CS 473: Algorithms

Chandra Chekuri
chekuri@cs.illinois.edu
3228 Siebel Center

University of Illinois, Urbana-Champaign

Fall 2009

Part I

Exponentiation, Binary Search

Exponentiation

Input Two numbers: a and integer $n \geq 0$

Goal Compute a^n

Exponentiation

Input Two numbers: a and integer $n \geq 0$

Goal Compute a^n

Obvious algorithm:

```
SlowPow(a,n):
    x = 1;
    for i = 1 to n do
        x = x*a
    Output x
```

$O(n)$ multiplications.

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,$\lfloor n/2 \rfloor$)
    x = x*x
    if (n is odd)
        x = x*a
    return x
```

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,$\lfloor n/2 \rfloor$)
    x = x*x
    if (n is odd)
        x = x*a
    return x
```

$T(n)$: number of multiplications for n

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,$\lfloor n/2 \rfloor$)
    x = x*x
    if (n is odd)
        x = x*a
    return x
```

$T(n)$: number of multiplications for n

$$T(n) = T(\lfloor n/2 \rfloor) + 2$$

$T(n) =$

Fast Exponentiation

Observation: $a^n = a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil} = a^{\lfloor n/2 \rfloor} a^{\lfloor n/2 \rfloor} a^{\lceil n/2 \rceil - \lfloor n/2 \rfloor}$.

```
FastPow(a,n):
    if (n = 0) return 1
    x = FastPow(a,$\lfloor n/2 \rfloor$)
    x = x*x
    if (n is odd)
        x = x*a
    return x
```

$T(n)$: number of multiplications for n

$$T(n) = T(\lfloor n/2 \rfloor) + 2$$

$T(n) = \Theta(\log n)$.

Complexity of Exponentiation

Question: Is SlowPow() a polynomial time algorithm? FastPow?

Complexity of Exponentiation

Question: Is SlowPow() a polynomial time algorithm? FastPow?

Input size: $\log a + \log n$

Complexity of Exponentiation

Question: Is SlowPow() a polynomial time algorithm? FastPow?

Input size: $\log a + \log n$
Output size:

Complexity of Exponentiation

Question: Is SlowPow() a polynomial time algorithm? FastPow?

Input size: $\log a + \log n$
Output size: $n \log a$. Not necessarily polynomial in input size!

Both SlowPow and FastPow are polynomial in output size.

Exponentiation modulo a given number

Exponentiation in applications:

Input Three integers: a, $n \geq 0$, $p \geq 2$ (typically a prime)

Goal Compute $a^n \mod p$

Exponentiation modulo a given number

Exponentiation in applications:

Input Three integers: a, $n \geq 0$, $p \geq 2$ (typically a prime)

Goal Compute $a^n \mod p$

Input size: $\log a + \log n + \log p$

Output size: $O(\log p)$ and hence polynomial in input size.

Exponentiation modulo a given number

Exponentiation in applications:

Input	Three integers: a, $n \geq 0$, $p \geq 2$ (typically a prime)
Goal	Compute $a^n \mod p$

Input size: $\log a + \log n + \log p$
Output size: $O(\log p)$ and hence polynomial in input size.

Observation: $xy \mod p = ((x \mod p)(y \mod p)) \mod p$

Exponentiation modulo a given number

Exponentiation in applications:

Input Three integers: a, $n \geq 0$, $p \geq 2$ (typically a prime)

Goal Compute $a^n \mod p$

Input size: $\log a + \log n + \log p$
Output size: $O(\log p)$ and hence polynomial in input size.

Observation: $xy \mod p = ((x \mod p)(y \mod p)) \mod p$

```
FastPowMod(a,n,p):
    if (n = 0) return 1
    x = FastPowMod(a,$\lfloor n/2 \rfloor$,p)
    x = x*x  mod  p
    if (n is odd)
        x = x*a  mod p
    return x
```

FastPowMod is a polynomial time algorithm. SlowPowMod is not.

Binary Search in Sorted Arrays

Input Sorted array A of n numbers and number x

Goal Is x in A?

Binary Search in Sorted Arrays

Input Sorted array A of n numbers and number x

Goal Is x in A?

```
BinarySearch(A[a..b], x):
    if (b-a <= 0) return NO
    mid = A[$\lfloor (a+b)/2 \rfloor$]
    if (x = mid) return YES
    else if (x < mid) return BinarySearch(A[a..$\lfloor (a+b)/2 \rfloor - 1$],x)
    else return BinarySearch(A[$\lfloor (a+b)/2 \rfloor + 1$..b],x)
```

Binary Search in Sorted Arrays

Input Sorted array A of n numbers and number x

Goal Is x in A?

