CS 473: Fundamental Algorithms

Chandra Chekuri
chekuri@illinois.edu
3228 SC

University of Illinois, Urbana-Champaign

Spring 2014

Administrivia, Introduction, Graph basics and DFS

Lecture 1

January 21, 2014

Part I

Administrivia

Instructional Staff

1. Instructor: Chandra Chekuri
2. Teaching Assistants:
    1. Shalmoli Gupta
    2. Konstantinos Koiliaris
    3. Kent Quanrud
    4. Yipu Wang
3. Office hours: See course webpage
4. Email: See course webpage

Online resources

1. Webpage: General information, homeworks, course policies
   `courses.engr.illinois.edu/cs473/sp2014/`
2. Moodle: Quizzes, solutions to homeworks, grades
   `https://learn.illinois.edu/course/view.php?id=1647`
3. Piazza: Announcements, online questions and discussion
   `https://piazza.com/#spring2014/cs473`

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. Lecture notes of Jeff Erickson, Sariel HarPeled and the instructor from previous editions of the course.
   2. Introduction to Algorithms: Cormen, Leiserson, Rivest, Stein.
   3. Computers and Intractability: Garey and Johnson.

Prerequisites

1. Asymptotic notation: $\mathbf{O()}, \mathbf{\Omega()}, \mathbf{o()}$.
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. Quizzes: 6%
2. Homeworks: 22%
3. Midterms: 42% ($\mathbf{2 \times 21\%}$)
4. Finals: 30% (covers the full course content)

Homeworks

1. One quiz every week. Due by midnight on Sunday. Short questions to be answered on *Moodle*.
2. One homework every week: Assigned on Tuesday and due the following Tuesday at noon.
3. Homeworks can be worked on in groups of up to 3 and each group submits *one* written solution (except Homework 0).

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 1109
   1. Tuesday
      5–5:50pm,
      6–6:50pm.
   2. Wednesday
      4–4:50pm,
      5–5:50pm.

Advice

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

Homeworks

1. HW 0 is posted on the class website. Quiz 0 available on Moodle
2. Quiz 0 due by Sunday Jan 26 at midnight
   HW 0 due on Tuesday January 28 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. Algorithmic thinking
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!!!

Goals

1. Algorithmic thinking
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

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

Public-Key Cryptography

Foundation of Electronic Commerce

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

Primality: Given a number $\mathbf{N}$, check if $\mathbf{N}$ is a prime or composite.

Factoring: Given a composite number $\mathbf{N}$, find a non-trivial factor

Public-Key Cryptography

Foundation of Electronic Commerce

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

Primality: Given a number $\mathbf{N}$, check if $\mathbf{N}$ is a prime or composite.

Factoring: Given a composite number $\mathbf{N}$, 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. Proving correctness
5. Analyzing running time, space and other resource usage
6. 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 $\mathbf{N} > \mathbf{0}$, is $\mathbf{N}$ a prime?

Is **435987243587234587249872435873451** prime?

SimpleAlgorithm:

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

Correctness? If $\mathbf{N}$ is composite, at least one factor in $\{\mathbf{2}, \ldots, \sqrt{\mathbf{N}}\}$

Running time? $\mathbf{O}(\sqrt{\mathbf{N}})$ divisions? Sub-linear in input size! Wrong!

Primality testing

Problem

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

Is $435987243587234587249872435873451$ 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

Problem

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

Is $435987243587234587249872435873451$ 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

Problem

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

Is $435987243587234587249872435873451$ 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

Problem

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

Is $435987243587234587249872435873451$ 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

Problem

Given an integer $\mathbf{N} > \mathbf{0}$, is $\mathbf{N}$ a prime?

Is $\mathbf{435987243587234587249872435873451}$ prime?

SimpleAlgorithm:

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

Correctness? If $\mathbf{N}$ is composite, at least one factor in $\{\mathbf{2}, \ldots, \sqrt{\mathbf{N}}\}$

Running time? $\mathbf{O}(\sqrt{\mathbf{N}})$ divisions? Sub-linear in input size! Wrong!

Primality testing

Problem

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

Is $435987243587234587249872435873451$ 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 $\mathbf{N}$ in binary? $\lceil \log \mathbf{N} \rceil$ bits.

Simple Algorithm takes $\sqrt{\mathbf{N}} = 2^{(\log \mathbf{N})/2}$ time.

Exponential in the input size $\mathbf{n} = \log \mathbf{N}$.

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 35 petaFlops/sec: 35×2^{50} floating point ops/sec.

Lesson:

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

Primality testing

...Polynomial means... in input size

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

1. Modern cryptography: binary numbers with 128, 256, 512 bits.
2. Simple Algorithm will take $\mathbf{2^{64}}$, $\mathbf{2^{128}}$, $\mathbf{2^{256}}$ steps!
3. Fastest computer today about 35 petaFlops/sec: $\mathbf{35 \times 2^{50}}$ 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.

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.

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*. $\mathbf{O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots}$ where $\mathbf{n}$ is size of the input.

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

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

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*. $\mathbf{O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots}$ where $\mathbf{n}$ is size of the input.

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

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

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*. $\mathbf{O(n), O(n \log n), O(n^2), O(n^3), O(n^{100}), \ldots}$ where $\mathbf{n}$ is size of the input.

Why? Is $\mathbf{n^{100}}$ 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. $\mathbf{n}$ = number of cities.
2. $\mathbf{n^2}$: size of input.
3. Number of possible solutions is

$$\mathbf{n * (n-1) * (n-2) * \ldots * 2 * 1 = n!.}$$

4. $\mathbf{n!}$ grows very quickly as $\mathbf{n}$ grows.
   $\mathbf{n = 10}$: $\mathbf{n! \approx 3628800}$
   $\mathbf{n = 50}$: $\mathbf{n! \approx 3 * 10^{64}}$
   $\mathbf{n = 100}$: $\mathbf{n! \approx 9 * 10^{157}}$

Solving TSP by a Computer

Fastest computer...

1. A good super computer can do (some what out dated)

$$\mathbf{2.5 * 10^{15}}$$

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

"No, Thursday's out. How about never—is never good for you?"

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 $\mathbf{O(\log^{12} N)}$ further improved to about $\mathbf{O(\log^{6} N)}$ by others. In terms of input size $\mathbf{n = \log N}$, time is $\mathbf{O(n^6)}$.

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: $\mathbf{O(\log^3 N)}$ time
2. if $\mathbf{N}$ is prime correctly says "yes".
3. if $\mathbf{N}$ is composite it says "yes" with probability at most $\mathbf{1/2^{100}}$ (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!

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:

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

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.

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

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

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

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

Fast Multiplication

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

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

Conjecture: there exists and $\mathbf{O(n \log n)}$ 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?

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" $\sqcup$
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

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

"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

"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

"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 $\mathbf{n}$-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 $\mathbf{2^k}$ where $\mathbf{k}$ 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 $\mathbf{G = (V, E)}$ is a $\mathbf{2}$-tuple:

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

Example

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

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 $(\mathbf{u}, \mathbf{v})$ 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 $G = (V, E)$ 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 $\mathbf{G = (V, E)}$:

1. A path is a sequence of *distinct* vertices $\mathbf{v_1, v_2, \ldots, v_k}$ such that $\mathbf{\{v_i, v_{i+1}\} \in E}$ for $\mathbf{1 \leq i \leq k - 1}$. The length of the path is $\mathbf{k - 1}$ (the number of edges in the path) and the path is from $\mathbf{v_1}$ to $\mathbf{v_k}$
2. A cycle is a sequence of *distinct* vertices $\mathbf{v_1, v_2, \ldots, v_k}$ such that $\mathbf{\{v_i, v_{i+1}\} \in E}$ for $\mathbf{1 \leq i \leq k - 1}$ and $\mathbf{\{v_1, v_k\} \in E}$. Caveat: Some times people use the term cycle to also allow vertices to be repeated; we will use the term tour.
3. A vertex $\mathbf{u}$ is connected to $\mathbf{v}$ if there is a path from $\mathbf{u}$ to $\mathbf{v}$.
4. The connected component of $\mathbf{u}$, $\mathrm{con}(\mathbf{u})$, is the set of all vertices connected to $\mathbf{u}$.

Connectivity contd

Define a relation **C** on $\mathbf{V} \times \mathbf{V}$ 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**.

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 $\mathbf{O(m+n)}$ 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
```

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!

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!

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 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 ($\mathbf{O(m+n)}$) 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 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)
```

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 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

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 Ajd(u) do
        if v is not marked
            add uv to T
            DFS(v)
```

Edges classified into two types: $\mathbf{uv} \in \mathbf{E}$ 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 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

pre and post numbers

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

Proposition

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

Proof.

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

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

pre and post numbers

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

Proposition

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

Proof.

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

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

pre and post numbers

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

Proposition

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

Proof.

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

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

pre and post numbers

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

Proposition

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

Proof.

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

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

pre and post numbers

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

Proposition

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

Proof.

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

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

pre and post numbers

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

Proposition

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

Proof.

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

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

Part VII

Directed Graphs and Decomposition

Directed Graphs

Definition

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

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

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

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 $\mathbf{(u, v) \notin E}$. $\mathbf{A[u, v]}$ 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.

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

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 $\mathbf{G}$.
$\text{SCC}(\mathbf{u})$: strongly connected component containing $\mathbf{u}$.

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

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 $\mathbf{G}$.
$\text{SCC}(\mathbf{u})$: strongly connected component containing $\mathbf{u}$.

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

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 $\mathbf{G}$.
$\text{SCC}(\mathbf{u})$: strongly connected component containing $\mathbf{u}$.

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 $\mathrm{rch}(\mathbf{u})$.
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 $\mathbf{G}$.

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.

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 $\mathbf{G}$.

First four problems can be solve 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**
2. A vertex **v** is in **T** if and only if $\mathbf{v} \in \mathrm{rch}(\mathbf{u})$
3. For any two vertices **x, y** the intervals $[\mathrm{pre}(\mathbf{x}), \mathrm{post}(\mathbf{x})]$ and $[\mathrm{pre}(\mathbf{y}), \mathrm{post}(\mathbf{y})]$ are either disjoint are one is contained in the other.
4. The running time of **DFS(u)** is $O(k)$ where $k = \sum_{v \in \mathrm{rch}(u)} |\mathrm{Adj}(v)|$ 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 $\mathrm{pre}(x) < \mathrm{pre}(y) < \mathrm{post}(y) < \mathrm{post}(x)$.
3. A **backward edge** is a non-tree edge (x, y) such that $\mathrm{pre}(y) < \mathrm{pre}(x) < \mathrm{post}(x) < \mathrm{post}(y)$.
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 $\mathrm{rch}(\mathbf{u})$.

Use $\mathbf{DFS(G, u)}$ to compute $\mathrm{rch}(\mathbf{u})$ 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 $\mathbf{u} \in \text{rch}(\mathbf{v})$.

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 $\mathbf{E}' = \{(\mathbf{y}, \mathbf{x}) \mid (\mathbf{x}, \mathbf{y}) \in \mathbf{E}\}$

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

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

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

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 $\mathbf{E}' = \{(\mathbf{y}, \mathbf{x}) \mid (\mathbf{x}, \mathbf{y}) \in \mathbf{E}\}$

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

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

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

Definition (Reverse graph.)

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

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

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

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 $\mathrm{SCC}(\mathbf{G}, \mathbf{u})$.

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

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

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 $\mathrm{SCC}(\mathbf{G}, \mathbf{u})$.

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

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

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 $\mathrm{SCC}(\mathbf{G}, \mathbf{u})$.

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

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

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 $\mathrm{SCC}(\mathbf{G}, \mathbf{u})$.

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

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

Algorithms via DFS- IV

1. Is **G** strongly connected?

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

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**.

```
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?

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

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

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?

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?

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)

Notes

Notes

Notes

Notes

DFS in Directed Graphs, Strong Connected Components, and DAGs

Lecture 2

January 23, 2014

Strong Connected Components (SCCs)

Algorithmic Problem

Find all SCCs of a given directed graph.

Previous lecture:
Saw an $\mathbf{O(n \cdot (n + m))}$ time algorithm.
This lecture: $\mathbf{O(n + m)}$ time algorithm.

Graph of SCCs

Graph G

Graph of SCCs G^{SCC}

Meta-graph of SCCs

Let $S_1, S_2, \ldots S_k$ be the strong connected components (i.e., SCCs) of G. The graph of SCCs is G^{SCC}

1. Vertices are $S_1, S_2, \ldots S_k$
2. There is an edge (S_i, S_j) if there is some $u \in S_i$ and $v \in S_j$ such that (u, v) is an edge in G.

Reversal and SCCs

Proposition

For any graph G, the graph of SCCs of $\mathbf{G^{rev}}$ is the same as the reversal of G^{SCC}.

Proof.

Exercise. □

SCCs and DAGs

Proposition

For any graph G, the graph G^{SCC} has no directed cycle.

Proof.

If G^{SCC} has a cycle $\mathbf{S_1}, \mathbf{S_2}, \ldots, \mathbf{S_k}$ then $\mathbf{S_1} \cup \mathbf{S_2} \cup \cdots \cup \mathbf{S_k}$ should be in the same SCC in G. Formal details: exercise. $\square$

Part I

Directed Acyclic Graphs

Directed Acyclic Graphs

Definition

A directed graph G is a **directed acyclic graph** (DAG) if there is no directed cycle in G.

Sources and Sinks

Definition

1. A vertex **u** is a **source** if it has no in-coming edges.
2. A vertex **u** is a **sink** if it has no out-going edges.

Simple DAG Properties

1. Every DAG G has at least one source and at least one sink.
2. If G is a DAG if and only if G^{rev} is a DAG.
3. G is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

Simple DAG Properties

1. Every DAG G has at least one source and at least one sink.
2. If G is a DAG if and only if G^{rev} is a DAG.
3. G is a DAG if and only each node is in its own strong connected component.

Formal proofs: exercise.

Topological Ordering/Sorting

Graph G

Topological Ordering of G

Definition

A **topological ordering**/**topological sorting** of $\mathbf{G = (V, E)}$ is an ordering $\prec$ on $\mathbf{V}$ such that if $\mathbf{(u, v)} \in \mathbf{E}$ then $\mathbf{u} \prec \mathbf{v}$.

Informal equivalent definition:

One can order the vertices of the graph along a line (say the **x**-axis) such that all edges are from left to right.

DAGs and Topological Sort

Lemma

A directed graph G can be topologically ordered iff it is a DAG.

Proof.

$\Longrightarrow$: Suppose G is not a DAG and has a topological ordering $\prec$. G has a cycle $\mathbf{C} = \mathbf{u_1, u_2, \ldots, u_k, u_1}$.
Then $\mathbf{u_1 \prec u_2 \prec \ldots \prec u_k \prec u_1}$!
That is... $\mathbf{u_1 \prec u_1}$.
A contradiction (to $\prec$ being an order).
Not possible to topologically order the vertices. $\square$

DAGs and Topological Sort

Lemma

A directed graph G can be topologically ordered iff it is a DAG.

Continued.

$\Leftarrow$: Consider the following algorithm:

1. Pick a source **u**, output it.
2. Remove **u** and all edges out of **u**.
3. Repeat until graph is empty.
4. Exercise: prove this gives an ordering.

□

Exercise: show above algorithm can be implemented in $\mathbf{O(m+n)}$ time.

Topological Sort: An Example

Output: 1 2 3 4

Topological Sort: An Example

Output: 1 2 3 4

Topological Sort: An Example

Output: 1 2 3 4

Topological Sort: An Example

Output: 1 2 3 4

Topological Sort: An Example

Output: 1 2 3 4

Topological Sort: Another Example

DAGs and Topological Sort

Note: A DAG G may have many different topological sorts.

Question: What is a DAG with the most number of distinct topological sorts for a given number **n** of vertices?

Question: What is a DAG with the least number of distinct topological sorts for a given number **n** of vertices?

Using DFS...

... to check for Acylicity and compute Topological Ordering

Question

Given G, is it a DAG? If it is, generate a topological sort.

DFS based algorithm:

1. Compute **DFS(G)**
2. If there is a back edge then G is not a DAG.
3. Otherwise output nodes in decreasing post-visit order.

Correctness relies on the following:

Proposition

G is a DAG iff there is no back-edge in **DFS(G)**.

Proposition

If G is a DAG and $\mathbf{post(v)} > \mathbf{post(u)}$, *then* $\mathbf{(u, v)}$ *is not in G.*

Using DFS...

... to check for Acylicity and compute Topological Ordering

Question

Given G, is it a DAG? If it is, generate a topological sort.

DFS based algorithm:

1. Compute **DFS(G)**
2. If there is a back edge then G is not a DAG.
3. Otherwise output nodes in decreasing post-visit order.

Correctness relies on the following:

Proposition

G is a DAG iff there is no back-edge in **DFS(G)**.

Proposition

If G is a DAG and $\mathbf{post(v)} > \mathbf{post(u)}$, *then* $\mathbf{(u, v)}$ *is not in G.*

Proof

Proposition

If G is a DAG *and* $\mathbf{post(v)} > \mathbf{post(u)}$, *then* $\mathbf{(u, v)}$ *is not in* G.

Proof.

Assume $\mathbf{post(v)} > \mathbf{post(u)}$ *and* $\mathbf{(u, v)}$ is an edge in $\mathbf{G}$. We derive a contradiction. One of two cases holds from DFS property.

- Case 1: $\mathbf{[pre(u), post(u)]}$ is contained in $\mathbf{[pre(v), post(v)]}$. Implies that $\mathbf{u}$ is explored during $\mathbf{DFS(v)}$ and hence is a descendent of $\mathbf{v}$. Edge $\mathbf{(u, v)}$ implies a cycle in G but G is assumed to be DAG!
- Case 2: $\mathbf{[pre(u), post(u)]}$ is disjoint from $\mathbf{[pre(v), post(v)]}$. This cannot happen since $\mathbf{v}$ would be explored from $\mathbf{u}$.

□

Example

Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS(G)**.

Proof.

If: (u, v) is a back edge implies there is a cycle C consisting of the path from v to u in **DFS** search tree and the edge (u, v).

Only if: Suppose there is a cycle $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$.
Let v_i be first node in C visited in **DFS**.
All other nodes in C are descendants of v_i since they are reachable from v_i.
Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge. □

Back edge and Cycles

Proposition

G has a cycle iff there is a back-edge in **DFS(G)**.

Proof.

If: (u, v) is a back edge implies there is a cycle C consisting of the path from v to u in **DFS** search tree and the edge (u, v).

Only if: Suppose there is a cycle $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$.
Let v_i be first node in C visited in **DFS**.
All other nodes in C are descendants of v_i since they are reachable from v_i.
Therefore, (v_{i-1}, v_i) (or (v_k, v_1) if $i = 1$) is a back edge. □

Topological sorting of a DAG

Input: DAG G. With **n** vertices and **m** edges.

O(n + m) algorithms for topological sorting

(A) Put source **s** of G as first in the order, remove **s**, and repeat. (Implementation not trivial.)

(B) Do **DFS** of G.
Compute post numbers.
Sort vertices by decreasing post number.

Topological sorting of a DAG

Input: DAG G. With **n** vertices and **m** edges.

O(n + m) algorithms for topological sorting

(A) Put source **s** of G as first in the order, remove **s**, and repeat. (Implementation not trivial.)

(B) Do **DFS** of G.
Compute post numbers.
Sort vertices by decreasing post number.

Question

How to avoid sorting?

Topological sorting of a DAG

Input: DAG G. With **n** vertices and **m** edges.

O(n + m) algorithms for topological sorting

(A) Put source **s** of G as first in the order, remove **s**, and repeat. (Implementation not trivial.)

(B) Do **DFS** of G.
Compute post numbers.
Sort vertices by decreasing post number.

Question

How to avoid sorting?
No need to sort - post numbering algorithm can output vertices...

DAGs and Partial Orders

Definition

A **partially ordered set** is a set $\mathbf{S}$ along with a binary relation $\preceq$ such that $\preceq$ is

1. **reflexive** ($\mathbf{a} \preceq \mathbf{a}$ for all $\mathbf{a} \in \mathbf{V}$),
2. **anti-symmetric** ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$ implies $\mathbf{b} \not\preceq \mathbf{a}$), and
3. **transitive** ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{b} \preceq \mathbf{c}$ implies $\mathbf{a} \preceq \mathbf{c}$).

Example: For numbers in the plane define $(\mathbf{x}, \mathbf{y}) \preceq (\mathbf{x}', \mathbf{y}')$ iff $\mathbf{x} \leq \mathbf{x}'$ and $\mathbf{y} \leq \mathbf{y}'$.

Observation: A *finite* partially ordered set is equivalent to a DAG. (No equal elements.)

Observation: A topological sort of a DAG corresponds to a complete (or total) ordering of the underlying partial order.

DAGs and Partial Orders

Definition

A **partially ordered set** is a set $\mathbf{S}$ along with a binary relation $\preceq$ such that $\preceq$ is

1. **reflexive** ($\mathbf{a} \preceq \mathbf{a}$ for all $\mathbf{a} \in \mathbf{V}$),
2. **anti-symmetric** ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$ implies $\mathbf{b} \not\preceq \mathbf{a}$), and
3. **transitive** ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{b} \preceq \mathbf{c}$ implies $\mathbf{a} \preceq \mathbf{c}$).

Example: For numbers in the plane define $(\mathbf{x}, \mathbf{y}) \preceq (\mathbf{x}', \mathbf{y}')$ iff $\mathbf{x} \leq \mathbf{x}'$ and $\mathbf{y} \leq \mathbf{y}'$.

Observation: A *finite* partially ordered set is equivalent to a DAG. (No equal elements.)

Observation: A topological sort of a DAG corresponds to a complete (or total) ordering of the underlying partial order.

DAGs and Partial Orders

Definition

A **partially ordered set** is a set $\mathbf{S}$ along with a binary relation $\preceq$ such that $\preceq$ is

1. **reflexive** ($\mathbf{a} \preceq \mathbf{a}$ for all $\mathbf{a} \in \mathbf{V}$),
2. **anti-symmetric** ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{a} \neq \mathbf{b}$ implies $\mathbf{b} \not\preceq \mathbf{a}$), and
3. **transitive** ($\mathbf{a} \preceq \mathbf{b}$ and $\mathbf{b} \preceq \mathbf{c}$ implies $\mathbf{a} \preceq \mathbf{c}$).

Example: For numbers in the plane define $(\mathbf{x}, \mathbf{y}) \preceq (\mathbf{x}', \mathbf{y}')$ iff $\mathbf{x} \leq \mathbf{x}'$ and $\mathbf{y} \leq \mathbf{y}'$.
Observation: A *finite* partially ordered set is equivalent to a DAG. (No equal elements.)

Observation: A topological sort of a DAG corresponds to a complete (or total) ordering of the underlying partial order.

What's DAG but a sweet old fashioned notion

Who needs a DAG...

Example

1. **V**: set of **n** products (say, **n** different types of tablets).
2. Want to buy one of them, so you do market research...
3. Online reviews compare only pairs of them.
   ...Not everything compared to everything.
4. Given this partial information:
   1. Decide what is the best product.
   2. Decide what is the ordering of products from best to worst.
   3. ...

What DAGs got to do with it?

Or why we should care about DAGs

1. DAGs enable us to represent partial ordering information we have about some set (very common situation in the real world).
2. Questions about DAGs:
   1. Is a graph G a DAG?

      $\Longleftrightarrow$

      Is the partial ordering information we have so far is consistent?
   2. Compute a topological ordering of a DAG.

      $\Longleftrightarrow$

      Find an a consistent ordering that agrees with our partial information.
   3. Find comparisons to do so DAG has a unique topological sort.

      $\Longleftrightarrow$

      Which elements to compare so that we have a consistent ordering of the items.

Part II

Linear time algorithm for finding all strong connected components of a directed graph

Is graph and its reverse together connected?

Let G be a directed graph, and let $G^{\mathbf{rev}}$ be its reverse graph. The graph $\mathbf{H} = G \cup G^{\mathbf{rev}}$ is

(A) always connected.

(B) always disconnected.

(C) connected, if and only if $\mathbf{H}^{\mathrm{SCC}}$ is a single vertex.

(D) disconnected, if and only if G is a DAG.

Finding all SCCs of a Directed Graph

Problem

Given a directed graph $\mathbf{G = (V, E)}$, output *all* its strong connected components.

Straightforward algorithm:

```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u):  Mark u as visited.
```

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

Is there an $\mathbf{O(n+m)}$ time algorithm?

Finding all SCCs of a Directed Graph

Problem

Given a directed graph $\mathbf{G = (V, E)}$, output *all* its strong connected components.

Straightforward algorithm:

```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u):  Mark u as visited.
```

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

Is there an $\mathbf{O(n+m)}$ time algorithm?

Finding all SCCs of a Directed Graph

Problem

Given a directed graph $\mathbf{G = (V, E)}$, output *all* its strong connected components.

Straightforward algorithm:

```
Mark all vertices in V as not visited.
for each vertex u ∈ V not visited yet do
    find SCC(G, u) the strong component of u:
        Compute rch(G, u) using DFS(G, u)
        Compute rch(G^rev, u) using DFS(G^rev, u)
        SCC(G, u) ⇐ rch(G, u) ∩ rch(G^rev, u)
        ∀u ∈ SCC(G, u):  Mark u as visited.
```

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

Is there an $\mathbf{O(n+m)}$ time algorithm?

Structure of a Directed Graph

Graph G

Graph of SCCs G^{SCC}

Reminder

G^{SCC} is created by collapsing every strong connected component to a single vertex.

Proposition

For a directed graph G, its meta-graph G^{SCC} is a DAG.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let **u** be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS(u)** to compute **SCC(u)**
3. Remove **SCC(u)** and repeat

Justification

1. **DFS(u)** only visits vertices (and edges) in **SCC(u)**
2.
3.
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let **u** be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS(u)** to compute **SCC(u)**
3. Remove **SCC(u)** and repeat

Justification

1. **DFS(u)** only visits vertices (and edges) in **SCC(u)**
2.
3.
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let **u** be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS(u)** to compute **SCC(u)**
3. Remove **SCC(u)** and repeat

Justification

1. **DFS(u)** only visits vertices (and edges) in **SCC(u)**
2. ... since there are no edges coming out a sink!
3.
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let **u** be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS(u)** to compute **SCC(u)**
3. Remove **SCC(u)** and repeat

Justification

1. **DFS(u)** only visits vertices (and edges) in **SCC(u)**
2. ... since there are no edges coming out a sink!
3. **DFS(u)** takes time proportional to size of **SCC(u)**
4.

Linear-time Algorithm for SCCs: Ideas

Exploit structure of meta-graph...

Wishful Thinking Algorithm

1. Let **u** be a vertex in a *sink* SCC of G^{SCC}
2. Do **DFS(u)** to compute **SCC(u)**
3. Remove **SCC(u)** and repeat

Justification

1. **DFS(u)** only visits vertices (and edges) in **SCC(u)**
2. ... since there are no edges coming out a sink!
3. **DFS(u)** takes time proportional to size of **SCC(u)**
4. Therefore, total time $\mathbf{O(n+m)}$!

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

Answer: **DFS(G)** gives some information!

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

Answer: **DFS(G)** gives some information!

Big Challenge(s)

How do we find a vertex in a sink SCC of G^{SCC}?

Can we obtain an *implicit* topological sort of G^{SCC} without computing G^{SCC}?

Answer: **DFS(G)** gives some information!

Find source/sink in a DAG using pre-numbers?

Given a DAG G, Consider a pre visit numbering of G using a **DFS**. Which of the following options is correct?

(A) The vertex **u** with minimum **pre(u)** is a sink.

(B) The vertex **u** with minimum **pre(u)** is a source.

(C) The vertex **u** with maximum **pre(u)** is a sink.

(D) The vertex **u** with maximum **pre(u)** is a source.

(E) None of the above.

Find source/sink in a DAG using post-numbers?

Given a DAG G, Consider a post visit numbering of G using a **DFS**. Which of the following options is correct?

(A) The vertex **u** with minimum **post(u)** is a sink.

(B) The vertex **u** with minimum **post(u)** is a source.

(C) The vertex **u** with maximum **post(u)** is a sink.

(D) The vertex **u** with maximum **post(u)** is a source.

(E) None of the above.

Post-visit times of SCCs

Definition

Given G and a SCC $\mathbf{S}$ of G, define $\mathrm{post}(\mathbf{S}) = \max_{u \in S} \mathrm{post}(u)$ where post numbers are with respect to some **DFS(G)**.

An Example

Graph G

Graph with pre-post times for **DFS**(**A**); black edges in tree

Figure : G^{SCC} with post times

Graph of strong connected components

... and post-visit times

Proposition

If $\mathbf{S}$ *and* $\mathbf{S'}$ *are* SCCs *in* G *and* $(\mathbf{S}, \mathbf{S'})$ *is an edge in* G^{SCC} *then* $\mathrm{post}(\mathbf{S}) > \mathrm{post}(\mathbf{S'})$.

Proof.

Let $\mathbf{u}$ be first vertex in $\mathbf{S} \cup \mathbf{S'}$ that is visited.

1. If $\mathbf{u} \in \mathbf{S}$ then all of $\mathbf{S'}$ will be explored before **DFS(u)** completes.
2. If $\mathbf{u} \in \mathbf{S'}$ then all of $\mathbf{S'}$ will be explored before any of $\mathbf{S}$.

□

A False Statement: If $\mathbf{S}$ and $\mathbf{S'}$ are SCCs in G and $(\mathbf{S}, \mathbf{S'})$ is an edge in G^{SCC} then for *every* $\mathbf{u} \in \mathbf{S}$ and $\mathbf{u'} \in \mathbf{S'}$, $\mathrm{post}(\mathbf{u}) > \mathrm{post}(\mathbf{u'})$.

Graph of strong connected components

... and post-visit times

Proposition

If $\mathbf{S}$ *and* $\mathbf{S'}$ *are* SCC*s in* G *and* $(\mathbf{S}, \mathbf{S'})$ *is an edge in* G^{SCC} *then* $\mathrm{post}(\mathbf{S}) > \mathrm{post}(\mathbf{S'})$.

Proof.

Let $\mathbf{u}$ be first vertex in $\mathbf{S} \cup \mathbf{S'}$ that is visited.

1. If $\mathbf{u} \in \mathbf{S}$ then all of $\mathbf{S'}$ will be explored before **DFS(u)** completes.
2. If $\mathbf{u} \in \mathbf{S'}$ then all of $\mathbf{S'}$ will be explored before any of $\mathbf{S}$.

□

A False Statement: If $\mathbf{S}$ and $\mathbf{S'}$ are SCCs in G and $(\mathbf{S}, \mathbf{S'})$ is an edge in G^{SCC} then for *every* $\mathbf{u} \in \mathbf{S}$ and $\mathbf{u'} \in \mathbf{S'}$, $\mathrm{post}(\mathbf{u}) > \mathrm{post}(\mathbf{u'})$.

Topological ordering of the strong components

Corollary

Ordering SCC*s in decreasing order of* $\mathbf{post(S)}$ *gives a topological ordering of* G^{SCC}

Recall: for a DAG, ordering nodes in decreasing post-visit order gives a topological sort.

So...
DFS(G) gives some information on topological ordering of $\mathbf{G}^{\mathrm{SCC}}$!

Topological ordering of the strong components

Corollary

Ordering SCC*s in decreasing order of* $\mathbf{post(S)}$ *gives a topological ordering of* G^{SCC}

Recall: for a DAG, ordering nodes in decreasing post-visit order gives a topological sort.

So...
DFS(G) gives some information on topological ordering of $\mathbf{G}^{\mathrm{SCC}}$!

Finding Sources

Proposition

The vertex $\mathbf{u}$ *with the highest post visit time belongs to a source* SCC *in* G^{SCC}

Proof.

1. $\mathrm{post}(\mathrm{SCC}(\mathbf{u})) = \mathrm{post}(\mathbf{u})$
2. Thus, $\mathrm{post}(\mathrm{SCC}(\mathbf{u}))$ is highest and will be output first in topological ordering of $\mathbf{G}^{\mathrm{SCC}}$.

□

Finding Sources

Proposition

The vertex $\mathbf{u}$ *with the highest post visit time belongs to a source* SCC *in* G^{SCC}

Proof.

1. $\mathbf{post(SCC(u)) = post(u)}$
2. Thus, $\mathbf{post(SCC(u))}$ is highest and will be output first in topological ordering of $\mathbf{G}^{\mathrm{SCC}}$.

□

Finding Sinks

Proposition

The vertex $\mathbf{u}$ *with highest post visit time in* $\mathbf{DFS(G^{rev})}$ *belongs to a sink SCC of* G.

Proof.

1. $\mathbf{u}$ belongs to source SCC of $\mathbf{G^{rev}}$
2. Since graph of SCCs of $\mathbf{G^{rev}}$ is the reverse of G^{SCC}, $\mathbf{SCC(u)}$ is sink SCC of G. □

Finding Sinks

Proposition

The vertex $\mathbf{u}$ *with highest post visit time in* $\mathbf{DFS(G^{rev})}$ *belongs to a sink SCC of* G.

Proof.

1. $\mathbf{u}$ belongs to source SCC of $\mathbf{G^{rev}}$
2. Since graph of SCCs of $\mathbf{G^{rev}}$ is the reverse of G^{SCC}, $\mathbf{SCC(u)}$ is sink SCC of G. □

Linear Time Algorithm

...for computing the strong connected components in **G**

```
do DFS(G^rev) and output vertices in decreasing post order.
Mark all nodes as unvisited
for each u in the computed order do
    if u is not visited then
        DFS(u)
        Let S_u be the nodes reached by u
        Output S_u as a strong connected component
        Remove S_u from G
```

Analysis

Running time is $O(n + m)$. (Exercise)

Linear Time Algorithm: An Example - Initial steps

Graph G:

$\Longrightarrow$

Reverse graph $\mathbf{G^{rev}}$:

DFS of reverse graph:

$\Longrightarrow$

Pre/Post **DFS** numbering of reverse graph:

$\Longrightarrow$

Linear Time Algorithm: An Example

Removing connected components: 1

Original graph G with rev post numbers:

$\Longrightarrow$

Do **DFS** from vertex G remove it.

SCC computed:
{G}

Linear Time Algorithm: An Example

Removing connected components: 2

Do **DFS** from vertex G remove it.

SCC computed:
{G}

$\Longrightarrow$

Do **DFS** from vertex **H**, remove it.

SCC computed:
{G}, {H}

Linear Time Algorithm: An Example

Removing connected components: 3

Do **DFS** from vertex **H**, remove it.

SCC computed:
{G}, {H}

$\Longrightarrow$

Do **DFS** from vertex **B**
Remove visited vertices:
{F, B, E}.

SCC computed:
{G}, {H}, {F, B, E}

Linear Time Algorithm: An Example

Removing connected components: 4

Do **DFS** from vertex **F**
Remove visited vertices:
$\{\mathbf{F}, \mathbf{B}, \mathbf{E}\}$.

$\Longrightarrow$

Do **DFS** from vertex **A**
Remove visited vertices:
$\{\mathbf{A}, \mathbf{C}, \mathbf{D}\}$.

SCC computed:
$\{\mathbf{G}\}, \{\mathbf{H}\}, \{\mathbf{F}, \mathbf{B}, \mathbf{E}\}$

SCC computed:
$\{\mathbf{G}\}, \{\mathbf{H}\}, \{\mathbf{F}, \mathbf{B}, \mathbf{E}\}, \{\mathbf{A}, \mathbf{C}, \mathbf{D}\}$

Linear Time Algorithm: An Example

Final result

SCC computed:

{G}, {H}, {F, B, E}, {A, C, D}

Which is the correct answer!

Obtaining the meta-graph...

Once the strong connected components are computed.

Exercise:

Given all the strong connected components of a directed graph $\mathbf{G = (V, E)}$ show that the meta-graph G^{SCC} can be obtained in $\mathbf{O(m + n)}$ time.

Correctness: more details

1. let $S_1, S_2, \ldots, S_k$ be strong components in G
2. Strong components of G^{rev} and G are same and meta-graph of G is reverse of meta-graph of G^{rev}.
3. consider $\mathbf{DFS}(G^{rev})$ and let $u_1, u_2, \ldots, u_k$ be such that $\mathrm{post}(u_i) = \mathrm{post}(S_i) = \max_{v \in S_i} \mathrm{post}(v)$.
4. Assume without loss of generality that $\mathrm{post}(u_k) > \mathrm{post}(u_{k-1}) \geq \ldots \geq \mathrm{post}(u_1)$ (renumber otherwise). Then $S_k, S_{k-1}, \ldots, S_1$ is a topological sort of meta-graph of G^{rev} and hence $S_1, S_2, \ldots, S_k$ is a topological sort of the meta-graph of G.
5. u_k has highest post number and $\mathbf{DFS}(u_k)$ will explore all of S_k which is a sink component in G.
6. After S_k is removed u_{k-1} has highest post number and $\mathbf{DFS}(u_{k-1})$ will explore all of S_{k-1} which is a sink component in remaining graph $G - S_k$. Formal proof by induction.

Part III

An Application to `make`

Make/Makefile

Clicker question

(A) I know what make/makefile is.

(B) I do NOT know what make/makefile is.

`make` Utility [Feldman]

1. Unix utility for automatically building large software applications
2. A makefile specifies
   1. Object files to be created,
   2. Source/object files to be used in creation, and
   3. How to create them

An Example makefile

```
project:  main.o utils.o command.o
    cc -o project main.o utils.o command.o

main.o:  main.c defs.h
    cc -c main.c
utils.o:  utils.c defs.h command.h
    cc -c utils.c
command.o:  command.c defs.h command.h
    cc -c command.c
```

makefile as a Digraph

Computational Problems for `make`

1. Is the `makefile` reasonable?
2. If it is reasonable, in what order should the object files be created?
3. If it is not reasonable, provide helpful debugging information.
4. If some file is modified, find the fewest compilations needed to make application consistent.

Algorithms for `make`

1. Is the `makefile` reasonable? Is G a DAG?
2. If it is reasonable, in what order should the object files be created? Find a topological sort of a DAG.
3. If it is not reasonable, provide helpful debugging information. Output a cycle. More generally, output all strong connected components.
4. If some file is modified, find the fewest compilations needed to make application consistent.
   1. Find all vertices reachable (using **DFS**/**BFS**) from modified files in directed graph, and recompile them in proper order. Verify that one can find the files to recompile and the ordering in linear time.

Take away Points

1. Given a directed graph G, its SCCs and the associated acyclic meta-graph G^{SCC} give a structural decomposition of G that should be kept in mind.
2. There is a **DFS** based linear time algorithm to compute all the SCCs and the meta-graph. Properties of **DFS** crucial for the algorithm.
3. DAGs arise in many application and topological sort is a key property in algorithm design. Linear time algorithms to compute a topological sort (there can be many possible orderings so not unique).

Notes

Notes

Notes

Notes

Breadth First Search, Dijkstra's Algorithm for Shortest Paths

Lecture 3

January 28, 2014

Part I

Breadth First Search

Breadth First Search (BFS)

Overview

(A) **BFS** is obtained from **BasicSearch** by processing edges using a data structure called a **queue**.

(B) It processes the vertices in the graph in the order of their shortest distance from the vertex **s** (the start vertex).

As such...

1. **DFS** good for exploring graph structure
2. **BFS** good for exploring *distances*

xkcd take on DFS

Queue Data Structure

Queues

A **queue** is a list of elements which supports the operations:

1. **enqueue**: Adds an element to the end of the list
2. **dequeue**: Removes an element from the front of the list

Elements are extracted in **first-in first-out (FIFO)** order, i.e., elements are picked in the order in which they were inserted.

BFS Algorithm

Given (undirected or directed) graph $G = (V, E)$ and node $s \in V$

```
BFS(s)
    Mark all vertices as unvisited
    Initialize search tree T to be empty
    Mark vertex s as visited
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u)
            if v is not visited then
                add edge (u, v) to T
                Mark v as visited and enq(v)
```

Proposition

BFS(s) *runs in* $O(n + m)$ *time.*

BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Undirected Graphs

1. [1]
2. [2,3]
3. [3,4,5]
4. [4,5,7,8]
5. [5,7,8]
6. [7,8,6]
7. [8,6]
8. [6]
9. []

BFS tree is the set of black edges.

BFS: An Example in Undirected Graphs

1.	[1]	4.	[4,5,7,8]	7.	[8,6]
2.	[2,3]	5.	[5,7,8]	8.	[6]
3.	[3,4,5]	6.	[7,8,6]	9.	[]

BFS tree is the set of black edges.

BFS: An Example in Directed Graphs

BFS with Distance

```
BFS(s)
    Mark all vertices as unvisited; for each v set dist(v) = ∞
    Initialize search tree T to be empty
    Mark vertex s as visited and set dist(s) = 0
    set Q to be the empty queue
    enq(s)
    while Q is nonempty do
        u = deq(Q)
        for each vertex v ∈ Adj(u) do
            if v is not visited do
                add edge (u, v) to T
                Mark v as visited, enq(v)
                and set dist(v) = dist(u) + 1
```

Properties of BFS: Undirected Graphs

Proposition

The following properties hold upon termination of **BFS**(**s**)

(A) *The search tree contains exactly the set of vertices in the connected component of* **s**.

(B) *If* $\mathbf{dist(u)} < \mathbf{dist(v)}$ *then* **u** *is visited before* **v**.

(C) *For every vertex* **u**, $\mathbf{dist(u)}$ *is the length of a shortest path (in terms of of edges) from* **s** *to* **u**.

(D) *If* $\mathbf{u, v}$ *are in connected component of* **s** *and* $\mathbf{e = \{u, v\}}$ *is an edge of* **G**, *then* $|\mathbf{dist(u)} - \mathbf{dist(v)}| \leq \mathbf{1}$.

Proof.

Exercise. □

Properties of BFS: Directed Graphs

Proposition

The following properties hold upon termination of **BFS**(**s**)*:*

(A) *The search tree contains exactly the set of vertices reachable from* **s**

(B) *If* $\mathbf{dist(u)} < \mathbf{dist(v)}$ *then* **u** *is visited before* **v**

(C) *For every vertex* **u**, $\mathbf{dist(u)}$ *is indeed the length of shortest path from* **s** *to* **u**

(D) *If* **u** *is reachable from* **s** *and* $\mathbf{e = (u, v)}$ *is an edge of* **G**, *then* $\mathbf{dist(v)} - \mathbf{dist(u)} \leq \mathbf{1}$. *Not necessarily the case that* $\mathbf{dist(u)} - \mathbf{dist(v)} \leq \mathbf{1}$.

Proof.

Exercise. □

BFS with Layers

```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
            initialize L_{i+1} to be an empty list
            for each u in L_i do
                for each edge (u, v) ∈ Adj(u) do
                if v is not visited
                        mark v as visited
                        add (u, v) to tree T
                        add v to L_{i+1}
            i = i + 1
```

Running time: $O(n + m)$

BFS with Layers

```
BFSLayers(s):
    Mark all vertices as unvisited and initialize T to be empty
    Mark s as visited and set L_0 = {s}
    i = 0
    while L_i is not empty do
            initialize L_{i+1} to be an empty list
            for each u in L_i do
                for each edge (u, v) ∈ Adj(u) do
                if v is not visited
                        mark v as visited
                        add (u, v) to tree T
                        add v to L_{i+1}
            i = i + 1
```

Running time: $O(n + m)$

Example

BFS with Layers: Properties

Proposition

The following properties hold on termination of **BFSLayers**(**s**).

1. **BFSLayers**(**s**) *outputs a* **BFS** *tree*
2. $\mathbf{L_i}$ *is the set of vertices at distance exactly* $\mathbf{i}$ *from* $\mathbf{s}$
3. *If* $\mathbf{G}$ *is undirected, each edge* $\mathbf{e} = \{\mathbf{u}, \mathbf{v}\}$ *is one of three types:*
   1. **tree** *edge between two* consecutive *layers*
   2. *non-tree* **forward**/**backward** *edge between two consecutive layers*
   3. *non-tree* **cross-edge** *with both* $\mathbf{u}, \mathbf{v}$ *in same layer*
   4. $\Longrightarrow$ *Every edge in the graph is either between two vertices that are either (i) in the same layer, or (ii) in two consecutive layers.*

BFS with Layers: Properties

For directed graphs

Proposition

The following properties hold on termination of **BFSLayers**(**s**)*, if* **G** *is directed.*

For each edge $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ *is one of four types:*

1. *a* **tree** *edge between consecutive layers,* $\mathbf{u} \in \mathbf{L_i}, \mathbf{v} \in \mathbf{L_{i+1}}$ *for some* $\mathbf{i} \geq \mathbf{0}$
2. *a non-tree* **forward** *edge between consecutive layers*
3. *a non-tree* **backward** *edge*
4. *a* **cross-edge** *with both* $\mathbf{u}, \mathbf{v}$ *in same layer*

Example

Part II

Bipartite Graphs and an application of BFS

Bipartite Graphs

Definition (Bipartite Graph)

Undirected graph $\mathbf{G = (V, E)}$ is a **bipartite graph** if $\mathbf{V}$ can be partitioned into $\mathbf{X}$ and $\mathbf{Y}$ s.t. all edges in $\mathbf{E}$ are between $\mathbf{X}$ and $\mathbf{Y}$.

Bipartite Graph Characterization

Question

When is a graph bipartite?

Proposition

Every tree is a bipartite graph.

Proof.

Root tree T at some node r. Let L_i be all nodes at level i, that is, L_i is all nodes at distance i from root r. Now define X to be all nodes at even levels and Y to be all nodes at odd level. Only edges in T are between levels. □

Proposition

An odd length cycle is not bipartite.

Bipartite Graph Characterization

Question

When is a graph bipartite?

Proposition

Every tree is a bipartite graph.

Proof.

Root tree T at some node r. Let L_i be all nodes at level i, that is, L_i is all nodes at distance i from root r. Now define X to be all nodes at even levels and Y to be all nodes at odd level. Only edges in T are between levels. □

Proposition

An odd length cycle is not bipartite.

Bipartite Graph Characterization

Question

When is a graph bipartite?

Proposition

Every tree is a bipartite graph.

Proof.

Root tree $\mathbf{T}$ at some node $\mathbf{r}$. Let $\mathbf{L_i}$ be all nodes at level $\mathbf{i}$, that is, $\mathbf{L_i}$ is all nodes at distance $\mathbf{i}$ from root $\mathbf{r}$. Now define $\mathbf{X}$ to be all nodes at even levels and $\mathbf{Y}$ to be all nodes at odd level. Only edges in $\mathbf{T}$ are between levels. □

Proposition

An odd length cycle is not bipartite.

Bipartite Graph Characterization

Question

When is a graph bipartite?

Proposition

Every tree is a bipartite graph.

Proof.

Root tree $\mathbf{T}$ at some node $\mathbf{r}$. Let $\mathbf{L_i}$ be all nodes at level $\mathbf{i}$, that is, $\mathbf{L_i}$ is all nodes at distance $\mathbf{i}$ from root $\mathbf{r}$. Now define $\mathbf{X}$ to be all nodes at even levels and $\mathbf{Y}$ to be all nodes at odd level. Only edges in $\mathbf{T}$ are between levels. □

Proposition

An odd length cycle is not bipartite.

Odd Cycles are not Bipartite

Proposition

An odd length cycle is not bipartite.

Proof.

Let $C = u_1, u_2, \ldots, u_{2k+1}, u_1$ be an odd cycle. Suppose C is a bipartite graph and let X, Y be the partition. Without loss of generality $u_1 \in X$. Implies $u_2 \in Y$. Implies $u_3 \in X$. Inductively, $u_i \in X$ if i is odd $u_i \in Y$ if i is even. But $\{u_1, u_{2k+1}\}$ is an edge and both belong to X! $\square$

Subgraphs

Definition

Given a graph $\mathbf{G = (V, E)}$ a **subgraph** of $\mathbf{G}$ is another graph $\mathbf{H = (V', E')}$ where $\mathbf{V' \subseteq V}$ and $\mathbf{E' \subseteq E}$.

Proposition

If $\mathbf{G}$ is bipartite then any subgraph $\mathbf{H}$ of $\mathbf{G}$ is also bipartite.

Proposition

A graph $\mathbf{G}$ is not bipartite if $\mathbf{G}$ has an odd cycle $\mathbf{C}$ as a subgraph.

Proof.

If $\mathbf{G}$ is bipartite then since $\mathbf{C}$ is a subgraph, $\mathbf{C}$ is also bipartite (by above proposition). However, $\mathbf{C}$ is not bipartite! □

Subgraphs

Definition

Given a graph $\mathbf{G = (V, E)}$ a **subgraph** of $\mathbf{G}$ is another graph $\mathbf{H = (V', E')}$ where $\mathbf{V' \subseteq V}$ and $\mathbf{E' \subseteq E}$.

Proposition

If $\mathbf{G}$ *is bipartite then any subgraph* $\mathbf{H}$ *of* $\mathbf{G}$ *is also bipartite.*

Proposition

A graph $\mathbf{G}$ *is not bipartite if* $\mathbf{G}$ *has an odd cycle* $\mathbf{C}$ *as a subgraph.*

Proof.

If $\mathbf{G}$ is bipartite then since $\mathbf{C}$ is a subgraph, $\mathbf{C}$ is also bipartite (by above proposition). However, $\mathbf{C}$ is not bipartite! □

Subgraphs

Definition

Given a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ a **subgraph** of $\mathbf{G}$ is another graph $\mathbf{H} = (\mathbf{V}', \mathbf{E}')$ where $\mathbf{V}' \subseteq \mathbf{V}$ and $\mathbf{E}' \subseteq \mathbf{E}$.

Proposition

If $\mathbf{G}$ is bipartite then any subgraph $\mathbf{H}$ of $\mathbf{G}$ is also bipartite.

Proposition

A graph $\mathbf{G}$ is not bipartite if $\mathbf{G}$ has an odd cycle $\mathbf{C}$ as a subgraph.

Proof.

If $\mathbf{G}$ is bipartite then since $\mathbf{C}$ is a subgraph, $\mathbf{C}$ is also bipartite (by above proposition). However, $\mathbf{C}$ is not bipartite! □

Subgraphs

Definition

Given a graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ a **subgraph** of $\mathbf{G}$ is another graph $\mathbf{H} = (\mathbf{V}', \mathbf{E}')$ where $\mathbf{V}' \subseteq \mathbf{V}$ and $\mathbf{E}' \subseteq \mathbf{E}$.

Proposition

If $\mathbf{G}$ is bipartite then any subgraph $\mathbf{H}$ of $\mathbf{G}$ is also bipartite.

Proposition

A graph $\mathbf{G}$ is not bipartite if $\mathbf{G}$ has an odd cycle $\mathbf{C}$ as a subgraph.

Proof.

If $\mathbf{G}$ is bipartite then since $\mathbf{C}$ is a subgraph, $\mathbf{C}$ is also bipartite (by above proposition). However, $\mathbf{C}$ is not bipartite! $\square$

Bipartite Graph Characterization

Theorem

A graph $\mathbf{G}$ *is bipartite if and only if it has no odd length cycle as subgraph.*

Proof.

Only If: $\mathbf{G}$ has an odd cycle implies $\mathbf{G}$ is not bipartite.

If: $\mathbf{G}$ has no odd length cycle. Assume without loss of generality that $\mathbf{G}$ is connected.

1. Pick $\mathbf{u}$ arbitrarily and do $\mathbf{BFS}(\mathbf{u})$
2. $\mathbf{X} = \cup_{\mathbf{i} \text{ is even}} \mathbf{L_i}$ and $\mathbf{Y} = \cup_{\mathbf{i} \text{ is odd}} \mathbf{L_i}$
3. **Claim:** $\mathbf{X}$ and $\mathbf{Y}$ is a valid partition if $\mathbf{G}$ has no odd length cycle.

□

Bipartite Graph Characterization

Theorem

A graph $\mathbf{G}$ *is bipartite if and only if it has no odd length cycle as subgraph.*

Proof.

Only If: $\mathbf{G}$ has an odd cycle implies $\mathbf{G}$ is not bipartite.

If: $\mathbf{G}$ has no odd length cycle. Assume without loss of generality that $\mathbf{G}$ is connected.

1. Pick $\mathbf{u}$ arbitrarily and do $\mathbf{BFS}(\mathbf{u})$
2. $\mathbf{X} = \cup_{\mathbf{i} \text{ is even}} \mathbf{L_i}$ and $\mathbf{Y} = \cup_{\mathbf{i} \text{ is odd}} \mathbf{L_i}$
3. **Claim:** $\mathbf{X}$ and $\mathbf{Y}$ is a valid partition if $\mathbf{G}$ has no odd length cycle.

□

Proof of Claim

Claim

In **BFS**$(\mathbf{u})$ *if* $\mathbf{a}, \mathbf{b} \in \mathbf{L_i}$ *and* $(\mathbf{a}, \mathbf{b})$ *is an edge then there is an odd length cycle containing* $(\mathbf{a}, \mathbf{b})$.

Proof.

Let $\mathbf{v}$ be least common ancestor of $\mathbf{a}, \mathbf{b}$ in **BFS** tree $\mathbf{T}$.
$\mathbf{v}$ is in some level $\mathbf{j} < \mathbf{i}$ (could be $\mathbf{u}$ itself).
Path from $\mathbf{v} \rightsquigarrow \mathbf{a}$ in $\mathbf{T}$ is of length $\mathbf{i} - \mathbf{j}$.
Path from $\mathbf{v} \rightsquigarrow \mathbf{b}$ in $\mathbf{T}$ is of length $\mathbf{i} - \mathbf{j}$.
These two paths plus $(\mathbf{a}, \mathbf{b})$ forms an odd cycle of length $2(\mathbf{i} - \mathbf{j}) + 1$. □

Proof of Claim

Claim

In **BFS**$(\mathbf{u})$ *if* $\mathbf{a}, \mathbf{b} \in \mathbf{L_i}$ *and* $(\mathbf{a}, \mathbf{b})$ *is an edge then there is an odd length cycle containing* $(\mathbf{a}, \mathbf{b})$.

Proof.

Let $\mathbf{v}$ be least common ancestor of $\mathbf{a}, \mathbf{b}$ in **BFS** tree $\mathbf{T}$.
$\mathbf{v}$ is in some level $\mathbf{j} < \mathbf{i}$ (could be $\mathbf{u}$ itself).
Path from $\mathbf{v} \rightsquigarrow \mathbf{a}$ in $\mathbf{T}$ is of length $\mathbf{i} - \mathbf{j}$.
Path from $\mathbf{v} \rightsquigarrow \mathbf{b}$ in $\mathbf{T}$ is of length $\mathbf{i} - \mathbf{j}$.
These two paths plus $(\mathbf{a}, \mathbf{b})$ forms an odd cycle of length $\mathbf{2(i - j) + 1}$. $\square$

Proof of Claim: Figure

Is this graph bipartite?

Is this graph bipartite?

Is this graph bipartite?

Is this graph bipartite?

Is this graph bipartite?

Is this graph bipartite?

Is this graph bipartite?

Is this graph bipartite?

Another tidbit

Corollary

There is an $\mathbf{O(n+m)}$ *time algorithm to check if* $\mathbf{G}$ *is bipartite and output an odd cycle if it is not.*

Question: Can you come up with an efficient algorithm to check whether a given graph $\mathbf{G}$ has an even length cycle and find one if it has? What is the running time of your algorithm?

Part III

Shortest Paths and Dijkstra's Algorithm

Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with edge lengths (or costs). For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

1. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.
3. Find shortest paths for all pairs of nodes.

Many applications!

Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with edge lengths (or costs). For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

1. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.
3. Find shortest paths for all pairs of nodes.

Many applications!

Single-Source Shortest Paths:

Non-Negative Edge Lengths

Single-Source Shortest Path Problems

1. Input: A (undirected or directed) graph $\mathbf{G = (V, E)}$ with non-negative edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.
2. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
3. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

1. Restrict attention to directed graphs
2. Undirected graph problem can be reduced to directed graph problem - how?
   1. Given undirected graph $\mathbf{G}$, create a new directed graph $\mathbf{G'}$ by replacing each edge $\{\mathbf{u, v}\}$ in $\mathbf{G}$ by $(\mathbf{u, v})$ and $(\mathbf{v, u})$ in $\mathbf{G'}$.
   2. set $\ell(\mathbf{u, v}) = \ell(\mathbf{v, u}) = \ell(\{\mathbf{u, v}\})$
   3. Exercise: show reduction works. Relies on non-negativity!

Single-Source Shortest Paths:

Non-Negative Edge Lengths

Single-Source Shortest Path Problems

1. Input: A (undirected or directed) graph $\mathbf{G = (V, E)}$ with non-negative edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.
2. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
3. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

1. Restrict attention to directed graphs
2. Undirected graph problem can be reduced to directed graph problem - how?
   1. Given undirected graph $\mathbf{G}$, create a new directed graph $\mathbf{G'}$ by replacing each edge $\{\mathbf{u, v}\}$ in $\mathbf{G}$ by $(\mathbf{u, v})$ and $(\mathbf{v, u})$ in $\mathbf{G'}$.
   2. set $\ell(\mathbf{u, v}) = \ell(\mathbf{v, u}) = \ell(\{\mathbf{u, v}\})$
   3. Exercise: show reduction works. Relies on non-negativity!

Single-Source Shortest Paths:

Non-Negative Edge Lengths

Single-Source Shortest Path Problems

1. Input: A (undirected or directed) graph $\mathbf{G = (V, E)}$ with non-negative edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.
2. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
3. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

1. Restrict attention to directed graphs
2. Undirected graph problem can be reduced to directed graph problem - how?
   1. Given undirected graph $\mathbf{G}$, create a new directed graph $\mathbf{G'}$ by replacing each edge $\{\mathbf{u, v}\}$ in $\mathbf{G}$ by $(\mathbf{u, v})$ and $(\mathbf{v, u})$ in $\mathbf{G'}$.
   2. set $\ell(\mathbf{u, v}) = \ell(\mathbf{v, u}) = \ell(\{\mathbf{u, v}\})$
   3. Exercise: show reduction works. Relies on non-negativity!

Single-Source Shortest Paths via BFS

Special case: All edge lengths are $\mathbf{1}$.

1. Run **BFS**($\mathbf{s}$) to get shortest path distances from s to all other nodes.
2. $\mathbf{O(m+n)}$ time algorithm.

Special case: Suppose $\ell(\mathbf{e})$ is an integer for all $\mathbf{e}$?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(\mathbf{e}) - \mathbf{1}$ dummy nodes on $\mathbf{e}$

Let $\mathbf{L} = \max_{\mathbf{e}} \ell(\mathbf{e})$. New graph has $\mathbf{O(mL)}$ edges and $\mathbf{O(mL+n)}$ nodes. **BFS** takes $\mathbf{O(mL+n)}$ time. Not efficient if $\mathbf{L}$ is large.

Single-Source Shortest Paths via BFS

Special case: All edge lengths are $\mathbf{1}$.

1. Run **BFS**(**s**) to get shortest path distances from s to all other nodes.
2. $\mathbf{O(m+n)}$ time algorithm.

Special case: Suppose $\ell(\mathbf{e})$ is an integer for all $\mathbf{e}$?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(\mathbf{e}) - \mathbf{1}$ dummy nodes on $\mathbf{e}$

Let $\mathbf{L} = \max_{\mathbf{e}} \ell(\mathbf{e})$. New graph has $\mathbf{O(mL)}$ edges and $\mathbf{O(mL+n)}$ nodes. **BFS** takes $\mathbf{O(mL+n)}$ time. Not efficient if $\mathbf{L}$ is large.

Single-Source Shortest Paths via BFS

Special case: All edge lengths are $\mathbf{1}$.

1. Run **BFS**($\mathbf{s}$) to get shortest path distances from s to all other nodes.
2. $\mathbf{O(m+n)}$ time algorithm.

Special case: Suppose $\ell(\mathbf{e})$ is an integer for all $\mathbf{e}$?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(\mathbf{e}) - \mathbf{1}$ dummy nodes on $\mathbf{e}$

Let $\mathbf{L} = \max_{\mathbf{e}} \ell(\mathbf{e})$. New graph has $\mathbf{O(mL)}$ edges and $\mathbf{O(mL+n)}$ nodes. **BFS** takes $\mathbf{O(mL+n)}$ time. Not efficient if $\mathbf{L}$ is large.

Single-Source Shortest Paths via BFS

Special case: All edge lengths are $\mathbf{1}$.

1. Run **BFS**($\mathbf{s}$) to get shortest path distances from s to all other nodes.
2. $\mathbf{O(m+n)}$ time algorithm.

Special case: Suppose $\ell(\mathbf{e})$ is an integer for all $\mathbf{e}$?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(\mathbf{e}) - \mathbf{1}$ dummy nodes on $\mathbf{e}$

Let $\mathbf{L} = \max_{\mathbf{e}} \ell(\mathbf{e})$. New graph has $\mathbf{O(mL)}$ edges and $\mathbf{O(mL+n)}$ nodes. **BFS** takes $\mathbf{O(mL+n)}$ time. Not efficient if $\mathbf{L}$ is large.

Single-Source Shortest Paths via BFS

Special case: All edge lengths are $\mathbf{1}$.

1. Run **BFS**($\mathbf{s}$) to get shortest path distances from s to all other nodes.
2. $\mathbf{O(m+n)}$ time algorithm.

Special case: Suppose $\ell(\mathbf{e})$ is an integer for all $\mathbf{e}$?
Can we use **BFS**? Reduce to unit edge-length problem by placing $\ell(\mathbf{e}) - \mathbf{1}$ dummy nodes on $\mathbf{e}$

Let $\mathbf{L} = \max_{\mathbf{e}} \ell(\mathbf{e})$. New graph has $\mathbf{O(mL)}$ edges and $\mathbf{O(mL+n)}$ nodes. **BFS** takes $\mathbf{O(mL+n)}$ time. Not efficient if $\mathbf{L}$ is large.

Towards an algorithm

Why does **BFS** work?

BFS(s) explores nodes in increasing distance from s

Lemma

Let G be a directed graph with non-negative edge lengths. Let $\mathrm{dist}(s, v)$ denote the shortest path length from s to v. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

1. $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ *is a shortest path from* s *to* v_i
2. $\mathrm{dist}(s, v_i) \leq \mathrm{dist}(s, v_k)$.

Proof.

Suppose not. Then for some $i < k$ there is a path P' from s to v_i of length strictly less than that of $s = v_0 \to v_1 \to \ldots \to v_i$. Then P' concatenated with $v_i \to v_{i+1} \ldots \to v_k$ contains a strictly shorter

Towards an algorithm

Why does **BFS** work?
BFS(s) explores nodes in increasing distance from **s**

Lemma

Let G *be a directed graph with non-negative edge lengths. Let* $\mathrm{dist}(s, v)$ *denote the shortest path length from* s *to* v*. If* $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ *is a shortest path from* s *to* v_k *then for* $1 \leq i < k$*:*

1. $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ *is a shortest path from* s *to* v_i
2. $\mathrm{dist}(s, v_i) \leq \mathrm{dist}(s, v_k)$.

Proof.

Suppose not. Then for some $i < k$ there is a path P' from s to v_i of length strictly less than that of $s = v_0 \to v_1 \to \ldots \to v_i$. Then P' concatenated with $v_i \to v_{i+1} \ldots \to v_k$ contains a strictly shorter

Towards an algorithm

Why does **BFS** work?
BFS(s) explores nodes in increasing distance from **s**

Lemma

Let $\mathbf{G}$ *be a directed graph with non-negative edge lengths. Let* $\mathrm{dist}(\mathbf{s}, \mathbf{v})$ *denote the shortest path length from* $\mathbf{s}$ *to* $\mathbf{v}$*. If* $\mathbf{s} = \mathbf{v}_0 \to \mathbf{v}_1 \to \mathbf{v}_2 \to \ldots \to \mathbf{v}_k$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v}_k$ *then for* $1 \leq i < k$*:*

1. $\mathbf{s} = \mathbf{v}_0 \to \mathbf{v}_1 \to \mathbf{v}_2 \to \ldots \to \mathbf{v}_i$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v}_i$
2. $\mathrm{dist}(\mathbf{s}, \mathbf{v}_i) \leq \mathrm{dist}(\mathbf{s}, \mathbf{v}_k)$.

Proof.

Suppose not. Then for some $i < k$ there is a path P' from s to v_i of length strictly less than that of $s = v_0 \to v_1 \to \ldots \to v_i$. Then P' concatenated with $v_i \to v_{i+1} \ldots \to v_k$ contains a strictly shorter

Towards an algorithm

Lemma

Let $\mathbf{G}$ *be a directed graph with non-negative edge lengths. Let* $\mathrm{dist}(\mathbf{s}, \mathbf{v})$ *denote the shortest path length from* $\mathbf{s}$ *to* $\mathbf{v}$*. If* $\mathbf{s} = \mathbf{v}_0 \to \mathbf{v}_1 \to \mathbf{v}_2 \to \ldots \to \mathbf{v}_k$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v}_k$ *then for* $1 \leq i < k$*:*

1. $\mathbf{s} = \mathbf{v}_0 \to \mathbf{v}_1 \to \mathbf{v}_2 \to \ldots \to \mathbf{v}_i$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v}_i$
2. $\mathrm{dist}(\mathbf{s}, \mathbf{v}_i) \leq \mathrm{dist}(\mathbf{s}, \mathbf{v}_k)$.

Proof.

Suppose not. Then for some $i < k$ there is a path P' from $\mathbf{s}$ to $\mathbf{v}_i$ of length strictly less than that of $\mathbf{s} = \mathbf{v}_0 \to \mathbf{v}_1 \to \ldots \to \mathbf{v}_i$. Then P' concatenated with $\mathbf{v}_i \to \mathbf{v}_{i+1} \ldots \to \mathbf{v}_k$ contains a strictly shorter path to $\mathbf{v}_k$ than $\mathbf{s} = \mathbf{v}_0 \to \mathbf{v}_1 \ldots \to \mathbf{v}_k$. For the second part, observe that edge lengths are non-negative. □

A proof by picture

Shortest path from v_0 to v_6

A proof by picture

A proof by picture

A Basic Strategy

Explore vertices in increasing order of distance from s:
(For simplicity assume that nodes are at different distances from s and that no edge has zero length)

```
Initialize for each node v, dist(s,v) = ∞
Initialize S = ∅,
for i = 1 to |V| do
    (* Invariant:  S contains the i − 1 closest nodes to s *)
    Among nodes in V \ S, find the node v that is the
            ith closest to s
    Update dist(s,v)
    S = S ∪ {v}
```

How can we implement the step in the for loop?

A Basic Strategy

Explore vertices in increasing order of distance from s:
(For simplicity assume that nodes are at different distances from s and that no edge has zero length)

```
Initialize for each node v, dist(s,v) = ∞
Initialize S = ∅,
for i = 1 to |V| do
    (* Invariant:  S contains the i − 1 closest nodes to s *)
    Among nodes in V \ S, find the node v that is the
            ith closest to s
    Update dist(s,v)
    S = S ∪ {v}
```

How can we implement the step in the for loop?

Finding the ith closest node

1. S contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - S$.

What do we know about the ith closest node?

Claim

Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to S.

Proof.

If P had an intermediate node u not in S then u will be closer to s than v. Implies v is not the ith closest node to s - recall that S already has the $i - 1$ closest nodes. □

Finding the ith closest node

1. S contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - S$.

What do we know about the ith closest node?

Claim

Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to S.

Proof.

If P had an intermediate node u not in S then u will be closer to s than v. Implies v is not the ith closest node to s - recall that S already has the $i - 1$ closest nodes. □

Finding the ith closest node

1. S contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - S$.

What do we know about the ith closest node?

Claim

Let P be a shortest path from s to v where v is the ith closest node. Then, all intermediate nodes in P belong to S.

Proof.

If P had an intermediate node u not in S then u will be closer to s than v. Implies v is not the ith closest node to s - recall that S already has the $i - 1$ closest nodes. □

Finding the ith closest node repeatedly

An example

Finding the ith closest node repeatedly

An example

Finding the ith closest node repeatedly

An example

Finding the ith closest node repeatedly

An example

Finding the ith closest node repeatedly

An example

Finding the ith closest node repeatedly

An example

Finding the ith closest node repeatedly

An example

Finding the ith closest node repeatedly

An example

Finding the ith closest node repeatedly

An example

Finding the ith closest node

Corollary

The **i***th closest node is adjacent to* **S***.*

Finding the ith closest node

1. S contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - S$.

1. For each $u \in V - S$ let $P(s, u, S)$ be a shortest path from s to u using only nodes in S as intermediate vertices.
2. Let $d'(s, u)$ be the length of $P(s, u, S)$

Observations: for each $u \in V - S$,

1. $\mathrm{dist}(s, u) \leq d'(s, u)$ since we are constraining the paths
2. $d'(s, u) = \min_{a \in S}(\mathrm{dist}(s, a) + \ell(a, u))$ - Why?

Lemma

If v is the ith closest node to s, then $d'(s, v) = \mathrm{dist}(s, v)$.

Finding the ith closest node

1. S contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - S$.

1. For each $u \in V - S$ let $P(s, u, S)$ be a shortest path from s to u using only nodes in S as intermediate vertices.
2. Let $d'(s, u)$ be the length of $P(s, u, S)$

Observations: for each $u \in V - S$,

1. $\mathrm{dist}(s, u) \leq d'(s, u)$ since we are constraining the paths
2. $d'(s, u) = \min_{a \in S}(\mathrm{dist}(s, a) + \ell(a, u))$ - Why?

Lemma

If v is the ith closest node to s, then $d'(s, v) = \mathrm{dist}(s, v)$.

Finding the ith closest node

1. S contains the $i - 1$ closest nodes to s
2. Want to find the ith closest node from $V - S$.

1. For each $u \in V - S$ let $P(s, u, S)$ be a shortest path from s to u using only nodes in S as intermediate vertices.
2. Let $d'(s, u)$ be the length of $P(s, u, S)$

Observations: for each $u \in V - S$,

1. $\mathrm{dist}(s, u) \leq d'(s, u)$ since we are constraining the paths
2. $d'(s, u) = \min_{a \in S}(\mathrm{dist}(s, a) + \ell(a, u))$ - Why?

Lemma

If v is the ith closest node to s, then $d'(s, v) = \mathrm{dist}(s, v)$.

Finding the ith closest node

Lemma

Given:

1. $\mathbf{S}$: *Set of* $\mathbf{i-1}$ *closest nodes to* $\mathbf{s}$.
2. $\mathbf{d'(s,u) = \min_{x \in S}(\mathrm{dist}(s,x) + \ell(x,u))}$

If $\mathbf{v}$ *is an* $\mathbf{i}$*th closest node to* $\mathbf{s}$*, then* $\mathbf{d'(s,v) = \mathrm{dist}(s,v)}$.

Proof.

Let $\mathbf{v}$ be the $\mathbf{i}$th closest node to $\mathbf{s}$. Then there is a shortest path $\mathbf{P}$ from $\mathbf{s}$ to $\mathbf{v}$ that contains only nodes in $\mathbf{S}$ as intermediate nodes (see previous claim). Therefore $\mathbf{d'(s,v) = \mathrm{dist}(s,v)}$. □

Finding the ith closest node

Lemma

If $\mathbf{v}$ *is an* $\mathbf{i}$*th closest node to* $\mathbf{s}$*, then* $\mathbf{d'(s,v)} = \mathrm{dist}(\mathbf{s},\mathbf{v})$.

Corollary

The $\mathbf{i}$*th closest node to* $\mathbf{s}$ *is the node* $\mathbf{v} \in \mathbf{V} - \mathbf{S}$ *such that* $\mathbf{d'(s,v)} = \mathbf{min}_{\mathbf{u} \in \mathbf{V}-\mathbf{S}}\, \mathbf{d'(s,u)}$.

Proof.

For every node $\mathbf{u} \in \mathbf{V} - \mathbf{S}$, $\mathrm{dist}(\mathbf{s},\mathbf{u}) \leq \mathbf{d'(s,u)}$ and for the $\mathbf{i}$th closest node $\mathbf{v}$, $\mathrm{dist}(\mathbf{s},\mathbf{v}) = \mathbf{d'(s,v)}$. Moreover, $\mathrm{dist}(\mathbf{s},\mathbf{u}) \geq \mathrm{dist}(\mathbf{s},\mathbf{v})$ for each $\mathbf{u} \in \mathbf{V} - \mathbf{S}$. □

Algorithm

```
Initialize for each node v:  dist(s,v) = ∞
Initialize S = ∅, d'(s,s) = 0
for i = 1 to |V| do
    (* Invariant:  S contains the i-1 closest nodes to s *)
    (* Invariant:  d'(s,u) is shortest path distance from u to
     using only S as intermediate nodes*)
    Let v be such that d'(s,v) = min_{u∈V−S} d'(s,u)
    dist(s,v) = d'(s,v)
    S = S ∪ {v}
    for each node u in V \ S do
        d'(s,u) ⇐ min_{a∈S} ( dist(s,a) + ℓ(a,u) )
```

Correctness: By induction on i using previous lemmas.

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

1. n outer iterations. In each iteration, $d'(s, u)$ for each u by scanning all edges out of nodes in S; $O(m + n)$ time/iteration.

Algorithm

```
Initialize for each node v:  dist(s,v) = ∞
Initialize S = ∅, d'(s,s) = 0
for i = 1 to |V| do
    (* Invariant:  S contains the i-1 closest nodes to s *)
    (* Invariant:  d'(s,u) is shortest path distance from u to
     using only S as intermediate nodes*)
    Let v be such that d'(s,v) = min_{u∈V−S} d'(s,u)
    dist(s,v) = d'(s,v)
    S = S ∪ {v}
    for each node u in V \ S do
        d'(s,u) ⇐ min_{a∈S} ( dist(s,a) + ℓ(a,u) )
```

Correctness: By induction on $\mathbf{i}$ using previous lemmas.

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

1. n outer iterations. In each iteration, $d'(s, u)$ for each u by scanning all edges out of nodes in S; $O(m + n)$ time/iteration.

Algorithm

```
Initialize for each node v:  dist(s,v) = ∞
Initialize S = ∅, d'(s,s) = 0
for i = 1 to |V| do
    (* Invariant:  S contains the i-1 closest nodes to s *)
    (* Invariant:  d'(s,u) is shortest path distance from u to
     using only S as intermediate nodes*)
    Let v be such that d'(s,v) = min_{u∈V−S} d'(s,u)
    dist(s,v) = d'(s,v)
    S = S ∪ {v}
    for each node u in V \ S do
        d'(s,u) ⇐ min_{a∈S} ( dist(s,a) + ℓ(a,u) )
```

Correctness: By induction on $\mathbf{i}$ using previous lemmas.

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

1. n outer iterations. In each iteration, $d'(s, u)$ for each u by scanning all edges out of nodes in S; $O(m + n)$ time/iteration.

Algorithm

```
Initialize for each node v:  dist(s,v) = ∞
Initialize S = ∅, d'(s,s) = 0
for i = 1 to |V| do
    (* Invariant:  S contains the i-1 closest nodes to s *)
    (* Invariant:  d'(s,u) is shortest path distance from u to
     using only S as intermediate nodes*)
    Let v be such that d'(s,v) = min_{u∈V−S} d'(s,u)
    dist(s,v) = d'(s,v)
    S = S ∪ {v}
    for each node u in V \ S do
        d'(s,u) ⇐ min_{a∈S} ( dist(s,a) + ℓ(a,u) )
```

Correctness: By induction on $\mathbf{i}$ using previous lemmas.

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

1. $\mathbf{n}$ outer iterations. In each iteration, $\mathbf{d'(s,u)}$ for each $\mathbf{u}$ by scanning all edges out of nodes in $\mathbf{S}$; $\mathbf{O(m + n)}$ time/iteration.

Example

s
2
3
4
5
6
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25

0
9
19
6
36
25
13
38
9
10
6
6
11
13
25

Example

s
2
3
4
5
6
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25

0
9
19
6
36
25
13
38
9
10
6
6
11
13
25

Example

s
2
3
6
4
5
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25

0
9
19
6
36
25
13
38
9
10
6
6
11
13
25

Example

s
2
3
4
5
6
7
t
9
10
6
18
6
6
30
11
13
20
8
19
6
16
25

0
9
6
13
9
10
19
6
6
11
36
25
13
25
38

Example

s
2
3
6
4
5
7
t
9
10
6
18
6
6
30
11
13
20
8
19
6
16
25

0
9
19
6
6
36
25
13
13
38
9
10
6
6
11
13
25

Example

s
2
3
6
4
5
7
t
9
10
6
18
6
6
30
11
13
20
8
19
6
16
25

0
9
19
6
25
36
13
38
9
10
6
6
11
13
25

Example

s
2
3
4
5
6
7
t
9
10
18
6
6
6
30
11
13
20
8
19
6
16
25

0
9
19
6
36
25
13
38
9
10
6
6
11
13
25

Example

s
2
3
6
4
5
7
t
9
10
6
18
6
6
30
11
13
20
8
19
6
16
25

0
9
19
6
36
25
13
38
9
10
6
6
11
13
25

Improved Algorithm

1. Main work is to compute the $\mathbf{d'(s,u)}$ values in each iteration
2. $\mathbf{d'(s,u)}$ changes from iteration $\mathbf{i}$ to $\mathbf{i+1}$ only because of the node $\mathbf{v}$ that is added to $\mathbf{S}$ in iteration $\mathbf{i}$.

```
Initialize for each node v, dist(s,v) = d'(s,v) = ∞
Initialize S = ∅, d'(s,s) = 0
for i = 1 to |V| do
    // S contains the i − 1 closest nodes to s,
    //        and the values of d'(s,u) are current
    v be node realizing d'(s,v) = min_{u∈V−S} d'(s,u)
    dist(s,v) = d'(s,v)
    S = S ∪ {v}
    Update d'(s,u) for each u in V − S as follows:
        d'(s,u) = min(d'(s,u), dist(s,v) + ℓ(v,u))
```

Running time: $\mathbf{O(m+n^2)}$ time.

1. $\mathbf{n}$ outer iterations and in each iteration following steps
2. updating $\mathbf{d'(s,u)}$ after $\mathbf{v}$ added takes $\mathbf{O(deg(v))}$ time so total

Improved Algorithm

1. Main work is to compute the $d'(s,u)$ values in each iteration
2. $d'(s,u)$ changes from iteration i to $i+1$ only because of the node v that is added to S in iteration i.

```
Initialize for each node v, dist(s,v) = d'(s,v) = ∞
Initialize S = ∅, d'(s,s) = 0
for i = 1 to |V| do
    // S contains the i − 1 closest nodes to s,
    //        and the values of d'(s,u) are current
    v be node realizing d'(s,v) = min_{u∈V−S} d'(s,u)
    dist(s,v) = d'(s,v)
    S = S ∪ {v}
    Update d'(s,u) for each u in V − S as follows:
        d'(s,u) = min(d'(s,u), dist(s,v) + ℓ(v,u))
```

Running time: $O(m + n^2)$ time.

1. n outer iterations and in each iteration following steps
2. updating $d'(s,u)$ after v added takes $O(\deg(v))$ time so total

Improved Algorithm

```
Initialize for each node v, dist(s,v) = d'(s,v) = ∞
Initialize S = ∅, d'(s,s) = 0
for i = 1 to |V| do
    // S contains the i − 1 closest nodes to s,
    //        and the values of d'(s,u) are current
    v be node realizing d'(s,v) = min_{u∈V−S} d'(s,u)
    dist(s,v) = d'(s,v)
    S = S ∪ {v}
    Update d'(s,u) for each u in V − S as follows:
        d'(s,u) = min(d'(s,u), dist(s,v) + ℓ(v,u))
```

Running time: $O(m + n^2)$ time.

1. n outer iterations and in each iteration following steps
2. updating $d'(s,u)$ after v added takes $O(\deg(v))$ time so total work is $O(m)$ since a node enters S only once
3. Finding v from $d'(s,u)$ values is $O(n)$ time

Dijkstra's Algorithm

1. eliminate $\mathbf{d'(s,u)}$ and let $\mathbf{dist(s,u)}$ maintain it
2. update **dist** values after adding **v** by scanning edges out of **v**

```
Initialize for each node v, dist(s,v) = ∞
Initialize S = {}, dist(s,s) = 0
for i = 1 to |V| do
    Let v be such that dist(s,v) = min_{u∈V−S} dist(s,u)
    S = S ∪ {v}
    for each u in Adj(v) do
        dist(s,u) = min(dist(s,u), dist(s,v) + ℓ(v,u))
```

Priority Queues to maintain **dist** values for faster running time

1. Using heaps and standard priority queues: $O((m+n)\log n)$
2. Using Fibonacci heaps: $O(m + n\log n)$.

Dijkstra's Algorithm

1. eliminate $\mathbf{d'(s,u)}$ and let $\mathbf{dist(s,u)}$ maintain it
2. update **dist** values after adding **v** by scanning edges out of **v**

```
Initialize for each node v, dist(s,v) = ∞
Initialize S = {}, dist(s,s) = 0
for i = 1 to |V| do
    Let v be such that dist(s,v) = min_{u∈V−S} dist(s,u)
    S = S ∪ {v}
    for each u in Adj(v) do
        dist(s,u) = min(dist(s,u), dist(s,v) + ℓ(v,u))
```

Priority Queues to maintain **dist** values for faster running time

1. Using heaps and standard priority queues: $\mathbf{O((m+n)\log n)}$
2. Using Fibonacci heaps: $\mathbf{O(m + n\log n)}$.

Priority Queues

Data structure to store a set $\mathbf{S}$ of $\mathbf{n}$ elements where each element $\mathbf{v} \in \mathbf{S}$ has an associated real/integer key $\mathbf{k(v)}$ such that the following operations:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in $\mathbf{S}$.
3. **extractMin**: Remove $\mathbf{v} \in \mathbf{S}$ with smallest key and return it.
4. **insert**$(\mathbf{v}, \mathbf{k(v)})$: Add new element $\mathbf{v}$ with key $\mathbf{k(v)}$ to $\mathbf{S}$.
5. **delete**$(\mathbf{v})$: Remove element $\mathbf{v}$ from $\mathbf{S}$.
6. **decreaseKey**$(\mathbf{v}, \mathbf{k'(v)})$: *decrease* key of $\mathbf{v}$ from $\mathbf{k(v)}$ (current key) to $\mathbf{k'(v)}$ (new key). Assumption: $\mathbf{k'(v)} \leq \mathbf{k(v)}$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $\mathbf{O(\log n)}$ time.
decreaseKey is implemented via **delete** and **insert**.

Priority Queues

Data structure to store a set $\mathbf{S}$ of $\mathbf{n}$ elements where each element $\mathbf{v} \in \mathbf{S}$ has an associated real/integer key $\mathbf{k(v)}$ such that the following operations:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in $\mathbf{S}$.
3. **extractMin**: Remove $\mathbf{v} \in \mathbf{S}$ with smallest key and return it.
4. **insert**$(\mathbf{v}, \mathbf{k(v)})$: Add new element $\mathbf{v}$ with key $\mathbf{k(v)}$ to $\mathbf{S}$.
5. **delete**$(\mathbf{v})$: Remove element $\mathbf{v}$ from $\mathbf{S}$.
6. **decreaseKey**$(\mathbf{v}, \mathbf{k'(v)})$: *decrease* key of $\mathbf{v}$ from $\mathbf{k(v)}$ (current key) to $\mathbf{k'(v)}$ (new key). Assumption: $\mathbf{k'(v)} \leq \mathbf{k(v)}$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $\mathbf{O(\log n)}$ time.
decreaseKey is implemented via **delete** and **insert**.

Priority Queues

Data structure to store a set $\mathbf{S}$ of $\mathbf{n}$ elements where each element $\mathbf{v} \in \mathbf{S}$ has an associated real/integer key $\mathbf{k(v)}$ such that the following operations:

1. **makePQ**: create an empty queue.
2. **findMin**: find the minimum key in $\mathbf{S}$.
3. **extractMin**: Remove $\mathbf{v} \in \mathbf{S}$ with smallest key and return it.
4. **insert**$(\mathbf{v}, \mathbf{k(v)})$: Add new element $\mathbf{v}$ with key $\mathbf{k(v)}$ to $\mathbf{S}$.
5. **delete**$(\mathbf{v})$: Remove element $\mathbf{v}$ from $\mathbf{S}$.
6. **decreaseKey**$(\mathbf{v}, \mathbf{k'(v)})$: *decrease* key of $\mathbf{v}$ from $\mathbf{k(v)}$ (current key) to $\mathbf{k'(v)}$ (new key). Assumption: $\mathbf{k'(v)} \leq \mathbf{k(v)}$.
7. **meld**: merge two separate priority queues into one.

All operations can be performed in $\mathbf{O(\log n)}$ time.
decreaseKey is implemented via **delete** and **insert**.

Dijkstra's Algorithm using Priority Queues

```
Q ⇐ makePQ()
insert(Q, (s, 0))
for each node u ≠ s do
    insert(Q, (u, ∞))
S ⇐ ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    S = S ∪ {v}
    for each u in Adj(v) do
        decreaseKey(Q, (u, min(dist(s, u), dist(s, v) + ℓ(v, u))))
```

Priority Queue operations:

1. **O(n) insert** operations
2. **O(n) extractMin** operations
3. **O(m) decreaseKey** operations

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $\mathbf{O(\log n)}$ time

Dijkstra's algorithm can be implemented in $\mathbf{O((n+m)\log n)}$ time.

Implementing Priority Queues via Heaps

Using Heaps

Store elements in a heap based on the key value

1. All operations can be done in $\mathbf{O(\log n)}$ time

Dijkstra's algorithm can be implemented in $\mathbf{O((n+m)\log n)}$ time.

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $\mathbf{O(\log n)}$ time
2. **decreaseKey** in $\mathbf{O(1)}$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq \mathbf{n}$ take *together* $\mathbf{O(\ell)}$ time
3. Relaxed Heaps: **decreaseKey** in $\mathbf{O(1)}$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $\mathbf{O(n \log n + m)}$ time. If $\mathbf{m = \Omega(n \log n)}$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $\mathbf{O(\log n)}$ time
2. **decreaseKey** in $\mathbf{O(1)}$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq \mathbf{n}$ take *together* $\mathbf{O(\ell)}$ time
3. Relaxed Heaps: **decreaseKey** in $\mathbf{O(1)}$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $\mathbf{O(n \log n + m)}$ time. If $\mathbf{m = \Omega(n \log n)}$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $\mathbf{O(\log n)}$ time
2. **decreaseKey** in $\mathbf{O(1)}$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq \mathbf{n}$ take *together* $\mathbf{O(\ell)}$ time
3. Relaxed Heaps: **decreaseKey** in $\mathbf{O(1)}$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $\mathbf{O(n \log n + m)}$ time. If $\mathbf{m = \Omega(n \log n)}$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Priority Queues: Fibonacci Heaps/Relaxed Heaps

Fibonacci Heaps

1. **extractMin**, **insert**, **delete**, **meld** in $\mathbf{O(\log n)}$ time
2. **decreaseKey** in $\mathbf{O(1)}$ *amortized* time: ℓ **decreaseKey** operations for $\ell \geq \mathbf{n}$ take *together* $\mathbf{O(\ell)}$ time
3. Relaxed Heaps: **decreaseKey** in $\mathbf{O(1)}$ worst case time but at the expense of **meld** (not necessary for Dijkstra's algorithm)

1. Dijkstra's algorithm can be implemented in $\mathbf{O(n \log n + m)}$ time. If $\mathbf{m = \Omega(n \log n)}$, running time is linear in input size.
2. Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009!)

Shortest Path Tree

Dijkstra's algorithm finds the shortest path distances from s to **V**.
Question: How do we find the paths themselves?

```
Q = makePQ()
insert(Q, (s, 0))
prev(s) ⇐ null
for each node u ≠ s do
    insert(Q, (u, ∞) )
    prev(u) ⇐ null

S = ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    S = S ∪ {v}
    for each u in Adj(v) do
        if (dist(s, v) + ℓ(v, u) < dist(s, u) ) then
            decreaseKey(Q, (u, dist(s, v) + ℓ(v, u)) )
            prev(u) = v
```

Shortest Path Tree

Dijkstra's algorithm finds the shortest path distances from s to **V**.
Question: How do we find the paths themselves?

```
Q = makePQ()
insert(Q, (s, 0))
prev(s) ⇐ null
for each node u ≠ s do
    insert(Q, (u, ∞) )
    prev(u) ⇐ null

S = ∅
for i = 1 to |V| do
    (v, dist(s, v)) = extractMin(Q)
    S = S ∪ {v}
    for each u in Adj(v) do
        if (dist(s, v) + ℓ(v, u) < dist(s, u) ) then
            decreaseKey(Q, (u, dist(s, v) + ℓ(v, u)) )
            prev(u) = v
```

Shortest Path Tree

Lemma

The edge set $(\mathbf{u}, \mathrm{prev}(\mathbf{u}))$ *is the* reverse *of a shortest path tree rooted at* $\mathbf{s}$*. For each* $\mathbf{u}$*, the reverse of the path from* $\mathbf{u}$ *to* $\mathbf{s}$ *in the tree is a shortest path from* $\mathbf{s}$ *to* $\mathbf{u}$*.*

Proof Sketch.

1. The edge set $\{(\mathbf{u}, \mathrm{prev}(\mathbf{u})) \mid \mathbf{u} \in \mathbf{V}\}$ induces a directed in-tree rooted at $\mathbf{s}$ (Why?)
2. Use induction on $|\mathbf{S}|$ to argue that the tree is a shortest path tree for nodes in $\mathbf{V}$.

□

Shortest paths to s

Dijkstra's algorithm gives shortest paths from s to all nodes in V. How do we find shortest paths from all of V to s?

1. In undirected graphs shortest path from s to u is a shortest path from u to s so there is no need to distinguish.
2. In directed graphs, use Dijkstra's algorithm in G^{rev}!

Shortest paths to s

Dijkstra's algorithm gives shortest paths from s to all nodes in V. How do we find shortest paths from all of V to s?

1. In undirected graphs shortest path from s to u is a shortest path from u to s so there is no need to distinguish.
2. In directed graphs, use Dijkstra's algorithm in G^{rev}!

Notes

Notes

Notes

Notes

Shortest Path Algorithms

Lecture 4

January 30, 2014

Part I

Shortest Paths with Negative Length Edges

Single-Source Shortest Paths with Negative Edge Lengths

Single-Source Shortest Path Problems

Input: A *directed* graph $\mathbf{G = (V, E)}$ with arbitrary (including negative) edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

1. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

Single-Source Shortest Paths with Negative Edge Lengths

Single-Source Shortest Path Problems

Input: A *directed* graph $\mathbf{G = (V, E)}$ with arbitrary (including negative) edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

1. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

What are the distances computed by Dijkstra's algorithm?

The distance as computed by Dijkstra algorithm starting from **s**:

(A) $s = 0$, $x = 5$, $y = 1$, $z = 0$.

(B) $s = 0$, $x = 1$, $y = 2$, $z = 5$.

(C) $s = 0$, $x = 5$, $y = 1$, $z = 2$.

(D) IDK.

Negative Length Cycles

Definition

A cycle **C** is a negative length cycle if the sum of the edge lengths of **C** is negative.

Negative Length Cycles

Definition

A cycle **C** is a negative length cycle if the sum of the edge lengths of **C** is negative.

Shortest Paths and Negative Cycles

Given $\mathbf{G = (V, E)}$ with edge lengths and $\mathbf{s, t}$. Suppose

1. $\mathbf{G}$ has a negative length cycle $\mathbf{C}$, and
2. $\mathbf{s}$ can reach $\mathbf{C}$ and $\mathbf{C}$ can reach $\mathbf{t}$.

Question: What is the shortest **distance** from $\mathbf{s}$ to $\mathbf{t}$?

Possible answers: Define shortest distance to be:

1. undefined, that is $-\infty$, OR
2. the length of a shortest **simple** path from $\mathbf{s}$ to $\mathbf{t}$.

Lemma

If there is an efficient algorithm to find a shortest simple $\mathbf{s} \to \mathbf{t}$ path in a graph with negative edge lengths, then there is an efficient algorithm to find the longest *simple $\mathbf{s} \to \mathbf{t}$ path in a graph with positive edge lengths.*

Finding the $\mathbf{s} \to \mathbf{t}$ longest path is difficult. **NP-Hard**!

Shortest Paths and Negative Cycles

Given $\mathbf{G = (V, E)}$ with edge lengths and $\mathbf{s, t}$. Suppose

1. $\mathbf{G}$ has a negative length cycle $\mathbf{C}$, and
2. $\mathbf{s}$ can reach $\mathbf{C}$ and $\mathbf{C}$ can reach $\mathbf{t}$.

Question: What is the shortest **distance** from $\mathbf{s}$ to $\mathbf{t}$?
Possible answers: Define shortest distance to be:

1. undefined, that is $-\infty$, OR
2. the length of a shortest **simple** path from $\mathbf{s}$ to $\mathbf{t}$.

Lemma

If there is an efficient algorithm to find a shortest simple $\mathbf{s} \to \mathbf{t}$ *path in a graph with negative edge lengths, then there is an efficient algorithm to find the* longest *simple* $\mathbf{s} \to \mathbf{t}$ *path in a graph with positive edge lengths.*

Finding the $\mathbf{s} \to \mathbf{t}$ longest path is difficult. **NP-Hard**!

Alterantively: Finding Shortest Walks

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

1. A path is a sequence of *distinct* vertices $\mathbf{v_1, v_2, \ldots, v_k}$ such that $\mathbf{(v_i, v_{i+1}) \in E}$ for $\mathbf{1 \leq i \leq k - 1}$.
2. A walk is a sequence of vertices $\mathbf{v_1, v_2, \ldots, v_k}$ such that $\mathbf{(v_i, v_{i+1}) \in E}$ for $\mathbf{1 \leq i \leq k - 1}$. Vertices are allowed to repeat.

Define $\mathbf{dist(u, v)}$ to be the length of a shortest walk from $\mathbf{u}$ to $\mathbf{v}$.

1. If there is a walk from $\mathbf{u}$ to $\mathbf{v}$ that contains negative length cycle then $\mathbf{dist(u, v) = -\infty}$
2. Else there is a path whose length is equal to the length of a shortest walk and $\mathbf{dist(u, v)}$ is finite

Helpful to think about walks

Shortest Paths with Negative Edge Lengths

Problems

Algorithmic Problems

Input: A directed graph $\mathbf{G = (V, E)}$ with edge lengths (could be negative). For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

Questions:

1. Given nodes $\mathbf{s, t}$, either find a negative length cycle $\mathbf{C}$ that $\mathbf{s}$ can reach or find a shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$, either find a negative length cycle $\mathbf{C}$ that $\mathbf{s}$ can reach or find shortest path distances from $\mathbf{s}$ to all reachable nodes.
3. Check if $\mathbf{G}$ has a negative length cycle or not.

Shortest Paths with Negative Edge Lengths

In Undirected Graphs

Note: With negative lengths, shortest path problems and negative cycle detection in undirected graphs cannot be reduced to directed graphs by bi-directing each undirected edge. Why?

Problem can be solved efficiently in undirected graphs but algorithms are different and more involved than those for directed graphs. Beyond the scope of this class. If interested, ask instructor for references.

Why Negative Lengths?

Several Applications

1. Shortest path problems useful in modeling many situations — in some negative lenths are natural
2. Negative length cycle can be used to find arbitrage opportunities in currency trading
3. Important sub-routine in algorithms for more general problem: minimum-cost flow

Negative cycles

Application to Currency Trading

Currency Trading

Input: $\mathbf{n}$ currencies and for each ordered pair $(\mathbf{a}, \mathbf{b})$ the *exchange rate* for converting one unit of $\mathbf{a}$ into one unit of $\mathbf{b}$.

Questions:

1. Is there an arbitrage opportunity?
2. Given currencies $\mathbf{s}, \mathbf{t}$ what is the best way to convert $\mathbf{s}$ to $\mathbf{t}$ (perhaps via other intermediate currencies)?

Concrete example:

1. $\mathbf{1}$ Chinese Yuan $=$ $\mathbf{0.1116}$ Euro
2. $\mathbf{1}$ Euro $=$ $\mathbf{1.3617}$ US dollar
3. $\mathbf{1}$ US Dollar $=$ $\mathbf{7.1}$ Chinese Yuan.

Thus, if exchanging $\mathbf{1\ \$} \rightarrow$ Yuan $\rightarrow$ Euro $\rightarrow$ $\mathbf{\$}$, we get: $\mathbf{0.1116 * 1.3617 * 7.1 = 1.07896\$}$.

Reducing Currency Trading to Shortest Paths

Observation: If we convert currency $\mathbf{i}$ to $\mathbf{j}$ via intermediate currencies $\mathbf{k_1, k_2, \ldots, k_h}$ then one unit of $\mathbf{i}$ yields $\mathbf{exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)}$ units of $\mathbf{j}$.

Create currency trading *directed* graph $\mathbf{G = (V, E)}$:

1. For each currency $\mathbf{i}$ there is a node $\mathbf{v_i \in V}$
2. $\mathbf{E = V \times V}$: an edge for each pair of currencies
3. edge length $\ell(\mathbf{v_i, v_j}) = -\mathbf{log(exch(i, j))}$ can be negative

Exercise: Verify that

1. There is an arbitrage opportunity if and only if $\mathbf{G}$ has a negative length cycle.
2. The best way to convert currency $\mathbf{i}$ to currency $\mathbf{j}$ is via a shortest path in $\mathbf{G}$ from $\mathbf{i}$ to $\mathbf{j}$. If $\mathbf{d}$ is the distance from $\mathbf{i}$ to $\mathbf{j}$ then one unit of $\mathbf{i}$ can be converted into $\mathbf{2^d}$ units of $\mathbf{j}$.

Reducing Currency Trading to Shortest Paths

Observation: If we convert currency $\mathbf{i}$ to $\mathbf{j}$ via intermediate currencies $\mathbf{k_1, k_2, \ldots, k_h}$ then one unit of $\mathbf{i}$ yields $\mathbf{exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)}$ units of $\mathbf{j}$.

Create currency trading *directed* graph $\mathbf{G = (V, E)}$:

1. For each currency $\mathbf{i}$ there is a node $\mathbf{v_i \in V}$
2. $\mathbf{E = V \times V}$: an edge for each pair of currencies
3. edge length $\ell(\mathbf{v_i, v_j}) = -\log(\mathbf{exch(i, j)})$ can be negative

Exercise: Verify that

1. There is an arbitrage opportunity if and only if $\mathbf{G}$ has a negative length cycle.
2. The best way to convert currency $\mathbf{i}$ to currency $\mathbf{j}$ is via a shortest path in $\mathbf{G}$ from $\mathbf{i}$ to $\mathbf{j}$. If $\mathbf{d}$ is the distance from $\mathbf{i}$ to $\mathbf{j}$ then one unit of $\mathbf{i}$ can be converted into $\mathbf{2^d}$ units of $\mathbf{j}$.

Reducing Currency Trading to Shortest Paths

Observation: If we convert currency $\mathbf{i}$ to $\mathbf{j}$ via intermediate currencies $\mathbf{k_1, k_2, \ldots, k_h}$ then one unit of $\mathbf{i}$ yields $\mathbf{exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)}$ units of $\mathbf{j}$.

Create currency trading *directed* graph $\mathbf{G = (V, E)}$:

1. For each currency $\mathbf{i}$ there is a node $\mathbf{v_i \in V}$
2. $\mathbf{E = V \times V}$: an edge for each pair of currencies
3. edge length $\ell(\mathbf{v_i, v_j}) = -\mathbf{log(exch(i, j))}$ can be negative

Exercise: Verify that

1. There is an arbitrage opportunity if and only if $\mathbf{G}$ has a negative length cycle.
2. The best way to convert currency $\mathbf{i}$ to currency $\mathbf{j}$ is via a shortest path in $\mathbf{G}$ from $\mathbf{i}$ to $\mathbf{j}$. If $\mathbf{d}$ is the distance from $\mathbf{i}$ to $\mathbf{j}$ then one unit of $\mathbf{i}$ can be converted into $\mathbf{2^d}$ units of $\mathbf{j}$.

Reducing Currency Trading to Shortest Paths

Observation: If we convert currency $\mathbf{i}$ to $\mathbf{j}$ via intermediate currencies $\mathbf{k_1, k_2, \ldots, k_h}$ then one unit of $\mathbf{i}$ yields $\mathbf{exch(i, k_1) \times exch(k_1, k_2) \ldots \times exch(k_h, j)}$ units of $\mathbf{j}$.

Create currency trading *directed* graph $\mathbf{G = (V, E)}$:

1. For each currency $\mathbf{i}$ there is a node $\mathbf{v_i \in V}$
2. $\mathbf{E = V \times V}$: an edge for each pair of currencies
3. edge length $\ell(\mathbf{v_i, v_j}) = -\mathbf{log(exch(i, j))}$ can be negative

Exercise: Verify that

1. There is an arbitrage opportunity if and only if $\mathbf{G}$ has a negative length cycle.
2. The best way to convert currency $\mathbf{i}$ to currency $\mathbf{j}$ is via a shortest path in $\mathbf{G}$ from $\mathbf{i}$ to $\mathbf{j}$. If $\mathbf{d}$ is the distance from $\mathbf{i}$ to $\mathbf{j}$ then one unit of $\mathbf{i}$ can be converted into $\mathbf{2^d}$ units of $\mathbf{j}$.

Reducing Currency Trading to Shortest Paths

Math recall - relevant information

1. $\log(\alpha_1 * \alpha_2 * \cdots * \alpha_k) = \log \alpha_1 + \log \alpha_2 + \cdots + \log \alpha_k$.
2. $\log x > 0$ if and only if $x > 1$.

Shortest Paths with Negative Edge Lengths

Problems

Algorithmic Problems

Input: A directed graph $\mathbf{G = (V, E)}$ with edge lengths (could be negative). For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

Questions:

1. Given nodes $\mathbf{s, t}$, either find a negative length cycle $\mathbf{C}$ that $\mathbf{s}$ can reach or find a shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$, either find a negative length cycle $\mathbf{C}$ that $\mathbf{s}$ can reach or find shortest path distances from $\mathbf{s}$ to all reachable nodes.
3. Check if $\mathbf{G}$ has a negative length cycle or not.

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail

Dijkstra's Algorithm and Negative Lengths

With negative length edges, Dijkstra's algorithm can fail

False assumption: Dijkstra's algorithm is based on the assumption that if $s = v_0 \to v_1 \to v_2 \ldots \to v_k$ is a shortest path from s to v_k then $dist(s, v_i) \leq dist(s, v_{i+1})$ for $0 \leq i < k$. Holds true only for non-negative edge lengths.

Shortest Paths with Negative Lengths

Lemma

Let $\mathbf{G}$ *be a directed graph with arbitrary edge lengths. If* $\mathbf{s = v_0 \to v_1 \to v_2 \to \ldots \to v_k}$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v_k}$ *then for* $\mathbf{1 \leq i < k}$*:*

1. $\mathbf{s = v_0 \to v_1 \to v_2 \to \ldots \to v_i}$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v_i}$
2. *False:* $\mathbf{dist(s, v_i) \leq dist(s, v_k)}$ *for* $\mathbf{1 \leq i < k}$*. Holds true only for non-negative edge lengths.*

Cannot explore nodes in increasing order of distance! We need other strategies.

Shortest Paths with Negative Lengths

Lemma

Let $\mathbf{G}$ *be a directed graph with arbitrary edge lengths. If* $\mathbf{s = v_0 \to v_1 \to v_2 \to \ldots \to v_k}$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v_k}$ *then for* $\mathbf{1 \leq i < k}$*:*

1. $\mathbf{s = v_0 \to v_1 \to v_2 \to \ldots \to v_i}$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v_i}$
2. *False:* $\mathbf{dist(s, v_i) \leq dist(s, v_k)}$ *for* $\mathbf{1 \leq i < k}$*. Holds true only for non-negative edge lengths.*

Cannot explore nodes in increasing order of distance! We need other strategies.

Shortest Paths with Negative Lengths

Lemma

Let $\mathbf{G}$ *be a directed graph with arbitrary edge lengths. If* $\mathbf{s = v_0 \to v_1 \to v_2 \to \ldots \to v_k}$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v_k}$ *then for* $\mathbf{1 \leq i < k}$*:*

1. $\mathbf{s = v_0 \to v_1 \to v_2 \to \ldots \to v_i}$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v_i}$
2. *False:* $\mathbf{dist(s, v_i) \leq dist(s, v_k)}$ *for* $\mathbf{1 \leq i < k}$*. Holds true only for non-negative edge lengths.*

Cannot explore nodes in increasing order of distance! We need other strategies.

A Generic Shortest Path Algorithm

1. Start with distance estimate for each node $\mathbf{d(s,u)}$ set to ∞
2. Maintain the invariant that there is an $\mathbf{s} \rightarrow \mathbf{u}$ path of length $\mathbf{d(s,u)}$. Hence $\mathbf{d(s,u)} \geq \mathbf{dist(s,u)}$.
3. Iteratively refine $\mathbf{d(s,\cdot)}$ values until they reach the correct value $\mathbf{dist(s,\cdot)}$ values at termination

Must hold that...

$$\mathbf{d(s,v)} \leq \mathbf{d(s,u)} + \ell(\mathbf{u,v})$$

A Generic Shortest Path Algorithm

Question: How do we make progress?

Definition

Given distance estimates $d(s, u)$ for each $u \in V$, an edge $e = (u, v)$ is **tense** if $d(s, v) > d(s, u) + \ell(u, v)$.

Relax$(e = (u, v))$
  if $(d(s, v) > d(s, u) + \ell(u, v))$ **then**
    $d(s, v) = d(s, u) + \ell(u, v)$

A Generic Shortest Path Algorithm

Question: How do we make progress?

Definition

Given distance estimates $\mathbf{d(s,u)}$ for each $\mathbf{u \in V}$, an edge $\mathbf{e = (u,v)}$ is **tense** if $\mathbf{d(s,v) > d(s,u) + \ell(u,v)}$.

Relax$(\mathbf{e = (u,v)})$
  if $(\mathbf{d(s,v) > d(s,u) + \ell(u,v)})$ **then**
    $\mathbf{d(s,v) = d(s,u) + \ell(u,v)}$

A Generic Shortest Path Algorithm

Question: How do we make progress?

Definition

Given distance estimates $\mathbf{d(s,u)}$ for each $\mathbf{u \in V}$, an edge $\mathbf{e = (u,v)}$ is **tense** if $\mathbf{d(s,v) > d(s,u) + \ell(u,v)}$.

Relax$(\mathbf{e = (u,v)})$
  if $(\mathbf{d(s,v) > d(s,u) + \ell(u,v)})$ **then**
    $\mathbf{d(s,v) = d(s,u) + \ell(u,v)}$

A Generic Shortest Path Algorithm

Invariant

If a vertex $\mathbf{u}$ has value $\mathbf{d(s,u)}$ associated with it, then there is a $\mathbf{s} \rightsquigarrow \mathbf{u}$ walk of length $\mathbf{d(s,u)}$.

Proposition

Relax *maintains the invariant on* $\mathbf{d(s,u)}$ *values.*

Proof.

Indeed, if **Relax**($\mathbf{(u,v)}$) changed the value of $\mathbf{d(s,v)}$, then there is a walk to $\mathbf{u}$ of length $\mathbf{d(s,u)}$ (by invariant), and there is a walk of length $\mathbf{d(s,u)} + \ell\mathbf{(u,v)}$ to $\mathbf{v}$ through $\mathbf{u}$, which is the new value of $\mathbf{d(s,v)}$. $\square$

A Generic Shortest Path Algorithm

$\mathbf{d(s,s) = 0}$
for `each node` $\mathbf{u \neq s}$ **do**
    $\mathbf{d(s,u) = \infty}$

while `there is a tense edge` **do**
    `Pick a tense edge` $\mathbf{e}$
    Relax$\mathbf{(e)}$

`Output` $\mathbf{d(s,u)}$ `values`

Technical assumption: If $\mathbf{e = (u,v)}$ is an edge and $\mathbf{d(s,u) = d(s,v) = \infty}$ then edge is not tense.

Key property of generic algorithm

If estimate distance from source too large, then $\exists$ tense edge...

Lemma

If $\exists$ *walk* $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$\implies \ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1})$.

$$\begin{aligned}
\implies & d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \\
& \leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k) \\
& = \ell(\pi) < d(s, v_k).
\end{aligned}$$

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$

$\implies$ edge (v_{k-1}, v_k) is tense. $\square$

Key property of generic algorithm

If estimate distance from source too large, then $\exists$ tense edge...

Lemma

If $\exists$ *walk* $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$\implies \ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1})$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k)$

$\leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k)$

$= \ell(\pi) < d(s, v_k)$.

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$

$\implies$ edge (v_{k-1}, v_k) is tense. $\square$

Key property of generic algorithm

If estimate distance from source too large, then $\exists$ tense edge...

Lemma

If $\exists$ *walk* $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$$\begin{aligned}
&\implies \ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1}). \\
&\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \\
&\qquad \leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k) \\
&\qquad = \ell(\pi) < d(s, v_k). \\
&\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k) \\
&\implies \text{edge}\,(v_{k-1}, v_k) \text{ is tense.}
\end{aligned}$$

□

Key property of generic algorithm

If estimate distance from source too large, then $\exists$ tense edge...

Lemma

If $\exists$ *walk* $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$$\begin{aligned}
&\implies \ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1}). \\
&\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \\
&\qquad \leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k) \\
&\qquad = \ell(\pi) < d(s, v_k). \\
&\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k) \\
&\implies \text{edge}\,(v_{k-1}, v_k) \text{ is tense.}
\end{aligned}$$

□

Key property of generic algorithm

If estimate distance from source too large, then $\exists$ tense edge...

Lemma

If $\exists$ *walk* $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$\implies \ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1})$.

$$\begin{aligned} \implies & d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \\ & \leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k) \\ & = \ell(\pi) < d(s, v_k). \end{aligned}$$

$\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k)$

$\implies$ edge (v_{k-1}, v_k) is tense. $\square$

$$\ell(\pi) < d(s, v_k)$$

$$\ell(\pi') \geq d(s, v_{k-1})$$

$$\ell(\pi) = \ell(\pi') + \ell(v_{k-1}, v_k) < d(s, v_k)$$

Key property of generic algorithm

If estimate distance from source too large, then $\exists$ tense edge...

Lemma

If $\exists$ *walk* $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_{i+1}) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$$\begin{aligned}
&\implies \ell(v_1 \to \cdots v_{k-1}) \geq d(s, v_{k-1}). \\
&\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) \\
&\qquad \leq \ell(v_1 \to \cdots v_{k-1}) + \ell(v_{k-1}, v_k) \\
&\qquad = \ell(\pi) < d(s, v_k). \\
&\implies d(s, v_{k-1}) + \ell(v_{k-1}, v_k) < d(s, v_k) \\
&\implies \text{edge}\,(v_{k-1}, v_k) \text{ is tense.}
\end{aligned}$$

$\square$

Key property of generic algorithm

If estimate distance from source too large, then $\exists$ tense edge...

Lemma

If $\exists$ *walk* $\pi \equiv s = v_1 \to v_2 \to \cdots \to v_k = u$ *such that*

$$\ell(\pi) = \sum_{i=1}^{k-1} \ell(v_i, v_j) < d(s, u)$$

Then, there exists a tense edge in G.

Proof.

Assume π: shortest in number of edges (with property).

$$\cdots$$

$\implies$ edge (v_{k-1}, v_k) is tense. $\square$

$\implies$ If for any vertex u: $d(s, u) > \mathrm{dist}(s, u)$ then the algorithm will continue working!

Properties of the generic algorithm

Proposition

If $\mathbf{u}$ *is reachable from* $\mathbf{s}$ *and algorithm terminates then* $\mathbf{d(s,u)} \neq \infty$.

Proof.

Corollary of key property. □

Proposition

If $\mathbf{u}$ *is not reachable from* $\mathbf{s}$ *then* $\mathbf{d(s,u)}$ *remains at* ∞ *throughout the algorithm.*

Properties of the generic algorithm

Proposition

If $\mathbf{u}$ *is reachable from* $\mathbf{s}$ *and algorithm terminates then* $\mathbf{d(s,u)} \neq \infty$.

Proof.

Corollary of key property. □

Proposition

If $\mathbf{u}$ *is not reachable from* $\mathbf{s}$ *then* $\mathbf{d(s,u)}$ *remains at* ∞ *throughout the algorithm.*

Properties of the generic algorithm

Proposition

If a negative length cycle **C** *is reachable by* **s** *then there is always a tense edge and hence the algorithm never terminates.*

Properties of the generic algorithm

Proposition

If a negative length cycle $\mathbf{C}$ *is reachable by* $\mathbf{s}$ *then there is always a tense edge and hence the algorithm never terminates.*

Proof.

Also corollary of key property. If algorithm terminates then for each node $\mathbf{u} \in \mathbf{C}$, $\mathbf{d(s, u)}$ is a finite value, however there is a walk of length $< \mathbf{d(s, u)}$ — in fact for any finite value. $\square$

$$d(s,v_1) \le d(s,v_0) + \ell(e_1)$$

$$d(s,v_2) \le d(s,v_1) + \ell(e_2)$$

$$\vdots$$

$$d(s,v_0) \le d(s,v_k) + \ell(e_k)$$

$$0 \le \sum \ell(e_i)$$

Properties of the generic algorithm

Proposition

If a negative length cycle $\mathbf{C}$ *is reachable by* $\mathbf{s}$ *then there is always a tense edge and hence the algorithm never terminates.*

A more direct proof.

