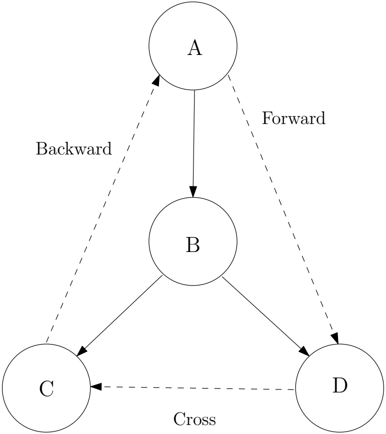
Note: Not obvious whether $\mathbf{DFS(G)}$ is useful in dir graphs but it is.

DFS Tree

Edges of **G** can be classified with respect to the **DFS** tree **T** as:

1. **Tree edges** that belong to **T**
2. A **forward edge** is a non-tree edges (x, y) such that $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
3. A **backward edge** is a non-tree edge (y, x) such that $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
4. A **cross edge** is a non-tree edges (x, y) such that the intervals $[\mathrm{pre}(x), \mathrm{post}(x)]$ and $[\mathrm{pre}(y), \mathrm{post}(y)]$ are disjoint.

Types of Edges



Directed Graph Connectivity Problems

1. Given $\mathbf{G}$ and nodes $\mathbf{u}$ and $\mathbf{v}$, can $\mathbf{u}$ reach $\mathbf{v}$?
2. Given $\mathbf{G}$ and $\mathbf{u}$, compute $\mathrm{rch}(\mathbf{u})$.
3. Given $\mathbf{G}$ and $\mathbf{u}$, compute all $\mathbf{v}$ that can reach $\mathbf{u}$, that is all $\mathbf{v}$ such that $\mathbf{u} \in \mathrm{rch}(\mathbf{v})$.
4. Find the strongly connected component containing node $\mathbf{u}$, that is $\mathrm{SCC}(\mathbf{u})$.
5. Is $\mathbf{G}$ strongly connected (a single strong component)?
6. Compute *all* strongly connected components of $\mathbf{G}$.

Algorithms via DFS- I

1. Given $\mathbf{G}$ and nodes $\mathbf{u}$ and $\mathbf{v}$, can $\mathbf{u}$ reach $\mathbf{v}$?
2. Given $\mathbf{G}$ and $\mathbf{u}$, compute $\text{rch}(\mathbf{u})$.

Use $\mathbf{DFS(G, u)}$ to compute $\text{rch}(\mathbf{u})$ in $\mathbf{O(n + m)}$ time.

Algorithms via DFS- II

1. Given $\mathbf{G}$ and $\mathbf{u}$, compute all $\mathbf{v}$ that can reach $\mathbf{u}$, that is all $\mathbf{v}$ such that $\mathbf{u} \in \text{rch}(\mathbf{v})$.

Algorithms via DFS- II

1. Given $\mathbf{G}$ and $\mathbf{u}$, compute all $\mathbf{v}$ that can reach $\mathbf{u}$, that is all $\mathbf{v}$ such that $\mathbf{u} \in \text{rch}(\mathbf{v})$.

   Naive: $\mathbf{O(n(n+m))}$

Algorithms via DFS- II

1. Given $\mathbf{G}$ and $\mathbf{u}$, compute all $\mathbf{v}$ that can reach $\mathbf{u}$, that is all $\mathbf{v}$ such that $\mathbf{u} \in \text{rch}(\mathbf{v})$.

   Naive: $\mathbf{O(n(n+m))}$

Definition (Reverse graph.)

Given $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, $\mathbf{G}^{\text{rev}}$ is the graph with edge directions reversed $\mathbf{G}^{\text{rev}} = (\mathbf{V}, \mathbf{E}')$ where $\mathbf{E}' = \{(\mathbf{y}, \mathbf{x}) \mid (\mathbf{x}, \mathbf{y}) \in \mathbf{E}\}$

Algorithms via DFS- II

1. Given $\mathbf{G}$ and $\mathbf{u}$, compute all $\mathbf{v}$ that can reach $\mathbf{u}$, that is all $\mathbf{v}$ such that $\mathbf{u} \in \text{rch}(\mathbf{v})$.