```
BinarySearch(A[a..b], x):
    if (b-a <= 0) return NO
    mid = A[$\lfloor (a+b)/2 \rfloor$]
    if (x = mid) return YES
    else if (x < mid) return BinarySearch(A[a..$\lfloor (a+b)/2 \rfloor - 1$],x)
    else return BinarySearch(A[$\lfloor (a+b)/2 \rfloor + 1$..b],x)
```

Analysis: $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. $T(n) = O(\log n)$.
Observation: After k steps, size of array left is $n/2^k$

Another common use of binary search

- Optimization version: find solution of best (say minimum) value
- Decision version: is there a solution of value at most a given value v?

Another common use of binary search

- Optimization version: find solution of best (say minimum) value
- Decision version: is there a solution of value at most a given value v?

Reduce optimization to decision (may be easier to think about):

- Given instance I ompute upper bound $U(I)$ on best value
- Compute lower bound $L(I)$ on best value
- Do binary search on interval $[L(I), U(I)]$ using decision version as black box
- $O(\log(U(I) - L(I)))$ calls to decision version if $U(I), L(I)$ are integers

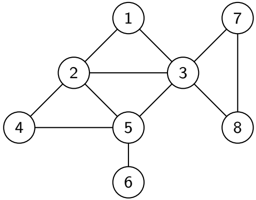
Part II

Graph Basics

Why Graphs?

- Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links) etc etc.
- Fundamental objects in Computer Science, Optimization, Combinatorics
- Many important and useful optimization problems are graph problems
- Graph theory: elegant, fun and deep mathematics

Graph



Definition

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

- V is a set of vertices (also referred to as nodes/points)
- 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 $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$.

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 (u, v) for $\{u, v\}$ when it is clear from the context that the graph is undirected.

- u and v are the end points of an edge $\{u, v\}$
- Multi-graphs allow
  - *loops* which are edges with the same node appearing as both end points
  - *multi-edges*: different edges between same pairs of nodes
- 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

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

Graph Representation II

Adjacency Lists

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

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

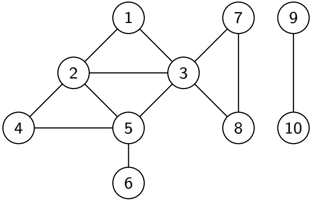
Note: In this class we will assume that by default, graphs are represented using plain vanilla (unsorted) adjacency lists.

Connectivity

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

- 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 $k-1$ and the path is from v_1 to v_k
- 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 $\{v_1, v_k\} \in E$.
- A vertex u is connected to v if there is a path from u to v.
- The connected component of u, $\mathrm{con}(u)$ is the set of all vertices connected to u.

Connectivity contd



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

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

Connectivity Problems

Fudnamental Algorithmic Problems

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

Connectivity Problems

Fudnamental Algorithmic Problems

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

Can be accomplished in $O(m + n)$ 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 $x \in S$ and $y \notin S$
        add $y$ to $S$
```

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 $x \in S$ and $y \notin S$
        add $y$ to $S$
```

Proposition

Explore(u) terminates with $S = con(u)$.

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 $x \in S$ and $y \notin S$
        add $y$ to $S$
```

Proposition

Explore(u) terminates with $S = con(u)$.

Running time: depends on implementation

- Breadth First Search (BFS): use queue data structure
- Depth First Search (DFS): use stack data structure
- Review CS 225 material!

Part III

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 ($O(m+n)$) time algorithms for

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

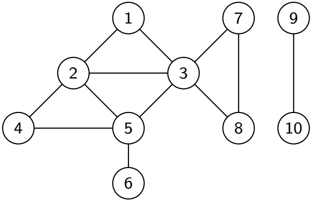
DFS in Undirected Graphs

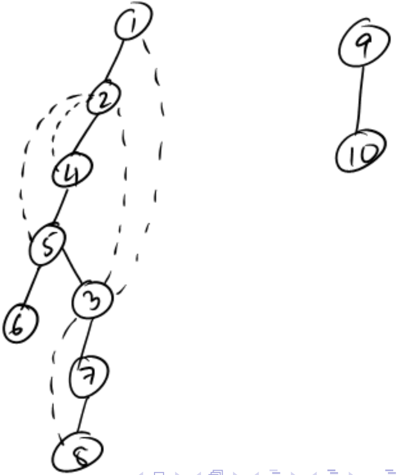
Recursive version.

