OLD CS 473: Fundamental Algorithms

Sariel Har-Peled
sariel@illinois.edu
3306 SC

University of Illinois, Urbana-Champaign

Spring 2015

OLD CS 473: Fundamental Algorithms, Spring 2015

Administrivia, Introduction, Graph basics and DFS

Lecture 1
January 20, 2015

The word "algorithm" comes from...

Muhammad ibn Musa al-Khwarizmi
780-850 AD
The word "algebra" is taken from the title of one of his books.

Part I

Administrivia

Instructional Staff

1. Instructor:
   - Sariel Har-Peled (`sariel`)
2. Teaching Assistants:
   1. Quanrud Kent.
   2. Agrawal Shashank.
3. Office hours:
   1. Instructor: See course webpage
   2. TAs: Friday, 10:00-13:00 in MEB 256.
      MEB = Mechanical engineering building.
4. Email: See course webpage

Online resources

1. Webpage:
   `http://courses.engr.illinois.edu/cs473/sp2015/`
   General information, homeworks, etc.
2. Moodle:
   `https://learn.illinois.edu/course/view.php?id=10239`
   Quizzes, solutions to homeworks.
3. Online questions/announcements: Piazza
   `http://piazza.com/illinois/spring2015/cs473/home`
   Online discussions, etc.

Textbooks

1. Prerequisites: CS 173 (discrete math), CS 225 (data structures) and CS 373 (theory of computation)
2. Recommended books:
   1. Algorithms by Dasgupta, Papadimitriou & Vazirani.
      Available online for free!
   2. Algorithm Design by Kleinberg & Tardos
3. Lecture notes: Available on the web-page after every class.
4. Additional References
   1. Previous class notes of Jeff Erickson, and the instructor.
   2. Introduction to Algorithms: Cormen, Leiserson, Rivest, Stein.
   3. Computers and Intractability: Garey and Johnson.

Prerequisites

1. Asymptotic notation: $O()$, $\Omega()$, $o()$.
2. Discrete Structures: sets, functions, relations, equivalence classes, partial orders, trees, graphs
3. Logic: predicate logic, boolean algebra
4. Proofs: **by induction**, by contradiction
5. Basic sums and recurrences: sum of a geometric series, unrolling of recurrences, basic calculus
6. Data Structures: arrays, multi-dimensional arrays, linked lists, trees, balanced search trees, heaps
7. Abstract Data Types: lists, stacks, queues, dictionaries, priority queues
8. Algorithms: sorting (merge, quick, insertion), pre/post/in order traversal of trees, depth/breadth first search of trees (maybe graphs)
9. Basic analysis of algorithms: loops and nested loops, deriving recurrences from a recursive program
10. Concepts from Theory of Computation: languages, automata, Turing machine, undecidability, non-determinism
11. Programming: in some general purpose language
12. Elementary Discrete Probability: event, random variable, independence
13. Mathematical maturity

Grading Policy: Overview

1. Attendance/clickers: 5%
2. Quizzes: 5%
3. Homeworks: 20%
4. Midterms: 40% (**2** $\times$ **20%**)
5. Final: 30% (covers the full course content)

Homeworks

1. One quiz every week: Due by midnight on Sunday.
2. One homework every week: Assigned on Tuesday and due the following Monday at noon.
3. Submit in homework box in the basement.
4. Homeworks can be worked on in groups of up to 3 and each group submits *one* written solution (except Homework 0).
    1. Short quiz-style questions are to be answered individually on *Moodle*.
5. Groups can be changed a *few* times only
6. Unlike previous years no *oral* homework this semester due to large enrollment.

More on Homeworks

1. No extensions or late homeworks accepted.
2. To compensate, the homework with the least score will be dropped in calculating the homework average.
3. Important: Read homework faq/instructions on website.

Discussion Sessions

1. 50min problem solving session led by TAs
2. Four sections all in SC 1214.
    1. Thursday
       4–4:50pm,
       5–5:50pm.
3. Attendance is required for both discussion sections/lectures.