   Naive: $\mathbf{O(n(n + m))}$

Definition (Reverse graph.)

Given $\mathbf{G = (V, E)}$, $\mathbf{G^{rev}}$ is the graph with edge directions reversed $\mathbf{G^{rev} = (V, E')}$ where $\mathbf{E' = \{(y, x) \mid (x, y) \in E\}}$

Compute $\text{rch}(\mathbf{u})$ in $\mathbf{G^{rev}}$!

1. **Correctness:** exercise
2. **Running time:** $\mathbf{O(n + m)}$ to obtain $\mathbf{G^{rev}}$ from $\mathbf{G}$ and $\mathbf{O(n + m)}$ time to compute $\text{rch}(\mathbf{u})$ via **DFS**. If both $\mathbf{Out(v)}$ and $\mathbf{In(v)}$ are available at each $\mathbf{v}$ then no need to explicitly compute $\mathbf{G^{rev}}$. Can do $\mathbf{DFS(u)}$ in $\mathbf{G^{rev}}$ implicitly.

Algorithms via DFS- III

$\mathbf{SC(G, u) = \{v \mid u}$ is strongly connected to $\mathbf{v\}}$

Algorithms via DFS- III

$\mathbf{SC(G, u)} = \{\mathbf{v} \mid \mathbf{u}$ is strongly connected to $\mathbf{v}\}$

1. Find the strongly connected component containing node $\mathbf{u}$. That is, compute $\mathrm{SCC}(\mathbf{G}, \mathbf{u})$.

Algorithms via DFS- III

$\mathbf{SC(G, u) = \{v \mid u \text{ is strongly connected to } v\}}$

1. Find the strongly connected component containing node **u**. That is, compute $\mathrm{SCC}(\mathbf{G}, \mathbf{u})$.

$\mathrm{SCC}(\mathbf{G}, \mathbf{u}) = \mathrm{rch}(\mathbf{G}, \mathbf{u}) \cap \mathrm{rch}(\mathbf{G}^{\mathrm{rev}}, \mathbf{u})$

Algorithms via DFS- III

$\mathbf{SC(G, u) = \{v \mid u}$ is strongly connected to $\mathbf{v\}}$

1. Find the strongly connected component containing node $\mathbf{u}$. That is, compute $\mathrm{SCC}(\mathbf{G}, \mathbf{u})$.

$\mathrm{SCC}(\mathbf{G}, \mathbf{u}) = \mathsf{rch}(\mathbf{G}, \mathbf{u}) \cap \mathsf{rch}(\mathbf{G}^{\mathsf{rev}}, \mathbf{u})$

Hence, $\mathrm{SCC}(\mathbf{G}, \mathbf{u})$ can be computed with two **DFS**es, one in $\mathbf{G}$ and the other in $\mathbf{G}^{\mathsf{rev}}$. Total $\mathbf{O(n + m)}$ time.

Why can $\mathsf{rch}(\mathbf{G}, \mathbf{u}) \cap \mathsf{rch}(\mathbf{G}^{\mathsf{rev}}, \mathbf{u})$ be done in $\mathbf{O(n)}$ time?

Algorithms via DFS- IV

1. Is **G** strongly connected?

Algorithms via DFS- IV

1. Is $\mathbf{G}$ strongly connected?

Pick arbitrary vertex $\mathbf{u}$. Check if $\mathbf{SC(G, u) = V}$.

Algorithms via DFS- V

1. Find *all* strongly connected components of **G**.

Algorithms via DFS- V

1. Find *all* strongly connected components of G.

```
for each vertex u ∈ V do
        find SC(G, u)
```

Algorithms via DFS- V

1. Find *all* strongly connected components of $\mathbf{G}$.

```
for each vertex u ∈ V do
        find SC(G, u)
```

Running time: $\mathbf{O(n(n+m))}$.

Algorithms via DFS- V

1. Find *all* strongly connected components of $\mathbf{G}$.

```
for each vertex u ∈ V do
    find SC(G, u)
```

Running time: $\mathbf{O(n(n+m))}$.

Q: Can we do it in $\mathbf{O(n+m)}$ time?