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

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

Global array Mark for all recursive calls.

Example





DFS Tree/Forest

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to $\emptyset$
    While there is an unvisited node u do
        DFS(u)
    Output T

DFS(u)
    Mark u as visited
    for each edge (u,v) in Ajd(u) do
        if v is not marked
            add edge (u,v) to T
            DFS(v)
```

DFS Tree/Forest

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to $\emptyset$
    While there is an unvisited node u do
        DFS(u)
    Output T

DFS(u)
    Mark u as visited
    for each edge (u,v) in Ajd(u) do
        if v is not marked
            add edge (u,v) to T
            DFS(v)
```

Edges classified into two types: $(u, v) \in E$ is a

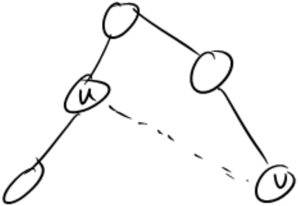
- tree edge: belongs to T
- non-tree edge: does not belong to T

Properties of DFS tree

Proposition

- *T is a forest and connected components of T are same as those of G.*
- *If (u, v) is a non-tree edge then, in T, either u is an ancestor of v or v is an ancestor of u.*

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

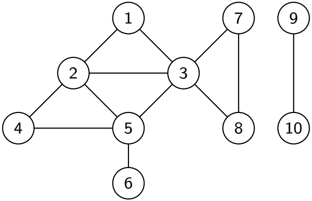


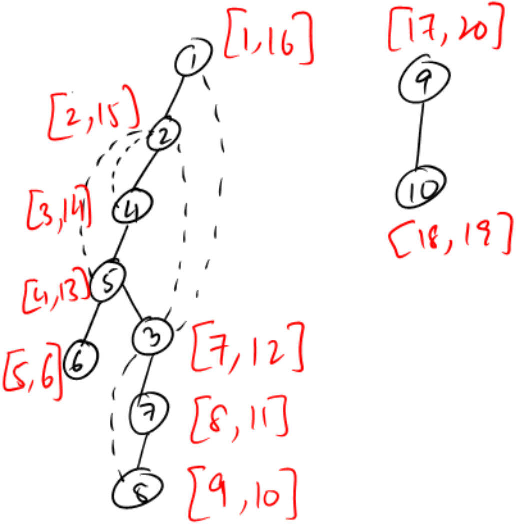
DFS with Visit Times

Keep track of when nodes are visited.

```
DFS(G)
    Mark all nodes u as unvisited
    T is set to $\emptyset$
    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


1
2
3
4
5
6
7
8
9
10


[1,16]
1
[17,20]
9
[2,15]
2
10
[18,19]
[3,14]
4
[4,13]
5
[5,6]
6
3
[7,12]
7
[8,11]
8
[9,10]

Pre and Post numbers

Node *u* is *active* in time interval [pre(u),post(u)]

Proposition

For any two nodes u and v, the two intervals [pre(u),post(u)] and [pre(v),post(v)] are disjoint or one is contained in the other.

[5, 20] u

[10, 25] v

Pre and Post numbers

Node u is *active* in time interval [pre(u),post(u)]

Proposition

For any two nodes u and v, the two intervals [pre(u),post(u)] and [pre(v),post(v)] are disjoint or one is contained in the other.

Proof.

Pre and Post numbers

Node u is *active* in time interval [pre(u),post(u)]

Proposition

For any two nodes u and v, the two intervals [pre(u),post(u)] and [pre(v),post(v)] are disjoint or one is contained in the other.

Proof.

- Suppose pre(u) $<$ pre(v). Implies v visited after u.

Pre and Post numbers

Node u is *active* in time interval [pre(u),post(u)]

Proposition

For any two nodes u and v, the two intervals [pre(u),post(u)] and [pre(v),post(v)] are disjoint or one is contained in the other.

Proof.

- Suppose pre(u) $<$ pre(v). Implies v visited after u.
- If DFS(v) invoked before DFS(u) finished, then post(u) $>$ post(v).

Pre and Post numbers

Node *u* is *active* in time interval [pre(u),post(u)]

Proposition

For any two nodes u and v, the two intervals [pre(u),post(u)] and [pre(v),post(v)] are disjoint or one is contained in the other.

Proof.

- Suppose pre(u) $<$ pre(v). Implies *v* visited after *u*.
- If DFS(v) invoked before DFS(u) finished, then post(u) $>$ post(v).
- If DFS(v) invoked after DFS(u) finished, then pre(v) $>$ post(u).

□

Pre and Post numbers

Node u is *active* in time interval [pre(u),post(u)]

Proposition

For any two nodes u and v, the two intervals [pre(u),post(u)] and [pre(v),post(v)] are disjoint or one is contained in the other.

Proof.

- Suppose pre(u) $<$ pre(v). Implies v visited after u.
- If DFS(v) invoked before DFS(u) finished, then post(u) $>$ post(v).
- If DFS(v) invoked after DFS(u) finished, then pre(v) $>$ post(u).

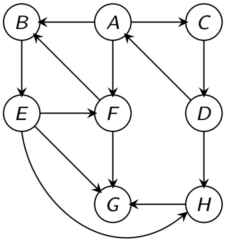