Advice

1. Attend lectures, please ask plenty of questions.
2. Clickers...
3. Attend discussion sessions.
4. Don't skip homework and don't copy homework solutions.
5. Study regularly and keep up with the course.
6. Ask for help promptly. Make use of office hours.

Homeworks

1. HW 0 is posted on the class website. Quiz 0 available
2. Quiz 0 due by Sunday January 25 midnight
   HW 0 due on Monday, January 26 at noon.
3. HW 0 to be submitted individually.

Part II

Course Goals and Overview

Topics

1. Some fundamental algorithms
2. Broadly applicable techniques in algorithm design
   1. Understanding problem structure
   2. Brute force enumeration and backtrack search
   3. Reductions
   4. Recursion
      1. Divide and Conquer
      2. Dynamic Programming
   5. Greedy methods
   6. Network Flows and Linear/Integer Programming (optional)
3. Analysis techniques
   1. Correctness of algorithms via induction and other methods
   2. Recurrences
   3. Amortization and elementary potential functions
4. Polynomial-time Reductions, NP-Completeness, Heuristics

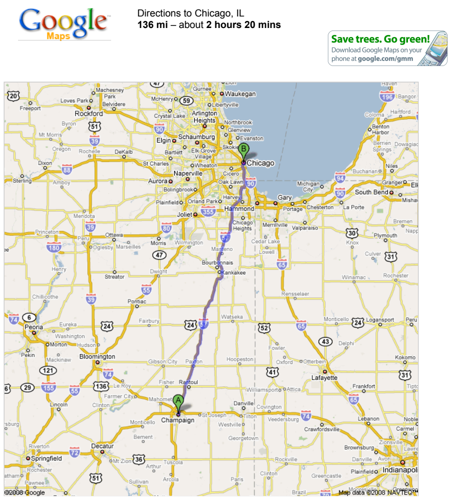
Goals

1.
2. Learn/remember some basic tricks, algorithms, problems, ideas
3. Understand/appreciate limits of computation (intractability)
4. Appreciate the importance of algorithms in computer science and beyond (engineering, mathematics, natural sciences, social sciences, ...)
5. Have fun!!!

Part III

Some Algorithmic Problems in the Real World

Shortest Paths



Shortest Paths - Paris to Berlin

Digital Information: Compression and Coding

Compression: reduce size for storage and transmission
Coding: add redundancy to protect against errors in storage and transmission

Efficient algorithms for compression/coding and decompressing/decoding part of most modern gadgets (computers, phones, music/video players ...)

Search and Indexing

String Matching and Link Analysis

1. Web search: Google, Yahoo!, Microsoft, Ask, ...
2. Text search: Text editors (Emacs, Word, Browsers, ...)
3. Regular expression search: grep, egrep, emacs, Perl, Awk, compilers

Public-Key Cryptography

Foundation of Electronic Commerce

RSA Crypto-system: generate key $n = pq$ where p, q are *primes*

Primality: Given a number N, check if N is a prime or composite.

Factoring: Given a composite number N, find a non-trivial factor

Programming: Parsing and Debugging

[godavari: /temp/test] chekuri % gcc main.c

Parsing: Is main.c a syntactically valid C program?

Debugging: Will main.c go into an infinite loop on some input?

Easier problem ??? Will main.c halt on the specific input 10?

Optimization

Find the cheapest of most profitable way to do things

1. Airline schedules - AA, Delta, ...
2. Vehicle routing - trucking and transportation (UPS, FedEx, Union Pacific, ...)
3. Network Design - AT&T, Sprint, Level3 ...

Linear and Integer programming problems

Part IV

Algorithm Design

Important Ingredients in Algorithm Design

1. What is the problem (really)?
   1. What is the input? How is it represented?
   2. What is the output?
2. What is the model of computation? What basic operations are allowed?
3. Algorithm design
4. Analysis of correctness, running time, space etc.
5. Algorithmic engineering: evaluating and understanding of algorithm's performance in practice, performance tweaks, comparison with other algorithms etc. (Not covered in this course)

Primality testing

Problem

Given an integer $N > 0$, is N a prime?