Let $\mathbf{C} = \mathbf{v_0, v_1, \ldots, v_k}$ be a negative length cycle reachable from $\mathbf{s}$. Suppose algorithm terminates. By previous proposition, $\mathbf{d(s, v_i)} < \infty$ for all $\mathbf{i}$. Since no edge of $\mathbf{C}$ was tense, for $\mathbf{i = 1, 2, \ldots, k}$ we have $\mathbf{d(s, v_i)} \leq \mathbf{d(s, v_{i-1})} + \ell\mathbf{(v_{i-1}, v_i)}$ and $\mathbf{d(s, v_0)} \leq \mathbf{d(s, v_k)} + \ell\mathbf{(v_k, v_0)}$. Adding up all the inequalities we obtain that length of $\mathbf{C}$ is non-negative! $\square$

Corollary

Alg. terminates implies no negative length cycle $\mathbf{C}$ *reachable from* $\mathbf{s}$.

Properties of the generic algorithm

Lemma

If the algorithm terminates then $\mathbf{d(s,u) = dist(s,u)}$ *for each node* $\mathbf{u}$ *(and* $\mathbf{s}$ *cannot reach a negative cycle).*

Proof follows from key property.

Question: How do we ensure termination?

Properties of the generic algorithm

Lemma

If the algorithm terminates then $\mathbf{d(s,u) = dist(s,u)}$ *for each node* $\mathbf{u}$ *(and* $\mathbf{s}$ *cannot reach a negative cycle).*

Proof follows from key property.

Question: How do we ensure termination?

Generic Algorithm: Ordering Relax operations

```
d(s,s) = 0
for each node u ≠ s do
    d(s,u) = ∞

While there is a tense edge do
    Pick a tense edge e
    Relax(e)

Output d(s,u) values for u ∈ V(G)
```

Question: How do we pick edges to relax?

Observation: Suppose $s \to v_1 \to \ldots \to v_k$ is a shortest path.

If $\mathbf{Relax}(s, v_1)$, $\mathbf{Relax}(v_1, v_2)$, $\ldots$, $\mathbf{Relax}(v_{k-1}, v_k)$ are done in *order* then $d(s, v_k) = \mathbf{dist}(s, v_k)$!

Generic Algorithm: Ordering Relax operations

```
d(s,s) = 0
for each node u ≠ s do
    d(s,u) = ∞

While there is a tense edge do
    Pick a tense edge e
    Relax(e)

Output d(s,u) values for u ∈ V(G)
```

Question: How do we pick edges to relax?

Observation: Suppose $s \to v_1 \to \ldots \to v_k$ is a shortest path.

If $\text{Relax}(s, v_1)$, $\text{Relax}(v_1, v_2)$, $\ldots$, $\text{Relax}(v_{k-1}, v_k)$ are done in *order* then $d(s, v_k) = \text{dist}(s, v_k)$!

Ordering Relax operations

1. **Observation:** Suppose $\mathbf{s} \rightarrow \mathbf{v_1} \rightarrow \ldots \rightarrow \mathbf{v_k}$ is a shortest path.

   If $\mathbf{Relax}(\mathbf{s}, \mathbf{v_1})$, $\mathbf{Relax}(\mathbf{v_1}, \mathbf{v_2})$, $\ldots$, $\mathbf{Relax}(\mathbf{v_{k-1}}, \mathbf{v_k})$ are done in *order* then $\mathbf{d(s, v_k) = dist(s, v_k)}$! (Why?)

2. We don't know the shortest paths so how do we know the order to do the Relax operations?

Ordering Relax operations

1. **Observation:** Suppose $\mathbf{s} \rightarrow \mathbf{v_1} \rightarrow \ldots \rightarrow \mathbf{v_k}$ is a shortest path.

   If $\mathbf{Relax}(\mathbf{s}, \mathbf{v_1})$, $\mathbf{Relax}(\mathbf{v_1}, \mathbf{v_2})$, $\ldots$, $\mathbf{Relax}(\mathbf{v_{k-1}}, \mathbf{v_k})$ are done in *order* then $\mathbf{d(s, v_k) = dist(s, v_k)}$! (Why?)
2. We don't know the shortest paths so how do we know the order to do the Relax operations?

Ordering Relax operations

1. We don't know the shortest paths so how do we know the order to do the Relax operations?
2. We don't!
   1. Relax *all* edges (even those not tense) in some arbitrary order
   2. Iterate $|V| - 1$ times
   3. First iteration will do $\mathbf{Relax}(s, v_1)$ (and other edges), second round $\mathbf{Relax}(v_1, v_2)$ and in iteration k we do $\mathbf{Relax}(v_{k-1}, v_k)$.

Ordering Relax operations

1. We don't know the shortest paths so how do we know the order to do the Relax operations?
2. We don't!
   1. Relax *all* edges (even those not tense) in some arbitrary order
   2. Iterate $|\mathbf{V}| - \mathbf{1}$ times
   3. First iteration will do **Relax**$(\mathbf{s}, \mathbf{v_1})$ (and other edges), second round **Relax**$(\mathbf{v_1}, \mathbf{v_2})$ and in iteration $\mathbf{k}$ we do **Relax**$(\mathbf{v_{k-1}}, \mathbf{v_k})$.

Bellman-Ford Algorithm

```
for each u ∈ V do
        d(s, u) ← ∞
d(s, s) ← 0

for i = 1 to |V| − 1 do
        for each edge e = (u, v) do
            Relax(e)

for each u ∈ V do
        dist(s, u) ← d(s, u)
```

Bellman-Ford Algorithm: Scanning Edges

One possible way to scan edges in each iteration.

```
Q is an empty queue
for each u ∈ V do
        d(s,u) = ∞
        enq(Q, u)
d(s,s) = 0

for i = 1 to |V| − 1 do
        for j = 1 to |V| do
            u = deq(Q)
            for each edge e in Adj(u) do
                Relax(e)
            enq(Q, u)

for each u ∈ V do
        dist(s,u) = d(s,u)
```

Example

Step 0

Example

Step 0

$\Longrightarrow$

Step 1

Example

Step 1

$\Longrightarrow$

Step 2

Example

Step 2

$\Longrightarrow$

Step 3

Example

Step 3

$\Longrightarrow$

Step 4

Example

Step 4

$\Longrightarrow$

Step 5

Example

Step 5

$\Longrightarrow$

Step 6

Example

Step 6

$\Longrightarrow$

Step 7

Example

Step 7

$\Longrightarrow$

Step 8

Example

Step 8

$\Longrightarrow$

Step 9

Example

Step 9

$\Longrightarrow$

Step 10

Example

Step 10

$\Longrightarrow$

Step 11

Example

Step 11

$\Longrightarrow$

Step 12

We are done! No edge is tense.

Example

Figure : One iteration of Bellman-Ford that Relaxes all edges by processing nodes in the order **s, a, b, c, d, e, f**. Red edges indicate the prev pointers (in reverse)

Example

Figure : **6** iterations of Bellman-Ford starting with the first one from previous slide. No changes in **5**th iteration and **6**th iteration.

Correctness of the Bellman-Ford Algorithm

Lemma

After $\mathbf{i}$ *iterations of the Bellman-Ford algorithm, for each* $\mathbf{v} \in \mathbf{V}$, $\mathbf{d(s, v)}$ *is length of shortest walk from* $\mathbf{s}$ *to* $\mathbf{v}$ *with at most* $\mathbf{i}$ *hops.*

Proof.

By induction on $\mathbf{i}$.

1. Base case: $\mathbf{i = 0}$. $\mathbf{d(s, s) = 0}$ and $\mathbf{d(s, v)} = \infty$ for all $\mathbf{v \neq s}$.
2. Induction Step: Let $\mathbf{s \rightarrow v_1 \ldots \rightarrow v_j \rightarrow v}$ be a shortest walk from $\mathbf{s}$ to $\mathbf{v}$ with at most $\mathbf{i}$ hops. Let α be its length.
   1. If $\mathbf{j < i - 1}$, then by induction, after $\mathbf{i - 1}$ iterations $\mathbf{d(s, v)} \leq \alpha$ (Why?)
   2. If $\mathbf{j = i - 1}$, then by induction, after $\mathbf{i - 1}$ iterations $\mathbf{d(s, v_{i-1})}$ is equal to length of walk $\mathbf{s \rightarrow v_1 \ldots \rightarrow v_{i-1}}$. (Why?)
   3. In iteration $\mathbf{i}$, **Relax**$(\mathbf{v_{i-1}, v_i})$ will ensure that $\mathbf{d(s, v_i)} \leq \alpha$.
   4. Note: Relax does not increase $\mathbf{d(s, u)}$ value.

Correctness of Bellman-Ford Algorithm

Corollary

After $|\mathbf{V}| - \mathbf{1}$ *iterations of Bellman-Ford,* $\mathbf{d(s,u) = dist(s,u)}$ *for any node* $\mathbf{u}$ *such that* $\mathbf{dist(s,u)} > -\infty$.

Proof.

If $\mathbf{dist(s,u)} > -\infty$ then there exists a shortest walk from $\mathbf{s}$ to $\mathbf{u}$ with finite number of hops. In particular it will be a path (since any cycle on the walk cannot be negative, otherwise $\mathbf{dist(s,u)} = -\infty$) and hence has at most $\mathbf{n} - \mathbf{1}$ hops. □

Note: If there is a negative cycle $\mathbf{C}$ such that $\mathbf{s}$ can reach $\mathbf{C}$ then we do not know whether $\mathbf{d(s,u) = dist(s,u)}$ or whether $\mathbf{dist(s,u)} = \infty$!

Question: How do we know whether there is a negative cycle $\mathbf{C}$ reachable from $\mathbf{s}$?

Correctness of Bellman-Ford Algorithm

Corollary

After $|\mathbf{V}| - \mathbf{1}$ *iterations of Bellman-Ford,* $\mathbf{d(s,u) = dist(s,u)}$ *for any node* $\mathbf{u}$ *such that* $\mathbf{dist(s,u)} > -\infty$.

Proof.

If $\mathbf{dist(s,u)} > -\infty$ then there exists a shortest walk from $\mathbf{s}$ to $\mathbf{u}$ with finite number of hops. In particular it will be a path (since any cycle on the walk cannot be negative, otherwise $\mathbf{dist(s,u)} = -\infty$) and hence has at most $\mathbf{n} - \mathbf{1}$ hops. □

Note: If there is a negative cycle $\mathbf{C}$ such that $\mathbf{s}$ can reach $\mathbf{C}$ then we do not know whether $\mathbf{d(s,u) = dist(s,u)}$ or whether $\mathbf{dist(s,u)} = \infty$!

Question: How do we know whether there is a negative cycle $\mathbf{C}$ reachable from $\mathbf{s}$?

Correctness of Bellman-Ford Algorithm

Corollary

After $|\mathbf{V}| - \mathbf{1}$ *iterations of Bellman-Ford,* $\mathbf{d(s,u) = dist(s,u)}$ *for any node* $\mathbf{u}$ *such that* $\mathbf{dist(s,u)} > -\infty$.

Proof.

If $\mathbf{dist(s,u)} > -\infty$ then there exists a shortest walk from $\mathbf{s}$ to $\mathbf{u}$ with finite number of hops. In particular it will be a path (since any cycle on the walk cannot be negative, otherwise $\mathbf{dist(s,u)} = -\infty$) and hence has at most $\mathbf{n} - \mathbf{1}$ hops. □

Note: If there is a negative cycle $\mathbf{C}$ such that $\mathbf{s}$ can reach $\mathbf{C}$ then we do not know whether $\mathbf{d(s,u) = dist(s,u)}$ or whether $\mathbf{dist(s,u)} = \infty$!

Question: How do we know whether there is a negative cycle $\mathbf{C}$ reachable from $\mathbf{s}$?

Bellman-Ford to detect Negative Cycles

```
for each u ∈ V do
        d(s,u) = ∞
d(s,s) = 0

for i = 1 to |V| − 1 do
        for each edge e = (u,v) do
            Relax(e)

for each edge e = (u,v) do
        if e = (u,v) is tense then
            Stop and output that s can reach
                    a negative length cycle

Output for each u ∈ V:  d(s,u)
```

Correctness

Lemma

$\mathbf{G}$ *has a negative cycle reachable from* $\mathbf{s}$ *if and only if there is a tense edge* $\mathbf{e}$ *after* $|\mathbf{V}| - \mathbf{1}$ *iterations of Bellman-Ford.*

Proof Sketch.

$\mathbf{G}$ has no negative length cycle reachable from $\mathbf{s}$ implies that all nodes $\mathbf{u}$ reachable from $\mathbf{s}$ have $\mathbf{dist(s, u)} > -\infty$. Therefore $\mathbf{d(s, u) = dist(s, u)}$ after the $|\mathbf{V}| - \mathbf{1}$ iterations. Therefore, there cannot be any tense edges left.

If there is a negative cycle $\mathbf{C}$ then there is a tense edge after $|\mathbf{V}| - \mathbf{1}$ (in fact any number of) iterations. Recall key property of the generic shortest path algorithm. □

Correctness

Lemma

$\mathbf{G}$ *has a negative cycle reachable from* $\mathbf{s}$ *if and only if there is a tense edge* $\mathbf{e}$ *after* $|\mathbf{V}| - \mathbf{1}$ *iterations of Bellman-Ford.*

Proof Sketch.

$\mathbf{G}$ has no negative length cycle reachable from $\mathbf{s}$ implies that all nodes $\mathbf{u}$ reachable from $\mathbf{s}$ have $\mathbf{dist(s, u)} > -\infty$. Therefore $\mathbf{d(s, u) = dist(s, u)}$ after the $|\mathbf{V}| - \mathbf{1}$ iterations. Therefore, there cannot be any tense edges left.

If there is a negative cycle $\mathbf{C}$ then there is a tense edge after $|\mathbf{V}| - \mathbf{1}$ (in fact any number of) iterations. Recall key property of the generic shortest path algorithm. □

Finding the Paths and a Shortest Path Tree

```
for each u ∈ V do
        d(s,u) = ∞
        prev(u) = null
d(s,s) = 0
for i = 1 to |V| − 1 do
        for each edge e = (u,v) do
            Relax(e)
if there is a tense edge e then
        Output that s can reach a negative cycle C
else
        for each u ∈ V do
            output d(s,u)

Relax(e = (u,v))
        if (d(s,v) > d(s,u) + ℓ(u,v)) then
            d(s,v) = d(s,u) + ℓ(u,v)
            prev(v) = u
```

Note: **prev** pointers induce a shortest path tree.

Negative Cycle Detection

Negative Cycle Detection

Given directed graph **G** with arbitrary edge lengths, does it have a negative length cycle?

1. Bellman-Ford checks whether there is a negative cycle **C** that is reachable from a specific vertex **s**. There may negative cycles not reachable from **s**.
2. Run Bellman-Ford $|V|$ times, once from each node **u**?

Negative Cycle Detection

Negative Cycle Detection

Given directed graph **G** with arbitrary edge lengths, does it have a negative length cycle?

1. Bellman-Ford checks whether there is a negative cycle **C** that is reachable from a specific vertex **s**. There may negative cycles not reachable from **s**.
2. Run Bellman-Ford |**V**| times, once from each node **u**?

Negative Cycle Detection

1. Add a new node $\mathbf{s'}$ and connect it to all nodes of $\mathbf{G}$ with zero length edges. Bellman-Ford from $\mathbf{s'}$ will fill find a negative length cycle if there is one. Exercise: why does this work?
2. Negative cycle detection can be done with one Bellman-Ford invocation.

Running time for Bellman-Ford

1. Input graph $\mathbf{G = (V, E)}$ with $\mathbf{m} = |\mathbf{E}|$ and $\mathbf{n} = |\mathbf{V}|$.
2. $\mathbf{n}$ outer iterations and $\mathbf{m}$ **Relax**() operations in each iteration. Each **Relax**() operation is $\mathbf{O(1)}$ time.
3. Total running time: $\mathbf{O(mn)}$.

Note: Algorithm can be safely stopped if no tense edge in some iteration. Why?

Running time for Bellman-Ford

1. Input graph $\mathbf{G = (V, E)}$ with $\mathbf{m} = |\mathbf{E}|$ and $\mathbf{n} = |\mathbf{V}|$.
2. $\mathbf{n}$ outer iterations and $\mathbf{m}$ **Relax**() operations in each iteration. Each **Relax**() operation is $\mathbf{O(1)}$ time.
3. Total running time: $\mathbf{O(mn)}$.

Note: Algorithm can be safely stopped if no tense edge in some iteration. Why?

Dijkstra as an instantiation of the generic algorithm

The Dijkstra algorithm can be stated as the generic algorithm as:

(A) Relax all tense edges.

(B) Always relax the edge (u, v), such that $d(s, u)$ is minimal.

(C) Pick u minimizing $d(s, u)$ such that u was not visited yet. Mark as visited, and relax all its outgoing edges.

(D) Pick an unvisited u. Mark as visited, and relax all its outgoing edges.

Dijkstra's Algorithm with **Relax**()

```
for each node u ≠ s do
    d(s, u) = ∞
d(s, s) = 0
S = ∅
while (S ≠ V) do
    Let v be node in V − S with min d value
    S = S ∪ {v}
    for each edge e in Adj(v) do
        Relax(e)
```

Part II

Shortest Paths in DAGs

Shortest Paths in a DAG

Single-Source Shortest Path Problems

Input A directed acyclic graph $\mathbf{G = (V, E)}$ with arbitrary (including negative) edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u}, \mathbf{v})$ is its length.

1. Given nodes $\mathbf{s}, \mathbf{t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

Simplification of algorithms for DAGs

1. No cycles and hence no negative length cycles! Hence can find shortest paths even for negative length edges
2. Can order nodes using topological sort

Shortest Paths in a DAG

Single-Source Shortest Path Problems

Input A directed acyclic graph $\mathbf{G = (V, E)}$ with arbitrary (including negative) edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u}, \mathbf{v})$ is its length.

1. Given nodes $\mathbf{s}, \mathbf{t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

Simplification of algorithms for DAGs

1. No cycles and hence no negative length cycles! Hence can find shortest paths even for negative length edges
2. Can order nodes using topological sort

Algorithm for DAGs

1. Want to find shortest paths from s. Ignore nodes not reachable from s.
2. Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of G

Observation:

1. shortest path from s to v_i cannot use any node from $v_{i+1}, \ldots, v_n$
2. can find shortest paths in topological sort order.

Algorithm for DAGs

1. Want to find shortest paths from s. Ignore nodes not reachable from s.
2. Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of G

Observation:

1. shortest path from s to v_i cannot use any node from $v_{i+1}, \ldots, v_n$
2. can find shortest paths in topological sort order.

Algorithm for DAGs

for $i = 1$ `to` n **do**
    $d(s, v_i) = \infty$
$d(s, s) = 0$

for $i = 1$ `to` $n - 1$ **do**
    for `each edge` e `in` $\mathrm{Adj}(v_i)$ **do**
        Relax(e)

return $d(s, \cdot)$ `values computed`

Correctness: induction on i and observation in previous slide.
Running time: $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find *longest* paths in a DAG.

Takeaway Points

1. Shortest paths with potentially negative length edges arise in a variety of applications. Longest simple path problem is difficult (no known efficient algorithm and **NP-Hard**). We restrict attention to shortest walks and they are well defined only if there are no negative length cycles reachable from the source.
2. A generic shortest path algorithm starts with distance estimates to the source and iteratively refines them by considering edges one at a time. The algorithm is guaranteed to terminate with correct distances if there are no negative length cycle. If a negative length cycle is reachable from the source it is guaranteed not to terminate.
3. Dijkstra's algorithm can also be thought of as an instantiation of the generic algorithm.

Points continued

1. Bellman-Ford algorithm is an instantiation of the generic algorithm that in each iteration relaxes all the edges. It recognizes negative length cycles if there is a tense edges in the **n**th iteration. For a vertex **u** with a shortest path to the source with **i** edges the algorithm has the correct distance after **i** iterations. Running time of Bellman-Ford algorithm is **O(nm)**.
2. Bellman-Ford can be adapted to find a negative length cycle in the graph by adding a new vertex.
3. If we have a DAG then it has no negative length cycle and hence shortest paths exists even with negative lengths. One can compute single-source shortest paths in a DAG in linear time. This implies that one can also compute longest paths in a DAG in linear time.

Notes

Notes

Notes

Notes

Reductions, Recursion and Divide and Conquer

Lecture 5

February 4, 2014

Part I

Reductions and Recursion

Reduction

Reducing problem **A** to problem **B**:

1. Algorithm for **A** uses algorithm for **B** as a *black box*

Reduction

Reducing problem **A** to problem **B**:

1. Algorithm for **A** uses algorithm for **B** as a *black box*

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Reduction

Reducing problem **A** to problem **B**:

1. Algorithm for **A** uses algorithm for **B** as a *black box*

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Q: How do you hunt a red elephant?

A: Hold his trunk shut until he turns blue, and then shoot him with the blue elephant gun.

Reduction

Reducing problem **A** to problem **B**:

1. Algorithm for **A** uses algorithm for **B** as a *black box*

Q: How do you hunt a blue elephant?

A: With a blue elephant gun.

Q: How do you hunt a red elephant?

A: Hold his trunk shut until he turns blue, and then shoot him with the blue elephant gun.

Q: How do you shoot a white elephant?

A: Embarrass it till it becomes red. Now use your algorithm for hunting red elephants.

UNIQUENESS: Distinct Elements Problem

Problem Given an array **A** of **n** integers, are there any *duplicates* in **A**?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
return NO
```

Running time: $O(n^2)$

UNIQUENESS: Distinct Elements Problem

Problem Given an array **A** of **n** integers, are there any *duplicates* in **A**?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
return NO
```

Running time: $O(n^2)$

UNIQUENESS: Distinct Elements Problem

Problem Given an array **A** of **n** integers, are there any *duplicates* in **A**?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
return NO
```

Running time: $O(n^2)$

UNIQUENESS: Distinct Elements Problem

Problem Given an array **A** of **n** integers, are there any *duplicates* in **A**?

Naive algorithm:

```
for i = 1 to n − 1 do
    for j = i + 1 to n do
        if (A[i] = A[j])
            return YES
return NO
```

Running time: $O(n^2)$

Reduction to Sorting

```
Sort A
for i = 1 to n − 1 do
    if (A[i] = A[i + 1]) then
        return YES
return NO
```

Running time: $O(n)$ plus time to sort an array of n numbers

Important point: algorithm uses sorting as a black box

Reduction to Sorting

```
Sort A
for i = 1 to n − 1 do
    if (A[i] = A[i + 1]) then
        return YES
return NO
```

Running time: $O(n)$ plus time to sort an array of n numbers

Important point: algorithm uses sorting as a black box

Two sides of Reductions

Suppose problem **A** reduces to problem **B**

1. Positive direction: Algorithm for **B** implies an algorithm for **A**
2. Negative direction: Suppose there is no "efficient" algorithm for **A** then it implies no efficient algorithm for **B** (technical condition for reduction time necessary for this)

Example: Distinct Elements reduces to Sorting in $\mathbf{O(n)}$ time

1. An $\mathbf{O(n \log n)}$ time algorithm for Sorting implies an $\mathbf{O(n \log n)}$ time algorithm for Distinct Elements problem.
2. If there is *no* $\mathbf{o(n \log n)}$ time algorithm for Distinct Elements problem then there is *no* $\mathbf{o(n \log n)}$ time algorithm for Sorting.

Two sides of Reductions

Suppose problem **A** reduces to problem **B**

1. Positive direction: Algorithm for **B** implies an algorithm for **A**
2. Negative direction: Suppose there is no "efficient" algorithm for **A** then it implies no efficient algorithm for **B** (technical condition for reduction time necessary for this)

Example: Distinct Elements reduces to Sorting in $\mathbf{O(n)}$ time

1. An $\mathbf{O(n \log n)}$ time algorithm for Sorting implies an $\mathbf{O(n \log n)}$ time algorithm for Distinct Elements problem.
2. If there is *no* $\mathbf{o(n \log n)}$ time algorithm for Distinct Elements problem then there is *no* $\mathbf{o(n \log n)}$ time algorithm for Sorting.

Maximum Independent Set in a Graph

Definition

Given undirected graph $\mathbf{G = (V, E)}$ a subset of nodes $\mathbf{S \subseteq V}$ is an independent set (also called a stable set) if for there are no edges between nodes in $\mathbf{S}$. That is, if $\mathbf{u, v \in S}$ then $\mathbf{(u, v) \notin E}$.

Some independent sets in graph above: $\{A, C\}$, $\{A, C, F\}$

Maximum Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$

Goal Find maximum sized independent set in $\mathbf{G}$

Maximum Weight Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$, weights $\mathbf{w(v) \geq 0}$ for $\mathbf{v \in V}$

Goal Find maximum weight independent set in $\mathbf{G}$

Weighted Interval Scheduling

Input A set of jobs with start times, finish times and *weights* (or profits).

Goal Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!

Weighted Interval Scheduling

Input A set of jobs with start times, finish times and *weights* (or profits).

Goal Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!

Reduction from Interval Scheduling to MIS

Question: Can you reduce Weighted Interval Scheduling to Max Weight Independent Set Problem?

Weighted Circular Arc Scheduling

Input A set of arcs on a circle, each arc has a *weight* (or profit).

Goal Find a maximum weight subset of arcs that do not overlap.

Reductions

Question: Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

Question: Can you reduce Weighted Circular Arc Scheduling to Weighted Interval Scheduling? Yes!

```
cur-max = 0
for each arc C do
    Remove C and all arcs overlapping with C
    w_C = wt of opt. solution in resulting Interval problem
    w_C = w_C + wt(C)
    cur-max = max{cur-max, w_C}

return cur-max
```

n calls to the sub-routine for interval scheduling

Reductions

Question: Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

Question: Can you reduce Weighted Circular Arc Scheduling to Weighted Interval Scheduling? Yes!

```
cur-max = 0
for each arc C do
    Remove C and all arcs overlapping with C
    w_C = wt of opt. solution in resulting Interval problem
    w_C = w_C + wt(C)
    cur-max = max{cur-max, w_C}

return cur-max
```

n calls to the sub-routine for interval scheduling

Reductions

Question: Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

Question: Can you reduce Weighted Circular Arc Scheduling to Weighted Interval Scheduling? Yes!

```
cur-max = 0
for each arc C do
    Remove C and all arcs overlapping with C
    w_C = wt of opt. solution in resulting Interval problem
    w_C = w_C + wt(C)
    cur-max = max{cur-max, w_C}

return cur-max
```

n calls to the sub-routine for interval scheduling

Reductions

Question: Can you reduce Weighted Interval Scheduling to Weighted Circular Arc Scheduling?

Question: Can you reduce Weighted Circular Arc Scheduling to Weighted Interval Scheduling? Yes!

```
cur-max = 0
for each arc C do
    Remove C and all arcs overlapping with C
    w_C = wt of opt. solution in resulting Interval problem
    w_C = w_C + wt(C)
    cur-max = max{cur-max, w_C}

return cur-max
```

n calls to the sub-routine for interval scheduling

Illustration

Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

1. reduce problem to a *smaller* instance of *itself*
2. self-reduction

1. Problem instance of size $\mathbf{n}$ is reduced to *one or more* instances of size $\mathbf{n} - \mathbf{1}$ or less.
2. For termination, problem instances of small size are solved by some other method as *base cases*

Recursion

Reduction: reduce one problem to another

Recursion: a special case of reduction

1. reduce problem to a *smaller* instance of *itself*
2. self-reduction

1. Problem instance of size $\mathbf{n}$ is reduced to *one or more* instances of size $\mathbf{n-1}$ or less.
2. For termination, problem instances of small size are solved by some other method as *base cases*

Recursion

1. Recursion is a very powerful and fundamental technique
2. Basis for several other methods
   1. Divide and conquer
   2. Dynamic programming
   3. Enumeration and branch and bound etc
   4. Some classes of greedy algorithms
3. Makes proof of correctness easy (via induction)
4. Recurrences arise in analysis

Selection Sort

Sort a given array **A[1..n]** of integers.

Recursive version of Selection sort.

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.  Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

$T(n)$: time for **SelectSort** on an n element array.

$T(n) = T(n-1) + n$ for $n > 1$ and $T(1) = 1$ for $n = 1$

$T(n) = \Theta(n^2)$.

Selection Sort

Sort a given array **A[1..n]** of integers.

Recursive version of Selection sort.

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.  Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

$\mathbf{T(n)}$: time for **SelectSort** on an $\mathbf{n}$ element array.

$\mathbf{T(n) = T(n-1) + n}$ for $\mathbf{n > 1}$ and $\mathbf{T(1) = 1}$ for $\mathbf{n = 1}$

$\mathbf{T(n) = \Theta(n^2)}$.

Selection Sort

Sort a given array **A[1..n]** of integers.

Recursive version of Selection sort.

```
SelectSort(A[1..n]):
    if n = 1 return
    Find smallest number in A.  Let A[i] be smallest number
    Swap A[1] and A[i]
    SelectSort(A[2..n])
```

$\mathbf{T(n)}$: time for **SelectSort** on an $\mathbf{n}$ element array.

$\mathbf{T(n) = T(n-1) + n}$ for $\mathbf{n > 1}$ and $\mathbf{T(1) = 1}$ for $\mathbf{n = 1}$

$\mathbf{T(n) = \Theta(n^2)}$.

Tower of Hanoi

The Tower of Hanoi puzzle

Move stack of **n** disks from peg **0** to peg **2**, one disk at a time.
Rule: cannot put a larger disk on a smaller disk.
Question: what is a strategy and how many moves does it take?

Tower of Hanoi via Recursion

The Tower of Hanoi algorithm; ignore everything but the bottom disk

Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

$T(n)$: time to move n disks via recursive strategy

$$T(n) = 2T(n-1) + 1 \qquad n > 1 \qquad \text{and } T(1) = 1$$

Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

T(n): time to move **n** disks via recursive strategy

$$T(n) = 2T(n-1) + 1 \qquad n > 1 \qquad \text{and } T(1) = 1$$

Recursive Algorithm

```
Hanoi(n, src, dest, tmp):
    if (n > 0) then
        Hanoi(n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi(n − 1, tmp, dest, src)
```

$\mathbf{T(n)}$: time to move $\mathbf{n}$ disks via recursive strategy

$$\mathbf{T(n) = 2T(n-1) + 1} \qquad \mathbf{n > 1} \qquad \text{and } \mathbf{T(1) = 1}$$

Analysis

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2^2 T(n-2) + 2 + 1 \\
&= \ldots \\
&= 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 1 \\
&= \ldots \\
&= 2^{n-1} T(1) + 2^{n-2} + \ldots + 1 \\
&= 2^{n-1} + 2^{n-2} + \ldots + 1 \\
&= (2^n - 1)/(2-1) = 2^n - 1
\end{aligned}
$$

Non-Recursive Algorithms for Tower of Hanoi

Pegs numbered $\mathbf{0, 1, 2}$

Non-recursive Algorithm 1:

1. Always move smallest disk forward if $\mathbf{n}$ is even, backward if $\mathbf{n}$ is odd.
2. Never move the same disk twice in a row.
3. Done when no legal move.

Non-recursive Algorithm 2:

1. Let $\rho(\mathbf{n})$ be the smallest integer $\mathbf{k}$ such that $\mathbf{n}/\mathbf{2}^{\mathbf{k}}$ is *not* an integer. Example: $\rho(\mathbf{40}) = \mathbf{4}$, $\rho(\mathbf{18}) = \mathbf{2}$.
2. In step $\mathbf{i}$ move disk $\rho(\mathbf{i})$ forward if $\mathbf{n} - \mathbf{i}$ is even and backward if $\mathbf{n} - \mathbf{i}$ is odd.

Moves are exactly same as those of recursive algorithm. Prove by induction.

Non-Recursive Algorithms for Tower of Hanoi

Pegs numbered $\mathbf{0, 1, 2}$

Non-recursive Algorithm 1:

1. Always move smallest disk forward if $\mathbf{n}$ is even, backward if $\mathbf{n}$ is odd.
2. Never move the same disk twice in a row.
3. Done when no legal move.

Non-recursive Algorithm 2:

1. Let $\rho(\mathbf{n})$ be the smallest integer $\mathbf{k}$ such that $\mathbf{n}/\mathbf{2}^{\mathbf{k}}$ is *not* an integer. Example: $\rho(\mathbf{40}) = \mathbf{4}$, $\rho(\mathbf{18}) = \mathbf{2}$.
2. In step $\mathbf{i}$ move disk $\rho(\mathbf{i})$ forward if $\mathbf{n} - \mathbf{i}$ is even and backward if $\mathbf{n} - \mathbf{i}$ is odd.

Moves are exactly same as those of recursive algorithm. Prove by induction.

Non-Recursive Algorithms for Tower of Hanoi

Pegs numbered $\mathbf{0, 1, 2}$

Non-recursive Algorithm 1:

1. Always move smallest disk forward if $\mathbf{n}$ is even, backward if $\mathbf{n}$ is odd.
2. Never move the same disk twice in a row.
3. Done when no legal move.

Non-recursive Algorithm 2:

1. Let $\rho(\mathbf{n})$ be the smallest integer $\mathbf{k}$ such that $\mathbf{n}/\mathbf{2}^{\mathbf{k}}$ is *not* an integer. Example: $\rho(\mathbf{40}) = \mathbf{4}$, $\rho(\mathbf{18}) = \mathbf{2}$.
2. In step $\mathbf{i}$ move disk $\rho(\mathbf{i})$ forward if $\mathbf{n} - \mathbf{i}$ is even and backward if $\mathbf{n} - \mathbf{i}$ is odd.

Moves are exactly same as those of recursive algorithm. Prove by induction.

Part II

Divide and Conquer

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

1. Break problem instance into smaller instances - divide step
2. Recursively solve problem on smaller instances
3. Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
2. There are many examples of this particular type of recursion that it deserves its own treatment.

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

1. Break problem instance into smaller instances - divide step
2. Recursively solve problem on smaller instances
3. Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
2. There are many examples of this particular type of recursion that it deserves its own treatment.

Divide and Conquer Paradigm

Divide and Conquer is a common and useful type of recursion

Approach

1. Break problem instance into smaller instances - divide step
2. Recursively solve problem on smaller instances
3. Combine solutions to smaller instances to obtain a solution to the original instance - conquer step

Question: Why is this not plain recursion?

1. In divide and conquer, each smaller instance is typically at least a constant factor smaller than the original instance which leads to efficient running times.
2. There are many examples of this particular type of recursion that it deserves its own treatment.

Sorting

Input Given an array of **n** elements

Goal Rearrange them in ascending order

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

A L G O R I T H M S

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

$$\mathbf{A\ L\ G\ O\ R\ I\ T\ H\ M\ S}$$

2. Divide into subarrays $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$, where $\mathbf{m = \lfloor n/2 \rfloor}$

$$\mathbf{A\ L\ G\ O\ R \quad I\ T\ H\ M\ S}$$

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

   A L G O R I T H M S

2. Divide into subarrays $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$, where $\mathbf{m = \lfloor n/2 \rfloor}$

   A L G O R I T H M S

3. Recursively **MergeSort** $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$

   A G L O R H I M S T

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

   A L G O R I T H M S

2. Divide into subarrays $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$, where $\mathbf{m = \lfloor n/2 \rfloor}$

   A L G O R I T H M S

3. Recursively **MergeSort** $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$

   A G L O R H I M S T

4. Merge the sorted arrays

   A G H I L M O R S T

Merge Sort [von Neumann]

MergeSort

1. Input: Array $\mathbf{A[1 \ldots n]}$

   A L G O R I T H M S

2. Divide into subarrays $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$, where $\mathbf{m = \lfloor n/2 \rfloor}$

   A L G O R I T H M S

3. Recursively **MergeSort** $\mathbf{A[1 \ldots m]}$ and $\mathbf{A[m+1 \ldots n]}$

   A G L O R H I M S T

4. Merge the sorted arrays

   A G H I L M O R S T

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T

A G H I L M O R S T

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T

A G H I L M O R S T

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T

A G H I L M O R S T

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T

A G H I L M O R S T

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T
A G H I L M O R S T

Merging Sorted Arrays

1. Use a new array **C** to store the merged array
2. Scan **A** and **B** from left-to-right, storing elements in **C** in order

A G L O R H I M S T
A G H I L M O R S T

3. Merge two arrays using only constantly more extra space (in-place merge sort): doable but complicated and typically impractical.

Running Time

$\mathbf{T(n)}$: time for merge sort to sort an $\mathbf{n}$ element array

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

What do we want as a solution to the recurrence?

Almost always only an *asymptotically* tight bound. That is we want to know $f(n)$ such that $T(n) = \Theta(f(n))$.

1. $T(n) = O(f(n))$ - upper bound
2. $T(n) = \Omega(f(n))$ - lower bound

Running Time

$\mathbf{T(n)}$: time for merge sort to sort an $\mathbf{n}$ element array

$$\mathbf{T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn}$$

What do we want as a solution to the recurrence?

Almost always only an *asymptotically* tight bound. That is we want to know $\mathbf{f(n)}$ such that $\mathbf{T(n) = \Theta(f(n))}$.

1. $\mathbf{T(n) = O(f(n))}$ - upper bound
2. $\mathbf{T(n) = \Omega(f(n))}$ - lower bound

Running Time

$\mathbf{T(n)}$: time for merge sort to sort an $\mathbf{n}$ element array

$$\mathbf{T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn}$$

What do we want as a solution to the recurrence?

Almost always only an *asymptotically* tight bound. That is we want to know $\mathbf{f(n)}$ such that $\mathbf{T(n) = \Theta(f(n))}$.

1. $\mathbf{T(n) = O(f(n))}$ - upper bound
2. $\mathbf{T(n) = \Omega(f(n))}$ - lower bound

Solving Recurrences: Some Techniques

1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus
2. Expand the recurrence and spot a pattern and use simple math
3. Recursion tree method — imagine the computation as a tree
4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds

Albert Einstein: "Everything should be made as simple as possible, but not simpler."

Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Solving Recurrences: Some Techniques

1. Know some basic math: geometric series, logarithms, exponentials, elementary calculus
2. Expand the recurrence and spot a pattern and use simple math
3. Recursion tree method — imagine the computation as a tree
4. Guess and verify — useful for proving upper and lower bounds even if not tight bounds

Albert Einstein: "Everything should be made as simple as possible, but not simpler."

Know where to be loose in analysis and where to be tight. Comes with practice, practice, practice!

Recursion Trees

MergeSort: **n** is a power of **2**

1. Unroll the recurrence. $\mathbf{T(n) = 2T(n/2) + cn}$

Recursion Trees

MergeSort: **n** is a power of **2**

1. Unroll the recurrence. $\mathbf{T(n) = 2T(n/2) + cn}$

2. Identify a pattern. At the **i**th level total work is **cn**.

Recursion Trees

MergeSort: **n** is a power of **2**

1. Unroll the recurrence. $\mathbf{T(n) = 2T(n/2) + cn}$

2. Identify a pattern. At the **i**th level total work is **cn**.

Recursion Trees

MergeSort: **n** is a power of **2**

1. Unroll the recurrence. $\mathbf{T(n) = 2T(n/2) + cn}$

2. Identify a pattern. At the **i**th level total work is **cn**.
3. Sum over all levels. The number of levels is $\mathbf{\log n}$. So total is $\mathbf{cn \log n = O(n \log n)}$.

Recursion Trees

MergeSort: **n** is a power of **2**

1. Unroll the recurrence. $\mathbf{T(n) = 2T(n/2) + cn}$

2. Identify a pattern. At the **i**th level total work is **cn**.
3. Sum over all levels. The number of levels is $\mathbf{\log n}$. So total is $\mathbf{cn \log n = O(n \log n)}$.

Recursion Trees

An illustrated example...

Recursion Trees

An illustrated example...

Work in each node

Recursion Trees

An illustrated example...

Work in each node

Recursion Trees

An illustrated example...

$$\log n \left\{ \begin{array}{c} cn \\ \hline \frac{cn}{2} + \frac{cn}{2} \\ \hline \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} \\ \hline \vdots \\ \hline \end{array} \right. \begin{array}{l} = cn \\ = cn \\ = cn \\ \vdots \\ = cn \end{array}$$

Recursion Trees

An illustrated example...

$$\log n \left\{ \begin{array}{l} cn \\ \hline \frac{cn}{2} + \frac{cn}{2} \\ \hline \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} + \frac{cn}{4} \\ \hline \vdots \\ \hline \end{array} \right. \begin{array}{l} = cn \\ + \\ = cn \\ + \\ = cn \\ \vdots \\ = cn \end{array}$$

$$= cn \log n = O(n \log n)$$

MergeSort Analysis

When n is not a power of 2

1. When n is not a power of 2, the running time of **MergeSort** is expressed as

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

MergeSort Analysis

When n is not a power of 2

1. When n is not a power of 2, the running time of **MergeSort** is expressed as

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

2. $n_1 = 2^{k-1} < n \leq 2^k = n_2$ (n_1, n_2 powers of 2).

MergeSort Analysis

When n is not a power of 2

1. When n is not a power of 2, the running time of **MergeSort** is expressed as

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

2. $n_1 = 2^{k-1} < n \leq 2^k = n_2$ (n_1, n_2 powers of 2).
3. $T(n_1) < T(n) \leq T(n_2)$ (Why?).

MergeSort Analysis

When n is not a power of 2

1. When n is not a power of 2, the running time of **MergeSort** is expressed as

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

2. $n_1 = 2^{k-1} < n \leq 2^k = n_2$ (n_1, n_2 powers of 2).
3. $T(n_1) < T(n) \leq T(n_2)$ (Why?).
4. $T(n) = \Theta(n \log n)$ since $n/2 \leq n_1 < n \leq n_2 \leq 2n$.

Recursion Trees

MergeSort: **n** is not a power of **2**

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

Observation: For any number x, $\lfloor x/2 \rfloor + \lceil x/2 \rceil = x$.

MergeSort Analysis

When n is not a power of 2: Guess and Verify

If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
Verify? proof by induction!

Induction Hypothesis: $T(n) \leq 2cn \log n$ for all $n \geq 1$
Base Case: $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
Induction Step Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

MergeSort Analysis

When n is not a power of 2: Guess and Verify

If n is power of 2 we saw that $T(n) = \Theta(n \log n)$.
Can *guess* that $T(n) = \Theta(n \log n)$ for all n.
Verify? proof by induction!

Induction Hypothesis: $T(n) \leq 2cn \log n$ for all $n \geq 1$
Base Case: $n = 1$. $T(1) = 0$ since no need to do any work and $2cn \log n = 0$ for $n = 1$.
Induction Step Assume $T(k) \leq 2ck \log k$ for all $k < n$ and prove it for $k = n$.

Induction Step

We have

$$
\begin{aligned}
T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn \\
&\leq 2c\lfloor n/2 \rfloor \log \lfloor n/2 \rfloor + 2c\lceil n/2 \rceil \log \lceil n/2 \rceil + cn \quad \text{(by induc} \\
&\leq 2c\lfloor n/2 \rfloor \log \lceil n/2 \rceil + 2c\lceil n/2 \rceil \log \lceil n/2 \rceil + cn \\
&\leq 2c(\lfloor n/2 \rfloor + \lceil n/2 \rceil) \log \lceil n/2 \rceil + cn \\
&\leq 2cn \log \lceil n/2 \rceil + cn \\
&\leq 2cn \log(2n/3) + cn \quad (\text{since } \lceil n/2 \rceil \leq 2n/3 \text{ for all } n \geq 2) \\
&\leq 2cn \log n + cn(1 - 2\log 3/2) \\
&\leq 2cn \log n + cn(\log 2 - \log 9/4) \\
&\leq 2cn \log n
\end{aligned}
$$

Guess and Verify

The math worked out like magic!
Why was $\mathbf{2cn \log n}$ chosen instead of say $\mathbf{4cn \log n}$?

1. Do not know upfront what constant to choose.
2. Instead assume that $\mathbf{T(n) \leq \alpha cn \log n}$ for some constant α. α will be fixed later.
3. Need to prove that for α large enough the algebra succeeds.
4. In our case... need α such that $\alpha \log 3/2 > 1$.
5. Typically, do the algebra with α and then show that it works... ... if α is chosen to be sufficiently large constant.

How do we know which function to guess?
We don't so we try several "reasonable" functions. With practice and experience we get better at guessing the right function.

Guess and Verify

The math worked out like magic!
Why was $\mathbf{2cn \log n}$ chosen instead of say $\mathbf{4cn \log n}$?

1. Do not know upfront what constant to choose.
2. Instead assume that $\mathbf{T(n) \leq \alpha cn \log n}$ for some constant α. α will be fixed later.
3. Need to prove that for α large enough the algebra succeeds.
4. In our case... need α such that $\alpha \log 3/2 > 1$.
5. Typically, do the algebra with α and then show that it works... ... if α is chosen to be sufficiently large constant.

How do we know which function to guess?
We don't so we try several "reasonable" functions. With practice and experience we get better at guessing the right function.

Guess and Verify

What happens if the guess is wrong?

1. Guessed that the solution to the **MergeSort** recurrence is $\mathbf{T(n) = O(n)}$.
2. Try to prove by induction that $\mathbf{T(n)} \leq \alpha \mathbf{cn}$ for some const' α.
   Induction Step: attempt

$$\begin{aligned} \mathbf{T(n)} &= \mathbf{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn} \\ &\leq \alpha \mathbf{c\lfloor n/2 \rfloor} + \alpha \mathbf{c\lceil n/2 \rceil + cn} \\ &\leq \alpha \mathbf{cn + cn} \\ &\leq \mathbf{(}\alpha \mathbf{+ 1)cn} \end{aligned}$$

   But need to show that $\mathbf{T(n)} \leq \alpha \mathbf{cn}$!
3. So guess does not work for any constant α. Suggests that our guess is incorrect.

Selection Sort vs Merge Sort

1. Selection Sort spends $\mathbf{O(n)}$ work to reduce problem from $\mathbf{n}$ to $\mathbf{n-1}$ leading to $\mathbf{O(n^2)}$ running time.
2. Merge Sort spends $\mathbf{O(n)}$ time *after* reducing problem to two instances of size $\mathbf{n/2}$ each. Running time is $\mathbf{O(n \log n)}$

Question: Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $\mathbf{k}$ arrays of size $\mathbf{n/k}$ each?

Selection Sort vs Merge Sort

1. Selection Sort spends $\mathbf{O(n)}$ work to reduce problem from $\mathbf{n}$ to $\mathbf{n-1}$ leading to $\mathbf{O(n^2)}$ running time.
2. Merge Sort spends $\mathbf{O(n)}$ time *after* reducing problem to two instances of size $\mathbf{n/2}$ each. Running time is $\mathbf{O(n \log n)}$

Question: Merge Sort splits into 2 (roughly) equal sized arrays. Can we do better by splitting into more than 2 arrays? Say $\mathbf{k}$ arrays of size $\mathbf{n/k}$ each?

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is **O(n)**
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is **O(n)**
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Time Analysis

1. Let $\mathbf{k}$ be the rank of the chosen pivot. Then, $\mathbf{T(n) = T(k-1) + T(n-k) + O(n)}$

Time Analysis

1. Let $\mathbf{k}$ be the rank of the chosen pivot. Then, $\mathbf{T(n) = T(k-1) + T(n-k) + O(n)}$
2. If $\mathbf{k = \lceil n/2 \rceil}$ then $\mathbf{T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)}$. Then, $\mathbf{T(n) = O(n \log n)}$.

Time Analysis

1. Let $\mathbf{k}$ be the rank of the chosen pivot. Then, $\mathbf{T(n) = T(k-1) + T(n-k) + O(n)}$
2. If $\mathbf{k = \lceil n/2 \rceil}$ then $\mathbf{T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)}$. Then, $\mathbf{T(n) = O(n \log n)}$.
    1. Theoretically, median can be found in linear time.

Time Analysis

1. Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
   1. Theoretically, median can be found in linear time.
3. Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

Part III

Fast Multiplication

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: $\Theta(n)$
2. Number of partial products: $\Theta(n)$
3. Addition of partial products: $\Theta(n^2)$
4. Total time: $\Theta(n^2)$

A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $(a + bi)$ and $(c + di)$

$$(a + bi)(c + di) = ac - bd + (ad + bc)i$$

How many multiplications do we need?

Only 3! If we do extra additions and subtractions.
Compute $ac, bd, (a + b)(c + d)$. Then
$(ad + bc) = (a + b)(c + d) - ac - bd$

A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $\mathbf{(a + bi)}$ and $\mathbf{(c + di)}$

$$\mathbf{(a + bi)(c + di) = ac - bd + (ad + bc)i}$$

How many multiplications do we need?

Only 3! If we do extra additions and subtractions.
Compute $\mathbf{ac, bd, (a + b)(c + d)}$. Then
$\mathbf{(ad + bc) = (a + b)(c + d) - ac - bd}$

A Trick of Gauss

Carl Friedrich Gauss: 1777–1855 "Prince of Mathematicians"

Observation: Multiply two complex numbers: $\mathbf{(a + bi)}$ and $\mathbf{(c + di)}$

$$\mathbf{(a + bi)(c + di) = ac - bd + (ad + bc)i}$$

How many multiplications do we need?

Only 3! If we do extra additions and subtractions.
Compute $\mathbf{ac, bd, (a + b)(c + d)}$. Then
$\mathbf{(ad + bc) = (a + b)(c + d) - ac - bd}$

Divide and Conquer

Assume n is a power of 2 for simplicity and numbers are in decimal.

1. $x = x_{n-1}x_{n-2}\ldots x_0$ and $y = y_{n-1}y_{n-2}\ldots y_0$
2. $x = 10^{n/2}x_L + x_R$ where $x_L = x_{n-1}\ldots x_{n/2}$ and $x_R = x_{n/2-1}\ldots x_0$
3. $y = 10^{n/2}y_L + y_R$ where $y_L = y_{n-1}\ldots y_{n/2}$ and $y_R = y_{n/2-1}\ldots y_0$

Therefore

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Example

$$
\begin{aligned}
1234 \times 5678 &= (100 \times 12 + 34) \times (100 \times 56 + 78) \\
&= 10000 \times 12 \times 56 \\
&\quad +100 \times (12 \times 78 + 34 \times 56) \\
&\quad +34 \times 78
\end{aligned}
$$

Time Analysis

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

4 recursive multiplications of number of size $\mathbf{n/2}$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Time Analysis

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R)$$
$$= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

4 recursive multiplications of number of size $\mathbf{n/2}$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$n + 2n + 4n + \cdots + 2^{l-1}n = n\frac{(2^l - 1)}{2-1}$$

$$\mathbf{T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)}$$

$\mathbf{T(n) = \Theta(n^2)}$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Time Analysis

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

4 recursive multiplications of number of size $\mathbf{n/2}$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$T(n) = \Theta(n^2)$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Time Analysis

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

4 recursive multiplications of number of size $\mathbf{n/2}$ each plus 4 additions and left shifts (adding enough 0's to the right)

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1)$$

$\mathbf{T(n) = \Theta(n^2)}$. No better than grade school multiplication!

Can we invoke Gauss's trick here?

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

Improving the Running Time

$$\begin{aligned} xy &= (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) \\ &= 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

Gauss trick: $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

Recursively compute only $x_L y_L, x_R y_R, (x_L + x_R)(y_L + y_R)$.

Time Analysis

Running time is given by

$$T(n) = 3T(n/2) + O(n) \qquad T(1) = O(1)$$

which means $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

State of the Art

Schönhage-Strassen 1971: $\mathbf{O(n \log n \log \log n)}$ time using Fast-Fourier-Transform (FFT)

Martin Fürer 2007: $\mathbf{O(n \log n 2^{O(\log^* n)})}$ time

Conjecture

There is an $\mathbf{O(n \log n)}$ time algorithm.

Analyzing the Recurrences

1. Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
2. Saving a multiplication: $T(n) = 3T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^{1+\log 1.5})$

Use recursion tree method:

1. In both cases, depth of recursion $L = \log n$.
2. Work at depth i is $4^i n/2^i$ and $3^i n/2^i$ respectively: number of children at depth i times the work at each child
3. Total work is therefore $n \sum_{i=0}^{L} 2^i$ and $n \sum_{i=0}^{L} (3/2)^i$ respectively.

Analyzing the Recurrences

1. Basic divide and conquer: $T(n) = 4T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^2)$.
2. Saving a multiplication: $T(n) = 3T(n/2) + O(n)$, $T(1) = 1$. **Claim:** $T(n) = \Theta(n^{1+\log 1.5})$

Use recursion tree method:

1. In both cases, depth of recursion $L = \log n$.
2. Work at depth i is $4^i n/2^i$ and $3^i n/2^i$ respectively: number of children at depth i times the work at each child
3. Total work is therefore $n \sum_{i=0}^{L} 2^i$ and $n \sum_{i=0}^{L} (3/2)^i$ respectively.

Recursion tree analysis

$$n + 3\frac{n}{2} + \left(\frac{3}{2}\right)^2 n + \left(\frac{3}{2}\right)^3 n + \cdots \left(\frac{3}{2}\right)^{L-1} n$$

$$n\left(1 + \frac{3}{2} + \cdots\right) = n \cdot \frac{\left(\frac{3}{2}\right)^L - 1}{3/2 - 1} = \left(\frac{3}{2}\right)^{\log_2 n}$$

Notes

$$n \cdot \left(\frac{3}{2}\right)^{\log_2 n} = n \cdot 2^{\left(\log_2 \frac{3}{2}\right) \log n}$$

$$= n \cdot n^{\log_2 \frac{3}{2}}$$

$$= n^{1 + \log_2 \frac{3}{2}}$$

Notes

Notes

Notes

Recurrences, Closest Pair and Selection

Lecture 6

February 6, 2014

Part I

Recurrences

Solving Recurrences

Two general methods:

1. Recursion tree method: need to do sums
   1. elementary methods, geometric series
   2. integration
2. Guess and Verify
   1. guessing involves intuition, experience and trial & error
   2. verification is via induction

Recurrence: Example I

1. Consider $\mathbf{T(n) = 2T(n/2) + n/\log n}$. ith level has 2^i nodes, and problem size at each node is $n/2^i$ and hence work at each node is $\frac{n}{2^i} / \log \frac{n}{2^i}$.

Total work

$$= \frac{n}{\log n} + 2\left(\frac{n}{2\log\frac{n}{2}}\right) + 4\left(\frac{n}{4\log\frac{n}{4}}\right) + \cdots 2^L\left(\frac{n}{2^L\log\frac{n}{2^L}}\right)$$

$$= n\left(\frac{1}{\log n} + \frac{1}{\log n - 1} + \frac{1}{\log n - 2} + \cdots 1\right) = n H_{\log n} = \Theta(n \log\log n)$$

Recurrence: Example I

1. Consider $\mathbf{T(n) = 2T(n/2) + n/\log n}$.
2. Construct recursion tree, and observe pattern. $\mathbf{i}$th level has $\mathbf{2^i}$ nodes, and problem size at each node is $\mathbf{n/2^i}$ and hence work at each node is $\frac{\mathbf{n}}{\mathbf{2^i}} / \mathbf{\log} \frac{\mathbf{n}}{\mathbf{2^i}}$.

Recurrence: Example I

1. Consider $\mathbf{T(n) = 2T(n/2) + n/\log n}$.
2. Construct recursion tree, and observe pattern. $\mathbf{i}$th level has $\mathbf{2^i}$ nodes, and problem size at each node is $\mathbf{n/2^i}$ and hence work at each node is $\mathbf{\frac{n}{2^i}/\log \frac{n}{2^i}}$.

Recurrence: Example I

1. Consider $T(n) = 2T(n/2) + n/\log n$.
2. Construct recursion tree, and observe pattern. ith level has 2^i nodes, and problem size at each node is $n/2^i$ and hence work at each node is $\frac{n}{2^i}/\log\frac{n}{2^i}$.
3. Summing over all levels

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log n - 1} 2^i \left[\frac{(n/2^i)}{\log(n/2^i)}\right] \\
&= \sum_{i=0}^{\log n - 1} \frac{n}{\log n - i} \\
&= n \sum_{j=1}^{\log n} \frac{1}{j} = nH_{\log n} = \Theta(n \log\log n)
\end{aligned}
$$

Recurrence: Example II

1. Consider $\mathbf{T(n) = T(\sqrt{n}) + 1}$
2. What is the depth of recursion?

   $\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \ldots, O(1)$.

depth of recursion?

Size of problem at level $i = n^{\frac{1}{2^i}}$

L = depth of recursion

$n^{2^{-L}} = c$ ← base case

$\Rightarrow 2^L = \frac{\log n}{\log c} \Rightarrow L = \Theta(\log \log n)$

3. Number of levels: $n^{2^{-L}} = 2$ means $\mathbf{L = \log \log n}$.
4. Number of children at each level is $\mathbf{1}$, work at each node is $\mathbf{1}$
5. Thus, $\mathbf{T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n)}$.

Recurrence: Example II

1. Consider $T(n) = T(\sqrt{n}) + 1$
2. What is the depth of recursion?
   $\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \ldots, O(1)$.
3. Number of levels: $n^{2^{-L}} = 2$ means $L = \log \log n$.
4. Number of children at each level is 1, work at each node is 1
5. Thus, $T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n)$.

Recurrence: Example II

1. Consider $T(n) = T(\sqrt{n}) + 1$
2. What is the depth of recursion?
   $\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \ldots, O(1)$.
3. Number of levels: $n^{2^{-L}} = 2$ means $L = \log \log n$.
4. Number of children at each level is 1, work at each node is 1
5. Thus, $T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n)$.

Recurrence: Example II

1. Consider $T(n) = T(\sqrt{n}) + 1$
2. What is the depth of recursion?
   $\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \ldots, O(1)$.
3. Number of levels: $n^{2^{-L}} = 2$ means $L = \log \log n$.
4. Number of children at each level is 1, work at each node is 1
5. Thus, $T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n)$.

Recurrence: Example II

1. Consider $T(n) = T(\sqrt{n}) + 1$
2. What is the depth of recursion?
   $\sqrt{n}, \sqrt{\sqrt{n}}, \sqrt{\sqrt{\sqrt{n}}}, \ldots, O(1)$.
3. Number of levels: $n^{2^{-L}} = 2$ means $L = \log \log n$.
4. Number of children at each level is 1, work at each node is 1
5. Thus, $T(n) = \sum_{i=0}^{L} 1 = \Theta(L) = \Theta(\log \log n)$.

Recurrence: Example III

1. Consider $\mathbf{T(n) = \sqrt{n}T(\sqrt{n}) + n}$.
2. Using recursion trees: number of levels $\mathbf{L = \log\log n}$
3. Work at each level? Root is $\mathbf{n}$, next level is $\mathbf{\sqrt{n} \times \sqrt{n} = n}$.
   Can check that each level is $\mathbf{n}$.
4. Thus, $\mathbf{T(n) = \Theta(n \log\log n)}$

$n + \sqrt{n}\,n + n + \cdots$

$n \log\log n$

Recurrence: Example III

1. Consider $\mathbf{T(n) = \sqrt{n}T(\sqrt{n}) + n}$.
2. Using recursion trees: number of levels $\mathbf{L = \log \log n}$
3. Work at each level? Root is $\mathbf{n}$, next level is $\sqrt{\mathbf{n}} \times \sqrt{\mathbf{n}} = \mathbf{n}$. Can check that each level is $\mathbf{n}$.
4. Thus, $\mathbf{T(n) = \Theta(n \log \log n)}$

Recurrence: Example III

1. Consider $\mathbf{T(n) = \sqrt{n}T(\sqrt{n}) + n}$.
2. Using recursion trees: number of levels $\mathbf{L = \log \log n}$
3. Work at each level? Root is $\mathbf{n}$, next level is $\mathbf{\sqrt{n} \times \sqrt{n} = n}$. Can check that each level is $\mathbf{n}$.
4. Thus, $\mathbf{T(n) = \Theta(n \log \log n)}$

Recurrence: Example III

1. Consider $\mathbf{T(n) = \sqrt{n}T(\sqrt{n}) + n}$.
2. Using recursion trees: number of levels $\mathbf{L = \log \log n}$
3. Work at each level? Root is $\mathbf{n}$, next level is $\mathbf{\sqrt{n} \times \sqrt{n} = n}$. Can check that each level is $\mathbf{n}$.
4. Thus, $\mathbf{T(n) = \Theta(n \log \log n)}$

Recurrence: Example IV

1. Consider $\mathbf{T(n) = T(n/4) + T(3n/4) + n}$.
2. Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
3. Total work in any level is at most $\mathbf{n}$. Total work in any level without leaves is exactly $\mathbf{n}$.
4. Highest leaf is at level $\mathbf{\log_4 n}$ and lowest leaf is at level $\mathbf{\log_{4/3} n}$
5. Thus, $\mathbf{n \log_4 n \leq T(n) \leq n \log_{4/3} n}$ which means $\mathbf{T(n) = \Theta(n \log n)}$

$\left(\frac{3}{4}\right)^L n = c$

$L = \log_{\frac{4}{3}} n$

$n \log_4 n \leq W \leq n \log_{\frac{4}{3}} n$

Recurrence: Example IV

1. Consider $\mathbf{T(n) = T(n/4) + T(3n/4) + n}$.
2. Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
3. Total work in any level is at most $\mathbf{n}$. Total work in any level without leaves is exactly $\mathbf{n}$.
4. Highest leaf is at level $\mathbf{\log_4 n}$ and lowest leaf is at level $\mathbf{\log_{4/3} n}$
5. Thus, $\mathbf{n \log_4 n \leq T(n) \leq n \log_{4/3} n}$, which means $\mathbf{T(n) = \Theta(n \log n)}$

Recurrence: Example IV

1. Consider $\mathbf{T(n) = T(n/4) + T(3n/4) + n}$.
2. Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
3. Total work in any level is at most $\mathbf{n}$. Total work in any level without leaves is exactly $\mathbf{n}$.
4. Highest leaf is at level $\mathbf{\log_4 n}$ and lowest leaf is at level $\mathbf{\log_{4/3} n}$
5. Thus, $\mathbf{n \log_4 n \leq T(n) \leq n \log_{4/3} n}$, which means $\mathbf{T(n) = \Theta(n \log n)}$

Recurrence: Example IV

1. Consider $\mathbf{T(n) = T(n/4) + T(3n/4) + n}$.
2. Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
3. Total work in any level is at most $\mathbf{n}$. Total work in any level without leaves is exactly $\mathbf{n}$.
4. Highest leaf is at level $\mathbf{\log_4 n}$ and lowest leaf is at level $\mathbf{\log_{4/3} n}$
5. Thus, $\mathbf{n \log_4 n \leq T(n) \leq n \log_{4/3} n}$, which means $\mathbf{T(n) = \Theta(n \log n)}$

Recurrence: Example IV

1. Consider $\mathbf{T(n) = T(n/4) + T(3n/4) + n}$.
2. Using recursion tree, we observe the tree has leaves at different levels (a *lop-sided* tree).
3. Total work in any level is at most $\mathbf{n}$. Total work in any level without leaves is exactly $\mathbf{n}$.
4. Highest leaf is at level $\mathbf{\log_4 n}$ and lowest leaf is at level $\mathbf{\log_{4/3} n}$
5. Thus, $\mathbf{n \log_4 n \leq T(n) \leq n \log_{4/3} n}$, which means $\mathbf{T(n) = \Theta(n \log n)}$

Part II

Closest Pair

Closest Pair - the problem

Input Given a set $\mathbf{S}$ of $\mathbf{n}$ points on the plane

Goal Find $\mathbf{p}, \mathbf{q} \in \mathbf{S}$ such that $\mathbf{d(p, q)}$ is minimum

Closest Pair - the problem

Input Given a set $\mathbf{S}$ of $\mathbf{n}$ points on the plane

Goal Find $\mathbf{p}, \mathbf{q} \in \mathbf{S}$ such that $\mathbf{d(p, q)}$ is minimum

Applications

1. Basic primitive used in graphics, vision, molecular modelling
2. Ideas used in solving nearest neighbor, Voronoi diagrams, Euclidean MST

Algorithm: Brute Force

1. Compute distance between every pair of points and find minimum.
2. Takes $\mathbf{O(n^2)}$ time.
3. Can we do better?

Algorithm: Brute Force

1. Compute distance between every pair of points and find minimum.
2. Takes $\mathbf{O(n^2)}$ time.
3. Can we do better?

Algorithm: Brute Force

1. Compute distance between every pair of points and find minimum.
2. Takes $\mathbf{O(n^2)}$ time.
3. Can we do better?

Closest Pair: 1-d case

Input Given a set $\mathbf{S}$ of $\mathbf{n}$ points on a line

Goal Find $\mathbf{p}, \mathbf{q} \in \mathbf{S}$ such that $\mathbf{d(p, q)}$ is minimum

Algorithm

1. Sort points based on coordinate
2. Compute the distance between successive points, keeping track of the closest pair.

Running time $\mathbf{O(n \log n)}$

Can we do this in better running time?
Can reduce Distinct Elements Problem (see lecture 1) to this problem in $\mathbf{O(n)}$ time. Do you see how?

Closest Pair: 1-d case

Input Given a set $\mathbf{S}$ of $\mathbf{n}$ points on a line

Goal Find $\mathbf{p}, \mathbf{q} \in \mathbf{S}$ such that $\mathbf{d(p, q)}$ is minimum

Algorithm

1. Sort points based on coordinate
2. Compute the distance between successive points, keeping track of the closest pair.

Running time $\mathbf{O(n \log n)}$

Can we do this in better running time?
Can reduce Distinct Elements Problem (see lecture 1) to this problem in $\mathbf{O(n)}$ time. Do you see how?

Generalizing 1-d case

Can we generalize **1**-d algorithm to **2**-d?
Sort according to **x** or **y**-coordinate??
No easy generalization.

Generalizing 1-d case

Can we generalize **1**-d algorithm to **2**-d?
Sort according to **x** or **y**-coordinate??
No easy generalization.

First Attempt

Divide and Conquer I

1. Partition into 4 quadrants of roughly equal size.
2. Find closest pair in each quadrant recursively
3. Combine solutions

First Attempt

Divide and Conquer I

1. Partition into 4 quadrants of roughly equal size.Not always!
2. Find closest pair in each quadrant recursively
3. Combine solutions

New Algorithm

Divide and Conquer II

1. Divide the set of points into two equal parts via vertical line
2. Find closest pair in each half recursively
3. Find closest pair with one point in each half
4. Return the best pair among the above 3 solutions

New Algorithm

Divide and Conquer II

1. Divide the set of points into two equal parts via vertical line
2. Find closest pair in each half recursively
3. Find closest pair with one point in each half
4. Return the best pair among the above 3 solutions

New Algorithm

Divide and Conquer II

1. Divide the set of points into two equal parts via vertical line
2. Find closest pair in each half recursively
3. Find closest pair with one point in each half
4. Return the best pair among the above 3 solutions

New Algorithm

Divide and Conquer II

1. Divide the set of points into two equal parts via vertical line
2. Find closest pair in each half recursively
3. Find closest pair with one point in each half
4. Return the best pair among the above 3 solutions

1. Sort points based on **x**-coordinate and pick the median. Time $= O(n \log n)$
2. How to find closest pair with points in different halves? $O(n^2)$ is trivial. Better?

New Algorithm

Divide and Conquer II

1. Divide the set of points into two equal parts via vertical line
2. Find closest pair in each half recursively
3. Find closest pair with one point in each half
4. Return the best pair among the above 3 solutions

1. Sort points based on **x**-coordinate and pick the median. Time $= O(n \log n)$
2. How to find closest pair with points in different halves? $O(n^2)$ is trivial. Better?

New Algorithm

Divide and Conquer II

1. Divide the set of points into two equal parts via vertical line
2. Find closest pair in each half recursively
3. Find closest pair with one point in each half
4. Return the best pair among the above 3 solutions

1. Sort points based on **x**-coordinate and pick the median. Time $= O(n \log n)$
2. How to find closest pair with points in different halves? $O(n^2)$ is trivial. Better?

New Algorithm

Divide and Conquer II

1. Divide the set of points into two equal parts via vertical line
2. Find closest pair in each half recursively
3. Find closest pair with one point in each half
4. Return the best pair among the above 3 solutions

1. Sort points based on **x**-coordinate and pick the median. Time $= O(n \log n)$
2. How to find closest pair with points in different halves? $O(n^2)$ is trivial. Better?

New Algorithm

Divide and Conquer II

1. Divide the set of points into two equal parts via vertical line
2. Find closest pair in each half recursively
3. Find closest pair with one point in each half
4. Return the best pair among the above 3 solutions

1. Sort points based on **x**-coordinate and pick the median. Time $= O(n \log n)$
2. How to find closest pair with points in different halves? $O(n^2)$ is trivial. Better?

Combining Partial Solutions

1. Does it take $\mathbf{O(n^2)}$ to combine solutions?
2. Let δ be the distance between closest pairs, where both points belong to the same half.

Combining Partial Solutions

1. Let δ be the distance between closest pairs, where both points belong to the same half.
2. Need to consider points within δ of dividing line

Sparsity of Band XXX

Divide the band into square boxes of size $\delta/2$

Lemma

Each box has at most one point

Proof.

If not, then there are a pair of points (both belonging to one half) that are at most $\sqrt{2}\delta/2 < \delta$ apart! □

Sparsity of Band XXX

Divide the band into square boxes of size $\delta/2$

Lemma

Each box has at most one point

Proof.

If not, then there are a pair of points (both belonging to one half) that are at most $\sqrt{2}\delta/2 < \delta$ apart! □

Sparsity of Band XXX

Divide the band into square boxes of size $\delta/2$

Lemma

Each box has at most one point

Proof.

If not, then there are a pair of points (both belonging to one half) that are at most $\sqrt{2}\delta/2 < \delta$ apart! □

Searching within the Band

Lemma

Suppose $\mathbf{a}, \mathbf{b}$ *are both in the band* $\mathbf{d}(\mathbf{a}, \mathbf{b}) < \delta$ *then* $\mathbf{a}, \mathbf{b}$ *have at most two rows of boxes between them.*

Proof.

Each row of boxes has height $\delta/2$. If more than two rows then $\mathbf{d}(\mathbf{a}, \mathbf{b}) > 2 \cdot \delta/2$! □

Searching within the Band

Lemma

Suppose $\mathbf{a}, \mathbf{b}$ *are both in the band* $\mathbf{d}(\mathbf{a}, \mathbf{b}) < \delta$ *then* $\mathbf{a}, \mathbf{b}$ *have at most two rows of boxes between them.*

Proof.

Each row of boxes has height $\delta/2$. If more than two rows then $\mathbf{d}(\mathbf{a}, \mathbf{b}) > 2 \cdot \delta/2$! $\square$

Searching within the Band

Corollary

Order points according to their $\mathbf{y}$*-coordinate. If* $\mathbf{p}, \mathbf{q}$ *are such that* $\mathbf{d(p,q)} < \delta$ *then* $\mathbf{p}$ *and* $\mathbf{q}$ *are within* $\mathbf{11}$ *positions in the sorted list.*

Proof.

1. $\leq \mathbf{2}$ points between them if $\mathbf{p}$ and $\mathbf{q}$ in same row.
2. $\leq \mathbf{6}$ points between them if $\mathbf{p}$ and $\mathbf{q}$ in two consecutive rows.
3. $\leq \mathbf{10}$ points between if $\mathbf{p}$ and $\mathbf{q}$ one row apart.
4. $\implies$ More than ten points between them in the sorted $\mathbf{y}$ order than $\mathbf{p}$ and $\mathbf{q}$ are more than two rows apart.
5. $\implies \mathbf{d(p,q)} > \delta$. A contradiction. ■

The Algorithm

```
ClosestPair(P):
    1.  Split P into equal-sized sets P_1, P_2 via vertical line L
    2.  δ_1 ← ClosestPair(P_1).
    3.  δ_2 ← ClosestPair(P_2).
    4.  δ = min(δ_1, δ_2)
    5.  Delete points from P further than δ from L
    6.  Sort P based on y-coordinate into an array A
    7.  for i = 1 to |A| − 1 do
            for j = i + 1 to min{i + 11, |A|} do
                if (dist(A[i], A[j]) < δ) update δ and closest pair
```

1. Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
2. Step 5 takes $O(n)$ time.
3. Step 6 takes $O(n \log n)$ time

The Algorithm

```
ClosestPair(P):
    1.  Split P into equal-sized sets P_1, P_2 via vertical line L
    2.  δ_1 ← ClosestPair(P_1).
    3.  δ_2 ← ClosestPair(P_2).
    4.  δ = min(δ_1, δ_2)
    5.  Delete points from P further than δ from L
    6.  Sort P based on y-coordinate into an array A
    7.  for i = 1 to |A| − 1 do
            for j = i + 1 to min{i + 11, |A|} do
                if (dist(A[i], A[j]) < δ) update δ and closest pair
```

1. Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
2. Step 5 takes $O(n)$ time.
3. Step 6 takes $O(n \log n)$ time

The Algorithm

```
ClosestPair(P):
    1.  Split P into equal-sized sets P_1, P_2 via vertical line L
    2.  δ_1 ← ClosestPair(P_1).
    3.  δ_2 ← ClosestPair(P_2).
    4.  δ = min(δ_1, δ_2)
    5.  Delete points from P further than δ from L
    6.  Sort P based on y-coordinate into an array A
    7.  for i = 1 to |A| − 1 do
            for j = i + 1 to min{i + 11, |A|} do
                if (dist(A[i], A[j]) < δ) update δ and closest pair
```

1. Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
2. Step 5 takes $O(n)$ time.
3. Step 6 takes $O(n \log n)$ time

The Algorithm

```
ClosestPair(P):
    1.  Split P into equal-sized sets P_1, P_2 via vertical line L
    2.  δ_1 ← ClosestPair(P_1).
    3.  δ_2 ← ClosestPair(P_2).
    4.  δ = min(δ_1, δ_2)
    5.  Delete points from P further than δ from L
    6.  Sort P based on y-coordinate into an array A
    7.  for i = 1 to |A| − 1 do
            for j = i + 1 to min{i + 11, |A|} do
                if (dist(A[i], A[j]) < δ) update δ and closest pair
```

1. Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
2. Step 5 takes $O(n)$ time.
3. Step 6 takes $O(n \log n)$ time

The Algorithm

```
ClosestPair(P):
    1.  Split P into equal-sized sets P_1, P_2 via vertical line L
    2.  δ_1 ← ClosestPair(P_1).
    3.  δ_2 ← ClosestPair(P_2).
    4.  δ = min(δ_1, δ_2)
    5.  Delete points from P further than δ from L
    6.  Sort P based on y-coordinate into an array A
    7.  for i = 1 to |A| − 1 do
            for j = i + 1 to min{i + 11, |A|} do
                if (dist(A[i], A[j]) < δ) update δ and closest pair
```

1. Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
2. Step 5 takes $O(n)$ time.
3. Step 6 takes $O(n \log n)$ time

The Algorithm

```
ClosestPair(P):
    1.  Split P into equal-sized sets P_1, P_2 via vertical line L
    2.  δ_1 ← ClosestPair(P_1).
    3.  δ_2 ← ClosestPair(P_2).
    4.  δ = min(δ_1, δ_2)
    5.  Delete points from P further than δ from L
    6.  Sort P based on y-coordinate into an array A
    7.  for i = 1 to |A| − 1 do
            for j = i + 1 to min{i + 11, |A|} do
                if (dist(A[i], A[j]) < δ) update δ and closest pair
```

1. Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
2. Step 5 takes $O(n)$ time.
3. Step 6 takes $O(n \log n)$ time

The Algorithm

```
ClosestPair(P):
    1.  Split P into equal-sized sets P_1, P_2 via vertical line L
    2.  δ_1 ← ClosestPair(P_1).
    3.  δ_2 ← ClosestPair(P_2).
    4.  δ = min(δ_1, δ_2)
    5.  Delete points from P further than δ from L
    6.  Sort P based on y-coordinate into an array A
    7.  for i = 1 to |A| − 1 do
            for j = i + 1 to min{i + 11, |A|} do
                if (dist(A[i], A[j]) < δ) update δ and closest pair
```

1. Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
2. Step 5 takes $O(n)$ time.
3. Step 6 takes $O(n \log n)$ time

The Algorithm

```
ClosestPair(P):
    1.  Split P into equal-sized sets P_1, P_2 via vertical line L
    2.  δ_1 ← ClosestPair(P_1).
    3.  δ_2 ← ClosestPair(P_2).
    4.  δ = min(δ_1, δ_2)
    5.  Delete points from P further than δ from L
    6.  Sort P based on y-coordinate into an array A
    7.  for i = 1 to |A| − 1 do
            for j = i + 1 to min{i + 11, |A|} do
                if (dist(A[i], A[j]) < δ) update δ and closest pair
```

1. Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
2. Step 5 takes $O(n)$ time.
3. Step 6 takes $O(n \log n)$ time

The Algorithm

```
ClosestPair(P):
    1.  Split P into equal-sized sets P_1, P_2 via vertical line L
    2.  δ_1 ← ClosestPair(P_1).
    3.  δ_2 ← ClosestPair(P_2).
    4.  δ = min(δ_1, δ_2)
    5.  Delete points from P further than δ from L
    6.  Sort P based on y-coordinate into an array A
    7.  for i = 1 to |A| − 1 do
            for j = i + 1 to min{i + 11, |A|} do
                if (dist(A[i], A[j]) < δ) update δ and closest pair
```

1. Step 1, involves sorting and scanning. Takes $O(n \log n)$ time.
2. Step 5 takes $O(n)$ time.
3. Step 6 takes $O(n \log n)$ time
4. Step 7 takes $O(n)$ time.

Running Time

The running time of the algorithm is given by

$$T(n) \leq 2T(n/2) + \underbrace{O(n \log n)}_{n}$$

Thus, $T(n) = O(n \log^2 n)$.

Improved Algorithm

Avoid repeated sorting of points in band: two options

1. Sort all points by **y**-coordinate and store the list. In conquer step use this to avoid sorting
2. Each recursive call returns a list of points sorted by their **y**-coordinates. Merge in conquer step in linear time.

Analysis: $T(n) \leq 2T(n/2) + O(n) = O(n \log n)$

Running Time

The running time of the algorithm is given by

$$T(n) \leq 2T(n/2) + O(n \log n)$$

Thus, $T(n) = O(n \log^2 n)$.

Improved Algorithm

Avoid repeated sorting of points in band: two options

1. Sort all points by y-coordinate and store the list. In conquer step use this to avoid sorting
2. Each recursive call returns a list of points sorted by their y-coordinates. Merge in conquer step in linear time.

Analysis: $T(n) \leq 2T(n/2) + O(n) = O(n \log n)$

Running Time

The running time of the algorithm is given by

$$T(n) \leq 2T(n/2) + O(n \log n)$$

Thus, $T(n) = O(n \log^2 n)$.

Improved Algorithm

Avoid repeated sorting of points in band: two options

1. Sort all points by **y**-coordinate and store the list. In conquer step use this to avoid sorting
2. Each recursive call returns a list of points sorted by their **y**-coordinates. Merge in conquer step in linear time.

Analysis: $T(n) \leq 2T(n/2) + O(n) = O(n \log n)$

Part III

Selecting in Unsorted Lists

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is $O(n)$
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is **O(n)**
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Quick Sort

Quick Sort [Hoare]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself. Linear scan of array does it. Time is **O(n)**
3. Recursively sort the subarrays, and concatenate them.

Example:

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1
2. pivot: 16
3. split into 12, 14, 5, 3, 1 and 20, 19, 18 and recursively sort
4. put them together with pivot in middle

Time Analysis

1. Let $\mathbf{k}$ be the rank of the chosen pivot. Then, $\mathbf{T(n) = T(k-1) + T(n-k) + O(n)}$
2. If $\mathbf{k = \lceil n/2 \rceil}$ then $\mathbf{T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)}$. Then, $\mathbf{T(n) = O(n \log n)}$.
   1. Theoretically, median can be found in linear time.
3. Typically, pivot is the first or last element of array. Then,

$$\mathbf{T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))}$$

In the worst case $\mathbf{T(n) = T(n-1) + O(n)}$, which means $\mathbf{T(n) = O(n^2)}$. Happens if array is already sorted and pivot is always first element.

Time Analysis

1. Let $\mathbf{k}$ be the rank of the chosen pivot. Then, $\mathbf{T(n) = T(k-1) + T(n-k) + O(n)}$
2. If $\mathbf{k = \lceil n/2 \rceil}$ then $\mathbf{T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)}$. Then, $\mathbf{T(n) = O(n \log n)}$.
   1. Theoretically, median can be found in linear time.
3. Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

Time Analysis

1. Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
   1. Theoretically, median can be found in linear time.
3. Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

Time Analysis

1. Let k be the rank of the chosen pivot. Then, $T(n) = T(k-1) + T(n-k) + O(n)$
2. If $k = \lceil n/2 \rceil$ then $T(n) = T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \leq 2T(n/2) + O(n)$. Then, $T(n) = O(n \log n)$.
   1. Theoretically, median can be found in linear time.
3. Typically, pivot is the first or last element of array. Then,

$$T(n) = \max_{1 \leq k \leq n} (T(k-1) + T(n-k) + O(n))$$

In the worst case $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$. Happens if array is already sorted and pivot is always first element.

Prune and search

Consider an algorithm **alg** that is given an input of size **n**. In **O(n)** time, it either solves the problem, or solve it by calling recursively on an input of size, say, $\leq$ **(15/16)n**. The running time of **alg** is

(A) $O(n^2)$

(B) $O(n \log n)$

(C) $O(n)$

(D) There are not enough details.

$$1 + a + a^2 + \cdots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

$$T(n) = T\left(\frac{15}{16} n\right) + O(n)$$

Problem - Selection

Input Unsorted array $\mathbf{A}$ of $\mathbf{n}$ integers

Goal Find the $\mathbf{j}$th smallest number in $\mathbf{A}$ (*rank* $\mathbf{j}$ number)

Example

$\mathbf{A} = \{4, 6, 2, 1, 5, 8, 7\}$ and $\mathbf{j} = 4$. The $\mathbf{j}$th smallest element is 5.

Median: $\mathbf{j} = \lfloor (\mathbf{n} + 1)/2 \rfloor$

Simplifying assumption for sake of notation: elements of $\mathbf{A}$ are distinct

Problem - Selection

Input Unsorted array $\mathbf{A}$ of $\mathbf{n}$ integers

Goal Find the $\mathbf{j}$th smallest number in $\mathbf{A}$ (*rank* $\mathbf{j}$ number)

Example

$\mathbf{A} = \{4, 6, 2, 1, 5, 8, 7\}$ and $\mathbf{j} = 4$. The $\mathbf{j}$th smallest element is 5.

Median: $\mathbf{j} = \lfloor (\mathbf{n} + 1)/2 \rfloor$

Simplifying assumption for sake of notation: elements of $\mathbf{A}$ are distinct

Algorithm I

1. Sort the elements in **A**
2. Pick **j**th element in sorted order

Time taken = $\mathbf{O(n \log n)}$

Do we need to sort? Is there an $\mathbf{O(n)}$ time algorithm?

Algorithm I

1. Sort the elements in **A**
2. Pick **j**th element in sorted order

Time taken = $\mathbf{O(n \log n)}$

Do we need to sort? Is there an $\mathbf{O(n)}$ time algorithm?

Algorithm II

If $\mathbf{j}$ is small or $\mathbf{n - j}$ is small then

1. Find $\mathbf{j}$ smallest/largest elements in $\mathbf{A}$ in $\mathbf{O(jn)}$ time. (How?)
2. Time to find median is $\mathbf{O(n^2)}$.

$j = 100$

$l = 10$

QuickSelect (A, j)

pick pivot $p = A[1]$

Let $l = \text{rank}(p)$

If $l < j$

QuickSelect $(A_{right}, j - l)$

Else $l > j$

QuickSelect (A_{left}, j)

Else declare victory

Divide and Conquer Approach

1. Pick a pivot element $\mathbf{a}$ from $\mathbf{A}$
2. Partition $\mathbf{A}$ based on $\mathbf{a}$.
   $\mathbf{A}_{\mathbf{less}} = \{\mathbf{x} \in \mathbf{A} \mid \mathbf{x} \leq \mathbf{a}\}$ and $\mathbf{A}_{\mathbf{greater}} = \{\mathbf{x} \in \mathbf{A} \mid \mathbf{x} > \mathbf{a}\}$
3. $|\mathbf{A}_{\mathbf{less}}| = \mathbf{j}$: return $\mathbf{a}$
4. $|\mathbf{A}_{\mathbf{less}}| > \mathbf{j}$: recursively find $\mathbf{j}$th smallest element in $\mathbf{A}_{\mathbf{less}}$
5. $|\mathbf{A}_{\mathbf{less}}| < \mathbf{j}$: recursively find $\mathbf{k}$th smallest element in $\mathbf{A}_{\mathbf{greater}}$ where $\mathbf{k} = \mathbf{j} - |\mathbf{A}_{\mathbf{less}}|$.

Time Analysis

1. Partitioning step: $\mathbf{O(n)}$ time to scan $\mathbf{A}$
2. How do we choose pivot? Recursive running time?

Suppose we always choose pivot to be $\mathbf{A[1]}$.

Say $\mathbf{A}$ is sorted in increasing order and $\mathbf{j = n}$.
Exercise: show that algorithm takes $\mathbf{\Omega(n^2)}$ time

Time Analysis

1. Partitioning step: $\mathbf{O(n)}$ time to scan $\mathbf{A}$
2. How do we choose pivot? Recursive running time?

Suppose we always choose pivot to be $\mathbf{A[1]}$.

Say $\mathbf{A}$ is sorted in increasing order and $\mathbf{j = n}$.
Exercise: show that algorithm takes $\mathbf{\Omega(n^2)}$ time

Time Analysis

1. Partitioning step: $\mathbf{O(n)}$ time to scan $\mathbf{A}$
2. How do we choose pivot? Recursive running time?

Suppose we always choose pivot to be $\mathbf{A[1]}$.

Say $\mathbf{A}$ is sorted in increasing order and $\mathbf{j = n}$.
Exercise: show that algorithm takes $\mathbf{\Omega(n^2)}$ time

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4} \leq \ell \leq \mathbf{3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4} \leq |\mathbf{A}_{\text{less}}| \leq \mathbf{3n/4}$ *and* $\mathbf{n/4} \leq |\mathbf{A}_{\text{greater}}| \leq \mathbf{3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.

Can we choose pivot deterministically?

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4 \leq \ell \leq 3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4 \leq |A_{less}| \leq 3n/4}$ *and* $\mathbf{n/4 \leq |A_{greater}| \leq 3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.

Can we choose pivot deterministically?

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4 \leq \ell \leq 3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4 \leq |A_{less}| \leq 3n/4}$ *and* $\mathbf{n/4 \leq |A_{greater}| \leq 3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.

Can we choose pivot deterministically?

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4 \leq \ell \leq 3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4 \leq |A_{less}| \leq 3n/4}$ *and* $\mathbf{n/4 \leq |A_{greater}| \leq 3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.

Can we choose pivot deterministically?

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4} \leq \ell \leq \mathbf{3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4} \leq |\mathbf{A}_{\text{less}}| \leq \mathbf{3n/4}$ *and* $\mathbf{n/4} \leq |\mathbf{A}_{\text{greater}}| \leq \mathbf{3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.

Can we choose pivot deterministically?

A Better Pivot

Suppose pivot is the ℓth smallest element where $\mathbf{n/4} \leq \ell \leq \mathbf{3n/4}$.
That is pivot is *approximately* in the middle of $\mathbf{A}$
Then $\mathbf{n/4} \leq |\mathbf{A}_{\text{less}}| \leq \mathbf{3n/4}$ *and* $\mathbf{n/4} \leq |\mathbf{A}_{\text{greater}}| \leq \mathbf{3n/4}$. If we apply recursion,

$$\mathbf{T(n) \leq T(3n/4) + O(n)}$$

Implies $\mathbf{T(n) = O(n)}$!

How do we find such a pivot? Randomly? In fact works!
Analysis a little bit later.

Can we choose pivot deterministically?

Divide and Conquer Approach

A game of medians

Idea

1. Break input $\mathbf{A}$ into many subarrays: $\mathbf{L_1, \ldots L_k}$.
2. Find median $\mathbf{m_i}$ in each subarray $\mathbf{L_i}$.
3. Find the median $\mathbf{x}$ of the medians $\mathbf{m_1, \ldots, m_k}$.
4. Intuition: The median $\mathbf{x}$ should be close to being a good median of all the numbers in $\mathbf{A}$.
5. Use $\mathbf{x}$ as pivot in previous algorithm.

But we have to be...

More specific...

1. Size of each group?
2. How to find median of medians?

Choosing the pivot

A clash of medians

1. Partition array $\mathbf{A}$ into $\lceil n/5 \rceil$ lists of 5 items each.
   $L_1 = \{A[1], A[2], \ldots, A[5]\}$, $L_2 = \{A[6], \ldots, A[10]\}$, $\ldots$,
   $L_i = \{A[5i+1], \ldots, A[5i-4]\}$, $\ldots$,
   $L_{\lceil n/5 \rceil} = \{A[5\lceil n/5 \rceil - 4, \ldots, A[n]\}$.
2. For each i find median b_i of L_i using brute-force in $O(1)$ time. Total $O(n)$ time
3. Let $B = \{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}\}$
4. Find median b of B

Lemma

Median of B *is an* approximate *median of* A. *That is, if* b *is used a pivot to partition* A, *then* $|A_{less}| \leq 7n/10 + 6$ and $|A_{greater}| \leq 7n/10 + 6$.

Choosing the pivot

A clash of medians

1. Partition array $\mathbf{A}$ into $\lceil \mathbf{n}/\mathbf{5} \rceil$ lists of $\mathbf{5}$ items each.
   $\mathbf{L_1} = \{\mathbf{A[1], A[2], \ldots, A[5]}\}$, $\mathbf{L_2} = \{\mathbf{A[6], \ldots, A[10]}\}$, $\ldots$,
   $\mathbf{L_i} = \{\mathbf{A[5i+1], \ldots, A[5i-4]}\}$, $\ldots$,
   $\mathbf{L_{\lceil n/5 \rceil}} = \{\mathbf{A[5\lceil n/5 \rceil - 4, \ldots, A[n]}\}$.
2. For each $\mathbf{i}$ find median $\mathbf{b_i}$ of $\mathbf{L_i}$ using brute-force in $\mathbf{O(1)}$ time.
   Total $\mathbf{O(n)}$ time
3. Let $\mathbf{B} = \{\mathbf{b_1, b_2, \ldots, b_{\lceil n/5 \rceil}}\}$
4. Find median $\mathbf{b}$ of $\mathbf{B}$

Lemma

Median of $\mathbf{B}$ *is an* approximate *median of* $\mathbf{A}$. *That is, if* $\mathbf{b}$ *is used a pivot to partition* $\mathbf{A}$, *then* $|\mathbf{A}_{less}| \leq \mathbf{7n/10 + 6}$ and $|\mathbf{A}_{greater}| \leq \mathbf{7n/10 + 6}$.

Algorithm for Selection

A storm of medians

```
select(A, j):
    Form lists L_1, L_2, ..., L_⌈n/5⌉ where L_i = {A[5i − 4], ..., A[5i]}
    Find median b_i of each L_i using brute-force
    Find median b of B = {b_1, b_2, ..., b_⌈n/5⌉}
    Partition A into A_less and A_greater using b as pivot
    if (|A_less|) = j return b
    else if (|A_less|) > j)
        return select(A_less, j)
    else
        return select(A_greater, j − |A_less|)
```

How do we find median of B?

Algorithm for Selection

A storm of medians

```
select(A, j):
    Form lists L_1, L_2, ..., L_⌈n/5⌉ where L_i = {A[5i − 4], ..., A[5i]}
    Find median b_i of each L_i using brute-force
    Find median b of B = {b_1, b_2, ..., b_⌈n/5⌉}
    Partition A into A_less and A_greater using b as pivot
    if (|A_less|) = j return b
    else if (|A_less|) > j)
        return select(A_less, j)
    else
        return select(A_greater, j − |A_less|)
```

How do we find median of B? Recursively!

Algorithm for Selection

A storm of medians

```
select(A, j):
    Form lists L_1, L_2, ..., L_⌈n/5⌉ where L_i = {A[5i − 4], ..., A[5i]}
    Find median b_i of each L_i using brute-force
    Find median b of B = {b_1, b_2, ..., b_⌈n/5⌉}
    Partition A into A_less and A_greater using b as pivot
    if (|A_less|) = j return b
    else if (|A_less|) > j)
        return select(A_less, j)
    else
        return select(A_greater, j − |A_less|)
```

How do we find median of $\mathbf{B}$? Recursively!

Algorithm for Selection

A storm of medians

```
select(A, j):
    Form lists L_1, L_2, ..., L_⌈n/5⌉ where L_i = {A[5i − 4], ..., A[5i]}
    Find median b_i of each L_i using brute-force
    B = [b_1, b_2, ..., b_⌈n/5⌉]
    b = select(B, ⌈n/10⌉)
    Partition A into A_less and A_greater using b as pivot
    if (|A_less|) = j return b
    else if (|A_less|) > j)
        return select(A_less, j)
    else
        return select(A_greater, j − |A_less|)
```

Running time of deterministic median selection

A dance with recurrences

$$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}})|\} + O(n)$$

From Lemma,

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$$

and

$$T(n) = O(1) \qquad n < 10$$

Exercise: show that $T(n) = O(n)$

Running time of deterministic median selection

A dance with recurrences

$$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}})|\} + O(n)$$

From Lemma,

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$$

and

$$T(n) = O(1) \qquad n < 10$$

Exercise: show that $T(n) = O(n)$

Running time of deterministic median selection

A dance with recurrences

$$T(n) = T(\lceil n/5 \rceil) + \max\{T(|A_{\text{less}}|), T(|A_{\text{greater}})|\} + O(n)$$

From Lemma,

$$T(n) \leq T(\lceil n/5 \rceil) + T(\lfloor 7n/10 + 6 \rfloor) + O(n)$$

and

$$T(n) = O(1) \qquad n < 10$$

Exercise: show that $T(n) = O(n)$

Median of Medians: Proof of Lemma

Figure : Shaded elements are all greater than **b**

Proposition

There are at least $\mathbf{3n/10 - 6}$ *elements greater than the median of medians* **b**.

Proof.

At least half of the $\lceil \mathbf{n/5} \rceil$ groups have at least 3 elements larger than **b**, except for last group and the group containing **b**. Hence number of elements greater than **b** is:

$$3(\lceil (1/2)\lceil n/5 \rceil \rceil - 2) \geq 3n/10 - 6$$

□

Median of Medians: Proof of Lemma

Figure : Shaded elements are all greater than **b**

Proposition

There are at least $\mathbf{3n/10 - 6}$ *elements greater than the median of medians* **b**.

Proof.

At least half of the $\lceil \mathbf{n/5} \rceil$ groups have at least 3 elements larger than **b**, except for last group and the group containing **b**. Hence number of elements greater than **b** is:

$$\mathbf{3(\lceil (1/2)\lceil n/5 \rceil \rceil - 2) \geq 3n/10 - 6}$$

□

Median of Medians: Proof of Lemma

Proposition

There are at least $\mathbf{3n/10 - 6}$ *elements greater than the median of medians* $\mathbf{b}$.

Corollary

$|\mathbf{A}_{less}| \leq \mathbf{7n/10 + 6}$.

Via symmetric argument,

Corollary

$|\mathbf{A}_{greater}| \leq \mathbf{7n/10 + 6}$.

$$T(n) \le T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + cn$$

$n \longleftarrow$

$\frac{9n}{10} \longleftarrow$

$\left(\frac{9}{10}\right)^2 n$

n

$\frac{n}{5}$

$\frac{7n}{10}$

$$\underbrace{T(n) \le T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)} + n$$

$$T(n) \le \alpha \cdot n$$

$$T(n) \le \alpha \cdot \frac{n}{5} + \alpha \cdot \frac{7n}{10} + n$$

$$\le \alpha\left(\frac{9n}{10}\right) + n$$

$$\alpha\left(\frac{9n}{10}\right) + n \le \alpha n$$

Questions to ponder

1. Why did we choose lists of size **5**? Will lists of size **3** work?
2. Write a recurrence to analyze the algorithm's running time if we choose a list of size **k**.

Median of Medians Algorithm

Due to:
M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.
"Time bounds for selection".
Journal of Computer System Sciences (JCSS), 1973.

How many Turing Award winners in the author list?
All except Vaughn Pratt!

Median of Medians Algorithm

Due to:
M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.
"Time bounds for selection".
Journal of Computer System Sciences (JCSS), 1973.

How many Turing Award winners in the author list?
All except Vaughn Pratt!

Median of Medians Algorithm

Due to:
M. Blum, R. Floyd, D. Knuth, V. Pratt, R. Rivest, and R. Tarjan.
"Time bounds for selection".
Journal of Computer System Sciences (JCSS), 1973.

How many Turing Award winners in the author list?
All except Vaughn Pratt!

Takeaway Points

1. Recursion tree method and guess and verify are the most reliable methods to analyze recursions in algorithms.
2. Recursive algorithms naturally lead to recurrences.
3. Some times one can look for certain type of recursive algorithms (reverse engineering) by understanding recurrences and their behavior.

Notes

Notes

Notes

Notes

Binary Search, Introduction to Dynamic Programming

Lecture 7

February 11, 2014

Part I

Exponentiation, Binary Search

Exponentiation

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

Goal Compute $\mathbf{a^n}$

Obvious algorithm:

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

O(n) multiplications.

Exponentiation

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

Goal Compute $\mathbf{a^n}$

Obvious algorithm:

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

O(n) multiplications.

How many bits...

Let $a > 1$ and $n > 1$ be two integer numbers. Representing a^n in base 2 requires

(A) $O(\log a + \log n)$ bits.

(B) $O(n \log a)$ bits.

(C) $O(a \log n)$ bits.

(D) $O(\log a \log n)$ bits.

(E) $O\left((\log a)^{\log n}\right)$ bits.

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, ⌊n/2⌋)
        x = x * x
        if (n is odd) then
            x = x * a
        return x
```

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

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

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

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, ⌊n/2⌋)
        x = x * x
        if (n is odd) then
            x = x * a
        return x
```

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

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

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

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, ⌊n/2⌋)
        x = x * x
        if (n is odd) then
            x = x * a
        return x
```

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

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

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

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, ⌊n/2⌋)
        x = x * x
        if (n is odd) then
            x = x * a
        return x
```

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

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

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

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, ⌊n/2⌋)
        x = x * x
        if (n is odd) then
            x = x * a
        return x
```

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

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

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

Complexity of Exponentiation

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

Input size: $O(\log a + \log n)$

Output size: $O(n \log a)$.

Not necessarily polynomial in input size!

Both **SlowPow** and **FastPow** are polynomial in output size.

Complexity of Exponentiation

Question: Is **SlowPow**() a polynomial time algorithm? **FastPow**?
Input size: $\mathbf{O(\log a + \log n)}$
Output size: $\mathbf{O(n \log a)}$.

Not necessarily polynomial in input size!

Both **SlowPow** and **FastPow** are polynomial in output size.

Complexity of Exponentiation

Question: Is **SlowPow**() a polynomial time algorithm? **FastPow**?
Input size: $\mathbf{O(\log a + \log n)}$
Output size: $\mathbf{O(n \log a)}$.

Not necessarily polynomial in input size!

Both **SlowPow** and **FastPow** are polynomial in output size.

Complexity of Exponentiation

Question: Is **SlowPow**() a polynomial time algorithm? **FastPow**?
Input size: $\mathbf{O(\log a + \log n)}$
Output size: $\mathbf{O(n \log a)}$.

Not necessarily polynomial in input size!

Both **SlowPow** and **FastPow** are polynomial in output size.

26 mod 7 is?

(A) 0
(B) 1
(C) 3
(D) 5
(E) 7

Exponentiation modulo a given number

Exponentiation in applications:

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

Goal Compute $\mathbf{a^n \mod p}$

Input size: $\Theta(\log a + \log n + \log p)$
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: $\mathbf{a}$, $\mathbf{n \geq 0}$, $\mathbf{p \geq 2}$ (typically a prime)

Goal Compute $\mathbf{a^n \mod p}$

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

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

Exponentiation modulo a given number

Exponentiation in applications:

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

Goal Compute $\mathbf{a^n \mod p}$

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

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

Exponentiation modulo a given number

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

Goal Compute $\mathbf{a^n} \mod \mathbf{p}$

```
FastPowMod(a,n,p):
        if (n = 0) return 1
        x =FastPowMod(a,⌊n/2⌋,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 (why?).

Exponentiation modulo a given number

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

Goal Compute $\mathbf{a^n} \mod \mathbf{p}$

```
FastPowMod(a,n,p):
        if (n = 0) return 1
        x =FastPowMod(a,⌊n/2⌋,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 (why?).

Binary Search in Sorted Arrays

Input Sorted array $\mathbf{A}$ of $\mathbf{n}$ numbers and number $\mathbf{x}$

Goal Is $\mathbf{x}$ in $\mathbf{A}$?

```
BinarySearch(A[a..b], x):
    if (b − a < 0) return NO
    mid = A[⌊(a + b)/2⌋]
    if (x = mid) return YES
    if (x < mid)
        return BinarySearch(A[a..⌊(a + b)/2⌋ − 1], x)
    else
        return BinarySearch(A[⌊(a + b)/2⌋ + 1..b], x)
```

Analysis: $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. $T(n) = O(\log n)$.

Observation: After $\mathbf{k}$ steps, size of array left is $n/2^k$

Binary Search in Sorted Arrays

Input Sorted array $\mathbf{A}$ of $\mathbf{n}$ numbers and number $\mathbf{x}$

Goal Is $\mathbf{x}$ in $\mathbf{A}$?

```
BinarySearch(A[a..b], x):
    if (b − a < 0) return NO
    mid = A[⌊(a + b)/2⌋]
    if (x = mid) return YES
    if (x < mid)
        return BinarySearch(A[a..⌊(a + b)/2⌋ − 1], x)
    else
        return BinarySearch(A[⌊(a + b)/2⌋ + 1..b],x)
```

Analysis: $T(n) = T(\lfloor n/2 \rfloor) + O(1)$. $T(n) = O(\log n)$.

Observation: After $\mathbf{k}$ steps, size of array left is $n/2^k$

Binary Search in Sorted Arrays

Input Sorted array $\mathbf{A}$ of $\mathbf{n}$ numbers and number $\mathbf{x}$

Goal Is $\mathbf{x}$ in $\mathbf{A}$?

```
BinarySearch(A[a..b], x):
    if (b − a < 0) return NO
    mid = A[⌊(a + b)/2⌋]
    if (x = mid) return YES
    if (x < mid)
        return BinarySearch(A[a..⌊(a + b)/2⌋ − 1], x)
    else
        return BinarySearch(A[⌊(a + b)/2⌋ + 1..b],x)
```

Analysis: $\mathbf{T(n) = T(\lfloor n/2 \rfloor) + O(1)}$. $\mathbf{T(n) = O(\log n)}$.

Observation: After $\mathbf{k}$ steps, size of array left is $\mathbf{n/2^k}$

Another common use of binary search

1. Optimization version: find solution of best (say minimum) value
2. Decision version: is there a solution of value at most a given value $\mathbf{v}$?

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

1. Given instance $\mathbf{I}$ compute upper bound $\mathbf{U(I)}$ on best value
2. Compute lower bound $\mathbf{L(I)}$ on best value
3. Do binary search on interval $\mathbf{[L(I), U(I)]}$ using decision version as black box
4. $\mathbf{O(\log(U(I) - L(I)))}$ calls to decision version if $\mathbf{U(I), L(I)}$ are integers

Another common use of binary search

1. Optimization version: find solution of best (say minimum) value
2. Decision version: is there a solution of value at most a given value $\mathbf{v}$?

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

1. Given instance $\mathbf{I}$ compute upper bound $\mathbf{U(I)}$ on best value
2. Compute lower bound $\mathbf{L(I)}$ on best value
3. Do binary search on interval $\mathbf{[L(I), U(I)]}$ using decision version as black box
4. $\mathbf{O(\log(U(I) - L(I)))}$ calls to decision version if $\mathbf{U(I), L(I)}$ are integers

Example

1. Problem: shortest paths in a graph.
2. Decision version: given $\mathbf{G}$ with non-negative integer edge lengths, nodes $\mathbf{s}, \mathbf{t}$ and bound $\mathbf{B}$, is there an $\mathbf{s}$-$\mathbf{t}$ path in $\mathbf{G}$ of length at most $\mathbf{B}$?
3. Optimization version: find the length of a shortest path between $\mathbf{s}$ and $\mathbf{t}$ in $\mathbf{G}$.

Question: given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?

Example continued

Question: given a black box algorithm for the decision version, can we obtain an algorithm for the optimization version?

1. Let $\mathbf{U}$ be maximum edge length in $\mathbf{G}$.
2. Minimum edge length is $\mathbf{L}$.
3. $\mathbf{s}$-$\mathbf{t}$ shortest path length is at most $\mathbf{(n-1)U}$ and at least $\mathbf{L}$.
4. Apply binary search on the interval $\mathbf{[L, (n-1)U]}$ via the algorithm for the decision problem.
5. $\mathbf{O(\log((n-1)U - L))}$ calls to the decision problem algorithm sufficient. Polynomial in input size.

Question

$\mathbf{G = (V, E)}$ is a directed graph with non-negative edge lengths; $\ell(\mathbf{e})$ lenght of edge $\mathbf{e}$. Want to find cycle $\mathbf{C}$ to minimize $\ell(\mathbf{C})/|\mathbf{C}|$, that is, the average length of the cycle.

Recall discussion question: given $\boldsymbol{\lambda}$ can reduce checking whether $\mathbf{G}$ has cycle of average length $\leq \boldsymbol{\lambda}$ to negative cycle detection.

Question: Suppose we do binary search using the preceding algorithm to find the minize the average length of a cycle? What is the search range? How many times do we need to call the algorithm for negative cycle detection?

Part II

Introduction to Dynamic Programming

Recursion

Reduction:

Reduce one problem to another

Recursion

A special case of reduction

1. reduce problem to a *smaller* instance of *itself*
2. self-reduction

1. Problem instance of size $\mathbf{n}$ is reduced to one or more instances of size $\mathbf{n} - \mathbf{1}$ or less.
2. For termination, problem instances of small size are solved by some other method as **base cases**.

Recursion

Reduction:

Reduce one problem to another

Recursion

A special case of reduction

1. reduce problem to a *smaller* instance of *itself*
2. self-reduction

1. Problem instance of size $\mathbf{n}$ is reduced to one or more instances of size $\mathbf{n-1}$ or less.
2. For termination, problem instances of small size are solved by some other method as **base cases**.

Recursion in Algorithm Design

1. **Tail Recursion**: problem reduced to a *single* recursive call after some work. Easy to convert algorithm into iterative or greedy algorithms. Examples: Interval scheduling, MST algorithms, etc.
2. **Divide and Conquer**: Problem reduced to multiple **independent** sub-problems that are solved separately. Conquer step puts together solution for bigger problem.

   Examples: Closest pair, deterministic median selection, quick sort.
3. **Dynamic Programming**: problem reduced to multiple (typically) *dependent or overlapping* sub-problems. Use **memoization** to avoid recomputation of common solutions leading to *iterative bottom-up* algorithm.

Fibonacci Numbers

Fibonacci numbers defined by recurrence:

$$\mathbf{F(n) = F(n-1) + F(n-2)} \text{ and } \mathbf{F(0) = 0, F(1) = 1.}$$

These numbers have many interesting and amazing properties.
A journal *The Fibonacci Quarterly*!

1. $\mathbf{F(n) = (\phi^n - (1-\phi)^n)/\sqrt{5}}$ where ϕ is the golden ratio $\mathbf{(1+\sqrt{5})/2 \simeq 1.618}$.
2. $\lim_{\mathbf{n}\to\infty}\mathbf{F(n+1)/F(n)} = \phi$

How many bits?

Consider the **n**th Fibonacci number $\mathbf{F(n)}$. Writing the number $\mathbf{F(n)}$ in base **2** requires

(A) $\Theta(n^2)$ bits.

(B) $\Theta(n)$ bits.

(C) $\Theta(\log n)$ bits.

(D) $\Theta(\log \log n)$ bits.

Recursive Algorithm for Fibonacci Numbers

Question: Given $\mathbf{n}$, compute $\mathbf{F(n)}$.

```
Fib(n):
        if (n = 0)
            return 0
        else if (n = 1)
            return 1
        else
            return Fib(n − 1) + Fib(n − 2)
```

Running time? Let $\mathbf{T(n)}$ be the number of additions in Fib(n).

$$\mathbf{T(n) = T(n-1) + T(n-2) + 1} \text{ and } \mathbf{T(0) = T(1) = 0}$$

Recursive Algorithm for Fibonacci Numbers

Question: Given $\mathbf{n}$, compute $\mathbf{F(n)}$.

```
Fib(n):
        if (n = 0)
            return 0
        else if (n = 1)
            return 1
        else
            return Fib(n − 1) + Fib(n − 2)
```

Running time? Let $\mathbf{T(n)}$ be the number of additions in Fib(n).

$$T(n) = T(n-1) + T(n-2) + 1 \text{ and } T(0) = T(1) = 0$$

Recursive Algorithm for Fibonacci Numbers

Question: Given $\mathbf{n}$, compute $\mathbf{F(n)}$.

```
Fib(n):
        if (n = 0)
            return 0
        else if (n = 1)
            return 1
        else
            return Fib(n − 1) + Fib(n − 2)
```

Running time? Let $\mathbf{T(n)}$ be the number of additions in Fib(n).

$$\mathbf{T(n) = T(n-1) + T(n-2) + 1} \text{ and } \mathbf{T(0) = T(1) = 0}$$

Recursive Algorithm for Fibonacci Numbers

Question: Given $\mathbf{n}$, compute $\mathbf{F(n)}$.

```
Fib(n):
        if (n = 0)
            return 0
        else if (n = 1)
            return 1
        else
            return Fib(n − 1) + Fib(n − 2)
```

Running time? Let $\mathbf{T(n)}$ be the number of additions in Fib(n).

$$\mathbf{T(n) = T(n-1) + T(n-2) + 1} \text{ and } \mathbf{T(0) = T(1) = 0}$$

Roughly same as $\mathbf{F(n)}$

$$\mathbf{T(n) = \Theta(\phi^n)}$$

The number of additions is exponential in $\mathbf{n}$. Can we do better?

Running time of binom?

```
binom(t, b)     // computes (t choose b)
// Using the identity:  (t choose b) = (t-1 choose b-1) + (t-1 choose b)
    if t = 0 then return 0
    if b = t or b = 0 then return 1
    return binom(t − 1, b − 1) + binom(t − 1, b).
```

Assuming each arithmetic operation takes $O(1)$ time, the running time of $\text{binom}(n, \lfloor n/2 \rfloor)$ is

(A) $\Theta(1)$.

(B) $\Theta(n)$.

(C) $\Theta(n \log n)$.

(D) $\Theta(n^2)$.

(E) $\Theta\left(\binom{n}{\lfloor n/2 \rfloor}\right)$.

An iterative algorithm for Fibonacci numbers

```
FibIter(n):
        if (n = 0) then
            return 0
        if (n = 1) then
            return 1
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] ⇐ F[i − 1] + F[i − 2]
        return F[n]
```

What is the running time of the algorithm? $O(n)$ additions.

An iterative algorithm for Fibonacci numbers

```
FibIter(n):
        if (n = 0) then
            return 0
        if (n = 1) then
            return 1
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] ⇐ F[i − 1] + F[i − 2]
        return F[n]
```

What is the running time of the algorithm? $O(n)$ additions.

An iterative algorithm for Fibonacci numbers

```
FibIter(n):
        if (n = 0) then
            return 0
        if (n = 1) then
            return 1
        F[0] = 0
        F[1] = 1
        for i = 2 to n do
            F[i] ⇐ F[i − 1] + F[i − 2]
        return F[n]
```

What is the running time of the algorithm? **O(n)** additions.

What is the difference?

1. Recursive algorithm is computing the same numbers again and again.
2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

Dynamic Programming:

Fnding a recursion that can be *effectively/efficiently* memoized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.

What is the difference?

1. Recursive algorithm is computing the same numbers again and again.
2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

Dynamic Programming:

Fnding a recursion that can be *effectively/efficiently* memoized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.

What is the difference?

1. Recursive algorithm is computing the same numbers again and again.
2. Iterative algorithm is storing computed values and building bottom up the final value. Memoization.

Dynamic Programming:

Fnding a recursion that can be *effectively/efficiently* memoized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (Fib(n) was previously computed)
            return stored value of Fib(n)
        else
            return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?
Two methods: explicitly and implicitly (via data structure)

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (Fib(n) was previously computed)
            return stored value of Fib(n)
        else
            return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?
Two methods: explicitly and implicitly (via data structure)

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (Fib(n) was previously computed)
            return stored value of Fib(n)
        else
            return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?

Two methods: explicitly and implicitly (via data structure)

Automatic Memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (Fib(n) was previously computed)
            return stored value of Fib(n)
        else
            return Fib(n − 1) + Fib(n − 2)
```

How do we keep track of previously computed values?
Two methods: explicitly and implicitly (via data structure)

Automatic explicit memoization

Initialize table/array **M** of size **n** such that $\mathbf{M[i] = -1}$ for $\mathbf{i = 0, \ldots, n}$.

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (M[n] ≠ −1) (* M[n] has stored value of Fib(n) *)
            return M[n]
        M[n] ⇐ Fib(n − 1) + Fib(n − 2)
        return M[n]
```

Need to know upfront the number of subproblems to allocate memory

Automatic explicit memoization

Initialize table/array $\mathbf{M}$ of size $\mathbf{n}$ such that $\mathbf{M[i] = -1}$ for $\mathbf{i = 0, \ldots, n}$.