☐

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

Part IV

Directed Graphs and Decomposition

Directed Graphs



Definition

A directed graph $G = (V, E)$ consists of

- set of vertices/nodes V and
- a set of edges/arcs $E \subseteq V \times V$.

An edge is an *ordered* pair of vertices. (u, v) different from (v, u).

Examples of Directed Graphs

In many situations relationship between vertices is asymmetric:

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

Representation

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

Adjacency Matrix $n \times n$ asymmetric matrix A. $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ if $(u, v) \notin E$. $A[u, v]$ is not same as $A[v, u]$.

Adjacency Lists for each node u, $Out(u)$ (also referred to as $Adj(u)$) and $In(u)$ store out-going edges and in-coming edges from u.

Default representation is adjacency lists.

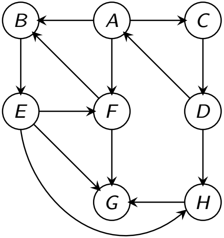
Directed Connectivity

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

- 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
- 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 $(v_k, v_1) \in E$.
- A vertex u can reach v if there is a path from u to v. Alternatively v can be reached from u
- Let $\mathrm{rch}(u)$ be the set of all vertices reachable from u.

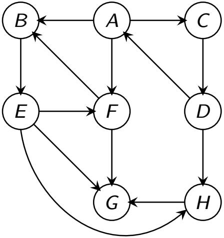
Connectivity contd

Asymmetricity: A can reach B but B cannot reach A



Connectivity contd

Asymmetricity: A can reach B but B cannot reach A



Questions:

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

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 $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

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 $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

Define relation C where uCv if u is (strongly) connected to 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 $v \in \mathrm{rch}(u)$ and $u \in \mathrm{rch}(v)$.

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 G, u is strongly connected to v if u can reach v *and* v can reach u. In other words $v \in \text{rch}(u)$ and $u \in \text{rch}(v)$.

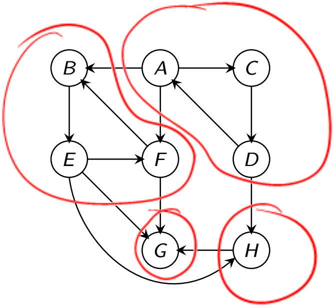
Define relation C where uCv if u is (strongly) connected to v.

Proposition

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

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

Strong Connected Components: Example



Directed Graph Connectivity Problems

- Given G and nodes u and v, can u reach v?
- Given G and u, compute $\text{rch}(u)$.
- Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.
- Find the strong component containing node u, that is $\text{SC}(u)$.
- Is G strongly connected (a single strong component)?
- Compute *all* strong components of G.

Directed Graph Connectivity Problems

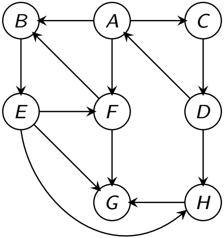
- Given G and nodes u and v, can u reach v?
- Given G and u, compute $\text{rch}(u)$.
- Given G and u, compute all v that can reach u, that is all v such that $u \in \text{rch}(v)$.
- Find the strong component containing node u, that is $\text{SC}(u)$.
- Is G strongly connected (a single strong component)?
- Compute *all* strong components of G.

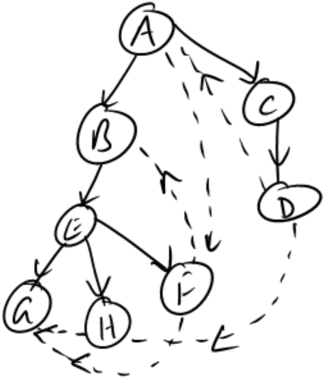
First four problems can be solve in $O(n + m)$ time by adapting BFS/DFS to directed graphs. The last one requires a clever DFS based algorithm.

DFS in Directed Graphs

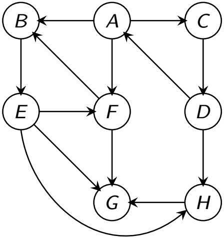
```
DFS(G)
    Mark all nodes u as unvisited
    T is set to $\emptyset$
    time = 0
    While there is an unvisited node $u$ do
        DFS(u)