SimpleAlgorithm:

```
for i = 2 to ⌊√N⌋ do
    if i divides N then
        return ``COMPOSITE''
return ``PRIME''
```

Correctness? If N is composite, at least one factor in $\{2, \ldots, \sqrt{N}\}$
Running time? $O(\sqrt{N})$ divisions? Sub-linear in input size! Wrong!

Primality testing

...Polynomial means... in input size

How many bits to represent N in binary? $\lceil \log N \rceil$ bits.
Simple Algorithm takes $\sqrt{N} = 2^{(\log N)/2}$ time.
Exponential in the input size $n = \log N$.

1. Modern cryptography: binary numbers with 128, 256, 512 bits.
2. Simple Algorithm will take 2^{64}, 2^{128}, 2^{256} steps!
3. Fastest computer today about 3 petaFlops/sec: 3×2^{50} floating point ops/sec.

Lesson:
Pay attention to representation size in analyzing efficiency of algorithms. Especially in *number* problems.

Efficient algorithms

So, is there an *efficient/good/effective* algorithm for primality?

Question:
What does efficiency mean?

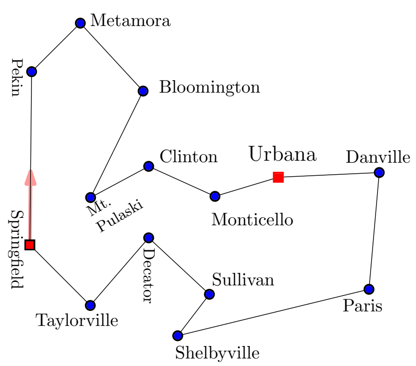
In this class *efficiency* is broadly equated to *polynomial time*.
$O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots$ where n is size of the input.

Why? Is n^{100} really efficient/practical? Etc.

Short answer: polynomial time is a robust, mathematically sound way to define efficiency. Has been useful for several decades.

TSP problem

Lincoln's tour



1. Circuit court - ride through counties staying a few days in each town.
2. Lincoln was a lawyer traveling with the Eighth Judicial Circuit.
3. Picture: travel during 1850.
    1. Very close to optimal tour.
    2. Might have been optimal at the time..

Solving TSP by a Computer

Is it hard?

1. n = number of cities.
2. n^2: size of input.
3. Number of possible solutions is
$$n * (n-1) * (n-2) * \ldots * 2 * 1 = n!.$$
4. $n!$ grows very quickly as n grows.
$n = 10$: $n! \approx 3628800$
$n = 50$: $n! \approx 3 * 10^{64}$
$n = 100$: $n! \approx 9 * 10^{157}$

Solving TSP by a Computer

Fastest computer...

1. Fastest super computer can do (roughly)

$$2.5 * 10^{15}$$

operations a second.

2. Assume: computer checks $2.5 * 10^{15}$ solutions every second, then...
    1. $n = 20 \implies$ 2 hours.
    2. $n = 25 \implies$ 200 years.
    3. $n = 37 \implies 2 * 10^{20}$ years!!!

What is a good algorithm?

Running time...

Input size	n^2 ops	n^3 ops	n^4 ops	$n!$ ops
5	0 secs	0 secs	0 secs	0 secs
20	0 secs	0 secs	0 secs	16 mins
30	0 secs	0 secs	0 secs	$3 \cdot 10^9$ years
100	0 secs	0 secs	0 secs	never
8000	0 secs	0 secs	1 secs	never
16000	0 secs	0 secs	26 secs	never
32000	0 secs	0 secs	6 mins	never
64000	0 secs	0 secs	111 mins	never
200,000	0 secs	3 secs	7 days	never
2,000,000	0 secs	53 mins	202.943 years	never
10^8	4 secs	12.6839 years	10^9 years	never
10^9	6 mins	12683.9 years	10^{13} years	never

What is a good algorithm?

Running time...

Primes is in P!