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (M[n] ≠ −1) (* M[n] has stored value of Fib(n) *)
            return M[n]
        M[n] ⇐ Fib(n − 1) + Fib(n − 2)
        return M[n]
```

Need to know upfront the number of subproblems to allocate memory

Automatic implicit memoization

Initialize a (dynamic) dictionary data structure **D** to empty

```
Fib(n):
        if (n = 0)
            return 0
        if (n = 1)
            return 1
        if (n is already in D)
            return value stored with n in D
        val ⇐ Fib(n − 1) + Fib(n − 2)
        Store (n, val) in D
        return val
```

Explicit vs Implicit Memoization

1. Explicit memoization or iterative algorithm preferred if one can analyze problem ahead of time. Allows for efficient memory allocation and access.
2. Implicit and automatic memoization used when problem structure or algorithm is either not well understood or in fact unknown to the underlying system.
    1. Need to pay overhead of data-structure.
    2. Functional languages such as LISP automatically do memoization, usually via hashing based dictionaries.

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $\mathbf{O(n)}$ time?

1. input is $\mathbf{n}$ and hence input size is $\mathbf{\Theta(\log n)}$
2. output is $\mathbf{F(n)}$ and output size is $\mathbf{\Theta(n)}$. Why?
3. Hence output size is exponential in input size so no polynomial time algorithm possible!
4. Running time of iterative algorithm: $\mathbf{\Theta(n)}$ additions but number sizes are $\mathbf{O(n)}$ bits long! Hence total time is $\mathbf{O(n^2)}$, in fact $\mathbf{\Theta(n^2)}$. Why?
5. Running time of recursive algorithm is $\mathbf{O(n\phi^n)}$ but can in fact shown to be $\mathbf{O(\phi^n)}$ by being careful. Doubly exponential in input size and exponential even in output size.

Back to Fibonacci Numbers

Is the iterative algorithm a *polynomial* time algorithm? Does it take $\mathbf{O(n)}$ time?

1. input is $\mathbf{n}$ and hence input size is $\mathbf{\Theta(\log n)}$
2. output is $\mathbf{F(n)}$ and output size is $\mathbf{\Theta(n)}$. Why?
3. Hence output size is exponential in input size so no polynomial time algorithm possible!
4. Running time of iterative algorithm: $\mathbf{\Theta(n)}$ additions but number sizes are $\mathbf{O(n)}$ bits long! Hence total time is $\mathbf{O(n^2)}$, in fact $\mathbf{\Theta(n^2)}$. Why?
5. Running time of recursive algorithm is $\mathbf{O(n\phi^n)}$ but can in fact shown to be $\mathbf{O(\phi^n)}$ by being careful. Doubly exponential in input size and exponential even in output size.

How many distinct calls?

```
binom(t, b)     // computes (t choose b)
    if t = 0 then return 0
    if b = t or b = 0 then return 1
    return binom(t − 1, b − 1) + binom(t − 1, b).
```

How many distinct calls does **binom(n, $\lfloor n/2 \rfloor$)** makes during its recursive execution?

(A) $\Theta(1)$.

(B) $\Theta(n)$.

(C) $\Theta(n \log n)$.

(D) $\Theta(n^2)$.

(E) $\Theta\left(\binom{n}{\lfloor n/2 \rfloor}\right)$.

That is, if the algorithm calls recursively **binom(17, 5)** about 5000 times during the computation, we count this is a single distinct call.

Running time of memoized binom?

```
D:  Initially an empty dictionary.
binomM(t, b)     // computes (t choose b)
    if b = t then return 1
    if b = 0 then return 0
    if D[t, b] is defined then return D[t, b]
    D[t, b] ⇐ binomM(t − 1, b − 1) + binomM(t − 1, b).
    return D[t, b]
```

Assuming that every arithmetic operation takes $O(1)$ time, What is the running time of $\text{binomM}(n, \lfloor n/2 \rfloor)$?

(A) $\Theta(1)$.

(B) $\Theta(n)$.

(C) $\Theta(n^2)$.

(D) $\Theta(n^3)$.

(E) $\Theta\left(\binom{n}{\lfloor n/2 \rfloor}\right)$.

Part III

Brute Force Search, Recursion and Backtracking

Maximum Independent Set in a Graph

Definition

Given undirected graph $\mathbf{G = (V, E)}$ a subset of nodes $\mathbf{S \subseteq V}$ is an independent set (also called a stable set) if for there are no edges between nodes in $\mathbf{S}$. That is, if $\mathbf{u, v \in S}$ then $\mathbf{(u, v) \notin E}$.

Some independent sets in graph above:

Maximum Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$

Goal Find maximum sized independent set in $\mathbf{G}$

Maximum Weight Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$, weights $\mathbf{w(v) \geq 0}$ for $\mathbf{v \in V}$

Goal Find maximum weight independent set in $\mathbf{G}$

Maximum Weight Independent Set Problem

1. No one knows an *efficient* (polynomial time) algorithm for this problem
2. Problem is **NP-Complete** and it is *believed* that there is no polynomial time algorithm

Brute-force algorithm:

Try all subsets of vertices.

Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

```
MaxIndSet(G = (V, E)):
    max = 0
    for each subset S ⊆ V do
        check if S is an independent set
        if S is an independent set and w(S) > max then
            max = w(S)
    Output max
```

Running time: suppose G has n vertices and m edges

1. 2^n subsets of V
2. checking each subset S takes $O(m)$ time
3. total time is $O(m2^n)$

Brute-force enumeration

Algorithm to find the size of the maximum weight independent set.

```
MaxIndSet(G = (V, E)):
    max = 0
    for each subset S ⊆ V do
        check if S is an independent set
        if S is an independent set and w(S) > max then
            max = w(S)
    Output max
```

Running time: suppose G has n vertices and m edges

1. 2^n subsets of V
2. checking each subset S takes $O(m)$ time
3. total time is $O(m2^n)$

A Recursive Algorithm

Let $\mathbf{V} = \{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$.
For a vertex $\mathbf{u}$ let $\mathbf{N(u)}$ be its neighbors.

Observation

$\mathbf{v_n}$: *Vertex in the graph.*
One of the following two cases is true

Case 1 $\mathbf{v_n}$ *is in* some *maximum independent set.*

Case 2 $\mathbf{v_n}$ *is in* no *maximum independent set.*

```
RecursiveMIS(G):
    if G is empty then Output 0
    a = RecursiveMIS(G − v_n)
    b = w(v_n) + RecursiveMIS(G − v_n − N(v_n))
    Output max(a, b)
```

A Recursive Algorithm

Let $\mathbf{V} = \{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$.
For a vertex $\mathbf{u}$ let $\mathbf{N(u)}$ be its neighbors.

Observation

$\mathbf{v_n}$: *Vertex in the graph.*
One of the following two cases is true

Case 1 $\mathbf{v_n}$ *is in* some *maximum independent set.*

Case 2 $\mathbf{v_n}$ *is in* no *maximum independent set.*

```
RecursiveMIS(G):
    if G is empty then Output 0
    a = RecursiveMIS(G − v_n)
    b = w(v_n) + RecursiveMIS(G − v_n − N(v_n))
    Output max(a, b)
```

A Recursive Algorithm

Let $\mathbf{V} = \{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\}$.
For a vertex $\mathbf{u}$ let $\mathbf{N(u)}$ be its neighbors.

Observation

$\mathbf{v_n}$: *Vertex in the graph.*
One of the following two cases is true

Case 1 $\mathbf{v_n}$ *is in* some *maximum independent set.*

Case 2 $\mathbf{v_n}$ *is in* no *maximum independent set.*

```
RecursiveMIS(G):
    if G is empty then Output 0
    a = RecursiveMIS(G − v_n)
    b = w(v_n) + RecursiveMIS(G − v_n − N(v_n))
    Output max(a, b)
```

Recursive Algorithms

..for Maximum Independent Set

Running time:

$$T(n) = T(n-1) + T\Big(n - 1 - \deg(v_n)\Big) + O(1 + \deg(v_n))$$

where $\deg(v_n)$ is the degree of v_n. $T(0) = T(1) = 1$ is base case.

Worst case is when $\deg(v_n) = 0$ when the recurrence becomes

$$T(n) = 2T(n-1) + O(1)$$

Solution to this is $T(n) = O(2^n)$.

Backtrack Search via Recursion

1. Recursive algorithm generates a tree of computation where each node is a smaller problem (subproblem)
2. Simple recursive algorithm computes/explores the whole tree blindly in some order.
3. Backtrack search is a way to explore the tree intelligently to prune the search space
    1. Some subproblems may be so simple that we can stop the recursive algorithm and solve it directly by some other method
    2. Memoization to avoid recomputing same problem
    3. Stop the recursion at a subproblem if it is clear that there is no need to explore further.
    4. Leads to a number of heuristics that are widely used in practice although the worst case running time may still be exponential.

Example

Notes

Notes

Notes

Notes

Dynamic Programming

Lecture 8

February 13, 2014

Part I

Longest Increasing Subsequence

Sequences

Definition

Sequence: an ordered list $a_1, a_2, \ldots, a_n$. **Length** of a sequence is number of elements in the list.

Definition

$a_{i_1}, \ldots, a_{i_k}$ is a **subsequence** of $a_1, \ldots, a_n$ if $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

Definition

A sequence is **increasing** if $a_1 < a_2 < \ldots < a_n$. It is **non-decreasing** if $a_1 \leq a_2 \leq \ldots \leq a_n$. Similarly **decreasing** and **non-increasing**.

Sequences

Example...

Example

1. Sequence: $\mathbf{6, 3, 5, 2, 7, 8, 1, 9}$
2. Subsequence of above sequence: $\mathbf{5, 2, 1}$
3. Increasing sequence: $\mathbf{3, 5, 9, 17, 54}$
4. Decreasing sequence: $\mathbf{34, 21, 7, 5, 1}$
5. Increasing subsequence of the first sequence: $\mathbf{2, 7, 9}$.

Longest Increasing Subsequence Problem

Input A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal Find an **increasing subsequence** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8

Longest Increasing Subsequence Problem

Input A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal Find an **increasing subsequence** $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8

Naïve Enumeration

Assume $a_1, a_2, \ldots, a_n$ is contained in an array A

```
algLISNaive(A[1..n]):
    max = 0
    for each subsequence B of A do
        if B is increasing and |B| > max then
            max = |B|

    Output max
```

Running time: $O(n2^n)$.

2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

Naïve Enumeration

Assume $a_1, a_2, \ldots, a_n$ is contained in an array A

```
algLISNaive(A[1..n]):
    max = 0
    for each subsequence B of A do
        if B is increasing and |B| > max then
            max = |B|

    Output max
```

Running time: $O(n2^n)$.

2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

Naïve Enumeration

Assume $a_1, a_2, \ldots, a_n$ is contained in an array A

```
algLISNaive(A[1..n]):
    max = 0
    for each subsequence B of A do
        if B is increasing and |B| > max then
            max = |B|

    Output max
```

Running time: $O(n2^n)$.

2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS(**A[1..n]**):

1. Case 1: Does not contain **A[n]** in which case LIS(**A[1..n]**) = LIS(**A[1..(n − 1)]**)
2. Case 2: contains **A[n]** in which case LIS(**A[1..n]**) is not so clear.

Observation

if **A[n]** *is in the longest increasing subsequence then all the elements before it must be smaller.*

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS(**A[1..n]**):

1. Case 1: Does not contain **A[n]** in which case LIS(**A[1..n]**) = LIS(**A[1..(n − 1)]**)
2. Case 2: contains **A[n]** in which case LIS(**A[1..n]**) is not so clear.

Observation

if **A[n]** *is in the longest increasing subsequence then all the elements before it must be smaller.*

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS(**A[1..n]**):

1. Case 1: Does not contain **A[n]** in which case LIS(**A[1..n]**) = LIS(**A[1..(n − 1)]**)
2. Case 2: contains **A[n]** in which case LIS(**A[1..n]**) is not so clear.

Observation

if **A[n]** *is in the longest increasing subsequence then all the elements before it must be smaller.*

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS(**A[1..n]**):

1. Case 1: Does not contain **A[n]** in which case LIS(**A[1..n]**) = LIS(**A[1..(n − 1)]**)
2. Case 2: contains **A[n]** in which case LIS(**A[1..n]**) is not so clear.

Observation

if **A[n]** *is in the longest increasing subsequence then all the elements before it must be smaller.*

Recursive Approach: Take 1

```
algLIS(A[1..n]):
    if (n = 0) then return 0
    m = algLIS(A[1..(n − 1)])
    B is subsequence of A[1..(n − 1)] with
            only elements less than A[n]
    (* let h be size of B, h ≤ n − 1 *)
    m = max(m, 1 + algLIS(B[1..h]))
    Output m
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(n)$.

Easy to see that $T(n)$ is $O(n2^n)$.

Recursive Approach: Take 1

```
algLIS(A[1..n]):
    if (n = 0) then return 0
    m = algLIS(A[1..(n − 1)])
    B is subsequence of A[1..(n − 1)] with
            only elements less than A[n]
    (* let h be size of B, h ≤ n − 1 *)
    m = max(m, 1 + algLIS(B[1..h]))
    Output m
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(n)$.

Easy to see that $T(n)$ is $O(n2^n)$.

Recursive Approach: Take 1

```
algLIS(A[1..n]):
    if (n = 0) then return 0
    m = algLIS(A[1..(n − 1)])
    B is subsequence of A[1..(n − 1)] with
            only elements less than A[n]
    (* let h be size of B, h ≤ n − 1 *)
    m = max(m, 1 + algLIS(B[1..h]))
    Output m
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(n)$.
Easy to see that $T(n)$ is $O(n2^n)$.

How many different recursive calls does $\mathbf{algLIS_1(A[1..n])}$ <u>really</u> make?

```
algLIS(A[1..n]):
    if (n = 0) then return 0
    m = algLIS(A[1..(n − 1)])
    B is subsequence of A[1..(n − 1)] with
            only elements less than A[n]
    (* let h be size of B, h ≤ n − 1 *)
    m = max(m, 1 + algLIS(B[1..h]))
    Output m
```

(A) $\Theta(n^2)$

(B) $\Theta(2^n)$

(C) $\Theta(n2^n)$

(D) $\Theta(2^{n^2})$

(E) $\Theta(n^n)$

Recursive Approach: Take 2

$\mathrm{LIS}(\mathbf{A[1..n]})$:

1. Case 1: Does not contain $\mathbf{A[n]}$ in which case $\mathrm{LIS}(\mathbf{A[1..n]}) = \mathrm{LIS}(\mathbf{A[1..(n-1)]})$
2. Case 2: contains $\mathbf{A[n]}$ in which case $\mathrm{LIS}(\mathbf{A[1..n]})$ is not so clear.

Observation

For second case we want to find a subsequence in $\mathbf{A[1..(n-1)]}$ *that is restricted to numbers less than* $\mathbf{A[n]}$*. This suggests that a more general problem is* $\mathbf{LIS_smaller(A[1..n], x)}$ *which gives the longest increasing subsequence in* $\mathbf{A}$ *where each number in the sequence is less than* $\mathbf{x}$*.*

Recursive Approach: Take 2

LIS_smaller(A[1..n], x) : length of longest increasing subsequence in **A[1..n]** with all numbers in subsequence less than **x**

```
LIS_smaller(A[1..n], x):
    if (n = 0) then return 0
    m = LIS_smaller(A[1..(n − 1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS_smaller(A[1..(n − 1)], A[n]))
    Output m
```

```
LIS(A[1..n]):
        return LIS_smaller(A[1..n], ∞)
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(1)$.

Question: Is there any advantage?

Recursive Approach: Take 2

LIS_smaller(A[1..n], x) : length of longest increasing subsequence in **A[1..n]** with all numbers in subsequence less than **x**

```
LIS_smaller(A[1..n], x):
    if (n = 0) then return 0
    m = LIS_smaller(A[1..(n − 1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS_smaller(A[1..(n − 1)], A[n]))
    Output m
```

```
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(1)$.

Question: Is there any advantage?

Recursive Approach: Take 2

LIS_smaller(A[1..n], x) : length of longest increasing subsequence in **A[1..n]** with all numbers in subsequence less than **x**

```
LIS_smaller(A[1..n], x):
    if (n = 0) then return 0
    m = LIS_smaller(A[1..(n − 1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS_smaller(A[1..(n − 1)], A[n]))
    Output m
```

```
LIS(A[1..n]):
        return LIS_smaller(A[1..n], ∞)
```

Recursion for running time: $T(n) \leq 2T(n-1) + O(1)$.

Question: Is there any advantage?

Recursive Algorithm: Take 2

Observation

The number of different *subproblems generated by* **LIS_smaller(A[1..n], x)** *is* $O(n^2)$.

Memoization the recursive algorithm leads to an $O(n^2)$ running time!

Question: What are the recursive subproblem generated by **LIS_smaller(A[1..n], x)**?

1. For $0 \leq i < n$ **LIS_smaller(A[1..i], y)** where **y** is either **x** or one of $A[i+1], \ldots, A[n]$.

Observation

previous recursion also generates only $O(n^2)$ *subproblems. Slightly harder to see.*

Recursive Algorithm: Take 2

Observation

The number of different *subproblems generated by* **LIS_smaller(A[1..n], x)** *is* $O(n^2)$.

Memoization the recursive algorithm leads to an $O(n^2)$ running time!

Question: What are the recursive subproblem generated by **LIS_smaller(A[1..n], x)**?

1. For $0 \leq i < n$ **LIS_smaller(A[1..i], y)** where **y** is either **x** or one of $A[i+1], \ldots, A[n]$.

Observation

previous recursion also generates only $O(n^2)$ *subproblems. Slightly harder to see.*

Recursive Algorithm: Take 2

Observation

The number of different *subproblems generated by* **LIS_smaller(A[1..n], x)** *is* $O(n^2)$.

Memoization the recursive algorithm leads to an $O(n^2)$ running time!

Question: What are the recursive subproblem generated by **LIS_smaller(A[1..n], x)?**

1. For $0 \leq i < n$ **LIS_smaller(A[1..i], y)** where **y** is either **x** or one of $A[i+1], \ldots, A[n]$.

Observation

previous recursion also generates only $O(n^2)$ *subproblems. Slightly harder to see.*

Recursive Algorithm: Take 2

Observation

The number of different *subproblems generated by* **LIS_smaller(A[1..n], x)** *is* $O(n^2)$.

Memoization the recursive algorithm leads to an $O(n^2)$ running time!

Question: What are the recursive subproblem generated by **LIS_smaller(A[1..n], x)?**

1. For $0 \leq i < n$ **LIS_smaller(A[1..i], y)** where **y** is either **x** or one of $A[i+1], \ldots, A[n]$.

Observation

previous recursion also generates only $O(n^2)$ *subproblems. Slightly harder to see.*

Recursive Algorithm: Take 2

Observation

The number of different *subproblems generated by* **LIS_smaller(A[1..n], x)** *is* $O(n^2)$.

Memoization the recursive algorithm leads to an $O(n^2)$ running time!

Question: What are the recursive subproblem generated by **LIS_smaller(A[1..n], x)?**

1. For $0 \leq i < n$ **LIS_smaller(A[1..i], y)** where y is either x or one of $A[i+1], \ldots, A[n]$.

Observation

previous recursion also generates only $O(n^2)$ *subproblems. Slightly harder to see.*

Increasing/decreasing sequence

Given a sequence $a_1, a_2, \ldots, a_n$ of numbers, there is always either an increasing or decreasing subsequence of length (longer is better):

(A) $\Theta(1)$.

(B) $\Theta(\log \log n)$.

(C) $\Theta(\log n)$.

(D) $\Theta(\sqrt{n})$.

(E) $\Theta(n)$.

Recursive Algorithm: Take 3

Definition

LISEnding(A[1..n]): length of longest increasing sub-sequence that *ends* in **A[n]**.

Question: can we obtain a recursive expression?

$$\mathbf{LISEnding(A[1..n])} = \max_{\mathbf{i:A[i]<A[n]}} \left(\mathbf{1 + LISEnding(A[1..i])}\right)$$

Recursive Algorithm: Take 3

Definition

LISEnding(A[1..n]): length of longest increasing sub-sequence that *ends* in **A[n]**.

Question: can we obtain a recursive expression?

$$\mathbf{LISEnding(A[1..n])} = \max_{\mathbf{i:A[i]<A[n]}} \Big(\mathbf{1 + LISEnding(A[1..i])}\Big)$$

Recursive Algorithm: Take 3

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))

    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1 ... i])
```

Question:

How many distinct subproblems generated by **LIS_ending_alg(A[1..n])**? **n**.

Recursive Algorithm: Take 3

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))

    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1 ... i])
```

Question:

How many distinct subproblems generated by **LIS_ending_alg(A[1..n])**? n.

Recursive Algorithm: Take 3

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))

    return m
```

```
LIS(A[1..n]):
        return max_{i=1}^{n} LIS_ending_alg(A[1 ... i])
```

Question:

How many distinct subproblems generated by **LIS_ending_alg(A[1..n])**? **n**.

Iterative Algorithm via Memoization

Compute the values **LIS_ending_alg(A[1..i])** iteratively in a bottom up fashion.

```
LIS_ending_alg(A[1..n]):
    Array L[1..n]  (* L[i] = value of LIS_ending_alg(A[1..i]) *)
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
    return L
```

```
LIS(A[1..n]):
        L = LIS_ending_alg(A[1..n])
        return the maximum value in L
```

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
    Array L[1..n]  (* L[i] stores the value LISEnding(A[1..i]) *)
    m = 0
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
    return m
```

Correctness: Via induction following the recursion

Running time: $O(n^2)$, Space: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
    Array L[1..n]  (* L[i] stores the value LISEnding(A[1..i]) *)
    m = 0
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
    return m
```

Correctness: Via induction following the recursion

Running time: $O(n^2)$, Space: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
    Array L[1..n]  (* L[i] stores the value LISEnding(A[1..i]) *)
    m = 0
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
    return m
```

Correctness: Via induction following the recursion

Running time: $O(n^2)$, Space: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
    Array L[1..n]  (* L[i] stores the value LISEnding(A[1..i]) *)
    m = 0
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
    return m
```

Correctness: Via induction following the recursion

Running time: $O(n^2)$, Space: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
    Array L[1..n]  (* L[i] stores the value LISEnding(A[1..i]) *)
    m = 0
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
    return m
```

Correctness: Via induction following the recursion

Running time: $O(n^2)$, Space: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

```
LIS(A[1..n]):
    Array L[1..n]  (* L[i] stores the value LISEnding(A[1..i]) *)
    m = 0
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
        m = max(m, L[i])
    return m
```

Correctness: Via induction following the recursion

Running time: $O(n^2)$, Space: $\Theta(n)$

Example

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Longest increasing subsequence: 3, 5, 7, 8

$L[1] = 1$

$L[2] = 1$

$L[3] = \max\{1, 1+1\} = 2$

$L[4] = 1$

$L[5] = \max\{1, 1+1, 1+1, 1+2, 1+1\} = 3$

1. **L[i]** is value of longest increasing subsequence ending in **A[i]**
2. Recursive algorithm computes **L[i]** from **L[1]** to **L[i − 1]**
3. Iterative algorithm builds up the values from **L[1]** to **L[n]**

Example

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Longest increasing subsequence: 3, 5, 7, 8

1. **L[i]** is value of longest increasing subsequence ending in **A[i]**
2. Recursive algorithm computes **L[i]** from **L[1]** to **L[i − 1]**
3. Iterative algorithm builds up the values from **L[1]** to **L[n]**

Memoizing LIS_smaller

```
LIS(A[1..n]):
    A[n+1] = ∞ (* add a sentinel at the end *)
    Array L[(n+1),(n+1)] (* two-dimensional array*)
        (* L[i,j] for j ≥ i stores the value LIS_smaller(A[1..i],A[j]) *
    for j = 1 to n+1 do
        L[0,j] = 0
    for i = 1 to n+1 do
        for j = i to n+1 do
            L[i,j] = L[i−1,j]
            if (A[i] < A[j]) then
                L[i,j] = max(L[i,j], 1 + L[i−1,i])

    return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

Running time: $O(n^2)$, Space: $\Theta(n^2)$

Memoizing LIS_smaller

```
LIS(A[1..n]):
    A[n+1] = ∞ (* add a sentinel at the end *)
    Array L[(n+1),(n+1)] (* two-dimensional array*)
        (* L[i,j] for j ≥ i stores the value LIS_smaller(A[1..i],A[j]) *
    for j = 1 to n+1 do
        L[0,j] = 0
    for i = 1 to n+1 do
        for j = i to n+1 do
            L[i,j] = L[i−1,j]
            if (A[i] < A[j]) then
                L[i,j] = max(L[i,j], 1 + L[i−1,i])

    return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

Running time: $O(n^2)$, Space: $\Theta(n^2)$

Memoizing LIS_smaller

```
LIS(A[1..n]):
    A[n+1] = ∞ (* add a sentinel at the end *)
    Array L[(n+1),(n+1)] (* two-dimensional array*)
        (* L[i,j] for j ≥ i stores the value LIS_smaller(A[1..i],A[j]) *
    for j = 1 to n+1 do
        L[0,j] = 0
    for i = 1 to n+1 do
        for j = i to n+1 do
            L[i,j] = L[i−1,j]
            if (A[i] < A[j]) then
                L[i,j] = max(L[i,j], 1 + L[i−1,i])

    return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

Running time: $O(n^2)$, Space: $\Theta(n^2)$

Memoizing LIS_smaller

```
LIS(A[1..n]):
    A[n+1] = ∞ (* add a sentinel at the end *)
    Array L[(n+1),(n+1)] (* two-dimensional array*)
        (* L[i,j] for j ≥ i stores the value LIS_smaller(A[1..i],A[j]) *
    for j = 1 to n+1 do
        L[0,j] = 0
    for i = 1 to n+1 do
        for j = i to n+1 do
            L[i,j] = L[i−1,j]
            if (A[i] < A[j]) then
                L[i,j] = max(L[i,j], 1 + L[i−1,i])

    return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

Running time: $O(n^2)$, Space: $\Theta(n^2)$

Memoizing LIS_smaller

```
LIS(A[1..n]):
    A[n+1] = ∞ (* add a sentinel at the end *)
    Array L[(n+1),(n+1)] (* two-dimensional array*)
        (* L[i,j] for j ≥ i stores the value LIS_smaller(A[1..i],A[j]) *
    for j = 1 to n+1 do
        L[0,j] = 0
    for i = 1 to n+1 do
        for j = i to n+1 do
            L[i,j] = L[i−1,j]
            if (A[i] < A[j]) then
                L[i,j] = max(L[i,j], 1 + L[i−1,i])

    return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

Running time: $O(n^2)$, Space: $\Theta(n^2)$

Memoizing LIS_smaller

```
LIS(A[1..n]):
    A[n+1] = ∞ (* add a sentinel at the end *)
    Array L[(n+1),(n+1)] (* two-dimensional array*)
        (* L[i,j] for j ≥ i stores the value LIS_smaller(A[1..i],A[j]) *
    for j = 1 to n+1 do
        L[0,j] = 0
    for i = 1 to n+1 do
        for j = i to n+1 do
            L[i,j] = L[i−1,j]
            if (A[i] < A[j]) then
                L[i,j] = max(L[i,j], 1 + L[i−1,i])

    return L[n,(n+1)]
```

Correctness: Via induction following the recursion (take 2)

Running time: $O(n^2)$, Space: $\Theta(n^2)$

Longest increasing subsequence

Another way to get quadratic time algorithm

1. $\mathbf{G} = (\{\mathbf{s}, \mathbf{1}, \ldots, \mathbf{n}\}, \{\})$: directed graph.
   1. $\forall \mathbf{i}, \mathbf{j}$: If $\mathbf{i} < \mathbf{j}$ and $\mathbf{A[i]} < \mathbf{A[j]}$ then add the edge $\mathbf{i} \rightarrow \mathbf{j}$ to $\mathbf{G}$.
   2. $\forall \mathbf{i}$: Add $\mathbf{s} \rightarrow \mathbf{i}$.
2. The graph $\mathbf{G}$ is a DAG. LIS corresponds to longest path in $\mathbf{G}$ starting at $\mathbf{s}$.
3. We know how to compute this in $\mathbf{O}(|\mathbf{V}(\mathbf{G})| + |\mathbf{E}(\mathbf{G})|) = \mathbf{O}(\mathbf{n}^2)$.

Comment: One can compute LIS in $\mathbf{O}(\mathbf{n} \log \mathbf{n})$ time with a bit more work.

Dynamic Programming

1. Find a "smart" recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.
2. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. This gives an upper bound on the total running time if we use automatic memoization.
3. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation. This leads to an explicit algorithm.
4. Optimize the resulting algorithm further

Part II

Weighted Interval Scheduling

Weighted Interval Scheduling

Input A set of jobs with start times, finish times and *weights* (or profits).

Goal Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!

Weighted Interval Scheduling

Input A set of jobs with start times, finish times and *weights* (or profits).

Goal Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!

Interval Scheduling

Greedy Solution

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight **1**.

Goal Schedule as many jobs as possible.

1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).

Interval Scheduling

Greedy Solution

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight **1**.

Goal Schedule as many jobs as possible.

1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).

Interval Scheduling

Greedy Solution

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight **1**.

Goal Schedule as many jobs as possible.

1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).

Interval Scheduling

Greedy Solution

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight **1**.

Goal Schedule as many jobs as possible.

1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).

Interval Scheduling

Greedy Solution

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight **1**.

Goal Schedule as many jobs as possible.

1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).

Interval Scheduling

Greedy Solution

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight **1**.

Goal Schedule as many jobs as possible.

1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).

Interval Scheduling

Greedy Solution

Input A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight **1**.

Goal Schedule as many jobs as possible.

1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).

Greedy Strategies

1. Earliest finish time first
2. Largest weight/profit first
3. Largest weight to length ratio first
4. Shortest length first
5. ...

None of the above strategies lead to an optimum solution.

Moral: Greedy strategies often don't work!

Greedy Strategies

1. Earliest finish time first
2. Largest weight/profit first
3. Largest weight to length ratio first
4. Shortest length first
5. ...

None of the above strategies lead to an optimum solution.

Moral: Greedy strategies often don't work!

Reduction to...

Max Weight Independent Set Problem

1. Given weighted interval scheduling instance $\mathbf{I}$ create an instance of max weight independent set on a graph $\mathbf{G(I)}$ as follows.
   1. For each interval $\mathbf{i}$ create a vertex $\mathbf{v_i}$ with weight $\mathbf{w_i}$.
   2. Add an edge between $\mathbf{v_i}$ and $\mathbf{v_j}$ if $\mathbf{i}$ and $\mathbf{j}$ overlap.
2. Claim: max weight independent set in $\mathbf{G(I)}$ has weight equal to max weight set of intervals in $\mathbf{I}$ that do not overlap

Reduction to...

Max Weight Independent Set Problem

1. Given weighted interval scheduling instance $\mathbf{I}$ create an instance of max weight independent set on a graph $\mathbf{G(I)}$ as follows.
   1. For each interval $\mathbf{i}$ create a vertex $\mathbf{v_i}$ with weight $\mathbf{w_i}$.
   2. Add an edge between $\mathbf{v_i}$ and $\mathbf{v_j}$ if $\mathbf{i}$ and $\mathbf{j}$ overlap.
2. Claim: max weight independent set in $\mathbf{G(I)}$ has weight equal to max weight set of intervals in $\mathbf{I}$ that do not overlap

Reduction to...

Max Weight Independent Set Problem

1. There is a reduction from **Weighted Interval Scheduling** to **Independent Set**.
2. Can use structure of original problem for efficient algorithm?
3. **Independent Set** in general is **NP-Complete**.

Reduction to...

Max Weight Independent Set Problem

1. There is a reduction from **Weighted Interval Scheduling** to **Independent Set**.
2. Can use structure of original problem for efficient algorithm?
3. **Independent Set** in general is **NP-Complete**.

Conventions

Definition

1. Let the requests be sorted according to finish time, i.e., $\mathbf{i} < \mathbf{j}$ implies $\mathbf{f_i} \leq \mathbf{f_j}$
2. Define $\mathbf{p(j)}$ to be the largest $\mathbf{i}$ (less than $\mathbf{j}$) such that job $\mathbf{i}$ and job $\mathbf{j}$ are not in conflict

Example

Towards a Recursive Solution

Observation

Consider an optimal schedule $\mathcal{O}$

Case $\mathbf{n} \in \mathcal{O}$: *None of the jobs between* $\mathbf{n}$ *and* $\mathbf{p(n)}$ *can be scheduled. Moreover* $\mathcal{O}$ *must contain an optimal schedule for the first* $\mathbf{p(n)}$ *jobs.*

Case $\mathbf{n} \notin \mathcal{O}$: $\mathcal{O}$ *is an optimal schedule for the first* $\mathbf{n} - \mathbf{1}$ *jobs.*

Towards a Recursive Solution

Observation

Consider an optimal schedule $\mathcal{O}$

Case $\mathbf{n} \in \mathcal{O}$: *None of the jobs between* $\mathbf{n}$ *and* $\mathbf{p(n)}$ *can be scheduled. Moreover* $\mathcal{O}$ *must contain an optimal schedule for the first* $\mathbf{p(n)}$ *jobs.*

Case $\mathbf{n} \notin \mathcal{O}$: $\mathcal{O}$ *is an optimal schedule for the first* $\mathbf{n-1}$ *jobs.*

A Recursive Algorithm

Let O_i be value of an optimal schedule for the first i jobs.

```
Schedule(n):
        if n = 0 then return 0
        if n = 1 then return w(v_1)
        O_{p(n)} ← Schedule(p(n))
        O_{n-1} ← Schedule(n - 1)
        if (O_{p(n)} + w(v_n) < O_{n-1}) then
            O_n = O_{n-1}
        else
            O_n = O_{p(n)} + w(v_n)
        return O_n
```

Time Analysis

Running time is $T(n) = T(p(n)) + T(n-1) + O(1)$ which is ...

A Recursive Algorithm

Let O_i be value of an optimal schedule for the first i jobs.

```
Schedule(n):
        if n = 0 then return 0
        if n = 1 then return w(v_1)
        O_p(n) ←Schedule(p(n))
        O_{n-1} ←Schedule(n - 1)
        if (O_p(n) + w(v_n) < O_{n-1}) then
            O_n = O_{n-1}
        else
            O_n = O_p(n) + w(v_n)
        return O_n
```

Time Analysis

Running time is $T(n) = T(p(n)) + T(n-1) + O(1)$ which is ...

The solution to the following recurrence is?

$$T(n) = T(n-2) + T(n-17) + 65$$

(A) $2^{\Theta(n)}$.

(B) $\Theta(n)$.

(C) 65.

(D) $\Theta(F_n)$, where F_n is the nth Fibonacci number..

(E) $\Theta(0)$.

Bad Example

Figure : Bad instance for recursive algorithm

Running time on this instance is

$$T(n) = T(n-1) + T(n-2) + O(1) = \Theta(\phi^n)$$

where $\phi \approx 1.618$ is the golden ratio.

Bad Example

Figure : Bad instance for recursive algorithm

Running time on this instance is

$$T(n) = T(n-1) + T(n-2) + O(1) = \Theta(\phi^n)$$

where $\phi \approx 1.618$ is the golden ratio.

Analysis of the Problem

Figure : Label of node indicates size of sub-problem. Tree of sub-problems grows very quickly

Memo(r)ization

Observation

1. *Number of different sub-problems in recursive algorithm is* $\mathbf{O(n)}$*; they are* $\mathbf{O_1, O_2, \ldots, O_{n-1}}$
2. *Exponential time is due to recomputation of solutions to sub-problems*

Solution

Store optimal solution to different sub-problems, and perform recursive call only if not already computed.

Recursive Solution with Memoization

```
schdIMem(j)
    if j = 0 then return 0
    if M[j] is defined then (* sub-problem already solved *)
        return M[j]
    if M[j] is not defined then
        M[j] = max( w(v_j) + schdIMem(p(j)),   schdIMem(j − 1) )
        return M[j]
```

Time Analysis

- Each invocation, $O(1)$ time plus: either return a computed value, or generate 2 recursive calls and fill one $M[\cdot]$
- Initially no entry of $M[]$ is filled. At the end all entries of $M[]$ are filled
- So total time is $O(n)$ (Assuming input is presorted...)

Recursive Solution with Memoization

```
schdIMem(j)
    if j = 0 then return 0
    if M[j] is defined then (* sub-problem already solved *)
        return M[j]
    if M[j] is not defined then
        M[j] = max( w(v_j) + schdIMem(p(j)),   schdIMem(j − 1) )
        return M[j]
```

Time Analysis

- Each invocation, **O(1)** time plus: either return a computed value, or generate 2 recursive calls and fill one **M[·]**
- Initially no entry of **M[]** is filled; at the end all entries of **M[]** are filled
- So total time is **O(n)** (Assuming input is presorted...)

Recursive Solution with Memoization

```
schdIMem(j)
    if j = 0 then return 0
    if M[j] is defined then (* sub-problem already solved *)
        return M[j]
    if M[j] is not defined then
        M[j] = max( w(v_j) + schdIMem(p(j)),   schdIMem(j − 1) )
        return M[j]
```

Time Analysis

- Each invocation, **O(1)** time plus: either return a computed value, or generate 2 recursive calls and fill one **M[·]**
- Initially no entry of **M[]** is filled; at the end all entries of **M[]** are filled
- So total time is **O(n)** (Assuming input is presorted...)

Recursive Solution with Memoization

```
schdIMem(j)
    if j = 0 then return 0
    if M[j] is defined then (* sub-problem already solved *)
        return M[j]
    if M[j] is not defined then
        M[j] = max( w(v_j) + schdIMem(p(j)),   schdIMem(j − 1) )
        return M[j]
```

Time Analysis

- Each invocation, $O(1)$ time plus: either return a computed value, or generate 2 recursive calls and fill one $M[\cdot]$
- Initially no entry of $M[]$ is filled; at the end all entries of $M[]$ are filled
- So total time is $O(n)$ (Assuming input is presorted...)

Recursive Solution with Memoization

```
schdIMem(j)
    if j = 0 then return 0
    if M[j] is defined then (* sub-problem already solved *)
        return M[j]
    if M[j] is not defined then
        M[j] = max( w(v_j) + schdIMem(p(j)),   schdIMem(j − 1) )
        return M[j]
```

Time Analysis

- Each invocation, **O(1)** time plus: either return a computed value, or generate 2 recursive calls and fill one **M[·]**
- Initially no entry of **M[]** is filled; at the end all entries of **M[]** are filled
- So total time is **O(n)** (Assuming input is presorted...)

Automatic Memoization

Fact

Many functional languages (like LISP) automatically do memoization for recursive function calls!

Back to Weighted Interval Scheduling

Iterative Solution

$$
\begin{array}{l}
M[0] = 0 \\
\textbf{for } i = 1 \texttt{ to } n \textbf{ do} \\
\qquad M[i] = \max\Big(w(v_i) + M[p(i)], M[i-1]\Big)
\end{array}
$$

M: table of subproblems

1. Implicitly dynamic programming fills the values of M.
2. Recursion determines order in which table is filled up.
3. Think of decomposing problem first (recursion) and then worry about setting up table — this comes naturally from recursion.

Back to Weighted Interval Scheduling

Iterative Solution

$$
\begin{array}{l}
\mathbf{M[0] = 0} \\
\textbf{for } \mathbf{i = 1} \texttt{ to } \mathbf{n} \textbf{ do} \\
\qquad \mathbf{M[i] = \max\Big(w(v_i) + M[p(i)], M[i-1]\Big)}
\end{array}
$$

M: table of subproblems

1. Implicitly dynamic programming fills the values of **M**.
2. Recursion determines order in which table is filled up.
3. Think of decomposing problem first (recursion) and then worry about setting up table — this comes naturally from recursion.

Example

$p(5) = 2,\ p(4) = 1,\ p(3) = 1,\ p(2) = 0,\ p(1) = 0$

$M[0] = 0 \quad D[0] = 0$

$M[1] = 30 \quad D[1] = 1$

$M[2] = \max\{M[1], 20 + 0\} = 30 \quad D[2] = 0$

$M[3] = \max\{M[2], 70 + M[1]\} = 100 \quad D[3] = 1$

$M[4] = \max\{M[3], 80 + M[1]\} = 110 \quad D[4] = 1$

$M[5] = \max\{M[4], 10 + M[2]\} = 110 \quad D[5] = 0$

Computing Solutions + First Attempt

1. Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n do
    M[i] = max(w(v_i) + M[p(i)], M[i − 1])
    if w(v_i) + M[p(i)] < M[i − 1] then
        S[i] = S[i − 1]
    else
        S[i] = S[p(i)] ∪ {i}
```

2. Naïvely updating **S[]** takes $O(n)$ time
3. Total running time is $O(n^2)$
4. Using pointers and linked lists running time can be improved to $O(n)$.

Computing Solutions + First Attempt

1. Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n do
    M[i] = max(w(v_i) + M[p(i)], M[i − 1])
    if w(v_i) + M[p(i)] < M[i − 1] then
        S[i] = S[i − 1]
    else
        S[i] = S[p(i)] ∪ {i}
```

2. Naïvely updating $S[]$ takes $O(n)$ time
3. Total running time is $O(n^2)$
4. Using pointers and linked lists running time can be improved to $O(n)$.

Computing Solutions + First Attempt

1. Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n do
    M[i] = max(w(v_i) + M[p(i)], M[i − 1])
    if w(v_i) + M[p(i)] < M[i − 1] then
        S[i] = S[i − 1]
    else
        S[i] = S[p(i)] ∪ {i}
```

2. Naïvely updating $S[]$ takes $O(n)$ time
3. Total running time is $O(n^2)$
4. Using pointers and linked lists running time can be improved to $O(n)$.

Computing Solutions + First Attempt

1. Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n do
    M[i] = max(w(v_i) + M[p(i)], M[i − 1])
    if w(v_i) + M[p(i)] < M[i − 1] then
        S[i] = S[i − 1]
    else
        S[i] = S[p(i)] ∪ {i}
```

2. Naïvely updating **S[]** takes **O(n)** time
3. Total running time is $O(n^2)$
4. Using pointers and linked lists running time can be improved to **O(n)**.

Computing Solutions + First Attempt

1. Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n do
    M[i] = max(w(v_i) + M[p(i)], M[i − 1])
    if w(v_i) + M[p(i)] < M[i − 1] then
        S[i] = S[i − 1]
    else
        S[i] = S[p(i)] ∪ {i}
```

2. Naïvely updating **S[]** takes **O(n)** time
3. Total running time is $\mathbf{O(n^2)}$
4. Using pointers and linked lists running time can be improved to **O(n)**.

Computing Solutions + First Attempt

1. Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

```
M[0] = 0
S[0] is empty schedule
for i = 1 to n do
    M[i] = max(w(v_i) + M[p(i)], M[i − 1])
    if w(v_i) + M[p(i)] < M[i − 1] then
        S[i] = S[i − 1]
    else
        S[i] = S[p(i)] ∪ {i}
```

2. Naïvely updating **S[]** takes **O(n)** time
3. Total running time is $\mathbf{O(n^2)}$
4. Using pointers and linked lists running time can be improved to **O(n)**.

Computing Implicit Solutions

Observation

Solution can be obtained from **M[]** *in* **O(n)** *time, without any additional information*

```
findSolution( j )
    if (j = 0) then return empty schedule
    if (v_j + M[p(j)] > M[j − 1]) then
        return findSolution(p(j)) ∪ {j}
    else
        return findSolution(j − 1)
```

Makes **O(n)** *recursive calls, so* **findSolution** *runs in* **O(n)** *time.*

Computing Implicit Solutions

A generic strategy for computing solutions in dynamic programming:

1. Keep track of the *decision* in computing the optimum value of a sub-problem. decision space depends on recursion
2. Once the optimum values are computed, go back and use the decision values to compute an optimum solution.

Question: What is the decision in computing **M[i]**?

A: Whether to include **i** or not.

Computing Implicit Solutions

A generic strategy for computing solutions in dynamic programming:

1. Keep track of the *decision* in computing the optimum value of a sub-problem. decision space depends on recursion
2. Once the optimum values are computed, go back and use the decision values to compute an optimum solution.

Question: What is the decision in computing **M[i]**?
A: Whether to include **i** or not.

Computing Implicit Solutions

```
M[0] = 0
for i = 1 to n do
    M[i] = max(v_i + M[p(i)], M[i − 1])
    if (v_i + M[p(i)] > M[i − 1]) then
        Decision[i] = 1 (* 1:  i included in solution M[i] *)
    else
        Decision[i] = 0 (* 0:  i not included in solution M[i] *)

S = ∅, i = n
while (i > 0) do
    if (Decision[i] = 1) then
        S = S ∪ {i}
        i = p(i)
    else
        i = i − 1
return S
```

Running time with memoization?

If we memoize the following function, what would be the running time of the resulting function, if we call **Confused**$(\mathbf{n}, \mathbf{n})$?

```
Confused(x, y)
    if x > y or x < 0 then if x = 0 then return 2y
    α = Confused(x − 1, y),   β = Confused(x − 1, y − 1),
    γ = Confused(x − 1, y − 1),   δ = Confused(x − 1, y − 17),
    μ = Confused(x − 32, y − 17),
    return 1 + max(α, β, γ, δ, μ)
```

(A) $\Theta(n)$

(B) $\Theta(n^2)$

(C) $\Theta(n^3)$

(D) $\Theta(n^4)$

(E) $\Theta(n^5)$

Notes

Notes

Notes

Notes

More Dynamic Programming

Lecture 9

February 18, 2014

What is the running time of the following?

Consider computing $\mathbf{f(x, y)}$ by recursive function + memoization.

$$\mathbf{f(x,y) = \sum_{i=1}^{x+y-1} x * f(x+y-i, i-1),}$$

$$\mathbf{f(0,y) = y \qquad f(x,0) = x.}$$

The resulting algorithm when computing $\mathbf{f(n, n)}$ would take:

(A) $\mathbf{O(n)}$

(B) $\mathbf{O(n \log n)}$

(C) $\mathbf{O(n^2)}$

(D) $\mathbf{O(n^3)}$

(E) The function is ill defined - it can not be computed.

Part I

Maximum Weighted Independent Set in Trees

Maximum Weight Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$ and weights $\mathbf{w(v) \geq 0}$ for each $\mathbf{v \in V}$

Goal Find maximum weight independent set in $\mathbf{G}$

Maximum weight independent set in above graph: $\{\mathbf{B, D}\}$

Maximum Weight Independent Set Problem

Input Graph $\mathbf{G = (V, E)}$ and weights $\mathbf{w(v) \geq 0}$ for each $\mathbf{v \in V}$

Goal Find maximum weight independent set in $\mathbf{G}$

Maximum weight independent set in above graph: $\mathbf{\{B, D\}}$

Maximum Weight Independent Set in a Tree

Input Tree $\mathbf{T = (V, E)}$ and weights $\mathbf{w(v) \geq 0}$ for each $\mathbf{v \in V}$

Goal Find maximum weight independent set in $\mathbf{T}$

Maximum weight independent set in above tree: ??

Independent set in a tree...

In a tree with **n** nodes, there is always an independent set of size (bigger is better [this is America!])

(A) $\Omega(1)$
(B) $\Omega(\log n)$
(C) $\Omega(\sqrt{n})$
(D) $n/2$
(E) $n - 5$

Towards a Recursive Solution

For an arbitrary graph $\mathbf{G}$:

1. Number vertices as $\mathbf{v_1, v_2, \ldots, v_n}$
2. Find recursively optimum solutions without $\mathbf{v_n}$ (recurse on $\mathbf{G - v_n}$) and with $\mathbf{v_n}$ (recurse on $\mathbf{G - v_n - N(v_n)}$ & include $\mathbf{v_n}$).
3. Saw that if graph $\mathbf{G}$ is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree? Natural candidate for $\mathbf{v_n}$ is root $\mathbf{r}$ of $\mathbf{T}$?

Towards a Recursive Solution

For an arbitrary graph $\mathbf{G}$:

1. Number vertices as $\mathbf{v_1, v_2, \ldots, v_n}$
2. Find recursively optimum solutions without $\mathbf{v_n}$ (recurse on $\mathbf{G - v_n}$) and with $\mathbf{v_n}$ (recurse on $\mathbf{G - v_n - N(v_n)}$ & include $\mathbf{v_n}$).
3. Saw that if graph $\mathbf{G}$ is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree? Natural candidate for $\mathbf{v_n}$ is root $\mathbf{r}$ of $\mathbf{T}$?

Towards a Recursive Solution

For an arbitrary graph $\mathbf{G}$:

1. Number vertices as $\mathbf{v_1, v_2, \ldots, v_n}$
2. Find recursively optimum solutions without $\mathbf{v_n}$ (recurse on $\mathbf{G - v_n}$) and with $\mathbf{v_n}$ (recurse on $\mathbf{G - v_n - N(v_n)}$ & include $\mathbf{v_n}$).
3. Saw that if graph $\mathbf{G}$ is arbitrary there was no good ordering that resulted in a small number of subproblems.

What about a tree? Natural candidate for $\mathbf{v_n}$ is root $\mathbf{r}$ of $\mathbf{T}$?

Towards a Recursive Solution

Natural candidate for $\mathbf{v_n}$ is root $\mathbf{r}$ of $\mathbf{T}$? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $\mathbf{r} \notin \mathcal{O}$: Then $\mathcal{O}$ contains an optimum solution for each subtree of $\mathbf{T}$ hanging at a child of $\mathbf{r}$.

Case $\mathbf{r} \in \mathcal{O}$: None of the children of $\mathbf{r}$ can be in $\mathcal{O}$. $\mathcal{O} - \{\mathbf{r}\}$ contains an optimum solution for each subtree of $\mathbf{T}$ hanging at a grandchild of $\mathbf{r}$.

Subproblems? Subtrees of $\mathbf{T}$ hanging at nodes in $\mathbf{T}$.

Towards a Recursive Solution

Natural candidate for $\mathbf{v_n}$ is root $\mathbf{r}$ of $\mathbf{T}$? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $\mathbf{r} \notin \mathcal{O}$: Then $\mathcal{O}$ contains an optimum solution for each subtree of $\mathbf{T}$ hanging at a child of $\mathbf{r}$.

Case $\mathbf{r} \in \mathcal{O}$: None of the children of $\mathbf{r}$ can be in $\mathcal{O}$. $\mathcal{O} - \{\mathbf{r}\}$ contains an optimum solution for each subtree of $\mathbf{T}$ hanging at a grandchild of $\mathbf{r}$.

Subproblems? Subtrees of $\mathbf{T}$ hanging at nodes in $\mathbf{T}$.

Towards a Recursive Solution

Natural candidate for $\mathbf{v_n}$ is root $\mathbf{r}$ of $\mathbf{T}$? Let $\mathcal{O}$ be an optimum solution to the whole problem.

Case $\mathbf{r} \notin \mathcal{O}$: Then $\mathcal{O}$ contains an optimum solution for each subtree of $\mathbf{T}$ hanging at a child of $\mathbf{r}$.

Case $\mathbf{r} \in \mathcal{O}$: None of the children of $\mathbf{r}$ can be in $\mathcal{O}$. $\mathcal{O} - \{\mathbf{r}\}$ contains an optimum solution for each subtree of $\mathbf{T}$ hanging at a grandchild of $\mathbf{r}$.

Subproblems? Subtrees of $\mathbf{T}$ hanging at nodes in $\mathbf{T}$.

A Recursive Solution

T(u): subtree of **T** hanging at node **u**
OPT(u): max weighted independent set value in **T(u)**

$$\mathbf{OPT(u)} = \max \begin{cases} \sum_{\mathbf{v} \text{ child of } \mathbf{u}} \mathbf{OPT(v)}, \\ \mathbf{w(u)} + \sum_{\mathbf{v} \text{ grandchild of } \mathbf{u}} \mathbf{OPT(v)} \end{cases}$$

A Recursive Solution

T(u): subtree of **T** hanging at node **u**
OPT(u): max weighted independent set value in **T(u)**

$$\mathbf{OPT(u)} = \mathbf{max} \begin{cases} \sum_{\mathbf{v} \text{ child of } \mathbf{u}} \mathbf{OPT(v)}, \\ \mathbf{w(u)} + \sum_{\mathbf{v} \text{ grandchild of } \mathbf{u}} \mathbf{OPT(v)} \end{cases}$$

Iterative Algorithm

1. Compute **OPT(u)** bottom up. To evaluate **OPT(u)** need to have computed values of all children and grandchildren of **u**
2. What is an ordering of nodes of a tree **T** to achieve above?
   Post-order traversal of a tree.

Iterative Algorithm

1. Compute **OPT(u)** bottom up. To evaluate **OPT(u)** need to have computed values of all children and grandchildren of **u**
2. What is an ordering of nodes of a tree **T** to achieve above? Post-order traversal of a tree.

Iterative Algorithm

```
MIS-Tree(T):
    Let v_1, v_2, ..., v_n be a post-order traversal of nodes of T
    for i = 1 to n do
        M[v_i] = max( Σ_{v_j child of v_i} M[v_j],
                      w(v_i) + Σ_{v_j grandchild of v_i} M[v_j] )
    return M[v_n] (* Note: v_n is the root of T *)
```

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T
for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T
Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(**T**):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array}\right)$$

return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Iterative Algorithm

MIS-Tree(T):

Let $v_1, v_2, \ldots, v_n$ be a post-order traversal of nodes of T

for $i = 1$ to n **do**

$$M[v_i] = \max\left(\begin{array}{c} \sum_{v_j \text{ child of } v_i} M[v_j], \\ w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \end{array} \right)$$

return $M[v_n]$ (* Note: v_n is the root of T *)

Space: $O(n)$ to store the value at each node of T

Running time:

1. Naive bound: $O(n^2)$ since each $M[v_i]$ evaluation may take $O(n)$ time and there are n evaluations.
2. Better bound: $O(n)$. A value $M[v_j]$ is accessed only by its parent and grand parent.

Example

Part II

Edit Distance and Sequence Alignment

Spell Checking Problem

Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?

What does nearness mean?

Question: Given two strings $\mathbf{x}_1\mathbf{x}_2 \ldots \mathbf{x}_n$ and $\mathbf{y}_1\mathbf{y}_2 \ldots \mathbf{y}_m$ what is a *distance* between them?

Edit Distance: minimum number of "edits" to transform $\mathbf{x}$ into $\mathbf{y}$.

Spell Checking Problem

Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?

What does nearness mean?

Question: Given two strings $\mathbf{x_1 x_2 \ldots x_n}$ and $\mathbf{y_1 y_2 \ldots y_m}$ what is a *distance* between them?

Edit Distance: minimum number of "edits" to transform $\mathbf{x}$ into $\mathbf{y}$.

Spell Checking Problem

Given a string "exponen" that is not in the dictionary, how should a spell checker suggest a *nearby* string?

What does nearness mean?

Question: Given two strings $\mathbf{x_1 x_2 \ldots x_n}$ and $\mathbf{y_1 y_2 \ldots y_m}$ what is a *distance* between them?

Edit Distance: minimum number of "edits" to transform $\mathbf{x}$ into $\mathbf{y}$.

Edit Distance

Definition

Edit distance between two words **X** and **Y** is the number of letter insertions, letter deletions and letter substitutions required to obtain **Y** from **X**.

Example

The edit distance between FOOD and MONEY is at most **4**:

$$\underline{\mathbf{F}}\mathbf{OOD} \rightarrow \mathbf{MO}\underline{\mathbf{O}}\mathbf{D} \rightarrow \mathbf{MON}\underline{\mathbf{O}}\mathbf{D} \rightarrow \mathbf{MONE}\underline{\mathbf{D}} \rightarrow \mathbf{MONEY}$$

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set $\mathbf{M}$ of pairs $(\mathbf{i}, \mathbf{j})$ such that each index appears at most once, and there is no "crossing": $\mathbf{i} < \mathbf{i}'$ and $\mathbf{i}$ is matched to $\mathbf{j}$ implies $\mathbf{i}'$ is matched to $\mathbf{j}' > \mathbf{j}$. In the above example, this is $\mathbf{M} = \{(1,1),(2,2),(3,3),(4,5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set $\mathbf{M}$ of pairs $(\mathbf{i}, \mathbf{j})$ such that each index appears at most once, and there is no "crossing": $\mathbf{i} < \mathbf{i}'$ and $\mathbf{i}$ is matched to $\mathbf{j}$ implies $\mathbf{i}'$ is matched to $\mathbf{j}' > \mathbf{j}$. In the above example, this is $\mathbf{M} = \{(1,1),(2,2),(3,3),(4,5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

Edit Distance: Alternate View

Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

$$\begin{array}{ccccc} \mathbf{F} & \mathbf{O} & \mathbf{O} & & \mathbf{D} \\ \mathbf{M} & \mathbf{O} & \mathbf{N} & \mathbf{E} & \mathbf{Y} \end{array}$$

Formally, an alignment is a set $\mathbf{M}$ of pairs $(\mathbf{i}, \mathbf{j})$ such that each index appears at most once, and there is no "crossing": $\mathbf{i} < \mathbf{i}'$ and $\mathbf{i}$ is matched to $\mathbf{j}$ implies $\mathbf{i}'$ is matched to $\mathbf{j}' > \mathbf{j}$. In the above example, this is $\mathbf{M} = \{(1,1),(2,2),(3,3),(4,5)\}$. Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

Applications

1. Spell-checkers and Dictionaries
2. Unix `diff`
3. DNA sequence alignment ... but, we need a new metric

Similarity Metric

Definition

For two strings $\mathbf{X}$ and $\mathbf{Y}$, the cost of alignment $\mathbf{M}$ is

1. [Gap penalty] For each gap in the alignment, we incur a cost δ.
2. [Mismatch cost] For each pair $\mathbf{p}$ and $\mathbf{q}$ that have been matched in $\mathbf{M}$, we incur cost $\alpha_{\mathbf{pq}}$; typically $\alpha_{\mathbf{pp}} = 0$.

Edit distance is special case when $\delta = \alpha_{\mathbf{pq}} = 1$.

Similarity Metric

Definition

For two strings $\mathbf{X}$ and $\mathbf{Y}$, the cost of alignment $\mathbf{M}$ is

1. [Gap penalty] For each gap in the alignment, we incur a cost δ.
2. [Mismatch cost] For each pair $\mathbf{p}$ and $\mathbf{q}$ that have been matched in $\mathbf{M}$, we incur cost $\alpha_{\mathbf{pq}}$; typically $\alpha_{\mathbf{pp}} = 0$.

Edit distance is special case when $\delta = \alpha_{\mathbf{pq}} = 1$.

An Example

Example

o		c	u	r	r	a	n	c	e
o	c	c	u	r	r	e	n	c	e

Cost $= \delta + \alpha_{\mathbf{ae}}$

Alternative:

o		c	u	r	r		a	n	c	e
o	c	c	u	r	r	e		n	c	e

Cost $= 3\delta$

Or a really stupid solution (delete string, insert other string):

o	c	u	r	r	a	n	c	e										
									o	c	c	u	r	r	e	n	c	e

Cost $= 19\delta$.

What is the edit distance between...

What is the minimum edit distance for the following two strings, if insertion/deletion/change of a single character cost **1** unit?

HAR–PELED

SHARP␣EYED

(A) 1
(B) 2
(C) 3
(D) 4
(E) 5

Sequence Alignment

Input Given two words $\mathbf{X}$ and $\mathbf{Y}$, and gap penalty δ and mismatch costs $\alpha_{\mathbf{pq}}$

Goal Find alignment of minimum cost

Edit distance

Basic observation

Let $\mathbf{X} = \alpha\mathbf{x}$ and $\mathbf{Y} = \beta\mathbf{y}$

α, β: strings.

$\mathbf{x}$ and $\mathbf{y}$ single characters.

Think about optimal edit distance between $\mathbf{X}$ and $\mathbf{Y}$ as alignment, and consider last column of alignment of the two strings:

α	$\mathbf{x}$
β	$\mathbf{y}$

or

α	$\mathbf{x}$
$\beta\mathbf{y}$	

or

$\alpha\mathbf{x}$	
β	$\mathbf{y}$

Observation

Prefixes must have optimal alignment!

Problem Structure

Observation

Let $\mathbf{X} = \mathbf{x_1 x_2 \cdots x_m}$ *and* $\mathbf{Y} = \mathbf{y_1 y_2 \cdots y_n}$. *If* $\mathbf{(m, n)}$ *are not matched then either the* $\mathbf{m}$*th position of* $\mathbf{X}$ *remains unmatched or the* $\mathbf{n}$*th position of* $\mathbf{Y}$ *remains unmatched.*

1. Case $\mathbf{x_m}$ and $\mathbf{y_n}$ are matched.
   1. Pay mismatch cost $\alpha_{\mathbf{x_m y_n}}$ plus cost of aligning strings $\mathbf{x_1 \cdots x_{m-1}}$ and $\mathbf{y_1 \cdots y_{n-1}}$
2. Case $\mathbf{x_m}$ is unmatched.
   1. Pay gap penalty plus cost of aligning $\mathbf{x_1 \cdots x_{m-1}}$ and $\mathbf{y_1 \cdots y_n}$
3. Case $\mathbf{y_n}$ is unmatched.
   1. Pay gap penalty plus cost of aligning $\mathbf{x_1 \cdots x_m}$ and $\mathbf{y_1 \cdots y_{n-1}}$

Subproblems and Recurrence

Optimal Costs

Let $\mathrm{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\mathrm{Opt}(i,j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{Opt}(i-1, j-1), \\ \delta + \mathrm{Opt}(i-1, j), \\ \delta + \mathrm{Opt}(i, j-1) \end{cases}$$

Base Cases: $\mathrm{Opt}(i, 0) = \delta \cdot i$ and $\mathrm{Opt}(0, j) = \delta \cdot j$

Subproblems and Recurrence

Optimal Costs

Let $\mathrm{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\mathrm{Opt}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{Opt}(i-1, j-1), \\ \delta + \mathrm{Opt}(i-1, j), \\ \delta + \mathrm{Opt}(i, j-1) \end{cases}$$

Base Cases: $\mathrm{Opt}(i, 0) = \delta \cdot i$ and $\mathrm{Opt}(0, j) = \delta \cdot j$

Dynamic Programming Solution

for `all` i **do** $M[i,0] = i\delta$
for `all` j **do** $M[0,j] = j\delta$

for $i = 1$ `to` m **do**
  for $j = 1$ `to` n **do**

$$M[i,j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1, j-1], \\ \delta + M[i-1, j], \\ \delta + M[i, j-1] \end{cases}$$

Analysis

1. Running time is $O(mn)$.

Dynamic Programming Solution

for `all` i **do** $M[i,0] = i\delta$
for `all` j **do** $M[0,j] = j\delta$

for $i = 1$ `to` m **do**
  for $j = 1$ `to` n **do**

$$M[i,j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1, j-1], \\ \delta + M[i-1, j], \\ \delta + M[i, j-1] \end{cases}$$

Analysis

1. Running time is $O(mn)$.

Dynamic Programming Solution

for `all` i **do** $M[i,0] = i\delta$
for `all` j **do** $M[0,j] = j\delta$

for $i = 1$ `to` m **do**
  for $j = 1$ `to` n **do**

$$M[i,j] = \min \begin{cases} \alpha_{x_i y_j} + M[i-1, j-1], \\ \delta + M[i-1, j], \\ \delta + M[i, j-1] \end{cases}$$

Analysis

1. Running time is $O(mn)$.
2. Space used is $O(mn)$.

Matrix and DAG of Computation

Figure : Iterative algorithm in previous slide computes values in row order. Optimal value is a shortest path from $(0, 0)$ to (m, n) in .

Sequence Alignment in Practice

1. Typically the DNA sequences that are aligned are about 10^5 letters long!
2. So about 10^{10} operations and 10^{10} bytes needed
3. The killer is the 10GB storage
4. Can we reduce space requirements?

Optimizing Space

1. Recall

$$M(i,j) = \min \begin{cases} \alpha_{x_i y_j} + M(i-1, j-1), \\ \delta + M(i-1, j), \\ \delta + M(i, j-1) \end{cases}$$

2. Entries in jth column only depend on $(j-1)$st column and earlier entries in jth column
3. Only store the current column and the previous column reusing space; $N(i,0)$ stores $M(i,j-1)$ and $N(i,1)$ stores $M(i,j)$

Computing in column order to save space

Figure : **M(i, j)** only depends on previous column values. Keep only two columns and compute in column order.

Space Efficient Algorithm

```
for all i do N[i,0] = iδ
for j = 1 to n do
    N[0,1] = jδ (* corresponds to M(0,j) *)
    for i = 1 to m do
        N[i,1] = min { α_{x_i y_j} + N[i-1,0]
                     { δ + N[i-1,1]
                     { δ + N[i,0]
    for i = 1 to m do
        Copy N[i,0] = N[i,1]
```

Analysis

Running time is $O(mn)$ and space used is $O(2m) = O(m)$

Analyzing Space Efficiency

1. From the $\mathbf{m} \times \mathbf{n}$ matrix $\mathbf{M}$ we can construct the actual alignment (exercise)
2. Matrix $\mathbf{N}$ computes cost of optimal alignment but no way to construct the actual alignment
3. Space efficient computation of alignment? More complicated algorithm — see text book.

Part III

DAGs and Dynamic Programming

Recursion and DAGs

Observation

Let **A** *be a recursive algorithm for problem* **Π**. *For each instance* **I** *of* **Π** *there is an associated* DAG **G(I)**.

1. Create directed graph **G(I)** as follows...
2. For each sub-problem in the execution of **A** on **I** create a node.
3. If sub-problem **v** *depends* on or *recursively calls* sub-problem **u** add *directed* edge **(u, v)** to graph.
4. **G(I)** is a DAG. Why? If **G(I)** has a cycle then **A** will not terminate on **I**.

Recursion and DAGs

Observation

Let **A** *be a recursive algorithm for problem* **Π**. *For each instance* **I** *of* **Π** *there is an associated* DAG **G(I)**.

1. Create directed graph **G(I)** as follows...
2. For each sub-problem in the execution of **A** on **I** create a node.
3. If sub-problem **v** *depends* on or *recursively calls* sub-problem **u** add *directed* edge **(u, v)** to graph.
4. **G(I)** is a DAG. Why? If **G(I)** has a cycle then **A** will not terminate on **I**.

Iterative Algorithm for...

Dynamic Programming and DAGs

Observation

An iterative algorithm **B** *obtained from a recursive algorithm* **A** *for a problem* **Π** *does the following:*

> *For each instance* **I** *of* **Π**, *it computes a topological sort of* **G(I)** *and evaluates sub-problems according to the topological ordering.*

1. Sometimes the DAG **G(I)** can be obtained directly without thinking about the recursive algorithm **A**
2. In some cases (not all) the computation of an optimal solution reduces to a shortest/longest path in DAG **G(I)**
3. Topological sort based shortest/longest path computation is dynamic programming!

A quick reminder...

A Recursive Algorithm for weighted interval scheduling

Let O_i be value of an optimal schedule for the first i jobs.

```
Schedule(n):
        if n = 0 then return 0
        if n = 1 then return w(v_1)
        O_p(n) ←Schedule(p(n))
        O_n−1 ←Schedule(n − 1)
        if (O_p(n) + w(v_n) < O_n−1) then
            O_n = O_n−1
        else
            O_n = O_p(n) + w(v_n)
        return O_n
```

Weighted Interval Scheduling via...

Longest Path in a DAG

Given intervals, create a DAG as follows:

1. Create one node for each interval, plus a dummy sink node $\mathbf{0}$ for interval $\mathbf{0}$, plus a dummy source node $\mathbf{s}$.
2. For each interval $\mathbf{i}$ add edge $\mathbf{(i, p(i))}$ of the length/weight of $\mathbf{v_i}$.
3. Add an edge from $\mathbf{s}$ to $\mathbf{n}$ of length $\mathbf{0}$.
4. For each interval $\mathbf{i}$ add edge $\mathbf{(i, i-1)}$ of length $\mathbf{0}$.

Example

$p(5) = 2$, $p(4) = 1$, $p(3) = 1$, $p(2) = 0$, $p(1) = 0$

Relating Optimum Solution

Given interval problem instance $\mathbf{I}$ let $\mathbf{G(I)}$ denote the DAG constructed as described.

Claim

Optimum solution to weighted interval scheduling instance $\mathbf{I}$ *is given by* longest *path from* $\mathbf{s}$ *to* $\mathbf{0}$ *in* $\mathbf{G(I)}$.

Assuming claim is true,

1. If $\mathbf{I}$ has $\mathbf{n}$ intervals, DAG $\mathbf{G(I)}$ has $\mathbf{n+2}$ nodes and $\mathbf{O(n)}$ edges. Creating $\mathbf{G(I)}$ takes $\mathbf{O(n \log n)}$ time: to find $\mathbf{p(i)}$ for each $\mathbf{i}$. How?
2. Longest path can be computed in $\mathbf{O(n)}$ time — recall $\mathbf{O(m+n)}$ algorithm for shortest/longest paths in DAGs.

Relating Optimum Solution

Given interval problem instance $\mathbf{I}$ let $\mathbf{G(I)}$ denote the DAG constructed as described.

Claim

Optimum solution to weighted interval scheduling instance $\mathbf{I}$ *is given by* longest *path from* $\mathbf{s}$ *to* $\mathbf{0}$ *in* $\mathbf{G(I)}$.

Assuming claim is true,

1. If $\mathbf{I}$ has $\mathbf{n}$ intervals, DAG $\mathbf{G(I)}$ has $\mathbf{n+2}$ nodes and $\mathbf{O(n)}$ edges. Creating $\mathbf{G(I)}$ takes $\mathbf{O(n \log n)}$ time: to find $\mathbf{p(i)}$ for each $\mathbf{i}$. How?
2. Longest path can be computed in $\mathbf{O(n)}$ time — recall $\mathbf{O(m+n)}$ algorithm for shortest/longest paths in DAGs.

DAG for Longest Increasing Sequence

Given sequence $a_1, a_2, \ldots, a_n$ create DAG as follows:

1. add sentinel a_0 to sequence where a_0 is less than smallest element in sequence
2. for each i there is a node v_i
3. if $i < j$ and $a_i < a_j$ add an edge (v_i, v_j)
4. find longest path from v_0

DAG for Longest Increasing Sequence

Given sequence $a_1, a_2, \ldots, a_n$ create DAG as follows:

1. add sentinel a_0 to sequence where a_0 is less than smallest element in sequence
2. for each i there is a node v_i
3. if $i < j$ and $a_i < a_j$ add an edge (v_i, v_j)
4. find longest path from v_0

Takeaway Points

1. Dynamic programming is based on finding a recursive way to solve the problem. Need a recursion that generates a small number of subproblems.
2. Given a recursive algorithm there is a natural DAG associated with the subproblems that are generated for given instance; this is the dependency graph. An iterative algorithm simply evaluates the subproblems in some topological sort of this DAG.
3. The space required to evaluate the answer can be reduced in some cases by a careful examination of that dependency DAG of the subproblems and keeping only a subset of the DAG at any time.

Notes

Notes

Notes

Notes

More Dynamic Programming

Lecture 10

February 20, 2014

Part I

All Pairs Shortest Paths

Shortest Path Problems

Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with edge lengths (or costs). For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

1. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.
3. Find shortest paths for all pairs of nodes.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

1. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

Dijkstra's algorithm for non-negative edge lengths. Running time: $\mathbf{O((m + n) \log n)}$ with heaps and $\mathbf{O(m + n \log n)}$ with advanced priority queues.

Bellman-Ford algorithm for arbitrary edge lengths. Running time: $\mathbf{O(nm)}$.

Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

1. Given nodes $\mathbf{s, t}$ find shortest path from $\mathbf{s}$ to $\mathbf{t}$.
2. Given node $\mathbf{s}$ find shortest path from $\mathbf{s}$ to all other nodes.

Dijkstra's algorithm for non-negative edge lengths. Running time: $\mathbf{O((m + n) \log n)}$ with heaps and $\mathbf{O(m + n \log n)}$ with advanced priority queues.

Bellman-Ford algorithm for arbitrary edge lengths. Running time: $\mathbf{O(nm)}$.

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms $\mathbf{n}$ times, once for each vertex.

1. Non-negative lengths. $\mathbf{O(nm \log n)}$ with heaps and $\mathbf{O(nm + n^2 \log n)}$ using advanced priority queues.
2. Arbitrary edge lengths: $\mathbf{O(n^2 m)}$. $\Theta(\mathbf{n}^4)$ if $\mathbf{m} = \Omega(\mathbf{n}^2)$.

Can we do better?

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms n times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.
2. Arbitrary edge lengths: $O(n^2 m)$. $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?

All-Pairs Shortest Paths

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $\mathbf{G = (V, E)}$ with edge lengths. For edge $\mathbf{e = (u, v)}$, $\ell(\mathbf{e}) = \ell(\mathbf{u, v})$ is its length.

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms $\mathbf{n}$ times, once for each vertex.

1. Non-negative lengths. $\mathbf{O(nm \log n)}$ with heaps and $\mathbf{O(nm + n^2 \log n)}$ using advanced priority queues.
2. Arbitrary edge lengths: $\mathbf{O(n^2 m)}$. $\mathbf{\Theta(n^4)}$ if $\mathbf{m = \Omega(n^2)}$.

Can we do better?

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

1. $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ *is a shortest path from* s *to* v_i

Sub-problem idea: paths of fewer hops/edges

Shortest Paths and Recursion

1. Compute the shortest path distance from s to t recursively?
2. What are the smaller sub-problems?

Lemma

Let G be a directed graph with arbitrary edge lengths. If $s = v_0 \to v_1 \to v_2 \to \ldots \to v_k$ is a shortest path from s to v_k then for $1 \leq i < k$:

1. $s = v_0 \to v_1 \to v_2 \to \ldots \to v_i$ *is a shortest path from* s *to* v_i

Sub-problem idea: paths of fewer hops/edges

Shortest Paths and Recursion

1. Compute the shortest path distance from $\mathbf{s}$ to $\mathbf{t}$ recursively?
2. What are the smaller sub-problems?

Lemma

Let $\mathbf{G}$ *be a directed graph with arbitrary edge lengths. If* $\mathbf{s} = \mathbf{v_0} \rightarrow \mathbf{v_1} \rightarrow \mathbf{v_2} \rightarrow \ldots \rightarrow \mathbf{v_k}$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v_k}$ *then for* $\mathbf{1} \leq \mathbf{i} < \mathbf{k}$*:*

1. $\mathbf{s} = \mathbf{v_0} \rightarrow \mathbf{v_1} \rightarrow \mathbf{v_2} \rightarrow \ldots \rightarrow \mathbf{v_i}$ *is a shortest path from* $\mathbf{s}$ *to* $\mathbf{v_i}$

Sub-problem idea: paths of fewer hops/edges

Hop-based Recur': Single-Source Shortest Paths

Single-source problem: fix source $\mathbf{s}$.

$\mathbf{OPT(v, k)}$: shortest walk length from $\mathbf{s}$ to $\mathbf{v}$ using at most $\mathbf{k}$ edges.

Note: $\mathbf{dist(s, v) = OPT(v, n-1)}$. Recursion for $\mathbf{OPT(v, k)}$:

$$\mathbf{OPT(v, k)} = \mathbf{min} \begin{cases} \mathbf{min_{u \in V}(OPT(u, k-1) + c(u, v)).} \\ \mathbf{OPT(v, k-1)} \end{cases}$$

Base case: $\mathbf{OPT(v, 1) = c(s, v)}$ if $\mathbf{(s, v) \in E}$ otherwise ∞
Leads to Bellman-Ford algorithm

$\mathbf{OPT(v, k)}$ values are also of independent interest: shortest paths with at most $\mathbf{k}$ hops

Hop-based Recur': Single-Source Shortest Paths

Single-source problem: fix source $\mathbf{s}$.

$\mathbf{OPT(v, k)}$: shortest walk length from $\mathbf{s}$ to $\mathbf{v}$ using at most $\mathbf{k}$ edges.
Note: $\mathbf{dist(s, v) = OPT(v, n-1)}$. Recursion for $\mathbf{OPT(v, k)}$:

$$\mathbf{OPT(v, k)} = \min \begin{cases} \min_{u \in V}(\mathbf{OPT(u, k-1) + c(u, v)}). \\ \mathbf{OPT(v, k-1)} \end{cases}$$

Base case: $\mathbf{OPT(v, 1) = c(s, v)}$ if $\mathbf{(s, v) \in E}$ otherwise ∞
Leads to Bellman-Ford algorithm

$\mathbf{OPT(v, k)}$ values are also of independent interest: shortest paths with at most $\mathbf{k}$ hops

Hop-based Recur': Single-Source Shortest Paths

Single-source problem: fix source $\mathbf{s}$.

$\mathbf{OPT(v,k)}$: shortest walk length from $\mathbf{s}$ to $\mathbf{v}$ using at most $\mathbf{k}$ edges.
Note: $\mathbf{dist(s,v) = OPT(v,n-1)}$. Recursion for $\mathbf{OPT(v,k)}$:

$$\mathbf{OPT(v,k)} = \min \begin{cases} \min_{u \in V}(\mathbf{OPT(u,k-1)} + \mathbf{c(u,v)}). \\ \mathbf{OPT(v,k-1)} \end{cases}$$

Base case: $\mathbf{OPT(v,1) = c(s,v)}$ if $\mathbf{(s,v) \in E}$ otherwise ∞
Leads to Bellman-Ford algorithm

$\mathbf{OPT(v,k)}$ values are also of independent interest: shortest paths with at most $\mathbf{k}$ hops

Hop-based Recur': Single-Source Shortest Paths

Single-source problem: fix source $\mathbf{s}$.

$\mathbf{OPT(v,k)}$: shortest walk length from $\mathbf{s}$ to $\mathbf{v}$ using at most $\mathbf{k}$ edges.
Note: $\mathbf{dist(s,v) = OPT(v,n-1)}$. Recursion for $\mathbf{OPT(v,k)}$:

$$\mathbf{OPT(v,k) = min} \begin{cases} \mathbf{min_{u \in V}(OPT(u,k-1) + c(u,v)).} \\ \mathbf{OPT(v,k-1)} \end{cases}$$

Base case: $\mathbf{OPT(v,1) = c(s,v)}$ if $\mathbf{(s,v) \in E}$ otherwise ∞
Leads to Bellman-Ford algorithm

$\mathbf{OPT(v,k)}$ values are also of independent interest: shortest paths with at most $\mathbf{k}$ hops

Hop-based Recur': Single-Source Shortest Paths

Single-source problem: fix source $\mathbf{s}$.

$\mathbf{OPT(v, k)}$: shortest walk length from $\mathbf{s}$ to $\mathbf{v}$ using at most $\mathbf{k}$ edges.
Note: $\mathbf{dist(s, v) = OPT(v, n-1)}$. Recursion for $\mathbf{OPT(v, k)}$:

$$\mathbf{OPT(v,k) = min} \begin{cases} \mathbf{min_{u \in V}(OPT(u, k-1) + c(u, v)).} \\ \mathbf{OPT(v, k-1)} \end{cases}$$

Base case: $\mathbf{OPT(v, 1) = c(s, v)}$ if $\mathbf{(s, v) \in E}$ otherwise ∞
Leads to Bellman-Ford algorithm

$\mathbf{OPT(v, k)}$ values are also of independent interest: shortest paths with at most $\mathbf{k}$ hops

Hop-based Recur': Single-Source Shortest Paths

Single-source problem: fix source $\mathbf{s}$.

$\mathbf{OPT(v,k)}$: shortest walk length from $\mathbf{s}$ to $\mathbf{v}$ using at most $\mathbf{k}$ edges.
Note: $\mathbf{dist(s,v) = OPT(v,n-1)}$. Recursion for $\mathbf{OPT(v,k)}$:

$$\mathbf{OPT(v,k) = min} \begin{cases} \mathbf{min_{u \in V}(OPT(u,k-1) + c(u,v)).} \\ \mathbf{OPT(v,k-1)} \end{cases}$$

Base case: $\mathbf{OPT(v,1) = c(s,v)}$ if $\mathbf{(s,v) \in E}$ otherwise ∞
Leads to Bellman-Ford algorithm

$\mathbf{OPT(v,k)}$ values are also of independent interest: shortest paths with at most $\mathbf{k}$ hops

All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $\mathbf{v_1, v_2, \ldots, v_n}$
2. **dist(i, j, k)**: shortest path distance between $\mathbf{v_i}$ and $\mathbf{v_j}$ among all paths in which the largest index of an *intermediate node* is at most **k**

$$\mathbf{dist(i,j,0)} = 100$$
$$\mathbf{dist(i,j,1)} = 9$$
$$\mathbf{dist(i,j,2)} = 8$$
$$\mathbf{dist(i,j,3)} = 5$$

All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $\mathbf{v_1, v_2, \ldots, v_n}$
2. **dist(i, j, k)**: shortest path distance between $\mathbf{v_i}$ and $\mathbf{v_j}$ among all paths in which the largest index of an *intermediate node* is at most **k**

$$\begin{aligned}
\mathbf{dist(i,j,0)} &= \mathbf{100} \\
\mathbf{dist(i,j,1)} &= 9 \\
\mathbf{dist(i,j,2)} &= 8 \\
\mathbf{dist(i,j,3)} &= 5
\end{aligned}$$

All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $\mathbf{v_1, v_2, \ldots, v_n}$
2. **dist(i, j, k)**: shortest path distance between $\mathbf{v_i}$ and $\mathbf{v_j}$ among all paths in which the largest index of an *intermediate node* is at most **k**

$$\begin{aligned}
\mathbf{dist(i,j,0)} &= \mathbf{100} \\
\mathbf{dist(i,j,1)} &= \mathbf{9} \\
\mathbf{dist(i,j,2)} &= 8 \\
\mathbf{dist(i,j,3)} &= 5
\end{aligned}$$

All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $\mathbf{v_1, v_2, \ldots, v_n}$
2. **dist(i, j, k)**: shortest path distance between $\mathbf{v_i}$ and $\mathbf{v_j}$ among all paths in which the largest index of an *intermediate node* is at most **k**

$$\begin{aligned}
\mathbf{dist(i,j,0)} &= \mathbf{100} \\
\mathbf{dist(i,j,1)} &= \mathbf{9} \\
\mathbf{dist(i,j,2)} &= \mathbf{8} \\
\mathbf{dist(i,j,3)} &= \mathbf{5}
\end{aligned}$$

All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $\mathbf{v_1, v_2, \ldots, v_n}$
2. $\mathbf{dist(i, j, k)}$: shortest path distance between $\mathbf{v_i}$ and $\mathbf{v_j}$ among all paths in which the largest index of an *intermediate node* is at most $\mathbf{k}$

$$\mathbf{dist(i, j, 0)} = \mathbf{100}$$
$$\mathbf{dist(i, j, 1)} = \mathbf{9}$$
$$\mathbf{dist(i, j, 2)} = \mathbf{8}$$
$$\mathbf{dist(i, j, 3)} = \mathbf{5}$$

For the following graph, **dist(i, j, 2)** is...

(A) 9

(B) 10

(C) 11

(D) 12

(E) 15

All-Pairs: Recursion on index of intermediate nodes

$$\mathbf{dist(i,j,k) = min} \begin{cases} \mathbf{dist(i,j,k-1)} \\ \mathbf{dist(i,k,k-1) + dist(k,j,k-1)} \end{cases}$$

Base case: $\mathbf{dist(i,j,0) = c(i,j)}$ if $\mathbf{(i,j) \in E}$, otherwise ∞

Correctness: If $\mathbf{i \to j}$ shortest path goes through $\mathbf{k}$ then $\mathbf{k}$ occurs only once on the path — otherwise there is a negative length cycle.

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
Check if G has a negative cycle // Bellman-Ford:  O(mn) time
if there is a negative cycle then return ``Negative cycle''

for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = c(i,j)  (* c(i,j) = ∞ if (i,j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i,j,k) = min { dist(i,j,k-1),
                                dist(i,k,k-1) + dist(k,j,k-1) }
```

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
Check if G has a negative cycle // Bellman-Ford:  O(mn) time
if there is a negative cycle then return ``Negative cycle''

for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = c(i,j)  (* c(i,j) = ∞ if (i,j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i,j,k) = min { dist(i,j,k-1),
                                dist(i,k,k-1) + dist(k,j,k-1) }
```

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

```
Check if G has a negative cycle // Bellman-Ford:  O(mn) time
if there is a negative cycle then return ``Negative cycle''

for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = c(i,j)  (* c(i,j) = ∞ if (i,j) ∉ E, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i,j,k) = min { dist(i,j,k-1),
                                dist(i,k,k-1) + dist(k,j,k-1) }
```

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

Do we need a separate algorithm to check if there is negative cycle?

```
for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = c(i,j)   (* c(i,j) = ∞ if (i,j) ∉ E, 0 if i = j *)
not edge, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i,j,k) = min(dist(i,j,k − 1), dist(i,k,k − 1) + dist(k,j,k −

for i = 1 to n do
    if (dist(i,i,n) < 0) then
        Output that there is a negative length cycle in G
```

Correctness: exercise

Floyd-Warshall Algorithm

for All-Pairs Shortest Paths

Do we need a separate algorithm to check if there is negative cycle?

```
for i = 1 to n do
    for j = 1 to n do
        dist(i,j,0) = c(i,j)   (* c(i,j) = ∞ if (i,j) ∉ E, 0 if i = j *)
not edge, 0 if i = j *)

for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            dist(i,j,k) = min(dist(i,j,k − 1), dist(i,k,k − 1) + dist(k,j,k −

for i = 1 to n do
    if (dist(i,i,n) < 0) then
        Output that there is a negative length cycle in G
```

Correctness: exercise

Floyd-Warshall Algorithm: Finding the Paths

Question: Can we find the paths in addition to the distances?

1. Create a $\mathbf{n} \times \mathbf{n}$ array Next that stores the next vertex on shortest path for each pair of vertices
2. With array Next, for any pair of given vertices $\mathbf{i}, \mathbf{j}$ can compute a shortest path in $\mathbf{O(n)}$ time.

Floyd-Warshall Algorithm: Finding the Paths

Question: Can we find the paths in addition to the distances?

1. Create a $\mathbf{n} \times \mathbf{n}$ array Next that stores the next vertex on shortest path for each pair of vertices
2. With array Next, for any pair of given vertices $\mathbf{i, j}$ can compute a shortest path in $\mathbf{O(n)}$ time.

Floyd-Warshall Algorithm

Finding the Paths

```
for i = 1 to n do
    for j = 1 to n do
        dist(i, j, 0) = c(i, j) (* c(i, j) = ∞ if (i, j) not edge, 0 if i = j *
        Next(i, j) = −1
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            if (dist(i, j, k − 1) > dist(i, k, k − 1) + dist(k, j, k − 1)) then
                dist(i, j, k) = dist(i, k, k − 1) + dist(k, j, k − 1)
                Next(i, j) = k

for i = 1 to n do
    if (dist(i, i, n) < 0) then
        Output that there is a negative length cycle in G
```

Exercise: Given **Next** array and any two vertices **i, j** describe an **O(n)** algorithm to find a **i-j** shortest path.

Summary of results on shortest paths

Single source		
No negative edges	Dijkstra	$O(n \log n + m)$
Edge lengths can be negative	Bellman Ford	$O(nm)$

All Pairs Shortest Paths

No negative edges	n * Dijkstra	$O(n^2 \log n + nm)$
No negative cycles	n * Bellman Ford	$O(n^2 m) = O(n^4)$
No negative cycles	BF + n * Dijkstra	$O(nm + n^2 \log n)$
No negative cycles	Floyd-Warshall	$O(n^3)$
Unweighted	Matrix multiplication	$O(n^{2.38})$, $O(n^{2.58})$

Part II

Knapsack

Knapsack Problem

Input Given a Knapsack of capacity $\mathbf{W}$ lbs. and $\mathbf{n}$ objects with $\mathbf{i}$th object having weight $\mathbf{w_i}$ and value $\mathbf{v_i}$; assume $\mathbf{W}, \mathbf{w_i}, \mathbf{v_i}$ are all positive integers

Goal Fill the Knapsack without exceeding weight limit while maximizing value.

Basic problem that arises in many applications as a sub-problem.

Knapsack Problem

Input Given a Knapsack of capacity $\mathbf{W}$ lbs. and $\mathbf{n}$ objects with $\mathbf{i}$th object having weight $\mathbf{w_i}$ and value $\mathbf{v_i}$; assume $\mathbf{W}, \mathbf{w_i}, \mathbf{v_i}$ are all positive integers

Goal Fill the Knapsack without exceeding weight limit while maximizing value.

Basic problem that arises in many applications as a sub-problem.

Knapsack Example

Example

Item	I_1	I_2	I_3	I_4	I_5
Value	1	6	18	22	28
Weight	1	2	5	6	7

If $W = 11$, the best is $\{I_3, I_4\}$ giving value 40.

Special Case

When $v_i = w_i$, the Knapsack problem is called the Subset Sum Problem.

Knapsack

For the following instance of Knapsack:

Item	I_1	I_2	I_3	I_4	I_5
Value	1	6	16	22	28
Weight	1	2	5	6	7

and weight limit $W = 15$. The best solution has value:

(A) 22
(B) 28
(C) 38
(D) 50
(E) 56

Greedy Approach

1. Pick objects with greatest value
   1. Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$
2. Pick objects with smallest weight
   1. Let $W = 2$, $w_1 = 1$, $w_2 = 2$, $v_1 = 1$ and $v_2 = 3$; greedy strategy will pick $\{1\}$, but the optimal is $\{2\}$
3. Pick objects with largest v_i/w_i ratio
   1. Let $W = 4$, $w_1 = w_2 = 2$, $w_3 = 3$, $v_1 = v_2 = 3$ and $v_3 = 5$; greedy strategy will pick $\{3\}$, but the optimal is $\{1, 2\}$
   2. Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to W.

Towards a Recursive Solution

First guess: $\mathbf{Opt(i)}$ is the optimum solution value for items $\mathbf{1, \ldots, i}$.

Observation

Consider an optimal solution $\mathcal{O}$ *for* $\mathbf{1, \ldots, i}$

Case item $\mathbf{i} \notin \mathcal{O}$ $\mathcal{O}$ *is an optimal solution to items* $\mathbf{1}$ *to* $\mathbf{i - 1}$

Case item $\mathbf{i} \in \mathcal{O}$ *Then* $\mathcal{O} - \{\mathbf{i}\}$ *is an optimum solution for items* $\mathbf{1}$ *to* $\mathbf{n - 1}$ *in knapsack of capacity* $\mathbf{W - w_i}$.

Subproblems depend also on remaining *capacity. Cannot write subproblem only in terms of* $\mathbf{Opt(1), \ldots, Opt(i - 1)}$.

$\mathbf{Opt(i, w)}$: optimum profit for items $\mathbf{1}$ to $\mathbf{i}$ in knapsack of size $\mathbf{w}$

Goal: compute $\mathbf{Opt(n, W)}$

Towards a Recursive Solution

First guess: $\mathbf{Opt(i)}$ is the optimum solution value for items $\mathbf{1, \ldots, i}$.

Observation

Consider an optimal solution $\mathcal{O}$ *for* $\mathbf{1, \ldots, i}$

Case item $\mathbf{i} \notin \mathcal{O}$ $\mathcal{O}$ *is an optimal solution to items* $\mathbf{1}$ *to* $\mathbf{i - 1}$

Case item $\mathbf{i} \in \mathcal{O}$ *Then* $\mathcal{O} - \{\mathbf{i}\}$ *is an optimum solution for items* $\mathbf{1}$ *to* $\mathbf{n - 1}$ *in knapsack of capacity* $\mathbf{W - w_i}$.
Subproblems depend also on remaining *capacity. Cannot write subproblem only in terms of* $\mathbf{Opt(1), \ldots, Opt(i - 1)}$.

$\mathbf{Opt(i, w)}$: optimum profit for items $\mathbf{1}$ to $\mathbf{i}$ in knapsack of size $\mathbf{w}$
Goal: compute $\mathbf{Opt(n, W)}$

Towards a Recursive Solution

First guess: $\mathbf{Opt(i)}$ is the optimum solution value for items $\mathbf{1, \ldots, i}$.

Observation

Consider an optimal solution $\mathcal{O}$ *for* $\mathbf{1, \ldots, i}$

Case item $\mathbf{i} \notin \mathcal{O}$ $\mathcal{O}$ *is an optimal solution to items* $\mathbf{1}$ *to* $\mathbf{i - 1}$

Case item $\mathbf{i} \in \mathcal{O}$ *Then* $\mathcal{O} - \{\mathbf{i}\}$ *is an optimum solution for items* $\mathbf{1}$ *to* $\mathbf{n - 1}$ *in knapsack of capacity* $\mathbf{W - w_i}$.
Subproblems depend also on remaining *capacity. Cannot write subproblem only in terms of* $\mathbf{Opt(1), \ldots, Opt(i - 1)}$.

$\mathbf{Opt(i, w)}$: optimum profit for items $\mathbf{1}$ to $\mathbf{i}$ in knapsack of size $\mathbf{w}$
Goal: compute $\mathbf{Opt(n, W)}$

Dynamic Programming Solution

Definition

Let $\mathrm{Opt}(i, w)$ be the optimal way of picking items from 1 to i, with total weight not exceeding w.

$$\mathrm{Opt}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \mathrm{Opt}(i-1, w) & \text{if } w_i > w \\ \max \begin{cases} \mathrm{Opt}(i-1, w) \\ \mathrm{Opt}(i-1, w - w_i) + v_i \end{cases} & \text{otherwise} \end{cases}$$

An Iterative Algorithm

```
for w = 0 to W do
    M[0, w] = 0
for i = 1 to n do
    for w = 1 to W do
        if (w_i > w) then
            M[i, w] = M[i − 1, w]
        else
            M[i, w] = max(M[i − 1, w], M[i − 1, w − w_i] + v_i)
```

Running Time

1. Time taken is $O(nW)$
2. Input has size $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$; so running time not polynomial but "pseudo-polynomial"!

An Iterative Algorithm

```
for w = 0 to W do
    M[0, w] = 0
for i = 1 to n do
    for w = 1 to W do
        if (w_i > w) then
            M[i, w] = M[i − 1, w]
        else
            M[i, w] = max(M[i − 1, w], M[i − 1, w − w_i] + v_i)
```

Running Time

1. Time taken is $O(nW)$
2. Input has size $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$; so running time not polynomial but "pseudo-polynomial"!

An Iterative Algorithm

```
for w = 0 to W do
    M[0, w] = 0
for i = 1 to n do
    for w = 1 to W do
        if (w_i > w) then
            M[i, w] = M[i − 1, w]
        else
            M[i, w] = max(M[i − 1, w], M[i − 1, w − w_i] + v_i)
```

Running Time

1. Time taken is $O(nW)$
2. Input has size $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$; so running time not polynomial but "pseudo-polynomial"!

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the **combinatorial size** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n} (\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the **combinatorial size** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the **combinatorial size** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the **combinatorial size** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the **combinatorial size** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the **combinatorial size** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

Knapsack Algorithm and Polynomial time

1. Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
2. Running time of dynamic programming algorithm: $O(nW)$.
3. Not a polynomial time algorithm.
4. Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
5. Algorithm is called a **pseudo-polynomial** time algorithm because running time is polynomial if *numbers* in input are of size polynomial in the **combinatorial size** of problem.
6. Knapsack is **NP-Hard** if numbers are not polynomial in n.

$n = 2$

$w_1 = 10^9 - 1 \quad v_1 = 2 \qquad W = 10^9$

$w_2 = 10^9 - 3, \quad v_2 = 3$

How much is n!?

(A) $n! = \Theta(n^n)$

(B) $n! = 2^{\Theta(n)}$

(C) $n! = \Theta(2^n)$

(D) $n! = 2^{\Theta(n \log n)}$

(E) $n! = \Theta\left(2^{n \log n}\right)$

Part III

Traveling Salesman Problem

Traveling Salesman Problem

Input A graph $\mathbf{G = (V, E)}$ with non-negative edge costs/lengths. $\mathbf{c(e)}$ for edge $\mathbf{e}$

Goal Find a tour of minimum cost that visits each node.

No polynomial time algorithm known. Problem is **NP-Hard**.

Traveling Salesman Problem

Input A graph $\mathbf{G = (V, E)}$ with non-negative edge costs/lengths. $\mathbf{c(e)}$ for edge $\mathbf{e}$

Goal Find a tour of minimum cost that visits each node.

No polynomial time algorithm known. Problem is **NP-Hard**.

Drawings using TSP

Drawings using TSP

Example: optimal tour for cities of a country (which one?)

An Exponential Time Algorithm

How many different tours are there? $n!$

Stirling's formula: $n! \simeq \sqrt{n}(n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?

An Exponential Time Algorithm

How many different tours are there? **$n!$**

Stirling's formula: $n! \simeq \sqrt{n}(n/e)^n$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?

An Exponential Time Algorithm

How many different tours are there? $\mathbf{n!}$

Stirling's formula: $\mathbf{n! \simeq \sqrt{n}(n/e)^n}$ which is $\Theta(2^{cn \log n})$ for some constant $c > 1$

Can we do better? Can we get a $2^{O(n)}$ time algorithm?

An Exponential Time Algorithm

How many different tours are there? $\mathbf{n!}$

Stirling's formula: $\mathbf{n! \simeq \sqrt{n}(n/e)^n}$ which is $\mathbf{\Theta(2^{cn \log n})}$ for some constant $\mathbf{c > 1}$

Can we do better? Can we get a $\mathbf{2^{O(n)}}$ time algorithm?

An Exponential Time Algorithm

How many different tours are there? $\mathbf{n!}$

Stirling's formula: $\mathbf{n! \simeq \sqrt{n}(n/e)^n}$ which is $\mathbf{\Theta(2^{cn \log n})}$ for some constant $\mathbf{c > 1}$

Can we do better? Can we get a $\mathbf{2^{O(n)}}$ time algorithm?

Towards a Recursive Solution

1. Order vertices as $\mathbf{v_1, v_2, \ldots, v_n}$
2. **OPT(S)**: optimum TSP tour for the vertices $\mathbf{S} \subseteq \mathbf{V}$ in the graph restricted to **S**. Want **OPT(V)**.

Can we compute **OPT(S)** recursively?

1. Say $\mathbf{v} \in \mathbf{S}$. What are the two neighbors of **v** in optimum tour in **S**?
2. If **u, w** are neighbors of **v** in an optimum tour of **S** then removing **v** gives an optimum *path* from **u** to **w** visiting all nodes in $\mathbf{S} - \{\mathbf{v}\}$.

Path from **u** to **w** is not a recursive subproblem! Need to find a more general problem to allow recursion.

Towards a Recursive Solution

1. Order vertices as $\mathbf{v_1, v_2, \ldots, v_n}$
2. **OPT(S)**: optimum TSP tour for the vertices $\mathbf{S} \subseteq \mathbf{V}$ in the graph restricted to **S**. Want **OPT(V)**.

Can we compute **OPT(S)** recursively?

1. Say $\mathbf{v} \in \mathbf{S}$. What are the two neighbors of **v** in optimum tour in **S**?
2. If **u, w** are neighbors of **v** in an optimum tour of **S** then removing **v** gives an optimum *path* from **u** to **w** visiting all nodes in $\mathbf{S} - \{\mathbf{v}\}$.

Path from **u** to **w** is not a recursive subproblem! Need to find a more general problem to allow recursion.

Towards a Recursive Solution

1. Order vertices as $\mathbf{v_1, v_2, \ldots, v_n}$
2. **OPT(S)**: optimum TSP tour for the vertices $\mathbf{S} \subseteq \mathbf{V}$ in the graph restricted to **S**. Want **OPT(V)**.

Can we compute **OPT(S)** recursively?

1. Say $\mathbf{v} \in \mathbf{S}$. What are the two neighbors of **v** in optimum tour in **S**?
2. If **u, w** are neighbors of **v** in an optimum tour of **S** then removing **v** gives an optimum *path* from **u** to **w** visiting all nodes in $\mathbf{S} - \{\mathbf{v}\}$.

Path from **u** to **w** is not a recursive subproblem! Need to find a more general problem to allow recursion.

A More General Problem: TSP Path

Input A graph $\mathbf{G = (V, E)}$ with non-negative edge costs/lengths($\mathbf{c(e)}$ for edge $\mathbf{e}$) and two nodes $\mathbf{s, t}$

Goal Find a path from $\mathbf{s}$ to $\mathbf{t}$ of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

1. $\mathbf{OPT(u, v, S)}$: optimum TSP Path from $\mathbf{u}$ to $\mathbf{v}$ in the graph restricted to $\mathbf{S}$ (here $\mathbf{u, v} \in \mathbf{S}$).

A More General Problem: TSP Path

Input A graph $\mathbf{G = (V, E)}$ with non-negative edge costs/lengths($\mathbf{c(e)}$ for edge $\mathbf{e}$) and two nodes $\mathbf{s, t}$

Goal Find a path from $\mathbf{s}$ to $\mathbf{t}$ of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

1. $\mathbf{OPT(u, v, S)}$: optimum TSP Path from $\mathbf{u}$ to $\mathbf{v}$ in the graph restricted to $\mathbf{S}$ (here $\mathbf{u, v} \in \mathbf{S}$).

A More General Problem: TSP Path

Input A graph $\mathbf{G = (V, E)}$ with non-negative edge costs/lengths($\mathbf{c(e)}$ for edge $\mathbf{e}$) and two nodes $\mathbf{s, t}$

Goal Find a path from $\mathbf{s}$ to $\mathbf{t}$ of minimum cost that visits each node exactly once.

Can solve TSP using above. Do you see how?

Recursion for optimum TSP Path problem:

1. $\mathbf{OPT(u, v, S)}$: optimum TSP Path from $\mathbf{u}$ to $\mathbf{v}$ in the graph restricted to $\mathbf{S}$ (here $\mathbf{u, v} \in \mathbf{S}$).

A More General Problem: TSP Path

Continued...

What is the next node in the optimum path from **u** to **v**? Suppose it is **w**. Then what is $\mathbf{OPT(u, v, S)}$?

$$\mathbf{OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})}$$

We do not know **w**! So try all possibilities for **w**.

A More General Problem: TSP Path

Continued...

What is the next node in the optimum path from $\mathbf{u}$ to $\mathbf{v}$? Suppose it is $\mathbf{w}$. Then what is $\mathbf{OPT(u, v, S)}$?

$$\mathbf{OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})}$$

We do not know $\mathbf{w}$! So try all possibilities for $\mathbf{w}$.

A Recursive Solution

$$\mathbf{OPT(u, v, S) = \min_{w \in S, w \neq u,v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)}$$

What are the subproblems for the original problem $\mathbf{OPT(s, t, V)}$? $\mathbf{OPT(u, v, S)}$ for $\mathbf{u, v \in S}$, $\mathbf{S \subseteq V}$.

How many subproblems?

1. number of distinct subsets $\mathbf{S}$ of $\mathbf{V}$ is at most $\mathbf{2^n}$
2. number of pairs of nodes in a set $\mathbf{S}$ is at most $\mathbf{n^2}$
3. hence number of subproblems is $\mathbf{O(n^2 2^n)}$

Exercise: Show that one can compute TSP using above dynamic program in $\mathbf{O(n^3 2^n)}$ time and $\mathbf{O(n^2 2^n)}$ space.

Disadvantage of dynamic programming solution: memory!

A Recursive Solution

$$\mathbf{OPT(u, v, S) = \min_{w \in S, w \neq u,v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)}$$

What are the subproblems for the original problem $\mathbf{OPT(s, t, V)}$?
$\mathbf{OPT(u, v, S)}$ for $\mathbf{u, v \in S}$, $\mathbf{S \subseteq V}$.

How many subproblems?

1. number of distinct subsets $\mathbf{S}$ of $\mathbf{V}$ is at most $\mathbf{2^n}$
2. number of pairs of nodes in a set $\mathbf{S}$ is at most $\mathbf{n^2}$
3. hence number of subproblems is $\mathbf{O(n^2 2^n)}$

Exercise: Show that one can compute TSP using above dynamic program in $\mathbf{O(n^3 2^n)}$ time and $\mathbf{O(n^2 2^n)}$ space.

Disadvantage of dynamic programming solution: memory!

A Recursive Solution

$$\mathbf{OPT(u, v, S) = \min_{w \in S, w \neq u,v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)}$$

What are the subproblems for the original problem $\mathbf{OPT(s, t, V)}$?
$\mathbf{OPT(u, v, S)}$ for $\mathbf{u, v \in S}$, $\mathbf{S \subseteq V}$.

How many subproblems?

1. number of distinct subsets $\mathbf{S}$ of $\mathbf{V}$ is at most $\mathbf{2^n}$
2. number of pairs of nodes in a set $\mathbf{S}$ is at most $\mathbf{n^2}$
3. hence number of subproblems is $\mathbf{O(n^2 2^n)}$

Exercise: Show that one can compute TSP using above dynamic program in $\mathbf{O(n^3 2^n)}$ time and $\mathbf{O(n^2 2^n)}$ space.

Disadvantage of dynamic programming solution: memory!

A Recursive Solution

$$\mathbf{OPT(u, v, S) = \min_{w \in S, w \neq u,v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)}$$

What are the subproblems for the original problem $\mathbf{OPT(s, t, V)}$?
$\mathbf{OPT(u, v, S)}$ for $\mathbf{u, v \in S}$, $\mathbf{S \subseteq V}$.

How many subproblems?

1. number of distinct subsets $\mathbf{S}$ of $\mathbf{V}$ is at most $\mathbf{2^n}$
2. number of pairs of nodes in a set $\mathbf{S}$ is at most $\mathbf{n^2}$
3. hence number of subproblems is $\mathbf{O(n^2 2^n)}$

Exercise: Show that one can compute TSP using above dynamic program in $\mathbf{O(n^3 2^n)}$ time and $\mathbf{O(n^2 2^n)}$ space.

Disadvantage of dynamic programming solution: memory!

A Recursive Solution

$$\mathbf{OPT(u, v, S) = \min_{w \in S, w \neq u,v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)}$$

What are the subproblems for the original problem $\mathbf{OPT(s, t, V)}$?
$\mathbf{OPT(u, v, S)}$ for $\mathbf{u, v \in S}$, $\mathbf{S \subseteq V}$.

How many subproblems?

1. number of distinct subsets $\mathbf{S}$ of $\mathbf{V}$ is at most $\mathbf{2^n}$
2. number of pairs of nodes in a set $\mathbf{S}$ is at most $\mathbf{n^2}$
3. hence number of subproblems is $\mathbf{O(n^2 2^n)}$

Exercise: Show that one can compute TSP using above dynamic program in $\mathbf{O(n^3 2^n)}$ time and $\mathbf{O(n^2 2^n)}$ space.

Disadvantage of dynamic programming solution: memory!

A Recursive Solution

$$\mathbf{OPT(u, v, S) = \min_{w \in S, w \neq u,v} \Big(c(u, w) + OPT(w, v, S - \{u\}) \Big)}$$

What are the subproblems for the original problem $\mathbf{OPT(s, t, V)}$?
$\mathbf{OPT(u, v, S)}$ for $\mathbf{u, v \in S}$, $\mathbf{S \subseteq V}$.

How many subproblems?

1. number of distinct subsets $\mathbf{S}$ of $\mathbf{V}$ is at most $\mathbf{2^n}$
2. number of pairs of nodes in a set $\mathbf{S}$ is at most $\mathbf{n^2}$
3. hence number of subproblems is $\mathbf{O(n^2 2^n)}$

Exercise: Show that one can compute TSP using above dynamic program in $\mathbf{O(n^3 2^n)}$ time and $\mathbf{O(n^2 2^n)}$ space.

Disadvantage of dynamic programming solution: memory!

Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

1. How to come up with the recursion?
2. How to recognize that dynamic programming may apply?

Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

1. How to come up with the recursion?
2. How to recognize that dynamic programming may apply?

Some Tips

1. Problems where there is a *natural* linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.
2. Problems involving trees: recursion based on subtrees.
3. More generally:
   1. Problem admits a natural recursive divide and conquer
   2. If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
   3. If optimal solution depends on all pieces then can apply dynamic programming if *interface/interaction* between pieces is *limited*. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.

Examples

1. Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?
2. Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?
3. Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the subtrees?
4. Independent Set in an graph: break graph into two graphs. What is the interaction? Very high!
5. Knapsack: Split items into two sets of half each. What is the interaction?

Notes

$x_1 \; x_2, x_3 \; \ldots \; x_n$

$y_1 \; y_2 \; y_3 \; \ldots \; y_m$

$x_1 \; x_2 \; \ldots \; x_{\frac{n}{2}} \quad | \quad x_{\frac{n}{2}+1} \; \ldots \; y_n$

$y_1 \; \ldots \; y_{\frac{m}{2}} \quad | \quad y_{\frac{m}{2}+1} \; \ldots \; y_m$

Notes

Notes

$n = 2$

$w_1 = 10^9 - 1$ $\quad w_2 = 10^9 - 5$ $\quad W = 10^9$

$v_1 = 3$ $\quad v_2 = 2$

$n = 100$

$\frac{10^9}{100} + 77$

2^n

nW

$W \leq n^2$

Notes

Greedy Algorithms

Lecture 11

February 25, 2014

Part I

Problems and Terminology

Problem Types

1. Decision Problem: Is the input a YES or NO input?
   Example: Given graph **G**, nodes **s, t**, is there a path from **s** to **t** in **G**?
2. Search Problem: Find a *solution* if input is a YES input.
   Example: Given graph **G**, nodes **s, t**, find an **s**-**t** path.
3. Optimization Problem: Find a *best* solution among all solutions for the input.
   Example: Given graph **G**, nodes **s, t**, find a shortest **s**-**t** path.

Terminology

1. A problem Π consists of an *infinite* collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. For an instance I, $sol(I)$ is a set of feasible solutions to I. *Typical implicit assumption:* given instance I and $y \in \Sigma^*$, there is a way to check (efficiently!) if $y \in sol(I)$. In other words, problem is in **NP**.
4. For optimization problems each solution $s \in sol(I)$ has an associated value. *Typical implicit assumption:* given s, can compute value efficiently.

Problem Types

1. Decision Problem: Given $\mathbf{I}$ output whether $\mathbf{sol(I)} = \emptyset$ or not.
2. Search Problem: Given $\mathbf{I}$, find a solution $\mathbf{s} \in \mathbf{sol(I)}$ if $\mathbf{sol(I)} \neq \emptyset$.
3. Optimization Problem: Given $\mathbf{I}$,
   1. Minimization problem. Find a solution $\mathbf{s} \in \mathbf{sol(I)}$ of minimum value
   2. Maximization problem. Find a solution $\mathbf{s} \in \mathbf{sol(I)}$ of maximum value
   3. Notation: $\mathbf{opt(I)}$: interchangeably (when there is no confusion) used to denote the value of an optimum solution or some fixed optimum solution.

Part II

Greedy Algorithms: Tools and Techniques

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. make decision incrementally in small steps *without backtracking*
2. decision at each step is based on improving *local or current* state in a myopic fashion without paying attention to the *global* situation
3. decisions often based on some fixed and simple *priority* rules

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. make decision incrementally in small steps *without backtracking*
2. decision at each step is based on improving *local or current* state in a myopic fashion without paying attention to the *global* situation
3. decisions often based on some fixed and simple *priority* rules

What is a Greedy Algorithm?

No real consensus on a universal definition.

Greedy algorithms:

1. make decision incrementally in small steps *without backtracking*
2. decision at each step is based on improving *local or current* state in a myopic fashion without paying attention to the *global* situation
3. decisions often based on some fixed and simple *priority* rules

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to a first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work. Easy to lull oneself into believing they work
2. Many greedy algorithms possible for a problem and no structured way to find effective ones

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to a first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work. Easy to lull oneself into believing they work
2. Many greedy algorithms possible for a problem and no structured way to find effective ones

CS 473: Every greedy algorithm needs a proof of correctness

Pros and Cons of Greedy Algorithms

Pros:

1. Usually (too) easy to design greedy algorithms
2. Easy to implement and often run fast since they are simple
3. Several important cases where they are effective/optimal
4. Lead to a first-cut heuristic when problem not well understood

Cons:

1. **Very often** greedy algorithms don't work. Easy to lull oneself into believing they work
2. Many greedy algorithms possible for a problem and no structured way to find effective ones

CS 473: Every greedy algorithm needs a proof of correctness

Greedy Algorithm Types

Crude classification:

1. Non-adaptive: fix some ordering of decisions a priori and stick with the order
2. Adaptive: make decisions adaptively but greedily/locally at each step

Plan:

1. See several examples
2. Pick up some proof techniques

Greedy Algorithm Types

Crude classification:

1. Non-adaptive: fix some ordering of decisions a priori and stick with the order
2. Adaptive: make decisions adaptively but greedily/locally at each step

Plan:

1. See several examples
2. Pick up some proof techniques

Interval Scheduling

Problem (Interval Scheduling)

Input: *A set of jobs with start and finish times to be scheduled on a resource (example: classes and class rooms).*

Goal: *Schedule as many jobs as possible*

1. *Two jobs with overlapping intervals cannot both be scheduled!*

Interval Scheduling

Problem (Interval Scheduling)

Input: *A set of jobs with start and finish times to be scheduled on a resource (example: classes and class rooms).*

Goal: *Schedule as many jobs as possible*

1. *Two jobs with overlapping intervals cannot both be scheduled!*

Greedy Template

```
R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty do
    choose i ∈ R
    add i to X
    remove from R all requests that overlap with i
return the set X
```

Main task: Decide the order in which to process requests in R

ES SP FC EF

Greedy Template

```
R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty do
    choose i ∈ R
    add i to X
    remove from R all requests that overlap with i
return the set X
```

Main task: Decide the order in which to process requests in **R**

ES SP FC EF

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Figure : Counter example for earliest start time

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Figure : Counter example for earliest start time

Earliest Start Time

Process jobs in the order of their starting times, beginning with those that start earliest.

Figure : Counter example for earliest start time

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Figure : Counter example for smallest processing time

Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Figure : Counter example for smallest processing time

Back Counter

Smallest Processing Time

Process jobs in the order of processing time, starting with jobs that require the shortest processing.

Figure : Counter example for smallest processing time

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Figure : Counter example for fewest conflicts

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Figure : Counter example for fewest conflicts

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Figure : Counter example for fewest conflicts

Back Counter

Fewest Conflicts

Process jobs in that have the fewest "conflicts" first.

Figure : Counter example for fewest conflicts

Back Counter

Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.

Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.

Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.

Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.

Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.

Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.

Earliest Finish Time

Process jobs in the order of their finishing times, beginning with those that finish earliest.

Optimal Greedy Algorithm

```
R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    add i to X
    remove from R all requests that overlap with i
return X
```

Theorem

The greedy algorithm that picks jobs in the order of their finishing times is optimal.

Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests $\mathbf{R}$, let $\mathbf{O}$ be an optimal set and let $\mathbf{X}$ be the set returned by the greedy algorithm. Then $\mathbf{O} = \mathbf{X}$? Not likely!

Instead we will show that $|\mathbf{O}| = |\mathbf{X}|$

Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests $\mathbf{R}$, let $\mathbf{O}$ be an optimal set and let $\mathbf{X}$ be the set returned by the greedy algorithm. Then $\mathbf{O} = \mathbf{X}$? Not likely!

Instead we will show that $|\mathbf{O}| = |\mathbf{X}|$

Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests $\mathbf{R}$, let $\mathbf{O}$ be an optimal set and let $\mathbf{X}$ be the set returned by the greedy algorithm. Then $\mathbf{O} = \mathbf{X}$? Not likely!

Instead we will show that $|\mathbf{O}| = |\mathbf{X}|$

Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests **R**, let **O** be an optimal set and let **X** be the set returned by the greedy algorithm. Then **O** = **X**? Not likely!

Instead we will show that $|\mathbf{O}| = |\mathbf{X}|$

Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests **R**, let **O** be an optimal set and let **X** be the set returned by the greedy algorithm. Then **O** = **X**? Not likely!

Instead we will show that $|\mathbf{O}| = |\mathbf{X}|$

Proving Optimality

1. Correctness: Clearly the algorithm returns a set of jobs that does not have any conflicts
2. For a set of requests $\mathbf{R}$, let $\mathbf{O}$ be an optimal set and let $\mathbf{X}$ be the set returned by the greedy algorithm. Then $\mathbf{O} = \mathbf{X}$? Not likely!

Instead we will show that $|\mathbf{O}| = |\mathbf{X}|$

Proof of Optimality: Key Lemma

Lemma

Let $\mathbf{i_1}$ be first interval picked by Greedy. There exists an optimum solution that contains $\mathbf{i_1}$.

Proof.

Let $\mathbf{O}$ be an *arbitrary* optimum solution. If $\mathbf{i_1} \in \mathbf{O}$ we are done.

Claim: If $\mathbf{i_1} \notin \mathbf{O}$ then there is exactly one interval $\mathbf{j_1} \in \mathbf{O}$ that conflicts with $\mathbf{i_1}$. (proof later)

1. Form a new set $\mathbf{O'}$ by removing $\mathbf{j_1}$ from $\mathbf{O}$ and adding $\mathbf{i_1}$, that is $\mathbf{O'} = (\mathbf{O} - \{\mathbf{j_1}\}) \cup \{\mathbf{i_1}\}$.
2. From claim, $\mathbf{O'}$ is a *feasible* solution (no conflicts).
3. Since $|\mathbf{O'}| = |\mathbf{O}|$, $\mathbf{O'}$ is also an optimum solution and it contains $\mathbf{i_1}$. □

Proof of Optimality: Key Lemma

Lemma

Let $\mathbf{i_1}$ be first interval picked by Greedy. There exists an optimum solution that contains $\mathbf{i_1}$.

Proof.

Let $\mathbf{O}$ be an *arbitrary* optimum solution. If $\mathbf{i_1} \in \mathbf{O}$ we are done.
Claim: If $\mathbf{i_1} \notin \mathbf{O}$ then there is exactly one interval $\mathbf{j_1} \in \mathbf{O}$ that conflicts with $\mathbf{i_1}$. (proof later)

1. Form a new set $\mathbf{O'}$ by removing $\mathbf{j_1}$ from $\mathbf{O}$ and adding $\mathbf{i_1}$, that is $\mathbf{O'} = (\mathbf{O} - \{\mathbf{j_1}\}) \cup \{\mathbf{i_1}\}$.
2. From claim, $\mathbf{O'}$ is a *feasible* solution (no conflicts).
3. Since $|\mathbf{O'}| = |\mathbf{O}|$, $\mathbf{O'}$ is also an optimum solution and it contains $\mathbf{i_1}$. □

Proof of Optimality: Key Lemma

Lemma

Let $\mathbf{i_1}$ be first interval picked by Greedy. There exists an optimum solution that contains $\mathbf{i_1}$.

Proof.

Let $\mathbf{O}$ be an *arbitrary* optimum solution. If $\mathbf{i_1} \in \mathbf{O}$ we are done.
Claim: If $\mathbf{i_1} \notin \mathbf{O}$ then there is exactly one interval $\mathbf{j_1} \in \mathbf{O}$ that conflicts with $\mathbf{i_1}$. (proof later)

1. Form a new set $\mathbf{O'}$ by removing $\mathbf{j_1}$ from $\mathbf{O}$ and adding $\mathbf{i_1}$, that is $\mathbf{O'} = (\mathbf{O} - \{\mathbf{j_1}\}) \cup \{\mathbf{i_1}\}$.
2. From claim, $\mathbf{O'}$ is a *feasible* solution (no conflicts).
3. Since $|\mathbf{O'}| = |\mathbf{O}|$, $\mathbf{O'}$ is also an optimum solution and it contains $\mathbf{i_1}$. □

Proof of Claim

Claim

If $\mathbf{i_1} \notin \mathbf{O}$ *then there is exactly one interval* $\mathbf{j_1} \in \mathbf{O}$ *that conflicts with* $\mathbf{i_1}$.

Proof.

1. Suppose $\mathbf{j_1}, \mathbf{j_2} \in \mathbf{O}$ such that $\mathbf{j_1} \neq \mathbf{j_2}$ and both $\mathbf{j_1}$ and $\mathbf{j_2}$ conflict with $\mathbf{i_1}$.
2. Since $\mathbf{i_1}$ has earliest finish time, $\mathbf{j_1}$ and $\mathbf{i_1}$ overlap at $\mathbf{f(i_1)}$.
3. For same reason $\mathbf{j_2}$ also overlaps with $\mathbf{i_1}$ at $\mathbf{f(i_1)}$.
4. Implies that $\mathbf{j_1}, \mathbf{j_2}$ overlap at $\mathbf{f(i_1)}$ contradicting the feasibility of $\mathbf{O}$.

See figure in next slide. □

Figure for proof of Claim

Figure : Since $\mathbf{i_1}$ has the earliest finish time, any interval that conflicts with it does so at $\mathbf{f(i_1)}$. This implies $\mathbf{j_1}$ and $\mathbf{j_2}$ conflict.

Proof of Optimality of Earliest Finish Time First

Proof by Induction on number of intervals.

Base Case: $n = 1$. Trivial since Greedy picks one interval.
Induction Step: Assume theorem holds for $i < n$.
Let I be an instance with n intervals
I': I with i_1 and all intervals that overlap with i_1 removed
$G(I), G(I')$: Solution produced by Greedy on I and I'
From Lemma, there is an optimum solution O to I and $i_1 \in O$.
Let $O' = O - \{i_1\}$. O' is a solution to I'.

$$\begin{aligned} |G(I)| &= 1 + |G(I')| \quad \text{(from Greedy description)} \\ &\geq 1 + |O'| \quad \text{(By induction, } G(I') \text{ is optimum for } I') \\ &= |O| \end{aligned}$$

□

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $O(n \log n)$ time
2. Now choosing least finishing time is $O(1)$
3. Keep track of the finishing time of the last request added to **A**. Then check if starting time of **i** later than that
4. Thus, checking non-overlapping is $O(1)$
5. Total time $O(n \log n + n) = O(n \log n)$

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $O(n \log n)$ time
2. Now choosing least finishing time is $O(1)$
3. Keep track of the finishing time of the last request added to **A**. Then check if starting time of **i** later than that
4. Thus, checking non-overlapping is $O(1)$
5. Total time $O(n \log n + n) = O(n \log n)$

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $\mathbf{O(n \log n)}$ time
2. Now choosing least finishing time is $\mathbf{O(1)}$
3. Keep track of the finishing time of the last request added to **A**. Then check if starting time of **i** later than that
4. Thus, checking non-overlapping is $\mathbf{O(1)}$
5. Total time $\mathbf{O(n \log n + n) = O(n \log n)}$

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $\mathbf{O(n \log n)}$ time
2. Now choosing least finishing time is $\mathbf{O(1)}$
3. Keep track of the finishing time of the last request added to **A**. Then check if starting time of **i** later than that
4. Thus, checking non-overlapping is $\mathbf{O(1)}$
5. Total time $\mathbf{O(n \log n + n) = O(n \log n)}$

Implementation and Running Time

```
Initially R is the set of all requests
X is empty (* X will store all the jobs that will be scheduled
while R is not empty
    choose i ∈ R such that finishing time of i is least
    if i does not overlap with requests in X
        add i to X
    remove i from R
return the set X
```

1. Presort all requests based on finishing time. $\mathbf{O(n \log n)}$ time
2. Now choosing least finishing time is $\mathbf{O(1)}$
3. Keep track of the finishing time of the last request added to **A**. Then check if starting time of **i** later than that
4. Thus, checking non-overlapping is $\mathbf{O(1)}$
5. Total time $\mathbf{O(n \log n + n) = O(n \log n)}$

Comments

1. Interesting Exercise: smallest interval first picks at least half the optimum number of intervals.
2. All requests need not be known at the beginning. Such *online* algorithms are a subject of research

Weighted Interval Scheduling

Suppose we are given $\mathbf{n}$ jobs. Each job $\mathbf{i}$ has a start time $\mathbf{s_i}$, a finish time $\mathbf{f_i}$, and a weight $\mathbf{w_i}$. We would like to find a set $\mathbf{S}$ of compatible jobs whose total weight is maximized. Which of the following greedy algorithms finds the optimum schedule?

(A) Earliest start time first.

(B) Earliest finish time fist.

(C) Highest weight first.

(D) None of the above.

(E) IDK.

Scheduling all Requests

Input A set of lectures, with start and end times

Goal Find the minimum number of classrooms, needed to schedule all the lectures such two lectures do not occur at the same time in the same room.

Figure : A schedule requiring 3 classrooms

Figure : A schedule requiring 4 classrooms

Scheduling all Requests

Input A set of lectures, with start and end times

Goal Find the minimum number of classrooms, needed to schedule all the lectures such two lectures do not occur at the same time in the same room.

Figure : A schedule requiring 4 classrooms

Figure : A schedule requiring 3 classrooms

Scheduling all Requests

Input A set of lectures, with start and end times

Goal Find the minimum number of classrooms, needed to schedule all the lectures such two lectures do not occur at the same time in the same room.

Figure : A schedule requiring 4 classrooms

Figure : A schedule requiring 3 classrooms

Greedy Algorithm

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty do
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1
                and schedule lecture i in d + 1
        d = d + 1
```

What order should we process requests in? According to start times (breaking ties arbitrarily)

Greedy Algorithm

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty do
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1
                and schedule lecture i in d + 1
        d = d + 1
```

What order should we process requests in? According to start times (breaking ties arbitrarily)

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Example of algorithm execution

"Few things are harder to put up with than a good example." – Mark Twain

Depth of Lectures

Definition

1. For a set of lectures **R**, **k** are said to be in conflict if there is some time **t** such that there are **k** lectures going on at time **t**.
2. The depth of a set of lectures **R** is the maximum number of lectures in conflict at any time.

Depth of Lectures

Definition

1. For a set of lectures **R**, **k** are said to be in conflict if there is some time **t** such that there are **k** lectures going on at time **t**.
2. The depth of a set of lectures **R** is the maximum number of lectures in conflict at any time.

Depth of Lectures

Definition

1. For a set of lectures **R**, **k** are said to be in conflict if there is some time **t** such that there are **k** lectures going on at time **t**.
2. The depth of a set of lectures **R** is the maximum number of lectures in conflict at any time.

Depth and Number of Class-rooms

Lemma

For any set **R** *of lectures, the number of class-rooms required is at least the depth of* **R**.

Proof.

All lectures that are in conflict must be scheduled in different rooms. □

Depth and Number of Class-rooms

Lemma

For any set **R** *of lectures, the number of class-rooms required is at least the depth of* **R**.

Proof.

All lectures that are in conflict must be scheduled in different rooms. □

Number of Class-rooms used by Greedy Algorithm

Lemma

Let $\mathbf{d}$ *be the depth of the set of lectures* $\mathbf{R}$*. The number of class-rooms used by the greedy algorithm is* $\mathbf{d}$*.*

Proof.

1. Suppose the greedy algorithm uses more that $\mathbf{d}$ rooms. Let $\mathbf{j}$ be the first lecture that is scheduled in room $\mathbf{d+1}$.
2. Since we process lectures according to start times, there are $\mathbf{d}$ lectures that start (at or) before $\mathbf{j}$ and which are in conflict with $\mathbf{j}$.
3. Thus, *at the start time of* $\mathbf{j}$, there are at least $\mathbf{d+1}$ lectures in conflict, which contradicts the fact that the depth is $\mathbf{d}$. □

Number of Class-rooms used by Greedy Algorithm

Lemma

Let $\mathbf{d}$ *be the depth of the set of lectures* $\mathbf{R}$*. The number of class-rooms used by the greedy algorithm is* $\mathbf{d}$*.*

Proof.

1. Suppose the greedy algorithm uses more that $\mathbf{d}$ rooms. Let $\mathbf{j}$ be the first lecture that is scheduled in room $\mathbf{d+1}$.
2. Since we process lectures according to start times, there are $\mathbf{d}$ lectures that start (at or) before $\mathbf{j}$ and which are in conflict with $\mathbf{j}$.
3. Thus, *at the start time of* $\mathbf{j}$, there are at least $\mathbf{d+1}$ lectures in conflict, which contradicts the fact that the depth is $\mathbf{d}$. □

Number of Class-rooms used by Greedy Algorithm

Lemma

Let $\mathbf{d}$ *be the depth of the set of lectures* $\mathbf{R}$*. The number of class-rooms used by the greedy algorithm is* $\mathbf{d}$*.*

Proof.

1. Suppose the greedy algorithm uses more that $\mathbf{d}$ rooms. Let $\mathbf{j}$ be the first lecture that is scheduled in room $\mathbf{d+1}$.
2. Since we process lectures according to start times, there are $\mathbf{d}$ lectures that start (at or) before $\mathbf{j}$ and which are in conflict with $\mathbf{j}$.
3. Thus, *at the start time of* $\mathbf{j}$, there are at least $\mathbf{d+1}$ lectures in conflict, which contradicts the fact that the depth is $\mathbf{d}$. □

Figure

Correctness

Observation

The greedy algorithm does not schedule two overlapping lectures in the same room.

Theorem

The greedy algorithm is correct and uses the optimal number of class-rooms.

Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3.
4. Total time

Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3.
4. Total time

Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3.
4. Total time

Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3. Checking conflict takes $O(d)$ time.
4. Total time $= O(n \log n + nd)$

Implementation and Running Time

```
Initially R is the set of all requests
d = 0 (* number of classrooms *)
while R is not empty
    choose i ∈ R such that start time of i is earliest
    if i can be scheduled in some class-room k ≤ d
        schedule lecture i in class-room k
    else
        allocate a new class-room d + 1 and schedule lecture i i
        d = d + 1
```

1. Presort according to start times. Picking lecture with earliest start time can be done in $O(1)$ time.
2. Keep track of the finish time of last lecture in each room.
3. With priority queues, checking conflict takes $O(\log d)$ time.
4. Total time $= O(n \log n + n \log d) = O(n \log n)$

Scheduling to Minimize Lateness

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job $\mathbf{i}$ starts at time $\mathbf{s_i}$ then it will finish at time $\mathbf{f_i = s_i + t_i}$, where $\mathbf{t_i}$ is its processing time. $\mathbf{d_i}$: deadline.
3. The lateness of a job is $\mathbf{l_i = \max(0, f_i - d_i)}$.
4. Schedule all jobs such that $\mathbf{L = \max l_i}$ is minimized.

	1	2	3	4	5	6
t_i	3	2	1	4	3	2
d_i	6	8	9	9	14	15

Scheduling to Minimize Lateness

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job $\mathbf{i}$ starts at time $\mathbf{s_i}$ then it will finish at time $\mathbf{f_i = s_i + t_i}$, where $\mathbf{t_i}$ is its processing time. $\mathbf{d_i}$: deadline.
3. The lateness of a job is $\mathbf{l_i = max(0, f_i - d_i)}$.
4. Schedule all jobs such that $\mathbf{L = max\, l_i}$ is minimized.

	1	2	3	4	5	6
$\mathbf{t_i}$	3	2	1	4	3	2
$\mathbf{d_i}$	6	8	9	9	14	15

A Simpler Feasibility Problem

1. Given jobs with deadlines and processing times to be scheduled on a single resource.
2. If a job $\mathbf{i}$ starts at time $\mathbf{s_i}$ then it will finish at time $\mathbf{f_i = s_i + t_i}$, where $\mathbf{t_i}$ is its processing time.
3. Schedule all jobs such that each of them completes before its deadline (in other words $\mathbf{L = \max_i l_i = 0}$).

Definition

A schedule is feasible if all jobs finish before their deadline.

Greedy Template

```
Initially R is the set of all requests
curr_time = 0
while R is not empty do
    choose i ∈ R
    curr_time = curr_time + t_i
    if (curr_time > d_i) then
        return ``no feasible schedule''

return ``found feasible schedule''
```

Main task: Decide the order in which to process jobs in R

Greedy Template

```
Initially R is the set of all requests
curr_time = 0
while R is not empty do
    choose i ∈ R
    curr_time = curr_time + t_i
    if (curr_time > d_i) then
        return ``no feasible schedule''

return ``found feasible schedule''
```

Main task: Decide the order in which to process jobs in **R**

Three Algorithms

1. Shortest job first — sort according to $\mathbf{t_i}$.
2. Shortest slack first — sort according to $\mathbf{d_i} - \mathbf{t_i}$.
3. EDF = Earliest deadline first — sort according to $\mathbf{d_i}$.

Counter examples for first two: exercise

Three Algorithms

1. Shortest job first — sort according to $\mathbf{t_i}$.
2. Shortest slack first — sort according to $\mathbf{d_i} - \mathbf{t_i}$.
3. EDF = Earliest deadline first — sort according to $\mathbf{d_i}$.

Counter examples for first two: exercise

Earliest Deadline First

Theorem

Greedy with EDF *rule for picking requests correctly decides if there is a feasible schedule.*

Proof via an exchange argument.

Idle time: time during which machine is not working.

Lemma

If there is a feasible schedule then there is one with no idle time before all jobs are finished.

Earliest Deadline First

Theorem

Greedy with EDF *rule for picking requests correctly decides if there is a feasible schedule.*

Proof via an exchange argument.

Idle time: time during which machine is not working.

Lemma

If there is a feasible schedule then there is one with no idle time before all jobs are finished.

Earliest Deadline First

Theorem

Greedy with EDF *rule for picking requests correctly decides if there is a feasible schedule.*

Proof via an exchange argument.

Idle time: time during which machine is not working.

Lemma

If there is a feasible schedule then there is one with no idle time before all jobs are finished.

Earliest Deadline First

Theorem

Greedy with EDF *rule for picking requests correctly decides if there is a feasible schedule.*

Proof via an exchange argument.

Idle time: time during which machine is not working.

Lemma

If there is a feasible schedule then there is one with no idle time before all jobs are finished.

Inversions

Definition

A schedule $\mathbf{S}$ is said to have an inversion if there are jobs $\mathbf{i}$ and $\mathbf{j}$ such that $\mathbf{S}$ schedules $\mathbf{i}$ before $\mathbf{j}$, but $\mathbf{d_i} > \mathbf{d_j}$.

Claim

If a schedule $\mathbf{S}$ *has an inversion then there is an inversion between two* adjacently *scheduled jobs.*

Proof: exercise.

Inversions

Definition

A schedule $\mathbf{S}$ is said to have an inversion if there are jobs $\mathbf{i}$ and $\mathbf{j}$ such that $\mathbf{S}$ schedules $\mathbf{i}$ before $\mathbf{j}$, but $\mathbf{d_i} > \mathbf{d_j}$.

Claim

If a schedule $\mathbf{S}$ *has an inversion then there is an inversion between two* adjacently *scheduled jobs.*

Proof: exercise.

Main Lemma

Lemma

If there is a feasible schedule, then there is one with no inversions.

Proof Sketch.

Let **S** be a schedule with minimum number of inversions.

1. If **S** has **0** inversions, done.
2. Suppose **S** has one or more inversions. By claim there are two adjacent jobs **i** and **j** that define an inversion.
3. Swap positions of **i** and **j**.
4. New schedule is still feasible. (Why?)
5. New schedule has one fewer inversion — contradiction!

□

Back to Minimizing Lateness

Goal: schedule to minimize $\mathbf{L} = \mathbf{max_i\, l_i}$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound $\mathbf{L}$, can we check if there is a schedule such that $\mathbf{max_i\, l_i} \leq \mathbf{L}$?

Yes! Set $\mathbf{d'_i = d_i + L}$ for each job $\mathbf{i}$. Use feasibility algorithm with new deadlines.

How can we find *minimum* $\mathbf{L}$? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $\mathbf{L = \max_i l_i}$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound $\mathbf{L}$, can we check if there is a schedule such that $\mathbf{\max_i l_i \leq L}$?

Yes! Set $\mathbf{d'_i = d_i + L}$ for each job $\mathbf{i}$. Use feasibility algorithm with new deadlines.

How can we find *minimum* $\mathbf{L}$? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $\mathbf{L} = \mathbf{max_i\, l_i}$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound $\mathbf{L}$, can we check if there is a schedule such that $\mathbf{max_i\, l_i} \leq \mathbf{L}$?

Yes! Set $\mathbf{d'_i = d_i + L}$ for each job $\mathbf{i}$. Use feasibility algorithm with new deadlines.

How can we find *minimum* $\mathbf{L}$? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $\mathbf{L = \max_i l_i}$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound $\mathbf{L}$, can we check if there is a schedule such that $\mathbf{\max_i l_i \leq L}$?

Yes! Set $\mathbf{d'_i = d_i + L}$ for each job $\mathbf{i}$. Use feasibility algorithm with new deadlines.

How can we find *minimum* $\mathbf{L}$? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $\mathbf{L} = \mathbf{max_i\, l_i}$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound $\mathbf{L}$, can we check if there is a schedule such that $\mathbf{max_i\, l_i} \leq \mathbf{L}$?

Yes! Set $\mathbf{d'_i = d_i + L}$ for each job $\mathbf{i}$. Use feasibility algorithm with new deadlines.

How can we find *minimum* $\mathbf{L}$? Binary search!

Back to Minimizing Lateness

Goal: schedule to minimize $\mathbf{L = \max_i l_i}$.

How can we use algorithm for simpler feasibility problem?

Given a lateness bound $\mathbf{L}$, can we check if there is a schedule such that $\mathbf{\max_i l_i \leq L}$?

Yes! Set $\mathbf{d'_i = d_i + L}$ for each job $\mathbf{i}$. Use feasibility algorithm with new deadlines.

How can we find *minimum* $\mathbf{L}$? Binary search!

Binary search for finding minimum lateness

```
L = L_min = 0
L_max = Σ_i t_i // why is this sufficient?
While L_min < L_max do
    L = ⌊(L_max + L_min)/2⌋
    check if there is a feasible schedule with lateness L
    if ''yes'' then L_max = L
    else L_min = L + 1
end while
return L
```

Running time: $O(n \log n \cdot \log T)$ where $T = \sum_i t_i$

1. $O(n \log n)$ for feasibility test (sort by deadlines)
2. $O(\log T)$ calls to feasibility test in binary search

Binary search for finding minimum lateness

```
L = L_min = 0
L_max = Σ_i t_i // why is this sufficient?
While L_min < L_max do
    L = ⌊(L_max + L_min)/2⌋
    check if there is a feasible schedule with lateness L
    if ``yes'' then L_max = L
    else L_min = L + 1
end while
return L
```

Running time: $O(n \log n \cdot \log T)$ where $T = \sum_i t_i$

1. $O(n \log n)$ for feasibility test (sort by deadlines)
2. $O(\log T)$ calls to feasibility test in binary search

Do we need binary search?

What happens in each call?
EDF algorithm with deadlines $d'_i = d_i + L$.

Greedy with EDF schedules the jobs in the same order for all L!!!

Maybe there is a direct greedy algorithm for minimizing maximum lateness?

Do we need binary search?

What happens in each call?
EDF algorithm with deadlines $d'_i = d_i + L$.

Greedy with EDF schedules the jobs in the same order for all L!!!

Maybe there is a direct greedy algorithm for minimizing maximum lateness?

Do we need binary search?

What happens in each call?
EDF algorithm with deadlines $\mathbf{d'_i = d_i + L}$.

Greedy with EDF schedules the jobs in the same order for all **L**!!!

Maybe there is a direct greedy algorithm for minimizing maximum lateness?

Greedy Algorithm for Minimizing Lateness

```
Initially R is the set of all requests
curr_time = 0
curr_late = 0
while R is not empty
    choose i ∈ R with earliest deadline
    curr_time = curr_time + t_i
    late = curr_time − d_i
    curr_late = max(late, curr_late)
return curr_late
```

Exercise: argue directly that above algorithm is correct

Can be easily implemented in $O(n \log n)$ time after sorting jobs.

Greedy Algorithm for Minimizing Lateness

```
Initially R is the set of all requests
curr_time = 0
curr_late = 0
while R is not empty
    choose i ∈ R with earliest deadline
    curr_time = curr_time + t_i
    late = curr_time − d_i
    curr_late = max(late, curr_late)
return curr_late
```

Exercise: argue directly that above algorithm is correct

Can be easily implemented in $O(n \log n)$ time after sorting jobs.

Greedy Algorithm for Minimizing Lateness

```
Initially R is the set of all requests
curr_time = 0
curr_late = 0
while R is not empty
    choose i ∈ R with earliest deadline
    curr_time = curr_time + t_i
    late = curr_time − d_i
    curr_late = max(late, curr_late)
return curr_late
```

Exercise: argue directly that above algorithm is correct

Can be easily implemented in $O(n \log n)$ time after sorting jobs.

Greedy Analysis: Overview

1. Greedy's first step leads to an optimum solution. Show that there is an optimum solution leading from the first step of Greedy and then use induction. Example, Interval Scheduling.
2. Greedy algorithm stays ahead. Show that after each step the solution of the greedy algorithm is at least as good as the solution of any other algorithm. Example, Interval scheduling.
3. Structural property of solution. Observe some structural bound of every solution to the problem, and show that greedy algorithm achieves this bound. Example, Interval Partitioning.
4. Exchange argument. Gradually transform any optimal solution to the one produced by the greedy algorithm, without hurting its optimality. Example, Minimizing lateness.

Takeaway Points

1. Greedy algorithms come naturally but often are incorrect. A proof of correctness is an absolute necessity.
2. *Exchange* arguments are often the key proof ingredient. Focus on why the first step of the algorithm is correct: need to show that there is an optimum/correct solution with the first step of the algorithm.
3. Thinking about correctness is also a good way to figure out which of the many greedy strategies is likely to work.

Notes

Notes

Notes

Notes

Greedy Algorithms for Minimum Spanning Trees

Lecture 12

March 4, 2014

Part I

Greedy Algorithms: Minimum Spanning Tree

Minimum Spanning Tree

Input Connected graph $\mathbf{G = (V, E)}$ with edge costs

Goal Find $\mathbf{T} \subseteq \mathbf{E}$ such that $\mathbf{(V, T)}$ is connected and total cost of all edges in $\mathbf{T}$ is smallest

1. $\mathbf{T}$ is the minimum spanning tree (MST) of $\mathbf{G}$

Minimum Spanning Tree

Input Connected graph $\mathbf{G = (V, E)}$ with edge costs

Goal Find $\mathbf{T} \subseteq \mathbf{E}$ such that $\mathbf{(V, T)}$ is connected and total cost of all edges in $\mathbf{T}$ is smallest

1. $\mathbf{T}$ is the minimum spanning tree (MST) of $\mathbf{G}$

Applications

1. Network Design
   1. Designing networks with minimum cost but maximum connectivity
2. Approximation algorithms
   1. Can be used to bound the optimality of algorithms to approximate Traveling Salesman Problem, Steiner Trees, etc.
3. Cluster Analysis

Greedy Template

```
Initially E is the set of all edges in G
T is empty (* T will store edges of a MST *)
while E is not empty do
    choose i ∈ E
    if (i satisfies condition)
        add i to T
return the set T
```

Main Task: In what order should edges be processed? When should we add edge to spanning tree?

KA PA RD

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to **T** as long as they don't form a cycle.

Figure : Graph **G**

Figure : MST of **G**

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to **T** as long as they don't form a cycle.

Figure : Graph **G**

Figure : MST of **G**

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to **T** as long as they don't form a cycle.

Figure : Graph **G**

Figure : MST of **G**

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to **T** as long as they don't form a cycle.

Figure : Graph **G**

Figure : MST of **G**

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to **T** as long as they don't form a cycle.

Figure : Graph **G**

Figure : MST of **G**

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to **T** as long as they don't form a cycle.

Figure : Graph **G**

Figure : MST of **G**

Kruskal's Algorithm

Process edges in the order of their costs (starting from the least) and add edges to **T** as long as they don't form a cycle.

Figure : Graph **G**

Figure : MST of **G**

Prim's Algorithm

T maintained by algorithm will be a tree. Start with a node in **T**. In each iteration, pick edge with least attachment cost to **T**.

Figure : Graph **G**

Figure : MST of **G**

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Start with a node in **T**. In each iteration, pick edge with least attachment cost to **T**.

Figure : Graph **G**

Figure : MST of **G**

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Start with a node in **T**. In each iteration, pick edge with least attachment cost to **T**.

Figure : Graph **G**

Figure : MST of **G**

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Start with a node in **T**. In each iteration, pick edge with least attachment cost to **T**.

Figure : Graph **G**

Figure : MST of **G**

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Start with a node in **T**. In each iteration, pick edge with least attachment cost to **T**.

Figure : Graph **G**

Figure : MST of **G**

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Start with a node in **T**. In each iteration, pick edge with least attachment cost to **T**.

Figure : Graph **G**

Figure : MST of **G**

Back

Prim's Algorithm

T maintained by algorithm will be a tree. Start with a node in **T**. In each iteration, pick edge with least attachment cost to **T**.

Figure : Graph **G**

Figure : MST of **G**

Back

Reverse Delete Algorithm

```
Initially E is the set of all edges in G
T is E (* T will store edges of a MST *)
while E is not empty do
    choose i ∈ E of largest cost
    if removing i does not disconnect T then
        remove i from T
return the set T
```

Returns a minimum spanning tree.

Back

Correctness of MST Algorithms

1. Many different MST algorithms
2. All of them rely on some basic properties of MSTs, in particular the **Cut Property** to be seen shortly.

Assumption

And for now . . .

Assumption

Edge costs are distinct, that is no two edge costs are equal.

Cuts

Definition

Given a graph $\mathbf{G = (V, E)}$, a **cut** is a partition of the vertices of the graph into two sets $\mathbf{(S, V \setminus S)}$.

Edges having an endpoint on both sides are the **edges of the cut**.

A cut edge is **crossing** the cut.

Cuts

Definition

Given a graph $\mathbf{G = (V, E)}$, a **cut** is a partition of the vertices of the graph into two sets $\mathbf{(S, V \setminus S)}$.

Edges having an endpoint on both sides are the **edges of the cut**.

A cut edge is **crossing** the cut.

Safe and Unsafe Edges

Definition

An edge $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ is a safe edge if there is some partition of $\mathbf{V}$ into $\mathbf{S}$ and $\mathbf{V} \setminus \mathbf{S}$ and $\mathbf{e}$ is the unique minimum cost edge crossing $\mathbf{S}$ (one end in $\mathbf{S}$ and the other in $\mathbf{V} \setminus \mathbf{S}$).

Definition

An edge $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ is an unsafe edge if there is some cycle $\mathbf{C}$ such that $\mathbf{e}$ is the unique maximum cost edge in $\mathbf{C}$.

Proposition

If edge costs are distinct then every edge is either safe or unsafe.

Proof.

Exercise. □

Safe and Unsafe Edges

Definition

An edge $\mathbf{e = (u, v)}$ is a safe edge if there is some partition of $\mathbf{V}$ into $\mathbf{S}$ and $\mathbf{V \setminus S}$ and $\mathbf{e}$ is the unique minimum cost edge crossing $\mathbf{S}$ (one end in $\mathbf{S}$ and the other in $\mathbf{V \setminus S}$).

Definition

An edge $\mathbf{e = (u, v)}$ is an unsafe edge if there is some cycle $\mathbf{C}$ such that $\mathbf{e}$ is the unique maximum cost edge in $\mathbf{C}$.

Proposition

If edge costs are distinct then every edge is either safe or unsafe.

Proof.

Exercise. □

Safe and Unsafe Edges

Definition

An edge $\mathbf{e = (u, v)}$ is a safe edge if there is some partition of $\mathbf{V}$ into $\mathbf{S}$ and $\mathbf{V \setminus S}$ and $\mathbf{e}$ is the unique minimum cost edge crossing $\mathbf{S}$ (one end in $\mathbf{S}$ and the other in $\mathbf{V \setminus S}$).

Definition

An edge $\mathbf{e = (u, v)}$ is an unsafe edge if there is some cycle $\mathbf{C}$ such that $\mathbf{e}$ is the unique maximum cost edge in $\mathbf{C}$.

Proposition

If edge costs are distinct then every edge is either safe or unsafe.

Proof.

Exercise. □

Safe edge

Example...

Every cut identifies one safe edge...

...the cheapest edge in the cut.

Note: An edge **e** may be a safe edge for *many* cuts!

Safe edge

Example...

Every cut identifies one safe edge...

...the cheapest edge in the cut.

Note: An edge **e** may be a safe edge for *many* cuts!

Unsafe edge

Example...

Every cycle identifies one **unsafe** edge...

...the most expensive edge in the cycle.

Unsafe edge

Example...

Every cycle identifies one **unsafe** edge...

...the most expensive edge in the cycle.

Example

Figure : Graph with unique edge costs. Safe edges are red, rest are unsafe.

And all safe edges are in the MST in this case...

Example

Figure : Graph with unique edge costs. Safe edges are red, rest are unsafe.

And all safe edges are in the MST in this case...

Example

Figure : Graph with unique edge costs. Safe edges are red, rest are unsafe.

And all safe edges are in the MST in this case...

Key Observation: Cut Property

Lemma

If $\mathbf{e}$ *is a safe edge then every minimum spanning tree contains* $\mathbf{e}$.

Proof.

1. Suppose (for contradiction) $\mathbf{e}$ is not in MST $\mathbf{T}$.
2. Since $\mathbf{e}$ is safe there is an $\mathbf{S} \subset \mathbf{V}$ such that $\mathbf{e}$ is the unique min cost edge crossing $\mathbf{S}$.
3. Since $\mathbf{T}$ is connected, there must be some edge $\mathbf{f}$ with one end in $\mathbf{S}$ and the other in $\mathbf{V} \setminus \mathbf{S}$
4. Since $c_f > c_e$, $T' = (T \setminus \{f\}) \cup \{e\}$ is a spanning tree of lower cost! Error: T' may not be a spanning tree!!

□

Key Observation: Cut Property

Lemma

If $\mathbf{e}$ *is a safe edge then every minimum spanning tree contains* $\mathbf{e}$.

Proof.

1. Suppose (for contradiction) $\mathbf{e}$ is not in MST $\mathbf{T}$.
2. Since $\mathbf{e}$ is safe there is an $\mathbf{S} \subset \mathbf{V}$ such that $\mathbf{e}$ is the unique min cost edge crossing $\mathbf{S}$.
3. Since $\mathbf{T}$ is connected, there must be some edge $\mathbf{f}$ with one end in $\mathbf{S}$ and the other in $\mathbf{V} \setminus \mathbf{S}$
4. Since $c_f > c_e$, $\mathbf{T}' = (\mathbf{T} \setminus \{\mathbf{f}\}) \cup \{\mathbf{e}\}$ is a spanning tree of lower cost! Error: $\mathbf{T}'$ may not be a spanning tree!!

□

Key Observation: Cut Property

Lemma

If $\mathbf{e}$ *is a safe edge then every minimum spanning tree contains* $\mathbf{e}$.

Proof.

1. Suppose (for contradiction) $\mathbf{e}$ is not in MST $\mathbf{T}$.
2. Since $\mathbf{e}$ is safe there is an $\mathbf{S} \subset \mathbf{V}$ such that $\mathbf{e}$ is the unique min cost edge crossing $\mathbf{S}$.
3. Since $\mathbf{T}$ is connected, there must be some edge $\mathbf{f}$ with one end in $\mathbf{S}$ and the other in $\mathbf{V} \setminus \mathbf{S}$
4. Since $\mathbf{c_f} > \mathbf{c_e}$, $\mathbf{T'} = (\mathbf{T} \setminus \{\mathbf{f}\}) \cup \{\mathbf{e}\}$ is a spanning tree of lower cost! Error: $\mathbf{T'}$ may not be a spanning tree!!

□

Error in Proof: Example

Problematic example. $\mathbf{S} = \{1, 2, 7\}$, $\mathbf{e} = (7, 3)$, $\mathbf{f} = (1, 6)$. $\mathbf{T} - \mathbf{f} + \mathbf{e}$ is not a spanning tree.

(A)

1. (A) Consider adding the edge **f**.

Error in Proof: Example

Problematic example. $\mathbf{S} = \{1, 2, 7\}$, $\mathbf{e} = (7, 3)$, $\mathbf{f} = (1, 6)$. $\mathbf{T} - \mathbf{f} + \mathbf{e}$ is not a spanning tree.

(B)

1. (A) Consider adding the edge **f**.
2. (B) It is safe because it is the cheapest edge in the cut.

Error in Proof: Example

Problematic example. $\mathbf{S} = \{1, 2, 7\}$, $\mathbf{e} = (7, 3)$, $\mathbf{f} = (1, 6)$. $\mathbf{T} - \mathbf{f} + \mathbf{e}$ is not a spanning tree.

(C)

1. (A) Consider adding the edge **f**.
2. (B) It is safe because it is the cheapest edge in the cut.
3. (C) Lets throw out the edge **e** currently in the spanning tree which is more expensive than **f** and is in the same cut. Put it **f** instead...

Error in Proof: Example

Problematic example. $\mathbf{S} = \{\mathbf{1}, \mathbf{2}, \mathbf{7}\}$, $\mathbf{e} = (\mathbf{7}, \mathbf{3})$, $\mathbf{f} = (\mathbf{1}, \mathbf{6})$. $\mathbf{T} - \mathbf{f} + \mathbf{e}$ is not a spanning tree.

(D)

1. (A) Consider adding the edge **f**.
2. (B) It is safe because it is the cheapest edge in the cut.
3. (C) Lets throw out the edge **e** currently in the spanning tree which is more expensive than **f** and is in the same cut. Put it **f** instead...
4. (D) New graph of selected edges is not a tree anymore. BUG.

Proof of Cut Property

Proof.

1. Suppose $e = (v, w)$ is not in MST T and e is min weight edge in cut $(S, V \setminus S)$. Assume $v \in S$.
2. T is spanning tree: there is a unique path P from v to w in T
3. Let w' be the first vertex in P belonging to $V \setminus S$; let v' be the vertex just before it on P, and let $e' = (v', w')$
4. $T' = (T \setminus \{e'\}) \cup \{e\}$ is spanning tree of lower cost. (Why?) □

Proof of Cut Property

Proof.

1. Suppose $\mathbf{e} = (\mathbf{v}, \mathbf{w})$ is not in MST $\mathbf{T}$ and $\mathbf{e}$ is min weight edge in cut $(\mathbf{S}, \mathbf{V} \setminus \mathbf{S})$. Assume $\mathbf{v} \in \mathbf{S}$.
2. $\mathbf{T}$ is spanning tree: there is a unique path $\mathbf{P}$ from $\mathbf{v}$ to $\mathbf{w}$ in $\mathbf{T}$
3. Let $\mathbf{w}'$ be the first vertex in $\mathbf{P}$ belonging to $\mathbf{V} \setminus \mathbf{S}$; let $\mathbf{v}'$ be the vertex just before it on $\mathbf{P}$, and let $\mathbf{e}' = (\mathbf{v}', \mathbf{w}')$
4. $\mathbf{T}' = (\mathbf{T} \setminus \{\mathbf{e}'\}) \cup \{\mathbf{e}\}$ is spanning tree of lower cost. (Why?) □

Proof of Cut Property

Proof.

1. Suppose $\mathbf{e} = (\mathbf{v}, \mathbf{w})$ is not in MST $\mathbf{T}$ and $\mathbf{e}$ is min weight edge in cut $(\mathbf{S}, \mathbf{V} \setminus \mathbf{S})$. Assume $\mathbf{v} \in \mathbf{S}$.
2. $\mathbf{T}$ is spanning tree: there is a unique path $\mathbf{P}$ from $\mathbf{v}$ to $\mathbf{w}$ in $\mathbf{T}$
3. Let $\mathbf{w}'$ be the first vertex in $\mathbf{P}$ belonging to $\mathbf{V} \setminus \mathbf{S}$; let $\mathbf{v}'$ be the vertex just before it on $\mathbf{P}$, and let $\mathbf{e}' = (\mathbf{v}', \mathbf{w}')$
4. $\mathbf{T}' = (\mathbf{T} \setminus \{\mathbf{e}'\}) \cup \{\mathbf{e}\}$ is spanning tree of lower cost. (Why?) □

Proof of Cut Property

Proof.

1. Suppose $\mathbf{e} = (\mathbf{v}, \mathbf{w})$ is not in MST $\mathbf{T}$ and $\mathbf{e}$ is min weight edge in cut $(\mathbf{S}, \mathbf{V} \setminus \mathbf{S})$. Assume $\mathbf{v} \in \mathbf{S}$.
2. $\mathbf{T}$ is spanning tree: there is a unique path $\mathbf{P}$ from $\mathbf{v}$ to $\mathbf{w}$ in $\mathbf{T}$
3. Let $\mathbf{w}'$ be the first vertex in $\mathbf{P}$ belonging to $\mathbf{V} \setminus \mathbf{S}$; let $\mathbf{v}'$ be the vertex just before it on $\mathbf{P}$, and let $\mathbf{e}' = (\mathbf{v}', \mathbf{w}')$
4. $\mathbf{T}' = (\mathbf{T} \setminus \{\mathbf{e}'\}) \cup \{\mathbf{e}\}$ is spanning tree of lower cost. (Why?) □

Proof of Cut Property

Proof.

1. Suppose $\mathbf{e} = (\mathbf{v}, \mathbf{w})$ is not in MST $\mathbf{T}$ and $\mathbf{e}$ is min weight edge in cut $(\mathbf{S}, \mathbf{V} \setminus \mathbf{S})$. Assume $\mathbf{v} \in \mathbf{S}$.
2. $\mathbf{T}$ is spanning tree: there is a unique path $\mathbf{P}$ from $\mathbf{v}$ to $\mathbf{w}$ in $\mathbf{T}$
3. Let $\mathbf{w}'$ be the first vertex in $\mathbf{P}$ belonging to $\mathbf{V} \setminus \mathbf{S}$; let $\mathbf{v}'$ be the vertex just before it on $\mathbf{P}$, and let $\mathbf{e}' = (\mathbf{v}', \mathbf{w}')$
4. $\mathbf{T}' = (\mathbf{T} \setminus \{\mathbf{e}'\}) \cup \{\mathbf{e}\}$ is spanning tree of lower cost. (Why?) $\square$

Proof of Cut Property (contd)

Observation

$\mathbf{T'} = (\mathbf{T} \setminus \{\mathbf{e'}\}) \cup \{\mathbf{e}\}$ *is a spanning tree.*

Proof.

$\mathbf{T'}$ is connected.

Removed $\mathbf{e'} = (\mathbf{v'}, \mathbf{w'})$ from $\mathbf{T}$ but $\mathbf{v'}$ and $\mathbf{w'}$ are connected by the path $\mathbf{P} - \mathbf{f} + \mathbf{e}$ in $\mathbf{T'}$. Hence $\mathbf{T'}$ is connected if $\mathbf{T}$ is.

$\mathbf{T'}$ is a tree

$\mathbf{T'}$ is connected and has $\mathbf{n} - \mathbf{1}$ edges (since $\mathbf{T}$ had $\mathbf{n} - \mathbf{1}$ edges) and hence $\mathbf{T'}$ is a tree

□

Proof of Cut Property (contd)

Observation

$\mathbf{T'} = (\mathbf{T} \setminus \{\mathbf{e'}\}) \cup \{\mathbf{e}\}$ *is a spanning tree.*

Proof.

$\mathbf{T'}$ is connected.

Removed $\mathbf{e'} = (\mathbf{v'}, \mathbf{w'})$ from $\mathbf{T}$ but $\mathbf{v'}$ and $\mathbf{w'}$ are connected by the path $\mathbf{P} - \mathbf{f} + \mathbf{e}$ in $\mathbf{T'}$. Hence $\mathbf{T'}$ is connected if $\mathbf{T}$ is.

$\mathbf{T'}$ is a tree

$\mathbf{T'}$ is connected and has $\mathbf{n} - \mathbf{1}$ edges (since $\mathbf{T}$ had $\mathbf{n} - \mathbf{1}$ edges) and hence $\mathbf{T'}$ is a tree

□

Proof of Cut Property (contd)

Observation

$\mathbf{T'} = (\mathbf{T} \setminus \{\mathbf{e'}\}) \cup \{\mathbf{e}\}$ *is a spanning tree.*

Proof.

$\mathbf{T'}$ is connected.

Removed $\mathbf{e'} = (\mathbf{v'}, \mathbf{w'})$ from $\mathbf{T}$ but $\mathbf{v'}$ and $\mathbf{w'}$ are connected by the path $\mathbf{P} - \mathbf{f} + \mathbf{e}$ in $\mathbf{T'}$. Hence $\mathbf{T'}$ is connected if $\mathbf{T}$ is.

$\mathbf{T'}$ is a tree

$\mathbf{T'}$ is connected and has $\mathbf{n} - \mathbf{1}$ edges (since $\mathbf{T}$ had $\mathbf{n} - \mathbf{1}$ edges) and hence $\mathbf{T'}$ is a tree

□

Safe Edges form a Tree

Lemma

Let **G** *be a connected graph with distinct edge costs, then the set of safe edges form a connected graph.*

Proof.

1. Suppose not. Let **S** be a connected component in the graph induced by the safe edges.
2. Consider the edges crossing **S**, there must be a safe edge among them since edge costs are distinct and so we must have picked it.

□

Safe Edges form an MST

Corollary

Let **G** *be a connected graph with distinct edge costs, then set of safe edges form the unique* MST *of* **G**.

Consequence: Every correct MST algorithm when **G** has unique edge costs includes exactly the safe edges.

Safe Edges form an MST

Corollary

Let **G** *be a connected graph with distinct edge costs, then set of safe edges form the unique* MST *of* **G**.

Consequence: Every correct MST algorithm when **G** has unique edge costs includes exactly the safe edges.

Cycle Property

Lemma

If **e** *is an unsafe edge then no* MST *of* **G** *contains* **e**.

Proof.

Exercise. □

Note: Cut and Cycle properties hold even when edge costs are not distinct. Safe and unsafe definitions do not rely on distinct cost assumption.

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

Proof of correctness.

1. If e is added to tree, then e is safe and belongs to every MST.
   1. Let S be the vertices connected by edges in T when e is added.
   2. e is edge of lowest cost with one end in S and the other in $V \setminus S$ and hence e is safe.
2. Set of edges output is a spanning tree
   1. Set of edges output forms a connected graph: by induction, S is connected in each iteration and eventually $S = V$.
   2. Only safe edges added and they do not have a cycle $\square$

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

Proof of correctness.

1. If **e** is added to tree, then **e** is safe and belongs to every MST.
   1. Let **S** be the vertices connected by edges in **T** when **e** is added.
   2. **e** is edge of lowest cost with one end in **S** and the other in $\mathbf{V} \setminus \mathbf{S}$ and hence **e** is safe.
2. Set of edges output is a spanning tree
   1. Set of edges output forms a connected graph: by induction, **S** is connected in each iteration and eventually $\mathbf{S} = \mathbf{V}$.
   2. Only safe edges added and they do not have a cycle □

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

Proof of correctness.

1. If **e** is added to tree, then **e** is safe and belongs to every MST.
   1. Let **S** be the vertices connected by edges in **T** when **e** is added.
   2. **e** is edge of lowest cost with one end in **S** and the other in $\mathbf{V} \setminus \mathbf{S}$ and hence **e** is safe.
2. Set of edges output is a spanning tree
   1. Set of edges output forms a connected graph: by induction, **S** is connected in each iteration and eventually $\mathbf{S} = \mathbf{V}$.
   2. Only safe edges added and they do not have a cycle □

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

Proof of correctness.

1. If $\mathbf{e}$ is added to tree, then $\mathbf{e}$ is safe and belongs to every MST.
   1. Let $\mathbf{S}$ be the vertices connected by edges in $\mathbf{T}$ when $\mathbf{e}$ is added.
   2. $\mathbf{e}$ is edge of lowest cost with one end in $\mathbf{S}$ and the other in $\mathbf{V} \setminus \mathbf{S}$ and hence $\mathbf{e}$ is safe.
2. Set of edges output is a spanning tree
   1. Set of edges output forms a connected graph: by induction, $\mathbf{S}$ is connected in each iteration and eventually $\mathbf{S} = \mathbf{V}$.
   2. Only safe edges added and they do not have a cycle $\square$

Correctness of Prim's Algorithm

Prim's Algorithm

Pick edge with minimum attachment cost to current tree, and add to current tree.

Proof of correctness.

1. If $\mathbf{e}$ is added to tree, then $\mathbf{e}$ is safe and belongs to every MST.
   1. Let $\mathbf{S}$ be the vertices connected by edges in $\mathbf{T}$ when $\mathbf{e}$ is added.
   2. $\mathbf{e}$ is edge of lowest cost with one end in $\mathbf{S}$ and the other in $\mathbf{V} \setminus \mathbf{S}$ and hence $\mathbf{e}$ is safe.
2. Set of edges output is a spanning tree
   1. Set of edges output forms a connected graph: by induction, $\mathbf{S}$ is connected in each iteration and eventually $\mathbf{S} = \mathbf{V}$.
   2. Only safe edges added and they do not have a cycle $\square$

Correctness of Kruskal's Algorithm

Kruskal's Algorithm

Pick edge of lowest cost and add if it does not form a cycle with existing edges.

Proof of correctness.

1. If $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ is added to tree, then $\mathbf{e}$ is safe
   1. When algorithm adds $\mathbf{e}$ let $\mathbf{S}$ and $\mathbf{S}$' be the connected components containing $\mathbf{u}$ and $\mathbf{v}$ respectively
   2. $\mathbf{e}$ is the lowest cost edge crossing $\mathbf{S}$ (and also $\mathbf{S}$').
   3. If there is an edge $\mathbf{e}'$ crossing $\mathbf{S}$ and has lower cost than $\mathbf{e}$, then $\mathbf{e}'$ would come before $\mathbf{e}$ in the sorted order and would be added by the algorithm to $\mathbf{T}$
2. Set of edges output is a spanning tree : exercise

□

Correctness of Kruskal's Algorithm

Kruskal's Algorithm

Pick edge of lowest cost and add if it does not form a cycle with existing edges.

Proof of correctness.

1. If $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ is added to tree, then $\mathbf{e}$ is safe
   1. When algorithm adds $\mathbf{e}$ let $\mathbf{S}$ and $\mathbf{S}$' be the connected components containing $\mathbf{u}$ and $\mathbf{v}$ respectively
   2. $\mathbf{e}$ is the lowest cost edge crossing $\mathbf{S}$ (and also $\mathbf{S}$').
   3. If there is an edge $\mathbf{e}'$ crossing $\mathbf{S}$ and has lower cost than $\mathbf{e}$, then $\mathbf{e}'$ would come before $\mathbf{e}$ in the sorted order and would be added by the algorithm to $\mathbf{T}$
2. Set of edges output is a spanning tree : exercise

□

Correctness of Kruskal's Algorithm

Kruskal's Algorithm

Pick edge of lowest cost and add if it does not form a cycle with existing edges.

Proof of correctness.

1. If $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ is added to tree, then $\mathbf{e}$ is safe
   1. When algorithm adds $\mathbf{e}$ let $\mathbf{S}$ and $\mathbf{S}$' be the connected components containing $\mathbf{u}$ and $\mathbf{v}$ respectively
   2. $\mathbf{e}$ is the lowest cost edge crossing $\mathbf{S}$ (and also $\mathbf{S}$').
   3. If there is an edge $\mathbf{e}'$ crossing $\mathbf{S}$ and has lower cost than $\mathbf{e}$, then $\mathbf{e}'$ would come before $\mathbf{e}$ in the sorted order and would be added by the algorithm to $\mathbf{T}$
2. Set of edges output is a spanning tree : exercise

□

Correctness of Kruskal's Algorithm

Kruskal's Algorithm

Pick edge of lowest cost and add if it does not form a cycle with existing edges.

Proof of correctness.

1. If $\mathbf{e} = (\mathbf{u}, \mathbf{v})$ is added to tree, then $\mathbf{e}$ is safe
   1. When algorithm adds $\mathbf{e}$ let $\mathbf{S}$ and $\mathbf{S}$' be the connected components containing $\mathbf{u}$ and $\mathbf{v}$ respectively
   2. $\mathbf{e}$ is the lowest cost edge crossing $\mathbf{S}$ (and also $\mathbf{S}$').
   3. If there is an edge $\mathbf{e}'$ crossing $\mathbf{S}$ and has lower cost than $\mathbf{e}$, then $\mathbf{e}'$ would come before $\mathbf{e}$ in the sorted order and would be added by the algorithm to $\mathbf{T}$
2. Set of edges output is a spanning tree : exercise

□

Correctness of Reverse Delete Algorithm

Reverse Delete Algorithm

Consider edges in decreasing cost and remove an edge if it does not disconnect the graph

Proof of correctness.

Argue that only unsafe edges are removed. □

When edge costs are not distinct

Heuristic argument: Make edge costs distinct by adding a small tiny and different cost to each edge

Formal argument: Order edges lexicographically to break ties

1. $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or ($c(e_i) = c(e_j)$ and $i < j$)
2. Lexicographic ordering extends to sets of edges. If $A, B \subseteq E$, $A \neq B$ then $A \prec B$ if either $c(A) < c(B)$ or ($c(A) = c(B)$ and $A \setminus B$ has a lower indexed edge than $B \setminus A$)
3. Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

Prim's, Kruskal, and Reverse Delete Algorithms are optimal with respect to lexicographic ordering.

When edge costs are not distinct

Heuristic argument: Make edge costs distinct by adding a small tiny and different cost to each edge

Formal argument: Order edges lexicographically to break ties

1. $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or ($c(e_i) = c(e_j)$ and $i < j$)
2. Lexicographic ordering extends to sets of edges. If $A, B \subseteq E$, $A \neq B$ then $A \prec B$ if either $c(A) < c(B)$ or ($c(A) = c(B)$ and $A \setminus B$ has a lower indexed edge than $B \setminus A$)
3. Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

Prim's, Kruskal, and Reverse Delete Algorithms are optimal with respect to lexicographic ordering.

When edge costs are not distinct

Heuristic argument: Make edge costs distinct by adding a small tiny and different cost to each edge

Formal argument: Order edges lexicographically to break ties

1. $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or ($c(e_i) = c(e_j)$ and $i < j$)
2. Lexicographic ordering extends to sets of edges. If $A, B \subseteq E$, $A \neq B$ then $A \prec B$ if either $c(A) < c(B)$ or ($c(A) = c(B)$ and $A \setminus B$ has a lower indexed edge than $B \setminus A$)
3. Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

Prim's, Kruskal, and Reverse Delete Algorithms are optimal with respect to lexicographic ordering.

When edge costs are not distinct

Heuristic argument: Make edge costs distinct by adding a small tiny and different cost to each edge

Formal argument: Order edges lexicographically to break ties

1. $e_i \prec e_j$ if either $c(e_i) < c(e_j)$ or ($c(e_i) = c(e_j)$ and $i < j$)
2. Lexicographic ordering extends to sets of edges. If $A, B \subseteq E$, $A \neq B$ then $A \prec B$ if either $c(A) < c(B)$ or ($c(A) = c(B)$ and $A \setminus B$ has a lower indexed edge than $B \setminus A$)
3. Can order all spanning trees according to lexicographic order of their edge sets. Hence there is a unique MST.

Prim's, Kruskal, and Reverse Delete Algorithms are optimal with respect to lexicographic ordering.

Edge Costs: Positive and Negative

1. Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
2. Another way to see this: make edge costs non-negative by adding to each edge a large enough positive number. Why does this work for MSTs but not for shortest paths?
3. Can compute *maximum* weight spanning tree by negating edge costs and then computing an MST.

   Question: Why does this not work for shortest paths?

Edge Costs: Positive and Negative

1. Algorithms and proofs don't assume that edge costs are non-negative! MST algorithms work for arbitrary edge costs.
2. Another way to see this: make edge costs non-negative by adding to each edge a large enough positive number. Why does this work for MSTs but not for shortest paths?
3. Can compute *maximum* weight spanning tree by negating edge costs and then computing an MST.
   Question: Why does this not work for shortest paths?

Part II

Data Structures for MST: Priority Queues and Union-Find

Implementing Prim's Algorithm

Implementing Prim's Algorithm

Prim_ComputeMST

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
while S ≠ V do
    pick e = (v, w) ∈ E such that
        v ∈ S and w ∈ V − S
        e has minimum cost
    T = T ∪ e
    S = S ∪ w
return the set T
```

Analysis

1. Number of iterations = $O(n)$, where n is number of vertices
2. Picking e is $O(m)$ where m is the number of edges
3. Total time $O(nm)$

Implementing Prim's Algorithm

Implementing Prim's Algorithm

Prim_ComputeMST

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
while S ≠ V do
    pick e = (v, w) ∈ E such that
        v ∈ S and w ∈ V − S
        e has minimum cost
    T = T ∪ e
    S = S ∪ w
return the set T
```

Analysis

1. Number of iterations = $O(n)$, where n is number of vertices
2. Picking e is $O(m)$ where m is the number of edges
3. Total time $O(nm)$

Implementing Prim's Algorithm

Implementing Prim's Algorithm

Prim_ComputeMST

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
while S ≠ V do
    pick e = (v, w) ∈ E such that
        v ∈ S and w ∈ V − S
        e has minimum cost
    T = T ∪ e
    S = S ∪ w
return the set T
```

Analysis

1. Number of iterations = $O(n)$, where n is number of vertices
2. Picking e is $O(m)$ where m is the number of edges
3. Total time $O(nm)$

Implementing Prim's Algorithm

Implementing Prim's Algorithm

Prim_ComputeMST

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
while S ≠ V do
    pick e = (v, w) ∈ E such that
        v ∈ S and w ∈ V − S
        e has minimum cost
    T = T ∪ e
    S = S ∪ w
return the set T
```

Analysis

1. Number of iterations = $O(n)$, where n is number of vertices
2. Picking e is $O(m)$ where m is the number of edges
3. Total time $O(nm)$

Implementing Prim's Algorithm

More Efficient Implementation

Prim_ComputeMST

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v ∉ S, a(v) = min_{w∈S} c(w, v)
for v ∉ S, e(v) = w such that w ∈ S and c(w, v) is minimum
while S ≠ V do
    pick v with minimum a(v)
    T = T ∪ {(e(v), v)}
    S = S ∪ {v}
    update arrays a and e
return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key $a(v)$.

Implementing Prim's Algorithm

More Efficient Implementation

Prim_ComputeMST

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v ∉ S, a(v) = min_{w∈S} c(w, v)
for v ∉ S, e(v) = w such that w ∈ S and c(w, v) is minimum
while S ≠ V do
    pick v with minimum a(v)
    T = T ∪ {(e(v), v)}
    S = S ∪ {v}
    update arrays a and e
return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key $a(v)$.

Implementing Prim's Algorithm

More Efficient Implementation

Prim_ComputeMST

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v ∉ S, a(v) = min_{w∈S} c(w, v)
for v ∉ S, e(v) = w such that w ∈ S and c(w, v) is minimum
while S ≠ V do
    pick v with minimum a(v)
    T = T ∪ {(e(v), v)}
    S = S ∪ {v}
    update arrays a and e
return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key $a(v)$.

Priority Queues

Data structure to store a set $\mathbf{S}$ of $\mathbf{n}$ elements where each element $\mathbf{v} \in \mathbf{S}$ has an associated real/integer key $\mathbf{k(v)}$ such that the following operations

1. **makeQ**: create an empty queue
2. **findMin**: find the minimum key in $\mathbf{S}$
3. **extractMin**: Remove $\mathbf{v} \in \mathbf{S}$ with smallest key and return it
4. **add**($\mathbf{v}$, $\mathbf{k(v)}$): Add new element $\mathbf{v}$ with key $\mathbf{k(v)}$ to $\mathbf{S}$
5. **Delete**($\mathbf{v}$): Remove element $\mathbf{v}$ from $\mathbf{S}$
6. **decreaseKey** ($\mathbf{v}$, $\mathbf{k'(v)}$): *decrease* key of $\mathbf{v}$ from $\mathbf{k(v)}$ (current key) to $\mathbf{k'(v)}$ (new key). Assumption: $\mathbf{k'(v)} \leq \mathbf{k(v)}$
7. **meld**: merge two separate priority queues into one

Prim's using priority queues

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v ∉ S, a(v) = min_{w∈S} c(w, v)
for v ∉ S, e(v) = w such that w ∈ S and c(w, v) is minimum
while S ≠ V do
    pick v with minimum a(v)
    T = T ∪ {(e(v), v)}
    S = S ∪ {v}
    update arrays a and e
return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key $a(v)$

1. Requires $O(n)$ **extractMin** operations
2. Requires $O(m)$ **decreaseKey** operations

Prim's using priority queues

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v ∉ S, a(v) = min_{w∈S} c(w, v)
for v ∉ S, e(v) = w such that w ∈ S and c(w, v) is minimum
while S ≠ V do
    pick v with minimum a(v)
    T = T ∪ {(e(v), v)}
    S = S ∪ {v}
    update arrays a and e
return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key $a(v)$

1. Requires $O(n)$ **extractMin** operations
2. Requires $O(m)$ **decreaseKey** operations

Prim's using priority queues

```
E is the set of all edges in G
S = {1}
T is empty (* T will store edges of a MST *)
for v ∉ S, a(v) = min_{w∈S} c(w, v)
for v ∉ S, e(v) = w such that w ∈ S and c(w, v) is minimum
while S ≠ V do
    pick v with minimum a(v)
    T = T ∪ {(e(v), v)}
    S = S ∪ {v}
    update arrays a and e
return the set T
```

Maintain vertices in $V \setminus S$ in a priority queue with key $a(v)$

1. Requires $O(n)$ **extractMin** operations
2. Requires $O(m)$ **decreaseKey** operations

Running time of Prim's Algorithm

O(n) **extractMin** operations and **O(m)** **decreaseKey** operations

1. Using standard Heaps, **extractMin** and **decreaseKey** take $\mathbf{O(\log n)}$ time. Total: $\mathbf{O((m+n)\log n)}$
2. Using Fibonacci Heaps, $\mathbf{O(\log n)}$ for **extractMin** and $\mathbf{O(1)}$ (amortized) for **decreaseKey**. Total: $\mathbf{O(n \log n + m)}$.

Prim's algorithm and Dijkstra's algorithms are similar. Where is the difference?

Running time of Prim's Algorithm

O(n) **extractMin** operations and **O(m)** **decreaseKey** operations

1. Using standard Heaps, **extractMin** and **decreaseKey** take $\mathbf{O(\log n)}$ time. Total: $\mathbf{O((m+n)\log n)}$
2. Using Fibonacci Heaps, $\mathbf{O(\log n)}$ for **extractMin** and $\mathbf{O(1)}$ (amortized) for **decreaseKey**. Total: $\mathbf{O(n \log n + m)}$.

Prim's algorithm and Dijkstra's algorithms are similar. Where is the difference?

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

1. Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
2. Do **BFS**/**DFS** on $T \cup \{e\}$. Takes $O(n)$ time
3. Total time $O(m \log m) + O(mn) = O(mn)$

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

1. Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
2. Do **BFS**/**DFS** on $T \cup \{e\}$. Takes $O(n)$ time
3. Total time $O(m \log m) + O(mn) = O(mn)$

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

1. Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
2. Do BFS/DFS on $T \cup \{e\}$. Takes $O(n)$ time
3. Total time $O(m \log m) + O(mn) = O(mn)$

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

1. Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
2. Do BFS/DFS on $T \cup \{e\}$. Takes $O(n)$ time
3. Total time $O(m \log m) + O(mn) = O(mn)$

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

1. Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
2. Do **BFS**/**DFS** on $T \cup \{e\}$. Takes $O(n)$ time
3. Total time $O(m \log m) + O(mn) = O(mn)$

Kruskal's Algorithm

```
Kruskal_ComputeMST
    Initially E is the set of all edges in G
    T is empty (* T will store edges of a MST *)
    while E is not empty do
        choose e ∈ E of minimum cost
        if (T ∪ {e} does not have cycles)
            add e to T
    return the set T
```

1. Presort edges based on cost. Choosing minimum can be done in $O(1)$ time
2. Do **BFS**/**DFS** on $T \cup \{e\}$. Takes $O(n)$ time
3. Total time $O(m \log m) + O(mn) = O(mn)$

Implementing Kruskal's Algorithm Efficiently

```
Kruskal_ComputeMST
    Sort edges in E based on cost
    T is empty (* T will store edges of a MST *)
    each vertex u is placed in a set by itself
    while E is not empty do
        pick e = (u, v) ∈ E of minimum cost
        if u and v belong to different sets
            add e to T
            merge the sets containing u and v
    return the set T
```

Need a data structure to check if two elements belong to same set and to merge two sets.

Implementing Kruskal's Algorithm Efficiently

```
Kruskal_ComputeMST
    Sort edges in E based on cost
    T is empty (* T will store edges of a MST *)
    each vertex u is placed in a set by itself
    while E is not empty do
        pick e = (u, v) ∈ E of minimum cost
        if u and v belong to different sets
            add e to T
            merge the sets containing u and v
    return the set T
```

Need a data structure to check if two elements belong to same set and to merge two sets.

Implementing Kruskal's Algorithm Efficiently

```
Kruskal_ComputeMST
    Sort edges in E based on cost
    T is empty (* T will store edges of a MST *)
    each vertex u is placed in a set by itself
    while E is not empty do
        pick e = (u, v) ∈ E of minimum cost
        if u and v belong to different sets
            add e to T
            merge the sets containing u and v
    return the set T
```

Need a data structure to check if two elements belong to same set and to merge two sets.

MST for really sparse graphs?

Given a graph G with $\mathbf{n}$ vertices, and $\mathbf{n + 20}$ edges, its MST can be computed in

(A) $\mathbf{O(n^2)}$.

(B) $\mathbf{O(n \log n)}$.

(C) $\mathbf{O(n \log \log n)}$.

(D) $\mathbf{O(n \log^* n)}$.

(E) $\mathbf{O(n)}$.

Union-Find Data Structure

Data Structure

Store disjoint sets of elements that supports the following operations

1. **makeUnionFind(S)** returns a data structure where each element of **S** is in a separate set
2. **find(u)** returns the *name* of set containing element **u**. Thus, **u** and **v** belong to the same set if and only if **find(u) = find(v)**
3. **union(A, B)** merges two sets **A** and **B**. Here **A** and **B** are the names of the sets. Typically the name of a set is some element in the set.

Assumption: **S** is indexed by integers **1** to $|S|$.

Union-Find Data Structure

Data Structure

Store disjoint sets of elements that supports the following operations

1. **makeUnionFind(S)** returns a data structure where each element of **S** is in a separate set
2. **find(u)** returns the *name* of set containing element **u**. Thus, **u** and **v** belong to the same set if and only if **find(u) = find(v)**
3. **union(A, B)** merges two sets **A** and **B**. Here **A** and **B** are the names of the sets. Typically the name of a set is some element in the set.

Assumption: **S** is indexed by integers **1** to $|S|$.

Union-Find Data Structure

Data Structure

Store disjoint sets of elements that supports the following operations

1. **makeUnionFind(S)** returns a data structure where each element of **S** is in a separate set
2. **find(u)** returns the *name* of set containing element **u**. Thus, **u** and **v** belong to the same set if and only if **find(u) = find(v)**
3. **union(A, B)** merges two sets **A** and **B**. Here **A** and **B** are the names of the sets. Typically the name of a set is some element in the set.

Assumption: **S** is indexed by integers **1** to $|S|$.

Implementing Union-Find using Arrays and Lists

Using lists

1. Each set stored as list with a name associated with the list.
2. For each element $\mathbf{u} \in \mathbf{S}$ a pointer to the its set. Array for pointers: `component`**[u]** is pointer for **u**.
3. **makeUnionFind** (**S**) takes **O(n)** time and space.

Example

Implementing Union-Find using Arrays and Lists

1. **find**(**u**) reads the entry `component`[**u**]: **O(1)** time
2. **union**(**A**,**B**) involves updating the entries `component`[**u**] for all elements **u** in **A** and **B**: **O(|A| + |B|)** which is **O(n)**

Implementing Union-Find using Arrays and Lists

1. **find**(**u**) reads the entry `component`[**u**]: **O(1)** time
2. **union**(**A**,**B**) involves updating the entries `component`[**u**] for all elements **u** in **A** and **B**: **O(|A| + |B|)** which is **O(n)**

Implementing Union-Find using Arrays and Lists

1. **find**(**u**) reads the entry `component`[**u**]: **O(1)** time
2. **union**(**A**,**B**) involves updating the entries `component`[**u**] for all elements **u** in **A** and **B**: **O(|A| + |B|)** which is **O(n)**

Improving the List Implementation for Union

New Implementation

As before use `component`[**u**] to store set of **u**.
Change to **union**(**A**,**B**):

1. with each set, keep track of its size
2. assume $|\mathbf{A}| \leq |\mathbf{B}|$ for now
3. Merge the list of **A** into that of **B**: $\mathbf{O(1)}$ time (linked lists)
4. Update `component`**[u]** only for elements in the smaller set **A**
5. Total $\mathbf{O(|A|)}$ time. Worst case is still $\mathbf{O(n)}$.

find still takes $\mathbf{O(1)}$ time

Improving the List Implementation for Union

New Implementation

As before use `component`[**u**] to store set of **u**.
Change to **union**(**A**,**B**):

1. with each set, keep track of its size
2. assume $|\mathbf{A}| \leq |\mathbf{B}|$ for now
3. Merge the list of **A** into that of **B**: $\mathbf{O(1)}$ time (linked lists)
4. Update `component`**[u]** only for elements in the smaller set **A**
5. Total $\mathbf{O(|A|)}$ time. Worst case is still $\mathbf{O(n)}$.

find still takes $\mathbf{O(1)}$ time

Improving the List Implementation for Union

New Implementation

As before use `component`[**u**] to store set of **u**.
Change to **union**(**A**,**B**):

1. with each set, keep track of its size
2. assume $|\mathbf{A}| \leq |\mathbf{B}|$ for now
3. Merge the list of **A** into that of **B**: $\mathbf{O(1)}$ time (linked lists)
4. Update `component`**[u]** only for elements in the smaller set **A**
5. Total $\mathbf{O(|A|)}$ time. Worst case is still $\mathbf{O(n)}$.

find still takes $\mathbf{O(1)}$ time

Example

The smaller set (list) is appended to the largest set (list)

Mergers

Consider an element $\mathbf{x}$. Assume $\mathbf{x}$ is in a set $\mathbf{X}$, and let $\mathbf{Y}$ be a bigger set. After **union**($\mathbf{X}$, $\mathbf{Y}$) the size of the set containing $\mathbf{x}$ is at least:

(A) At least double what it was.

(B) Same.

(C) Maybe bigger, maybe the same size.

(D) $|\mathbf{X}| * |\mathbf{Y}|$.

(E) $|\mathbf{X}|(|\mathbf{Y}| - |\mathbf{X}|)$.

Mergers

Consider starting with $\mathbf{n}$ singletons. Consider an element $\mathbf{x}$. The element $\mathbf{x}$ can be participate in at most

(A) $\Theta(1)$.
(B) $\Theta(\log n)$.
(C) $\Theta(\sqrt{n})$.
(D) $\Theta(n)$.

mergers where it belongs to the smaller set, throughout the execution of **Union-Find**.

Improving the List Implementation for Union

Question

Is the improved implementation provably better or is it simply a nice heuristic?

Theorem

Any sequence of **k union** *operations, starting from* **makeUnionFind(S)** *on set* **S** *of size* **n**, *takes at most* $O(k \log k)$.

Corollary

Kruskal's algorithm can be implemented in $O(m \log m)$ *time.*

Sorting takes $O(m \log m)$ time, $O(m)$ finds take $O(m)$ time and $O(n)$ unions take $O(n \log n)$ time.

Improving the List Implementation for Union

Question

Is the improved implementation provably better or is it simply a nice heuristic?

Theorem

Any sequence of **k union** *operations, starting from* **makeUnionFind(S)** *on set* **S** *of size* **n**, *takes at most* $O(k \log k)$.

Corollary

Kruskal's algorithm can be implemented in $O(m \log m)$ *time.*

Sorting takes $O(m \log m)$ time, $O(m)$ finds take $O(m)$ time and $O(n)$ unions take $O(n \log n)$ time.

Improving the List Implementation for Union

Question

Is the improved implementation provably better or is it simply a nice heuristic?

Theorem

Any sequence of **k union** *operations, starting from* **makeUnionFind(S)** *on set* **S** *of size* **n**, *takes at most* $\mathbf{O(k \log k)}$.

Corollary

Kruskal's algorithm can be implemented in $\mathbf{O(m \log m)}$ *time.*

Sorting takes $\mathbf{O(m \log m)}$ time, $\mathbf{O(m)}$ finds take $\mathbf{O(m)}$ time and $\mathbf{O(n)}$ unions take $\mathbf{O(n \log n)}$ time.

Amortized Analysis

Why does theorem work?

Key Observation

union(**A**,**B**) takes $\mathbf{O(|A|)}$ time where $|\mathbf{A}| \leq |\mathbf{B}|$. Size of new set is $\geq \mathbf{2}|\mathbf{A}|$. Cannot double too many times.

Proof of Theorem

Proof.

1. Any union operation involves at most 2 of the original one-element sets; thus at least $\mathbf{n - 2k}$ elements have never been involved in a union
2. Also, maximum size of any set (after $\mathbf{k}$ unions) is $\mathbf{2k}$
3. **union**($\mathbf{A}$,$\mathbf{B}$) takes $\mathbf{O(|A|)}$ time where $|\mathbf{A}| \leq |\mathbf{B}|$.
4. *Charge* each element in $\mathbf{A}$ constant time to *pay* for $\mathbf{O(|A|)}$ time.
5. How much does any element get charged?
6. If `component`[$\mathbf{v}$] is updated, set containing $\mathbf{v}$ *doubles* in size
7. `component`[$\mathbf{v}$] is updated at most $\mathbf{\log 2k}$ times
8. Total number of updates is $\mathbf{2k \log 2k = O(k \log k)}$ □

Improving Worst Case Time

Better data structure

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root's name.

1. **find(u)**: Traverse from **u** to the root
2. **union(A, B)**: Make root of **A** (smaller set) point to root of **B**. Takes **O(1)** time.

Improving Worst Case Time

Better data structure

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root's name.

1. **find(u)**: Traverse from **u** to the root
2. **union(A, B)**: Make root of **A** (smaller set) point to root of **B**. Takes **O(1)** time.

Improving Worst Case Time

Better data structure

Maintain elements in a forest of *in-trees*; all elements in one tree belong to a set with root's name.

1. **find(u)**: Traverse from **u** to the root
2. **union(A, B)**: Make root of **A** (smaller set) point to root of **B**. Takes **O(1)** time.

Details of Implementation

Each element $\mathbf{u} \in \mathbf{S}$ has a pointer $\mathbf{parent(u)}$ to its ancestor.

```
makeUnionFind(S)
    for each u in S do
        parent(u) = u
```

```
find(u)
    while (parent(u) ≠ u) do
        u = parent(u)
    return u
```

```
union(component(u), component(v))
        (* parent(u) = u & parent(v) = v *)
    if (|component(u)| ≤ |component(v)|) then
        parent(u) = v
    else
        parent(v) = u
    set new component size to |component(u)| + |component(v)|
```

Details of Implementation

Each element $u \in S$ has a pointer $\mathrm{parent}(u)$ to its ancestor.

```
makeUnionFind(S)
    for each u in S do
        parent(u) = u
```

```
find(u)
    while (parent(u) ≠ u) do
        u = parent(u)
    return u
```

```
union(component(u), component(v))
        (* parent(u) = u & parent(v) = v *)
    if (|component(u)| ≤ |component(v)|) then
        parent(u) = v
    else
        parent(v) = u
    set new component size to |component(u)| + |component(v)|
```

Details of Implementation

Each element $u \in S$ has a pointer $\mathrm{parent}(u)$ to its ancestor.

```
makeUnionFind(S)
    for each u in S do
        parent(u) = u
```

```
find(u)
    while (parent(u) ≠ u) do
        u = parent(u)
    return u
```

```
union(component(u), component(v))
        (* parent(u) = u & parent(v) = v *)
    if (|component(u)| ≤ |component(v)|) then
        parent(u) = v
    else
        parent(v) = u
    set new component size to |component(u)| + |component(v)|
```

Details of Implementation

Each element $u \in S$ has a pointer $\mathrm{parent}(u)$ to its ancestor.

```
makeUnionFind(S)
    for each u in S do
        parent(u) = u
```

```
find(u)
    while (parent(u) ≠ u) do
        u = parent(u)
    return u
```

```
union(component(u), component(v))
        (* parent(u) = u & parent(v) = v *)
    if (|component(u)| ≤ |component(v)|) then
        parent(u) = v
    else
        parent(v) = u
    set new component size to |component(u)| + |component(v)|
```

Analysis

Theorem

The forest based implementation for a set of size **n**, *has the following complexity for the various operations:* **makeUnionFind** *takes* **O(n)**, **union** *takes* **O(1)**, *and* **find** *takes* **O(log n)**.

Proof.

1. **find(u)** depends on the height of tree containing **u**.
2. Height of **u** increases by at most **1** only when the set containing **u** changes its name.
3. If height of **u** increases then size of the set containing **u** (at least) doubles.
4. Maximum set size is **n**; so height of any tree is at most **O(log n)**. □

Further Improvements: Path Compression

Observation

Consecutive calls of **find(u)** *take* $\mathbf{O(\log n)}$ *time each, but they traverse the same sequence of pointers.*

Idea: Path Compression

Make all nodes encountered in the **find(u)** point to root.

Further Improvements: Path Compression

Observation

Consecutive calls of **find(u)** *take* $\mathbf{O(\log n)}$ *time each, but they traverse the same sequence of pointers.*

Idea: Path Compression

Make all nodes encountered in the **find(u)** point to root.

Path Compression: Example

Path Compression

```
find(u):
    if (parent(u) ≠ u) then
        parent(u) = find(parent(u))
    return parent(u)
```

Question

Does Path Compression help?

Yes!

Theorem

With Path Compression, $\mathbf{k}$ *operations (***find** *and/or* **union***) take* $O(k\alpha(k, \min\{k, n\}))$ *time where* α *is the* **inverse Ackermann function**.

Path Compression

```
find(u):
    if (parent(u) ≠ u) then
        parent(u) = find(parent(u))
    return parent(u)
```

Question

Does Path Compression help?

Yes!

Theorem

With Path Compression, $\mathbf{k}$ *operations (***find** *and/or* **union***) take* $O(k\alpha(k, \min\{k, n\}))$ *time where* α *is the* **inverse Ackermann function**.

Path Compression

```
find(u):
    if (parent(u) ≠ u) then
        parent(u) = find(parent(u))
    return parent(u)
```

Question

Does Path Compression help?

Yes!

Theorem

With Path Compression, **k** *operations (***find** *and/or* **union***) take* $O(k\alpha(k, \min\{k, n\}))$ *time where* α *is the* **inverse Ackermann function**.

Ackermann and Inverse Ackermann Functions

Ackermann function $\mathbf{A(m,n)}$ defined for $\mathbf{m,n \geq 0}$ recursively

$$\mathbf{A(m,n)} = \begin{cases} \mathbf{n+1} & \text{if } \mathbf{m=0} \\ \mathbf{A(m-1,1)} & \text{if } \mathbf{m>0} \text{ and } \mathbf{n=0} \\ \mathbf{A(m-1,A(m,n-1))} & \text{if } \mathbf{m>0} \text{ and } \mathbf{n>0} \end{cases}$$

$\mathbf{A(3,n)} = 2^{\mathbf{n+3}} - 3$
$\mathbf{A(4,3)} = 2^{65536} - 3$

$\alpha\mathbf{(m,n)}$ is inverse Ackermann function defined as

$$\alpha(\mathbf{m,n}) = \min\{\mathbf{i} \mid \mathbf{A(i,\lfloor m/n \rfloor)} \geq \log_2 \mathbf{n}\}$$

For **all practical** purposes $\alpha\mathbf{(m,n)} \leq \mathbf{5}$

Ackermann and Inverse Ackermann Functions

Ackermann function $\mathbf{A(m,n)}$ defined for $\mathbf{m,n \geq 0}$ recursively

$$\mathbf{A(m,n)} = \begin{cases} \mathbf{n+1} & \text{if } \mathbf{m=0} \\ \mathbf{A(m-1,1)} & \text{if } \mathbf{m>0} \text{ and } \mathbf{n=0} \\ \mathbf{A(m-1,A(m,n-1))} & \text{if } \mathbf{m>0} \text{ and } \mathbf{n>0} \end{cases}$$

$\mathbf{A(3,n) = 2^{n+3} - 3}$
$\mathbf{A(4,3) = 2^{65536} - 3}$

$\alpha\mathbf{(m,n)}$ is inverse Ackermann function defined as

$$\alpha\mathbf{(m,n) = \min\{i \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n\}}$$

For **all practical** purposes $\alpha\mathbf{(m,n) \leq 5}$

Ackermann and Inverse Ackermann Functions

Ackermann function $\mathbf{A(m,n)}$ defined for $\mathbf{m,n \geq 0}$ recursively

$$\mathbf{A(m,n)} = \begin{cases} \mathbf{n+1} & \text{if } \mathbf{m=0} \\ \mathbf{A(m-1,1)} & \text{if } \mathbf{m>0} \text{ and } \mathbf{n=0} \\ \mathbf{A(m-1,A(m,n-1))} & \text{if } \mathbf{m>0} \text{ and } \mathbf{n>0} \end{cases}$$

$\mathbf{A(3,n) = 2^{n+3} - 3}$
$\mathbf{A(4,3) = 2^{65536} - 3}$

$\alpha\mathbf{(m,n)}$ is inverse Ackermann function defined as

$$\alpha\mathbf{(m,n) = \min\{i \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n\}}$$

For **all practical** purposes $\alpha\mathbf{(m,n) \leq 5}$

Lower Bound for Union-Find Data Structure

Amazing result:

Theorem (Tarjan)

For **Union-Find**, *any data structure in the pointer model requires* $\mathbf{\Omega(m\alpha(m,n))}$ *time for* $\mathbf{m}$ *operations.*

Running time of Kruskal's Algorithm

Using Union-Find data structure:

1. **O(m) find** operations (two for each edge)
2. **O(n) union** operations (one for each edge added to **T**)
3. Total time: $\mathbf{O(m \log m)}$ for sorting plus $\mathbf{O(m\alpha(n))}$ for union-find operations. Thus $\mathbf{O(m \log m)}$ time despite the improved Union-Find data structure.

Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps: $\mathbf{O(n \log n + m)}$.
If $\mathbf{m}$ is $\mathbf{O(n)}$ then running time is $\mathbf{\Omega(n \log n)}$.

Question

Is there a linear time ($\mathbf{O(m+n)}$ time) algorithm for MST?

1. $\mathbf{O(m \log^* m)}$ time **Fredman and Tarjan [1987]**.
2. $\mathbf{O(m+n)}$ time using bit operations in RAM model **Fredman and Willard [1994]**.
3. $\mathbf{O(m+n)}$ expected time (randomized algorithm) **Karger et al. [1995]**.
4. $\mathbf{O((n+m)\alpha(m,n))}$ time **Chazelle [2000]**.
5. Still open: Is there an $\mathbf{O(n+m)}$ time deterministic algorithm in the comparison model?

Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps: $\mathbf{O(n \log n + m)}$.
If $\mathbf{m}$ is $\mathbf{O(n)}$ then running time is $\mathbf{\Omega(n \log n)}$.

Question

Is there a linear time ($\mathbf{O(m + n)}$ time) algorithm for MST?

1. $\mathbf{O(m \log^* m)}$ time **Fredman and Tarjan [1987]**.
2. $\mathbf{O(m + n)}$ time using bit operations in RAM model **Fredman and Willard [1994]**.
3. $\mathbf{O(m + n)}$ expected time (randomized algorithm) **Karger et al. [1995]**.
4. $\mathbf{O((n + m)\alpha(m, n))}$ time **Chazelle [2000]**.
5. Still open: Is there an $\mathbf{O(n + m)}$ time deterministic algorithm in the comparison model?

Best Known Asymptotic Running Times for MST

Prim's algorithm using Fibonacci heaps: $\mathbf{O(n \log n + m)}$.
If $\mathbf{m}$ is $\mathbf{O(n)}$ then running time is $\mathbf{\Omega(n \log n)}$.

Question

Is there a linear time ($\mathbf{O(m + n)}$ time) algorithm for MST?

1. $\mathbf{O(m \log^* m)}$ time **Fredman and Tarjan [1987]**.
2. $\mathbf{O(m + n)}$ time using bit operations in RAM model **Fredman and Willard [1994]**.
3. $\mathbf{O(m + n)}$ expected time (randomized algorithm) **Karger et al. [1995]**.
4. $\mathbf{O((n + m)\alpha(m, n))}$ time **Chazelle [2000]**.
5. Still open: Is there an $\mathbf{O(n + m)}$ time deterministic algorithm in the comparison model?

Notes

Notes

Notes

Notes

Chazelle, B. (2000). A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. Assoc. Comput. Mach.*, 47(6):1028–1047.

Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.*, 34(3):596–615.

Fredman, M. L. and Willard, D. E. (1994). Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Sys. Sci.*, 48(3):533–551.

Karger, D. R., Klein, P. N., and Tarjan, R. E. (1995). A randomized linear-time algorithm to find minimum spanning trees. *J. Assoc. Comput. Mach.*, 42(2):321–328.

Introduction to Randomized Algorithms: QuickSort and QuickSelect

Lecture 13

March 6, 2014

Part I

Introduction to Randomized Algorithms

Randomized Algorithms

Randomized Algorithms

Example: Randomized **QuickSort**

QuickSort Hoare [1962]

1. Pick a pivot element from array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Randomized **QuickSort**

1. Pick a pivot element *uniformly at random* from the array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Example: Randomized Quicksort

Recall: **QuickSort** can take $\Omega(n^2)$ time to sort array of size n.

Theorem

Randomized **QuickSort** *sorts a given array of length* n *in* $O(n \log n)$ *expected time.*

Note: On *every* input randomized **QuickSort** takes $O(n \log n)$ time in expectation. On *every* input it may take $\Omega(n^2)$ time with some small probability.

Example: Randomized Quicksort

Recall: **QuickSort** can take $\Omega(n^2)$ time to sort array of size n.

Theorem

Randomized **QuickSort** *sorts a given array of length* n *in* $O(n \log n)$ *expected time.*

Note: On *every* input randomized **QuickSort** takes $O(n \log n)$ time in expectation. On *every* input it may take $\Omega(n^2)$ time with some small probability.

Example: Randomized Quicksort

Recall: **QuickSort** can take $\Omega(n^2)$ time to sort array of size n.

Theorem

Randomized **QuickSort** *sorts a given array of length* n *in* $O(n \log n)$ *expected time.*

Note: On *every* input randomized **QuickSort** takes $O(n \log n)$ time in expectation. On *every* input it may take $\Omega(n^2)$ time with some small probability.

Example: Verifying Matrix Multiplication

Problem

Given three $\mathbf{n} \times \mathbf{n}$ matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ is $\mathbf{AB} = \mathbf{C}$?

Deterministic algorithm:

1. Multiply $\mathbf{A}$ and $\mathbf{B}$ and check if equal to $\mathbf{C}$.
2. Running time? $O(n^3)$ by straight forward approach. $O(n^{2.37})$ with fast matrix multiplication (complicated and impractical).

Example: Verifying Matrix Multiplication

Problem

Given three $\mathbf{n} \times \mathbf{n}$ matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ is $\mathbf{AB} = \mathbf{C}$?

Deterministic algorithm:

1. Multiply $\mathbf{A}$ and $\mathbf{B}$ and check if equal to $\mathbf{C}$.
2. Running time? $O(n^3)$ by straight forward approach. $O(n^{2.37})$ with fast matrix multiplication (complicated and impractical).

Example: Verifying Matrix Multiplication

Problem

Given three $\mathbf{n} \times \mathbf{n}$ matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ is $\mathbf{AB} = \mathbf{C}$?

Deterministic algorithm:

1. Multiply $\mathbf{A}$ and $\mathbf{B}$ and check if equal to $\mathbf{C}$.
2. Running time? $\mathbf{O(n^3)}$ by straight forward approach. $\mathbf{O(n^{2.37})}$ with fast matrix multiplication (complicated and impractical).

Example: Verifying Matrix Multiplication

Problem

Given three $\mathbf{n} \times \mathbf{n}$ matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ is $\mathbf{AB} = \mathbf{C}$?

Randomized algorithm:

1. Pick a random $\mathbf{n} \times \mathbf{1}$ vector $\mathbf{r}$.
2. Return the answer of the equality $\mathbf{ABr} = \mathbf{Cr}$.
3. Running time? $\mathbf{O(n^2)}$!

Theorem

If $\mathbf{AB} = \mathbf{C}$ then the algorithm will always say YES. If $\mathbf{AB} \neq \mathbf{C}$ then the algorithm will say YES with probability at most $\mathbf{1/2}$. Can repeat the algorithm $\mathbf{100}$ times independently to reduce the probability of a false positive to $\mathbf{1/2^{100}}$.

Example: Verifying Matrix Multiplication

Problem

Given three $\mathbf{n} \times \mathbf{n}$ matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ is $\mathbf{AB} = \mathbf{C}$?

Randomized algorithm:

1. Pick a random $\mathbf{n} \times \mathbf{1}$ vector $\mathbf{r}$.
2. Return the answer of the equality $\mathbf{ABr} = \mathbf{Cr}$.
3. Running time? $O(n^2)$!

Theorem

If $\mathbf{AB} = \mathbf{C}$ then the algorithm will always say YES. If $\mathbf{AB} \neq \mathbf{C}$ then the algorithm will say YES with probability at most $1/2$. Can repeat the algorithm 100 times independently to reduce the probability of a false positive to $1/2^{100}$.

Example: Verifying Matrix Multiplication

Problem

Given three $\mathbf{n} \times \mathbf{n}$ matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ is $\mathbf{AB} = \mathbf{C}$?

Randomized algorithm:

1. Pick a random $\mathbf{n} \times \mathbf{1}$ vector $\mathbf{r}$.
2. Return the answer of the equality $\mathbf{ABr} = \mathbf{Cr}$.
3. Running time? $\mathbf{O(n^2)}$!

Theorem

If $\mathbf{AB} = \mathbf{C}$ then the algorithm will always say YES. If $\mathbf{AB} \neq \mathbf{C}$ then the algorithm will say YES with probability at most $\mathbf{1/2}$. Can repeat the algorithm $\mathbf{100}$ times independently to reduce the probability of a false positive to $\mathbf{1/2^{100}}$.

Example: Verifying Matrix Multiplication

Problem

Given three $\mathbf{n} \times \mathbf{n}$ matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ is $\mathbf{AB} = \mathbf{C}$?

Randomized algorithm:

1. Pick a random $\mathbf{n} \times \mathbf{1}$ vector $\mathbf{r}$.
2. Return the answer of the equality $\mathbf{ABr} = \mathbf{Cr}$.
3. Running time? $\mathbf{O(n^2)}$!

Theorem

If $\mathbf{AB} = \mathbf{C}$ then the algorithm will always say YES. If $\mathbf{AB} \neq \mathbf{C}$ then the algorithm will say YES with probability at most $\mathbf{1/2}$. Can repeat the algorithm $\mathbf{100}$ times independently to reduce the probability of a false positive to $\mathbf{1/2^{100}}$.

Why randomized algorithms?

1. Many many applications in algorithms, data structures and computer science!
2. In some cases only known algorithms are randomized or randomness is provably necessary.
3. Often randomized algorithms are (much) simpler and/or more efficient.
4. Several deep connections to mathematics, physics etc.
5. ...
6. Lots of fun!

Where do I get random bits?

Question: Are true random bits available in practice?

1. Buy them!
2. CPUs use physical phenomena to generate random bits.
3. Can use pseudo-random bits or semi-random bits from nature. Several fundamental unresolved questions in complexity theory on this topic. Beyond the scope of this course.
4. In practice pseudo-random generators work quite well in many applications.
5. The model is interesting to think in the abstract and is very useful even as a theoretical construct. One can *derandomize* randomized algorithms to obtain deterministic algorithms.

Average case analysis vs Randomized algorithms

Average case analysis:

1. Fix a deterministic algorithm.
2. Assume inputs comes from a probability distribution.
3. Analyze the algorithm's *average* performance over the distribution over inputs.

Randomized algorithms:

1. Algorithm uses random bits in addition to input.
2. Analyze algorithms *average* performance over the given input where the average is over the random bits that the algorithm uses.
3. On each input behaviour of algorithm is random. Analyze worst-case over all inputs of the (average) performance.

Discrete Probability

We restrict attention to finite probability spaces.

Definition

A discrete probability space is a pair $(\Omega, \Pr)$ consists of finite set Ω of **elementary events** and function $p : \Omega \to [0, 1]$ which assigns a probability $\Pr[\omega]$ for each $\omega \in \Omega$ such that $\sum_{\omega \in \Omega} \Pr[\omega] = 1$.

Example

An unbiased coin. $\Omega = \{H, T\}$ and $\Pr[H] = \Pr[T] = 1/2$.

Example

A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[i] = 1/6$ for $1 \leq i \leq 6$.

Discrete Probability

We restrict attention to finite probability spaces.

Definition

A discrete probability space is a pair $(\Omega, \Pr)$ consists of finite set Ω of **elementary events** and function $p : \Omega \to [0, 1]$ which assigns a probability $\Pr[\omega]$ for each $\omega \in \Omega$ such that $\sum_{\omega \in \Omega} \Pr[\omega] = 1$.

Example

An unbiased coin. $\Omega = \{H, T\}$ and $\Pr[H] = \Pr[T] = 1/2$.

Example

A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[i] = 1/6$ for $1 \leq i \leq 6$.

Discrete Probability

And more examples

Example

A biased coin. $\Omega = \{H, T\}$ and $Pr[H] = 2/3, Pr[T] = 1/3$.

Example

Two independent unbiased coins. $\Omega = \{HH, TT, HT, TH\}$ and $Pr[HH] = Pr[TT] = Pr[HT] = Pr[TH] = 1/4$.

Example

A pair of (highly) correlated dice.
$\Omega = \{(i, j) \mid 1 \leq i \leq 6, 1 \leq j \leq 6\}$.
$Pr[i, i] = 1/6$ for $1 \leq i \leq 6$ and $Pr[i, j] = 0$ if $i \neq j$.

Events

Definition

Given a probability space $(\Omega, \Pr)$ an **event** is a subset of Ω. In other words an event is a collection of elementary events. The probability of an event A, denoted by $\Pr[A]$, is $\sum_{\omega \in A} \Pr[\omega]$.

The **complement event** of an event $A \subseteq \Omega$ is the event $\Omega \setminus A$ frequently denoted by $\bar{A}$.

Events

Examples

Example

A pair of independent dice. $\Omega = \{(i,j) \mid 1 \leq i \leq 6, 1 \leq j \leq 6\}$.

1. Let A be the event that the sum of the two numbers on the dice is even.
   Then $A = \left\{(i,j) \in \Omega \;\middle|\; (i+j) \text{ is even}\right\}$.
   $\Pr[A] = |A|/36 = 1/2$.

2. Let B be the event that the first die has 1. Then
   $B = \left\{(1,1),(1,2),(1,3),(1,4),(1,5),(1,6)\right\}$.
   $\Pr[B] = 6/36 = 1/6$.

Independent Events

Definition

Given a probability space $(\Omega, \Pr)$ and two events A, B are **independent** if and only if $\Pr[A \cap B] = \Pr[A]\Pr[B]$. Otherwise they are *dependent*. In other words A, B independent implies one does not affect the other.

Example

Two coins. $\Omega = \{HH, TT, HT, TH\}$ and $\Pr[HH] = \Pr[TT] = \Pr[HT] = \Pr[TH] = 1/4$.

1. A is the event that the first coin is heads and B is the event that second coin is tails. A, B are independent.
2. A is the event that the two coins are different. B is the event that the second coin is heads. A, B independent.

Independent Events

Definition

Given a probability space $(\Omega, \Pr)$ and two events A, B are **independent** if and only if $\Pr[A \cap B] = \Pr[A]\Pr[B]$. Otherwise they are *dependent*. In other words A, B independent implies one does not affect the other.

Example

Two coins. $\Omega = \{HH, TT, HT, TH\}$ and $\Pr[HH] = \Pr[TT] = \Pr[HT] = \Pr[TH] = 1/4$.

1. A is the event that the first coin is heads and B is the event that second coin is tails. A, B are independent.
2. A is the event that the two coins are different. B is the event that the second coin is heads. A, B independent.

Independent Events

Examples

Example

A is the event that both are not tails and **B** is event that second coin is heads. **A, B** are dependent.

Dependent or independent?

Consider two independent rolls of the dice.

1. **A** = the event that the first roll is odd.
2. **B** = the event that the sum of the two rolls is odd.

The events **A** and **B** are

(A) dependent.

(B) independent.

Union bound

The probability of the union of two events, is no bigger than the probability of the sum of their probabilities.

Lemma

For any two events $\mathcal{E}$ and $\mathcal{F}$, we have that
$\mathbf{Pr}\left[\mathcal{E} \cup \mathcal{F}\right] \leq \mathbf{Pr}\left[\mathcal{E}\right] + \mathbf{Pr}\left[\mathcal{F}\right]$.

Proof.

Consider $\mathcal{E}$ and $\mathcal{F}$ to be a collection of elmentery events (which they are). We have

$$\begin{aligned}\mathbf{Pr}\left[\mathcal{E} \cup \mathcal{F}\right] &= \sum_{\mathbf{x} \in \mathcal{E} \cup \mathcal{F}} \mathbf{Pr}[\mathbf{x}] \\ &\leq \sum_{\mathbf{x} \in \mathcal{E}} \mathbf{Pr}[\mathbf{x}] + \sum_{\mathbf{x} \in \mathcal{F}} \mathbf{Pr}[\mathbf{x}] = \mathbf{Pr}\left[\mathcal{E}\right] + \mathbf{Pr}\left[\mathcal{F}\right].\end{aligned}$$

Random Variables

Definition

Given a probability space $(\Omega, \Pr)$ a (real-valued) random variable X over Ω is a function that maps each elementary event to a real number. In other words $X : \Omega \to \mathbb{R}$.

Example

A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[i] = 1/6$ for $1 \leq i \leq 6$.

1. $X : \Omega \to \mathbb{R}$ where $X(i) = i \mod 2$.
2. $Y : \Omega \to \mathbb{R}$ where $Y(i) = i^2$.

Definition

A **binary random variable** is one that takes on values in $\{0, 1\}$.

Random Variables

Definition

Given a probability space $(\Omega, \Pr)$ a (real-valued) random variable X over Ω is a function that maps each elementary event to a real number. In other words $X : \Omega \to \mathbb{R}$.

Example

A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[i] = 1/6$ for $1 \leq i \leq 6$.

1. $X : \Omega \to \mathbb{R}$ where $X(i) = i \mod 2$.
2. $Y : \Omega \to \mathbb{R}$ where $Y(i) = i^2$.

Definition

A **binary random variable** is one that takes on values in $\{0, 1\}$.

Random Variables

Definition

Given a probability space $(\Omega, \Pr)$ a (real-valued) random variable X over Ω is a function that maps each elementary event to a real number. In other words $X : \Omega \to \mathbb{R}$.

Example

A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[i] = 1/6$ for $1 \leq i \leq 6$.

1. $X : \Omega \to \mathbb{R}$ where $X(i) = i \mod 2$.
2. $Y : \Omega \to \mathbb{R}$ where $Y(i) = i^2$.

Definition

A **binary random variable** is one that takes on values in $\{0, 1\}$.

Indicator Random Variables

Special type of random variables that are quite useful.

Definition

Given a probability space $(\Omega, \Pr)$ and an event $A \subsetneq \Omega$ the indicator random variable X_A is a binary random variable where $X_A(\omega) = 1$ if $\omega \in A$ and $X_A(\omega) = 0$ if $\omega \notin A$.

Example

A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[i] = 1/6$ for $1 \leq i \leq 6$. Let A be the even that i is divisible by 3. Then $X_A(i) = 1$ if $i = 3, 6$ and 0 otherwise.

Indicator Random Variables

Special type of random variables that are quite useful.

Definition

Given a probability space $(\Omega, \Pr)$ and an event $A \subsetneq \Omega$ the indicator random variable X_A is a binary random variable where $X_A(\omega) = 1$ if $\omega \in A$ and $X_A(\omega) = 0$ if $\omega \notin A$.

Example

A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[i] = 1/6$ for $1 \leq i \leq 6$. Let A be the even that i is divisible by 3. Then $X_A(i) = 1$ if $i = 3, 6$ and 0 otherwise.

Expectation

Definition

For a random variable $\mathbf{X}$ over a probability space $(\Omega, \Pr)$ the **expectation** of $\mathbf{X}$ is defined as $\sum_{\omega \in \Omega} \Pr[\omega]\, \mathbf{X}(\omega)$. In other words, the expectation is the average value of $\mathbf{X}$ according to the probabilities given by $\Pr[\cdot]$.

Example

A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[i] = 1/6$ for $1 \leq i \leq 6$.

1. $X : \Omega \to \mathbb{R}$ where $X(i) = i \mod 2$. Then $E[X] = 1/2$.
2. $Y : \Omega \to \mathbb{R}$ where $Y(i) = i^2$. Then $E[Y] = \sum_{i=1}^{6} \frac{1}{6} \cdot i^2 = 91/6$.

Expectation

Definition

For a random variable $\mathbf{X}$ over a probability space $(\Omega, \Pr)$ the **expectation** of $\mathbf{X}$ is defined as $\sum_{\omega \in \Omega} \Pr[\omega]\, \mathbf{X}(\omega)$. In other words, the expectation is the average value of $\mathbf{X}$ according to the probabilities given by $\Pr[\cdot]$.

Example

A 6-sided unbiased die. $\Omega = \{1, 2, 3, 4, 5, 6\}$ and $\Pr[i] = 1/6$ for $1 \leq i \leq 6$.

1. $\mathbf{X} : \Omega \to \mathbb{R}$ where $\mathbf{X}(i) = i \mod 2$. Then $\mathbf{E}[\mathbf{X}] = 1/2$.
2. $\mathbf{Y} : \Omega \to \mathbb{R}$ where $\mathbf{Y}(i) = i^2$. Then $\mathbf{E}[\mathbf{Y}] = \sum_{i=1}^{6} \frac{1}{6} \cdot i^2 = 91/6$.

Expected number of vertices?

Let $G = (V, E)$ be a graph with $\mathbf{n}$ vertices and $\mathbf{m}$ edges. Let H be the graph resulting from independently deleting every vertex of G with probability $\mathbf{1/2}$. The expected number of vertices in H is

(A) $n/2$.

(B) $n/4$.

(C) $m/2$.

(D) $m/4$.

(E) none of the above.

Expected number of edges?

Let $G = (V, E)$ be a graph with $\mathbf{n}$ vertices and $\mathbf{m}$ edges. Let H be the graph resulting from independently deleting every vertex of G with probability $\mathbf{1/2}$. The expected number of edges in H is

(A) $n/2$.

(B) $n/4$.

(C) $m/2$.

(D) $m/4$.

(E) none of the above.

Expectation

Proposition

For an indicator variable X_A, $E[X_A] = Pr[A]$.

Proof.

$$
\begin{aligned}
E[X_A] &= \sum_{y \in \Omega} X_A(y)\, Pr[y] \\
&= \sum_{y \in A} 1 \cdot Pr[y] + \sum_{y \in \Omega \setminus A} 0 \cdot Pr[y] \\
&= \sum_{y \in A} Pr[y] \\
&= Pr[A]\,.
\end{aligned}
$$

$\square$

Linearity of Expectation

Lemma

Let $\mathbf{X}, \mathbf{Y}$ *be two random variables (not necessarily independent) over a probability space* $(\Omega, \Pr)$. *Then* $\mathbf{E}[\mathbf{X} + \mathbf{Y}] = \mathbf{E}[\mathbf{X}] + \mathbf{E}[\mathbf{Y}]$.

Proof.

$$\begin{aligned}\mathbf{E}[\mathbf{X} + \mathbf{Y}] &= \sum_{\omega \in \Omega} \Pr[\omega] \left(\mathbf{X}(\omega) + \mathbf{Y}(\omega)\right) \\ &= \sum_{\omega \in \Omega} \Pr[\omega]\, \mathbf{X}(\omega) + \sum_{\omega \in \Omega} \Pr[\omega]\, \mathbf{Y}(\omega) = \mathbf{E}[\mathbf{X}] + \mathbf{E}[\mathbf{Y}]\,.\end{aligned}$$

□

Corollary

$\mathbf{E}[a_1 X_1 + a_2 X_2 + \ldots + a_n X_n] = \sum_{i=1}^{n} a_i\, \mathbf{E}[X_i]$.

Linearity of Expectation

Lemma

Let $\mathbf{X}, \mathbf{Y}$ *be two random variables (not necessarily independent) over a probability space* $(\Omega, \Pr)$. *Then* $\mathbf{E}[\mathbf{X} + \mathbf{Y}] = \mathbf{E}[\mathbf{X}] + \mathbf{E}[\mathbf{Y}]$.

Proof.

$$\begin{aligned}\mathbf{E}[\mathbf{X} + \mathbf{Y}] &= \sum_{\omega \in \Omega} \Pr[\omega] \left(\mathbf{X}(\omega) + \mathbf{Y}(\omega)\right) \\ &= \sum_{\omega \in \Omega} \Pr[\omega]\, \mathbf{X}(\omega) + \sum_{\omega \in \Omega} \Pr[\omega]\, \mathbf{Y}(\omega) = \mathbf{E}[\mathbf{X}] + \mathbf{E}[\mathbf{Y}]\,.\end{aligned}$$

□

Corollary

$\mathbf{E}[a_1 X_1 + a_2 X_2 + \ldots + a_n X_n] = \sum_{i=1}^{n} a_i\, \mathbf{E}[X_i]$.

Expected number of edges?

Let $G = (V, E)$ be a graph with $\mathbf{n}$ vertices and $\mathbf{m}$ edges. Let H be the graph resulting from independently deleting every vertex of G with probability $\mathbf{1/2}$. The expected number of edges in H is

(A) $n/2$.

(B) $n/4$.

(C) $m/2$.

(D) $m/4$.

(E) none of the above.

Expected number of triangles?

Let $G = (V, E)$ be a graph with $\mathbf{n}$ vertices and $\mathbf{m}$ edges. Assume G has $\mathbf{t}$ triangles (i.e., a triangle is a simple cycle with three vertices). Let H be the graph resulting from deleting independently each vertex of G with probability $\mathbf{1/2}$. The expected number of triangles in H is

(A) t/2.

(B) t/4.

(C) t/8.

(D) t/16.

(E) none of the above.

Types of Randomized Algorithms

Typically one encounters the following types:

1. **Las Vegas randomized algorithms:** for a given input $\mathbf{x}$ output of algorithm is always correct but the running time is a random variable. In this case we are interested in analyzing the *expected* running time.
2. **Monte Carlo randomized algorithms:** for a given input $\mathbf{x}$ the running time is deterministic but the output is random; correct with some probability. In this case we are interested in analyzing the *probability* of the correct output (and also the running time).
3. Algorithms whose running time and output may both be random.

Analyzing Las Vegas Algorithms

Deterministic algorithm $\mathbf{Q}$ for a problem $\mathbf{\Pi}$:

1. Let $\mathbf{Q(x)}$ be the time for $\mathbf{Q}$ to run on input $\mathbf{x}$ of length $|\mathbf{x}|$.
2. Worst-case analysis: run time on worst input for a given size $\mathbf{n}$.

$$\mathbf{T_{wc}(n) = \max_{x:|x|=n} Q(x).}$$

Randomized algorithm $\mathbf{R}$ for a problem $\mathbf{\Pi}$:

1. Let $\mathbf{R(x)}$ be the time for $\mathbf{Q}$ to run on input $\mathbf{x}$ of length $|\mathbf{x}|$.
2. $\mathbf{R(x)}$ is a random variable: depends on random bits used by $\mathbf{R}$.
3. $\mathbf{E[R(x)]}$ is the expected running time for $\mathbf{R}$ on $\mathbf{x}$
4. Worst-case analysis: expected time on worst input of size $\mathbf{n}$

$$\mathbf{T_{rand-wc}(n) = \max_{x:|x|=n} E[Q(x)]}\,.$$

Analyzing Las Vegas Algorithms

Deterministic algorithm $\mathbf{Q}$ for a problem $\mathbf{\Pi}$:

1. Let $\mathbf{Q(x)}$ be the time for $\mathbf{Q}$ to run on input $\mathbf{x}$ of length $|\mathbf{x}|$.
2. Worst-case analysis: run time on worst input for a given size $\mathbf{n}$.

$$T_{wc}(n) = \max_{x:|x|=n} Q(x).$$

Randomized algorithm $\mathbf{R}$ for a problem $\mathbf{\Pi}$:

1. Let $\mathbf{R(x)}$ be the time for $\mathbf{Q}$ to run on input $\mathbf{x}$ of length $|\mathbf{x}|$.
2. $\mathbf{R(x)}$ is a random variable: depends on random bits used by $\mathbf{R}$.
3. $\mathbf{E[R(x)]}$ is the expected running time for $\mathbf{R}$ on $\mathbf{x}$
4. Worst-case analysis: expected time on worst input of size $\mathbf{n}$

$$T_{rand-wc}(n) = \max_{x:|x|=n} E[Q(x)]\,.$$

Analyzing Monte Carlo Algorithms

Randomized algorithm $\mathbf{M}$ for a problem $\mathbf{\Pi}$:

1. Let $\mathbf{M(x)}$ be the time for $\mathbf{M}$ to run on input $\mathbf{x}$ of length $|\mathbf{x}|$. For Monte Carlo, assumption is that run time is deterministic.
2. Let $\mathbf{Pr[x]}$ be the probability that $\mathbf{M}$ is correct on $\mathbf{x}$.
3. $\mathbf{Pr[x]}$ is a random variable: depends on random bits used by $\mathbf{M}$.
4. Worst-case analysis: success probability on worst input

$$\mathbf{P_{rand-wc}(n) = \min_{x:|x|=n} Pr[x]\,.}$$

Part II

Why does randomization help?

Ping and find.

Consider a deterministic algorithm **A** that is trying to find an element in an array **X** of size **n**. At every step it is allowed to ask the value of one cell in the array, and the adversary is allowed after each such ping, to shuffle elements around in the array in any way it seems fit. For the best possible deterministic algorithm the number of rounds it has to play this game till it finds the required element is

(A) $O(1)$
(B) $O(n)$
(C) $O(n \log n)$
(D) $O(n^2)$
(E) ∞.

Ping and find randomized.

Consider an algorithm **randFind** that is trying to find an element in an array **X** of size **n**. At every step it asks the value of one random cell in the array, and the adversary is allowed after each such ping, to shuffle elements around in the array in any way it seems fit. This algorithm would stop in expectation after

(A) $O(1)$
(B) $O(\log n)$
(C) $O(n)$
(D) $O(n^2)$
(E) ∞.

steps.

Abundance of witnesses

Consider the problem of finding an "approximate median" of an unsorted array $\mathbf{A[1..n]}$: an element of $\mathbf{A}$ with rank between $\mathbf{n/4}$ and $\mathbf{3n/4}$.

- We saw a complicated deterministic algorithm. Finding an approximate median is not any easier than a proper median.
- $\mathbf{n/2}$ elements of $\mathbf{A}$ qualify as approximate medians and hence a random element is good with probability $\mathbf{1/2}$!

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

Consider flipping a fair coin **n** times independently, head given **1**, tail gives zero. How many heads? ...we get a binomial distribution.

Massive randomness.. Is not that random.

This is known as **concentration of mass**.

This is a very special case of the **law of large numbers**.

Side note...

Law of large numbers (weakest form)...

Informal statement of law of large numbers

For **n** large enough, the middle portion of the binomial distribution looks like (converges to) the normal/Gaussian distribution.

Massive randomness.. Is not that random.

Intuitive conclusion

Randomized algorithm are unpredictable in the tactical level, but very predictable in the strategic level.

Binomial distribution

$\mathbf{X_n}$ = numbers of heads when flipping a coin $\mathbf{n}$ times.