    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
```

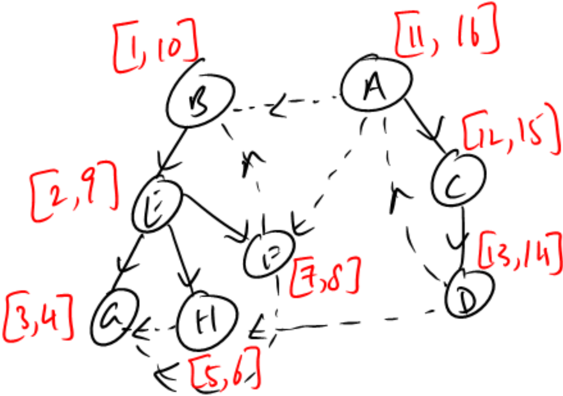
Example





Example





DFS Properties

Generalizing ideas from undirected graphs:

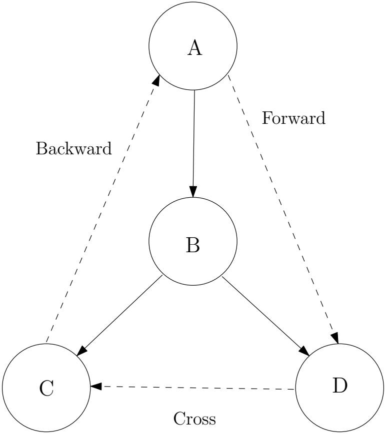
- $DFS(u)$ outputs a directed out-tree T rooted at u
- A vertex v is in T if and only if $v \in \text{rch}(u)$
- For any two vertices x, y the intervals $[pre(x), post(x)]$ and $[pre(y), post(y)]$ are either disjoint are one is contained in the other.
- The running time of $DFS(u)$ is $O(k)$ where $k = \sum_{v \in \text{rch}(u)} |Adj(v)|$ plus the time to initialize the Mark array.
- $DFS(G)$ takes $O(m + n)$ time. Edges in T form a disjoint collection of of out-trees. Output of $DFS(G)$ depends on the order in which vertices are considered.

DFS Tree

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

- Tree edges that belong to T
- A forward edge is a non-tree edges (x, y) such that $pre(x) < pre(y) < post(y) < post(x)$.
- A backward edge is a non-tree edge (x, y) such that $pre(y) < pre(x) < post(x) < post(y)$.
- A cross edge is a non-tree edges (x, y) such that the intervals $[pre(x), post(x)]$ and $[pre(y), post(y)]$ are disjoint.

Types of Edges



Directed Graph Connectivity Problems

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

Algorithms via DFS - I

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

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

Algorithms via DFS - II

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

Algorithms via DFS - II

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

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

Algorithms via DFS - II

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

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

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

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

Algorithms via DFS - III

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

Algorithms via DFS - III

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

- Find the strong component containing node u. That is, compute $SC(G, u)$.

Algorithms via DFS - III

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

- Find the strong component containing node u. That is, compute $SC(G, u)$.

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

Algorithms via DFS - III

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

- Find the strong component containing node u. That is, compute $SC(G, u)$.

$SC(G, u) = \text{rch}(G, u) \cap \text{rch}(G^{rev}, u)$

Hence, $SC(G, u)$ can be computed with two DFSes, one in G and the other in G^{rev}. Total $O(n + m)$ time.

Algorithms via DFS - IV

- Is G strongly connected?

Algorithms via DFS - IV

- Is G strongly connected?

Pick arbitrary vertex u. Check if $SC(G, u) = V$.

Algorithms via DFS - V

- Find *all* strongly connected components of G.

Algorithms via DFS - V

- Find *all* strongly connected components of G.

```
For each vertex $u \in V$ do
        find $SC(G, u)$
```

Running time: $O(n(n+m))$.

Algorithms via DFS - V

- Find *all* strongly connected components of G.

```
For each vertex $u \in V$ do
        find $SC(G, u)$
```

Running time: $O(n(n+m))$.

Can we do it in $O(n+m)$ time?