Theorem (Agrawal-Kayal-Saxena'02)

There is a polynomial time algorithm for primality.

First polynomial time algorithm for testing primality. Running time is $O(\log^{12} N)$ further improved to about $O(\log^6 N)$ by others. In terms of input size $n = \log N$, time is $O(n^6)$.

Breakthrough announced in August 2002. Three days later announced in New York Times. Only 9 pages!

Neeraj Kayal and Nitin Saxena were undergraduates at IIT-Kanpur!

What about before 2002?

Primality testing a key part of cryptography. What was the algorithm being used before 2002?
Miller-Rabin *randomized* algorithm:

1. runs in polynomial time: $O(\log^3 N)$ time
2. if N is prime correctly says "yes".
3. if N is composite it says "yes" with probability at most $1/2^{100}$ (can be reduced further at the expense of more running time).

Based on Fermat's little theorem and some basic number theory.

Factoring

1. Modern public-key cryptography based on RSA (Rivest-Shamir-Adelman) system.
2. Relies on the difficulty of factoring a composite number into its prime factors.
3. There is a polynomial time algorithm that decides whether a given number N is prime or not (hence composite or not) but no known polynomial time algorithm to factor a given number.

Lesson

Intractability can be useful!

Digression: decision, search and optimization

Three variants of problems.

1. Decision problem: answer is yes or no.
   Example: Given integer N, is it a composite number?
2. Search problem: answer is a feasible solution if it exists.
   Example: Given integer N, if N is composite output *a* non-trivial factor p of N.
3. Optimization problem: answer is the *best* feasible solution (if one exists).
   Example: Given integer N, if N is composite output the *smallest* non-trivial factor p of N.

For a given underlying problem:

$$\text{Optimization} \geq \text{Search} \geq \text{Decision}$$

Quantum Computing

Theorem (Shor'1994)

There is a polynomial time algorithm for factoring on a quantum *computer.*

RSA and current commercial cryptographic systems can be broken if a quantum computer can be built!

Lesson

Pay attention to the model of computation.

Problems and Algorithms

Many many different problems.

1. Adding two numbers: efficient and simple algorithm
2. Sorting: efficient and not too difficult to design algorithm
3. Primality testing: simple and basic problem, took a long time to find efficient algorithm
4. Factoring: no efficient algorithm known.
5. Halting problem: important problem in practice, undecidable!

Multiplying Numbers

Problem Given two n-digit numbers x and y, compute their product.

Grade School Multiplication

Compute "partial product" by multiplying each digit of y with x and adding the partial products.

$$\begin{array}{r} 3141 \\ \times 2718 \\ \hline 25128 \\ 3141 \\ 21987 \\ 6282 \\ \hline 8537238 \end{array}$$

Time analysis of grade school multiplication

1. Each partial product: $\Theta(n)$ time
2. Number of partial products: $\leq n$
3. Adding partial products: n additions each $\Theta(n)$ (Why?)
4. Total time: $\Theta(n^2)$
5. Is there a faster way?

Fast Multiplication

Best known algorithm: $O(n \log n \cdot 2^{O(\log^* n)})$ time [Furer 2008]

Previous best time: $O(n \log n \log \log n)$ [Schonhage-Strassen 1971]

Conjecture: there exists and $O(n \log n)$ time algorithm

We don't fully understand multiplication!
Computation and algorithm design is non-trivial!

Course Approach

Algorithm design requires a mix of skill, experience, mathematical background/maturity and ingenuity.

Approach in this class and many others:

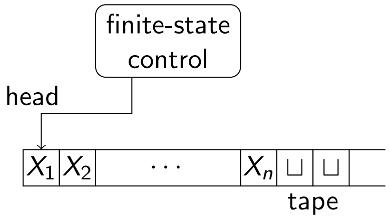
1. Improve skills by showing various tools in the abstract and with concrete examples
2. Improve experience by giving many problems to solve
3. Motivate and inspire
4. Creativity: you are on your own!

What model of computation do we use?

Turing Machine?