Claim

$$\Pr\left[X_n = i\right] = \frac{\binom{n}{i}}{2^n}.$$

Where: $\binom{n}{k} = \frac{n!}{(n-k)!k!}$.
Indeed, $\binom{n}{i}$ is the number of ways to choose $\mathbf{i}$ elements out of $\mathbf{n}$ elements (i.e., pick which $\mathbf{i}$ coin flip come up heads).
Each specific such possibility (say **0100010...**) had probability $1/2^n$.
We are interested in the bad event $\Pr[X_n \leq n/4]$ (way too few heads). We are going to prove this probability is tiny.

Binomial distribution

Playing around with binomial coefficients

Lemma

$n! \geq (n/e)^n$.

Proof.

$$\frac{n^n}{n!} \leq \sum_{i=0}^{\infty} \frac{n^i}{i!} = e^n,$$

by the Taylor expansion of $e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$. This implies that $(n/e)^n \leq n!$, as required. $\square$

Binomial distribution

Playing around with binomial coefficients

Lemma

For any $k \leq n$, *we have* $\binom{n}{k} \leq \left(\frac{ne}{k}\right)^k$.

Proof.

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} = \frac{n(n-1)(n-2)\ldots(n-k+1)}{k!}$$
$$\leq \frac{n^k}{k!} \leq \frac{n^k}{\left(\frac{k}{e}\right)^k} = \left(\frac{ne}{k}\right)^k.$$

since $k! \geq (k/e)^k$ (by previous lemma). □

Binomial distribution

Playing around with binomial coefficients

$$\Pr\left[X_n \leq \frac{n}{4}\right] = \sum_{k=0}^{n/4} \frac{1}{2^n}\binom{n}{k} \leq \frac{1}{2^n} 2 \cdot \binom{n}{n/4}$$

For $k \leq n/4$ the above sequence behave like a geometric variable.

$$\binom{n}{k+1} / \binom{n}{k} = \frac{n!}{(k+1)!(n-k-1)!} / \frac{n!}{(k)!(n-k)!}$$
$$= \frac{n-k}{k+1} \geq \frac{(3/4)n}{n/4+1} \geq 2.$$

Binomial distribution

Playing around with binomial coefficients

$$\Pr\left[X_n \leq \frac{n}{4}\right] \leq \frac{1}{2^n} 2 \cdot \binom{n}{n/4} \leq \frac{1}{2^n} 2 \cdot \left(\frac{ne}{n/4}\right)^{n/4} \leq 2 \cdot \left(\frac{4e}{2^4}\right)^{n/4}$$
$$\leq 2 \cdot 0.68^{n/4}.$$

We just proved the following theorem.

Theorem

Let X_n *be the random variable which is the number of heads when flipping an unbiased coin independently* n *times. Then*

$$\Pr\left[X_n \leq \frac{n}{4}\right] \leq 2 \cdot 0.68^{n/4} \text{ and } \Pr\left[X_n \geq \frac{3n}{4}\right] \leq 2 \cdot 0.68^{n/4}.$$

Flipping a coin.

If you flip an unbiased coin **1000** times, the probability you get at most **250** heads is at most

(A) $\leq 2 \cdot 0.68^{2000} \leq 10^{-336}$.

(B) $\leq 2 \cdot 0.68^{1000} \leq 10^{-168}$.

(C) $\leq 2 \cdot 0.68^{500} \leq 3.593 \cdot 10^{-84}$.

(D) $\leq 2 \cdot 0.68^{250} \leq 3 \cdot 10^{-42}$.

(E) 0.1.

Part III

Randomized Quick Sort and Selection

Randomized **QuickSort**

Randomized **QuickSort**

1. Pick a pivot element *uniformly at random* from the array.
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Recursively sort the subarrays, and concatenate them.

Example

1. array: 16, 12, 14, 20, 5, 3, 18, 19, 1

Analysis via Recurrence

1. Given array $\mathbf{A}$ of size $\mathbf{n}$, let $\mathbf{Q(A)}$ be number of comparisons of randomized **QuickSort** on $\mathbf{A}$.
2. Note that $\mathbf{Q(A)}$ is a random variable.
3. Let $\mathbf{A}^{\mathbf{i}}_{\text{left}}$ and $\mathbf{A}^{\mathbf{i}}_{\text{right}}$ be the left and right arrays obtained if:

pivot is of rank $\mathbf{i}$ in $\mathbf{A}$.

$$\mathbf{Q(A)} = \mathbf{n} + \sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{Pr}\left[\text{pivot has rank } \mathbf{i}\right] \left(\mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{left}}) + \mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{right}})\right).$$

Since each element of $\mathbf{A}$ has probability exactly of $\mathbf{1/n}$ of being chosen:

$$\mathbf{Q(A)} = \mathbf{n} + \sum_{\mathbf{i=1}}^{\mathbf{n}} \frac{\mathbf{1}}{\mathbf{n}} \left(\mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{left}}) + \mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{right}})\right).$$

Analysis via Recurrence

1. Given array $\mathbf{A}$ of size $\mathbf{n}$, let $\mathbf{Q(A)}$ be number of comparisons of randomized **QuickSort** on $\mathbf{A}$.
2. Note that $\mathbf{Q(A)}$ is a random variable.
3. Let $\mathbf{A}^{\mathbf{i}}_{\text{left}}$ and $\mathbf{A}^{\mathbf{i}}_{\text{right}}$ be the left and right arrays obtained if:

$$\text{pivot is of rank } \mathbf{i} \text{ in } \mathbf{A}.$$

$$\mathbf{Q(A)} = \mathbf{n} + \sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{Pr}\left[\text{pivot has rank } \mathbf{i}\right] \left(\mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{left}}) + \mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{right}})\right).$$

Since each element of $\mathbf{A}$ has probability exactly of $\mathbf{1/n}$ of being chosen:

$$\mathbf{Q(A)} = \mathbf{n} + \sum_{\mathbf{i=1}}^{\mathbf{n}} \frac{\mathbf{1}}{\mathbf{n}} \left(\mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{left}}) + \mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{right}})\right).$$

Analysis via Recurrence

Let $\mathbf{T(n) = \max_{A:|A|=n} E[Q(A)]}$ be the worst-case expected running time of randomized **QuickSort** on arrays of size $\mathbf{n}$.

We have, for any $\mathbf{A}$:

$$Q(A) = n + \sum_{i=1}^{n} \Pr\left[\text{pivot has rank } i\right] \left(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}})\right)$$

Therefore, by linearity of expectation:

$$E\left[Q(A)\right] = n + \sum_{i=1}^{n} \Pr\left[\begin{array}{c}\text{pivot is}\\ \text{of rank } i\end{array}\right] \left(E\left[Q(A^i_{\text{left}})\right] + E\left[Q(A^i_{\text{right}})\right]\right).$$

$$\Rightarrow \quad E\left[Q(A)\right] \leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right).$$

Analysis via Recurrence

Let $\mathbf{T(n) = \max_{A:|A|=n} E[Q(A)]}$ be the worst-case expected running time of randomized **QuickSort** on arrays of size $\mathbf{n}$.

We have, for any $\mathbf{A}$:

$$\mathbf{Q(A) = n + \sum_{i=1}^{n} Pr\Big[\text{pivot has rank } i\Big] \Big(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}})\Big)}$$

Therefore, by linearity of expectation:

$$\mathbf{E\Big[Q(A)\Big] = n + \sum_{i=1}^{n} Pr\left[\begin{array}{c}\text{pivot is}\\ \text{of rank } i\end{array}\right]\Big(E\Big[Q(A^i_{\text{left}})\Big] + E\Big[Q(A^i_{\text{right}})\Big]\Big).}$$

$$\mathbf{\Rightarrow \quad E\Big[Q(A)\Big] \le n + \sum_{i=1}^{n} \frac{1}{n}\left(T(i-1) + T(n-i)\right).}$$

Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} E[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.

We have, for any A:

$$Q(A) = n + \sum_{i=1}^{n} \Pr\Big[\text{pivot has rank } i\Big] \Big(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}})\Big)$$

Therefore, by linearity of expectation:

$$E\Big[Q(A)\Big] = n + \sum_{i=1}^{n} \Pr\left[\begin{array}{c}\text{pivot is}\\ \text{of rank } i\end{array}\right] \Big(E\Big[Q(A^i_{\text{left}})\Big] + E\Big[Q(A^i_{\text{right}})\Big]\Big).$$

$$\Rightarrow \quad E\Big[Q(A)\Big] \leq n + \sum_{i=1}^{n} \frac{1}{n}\left(T(i-1) + T(n-i)\right).$$

Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} E[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.

We have, for any A:

$$Q(A) = n + \sum_{i=1}^{n} \Pr\Big[\text{pivot has rank } i\Big] \Big(Q(A^i_{\text{left}}) + Q(A^i_{\text{right}})\Big)$$

Therefore, by linearity of expectation:

$$E\Big[Q(A)\Big] = n + \sum_{i=1}^{n} \Pr\left[\begin{array}{c}\text{pivot is}\\ \text{of rank } i\end{array}\right] \Big(E\Big[Q(A^i_{\text{left}})\Big] + E\Big[Q(A^i_{\text{right}})\Big]\Big).$$

$$\Rightarrow \quad E\Big[Q(A)\Big] \leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right).$$

Analysis via Recurrence

Let $\mathbf{T(n) = \max_{A:|A|=n} E[Q(A)]}$ be the worst-case expected running time of randomized **QuickSort** on arrays of size $\mathbf{n}$.

We derived:

$$\mathbf{E\left[Q(A)\right] \leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right).}$$

Note that above holds for any $\mathbf{A}$ of size $\mathbf{n}$. Therefore

$$\mathbf{\max_{A:|A|=n} E[Q(A)] = T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right).}$$

Analysis via Recurrence

Let $T(n) = \max_{A:|A|=n} E[Q(A)]$ be the worst-case expected running time of randomized **QuickSort** on arrays of size n.
We derived:

$$E\Big[Q(A)\Big] \leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right).$$

Note that above holds for any A of size n. Therefore

$$\max_{A:|A|=n} E[Q(A)] = T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} \left(T(i-1) + T(n-i)\right).$$

Solving the Recurrence

$$T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} (T(i-1) + T(n-i))$$

with base case $T(1) = 0$.

Lemma

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

Proof.

(Guess and) Verify by induction. □

Solving the Recurrence

$$T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} (T(i-1) + T(n-i))$$

with base case $T(1) = 0$.

Lemma

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

Proof.

(Guess and) Verify by induction. □

Solving the Recurrence

$$T(n) \leq n + \sum_{i=1}^{n} \frac{1}{n} (T(i-1) + T(n-i))$$

with base case $T(1) = 0$.

Lemma

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

Proof.

(Guess and) Verify by induction. □

Notes

Notes

Notes

Notes

Hoare, C. A. R. (1962). Quicksort. *Comput. J.*, 5(1):10–15.

Randomized Algorithms: QuickSort and QuickSelect

Lecture 14

March 11, 2014

Red, blue, and white balls

$\mathbf{n}$ balls, $\mathbf{k-2}$ blue balls, and $\mathbf{2}$ red balls and rest are white.

Game

Pick a ball uniformly at random, and throw it out. Repeat until a red or blue ball is thrown out for the first time.

Question: What is the probability that the last ball thrown out is red?

(A) $1/2$
(B) $(k-2)/n$.
(C) $2/n$.
(D) $2/k$.
(E) $2/(k-2)$.

Part I

Slick analysis of QuickSort

A Slick Analysis of **QuickSort**

Let $\mathbf{Q(A)}$ be number of comparisons done on input array $\mathbf{A}$:

1. For $1 \leq i < j < n$ let R_{ij} be the event that rank i element is compared with rank j element.
2. X_{ij} is the indicator random variable for R_{ij}. That is, $X_{ij} = 1$ if rank i is compared with rank j element, otherwise 0.

$$Q(A) = \sum_{1 \leq i < j \leq n} X_{ij}$$

and hence by linearity of expectation,

$$E\left[Q(A)\right] = \sum_{1 \leq i < j \leq n} E\left[X_{ij}\right] = \sum_{1 \leq i < j \leq n} Pr\left[R_{ij}\right].$$

A Slick Analysis of **QuickSort**

Let $\mathbf{Q(A)}$ be number of comparisons done on input array $\mathbf{A}$:

1. For $1 \leq i < j < n$ let R_{ij} be the event that rank i element is compared with rank j element.
2. X_{ij} is the indicator random variable for R_{ij}. That is, $X_{ij} = 1$ if rank i is compared with rank j element, otherwise 0.

$$Q(A) = \sum_{1 \leq i < j \leq n} X_{ij}$$

and hence by linearity of expectation,

$$E\left[Q(A)\right] = \sum_{1 \leq i < j \leq n} E\left[X_{ij}\right] = \sum_{1 \leq i < j \leq n} Pr\left[R_{ij}\right].$$

A Slick Analysis of **QuickSort**

$\mathbf{R_{ij}}$ = rank **i** element is compared with rank **j** element.

Question: What is $\mathbf{Pr[R_{ij}]}$?

7	5	9	1	3	4	8	6

A Slick Analysis of **QuickSort**

$\mathbf{R_{ij}}$ = rank **i** element is compared with rank **j** element.

Question: What is $\mathbf{Pr[R_{ij}]}$?

	7	5	9	1	3	4	8	6
With ranks:	6	4	8	1	2	3	7	5

A Slick Analysis of **QuickSort**

$\mathbf{R_{ij}}$ = rank **i** element is compared with rank **j** element.

Question: What is $\mathbf{Pr[R_{ij}]}$?

7	5	9	1	3	4	8	6
6	4	8	1	2	3	7	5

With ranks: 6 4 8 1 2 3 7 5

As such, probability of comparing **5** to **8** is $\mathbf{Pr[R_{4,7}]}$.

A Slick Analysis of **QuickSort**

$\mathbf{R_{ij}}$ = rank **i** element is compared with rank **j** element.

Question: What is $\mathbf{Pr[R_{ij}]}$?

7	5	9	1	3	4	8	6
6	4	8	1	2	3	7	5

With ranks: 6 4 8 1 2 3 7 5

1. If pivot too small (say **3** [rank 2]). Partition and call recursively:

7	5	9	1	3	4	8	6

$\Longrightarrow$

1	3	7	5	9	4	8	6

Decision if to compare **5** to **8** is moved to subproblem.

A Slick Analysis of **QuickSort**

$\mathbf{R_{ij}}$ = rank **i** element is compared with rank **j** element.

Question: What is $\mathbf{Pr[R_{ij}]}$?

7	5	9	1	3	4	8	6

With ranks: 6 4 8 1 2 3 7 5

1. If pivot too small (say **3** [rank 2]). Partition and call recursively:

7	5	9	1	3	4	8	6

$\Longrightarrow$

1	3	7	5	9	4	8	6

Decision if to compare **5** to **8** is moved to subproblem.

2. If pivot too large (say **9** [rank 8]):

7	5	9	1	3	4	8	6

$\Longrightarrow$

7	5	1	3	4	8	6		9

Decision if to compare **5** to **8** moved to subproblem.

A Slick Analysis of **QuickSort**

Question: What is $\mathbf{Pr[R_{i,j}]}$?

7	5	9	1	3	4	8	6
6	4	8	1	2	3	7	5

As such, probability of comparing **5** to **8** is $\mathbf{Pr[R_{4,7}]}$.

1. If pivot is **5** (rank 4). Bingo!

7	5	9	1	3	4	8	6

$\Longrightarrow$

1	3	4	5	7	9	8	6

A Slick Analysis of **QuickSort**

Question: What is $\mathbf{Pr[R_{i,j}]}$?

7	5	9	1	3	4	8	6
6	4	8	1	2	3	7	5

As such, probability of comparing **5** to **8** is $\mathbf{Pr[R_{4,7}]}$.

1. If pivot is **5** (rank 4). Bingo!

7	5	9	1	3	4	8	6

$\Longrightarrow$

1	3	4	5	7	9	8	6

2. If pivot is **8** (rank 7). Bingo!

7	5	9	1	3	4	8	6

$\Longrightarrow$

7	5	1	3	4	6	8	9

A Slick Analysis of **QuickSort**

Question: What is $\mathbf{Pr[R_{i,j}]}$?

7	5	9	1	3	4	8	6
6	4	8	1	2	3	7	5

As such, probability of comparing **5** to **8** is $\mathbf{Pr[R_{4,7}]}$.

1. If pivot is **5** (rank 4). Bingo!

7	5	9	1	3	4	8	6

$\Longrightarrow$

1	3	4	5	7	9	8	6

2. If pivot is **8** (rank 7). Bingo!

7	5	9	1	3	4	8	6

$\Longrightarrow$

7	5	1	3	4	6	8	9

3. If pivot in between the two numbers (say **6** [rank 5]):

7	5	9	1	3	4	8	6

$\Longrightarrow$

5	1	3	4	6	7	8	9

5 and **8** will never be compared to each other.

A Slick Analysis of **QuickSort**

Question: What is $\mathbf{Pr}[\mathbf{R}_{i,j}]$?

Conclusion:

$\mathbf{R}_{i,j}$ happens if and only if:

> **i**th or **j**th ranked element is the first pivot out of **i**th to **j**th ranked elements.

How to analyze this?

Thinking acrobatics!

1. Assign every element in the array a random priority (say in $[0, 1]$).
2. Choose pivot to be the element with lowest priority in subproblem.
3. Equivalent to picking pivot uniformly at random (as **QuickSort** do).

A Slick Analysis of **QuickSort**

Question: What is $\Pr[R_{i,j}]$?

How to analyze this?

Thinking acrobatics!

1. Assign every element in the array a random priority (say in $[0, 1]$).
2. Choose pivot to be the element with lowest priority in subproblem.

$\Longrightarrow$ $R_{i,j}$ happens if either i or j have lowest priority out of elements rank i to j,

There are $k = j - i + 1$ relevant elements.

$$\Pr\left[R_{i,j}\right] = \frac{2}{k} = \frac{2}{j - i + 1}.$$

A Slick Analysis of **QuickSort**

Question: What is $\Pr[R_{i,j}]$?

How to analyze this?

Thinking acrobatics!

1. Assign every element in the array a random priority (say in $[0, 1]$).
2. Choose pivot to be the element with lowest priority in subproblem.

$\implies$ $R_{i,j}$ happens if either i or j have lowest priority out of elements rank i to j,
There are $k = j - i + 1$ relevant elements.

$$\Pr\left[R_{i,j}\right] = \frac{2}{k} = \frac{2}{j - i + 1}.$$

A Slick Analysis of **QuickSort**

Question: What is $\Pr[R_{ij}]$?

Lemma

$\Pr\left[R_{ij}\right] = \frac{2}{j-i+1}$.

Proof.

Let $a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n$ be elements of A in sorted order. Let $S = \{a_i, a_{i+1}, \ldots, a_j\}$

Observation: If pivot is chosen outside S then all of S either in left array or right array.

Observation: a_i and a_j separated when a pivot is chosen from S for the first time. Once separated no comparison.

Observation: a_i is compared with a_j if and only if either a_i or a_j is chosen as a pivot from S at separation... □

A Slick Analysis of **QuickSort**

Question: What is $\Pr[R_{ij}]$?

Lemma

$$\Pr\left[R_{ij}\right] = \frac{2}{j-i+1}.$$

Proof.

Let $a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n$ be elements of A in sorted order. Let $S = \{a_i, a_{i+1}, \ldots, a_j\}$

Observation: If pivot is chosen outside S then all of S either in left array or right array.

Observation: a_i and a_j separated when a pivot is chosen from S for the first time. Once separated no comparison.

Observation: a_i is compared with a_j if and only if either a_i or a_j is chosen as a pivot from S at separation... □

A Slick Analysis of **QuickSort**

Question: What is $\mathbf{Pr[R_{ij}]}$?

Lemma

$\mathbf{Pr}\left[\mathbf{R_{ij}}\right] = \frac{2}{j-i+1}$.

Proof.

Let $\mathbf{a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n}$ be elements of $\mathbf{A}$ in sorted order. Let $\mathbf{S = \{a_i, a_{i+1}, \ldots, a_j\}}$

Observation: If pivot is chosen outside $\mathbf{S}$ then all of $\mathbf{S}$ either in left array or right array.

Observation: $\mathbf{a_i}$ and $\mathbf{a_j}$ separated when a pivot is chosen from $\mathbf{S}$ for the first time. Once separated no comparison.

Observation: $\mathbf{a_i}$ is compared with $\mathbf{a_j}$ if and only if either $\mathbf{a_i}$ or $\mathbf{a_j}$ is chosen as a pivot from $\mathbf{S}$ at separation... □

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\Pr\left[R_{ij}\right] = \frac{2}{j-i+1}$.

Proof.

Let $a_1, \ldots, a_i, \ldots, a_j, \ldots, a_n$ be sort of A. Let $S = \{a_i, a_{i+1}, \ldots, a_j\}$

Observation: a_i is compared with a_j if and only if either a_i or a_j is chosen as a pivot from S at separation.

Observation: Given that pivot is chosen from S the probability that it is a_i or a_j is exactly $2/|S| = 2/(j-i+1)$ since the pivot is chosen uniformly at random from the array. □

How much is this?

$\mathbf{H_n} = \sum_{\mathbf{i=1}}^{\mathbf{n}} \frac{\mathbf{1}}{\mathbf{i}}$ is the $\mathbf{n}$'th harmonic number

(A) $\mathbf{H_n = \Theta(1)}$.

(B) $\mathbf{H_n = \Theta(\log \log n)}$.

(C) $\mathbf{H_n = \Theta(\sqrt{\log n})}$.

(D) $\mathbf{H_n = \Theta(\log n)}$.

(E) $\mathbf{H_n = \Theta(\log^2 n)}$.

And how much is this?

$$T_n = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} \frac{1}{j}$$

is equal to

(A) $T_n = \Theta(n)$.

(B) $T_n = \Theta(n \log n)$.

(C) $T_n = \Theta(n \log^2 n)$.

(D) $T_n = \Theta(n^2)$.

(E) $T_n = \Theta(n^3)$.

A Slick Analysis of **QuickSort**

Continued...

$$E\Big[Q(A)\Big] = \sum_{1 \le i < j \le n} E[X_{ij}] = \sum_{1 \le i < j \le n} Pr[R_{ij}]\,.$$

Lemma

$Pr[R_{ij}] = \frac{2}{j-i+1}$.

$$E\Big[Q(A)\Big] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\Pr[R_{ij}] = \frac{2}{j-i+1}$.

$$E\Big[Q(A)\Big] = \sum_{1 \le i < j \le n} \Pr\Big[R_{ij}\Big] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\Pr[R_{ij}] = \frac{2}{j-i+1}$.

$$E\Big[Q(A)\Big] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\Pr[R_{ij}] = \frac{2}{j-i+1}$.

$$E\left[Q(A)\right] = \sum_{1 \le i < j \le n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\mathbf{Pr[R_{ij}] = \frac{2}{j-i+1}}$.

$$\mathbf{E\Big[Q(A)\Big] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}}$$

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\Pr[R_{ij}] = \frac{2}{j-i+1}$.

$$E\Big[Q(A)\Big] = 2\sum_{i=1}^{n-1}\sum_{i<j}^{n}\frac{1}{j-i+1}$$

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\Pr[R_{ij}] = \frac{2}{j-i+1}$.

$$E\Big[Q(A)\Big] = 2\sum_{i=1}^{n-1}\sum_{i<j}^{n}\frac{1}{j-i+1}$$

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\Pr[R_{ij}] = \frac{2}{j-i+1}$.

$$E\Big[Q(A)\Big] = 2\sum_{i=1}^{n-1}\sum_{i<j}^{n}\frac{1}{j-i+1} \leq 2\sum_{i=1}^{n-1}\sum_{\Delta=2}^{n-i+1}\frac{1}{\Delta}$$

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\Pr[R_{ij}] = \frac{2}{j-i+1}$.

$$E\Big[Q(A)\Big] = 2\sum_{i=1}^{n-1}\sum_{i<j}^{n}\frac{1}{j-i+1} \leq 2\sum_{i=1}^{n-1}\sum_{\Delta=2}^{n-i+1}\frac{1}{\Delta}$$

$$\leq 2\sum_{i=1}^{n-1}(H_{n-i+1}-1) \;\leq\; 2\sum_{1\leq i<n} H_n$$

A Slick Analysis of **QuickSort**

Continued...

Lemma

$\Pr[R_{ij}] = \frac{2}{j-i+1}$.

$$E\Big[Q(A)\Big] = 2\sum_{i=1}^{n-1}\sum_{i<j}^{n}\frac{1}{j-i+1} \leq 2\sum_{i=1}^{n-1}\sum_{\Delta=2}^{n-i+1}\frac{1}{\Delta}$$

$$\leq 2\sum_{i=1}^{n-1}(H_{n-i+1}-1) \;\leq\; 2\sum_{1\leq i<n} H_n$$

$$\leq 2nH_n = O(n\log n)$$

Yet another analysis of QuickSort

You should never trust a man who has only one way to spell a word

Consider element $\mathbf{e}$ in the array.
Consider the subproblems it participates in during **QuickSort** execution:
$\mathbf{S_1, S_2, \ldots, S_k}$.

Definition

$\mathbf{e}$ is lucky in the $\mathbf{j}$th iteration if $|\mathbf{S_j}| \leq \mathbf{(3/4)}\,|\mathbf{S_{j-1}}|$.

Key observation

The event $\mathbf{e}$ is lucky in $\mathbf{j}$th iteration
is independent of
the event that $\mathbf{e}$ is lucky in $\mathbf{k}$th iteration,
(If $\mathbf{j} \neq \mathbf{k}$)

$\mathbf{X_j = 1}$ **iff** $\mathbf{e}$ is lucky in the $\mathbf{j}$th iteration.

Yet another analysis of QuickSort

Continued...

Claim

$\Pr[X_j = 1] = 1/2$.

Proof.

1. X_j determined by j recursive subproblem.
2. Subproblem has $n_{j-1} = |X_{j-1}|$ elements.
3. If jth pivot rank $\in [n_{j-1}/4, (3/4)n_{j-1}]$, then e lucky in jth iter.
4. Prob. e is lucky $\geq |[n_{j-1}/4, (3/4)n_{j-1}]| / n_{j-1} = 1/2$. $\square$

Observation

If $X_1 + X_2 + \ldots X_k = \lceil \log_{4/3} n \rceil$ then e subproblem is of size one. Done!

Yet another analysis of QuickSort

Continued...

Observation

Probability $\mathbf{e}$ participates in $\geq \mathbf{k} = \mathbf{40}\lceil \mathbf{\log_{4/3} n} \rceil$ subproblems. Is equal to

$$\begin{aligned}
\mathbf{Pr}\Big[\mathbf{X_1 + X_2 + \ldots + X_k} \leq \lceil \mathbf{\log_{4/3} n} \rceil\Big] \\
\leq \mathbf{Pr}[\mathbf{X_1 + X_2 + \ldots + X_k} \leq \mathbf{k/4}] \\
\leq \mathbf{2 \cdot 0.68^{k/4}} \leq \mathbf{1/n^5}.
\end{aligned}$$

Conclusion

QuickSort takes $\mathbf{O(n \log n)}$ time with high probability.

Because...

Theorem

Let $\mathbf{X_n}$ *be the number heads when flipping a coin indepdently* $\mathbf{n}$ *times. Then*

$$\mathbf{Pr}\left[\mathbf{X_n} \leq \frac{\mathbf{n}}{\mathbf{4}}\right] \leq \mathbf{2 \cdot 0.68^{n/4}} \textit{ and } \mathbf{Pr}\left[\mathbf{X_n} \geq \frac{\mathbf{3n}}{\mathbf{4}}\right] \leq \mathbf{2 \cdot 0.68^{n/4}}$$

Randomized Quick Selection

Input Unsorted array **A** of **n** integers

Goal Find the **j**th smallest number in **A** (*rank* **j** number)

Randomized Quick Selection

1. Pick a pivot element *uniformly at random* from the array
2. Split array into 3 subarrays: those smaller than pivot, those larger than pivot, and the pivot itself.
3. Return pivot if rank of pivot is **j**.
4. Otherwise recurse on one of the arrays depending on **j** and their sizes.

Algorithm for Randomized Selection

Assume for simplicity that $\mathbf{A}$ has distinct elements.

```
QuickSelect(A, j):
    Pick pivot x uniformly at random from A
    Partition A into A_less, x, and A_greater using x as pivot
    if (|A_less| = j − 1) then
        return x
    if (|A_less| ≥ j) then
        return QuickSelect(A_less, j)
    else
        return QuickSelect(A_greater, j − |A_less| − 1)
```

QuickSelect analysis

1. $S_1, S_2, \ldots, S_k$ be the subproblems considered by the algorithm. Here $|S_1| = n$.
2. S_i would be **successful** if $|S_i| \leq (3/4)\,|S_{i-1}|$
3. Y_1 = number of recursive calls till first successful iteration. Clearly, total work till this happens is $O(Y_1 n)$.
4. n_i = size of the subproblem immediately after the $(i-1)$th successful iteration.
5. Y_i = number of recursive calls after the $(i-1)$th successful call, till the ith successful iteration.
6. Running time is $O(\sum_i n_i Y_i)$.

QuickSelect analysis

Example

S_i = subarray used in ith recursive call

$|S_i|$ = size of this subarray

Red indicates successful iteration.

Inst'	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
$\lvert S_i \rvert$	100	70	60	50	40	30	25	5	2
Succ'	$Y_1 = 2$		$Y_2 = 4$				$Y_3 = 2$		$Y_4 = 1$
$n_i =$	$n_1 = 100$		$n_2 = 60$				$n_3 = 25$		$n_4 = 2$

1. All the subproblems after $(i-1)$th successful iteration till ith successful iteration have size $\leq n_i$.
2. Total work: $O(\sum_i n_i Y_i)$.

QuickSelect analysis

Total work: $O(\sum_i n_i Y_i)$.
We have:

1. $n_i \leq (3/4)n_{i-1} \leq (3/4)^{i-1}n$.
2. Y_i is a random variable with geometric distribution Probability of $Y_i = k$ is $1/2^i$.
3. $E[Y_i] = 2$.

As such, expected work is proportional to

$$E\left[\sum_i n_i Y_i\right] = \sum_i E\left[n_i Y_i\right] \leq \sum_i E\left[(3/4)^{i-1} n Y_i\right]$$
$$= n \sum_i (3/4)^{i-1} E\left[Y_i\right] = n \sum_{i=1} (3/4)^{i-1} 2 \leq 8n.$$

QuickSelect analysis

Theorem

The expected running time of **QuickSelect** *is* $\mathbf{O(n)}$.

QuickSelect analysis

Analysis via Recurrence

1. Given array $\mathbf{A}$ of size $\mathbf{n}$ let $\mathbf{Q(A)}$ be number of comparisons of randomized selection on $\mathbf{A}$ for selecting rank $\mathbf{j}$ element.
2. Note that $\mathbf{Q(A)}$ is a random variable
3. Let $\mathbf{A}^{\mathbf{i}}_{\text{less}}$ and $\mathbf{A}^{\mathbf{i}}_{\text{greater}}$ be the left and right arrays obtained if pivot is rank $\mathbf{i}$ element of $\mathbf{A}$.
4. Algorithm recurses on $\mathbf{A}^{\mathbf{i}}_{\text{less}}$ if $\mathbf{j} < \mathbf{i}$ and recurses on $\mathbf{A}^{\mathbf{i}}_{\text{greater}}$ if $\mathbf{j} > \mathbf{i}$ and terminates if $\mathbf{j} = \mathbf{i}$.

$$\mathbf{Q(A)} = \mathbf{n} + \sum_{\mathbf{i}=1}^{\mathbf{j}-1} \mathbf{Pr}[\text{pivot has rank } \mathbf{i}]\, \mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{greater}}) + \sum_{\mathbf{i}=\mathbf{j}+1}^{\mathbf{n}} \mathbf{Pr}[\text{pivot has rank } \mathbf{i}]\, \mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{less}})$$

QuickSelect analysis

Analysis via Recurrence

1. Given array $\mathbf{A}$ of size $\mathbf{n}$ let $\mathbf{Q(A)}$ be number of comparisons of randomized selection on $\mathbf{A}$ for selecting rank $\mathbf{j}$ element.
2. Note that $\mathbf{Q(A)}$ is a random variable
3. Let $\mathbf{A}^{\mathbf{i}}_{\text{less}}$ and $\mathbf{A}^{\mathbf{i}}_{\text{greater}}$ be the left and right arrays obtained if pivot is rank $\mathbf{i}$ element of $\mathbf{A}$.
4. Algorithm recurses on $\mathbf{A}^{\mathbf{i}}_{\text{less}}$ if $\mathbf{j} < \mathbf{i}$ and recurses on $\mathbf{A}^{\mathbf{i}}_{\text{greater}}$ if $\mathbf{j} > \mathbf{i}$ and terminates if $\mathbf{j} = \mathbf{i}$.

$$\begin{aligned} \mathbf{Q(A)} &= \mathbf{n} + \sum_{\mathbf{i}=1}^{\mathbf{j}-1} \mathbf{Pr}[\text{pivot has rank } \mathbf{i}]\, \mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{greater}}) \\ &\quad + \sum_{\mathbf{i}=\mathbf{j}+1}^{\mathbf{n}} \mathbf{Pr}[\text{pivot has rank } \mathbf{i}]\, \mathbf{Q}(\mathbf{A}^{\mathbf{i}}_{\text{less}}) \end{aligned}$$

Analyzing the Recurrence

As in **QuickSort** we obtain the following recurrence where $\mathbf{T(n)}$ is the worst-case expected time.

$$\mathbf{T(n) \leq n + \frac{1}{n}\left(\sum_{i=1}^{j-1} T(n-i) + \sum_{i=j}^{n} T(i-1)\right).}$$

Theorem

$\mathbf{T(n) = O(n)}$.

Proof.

(Guess and) Verify by induction (see next slide). □

Analyzing the recurrence

Theorem

$T(n) = O(n)$.

Prove by induction that $T(n) \leq \alpha n$ for some constant $\alpha \geq 1$ to be fixed later.

Base case: $n = 1$, we have $T(1) = 0$ since no comparisons needed and hence $T(1) \leq \alpha$.

Induction step: Assume $T(k) \leq \alpha k$ for $1 \leq k < n$ and prove it for $T(n)$. We have by the recurrence:

$$\begin{aligned} T(n) &\leq n + \frac{1}{n}\left(\sum_{i=1}^{j-1} T(n-i) + \sum_{i=j^n} T(i-1)\right) \\ &\leq n + \frac{\alpha}{n}\left(\sum_{i=1}^{j-1}(n-i) + \sum_{i=j}^{n}(i-1)\right) \quad \text{by applying induction} \end{aligned}$$

Analyzing the recurrence

$$
\begin{aligned}
T(n) &\leq n + \frac{\alpha}{n}\left(\sum_{i=1}^{j-1}(n-i) + \sum_{i=j}^{n}(i-1)\right) \\
&\leq n + \frac{\alpha}{n}((j-1)(2n-j)/2 + (n-j+1)(n+j-2)/2) \\
&\leq n + \frac{\alpha}{2n}(n^2 + 2nj - 2j^2 - 3n + 4j - 2) \\
& \quad \text{above expression maximized when } j = (n+1)/2\text{: calculus} \\
&\leq n + \frac{\alpha}{2n}(3n^2/2 - n) \quad \text{substituting } (n+1)/2 \text{ for } j \\
&\leq n + 3\alpha n/4 \\
&\leq \alpha n \quad \text{for any constant } \alpha \geq 4
\end{aligned}
$$

Comments on analyzing the recurrence

1. Algebra looks messy but intuition suggest that the median is the hardest case and hence can plug $\mathbf{j = n/2}$ to simplify without calculus
2. Analyzing recurrences comes with practice and after a while one can see things more intuitively

John Von Neumann:
Young man, in mathematics you don't understand things. You just get used to them.

Notes

Notes

Notes

Notes

Hashing

Lecture 15

March 13, 2014

Part I

Hash Tables

Dictionary Data Structure

1. $\mathcal{U}$: universe of keys with total order: numbers, strings, etc.
2. Data structure to store a subset $\mathbf{S} \subseteq \mathcal{U}$
3. **Operations:**
   1. **Search/lookup**: given $\mathbf{x} \in \mathcal{U}$ is $\mathbf{x} \in \mathbf{S}$?
   2. **Insert**: given $\mathbf{x} \notin \mathbf{S}$ add $\mathbf{x}$ to $\mathbf{S}$.
   3. **Delete**: given $\mathbf{x} \in \mathbf{S}$ delete $\mathbf{x}$ from $\mathbf{S}$
4. **Static** structure: $\mathbf{S}$ given in advance or changes very infrequently, main operations are lookups.
5. **Dynamic** structure: $\mathbf{S}$ changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structures

Common solutions:

1. Static:
   1. Store **S** as a *sorted* array
   2. **Lookup**: Binary search in $\mathbf{O(\log |S|)}$ time (comparisons)
2. Dynamic:
   1. Store **S** in a *balanced* binary search tree
   2. Lookup, Insert, Delete in $\mathbf{O(\log |S|)}$ time (comparisons)

Dictionary Data Structures

Question: "Should Tables be Sorted?"
(also title of famous paper by Turing award winner Andy Yao)

Hashing is a widely used & powerful technique for dictionaries.

Motivation:

1. Universe $\mathcal{U}$ may not be (naturally) totally ordered.
2. Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive.
3. Want to improve "average" performance of lookups to $O(1)$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Dictionary Data Structures

Question: "Should Tables be Sorted?"
(also title of famous paper by Turing award winner Andy Yao)

Hashing is a widely used & powerful technique for dictionaries.

Motivation:

1. Universe $\mathcal{U}$ may not be (naturally) totally ordered.
2. Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive.
3. Want to improve "average" performance of lookups to $\mathbf{O(1)}$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Hashing and Hash Tables

Hash Table data structure:

1. A (hash) table/array $\mathbf{T}$ of size $\mathbf{m}$ (the table **size**).
2. A hash function $\mathbf{h} : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
3. Item $\mathbf{x} \in \mathcal{U}$ hashes to slot $\mathbf{h(x)}$ in $\mathbf{T}$.

Given $\mathbf{S} \subseteq \mathcal{U}$. How do we store $\mathbf{S}$ and how do we do lookups?

Ideal situation:

1. Each element $\mathbf{x} \in \mathbf{S}$ hashes to a distinct slot in $\mathbf{T}$. Store $\mathbf{x}$ in slot $\mathbf{h(x)}$
2. **Lookup**: Given $\mathbf{y} \in \mathcal{U}$ check if $\mathbf{T[h(y)] = y}$. $\mathbf{O(1)}$ time!

Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

Hash Table data structure:

1. A (hash) table/array $\mathbf{T}$ of size $\mathbf{m}$ (the table **size**).
2. A hash function $\mathbf{h} : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
3. Item $\mathbf{x} \in \mathcal{U}$ hashes to slot $\mathbf{h(x)}$ in $\mathbf{T}$.

Given $\mathbf{S} \subseteq \mathcal{U}$. How do we store $\mathbf{S}$ and how do we do lookups?

Ideal situation:

1. Each element $\mathbf{x} \in \mathbf{S}$ hashes to a distinct slot in $\mathbf{T}$. Store $\mathbf{x}$ in slot $\mathbf{h(x)}$
2. **Lookup**: Given $\mathbf{y} \in \mathcal{U}$ check if $\mathbf{T[h(y)] = y}$. $\mathbf{O(1)}$ time!

Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

Hash Table data structure:

1. A (hash) table/array $\mathbf{T}$ of size $\mathbf{m}$ (the table **size**).
2. A hash function $\mathbf{h} : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
3. Item $\mathbf{x} \in \mathcal{U}$ hashes to slot $\mathbf{h(x)}$ in $\mathbf{T}$.

Given $\mathbf{S} \subseteq \mathcal{U}$. How do we store $\mathbf{S}$ and how do we do lookups?

Ideal situation:

1. Each element $\mathbf{x} \in \mathbf{S}$ hashes to a distinct slot in $\mathbf{T}$. Store $\mathbf{x}$ in slot $\mathbf{h(x)}$
2. **Lookup**: Given $\mathbf{y} \in \mathcal{U}$ check if $\mathbf{T[h(y)] = y}$. $\mathbf{O(1)}$ time!

Collisions unavoidable. Several different techniques to handle them.

Hashing and Hash Tables

Hash Table data structure:

1. A (hash) table/array $\mathbf{T}$ of size $\mathbf{m}$ (the table **size**).
2. A hash function $\mathbf{h} : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$.
3. Item $\mathbf{x} \in \mathcal{U}$ hashes to slot $\mathbf{h(x)}$ in $\mathbf{T}$.

Given $\mathbf{S} \subseteq \mathcal{U}$. How do we store $\mathbf{S}$ and how do we do lookups?

Ideal situation:

1. Each element $\mathbf{x} \in \mathbf{S}$ hashes to a distinct slot in $\mathbf{T}$. Store $\mathbf{x}$ in slot $\mathbf{h(x)}$
2. **Lookup**: Given $\mathbf{y} \in \mathcal{U}$ check if $\mathbf{T[h(y)] = y}$. $\mathbf{O(1)}$ time!

Collisions unavoidable. Several different techniques to handle them.

Handling Collisions: Chaining

Collision: $\mathbf{h(x) = h(y)}$ for some $\mathbf{x \neq y}$.

Chaining to handle collisions:

1. For each slot $\mathbf{i}$ store all items hashed to slot $\mathbf{i}$ in a linked list. $\mathbf{T[i]}$ points to the linked list
2. **Lookup**: to find if $\mathbf{y} \in \mathcal{U}$ is in $\mathbf{T}$, check the linked list at $\mathbf{T[h(y)]}$. Time proportion to size of linked list.

This is also known as **Open hashing**.

Handling Collisions

Several other techniques:

1. Open addressing.
   Every element has a list of places it can be (in certain order). Check in this order.
2. ...
3. Cuckoo hashing.
   Every value has two possible locations. When inserting, insert in one of the locations, otherwise, kick stored value to its other location. Repeat till stable. if no stability then rebuild table.
4. Others.

Understanding Hashing

Does hashing give **O(1)** time per operation for dictionaries?

Questions:

1. Complexity of evaluating **h** on a given element?
2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored **S**.
3. Size of table relative to size of **S**.
4. Worst-case vs average-case vs randomized (expected) time?
5. How do we choose **h**?

Understanding Hashing

Does hashing give **O(1)** time per operation for dictionaries?

Questions:

1. Complexity of evaluating **h** on a given element?
2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored **S**.
3. Size of table relative to size of **S**.
4. Worst-case vs average-case vs randomized (expected) time?
5. How do we choose **h**?

Understanding Hashing

1. Complexity of evaluating $\mathbf{h}$ on a given element? Should be small.
2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored $\mathbf{S}$: typically $|\mathcal{U}| \gg |\mathbf{S}|$.
3. Size of table relative to size of $\mathbf{S}$. The **load factor** of $\mathbf{T}$ is the ratio $\mathbf{n}/\mathbf{t}$ where $\mathbf{n} = |\mathbf{S}|$ and $\mathbf{m} = |\mathbf{T}|$. Typically $\mathbf{n}/\mathbf{t}$ is a small constant smaller than $\mathbf{1}$.
   Also known as the **fill factor**.

Main and interrelated questions:

1. Worst-case vs average-case vs randomized (expected) time?
2. How do we choose $\mathbf{h}$?

Understanding Hashing

1. Complexity of evaluating $\mathbf{h}$ on a given element? Should be small.
2. Relative sizes of the universe $\mathcal{U}$ and the set to be stored $\mathbf{S}$: typically $|\mathcal{U}| \gg |\mathbf{S}|$.
3. Size of table relative to size of $\mathbf{S}$. The **load factor** of $\mathbf{T}$ is the ratio $\mathbf{n}/\mathbf{t}$ where $\mathbf{n} = |\mathbf{S}|$ and $\mathbf{m} = |\mathbf{T}|$. Typically $\mathbf{n}/\mathbf{t}$ is a small constant smaller than $\mathbf{1}$.
   Also known as the **fill factor**.

Main and interrelated questions:

1. Worst-case vs average-case vs randomized (expected) time?
2. How do we choose $\mathbf{h}$?

Single hash function

1. $\mathcal{U}$: universe (very large).
2. Assume $\mathbf{N} = |\mathcal{U}| \gg \mathbf{m}$ where $\mathbf{m}$ is size of table $\mathbf{T}$. In particular assume $\mathbf{N} \geq \mathbf{m}^2$ (very conservative).
3. Fix hash function $\mathbf{h} : \mathcal{U} \rightarrow \{0, \ldots, \mathbf{m} - 1\}$.
4. $\mathbf{N}$ items hashed to $\mathbf{m}$ slots. By pigeon hole principle there is some $\mathbf{i} \in \{0, \ldots, \mathbf{m} - 1\}$ such that $\mathbf{N}/\mathbf{m} \geq \mathbf{m}$ elements of $\mathcal{U}$ get hashed to $\mathbf{i}$ (!).
5. Implies that there is a set $\mathbf{S} \subseteq \mathcal{U}$ where $|\mathbf{S}| = \mathbf{m}$ such that all of $\mathbf{S}$ hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

Single hash function

1. $\mathcal{U}$: universe (very large).
2. Assume $\mathbf{N} = |\mathcal{U}| \gg \mathbf{m}$ where $\mathbf{m}$ is size of table $\mathbf{T}$. In particular assume $\mathbf{N} \geq \mathbf{m}^2$ (very conservative).
3. Fix hash function $\mathbf{h} : \mathcal{U} \rightarrow \{0, \ldots, \mathbf{m} - 1\}$.
4. $\mathbf{N}$ items hashed to $\mathbf{m}$ slots. By pigeon hole principle there is some $\mathbf{i} \in \{0, \ldots, \mathbf{m} - 1\}$ such that $\mathbf{N}/\mathbf{m} \geq \mathbf{m}$ elements of $\mathcal{U}$ get hashed to $\mathbf{i}$ (!).
5. Implies that there is a set $\mathbf{S} \subseteq \mathcal{U}$ where $|\mathbf{S}| = \mathbf{m}$ such that all of $\mathbf{S}$ hashes to same slot. Ooops.

Lesson: For every hash function there is a very bad set. Bad set. Bad.

How many hash functions are there, anyway?

Let $\mathcal{H}$ be the set of all functions from $\mathcal{U} = \{1, \ldots, U\}$ to $\{1, \ldots, m\}$. The number of functions in $\mathcal{H}$ is

(A) $U + m$.

(B) Um.

(C) U^m.

(D) m^U.

(E) $\binom{U+m}{m}$.

(F) The answer is blowing in the wind.

How many bits one need?

Let $\mathcal{H}$ be a set of functions from $\mathcal{U} = \{1, \ldots, U\}$ to $\{1, \ldots, m\}$. Specifying a function in $\mathcal{H}$ requires:

(A) $O(U + m)$ bits.

(B) $O(Um)$ bits.

(C) $O(U^m)$ bits.

(D) $O\left(m^U\right)$ bits.

(E) $O(\log |\mathcal{H}|)$ bits.

(F) Many many bits. At least two.

Picking a hash function

1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.

Parameters: $\mathbf{N} = |\mathcal{U}|$, $\mathbf{m} = |\mathbf{T}|$, $\mathbf{n} = |\mathbf{S}|$

1. $\mathcal{H}$ is a **family** of hash functions: each function $\mathbf{h} \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $\mathbf{h(x)}$).
2. $\mathbf{h}$ is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
3. Randomized guarantee: should have the property that for any *fixed* set $\mathbf{S} \subseteq \mathcal{U}$ of size $\mathbf{m}$ the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of $\mathbf{h}$.

Picking a hash function

1. Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
2. Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.

Parameters: $\mathbf{N} = |\mathcal{U}|$, $\mathbf{m} = |\mathbf{T}|$, $\mathbf{n} = |\mathbf{S}|$

1. $\mathcal{H}$ is a **family** of hash functions: each function $\mathbf{h} \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $\mathbf{h(x)}$).
2. $\mathbf{h}$ is chosen **randomly** from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
3. Randomized guarantee: should have the property that for any *fixed* set $\mathbf{S} \subseteq \mathcal{U}$ of size $\mathbf{m}$ the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of $\mathbf{h}$.

Picking a hash function

Question: Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?

1. Too many functions! A random function has high complexity!
   # of functions: $M = m^{|\mathcal{U}|}$.
   Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.

Question: Are there good and compact families $\mathcal{H}$?

1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Picking a hash function

Question: Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?

1. Too many functions! A random function has high complexity!
   # of functions: $M = m^{|\mathcal{U}|}$.
   Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.

Question: Are there good and compact families $\mathcal{H}$?

1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Picking a hash function

Question: Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?

1. Too many functions! A random function has high complexity!
   # of functions: $M = m^{|\mathcal{U}|}$.
   Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.

Question: Are there good and compact families $\mathcal{H}$?

1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Picking a hash function

Question: Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?

1. Too many functions! A random function has high complexity!
   # of functions: $M = m^{|\mathcal{U}|}$.
   Bits to encode such a function $\approx \log M = |\mathcal{U}| \log m$.

Question: Are there good and compact families $\mathcal{H}$?

1. Yes... But what it means for $\mathcal{H}$ to be good and compact.

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \leq i < m$.
2. Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $\mathbf{x} \in \mathcal{U}$. Then if $\mathbf{h} \in \mathcal{H}$ is picked randomly then $\mathbf{x}$ should go into a random slot in $\mathbf{T}$. In other words $\Pr[\mathbf{h(x) = i}] = \mathbf{1/m}$ for every $\mathbf{0 \leq i < m}$.
2. Consider any two distinct elements $\mathbf{x, y} \in \mathcal{U}$. Then if $\mathbf{h} \in \mathcal{H}$ is picked randomly then the probability of a collision between $\mathbf{x}$ and $\mathbf{y}$ should be at most $\mathbf{1/m}$. In other words $\Pr[\mathbf{h(x) = h(y)}] = \mathbf{1/m}$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $\mathbf{x, y} \in \mathcal{U}$, $\Pr[\mathbf{h(x) = h(y)}] = \mathbf{1/m}$ where $\mathbf{m}$ is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $\mathbf{x} \in \mathcal{U}$. Then if $\mathbf{h} \in \mathcal{H}$ is picked randomly then $\mathbf{x}$ should go into a random slot in $\mathbf{T}$. In other words $\Pr[\mathbf{h(x) = i}] = \mathbf{1/m}$ for every $\mathbf{0 \leq i < m}$.
2. Consider any two distinct elements $\mathbf{x, y} \in \mathcal{U}$. Then if $\mathbf{h} \in \mathcal{H}$ is picked randomly then the probability of a collision between $\mathbf{x}$ and $\mathbf{y}$ should be at most $\mathbf{1/m}$. In other words $\Pr[\mathbf{h(x) = h(y)}] = \mathbf{1/m}$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $\mathbf{x, y} \in \mathcal{U}$, $\Pr[\mathbf{h(x) = h(y)}] = \mathbf{1/m}$ where $\mathbf{m}$ is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $\mathbf{x} \in \mathcal{U}$. Then if $\mathbf{h} \in \mathcal{H}$ is picked randomly then $\mathbf{x}$ should go into a random slot in $\mathbf{T}$. In other words $\Pr[\mathbf{h(x) = i}] = \mathbf{1/m}$ for every $\mathbf{0 \leq i < m}$.
2. Consider any two distinct elements $\mathbf{x, y} \in \mathcal{U}$. Then if $\mathbf{h} \in \mathcal{H}$ is picked randomly then the probability of a collision between $\mathbf{x}$ and $\mathbf{y}$ should be at most $\mathbf{1/m}$. In other words $\Pr[\mathbf{h(x) = h(y)}] = \mathbf{1/m}$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $\mathbf{x, y} \in \mathcal{U}$, $\Pr[\mathbf{h(x) = h(y)}] = \mathbf{1/m}$ where $\mathbf{m}$ is the table size.

Note: The set of all hash functions satisfies stronger properties!

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

1. Consider any element $\mathbf{x} \in \mathcal{U}$. Then if $\mathbf{h} \in \mathcal{H}$ is picked randomly then $\mathbf{x}$ should go into a random slot in $\mathbf{T}$. In other words $\Pr[\mathbf{h(x) = i}] = \mathbf{1/m}$ for every $\mathbf{0 \leq i < m}$.
2. Consider any two distinct elements $\mathbf{x, y} \in \mathcal{U}$. Then if $\mathbf{h} \in \mathcal{H}$ is picked randomly then the probability of a collision between $\mathbf{x}$ and $\mathbf{y}$ should be at most $\mathbf{1/m}$. In other words $\Pr[\mathbf{h(x) = h(y)}] = \mathbf{1/m}$ (cannot be smaller).
3. Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $\mathbf{x, y} \in \mathcal{U}$, $\Pr[\mathbf{h(x) = h(y)}] = \mathbf{1/m}$ where $\mathbf{m}$ is the table size.

Note: The set of all hash functions satisfies stronger properties!

Analyzing Uniform Hashing

1. $\mathbf{T}$ is hash table of size $\mathbf{m}$.
2. $\mathbf{S} \subseteq \mathcal{U}$ is a **fixed** set of size $\leq \mathbf{m}$.
3. $\mathbf{h}$ is chosen randomly from a uniform hash family $\mathcal{H}$.
4. $\mathbf{x}$ is a *fixed* element of $\mathcal{U}$. Assume for simplicity that $\mathbf{x} \notin \mathbf{S}$.

Question: What is the *expected* time to look up $\mathbf{x}$ in $\mathbf{T}$ using $\mathbf{h}$ assuming chaining used to resolve collisions?

Analyzing Uniform Hashing

Question: What is the *expected* time to look up $\mathbf{x}$ in $\mathbf{T}$ using $\mathbf{h}$ assuming chaining used to resolve collisions?

1. The time to look up $\mathbf{x}$ is the size of the list at $\mathbf{T[h(x)]}$: same as the number of elements in $\mathbf{S}$ that collide with $\mathbf{x}$ under $\mathbf{h}$.
2. Let $\ell(\mathbf{x})$ be this number. We want $\mathrm{E}[\ell(\mathbf{x})]$
3. For $\mathbf{y} \in \mathbf{S}$ let $\mathbf{A_y}$ be the even that $\mathbf{x}, \mathbf{y}$ collide and $\mathbf{D_y}$ be the corresponding indicator variable.

Analyzing Uniform Hashing

Continued...

Number of elements colliding with $\mathbf{x}$: $\ell(\mathbf{x}) = \sum_{\mathbf{y} \in \mathbf{S}} \mathbf{D_y}$.

$$\begin{aligned}
\Rightarrow \mathrm{E}[\ell(\mathbf{x})] &= \sum_{\mathbf{y} \in \mathbf{S}} \mathrm{E}[\mathbf{D_y}] && \text{linearity of expectation} \\
&= \sum_{\mathbf{y} \in \mathbf{S}} \Pr[\mathbf{h(x) = h(y)}] \\
&= \sum_{\mathbf{y} \in \mathbf{S}} \frac{\mathbf{1}}{\mathbf{m}} && \text{since } \mathcal{H} \text{ is a uniform hash family} \\
&= |\mathbf{S}|/\mathbf{m} \\
&\leq \mathbf{1} \quad \text{if } |\mathbf{S}| \leq \mathbf{m}
\end{aligned}$$

Analyzing Uniform Hashing

Question: What is the *expected* time to look up $\mathbf{x}$ in $\mathbf{T}$ using $\mathbf{h}$ assuming chaining used to resolve collisions?

Answer: $\mathbf{O(n/m)}$.

Comments:

1. $\mathbf{O(1)}$ expected time also holds for insertion.
2. Analysis assumes static set $\mathbf{S}$ but holds as long as $\mathbf{S}$ is a set formed with at most $\mathbf{O(m)}$ insertions and deletions.
3. **Worst-case**: look up time can be large! How large?
   $\mathbf{\Omega(\log n / \log \log n)}$
   [Lower bound holds even under stronger assumptions.]

Analyzing Uniform Hashing

Question: What is the *expected* time to look up $\mathbf{x}$ in $\mathbf{T}$ using $\mathbf{h}$ assuming chaining used to resolve collisions?

Answer: $\mathbf{O(n/m)}$.

Comments:

1. $\mathbf{O(1)}$ expected time also holds for insertion.
2. Analysis assumes static set $\mathbf{S}$ but holds as long as $\mathbf{S}$ is a set formed with at most $\mathbf{O(m)}$ insertions and deletions.
3. **Worst-case**: look up time can be large! How large? $\mathbf{\Omega(\log n / \log \log n)}$
   [Lower bound holds even under stronger assumptions.]

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed $\mathbf{S}$ of size $\simeq \mathbf{m}$.

Question: What happens as items are inserted and deleted?

1. If $|\mathbf{S}|$ grows to more than $\mathbf{cm}$ for some constant $\mathbf{c}$ then hash table performance clearly degrades.
2. If $|\mathbf{S}|$ stays around $\simeq \mathbf{m}$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

1. Choose a new table size based on current number of elements in table.
2. Choose a new random hash function and rehash the elements.
3. Discard old table and hash function.

Question: When to rebuild? How expensive?

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed $\mathbf{S}$ of size $\simeq \mathbf{m}$.

Question: What happens as items are inserted and deleted?

1. If $|\mathbf{S}|$ grows to more than $\mathbf{cm}$ for some constant $\mathbf{c}$ then hash table performance clearly degrades.
2. If $|\mathbf{S}|$ stays around $\simeq \mathbf{m}$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

1. Choose a new table size based on current number of elements in table.
2. Choose a new random hash function and rehash the elements.
3. Discard old table and hash function.

Question: When to rebuild? How expensive?

Rebuilding the hash table

1. Start with table size $\mathbf{m}$ where $\mathbf{m}$ is some estimate of $|\mathbf{S}|$ (can be some large constant).
2. If $|\mathbf{S}|$ grows to more than twice current table size, build new hash table (choose a new random hash function) with double the current number of elements. Can also use similar trick if table size falls below quarter the size.
3. If $|\mathbf{S}|$ stays roughly the same but more than $\mathbf{c}|\mathbf{S}|$ operations on table for some chosen constant $\mathbf{c}$ (say $\mathbf{10}$), rebuild.

The **amortize** cost of rebuilding to previously performed operations. Rebuilding ensures $\mathbf{O(1)}$ expected analysis holds even when $\mathbf{S}$ changes. Hence $\mathbf{O(1)}$ expected look up/insert/delete time *dynamic* data dictionary data structure!

Some math required...

Lemma

Let $\mathbf{p}$ *be a prime number,*
$\mathbf{x}$*: an integer number in* $\{\mathbf{1}, \ldots, \mathbf{p} - \mathbf{1}\}$.
$\Longrightarrow$ *There exists a unique* $\mathbf{y}$ *s.t.* $\mathbf{xy} = \mathbf{1} \mod \mathbf{p}$.

In other words: For every element there is a unique inverse.
$\Longrightarrow$ $\mathbb{Z}_{\mathbf{p}} = \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{p} - \mathbf{1}\}$ when working modulo $\mathbf{p}$ is a *field*.

Proof of lemma

Claim

Let $\mathbf{p}$ *be a prime number. For any* $\alpha, \beta, \mathbf{i} \in \{1, \ldots, \mathbf{p}-1\}$ *s.t.* $\alpha \neq \beta$*, we have that* $\alpha \mathbf{i} \neq \beta \mathbf{i} \mod \mathbf{p}$.

Proof.

Assume for the sake of contradiction $\alpha \mathbf{i} = \beta \mathbf{i} \mod \mathbf{p}$. Then

$$\begin{aligned} & \mathbf{i}(\alpha - \beta) = 0 \mod \mathbf{p} \\ \implies & \quad \mathbf{p} \text{ divides } \mathbf{i}(\alpha - \beta) \\ \implies & \quad \mathbf{p} \text{ divides } \alpha - \beta \\ \implies & \quad \alpha - \beta = 0 \\ \implies & \quad \alpha = \beta. \end{aligned}$$

And that is a contradiction. □

Proof of lemma...

Uniqueness.

Lemma

Let $\mathbf{p}$ *be a prime number,*
$\mathbf{x}$*: an integer number in* $\{1, \ldots, p-1\}$.
$\implies$ *There exists a unique* $\mathbf{y}$ *s.t.* $xy = 1 \mod p$.

Proof.

Assume the lemma is false and there are two distinct numbers $y, z \in \{1, \ldots, p-1\}$ such that

$$xy = 1 \mod p \quad \text{and} \quad xz = 1 \mod p.$$

But this contradicts the above claim (set $i = x$, $\alpha = y$ and $\beta = z$). $\square$

Proof of lemma...

Existence

Proof.

By claim, for any $\alpha \in \{1, \ldots, p-1\}$ we have that $\{\alpha * 1 \mod p, \alpha * 2 \mod p, \ldots, \alpha * (p-1) \mod p\} = \{1, 2, \ldots, p-1\}$.

$\implies$ There exists a number $y \in \{1, \ldots, p-1\}$ such that $\alpha y = 1 \mod p$. $\square$

Constructing Universal Hash Families

Parameters: $\mathbf{N} = |\mathcal{U}|$, $\mathbf{m} = |\mathbf{T}|$, $\mathbf{n} = |\mathbf{S}|$

1. Choose a **prime** number $\mathbf{p} \geq \mathbf{N}$. $\mathbb{Z}_\mathbf{p} = \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{p} - \mathbf{1}\}$ is a field.
2. For $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_\mathbf{p}$, $\mathbf{a} \neq \mathbf{0}$, define the hash function $\mathbf{h_{a,b}}$ as $\mathbf{h_{a,b}(x) = ((ax + b) \mod p) \mod m}$.
3. Let $\mathcal{H} = \{\mathbf{h_{a,b}} \mid \mathbf{a}, \mathbf{b} \in \mathbb{Z}_\mathbf{p}, \mathbf{a} \neq \mathbf{0}\}$. Note that $|\mathcal{H}| = \mathbf{p(p - 1)}$.

Theorem

$\mathcal{H}$ is a $\mathbf{2}$-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function ($\mathbf{a}, \mathbf{b}$ have to be stored) and evaluate it.

Constructing Universal Hash Families

Parameters: $\mathbf{N} = |\mathcal{U}|$, $\mathbf{m} = |\mathbf{T}|$, $\mathbf{n} = |\mathbf{S}|$

1. Choose a **prime** number $\mathbf{p} \geq \mathbf{N}$. $\mathbb{Z}_\mathbf{p} = \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{p} - \mathbf{1}\}$ is a field.
2. For $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_\mathbf{p}$, $\mathbf{a} \neq \mathbf{0}$, define the hash function $\mathbf{h_{a,b}}$ as $\mathbf{h_{a,b}(x) = ((ax + b) \mod p) \mod m}$.
3. Let $\mathcal{H} = \{\mathbf{h_{a,b}} \mid \mathbf{a}, \mathbf{b} \in \mathbb{Z}_\mathbf{p}, \mathbf{a} \neq \mathbf{0}\}$. Note that $|\mathcal{H}| = \mathbf{p(p - 1)}$.

Theorem

$\mathcal{H}$ is a $\mathbf{2}$-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function ($\mathbf{a}, \mathbf{b}$ have to be stored) and evaluate it.

Constructing Universal Hash Families

Parameters: $\mathbf{N} = |\mathcal{U}|$, $\mathbf{m} = |\mathbf{T}|$, $\mathbf{n} = |\mathbf{S}|$

1. Choose a **prime** number $\mathbf{p} \geq \mathbf{N}$. $\mathbb{Z}_\mathbf{p} = \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{p} - \mathbf{1}\}$ is a field.
2. For $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_\mathbf{p}$, $\mathbf{a} \neq \mathbf{0}$, define the hash function $\mathbf{h_{a,b}}$ as $\mathbf{h_{a,b}(x) = ((ax + b) \mod p) \mod m}$.
3. Let $\mathcal{H} = \{\mathbf{h_{a,b}} \mid \mathbf{a}, \mathbf{b} \in \mathbb{Z}_\mathbf{p}, \mathbf{a} \neq \mathbf{0}\}$. Note that $|\mathcal{H}| = \mathbf{p(p - 1)}$.

Theorem

$\mathcal{H}$ is a $\mathbf{2}$-universal hash family.

Comments:

1. Hash family is of small size, easy to sample from.
2. Easy to store a hash function ($\mathbf{a}, \mathbf{b}$ have to be stored) and evaluate it.

What the is going on?

$\mathbf{h_{a,b}(x) = ((ax + b) \mod p) \mod m}$

First map $\mathbf{x \neq y}$ to $\mathbf{r = h(x)}$ and $\mathbf{s = h(y)}$.

This is a random uniform mapping (choosing **a** and **b**) – every cell has the same probability to be the target, for fixed **x** and **y**.

What the is going on?

$\mathbf{h_{a,b}(x) = ((ax + b) \mod p) \mod m}$

First map $\mathbf{x \neq y}$ to $\mathbf{r = h(x)}$ and $\mathbf{s = h(y)}$.

This is a random uniform mapping (choosing **a** and **b**) – every cell has the same probability to be the target, for fixed **x** and **y**.

What the is going on?

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m$$

What the is going on?

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m$$

What the is going on?

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m$$

1. First part of mapping maps (x, y) to a random location $(h_{a,b}(x), h_{a,b}(y))$ in the "matrix".
2. $(h_{a,b}(x), h_{a,b}(y))$ is not on main diagonal.
3. All blue locations are "bad" – map by $\mod m$ to a location of collusion.
4. But... at most $1/m$ fraction of allowable locations in the matrix are bad.

Constructing Universal Hash Families

Theorem

$\mathcal{H}$ is a (2)-universal hash family.

Proof.

Fix $\mathbf{x}, \mathbf{y} \in \mathcal{U}$. What is the probability they will collide if $\mathbf{h}$ is picked randomly from $\mathcal{H}$?

1. Let $\mathbf{a}, \mathbf{b}$ be *bad* for $\mathbf{x}, \mathbf{y}$ if $\mathbf{h_{a,b}(x) = h_{a,b}(y)}$.
2. **Claim:** Number of bad pairs is at most $\mathbf{p(p-1)/m}$.
3. Total number of hash functions is $\mathbf{p(p-1)}$ and hence probability of a collision is $\leq \mathbf{1/m}$. □

Constructing Universal Hash Families

Theorem

$\mathcal{H}$ is a (2)-universal hash family.

Proof.

Fix $\mathbf{x}, \mathbf{y} \in \mathcal{U}$. What is the probability they will collide if $\mathbf{h}$ is picked randomly from $\mathcal{H}$?

1. Let $\mathbf{a}, \mathbf{b}$ be *bad* for $\mathbf{x}, \mathbf{y}$ if $\mathbf{h_{a,b}(x) = h_{a,b}(y)}$.
2. **Claim:** Number of bad pairs is at most $\mathbf{p(p-1)/m}$.
3. Total number of hash functions is $\mathbf{p(p-1)}$ and hence probability of a collision is $\leq \mathbf{1/m}$. □

Some Lemmas

Lemma

If $x \neq y$ *then for any* $a, b \in \mathbb{Z}_p$ *such that* $a \neq 0$*, we have*

$$ax + b \mod p \neq ay + b \mod p.$$

Proof.

If $ax + b \mod p = ay + b \mod p$ then $a(x - y) \mod p = 0$ and $a \neq 0$ and $(x - y) \neq 0$. However, a and $(x - y)$ cannot divide p since p is prime and $a < p$ and $(x - y) < p$. $\square$

Some Lemmas

Lemma

If $x \neq y$ *then for each* (r, s) *such that* $r \neq s$ *and* $0 \leq r, s \leq p - 1$ *there is exactly* **one** a, b *such that*

$$ax + b \mod p = r \text{ and } ay + b \mod p = s$$

.

Proof.

Solve the two equations:

$$ax + b = r \mod p \qquad \text{and} \qquad ay + b = s \mod p$$

We get $a = \frac{r-s}{x-y} \mod p$ and $b = r - ax \mod p$. □

Understanding the hashing

Once we fix $\mathbf{a}$ and $\mathbf{b}$, and we are given a value $\mathbf{x}$, we compute the hash value of $\mathbf{x}$ in two stages:

1. **Compute**: $\mathbf{r} \leftarrow (\mathbf{ax} + \mathbf{b}) \mod \mathbf{p}$.
2. **Fold**: $\mathbf{r}' \leftarrow \mathbf{r} \mod \mathbf{m}$

Collision...

Given two distinct values $\mathbf{x}$ and $\mathbf{y}$ they might collide only because of folding.

Lemma

 # not equal pairs of $\mathbb{Z}_\mathbf{p} \times \mathbb{Z}_\mathbf{p}$ *that are folded to the same number is* $\mathbf{p}(\mathbf{p}-1)/\mathbf{m}$.

Folding numbers

Lemma

 # not pairs (r, s) $\mathbb{Z}_p \times \mathbb{Z}_p$ *such that* $r \neq s$ *and* $r \mod m = s \mod m$ *(folded to the same number) is* $p(p-1)/m$.

Proof.

Consider a pair $(r, s) \in \{0, 1, \ldots, p-1\}^2$ s.t. $r \neq s$. Fix r:

1. There are $\lceil p/m \rceil$ values of s that fold into x. That is

$$r \mod m = s \mod m.$$

2. One of them is when $r = s$.
3. $\implies$ # of colliding pairs $(\lceil p/m \rceil - 1)p \leq (p-1)p/m$

$\square$

Proof of Claim

of bad pairs is $p(p-1)/m$

Proof.

Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$.

1. Let $ax + b \mod p = r$ and $ay + b = s \mod p$.
2. Collision if and only if $r = s \mod m$.
3. (Folding error): Number of pairs (r, s) such that $r \neq s$ and $0 \leq r, s \leq p - 1$ and $r = s \mod m$ is $p(p-1)/m$.
4. From previous lemma for each bad pair (a, b) there is a unique pair (r, s) such that $r = s \mod m$. Hence total number of bad pairs is $p(p-1)/m$.

$\square$

Prob of x and y to collide: $\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$.

Proof of Claim

of bad pairs is $p(p-1)/m$

Proof.

Let $a, b \in \mathbb{Z}_p$ such that $a \neq 0$ and $h_{a,b}(x) = h_{a,b}(y)$.

1. Let $ax + b \mod p = r$ and $ay + b = s \mod p$.
2. Collision if and only if $r = s \mod m$.
3. (Folding error): Number of pairs (r, s) such that $r \neq s$ and $0 \leq r, s \leq p - 1$ and $r = s \mod m$ is $p(p-1)/m$.
4. From previous lemma for each bad pair (a, b) there is a unique pair (r, s) such that $r = s \mod m$. Hence total number of bad pairs is $p(p-1)/m$.

$\square$

Prob of x and y to collide: $\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{p(p-1)/m}{p(p-1)} = \frac{1}{m}$.

Perfect Hashing

Question: Can we make look up time $\mathbf{O(1)}$ in worst case?

Yes for static dictionaries but then space usage is $\mathbf{O(m)}$ only in expectation.

Practical Issues

Hashing used typically for integers, vectors, strings etc.

- Universal hashing is defined for integers. To implement for other objects need to map objects in some fashion to integers (via representation)
- Practical methods for various important cases such as vectors, strings are studied extensively. See http://en.wikipedia.org/wiki/Universal_hashing for some pointers.
- Details on Cuckoo hashing and its advantage over chaining http://en.wikipedia.org/wiki/Cuckoo_hashing.
- Recent important paper briding theory and practice of hashing. "The power of simple tabulation hashing" by Mikkel Thorup and Mihai Patrascu, 2011. See http://en.wikipedia.org/wiki/Tabulation_hashing
- Cryptographic hash functions have a different motivation and

Bloom Filters

Hashing:

1. To insert $\mathbf{x}$ in dictionary store $\mathbf{x}$ in table in location $\mathbf{h(x)}$
2. To lookup $\mathbf{y}$ in dictionary check contents of location $\mathbf{h(y)}$

Bloom Filter: tradeoff space for false positives

1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
2. To insert $\mathbf{x}$ in dictionary set *bit* to $\mathbf{1}$ in location $\mathbf{h(x)}$ (initially all bits are set to $\mathbf{0}$)
3. To lookup $\mathbf{y}$ if bit in location $\mathbf{h(y)}$ is $\mathbf{1}$ say yes, else no.

Bloom Filters

Hashing:

1. To insert **x** in dictionary store **x** in table in location **h(x)**
2. To lookup **y** in dictionary check contents of location **h(y)**

Bloom Filter: tradeoff space for false positives

1. Storing items in dictionary expensive in terms of memory, especially if items are unwieldy objects such a long strings, images, etc with *non-uniform* sizes.
2. To insert **x** in dictionary set *bit* to **1** in location **h(x)** (initially all bits are set to **0**)
3. To lookup **y** if bit in location **h(y)** is **1** say yes, else no.

Bloom Filters

Bloom Filter: tradeoff space for false positives

1. To insert $\mathbf{x}$ in dictionary set *bit* to $\mathbf{1}$ in location $\mathbf{h(x)}$ (initially all bits are set to $\mathbf{0}$)
2. To lookup $\mathbf{y}$ if bit in location $\mathbf{h(y)}$ is $\mathbf{1}$ say yes, else no
3. No false negatives but false positives possible due to collisions

Reducing false positives:

1. Pick $\mathbf{k}$ hash functions $\mathbf{h_1, h_2, \ldots, h_k}$ *independently*
2. To insert $\mathbf{x}$ for $\mathbf{1 \leq i \leq k}$ set bit in location $\mathbf{h_i(x)}$ in table $\mathbf{i}$ to $\mathbf{1}$
3. To lookup $\mathbf{y}$ compute $\mathbf{h_i(y)}$ for $\mathbf{1 \leq i \leq k}$ and say yes only if each bit in the corresponding location is $\mathbf{1}$, otherwise say no. If probability of false positive for one hash function is $\alpha < \mathbf{1}$ then with $\mathbf{k}$ independent hash function it is $\alpha^{\mathbf{k}}$.

Bloom Filters

Bloom Filter: tradeoff space for false positives

1. To insert $\mathbf{x}$ in dictionary set *bit* to $\mathbf{1}$ in location $\mathbf{h(x)}$ (initially all bits are set to $\mathbf{0}$)
2. To lookup $\mathbf{y}$ if bit in location $\mathbf{h(y)}$ is $\mathbf{1}$ say yes, else no
3. No false negatives but false positives possible due to collisions

Reducing false positives:

1. Pick $\mathbf{k}$ hash functions $\mathbf{h_1, h_2, \ldots, h_k}$ *independently*
2. To insert $\mathbf{x}$ for $\mathbf{1 \leq i \leq k}$ set bit in location $\mathbf{h_i(x)}$ in table $\mathbf{i}$ to $\mathbf{1}$
3. To lookup $\mathbf{y}$ compute $\mathbf{h_i(y)}$ for $\mathbf{1 \leq i \leq k}$ and say yes only if each bit in the corresponding location is $\mathbf{1}$, otherwise say no. If probability of false positive for one hash function is $\alpha < 1$ then with $\mathbf{k}$ independent hash function it is $\alpha^{\mathbf{k}}$.

Take away points

1. Hashing is a powerful and important technique for dictionaries. Many practical applications.
2. Randomization fundamental to understanding hashing.
3. Good and efficient hashing possible in theory and practice with proper definitions (universal, perfect, etc).
4. Related ideas of creating a compact fingerprint/sketch for objects is very powerful in theory and practice.
5. Many applications in practice.

Notes

Notes

Notes

Notes

Network Flows

Lecture 16

March 20, 2014

How many edges to cut?

For the graph depicted on the right. How many edges have to be cut before there is no path between **s** and **t**:

(A) 1
(B) 2
(C) 3
(D) 4
(E) 5

Everything flows

Panta rei – everything flows (literally).
Heraclitus (535–475 BC)

Part I

Network Flows: Introduction and Setup

Transportation/Road Network

Internet Backbone Network

Common Features of Flow Networks

1. **Network** represented by a (directed) *graph* $\mathbf{G = (V, E)}$.
2. Each edge $\mathbf{e}$ has a **capacity** $\mathbf{c(e) \geq 0}$ that limits amount of *traffic* on $\mathbf{e}$.
3. *Source(s)* of traffic/data.
4. *Sink(s)* of traffic/data.
5. Traffic *flows* from sources to sinks.
6. Traffic is *switched/interchanged* at nodes.

Flow abstract term to indicate stuff (traffic/data/etc) that **flows** from sources to sinks.

Single Source/Single Sink Flows

Simple setting:

1. Single source **s** and single sink **t**.
2. Every other node **v** is an **internal** node.
3. Flow originates at **s** and terminates at **t**.

1. Each edge **e** has a capacity $\mathbf{c(e) \geq 0}$.
2. Sometimes assume:
   Source $\mathbf{s} \in \mathbf{V}$ has no incoming edges, and sink $\mathbf{t} \in \mathbf{V}$ has no outgoing edges.

Assumptions: All capacities are integer, and every vertex has at least one edge incident to it.

Definition of Flow

Two ways to define flows:

1. edge based, or
2. path based.

Essentially equivalent but have different uses.

Edge based definition is more compact.

Edge Based Definition of Flow

Definition

Flow in network $G = (V, E)$, is function $f : E \to \mathbb{R}^{\geq 0}$ s.t.

1. Capacity Constraint: For each edge e, $f(e) \leq c(e)$.

Figure : Flow with value.

Edge Based Definition of Flow

Definition

Flow in network $G = (V, E)$, is function $\mathbf{f : E \to \mathbb{R}^{\geq 0}}$ s.t.

1. Capacity Constraint: For each edge $\mathbf{e}$, $\mathbf{f(e) \leq c(e)}$.

Figure : Flow with value.

Edge Based Definition of Flow

Definition

Flow in network $G = (V, E)$, is function $\mathbf{f : E \to \mathbb{R}^{\geq 0}}$ s.t.

1. Capacity Constraint: For each edge $\mathbf{e}$, $\mathbf{f(e) \leq c(e)}$.
2. Conservation Constraint: For each vertex $\mathbf{v \neq s, t}$.

$$\sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f(e)} = \sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f(e)}$$

Figure : Flow with value.

Edge Based Definition of Flow

Definition

Flow in network $G = (V, E)$, is function $\mathbf{f : E \to \mathbb{R}^{\geq 0}}$ s.t.

Figure : Flow with value.

1. Capacity Constraint: For each edge $\mathbf{e}$, $\mathbf{f(e) \leq c(e)}$.
2. Conservation Constraint: For each vertex $\mathbf{v \neq s, t}$.

$$\sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f(e)} = \sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f(e)}$$

3. Value of flow= (total flow out of source) − (total flow in to source).

Flow...

Conservation of flow law is also known as **Kirchhoff's law**.

More Definitions and Notation

Notation

1. The inflow into a vertex $\mathbf{v}$ is $\mathbf{f^{in}(v) = \sum_{e \text{ into } v} f(e)}$ and the outflow is $\mathbf{f^{out}(v) = \sum_{e \text{ out of } v} f(e)}$
2. For a set of vertices $\mathbf{A}$, $\mathbf{f^{in}(A) = \sum_{e \text{ into } A} f(e)}$. Outflow $\mathbf{f^{out}(A)}$ is defined analogously

Definition

For a network $\mathbf{G = (V, E)}$ with source $\mathbf{s}$, the **value** of flow $\mathbf{f}$ is defined as $\mathbf{v(f) = f^{out}(s) - f^{in}(s)}$.

Value of flow?

In the flow depicted on the right, the value of the flow is.

(A) 6.

(B) 13.

(C) 18.

(D) 28.

(E) 43.

A Path Based Definition of Flow

Intuition: Flow goes from source $\mathbf{s}$ to sink $\mathbf{t}$ along a path.

$\mathcal{P}$: set of all paths from $\mathbf{s}$ to $\mathbf{t}$. $|\mathcal{P}|$ can be **exponential** in $\mathbf{n}$.

Definition (Flow by paths.)

A **flow** in network $G = (V, E)$, is function $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ s.t.

1. Capacity Constraint: For each edge $\mathbf{e}$, total flow on $\mathbf{e}$ is $\leq \mathbf{c(e)}$.

$$\sum_{p \in \mathcal{P} : e \in p} f(p) \leq c(e)$$

2. Conservation Constraint: No need! Automatic.

Value of flow: $\sum_{p \in \mathcal{P}} f(p)$.

A Path Based Definition of Flow

Intuition: Flow goes from source $\mathbf{s}$ to sink $\mathbf{t}$ along a path.

$\mathcal{P}$: set of all paths from $\mathbf{s}$ to $\mathbf{t}$. $|\mathcal{P}|$ can be **exponential** in $\mathbf{n}$.

Definition (Flow by paths.)

A **flow** in network $G = (V, E)$, is function $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ s.t.

1. Capacity Constraint: For each edge $\mathbf{e}$, total flow on $\mathbf{e}$ is $\leq \mathbf{c(e)}$.

$$\sum_{\mathbf{p} \in \mathcal{P} : \mathbf{e} \in \mathbf{p}} \mathbf{f(p)} \leq \mathbf{c(e)}$$

2. Conservation Constraint: No need! Automatic.

Value of flow: $\sum_{\mathbf{p} \in \mathcal{P}} \mathbf{f(p)}$.

A Path Based Definition of Flow

Intuition: Flow goes from source $\mathbf{s}$ to sink $\mathbf{t}$ along a path.

$\mathcal{P}$: set of all paths from $\mathbf{s}$ to $\mathbf{t}$. $|\mathcal{P}|$ can be **exponential** in $\mathbf{n}$.

Definition (Flow by paths.)

A **flow** in network $G = (V, E)$, is function $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ s.t.

1. Capacity Constraint: For each edge $\mathbf{e}$, total flow on $\mathbf{e}$ is $\leq \mathbf{c(e)}$.

$$\sum_{\mathbf{p} \in \mathcal{P} : \mathbf{e} \in \mathbf{p}} \mathbf{f(p)} \leq \mathbf{c(e)}$$

2. Conservation Constraint: No need! Automatic.

Value of flow: $\sum_{\mathbf{p} \in \mathcal{P}} \mathbf{f(p)}$.

A Path Based Definition of Flow

Intuition: Flow goes from source $\mathbf{s}$ to sink $\mathbf{t}$ along a path.

$\mathcal{P}$: set of all paths from $\mathbf{s}$ to $\mathbf{t}$. $|\mathcal{P}|$ can be **exponential** in $\mathbf{n}$.

Definition (Flow by paths.)

A **flow** in network $G = (V, E)$, is function $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ s.t.

1. Capacity Constraint: For each edge $\mathbf{e}$, total flow on $\mathbf{e}$ is $\leq \mathbf{c(e)}$.

$$\sum_{\mathbf{p} \in \mathcal{P} : \mathbf{e} \in \mathbf{p}} \mathbf{f(p)} \leq \mathbf{c(e)}$$

2. Conservation Constraint: No need! Automatic.

Value of flow: $\sum_{\mathbf{p} \in \mathcal{P}} \mathbf{f(p)}$.

A Path Based Definition of Flow

Intuition: Flow goes from source $\mathbf{s}$ to sink $\mathbf{t}$ along a path.

$\mathcal{P}$: set of all paths from $\mathbf{s}$ to $\mathbf{t}$. $|\mathcal{P}|$ can be **exponential** in $\mathbf{n}$.

Definition (Flow by paths.)

A **flow** in network $G = (V, E)$, is function $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ s.t.

1. Capacity Constraint: For each edge $\mathbf{e}$, total flow on $\mathbf{e}$ is $\leq \mathbf{c(e)}$.

$$\sum_{\mathbf{p} \in \mathcal{P} : \mathbf{e} \in \mathbf{p}} \mathbf{f(p)} \leq \mathbf{c(e)}$$

2. Conservation Constraint: No need! Automatic.

Value of flow: $\sum_{\mathbf{p} \in \mathcal{P}} \mathbf{f(p)}$.

Example

$\mathcal{P} = \{p_1, p_2, p_3\}$

$p_1 : s \rightarrow u \rightarrow t$

$p_2 : s \rightarrow u \rightarrow v \rightarrow t$

$p_3 : s \rightarrow v \rightarrow t$

$f(p_1) = 10, f(p_2) = 4, f(p_3) = 6$

Example

$\mathcal{P} = \{p_1, p_2, p_3\}$

$p_1 : s \to u \to t$

$p_2 : s \to u \to v \to t$

$p_3 : s \to v \to t$

$f(p_1) = 10, f(p_2) = 4, f(p_3) = 6$

Path based flow implies edge based flow

Lemma

Given a path based flow $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ *there is an edge based flow* $\mathbf{f'} : \mathbf{E} \rightarrow \mathbb{R}^{\geq 0}$ *of the same value.*

Proof.

For each edge $\mathbf{e}$ define $\mathbf{f'(e)} = \sum_{\mathbf{p:e \in p}} \mathbf{f(p)}$.

Exercise: Verify capacity and conservation constraints for $\mathbf{f'}$.

Exercise: Verify that value of $\mathbf{f}$ and $\mathbf{f'}$ are equal □

Path based flow implies edge based flow

Lemma

Given a path based flow $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ *there is an edge based flow* $\mathbf{f'} : \mathbf{E} \rightarrow \mathbb{R}^{\geq 0}$ *of the same value.*

Proof.

For each edge $\mathbf{e}$ define $\mathbf{f'(e)} = \sum_{\mathbf{p:e \in p}} \mathbf{f(p)}$.
Exercise: Verify capacity and conservation constraints for $\mathbf{f'}$.
Exercise: Verify that value of $\mathbf{f}$ and $\mathbf{f'}$ are equal □

Example

$\mathcal{P} = \{\mathbf{p_1}, \mathbf{p_2}, \mathbf{p_3}\}$

$\mathbf{p_1} : \mathbf{s} \rightarrow \mathbf{u} \rightarrow \mathbf{t}$

$\mathbf{p_2} : \mathbf{s} \rightarrow \mathbf{u} \rightarrow \mathbf{v} \rightarrow \mathbf{t}$

$\mathbf{p_3} : \mathbf{s} \rightarrow \mathbf{v} \rightarrow \mathbf{t}$

$\mathbf{f(p_1) = 10, f(p_2) = 4, f(p_3) = 6}$

Example

$\mathcal{P} = \{p_1, p_2, p_3\}$

$p_1 : s \to u \to t$

$p_2 : s \to u \to v \to t$

$p_3 : s \to v \to t$

$f(p_1) = 10, f(p_2) = 4, f(p_3) = 6$

$f'(s \to u) = 14$

$f'(u \to v) = 4$

$f'(s \to v) = 6$

$f'(u \to t) = 10$

$f'(v \to t) = 10$

Flow Decomposition

Edge based flow to Path based Flow

Lemma

Given an edge based flow $\mathbf{f'} : \mathbf{E} \rightarrow \mathbb{R}^{\geq 0}$*, there is a path based flow* $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ *of same value. Moreover,* $\mathbf{f}$ *assigns non-negative flow to at most* $\mathbf{m}$ *paths where* $|\mathbf{E}| = \mathbf{m}$ *and* $|\mathbf{V}| = \mathbf{n}$*. Given* $\mathbf{f'}$*, the path based flow can be computed in* $\mathbf{O(mn)}$ *time.*

Flow Decomposition

Edge based flow to Path based Flow

Proof Idea.

1. Remove all edges with $\mathbf{f'(e) = 0}$.
2. Find a path $\mathbf{p}$ from $\mathbf{s}$ to $\mathbf{t}$.
3. Assign $\mathbf{f(p)}$ to be $\mathbf{\min_{e \in p} f'(e)}$.
4. Reduce $\mathbf{f'(e)}$ for all $\mathbf{e \in p}$ by $\mathbf{f(p)}$.
5. Repeat until no path from $\mathbf{s}$ to $\mathbf{t}$.
6. In each iteration at least on edge has flow reduced to zero.
7. Hence, at most $\mathbf{m}$ iterations. Can be implemented in $\mathbf{O(m(m+n))}$ time. $\mathbf{O(mn)}$ time requires care. □

Flow Decomposition

Edge based flow to Path based Flow

Proof Idea.

1. Remove all edges with $f'(e) = 0$.
2. Find a path p from s to t.
3. Assign $f(p)$ to be $\min_{e \in p} f'(e)$.
4. Reduce $f'(e)$ for all $e \in p$ by $f(p)$.
5. Repeat until no path from s to t.
6. In each iteration at least on edge has flow reduced to zero.
7. Hence, at most m iterations. Can be implemented in $O(m(m+n))$ time. $O(mn)$ time requires care. □

Example

s
1
2
3
4
5
6
t
10/10
4/5
14/15
0/4
0/4
9/9
1/15
8/8
4/6
14/30
0/15
0/15
9/10
9/10
10/10

Edge vs Path based Definitions of Flow

Edge based flows:

1. **compact** representation, only **m** values to be specified, and
2. need to check flow conservation explicitly at each internal node.

Path flows:

1. in some applications, paths more natural,
2. not compact,
3. no need to check flow conservation constraints.

Equivalence shows that we can go back and forth easily.

The Maximum-Flow Problem

Problem

Input A network **G** with capacity **c** and source **s** and sink **t**.

Goal Find flow of **maximum** value.

Question: Given a flow network, what is an *upper bound* on the maximum flow between source and sink?

The Maximum-Flow Problem

Problem

Input A network **G** with capacity **c** and source **s** and sink **t**.

Goal Find flow of **maximum** value.

Question: Given a flow network, what is an *upper bound* on the maximum flow between source and sink?

Cuts

Definition (s-t cut)

Given a flow network an **s-t cut** is a set of edges $\mathbf{E'} \subset \mathbf{E}$ such that removing $\mathbf{E'}$ *disconnects* $\mathbf{s}$ from $\mathbf{t}$: in other words there is no directed $\mathbf{s} \rightarrow \mathbf{t}$ path in $\mathbf{E} - \mathbf{E'}$.
The **capacity** of a cut $\mathbf{E'}$ is $\mathbf{c(E')} = \sum_{\mathbf{e} \in \mathbf{E'}} \mathbf{c(e)}$.

Caution:

1. Cut may leave $\mathbf{t} \rightarrow \mathbf{s}$ paths!
2. There might be many $\mathbf{s}$-$\mathbf{t}$ cuts.

Cuts

Definition (s-t cut)

Given a flow network an **s-t cut** is a set of edges $\mathbf{E'} \subset \mathbf{E}$ such that removing $\mathbf{E'}$ *disconnects* $\mathbf{s}$ from $\mathbf{t}$: in other words there is no directed $\mathbf{s} \rightarrow \mathbf{t}$ path in $\mathbf{E} - \mathbf{E'}$.
The **capacity** of a cut $\mathbf{E'}$ is $\mathbf{c(E')} = \sum_{\mathbf{e} \in \mathbf{E'}} \mathbf{c(e)}$.

Caution:

1. Cut may leave $\mathbf{t} \rightarrow \mathbf{s}$ paths!
2. There might be many $\mathbf{s}$-$\mathbf{t}$ cuts.

Cuts

Definition (s-t cut)

Given a flow network an **s-t cut** is a set of edges $\mathbf{E'} \subset \mathbf{E}$ such that removing $\mathbf{E'}$ *disconnects* $\mathbf{s}$ from $\mathbf{t}$: in other words there is no directed $\mathbf{s} \rightarrow \mathbf{t}$ path in $\mathbf{E} - \mathbf{E'}$.
The **capacity** of a cut $\mathbf{E'}$ is $\mathbf{c(E')} = \sum_{\mathbf{e} \in \mathbf{E'}} \mathbf{c(e)}$.

Caution:

1. Cut may leave $\mathbf{t} \rightarrow \mathbf{s}$ paths!
2. There might be many **s-t** cuts.

s — t cuts

A death by a thousand cuts

Minimal Cut

Definition (Minimal **s**-**t** cut.)

Given a **s**-**t** flow network $G = (V, E)$, $E' \subseteq E$ is a **minimal cut** if for all $\mathbf{e} \in E'$, if $E' \setminus \{\mathbf{e}\}$ is not a cut.

Observation: given a cut E', can check efficiently whether E' is a minimal cut or not. How?

Minimal Cut

Definition (Minimal **s**-**t** cut.)

Given a **s**-**t** flow network $G = (V, E)$, $E' \subseteq E$ is a **minimal cut** if for all $\mathbf{e} \in E'$, if $E' \setminus \{\mathbf{e}\}$ is not a cut.

Observation: given a cut $\mathbf{E'}$, can check efficiently whether $\mathbf{E'}$ is a minimal cut or not. How?

Is this a minimal cut?

Definition (Minimal s-t cut.)

Given a **s**-**t** flow network $G = (V, E)$ with **n** vertices and **m** edges, $E' \subseteq E$ is a **minimal cut** if for all $\mathbf{e} \in E'$, $E' \setminus \{\mathbf{e}\}$ is not a cut.

Checking if a set $\mathbf{E'}$ forms a minimal **s**-**t** cut can be done in

(A) $O(n + m)$.

(B) $O(n \log n + m)$.

(C) $O((n + m) \log n)$.

(D) $O(nm)$.

(E) $O(nm \log n)$.

(F) You flow, me cut.

Cuts as Vertex Partitions

Let $\mathbf{A} \subset \mathbf{V}$ such that

1. $\mathbf{s} \in \mathbf{A}$, $\mathbf{t} \notin \mathbf{A}$, and
2. $\mathbf{B} = \mathbf{V} \setminus \mathbf{A}$ (hence $\mathbf{t} \in \mathbf{B}$).

The **cut** $(\mathbf{A}, \mathbf{B})$ is the set of edges

$$(\mathbf{A}, \mathbf{B}) = \{(\mathbf{u}, \mathbf{v}) \in \mathbf{E} \mid \mathbf{u} \in \mathbf{A}, \mathbf{v} \in \mathbf{B}\}.$$

Cut $(\mathbf{A}, \mathbf{B})$ is set of edges leaving $\mathbf{A}$.

Claim

$(\mathbf{A}, \mathbf{B})$ *is an* $\mathbf{s}$-$\mathbf{t}$ *cut.*

Proof.

Let $\mathbf{P}$ be any $\mathbf{s} \to \mathbf{t}$ path in $\mathbf{G}$. Since $\mathbf{t}$ is not in $\mathbf{A}$, $\mathbf{P}$ has to leave $\mathbf{A}$ via some edge $(\mathbf{u}, \mathbf{v})$ in $(\mathbf{A}, \mathbf{B})$. □

Cuts as Vertex Partitions

Lemma

Suppose $\mathbf{E'}$ *is an* $\mathbf{s}$-$\mathbf{t}$ *cut. Then there is a cut* $(\mathbf{A}, \mathbf{B})$ *such that* $(\mathbf{A}, \mathbf{B}) \subsetneq \mathbf{E'}$.

Proof.

$\mathbf{E'}$ is an $\mathbf{s}$-$\mathbf{t}$ cut implies no path from $\mathbf{s}$ to $\mathbf{t}$ in $(\mathbf{V}, \mathbf{E} - \mathbf{E'})$.

1. Let $\mathbf{A}$ be set of all nodes reachable by $\mathbf{s}$ in $(\mathbf{V}, \mathbf{E} - \mathbf{E'})$.
2. Since $\mathbf{E'}$ is a cut, $\mathbf{t} \notin \mathbf{A}$.
3. $(\mathbf{A}, \mathbf{B}) \subseteq \mathbf{E'}$. Why?

□

Corollary

Every minimal $\mathbf{s}$-$\mathbf{t}$ *cut* $\mathbf{E'}$ *is a cut of the form* $(\mathbf{A}, \mathbf{B})$.

Cuts as Vertex Partitions

Lemma

Suppose $\mathbf{E'}$ *is an* $\mathbf{s}$-$\mathbf{t}$ *cut. Then there is a cut* $(\mathbf{A}, \mathbf{B})$ *such that* $(\mathbf{A}, \mathbf{B}) \subseteq \mathbf{E'}$.

Proof.

$\mathbf{E'}$ is an $\mathbf{s}$-$\mathbf{t}$ cut implies no path from $\mathbf{s}$ to $\mathbf{t}$ in $(\mathbf{V}, \mathbf{E} - \mathbf{E'})$.

1. Let $\mathbf{A}$ be set of all nodes reachable by $\mathbf{s}$ in $(\mathbf{V}, \mathbf{E} - \mathbf{E'})$.
2. Since $\mathbf{E'}$ is a cut, $\mathbf{t} \notin \mathbf{A}$.
3. $(\mathbf{A}, \mathbf{B}) \subseteq \mathbf{E'}$. Why?

□

Corollary

Every minimal $\mathbf{s}$-$\mathbf{t}$ *cut* $\mathbf{E'}$ *is a cut of the form* $(\mathbf{A}, \mathbf{B})$.

Cuts as Vertex Partitions

Lemma

Suppose $\mathbf{E'}$ *is an* $\mathbf{s}$-$\mathbf{t}$ *cut. Then there is a cut* $(\mathbf{A}, \mathbf{B})$ *such that* $(\mathbf{A}, \mathbf{B}) \subsetneq \mathbf{E'}$.

Proof.

$\mathbf{E'}$ is an $\mathbf{s}$-$\mathbf{t}$ cut implies no path from $\mathbf{s}$ to $\mathbf{t}$ in $(\mathbf{V}, \mathbf{E} - \mathbf{E'})$.

1. Let $\mathbf{A}$ be set of all nodes reachable by $\mathbf{s}$ in $(\mathbf{V}, \mathbf{E} - \mathbf{E'})$.
2. Since $\mathbf{E'}$ is a cut, $\mathbf{t} \notin \mathbf{A}$.
3. $(\mathbf{A}, \mathbf{B}) \subsetneq \mathbf{E'}$. Why?If some edge $(\mathbf{u}, \mathbf{v}) \in (\mathbf{A}, \mathbf{B})$ is not in $\mathbf{E'}$ then $\mathbf{v}$ will be reachable by $\mathbf{s}$ and should be in $\mathbf{A}$, hence a contradiction. □

Corollary

Every minimal $\mathbf{s}$-$\mathbf{t}$ *cut* $\mathbf{E'}$ *is a cut of the form* $(\mathbf{A}, \mathbf{B})$.

Cuts as Vertex Partitions

Lemma

Suppose $\mathbf{E'}$ *is an* $\mathbf{s}$-$\mathbf{t}$ *cut. Then there is a cut* $(\mathbf{A}, \mathbf{B})$ *such that* $(\mathbf{A}, \mathbf{B}) \subseteq \mathbf{E'}$.

Proof.

$\mathbf{E'}$ is an $\mathbf{s}$-$\mathbf{t}$ cut implies no path from $\mathbf{s}$ to $\mathbf{t}$ in $(\mathbf{V}, \mathbf{E} - \mathbf{E'})$.

1. Let $\mathbf{A}$ be set of all nodes reachable by $\mathbf{s}$ in $(\mathbf{V}, \mathbf{E} - \mathbf{E'})$.
2. Since $\mathbf{E'}$ is a cut, $\mathbf{t} \notin \mathbf{A}$.
3. $(\mathbf{A}, \mathbf{B}) \subseteq \mathbf{E'}$. Why?If some edge $(\mathbf{u}, \mathbf{v}) \in (\mathbf{A}, \mathbf{B})$ is not in $\mathbf{E'}$ then $\mathbf{v}$ will be reachable by $\mathbf{s}$ and should be in $\mathbf{A}$, hence a contradiction. $\square$

Corollary

Every minimal $\mathbf{s}$-$\mathbf{t}$ *cut* $\mathbf{E'}$ *is a cut of the form* $(\mathbf{A}, \mathbf{B})$.

Minimum Cut

Definition

Given a flow network an **s-t minimum** cut is a cut **E′** of smallest capacity amongst all **s-t** cuts.

The minimum cut in the network flow depicted is:

- **(A)** 10
- **(B)** 18
- **(C)** 28
- **(D)** 30
- **(E)** 48.
- **(F)** No minimum cut, no cry.

Minimum Cut

Definition

Given a flow network an **s-t minimum** cut is a cut **E′** of smallest capacity amongst all **s-t** cuts.

Observation: exponential number of **s-t** cuts and no "easy" algorithm to find a minimum cut.

Minimum Cut

Definition

Given a flow network an **s-t minimum** cut is a cut **E′** of smallest capacity amongst all **s-t** cuts.

Observation: exponential number of **s-t** cuts and no "easy" algorithm to find a minimum cut.

The Minimum-Cut Problem

Problem

Input A flow network **G**

Goal Find the capacity of a *minimum* **s**-**t** cut

Flows and Cuts

Lemma

For any **s-t** *cut* $\mathbf{E'}$, **maximum s-t** *flow* $\leq$ *capacity of* $\mathbf{E'}$.

Proof.

Formal proof easier with path based definition of flow.
Suppose $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ is a max-flow.

Every path $\mathbf{p} \in \mathcal{P}$ contains an edge $\mathbf{e} \in \mathbf{E'}$. Why?
Assign each path $\mathbf{p} \in \mathcal{P}$ to exactly one edge $\mathbf{e} \in \mathbf{E'}$.
Let $\mathcal{P}_\mathbf{e}$ be paths assigned to $\mathbf{e} \in \mathbf{E'}$. Then

$$\mathbf{v(f)} = \sum_{\mathbf{p} \in \mathcal{P}} \mathbf{f(p)} = \sum_{\mathbf{e} \in \mathbf{E'}} \sum_{\mathbf{p} \in \mathcal{P}_\mathbf{e}} \mathbf{f(p)} \leq \sum_{\mathbf{e} \in \mathbf{E'}} \mathbf{c(e)}.$$

□

Flows and Cuts

Lemma

For any **s-t** *cut* $\mathbf{E'}$, **maximum s-t** *flow* $\leq$ *capacity of* $\mathbf{E'}$.

Proof.

Formal proof easier with path based definition of flow.
Suppose $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ is a max-flow.

Every path $\mathbf{p} \in \mathcal{P}$ contains an edge $\mathbf{e} \in \mathbf{E'}$. Why?
Assign each path $\mathbf{p} \in \mathcal{P}$ to exactly one edge $\mathbf{e} \in \mathbf{E'}$.
Let $\mathcal{P}_\mathbf{e}$ be paths assigned to $\mathbf{e} \in \mathbf{E'}$. Then

$$\mathbf{v(f)} = \sum_{\mathbf{p} \in \mathcal{P}} \mathbf{f(p)} = \sum_{\mathbf{e} \in \mathbf{E'}} \sum_{\mathbf{p} \in \mathcal{P}_\mathbf{e}} \mathbf{f(p)} \leq \sum_{\mathbf{e} \in \mathbf{E'}} \mathbf{c(e)}.$$

□

Flows and Cuts

Lemma

For any **s-t** *cut* $\mathbf{E'}$, **maximum s-t** *flow* $\leq$ *capacity of* $\mathbf{E'}$.

Proof.

Formal proof easier with path based definition of flow.
Suppose $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ is a max-flow.

Every path $\mathbf{p} \in \mathcal{P}$ contains an edge $\mathbf{e} \in \mathbf{E'}$. Why?
Assign each path $\mathbf{p} \in \mathcal{P}$ to exactly one edge $\mathbf{e} \in \mathbf{E'}$.
Let $\mathcal{P}_\mathbf{e}$ be paths assigned to $\mathbf{e} \in \mathbf{E'}$. Then

$$v(f) = \sum_{p \in \mathcal{P}} f(p) = \sum_{e \in E'} \sum_{p \in \mathcal{P}_e} f(p) \leq \sum_{e \in E'} c(e).$$

□

Flows and Cuts

Lemma

For any **s-t** *cut* $\mathbf{E'}$, **maximum s-t** *flow* $\leq$ *capacity of* $\mathbf{E'}$.

Proof.

Formal proof easier with path based definition of flow.
Suppose $\mathbf{f} : \mathcal{P} \rightarrow \mathbb{R}^{\geq 0}$ is a max-flow.

Every path $\mathbf{p} \in \mathcal{P}$ contains an edge $\mathbf{e} \in \mathbf{E'}$. Why?
Assign each path $\mathbf{p} \in \mathcal{P}$ to exactly one edge $\mathbf{e} \in \mathbf{E'}$.
Let $\mathcal{P}_\mathbf{e}$ be paths assigned to $\mathbf{e} \in \mathbf{E'}$. Then

$$\mathbf{v(f)} = \sum_{\mathbf{p} \in \mathcal{P}} \mathbf{f(p)} = \sum_{\mathbf{e} \in \mathbf{E'}} \sum_{\mathbf{p} \in \mathcal{P}_\mathbf{e}} \mathbf{f(p)} \leq \sum_{\mathbf{e} \in \mathbf{E'}} \mathbf{c(e)}.$$

□

Flows and Cuts

Lemma

For any **s**-**t** *cut* $\mathbf{E}'$, **maximum** **s**-**t** *flow* $\leq$ *capacity of* $\mathbf{E}'$.

Corollary

Maximum **s**-**t** *flow* $\leq$ *minimum* **s**-**t** *cut.*

Max-Flow Min-Cut Theorem

Theorem

In any flow network the maximum **s**-**t** *flow is equal to the minimum* **s**-**t** *cut.*

Can compute minimum-cut from maximum flow and vice-versa!
Proof coming shortly.
Many applications:

1. optimization
2. graph theory
3. combinatorics

Max-Flow Min-Cut Theorem

Theorem

In any flow network the maximum **s**-**t** *flow is equal to the minimum* **s**-**t** *cut.*

Can compute minimum-cut from maximum flow and vice-versa!

Proof coming shortly.

Many applications:

1. optimization
2. graph theory
3. combinatorics

Max-Flow Min-Cut Theorem

Theorem

In any flow network the maximum **s**-**t** *flow is equal to the minimum* **s**-**t** *cut.*

Can compute minimum-cut from maximum flow and vice-versa!
Proof coming shortly.
Many applications:

1. optimization
2. graph theory
3. combinatorics

The Maximum-Flow Problem

Problem

Input A network $\mathbf{G}$ with capacity $\mathbf{c}$ and source $\mathbf{s}$ and sink $\mathbf{t}$.

Goal Find flow of **maximum** value from $\mathbf{s}$ to $\mathbf{t}$.

Exercise: Given $\mathbf{G}, \mathbf{s}, \mathbf{t}$ as above, show that one can remove all edges into $\mathbf{s}$ and all edges out of $\mathbf{t}$ without affecting the flow value between $\mathbf{s}$ and $\mathbf{t}$.

The Maximum-Flow Problem

Problem

Input A network $\mathbf{G}$ with capacity $\mathbf{c}$ and source $\mathbf{s}$ and sink $\mathbf{t}$.

Goal Find flow of **maximum** value from $\mathbf{s}$ to $\mathbf{t}$.

Exercise: Given $\mathbf{G}, \mathbf{s}, \mathbf{t}$ as above, show that one can remove all edges into $\mathbf{s}$ and all edges out of $\mathbf{t}$ without affecting the flow value between $\mathbf{s}$ and $\mathbf{t}$.

Notes

Notes

Notes

Notes

Network Flow Algorithms

Lecture 17

April 1, 2014

Part I

Algorithm(s) for Maximum Flow

Max-flow and min-cut?

Given a network G with capacities on the edges, and vertices $\mathbf{s}$ and $\mathbf{t}$, consider the maximum flow $\mathbf{f}$ between $\mathbf{s}$ and $\mathbf{t}$, and the minimum cut $(\mathbf{S}, \mathbf{T})$ between $\mathbf{s}$ and $\mathbf{t}$. Then, we have that

(A) $\mathbf{v(f) < c(S, T)}$.

(B) $\mathbf{v(f) \leq c(S, T)}$.

(C) $\mathbf{v(f) > c(S, T)}$.

(D) $\mathbf{v(f) \geq c(S, T)}$.

(E) $\mathbf{v(f) = c(S, T)}$.

Greedy Approach

1. Begin with $\mathbf{f(e) = 0}$ for each edge.
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$.
3. **Augment** flow along this path.
4. Repeat augmentation for as long as possible.

Greedy Approach

1. Begin with $\mathbf{f(e) = 0}$ for each edge.
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$.
3. **Augment** flow along this path.
4. Repeat augmentation for as long as possible.

Greedy Approach

1. Begin with $\mathbf{f(e) = 0}$ for each edge.
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$.
3. **Augment** flow along this path.
4. Repeat augmentation for as long as possible.

Greedy Approach

1. Begin with $\mathbf{f(e) = 0}$ for each edge.
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$.
3. **Augment** flow along this path.
4. Repeat augmentation for as long as possible.

Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $\mathbf{f(e) = 0}$ for each edge
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow!
Need to "push-back" flow along edge $\mathbf{(u, v)}$.

Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $\mathbf{f(e) = 0}$ for each edge
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow!
Need to "push-back" flow along edge $\mathbf{(u, v)}$.

Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $\mathbf{f(e) = 0}$ for each edge
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow!

Need to "push-back" flow along edge $\mathbf{(u, v)}$.

Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $\mathbf{f(e) = 0}$ for each edge
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow!

Need to "push-back" flow along edge $\mathbf{(u, v)}$.

Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $\mathbf{f(e) = 0}$ for each edge
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow!
Need to "push-back" flow along edge $\mathbf{(u, v)}$.

Greedy Approach: Issues

Issues = What is this nonsense?

1. Begin with $\mathbf{f(e) = 0}$ for each edge
2. Find a **s-t** path **P** with $\mathbf{f(e) < c(e)}$ for every edge $\mathbf{e \in P}$
3. Augment flow along this path
4. Repeat augmentation for as long as possible.

Greedy can get stuck in sub-optimal flow!
Need to "push-back" flow along edge $\mathbf{(u, v)}$.

Residual Graph

The "leftover" graph

Definition

For a network $\mathbf{G = (V, E)}$ and flow $\mathbf{f}$, the **residual graph** $\mathbf{G_f = (V', E')}$ of $\mathbf{G}$ with respect to $\mathbf{f}$ is

1. $\mathbf{V' = V}$,
2. **Forward Edges**: For each edge $\mathbf{e \in E}$ with $\mathbf{f(e) < c(e)}$, we add $\mathbf{e \in E'}$ with capacity $\mathbf{c(e) - f(e)}$.
3. **Backward Edges**: For each edge $\mathbf{e = (u, v) \in E}$ with $\mathbf{f(e) > 0}$, we add $\mathbf{(v, u) \in E'}$ with capacity $\mathbf{f(e)}$.

Residual Graph Example

Figure : Flow on edges is indicated in red

Figure : Residual Graph

Residual graph has...

Given a network with $\mathbf{n}$ vertices and $\mathbf{m}$ edges, and a valid flow $\mathbf{f}$ in it, the residual network $\mathbf{G_f}$, has

(A) $\mathbf{m}$ edges.

(B) $\leq \mathbf{2m}$ edges.

(C) $\leq \mathbf{2m + n}$ edges.

(D) $\mathbf{4m + 2n}$ edges.

(E) $\mathbf{nm}$ edges.

(F) just the right number of edges - not too many, not too few.

Residual Graph Property

Observation: Residual graph captures the "residual" problem exactly.

Lemma

Let $\mathbf{f}$ be a flow in $\mathbf{G}$ and $\mathbf{G_f}$ be the residual graph. If $\mathbf{f'}$ is a flow in $\mathbf{G_f}$ then $\mathbf{f} + \mathbf{f'}$ is a flow in $\mathbf{G}$ of value $\mathbf{v(f)} + \mathbf{v(f')}$.

Lemma

Let $\mathbf{f}$ and $\mathbf{f'}$ be two flows in $\mathbf{G}$ with $\mathbf{v(f')} \geq \mathbf{v(f)}$. Then there is a flow $\mathbf{f''}$ of value $\mathbf{v(f')} - \mathbf{v(f)}$ in $\mathbf{G_f}$.

Definition of + and - for flows is intuitive and the above lemmas are easy in some sense but a bit messy to formally prove.

Residual Graph Property

Observation: Residual graph captures the "residual" problem exactly.

Lemma

Let $\mathbf{f}$ *be a flow in* $\mathbf{G}$ *and* $\mathbf{G_f}$ *be the residual graph. If* $\mathbf{f'}$ *is a flow in* $\mathbf{G_f}$ *then* $\mathbf{f} + \mathbf{f'}$ *is a flow in* $\mathbf{G}$ *of value* $\mathbf{v(f) + v(f')}$.

Lemma

Let $\mathbf{f}$ *and* $\mathbf{f'}$ *be two flows in* $\mathbf{G}$ *with* $\mathbf{v(f') \geq v(f)}$. *Then there is a flow* $\mathbf{f''}$ *of value* $\mathbf{v(f') - v(f)}$ *in* $\mathbf{G_f}$.

Definition of + and - for flows is intuitive and the above lemmas are easy in some sense but a bit messy to formally prove.

Residual Graph Property

Observation: Residual graph captures the "residual" problem exactly.

Lemma

Let $\mathbf{f}$ *be a flow in* $\mathbf{G}$ *and* $\mathbf{G_f}$ *be the residual graph. If* $\mathbf{f'}$ *is a flow in* $\mathbf{G_f}$ *then* $\mathbf{f} + \mathbf{f'}$ *is a flow in* $\mathbf{G}$ *of value* $\mathbf{v(f)} + \mathbf{v(f')}$.

Lemma

Let $\mathbf{f}$ *and* $\mathbf{f'}$ *be two flows in* $\mathbf{G}$ *with* $\mathbf{v(f')} \geq \mathbf{v(f)}$. *Then there is a flow* $\mathbf{f''}$ *of value* $\mathbf{v(f')} - \mathbf{v(f)}$ *in* $\mathbf{G_f}$.

Definition of + and - for flows is intuitive and the above lemmas are easy in some sense but a bit messy to formally prove.

Residual Graph Property

Observation: Residual graph captures the "residual" problem exactly.

Lemma

Let $\mathbf{f}$ *be a flow in* $\mathbf{G}$ *and* $\mathbf{G_f}$ *be the residual graph. If* $\mathbf{f'}$ *is a flow in* $\mathbf{G_f}$ *then* $\mathbf{f} + \mathbf{f'}$ *is a flow in* $\mathbf{G}$ *of value* $\mathbf{v(f)} + \mathbf{v(f')}$.