Turing Machines: Recap

1. Infinite tape
2. Finite state control
3. Input at beginning of tape
4. Special tape letter "blank" ⊔
5. Head can move only one cell to left or right



Turing Machines

1. Basic unit of data is a bit (or a single character from a finite alphabet)
2. Algorithm is the finite control
3. Time is number of steps/head moves

Pros and Cons:

1. theoretically sound, robust and simple model that underpins computational complexity.
2. polynomial time equivalent to any reasonable "real" computer: Church-Turing thesis
3. too low-level and cumbersome, does not model actual computers for many realistic settings

“Real” Computers vs Turing Machines

How do “real” computers differ from TMs?

1. random access to memory
2. pointers
3. arithmetic operations (addition, subtraction, multiplication, division) in constant time

How do they do it?

1. basic data type is a word: currently 64 bits
2. arithmetic on words are basic instructions of computer
3. memory requirements assumed to be $\leq 2^{64}$ which allows for pointers and indirect addressing as well as random access

Unit-Cost RAM Model

Informal description:

1. Basic data type is an integer/floating point number
2. Numbers in input fit in a word
3. Arithmetic/comparison operations on words take constant time
4. Arrays allow random access (constant time to access $A[i]$)
5. Pointer based data structures via storing addresses in a word

Example

Sorting: input is an array of n numbers

1. input size is n (ignore the bits in each number),
2. comparing two numbers takes $O(1)$ time,
3. random access to array elements,
4. addition of indices takes constant time,
5. basic arithmetic operations take constant time,
6. reading/writing one word from/to memory takes constant time.

We will usually not allow (or be careful about allowing):

1. bitwise operations (and, or, xor, shift, etc).
2. floor function.
3. limit word size (usually assume unbounded word size).

Caveats of RAM Model

Unit-Cost RAM model is applicable in wide variety of settings in practice. However it is not a proper model in several important situations so one has to be careful.

1. For some problems such as basic arithmetic computation, unit-cost model makes no sense. Examples: multiplication of two n-digit numbers, primality etc.
2. Input data is very large and does not satisfy the assumptions that individual numbers fit into a word or that total memory is bounded by 2^k where k is word length.
3. Assumptions valid only for certain type of algorithms that do not create large numbers from initial data. For example, exponentiation creates very big numbers from initial numbers.

Models used in class

In this course:

1. Assume unit-cost RAM by default.
2. We will explicitly point out where unit-cost RAM is not applicable for the problem at hand.

Part V

Graph Basics

Why Graphs?

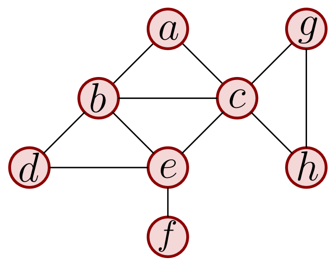
1. Graphs help model networks which are ubiquitous: transportation networks (rail, roads, airways), social networks (interpersonal relationships), information networks (web page links) etc etc.
2. Fundamental objects in Computer Science, Optimization, Combinatorics
3. Many important and useful optimization 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 $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\}$.

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.

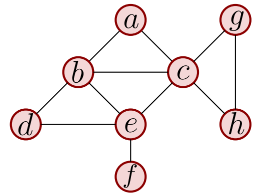
1. u and v are the end points of an edge $\{u, v\}$
2. Multi-graphs allow
   1. *loops* which are edges with the same node appearing as both end points
   2. *multi-edges*: different edges between same pairs of nodes
3. In this class we will assume that a graph is a simple graph unless explicitly stated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



	a	b	c	d	e	f	g	h
a	0	1	1	0	0	0	0	0
b	1	0	1	1	1	0	0	0
c	1	1	0	0	1	0	1	1
d	0	1	0	0	1	0	0	0
e	0	1	1	1	0	1	0	0
f	0	0	0	0	1	0	0	0
g	0	0	1	0	0	0	0	1
h	0	0	1	0	0	0	1	0