Lemma

Let $\mathbf{f}$ *and* $\mathbf{f'}$ *be two flows in* $\mathbf{G}$ *with* $\mathbf{v(f')} \geq \mathbf{v(f)}$. *Then there is a flow* $\mathbf{f''}$ *of value* $\mathbf{v(f')} - \mathbf{v(f)}$ *in* $\mathbf{G_f}$.

Definition of $+$ and - for flows is intuitive and the above lemmas are easy in some sense but a bit messy to formally prove.

Residual Graph Property: Implication

Recursive algorithm for finding a maximum flow:

```
MaxFlow(G, s, t):
    if the flow from s to t is 0 then
        return 0
    Find any flow f with v(f) > 0 in G
    Recursively compute a maximum flow f′ in G_f
    Output the flow f + f′
```

Iterative algorithm for finding a maximum flow:

```
MaxFlow(G, s, t):
    Start with flow f that is 0 on all edges
    while there is a flow f′ in G_f with v(f′) > 0 do
        f = f + f′
        Update G_f

    Output f
```

Residual Graph Property: Implication

Recursive algorithm for finding a maximum flow:

```
MaxFlow(G, s, t):
    if the flow from s to t is 0 then
        return 0
    Find any flow f with v(f) > 0 in G
    Recursively compute a maximum flow f' in G_f
    Output the flow f + f'
```

Iterative algorithm for finding a maximum flow:

```
MaxFlow(G, s, t):
    Start with flow f that is 0 on all edges
    while there is a flow f' in G_f with v(f') > 0 do
        f = f + f'
        Update G_f

    Output f
```

Ford-Fulkerson Algorithm

```
algFordFulkerson
    for every edge e, f(e) = 0
    G_f is residual graph of G with respect to f
    while G_f has a simple s-t path do
        let P be simple s-t path in G_f
        f = augment(f, P)
        Construct new residual graph G_f.
```

```
augment(f,P)
    let b be bottleneck capacity,
        i.e., min capacity of edges in P (in G_f)
    for each edge (u,v) in P do
        if e = (u,v) is a forward edge then
            f(e) = f(e) + b
        else (* (u,v) is a backward edge *)
            let e = (v,u) (* (v,u) is in G *)
            f(e) = f(e) - b
    return f
```

Ford-Fulkerson Algorithm

```
algFordFulkerson
    for every edge e, f(e) = 0
    G_f is residual graph of G with respect to f
    while G_f has a simple s-t path do
        let P be simple s-t path in G_f
        f = augment(f, P)
        Construct new residual graph G_f.
```

```
augment(f,P)
    let b be bottleneck capacity,
        i.e., min capacity of edges in P (in G_f)
    for each edge (u,v) in P do
        if e = (u,v) is a forward edge then
            f(e) = f(e) + b
        else (* (u,v) is a backward edge *)
            let e = (v,u) (* (v,u) is in G *)
            f(e) = f(e) − b
    return f
```

Example

Example continued

Example continued

Example continued

Properties about Augmentation: Flow

Lemma

If $\mathbf{f}$ *is a flow and* $\mathbf{P}$ *is a simple* $\mathbf{s}$-$\mathbf{t}$ *path in* $\mathbf{G_f}$*, then* $\mathbf{f'} = \texttt{augment}(\mathbf{f}, \mathbf{P})$ *is also a flow.*

Proof.

Verify that $\mathbf{f'}$ is a flow. Let $\mathbf{b}$ be augmentation amount.

1. Capacity constraint: If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a forward edge then $\mathbf{f'(e) = f(e) + b}$ and $\mathbf{b \leq c(e) - f(e)}$. If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a backward edge, then letting $\mathbf{e = (v, u)}$, $\mathbf{f'(e) = f(e) - b}$ and $\mathbf{b \leq f(e)}$. Both cases $\mathbf{0 \leq f'(e) \leq c(e)}$.
2. Conservation constraint: Let $\mathbf{v}$ be an internal node. Let $\mathbf{e_1, e_2}$ be edges of $\mathbf{P}$ incident to $\mathbf{v}$. Four cases based on whether $\mathbf{e_1, e_2}$ are forward or backward edges. Check cases (see fig next slide). □

Properties about Augmentation: Flow

Lemma

If $\mathbf{f}$ *is a flow and* $\mathbf{P}$ *is a simple* $\mathbf{s}$-$\mathbf{t}$ *path in* $\mathbf{G_f}$*, then* $\mathbf{f'} = \texttt{augment}(\mathbf{f}, \mathbf{P})$ *is also a flow.*

Proof.

Verify that $\mathbf{f'}$ is a flow. Let $\mathbf{b}$ be augmentation amount.

1. Capacity constraint: If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a forward edge then $\mathbf{f'(e)} = \mathbf{f(e)} + \mathbf{b}$ and $\mathbf{b} \leq \mathbf{c(e)} - \mathbf{f(e)}$. If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a backward edge, then letting $\mathbf{e} = (\mathbf{v}, \mathbf{u})$, $\mathbf{f'(e)} = \mathbf{f(e)} - \mathbf{b}$ and $\mathbf{b} \leq \mathbf{f(e)}$. Both cases $\mathbf{0} \leq \mathbf{f'(e)} \leq \mathbf{c(e)}$.
2. Conservation constraint: Let $\mathbf{v}$ be an internal node. Let $\mathbf{e_1}, \mathbf{e_2}$ be edges of $\mathbf{P}$ incident to $\mathbf{v}$. Four cases based on whether $\mathbf{e_1}, \mathbf{e_2}$ are forward or backward edges. Check cases (see fig next slide). □

Properties about Augmentation: Flow

Lemma

If $\mathbf{f}$ *is a flow and* $\mathbf{P}$ *is a simple* $\mathbf{s}$-$\mathbf{t}$ *path in* $\mathbf{G_f}$*, then* $\mathbf{f'} = \texttt{augment}(\mathbf{f}, \mathbf{P})$ *is also a flow.*

Proof.

Verify that $\mathbf{f'}$ is a flow. Let $\mathbf{b}$ be augmentation amount.

1. Capacity constraint: If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a forward edge then $\mathbf{f'(e) = f(e) + b}$ and $\mathbf{b \leq c(e) - f(e)}$. If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a backward edge, then letting $\mathbf{e = (v, u)}$, $\mathbf{f'(e) = f(e) - b}$ and $\mathbf{b \leq f(e)}$. Both cases $\mathbf{0 \leq f'(e) \leq c(e)}$.
2. Conservation constraint: Let $\mathbf{v}$ be an internal node. Let $\mathbf{e_1, e_2}$ be edges of $\mathbf{P}$ incident to $\mathbf{v}$. Four cases based on whether $\mathbf{e_1, e_2}$ are forward or backward edges. Check cases (see fig next slide). □

Properties about Augmentation: Flow

Lemma

If $\mathbf{f}$ *is a flow and* $\mathbf{P}$ *is a simple* $\mathbf{s}$-$\mathbf{t}$ *path in* $\mathbf{G_f}$*, then* $\mathbf{f'} = \texttt{augment}(\mathbf{f}, \mathbf{P})$ *is also a flow.*

Proof.

Verify that $\mathbf{f'}$ is a flow. Let $\mathbf{b}$ be augmentation amount.

1. Capacity constraint: If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a forward edge then $\mathbf{f'(e) = f(e) + b}$ and $\mathbf{b \leq c(e) - f(e)}$. If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a backward edge, then letting $\mathbf{e = (v, u)}$, $\mathbf{f'(e) = f(e) - b}$ and $\mathbf{b \leq f(e)}$. Both cases $\mathbf{0 \leq f'(e) \leq c(e)}$.
2. Conservation constraint: Let $\mathbf{v}$ be an internal node. Let $\mathbf{e_1, e_2}$ be edges of $\mathbf{P}$ incident to $\mathbf{v}$. Four cases based on whether $\mathbf{e_1, e_2}$ are forward or backward edges. Check cases (see fig next slide). □

Properties about Augmentation: Flow

Lemma

If $\mathbf{f}$ *is a flow and* $\mathbf{P}$ *is a simple* $\mathbf{s}$-$\mathbf{t}$ *path in* $\mathbf{G_f}$*, then* $\mathbf{f'} = \texttt{augment}(\mathbf{f}, \mathbf{P})$ *is also a flow.*

Proof.

Verify that $\mathbf{f'}$ is a flow. Let $\mathbf{b}$ be augmentation amount.

1. Capacity constraint: If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a forward edge then $\mathbf{f'(e) = f(e) + b}$ and $\mathbf{b \leq c(e) - f(e)}$. If $(\mathbf{u}, \mathbf{v}) \in \mathbf{P}$ is a backward edge, then letting $\mathbf{e = (v, u)}$, $\mathbf{f'(e) = f(e) - b}$ and $\mathbf{b \leq f(e)}$. Both cases $\mathbf{0 \leq f'(e) \leq c(e)}$.
2. Conservation constraint: Let $\mathbf{v}$ be an internal node. Let $\mathbf{e_1, e_2}$ be edges of $\mathbf{P}$ incident to $\mathbf{v}$. Four cases based on whether $\mathbf{e_1, e_2}$ are forward or backward edges. Check cases (see fig next slide). □

Properties of Augmentation

Conservation Constraint

Figure : Augmenting path **P** in $\mathbf{G_f}$ and corresponding change of flow in **G**. Red edges are backward edges.

Rational, integer or real?

Consider a network flow instance where all the numbers are integers. **algFordFulkerson** on this network outputs a flow such that its value is

(A) Since the algorithm runs on a RAM machine, and it can perform any arithmetic operation, the output is a real number.

(B) The algorithm does only subtract, add, divide and multiply operations. Thus the output is a rational number.

(C) The algorithm does only subtract and add operations on numbers. Thus the output is an integer number.

(D) **algFordFulkerson** does not necessarily terminates, so the question is ill defined.

(E) If the capacities are negative, the algorithm might output $+\infty$ (which is not an integer, rational or real number).

Properties of Augmentation

Integer Flow

Lemma

At every stage of the Ford-Fulkerson algorithm, the flow values on the edges (i.e., $\mathbf{f(e)}$*, for all edges* $\mathbf{e}$*) and the residual capacities in* $\mathbf{G_f}$ *are integers.*

Proof.

Initial flow and residual capacities are integers. Suppose lemma holds for $\mathbf{j}$ iterations. Then in $\mathbf{(j+1)}$st iteration, minimum capacity edge $\mathbf{b}$ is an integer, and so flow after augmentation is an integer. □

Progress in Ford-Fulkerson

Proposition

Let $\mathbf{f}$ *be a flow and* $\mathbf{f'}$ *be flow after one augmentation. Then* $\mathbf{v(f)} < \mathbf{v(f')}$.

Proof.

Let $\mathbf{P}$ be an augmenting path, i.e., $\mathbf{P}$ is a simple $\mathbf{s}$-$\mathbf{t}$ path in residual graph. We have the following.

1. First edge $\mathbf{e}$ in $\mathbf{P}$ must leave $\mathbf{s}$.
2. Original network $\mathbf{G}$ has no incoming edges to $\mathbf{s}$; hence $\mathbf{e}$ is a forward edge.
3. $\mathbf{P}$ is simple and so never returns to $\mathbf{s}$.
4. Thus, value of flow increases by the flow on edge $\mathbf{e}$. □

Termination proof for integral flow

Theorem

Let $\mathbf{C}$ *be the minimum cut value; in particular* $\mathbf{C} \leq \sum_{\mathbf{e}\ \textit{out of}\ \mathbf{s}} \mathbf{c(e)}$. *Ford-Fulkerson algorithm terminates after finding at most* $\mathbf{C}$ *augmenting paths.*

Proof.

The value of the flow increases by at least $\mathbf{1}$ after each augmentation. Maximum value of flow is at most $\mathbf{C}$. □

Running time

1. Number of iterations $\leq \mathbf{C}$.
2. Number of edges in $\mathbf{G_f} \leq \mathbf{2m}$.
3. Time to find augmenting path is $\mathbf{O(n+m)}$.
4. Running time is $\mathbf{O(C(n+m))}$ (or $\mathbf{O(mC)}$).

Efficiency of Ford-Fulkerson

Running time = $\mathbf{O(mC)}$ is not polynomial. Can the running time be as $\mathbf{\Omega(mC)}$ or is our analysis weak?

Ford-Fulkerson can take $\mathbf{\Omega(C)}$ iterations.

Efficiency of Ford-Fulkerson

Running time = $\mathbf{O(mC)}$ is not polynomial. Can the running time be as $\mathbf{\Omega(mC)}$ or is our analysis weak?

Ford-Fulkerson can take $\mathbf{\Omega(C)}$ iterations.

Efficiency of Ford-Fulkerson

Running time = $\mathbf{O(mC)}$ is not polynomial. Can the running time be as $\mathbf{\Omega(mC)}$ or is our analysis weak?

Ford-Fulkerson can take $\mathbf{\Omega(C)}$ iterations.

Correctness of Ford-Fulkerson

Why the augmenting path approach works

Question: When the algorithm terminates, is the flow computed the maximum **s**-**t** flow?

Proof idea: show a cut of value equal to the flow. Also shows that maximum flow is equal to minimum cut!

Correctness of Ford-Fulkerson

Why the augmenting path approach works

Question: When the algorithm terminates, is the flow computed the maximum **s**-**t** flow?

Proof idea: show a cut of value equal to the flow. Also shows that maximum flow is equal to minimum cut!

Recalling Cuts

Definition

Given a flow network an **s-t cut** is a set of edges $E' \subset E$ such that removing E' *disconnects* s from t: in other words there is no directed $s \to t$ path in $E - E'$. **Capacity** of cut E' is $\sum_{e \in E'} c(e)$.

Let $A \subset V$ such that

1. $s \in A$, $t \notin A$, and
2. $B = V \setminus -A$ and hence $t \in B$.

Define $(A, B) = \{(u, v) \in E \mid u \in A, v \in B\}$

Claim

(A, B) *is an* s-t *cut.*

Recall: Every *minimal* s-t cut E' is a cut of the form (A, B).

Ford-Fulkerson Correctness

Lemma

If there is no s-t *path in* G_f *then there is some cut* (A, B) *such that* $v(f) = c(A, B)$

Proof.

Let A be all vertices reachable from s in G_f; $B = V \setminus A$.

1. $s \in A$ and $t \in B$. So (A, B) is an s-t cut in G.
2. If $e = (u, v) \in G$ with $u \in A$ and $v \in B$, then $f(e) = c(e)$ (saturated edge) because otherwise v is reachable from s in G_f.

□

Ford-Fulkerson Correctness

Lemma

If there is no s-t *path in* G_f *then there is some cut* (A, B) *such that* $v(f) = c(A, B)$

Proof.

Let A be all vertices reachable from s in G_f; $B = V \setminus A$.

1. $s \in A$ and $t \in B$. So (A, B) is an s-t cut in G.
2. If $e = (u, v) \in G$ with $u \in A$ and $v \in B$, then $f(e) = c(e)$ (saturated edge) because otherwise v is reachable from s in G_f.

□

Ford-Fulkerson Correctness

Lemma

If there is no $\mathbf{s}$-$\mathbf{t}$ *path in* $\mathbf{G_f}$ *then there is some cut* $(\mathbf{A}, \mathbf{B})$ *such that* $\mathbf{v(f) = c(A, B)}$

Proof.

Let $\mathbf{A}$ be all vertices reachable from $\mathbf{s}$ in $\mathbf{G_f}$; $\mathbf{B = V \setminus A}$.

1. $\mathbf{s \in A}$ and $\mathbf{t \in B}$. So $(\mathbf{A}, \mathbf{B})$ is an $\mathbf{s}$-$\mathbf{t}$ cut in $\mathbf{G}$.
2. If $\mathbf{e = (u, v) \in G}$ with $\mathbf{u \in A}$ and $\mathbf{v \in B}$, then $\mathbf{f(e) = c(e)}$ (saturated edge) because otherwise $\mathbf{v}$ is reachable from $\mathbf{s}$ in $\mathbf{G_f}$.

□

Ford-Fulkerson Correctness

Lemma

If there is no s-t *path in* G_f *then there is some cut* (A, B) *such that* $v(f) = c(A, B)$

Proof.

Let A be all vertices reachable from s in G_f; $B = V \setminus A$.

1. $s \in A$ and $t \in B$. So (A, B) is an s-t cut in G.
2. If $e = (u, v) \in G$ with $u \in A$ and $v \in B$, then $f(e) = c(e)$ (saturated edge) because otherwise v is reachable from s in G_f.

□

Lemma Proof Continued

Proof.

1. If $e = (u', v') \in G$ with $u' \in B$ and $v' \in A$, then $f(e) = 0$ because otherwise u' is reachable from s in G_f
2. Thus,

$$\begin{aligned} v(f) &= f^{out}(A) - f^{in}(A) \\ &= f^{out}(A) - 0 \\ &= c(A, B) - 0 \\ &= c(A, B). \end{aligned}$$

□

Example

Flow f

Flow f

Residual graph G_f: no s-t path

A is reachable set from s in G_f

Example

Flow f

Flow f

Residual graph G_f: no s-t path

A is reachable set from s in G_f

Ford-Fulkerson Correctness

Theorem

The flow returned by the algorithm is the maximum flow.

Proof.

1. For any flow $\mathbf{f}$ and $\mathbf{s}$-$\mathbf{t}$ cut $(\mathbf{A}, \mathbf{B})$, $\mathbf{v}(\mathbf{f}) \leq \mathbf{c}(\mathbf{A}, \mathbf{B})$.
2. For flow $\mathbf{f}^*$ returned by algorithm, $\mathbf{v}(\mathbf{f}^*) = \mathbf{c}(\mathbf{A}^*, \mathbf{B}^*)$ for some $\mathbf{s}$-$\mathbf{t}$ cut $(\mathbf{A}^*, \mathbf{B}^*)$.
3. Hence, $\mathbf{f}^*$ is maximum.

□

Max-Flow Min-Cut Theorem and Integrality of Flows

Theorem

For any network **G**, *the value of a maximum* **s**-**t** *flow is equal to the capacity of the minimum* **s**-**t** *cut.*

Proof.

Ford-Fulkerson algorithm terminates with a maximum flow of value equal to the capacity of a (minimum) cut. □

Max-Flow Min-Cut Theorem and Integrality of Flows

Theorem

For any network **G** *with integer capacities, there is a maximum* **s**-**t** *flow that is integer valued.*

Proof.

Ford-Fulkerson algorithm produces an integer valued flow when capacities are integers. □

Does it terminate?

(A) **algFordFulkerson** always terminates.

(B) **algFordFulkerson** might not terminate if the input has real numbers.

(C) **algFordFulkerson** might not terminate if the input has rational numbers.

(D) **algFordFulkerson** might not terminate if the input is only integer numbers that are sufficiently large.

Efficiency of Ford-Fulkerson

Running time = **O(mC)** is not polynomial. Can the upper bound be achieved?

Efficiency of Ford-Fulkerson

Running time = **O(mC)** is not polynomial. Can the upper bound be achieved?

Efficiency of Ford-Fulkerson

Running time = **O(mC)** is not polynomial. Can the upper bound be achieved?

Polynomial Time Algorithms

Question: Is there a polynomial time algorithm for maxflow?

Question: Is there a variant of Ford-Fulkerson that leads to a polynomial time algorithm? Can we choose an augmenting path in some clever way? Yes! Two variants.

1. Choose the augmenting path with largest bottleneck capacity.
2. Choose the shortest augmenting path.

Polynomial Time Algorithms

Question: Is there a polynomial time algorithm for maxflow?

Question: Is there a variant of Ford-Fulkerson that leads to a polynomial time algorithm? Can we choose an augmenting path in some clever way? Yes! Two variants.

1. Choose the augmenting path with largest bottleneck capacity.
2. Choose the shortest augmenting path.

Polynomial Time Algorithms

Question: Is there a polynomial time algorithm for maxflow?

Question: Is there a variant of Ford-Fulkerson that leads to a polynomial time algorithm? Can we choose an augmenting path in some clever way? Yes! Two variants.

1. Choose the augmenting path with largest bottleneck capacity.
2. Choose the shortest augmenting path.

Finding path with largest bottleneck capacity

$\mathbf{G_f}$ - residual network with (residual) capacities.
$\mathbf{n}$ vertices and $\mathbf{m}$ edges.
Finding the $\mathbf{s}$-$\mathbf{t}$ path with largest bottleneck capacity can be done (faster is better) in:

(A) $O(n + m)$
(B) $O(n \log + m)$
(C) $O(nm)$
(D) $O(m^2)$
(E) $O(m^3)$

time (expected or deterministic is fine here).

Augmenting Paths with Large Bottleneck Capacity

1. Pick augmenting paths with largest bottleneck capacity in each iteration of Ford-Fulkerson.
2. How do we find path with largest bottleneck capacity?
   1. Assume we know Δ the bottleneck capacity
   2. Remove all edges with residual capacity $\leq \Delta$
   3. Check if there is a path from s to t
   4. Do binary search to find largest Δ
   5. Running time: $O(m \log C)$
3. Can we bound the number of augmentations? Can show that in $O(m \log C)$ augmentations the algorithm reaches a max flow. This leads to an $O(m^2 \log^2 C)$ time algorithm.

Augmenting Paths with Large Bottleneck Capacity

1. Pick augmenting paths with largest bottleneck capacity in each iteration of Ford-Fulkerson.
2. How do we find path with largest bottleneck capacity?
   1. Assume we know Δ the bottleneck capacity
   2. Remove all edges with residual capacity $\leq \Delta$
   3. Check if there is a path from s to t
   4. Do binary search to find largest Δ
   5. Running time: $O(m \log C)$
3. Can we bound the number of augmentations? Can show that in $O(m \log C)$ augmentations the algorithm reaches a max flow. This leads to an $O(m^2 \log^2 C)$ time algorithm.

Augmenting Paths with Large Bottleneck Capacity

1. Pick augmenting paths with largest bottleneck capacity in each iteration of Ford-Fulkerson.
2. How do we find path with largest bottleneck capacity?
   1. Assume we know Δ the bottleneck capacity
   2. Remove all edges with residual capacity $\leq \Delta$
   3. Check if there is a path from s to t
   4. Do binary search to find largest Δ
   5. Running time: $O(m \log C)$
3. Can we bound the number of augmentations? Can show that in $O(m \log C)$ augmentations the algorithm reaches a max flow. This leads to an $O(m^2 \log^2 C)$ time algorithm.

Augmenting Paths with Large Bottleneck Capacity

How do we find path with largest bottleneck capacity?

1. Max bottleneck capacity is one of the edge capacities. Why?
2. Can do binary search on the edge capacities. First, sort the edges by their capacities and then do binary search on that array as before.
3. Algorithm's running time is $\mathbf{O(m \log m)}$.
4. Different algorithm that also leads to $\mathbf{O(m \log m)}$ time algorithm by adapting Prim's algorithm.
5. $\mathbf{O(m)}$ time using linear-time median selection algorithm.

Removing Dependence on C

1. **Dinic [1970], Edmonds and Karp [1972]**
   Picking augmenting paths with fewest number of edges yields a $O(m^2n)$ algorithm, i.e., independent of C. Such an algorithm is called a strongly polynomial time algorithm since the running time does not depend on the numbers (assuming RAM model). (Many implementation of Ford-Fulkerson would actually use shortest augmenting path if they use **BFS** to find an s-t path).
2. Further improvements can yield algorithms running in $O(mn \log n)$, or $O(n^3)$.

Ford-Fulkerson Algorithm

```
algEdmondsKarp
    for every edge e, f(e) = 0
    G_f is residual graph of G with respect to f
    while G_f has a simple s-t path do
        Perform BFS in G_f
        P:  shortest s-t path in G_f
        f = augment(f, P)
        Construct new residual graph G_f.
```

Running time $O(m^2 n)$.

Finding a Minimum Cut

Question: How do we find an actual minimum **s**-**t** cut?

Proof gives the algorithm!

1. Compute an **s**-**t** maximum flow **f** in **G**
2. Obtain the residual graph $\mathbf{G_f}$
3. Find the nodes **A** reachable from **s** in $\mathbf{G_f}$
4. Output the cut $(\mathbf{A}, \mathbf{B}) = \{(\mathbf{u}, \mathbf{v}) \mid \mathbf{u} \in \mathbf{A}, \mathbf{v} \in \mathbf{B}\}$. Note: The cut is found in **G** while **A** is found in $\mathbf{G_f}$

Running time is essentially the same as finding a maximum flow.

Note: Given **G** and a flow **f** there is a linear time algorithm to check if **f** is a maximum flow and if it is, outputs a minimum cut. How?

Finding a Minimum Cut

Question: How do we find an actual minimum **s**-**t** cut?
Proof gives the algorithm!

1. Compute an **s**-**t** maximum flow **f** in **G**
2. Obtain the residual graph $\mathbf{G_f}$
3. Find the nodes **A** reachable from **s** in $\mathbf{G_f}$
4. Output the cut $(\mathbf{A}, \mathbf{B}) = \{(\mathbf{u}, \mathbf{v}) \mid \mathbf{u} \in \mathbf{A}, \mathbf{v} \in \mathbf{B}\}$. Note: The cut is found in **G** while **A** is found in $\mathbf{G_f}$

Running time is essentially the same as finding a maximum flow.

Note: Given **G** and a flow **f** there is a linear time algorithm to check if **f** is a maximum flow and if it is, outputs a minimum cut. How?

Notes

Notes

Notes

Notes

Dinic, E. A. (1970). Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Doklady*, 11:1277–1280.

Edmonds, J. and Karp, R. M. (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *J. Assoc. Comput. Mach.*, 19(2):248–264.

Applications of Network Flows

Lecture 18

April 3, 2014

Is the flow always integral?

Let G be an integral instance of network flow (i.e., all numbers are integers). Consider the following statements:

(I) The value of the maximum flow is an integer number.

(II) If $\mathbf{f}$ is a maximum flow, then $\mathbf{f(e)}$ is an integer, for any edge $\mathbf{e} \in E(G)$.

(III) There always exists a max flow $\mathbf{g}$, such that $\mathbf{g}$ is a maximum flow, and $\mathbf{g(e)}$ is an integer, for any edge $\mathbf{e} \in E(G)$.

We have the following:

(A) All the above statements are false.

(B) All the above statements are true.

(C) (I) is true, (II) and (III) are false.

(D) (I) and (II) are true, and (III) is false.

(E) (I) and (III) are true, and (II) is false.

Why max-flow does not have to be integral...

...but the one we compute always is!

Consider the graph with all capacities being one.

Why max-flow does not have to be integral...

...but the one we compute always is!

Consider the graph with all capacities being one.

One possible max flow:

Why max-flow does not have to be integral...

...but the one we compute always is!

Consider the graph with all capacities being one.

One possible max flow:

Max flow as computed by **algEdmondsKarp** or **algFordFulkerson**:

Network Flow: Facts to Remember

Flow network: directed graph $\mathbf{G}$, capacities $\mathbf{c}$, source $\mathbf{s}$, sink $\mathbf{t}$.

1. Maximum $\mathbf{s}$-$\mathbf{t}$ flow can be computed:
   1. Using Ford-Fulkerson algorithm in $\mathbf{O(mC)}$ time when capacities are integral and $\mathbf{C}$ is an upper bound on the flow.
   2. Using variant of algorithm, in $\mathbf{O(m^2 \log C)}$ time, when capacities are integral. (Polynomial time.)
   3. Using Edmonds-Karp algorithm, in $\mathbf{O(m^2 n)}$ time, when capacities are rational (strongly polynomial time algorithm).

Network Flow

Even more facts to remember

1. If capacities are integral then there is a maximum flow that is integral and above algorithms give an integral max flow. This is known as **integrality of flow**.
2. Given a flow of value $\mathbf{v}$, can decompose into $\mathbf{O(m+n)}$ flow paths of same total value $\mathbf{v}$. Integral flow implies integral flow on paths.
3. Maximum flow is equal to the minimum cut and minimum cut can be found in $\mathbf{O(m+n)}$ time given any maximum flow.

Paths, Cycles and Acyclicity of Flows

Definition

Given a flow network $\mathbf{G = (V, E)}$ and a flow $\mathbf{f : E \rightarrow \mathbb{R}^{\geq 0}}$ on the edges, the **support** of $\mathbf{f}$ is the set of edges $\mathbf{E' \subseteq E}$ with non-zero flow on them. That is, $\mathbf{E' = \{e \in E \mid f(e) > 0\}}$.

Question: Given a flow $\mathbf{f}$, can there by cycles in its support?

Paths, Cycles and Acyclicity of Flows

Definition

Given a flow network $\mathbf{G = (V, E)}$ and a flow $\mathbf{f : E \to \mathbb{R}^{\geq 0}}$ on the edges, the **support** of $\mathbf{f}$ is the set of edges $\mathbf{E' \subseteq E}$ with non-zero flow on them. That is, $\mathbf{E' = \{e \in E \mid f(e) > 0\}}$.

Question: Given a flow $\mathbf{f}$, can there by cycles in its support?

Paths, Cycles and Acyclicity of Flows

Definition

Given a flow network $\mathbf{G = (V, E)}$ and a flow $\mathbf{f : E \to \mathbb{R}^{\geq 0}}$ on the edges, the **support** of $\mathbf{f}$ is the set of edges $\mathbf{E' \subseteq E}$ with non-zero flow on them. That is, $\mathbf{E' = \{e \in E \mid f(e) > 0\}}$.

Question: Given a flow $\mathbf{f}$, can there by cycles in its support?

Paths, Cycles and Acyclicity of Flows

Definition

Given a flow network $\mathbf{G = (V, E)}$ and a flow $\mathbf{f : E \to \mathbb{R}^{\geq 0}}$ on the edges, the **support** of $\mathbf{f}$ is the set of edges $\mathbf{E' \subseteq E}$ with non-zero flow on them. That is, $\mathbf{E' = \{e \in E \mid f(e) > 0\}}$.

Question: Given a flow $\mathbf{f}$, can there by cycles in its support?

How fast can we detect a cycle in the flow

Given a flow network G with $\mathbf{n}$ vertices, and $\mathbf{m}$ edges, and a flow $\mathbf{f}$ on it, then detecting a cycle in the flow can be done in time

(A) $\mathbf{O(m + n)}$.
(B) $\mathbf{O(mC)}$.
(C) $\mathbf{O(mn)}$.
(D) $\mathbf{O(m^2n)}$.
(E) $\mathbf{O(mn^2)}$.

Acyclicity of Flows

Proposition

In any flow network, if $\mathbf{f}$ *is a flow then there is another flow* $\mathbf{f'}$ *such that the support of* $\mathbf{f'}$ *is an* acyclic *graph and* $\mathbf{v(f') = v(f)}$. *Further if* $\mathbf{f}$ *is an integral flow then so is* $\mathbf{f'}$.

Proof.

1. $\mathbf{E' = \{e \in E \mid f(e) > 0\}}$, support of $\mathbf{f}$.
2. Suppose there is a directed cycle $\mathbf{C}$ in $\mathbf{E'}$
3. Let $\mathbf{e'}$ be the edge in $\mathbf{C}$ with least amount of flow
4. For each $\mathbf{e \in C}$, reduce flow by $\mathbf{f(e')}$. Remains a flow. Why?
5. Flow on $\mathbf{e'}$ is reduced to $\mathbf{0}$.
6. Claim: Flow value from $\mathbf{s}$ to $\mathbf{t}$ does not change. Why?
7. Iterate until no cycles $\square$

Example

Example

Throw away edge with no flow on it

Example

Find a cycle in the support/flow

Example

Reduce flow on cycle as much as possible

Example

Throw away edge with no flow on it

Example

Find a cycle in the support/flow

Example

Reduce flow on cycle as much as possible

Example

Throw away edge with no flow on it

Example

Viola!!! An equivalent flow with no cycles in it. Original flow:

Flow Decomposition

Lemma

Given an edge based flow $\mathbf{f} : \mathbf{E} \rightarrow \mathbb{R}^{\geq 0}$, *there exists a collection of paths* $\mathcal{P}$ *and cycles* $\mathcal{C}$ *and an assignment of flow to them* $\mathbf{f}' : \mathcal{P} \cup \mathcal{C} \rightarrow \mathbb{R}^{\geq 0}$ *such that:*

1. $|\mathcal{P} \cup \mathcal{C}| \leq \mathbf{m}$
2. *for each* $\mathbf{e} \in \mathbf{E}$, $\sum_{\mathbf{P} \in \mathcal{P} : \mathbf{e} \in \mathbf{P}} \mathbf{f}'(\mathbf{P}) + \sum_{\mathbf{C} \in \mathcal{C} : \mathbf{e} \in \mathbf{C}} \mathbf{f}'(\mathbf{C}) = \mathbf{f}(\mathbf{e})$
3. $\mathbf{v}(\mathbf{f}) = \sum_{\mathbf{P} \in \mathcal{P}} \mathbf{f}'(\mathbf{P})$.
4. *if* $\mathbf{f}$ *is integral then so are* $\mathbf{f}'(\mathbf{P})$ *and* $\mathbf{f}'(\mathbf{C})$ *for all* $\mathbf{P}$ *and* $\mathbf{C}$

Proof Idea.

1. Remove all cycles as in previous proposition.
2. Next, decompose into paths as in previous lecture.
3. Exercise: verify claims. □

Flow Decomposition

Lemma

Given an edge based flow $\mathbf{f} : \mathbf{E} \rightarrow \mathbb{R}^{\geq 0}$*, there exists a collection of paths* $\mathcal{P}$ *and cycles* $\mathcal{C}$ *and an assignment of flow to them* $\mathbf{f}' : \mathcal{P} \cup \mathcal{C} \rightarrow \mathbb{R}^{\geq 0}$ *such that:*

1. $|\mathcal{P} \cup \mathcal{C}| \leq \mathbf{m}$
2. *for each* $\mathbf{e} \in \mathbf{E}$, $\sum_{\mathbf{P} \in \mathcal{P} : \mathbf{e} \in \mathbf{P}} \mathbf{f}'(\mathbf{P}) + \sum_{\mathbf{C} \in \mathcal{C} : \mathbf{e} \in \mathbf{C}} \mathbf{f}'(\mathbf{C}) = \mathbf{f}(\mathbf{e})$
3. $\mathbf{v}(\mathbf{f}) = \sum_{\mathbf{P} \in \mathcal{P}} \mathbf{f}'(\mathbf{P})$.
4. *if* $\mathbf{f}$ *is integral then so are* $\mathbf{f}'(\mathbf{P})$ *and* $\mathbf{f}'(\mathbf{C})$ *for all* $\mathbf{P}$ *and* $\mathbf{C}$

Proof Idea.

1. Remove all cycles as in previous proposition.
2. Next, decompose into paths as in previous lecture.
3. Exercise: verify claims. □

Example

Find cycles as shown before

Example

Find a source to sink path, and push max flow along it (5 unites)

Example

Compute remaining flow

Example

Find a source to sink path, and push max flow along it (5 unites). Edges with **0** flow on them can not be used as they are no longer in the support of the flow.

Example

Compute remaining flow

Example

Find a source to sink path, and push max flow along it (10 unites).

Example

Compute remaining flow

Example

Find a source to sink path, and push max flow along it (5 unites).

Example

Compute remaining flow

Example

No flow remains in the graph. We fully decomposed the flow into flow on paths. Together with the cycles, we get a decomposition of the original flow into **m** flows on paths and cycles.

Flow Decomposition

Lemma

Given an edge based flow $\mathbf{f} : \mathbf{E} \rightarrow \mathbb{R}^{\geq 0}$*, there exists a collection of paths* $\mathcal{P}$ *and cycles* $\mathcal{C}$ *and an assignment of flow to them* $\mathbf{f}' : \mathcal{P} \cup \mathcal{C} \rightarrow \mathbb{R}^{\geq 0}$ *such that:*

1. $|\mathcal{P} \cup \mathcal{C}| \leq \mathbf{m}$
2. *for each* $\mathbf{e} \in \mathbf{E}$, $\sum_{\mathbf{P} \in \mathcal{P} : \mathbf{e} \in \mathbf{P}} \mathbf{f}'(\mathbf{P}) + \sum_{\mathbf{C} \in \mathcal{C} : \mathbf{e} \in \mathbf{C}} \mathbf{f}'(\mathbf{C}) = \mathbf{f}(\mathbf{e})$
3. $\mathbf{v}(\mathbf{f}) = \sum_{\mathbf{P} \in \mathcal{P}} \mathbf{f}'(\mathbf{P})$.
4. *if* $\mathbf{f}$ *is integral then so are* $\mathbf{f}'(\mathbf{P})$ *and* $\mathbf{f}'(\mathbf{C})$ *for all* $\mathbf{P}$ *and* $\mathbf{C}$.

Above flow decomposition can be computed in $\mathbf{O}(\mathbf{m}^2)$ *time.*

Flow decomposition into paths and cycles

Consider an integral flow network G, and two maximum flows $\mathbf{f}$ and $\mathbf{g}$ in G. Assume both $\mathbf{f}$ and $\mathbf{g}$ are acyclic. Let $\mathbf{P_f}$ and $\mathbf{P_g}$ be the decomposition of the two flows into paths. Then:

(A) $\mathbf{P_f} = \mathbf{P_g}$ (paths are the same).

(B) $|\mathbf{P_f}| = |\mathbf{P_g}|$ (i.e., number of paths is the same).

(C) $|\mathbf{P_f}| + |\mathbf{P_g}| = \mathbf{m}$.

(D) $|\mathbf{P_f}| * |\mathbf{P_g}| = \mathbf{nm}$.

(E) None of the above.

Part I

Network Flow Applications I

Edge-Disjoint Paths in Directed Graphs

Definition

A set of paths is **edge disjoint** if no two paths share an edge.

Problem

Given a directed graph with two special vertices **s** and **t**, find the *maximum* number of edge disjoint paths from **s** to **t**.

Applications: Fault tolerance in routing — edges/nodes in networks can fail. Disjoint paths allow for planning backup routes in case of failures.

Edge-Disjoint Paths in Directed Graphs

Definition

A set of paths is **edge disjoint** if no two paths share an edge.

Problem

Given a directed graph with two special vertices **s** and **t**, find the *maximum* number of edge disjoint paths from **s** to **t**.

Applications: Fault tolerance in routing — edges/nodes in networks can fail. Disjoint paths allow for planning backup routes in case of failures.

Reduction to Max-Flow

Problem

Given a directed graph $\mathbf{G}$ with two special vertices $\mathbf{s}$ and $\mathbf{t}$, find the maximum number of edge disjoint paths from $\mathbf{s}$ to $\mathbf{t}$.

Reduction

Consider $\mathbf{G}$ as a flow network with edge capacities $\mathbf{1}$, and compute max-flow.

Correctness of Reduction

Lemma

If $\mathbf{G}$ *has* $\mathbf{k}$ *edge disjoint paths* $\mathbf{P_1, P_2, \ldots, P_k}$ *then there is an* $\mathbf{s}$-$\mathbf{t}$ *flow of value* $\mathbf{k}$ *in* $\mathbf{G}$.

Proof.

Set $\mathbf{f(e) = 1}$ if $\mathbf{e}$ belongs to one of the paths $\mathbf{P_1, P_2, \ldots, P_k}$; other-wise set $\mathbf{f(e) = 0}$. This defines a flow of value $\mathbf{k}$. □

Correctness of Reduction

Lemma

If $\mathbf{G}$ *has* $\mathbf{k}$ *edge disjoint paths* $\mathbf{P_1, P_2, \ldots, P_k}$ *then there is an* $\mathbf{s}$-$\mathbf{t}$ *flow of value* $\mathbf{k}$ *in* $\mathbf{G}$.

Proof.

Set $\mathbf{f(e) = 1}$ if $\mathbf{e}$ belongs to one of the paths $\mathbf{P_1, P_2, \ldots, P_k}$; other-wise set $\mathbf{f(e) = 0}$. This defines a flow of value $\mathbf{k}$. □

Correctness of Reduction

Lemma

If $\mathbf{G}$ *has a flow of value* $\mathbf{k}$ *then there are* $\mathbf{k}$ *edge disjoint paths between* $\mathbf{s}$ *and* $\mathbf{t}$.

Proof.

1. Capacities are all 1 and hence there is integer flow of value k, that is $f(e) = 0$ or $f(e) = 1$ for each e.
2. Decompose flow into paths.
3. Flow on each path is either 1 or 0.
4. Hence there are k paths $P_1, P_2, \ldots, P_k$ with flow of 1 each.
5. Paths are edge-disjoint since capacities are 1. □

Correctness of Reduction

Lemma

If $\mathbf{G}$ *has a flow of value* $\mathbf{k}$ *then there are* $\mathbf{k}$ *edge disjoint paths between* $\mathbf{s}$ *and* $\mathbf{t}$.

Proof.

1. Capacities are all $\mathbf{1}$ and hence there is integer flow of value $\mathbf{k}$, that is $\mathbf{f(e) = 0}$ or $\mathbf{f(e) = 1}$ for each $\mathbf{e}$.
2. Decompose flow into paths.
3. Flow on each path is either $\mathbf{1}$ or $\mathbf{0}$.
4. Hence there are $\mathbf{k}$ paths $\mathbf{P_1, P_2, \ldots, P_k}$ with flow of $\mathbf{1}$ each.
5. Paths are edge-disjoint since capacities are $\mathbf{1}$. □

Running Time

Theorem

The number of edge disjoint paths in a simple graph $\mathbf{G}$ *can be found in* $\mathbf{O(mn)}$ *time.*

Proof.

1. Set capacities of edges in $\mathbf{G}$ to $\mathbf{1}$.
2. Run Ford-Fulkerson algorithm.
3. Maximum value of flow is $\mathbf{n}$ and hence run-time is $\mathbf{O(nm)}$.
4. Decompose flow into $\mathbf{k}$ paths ($\mathbf{k} \leq \mathbf{n}$).
   Takes $\mathbf{O(k \times m) = O(km) = O(mn)}$ time. □

Remark

The algorithm also computes a set of edge-disjoint paths realizing this optimal solution.

Running Time

Theorem

The number of edge disjoint paths in a simple graph $\mathbf{G}$ *can be found in* $\mathbf{O(mn)}$ *time.*

Proof.

1. Set capacities of edges in $\mathbf{G}$ to $\mathbf{1}$.
2. Run Ford-Fulkerson algorithm.
3. Maximum value of flow is $\mathbf{n}$ and hence run-time is $\mathbf{O(nm)}$.
4. Decompose flow into $\mathbf{k}$ paths ($\mathbf{k} \leq \mathbf{n}$).
   Takes $\mathbf{O(k \times m) = O(km) = O(mn)}$ time. □

Remark

The algorithm also computes a set of edge-disjoint paths realizing this optimal solution.

Menger's Theorem

Theorem (Menger [1927])

Let **G** *be a directed graph. The minimum number of edges whose removal disconnects* **s** *from* **t** *(the minimum-cut between* **s** *and* **t***) is equal to the maximum number of edge-disjoint paths in* **G** *between* **s** *and* **t***.*

Proof.

Maxflow-mincut theorem and integrality of flow. □

Menger proved his theorem before Maxflow-Mincut theorem!
Maxflow-Mincut theorem is a generalization of Menger's theorem to capacitated graphs.

Menger's Theorem

Theorem (Menger [1927])

Let **G** *be a directed graph. The minimum number of edges whose removal disconnects* **s** *from* **t** *(the minimum-cut between* **s** *and* **t***) is equal to the maximum number of edge-disjoint paths in* **G** *between* **s** *and* **t***.*

Proof.

Maxflow-mincut theorem and integrality of flow. □

Menger proved his theorem before Maxflow-Mincut theorem!
Maxflow-Mincut theorem is a generalization of Menger's theorem to capacitated graphs.

Menger's Theorem

Theorem (Menger [1927])

Let **G** *be a directed graph. The minimum number of edges whose removal disconnects* **s** *from* **t** *(the minimum-cut between* **s** *and* **t***) is equal to the maximum number of edge-disjoint paths in* **G** *between* **s** *and* **t***.*

Proof.

Maxflow-mincut theorem and integrality of flow. □

Menger proved his theorem before Maxflow-Mincut theorem!
Maxflow-Mincut theorem is a generalization of Menger's theorem to capacitated graphs.

Edge Disjoint Paths in Undirected Graphs

Problem

Given an undirected graph $\mathbf{G}$, find the maximum number of edge disjoint paths in $\mathbf{G}$

Reduction:

1. create directed graph $\mathbf{H}$ by adding directed edges $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$ for each edge $\mathbf{uv}$ in $\mathbf{G}$.
2. compute maximum $\mathbf{s}$-$\mathbf{t}$ flow in $\mathbf{H}$.

Problem: Both edges $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$ may have non-zero flow!

Not a Problem! Can assume maximum flow in $\mathbf{H}$ is acyclic and hence cannot have non-zero flow on both $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$. Reduction works. See book for more details.

Edge Disjoint Paths in Undirected Graphs

Problem

Given an undirected graph $\mathbf{G}$, find the maximum number of edge disjoint paths in $\mathbf{G}$

Reduction:

1. create directed graph $\mathbf{H}$ by adding directed edges $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$ for each edge $\mathbf{uv}$ in $\mathbf{G}$.
2. compute maximum $\mathbf{s}$-$\mathbf{t}$ flow in $\mathbf{H}$.

Problem: Both edges $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$ may have non-zero flow!

Not a Problem! Can assume maximum flow in $\mathbf{H}$ is acyclic and hence cannot have non-zero flow on both $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$. Reduction works. See book for more details.

Edge Disjoint Paths in Undirected Graphs

Problem

Given an undirected graph $\mathbf{G}$, find the maximum number of edge disjoint paths in $\mathbf{G}$

Reduction:

1. create directed graph $\mathbf{H}$ by adding directed edges $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$ for each edge $\mathbf{uv}$ in $\mathbf{G}$.
2. compute maximum $\mathbf{s}$-$\mathbf{t}$ flow in $\mathbf{H}$.

Problem: Both edges $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$ may have non-zero flow!

Not a Problem! Can assume maximum flow in $\mathbf{H}$ is acyclic and hence cannot have non-zero flow on both $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$. Reduction works. See book for more details.

Edge Disjoint Paths in Undirected Graphs

Problem

Given an undirected graph $\mathbf{G}$, find the maximum number of edge disjoint paths in $\mathbf{G}$

Reduction:

1. create directed graph $\mathbf{H}$ by adding directed edges $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$ for each edge $\mathbf{uv}$ in $\mathbf{G}$.
2. compute maximum $\mathbf{s}$-$\mathbf{t}$ flow in $\mathbf{H}$.

Problem: Both edges $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$ may have non-zero flow!

Not a Problem! Can assume maximum flow in $\mathbf{H}$ is acyclic and hence cannot have non-zero flow on both $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{v}, \mathbf{u})$. Reduction works. See book for more details.

Node Disjoint Paths

Definition

A set of **s**-**t** paths $\mathcal{P}$ are *internally* node-disjoint if no two paths in $\mathcal{P}$ share a node other than **s, t**.

Theorem (Menger [1927])

Let **G** *be an undirected graph. The minimum number of nodes in* $V \setminus \{s, t\}$ *whose removal disconnects* **s** *from* **t** *is equal to the maximum number of internally node-disjoint paths in* **G** *between* **s** *and* **t**.

Theorem

The max number of internally node-disjoint paths between **s** *and* **t** *in* **G** *can be computed in* $O(mn)$ *time.*

Via reductions to directed graph *edge*-disjoint case!

Node Disjoint Paths

Definition

A set of $\mathbf{s}$-$\mathbf{t}$ paths $\mathcal{P}$ are *internally* node-disjoint if no two paths in $\mathcal{P}$ share a node other than $\mathbf{s}, \mathbf{t}$.

Theorem (Menger [1927])

Let $\mathbf{G}$ *be an undirected graph. The minimum number of nodes in* $\mathbf{V} \setminus \{\mathbf{s}, \mathbf{t}\}$ *whose removal disconnects* $\mathbf{s}$ *from* $\mathbf{t}$ *is equal to the maximum number of internally node-disjoint paths in* $\mathbf{G}$ *between* $\mathbf{s}$ *and* $\mathbf{t}$.

Theorem

The max number of internally node-disjoint paths between $\mathbf{s}$ *and* $\mathbf{t}$ *in* $\mathbf{G}$ *can be computed in* $\mathbf{O(mn)}$ *time.*

Via reductions to directed graph *edge*-disjoint case!

Node Disjoint Paths

Definition

A set of $\mathbf{s}$-$\mathbf{t}$ paths $\mathcal{P}$ are *internally* node-disjoint if no two paths in $\mathcal{P}$ share a node other than $\mathbf{s}, \mathbf{t}$.

Theorem (Menger [1927])

Let $\mathbf{G}$ *be an undirected graph. The minimum number of nodes in* $\mathbf{V} \setminus \{\mathbf{s}, \mathbf{t}\}$ *whose removal disconnects* $\mathbf{s}$ *from* $\mathbf{t}$ *is equal to the maximum number of internally node-disjoint paths in* $\mathbf{G}$ *between* $\mathbf{s}$ *and* $\mathbf{t}$.

Theorem

The max number of internally node-disjoint paths between $\mathbf{s}$ *and* $\mathbf{t}$ *in* $\mathbf{G}$ *can be computed in* $\mathbf{O(mn)}$ *time.*

Via reductions to directed graph *edge*-disjoint case!

Multiple Sources and Sinks

Input:

1. A directed graph $\mathbf{G}$ with edge capacities $\mathbf{c(e)}$.
2. Source nodes $\mathbf{s_1, s_2, \ldots, s_k}$.
3. Sink nodes $\mathbf{t_1, t_2, \ldots, t_\ell}$.
4. Sources and sinks are *disjoint*.

Maximum Flow: Send as much flow as possible from the sources to the sinks. *Sinks don't care which source they get flow from.*

Minimum Cut: Find a minimum capacity set of edge $\mathbf{E'}$ such that removing $\mathbf{E'}$ disconnects every source from every sink.

Multiple Sources and Sinks

Input:

1. A directed graph $\mathbf{G}$ with edge capacities $\mathbf{c(e)}$.
2. Source nodes $\mathbf{s_1, s_2, \ldots, s_k}$.
3. Sink nodes $\mathbf{t_1, t_2, \ldots, t_\ell}$.
4. Sources and sinks are *disjoint*.

Maximum Flow: Send as much flow as possible from the sources to the sinks. *Sinks don't care which source they get flow from.*

Minimum Cut: Find a minimum capacity set of edge $\mathbf{E'}$ such that removing $\mathbf{E'}$ disconnects every source from every sink.

Multiple Sources and Sinks

Input:

1. A directed graph $\mathbf{G}$ with edge capacities $\mathbf{c(e)}$.
2. Source nodes $\mathbf{s_1, s_2, \ldots, s_k}$.
3. Sink nodes $\mathbf{t_1, t_2, \ldots, t_\ell}$.
4. Sources and sinks are *disjoint*.

Maximum Flow: Send as much flow as possible from the sources to the sinks. *Sinks don't care which source they get flow from.*

Minimum Cut: Find a minimum capacity set of edge $\mathbf{E'}$ such that removing $\mathbf{E'}$ disconnects every source from every sink.

Multiple Sources and Sinks: Formal Definition

Input:

1. A directed graph $\mathbf{G}$ with edge capacities $\mathbf{c(e)}$.
2. Source nodes $\mathbf{s_1, s_2, \ldots, s_k}$.
3. Sink nodes $\mathbf{t_1, t_2, \ldots, t_\ell}$.
4. Sources and sinks are *disjoint*.

A function $\mathbf{f : E \rightarrow \mathbb{R}^{\geq 0}}$ is a **flow** if:

1. For each $\mathbf{e \in E}$, $\mathbf{f(e) \leq c(e)}$, and
2. for each $\mathbf{v}$ which is not a source or a sink $\mathbf{f^{in}(v) = f^{out}(v)}$.

Goal: max $\sum_{i=1}^{k}(\mathbf{f^{out}(s_i) - f^{in}(s_i)})$, that is, flow out of sources.

Reduction to Single-Source Single-Sink

1. Add a **source** node $\mathbf{s}$ and a **sink** node $\mathbf{t}$.
2. Add edges $(\mathbf{s}, \mathbf{s_1}), (\mathbf{s}, \mathbf{s_2}), \ldots, (\mathbf{s}, \mathbf{s_k})$.
3. Add edges $(\mathbf{t_1}, \mathbf{t}), (\mathbf{t_2}, \mathbf{t}), \ldots, (\mathbf{t_\ell}, \mathbf{t})$.
4. Set the capacity of the new edges to be ∞.

Reduction to Single-Source Single-Sink

1. Add a **source** node **s** and a **sink** node **t**.
2. Add edges $(s, s_1), (s, s_2), \ldots, (s, s_k)$.
3. Add edges $(t_1, t), (t_2, t), \ldots, (t_\ell, t)$.
4. Set the capacity of the new edges to be ∞.

Supplies and Demands

A further generalization:

1. source s_i has a supply of $S_i \geq 0$
2. since t_j has a demand of $D_j \geq 0$ units

Question: is there a flow from source to sinks such that supplies are not exceeded and demands are met? Formally we have the additional constraints that $f^{out}(s_i) - f^{in}(s_i) \leq S_i$ for each source s_i and $f^{in}(t_j) - f^{out}(t_j) \geq D_j$ for each sink t_j.

Supplies and Demands

A further generalization:

1. source s_i has a supply of $S_i \geq 0$
2. since t_j has a demand of $D_j \geq 0$ units

Question: is there a flow from source to sinks such that supplies are not exceeded and demands are met? Formally we have the additional constraints that $f^{out}(s_i) - f^{in}(s_i) \leq S_i$ for each source s_i and $f^{in}(t_j) - f^{out}(t_j) \geq D_j$ for each sink t_j.

Matching

Problem (Matching)

Input: *Given a (undirected) graph* $\mathbf{G = (V, E)}$.
Goal: *Find a matching of maximum cardinality.*

1. *A matching is* $\mathbf{M} \subseteq \mathbf{E}$ *such that at most one edge in* $\mathbf{M}$ *is incident on any vertex*

Matching

Problem (Matching)

Input: *Given a (undirected) graph* $\mathbf{G = (V, E)}$.
Goal: *Find a matching of maximum cardinality.*

1. *A matching is* $\mathbf{M} \subseteq \mathbf{E}$ *such that at most one edge in* $\mathbf{M}$ *is incident on any vertex*

Bipartite Matching

Problem (Bipartite matching)

Input: *Given a bipartite graph* $\mathbf{G = (L \cup R, E)}$.
Goal: *Find a matching of maximum cardinality*

Maximum matching has 4 edges

Bipartite Matching

Problem (Bipartite matching)

Input: *Given a bipartite graph* $\mathbf{G = (L \cup R, E)}$.
Goal: *Find a matching of maximum cardinality*

Maximum matching has 4 edges

Bipartite Matching

Problem (Bipartite matching)

Input: *Given a bipartite graph* $\mathbf{G = (L \cup R, E)}$.
Goal: *Find a matching of maximum cardinality*

Maximum matching has 4 edges

Reduction of bipartite matching to max-flow

Max-Flow Construction

Given graph $\mathbf{G = (L \cup R, E)}$ create flow-network $\mathbf{G' = (V', E')}$ as follows:

1. $\mathbf{V' = L \cup R \cup \{s, t\}}$ where $\mathbf{s}$ and $\mathbf{t}$ are the new source and sink.
2. Direct all edges in $\mathbf{E}$ from $\mathbf{L}$ to $\mathbf{R}$, and add edges from $\mathbf{s}$ to all vertices in $\mathbf{L}$ and from each vertex in $\mathbf{R}$ to $\mathbf{t}$.
3. Capacity of every edge is $\mathbf{1}$.

Reduction of bipartite matching to max-flow

Max-Flow Construction

Given graph $\mathbf{G = (L \cup R, E)}$ create flow-network $\mathbf{G' = (V', E')}$ as follows:

1. $\mathbf{V' = L \cup R \cup \{s, t\}}$ where $\mathbf{s}$ and $\mathbf{t}$ are the new source and sink.
2. Direct all edges in $\mathbf{E}$ from $\mathbf{L}$ to $\mathbf{R}$, and add edges from $\mathbf{s}$ to all vertices in $\mathbf{L}$ and from each vertex in $\mathbf{R}$ to $\mathbf{t}$.
3. Capacity of every edge is $\mathbf{1}$.

Reduction of bipartite matching to max-flow

Max-Flow Construction

Given graph $\mathbf{G = (L \cup R, E)}$ create flow-network $\mathbf{G' = (V', E')}$ as follows:

1. $\mathbf{V' = L \cup R \cup \{s, t\}}$ where $\mathbf{s}$ and $\mathbf{t}$ are the new source and sink.
2. Direct all edges in $\mathbf{E}$ from $\mathbf{L}$ to $\mathbf{R}$, and add edges from $\mathbf{s}$ to all vertices in $\mathbf{L}$ and from each vertex in $\mathbf{R}$ to $\mathbf{t}$.
3. Capacity of every edge is 1.

Reduction of bipartite matching to max-flow

Max-Flow Construction

Given graph $\mathbf{G = (L \cup R, E)}$ create flow-network $\mathbf{G' = (V', E')}$ as follows:

1. $\mathbf{V' = L \cup R \cup \{s, t\}}$ where $\mathbf{s}$ and $\mathbf{t}$ are the new source and sink.
2. Direct all edges in $\mathbf{E}$ from $\mathbf{L}$ to $\mathbf{R}$, and add edges from $\mathbf{s}$ to all vertices in $\mathbf{L}$ and from each vertex in $\mathbf{R}$ to $\mathbf{t}$.
3. Capacity of every edge is $\mathbf{1}$.

Correctness: Matching to Flow

Proposition

If G has a matching of size k then G' has a flow of value k.

Proof.

Let M be matching of size k. Let $M = \{(u_1, v_1), \ldots, (u_k, v_k)\}$. Consider following flow f in G':

1. $f(s, u_i) = 1$ and $f(v_i, t) = 1$ for $1 \leq i \leq k$
2. $f(u_i, v_i) = 1$ for $1 \leq i \leq k$
3. for all other edges flow is zero.

Verify that f is a flow of value k (because M is a matching). □

Correctness: Matching to Flow

Proposition

If $\mathbf{G}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G'}$ *has a flow of value* $\mathbf{k}$.

Proof.

Let $\mathbf{M}$ be matching of size $\mathbf{k}$. Let $\mathbf{M} = \{(u_1, v_1), \ldots, (u_k, v_k)\}$. Consider following flow $\mathbf{f}$ in $\mathbf{G'}$:

1. $f(s, u_i) = 1$ and $f(v_i, t) = 1$ for $1 \leq i \leq k$
2. $f(u_i, v_i) = 1$ for $1 \leq i \leq k$
3. for all other edges flow is zero.

Verify that $\mathbf{f}$ is a flow of value $\mathbf{k}$ (because $\mathbf{M}$ is a matching). □

Correctness: Flow to Matching

Proposition

If $\mathbf{G'}$ *has a flow of value* $\mathbf{k}$ *then* $\mathbf{G}$ *has a matching of size* $\mathbf{k}$.

Proof.

Consider flow $\mathbf{f}$ of value $\mathbf{k}$.

1. Can assume $\mathbf{f}$ is integral. Thus each edge has flow $\mathbf{1}$ or $\mathbf{0}$.
2. Consider the set $\mathbf{M}$ of edges from $\mathbf{L}$ to $\mathbf{R}$ that have flow 1.
   1. $\mathbf{M}$ has $\mathbf{k}$ edges because value of flow is equal to the number of non-zero flow edges crossing cut $(\mathbf{L} \cup \{\mathbf{s}\}, \mathbf{R} \cup \{\mathbf{t}\})$
   2. Each vertex has at most one edge in $\mathbf{M}$ incident upon it. Why?

□

Correctness of Reduction

Theorem

The maximum flow value in $\mathbf{G}'$ *= maximum cardinality of matching in* $\mathbf{G}$*.*

Consequence

Thus, to find maximum cardinality matching in $\mathbf{G}$, we construct $\mathbf{G}'$ and find the maximum flow in $\mathbf{G}'$. Note that the matching itself (not just the value) can be found efficiently from the flow.

Running Time

For graph $\mathbf{G}$ with $\mathbf{n}$ vertices and $\mathbf{m}$ edges $\mathbf{G'}$ has $\mathbf{O(n+m)}$ edges, and $\mathbf{O(n)}$ vertices.

1. Generic Ford-Fulkerson: Running time is $\mathbf{O(mC) = O(nm)}$ since $\mathbf{C = n}$.
2. Capacity scaling: Running time is $\mathbf{O(m^2 \log C) = O(m^2 \log n)}$.

Better running time is known: $\mathbf{O(m\sqrt{n})}$.

Running Time

For graph $\mathbf{G}$ with $\mathbf{n}$ vertices and $\mathbf{m}$ edges $\mathbf{G'}$ has $\mathbf{O(n+m)}$ edges, and $\mathbf{O(n)}$ vertices.

1. Generic Ford-Fulkerson: Running time is $\mathbf{O(mC) = O(nm)}$ since $\mathbf{C = n}$.
2. Capacity scaling: Running time is $\mathbf{O(m^2 \log C) = O(m^2 \log n)}$.

Better running time is known: $\mathbf{O(m\sqrt{n})}$.

3sided matching

Clicker question

Given sets $\mathbf{A}, \mathbf{B}, \mathbf{C}$ of size $\mathbf{n}$, and a set of triples $\mathbf{T} = \left\{ (\mathbf{x_i}, \mathbf{y_i}, \mathbf{z_i}) \in \mathbf{A} \times \mathbf{B} \times \mathbf{C} \;\middle|\; \mathbf{i} = \mathbf{1}, \ldots, \mathbf{m} \right\}$, one can compute a maximum number of disjoint triplets in $\mathbf{T}$, by building the natural $\mathbf{3}$ layers graph as before (with an additional source and sink), and computing max-flow in this graph. This algorithm is

- **(A)** incorrect because one can not compute a max flow in such graph.
- **(B)** incorrect because of some other reason.
- **(C)** correct.
- **(D)** correct if you assume no element in $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ participates in more than $\mathbf{2}$ triplets.
- **(E)** None of the above.

Perfect Matchings

Definition

A matching **M** is said to be **perfect** if every vertex has one edge in **M** incident upon it.

Figure : This graph does not have a perfect matching

Characterizing Perfect Matchings

Problem

When does a bipartite graph have a perfect matching?

1. Clearly $|\mathbf{L}| = |\mathbf{R}|$
2. Are there any necessary and sufficient conditions?

A Necessary Condition

Lemma

If $\mathbf{G} = (\mathbf{L} \cup \mathbf{R}, \mathbf{E})$ *has a perfect matching then for any* $\mathbf{X} \subseteq \mathbf{L}$, $|\mathbf{N(X)}| \geq |\mathbf{X}|$, *where* $\mathbf{N(X)}$ *is the set of neighbors of vertices in* $\mathbf{X}$.

Proof.

Since $\mathbf{G}$ has a perfect matching, every vertex of $\mathbf{X}$ is matched to a different neighbor, and so $|\mathbf{N(X)}| \geq |\mathbf{X}|$. □

A Necessary Condition

Lemma

If $\mathbf{G = (L \cup R, E)}$ *has a perfect matching then for any* $\mathbf{X} \subseteq \mathbf{L}$, $|\mathbf{N(X)}| \geq |\mathbf{X}|$, *where* $\mathbf{N(X)}$ *is the set of neighbors of vertices in* $\mathbf{X}$.

Proof.

Since $\mathbf{G}$ has a perfect matching, every vertex of $\mathbf{X}$ is matched to a different neighbor, and so $|\mathbf{N(X)}| \geq |\mathbf{X}|$. □

Hall's Theorem

Theorem (Frobenius-Hall)

Let $\mathbf{G} = (\mathbf{L} \cup \mathbf{R}, \mathbf{E})$ *be a bipartite graph with* $|\mathbf{L}| = |\mathbf{R}|$. $\mathbf{G}$ *has a perfect matching if and only if for every* $\mathbf{X} \subseteq \mathbf{L}$, $|\mathbf{N}(\mathbf{X})| \geq |\mathbf{X}|$.

One direction is the necessary condition.

For the other direction we will show the following:

1. Create flow network $\mathbf{G'}$ from $\mathbf{G}$.
2. If $|\mathbf{N}(\mathbf{X})| \geq |\mathbf{X}|$ for all $\mathbf{X}$, show that minimum $\mathbf{s}$-$\mathbf{t}$ cut in $\mathbf{G'}$ is of capacity $\mathbf{n} = |\mathbf{L}| = |\mathbf{R}|$.
3. Implies that $\mathbf{G}$ has a perfect matching.

Hall's Theorem

Theorem (Frobenius-Hall)

Let $\mathbf{G} = (\mathbf{L} \cup \mathbf{R}, \mathbf{E})$ *be a bipartite graph with* $|\mathbf{L}| = |\mathbf{R}|$. $\mathbf{G}$ *has a perfect matching if and only if for every* $\mathbf{X} \subseteq \mathbf{L}$, $|\mathbf{N}(\mathbf{X})| \geq |\mathbf{X}|$.

One direction is the necessary condition.
For the other direction we will show the following:

1. Create flow network $\mathbf{G'}$ from $\mathbf{G}$.
2. If $|\mathbf{N}(\mathbf{X})| \geq |\mathbf{X}|$ for all $\mathbf{X}$, show that minimum $\mathbf{s}$-$\mathbf{t}$ cut in $\mathbf{G'}$ is of capacity $\mathbf{n} = |\mathbf{L}| = |\mathbf{R}|$.
3. Implies that $\mathbf{G}$ has a perfect matching.

Proof of Sufficiency

Assume $|\mathbf{N(X)}| \geq |\mathbf{X}|$ for any $\mathbf{X} \subseteq \mathbf{L}$. Then show that min **s**-**t** cut in $\mathbf{G'}$ is of capacity at least $\mathbf{n}$.

Let $(\mathbf{A}, \mathbf{B})$ be an *arbitrary* **s**-**t** cut in $\mathbf{G'}$

1. Let $\mathbf{X} = \mathbf{A} \cap \mathbf{L}$ and $\mathbf{Y} = \mathbf{A} \cap \mathbf{R}$.
2. Cut capacity is at least $(|\mathbf{L}| - |\mathbf{X}|) + |\mathbf{Y}| + |\mathbf{N(X)} \setminus \mathbf{Y}|$

Proof of Sufficiency

Assume $|\mathbf{N(X)}| \geq |\mathbf{X}|$ for any $\mathbf{X} \subseteq \mathbf{L}$. Then show that min **s**-**t** cut in $\mathbf{G'}$ is of capacity at least $\mathbf{n}$.

Let $\mathbf{(A, B)}$ be an *arbitrary* **s**-**t** cut in $\mathbf{G'}$

1. Let $\mathbf{X} = \mathbf{A} \cap \mathbf{L}$ and $\mathbf{Y} = \mathbf{A} \cap \mathbf{R}$.
2. Cut capacity is at least $(|\mathbf{L}| - |\mathbf{X}|) + |\mathbf{Y}| + |\mathbf{N(X)} \setminus \mathbf{Y}|$

Proof of Sufficiency

Assume $|\mathbf{N(X)}| \geq |\mathbf{X}|$ for any $\mathbf{X} \subseteq \mathbf{L}$. Then show that min **s**-**t** cut in $\mathbf{G'}$ is of capacity at least $\mathbf{n}$.

Let $\mathbf{(A, B)}$ be an *arbitrary* **s**-**t** cut in $\mathbf{G'}$

1. Let $\mathbf{X} = \mathbf{A} \cap \mathbf{L}$ and $\mathbf{Y} = \mathbf{A} \cap \mathbf{R}$.
2. Cut capacity is at least $(|\mathbf{L}| - |\mathbf{X}|) + |\mathbf{Y}| + |\mathbf{N(X)} \setminus \mathbf{Y}|$

Proof of Sufficiency

Assume $|\mathbf{N(X)}| \geq |\mathbf{X}|$ for any $\mathbf{X} \subseteq \mathbf{L}$. Then show that min **s**-**t** cut in $\mathbf{G'}$ is of capacity at least **n**.

Let $(\mathbf{A}, \mathbf{B})$ be an *arbitrary* **s**-**t** cut in $\mathbf{G'}$

1. Let $\mathbf{X} = \mathbf{A} \cap \mathbf{L}$ and $\mathbf{Y} = \mathbf{A} \cap \mathbf{R}$.
2. Cut capacity is at least $(|\mathbf{L}| - |\mathbf{X}|) + |\mathbf{Y}| + |\mathbf{N(X)} \setminus \mathbf{Y}|$

Because there are...

1. $|\mathbf{L}| - |\mathbf{X}|$ edges from **s** to $\mathbf{L} \cap \mathbf{B}$.
2. $|\mathbf{Y}|$ edges from **Y** to **t**.
3. there are at least $|\mathbf{N(X)} \setminus \mathbf{Y}|$ edges from **X** to vertices on the right side that are not in **Y**.

Proof of Sufficiency

Continued...

1. By the above, cut capacity is at least
$$\alpha = (|\mathbf{L}| - |\mathbf{X}|) + |\mathbf{Y}| + |\mathbf{N(X)} \setminus \mathbf{Y}|.$$
2. $|\mathbf{N(X)} \setminus \mathbf{Y}| \geq |\mathbf{N(X)}| - |\mathbf{Y}|$.
(This holds for any two sets.)
3. By assumption $|\mathbf{N(X)}| \geq |\mathbf{X}|$ and hence
$$|\mathbf{N(X)} \setminus \mathbf{Y}| \geq |\mathbf{N(X)}| - |\mathbf{Y}| \geq |\mathbf{X}| - |\mathbf{Y}|.$$
4. Cut capacity is therefore at least
$$\begin{aligned}\alpha &= (|\mathbf{L}| - |\mathbf{X}|) + |\mathbf{Y}| + |\mathbf{N(X)} \setminus \mathbf{Y}| \\ &\geq |\mathbf{L}| - |\mathbf{X}| + |\mathbf{Y}| + |\mathbf{X}| - |\mathbf{Y}| \geq |\mathbf{L}| = \mathbf{n}.\end{aligned}$$
5. Any **s**-**t** cut capacity is at least **n** $\implies$ max flow at least **n** units $\implies$ perfect matching. **QED**

Hall's Theorem: Generalization

Theorem (Frobenius-Hall)

Let $\mathbf{G} = (\mathbf{L} \cup \mathbf{R}, \mathbf{E})$ *be a bipartite graph with* $|\mathbf{L}| \leq |\mathbf{R}|$. $\mathbf{G}$ *has a matching that matches all nodes in* $\mathbf{L}$ *if and only if for every* $\mathbf{X} \subseteq \mathbf{L}$, $|\mathbf{N}(\mathbf{X})| \geq |\mathbf{X}|$.

Proof is essentially the same as the previous one.

Assigning jobs to people

1. $\mathbf{n}$ jobs, $\mathbf{n/2}$ people
2. For each job: a set of people who can do that job.
3. Each person $\mathbf{j}$ has to do exactly two jobs.
4. Goal: find an assignment of 2 jobs to each person, such that all jobs are assigned.

Solution: Build bipartite graph, compute maximum matching, remove it, compute another maximum matching. Both matchings together form a valid solution if it exists. This algorithm is

(A) Correct.

(B) Incorrect.

Application: Assigning jobs to people

1. $\mathbf{n}$ jobs or tasks
2. $\mathbf{m}$ people
3. for each job a set of people who can do that job
4. for each person $\mathbf{j}$ a limit on number of jobs $\mathbf{k_j}$
5. Goal: find an assignment of jobs to people so that all jobs are assigned and no person is overloaded

Reduce to max-flow similar to matching.

Arises in many settings. Using *minimum-cost flows* can also handle the case when assigning a job $\mathbf{i}$ to person $\mathbf{j}$ costs $\mathbf{c_{ij}}$ and goal is assign all jobs but minimize cost of assignment.

Application: Assigning jobs to people

1. $\mathbf{n}$ jobs or tasks
2. $\mathbf{m}$ people
3. for each job a set of people who can do that job
4. for each person $\mathbf{j}$ a limit on number of jobs $\mathbf{k_j}$
5. Goal: find an assignment of jobs to people so that all jobs are assigned and no person is overloaded

Reduce to max-flow similar to matching.

Arises in many settings. Using *minimum-cost flows* can also handle the case when assigning a job $\mathbf{i}$ to person $\mathbf{j}$ costs $\mathbf{c_{ij}}$ and goal is assign all jobs but minimize cost of assignment.

Reduction to Maximum Flow

1. Create directed graph $\mathbf{G = (V, E)}$ as follows
   1. $\mathbf{V = \{s, t\} \cup L \cup R}$: $\mathbf{L}$ set of $\mathbf{n}$ jobs, $\mathbf{R}$ set of $\mathbf{m}$ people
   2. add edges $\mathbf{(s, i)}$ for each job $\mathbf{i \in L}$, capacity $\mathbf{1}$
   3. add edges $\mathbf{(j, t)}$ for each person $\mathbf{j \in R}$, capacity $\mathbf{k_j}$
   4. if job $\mathbf{i}$ can be done by person $\mathbf{j}$ add an edge $\mathbf{(i, j)}$, capacity $\mathbf{1}$
2. Compute max $\mathbf{s}$-$\mathbf{t}$ flow. There is an assignment if and only if flow value is $\mathbf{n}$.

Matchings in General Graphs

Matchings in general graphs more complicated.

There is a polynomial time algorithm to compute a maximum matching in a general graph. Best known running time is $\mathbf{O(m\sqrt{n})}$.

Notes

Notes

Notes

Notes

Menger, K. (1927). Zur allgemeinen kruventheorie. *Fund. Math.*, 10:96–115.

More Network Flow Applications

Lecture 19

April 8, 2014

Part I

Baseball Pennant Race

Pennant Race

TUESDAY, SEPTEMBER 10, 1996

San Francisco Chronicle

The Gate
Sports Online
▶ http://www.sfgate.com

SPORTING G

49ers, Young Get Big Brea

Quarterback m

By Gary Swan
Chronicle Staff Writer

The bye week has come at a perfect time for the 49ers and quarterback Steve Young. If they had a game next Sunday, there's a good chance Young would not play.

But the pulled groin muscle on his up-

Giants Officially Leave the NL West Race

By Nancy Gay
Chronicle Staff Writer

With the smack of another National League West bat 500 miles away, the Giants' run at the division title ended last night, just as they were handing the visiting St. Louis Cardinals an even bigger lead in the NL Central.

CARDINALS 6
GIANTS 2

In San Diego, Greg Vaughn's three-run homer in the eighth pushed the Padres over the Pirates and officially shoved the rest of the Giants' season into the background. On the heels of their tedious 6-2 loss before an announced crowd of 10,307 at Candlestick Park, the Giants fell 19½ games off the lead.

As it is, the worst the Padres (80-65) can finish is 80-82. The Giants have fallen to 59-83 with 20 games left; they cannot win 80 games. Coming off a miserable 2-8 mark on a three-city road trip that saw their road record drop to 27-47, the Giants were hoping to get off on the right foot in their longest homestand of the year (15 games, 14 days).

Financing In Place For Giants' New Stadium
SEE PAGE B1, MAIN NEWS

"Where we are, you're going to be eliminated sooner or later," Baker said quietly. "But it doesn't alter the fact that we've still got to play ball. You've still got to play hard, the fans come out to watch you play. You've got to play for the fact of loving to play, no matter where you are in the standings.

"You've got to play the role of spoiler, to not make it easier on

GIANTS: Page D5 Col. 3

Pennant Race: Example

Example

Team	Won	Left
New York	92	2
Baltimore	91	3
Toronto	91	3
Boston	89	2

Can Boston win the pennant?

No, because Boston can win at most 91 games.

Pennant Race: Example

Example

Team	Won	Left
New York	92	2
Baltimore	91	3
Toronto	91	3
Boston	89	2

Can Boston win the pennant?
No, because Boston can win at most 91 games.

Another Example

Example

Team	Won	Left
New York	92	2
Baltimore	91	3
Toronto	91	3
Boston	90	2

Can Boston win the pennant?

Not clear unless we know what the remaining games are!

Another Example

Example

Team	Won	Left
New York	92	2
Baltimore	91	3
Toronto	91	3
Boston	90	2

Can Boston win the pennant?
Not clear unless we know what the remaining games are!

Refining the Example

Example

Team	Won	Left	NY	Bal	Tor	Bos
New York	92	2	–	1	1	0
Baltimore	91	3	1	–	1	1
Toronto	91	3	1	1	–	1
Boston	90	2	0	1	1	–

Can Boston win the pennant? Suppose Boston does

1. Boston wins both its games to get 92 wins
2. New York must lose both games; now both Baltimore and Toronto have at least 92
3. Winner of Baltimore-Toronto game has 93 wins!

Refining the Example

Example

Team	Won	Left	NY	Bal	Tor	Bos
New York	92	2	–	1	1	0
Baltimore	91	3	1	–	1	1
Toronto	91	3	1	1	–	1
Boston	90	2	0	1	1	–

Can Boston win the pennant? Suppose Boston does

1. Boston wins both its games to get 92 wins
2. New York must lose both games; now both Baltimore and Toronto have at least 92
3. Winner of Baltimore-Toronto game has 93 wins!

Refining the Example

Example

Team	Won	Left	NY	Bal	Tor	Bos
New York	92	2	–	1	1	0
Baltimore	91	3	1	–	1	1
Toronto	91	3	1	1	–	1
Boston	90	2	0	1	1	–

Can Boston win the pennant? Suppose Boston does

1. Boston wins both its games to get 92 wins
2. New York must lose both games; now both Baltimore and Toronto have at least 92
3. Winner of Baltimore-Toronto game has 93 wins!

Refining the Example

Example

Team	Won	Left	NY	Bal	Tor	Bos
New York	92	2	–	1	1	0
Baltimore	91	3	1	–	1	1
Toronto	91	3	1	1	–	1
Boston	90	2	0	1	1	–

Can Boston win the pennant? Suppose Boston does

1. Boston wins both its games to get 92 wins
2. New York must lose both games; now both Baltimore and Toronto have at least 92
3. Winner of Baltimore-Toronto game has 93 wins!

Refining the Example

Example

Team	Won	Left	NY	Bal	Tor	Bos
New York	92	2	–	1	1	0
Baltimore	91	3	1	–	1	1
Toronto	91	3	1	1	–	1
Boston	90	2	0	1	1	–

Can Boston win the pennant? Suppose Boston does

1. Boston wins both its games to get 92 wins
2. New York must lose both games; now both Baltimore and Toronto have at least 92
3. Winner of Baltimore-Toronto game has 93 wins!

Refining the Example

Example

Team	Won	Left	NY	Bal	Tor	Bos
New York	92	2	–	1	1	0
Baltimore	91	3	1	–	1	1
Toronto	91	3	1	1	–	1
Boston	90	2	0	1	1	–

Can Boston win the pennant? Suppose Boston does

1. Boston wins both its games to get 92 wins
2. New York must lose both games; now both Baltimore and Toronto have at least 92
3. Winner of Baltimore-Toronto game has 93 wins!

Can Boston win the penant?

Team	Won	Left	NY	Bal	Tor	Bos
New York	3	6	—	2	3	1
Baltimore	5	4	2	—	1	1
Toronto	4	6	3	1	—	2
Boston	2	4	1	1	2	—

(A) Yes.

(B) No.

Abstracting the Problem

Given

1. A set of teams $\mathbf{S}$
2. For each $\mathbf{x} \in \mathbf{S}$, the current number of wins $\mathbf{w_x}$
3. For any $\mathbf{x}, \mathbf{y} \in \mathbf{S}$, the number of remaining games $\mathbf{g_{xy}}$ between $\mathbf{x}$ and $\mathbf{y}$
4. A team $\mathbf{z}$

Can $\mathbf{z}$ win the pennant?

Towards a Reduction

$\bar{z}$ can win the pennant if

1. $\bar{z}$ wins at least m games
2. no other team wins more than m games

Towards a Reduction

$\bar{z}$ can win the pennant if

1. $\bar{z}$ wins at least $\mathbf{m}$ games
   1. to maximize $\bar{z}$'s chances we make $\bar{z}$ win all its remaining games and hence $\mathbf{m} = \mathbf{w}_{\bar{z}} + \sum_{x \in S} \mathbf{g}_{x\bar{z}}$
2. no other team wins more than $\mathbf{m}$ games

Towards a Reduction

$\bar{z}$ can win the pennant if

1. $\bar{z}$ wins at least m games
   1. to maximize $\bar{z}$'s chances we make $\bar{z}$ win all its remaining games and hence $m = w_{\bar{z}} + \sum_{x \in S} g_{x\bar{z}}$
2. no other team wins more than m games
   1. for each $x, y \in S$ the g_{xy} games between them have to be *assigned* to either x or y.
   2. each team $x \neq \bar{z}$ can win at most $m - w_x - g_{x\bar{z}}$ remaining games

Is there an assignment of remaining games to teams such that no team $x \neq \bar{z}$ wins more than $m - w_x$ games?

Flow Network: The basic gadget

1. **s**: source
2. **t**: sink
3. **x**, **y**: two teams
4. $\mathbf{g_{xy}}$: number of games remaining between **x** and **y**.
5. $\mathbf{w_x}$: number of points **x** has.
6. **m**: maximum number of points **x** can win before team of interest is eliminated.

Flow Network: An Example

Can Boston win?

Team	Won	Left	NY	Bal	Tor	Bos
New York	90	11	–	1	6	4
Baltimore	88	6	1	–	1	4
Toronto	87	11	6	1	–	4
Boston	79	12	4	4	4	–

1. $\mathbf{m = 79 + 12 = 91}$: Boston can get at most **91** points.

Constructing Flow Network

Notations

1. S: set of teams,
2. w_x wins for each team, and
3. g_{xy} games left between x and y.
4. m be the maximum number of wins for $\bar{z}$,
5. and $S' = S \setminus \{\bar{z}\}$.

Reduction

Construct the flow network G as follows

1. One vertex v_x for each team $x \in S'$, one vertex u_{xy} for each pair of teams x and y in S'
2. A new source vertex s and sink t
3. Edges (u_{xy}, v_x) and (u_{xy}, v_y) of capacity ∞
4. Edges (s, u_{xy}) of capacity g_{xy}
5. Edges (v_x, t) of capacity equal $m - w_x$

Correctness of reduction

Theorem

$\mathbf{G'}$ *has a maximum flow of value* $\mathbf{g}^* = \sum_{\mathbf{x},\mathbf{y} \in \mathbf{S'}} \mathbf{g}_{\mathbf{xy}}$ *if and only if* $\bar{\mathbf{z}}$ *can win the most number of games (including possibly tie with other teams).*

Proof of Correctness

Proof.

Existence of g^* flow $\Rightarrow$ $\bar{z}$ wins pennant

1. An integral flow saturating edges out of s, ensures that each remaining game between x and y is added to win total of either x or y
2. Capacity on (v_x, t) edges ensures that no team wins more than m games

Conversely, $\bar{z}$ wins pennant $\Rightarrow$ flow of value g^*

1. Scenario determines flow on edges; if x wins k of the games against y, then flow on (u_{xy}, v_x) edge is k and on (u_{xy}, v_y) edge is $g_{xy} - k$ $\square$

Proof that $\bar{z}$ cannot with the pennant

1. Suppose $\bar{z}$ cannot win the pennant since $\mathbf{g}^* < \mathbf{g}$. How do we *prove* to some one *compactly* that $\bar{z}$ cannot win the pennant?
2. Show them the min-cut in the reduction flow network!
3. See Kleinberg-Tardos book for a natural interpretation of the min-cut as a certificate.

Proof that $\bar{z}$ cannot with the pennant

1. Suppose $\bar{z}$ cannot win the pennant since $\mathbf{g}^* < \mathbf{g}$. How do we *prove* to some one *compactly* that $\bar{z}$ cannot win the pennant?
2. Show them the min-cut in the reduction flow network!
3. See Kleinberg-Tardos book for a natural interpretation of the min-cut as a certificate.

Proof that $\bar{\mathbf{z}}$ cannot with the pennant

1. Suppose $\bar{\mathbf{z}}$ cannot win the pennant since $\mathbf{g}^* < \mathbf{g}$. How do we *prove* to some one *compactly* that $\bar{\mathbf{z}}$ cannot win the pennant?
2. Show them the min-cut in the reduction flow network!
3. See Kleinberg-Tardos book for a natural interpretation of the min-cut as a certificate.

The biggest loser?

Given an input as above for the pennant competition, deciding if a team can come in the last place can be done in

(A) Can be done using the same reduction as just seen.

(B) Can not be done using the same reduction as just seen.

(C) Can be done using flows but we need lower bounds on the flow, instead of upper bounds.

(D) The problem is **NP-Hard** and requires exponential time.

(E) Can be solved by negating all the numbers, and using the above reduction.

(F) Can be solved efficiently only by running a reality show on the problem.

Part II

An Application of Min-Cut to Project Scheduling

Project Scheduling

Problem:

1. $\mathbf{n}$ projects/tasks $\mathbf{1, 2, \ldots, n}$
2. *dependencies* between projects: $\mathbf{i}$ depends on $\mathbf{j}$ implies $\mathbf{i}$ cannot be done unless $\mathbf{j}$ is done. dependency graph is *acyclic*
3. each project $\mathbf{i}$ has a cost/profit $\mathbf{p_i}$
   1. $\mathbf{p_i} < \mathbf{0}$ implies $\mathbf{i}$ requires a cost of $-\mathbf{p_i}$ units
   2. $\mathbf{p_i} > \mathbf{0}$ implies that $\mathbf{i}$ generates $\mathbf{p_i}$ profit

Goal: Find projects to do so as to *maximize* profit.

Example

Notation

For a set $\mathbf{A}$ of projects:

1. $\mathbf{A}$ is a *valid* solution if $\mathbf{A}$ is *dependency closed*, that is for every $\mathbf{i} \in \mathbf{A}$, all projects that $\mathbf{i}$ depends on are also in $\mathbf{A}$.
2. $\mathbf{profit(A)} = \sum_{\mathbf{i} \in \mathbf{A}} \mathbf{p_i}$. Can be negative or positive.

Goal: find valid $\mathbf{A}$ to maximize $\mathbf{profit(A)}$.

Notation

For a set $\mathbf{A}$ of projects:

1. $\mathbf{A}$ is a *valid* solution if $\mathbf{A}$ is *dependency closed*, that is for every $\mathbf{i} \in \mathbf{A}$, all projects that $\mathbf{i}$ depends on are also in $\mathbf{A}$.
2. $\mathbf{profit(A)} = \sum_{\mathbf{i} \in \mathbf{A}} \mathbf{p_i}$. Can be negative or positive.

Goal: find valid $\mathbf{A}$ to maximize $\mathbf{profit(A)}$.

Notation

For a set $\mathbf{A}$ of projects:

1. $\mathbf{A}$ is a *valid* solution if $\mathbf{A}$ is *dependency closed*, that is for every $\mathbf{i} \in \mathbf{A}$, all projects that $\mathbf{i}$ depends on are also in $\mathbf{A}$.
2. $\mathbf{profit(A)} = \sum_{\mathbf{i} \in \mathbf{A}} \mathbf{p_i}$. Can be negative or positive.

Goal: find valid $\mathbf{A}$ to maximize $\mathbf{profit(A)}$.

Idea: Reduction to Minimum-Cut

Finding a set of projects is partitioning the projects into two sets: those that are done and those that are not done.

Can we express this is a minimum cut problem?

Several issues:

1. We are interested in maximizing profit but we can solve minimum cuts.
2. We need to convert negative profits into positive capacities.
3. Need to ensure that chosen projects is a valid set.
4. The cut value captures the profit of the chosen set of projects.

Idea: Reduction to Minimum-Cut

Finding a set of projects is partitioning the projects into two sets: those that are done and those that are not done.

Can we express this is a minimum cut problem?

Several issues:

1. We are interested in maximizing profit but we can solve minimum cuts.
2. We need to convert negative profits into positive capacities.
3. Need to ensure that chosen projects is a valid set.
4. The cut value captures the profit of the chosen set of projects.

Reduction to Minimum-Cut

Note: We are reducing a *maximization* problem to a *minimization* problem.

1. projects represented as nodes in a graph
2. if $\mathbf{i}$ depends on $\mathbf{j}$ then $\mathbf{(i, j)}$ is an edge
3. add source $\mathbf{s}$ and sink $\mathbf{t}$
4. for each $\mathbf{i}$ with $\mathbf{p_i > 0}$ add edge $\mathbf{(s, i)}$ with capacity $\mathbf{p_i}$
5. for each $\mathbf{i}$ with $\mathbf{p_i < 0}$ add edge $\mathbf{(i, t)}$ with capacity $\mathbf{-p_i}$
6. for each dependency edge $\mathbf{(i, j)}$ put capacity ∞ (more on this later)

Reduction: Flow Network Example

Reduction contd

Algorithm:

1. form graph as in previous slide
2. compute **s**-**t** minimum cut $(\mathbf{A}, \mathbf{B})$
3. output the projects in $\mathbf{A} - \{\mathbf{s}\}$

Understanding the Reduction

Let $\mathbf{C} = \sum_{\mathbf{i}:\mathbf{p_i}>0} \mathbf{p_i}$: maximum possible profit.

Observation: The minimum $\mathbf{s}$-$\mathbf{t}$ cut value is $\leq \mathbf{C}$. Why?

Lemma

Suppose $(\mathbf{A}, \mathbf{B})$ *is an* $\mathbf{s}$-$\mathbf{t}$ *cut of finite capacity (no* ∞*) edges. Then projects in* $\mathbf{A} - \{\mathbf{s}\}$ *are a* valid *solution.*

Proof.

If $\mathbf{A} - \{\mathbf{s}\}$ is not a valid solution then there is a project $\mathbf{i} \in \mathbf{A}$ and a project $\mathbf{j} \notin \mathbf{A}$ such that $\mathbf{i}$ depends on $\mathbf{j}$

Since $(\mathbf{i}, \mathbf{j})$ capacity is ∞, implies $(\mathbf{A}, \mathbf{B})$ capacity is ∞, contradicting assumption. □

Understanding the Reduction

Let $\mathbf{C} = \sum_{\mathbf{i}:\mathbf{p_i}>0} \mathbf{p_i}$: maximum possible profit.

Observation: The minimum **s**-**t** cut value is $\leq \mathbf{C}$. Why?

Lemma

Suppose $(\mathbf{A}, \mathbf{B})$ *is an* **s**-**t** *cut of finite capacity (no* ∞*) edges. Then projects in* $\mathbf{A} - \{\mathbf{s}\}$ *are a* valid *solution.*

Proof.

If $\mathbf{A} - \{\mathbf{s}\}$ is not a valid solution then there is a project $\mathbf{i} \in \mathbf{A}$ and a project $\mathbf{j} \notin \mathbf{A}$ such that $\mathbf{i}$ depends on $\mathbf{j}$

Since $(\mathbf{i}, \mathbf{j})$ capacity is ∞, implies $(\mathbf{A}, \mathbf{B})$ capacity is ∞, contradicting assumption. □

Understanding the Reduction

Let $\mathbf{C} = \sum_{\mathbf{i}:\mathbf{p_i}>0} \mathbf{p_i}$: maximum possible profit.

Observation: The minimum **s**-**t** cut value is $\leq \mathbf{C}$. Why?

Lemma

Suppose $(\mathbf{A}, \mathbf{B})$ *is an* **s**-**t** *cut of finite capacity (no* ∞*) edges. Then projects in* $\mathbf{A} - \{\mathbf{s}\}$ *are a* valid *solution.*

Proof.

If $\mathbf{A} - \{\mathbf{s}\}$ is not a valid solution then there is a project $\mathbf{i} \in \mathbf{A}$ and a project $\mathbf{j} \notin \mathbf{A}$ such that $\mathbf{i}$ depends on $\mathbf{j}$

Since $(\mathbf{i}, \mathbf{j})$ capacity is ∞, implies $(\mathbf{A}, \mathbf{B})$ capacity is ∞, contradicting assumption. □

Understanding the Reduction

Let $\mathbf{C} = \sum_{\mathbf{i}:\mathbf{p_i}>0} \mathbf{p_i}$: maximum possible profit.

Observation: The minimum **s**-**t** cut value is $\leq \mathbf{C}$. Why?

Lemma

Suppose $(\mathbf{A}, \mathbf{B})$ *is an* **s**-**t** *cut of finite capacity (no* ∞*) edges. Then projects in* $\mathbf{A} - \{\mathbf{s}\}$ *are a* valid *solution.*

Proof.

If $\mathbf{A} - \{\mathbf{s}\}$ is not a valid solution then there is a project $\mathbf{i} \in \mathbf{A}$ and a project $\mathbf{j} \notin \mathbf{A}$ such that $\mathbf{i}$ depends on $\mathbf{j}$

Since $(\mathbf{i}, \mathbf{j})$ capacity is ∞, implies $(\mathbf{A}, \mathbf{B})$ capacity is ∞, contradicting assumption. □

Understanding the Reduction

Let $\mathbf{C} = \sum_{\mathbf{i}:\mathbf{p_i}>0} \mathbf{p_i}$: maximum possible profit.

Observation: The minimum **s**-**t** cut value is $\leq \mathbf{C}$. Why?

Lemma

Suppose $(\mathbf{A}, \mathbf{B})$ *is an* **s**-**t** *cut of finite capacity (no* ∞*) edges. Then projects in* $\mathbf{A} - \{\mathbf{s}\}$ *are a* valid *solution.*

Proof.

If $\mathbf{A} - \{\mathbf{s}\}$ is not a valid solution then there is a project $\mathbf{i} \in \mathbf{A}$ and a project $\mathbf{j} \notin \mathbf{A}$ such that $\mathbf{i}$ depends on $\mathbf{j}$

Since $(\mathbf{i}, \mathbf{j})$ capacity is ∞, implies $(\mathbf{A}, \mathbf{B})$ capacity is ∞, contradicting assumption. $\square$

Example

Example

Correctness of Reduction

Recall that for a set of projects $\mathbf{X}$, $\mathbf{profit(X)} = \sum_{i \in X} p_i$.

Lemma

Suppose (A, B) *is an* s-t *cut of finite capacity (no* ∞*) edges. Then* $c(A, B) = C - \text{profit}(A - \{s\})$.

Proof.

Edges in (A, B):

1. (s, i) for $i \in B$ and $p_i > 0$: capacity is p_i
2. (i, t) for $i \in A$ and $p_i < 0$: capacity is $-p_i$
3. cannot have ∞ edges

□

Correctness of Reduction

Recall that for a set of projects $\mathbf{X}$, $\mathbf{profit(X)} = \sum_{i \in X} p_i$.

Lemma

Suppose (A, B) *is an* s-t *cut of finite capacity (no* ∞*) edges. Then* $c(A, B) = C - \mathbf{profit}(A - \{s\})$.

Proof.

Edges in (A, B):

1. (s, i) for $i \in B$ and $p_i > 0$: capacity is p_i
2. (i, t) for $i \in A$ and $p_i < 0$: capacity is $-p_i$
3. cannot have ∞ edges

□

Correctness of Reduction

Recall that for a set of projects $\mathbf{X}$, $\mathbf{profit(X)} = \sum_{i \in X} p_i$.

Lemma

Suppose (A, B) *is an* s-t *cut of finite capacity (no* ∞*) edges. Then* $c(A, B) = C - profit(A - \{s\})$.

Proof.

Edges in (A, B):

1. (s, i) for $i \in B$ and $p_i > 0$: capacity is p_i
2. (i, t) for $i \in A$ and $p_i < 0$: capacity is $-p_i$
3. cannot have ∞ edges

□

Proof contd

For project set $\mathbf{A}$ let

1. $\mathbf{cost(A)} = \sum_{\mathbf{i \in A : p_i < 0}} -\mathbf{p_i}$
2. $\mathbf{benefit(A)} = \sum_{\mathbf{i \in A : p_i > 0}} \mathbf{p_i}$
3. $\mathbf{profit(A)} = \mathbf{benefit(A)} - \mathbf{cost(A)}$.

Proof.

Let $\mathbf{A'} = \mathbf{A} \cup \{\mathbf{s}\}$.

$$\begin{aligned}
\mathbf{c(A', B)} &= \mathbf{cost(A)} + \mathbf{benefit(B)} \\
&= \mathbf{cost(A)} - \mathbf{benefit(A)} + \mathbf{benefit(A)} + \mathbf{benefit(B)} \\
&= -\mathbf{profit(A)} + \mathbf{C} \\
&= \mathbf{C} - \mathbf{profit(A)}
\end{aligned}$$

□

Correctness of Reduction contd

We have shown that if $(\mathbf{A}, \mathbf{B})$ is an $\mathbf{s}$-$\mathbf{t}$ cut in $\mathbf{G}$ with finite capacity then

1. $\mathbf{A} - \{\mathbf{s}\}$ is a valid set of projects
2. $\mathbf{c}(\mathbf{A}, \mathbf{B}) = \mathbf{C} - \mathbf{profit}(\mathbf{A} - \{\mathbf{s}\})$

Therefore a *minimum* $\mathbf{s}$-$\mathbf{t}$ cut $(\mathbf{A}^*, \mathbf{B}^*)$ gives a *maximum* profit set of projects $\mathbf{A}^* - \{\mathbf{s}\}$ since $\mathbf{C}$ is fixed.

Question: How can we use ∞ in a real algorithm?

Set capacity of ∞ arcs to $\mathbf{C} + \mathbf{1}$ instead. Why does this work?

Correctness of Reduction contd

We have shown that if $(\mathbf{A}, \mathbf{B})$ is an $\mathbf{s}$-$\mathbf{t}$ cut in $\mathbf{G}$ with finite capacity then

1. $\mathbf{A} - \{\mathbf{s}\}$ is a valid set of projects
2. $\mathbf{c}(\mathbf{A}, \mathbf{B}) = \mathbf{C} - \mathbf{profit}(\mathbf{A} - \{\mathbf{s}\})$

Therefore a *minimum* $\mathbf{s}$-$\mathbf{t}$ cut $(\mathbf{A}^*, \mathbf{B}^*)$ gives a *maximum* profit set of projects $\mathbf{A}^* - \{\mathbf{s}\}$ since $\mathbf{C}$ is fixed.

Question: How can we use ∞ in a real algorithm?

Set capacity of ∞ arcs to $\mathbf{C} + \mathbf{1}$ instead. Why does this work?

Correctness of Reduction contd

We have shown that if $(\mathbf{A}, \mathbf{B})$ is an $\mathbf{s}$-$\mathbf{t}$ cut in $\mathbf{G}$ with finite capacity then

1. $\mathbf{A} - \{\mathbf{s}\}$ is a valid set of projects
2. $\mathbf{c}(\mathbf{A}, \mathbf{B}) = \mathbf{C} - \mathbf{profit}(\mathbf{A} - \{\mathbf{s}\})$

Therefore a *minimum* $\mathbf{s}$-$\mathbf{t}$ cut $(\mathbf{A}^*, \mathbf{B}^*)$ gives a *maximum* profit set of projects $\mathbf{A}^* - \{\mathbf{s}\}$ since $\mathbf{C}$ is fixed.

Question: How can we use ∞ in a real algorithm?

Set capacity of ∞ arcs to $\mathbf{C} + \mathbf{1}$ instead. Why does this work?

Correctness of Reduction contd

We have shown that if $(\mathbf{A}, \mathbf{B})$ is an $\mathbf{s}$-$\mathbf{t}$ cut in $\mathbf{G}$ with finite capacity then

1. $\mathbf{A} - \{\mathbf{s}\}$ is a valid set of projects
2. $\mathbf{c}(\mathbf{A}, \mathbf{B}) = \mathbf{C} - \mathbf{profit}(\mathbf{A} - \{\mathbf{s}\})$

Therefore a *minimum* $\mathbf{s}$-$\mathbf{t}$ cut $(\mathbf{A}^*, \mathbf{B}^*)$ gives a *maximum* profit set of projects $\mathbf{A}^* - \{\mathbf{s}\}$ since $\mathbf{C}$ is fixed.

Question: How can we use ∞ in a real algorithm?

Set capacity of ∞ arcs to $\mathbf{C} + \mathbf{1}$ instead. Why does this work?

Shortest path always present?

Let $\mathbf{G}$ be an directed graph, and let $\mathbf{\Pi} = \{\pi_1, \ldots, \pi_k\}$ be the (largest) set of edge disjoint paths in $\mathbf{G}$ from $\mathbf{s}$ to $\mathbf{t}$, computed using network flow.

(A) The shortest path in $\mathbf{G}$ must be one of the paths in $\mathbf{\Pi}$.

(B) The shortest path in $\mathbf{G}$ must intersect exactly <u>one</u> of the paths in $\mathbf{\Pi}$.

(C) The shortest path in $\mathbf{G}$ must intersects <u>all</u> of the paths in $\mathbf{\Pi}$.

(D) The shortest path in $\mathbf{G}$ must intersects at least one of the paths in $\mathbf{\Pi}$, and it can intersect all of them.

Part III

Extensions to Maximum-Flow Problem

Lower Bounds and Costs

Two generalizations:

1. flow satisfies $\mathbf{f(e)} \leq \mathbf{c(e)}$ for all $\mathbf{e}$. suppose we are given *lower bounds* $\ell(\mathbf{e})$ for each $\mathbf{e}$. can we find a flow such that $\ell(\mathbf{e}) \leq \mathbf{f(e)} \leq \mathbf{c(e)}$ for all $\mathbf{e}$?
2. suppose we are given a cost $\mathbf{w(e)}$ for each edge. cost of routing flow $\mathbf{f(e)}$ on edge $\mathbf{e}$ is $\mathbf{w(e)f(e)}$. can we (efficiently) find a flow (of at least some given quantity) at minimum cost?

Many applications.

Lower Bounds and Costs

Two generalizations:

1. flow satisfies $\mathbf{f(e)} \leq \mathbf{c(e)}$ for all $\mathbf{e}$. suppose we are given *lower bounds* $\ell(\mathbf{e})$ for each $\mathbf{e}$. can we find a flow such that $\ell(\mathbf{e}) \leq \mathbf{f(e)} \leq \mathbf{c(e)}$ for all $\mathbf{e}$?
2. suppose we are given a cost $\mathbf{w(e)}$ for each edge. cost of routing flow $\mathbf{f(e)}$ on edge $\mathbf{e}$ is $\mathbf{w(e)f(e)}$. can we (efficiently) find a flow (of at least some given quantity) at minimum cost?

Many applications.

Lower Bounds and Costs

Two generalizations:

1. flow satisfies $\mathbf{f(e)} \leq \mathbf{c(e)}$ for all $\mathbf{e}$. suppose we are given *lower bounds* $\ell(\mathbf{e})$ for each $\mathbf{e}$. can we find a flow such that $\ell(\mathbf{e}) \leq \mathbf{f(e)} \leq \mathbf{c(e)}$ for all $\mathbf{e}$?
2. suppose we are given a cost $\mathbf{w(e)}$ for each edge. cost of routing flow $\mathbf{f(e)}$ on edge $\mathbf{e}$ is $\mathbf{w(e)f(e)}$. can we (efficiently) find a flow (of at least some given quantity) at minimum cost?

Many applications.

Flows with Lower Bounds

Definition

A flow in a network $\mathbf{G = (V, E)}$, is a function $\mathbf{f : E \to \mathbb{R}^{\geq 0}}$ such that

1. Capacity Constraint: For each edge $\mathbf{e}$, $\mathbf{f(e) \leq c(e)}$
2. Lower Bound Constraint: For each edge $\mathbf{e}$, $\mathbf{f(e) \geq \ell(e)}$
3. Conservation Constraint: For each vertex $\mathbf{v}$

$$\sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f(e)} = \sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f(e)}$$

Question: Given $\mathbf{G}$ and $\mathbf{c(e)}$ and $\ell(\mathbf{e})$ for each $\mathbf{e}$, is there a flow?
As difficult as finding an $\mathbf{s}$-$\mathbf{t}$ maximum-flow without lower bounds!

Flows with Lower Bounds

Definition

A flow in a network $\mathbf{G = (V, E)}$, is a function $\mathbf{f : E \to \mathbb{R}^{\geq 0}}$ such that

1. Capacity Constraint: For each edge $\mathbf{e}$, $\mathbf{f(e) \leq c(e)}$
2. Lower Bound Constraint: For each edge $\mathbf{e}$, $\mathbf{f(e) \geq \ell(e)}$
3. Conservation Constraint: For each vertex $\mathbf{v}$

$$\sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f(e)} = \sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f(e)}$$

Question: Given $\mathbf{G}$ and $\mathbf{c(e)}$ and $\boldsymbol{\ell}\mathbf{(e)}$ for each $\mathbf{e}$, is there a flow?

As difficult as finding an $\mathbf{s}$-$\mathbf{t}$ maximum-flow without lower bounds!

Flows with Lower Bounds

Definition

A flow in a network $\mathbf{G = (V, E)}$, is a function $\mathbf{f : E \to \mathbb{R}^{\geq 0}}$ such that

1. Capacity Constraint: For each edge $\mathbf{e}$, $\mathbf{f(e) \leq c(e)}$
2. Lower Bound Constraint: For each edge $\mathbf{e}$, $\mathbf{f(e) \geq \ell(e)}$
3. Conservation Constraint: For each vertex $\mathbf{v}$

$$\sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f(e)} = \sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f(e)}$$

Question: Given $\mathbf{G}$ and $\mathbf{c(e)}$ and $\boldsymbol{\ell}\mathbf{(e)}$ for each $\mathbf{e}$, is there a flow?
As difficult as finding an $\mathbf{s}$-$\mathbf{t}$ maximum-flow without lower bounds!

Regular flow via lower bounds

Given usual flow network $\mathbf{G}$ with source $\mathbf{s}$ and sink $\mathbf{t}$, create lower-bound flow network $\mathbf{G'}$ as follows:

1. set $\ell(\mathbf{e}) = \mathbf{0}$ for each $\mathbf{e}$ in $\mathbf{G}$
2. add new edge $(\mathbf{t}, \mathbf{s})$ with lower bound $\mathbf{v}$ and upper bound ∞

Claim

There exists a flow of value $\mathbf{v}$ *from* $\mathbf{s}$ *to* $\mathbf{t}$ *in* $\mathbf{G}$ *if and only if there exists a feasible flow with lower bounds in* $\mathbf{G'}$.

Above reduction show that lower bounds on flows are naturally related to circulations. With lower bounds, cannot guarantee acyclic flows from $\mathbf{s}$ to $\mathbf{t}$.

Regular flow via lower bounds

Given usual flow network $\mathbf{G}$ with source $\mathbf{s}$ and sink $\mathbf{t}$, create lower-bound flow network $\mathbf{G'}$ as follows:

1. set $\ell(\mathbf{e}) = \mathbf{0}$ for each $\mathbf{e}$ in $\mathbf{G}$
2. add new edge $(\mathbf{t}, \mathbf{s})$ with lower bound $\mathbf{v}$ and upper bound ∞

Claim

There exists a flow of value $\mathbf{v}$ *from* $\mathbf{s}$ *to* $\mathbf{t}$ *in* $\mathbf{G}$ *if and only if there exists a feasible flow with lower bounds in* $\mathbf{G'}$.

Above reduction show that lower bounds on flows are naturally related to circulations. With lower bounds, cannot guarantee acyclic flows from $\mathbf{s}$ to $\mathbf{t}$.

Regular flow via lower bounds

Given usual flow network $\mathbf{G}$ with source $\mathbf{s}$ and sink $\mathbf{t}$, create lower-bound flow network $\mathbf{G'}$ as follows:

1. set $\ell(\mathbf{e}) = \mathbf{0}$ for each $\mathbf{e}$ in $\mathbf{G}$
2. add new edge $(\mathbf{t}, \mathbf{s})$ with lower bound $\mathbf{v}$ and upper bound ∞

Claim

There exists a flow of value $\mathbf{v}$ *from* $\mathbf{s}$ *to* $\mathbf{t}$ *in* $\mathbf{G}$ *if and only if there* exists *a feasible flow with lower bounds in* $\mathbf{G'}$.

Above reduction show that lower bounds on flows are naturally related to circulations. With lower bounds, cannot guarantee acyclic flows from $\mathbf{s}$ to $\mathbf{t}$.

Flows with Lower Bounds

1. Flows with lower bounds can be reduced to standard maximum flow problem. See Kleinberg-Tardos book. Reduction goes via circulations.
2. If all bounds are integers then there is a flow that is integral. Useful in applications.

Combining max flows?

Given distinct max flows $\mathbf{f}$ and $\mathbf{g}$ in $\mathbf{G}$, the function $\mathbf{h(e) = (f(e) + g(e))/2}$, for all $\mathbf{e} \in \mathbf{E(G)}$, describes a valid max flow in $\mathbf{G}$. This is

(A) True.

(B) False.

Survey Design

Application of Flows with Lower Bounds

1. Design survey to find information about $\mathbf{n_1}$ products from $\mathbf{n_2}$ customers.
2. Can ask customer questions only about products purchased in the past.
3. Customer can only be asked about at most $\mathbf{c_i'}$ products and at least $\mathbf{c_i}$ products.
4. For each product need to ask at east $\mathbf{p_i}$ consumers and at most $\mathbf{p_i'}$ consumers.

Reduction to Circulation

1. include edge (i, j) is customer i has bought product j
2. Add edge (t, s) with lower bound 0 and upper bound ∞.
   1. Consumer i is asked about product j if the integral flow on edge (i, j) is 1

Minimum Cost Flows

1. **Input:** Given a flow network $\mathbf{G}$ and also edge costs, $\mathbf{w(e)}$ for edge $\mathbf{e}$, and a flow requirement $\mathbf{F}$.
2. **Goal;** Find a *minimum cost* flow of value $\mathbf{F}$ from $\mathbf{s}$ to $\mathbf{t}$

Given flow $\mathbf{f : E \rightarrow R^+}$, cost of flow $= \sum_{\mathbf{e \in E}} \mathbf{w(e)f(e)}$.

Much more geeneral than the shortest path problem.

Minimum Cost Flows

1. **Input:** Given a flow network $\mathbf{G}$ and also edge costs, $\mathbf{w(e)}$ for edge $\mathbf{e}$, and a flow requirement $\mathbf{F}$.
2. **Goal;** Find a *minimum cost* flow of value $\mathbf{F}$ from $\mathbf{s}$ to $\mathbf{t}$

Given flow $\mathbf{f : E \rightarrow R^+}$, cost of flow $= \sum_{\mathbf{e \in E}} \mathbf{w(e)f(e)}$.

Much more geeneral than the shortest path problem.

Minimum Cost Flow: Facts

1. problem can be solved efficiently in polynomial time
   1. $\mathbf{O(nm \log C \log(nW))}$ time algorithm where $\mathbf{C}$ is maximum edge capacity and $\mathbf{W}$ is maximum edge cost
   2. $\mathbf{O(m \log n(m + n \log n))}$ time strongly polynomial time algorithm
2. for integer capacities there is always an optimum solutions in which flow is integral

Min-Cost Flow: Successive Shortest Path Alg

Goal: Given $\mathbf{G}$ with integer capacities, non-negative weights, and integer $\mathbf{k}$, find $\mathbf{s}$-$\mathbf{t}$ flow of value $\mathbf{k}$ with minimum cost.

Successive-Shortest-Path-Alg

```
for every edge e, f(e) = 0
G_{f,w} is residual graph of G with respect to f
while v(f) < k and G_{f,w} has a simple s-t path do
    let P be a shortest s-t path in G_{f,w}
    Augment one unit of flow along P and update f
    Construct new residual graph G_{f,w}.
```

Residular graph $\mathbf{G_{f,w}}$: similar to $\mathbf{G_f}$, for edge $\mathbf{e}$ with cost $\mathbf{w(e)}$, cost of forward edge is $\mathbf{w(e)}$ and cost of backward edge is $-\mathbf{w(e)}$.

Algorithm gives optimum solution. Shows existence of integral optimum solution for integer capacities. Run time is $\mathbf{O(mk \log m)}$, and in the worst-case, $\mathbf{O(mC \log m)}$.

How much damage can a single path cause?

Consider the following network. All the edges have capacity **1**. Clearly the maximum flow in this network has value **4**.

The network

Why removing the shortest path might ruin everything

1. However... The shortest path between **s** and **t** is the blue path.
2. And if we remove the shortest path, **s** and **t** become disconnected, and the maximum flow drop to **0**.

Notes

Notes

Notes

Notes

Polynomial Time Reductions

Lecture 20

April 15, 2014

Part I

Introduction to Reductions

Reductions

A reduction from Problem **X** to Problem **Y** means (informally) that if we have an algorithm for Problem **Y**, we can use it to find an algorithm for Problem **X**.

Reductions

A reduction from Problem **X** to Problem **Y** means (informally) that if we have an algorithm for Problem **Y**, we can use it to find an algorithm for Problem **X**.

Using Reductions

1. We use reductions to find algorithms to solve problems.

Reductions

A reduction from Problem **X** to Problem **Y** means (informally) that if we have an algorithm for Problem **Y**, we can use it to find an algorithm for Problem **X**.

Using Reductions

1. We use reductions to find algorithms to solve problems.
2. We also use reductions to show that we can't find algorithms for some problems. (We say that these problems are hard.)

Reductions

A reduction from Problem **X** to Problem **Y** means (informally) that if we have an algorithm for Problem **Y**, we can use it to find an algorithm for Problem **X**.

Using Reductions

1. We use reductions to find algorithms to solve problems.
2. We also use reductions to show that we can't find algorithms for some problems. (We say that these problems are hard.)

Also, the right reductions might win you a million dollars!

Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?

Given a bipartite graph $\mathbf{G = (U \cup V, E)}$ and number $\mathbf{k}$, does $\mathbf{G}$ have a matching of size $\geq \mathbf{k}$?

Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?

Given a bipartite graph $\mathbf{G = (U \cup V, E)}$ and number $\mathbf{k}$, does $\mathbf{G}$ have a matching of size $\geq \mathbf{k}$?

Example 1: Bipartite Matching and Flows

How do we solve the Bipartite Matching Problem?

Given a bipartite graph $\mathbf{G = (U \cup V, E)}$ and number $\mathbf{k}$, does $\mathbf{G}$ have a matching of size $\geq \mathbf{k}$?

Example 1: Bipartite Matching and Flows

How do we solve the **Bipartite Matching** Problem?

Given a bipartite graph $\mathbf{G = (U \cup V, E)}$ and number $\mathbf{k}$, does $\mathbf{G}$ have a matching of size $\geq \mathbf{k}$?

Solution

Reduce it to **Max-Flow**. G has a matching of size $\geq \mathbf{k}$ iff there is a flow from $\mathbf{s}$ to $\mathbf{t}$ of value $\geq \mathbf{k}$.

Types of Problems

Decision, Search, and Optimization

1. **Decision problem**. Example: given **n**, is **n** prime?.

Types of Problems

Decision, Search, and Optimization

1. **Decision problem**. Example: given **n**, is **n** prime?.
2. **Search problem**. Example: given **n**, find a factor of **n** if it exists.

Types of Problems

Decision, Search, and Optimization

1. **Decision problem**. Example: given **n**, is **n** prime?.
2. **Search problem**. Example: given **n**, find a factor of **n** if it exists.
3. **Optimization problem**. Example: find the smallest prime factor of **n**.

Optimization and Decision problems

For max flow...

Problem (Max-Flow optimization version)

Given an instance G of network flow, find the maximum flow between **s** *and* **t**.

Optimization and Decision problems

For max flow...

Problem (Max-Flow optimization version)

Given an instance G of network flow, find the maximum flow between **s** *and* **t**.

Problem (Max-Flow decision version)

Given an instance G of network flow and a parameter **K**, *is there a flow in G, from* **s** *to* **t**, *of value at least* **K**?

Optimization and Decision problems

For max flow...

Problem (Max-Flow optimization version)

Given an instance G of network flow, find the maximum flow between **s** *and* **t**.

Problem (Max-Flow decision version)

Given an instance G of network flow and a parameter **K**, *is there a flow in G, from* **s** *to* **t**, *of value at least* **K**?

While using reductions and comparing problems, we typically work with the decision versions. Decision problems have Yes/No answers. This makes them easy to work with.

Problems vs Instances

1. A problem Π consists of an **infinite** collection of inputs $\{I_1, I_2, \ldots, \}$. Each input is referred to as an instance.
2. The size of an instance I is the number of bits in its representation.
3. For an instance I, $sol(I)$ is a set of feasible solutions to I.
4. For optimization problems each solution $s \in sol(I)$ has an associated value.

Examples

Example

An instance of **Bipartite Matching** is a bipartite graph, and an integer **k**.

Examples

Example

An instance of **Bipartite Matching** is a bipartite graph, and an integer $\mathbf{k}$. The solution to this instance is "YES" if the graph has a matching of size $\geq \mathbf{k}$, and "NO" otherwise.

Examples

Example

An instance of **Bipartite Matching** is a bipartite graph, and an integer $\mathbf{k}$. The solution to this instance is "YES" if the graph has a matching of size $\geq \mathbf{k}$, and "NO" otherwise.

Example

An instance of **Max-Flow** is a graph $\mathbf{G}$ with edge-capacities, two vertices $\mathbf{s}, \mathbf{t}$, and an integer $\mathbf{k}$. The solution to this instance is "YES" if there is a flow from $\mathbf{s}$ to $\mathbf{t}$ of value $\geq \mathbf{k}$, else 'NO".

Examples

Example

An instance of **Bipartite Matching** is a bipartite graph, and an integer $\mathbf{k}$. The solution to this instance is "YES" if the graph has a matching of size $\geq \mathbf{k}$, and "NO" otherwise.

Example

An instance of **Max-Flow** is a graph $\mathbf{G}$ with edge-capacities, two vertices $\mathbf{s}, \mathbf{t}$, and an integer $\mathbf{k}$. The solution to this instance is "YES" if there is a flow from $\mathbf{s}$ to $\mathbf{t}$ of value $\geq \mathbf{k}$, else 'NO".

What is an algorithm for a decision Problem X?

It takes as input an instance of $\mathbf{X}$, and outputs either "YES" or "NO".

Encoding an instance into a string

1. **I**; Instance of some problem.
2. **I** can be fully and precisely described (say in a text file).
3. Resulting text file is a binary string.
4. $\Longrightarrow$ Any input can be interpreted as a binary string **S**.
5. ... Running time of algorithm: Function of length of **S** (i.e., **n**).

Decision Problems and Languages

1. A finite alphabet Σ. Σ^* is set of all finite strings on Σ.
2. A language L is simply a subset of Σ^*; a set of strings.

Decision Problems and Languages

1. A finite alphabet Σ. Σ^* is set of all finite strings on Σ.
2. A language L is simply a subset of Σ^*; a set of strings.

For every language L there is an associated decision problem Π_L and conversely, for every decision problem Π there is an associated language L_Π.

Decision Problems and Languages

1. A finite alphabet Σ. Σ^* is set of all finite strings on Σ.
2. A language L is simply a subset of Σ^*; a set of strings.

For every language L there is an associated decision problem Π_L and conversely, for every decision problem Π there is an associated language L_Π.

1. Given L, Π_L is the following decision problem: Given $x \in \Sigma^*$, is $x \in L$? Each string in Σ^* is an instance of Π_L and L is the set of instances for which the answer is YES.
2. Given Π the associated language is

$$L_\Pi = \left\{ I \;\middle|\; I \text{ is an instance of } \Pi \text{ for which answer is YES} \right\}.$$

Thus, decision problems and languages are used interchangeably.

Example

1. The decision problem **Primality**, and the language

$$\mathbf{L} = \left\{ \#\mathbf{p} \;\middle|\; \mathbf{p} \text{ is a prime number} \right\}.$$

   Here $\#\mathbf{p}$ is the string in base $\mathbf{10}$ representing $\mathbf{p}$.

2. **Bipartite** (is given graph bipartite?). The language is

$$\mathbf{L} = \left\{ \mathcal{S}(\mathrm{G}) \;\middle|\; \mathrm{G} \text{ is a bipartite graph} \right\}.$$

   Here $\mathcal{S}(\mathrm{G})$ is the string encoding the graph G.

DFAs and Regular Languages

What is the language accepted?

DFAs and Regular Languages

What is the language accepted?

$\mathbf{L} = \{\mathbf{s} \in \{\mathbf{a}, \mathbf{b}\}^* \mid \mathbf{s}$ contains at most one pair of consecutive a's$\}$

NFAs and Regular Languages

What is the language accepted?

NFAs and Regular Languages

What is the language accepted?

$\mathbf{L} = \{\mathbf{s} \in \{\mathbf{0}, \mathbf{1}\}^* \mid$ there is a $\mathbf{1}$ in the last 5 positions of $\mathbf{s}\}$

Are regular languages good?

Let **L** be a regular language. Then the decision problem associated with **L** can be solved in

(A) Constant time.

(B) Linear time.

(C) Quadratic time.

(D) Exponential time.

(E) Doubly exponential time (i.e., 2^{2^n}).

(F) Octly exponential time (i.e., $2^{2^{2^{2^{2^{2^{2^{2^n}}}}}}}$).

Reductions, revised.

For decision problems $\mathbf{X}, \mathbf{Y}$, a **reduction from $\mathbf{X}$ to $\mathbf{Y}$** is:

1. An algorithm ...
2. Input: $\mathbf{I_X}$, an instance of $\mathbf{X}$.
3. Output: $\mathbf{I_Y}$ an instance of $\mathbf{Y}$.
4. Such that:

$$\boxed{\mathbf{I_Y} \text{ is YES instance of } \mathbf{Y}} \iff \boxed{\mathbf{I_X} \text{ is YES instance of } \mathbf{X}}$$

Reductions, revised.

For decision problems **X, Y**, a **reduction from X to Y** is:

1. An algorithm ...
2. Input: $\mathbf{I_X}$, an instance of **X**.
3. Output: $\mathbf{I_Y}$ an instance of **Y**.
4. Such that:

$$\boxed{\mathbf{I_Y} \text{ is YES instance of } \mathbf{Y}} \Longleftrightarrow \boxed{\mathbf{I_X} \text{ is YES instance of } \mathbf{X}}$$

There are other kinds of reductions.

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $\mathbf{X} \rightarrow \mathbf{Y}$
2. $\mathcal{A}_{\mathbf{Y}}$: algorithm for $\mathbf{Y}$:

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $\mathbf{X} \rightarrow \mathbf{Y}$
2. $\mathcal{A}_{\mathbf{Y}}$: algorithm for $\mathbf{Y}$:
3. $\Longrightarrow$ New algorithm for $\mathbf{X}$:

$\mathcal{A}_{\mathbf{X}}(\mathbf{I}_{\mathbf{X}})$:

```
// I_X: instance of X.
```

$\mathbf{I}_{\mathbf{Y}} \Leftarrow \mathcal{R}(\mathbf{I}_{\mathbf{X}})$

return $\mathcal{A}_{\mathbf{Y}}(\mathbf{I}_{\mathbf{Y}})$

Using reductions to solve problems

1. $\mathcal{R}$: Reduction $\mathbf{X} \rightarrow \mathbf{Y}$
2. $\mathcal{A}_{\mathbf{Y}}$: algorithm for $\mathbf{Y}$:
3. $\Longrightarrow$ New algorithm for $\mathbf{X}$:

$\mathcal{A}_{\mathbf{X}}(\mathbf{I_X})$:

```
// I_X: instance of X.
```

$\mathbf{I_Y} \Leftarrow \mathcal{R}(\mathbf{I_X})$

return $\mathcal{A}_{\mathbf{Y}}(\mathbf{I_Y})$

If $\mathcal{R}$ and $\mathcal{A}_{\mathbf{Y}}$ polynomial-time $\Longrightarrow$ $\mathcal{A}_{\mathbf{X}}$ polynomial-time.

Comparing Problems

1. "Problem **X** is no harder to solve than Problem **Y**".
2. If Problem **X** reduces to Problem **Y** (we write $\mathbf{X} \leq \mathbf{Y}$), then **X** cannot be harder to solve than **Y**.
3. **Bipartite Matching** $\leq$ **Max-Flow**.
   Bipartite Matching cannot be harder than **Max-Flow**.
4. Equivalently,
   Max-Flow is at least as hard as **Bipartite Matching**.
5. $\mathbf{X} \leq \mathbf{Y}$:
   1. **X** is no harder than **Y**, or
   2. **Y** is at least as hard as **X**.

Part II

Examples of Reductions

Independent Sets and Cliques

Given a graph $\mathbf{G}$, a set of vertices $\mathbf{V'}$ is:

Independent Sets and Cliques

Given a graph $\mathbf{G}$, a set of vertices $\mathbf{V'}$ is:

1. **independent set**: no two vertices of $\mathbf{V'}$ connected by an edge.

Independent Sets and Cliques

Given a graph $\mathbf{G}$, a set of vertices $\mathbf{V'}$ is:

1. **independent set**: no two vertices of $\mathbf{V'}$ connected by an edge.
2. **clique**: *every* pair of vertices in $\mathbf{V'}$ is connected by an edge of $\mathbf{G}$.

Independent Sets and Cliques

Given a graph **G**, a set of vertices **V′** is:

1. **independent set**: no two vertices of **V′** connected by an edge.
2. **clique**: *every* pair of vertices in **V′** is connected by an edge of **G**.

Independent Sets and Cliques

Given a graph **G**, a set of vertices **V′** is:

1. **independent set**: no two vertices of **V′** connected by an edge.
2. **clique**: *every* pair of vertices in **V′** is connected by an edge of **G**.

Independent Sets and Cliques

Given a graph **G**, a set of vertices **V′** is:

1. **independent set**: no two vertices of **V′** connected by an edge.
2. **clique**: *every* pair of vertices in **V′** is connected by an edge of **G**.

The **Independent Set** and **Clique** Problems

Problem: Independent Set

Instance: A graph G and an integer $\mathbf{k}$.
Question: Does G has an independent set of size $\geq \mathbf{k}$?

The Independent Set and Clique Problems

Problem: Independent Set

Instance: A graph G and an integer $\mathbf{k}$.
Question: Does G has an independent set of size $\geq \mathbf{k}$?

Problem: Clique

Instance: A graph G and an integer $\mathbf{k}$.
Question: Does G has a clique of size $\geq \mathbf{k}$?

Recall

For decision problems $\mathbf{X}, \mathbf{Y}$, a reduction from $\mathbf{X}$ to $\mathbf{Y}$ is:

1. An algorithm ...
2. that takes $\mathbf{I_X}$, an instance of $\mathbf{X}$ as input ...
3. and returns $\mathbf{I_Y}$, an instance of $\mathbf{Y}$ as output ...
4. such that the solution (YES/NO) to $\mathbf{I_Y}$ is the same as the solution to $\mathbf{I_X}$.

Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph **G** and an integer **k**.

Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph **G** and an integer **k**.

Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $\mathbf{G}$ and an integer $\mathbf{k}$.

Convert $\mathbf{G}$ to $\overline{\mathbf{G}}$, in which $(\mathbf{u}, \mathbf{v})$ is an edge iff $(\mathbf{u}, \mathbf{v})$ is not an edge of $\mathbf{G}$. ($\overline{\mathbf{G}}$ is the *complement* of $\mathbf{G}$.)
We use $\overline{\mathbf{G}}$ and $\mathbf{k}$ as the instance of **Clique**.

Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $\mathbf{G}$ and an integer $\mathbf{k}$.

Convert $\mathbf{G}$ to $\overline{\mathbf{G}}$, in which $(\mathbf{u}, \mathbf{v})$ is an edge iff $(\mathbf{u}, \mathbf{v})$ is not an edge of $\mathbf{G}$. ($\overline{\mathbf{G}}$ is the *complement* of $\mathbf{G}$.)
We use $\overline{\mathbf{G}}$ and $\mathbf{k}$ as the instance of **Clique**.

Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $\mathbf{G}$ and an integer $\mathbf{k}$.

Convert $\mathbf{G}$ to $\overline{\mathbf{G}}$, in which $(\mathbf{u}, \mathbf{v})$ is an edge iff $(\mathbf{u}, \mathbf{v})$ is not an edge of $\mathbf{G}$. ($\overline{\mathbf{G}}$ is the *complement* of $\mathbf{G}$.)
We use $\overline{\mathbf{G}}$ and $\mathbf{k}$ as the instance of **Clique**.

Reducing **Independent Set** to **Clique**

An instance of **Independent Set** is a graph $\mathbf{G}$ and an integer $\mathbf{k}$.

Convert $\mathbf{G}$ to $\overline{\mathbf{G}}$, in which $(\mathbf{u}, \mathbf{v})$ is an edge iff $(\mathbf{u}, \mathbf{v})$ is not an edge of $\mathbf{G}$. ($\overline{\mathbf{G}}$ is the *complement* of $\mathbf{G}$.)
We use $\overline{\mathbf{G}}$ and $\mathbf{k}$ as the instance of **Clique**.

Independent Set and Clique

1. **Independent Set** $\leq$ **Clique**.

Independent Set and Clique

1. **Independent Set** $\leq$ **Clique**.
   What does this mean?
2. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.

Independent Set and Clique

1. **Independent Set** $\leq$ **Clique**.
   What does this mean?
2. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.

Independent Set and Clique

1. **Independent Set** $\leq$ **Clique**.
   What does this mean?
2. If have an algorithm for **Clique**, then we have an algorithm for **Independent Set**.
3. **Clique** is *at least as hard as* **Independent Set**.
4. Also... **Independent Set** is *at least as hard as* **Clique**.

Independent Set and Clique

Assume you can solve the **Clique** problem in $\mathbf{T(n)}$ time. Then you can solve the **Independent Set** problem in

(A) $\mathbf{O(T(n))}$ time.

(B) $\mathbf{O(n \log n + T(n))}$ time.

(C) $\mathbf{O(n^2 T(n^2))}$ time.

(D) $\mathbf{O(n^4 T(n^4))}$ time.

(E) $\mathbf{O(n^2 + T(n^2))}$ time.

(F) Does not matter - all these are polynomial if $\mathbf{T(n)}$ is polynomial, which is good enough for our purposes.

DFAs and NFAs

DFAs (Remember 373?) are automata that accept regular languages. NFAs are the same, except that they are non-deterministic, while DFAs are deterministic.

DFAs and NFAs

DFAs (Remember 373?) are automata that accept regular languages. NFAs are the same, except that they are non-deterministic, while DFAs are deterministic.

Every NFA can be converted to a DFA that accepts the same language using the subset construction.

(How long does this take?)

DFAs and NFAs

DFAs (Remember 373?) are automata that accept regular languages. NFAs are the same, except that they are non-deterministic, while DFAs are deterministic.

Every NFA can be converted to a DFA that accepts the same language using the subset construction.

(How long does this take?)
The smallest DFA equivalent to an NFA with **n** states may have $\approx 2^n$ states.

DFA Universality

A DFA **M** is universal if it accepts every string.
That is, $\mathbf{L(M) = \Sigma^*}$, the set of all strings.

DFA Universality

A DFA **M** is universal if it accepts every string.
That is, $\mathbf{L(M) = \Sigma^*}$, the set of all strings.

Problem (DFA universality)

Input: *A* DFA **M**.
Goal: *Is* **M** *universal?*

DFA Universality

A DFA **M** is universal if it accepts every string.
That is, $\mathbf{L(M) = \Sigma^*}$, the set of all strings.

Problem (DFA universality)

Input: *A* DFA **M**.
Goal: *Is* **M** *universal?*

How do we solve **DFA Universality**?

DFA Universality

A DFA **M** is universal if it accepts every string.
That is, $\mathbf{L(M) = \Sigma^*}$, the set of all strings.

Problem (DFA universality)

Input: *A* DFA **M**.
Goal: *Is* **M** *universal?*

How do we solve **DFA Universality**?
We check if **M** has *any* reachable non-final state.
Alternatively, minimize **M** to obtain **M′** and see if **M′** has a single state which is an accepting state.

NFA Universality

An NFA **N** is said to be universal if it accepts every string. That is, $\mathbf{L(N) = \Sigma^*}$, the set of all strings.

Problem (NFA universality)

Input: *A* NFA **M**.
Goal: *Is* **M** *universal?*

How do we solve **NFA Universality**?

NFA Universality

An NFA **N** is said to be universal if it accepts every string. That is, $\mathbf{L(N) = \Sigma^*}$, the set of all strings.

Problem (NFA universality)

Input: *A* NFA **M**.
Goal: *Is* **M** *universal?*

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?

NFA Universality

An NFA **N** is said to be universal if it accepts every string. That is, $\mathbf{L(N) = \Sigma^*}$, the set of all strings.

Problem (NFA universality)

Input: *A* NFA **M**.
Goal: *Is* **M** *universal?*

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?
Given an NFA **N**, convert it to an equivalent DFA **M**, and use the **DFA Universality** Algorithm.

NFA Universality

An NFA **N** is said to be universal if it accepts every string. That is, $\mathbf{L(N) = \Sigma^*}$, the set of all strings.

Problem (NFA universality)

Input: *A* NFA **M**.
Goal: *Is* **M** *universal?*

How do we solve **NFA Universality**?
Reduce it to **DFA Universality**?
Given an NFA **N**, convert it to an equivalent DFA **M**, and use the **DFA Universality** Algorithm.

The reduction takes exponential time! Problem is known to be PSPACE-Complete and we do not expect a polynomial-time algorithm.

Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.

Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.

If we have a polynomial-time reduction from problem $\mathbf{X}$ to problem $\mathbf{Y}$ (we write $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$), and a poly-time algorithm $\mathcal{A}_{\mathbf{Y}}$ for $\mathbf{Y}$, we have a polynomial-time/efficient algorithm for $\mathbf{X}$.

Polynomial-time reductions

We say that an algorithm is **efficient** if it runs in polynomial-time.

To find efficient algorithms for problems, we are only interested in polynomial-time reductions. Reductions that take longer are not useful.

If we have a polynomial-time reduction from problem $\mathbf{X}$ to problem $\mathbf{Y}$ (we write $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$), and a poly-time algorithm $\mathcal{A}_{\mathbf{Y}}$ for $\mathbf{Y}$, we have a polynomial-time/efficient algorithm for $\mathbf{X}$.

Polynomial-time Reduction

A polynomial time reduction from a *decision* problem $\mathbf{X}$ to a *decision* problem $\mathbf{Y}$ is an *algorithm* $\mathcal{A}$ that has the following properties:

1. given an instance $\mathbf{I_X}$ of $\mathbf{X}$, $\mathcal{A}$ produces an instance $\mathbf{I_Y}$ of $\mathbf{Y}$
2. $\mathcal{A}$ runs in time polynomial in $|\mathbf{I_X}|$.
3. Answer to $\mathbf{I_X}$ YES *iff* answer to $\mathbf{I_Y}$ is YES.

Proposition

If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ then a polynomial time algorithm for $\mathbf{Y}$ implies a polynomial time algorithm for $\mathbf{X}$.

Such a reduction is called a **Karp reduction**. Most reductions we will need are Karp reductions.

Reductions again...

Let **X** and **Y** be two decision problems, such that **X** can be solved in polynomial time, and $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$. Then

(A) **Y** can be solved in polynomial time.

(B) **Y** can NOT be solved in polynomial time.

(C) If **Y** is hard then **X** is also hard.

(D) None of the above.

(E) All of the above.

Polynomial-time reductions and hardness

For decision problems **X** and **Y**, if $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and **Y** has an efficient algorithm, **X** has an efficient algorithm.

Polynomial-time reductions and hardness

For decision problems **X** and **Y**, if $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and **Y** has an efficient algorithm, **X** has an efficient algorithm.

If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?

Polynomial-time reductions and hardness

For decision problems **X** and **Y**, if **X** $\leq_P$ **Y**, and **Y** has an efficient algorithm, **X** has an efficient algorithm.

If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?

Because we showed **Independent Set** $\leq_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!

Polynomial-time reductions and hardness

For decision problems **X** and **Y**, if **X** $\leq_P$ **Y**, and **Y** has an efficient algorithm, **X** has an efficient algorithm.

If you believe that **Independent Set** does not have an efficient algorithm, why should you believe the same of **Clique**?

Because we showed **Independent Set** $\leq_P$ **Clique**. If **Clique** had an efficient algorithm, so would **Independent Set**!

If **X** $\leq_P$ **Y** and **X** does not have an efficient algorithm, **Y** cannot have an efficient algorithm!

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time reduction from $\mathbf{X}$ to $\mathbf{Y}$. Then for any instance $\mathbf{I_X}$ of $\mathbf{X}$, the size of the instance $\mathbf{I_Y}$ of $\mathbf{Y}$ produced from $\mathbf{I_X}$ by $\mathcal{R}$ is polynomial in the size of $\mathbf{I_X}$.

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time reduction from $\mathbf{X}$ to $\mathbf{Y}$. Then for any instance $\mathbf{I_X}$ of $\mathbf{X}$, the size of the instance $\mathbf{I_Y}$ of $\mathbf{Y}$ produced from $\mathbf{I_X}$ by $\mathcal{R}$ is polynomial in the size of $\mathbf{I_X}$.

Proof.

$\mathcal{R}$ is a polynomial-time algorithm and hence on input $\mathbf{I_X}$ of size $|\mathbf{I_X}|$ it runs in time $\mathbf{p(|I_X|)}$ for some polynomial $\mathbf{p()}$.
$\mathbf{I_Y}$ is the output of $\mathcal{R}$ on input $\mathbf{I_X}$.
$\mathcal{R}$ can write at most $\mathbf{p(|I_X|)}$ bits and hence $|\mathbf{I_Y}| \leq \mathbf{p(|I_X|)}$. □

Polynomial-time reductions and instance sizes

Proposition

Let $\mathcal{R}$ be a polynomial-time reduction from $\mathbf{X}$ to $\mathbf{Y}$. Then for any instance $\mathbf{I_X}$ of $\mathbf{X}$, the size of the instance $\mathbf{I_Y}$ of $\mathbf{Y}$ produced from $\mathbf{I_X}$ by $\mathcal{R}$ is polynomial in the size of $\mathbf{I_X}$.

Proof.

$\mathcal{R}$ is a polynomial-time algorithm and hence on input $\mathbf{I_X}$ of size $|\mathbf{I_X}|$ it runs in time $\mathbf{p(|I_X|)}$ for some polynomial $\mathbf{p()}$.
$\mathbf{I_Y}$ is the output of $\mathcal{R}$ on input $\mathbf{I_X}$.
$\mathcal{R}$ can write at most $\mathbf{p(|I_X|)}$ bits and hence $|\mathbf{I_Y}| \leq \mathbf{p(|I_X|)}$. □

Note: Converse is not true. A reduction need not be polynomial-time even if output of reduction is of size polynomial in its input.

Polynomial-time Reduction

A polynomial time reduction from a *decision* problem $\mathbf{X}$ to a *decision* problem $\mathbf{Y}$ is an *algorithm* $\mathcal{A}$ that has the following properties:

1. Given an instance $\mathbf{I_X}$ of $\mathbf{X}$, $\mathcal{A}$ produces an instance $\mathbf{I_Y}$ of $\mathbf{Y}$.
2. $\mathcal{A}$ runs in time polynomial in $|\mathbf{I_X}|$. This implies that $|\mathbf{I_Y}|$ (size of $\mathbf{I_Y}$) is polynomial in $|\mathbf{I_X}|$.
3. Answer to $\mathbf{I_X}$ YES *iff* answer to $\mathbf{I_Y}$ is YES.

Proposition

If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ then a polynomial time algorithm for $\mathbf{Y}$ implies a polynomial time algorithm for $\mathbf{X}$.

Such a reduction is called a Karp reduction. Most reductions we will need are Karp reductions

Transitivity of Reductions

Proposition

$\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ *and* $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{Z}$ *implies that* $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Z}$.

Note: $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ does not imply that $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$ and hence it is very important to know the FROM and TO in a reduction.

To prove $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ you need to show a reduction FROM **X** TO **Y**
In other words show that an algorithm for **Y** implies an algorithm for **X**.

Vertex Cover

Given a graph $\mathbf{G = (V, E)}$, a set of vertices $\mathbf{S}$ is:

Vertex Cover

Given a graph $\mathbf{G = (V, E)}$, a set of vertices $\mathbf{S}$ is:

1. A **vertex cover** if every $\mathbf{e} \in \mathbf{E}$ has at least one endpoint in $\mathbf{S}$.

Vertex Cover

Given a graph $\mathbf{G = (V, E)}$, a set of vertices $\mathbf{S}$ is:

1. A **vertex cover** if every $\mathbf{e} \in \mathbf{E}$ has at least one endpoint in $\mathbf{S}$.

Vertex Cover

Given a graph $\mathbf{G = (V, E)}$, a set of vertices $\mathbf{S}$ is:

1. A **vertex cover** if every $\mathbf{e} \in \mathbf{E}$ has at least one endpoint in $\mathbf{S}$.

Vertex Cover

Given a graph $\mathbf{G = (V, E)}$, a set of vertices $\mathbf{S}$ is:

1. A **vertex cover** if every $\mathbf{e} \in \mathbf{E}$ has at least one endpoint in $\mathbf{S}$.

The **Vertex Cover** Problem

Problem (**Vertex Cover**)

Input: *A graph G and integer* $\mathbf{k}$.
Goal: *Is there a vertex cover of size* $\leq \mathbf{k}$ *in* G*?*

The **Vertex Cover** Problem

Problem (**Vertex Cover**)

Input: *A graph G and integer* $\mathbf{k}$.
Goal: *Is there a vertex cover of size* $\leq \mathbf{k}$ *in* G*?*

Can we relate **Independent Set** and **Vertex Cover**?

Relationship between...

Vertex Cover and Independent Set

Proposition

Let $\mathbf{G = (V, E)}$ *be a graph.* $\mathbf{S}$ *is an independent set if and only if* $\mathbf{V \setminus S}$ *is a vertex cover.*

Proof.

($\Rightarrow$) Let $\mathbf{S}$ be an independent set

1. Consider any edge $\mathbf{uv} \in \mathbf{E}$.
2. Since $\mathbf{S}$ is an independent set, either $\mathbf{u} \notin \mathbf{S}$ or $\mathbf{v} \notin \mathbf{S}$.
3. Thus, either $\mathbf{u} \in \mathbf{V} \setminus \mathbf{S}$ or $\mathbf{v} \in \mathbf{V} \setminus \mathbf{S}$.
4. $\mathbf{V} \setminus \mathbf{S}$ is a vertex cover.

($\Leftarrow$) Let $\mathbf{V} \setminus \mathbf{S}$ be some vertex cover:

1. Consider $\mathbf{u}, \mathbf{v} \in \mathbf{S}$
2. $\mathbf{uv}$ is not an edge of G, as otherwise $\mathbf{V} \setminus \mathbf{S}$ does not cover $\mathbf{uv}$.
3. $\implies$ $\mathbf{S}$ is thus an independent set. $\square$

Independent Set $\leq_P$ Vertex Cover

1. **G**: graph with **n** vertices, and an integer **k** be an instance of the **Independent Set** problem.

Independent Set $\leq_P$ Vertex Cover

1. **G**: graph with **n** vertices, and an integer **k** be an instance of the **Independent Set** problem.
2. **G** has an independent set of size $\geq$ **k** iff **G** has a vertex cover of size $\leq$ **n** $-$ **k**

Independent Set $\leq_P$ Vertex Cover

1. $\mathbf{G}$: graph with $\mathbf{n}$ vertices, and an integer $\mathbf{k}$ be an instance of the **Independent Set** problem.
2. $\mathbf{G}$ has an independent set of size $\geq \mathbf{k}$ iff $\mathbf{G}$ has a vertex cover of size $\leq \mathbf{n} - \mathbf{k}$
3. $(\mathbf{G}, \mathbf{k})$ is an instance of **Independent Set** , and $(\mathbf{G}, \mathbf{n} - \mathbf{k})$ is an instance of **Vertex Cover** with the same answer.

Independent Set $\leq_P$ Vertex Cover

1. **G**: graph with **n** vertices, and an integer **k** be an instance of the **Independent Set** problem.
2. **G** has an independent set of size $\geq$ **k** iff **G** has a vertex cover of size $\leq \mathbf{n} - \mathbf{k}$
3. $(\mathbf{G}, \mathbf{k})$ is an instance of **Independent Set** , and $(\mathbf{G}, \mathbf{n} - \mathbf{k})$ is an instance of **Vertex Cover** with the same answer.
4. Therefore, **Independent Set** $\leq_P$ **Vertex Cover**. Also **Vertex Cover** $\leq_P$ **Independent Set**.

What about edge cover?

Problem: Edge Cover

Instance: A graph G and integer $\mathbf{k}$.
Question: Is there a subset of $\mathbf{k}$ edges such that all the vertices of G are adjacent to one of these edges.

We have that:

(A) **Edge Cover** is polynomially equivalent to **Independent Set**.

(B) **Edge Cover** is polynomially equivalent to **Vertex Cover**.

(C) **Edge Cover** is polynomially equivalent to **Clique**.

(D) **Edge Cover** is polynomially equivalent to **3 COLORING**.

(E) None of the above.

Can you reduce between these problems

Problem: 2SAT

Instance: **F**: a 2CNF formula.
Question: Is there a satisfying assignment to **F**?

Problem: Max Flow

Instance: **G, s, t, k**: Instance of network flow.
Question: Is there a valid flow in **G** from **s** to **t** of value larger than **k**?

(A) **2SAT** $\leq_P$ **Max Flow**.

(B) **Max Flow** $\leq_P$ **2SAT**.

(C) **2SAT** $\leq_P$ **Max Flow** and **Max Flow** $\leq_P$ **2SAT**.

(D) There is NO polynomial time reduction from **2SAT** to **Max Flow**, or vice versa.

(E) All your reduction belong to us.

A problem of Languages

Suppose you work for the United Nations. Let **U** be the set of all languages spoken by people across the world. The United Nations also has a set of translators, all of whom speak English, and some other languages from **U**.

A problem of Languages

Suppose you work for the United Nations. Let **U** be the set of all languages spoken by people across the world. The United Nations also has a set of translators, all of whom speak English, and some other languages from **U**.

Due to budget cuts, you can only afford to keep **k** translators on your payroll. Can you do this, while still ensuring that there is someone who speaks every language in **U**?

A problem of Languages

Suppose you work for the United Nations. Let **U** be the set of all languages spoken by people across the world. The United Nations also has a set of translators, all of whom speak English, and some other languages from **U**.

Due to budget cuts, you can only afford to keep **k** translators on your payroll. Can you do this, while still ensuring that there is someone who speaks every language in **U**?

More General problem: Find/Hire a small group of people who can accomplish a large number of tasks.

The Set Cover Problem

Problem (Set Cover)

Input: *Given a set* $\mathbf{U}$ *of* $\mathbf{n}$ *elements, a collection* $\mathbf{S_1, S_2, \ldots S_m}$ *of subsets of* $\mathbf{U}$*, and an integer* $\mathbf{k}$*.*

Goal: *Is there a collection of at most* $\mathbf{k}$ *of these sets* $\mathbf{S_i}$ *whose union is equal to* $\mathbf{U}$*?*

The Set Cover Problem

Problem (Set Cover)

Input: *Given a set* U *of* n *elements, a collection* $S_1, S_2, \ldots S_m$ *of subsets of* U*, and an integer* k*.*

Goal: *Is there a collection of at most* k *of these sets* S_i *whose union is equal to* U*?*

Example

Let $U = \{1, 2, 3, 4, 5, 6, 7\}$, $k = 2$ with

$$\begin{array}{ll} S_1 = \{3, 7\} & S_2 = \{3, 4, 5\} \\ S_3 = \{1\} & S_4 = \{2, 4\} \\ S_5 = \{5\} & S_6 = \{1, 2, 6, 7\} \end{array}$$

The Set Cover Problem

Problem (Set Cover)

Input: *Given a set* U *of* n *elements, a collection* $S_1, S_2, \ldots S_m$ *of subsets of* U*, and an integer* k*.*

Goal: *Is there a collection of at most* k *of these sets* S_i *whose union is equal to* U*?*

Example

Let $U = \{1, 2, 3, 4, 5, 6, 7\}$, $k = 2$ with

$$\begin{array}{ll} S_1 = \{3, 7\} & S_2 = \{3, 4, 5\} \\ S_3 = \{1\} & S_4 = \{2, 4\} \\ S_5 = \{5\} & S_6 = \{1, 2, 6, 7\} \end{array}$$

$\{S_2, S_6\}$ is a set cover

Vertex Cover $\leq_P$ Set Cover

Given graph $\mathbf{G = (V, E)}$ and integer $\mathbf{k}$ as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

Vertex Cover $\leq_P$ Set Cover

Given graph $\mathbf{G = (V, E)}$ and integer $\mathbf{k}$ as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

1. Number $\mathbf{k}$ for the **Set Cover** instance is the same as the number $\mathbf{k}$ given for the **Vertex Cover** instance.

Vertex Cover $\leq_P$ Set Cover

Given graph $\mathbf{G = (V, E)}$ and integer $\mathbf{k}$ as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

1. Number $\mathbf{k}$ for the **Set Cover** instance is the same as the number $\mathbf{k}$ given for the **Vertex Cover** instance.
2. $\mathbf{U = E}$.

Vertex Cover $\leq_P$ Set Cover

Given graph $\mathbf{G = (V, E)}$ and integer $\mathbf{k}$ as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

1. Number $\mathbf{k}$ for the **Set Cover** instance is the same as the number $\mathbf{k}$ given for the **Vertex Cover** instance.
2. $\mathbf{U = E}$.
3. We will have one set corresponding to each vertex; $\mathbf{S_v = \{e \mid e}$ is incident on $\mathbf{v\}}$.

Vertex Cover $\leq_P$ Set Cover

Given graph $G = (V, E)$ and integer k as instance of **Vertex Cover**, construct an instance of **Set Cover** as follows:

1. Number k for the **Set Cover** instance is the same as the number k given for the **Vertex Cover** instance.
2. $U = E$.
3. We will have one set corresponding to each vertex; $S_v = \{e \mid e \text{ is incident on } v\}$.

Observe that G has vertex cover of size k if and only if $U, \{S_v\}_{v \in V}$ has a set cover of size k. (Exercise: Prove this.)

Vertex Cover $\leq_P$ Set Cover: Example

Vertex Cover $\leq_P$ Set Cover: Example

Let $\mathbf{U} = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}, \mathbf{g}\}$, $\mathbf{k} = \mathbf{2}$ with

$$\mathbf{S_1} = \{\mathbf{c}, \mathbf{g}\} \qquad \mathbf{S_2} = \{\mathbf{b}, \mathbf{d}\}$$
$$\mathbf{S_3} = \{\mathbf{c}, \mathbf{d}, \mathbf{e}\} \qquad \mathbf{S_4} = \{\mathbf{e}, \mathbf{f}\}$$
$$\mathbf{S_5} = \{\mathbf{a}\} \qquad \mathbf{S_6} = \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{g}\}$$

Vertex Cover $\leq_P$ Set Cover: Example

Let $\mathbf{U} = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}, \mathbf{g}\}$, $\mathbf{k} = \mathbf{2}$ with

$\mathbf{S_1} = \{\mathbf{c}, \mathbf{g}\}$ $\quad$ $\mathbf{S_2} = \{\mathbf{b}, \mathbf{d}\}$

$\mathbf{S_3} = \{\mathbf{c}, \mathbf{d}, \mathbf{e}\}$ $\quad$ $\mathbf{S_4} = \{\mathbf{e}, \mathbf{f}\}$

$\mathbf{S_5} = \{\mathbf{a}\}$ $\quad$ $\mathbf{S_6} = \{\mathbf{a}, \mathbf{b}, \mathbf{f}, \mathbf{g}\}$

$\{\mathbf{S_3}, \mathbf{S_6}\}$ is a set cover

$\{\mathbf{3}, \mathbf{6}\}$ is a vertex cover

Proving Reductions

To prove that $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ you need to give an algorithm $\mathcal{A}$ that:

1. Transforms an instance $\mathbf{I_X}$ of $\mathbf{X}$ into an instance $\mathbf{I_Y}$ of $\mathbf{Y}$.
2. Satisfies the property that answer to $\mathbf{I_X}$ is YES iff $\mathbf{I_Y}$ is YES.
   1. typical easy direction to prove: answer to $\mathbf{I_Y}$ is YES if answer to $\mathbf{I_X}$ is YES
   2. typical difficult direction to prove: answer to $\mathbf{I_X}$ is YES if answer to $\mathbf{I_Y}$ is YES (equivalently answer to $\mathbf{I_X}$ is NO if answer to $\mathbf{I_Y}$ is NO).
3. Runs in **polynomial** time.

Vertex cover and Set cover?

Clicker question

Consider the statement: **Set Cover** $\leq_P$ **Vertex Cover**.
This statement is

- **(A)** correct.
- **(B)** correct (although the reduction seen is in the other direction - so not clear why this is correct).
- **(C)** incorrect.
- **(D)** incorrect (the reduction seen is in the other direction!)

Example of incorrect reduction proof

Try proving **Matching** $\leq_P$ **Bipartite Matching** via following reduction:

1. Given graph $G = (V, E)$ obtain a bipartite graph $G' = (V', E')$ as follows.
   1. Let $V_1 = \{u_1 \mid u \in V\}$ and $V_2 = \{u_2 \mid u \in V\}$. We set $V' = V_1 \cup V_2$ (that is, we make two copies of V)
   2. $E' = \left\{ u_1 v_2 \;\middle|\; u \neq v \text{ and } uv \in E \right\}$
2. Given G and integer k the reduction outputs G' and k.

Example

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$.

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$.

Proof.

Exercise. □

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$.

Proof.

Exercise. □

Claim

If $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G}$ *has a matching of size* $\mathbf{k}$.

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$.

Proof.

Exercise. □

Claim

If $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G}$ *has a matching of size* $\mathbf{k}$.

Incorrect! Why?

"Proof"

Claim

Reduction is a poly-time algorithm. If $\mathbf{G}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$.

Proof.

Exercise. □

Claim

If $\mathbf{G'}$ *has a matching of size* $\mathbf{k}$ *then* $\mathbf{G}$ *has a matching of size* $\mathbf{k}$.

Incorrect! Why? Vertex $\mathbf{u} \in \mathbf{V}$ has two copies $\mathbf{u_1}$ and $\mathbf{u_2}$ in $\mathbf{G'}$. A matching in $\mathbf{G'}$ may use both copies!

Subset sum and Partition?

Problem: Subset Sum

Instance: S - set of positive integers, t: - an integer number (target).
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

Problem: Partition

Instance: A set S of n numbers.
Question: Is there a subset $T \subseteq S$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

Assume that we can solve **Subset Sum** in polynomial time, then we can solve **Partition** in polynomial time. This statement is

(A) True.
(B) Mostly true.
(C) False.
(D) Mostly false.

II: Partition and subset sum?

Problem: Partition

Instance: A set $\mathbf{S}$ of $\mathbf{n}$ numbers.
Question: Is there a subset $\mathbf{T} \subseteq \mathbf{S}$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

Problem: Subset Sum

Instance: $\mathbf{S}$ - set of positive integers, $\mathbf{t}$: - an integer number (target).
Question: Is there a subset $\mathbf{X} \subseteq \mathbf{S}$ such that $\sum_{x \in X} x = t$?

Assume that we can solve **Partition** in polynomial time, then we can solve **Subset Sum** in polynomial time. This statement is

(A) True.

(B) Mostly true.

(C) False.

(D) Mostly false.

III: Partition and Halting?

Problem: Halting

Instance: **P**: Program, **I**: Input.
Question: Does **P** stop on the input **I**?

Problem: Partition

Instance: A set **S** of **n** numbers.
Question: Is there a subset $\mathbf{T} \subseteq \mathbf{S}$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

Assume that we can solve **Halting** in polynomial time, then we can solve **Partition** in polynomial time. This statement is

(A) True.

(B) Mostly true.

(C) False.

(D) Mostly false.

IV: Halting and Partition?

Problem: Partition

Instance: A set $\mathbf{S}$ of $\mathbf{n}$ numbers.
Question: Is there a subset $\mathbf{T} \subseteq \mathbf{S}$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$?

Problem: Halting

Instance: $\mathbf{P}$: Program, $\mathbf{I}$: Input.
Question: Does $\mathbf{P}$ stop on the input $\mathbf{I}$?

Assume that we can solve **Partition** in polynomial time, then we can solve **Halting** in polynomial time. This statement is

(A) True.

(B) Mostly true.

(C) False.

(D) Mostly false.

What we know...

1. **Partition** $\approx_{\mathbf{P}}$ **Subset sum**.
2. **Halting** is way way way way way way **harder**.

Summary

We looked at polynomial-time reductions.

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and there is no efficient algorithm for $\mathbf{X}$, there is no efficient algorithm for $\mathbf{Y}$.

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and we have an efficient algorithm for $\mathbf{Y}$, we have an efficient algorithm for $\mathbf{X}$.
2. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and there is no efficient algorithm for $\mathbf{X}$, there is no efficient algorithm for $\mathbf{Y}$.

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and we have an efficient algorithm for $\mathbf{Y}$, we have an efficient algorithm for $\mathbf{X}$.
2. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and there is no efficient algorithm for $\mathbf{X}$, there is no efficient algorithm for $\mathbf{Y}$.

Summary

We looked at polynomial-time reductions.

Using polynomial-time reductions

1. If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$, and we have an efficient algorithm for $\mathbf{Y}$, we have an efficient algorithm for $\mathbf{X}$.

We looked at some examples of reductions between **Independent Set**, **Clique**, **Vertex Cover**, and **Set Cover**.

Notes

Notes

Notes

Notes

Reductions and NP

Lecture 21

April 17, 2014

Part I

Reductions Continued

Polynomial Time Reduction

Karp reduction

A **polynomial time reduction** from a *decision* problem $\mathbf{X}$ to a *decision* problem $\mathbf{Y}$ is an *algorithm* $\mathcal{A}$ that has the following properties:

1. given an instance $\mathbf{I_X}$ of $\mathbf{X}$, $\mathcal{A}$ produces an instance $\mathbf{I_Y}$ of $\mathbf{Y}$
2. $\mathcal{A}$ runs in time polynomial in $|\mathbf{I_X}|$. This implies that $|\mathbf{I_Y}|$ (size of $\mathbf{I_Y}$) is polynomial in $|\mathbf{I_X}|$
3. Answer to $\mathbf{I_X}$ YES *iff* answer to $\mathbf{I_Y}$ is YES.

Notation: $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ if $\mathbf{X}$ reduces to $\mathbf{Y}$

Proposition

If $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ then a polynomial time algorithm for $\mathbf{Y}$ implies a polynomial time algorithm for $\mathbf{X}$.

Such a reduction is called a **Karp reduction**. Most reductions we will need are Karp reductions.

A More General Reduction

Turing Reduction

Definition (Turing reduction.)

Problem $\mathbf{X}$ polynomial time reduces to $\mathbf{Y}$ if there is an algorithm $\mathcal{A}$ for $\mathbf{X}$ that has the following properties:

1. on any given instance $\mathbf{I_X}$ of $\mathbf{X}$, $\mathcal{A}$ uses polynomial in $|\mathbf{I_X}|$ "steps"
2. a step is either a standard computation step, or
3. a sub-routine call to an algorithm that solves $\mathbf{Y}$.

This is a **Turing reduction**.

A More General Reduction

Turing Reduction

Definition (Turing reduction.)

Problem $\mathbf{X}$ polynomial time reduces to $\mathbf{Y}$ if there is an algorithm $\mathcal{A}$ for $\mathbf{X}$ that has the following properties:

1. on any given instance $\mathbf{I_X}$ of $\mathbf{X}$, $\mathcal{A}$ uses polynomial in $|\mathbf{I_X}|$ "steps"
2. a step is either a standard computation step, or
3. a sub-routine call to an algorithm that solves $\mathbf{Y}$.

This is a **Turing reduction**.

Note: In making sub-routine call to algorithm to solve $\mathbf{Y}$, $\mathcal{A}$ can only ask questions of size polynomial in $|\mathbf{I_X}|$. Why?

Comparing reductions

1. Karp reduction:

2. Turing reduction:

Turing reduction

1. Algorithm to solve **X** can call solver for **Y** many times.
2. Conceptually, every call to the solver of **Y** takes constant time.

Relation between reductions

Consider two problems **X** and **Y**. Which of the following statements is correct?

(A) If there is a Turing reduction from **X** to **Y**, then there is a Karp reduction from **X** to **Y**.

(B) If there is a Karp reduction from **X** to **Y**, then there is a Turing reduction from **X** to **Y**.

(C) If there is a Karp reduction from **X** to **Y**, then there is a Karp reduction from **Y** to **X**.

(D) If there is a Turing reduction from **X** to **Y**, then there is a Turing reduction from **Y** to **X**.

(E) All of the above.

Example of Turing Reduction

Problem (Independent set in circular arcs graph.)

Input: *Collection of arcs on a circle.*
Goal: *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

Example of Turing Reduction

Problem (Independent set in circular arcs graph.)

Input: *Collection of arcs on a circle.*
Goal: *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

Problem (Independent set of intervals.)

Input: *Collection of intervals on the line.*
Goal: *Compute the maximum number of non-overlapping intervals.*

How?

Example of Turing Reduction

Problem (Independent set in circular arcs graph.)

Input: *Collection of arcs on a circle.*
Goal: *Compute the maximum number of non-overlapping arcs.*

Reduced to the following problem:?

Problem (Independent set of intervals.)

Input: *Collection of intervals on the line.*
Goal: *Compute the maximum number of non-overlapping intervals.*

How? Used algorithm for interval problem multiple times.

Turing vs Karp Reductions

1. Turing reductions more general than Karp reductions.
2. Turing reduction useful in obtaining algorithms via reductions.
3. Karp reduction is simpler and easier to use to prove hardness of problems.
4. Perhaps surprisingly, Karp reductions, although limited, suffice for most known **NP-Complete**ness proofs.
5. Karp reductions allow us to distinguish between **NP** and **co-NP** (more on this later).

Propositional Formulas

Definition

Consider a set of boolean variables $x_1, x_2, \ldots x_n$.

1. A **literal** is either a boolean variable x_i or its negation $\neg x_i$.
2. A **clause** is a disjunction of literals.
   For example, $x_1 \vee x_2 \vee \neg x_4$ is a clause.
3. A **formula in conjunctive normal form** (CNF) is propositional formula which is a conjunction of clauses
   1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is a CNF formula.

Propositional Formulas

Definition

Consider a set of boolean variables $x_1, x_2, \ldots x_n$.

1. A **literal** is either a boolean variable x_i or its negation $\neg x_i$.
2. A **clause** is a disjunction of literals.
   For example, $x_1 \vee x_2 \vee \neg x_4$ is a clause.
3. A **formula in conjunctive normal form** (CNF) is propositional formula which is a conjunction of clauses
   1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is a CNF formula.
4. A formula φ is a 3CNF:
   A CNF formula such that every clause has **exactly** 3 literals.
   1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_1)$ is a 3CNF formula, but $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is not.

Satisfiability

Problem: SAT

Instance: A CNF formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

Problem: 3SAT

Instance: A 3CNF formula φ.
Question: Is there a truth assignment to the variable of φ such that φ evaluates to true?

Satisfiability

SAT

Given a CNF formula φ, is there a truth assignment to variables such that φ evaluates to true?

Example

1. $(x_1 \vee x_2 \vee \neg x_4) \wedge (x_2 \vee \neg x_3) \wedge x_5$ is satisfiable; take $x_1, x_2, \ldots x_5$ to be all true
2. $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is not satisfiable.

3SAT

Given a 3CNF formula φ, is there a truth assignment to variables such that φ evaluates to true?

(More on **2SAT** in a bit...)

Importance of SAT and 3SAT

1. **SAT** and **3SAT** are basic constraint satisfaction problems.
2. Many different problems can reduced to them because of the simple yet powerful expressively of logical constraints.
3. Arise naturally in many applications involving hardware and software verification and correctness.
4. As we will see, it is a fundamental problem in theory of **NP-Complete**ness.

$z = \overline{x}$

Given two bits x, z which of the following **SAT** formulas is equivalent to the formula $z = \overline{x}$:

(A) $(\overline{z} \vee x) \wedge (z \vee \overline{x})$.

(B) $(z \vee x) \wedge (\overline{z} \vee \overline{x})$.

(C) $(\overline{z} \vee x) \wedge (\overline{z} \vee \overline{x}) \wedge (\overline{z} \vee \overline{x})$.

(D) $z \oplus x$.

(E) $(z \vee x) \wedge (\overline{z} \vee \overline{x}) \wedge (z \vee \overline{x}) \wedge (\overline{z} \vee x)$.

$z = x \wedge y$

Given three bits x, y, z which of the following **SAT** formulas is equivalent to the formula $z = x \wedge y$:

(A) $(\overline{z} \vee x \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(B) $(\overline{z} \vee x \vee y) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(C) $(\overline{z} \vee x \vee y) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(D) $(z \vee x \vee y) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y})$.

(E) $(z \vee x \vee y) \wedge (z \vee x \vee \overline{y}) \wedge (z \vee \overline{x} \vee y) \wedge (z \vee \overline{x} \vee \overline{y}) \wedge$
$(\overline{z} \vee x \vee y) \wedge (\overline{z} \vee x \vee \overline{y}) \wedge (\overline{z} \vee \overline{x} \vee y) \wedge (\overline{z} \vee \overline{x} \vee \overline{y})$.

$z = x \vee y$

Given three bits x, y, z which of the following **SAT** formulas is equivalent to the formula $z = x \vee y$:

(A) $(\bar{z} \vee x \vee y) \wedge (\bar{z} \vee \bar{x} \vee y) \wedge (z \vee \bar{x} \vee \bar{y})$.

(B) $(\bar{z} \vee x \vee y) \wedge (\bar{z} \vee \bar{x} \vee y) \wedge (z \vee \bar{x} \vee y) \wedge (z \vee \bar{x} \vee \bar{y})$.

(C) $(z \vee x \vee y) \wedge (\bar{z} \vee \bar{x} \vee y) \wedge (z \vee \bar{x} \vee y) \wedge (z \vee \bar{x} \vee \bar{y})$.

(D) $(z \vee x \vee y) \wedge (z \vee x \vee \bar{y}) \wedge (z \vee \bar{x} \vee y) \wedge (z \vee \bar{x} \vee \bar{y}) \wedge$
$(\bar{z} \vee x \vee y) \wedge (\bar{z} \vee x \vee \bar{y}) \wedge (\bar{z} \vee \bar{x} \vee y) \wedge (\bar{z} \vee \bar{x} \vee \bar{y})$.

(E) $(\bar{z} \vee x \vee y) \wedge (z \vee \bar{x} \vee y) \wedge (z \vee x \vee \bar{y}) \wedge (z \vee \bar{x} \vee \bar{y})$.

SAT $\leq_P$ 3SAT

How SAT is different from 3SAT?

In **SAT** clauses might have arbitrary length: $\mathbf{1, 2, 3, \ldots}$ variables:

$$\left(\mathbf{x} \vee \mathbf{y} \vee \mathbf{z} \vee \mathbf{w} \vee \mathbf{u}\right) \wedge \left(\neg\mathbf{x} \vee \neg\mathbf{y} \vee \neg\mathbf{z} \vee \mathbf{w} \vee \mathbf{u}\right) \wedge \left(\neg\mathbf{x}\right)$$

In **3SAT** every clause must have **exactly 3** different literals.

SAT $\leq_P$ 3SAT

How SAT is different from 3SAT?

In **SAT** clauses might have arbitrary length: $1, 2, 3, \ldots$ variables:

$$\left(\mathbf{x} \vee \mathbf{y} \vee \mathbf{z} \vee \mathbf{w} \vee \mathbf{u}\right) \wedge \left(\neg\mathbf{x} \vee \neg\mathbf{y} \vee \neg\mathbf{z} \vee \mathbf{w} \vee \mathbf{u}\right) \wedge \left(\neg\mathbf{x}\right)$$

In **3SAT** every clause must have **exactly 3** different literals.

To reduce from an instance of **SAT** to an instance of **3SAT**, we must make all clauses to have exactly **3** variables...

Basic idea

1. Pad short clauses so they have **3** literals.
2. Break long clauses into shorter clauses.
3. Repeat the above till we have a 3CNF.

3SAT $\leq_P$ SAT

1. **3SAT** $\leq_P$ **SAT**.
2. Because...
   A **3SAT** instance is also an instance of **SAT**.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ 3SAT.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ 3SAT.

Given φ a **SAT** formula we create a **3SAT** formula φ' such that

1. φ is satisfiable iff φ' is satisfiable.
2. φ' can be constructed from φ in time polynomial in $|\varphi|$.

SAT $\leq_P$ 3SAT

Claim

SAT $\leq_P$ 3SAT.

Given φ a **SAT** formula we create a **3SAT** formula φ' such that

1. φ is satisfiable iff φ' is satisfiable.
2. φ' can be constructed from φ in time polynomial in $|\varphi|$.

Idea: if a clause of φ is not of length **3**, replace it with several clauses of length exactly **3**.

SAT $\leq_P$ 3SAT

A clause with a single literal

Reduction Ideas

Challenge: Some of the clauses in φ may have less or more than $\mathbf{3}$ literals. For each clause with $< \mathbf{3}$ or $> \mathbf{3}$ literals, we will construct a set of logically equivalent clauses.

1. Case clause with one literal: Let $\mathbf{c}$ be a clause with a single literal (i.e., $\mathbf{c} = \ell$). Let $\mathbf{u}, \mathbf{v}$ be new variables. Consider

$$\begin{aligned} \mathbf{c}' = \quad & \left(\ell \vee \mathbf{u} \vee \mathbf{v}\right) \wedge \left(\ell \vee \mathbf{u} \vee \neg\mathbf{v}\right) \\ & \wedge \left(\ell \vee \neg\mathbf{u} \vee \mathbf{v}\right) \wedge \left(\ell \vee \neg\mathbf{u} \vee \neg\mathbf{v}\right). \end{aligned}$$

Observe that $\mathbf{c}'$ is satisfiable iff $\mathbf{c}$ is satisfiable

SAT $\leq_P$ 3SAT

A clause with two literals

Reduction Ideas: 2 and more literals

1. Case clause with 2 literals: Let $\mathbf{c} = \ell_1 \vee \ell_2$. Let $\mathbf{u}$ be a new variable. Consider

$$\mathbf{c}' = \left(\ell_1 \vee \ell_2 \vee \mathbf{u}\right) \wedge \left(\ell_1 \vee \ell_2 \vee \neg\mathbf{u}\right).$$

Again $\mathbf{c}$ is satisfiable iff $\mathbf{c}'$ is satisfiable

Breaking a clause

Lemma

For any boolean formulas $\mathbf{X}$ *and* $\mathbf{Y}$ *and* $\mathbf{z}$ *a new boolean variable. Then*

$$\mathbf{X} \vee \mathbf{Y} \text{ is satisfiable}$$

if and only if, $\mathbf{z}$ *can be assigned a value such that*

$$\left(\mathbf{X} \vee \mathbf{z}\right) \wedge \left(\mathbf{Y} \vee \neg\mathbf{z}\right) \text{ is satisfiable}$$

(with the same assignment to the variables appearing in $\mathbf{X}$ *and* $\mathbf{Y}$*).*

SAT $\leq_P$ 3SAT (contd)

Clauses with more than 3 literals

Let $\mathbf{c} = \ell_1 \vee \cdots \vee \ell_k$. Let $\mathbf{u_1, \ldots u_{k-3}}$ be new variables. Consider

$$\begin{aligned} \mathbf{c'} = & \left(\ell_1 \vee \ell_2 \vee \mathbf{u_1}\right) \wedge \left(\ell_3 \vee \neg \mathbf{u_1} \vee \mathbf{u_2}\right) \\ & \wedge \left(\ell_4 \vee \neg \mathbf{u_2} \vee \mathbf{u_3}\right) \wedge \\ & \cdots \wedge \left(\ell_{k-2} \vee \neg \mathbf{u_{k-4}} \vee \mathbf{u_{k-3}}\right) \wedge \left(\ell_{k-1} \vee \ell_k \vee \neg \mathbf{u_{k-3}}\right). \end{aligned}$$

Claim

$\mathbf{c}$ *is satisfiable iff* $\mathbf{c'}$ *is satisfiable.*

Another way to see it — reduce size of clause by one:

$$\mathbf{c'} = \left(\ell_1 \vee \ell_2 \ldots \vee \ell_{k-2} \vee \mathbf{u_{k-3}}\right) \wedge \left(\ell_{k-1} \vee \ell_k \vee \neg \mathbf{u_{k-3}}\right).$$

An Example

Example

$$\varphi = \left(\neg x_1 \vee \neg x_4\right) \wedge \left(x_1 \vee \neg x_2 \vee \neg x_3\right)$$
$$\wedge \left(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1\right) \wedge \left(x_1\right).$$

Equivalent form:

$$\psi = (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z)$$

An Example

Example

$$\varphi = \left(\neg x_1 \vee \neg x_4\right) \wedge \left(x_1 \vee \neg x_2 \vee \neg x_3\right)$$
$$\wedge \left(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1\right) \wedge \left(x_1\right).$$

Equivalent form:

$$\psi = (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z)$$
$$\wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$

An Example

Example

$$\varphi = \left(\neg x_1 \vee \neg x_4\right) \wedge \left(x_1 \vee \neg x_2 \vee \neg x_3\right)$$
$$\wedge \left(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1\right) \wedge \left(x_1\right).$$

Equivalent form:

$$\psi = (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z)$$
$$\wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$
$$\wedge (\neg x_2 \vee \neg x_3 \vee y_1) \wedge (x_4 \vee x_1 \vee \neg y_1)$$

An Example

Example

$$\varphi = \left(\neg x_1 \vee \neg x_4\right) \wedge \left(x_1 \vee \neg x_2 \vee \neg x_3\right)$$
$$\wedge \left(\neg x_2 \vee \neg x_3 \vee x_4 \vee x_1\right) \wedge \left(x_1\right).$$

Equivalent form:

$$\psi = (\neg x_1 \vee \neg x_4 \vee z) \wedge (\neg x_1 \vee \neg x_4 \vee \neg z)$$
$$\wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$
$$\wedge (\neg x_2 \vee \neg x_3 \vee y_1) \wedge (x_4 \vee x_1 \vee \neg y_1)$$
$$\wedge (x_1 \vee u \vee v) \wedge (x_1 \vee u \vee \neg v)$$
$$\wedge (x_1 \vee \neg u \vee v) \wedge (x_1 \vee \neg u \vee \neg v).$$

Overall Reduction Algorithm

Reduction from **SAT** to **3SAT**

```
ReduceSATTo3SAT(φ):
    // φ:  CNF formula.
    for each clause c of φ do
        if c does not have exactly 3 literals then
            construct c′ as before
        else
            c′ = c
    ψ is conjunction of all c′ constructed in loop
    return Solver3SAT(ψ)
```

Correctness (informal)

φ is satisfiable iff ψ is satisfiable because for each clause $\mathbf{c}$, the new 3CNF formula $\mathbf{c}'$ is logically equivalent to $\mathbf{c}$.

What about 2SAT?

2SAT can be solved in polynomial time! (specifically, linear time!)

No known polynomial time reduction from **SAT** (or **3SAT**) to **2SAT**. If there was, then **SAT** and **3SAT** would be solvable in polynomial time.

Why the reduction from 3SAT to 2SAT fails?

Consider a clause $(\mathbf{x} \vee \mathbf{y} \vee \mathbf{z})$. We need to reduce it to a collection of $\mathbf{2}\mathrm{CNF}$ clauses. Introduce a face variable α, and rewrite this as

$$(\mathbf{x} \vee \mathbf{y} \vee \alpha) \wedge (\neg\alpha \vee \mathbf{z}) \qquad \text{(bad! clause with 3 vars)}$$

$$\text{or} \quad (\mathbf{x} \vee \alpha) \wedge (\neg\alpha \vee \mathbf{y} \vee \mathbf{z}) \qquad \text{(bad! clause with 3 vars).}$$

(In animal farm language: **2SAT** good, **3SAT** bad.)

What about 2SAT?

A challenging exercise: Given a **2SAT** formula show to compute its satisfying assignment...
(Hint: Create a graph with two vertices for each variable (for a variable $\mathbf{x}$ there would be two vertices with labels $\mathbf{x = 0}$ and $\mathbf{x = 1}$). For ever $\mathbf{2}\mathrm{CNF}$ clause add two directed edges in the graph. The edges are implication edges: They state that if you decide to assign a certain value to a variable, then you must assign a certain value to some other variable.
Now compute the strong connected components in this graph, and continue from there...)

Independent Set

Problem: Independent Set

Instance: A graph G, integer $\mathbf{k}$.
Question: Is there an independent set in G of size $\mathbf{k}$?

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a 3CNF formula φ

Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a 3CNF formula φ

Goal: Construct a graph G_φ and number k such that G_φ has an independent set of size k if and only if φ is satisfiable.

G_φ should be constructable in time polynomial in size of φ

3SAT $\leq_P$ Independent Set

The reduction 3SAT $\leq_P$ Independent Set

Input: Given a $3\mathrm{CNF}$ formula φ
Goal: Construct a graph $\mathbf{G}_\varphi$ and number $\mathbf{k}$ such that $\mathbf{G}_\varphi$ has an independent set of size $\mathbf{k}$ if and only if φ is satisfiable.
$\mathbf{G}_\varphi$ should be constructable in time polynomial in size of φ

Importance of reduction: Although **3SAT** is much more expressive, it can be reduced to a seemingly specialized Independent Set problem.

Notice: We handle only $3\mathrm{CNF}$ formulas – reduction would not work for other kinds of boolean formulas.

Interpreting **3SAT**

There are two ways to think about **3SAT**

Interpreting **3SAT**

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.

Interpreting **3SAT**

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true

Interpreting **3SAT**

There are two ways to think about **3SAT**

1. Find a way to assign 0/1 (false/true) to the variables such that the formula evaluates to true, that is each clause evaluates to true.
2. Pick a literal from each clause and find a truth assignment to make all of them true. You will fail if two of the literals you pick are in conflict, i.e., you pick x_i and $\neg x_i$

We will take the second view of **3SAT** to construct the reduction.

The Reduction

1. $\mathbf{G}_\varphi$ will have one vertex for each literal in a clause

Figure : Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. $\mathbf{G}_\varphi$ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

Figure : Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. G_φ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true

Figure : Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. $\mathbf{G}_\varphi$ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict

Figure : Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

The Reduction

1. $\mathbf{G}_\varphi$ will have one vertex for each literal in a clause
2. Connect the 3 literals in a clause to form a triangle; the independent set will pick at most one vertex from each clause, which will correspond to the literal to be set to true
3. Connect 2 vertices if they label complementary literals; this ensures that the literals corresponding to the independent set do not have a conflict
4. Take $\mathbf{k}$ to be the number of clauses

Figure : Graph for
$\varphi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$

Correctness

Proposition

φ is satisfiable iff $\mathbf{G}_\varphi$ has an independent set of size $\mathbf{k}$ (= number of clauses in φ).

Proof.

$\Rightarrow$ Let $\mathbf{a}$ be the truth assignment satisfying φ

Correctness

Proposition

φ is satisfiable iff $\mathbf{G}_\varphi$ has an independent set of size $\mathbf{k}$ (= number of clauses in φ).

Proof.

$\Rightarrow$ Let $\mathbf{a}$ be the truth assignment satisfying φ

1. Pick one of the vertices, corresponding to true literals under $\mathbf{a}$, from each triangle. This is an independent set of the appropriate size $\square$

Correctness (contd)

Proposition

φ is satisfiable iff $\mathbf{G}_\varphi$ has an independent set of size $\mathbf{k}$ (= number of clauses in φ).

Proof.

$\Leftarrow$ Let $\mathbf{S}$ be an independent set of size $\mathbf{k}$

1. $\mathbf{S}$ must contain exactly one vertex from each clause
2. $\mathbf{S}$ cannot contain vertices labeled by conflicting clauses
3. Thus, it is possible to obtain a truth assignment that makes in the literals in $\mathbf{S}$ true; such an assignment satisfies one literal in every clause $\square$

Transitivity of Reductions

Lemma

$\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ *and* $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{Z}$ *implies that* $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Z}$.

Note: $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ does not imply that $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$ and hence it is very important to know the FROM and TO in a reduction.

To prove $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ you need to show a reduction FROM **X** TO **Y**
In other words show that an algorithm for **Y** implies an algorithm for **X**.

Part II

Definition of NP

Clique?

Given a graph G with $\mathbf{n}$ vertices and $\mathbf{m}$ edges, consider a certificate (i.e., proof) that G indeed has a clique of size $\mathbf{k}$. We have that:

(A) There is such a certificate of length $\mathbf{t = O(n)}$, and it can be verified in $\mathbf{O(2^t)}$ time.

(B) There is such a certificate of length $\mathbf{t = O(n2^n)}$, and it can be verified in $\mathbf{O(t)}$ time.

(C) There is such a certificate of length $\mathbf{t = O(k)}$, and it can be verified in $\mathbf{O(n^2)}$ time.

(D) There is no certificate for this problem.

(E) If there was such a certificate, then we could solve the problem in polynomial time.

Not Clique?

Given a graph G with $\mathbf{n}$ vertices and $\mathbf{m}$ edges, consider a certificate (i.e., proof) that G has **NO** clique of size $\mathbf{k}$. We have that:

(A) There is such a certificate of length $\mathbf{t = O(n)}$, and it can be verified in $\mathbf{O(2^t)}$ time.

(B) There is such a certificate of length $\mathbf{t = O(n2^n)}$, and it can be verified in $\mathbf{O(t)}$ time.

(C) There is such a certificate of length $\mathbf{t = O(k)}$, and it can be verified in $\mathbf{O(n^2)}$ time.

(D) There is no certificate for this problem.

(E) If there was such a certificate, then we could solve the problem in polynomial time.

Recap . . .

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Recap . . .

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Relationship

3SAT $\leq_P$ **Independent Set**

Recap . . .

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Relationship

$$\textbf{3SAT} \leq_P \textbf{Independent Set} \overset{\leq_P}{\geq_P} \textbf{Vertex Cover}$$

Recap . . .

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Relationship

$$\textbf{3SAT} \leq_P \textbf{Independent Set} \overset{\leq_P}{\geq_P} \textbf{Vertex Cover} \leq_P \textbf{Set Cover}$$

Recap . . .

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

Relationship

$$\text{3SAT} \leq_P \text{Independent Set} \overset{\leq_P}{\geq_P} \text{Vertex Cover} \leq_P \text{Set Cover}$$
$$\text{3SAT} \leq_P \text{SAT} \leq_P \text{3SAT}$$

Problems and Algorithms: Formal Approach

Decision Problems

1. Problem Instance: Binary string $\mathbf{s}$, with size $|\mathbf{s}|$
2. Problem: A set $\mathbf{X}$ of strings on which the answer should be "yes"; we call these YES instances of $\mathbf{X}$. Strings not in $\mathbf{X}$ are NO instances of $\mathbf{X}$.

Definition

1. $\mathbf{A}$ is an algorithm for problem $\mathbf{X}$ if $\mathbf{A(s)} =$ "yes" iff $\mathbf{s} \in \mathbf{X}$.
2. $\mathbf{A}$ is said to have a polynomial running time if there is a polynomial $\mathbf{p(\cdot)}$ such that for every string $\mathbf{s}$, $\mathbf{A(s)}$ terminates in at most $\mathbf{O(p(|s|))}$ steps.

Polynomial Time

Definition

Polynomial time (denoted by **P**) is the class of all (decision) problems that have an algorithm that solves it in polynomial time.

Polynomial Time

Definition

Polynomial time (denoted by **P**) is the class of all (decision) problems that have an algorithm that solves it in polynomial time.

Example

Problems in **P** include

1. Is there a shortest path from **s** to **t** of length $\leq$ **k** in **G**?
2. Is there a flow of value $\geq$ **k** in network **G**?
3. Is there an assignment to variables to satisfy given linear constraints?

Efficiency Hypothesis

A problem **X** *has an efficient algorithm iff* $\mathbf{X} \in \mathbf{P}$*, that is* **X** *has a polynomial time algorithm.*

Justifications:

1. Robustness of definition to variations in machines.
2. A sound theoretical definition.
3. Most known polynomial time algorithms for "natural" problems have small polynomial running times.

Problems with no known polynomial time algorithms

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

There are of course undecidable problems (no algorithm at all!) but many problems that we want to solve are of similar flavor to the above.

Question: What is common to above problems?

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance $\mathbf{I_X}$ of $\mathbf{X}$ there is a proof/certificate/solution that is of length poly($|\mathbf{I_X}|$) such that given a proof one can efficiently check that $\mathbf{I_X}$ is indeed a YES instance.

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance $\mathbf{I_X}$ of $\mathbf{X}$ there is a proof/certificate/solution that is of length poly($|\mathbf{I_X}|$) such that given a proof one can efficiently check that $\mathbf{I_X}$ is indeed a YES instance.

Examples:

1. **SAT** formula φ: proof is a satisfying assignment.
2. **Independent Set** in graph $\mathbf{G}$ and $\mathbf{k}$: a subset $\mathbf{S}$ of vertices.

Certifiers

Definition

An algorithm $\mathbf{C(\cdot,\cdot)}$ is a **certifier** for problem $\mathbf{X}$ if for every $\mathbf{s} \in \mathbf{X}$ there is some string $\mathbf{t}$ such that $\mathbf{C(s,t)} =$ "yes", and conversely, if for some $\mathbf{s}$ and $\mathbf{t}$, $\mathbf{C(s,t)} =$ "yes" then $\mathbf{s} \in \mathbf{X}$.
The string $\mathbf{t}$ is called a certificate or proof for $\mathbf{s}$.

Certifiers

Definition

An algorithm $\mathbf{C(\cdot,\cdot)}$ is a **certifier** for problem $\mathbf{X}$ if for every $\mathbf{s} \in \mathbf{X}$ there is some string $\mathbf{t}$ such that $\mathbf{C(s,t)} =$ "yes", and conversely, if for some $\mathbf{s}$ and $\mathbf{t}$, $\mathbf{C(s,t)} =$ "yes" then $\mathbf{s} \in \mathbf{X}$.
The string $\mathbf{t}$ is called a certificate or proof for $\mathbf{s}$.

Definition (Efficient Certifier.)

A certifier $\mathbf{C}$ is an **efficient certifier** for problem $\mathbf{X}$ if there is a polynomial $\mathbf{p(\cdot)}$ such that for every string $\mathbf{s}$, we have that

- ⋆ $\mathbf{s} \in \mathbf{X}$ if and only if
- ⋆ there is a string $\mathbf{t}$:
  1. $|\mathbf{t}| \leq \mathbf{p}(|\mathbf{s}|)$,
  2. $\mathbf{C(s,t)} =$ "yes",
  3. and $\mathbf{C}$ runs in polynomial time.

Example: Independent Set

1. Problem: Does $\mathbf{G = (V, E)}$ have an independent set of size $\geq \mathbf{k}$?
   1. Certificate: Set $\mathbf{S} \subseteq \mathbf{V}$.
   2. Certifier: Check $|\mathbf{S}| \geq \mathbf{k}$ and no pair of vertices in $\mathbf{S}$ is connected by an edge.

Example: Vertex Cover

1. Problem: Does $\mathbf{G}$ have a vertex cover of size $\leq \mathbf{k}$?
   1. Certificate: $\mathbf{S} \subseteq \mathbf{V}$.
   2. Certifier: Check $|\mathbf{S}| \leq \mathbf{k}$ and that for every edge at least one endpoint is in $\mathbf{S}$.

Example: SAT

1. Problem: Does formula φ have a satisfying truth assignment?
   1. Certificate: Assignment $\mathbf{a}$ of $\mathbf{0/1}$ values to each variable.
   2. Certifier: Check each clause under $\mathbf{a}$ and say "yes" if all clauses are true.

Example: Composites

Problem: Composite

Instance: A number $\mathbf{s}$.
Question: Is the number $\mathbf{s}$ a composite?

1. Problem: **Composite**.
   1. Certificate: A factor $\mathbf{t} \leq \mathbf{s}$ such that $\mathbf{t} \neq \mathbf{1}$ and $\mathbf{t} \neq \mathbf{s}$.
   2. Certifier: Check that $\mathbf{t}$ divides $\mathbf{s}$.

Not composite?

Problem: Not Composite

Instance: A number **s**.
Question: Is the number **s** not a composite?

The problem **Not Composite** is

(A) Can be solved in linear time.

(B) in **P**.

(C) Can be solved in exponential time.

(D) Does not have a certificate or an efficient certifier.

(E) The status of this problem is still open.

Example: NFA Universality

Problem: NFA Universality

Instance: Description of a NFA $\mathbf{M}$.
Question: Is $\mathbf{L(M) = \Sigma^*}$, that is, does $\mathbf{M}$ accept all strings?

1. Problem: **NFA Universality**.
   1. Certificate: A DFA $\mathbf{M'}$ equivalent to $\mathbf{M}$
   2. Certifier: Check that $\mathbf{L(M') = \Sigma^*}$

Example: NFA Universality

Problem: NFA Universality

Instance: Description of a NFA $\mathbf{M}$.
Question: Is $\mathbf{L(M) = \Sigma^*}$, that is, does $\mathbf{M}$ accept all strings?

1. Problem: **NFA Universality**.
   1. Certificate: A DFA $\mathbf{M'}$ equivalent to $\mathbf{M}$
   2. Certifier: Check that $\mathbf{L(M') = \Sigma^*}$

Certifier is efficient but certificate is not necessarily short! We do not know if the problem is in **NP**.

Example: A String Problem

Problem: PCP

Instance: Two sets of binary strings $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$
Question: Are there indices $i_1, i_2, \ldots, i_k$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

1. Problem: **PCP**
   1. Certificate: A sequence of indices $i_1, i_2, \ldots, i_k$
   2. Certifier: Check that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

Example: A String Problem

Problem: PCP

Instance: Two sets of binary strings $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$
Question: Are there indices $i_1, i_2, \ldots, i_k$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

1. Problem: **PCP**
   1. Certificate: A sequence of indices $i_1, i_2, \ldots, i_k$
   2. Certifier: Check that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

PCP = Posts Correspondence Problem and it is undecidable!
Implies no finite bound on length of certificate!

Nondeterministic Polynomial Time

Definition

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Nondeterministic Polynomial Time

Definition

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Example

Independent Set, **Vertex Cover**, **Set Cover**, **SAT**, **3SAT**, and **Composite** are all examples of problems in **NP**.

Why is it called...

Nondeterministic Polynomial Time

A certifier is an algorithm $\mathbf{C(I, c)}$ with two inputs:

1. **I**: instance.
2. **c**: proof/certificate that the instance is indeed a YES instance of the given problem.

One can think about **C** as an algorithm for the original problem, if:

1. Given **I**, the algorithm guesses (non-deterministically, and who knows how) a certificate **c**.
2. The algorithm now verifies the certificate **c** for the instance **I**.

NP can be equivalently described using Turing machines.

Asymmetry in Definition of NP

Note that only YES instances have a short proof/certificate. NO instances need not have a short certificate.

Example

SAT formula φ. No easy way to prove that φ is NOT satisfiable!

More on this and **co-NP** later on.

P versus NP

Proposition

$\mathbf{P} \subseteq \mathbf{NP}$.

P versus NP

Proposition

$\mathbf{P} \subseteq \mathbf{NP}$.

For a problem in **P** no need for a certificate!

Proof.

Consider problem $\mathbf{X} \in \mathbf{P}$ with algorithm $\mathbf{A}$. Need to demonstrate that $\mathbf{X}$ has an efficient certifier:

1. Certifier $\mathbf{C}$ on input $\mathbf{s}, \mathbf{t}$, runs $\mathbf{A}(\mathbf{s})$ and returns the answer.
2. $\mathbf{C}$ runs in polynomial time.
3. If $\mathbf{s} \in \mathbf{X}$, then for every $\mathbf{t}$, $\mathbf{C}(\mathbf{s}, \mathbf{t}) =$ "yes".
4. If $\mathbf{s} \notin \mathbf{X}$, then for every $\mathbf{t}$, $\mathbf{C}(\mathbf{s}, \mathbf{t}) =$ "no". $\square$

Exponential Time

Definition

Exponential Time (denoted **EXP**) is the collection of all problems that have an algorithm which on input **s** runs in exponential time, i.e., $\mathbf{O(2^{poly(|s|)})}$.

Exponential Time

Definition

Exponential Time (denoted **EXP**) is the collection of all problems that have an algorithm which on input **s** runs in exponential time, i.e., $O(2^{\mathrm{poly}(|s|)})$.

Example: $O(2^n)$, $O(2^{n \log n})$, $O(2^{n^3})$, ...

NP versus EXP

Proposition

$\mathbf{NP} \subseteq \mathbf{EXP}$.

Proof.

Let $\mathbf{X} \in \mathbf{NP}$ with certifier $\mathbf{C}$. Need to design an exponential time algorithm for $\mathbf{X}$.

1. For every $\mathbf{t}$, with $|\mathbf{t}| \leq \mathbf{p}(|\mathbf{s}|)$ run $\mathbf{C}(\mathbf{s}, \mathbf{t})$; answer "yes" if any one of these calls returns "yes".
2. The above algorithm correctly solves $\mathbf{X}$ (exercise).
3. Algorithm runs in $\mathbf{O}(\mathbf{q}(|\mathbf{s}| + |\mathbf{p}(\mathbf{s})|)2^{\mathbf{p}(|\mathbf{s}|)})$, where $\mathbf{q}$ is the running time of $\mathbf{C}$. □

Examples

1. **SAT**: try all possible truth assignment to variables.
2. **Independent Set**: try all possible subsets of vertices.
3. **Vertex Cover**: try all possible subsets of vertices.

Is **NP** efficiently solvable?

We know $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$.

Is NP efficiently solvable?

We know $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$.

Big Question

Is there are problem in **NP** that does not belong to **P**? Is $\mathbf{P} = \mathbf{NP}$?

If $\mathbf{P} = \mathbf{NP} \ldots$

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.

If $\mathbf{P} = \mathbf{NP} \ldots$

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.

If $\mathbf{P} = \mathbf{NP} \ldots$

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.

If $\mathbf{P} = \mathbf{NP} \ldots$

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce $\ldots$

If $\mathbf{P} = \mathbf{NP} \ldots$

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce $\ldots$
5. Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

If $\mathbf{P} = \mathbf{NP}$ this implies that...

(A) **Vertex Cover** can be solved in polynomial time.

(B) $\mathbf{P} = \mathbf{EXP}$.

(C) $\mathbf{EXP} \subseteq \mathbf{P}$.

(D) All of the above.

P versus NP

Status

Relationship between **P** and **NP** remains one of the most important open problems in mathematics/computer science.

Consensus: Most people feel/believe **P** $\neq$ **NP**.

Resolving **P** versus **NP** is a Clay Millennium Prize Problem. You can win a million dollars in addition to a Turing award and major fame!

Part III

Not for lecture: Converting any boolean formula into CNF

The dark art of formula conversion into CNF

Consider an arbitrary boolean formula ϕ defined over $\mathbf{k}$ variables. To keep the discussion concrete, consider the formula $\phi \equiv \mathbf{x_k = x_i \wedge x_j}$. We would like to convert this formula into an equivalent CNF formula.

Formula conversion into CNF

Step 1

Build a truth table for the boolean formula.

x_k	x_i	x_j	value of $x_k = x_i \wedge x_j$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Formula conversion into CNF

Step 1.5 - understand what a single CNF clause represents

Given an assignment, say, $x_k = 0$, $x_i = 0$ and $x_j = 1$, consider the CNF clause $x_k \vee x_i \vee \overline{x_j}$ (you negate a variable if it is assigned one). Its truth table is

x_k	x_i	x_j	$x_k \vee x_i \vee \overline{x_j}$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Observe that a single clause assigns zero to one row, and one everywhere else. An conjunction of several such clauses, as such, would result in a formula that is 0 in all the rows that corresponds to these clauses, and one everywhere else.

Formula conversion into CNF

Step 2

Write down the CNF clause for every row in the table that is zero.

x_k	x_i	x_j	$x_k = x_i \wedge x_j$	CNF clause
0	0	0	1	
0	0	1	1	
0	1	0	1	
0	1	1	0	$x_k \vee \overline{x_i} \vee \overline{x_j}$
1	0	0	0	$\overline{x_k} \vee x_i \vee x_j$
1	0	1	0	$\overline{x_k} \vee x_i \vee \overline{x_j}$
1	1	0	0	$\overline{x_k} \vee \overline{x_i} \vee x_j$
1	1	1	1	

The conjunction (i.e., and) of all these clauses is clearly equivalent to the original formula. In this case

$$\psi \equiv (x_k \vee \overline{x_i} \vee \overline{x_j}) \wedge (\overline{x_k} \vee x_i \vee x_j) \wedge (\overline{x_k} \vee x_i \vee \overline{x_j}) \wedge (\overline{x_k} \vee \overline{x_i} \vee x_j)$$

Formula conversion into CNF

Step 3 - simplify if you want to

Using that $(x \vee y) \wedge (x \vee \overline{y}) = x$, we have that:

1. $(\overline{x_k} \vee x_i \vee x_j) \wedge (\overline{x_k} \vee x_i \vee \overline{x_j})$ is equivalent to $(\overline{x_k} \vee x_i)$.
2. $(\overline{x_k} \vee x_i \vee x_j) \wedge (\overline{x_k} \vee \overline{x_i} \vee x_j)$ is equivalent to $(\overline{x_k} \vee x_j)$.

Using the above two observation, we have that our formula
$\psi \equiv (x_k \vee \overline{x_i} \vee \overline{x_j}) \wedge (\overline{x_k} \vee x_i \vee x_j) \wedge (\overline{x_k} \vee x_i \vee \overline{x_j}) \wedge (\overline{x_k} \vee \overline{x_i} \vee x_j)$
is equivalent to
$\psi \equiv (x_k \vee \overline{x_i} \vee \overline{x_j}) \wedge (\overline{x_k} \vee x_i) \wedge (\overline{x_k} \vee x_j)$.
We conclude:

Lemma

The formula $x_k = x_i \wedge x_j$ *is equivalent to the* CNF *formula*
$\psi \equiv (x_k \vee \overline{x_i} \vee \overline{x_j}) \wedge (\overline{x_k} \vee x_i) \wedge (\overline{x_k} \vee x_j)$.

Notes

Notes

Notes

Notes

NP Completeness and Cook-Levin Theorem

Lecture 22

April 22, 2014

P and NP and Turing Machines

1. **P**: set of decision problems that have polynomial time algorithms.
2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.

P and NP and Turing Machines

1. **P**: set of decision problems that have polynomial time algorithms.
2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.

Question: What is an algorithm?

P and NP and Turing Machines

1. **P**: set of decision problems that have polynomial time algorithms.
2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.

Question: What is an algorithm? Depends on the model of computation!

P and NP and Turing Machines

1. **P**: set of decision problems that have polynomial time algorithms.
2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.

Question: What is an algorithm? Depends on the model of computation!

What is our model of computation?

P and NP and Turing Machines

1. **P**: set of decision problems that have polynomial time algorithms.
2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.

Question: What is an algorithm? Depends on the model of computation!

What is our model of computation?

Formally speaking our model of computation is Turing Machines.

Turing Machines: Recap

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

Turing Machines: Formally

A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:

1. Q is set of states in finite control
2. q_0 start state, q_{accept} is accept state, q_{reject} is reject state
3. Σ is input alphabet, Γ is tape alphabet (includes $\sqcup$)
4. $\delta : Q \times \Gamma \rightarrow \{L, R\} \times \Gamma \times Q$ is transition function
   1. $\delta(q, a) = (q', b, L)$ means that M in state q and head seeing a on tape will move to state q' while replacing a on tape with b and head moves left.

$L(M)$: language accepted by M is set of all input strings s on which M accepts; that is:

1. TM is started in state q_0.
2. Initially, the tape head is located at the first cell.
3. The tape contain s on the tape followed by blanks.
4. The TM halts in the state q_{accept}.

P via TMs

Definition

M is a polynomial time TM if there is some polynomial $\mathbf{p(\cdot)}$ such that on all inputs **w**, **M** halts in $\mathbf{p(|w|)}$ steps.

Definition

L is a language in **P** iff there is a polynomial time TM **M** such that $\mathbf{L = L(M)}$.

NP via TMs

Definition

L is an **NP** language iff there is a *non-deterministic* polynomial time TM **M** such that **L = L(M)**.

NP via TMs

Definition

$\mathbf{L}$ is an **NP** language iff there is a *non-deterministic* polynomial time TM $\mathbf{M}$ such that $\mathbf{L} = \mathbf{L(M)}$.

Non-deterministic TM: each step has a choice of moves

1. $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$.
   1. Example: $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that $\mathbf{M}$ can non-deterministically choose one of the three possible moves from (q, a).
2. $\mathbf{L(M)}$: set of all strings $\mathbf{s}$ on which there *exists* some sequence of valid choices at each step that lead from q_0 to q_{accept}

Non-deterministic TMs vs certifiers

Two definition of **NP**:

1. **L** is in **NP** iff **L** has a polynomial time certifier $\mathbf{C}(\cdot, \cdot)$.
2. **L** is in **NP** iff **L** is decided by a non-deterministic polynomial time TM **M**.

Claim

Two definitions are equivalent.

Why?
Informal proof idea: the certificate **t** for **C** corresponds to non-deterministic choices of **M** and vice-versa.
In other words **L** is in **NP** iff **L** is accepted by a NTM which first guesses a proof **t** of length poly in input $|\mathbf{s}|$ and then acts as a *deterministic* TM.

Non-determinism, guessing and verification

1. A non-deterministic machine has choices at each step and accepts a string if there *exists* a set of choices which lead to a final state.
2. Equivalently the choices can be thought of as *guessing* a solution and then *verifying* that solution. In this view all the choices are made a priori and hence the verification can be deterministic. The "guess" is the "proof" and the "verifier" is the "certifier".
3. We reemphasize the asymmetry inherent in the definition of non-determinism. Strings in the language can be easily verified. No easy way to verify that a string is not in the language.

Algorithms: TMs vs RAM Model

Why do we use TMs some times and RAM Model other times?

1. TMs are very simple: no complicated instruction set, no jumps/pointers, no explicit loops etc.
   1. Simplicity is useful in proofs.
   2. The "right" formal bare-bones model when dealing with subtleties.
2. RAM model is a closer approximation to the running time/space usage of realistic computers for reasonable problem sizes
   1. Not appropriate for certain kinds of formal proofs when algorithms can take super-polynomial time and space

"Hardest" Problems

Question

What is the hardest problem in **NP**? How do we define it?

Towards a definition

1. Hardest problem must be in **NP**.
2. Hardest problem must be at least as "difficult" as every other problem in **NP**.

NP-Complete Problems

Definition

A problem **X** is said to be **NP-Complete** if

1. $\mathbf{X} \in \mathbf{NP}$, and
2. (Hardness) For any $\mathbf{Y} \in \mathbf{NP}$, $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$.

Solving NP-Complete Problems

Proposition

Suppose $\mathbf{X}$ *is* **NP-Complete**. *Then* $\mathbf{X}$ *can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Proof.

$\Rightarrow$ Suppose $\mathbf{X}$ can be solved in polynomial time

1. Let $\mathbf{Y} \in \mathbf{NP}$. We know $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$.
2. We showed that if $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$ and $\mathbf{X}$ can be solved in polynomial time, then $\mathbf{Y}$ can be solved in polynomial time.
3. Thus, every problem $\mathbf{Y} \in \mathbf{NP}$ is such that $\mathbf{Y} \in \mathbf{P}$; $\mathbf{NP} \subseteq \mathbf{P}$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $\mathbf{X} \in \mathbf{NP}$, we have a polynomial time algorithm for $\mathbf{X}$. $\square$

NP-Hard Problems

Definition

A problem **X** is said to be **NP-Hard** if

1. (Hardness) For any $\mathbf{Y} \in \mathbf{NP}$, we have that $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$.

An **NP-Hard** problem need not be in **NP**!

Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

Consequences of proving **NP-Completeness**

If **X** is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving **X** implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

Consequences of proving **NP-Completeness**

If **X** is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving **X** implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for **X**.

Consequences of proving **NP-Completeness**

If **X** is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving **X** implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for **X**.

Consequences of proving **NP-Completeness**

If **X** is **NP-Complete**

1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
2. and solving **X** implies $\mathbf{P} = \mathbf{NP}$.

X is unlikely to be efficiently solvable.

At the very least, many smart people before you have failed to find an efficient algorithm for **X**.
(This is proof by mob opinion — take with a grain of salt.)

NP-Complete Problems

Question

Are there any problems that are **NP-Complete**?

Answer

Yes! Many, many problems are **NP-Complete**.

Circuits

Definition

A circuit is a directed *acyclic* graph with

1. Input vertices (without incoming edges) labelled with $\mathbf{0}$, $\mathbf{1}$ or a distinct variable.
2. Every other vertex is labelled $\vee$, $\wedge$ or $\neg$.
3. Single node output vertex with no outgoing edges.

Circuits

Definition

A circuit is a directed *acyclic* graph with

1. Input vertices (without incoming edges) labelled with **0**, **1** or a distinct variable.
2. Every other vertex is labelled $\vee$, $\wedge$ or $\neg$.
3. Single node output vertex with no outgoing edges.

Circuits

Definition

A circuit is a directed *acyclic* graph with

1. Input vertices (without incoming edges) labelled with **0**, **1** or a distinct variable.
2. Every other vertex is labelled $\vee$, $\wedge$ or $\neg$.
3. Single node output vertex with no outgoing edges.

Cook-Levin Theorem

Definition (Circuit Satisfaction (CSAT).)

Given a circuit as input, is there an assignment to the input variables that causes the output to get value **1**?

Theorem (Cook-Levin)

CSAT *is* **NP-Complete**.

Need to show

1. **CSAT** is in **NP**.
2. *every* **NP** problem **X** reduces to **CSAT**.

Monotone CSAT?

Consider an instance of **CSAT** of size **n**, that does not contain any negations. This problem **Monotone CSAT** is

(A) **NP-Hard**.

(B) **NP-Complete**.

(C) **P**.

(D) Solvable in linear time.

(E) Solvable in $\mathbf{O(2^n)}$ time.

CSAT: Circuit Satisfaction

Claim

CSAT *is in* **NP**.

1. Certificate:
2. Certifier:

CSAT: Circuit Satisfaction

Claim

CSAT *is in* **NP**.

1. Certificate: Assignment to input variables.
2. Certifier: Evaluate the value of each gate in a topological sort of DAG and check the output gate value.

CSAT is NP-hard: Idea

Need to show that *every* **NP** problem **X** reduces to **CSAT**.

What does it mean that $\mathbf{X} \in \mathbf{NP}$?
$\mathbf{X} \in \mathbf{NP}$ implies that there are polynomials $\mathbf{p()}$ and $\mathbf{q()}$ and certifier/verifier program **C** such that for every string **s** the following is true:

1. If **s** is a YES instance ($\mathbf{s} \in \mathbf{X}$) then there is a *proof* **t** of length $\mathbf{p(|s|)}$ such that $\mathbf{C(s,t)}$ says YES.
2. If **s** is a NO instance ($\mathbf{s} \notin \mathbf{X}$) then for every string **t** of length at $\mathbf{p(|s|)}$, $\mathbf{C(s,t)}$ says NO.
3. $\mathbf{C(s,t)}$ runs in time $\mathbf{q(|s| + |t|)}$ time (hence polynomial time).

Reducing X to CSAT

X is in **NP** means we have access to $\mathbf{p(), q(), C(\cdot,\cdot)}$.
What is $\mathbf{C(\cdot,\cdot)}$? It is a program or equivalently a Turing Machine!
How are $\mathbf{p()}$ and $\mathbf{q()}$ given? As numbers.
Example: if **3** is given then $\mathbf{p(n) = n^3}$.

Thus an **NP** problem is essentially a three tuple $\langle \mathbf{p, q, C} \rangle$ where **C** is either a program or a TM.

Reducing X to CSAT

Thus an **NP** problem is essentially a three tuple $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ where $\mathbf{C}$ is either a program or TM.

Problem X: Given string $\mathbf{s}$, is $\mathbf{s} \in \mathbf{X}$?

Reducing X to CSAT

Thus an **NP** problem is essentially a three tuple $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ where $\mathbf{C}$ is either a program or TM.

Problem X: Given string $\mathbf{s}$, is $\mathbf{s} \in \mathbf{X}$?

Same as the following: is there a proof $\mathbf{t}$ of length $\mathbf{p}(|\mathbf{s}|)$ such that $\mathbf{C}(\mathbf{s}, \mathbf{t})$ says YES.

How do we reduce $\mathbf{X}$ to **CSAT**?

Reducing X to CSAT

Thus an **NP** problem is essentially a three tuple $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ where $\mathbf{C}$ is either a program or TM.

Problem X: Given string $\mathbf{s}$, is $\mathbf{s} \in \mathbf{X}$?

Same as the following: is there a proof $\mathbf{t}$ of length $\mathbf{p}(|\mathbf{s}|)$ such that $\mathbf{C}(\mathbf{s}, \mathbf{t})$ says YES.

How do we reduce $\mathbf{X}$ to **CSAT**? Need an algorithm $\mathcal{A}$ that

Reducing X to CSAT

Thus an **NP** problem is essentially a three tuple $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ where $\mathbf{C}$ is either a program or TM.

Problem X: Given string $\mathbf{s}$, is $\mathbf{s} \in \mathbf{X}$?

Same as the following: is there a proof $\mathbf{t}$ of length $\mathbf{p}(|\mathbf{s}|)$ such that $\mathbf{C}(\mathbf{s}, \mathbf{t})$ says YES.

How do we reduce $\mathbf{X}$ to **CSAT**? Need an algorithm $\mathcal{A}$ that

1. takes $\mathbf{s}$ (and $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$) and creates a circuit $\mathbf{G}$ in polynomial time in $|\mathbf{s}|$ (note that $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ are fixed).

Reducing X to CSAT

Thus an **NP** problem is essentially a three tuple $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ where $\mathbf{C}$ is either a program or TM.

Problem X: Given string $\mathbf{s}$, is $\mathbf{s} \in \mathbf{X}$?

Same as the following: is there a proof $\mathbf{t}$ of length $\mathbf{p}(|\mathbf{s}|)$ such that $\mathbf{C}(\mathbf{s}, \mathbf{t})$ says YES.

How do we reduce $\mathbf{X}$ to **CSAT**? Need an algorithm $\mathcal{A}$ that

1. takes $\mathbf{s}$ (and $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$) and creates a circuit $\mathbf{G}$ in polynomial time in $|\mathbf{s}|$ (note that $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ are fixed).
2. $\mathbf{G}$ is satisfiable if and only if there is a proof $\mathbf{t}$ such that $\mathbf{C}(\mathbf{s}, \mathbf{t})$ says YES.

Reducing **X** to **CSAT**

How do we reduce **X** to **CSAT**?

Reducing **X** to **CSAT**

How do we reduce **X** to **CSAT**? Need an algorithm $\mathcal{A}$ that

Reducing X to CSAT

How do we reduce **X** to **CSAT**? Need an algorithm $\mathcal{A}$ that

1. takes $\mathbf{s}$ (and $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$) and creates a circuit $\mathbf{G}$ in polynomial time in $|\mathbf{s}|$ (note that $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ are fixed).
2. $\mathbf{G}$ is satisfiable if and only if there is a proof $\mathbf{t}$ such that $\mathbf{C}(\mathbf{s}, \mathbf{t})$ says YES

Reducing X to CSAT

How do we reduce **X** to **CSAT**? Need an algorithm $\mathcal{A}$ that

1. takes $\mathbf{s}$ (and $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$) and creates a circuit $\mathbf{G}$ in polynomial time in $|\mathbf{s}|$ (note that $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ are fixed).
2. $\mathbf{G}$ is satisfiable if and only if there is a proof $\mathbf{t}$ such that $\mathbf{C}(\mathbf{s}, \mathbf{t})$ says YES

Simple but Big Idea: Programs are essentially the same as Circuits!

Reducing X to CSAT

How do we reduce **X** to **CSAT**? Need an algorithm $\mathcal{A}$ that

1. takes $\mathbf{s}$ (and $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$) and creates a circuit $\mathbf{G}$ in polynomial time in $|\mathbf{s}|$ (note that $\langle \mathbf{p}, \mathbf{q}, \mathbf{C} \rangle$ are fixed).
2. $\mathbf{G}$ is satisfiable if and only if there is a proof $\mathbf{t}$ such that $\mathbf{C(s, t)}$ says YES

Simple but Big Idea: Programs are essentially the same as Circuits!

1. Convert $\mathbf{C(s, t)}$ into a circuit $\mathbf{G}$ with $\mathbf{t}$ as unknown inputs (rest is known including $\mathbf{s}$)
2. We know that $|\mathbf{t}| = \mathbf{p}(|\mathbf{s}|)$ so express boolean string $\mathbf{t}$ as $\mathbf{p}(|\mathbf{s}|)$ variables $\mathbf{t_1, t_2, \ldots, t_k}$ where $\mathbf{k = p(|s|)}$.
3. Asking if there is a proof $\mathbf{t}$ that makes $\mathbf{C(s, t)}$ say YES is same as whether there is an assignment of values to "unknown" variables $\mathbf{t_1, t_2, \ldots, t_k}$ that will make $\mathbf{G}$ evaluate to true/YES.

Example: **Independent Set**

1. Problem: Does $\mathbf{G = (V, E)}$ have an **Independent Set** of size $\geq \mathbf{k}$?
   1. Certificate: Set $\mathbf{S} \subseteq \mathbf{V}$.
   2. Certifier: Check $|\mathbf{S}| \geq \mathbf{k}$ and no pair of vertices in $\mathbf{S}$ is connected by an edge.

Example: **Independent Set**

1. Problem: Does $\mathbf{G = (V, E)}$ have an **Independent Set** of size $\geq \mathbf{k}$?
   1. Certificate: Set $\mathbf{S} \subseteq \mathbf{V}$.
   2. Certifier: Check $|\mathbf{S}| \geq \mathbf{k}$ and no pair of vertices in $\mathbf{S}$ is connected by an edge.

Formally, why is **Independent Set** in **NP**?

Example: Independent Set

Formally why is **Independent Set** in **NP**?

1. Input:
   $< n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k >$ encodes $< G, k >$.
   1. n is number of vertices in G
   2. $y_{i,j}$ is a bit which is 1 if edge (i, j) is in G and 0 otherwise (adjacency matrix representation)
   3. k is size of independent set.
2. Certificate: $t = t_1 t_2 \ldots t_n$. Interpretation is that t_i is 1 if vertex i is in the independent set, 0 otherwise.

Certifier for **Independent Set**

Certifier $\mathbf{C(s,t)}$ for **Independent Set**:

```
if (t_1 + t_2 + ... + t_n < k) then
    return NO
else
    for each (i, j) do
        if (t_i ∧ t_j ∧ y_{i,j}) then
            return NO

return YES
```

Example: Independent Set

A certifier circuit for Independent Set

Figure : Graph **G** with $\mathbf{k = 2}$

What does the following formula compute?

The formula

$$F(x_1, \ldots, x_n) = \bigwedge_{i<j} (\overline{x_i} \vee \overline{x_j}) \,.$$

is true if and only if

(A) All the x_is are one.

(B) All the x_is are zero.

(C) There are exactly two ones in $x_1, \ldots, x_n$.

(D) There is at most one bit on in $x_1, \ldots, x_n$.

(E) There are at most two ones in $x_1, \ldots, x_n$.

What does the following formula compute?

The formula

$$H(x_1, \ldots, x_n) = \left(\bigwedge_{i<j} (\overline{x_i} \vee \overline{x_j}) \right) \wedge (x_1 \vee x_2 \vee \cdots x_n) .$$

is true if and only if

(A) All the x_is are one.

(B) There are exactly two ones in $x_1, \ldots, x_n$.

(C) There is exactly one bit on in $x_1, \ldots, x_n$.

(D) There is at most one bit on in $x_1, \ldots, x_n$.

(E) There are at most two ones in $x_1, \ldots, x_n$.

What does the following formula compute?

$\langle \mathbf{G} \rangle$: a vector of $\binom{n}{2}$ bits describing a graph with $\mathbf{n}$ vertices.
$\mathbf{I}(x_1, \ldots, x_n, \langle \mathbf{G} \rangle)$ formula true $\Leftrightarrow$ $x_1, \ldots, x_n$ independent set in $\mathbf{G}$.
Input: $\langle x_1^1, x_1^2, x_1^3, \quad x_2^1, x_2^2, x_2^3, \quad \ldots, \quad x_n^1, x_n^2, x_n^3, \quad \mathbf{G} \rangle$.
The formula

$$\left(\bigwedge_{i=1}^{n} \mathbf{H}\left(x_i^1, x_i^2, x_i^3\right) \right) \bigwedge \mathbf{I}\left(x_1^1, x_2^1, x_3^1, \ldots, x_n^1, \langle \mathbf{G} \rangle\right)$$
$$\bigwedge \mathbf{I}\left(x_1^2, x_2^2, x_3^2, \ldots, x_n^2, \langle \mathbf{G} \rangle\right) \bigwedge \mathbf{I}\left(x_1^3, x_2^3, x_3^3, \ldots, x_n^1, \langle \mathbf{G} \rangle\right)$$

is satisfiable if and only if

- **(A)** The graph $\mathbf{G}$ contains a clique.
- **(B)** The graph $\mathbf{G}$ can be colored by two colors.
- **(C)** The graph $\mathbf{G}$ can be colored by three colors.
- **(D)** The graph $\mathbf{G}$ encodes a satisfiable instance of 3DM.
- **(E)** None of the above.

Programs, Turing Machines and Circuits

Consider "program" **A** that takes $\mathbf{f(|s|)}$ steps on input string **s**.

Question: What computer is the program running on and what does *step* mean?

Programs, Turing Machines and Circuits

Consider "program" **A** that takes **f(|s|)** steps on input string **s**.

Question: What computer is the program running on and what does *step* mean?

Real computers difficult to reason with mathematically because

1. instruction set is too rich
2. pointers and control flow jumps in one step
3. assumption that pointer to code fits in one word

Turing Machines

1. simpler model of computation to reason with
2. can simulate real computers with *polynomial* slow down
3. all moves are *local* (head moves only one cell)

Certifiers that at TMs

Assume $\mathbf{C(\cdot,\cdot)}$ is a (deterministic) Turing Machine $\mathbf{M}$

Problem: Given $\mathbf{M}$, input $\mathbf{s}$, $\mathbf{p}$, $\mathbf{q}$ decide if there is a proof $\mathbf{t}$ of length $\mathbf{p(|s|)}$ such that $\mathbf{M}$ on $\mathbf{s, t}$ will halt in $\mathbf{q(|s|)}$ time and say YES.

There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.

1. $\mathcal{A}$ first computes $\mathbf{p(|s|)}$ and $\mathbf{q(|s|)}$.
2. Knows that $\mathbf{M}$ can use at most $\mathbf{q(|s|)}$ memory/tape cells
3. Knows that $\mathbf{M}$ can run for at most $\mathbf{q(|s|)}$ time
4. Simulates the evolution of the state of $\mathbf{M}$ and memory over time using a big circuit.

Simulation of Computation via Circuit

1. Think of $\mathbf{M}$'s state at time ℓ as a string $\mathbf{x}^\ell = \mathbf{x_1 x_2 \ldots x_k}$ where each $\mathbf{x_i} \in \{\mathbf{0, 1, B}\} \times \mathbf{Q} \cup \{\mathbf{q_{-1}}\}$.
2. At time $\mathbf{0}$ the state of $\mathbf{M}$ consists of input string $\mathbf{s}$ a guess $\mathbf{t}$ (unknown variables) of length $\mathbf{p(|s|)}$ and rest $\mathbf{q(|s|)}$ blank symbols.
3. At time $\mathbf{q(|s|)}$ we wish to know if $\mathbf{M}$ stops in $\mathbf{q_{accept}}$ with say all blanks on the tape.
4. We write a circuit $\mathbf{C}_\ell$ which captures the transition of $\mathbf{M}$ from time ℓ to time $\ell + \mathbf{1}$.
5. Composition of the circuits for all times $\mathbf{0}$ to $\mathbf{q(|s|)}$ gives a big (still poly) sized circuit $\mathcal{C}$
6. The final output of $\mathcal{C}$ should be true if and only if the entire state of $\mathbf{M}$ at the end leads to an accept state.

NP-Hardness of Circuit Satisfaction

Key Ideas in reduction:

1. Use TMs as the code for certifier for simplicity
2. Since **p()** and **q()** are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
3. Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time

NP-Hardness of Circuit Satisfaction

Key Ideas in reduction:

1. Use TMs as the code for certifier for simplicity
2. Since **p()** and **q()** are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
3. Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time

Note: Above reduction can be done to **SAT** as well. Reduction to **SAT** was the original proof of Steve Cook.

To show NP-Completeness

Clicker question

Let **X** be a decision problem.
We know that **CSAT** is **NP-Complete**.
To show that **X** is **NP-Complete** we need to:

- **(A)** Provide a polynomial time reduction from **X** to **CSAT**.
- **(B)** Provide a polynomial time reduction from **X** to **CSAT** and show that $\mathbf{X} \in \mathbf{NP}$.
- **(C)** Provide a polynomial time reduction from **CSAT** to **X**.
- **(D)** Provide a polynomial time reduction from **CSAT** to **X** and show that $\mathbf{X} \in \mathbf{NP}$.
- **(E)** Provide a polynomial time reduction from **CSAT** to **X** and show that $\mathbf{X} \notin \mathbf{P}$.

SAT is NP-Complete

1. We have seen that **SAT** $\in$ **NP**
2. To show **NP-Hardness**, we will reduce Circuit Satisfiability (**CSAT**) to **SAT**
   Instance of **CSAT** (we label each node):

Converting a circuit into a CNF formula

Label the nodes

(A) Input circuit

(B) Label the nodes.

Converting a circuit into a CNF formula

Introduce a variable for each node

(B) Label the nodes.

(C) Introduce var for each node.

Converting a circuit into a CNF formula

Write a sub-formula for each variable that is true if the var is computed correctly.

(C) Introduce var for each node.

$\mathbf{x_k}$ (Demand a sat' assignment!)

$\mathbf{x_k = x_i \wedge x_k}$

$\mathbf{x_j = x_g \wedge x_h}$

$\mathbf{x_i = \neg x_f}$

$\mathbf{x_h = x_d \vee x_e}$

$\mathbf{x_g = x_b \vee x_c}$

$\mathbf{x_f = x_a \wedge x_b}$

$\mathbf{x_d = 0}$

$\mathbf{x_a = 1}$

(D) Write a sub-formula for each variable that is true if the var is computed correctly.

Converting a circuit into a CNF formula

Convert each sub-formula to an equivalent CNF formula

x_k	x_k
$x_k = x_i \wedge x_j$	$(\neg x_k \vee x_i) \wedge (\neg x_k \vee x_j) \wedge (x_k \vee \neg x_i \vee \neg x_j)$
$x_j = x_g \wedge x_h$	$(\neg x_j \vee x_g) \wedge (\neg x_j \vee x_h) \wedge (x_j \vee \neg x_g \vee \neg x_h)$
$x_i = \neg x_f$	$(x_i \vee x_f) \wedge (\neg x_i \vee x_f)$
$x_h = x_d \vee x_e$	$(x_h \vee \neg x_d) \wedge (x_h \vee \neg x_e) \wedge (\neg x_h \vee x_d \vee x_e)$
$x_g = x_b \vee x_c$	$(x_g \vee \neg x_b) \wedge (x_g \vee \neg x_c) \wedge (\neg x_g \vee x_b \vee x_c)$
$x_f = x_a \wedge x_b$	$(\neg x_f \vee x_a) \wedge (\neg x_f \vee x_b) \wedge (x_f \vee \neg x_a \vee \neg x_b)$
$x_d = 0$	$\neg x_d$
$x_a = 1$	x_a

Converting a circuit into a CNF formula

Take the conjunction of all the CNF sub-formulas

$$
\begin{aligned}
&\mathbf{x_k} \wedge (\neg \mathbf{x_k} \vee \mathbf{x_i}) \wedge (\neg \mathbf{x_k} \vee \mathbf{x_j}) \\
&\wedge (\mathbf{x_k} \vee \neg \mathbf{x_i} \vee \neg \mathbf{x_j}) \wedge (\neg \mathbf{x_j} \vee \mathbf{x_g}) \\
&\wedge (\neg \mathbf{x_j} \vee \mathbf{x_h}) \wedge (\mathbf{x_j} \vee \neg \mathbf{x_g} \vee \neg \mathbf{x_h}) \\
&\wedge (\mathbf{x_i} \vee \mathbf{x_f}) \wedge (\neg \mathbf{x_i} \vee \mathbf{x_f}) \\
&\wedge (\mathbf{x_h} \vee \neg \mathbf{x_d}) \wedge (\mathbf{x_h} \vee \neg \mathbf{x_e}) \\
&\wedge (\neg \mathbf{x_h} \vee \mathbf{x_d} \vee \mathbf{x_e}) \wedge (\mathbf{x_g} \vee \neg \mathbf{x_b}) \\
&\wedge (\mathbf{x_g} \vee \neg \mathbf{x_c}) \wedge (\neg \mathbf{x_g} \vee \mathbf{x_b} \vee \mathbf{x_c}) \\
&\wedge (\neg \mathbf{x_f} \vee \mathbf{x_a}) \wedge (\neg \mathbf{x_f} \vee \mathbf{x_b}) \\
&\wedge (\mathbf{x_f} \vee \neg \mathbf{x_a} \vee \neg \mathbf{x_b}) \wedge (\neg \mathbf{x_d}) \wedge \mathbf{x_a}
\end{aligned}
$$

We got a CNF formula that is satisfiable if and only if the original circuit is satisfiable.

Reduction: CSAT $\leq_P$ SAT

1. For each gate (vertex) $\mathbf{v}$ in the circuit, create a variable $\mathbf{x_v}$
2. Case $\neg$: $\mathbf{v}$ is labeled $\neg$ and has one incoming edge from $\mathbf{u}$ (so $\mathbf{x_v} = \neg \mathbf{x_u}$). In **SAT** formula generate, add clauses $(\mathbf{x_u} \vee \mathbf{x_v})$, $(\neg \mathbf{x_u} \vee \neg \mathbf{x_v})$. Observe that

$$\mathbf{x_v} = \neg \mathbf{x_u} \text{ is true} \quad \Longleftrightarrow \quad \begin{array}{l} (\mathbf{x_u} \vee \mathbf{x_v}) \\ (\neg \mathbf{x_u} \vee \neg \mathbf{x_v}) \end{array} \text{ both true.}$$

Reduction: CSAT $\leq_P$ SAT

Continued...

1. Case $\vee$: So $\mathbf{x_v = x_u \vee x_w}$. In **SAT** formula generated, add clauses $\mathbf{(x_v \vee \neg x_u)}$, $\mathbf{(x_v \vee \neg x_w)}$, and $\mathbf{(\neg x_v \vee x_u \vee x_w)}$. Again, observe that

$$\left(\mathbf{x_v = x_u \vee x_w}\right) \text{ is true} \quad \Longleftrightarrow \quad \begin{array}{l} \mathbf{(x_v \vee \neg x_u)}, \\ \mathbf{(x_v \vee \neg x_w)}, \\ \mathbf{(\neg x_v \vee x_u \vee x_w)} \end{array} \quad \text{all true.}$$

Reduction: CSAT $\leq_P$ SAT

Continued...

1. Case $\wedge$: So $x_v = x_u \wedge x_w$. In **SAT** formula generated, add clauses $(\neg x_v \vee x_u)$, $(\neg x_v \vee x_w)$, and $(x_v \vee \neg x_u \vee \neg x_w)$. Again observe that

$$x_v = x_u \wedge x_w \text{ is true} \quad \Longleftrightarrow \quad \begin{array}{l} (\neg x_v \vee x_u), \\ (\neg x_v \vee x_w), \\ (x_v \vee \neg x_u \vee \neg x_w) \end{array} \quad \text{all true.}$$

Reduction: CSAT $\leq_P$ SAT

Continued...