Graph Representation II

Adjacency Lists

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

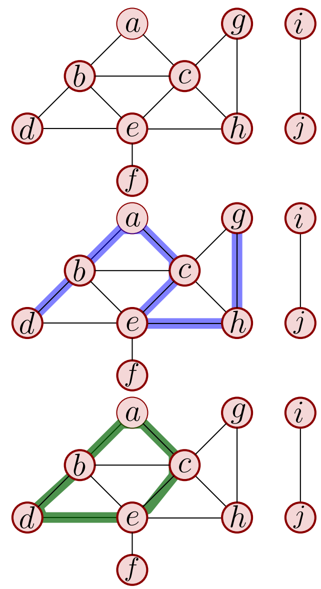
1. For each $u \in V$, $\text{Adj}(u) = \{v \mid \{u, v\} \in E\}$, that is neighbors of u. Sometimes $\text{Adj}(u)$ is the list of edges incident to u.
2. Advantage: space is $O(m + n)$
3. Disadvantage: cannot "easily" determine in $O(1)$ time whether $\{i, j\} \in E$
   1. By sorting each list, one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

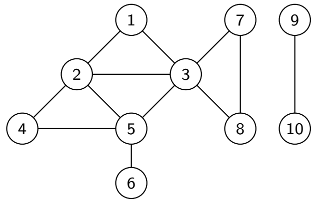
1. **path**: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$.
   For $i = 1, \ldots, k - 1$: $v_i v_{i+1} \in E$
   length of path $= k - 1$.
   The path is from v_1 to v_k
2. **cycle**: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$
   $\forall i \quad v_i v_{i+1} \in E$ and $\{v_1, v_k\} \in E$.
3. A vertex u is connected to v if there is a path from u to v.
4. The **connected component** of u, $\text{con}(u)$, 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

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.

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 ∈ S and y ∉ S do
        add y to S
```

Proposition

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

Running time: depends on implementation

1. Breadth First Search (**BFS**): use queue data structure
2. Depth First Search (**DFS**): use stack data structure
3. Review CS 225 material!

Part VI

DFS

Depth First Search

DFS: versatile graph exploration strategy. Hopcroft and Tarjan demonstrated the power of **DFS** to understand graph structure. **DFS** can be used to obtain linear time ($O(m + n)$) time 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

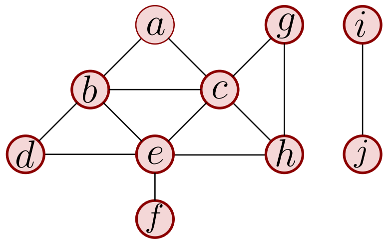
Recursive version.

```
DFS(G)
        Mark all nodes 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 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 as unvisited
    T is set to ∅
    while ∃ unvisited node u do
        DFS(u)
    Output T
DFS(u)
    Mark u as visited
    for uv in Ajd(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. T is a forest
2. connected components of T are same as those of G.
3. If $uv \in E$ is a non-tree edge then, in T, either:
    1. u is an ancestor of v, or
    2. v is an ancestor of u.

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

DFS with Visit Times

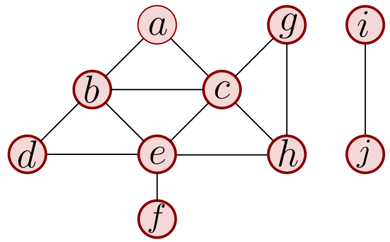
Keep track of when nodes are visited.

```
DFS(G)
    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
```

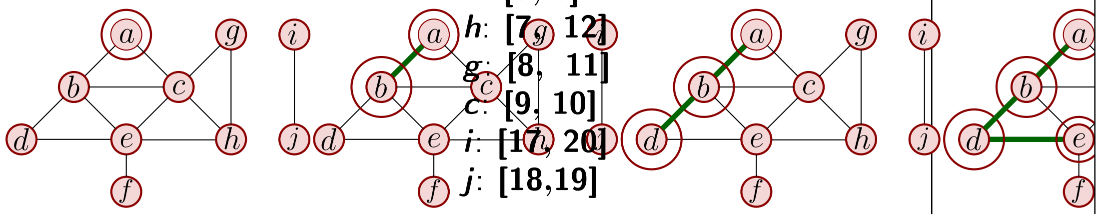

Scratch space

Example: DFS with visit times



Example

a: [1, 16]
b: [2, 15]
d: [3, 14]
e: [4, 13]
f: [5, 6]
h: [7, 12]
g: [8, 11]
c: [9, 10]
i: [17, 20]
j: [18,19]



pre and post numbers

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

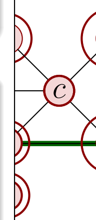
Proposition

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

Proof.

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

□



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

Part VII

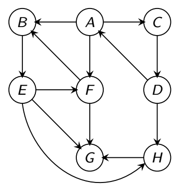
Directed Graphs and Decomposition

Directed Graphs

Definition

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

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



1. An edge is an *ordered* pair of vertices.
2. Directed edge written as (u, v) or $(u \to v)$.
3. $(u \to v)$ is different from $(v \to u)$.

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. 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.
3. Dependency graphs in variety of applications: link from x to y if y depends on x. Make files for compiling programs.
4. 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:

1. **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 $A[v, u]$.
2. **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.

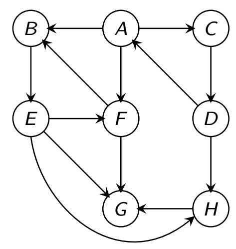
Directed Connectivity

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

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
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 $(v_k, v_1) \in E$.
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 $\text{rch}(u)$ be the set of all vertices reachable from u.

Connectivity contd

Asymmetricity: A can reach B but B cannot reach A



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

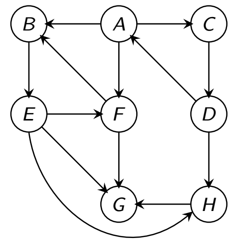
1. Define relation C where uCv if u is (strongly) connected to v.

Proposition

C is an equivalence relation $\implies$ reflexive, symmetric and transitive.

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

Strongly Connected Components: Example



Problems on Directed Graph Connectivity

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

DFS Properties

Generalizing ideas from undirected graphs:

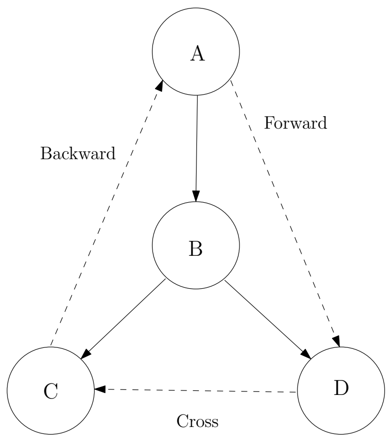
1. $DFS(u)$ outputs a directed out-tree T rooted at u
2. A vertex v is in T if and only if $v \in \text{rch}(u)$
3. For any two vertices x, y the intervals $[\text{pre}(x), \text{post}(x)]$ and $[\text{pre}(y), \text{post}(y)]$ are either disjoint are one is contained in the other.
4. The running time of $DFS(u)$ is $O(k)$ where $k = \sum_{v \in \text{rch}(u)} |Adj(v)|$ plus the time to initialize the Mark array.
5. **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:

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

Types of Edges



Directed Graph Connectivity Problems

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

Algorithms via DFS- I

1. Given G and nodes u and v, can u reach v?
2. 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

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

Definition (Reverse graph.)

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

1. **Correctness:** exercise
2. **Running time:** $O(n + m)$ to obtain G^{rev} from G and $O(n + m)$ time to compute $\text{rch}(u)$ 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

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

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

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

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

Algorithms via DFS- IV

1. Is G strongly connected?

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

Algorithms via DFS- V

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

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

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

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

Reading and Homework 0

Chapters 1 from Dasgupta etal book, Chapters 1-3 from Kleinberg-Tardos book.

Proving algorithms correct - Jeff Erickson's notes (see link on website)