1. If $\mathbf{v}$ is an input gate with a fixed value then we do the following. If $\mathbf{x_v = 1}$ add clause $\mathbf{x_v}$. If $\mathbf{x_v = 0}$ add clause $\neg \mathbf{x_v}$
2. Add the clause $\mathbf{x_v}$ where $\mathbf{v}$ is the variable for the output gate

Correctness of Reduction

Need to show circuit $\mathbf{C}$ is satisfiable iff $\varphi_{\mathbf{C}}$ is satisfiable

$\Rightarrow$ Consider a satisfying assignment $\mathbf{a}$ for $\mathbf{C}$

1. Find values of all gates in $\mathbf{C}$ under $\mathbf{a}$
2. Give value of gate $\mathbf{v}$ to variable $\mathbf{x_v}$; call this assignment $\mathbf{a'}$
3. $\mathbf{a'}$ satisfies $\varphi_{\mathbf{C}}$ (exercise)

$\Leftarrow$ Consider a satisfying assignment $\mathbf{a}$ for $\varphi_{\mathbf{C}}$

1. Let $\mathbf{a'}$ be the restriction of $\mathbf{a}$ to only the input variables
2. Value of gate $\mathbf{v}$ under $\mathbf{a'}$ is the same as value of $\mathbf{x_v}$ in $\mathbf{a}$
3. Thus, $\mathbf{a'}$ satisfies $\mathbf{C}$

Theorem

SAT *is* **NP-Complete**.

Proving that a problem **X** is **NP-Complete**

To prove **X** is **NP-Complete**, show

1. Show **X** is in **NP**.
   1. certificate/proof of polynomial size in input
   2. polynomial time certifier $\mathbf{C(s,t)}$
2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to **X**

Proving that a problem **X** is **NP-Complete**

To prove **X** is **NP-Complete**, show

1. Show **X** is in **NP**.
   1. certificate/proof of polynomial size in input
   2. polynomial time certifier $\mathbf{C(s,t)}$
2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to **X**

SAT $\leq_{\mathbf{P}}$ X implies that every **NP** problem **Y** $\leq_{\mathbf{P}}$ **X**. Why?

Proving that a problem **X** is **NP-Complete**

To prove **X** is **NP-Complete**, show

1. Show **X** is in **NP**.
   1. certificate/proof of polynomial size in input
   2. polynomial time certifier $\mathbf{C(s,t)}$
2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to **X**

SAT $\leq_P$ X implies that every **NP** problem $\mathbf{Y} \leq_P \mathbf{X}$. Why?
Transitivity of reductions:

$\mathbf{Y} \leq_P \mathbf{SAT}$ and $\mathbf{SAT} \leq_P \mathbf{X}$ and hence $\mathbf{Y} \leq_P \mathbf{X}$.

NP-Completeness via Reductions

1. **CSAT** is **NP-Complete**.
2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
5. **Vertex Cover** is **NP-Complete**.
6. **Clique** is **NP-Complete**.

NP-Completeness via Reductions

1. **CSAT** is **NP-Complete**.
2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
5. **Vertex Cover** is **NP-Complete**.
6. **Clique** is **NP-Complete**.

Hundreds and thousands of different problems from many areas of science and engineering have been shown to be **NP-Complete**.

A surprisingly frequent phenomenon!

Notes

Notes

Notes

Notes

More NP-Complete Problems

Lecture 23

April 24, 2014

Recap

NP: languages that have polynomial time certifiers/verifiers

Recap

NP: languages that have polynomial time certifiers/verifiers

A language **L** is **NP-Complete** iff

- **L** is in **NP**
- for every $\mathbf{L'}$ in **NP**, $\mathbf{L'} \leq_{\mathbf{P}} \mathbf{L}$

Recap

NP: languages that have polynomial time certifiers/verifiers

A language **L** is **NP-Complete** iff

- **L** is in **NP**
- for every $\mathbf{L'}$ in **NP**, $\mathbf{L'} \leq_{\mathbf{P}} \mathbf{L}$

L is **NP-Hard** if for every $\mathbf{L'}$ in **NP**, $\mathbf{L'} \leq_{\mathbf{P}} \mathbf{L}$.

Recap

NP: languages that have polynomial time certifiers/verifiers

A language **L** is **NP-Complete** iff

- **L** is in **NP**
- for every $\mathbf{L'}$ in **NP**, $\mathbf{L'} \leq_{\mathbf{P}} \mathbf{L}$

L is **NP-Hard** if for every $\mathbf{L'}$ in **NP**, $\mathbf{L'} \leq_{\mathbf{P}} \mathbf{L}$.

Theorem (Cook-Levin)

Circuit-SAT *and* **SAT** *are* **NP-Complete**.

Recap contd

Theorem (Cook-Levin)

Circuit-SAT and SAT are **NP-Complete**.

Establish **NP-Completeness** via reductions:

- SAT $\leq_{\mathbf{P}}$ 3-SAT and hence 3-SAT is **NP**-complete
- 3-SAT $\leq_{\mathbf{P}}$ Independent Set (which is in **NP**) and hence Independent Set is **NP-Complete**
- Vertex Cover is **NP-Complete**
- Clique is **NP-Complete**
- Set Cover is **NP-Complete**

Today

Prove

- Hamiltonian Cycle Problem is **NP-Complete**
- 3-Coloring is **NP-Complete**

Part I

NP-Completeness of Hamiltonian Cycle

Directed Hamiltonian Cycle

Input Given a directed graph $\mathbf{G = (V, E)}$ with $\mathbf{n}$ vertices

Goal Does $\mathbf{G}$ have a Hamiltonian cycle?

Directed Hamiltonian Cycle

Input Given a directed graph $\mathbf{G = (V, E)}$ with $\mathbf{n}$ vertices

Goal Does $\mathbf{G}$ have a Hamiltonian cycle?

- A Hamiltonian cycle is a cycle in the graph that visits every vertex in $\mathbf{G}$ exactly once

Is the following graph Hamiltonianan?

(A) Yes.

(B) No.

Directed Hamiltonian Cycle is **NP-Complete**

- Directed Hamiltonian Cycle is in **NP**
  - Certificate: Sequence of vertices
  - Certifier: Check if every vertex (except the first) appears exactly once, and that consecutive vertices are connected by a directed edge
- Hardness: We will show
  3-SAT $\leq_P$ Directed Hamiltonian Cycle

Reduction

Given 3-SAT formula φ create a graph $\mathbf{G}_\varphi$ such that

- $\mathbf{G}_\varphi$ has a Hamiltonian cycle if and only if φ is satisfiable
- $\mathbf{G}_\varphi$ should be constructible from φ by a polynomial time algorithm $\mathcal{A}$

Notation: φ has $\mathbf{n}$ variables $\mathbf{x_1, x_2, \ldots, x_n}$ and $\mathbf{m}$ clauses $\mathbf{C_1, C_2, \ldots, C_m}$.

Reduction: First Ideas

- Viewing SAT: Assign values to **n** variables, and each clauses has 3 ways in which it can be satisfied.
- Construct graph with 2^n Hamiltonian cycles, where each cycle corresponds to some boolean assignment.
- Then add more graph structure to encode constraints on assignments imposed by the clauses.

The Reduction: Phase I

- Traverse path **i** from left to right iff $\mathbf{x_i}$ is set to true
- Each path has $\mathbf{3(m+1)}$ nodes where **m** is number of clauses in φ; nodes numbered from left to right ($\mathbf{1}$ to $\mathbf{3m+3}$)

The Reduction: Phase II

- Add vertex $\mathbf{c_j}$ for clause $\mathbf{C_j}$. $\mathbf{c_j}$ has edge *from* vertex $\mathbf{3j}$ and *to* vertex $\mathbf{3j+1}$ on path $\mathbf{i}$ if $\mathbf{x_i}$ appears in clause $\mathbf{C_j}$, and has edge *from* vertex $\mathbf{3j+1}$ and *to* vertex $\mathbf{3j}$ if $\mathbf{\neg x_i}$ appears in $\mathbf{C_j}$.

The Reduction: Phase II

- Add vertex $\mathbf{c_j}$ for clause $\mathbf{C_j}$. $\mathbf{c_j}$ has edge *from* vertex $\mathbf{3j}$ and *to* vertex $\mathbf{3j+1}$ on path $\mathbf{i}$ if $\mathbf{x_i}$ appears in clause $\mathbf{C_j}$, and has edge *from* vertex $\mathbf{3j+1}$ and *to* vertex $\mathbf{3j}$ if $\mathbf{\neg x_i}$ appears in $\mathbf{C_j}$.

The Reduction: Phase II

- Add vertex $\mathbf{c_j}$ for clause $\mathbf{C_j}$. $\mathbf{c_j}$ has edge *from* vertex $\mathbf{3j}$ and *to* vertex $\mathbf{3j+1}$ on path $\mathbf{i}$ if $\mathbf{x_i}$ appears in clause $\mathbf{C_j}$, and has edge *from* vertex $\mathbf{3j+1}$ and *to* vertex $\mathbf{3j}$ if $\neg\mathbf{x_i}$ appears in $\mathbf{C_j}$.

The Reduction: Phase II

- Add vertex $\mathbf{c_j}$ for clause $\mathbf{C_j}$. $\mathbf{c_j}$ has edge *from* vertex $\mathbf{3j}$ and *to* vertex $\mathbf{3j+1}$ on path $\mathbf{i}$ if $\mathbf{x_i}$ appears in clause $\mathbf{C_j}$, and has edge *from* vertex $\mathbf{3j+1}$ and *to* vertex $\mathbf{3j}$ if $\mathbf{\neg x_i}$ appears in $\mathbf{C_j}$.

The Reduction: Phase II

- Add vertex $\mathbf{c_j}$ for clause $\mathbf{C_j}$. $\mathbf{c_j}$ has edge *from* vertex $\mathbf{3j}$ and *to* vertex $\mathbf{3j+1}$ on path $\mathbf{i}$ if $\mathbf{x_i}$ appears in clause $\mathbf{C_j}$, and has edge *from* vertex $\mathbf{3j+1}$ and *to* vertex $\mathbf{3j}$ if $\mathbf{\neg x_i}$ appears in $\mathbf{C_j}$.

The Reduction: Phase II

- Add vertex $\mathbf{c_j}$ for clause $\mathbf{C_j}$. $\mathbf{c_j}$ has edge *from* vertex $\mathbf{3j}$ and *to* vertex $\mathbf{3j+1}$ on path $\mathbf{i}$ if $\mathbf{x_i}$ appears in clause $\mathbf{C_j}$, and has edge *from* vertex $\mathbf{3j+1}$ and *to* vertex $\mathbf{3j}$ if $\mathbf{\neg x_i}$ appears in $\mathbf{C_j}$.

The Reduction: Phase II

- Add vertex $\mathbf{c_j}$ for clause $\mathbf{C_j}$. $\mathbf{c_j}$ has edge *from* vertex $\mathbf{3j}$ and *to* vertex $\mathbf{3j+1}$ on path $\mathbf{i}$ if $\mathbf{x_i}$ appears in clause $\mathbf{C_j}$, and has edge *from* vertex $\mathbf{3j+1}$ and *to* vertex $\mathbf{3j}$ if $\neg\mathbf{x_i}$ appears in $\mathbf{C_j}$.

Correctness Proof

Proposition

φ has a satisfying assignment iff $\mathbf{G}_\varphi$ has a Hamiltonian cycle.

Proof.

$\Rightarrow$ Let $\mathbf{a}$ be the satisfying assignment for φ. Define Hamiltonian cycle as follows

- If $\mathbf{a(x_i) = 1}$ then traverse path $\mathbf{i}$ from left to right
- If $\mathbf{a(x_i) = 0}$ then traverse path $\mathbf{i}$ from right to left
- For each clause, path of at least one variable is in the "right" direction to splice in the node corresponding to clause $\square$

Hamiltonian Cycle $\Rightarrow$ Satisfying assignment

Suppose Π is a Hamiltonian cycle in G_φ

- If Π enters c_j (vertex for clause C_j) from vertex $3j$ on path i then it must leave the clause vertex on edge to $3j+1$ on the *same path* i
  - If not, then only unvisited neighbor of $3j+1$ on path i is $3j+2$
  - Thus, we don't have two unvisited neighbors (one to enter from, and the other to leave) to have a Hamiltonian Cycle
- Similarly, if Π enters c_j from vertex $3j+1$ on path i then it must leave the clause vertex c_j on edge to $3j$ on path i

Example

Hamiltonian Cycle $\Longrightarrow$ Satisfying assignment (contd)

- Thus, vertices visited immediately before and after C_i are connected by an edge
- We can remove c_j from cycle, and get Hamiltonian cycle in $G - c_j$
- Consider Hamiltonian cycle in $G - \{c_1, \ldots c_m\}$; it traverses each path in only one direction, which determines the truth assignment

Is covering by cycles hard?

Given a directed graph **G**, deciding if **G** can be covered by vertex disjoint cycles (each of length at least two) is

(A) **NP-Hard**.

(B) **NP-Complete**.

(C) **P**.

(D) IDK.

Hamiltonian Cycle

Problem

Input *Given undirected graph* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$

Goal *Does* $\mathbf{G}$ *have a Hamiltonian cycle? That is, is there a cycle that visits every vertex exactly one (except start and end vertex)?*

NP-Completeness

Theorem

Hamiltonian cycle *problem for* **undirected** *graphs is* **NP-Complete**.

Proof.

- The problem is in **NP**; proof left as exercise.
- Hardness proved by reducing Directed Hamiltonian Cycle to this problem □

Reduction Sketch

Goal: Given directed graph **G**, need to construct undirected graph **G′** such that **G** has Hamiltonian Path iff **G′** has Hamiltonian path

Reduction

Reduction Sketch

Goal: Given directed graph $\mathbf{G}$, need to construct undirected graph $\mathbf{G'}$ such that $\mathbf{G}$ has Hamiltonian Path iff $\mathbf{G'}$ has Hamiltonian path

Reduction

- Replace each vertex $\mathbf{v}$ by 3 vertices: $\mathbf{v_{in}}$, $\mathbf{v}$, and $\mathbf{v_{out}}$

Reduction Sketch

Goal: Given directed graph $\mathbf{G}$, need to construct undirected graph $\mathbf{G'}$ such that $\mathbf{G}$ has Hamiltonian Path iff $\mathbf{G'}$ has Hamiltonian path

Reduction

- Replace each vertex $\mathbf{v}$ by 3 vertices: $\mathbf{v_{in}}, \mathbf{v}$, and $\mathbf{v_{out}}$
- A directed edge $(\mathbf{a}, \mathbf{b})$ is replaced by edge $(\mathbf{a_{out}}, \mathbf{b_{in}})$

Reduction Sketch

Goal: Given directed graph $\mathbf{G}$, need to construct undirected graph $\mathbf{G'}$ such that $\mathbf{G}$ has Hamiltonian Path iff $\mathbf{G'}$ has Hamiltonian path

Reduction

- Replace each vertex $\mathbf{v}$ by 3 vertices: $\mathbf{v_{in}}, \mathbf{v}$, and $\mathbf{v_{out}}$
- A directed edge $(\mathbf{a}, \mathbf{b})$ is replaced by edge $(\mathbf{a_{out}}, \mathbf{b_{in}})$

Reduction: Wrapup

- The reduction is polynomial time (exercise)
- The reduction is correct (exercise)

Part II

NP-Completeness of Graph Coloring

Graph Coloring

Problem: Graph Coloring

Instance: $\mathbf{G = (V, E)}$: Undirected graph, integer $\mathbf{k}$.
Question: Can the vertices of the graph be colored using $\mathbf{k}$ colors so that vertices connected by an edge do not get the same color?

Graph 3-Coloring

Problem: 3 Coloring

Instance: $G = (V, E)$: Undirected graph.
Question: Can the vertices of the graph be colored using **3** colors so that vertices connected by an edge do not get the same color?

Graph 3-Coloring

Problem: 3 Coloring

Instance: $G = (V, E)$: Undirected graph.
Question: Can the vertices of the graph be colored using **3** colors so that vertices connected by an edge do not get the same color?

Graph Coloring

Observation: If $\mathbf{G}$ is colored with $\mathbf{k}$ colors then each color class (nodes of same color) form an independent set in $\mathbf{G}$. Thus, $\mathbf{G}$ can be partitioned into $\mathbf{k}$ independent sets iff $\mathbf{G}$ is $\mathbf{k}$-colorable.

Graph Coloring

Observation: If $\mathbf{G}$ is colored with $\mathbf{k}$ colors then each color class (nodes of same color) form an independent set in $\mathbf{G}$. Thus, $\mathbf{G}$ can be partitioned into $\mathbf{k}$ independent sets iff $\mathbf{G}$ is $\mathbf{k}$-colorable.

Graph $\mathbf{2}$-Coloring can be decided in polynomial time.

Graph Coloring

Observation: If $\mathbf{G}$ is colored with $\mathbf{k}$ colors then each color class (nodes of same color) form an independent set in $\mathbf{G}$. Thus, $\mathbf{G}$ can be partitioned into $\mathbf{k}$ independent sets iff $\mathbf{G}$ is $\mathbf{k}$-colorable.

Graph $\mathbf{2}$-Coloring can be decided in polynomial time.

$\mathbf{G}$ is $\mathbf{2}$-colorable iff $\mathbf{G}$ is bipartite!

Graph Coloring

Observation: If **G** is colored with **k** colors then each color class (nodes of same color) form an independent set in **G**. Thus, **G** can be partitioned into **k** independent sets iff **G** is **k**-colorable.

Graph **2**-Coloring can be decided in polynomial time.

G is **2**-colorable iff **G** is bipartite! There is a linear time algorithm to check if **G** is bipartite using **BFS** (we saw this earlier).

Graph Coloring and Register Allocation

Register Allocation

Assign variables to (at most) **k** registers such that variables needed at the same time are not assigned to the same register

Interference Graph

Vertices are variables, and there is an edge between two vertices, if the two variables are "live" at the same time.

Observations

- [Chaitin] Register allocation problem is equivalent to coloring the interference graph with **k** colors
- Moreover, **3-COLOR** $\leq_P$ **k-Register Allocation**, for any $k \geq 3$

Class Room Scheduling

Given **n** classes and their meeting times, are **k** rooms sufficient?

Class Room Scheduling

Given $\mathbf{n}$ classes and their meeting times, are $\mathbf{k}$ rooms sufficient?

Reduce to Graph $\mathbf{k}$-Coloring problem

Create graph $\mathbf{G}$

- a node $\mathbf{v_i}$ for each class $\mathbf{i}$
- an edge between $\mathbf{v_i}$ and $\mathbf{v_j}$ if classes $\mathbf{i}$ and $\mathbf{j}$ *conflict*

Class Room Scheduling

Given $\mathbf{n}$ classes and their meeting times, are $\mathbf{k}$ rooms sufficient?

Reduce to Graph $\mathbf{k}$-Coloring problem

Create graph $\mathbf{G}$

- a node $\mathbf{v_i}$ for each class $\mathbf{i}$
- an edge between $\mathbf{v_i}$ and $\mathbf{v_j}$ if classes $\mathbf{i}$ and $\mathbf{j}$ *conflict*

Exercise: $\mathbf{G}$ is $\mathbf{k}$-colorable iff $\mathbf{k}$ rooms are sufficient

Frequency Assignments in Cellular Networks

Cellular telephone systems that use Frequency Division Multiple Access (FDMA) (example: GSM in Europe and Asia and AT&T in USA)

- Breakup a frequency range $[a, b]$ into disjoint *bands* of frequencies $[a_0, b_0], [a_1, b_1], \ldots, [a_k, b_k]$
- Each cell phone tower (simplifying) gets one band
- Constraint: nearby towers cannot be assigned same band, otherwise signals will interference

Frequency Assignments in Cellular Networks

Cellular telephone systems that use Frequency Division Multiple Access (FDMA) (example: GSM in Europe and Asia and AT&T in USA)

- Breakup a frequency range $[a, b]$ into disjoint *bands* of frequencies $[a_0, b_0], [a_1, b_1], \ldots, [a_k, b_k]$
- Each cell phone tower (simplifying) gets one band
- Constraint: nearby towers cannot be assigned same band, otherwise signals will interference

Problem: given k bands and some region with n towers, is there a way to assign the bands to avoid interference?

Can reduce to k-coloring by creating intereference/conflict graph on towers.

3 color this gadget.

You are given three colors: red, green and blue. Can the following graph be three colored in a valid way (assuming the two nodes are already colored as indicated).

(A) Yes.

(B) No.

3 color this gadget II

You are given three colors: red, green and blue. Can the following graph be three colored in a valid way (assuming the two nodes are already colored as indicated).

(A) Yes.

(B) No.

3-Coloring is NP-Complete

- **3-Coloring** is in **NP**.
  - Certificate: for each node a color from $\{1, 2, 3\}$.
  - Certifier: Check if for each edge (u, v), the color of u is different from that of v.
- Hardness: We will show **3-SAT** $\leq_P$ **3-Coloring**.

Reduction Idea

Start with **3SAT** formula (i.e., **3**CNF formula) φ with $\mathbf{n}$ variables $\mathbf{x_1, \ldots, x_n}$ and $\mathbf{m}$ clauses $\mathbf{C_1, \ldots, C_m}$. Create graph $\mathbf{G_\varphi}$ such that $\mathbf{G_\varphi}$ is 3-colorable iff φ is satisfiable

- need to establish truth assignment for $\mathbf{x_1, \ldots, x_n}$ via colors for some nodes in $\mathbf{G_\varphi}$.

Reduction Idea

Start with **3SAT** formula (i.e., **3**CNF formula) φ with $\mathbf{n}$ variables $\mathbf{x_1, \ldots, x_n}$ and $\mathbf{m}$ clauses $\mathbf{C_1, \ldots, C_m}$. Create graph $\mathbf{G_\varphi}$ such that $\mathbf{G_\varphi}$ is 3-colorable iff φ is satisfiable

- need to establish truth assignment for $\mathbf{x_1, \ldots, x_n}$ via colors for some nodes in $\mathbf{G_\varphi}$.
- create triangle with node True, False, Base

Reduction Idea

Start with **3SAT** formula (i.e., **3**CNF formula) φ with $\mathbf{n}$ variables $\mathbf{x_1, \ldots, x_n}$ and $\mathbf{m}$ clauses $\mathbf{C_1, \ldots, C_m}$. Create graph $\mathbf{G_\varphi}$ such that $\mathbf{G_\varphi}$ is 3-colorable iff φ is satisfiable

- need to establish truth assignment for $\mathbf{x_1, \ldots, x_n}$ via colors for some nodes in $\mathbf{G_\varphi}$.
- create triangle with node True, False, Base
- for each variable $\mathbf{x_i}$ two nodes $\mathbf{v_i}$ and $\bar{\mathbf{v}}_\mathbf{i}$ connected in a triangle with common Base

Reduction Idea

Start with **3SAT** formula (i.e., **3**CNF formula) φ with $\mathbf{n}$ variables $\mathbf{x_1, \ldots, x_n}$ and $\mathbf{m}$ clauses $\mathbf{C_1, \ldots, C_m}$. Create graph $\mathbf{G_\varphi}$ such that $\mathbf{G_\varphi}$ is 3-colorable iff φ is satisfiable

- need to establish truth assignment for $\mathbf{x_1, \ldots, x_n}$ via colors for some nodes in $\mathbf{G_\varphi}$.
- create triangle with node True, False, Base
- for each variable $\mathbf{x_i}$ two nodes $\mathbf{v_i}$ and $\bar{\mathbf{v}}_\mathbf{i}$ connected in a triangle with common Base
- If graph is 3-colored, either $\mathbf{v_i}$ or $\bar{\mathbf{v}}_\mathbf{i}$ gets the same color as True. Interpret this as a truth assignment to $\mathbf{v_i}$

Reduction Idea

Start with **3SAT** formula (i.e., **3**CNF formula) φ with $\mathbf{n}$ variables $\mathbf{x_1, \ldots, x_n}$ and $\mathbf{m}$ clauses $\mathbf{C_1, \ldots, C_m}$. Create graph $\mathbf{G_\varphi}$ such that $\mathbf{G_\varphi}$ is 3-colorable iff φ is satisfiable

- need to establish truth assignment for $\mathbf{x_1, \ldots, x_n}$ via colors for some nodes in $\mathbf{G_\varphi}$.
- create triangle with node True, False, Base
- for each variable $\mathbf{x_i}$ two nodes $\mathbf{v_i}$ and $\bar{\mathbf{v}}_\mathbf{i}$ connected in a triangle with common Base
- If graph is 3-colored, either $\mathbf{v_i}$ or $\bar{\mathbf{v}}_\mathbf{i}$ gets the same color as True. Interpret this as a truth assignment to $\mathbf{v_i}$
- Need to add constraints to ensure clauses are satisfied (next phase)

Figure

Clause Satisfiability Gadget

For each clause $\mathbf{C_j = (a \vee b \vee c)}$, create a small gadget graph

- gadget graph connects to nodes corresponding to **a, b, c**
- needs to implement OR

OR-gadget-graph:

OR-Gadget Graph

Property: if $\mathbf{a}, \mathbf{b}, \mathbf{c}$ are colored False in a 3-coloring then output node of OR-gadget has to be colored False.

Property: if one of $\mathbf{a}, \mathbf{b}, \mathbf{c}$ is colored True then OR-gadget can be 3-colored such that output node of OR-gadget is colored True.

Reduction

- create triangle with nodes True, False, Base
- for each variable $\mathbf{x_i}$ two nodes $\mathbf{v_i}$ and $\bar{\mathbf{v}}_\mathbf{i}$ connected in a triangle with common Base
- for each clause $\mathbf{C_j = (a \vee b \vee c)}$, add OR-gadget graph with input nodes $\mathbf{a, b, c}$ and connect output node of gadget to both False and Base

Reduction

Claim

No legal **3***-coloring of above graph (with coloring of nodes* **T, F, B** *fixed) in which* **a, b, c** *are colored False. If any of* **a, b, c** *are colored True then there is a legal* **3***-coloring of above graph.*

3 coloring of the clause gadget

FFF - **BAD**

FFT

FTF

FTT

TFF

TFT

TTF

TTT

Reduction Outline

Example

$\varphi = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$

Correctness of Reduction

φ is satisfiable implies $\mathbf{G}_\varphi$ is 3-colorable

- if $\mathbf{x_i}$ is assigned True, color $\mathbf{v_i}$ True and $\bar{\mathbf{v}}_\mathbf{i}$ False

Correctness of Reduction

φ is satisfiable implies $\mathbf{G}_\varphi$ is 3-colorable

- if $\mathbf{x_i}$ is assigned True, color $\mathbf{v_i}$ True and $\bar{\mathbf{v}}_\mathbf{i}$ False
- for each clause $\mathbf{C_j = (a \vee b \vee c)}$ at least one of $\mathbf{a, b, c}$ is colored True. OR-gadget for $\mathbf{C_j}$ can be 3-colored such that output is True.

Correctness of Reduction

φ is satisfiable implies $\mathbf{G}_\varphi$ is 3-colorable

- if $\mathbf{x_i}$ is assigned True, color $\mathbf{v_i}$ True and $\bar{\mathbf{v}}_\mathbf{i}$ False
- for each clause $\mathbf{C_j = (a \vee b \vee c)}$ at least one of $\mathbf{a, b, c}$ is colored True. OR-gadget for $\mathbf{C_j}$ can be 3-colored such that output is True.

Correctness of Reduction

φ is satisfiable implies $\mathbf{G}_\varphi$ is 3-colorable

- if $\mathbf{x_i}$ is assigned True, color $\mathbf{v_i}$ True and $\bar{\mathbf{v}}_\mathbf{i}$ False
- for each clause $\mathbf{C_j = (a \vee b \vee c)}$ at least one of $\mathbf{a, b, c}$ is colored True. OR-gadget for $\mathbf{C_j}$ can be 3-colored such that output is True.

$\mathbf{G}_\varphi$ is 3-colorable implies φ is satisfiable

- if $\mathbf{v_i}$ is colored True then set $\mathbf{x_i}$ to be True, this is a legal truth assignment

Correctness of Reduction

φ is satisfiable implies $\mathbf{G}_\varphi$ is 3-colorable

- if $\mathbf{x_i}$ is assigned True, color $\mathbf{v_i}$ True and $\bar{\mathbf{v}}_\mathbf{i}$ False
- for each clause $\mathbf{C_j = (a \vee b \vee c)}$ at least one of $\mathbf{a, b, c}$ is colored True. OR-gadget for $\mathbf{C_j}$ can be 3-colored such that output is True.

$\mathbf{G}_\varphi$ is 3-colorable implies φ is satisfiable

- if $\mathbf{v_i}$ is colored True then set $\mathbf{x_i}$ to be True, this is a legal truth assignment
- consider any clause $\mathbf{C_j = (a \vee b \vee c)}$. it cannot be that all $\mathbf{a, b, c}$ are False. If so, output of OR-gadget for $\mathbf{C_j}$ has to be colored False but output is connected to Base and False!

Graph generated in reduction...

... from 3SAT to 3COLOR

Part III

Hardness of **Subset Sum**

Subset Sum

Problem: Subset Sum

Instance: $\mathbf{S}$ - set of positive integers, $\mathbf{t}$: - an integer number (Target)
Question: Is there a subset $\mathbf{X} \subseteq \mathbf{S}$ such that $\sum_{x \in X} x = t$?

Claim

Subset Sum *is* **NP-Complete**.

Vec Subset Sum

We will prove following problem is **NP-Complete**...

Problem: Vec Subset Sum

Instance: $\mathbf{S}$ - set of $\mathbf{n}$ vectors of dimension $\mathbf{k}$, each vector has non-negative numbers for its coordinates, and a target vector $\overrightarrow{\mathbf{t}}$.
Question: Is there a subset $\mathbf{X} \subseteq \mathbf{S}$ such that $\sum_{\overrightarrow{\mathbf{x}} \in \mathbf{X}} \overrightarrow{\mathbf{x}} = \overrightarrow{\mathbf{t}}$?

Reduction from **3SAT**.

Vec Subset Sum

Handling a single clause

Think about vectors as being lines in a table.

First gadget

Selecting between two lines.

Target	??	??	01	???
$\mathbf{a}_1$	??	??	01	??
$\mathbf{a}_2$	??	??	01	??

Two rows for every variable $\mathbf{x}$: selecting either $\mathbf{x} = \mathbf{0}$ or $\mathbf{x} = \mathbf{1}$.

Handling a clause...

We will have a column for every clause...

numbers	...	$\mathbf{C} \equiv \mathbf{a} \vee \mathbf{b} \vee \overline{\mathbf{c}}$	...
$\mathbf{a}$	...	01	...
$\overline{\mathbf{a}}$	...	00	...
$\mathbf{b}$	...	01	...
$\overline{\mathbf{b}}$	...	00	...
$\mathbf{c}$	...	00	...
$\overline{\mathbf{c}}$	...	01	...
$\mathbf{C}$ fix-up 1	000	07	000
$\mathbf{C}$ fix-up 2	000	08	000
$\mathbf{C}$ fix-up 3	000	09	000
TARGET		10	

3SAT to Vec Subset Sum

numbers	$a \vee \bar{a}$	$b \vee \bar{b}$	$c \vee \bar{c}$	$d \vee \bar{d}$	$D \equiv \bar{b} \vee c \vee \bar{d}$	$C \equiv a \vee b \vee \bar{c}$
a	1	0	0	0	00	01
$\bar{a}$	1	0	0	0	00	00
b	0	1	0	0	00	01
$\bar{b}$	0	1	0	0	01	00
c	0	0	1	0	01	00
$\bar{c}$	0	0	1	0	00	01
d	0	0	0	1	00	00
$\bar{d}$	0	0	0	1	01	01
C fix-up 1	0	0	0	0	00	07
C fix-up 2	0	0	0	0	00	08
C fix-up 3	0	0	0	0	00	09
D fix-up 1	0	0	0	0	07	00
D fix-up 2	0	0	0	0	08	00
D fix-up 3	0	0	0	0	09	00
TARGET	1	1	1	1	10	10

Vec Subset Sum to Subset Sum

numbers
010000000001
010000000000
000100000001
000100000100
000001000100
000001000001
000000010000
000000010101
000000000007
000000000008
000000000009
000000000700
000000000800
000000000900

Other **NP-Complete** Problems

- 3-Dimensional Matching
- Subset Sum

Read book.

Need to Know **NP-Complete** Problems

- 3-SAT
- Circuit-SAT
- Independent Set
- Vertex Cover
- Clique
- Set Cover
- Hamiltonian Cycle in Directed/Undirected Graphs
- 3-Coloring
- 3-D Matching
- Subset Sum

Subset Sum and Knapsack

Subset Sum Problem: Given $\mathbf{n}$ integers $\mathbf{a_1, a_2, \ldots, a_n}$ and a target $\mathbf{B}$, is there a subset of $\mathbf{S}$ of $\{\mathbf{a_1, \ldots, a_n}\}$ such that the numbers in $\mathbf{S}$ add up *precisely* to $\mathbf{B}$?

Subset Sum and Knapsack

Subset Sum Problem: Given $\mathbf{n}$ integers $\mathbf{a_1, a_2, \ldots, a_n}$ and a target $\mathbf{B}$, is there a subset of $\mathbf{S}$ of $\{\mathbf{a_1, \ldots, a_n}\}$ such that the numbers in $\mathbf{S}$ add up *precisely* to $\mathbf{B}$?

Subset Sum is **NP-Complete**— see book.

Subset Sum and Knapsack

Subset Sum Problem: Given $\mathbf{n}$ integers $\mathbf{a_1, a_2, \ldots, a_n}$ and a target $\mathbf{B}$, is there a subset of $\mathbf{S}$ of $\{\mathbf{a_1, \ldots, a_n}\}$ such that the numbers in $\mathbf{S}$ add up *precisely* to $\mathbf{B}$?

Subset Sum is **NP-Complete**— see book.

Knapsack: Given $\mathbf{n}$ items with item $\mathbf{i}$ having size $\mathbf{s_i}$ and profit $\mathbf{p_i}$, a knapsack of capacity $\mathbf{B}$, and a target profit $\mathbf{P}$, is there a subset $\mathbf{S}$ of items that can be packed in the knapsack and the profit of $\mathbf{S}$ is at least $\mathbf{P}$?

Subset Sum and Knapsack

Subset Sum Problem: Given $\mathbf{n}$ integers $\mathbf{a_1, a_2, \ldots, a_n}$ and a target $\mathbf{B}$, is there a subset of $\mathbf{S}$ of $\{\mathbf{a_1, \ldots, a_n}\}$ such that the numbers in $\mathbf{S}$ add up *precisely* to $\mathbf{B}$?

Subset Sum is **NP-Complete**— see book.

Knapsack: Given $\mathbf{n}$ items with item $\mathbf{i}$ having size $\mathbf{s_i}$ and profit $\mathbf{p_i}$, a knapsack of capacity $\mathbf{B}$, and a target profit $\mathbf{P}$, is there a subset $\mathbf{S}$ of items that can be packed in the knapsack and the profit of $\mathbf{S}$ is at least $\mathbf{P}$?

Show Knapsack problem is **NP-Complete** via reduction from Subset Sum (exercise).

Subset Sum and Knapsack

Subset Sum can be solved in **O(nB)** time using dynamic programming (exercise).

Subset Sum and Knapsack

Subset Sum can be solved in $\mathbf{O(nB)}$ time using dynamic programming (exercise).

Implies that problem is hard only when numbers $\mathbf{a_1, a_2, \ldots, a_n}$ are exponentially large compared to $\mathbf{n}$. That is, each $\mathbf{a_i}$ requires polynomial in $\mathbf{n}$ bits.

Subset Sum and Knapsack

Subset Sum can be solved in $\mathbf{O(nB)}$ time using dynamic programming (exercise).

Implies that problem is hard only when numbers $\mathbf{a_1, a_2, \ldots, a_n}$ are exponentially large compared to $\mathbf{n}$. That is, each $\mathbf{a_i}$ requires polynomial in $\mathbf{n}$ bits.

Number problems of the above type are said to be **weakly NPComplete**.

Notes

Notes

Notes

Notes

S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Comput.*, 5(4):691–703, 1976.

co-NP, Self-Reductions

Lecture 24

April 29, 2014

Find this path

G: A directed graph G with weights on edges, vertices **s** and **t**, and a number $\Delta \geq 0$. Consider the following problems, that all ask if there exists a path π between **s** and **t** with a desired property. We have that

1. **ShortestP**: $\text{length}(\pi) \leq \Delta$.
2. **ExactP**: $\text{length}(\pi) = \Delta$.
3. **LongestP**: $\text{length}(\pi) \geq \Delta$.

(A) **ShortestP** $\in$ **P**, **ExactP** $\in$ **P**, **LongestP** $\in$ **P**.

(B) **ShortestP** $\in$ **P**, **ExactP** $\in$ **P**, **LongestP** $\in$ **NPC**.

(C) **ShortestP** $\in$ **NPC**, **ExactP** $\in$ **P**, **LongestP** $\in$ **P**.

(D) **ShortestP** $\in$ **P**, **ExactP** $\in$ **NPC**, **LongestP** $\in$ **NPC**.

(E) **ShortestP** $\in$ **NPC**, **ExactP** $\in$ **NPC**, **LongestP** $\in$ **NPC**.

Part I

Complementation and Self-Reduction

The class **P**

1. A language **L** (equivalently decision problem) is in the class **P** if there is a polynomial time algorithm **A** for deciding **L**; that is given a string **x**, **A** correctly decides if $\mathbf{x} \in \mathbf{L}$ and running time of **A** on **x** is polynomial in $|\mathbf{x}|$, the length of **x**.

The class NP

Two equivalent definitions:

1. Language $\mathbf{L}$ is in **NP** if there is a non-deterministic polynomial time algorithm $\mathbf{A}$ (Turing Machine) that decides $\mathbf{L}$.
   1. For $\mathbf{x} \in \mathbf{L}$, $\mathbf{A}$ has some non-deterministic choice of moves that will make $\mathbf{A}$ accept $\mathbf{x}$
   2. For $\mathbf{x} \notin \mathbf{L}$, no choice of moves will make $\mathbf{A}$ accept $\mathbf{x}$
2. $\mathbf{L}$ has an efficient certifier $\mathbf{C}(\cdot, \cdot)$.
   1. $\mathbf{C}$ is a polynomial time deterministic algorithm
   2. For $\mathbf{x} \in \mathbf{L}$ there is a string $\mathbf{y}$ (proof) of length polynomial in $|\mathbf{x}|$ such that $\mathbf{C}(\mathbf{x}, \mathbf{y})$ accepts
   3. For $\mathbf{x} \notin \mathbf{L}$, no string $\mathbf{y}$ will make $\mathbf{C}(\mathbf{x}, \mathbf{y})$ accept

Short certificates...

Let **X** be an **NP-Complete** problem, such that for any YES instance **I** of **X** of length **n**, there exists a certificate of length $\mathbf{O(\log n)}$ that it is correct.

If such a problem **X** exists then

(A) Nothing at all is implied by this.

(B) $\mathbf{P = NP}$.

(C) $\mathbf{P \neq NP}$.

Complementation

Definition

Given a decision problem $\mathbf{X}$, its **complement** $\overline{\mathbf{X}}$ is the collection of all instances $\mathbf{s}$ such that $\mathbf{s} \notin \mathbf{L(X)}$

Equivalently, in terms of languages:

Definition

Given a language $\mathbf{L}$ over alphabet $\mathbf{\Sigma}$, its **complement** $\overline{\mathbf{L}}$ is the language $\mathbf{\Sigma}^* \setminus \mathbf{L}$.

Examples

1. $\textbf{PRIME} = \{\mathbf{n} \mid \mathbf{n} \text{ is an integer and } \mathbf{n} \text{ is prime}\}$
   $\overline{\textbf{PRIME}} = \left\{\mathbf{n} \;\middle|\; \mathbf{n} \text{ is an integer and } \mathbf{n} \text{ is not a prime}\right\}$
   $\overline{\textbf{PRIME}} = \textbf{COMPOSITE}$.
2. $\textbf{SAT} = \left\{\varphi \;\middle|\; \varphi \text{ is a } \mathrm{CNF} \text{ formula and } \varphi \text{ is satisfiable}\right\}$
   $\overline{\textbf{SAT}} = \left\{\varphi \;\middle|\; \varphi \text{ is a } \mathrm{CNF} \text{ formula and } \varphi \text{ is not satisfiable}\right\}$.
   $\overline{\textbf{SAT}} = \textbf{UnSAT}$.

Examples

1. $\textbf{PRIME} = \{\mathbf{n} \mid \mathbf{n} \text{ is an integer and } \mathbf{n} \text{ is prime}\}$
   $\overline{\textbf{PRIME}} = \left\{\mathbf{n} \;\middle|\; \mathbf{n} \text{ is an integer and } \mathbf{n} \text{ is not a prime}\right\}$
   $\overline{\textbf{PRIME}} = \textbf{COMPOSITE}$.
2. $\textbf{SAT} = \left\{\varphi \;\middle|\; \varphi \text{ is a } \mathrm{CNF} \text{ formula and } \varphi \text{ is satisfiable}\right\}$
   $\overline{\textbf{SAT}} = \left\{\varphi \;\middle|\; \varphi \text{ is a } \mathrm{CNF} \text{ formula and } \varphi \text{ is not satisfiable}\right\}$.
   $\overline{\textbf{SAT}} = \textbf{UnSAT}$.

Technicality: $\overline{\textbf{SAT}}$ also includes strings that do not encode any valid CNF formula. Typically we ignore those strings because they are not interesting. In all problems of interest, we assume that it is "easy" to check whether a given string is a valid instance or not.

P is closed under complementation

Proposition

Decision problem $\mathbf{X}$ *is in* $\mathbf{P}$ *if and only if* $\overline{\mathbf{X}}$ *is in* $\mathbf{P}$.

Proof.

1. If $\mathbf{X}$ is in $\mathbf{P}$ let $\mathbf{A}$ be a polynomial time algorithm for $\mathbf{X}$.
2. Construct polynomial time algorithm $\mathbf{A}'$ for $\overline{\mathbf{X}}$ as follows: given input $\mathbf{x}$, $\mathbf{A}'$ runs $\mathbf{A}$ on $\mathbf{x}$ and if $\mathbf{A}$ accepts $\mathbf{x}$, $\mathbf{A}'$ rejects $\mathbf{x}$ and if $\mathbf{A}$ rejects $\mathbf{x}$ then $\mathbf{A}'$ accepts $\mathbf{x}$.
3. Only if direction is essentially the same argument.

□

Asymmetry of NP

Definition

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Observation

To show that a problem is in **NP** we only need short, efficiently checkable certificates for "yes"-instances. What about "no"-instances?

Asymmetry of NP

Definition

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Observation

To show that a problem is in **NP** we only need short, efficiently checkable certificates for "yes"-instances. What about "no"-instances?

Given a CNF formula φ, is φ unsatisfiable?

Asymmetry of NP

Definition

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Observation

To show that a problem is in **NP** we only need short, efficiently checkable certificates for "yes"-instances. What about "no"-instances?

Given a CNF formula φ, is φ unsatisfiable?

Easy to give a proof that φ is satisfiable (an assignment) but no easy (known) proof to show that φ is unsatisfiable!

Examples of complement problems

Some languages

1. **UnSAT**: CNF formulas φ that are not satisfiable
2. **No-Hamilton-Cycle**: graphs **G** that do not have a Hamilton cycle
3. **No-3-Color**: graphs **G** that are not 3-colorable

Examples of complement problems

Some languages

1. **UnSAT**: CNF formulas φ that are not satisfiable
2. **No-Hamilton-Cycle**: graphs **G** that do not have a Hamilton cycle
3. **No-3-Color**: graphs **G** that are not 3-colorable

Above problems are complements of known **NP** problems (viewed as languages).

NP and co-NP

NP

Decision problems with a polynomial certifier.
Examples: **SAT**, **Hamiltonian Cycle**, **3-Colorability**.

Definition

co-NP is the class of all decision problems $\mathbf{X}$ such that $\overline{\mathbf{X}} \in \mathbf{NP}$.
Examples: **UnSAT**, **No-Hamiltonian-Cycle**, **No-3-Colorable**.

co-NP

If $\mathbf{L}$ is a language in **co-NP** then that there is a polynomial time certifier/verifier $\mathbf{C(\cdot,\cdot)}$, such that:

1. for $\mathbf{s} \notin \mathbf{L}$ there is a proof $\mathbf{t}$ of size polynomial in $|\mathbf{s}|$ such that $\mathbf{C(s,t)}$ correctly says NO.
2. for $\mathbf{s} \in \mathbf{L}$ there is no proof $\mathbf{t}$ for which $\mathbf{C(s,t)}$ will say NO

co-NP

If $\mathbf{L}$ is a language in **co-NP** then that there is a polynomial time certifier/verifier $\mathbf{C(\cdot, \cdot)}$, such that:

1. for $\mathbf{s} \notin \mathbf{L}$ there is a proof $\mathbf{t}$ of size polynomial in $|\mathbf{s}|$ such that $\mathbf{C(s, t)}$ correctly says NO.
2. for $\mathbf{s} \in \mathbf{L}$ there is no proof $\mathbf{t}$ for which $\mathbf{C(s, t)}$ will say NO

co-NP has checkable proofs for strings NOT in the language.

Natural Problems in co-NP

1. **Tautology**: given a Boolean formula (not necessarily in CNF form), is it true for *all* possible assignments to the variables?
2. **Graph expansion**: given a graph $\mathbf{G}$, is it an *expander*? A graph $\mathbf{G = (V, E)}$ is an **expander** if and only if for each $\mathbf{S \subset V}$ with $|\mathbf{S}| \leq |\mathbf{V}|/\mathbf{2}$, $|\mathbf{N(S)}| \geq |\mathbf{S}|$. Expanders are very important graphs in theoretical computer science and mathematics.

Factorization, Primality

Problem: Primality

Instance: An integer $\mathbf{n}$.
Question: Is the number $\mathbf{n}$ prime?

Problem: Factoring

Instance: Integers $\mathbf{n}, \mathbf{k}$.
Question: Does the number $\mathbf{n}$ have a factor $\leq \mathbf{k}$? Formally, is there ℓ, such that $\mathbf{2} \leq \ell \leq \mathbf{k}$, such that ℓ divides $\mathbf{n}$?

1. **Primality** is in **P**.
2. **Factoring** is in **NP** $\cap$ **co-NP**.

Factoring is a very naughty problem

Problem: Factoring

Instance: Integers $\mathbf{n}, \mathbf{k}$.
Question: Does the number $\mathbf{n}$ has a factor $\leq \mathbf{k}$? Formally, is there ℓ, such that $\mathbf{2} \leq \ell \leq \mathbf{k}$, such that ℓ divides $\mathbf{n}$?

If answer is:

1. NO: certificate is all prime factors of $\mathbf{n}$. Certification: multiply the given numbers.
2. YES: Certificate is the factor ℓ. Verify it divides $\mathbf{n}$.

Belief: Unlikely **Factoring** is **NP-Complete**. Can be solved in polynomial time on a quantum computer.

P, NP, co-NP

co-P: complement of P. Language $\mathbf{X}$ is in **co-P** iff $\overline{\mathbf{X}} \in \mathbf{P}$

P, NP, co-NP

co-P: complement of P. Language $\mathbf{X}$ is in **co-P** iff $\overline{\mathbf{X}} \in \mathbf{P}$

Proposition

$\mathbf{P} = \mathbf{co\text{-}P}$.

P, NP, co-NP

co-P: complement of P. Language $\mathbf{X}$ is in **co-P** iff $\overline{\mathbf{X}} \in \mathbf{P}$

Proposition

$\mathbf{P} = \mathbf{co\text{-}P}$.

Proposition

$\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co\text{-}NP}$.

P, NP, co-NP

co-P: complement of P. Language $\mathbf{X}$ is in **co-P** iff $\overline{\mathbf{X}} \in \mathbf{P}$

Proposition

$\mathbf{P} = \mathbf{co\text{-}P}$.

Proposition

$\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co\text{-}NP}$.

Saw that $\mathbf{P} \subseteq \mathbf{NP}$. Same proof shows $\mathbf{P} \subseteq \mathbf{co\text{-}NP}$.

P, NP, and co-NP

Open Problems:

1. Does **NP = co-NP**?

P, NP, and co-NP

Open Problems:

1. Does **NP = co-NP**?

   Consensus opinion: No.

P, NP, and co-NP

Open Problems:

1. Does **NP = co-NP**?

   Consensus opinion: No.

2. Is **P = NP ∩ co-NP**?

P, NP, and co-NP

Open Problems:

1. Does **NP = co-NP**?
   Consensus opinion: No.
2. Is **P = NP ∩ co-NP**?
   No real consensus.

P, NP, and co-NP

Proposition

If **P** = **NP** *then* **NP** = **co-NP**.

P, NP, and co-NP

Proposition

If **P** = **NP** *then* **NP** = **co-NP**.

Proof.

P = **co-P**

If **P** = **NP** then **co-NP** = **co-P** = **P**. □

P, NP, and co-NP

We know that:

Proposition

If **P** $=$ **NP** *then* **NP** $=$ **co-NP**.

As such:

(A) If **P** $\neq$ **NP** then **NP** $\neq$ **co-NP**.

(B) If **NP** $\neq$ **co-NP** then **P** $\neq$ **NP**.

(C) If **NP** $=$ **co-NP** then **P** $\neq$ **NP**.

(D) If **NP** $\neq$ **co-NP** then **P** $=$ **NP**.

(E) None of the above.

P, NP, and co-NP

Which means that...

Corollary

If **NP** $\neq$ **co-NP** *then* **P** $\neq$ **NP**.

P, NP, and co-NP

Which means that...

Corollary

If **NP** $\neq$ **co-NP** *then* **P** $\neq$ **NP**.

Importance of corollary: try to prove **P** $\neq$ **NP** by proving that **NP** $\neq$ **co-NP**.

NP ∩ co-NP

Complexity Class NP ∩ co-NP

Problems in this class have

1. Efficient certifiers for yes-instances
2. Efficient disqualifiers for no-instances

Problems have a good characterization property, since for both yes and no instances we have short efficiently checkable proofs.

NP $\cap$ co-NP: Example

Example

Bipartite Matching: Given bipartite graph $G = (U \cup V, E)$, does G have a perfect matching?

Bipartite Matching $\in$ **NP** $\cap$ **co-NP**

NP $\cap$ co-NP: Example

Example

Bipartite Matching: Given bipartite graph $G = (U \cup V, E)$, does G have a perfect matching?

Bipartite Matching $\in$ **NP** $\cap$ **co-NP**

1. If G is a yes-instance, then proof is just the perfect matching.

Example (More interesting...)

Factoring $\in$ **NP** $\cap$ **co-NP**, and we do not know if it is in **P**!

NP $\cap$ co-NP: Example

Example

Bipartite Matching: Given bipartite graph $G = (U \cup V, E)$, does G have a perfect matching?

Bipartite Matching $\in$ **NP** $\cap$ **co-NP**

1. If G is a yes-instance, then proof is just the perfect matching.
2. If G is a no-instance, then by Hall's Theorem, there is a subset of vertices $A \subsetneq U$ such that $|N(A)| < |A|$.

Example (More interesting...)

Factoring $\in$ **NP** $\cap$ **co-NP**, and we do not know if it is in **P**!

Good Characterization $\stackrel{?}{=}$ Efficient Solution

1. Bipartite Matching has a polynomial time algorithm
2. Do all problems in **NP** $\cap$ **co-NP** have polynomial time algorithms? That is, is **P** $=$ **NP** $\cap$ **co-NP**?

Good Characterization $\stackrel{?}{=}$ Efficient Solution

1. Bipartite Matching has a polynomial time algorithm
2. Do all problems in **NP** $\cap$ **co-NP** have polynomial time algorithms? That is, is **P** $=$ **NP** $\cap$ **co-NP**?
   Problems in **NP** $\cap$ **co-NP** have been proved to be in **P** many years later
   1. Linear programming (Khachiyan 1979)
      1. Duality easily shows that it is in **NP** $\cap$ **co-NP**
   2. Primality Testing (Agarwal-Kayal-Saxena 2002)
      1. Easy to see that **PRIME** is in **co-NP** (why?)
      2. **PRIME** is in **NP** - not easy to show! (Vaughan Pratt 1975)

$\mathbf{P} \stackrel{?}{=} \mathbf{NP} \cap \text{co-}\mathbf{NP}$ (contd)

1. Some problems in $\mathbf{NP} \cap \mathbf{co\text{-}NP}$ still cannot be proved to have polynomial time algorithms
   1. Parity Games.
   2. Other more specialized problems.

co-NP Completeness

Definition

A problem **X** is said to be co-NP-Complete (**co-NPC**) if

1. $\mathbf{X} \in$ **co-NP**
2. (Hardness) For any $\mathbf{Y} \in$ **co-NP**, $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$

co-NP Completeness

Definition

A problem **X** is said to be co-NP-Complete (**co-NPC**) if

1. $\mathbf{X} \in$ **co-NP**
2. (Hardness) For any $\mathbf{Y} \in$ **co-NP**, $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$

co-NP-Complete problems are the hardest problems in **co-NP**.

co-NP Completeness

Definition

A problem **X** is said to be co-NP-Complete (**co-NPC**) if

1. $\mathbf{X} \in$ **co-NP**
2. (Hardness) For any $\mathbf{Y} \in$ **co-NP**, $\mathbf{Y} \leq_{\mathbf{P}} \mathbf{X}$

co-NP-Complete problems are the hardest problems in **co-NP**.

Lemma

X *is* **co-NPC** *if and only if* $\overline{\mathbf{X}}$ *is* **NP-Complete**.

Proof left as an exercise.

coNP-Complete?

Every **co-NP**-Complete problem has an exponential time algorithm; that is, an algorithm that runs in $2^{p(n)}$ time on an input of size n for some fixed polynomial $p(\cdot)$.

(A) True.

(B) False.

(C) True only if $P = NP$.

(D) True only if $P \neq NP$.

P, NP and co-NP

Possible scenarios:

1. **P** $=$ **NP**. Then **P** $=$ **NP** $=$ **co-NP**.
2. **NP** $=$ **co-NP** and **P** $\neq$ **NP** (and hence also **P** $\neq$ **co-NP**).
3. **NP** $\neq$ **co-NP**. Then **P** $\neq$ **NP** and also **P** $\neq$ **co-NP**.

Most people believe that the last scenario is the likely one.

P, NP and co-NP

Possible scenarios:

1. **P** $=$ **NP**. Then **P** $=$ **NP** $=$ **co-NP**.
2. **NP** $=$ **co-NP** and **P** $\neq$ **NP** (and hence also **P** $\neq$ **co-NP**).
3. **NP** $\neq$ **co-NP**. Then **P** $\neq$ **NP** and also **P** $\neq$ **co-NP**.

Most people believe that the last scenario is the likely one.

Question: Suppose **P** $\neq$ **NP**. Is every problem that is in **NP** $\setminus$ **P** is also **NP-Complete**?

P, NP and co-NP

Possible scenarios:

1. **P** $=$ **NP**. Then **P** $=$ **NP** $=$ **co-NP**.
2. **NP** $=$ **co-NP** and **P** $\neq$ **NP** (and hence also **P** $\neq$ **co-NP**).
3. **NP** $\neq$ **co-NP**. Then **P** $\neq$ **NP** and also **P** $\neq$ **co-NP**.

Most people believe that the last scenario is the likely one.

Question: Suppose **P** $\neq$ **NP**. Is every problem that is in **NP** $\setminus$ **P** is also **NP-Complete**?

Theorem (Ladner)

If **P** $\neq$ **NP** *then there is a problem/language* **X** $\in$ **NP** $\setminus$ **P** *such that* **X** *is not* **NP-Complete**.

Karp vs Turing Reduction and NP vs co-NP

Question: Why restrict to Karp reductions for NP-Completeness?

Lemma

If $\mathbf{X} \in$ **co-NP** *and* $\mathbf{Y}$ *is* **NP-Complete** *then* $\mathbf{X} \leq_{\mathbf{P}} \mathbf{Y}$ *under Turing reduction.*

Thus, Turing reductions cannot distinguish **NP** and **co-NP**.

Back to Decision versus Search

1. Recall, decision problems are those with yes/no answers, while search problems require an explicit solution for a yes instance

Example

1. Satisfiability
   1. Decision: Is the formula φ satisfiable?
   2. Search: Find assignment that satisfies φ
2. Graph coloring
   1. Decision: Is graph $\mathbf{G}$ 3-colorable?
   2. Search: Find a 3-coloring of the vertices of $\mathbf{G}$

Decision "reduces to" Search

1. Efficient algorithm for search implies efficient algorithm for decision.
2. If decision problem is difficult then search problem is also difficult.
3. Can an efficient algorithm for decision imply an efficient algorithm for search?
   Yes, for all the problems we have seen. In fact for all **NP-Complete** Problems.

Self Reduction

Definition

A problem is said to be **self reducible** if the search problem reduces (by Turing reduction) in polynomial time to decision problem. In other words, there is an algorithm to solve the search problem that has polynomially many steps, where each step is either

1. A conventional computational step, or
2. a call to subroutine solving the decision problem.

Max 3SAT...

Problem: MAX 3SAT

Instance: **F**: 3CNF Formula with **n** variables and **m** clauses.
Question: Find assignment satisfying as many clauses of **F** as possible.

(A) **MAX 3SAT** is **NP-Complete**.
(B) **MAX 3SAT** is **NP-Hard**.
(C) **MAX 3SAT** is **P**.
(D) **MAX 3SAT** is not a decision problem.

Back to SAT

Proposition

SAT *is self reducible.*

In other words, there is a polynomial time algorithm to find the satisfying assignment if one can periodically check if some formula is satisfiable.

Search Algorithm for SAT

given a Decision Algorithm for SAT

Input: SAT formula φ with n variables $x_1, x_2, \ldots, x_n$.

1. set $x_1 = 0$ in φ and get new formula φ_1. check if φ_1 is satisfiable using decision algorithm. if φ_1 is satisfiable, recursively find assignment to $x_2, x_3, \ldots, x_n$ that satisfy φ_1 and output $x_1 = 0$ along with the assignment to $x_2, \ldots, x_n$.
2. if φ_1 is not satisfiable then set $x_1 = 1$ in φ to get formula φ_2. if φ_2 is satisfiable , recursively find assignment to $x_2, x_3, \ldots, x_n$ that satisfy φ_2 and output $x_1 = 1$ along with the assignment to $x_2, \ldots, x_n$.
3. if φ_1 and φ_2 are both not satisfiable then φ is not satisfiable.

Algorithm runs in polynomial time if the decision algorithm for SAT runs in polynomial time. At most $2n$ calls to decision algorithm.

Self-Reduction for **NP-Complete** Problems

Theorem

Every **NP-Complete** *problem/language* **L** *is self-reducible.*

Proof is not hard but requires understanding of proof of Cook-Levin theorem.

Note that proof is only for complete languages, not for all languages in **NP**. Otherwise **Factoring** would be in polynomial time and we would not rely on it for our current security protocols.

Easy and instructive to prove self-reducibility for specific **NP-Complete** problems such as **Independent Set**, **Vertex Cover**, **Hamiltonian Cycle**, etc.
See discussion section problems.

Notes

Notes

Notes

Notes

Introduction to Linear Programming

Lecture 25

May 1, 2014

Easy or not easy?

Clicker question

Let $x_1, \ldots, x_n \in \{0, 1\}$ be boolean variables. You are given m constraints of the form:

$$1 + x_i + x_j - x_k \geq 1.$$

That is, each variable might have $+1$ or -1 as a coefficient, and each inequality has three variables, and a constant additive term. Deciding if such a problem has a feasible solution is

(A) **NP-Complete**.

(B) **NP-Hard**.

(C) **P**.

(D) Not a well defined question.

(E) Doable in polynomial time if Riemann's hypothesis is true.

Part I

Introduction to Linear Programming

Maximum Flow in Network

Maximum Flow in Network

Need to compute values $\mathbf{f_{s1}, f_{s2}, \ldots f_{25}, \ldots f_{5t}, f_{6t}}$ such that

Maximum Flow in Network

Need to compute values $f_{s1}, f_{s2}, \ldots f_{25}, \ldots f_{5t}, f_{6t}$ such that

$$
\begin{array}{lll}
f_{s1} \leq 15 & f_{s2} \leq 5 & f_{s3} \leq 10 \\
f_{14} \leq 30 & f_{21} \leq 4 & f_{25} \leq 8 \\
f_{32} \leq 4 & f_{35} \leq 15 & f_{36} \leq 9 \\
f_{42} \leq 6 & f_{4t} \leq 10 & f_{54} \leq 15 \\
f_{5t} \leq 10 & f_{65} \leq 15 & f_{6t} \leq 10
\end{array}
$$

Maximum Flow in Network

Need to compute values $f_{s1}, f_{s2}, \ldots f_{25}, \ldots f_{5t}, f_{6t}$ such that

$$\begin{array}{lll} f_{s1} \leq 15 & f_{s2} \leq 5 & f_{s3} \leq 10 \\ f_{14} \leq 30 & f_{21} \leq 4 & f_{25} \leq 8 \\ f_{32} \leq 4 & f_{35} \leq 15 & f_{36} \leq 9 \\ f_{42} \leq 6 & f_{4t} \leq 10 & f_{54} \leq 15 \\ f_{5t} \leq 10 & f_{65} \leq 15 & f_{6t} \leq 10 \end{array}$$

and

$$\begin{array}{lll} f_{s1} + f_{21} = f_{14} & f_{s2} + f_{32} = f_{21} + f_{25} & f_{s3} = f_{32} + f_{35} + f_{36} \\ f_{14} + f_{54} = f_{42} + f_{4t} & f_{25} + f_{35} + f_{65} = f_{54} + f_{5t} & f_{36} = f_{65} + f_{6t} \end{array}$$

Maximum Flow in Network

Need to compute values $f_{s1}, f_{s2}, \ldots f_{25}, \ldots f_{5t}, f_{6t}$ such that

$$\begin{array}{lll} f_{s1} \leq 15 & f_{s2} \leq 5 & f_{s3} \leq 10 \\ f_{14} \leq 30 & f_{21} \leq 4 & f_{25} \leq 8 \\ f_{32} \leq 4 & f_{35} \leq 15 & f_{36} \leq 9 \\ f_{42} \leq 6 & f_{4t} \leq 10 & f_{54} \leq 15 \\ f_{5t} \leq 10 & f_{65} \leq 15 & f_{6t} \leq 10 \end{array}$$

and

$$\begin{array}{lll} f_{s1} + f_{21} = f_{14} & f_{s2} + f_{32} = f_{21} + f_{25} & f_{s3} = f_{32} + f_{35} + f_{36} \\ f_{14} + f_{54} = f_{42} + f_{4t} & f_{25} + f_{35} + f_{65} = f_{54} + f_{5t} & f_{36} = f_{65} + f_{6t} \\ f_{s1} \geq 0 \quad f_{s2} \geq 0 & f_{s3} \geq 0 \quad \cdots \quad f_{4t} \geq 0 & f_{5t} \geq 0 \quad f_{6t} \geq 0 \end{array}$$

Maximum Flow in Network

Need to compute values $f_{s1}, f_{s2}, \ldots f_{25}, \ldots f_{5t}, f_{6t}$ such that

$$\begin{array}{lll} f_{s1} \leq 15 & f_{s2} \leq 5 & f_{s3} \leq 10 \\ f_{14} \leq 30 & f_{21} \leq 4 & f_{25} \leq 8 \\ f_{32} \leq 4 & f_{35} \leq 15 & f_{36} \leq 9 \\ f_{42} \leq 6 & f_{4t} \leq 10 & f_{54} \leq 15 \\ f_{5t} \leq 10 & f_{65} \leq 15 & f_{6t} \leq 10 \end{array}$$

and

$$\begin{array}{lll} f_{s1} + f_{21} = f_{14} & f_{s2} + f_{32} = f_{21} + f_{25} & f_{s3} = f_{32} + f_{35} + f_{36} \\ f_{14} + f_{54} = f_{42} + f_{4t} & f_{25} + f_{35} + f_{65} = f_{54} + f_{5t} & f_{36} = f_{65} + f_{6t} \\ f_{s1} \geq 0 \quad f_{s2} \geq 0 & f_{s3} \geq 0 \quad \cdots \quad f_{4t} \geq 0 & f_{5t} \geq 0 \quad f_{6t} \geq 0 \end{array}$$

and $f_{s1} + f_{s2} + f_{s3}$ is maximized.

Maximum Flow as a Linear Program

For a general flow network $\mathbf{G = (V, E)}$ with capacities $\mathbf{c_e}$ on edge $\mathbf{e \in E}$, we have variables $\mathbf{f_e}$ indicating flow on edge $\mathbf{e}$

$$\begin{aligned}
\text{Maximize} \quad & \sum_{\mathbf{e} \text{ out of } \mathbf{s}} \mathbf{f_e} \\
\text{subject to} \quad & \mathbf{f_e} \leq \mathbf{c_e} && \text{for each } \mathbf{e} \in \mathbf{E} \\
& \sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f_e} - \sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f_e} = \mathbf{0} && \forall \mathbf{v} \in \mathbf{V} \setminus \{\mathbf{s}, \mathbf{t}\} \\
& \mathbf{f_e} \geq \mathbf{0} && \text{for each } \mathbf{e} \in \mathbf{E}.
\end{aligned}$$

Maximum Flow as a Linear Program

For a general flow network $\mathbf{G = (V, E)}$ with capacities $\mathbf{c_e}$ on edge $\mathbf{e \in E}$, we have variables $\mathbf{f_e}$ indicating flow on edge $\mathbf{e}$

$$\begin{aligned}
\text{Maximize} \quad & \sum_{\mathbf{e} \text{ out of } \mathbf{s}} \mathbf{f_e} \\
\text{subject to} \quad & \mathbf{f_e} \leq \mathbf{c_e} && \text{for each } \mathbf{e} \in \mathbf{E} \\
& \sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f_e} - \sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f_e} = \mathbf{0} && \forall \mathbf{v} \in \mathbf{V} \setminus \{\mathbf{s}, \mathbf{t}\} \\
& \mathbf{f_e} \geq \mathbf{0} && \text{for each } \mathbf{e} \in \mathbf{E}.
\end{aligned}$$

Number of variables: $\mathbf{m}$, one for each edge.
Number of constraints: $\mathbf{m + n - 2 + m}$.

Minimum Cost Flow with Lower Bounds

... as a Linear Program

For a general flow network $\mathbf{G = (V, E)}$ with capacities $\mathbf{c_e}$, lower bounds $\ell_\mathbf{e}$, and costs $\mathbf{w_e}$, we have variables $\mathbf{f_e}$ indicating flow on edge $\mathbf{e}$. Suppose we want a min-cost flow of value at least $\mathbf{v}$.

$$\text{Minimize} \sum_{\mathbf{e} \in \mathbf{E}} \mathbf{w_e f_e}$$

$$\text{subject to} \sum_{\mathbf{e} \text{ out of } \mathbf{s}} \mathbf{f_e} \geq \mathbf{v}$$

$$\mathbf{f_e} \leq \mathbf{c_e} \qquad \mathbf{f_e} \geq \ell_\mathbf{e} \qquad \text{for each } \mathbf{e} \in \mathbf{E}$$

$$\sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f_e} - \sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f_e} = \mathbf{0} \qquad \text{for each } \mathbf{v} \in \mathbf{V} - \{\mathbf{s}, \mathbf{t}\}$$

$$\mathbf{f_e} \geq \mathbf{0} \qquad \text{for each } \mathbf{e} \in \mathbf{E}.$$

Minimum Cost Flow with Lower Bounds

... as a Linear Program

For a general flow network $\mathbf{G = (V, E)}$ with capacities $\mathbf{c_e}$, lower bounds $\ell_\mathbf{e}$, and costs $\mathbf{w_e}$, we have variables $\mathbf{f_e}$ indicating flow on edge $\mathbf{e}$. Suppose we want a min-cost flow of value at least $\mathbf{v}$.

$$\text{Minimize} \sum_{\mathbf{e} \in \mathbf{E}} \mathbf{w_e f_e}$$

$$\text{subject to} \sum_{\mathbf{e} \text{ out of } \mathbf{s}} \mathbf{f_e} \geq \mathbf{v}$$

$$\mathbf{f_e} \leq \mathbf{c_e} \qquad \mathbf{f_e} \geq \ell_\mathbf{e} \qquad \text{for each } \mathbf{e} \in \mathbf{E}$$

$$\sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f_e} - \sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f_e} = \mathbf{0} \qquad \text{for each } \mathbf{v} \in \mathbf{V} - \{\mathbf{s}, \mathbf{t}\}$$

$$\mathbf{f_e} \geq \mathbf{0} \qquad \text{for each } \mathbf{e} \in \mathbf{E}.$$

Number of variables: $\mathbf{m}$, one for each edge

Number of constraints: $\mathbf{1 + m + m + n - 2 + m = 3m + n - 1}$.

Linear Programs

Problem

Find a vector $\mathbf{x} \in \mathbb{R}^d$ that

$$\begin{array}{lll} \text{maximize/minimize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i & \text{for } i = 1 \ldots p \\ & \sum_{j=1}^{d} a_{ij} x_j = b_i & \text{for } i = p+1 \ldots q \\ & \sum_{j=1}^{d} a_{ij} x_j \geq b_i & \text{for } i = q+1 \ldots n \end{array}$$

Linear Programs

Problem

Find a vector $\mathbf{x} \in \mathbb{R}^d$ that

$$\begin{array}{lll} \text{maximize/minimize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i & \text{for } i = 1 \ldots p \\ & \sum_{j=1}^{d} a_{ij} x_j = b_i & \text{for } i = p+1 \ldots q \\ & \sum_{j=1}^{d} a_{ij} x_j \geq b_i & \text{for } i = q+1 \ldots n \end{array}$$

Input is matrix $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{n \times d}$, column vector $\mathbf{b} = (b_i) \in \mathbb{R}^n$, and row vector $\mathbf{c} = (c_j) \in \mathbb{R}^d$

Canonical Form of Linear Programs

Canonical Form

A linear program is in canonical form if it has the following structure

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i & \text{for } i = 1 \ldots n \\ & x_j \geq 0 & \text{for } j = 1 \ldots d \end{array}$$

Canonical Form of Linear Programs

Canonical Form

A linear program is in canonical form if it has the following structure

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i & \text{for } i = 1 \ldots n \\ & x_j \geq 0 & \text{for } j = 1 \ldots d \end{array}$$

Conversion to Canonical Form

1. Replace each variable x_j by $x_j^+ - x_j^-$ and inequalities $x_j^+ \geq 0$ and $x_j^- \geq 0$
2. Replace $\sum_j a_{ij} x_j = b_i$ by $\sum_j a_{ij} x_j \leq b_i$ and $-\sum_j a_{ij} x_j \leq -b_i$
3. Replace $\sum_j a_{ij} x_j \geq b_i$ by $-\sum_j a_{ij} x_j \leq -b_i$

Matrix Representation of Linear Programs

A linear program in canonical form can be written as

$$\begin{array}{ll} \text{maximize} & \mathbf{c} \cdot \mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

where $\mathbf{A} = (\mathbf{a_{ij}}) \in \mathbb{R}^{\mathbf{n} \times \mathbf{d}}$, column vector $\mathbf{b} = (\mathbf{b_i}) \in \mathbb{R}^{\mathbf{n}}$, row vector $\mathbf{c} = (\mathbf{c_j}) \in \mathbb{R}^{\mathbf{d}}$, and column vector $\mathbf{x} = (\mathbf{x_j}) \in \mathbb{R}^{\mathbf{d}}$

1. Number of variable is $\mathbf{d}$
2. Number of constraints is $\mathbf{n} + \mathbf{d}$

Other Standard Forms for Linear Programs

$$\begin{array}{ll} \text{maximize} & \mathbf{c} \cdot \mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

$$\begin{array}{ll} \text{minimize} & \mathbf{c} \cdot \mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

Linear Programming: A History

1. First formalized applied to problems in economics by Leonid Kantorovich in the 1930s
   1. However, work was ignored behind the Iron Curtain and unknown in the West
2. Rediscovered by Tjalling Koopmans in the 1940s, along with applications to economics
3. First algorithm (Simplex) to solve linear programs by George Dantzig in 1947
4. Kantorovich and Koopmans receive Nobel Prize for economics in 1975 ; Dantzig, however, was ignored
   1. Koopmans contemplated refusing the Nobel Prize to protest Dantzig's exclusion, but Kantorovich saw it as a vindication for using mathematics in economics, which had been written off as "a means for apologists of capitalism"

A Factory Example

Problem

Suppose a factory produces two products **1** and **2**. Each requires three resources **A, B, C**.

1. Producing one unit of Product **1** requires one unit each of resources **A** and **C**.
2. One unit of Product **2** requires one unit of resource **B** and one units of resource **C**.
3. We have 200 units of **A**, 300 units of **B**, and 400 units of **C**.
4. Product **1** can be sold for **$1** and product **2** for **$6**.

How many units of product **1** and product **2** should the factory manufacture to maximize profit?

A Factory Example

Problem

Suppose a factory produces two products **1** and **2**. Each requires three resources **A, B, C**.

1. Producing one unit of Product **1** requires one unit each of resources **A** and **C**.
2. One unit of Product **2** requires one unit of resource **B** and one units of resource **C**.
3. We have 200 units of **A**, 300 units of **B**, and 400 units of **C**.
4. Product **1** can be sold for **$1** and product **2** for **$6**.

How many units of product **1** and product **2** should the factory manufacture to maximize profit?

Solution:

A Factory Example

Problem

Suppose a factory produces two products **1** and **2**. Each requires three resources **A, B, C**.

1. Producing one unit of Product **1** requires one unit each of resources **A** and **C**.
2. One unit of Product **2** requires one unit of resource **B** and one units of resource **C**.
3. We have 200 units of **A**, 300 units of **B**, and 400 units of **C**.
4. Product **1** can be sold for **$1** and product **2** for **$6**.

How many units of product **1** and product **2** should the factory manufacture to maximize profit?

Solution: Formulate as a linear program.

A Factory Example

Problem

Suppose a factory produces two products **1** and **2**. Each requires three resources **A, B, C**.

1. Producing unit of product **1**: Req. one unit of **A**, **C**.
2. Producing unit of product **2**: Requ. one unit of **B, C**.
3. Have **A**: 200, **B**: 300 , and **C**: 400.
4. Price of **1**: **\$1**, and **2**: **\$6**.

How many units of **1** and **2** to manufacture to max profit?

A Factory Example

Problem

Suppose a factory produces two products **1** and **2**. Each requires three resources **A, B, C**.

1. Producing unit of product **1**: Req. one unit of **A**, **C**.
2. Producing unit of product **2**: Requ. one unit of **B, C**.
3. Have **A**: 200, **B**: 300 , and **C**: 400.
4. Price of **1**: **\$1**, and **2**: **\$6**.

How many units of **1** and **2** to manufacture to max profit?

$$
\begin{array}{lll}
\max & x_1 + 6x_2 & \\
\text{s.t.} & x_1 \leq 200 & \text{(A)} \\
 & x_2 \leq 300 & \text{(B)} \\
 & x_1 + x_2 \leq 400 & \text{(C)} \\
 & x_1 \geq 0 & \\
 & x_2 \geq 0 &
\end{array}
$$

Linear Programming Formulation

Let us produce x_1 units of product 1 and x_2 units of product 2. Our profit can be computed by solving

$$\begin{array}{lll} \text{maximize} & & x_1 + 6x_2 \\ \text{subject to} & x_1 \leq 200 \quad x_2 \leq 300 & x_1 + x_2 \leq 400 \\ & & x_1, x_2 \geq 0 \end{array}$$

Linear Programming Formulation

Let us produce x_1 units of product 1 and x_2 units of product 2. Our profit can be computed by solving

$$\begin{array}{lll} \text{maximize} & & x_1 + 6x_2 \\ \text{subject to} & x_1 \leq 200 \quad x_2 \leq 300 & x_1 + x_2 \leq 400 \\ & & x_1, x_2 \geq 0 \end{array}$$

What is the solution?

Solving the Factory Example

1. Feasible values of x_1 and x_2 are shaded region.

Solving the Factory Example

1. Feasible values of x_1 and x_2 are shaded region.
2. Objective function is a direction — the line represents all points with same value of the function

Solving the Factory Example

1. Feasible values of x_1 and x_2 are shaded region.
2. Objective function is a direction — the line represents all points with same value of the function; moving the line until it just leaves the feasible region, gives optimal values.

Linear Programming in 2-d

1. Each constraint a half plane
2. Feasible region is intersection of finitely many half planes — it forms a polygon
3. For a fixed value of objective function, we get a line. Parallel lines correspond to different values for objective function.
4. Optimum achieved when objective function line just leaves the feasible region

An Example in 3-d

$$\max \quad x_1 + 6x_2 + 13x_3$$

$$\begin{aligned} x_1 &\leq 200 && (1) \\ x_2 &\leq 300 && (2) \\ x_1 + x_2 + x_3 &\leq 400 && (3) \\ x_2 + 3x_3 &\leq 600 && (4) \\ x_1 &\geq 0 && (5) \\ x_2 &\geq 0 && (6) \\ x_3 &\geq 0 && (7) \end{aligned}$$

Figure from Dasgupta etal book.

Factory Example: Alternate View

Original Problem

Recall we have,

$$\begin{array}{ll} \text{maximize} & x_1 + 6x_2 \\ \text{subject to} & x_1 \leq 200 \quad x_2 \leq 300 \quad x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0 \end{array}$$

Factory Example: Alternate View

Original Problem

Recall we have,

$$\begin{array}{ll} \text{maximize} & x_1 + 6x_2 \\ \text{subject to} & x_1 \leq 200 \quad x_2 \leq 300 \quad x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0 \end{array}$$

Transformation

Consider new variable x_1' and x_2', such that $x_1 = -6x_1' + x_2'$ and $x_2 = x_1' + 6x_2'$. Then in terms of the new variables we have

$$\begin{array}{ll} \text{maximize} & 37x_2' \\ \text{subject to} & -6x_1' + x_2' \leq 200 \quad x_1' + 6x_2' \leq 300 \quad -5x_1' + 7x_2 \leq 40 \\ & -6x_1' + x_2' \geq 0 \qquad x_1' + 6x_2' \geq 0 \end{array}$$

Transformed Picture

Feasible region rotated, and optimal value at the highest point on polygon

Observations about the Transformation

Observations

1. Linear program can always be transformed to get a linear program where the optimal value is achieved at the point in the feasible region with highest **y**-coordinate
2. Optimum value attained at a vertex of the polygon
3. Since feasible region is convex, every local optimum is a global optimum

A Simple Algorithm in 2-d

1. optimum solution is at a vertex of the feasible region
2. a vertex is defined by the intersection of two lines (constraints)

A Simple Algorithm in 2-d

1. optimum solution is at a vertex of the feasible region
2. a vertex is defined by the intersection of two lines (constraints)

Algorithm:

1. find all intersections between the $\mathbf{n}$ lines — $\mathbf{n^2}$ points
2. for each intersection point $\mathbf{p = (p_1, p_2)}$
   1. check if $\mathbf{p}$ is in feasible region (how?)
   2. if $\mathbf{p}$ is feasible evaluate objective function at $\mathbf{p}$: $\mathbf{val(p) = c_1 p_1 + c_2 p_2}$
3. Output the feasible point with the largest value

A Simple Algorithm in 2-d

1. optimum solution is at a vertex of the feasible region
2. a vertex is defined by the intersection of two lines (constraints)

Algorithm:

1. find all intersections between the $\mathbf{n}$ lines — $\mathbf{n^2}$ points
2. for each intersection point $\mathbf{p = (p_1, p_2)}$
   1. check if $\mathbf{p}$ is in feasible region (how?)
   2. if $\mathbf{p}$ is feasible evaluate objective function at $\mathbf{p}$: $\mathbf{val(p) = c_1 p_1 + c_2 p_2}$
3. Output the feasible point with the largest value

Running time: $\mathbf{O(n^3)}$.

Simple Algorithm in General Case

Real problem: **d**-dimensions

Simple Algorithm in General Case

Real problem: $\mathbf{d}$-dimensions

1. optimum solution is at a vertex of the feasible region
2. a vertex is defined by the intersection of $\mathbf{d}$ hyperplanes
3. number of vertices can be $\mathbf{\Omega(n^d)}$

Running time: $\mathbf{O(n^{d+1})}$ which is not polynomial since problem size is at least $\mathbf{nd}$. Also not practical.

How do we find the intersection point of $\mathbf{d}$ hyperplanes in $\mathbb{R}^{\mathbf{d}}$?

Simple Algorithm in General Case

Real problem: $\mathbf{d}$-dimensions

1. optimum solution is at a vertex of the feasible region
2. a vertex is defined by the intersection of $\mathbf{d}$ hyperplanes
3. number of vertices can be $\Omega(n^d)$

Running time: $O(n^{d+1})$ which is not polynomial since problem size is at least nd. Also not practical.

How do we find the intersection point of $\mathbf{d}$ hyperplanes in $\mathbb{R}^d$? Using Gaussian elimination to solve $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a $\mathbf{d} \times \mathbf{d}$ matrix and $\mathbf{b}$ is a $\mathbf{d} \times 1$ matrix.

Simplex in 2-d

Simplex Algorithm

1. Start from some vertex of the feasible polygon
2. Compare value of objective function at current vertex with the value at "neighboring" vertices of polygon
3. If neighboring vertex improves objective function, move to this vertex, and repeat step 2
4. If current vertex is local optimum, then stop.

Linear Programming in **d**-dimensions

1. Each linear constraint defines a **halfspace**.
2. Feasible region, which is an intersection of halfspaces, is a convex **polyhedron**.
3. Optimal value attained at a vertex of the polyhedron.
4. Every local optimum is a global optimum.

Simplex in Higher Dimensions

1. Start at a vertex of the polytope.
2. Compare value of objective function at each of the **d** "neighbors".
3. Move to neighbor that improves objective function, and repeat step 2.
4. If local optimum, then stop

Simplex in Higher Dimensions

1. Start at a vertex of the polytope.
2. Compare value of objective function at each of the **d** "neighbors".
3. Move to neighbor that improves objective function, and repeat step 2.
4. If local optimum, then stop

Simplex is a greedy local-improvement algorithm! Works because a local optimum is also a global optimum — convexity of polyhedra.

Solving Linear Programming in Practice

1. Naïve implementation of Simplex algorithm can be very inefficient
   1. Choosing which neighbor to move to can significantly affect running time
   2. Very efficient Simplex-based algorithms exist
   3. Simplex algorithm takes exponential time in the worst case but works extremely well in practice with many improvements over the years
2. Non Simplex based methods like interior point methods work well for large problems.

Polynomial time Algorithm for Linear Programming

Major open problem for many years: is there a polynomial time algorithm for linear programming?

Polynomial time Algorithm for Linear Programming

Major open problem for many years: is there a polynomial time algorithm for linear programming?
Leonid Khachiyan in 1979 gave the first polynomial time algorithm using the Ellipsoid method.

1. major theoretical advance
2. highly impractical algorithm, not used at all in practice
3. routinely used in theoretical proofs.

Polynomial time Algorithm for Linear Programming

Major open problem for many years: is there a polynomial time algorithm for linear programming?
Leonid Khachiyan in 1979 gave the first polynomial time algorithm using the Ellipsoid method.

1. major theoretical advance
2. highly impractical algorithm, not used at all in practice
3. routinely used in theoretical proofs.

Narendra Karmarkar in 1984 developed another polynomial time algorithm, the interior point method.

1. very practical for some large problems and beats simplex
2. also revolutionized theory of interior point methods

Polynomial time Algorithm for Linear Programming

Major open problem for many years: is there a polynomial time algorithm for linear programming?
Leonid Khachiyan in 1979 gave the first polynomial time algorithm using the Ellipsoid method.

1. major theoretical advance
2. highly impractical algorithm, not used at all in practice
3. routinely used in theoretical proofs.

Narendra Karmarkar in 1984 developed another polynomial time algorithm, the interior point method.

1. very practical for some large problems and beats simplex
2. also revolutionized theory of interior point methods

Following interior point method success, Simplex has been improved enormously and is the method of choice.

Degeneracy

1. The linear program could be infeasible: No points satisfy the constraints.
2. The linear program could be unbounded: Polygon unbounded in the direction of the objective function.

Infeasibility: Example

$$\begin{array}{ll} \text{maximize} & x_1 + 6x_2 \\ \text{subject to} & x_1 \leq 2 \quad x_2 \leq 1 \quad x_1 + x_2 \geq 4 \\ & x_1, x_2 \geq 0 \end{array}$$

Infeasibility has to do only with constraints.

Unboundedness: Example

$$\begin{aligned} \text{maximize} \quad & x_2 \\ x_1 + x_2 &\geq 2 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Unboundedness depends on both constraints and the objective

Feasible Solutions and Lower Bounds

Consider the program

$$\begin{array}{lrrrl} \text{maximize} & 4x_1 + & x_2 + & 3x_3 & \\ \text{subject to} & x_1 + & 4x_2 & & \leq 1 \\ & 3x_1 - & x_2 + & x_3 & \leq 3 \\ & & & x_1, x_2, x_3 & \geq 0 \end{array}$$

Feasible Solutions and Lower Bounds

Consider the program

$$\begin{array}{lrrrl} \text{maximize} & 4x_1 + & x_2 + & 3x_3 & \\ \text{subject to} & x_1 + & 4x_2 & & \leq 1 \\ & 3x_1 - & x_2 + & x_3 & \leq 3 \\ & & & x_1, x_2, x_3 & \geq 0 \end{array}$$

1. $(1, 0, 0)$ satisfies all the constraints and gives value 4 for the objective function.

Feasible Solutions and Lower Bounds

Consider the program

$$\begin{array}{lrrrl} \text{maximize} & 4x_1 + & x_2 + & 3x_3 & \\ \text{subject to} & x_1 + & 4x_2 & & \leq 1 \\ & 3x_1 - & x_2 + & x_3 & \leq 3 \\ & & & x_1, x_2, x_3 & \geq 0 \end{array}$$

1. $(1, 0, 0)$ satisfies all the constraints and gives value 4 for the objective function.
2. Thus, optimal value σ^* is at least 4.

Feasible Solutions and Lower Bounds

Consider the program

$$\begin{array}{lrrrl} \text{maximize} & 4x_1 + & x_2 + & 3x_3 & \\ \text{subject to} & x_1 + & 4x_2 & & \leq 1 \\ & 3x_1 - & x_2 + & x_3 & \leq 3 \\ & & & x_1, x_2, x_3 & \geq 0 \end{array}$$

1. $(1, 0, 0)$ satisfies all the constraints and gives value 4 for the objective function.
2. Thus, optimal value σ^* is at least 4.
3. $(0, 0, 3)$ also feasible, and gives a better bound of 9.

Feasible Solutions and Lower Bounds

Consider the program

$$\begin{array}{lrrrl} \text{maximize} & 4x_1 + & x_2 + & 3x_3 & \\ \text{subject to} & x_1 + & 4x_2 & & \leq 1 \\ & 3x_1 - & x_2 + & x_3 & \leq 3 \\ & & & x_1, x_2, x_3 & \geq 0 \end{array}$$

1. $(1, 0, 0)$ satisfies all the constraints and gives value 4 for the objective function.
2. Thus, optimal value σ^* is at least 4.
3. $(0, 0, 3)$ also feasible, and gives a better bound of 9.
4. How good is 9 when compared with σ^*?

Obtaining Upper Bounds

1. Let us multiply the first constraint by 2 and the second by 3 and add the result

$$\begin{array}{rrrrl} 2(& x_1+ & 4x_2 & &) \leq 2(1) \\ +3(& 3x_1- & x_2+ & x_3 &) \leq 3(3) \\ \hline & 11x_1+ & 5x_2+ & 3x_3 & \leq 11 \end{array}$$

2. Since x_is are positive, compared to objective function $4x_1 + x_2 + 3x_3$, we have

$$4x_1 + x_2 + 3x_3 \leq 11x_1 + 5x_2 + 3x_3 \leq 11$$

3. Thus, 11 is an upper bound on the optimum value!

Generalizing . . .

1. Multiply first equation by y_1 and second by y_2 (both y_1, y_2 being positive) and add

$$\begin{array}{rrrrl} y_1(& x_1+ & 4x_2 & &) \leq y_1(1) \\ +y_2(& 3x_1- & x_2+ & x_3 &) \leq y_2(3) \\ \hline & (y_1+3y_2)x_1+ & (4y_1-y_2)x_2+ & (y_2)x_3 & \leq y_1+3y_2 \end{array}$$

2. $y_1 + 3y_2$ is an upper bound, provided coefficients of x_i are as large as in the objective function, i.e.,

$$y_1 + 3y_2 \geq 4 \quad 4y_1 - y_2 \geq 1 \quad y_2 \geq 3$$

3. The best upper bound is when $y_1 + 3y_2$ is minimized!

Dual LP: Example

Thus, the optimum value of program

$$\begin{aligned} \text{maximize} \quad & 4x_1 + x_2 + 3x_3 \\ \text{subject to} \quad & x_1 + 4x_2 \leq 1 \\ & 3x_1 - x_2 + x_3 \leq 3 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

is upper bounded by the optimal value of the program

$$\begin{aligned} \text{minimize} \quad & y_1 + 3y_2 \\ \text{subject to} \quad & y_1 + 3y_2 \geq 4 \\ & 4y_1 - y_2 \geq 1 \\ & y_2 \geq 3 \\ & y_1, y_2 \geq 0 \end{aligned}$$

Dual Linear Program

Given a linear program Π in canonical form

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i & i = 1, 2, \ldots n \\ & x_j \geq 0 & j = 1, 2, \ldots d \end{array}$$

the dual $\mathrm{Dual}(\Pi)$ is given by

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} b_i y_i & \\ \text{subject to} & \sum_{i=1}^{n} y_i a_{ij} \geq c_j & j = 1, 2, \ldots d \\ & y_i \geq 0 & i = 1, 2, \ldots n \end{array}$$

Dual Linear Program

Given a linear program Π in canonical form

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{d} c_j x_j & \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i & i = 1, 2, \ldots n \\ & x_j \geq 0 & j = 1, 2, \ldots d \end{array}$$

the dual $\mathrm{Dual}(\Pi)$ is given by

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{n} b_i y_i & \\ \text{subject to} & \sum_{i=1}^{n} y_i a_{ij} \geq c_j & j = 1, 2, \ldots d \\ & y_i \geq 0 & i = 1, 2, \ldots n \end{array}$$

Proposition

$\mathrm{Dual}(\mathrm{Dual}(\Pi))$ *is equivalent to* Π

Duality Theorem

Theorem (Weak Duality)

If $\mathbf{x}$ *is a feasible solution to* $\mathbf{\Pi}$ *and* $\mathbf{y}$ *is a feasible solution to* $\mathbf{Dual(\Pi)}$ *then* $\mathbf{c} \cdot \mathbf{x} \leq \mathbf{y} \cdot \mathbf{b}$.

Duality Theorem

Theorem (Weak Duality)

If $\mathbf{x}$ *is a feasible solution to* $\mathbf{\Pi}$ *and* $\mathbf{y}$ *is a feasible solution to* $\mathbf{Dual(\Pi)}$ *then* $\mathbf{c} \cdot \mathbf{x} \leq \mathbf{y} \cdot \mathbf{b}$.

Theorem (Strong Duality)

If $\mathbf{x}^*$ *is an optimal solution to* $\mathbf{\Pi}$ *and* $\mathbf{y}^*$ *is an optimal solution to* $\mathbf{Dual(\Pi)}$ *then* $\mathbf{c} \cdot \mathbf{x}^* = \mathbf{y}^* \cdot \mathbf{b}$.

Many applications! Maxflow-Mincut theorem can be deduced from duality.

Maximum Flow Revisited

For a general flow network $\mathbf{G = (V, E)}$ with capacities $\mathbf{c_e}$ on edge $\mathbf{e \in E}$, we have variables $\mathbf{f_e}$ indicating flow on edge $\mathbf{e}$

Maximize $\sum_{\mathbf{e} \text{ out of } \mathbf{s}} \mathbf{f_e}$	subject to
$\mathbf{f_e \leq c_e}$	for each $\mathbf{e \in E}$
$\sum_{\mathbf{e} \text{ out of } \mathbf{v}} \mathbf{f_e} - \sum_{\mathbf{e} \text{ into } \mathbf{v}} \mathbf{f_e = 0}$	for each $\mathbf{v \in V - \{s, t\}}$
$\mathbf{f_e \geq 0}$	for each $\mathbf{e \in E}$

Number of variables: $\mathbf{m}$, one for each edge
Number of constraints: $\mathbf{m + n - 2 + m}$

Maximum flow can be reduced to Linear Programming.

Integer Linear Programming

Problem

Find a vector $\mathbf{x} \in \mathbf{Z}^{\mathbf{d}}$ (integer values) that

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^{d} c_j x_j \\ \text{subject to} & \sum_{j=1}^{d} a_{ij} x_j \leq b_i \quad \text{for } i = 1 \ldots n \end{array}$$

Input is matrix $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{n \times d}$, column vector $\mathbf{b} = (b_i) \in \mathbb{R}^{n}$, and row vector $\mathbf{c} = (c_j) \in \mathbb{R}^{d}$

Factory Example

$$\begin{array}{ll} \text{maximize} & x_1 + 6x_2 \\ \text{subject to} & x_1 \leq 200 \quad x_2 \leq 300 \quad x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0 \end{array}$$

Suppose we want x_1, x_2 to be integer valued.

Factory Example Figure

1. Feasible values of x_1 and x_2 are integer points in shaded region
2. Optimization function is a line; moving the line until it just leaves the final integer point in feasible region, gives optimal values

Factory Example Figure

1. Feasible values of x_1 and x_2 are integer points in shaded region
2. Optimization function is a line; moving the line until it just leaves the final integer point in feasible region, gives optimal values

Factory Example Figure

1. Feasible values of x_1 and x_2 are integer points in shaded region
2. Optimization function is a line; moving the line until it just leaves the final integer point in feasible region, gives optimal values

Integer Programming

Can model many difficult discrete optimization problems as integer programs!

Therefore integer programming is a hard problem. NP-hard.

Can relax integer program to linear program and *approximate*.

Practice: integer programs are solved by a variety of methods

1. branch and bound
2. branch and cut
3. adding cutting planes
4. linear programming plays a fundamental role

Linear Programs with Integer Vertices

Suppose we know that for a linear program *all* vertices have integer coordinates.

Linear Programs with Integer Vertices

Suppose we know that for a linear program *all* vertices have integer coordinates.
Then solving linear program is same as solving integer program. We know how to solve linear programs efficiently (polynomial time) and hence we get an integer solution for free!

Linear Programs with Integer Vertices

Suppose we know that for a linear program *all* vertices have integer coordinates.
Then solving linear program is same as solving integer program. We know how to solve linear programs efficiently (polynomial time) and hence we get an integer solution for free!

Luck or *Structure*:

1. Linear program for flows with integer capacities have integer vertices
2. Linear program for matchings in bipartite graphs have integer vertices
3. A complicated linear program for matchings in general graphs have integer vertices.

All of above problems can hence be solved efficiently.

Linear Programs with Integer Vertices

Meta Theorem: A combinatorial optimization problem can be solved efficiently if and only if there is a linear program for problem with integer vertices.

Consequence of the Ellipsoid method for solving linear programming.

In a sense linear programming and other geometric generalizations such as convex programming are the most general problems that we can solve efficiently.

Summary

1. Linear Programming is a useful and powerful (modeling) problem.
2. Can be solved in polynomial time. Practical solvers available commercially as well as in open source. Whether there is a strongly polynomial time algorithm is a major open problem.
3. Geometry and linear algebra are important to understand the structure of LP and in algorithm design. Vertex solutions imply that LPs have poly-sized optimum solutions. This implies that LP is in **NP**.
4. Duality is a critical tool in the theory of linear programming. Duality implies the Linear Programming is in **co-NP**. Do you see why?
5. Integer Programming in **NP-Complete**. LP-based techniques critical in heuristically solving integer programs.

Notes

Notes

Notes

Notes

Heuristics, Closing Thoughts

Lecture 26

May 1, 2014

Part I

Heuristics

Coping with Intractability

Question: Many useful/important problems are **NP-Hard** or worse. How does one cope with them?

Coping with Intractability

Question: Many useful/important problems are **NP-Hard** or worse. How does one cope with them?

Some general things that people do.

1. Consider special cases of the problem which may be tractable.
2. Run inefficient algorithms (for example exponential time algorithms for **NP-Hard** problems) augmented with (very) clever heuristics
   1. stop algorithm when time/resources run out
   2. use massive computational power
3. Exploit properties of instances that arise in practice which may be much easier. Give up on hard instances, which is OK.
4. Settle for sub-optimal (aka approximate) solutions, especially for optimization problems

NP and EXP

EXP: all problems that have an exponential time algorithm.

Proposition

NP $\subseteq$ **EXP**.

NP and EXP

EXP: all problems that have an exponential time algorithm.

Proposition

$\mathbf{NP} \subseteq \mathbf{EXP}$.

Proof.

Let $\mathbf{X} \in \mathbf{NP}$ with certifier $\mathbf{C}$. To prove $\mathbf{X} \in \mathbf{EXP}$, here is an algorithm for $\mathbf{X}$. Given input $\mathbf{s}$,

1. For every $\mathbf{t}$, with $|\mathbf{t}| \leq \mathbf{p}(|\mathbf{s}|)$ run $\mathbf{C}(\mathbf{s}, \mathbf{t})$; answer "yes" if any one of these calls returns "yes", otherwise say "no". □

Every problem in **NP** has a brute-force "try all possibilities" algorithm that runs in exponential time.

Examples

1. **SAT**: try all possible truth assignment to variables.
2. **Independent set**: try all possible subsets of vertices.
3. **Vertex cover**: try all possible subsets of vertices.

Improving brute-force via intelligent backtracking

1. Backtrack search: enumeration with bells and whistles to "heuristically" cut down search space.
2. Works quite well in practice for several problems, especially for small enough problem sizes.

Backtrack Search Algorithm for SAT

Input: CNF Formula φ on $\mathbf{n}$ variables $\mathbf{x_1, \ldots, x_n}$ and $\mathbf{m}$ clauses
Output: Is φ satisfiable or not.

1. Pick a variable $\mathbf{x_i}$
2. φ' is CNF formula obtained by setting $\mathbf{x_i = 0}$ and simplifying
3. Run a simple (heuristic) check on φ': returns "yes", "no" or "not sure"
   1. If "not sure" recursively solve φ'
   2. If φ' is satisfiable, return "yes"
4. φ'' is CNF formula obtained by setting $\mathbf{x_i = 1}$
5. Run simple check on φ'': returns "yes", "no" or "not sure"
   1. If "not sure" recursively solve φ''
   2. If φ'' is satisfiable, return "yes"
6. Return "no"

Certain part of the search space is pruned.

Example

Figure : Backtrack search. Formula is not satisfiable.

Figure taken from Dasgupta etal book.

Backtrack Search Algorithm for SAT

How do we pick the order of variables?

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

1. pick variable that occurs in most clauses first
2. pick variable that appears in most size 2 clauses first
3. ...

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

1. pick variable that occurs in most clauses first
2. pick variable that appears in most size 2 clauses first
3. ...

What are quick tests for Satisfiability?

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

1. pick variable that occurs in most clauses first
2. pick variable that appears in most size 2 clauses first
3. ...

What are quick tests for Satisfiability?
Depends on known special cases and heuristics. Examples.

1. Obvious test: return "no" if empty clause, "yes" if no clauses left and otherwise "not sure"
2. Run obvious test and in addition if all clauses are of size **2** then run 2-SAT polynomial time algorithm
3. ...

Branch-and-Bound

Backtracking for optimization problems

Intelligent backtracking can be used also for optimization problems. Consider a minimization problem.

Notation: for instance **I**, **opt(I)** is optimum value on **I**.

$\mathbf{P_0}$ initial instance of given problem.

1. Keep track of the best solution value **B** found so far. Initialize **B** to be crude upper bound on **opt(I)**.
2. Let **P** be a subproblem at some stage of exploration.
3. If **P** is a complete solution, update **B**.
4. Else use a lower bounding heuristic to quickly/efficiently find a lower bound **b** on **opt(P)**.
   1. If $\mathbf{b} \geq \mathbf{B}$ then prune **P**
   2. Else explore **P** further by breaking it into subproblems and recurse on them.
5. Output best solution found.

Example: Vertex Cover

Given $\mathbf{G = (V, E)}$, find a minimum sized vertex cover in $\mathbf{G}$.

1. Initialize $\mathbf{B = n - 1}$.
2. Pick a vertex $\mathbf{u}$. Branch on $\mathbf{u}$: either choose $\mathbf{u}$ or discard it.
3. Let $\mathbf{b_1}$ be a lower bound on $\mathbf{G_1 = G - u}$.
4. If $\mathbf{1 + b_1 < B}$, recursively explore $\mathbf{G_1}$
5. Let $\mathbf{b_2}$ be a lower bound on $\mathbf{G_2 = G - u - N(u)}$ where $\mathbf{N(u)}$ is the set of neighbors of $\mathbf{u}$.
6. If $\mathbf{|N(u)| + b_2 < B}$, recursively explore $\mathbf{G_2}$
7. Output $\mathbf{B}$.

Example: Vertex Cover

Given $\mathbf{G = (V, E)}$, find a minimum sized vertex cover in $\mathbf{G}$.

1. Initialize $\mathbf{B = n - 1}$.
2. Pick a vertex $\mathbf{u}$. Branch on $\mathbf{u}$: either choose $\mathbf{u}$ or discard it.
3. Let $\mathbf{b_1}$ be a lower bound on $\mathbf{G_1 = G - u}$.
4. If $\mathbf{1 + b_1 < B}$, recursively explore $\mathbf{G_1}$
5. Let $\mathbf{b_2}$ be a lower bound on $\mathbf{G_2 = G - u - N(u)}$ where $\mathbf{N(u)}$ is the set of neighbors of $\mathbf{u}$.
6. If $\mathbf{|N(u)| + b_2 < B}$, recursively explore $\mathbf{G_2}$
7. Output $\mathbf{B}$.

How do we compute a lower bound?
One possibility: solve an LP relaxation.

Local Search

Local Search: a simple and broadly applicable heuristic method

1. Start with some arbitrary solution $\mathbf{s}$
2. Let $\mathbf{N(s)}$ be solutions in the "neighborhood" of $\mathbf{s}$ obtained from $\mathbf{s}$ via "local" moves/changes
3. If there is a solution $\mathbf{s'} \in \mathbf{N(s)}$ that is better than $\mathbf{s}$, move to $\mathbf{s'}$ and continue search with $\mathbf{s'}$
4. Else, stop search and output $\mathbf{s}$.

Local Search

Main ingredients in local search:

1. Initial solution.
2. Definition of neighborhood of a solution.
3. Efficient algorithm to find a good solution in the neighborhood.

Example: TSP

TSP: Given a complete graph $\mathbf{G = (V, E)}$ with $\mathbf{c_{ij}}$ denoting cost of edge $\mathbf{(i, j)}$, compute a Hamiltonian cycle/tour of minimum edge cost.

Example: TSP

TSP: Given a complete graph $G = (V, E)$ with c_{ij} denoting cost of edge (i, j), compute a Hamiltonian cycle/tour of minimum edge cost.

2-change local search:

1. Start with an arbitrary tour s_0
2. For a solution s define s' to be a neighbor if s' can be obtained from s by replacing two edges in s with two other edges.
3. For a solution s at most $O(n^2)$ neighbors and one can try all of them to find an improvement.

TSP: 2-change example

TSP: 2-change example

TSP: 2-change example

Figure below shows a bad local optimum for **2**-change heuristic...

TSP: 2-change example

Figure below shows a bad local optimum for **2**-change heuristic...

TSP: 2-change example

Figure below shows a bad local optimum for **2**-change heuristic...

TSP: 3-change example

3-change local search: swap **3** edges out.

Neighborhood of **s** has now increased to a size of $\mathbf{\Omega(n^3)}$
Can define **k**-change heuristic where **k** edges are swapped out.
Increases neighborhood size and makes each local improvement step less efficient.

TSP: 3-change example

3-change local search: swap **3** edges out.

Neighborhood of **s** has now increased to a size of $\mathbf{\Omega(n^3)}$
Can define **k**-change heuristic where **k** edges are swapped out.
Increases neighborhood size and makes each local improvement step less efficient.

Local Search Variants

Local search terminates with a local optimum which may be far from a global optimum. Many variants to improve plain local search.

1. Randomization and restarts. Initial solution may strongly influence the quality of the final solution. Try many random initial solutions.
2. Simulated annealing is a general method where one allows the algorithm to move to worse solutions with some probability. At the beginning this is done more aggressively and then slowly the algorithm converges to plain local search. Controlled by a parameter called "temperature".
3. Tabu search. Store already visited solutions and do not visit them again (they are "taboo").

Heuristics

Several other heuristics used in practice.

1. Heuristics for solving integer linear programs such as cutting planes, branch-and-cut etc are quite effective.
2. Heuristics to solve SAT (SAT-solvers) have gained prominence in recent years
3. Genetic algorithms
4. ...

Heuristics design is somewhat ad hoc and depends heavily on the problem and the instances that are of interest. Rigorous analysis is sometimes possible.

Approximation algorithms

Consider the following *optimization* problems:

1. **Max Knapsack**: Given knapsack of capacity $\mathbf{W}$, $\mathbf{n}$ items each with a value and weight, pack the knapsack with the most profitable subset of items whose weight does not exceed the knapsack capacity.
2. **Min Vertex Cover:** given a graph $\mathbf{G = (V, E)}$ find the minimum cardinality vertex cover.
3. **Min Set Cover:** given Set Cover instance, find the smallest number of sets that cover all elements in the universe.
4. **Max Independent Set:** given graph $\mathbf{G = (V, E)}$ find maximum independent set.
5. **Min Traveling Salesman Tour:** given a directed graph $\mathbf{G}$ with edge costs, find minimum length/cost Hamiltonian cycle in $\mathbf{G}$.

Solving one in polynomial time implies solving all the others.

Approximation algorithms

However, the problems behave very differently if one wants to solve them *approximately*.

Informal definition: An approximation algorithm for an optimization problem is an efficient (polynomial-time) algorithm that *guarantees* for every instance a solution of some given quality when compared to an optimal solution.

Some known approximation results

1. **Knapsack**: For every fixed $\epsilon > 0$ there is a polynomial time algorithm that guarantees a solution of quality $(1 - \epsilon)$ times the best solution for the given instance. Hence can get a 0.99-approximation efficiently.
2. **Min Vertex Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most 2 times the cost of an optimum solution.
3. **Min Set Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most $(\ln n + 1)$ times the cost of an optimal solution.
4. **Max Independent Set**: Unless $P = NP$, for any fixed $\epsilon > 0$, no polynomial time algorithm can give a $n^{1-\epsilon}$ relative approximation . Here n is number of vertices in the graph.
5. **Min TSP**: No polynomial factor relative approximation possible.

Approximation algorithms

1. Although **NP-Complete** problems are all equivalent with respect to polynomial-time solvability they behave quite differently under approximation (in both theory and practice).
2. Approximation is a useful lens to examine **NP-Complete** problems more closely.
3. Approximation also useful for problems that we can solve efficiently:
    1. We may have other constraints such a space (streaming problems) or time (need linear time or less for very large problems)
    2. Data may be uncertain (online and stochastic problems).

Example: Vertex Cover

```
Greedy(G):
        Initialize S to be ∅
        While there are edges in G do
            Let v be a vertex with maximum degree
            S ← S ∪ {v}
            G ← G − v
        endWhile
        Output S
```

Example: Vertex Cover

```
Greedy(G):
        Initialize S to be ∅
        While there are edges in G do
            Let v be a vertex with maximum degree
            S ← S ∪ {v}
            G ← G − v
        endWhile
        Output S
```

Theorem

$|S| \leq (\ln n + 1)OPT$ *where* OPT *is the value of an optimum set. Here* n *is number of nodes in* G*.*

Theorem

There is an infinite family of graphs where the solution S *output by Greedy is* $\Omega(\ln n)OPT$*.*

Example: Vertex Cover

```
MatchingHeuristic(G):
        Find a maximal matching M in G
        S is the set of end points of edges in M
        Output S
```

Example: Vertex Cover

```
MatchingHeuristic(G):
        Find a maximal matching M in G
        S is the set of end points of edges in M
        Output S
```

Lemma

$\mathbf{OPT} \geq |\mathbf{M}|$.

Lemma

$\mathbf{S}$ *is a feasible vertex cover.*

Analysis: $|\mathbf{S}| = 2|\mathbf{M}| \leq 2\mathbf{OPT}$. Algorithm is a 2-approximation.

Vertex Cover: LP Relaxation based approach

Write (weighted) vertex cover problem as an integer linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \in \{0, 1\} & \text{for each } v \in V \end{array}$$

Vertex Cover: LP Relaxation based approach

Write (weighted) vertex cover problem as an integer linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \in \{0, 1\} & \text{for each } v \in V \end{array}$$

Relax integer program to a linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Vertex Cover: LP Relaxation based approach

Write (weighted) vertex cover problem as an integer linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \in \{0, 1\} & \text{for each } v \in V \end{array}$$

Relax integer program to a linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Can solve linear program in polynomial time.
Let x^* be an optimum solution to the linear program.

Lemma

$\mathbf{OPT} \geq \sum_v w_v x_v^*$.

Vertex Cover: Rounding fractional solution

LP Relaxation

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Let x^* be an optimum solution to the linear program.
Rounding: $S = \{v \mid x^*_v \geq 1/2\}$. Output S.

Vertex Cover: Rounding fractional solution

LP Relaxation

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Let x^* be an optimum solution to the linear program.
Rounding: $S = \{v \mid x^*_v \geq 1/2\}$. Output S.

Lemma

S *is a feasible vertex cover for the given graph.*

Vertex Cover: Rounding fractional solution

LP Relaxation

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Let x^* be an optimum solution to the linear program.
Rounding: $S = \{v \mid x^*_v \geq 1/2\}$. Output S.

Lemma

S *is a feasible vertex cover for the given graph.*

Lemma

$w(S) \leq 2 \sum_v w_v x^*_v \leq 2OPT$.

Set Cover and Vertex Cover

Theorem

Greedy gives $(\ln n + 1)$*-approximation for Set Cover where* n *is number of elements.*

Theorem

Unless $P = NP$ *no* $(\ln n + \epsilon)$*-approximation for Set Cover.*

2-approximation is best known for Vertex Cover.

Theorem

Unless $P = NP$ *no* 1.36*-approximation for Vertex Cover.*

Conjecture: Unless $P = NP$ no $(2 - \epsilon)$-approximation for Vertex Cover for any fixed $\epsilon > 0$.

Part II

Closing Thoughts

Topics I wish I had time for

1. Data structures
2. Lower bounds on sorting and related problems
3. More on the use of randomization in algorithms
4. Algorithms for string processing, compression, coding
5. More on heuristics and applications

Theoretical Computer Science

1. Algorithms: find efficient ways to solve particular problems or broad category of problems
2. Computational Complexity: understand nature of computation — classification of problems into classes (**P**, **NP**, **co-NP**) and their relationships, limits of computation.
3. Logic, Languages and Formal Methods

Form the foundations for computer "science"

The Computational Lens

The Algorithm: Idiom of Modern Science by Bernard Chazelle
http://www.cs.princeton.edu/chazelle/pubs/algorithm.html

Computation has gained ground as *fundamental* artifact in mathematics and science.

1. nature of proofs, **P** vs **NP**, complexity, ...
2. quantum computation and information
3. computational biology and the biological processes, ...

Standard question in math and sciences: Is there a *solution/algorithm*?

The Computational Lens

The Algorithm: Idiom of Modern Science by Bernard Chazelle
http://www.cs.princeton.edu/chazelle/pubs/algorithm.html

Computation has gained ground as *fundamental* artifact in mathematics and science.

1. nature of proofs, **P** vs **NP**, complexity, ...
2. quantum computation and information
3. computational biology and the biological processes, ...

Standard question in math and sciences: Is there a *solution/algorithm*?
New: Is there an *efficient* solution/algorithm?

Related theory courses

1. Graduate algorithms (every year, offered this fall)
2. Computational complexity (every year, offered next spring)
3. Randomized algorithms (every other year)
4. Approximation algorithms (every other year)
5. Advanced data structures (every once in a while)
6. Cryptography and related topics (almost every year, offered this fall)
7. Algorithmic game theory, combinatorial optimization, computational geometry, logic and formal methods, coding theory, information theory, graph theory, combinatorics, ...

Technological and sociological changes

- Scale of data due to the web, digitization of knowledge, people
- Mobility, cloud, and distributed data
- Social and economic networks/activity in online/virtual fora
- Privacy and security
- Possibility of quantum computation: new model and algorithms
- Biology/DNA as digital information/computation
- End of Moore's law?
- Increasing integration of sensing and computational devices

Impact on Algorithms

- Scale of data implies efficiency of algorithms very important
  - Sub-linear time and data streaming (approximation) algorithms via sampling, estimation, ...
  - Various forms of parallel algorithms (Map-Reduce etc)
- Large data availability implies statistical machine learning can be very effective so many new applications
- Algorithmic aspects of strategic behavior of people: algorithmic game theory, sponsored search auctions, influence propagation in social networks, ...,
- Distributed computing, cyber-physical computing
- Interdisciplinary work: quatum information and computation, computational biology and chemistry

And...

Questions?

And...

Questions?

Final Exam: Friday May 9th, 7–10pm

Thanks!

Notes

Notes

Notes

Notes