Algorithms Design I

Prologue

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Instructor

Instructor and Teaching Assistants

Guoqiang LI

- Homepage: https://basics.sjtu.edu.cn/%7Eliguoqiang
- Canvas: https://oc.sjtu.edu.cn/courses/69730
- Email: li.g (AT) outlook (DOT) com
- Office: Rm. 1212, Building of Software
- Phone: 3420-4167

TA:

- Shuhan FENG: count_von (AT) sjtu (DOT) edu (DOT) cn
- Lianyi WU: edithwuly (AT) 163 (DOT) com

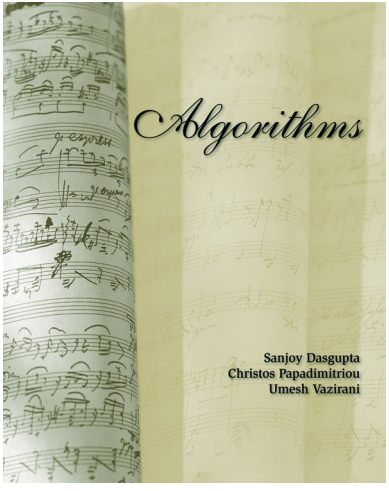
Office hour: Wed. 14:00-17:00 @ SEIEE 3-325

Reference Book

Textbook

Algorithms

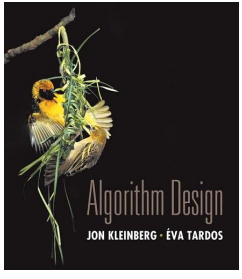
- Sanjoy Dasgupta
- San Diego Christos Papadimitriou
- Umesh Vazirani
- McGraw-Hill, 2007.



Reference Book

Algorithm Design

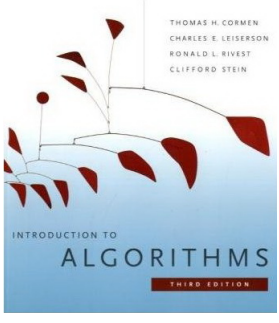
- Jon Kleinberg, Éva Tardos
- Addison-Wesley, 2005.



Reference Book

Introduction to Algorithms

- Thomas H. Cormen
- Charles E. Leiserson
- Ronald L. Rivest
- Clifford Stein
- The MIT Press (3rd edition), 2009.



Scoring Policy

0% Attendees.

40% Homework.

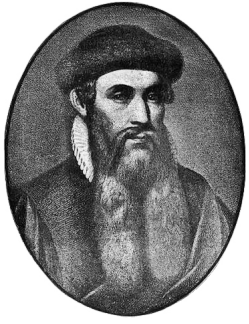
- Eight assignments.
- Each one is 5pts.
- Work out individually.
- Each assignment will be evaluated by A, B, C, D, F (Excellent(5), Good(5), Fair(4), Delay(3), Fail(0))

60% Final exam.

Any Questions?

Two Things Change the World

Johann Gutenberg



Johann Gutenberg (1398 - 1468)

In 1448 in the German city of Mainz a goldsmith named Johann Gutenberg discovered a way to print books by putting together movable metallic pieces.

Sheng BI

Bì Shēng (972-1051)

Bì Shēng was a Chinese artisan, engineer, and inventor of the world's first movable type technology, with printing being one of the Four Great Inventions of Ancient China.

Because of the typography, literacy spread, the Dark Ages ended, the human intellect was liberated, science and technology triumphed, the Industrial Revolution happened.

Many historians say we owe all this to typography.

Others insist that the key development was not typography, but algorithms.

Gutenberg would write the number 1448 as $MCDXLVIII$.

How to add two Roman numerals? What is

$$MCDXLVIII + DCCCXII$$

The decimal system was invented in India around AD 600. Using only 10 symbols, even very large numbers were written down compactly, and arithmetic is done efficiently by elementary steps.

Al Khwarizmi (780 - 850)

In the 12th century, Latin translations of his work on the Indian numerals, introduced the decimal system to the Western world. (Source: Wikipedia)

Algorithms

Al Khwarizmi laid out the basic methods for

- adding,
- multiplying,
- dividing numbers,
- extracting square roots,
- calculating digits of π.

These procedures were precise, unambiguous, mechanical, efficient, correct.

They were algorithms, a term coined to honor the wise man after the decimal system was finally adopted in Europe, many centuries later.

Chongzhi ZU

Chongzhi ZU (429 – 500)

A Chinese astronomer, inventor, mathematician, politician, and writer during the Liu Song and Southern Qi dynasties. He was most notable for calculating π as between 3.1415926 and 3.1415927, a record in precision which would not be surpassed for nearly 900 years.

What Is An Algorithm

What Is An Algorithm

A step by step procedure for solving a problem or accomplishing some end.

An abstract recipe, prescribing a process which may be carried out by a human, a computer or by other means.

Any well-defined computational procedure that makes some value, or set of values, as input and produces some value, of set of values, as output. An algorithm is thus a finite sequence of computational steps that transform the input into the output.

What Is An Algorithm

An algorithm is a procedure that consists of

- a finite set of instructions which,
- given an input from some set of possible inputs,
- enables us to obtain an output through a systematic execution of the instructions
- that terminates in a finite number of steps.

A program is

- an implementation of an algorithm, or algorithms.
- A program does not necessarily terminate.

Fibonacci Algorithm

Leonardo Fibonacci (1170 - 1250)

Fibonacci helped the spread of the decimal system in Europe, primarily through the publication in the early 13th century of his Book of Calculation, the Liber Abaci. (Source: Wikipedia)

Fibonacci Sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Formally,

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

Q: What is F_{100} or F_{200}?

An Exponential Algorithm


```
FIBO1(n)
a nature number n;

if n = 0 then return(0);
if n = 1 then return(1);
return(FIBO1(n − 1) +FIBO1(n − 2));
```

Three Questions about An Algorithm

1. Is it correct?
2. How much time does it take, as a function of n?
3. Can we do better?

The first question is trivial, as this algorithm is precisely Fibonacci's definition of F_n

How Much Time

Let $T(n)$ be the number of computer steps needed to compute `FIBO1`(n)

For $n \leq 1$,

$$T(n) \leq 2$$

For $n \geq 1$,

$$T(n) = T(n-1) + T(n-2) + 3$$

It is easy to shown, for all $n \in \mathbb{N}$,

$$T(n) \geq F_n$$

It is exponential to n.

Why Exponential Is Bad?

$$T(200) \geq F_{200} \geq 2^{138} \approx 2.56 \times 10^{42}$$

In 2010, the fastest computer in the world is the Tianhe-1A system at the National Supercomputer Center in Tianjin.

Its speed is

$$2.57 \times 10^{15}$$

steps per second.

Thus to compute F_{200} Tianhe-1A needs roughly

$$10^{27} \text{ seconds} \geq 10^{22} \text{ years.}$$

In 2022, the fastest is Frontier, 1.102×10^{18} per second.

Moore's Law:

Computer speeds have been doubling roughly every 18 months.

The running time of `FIBO1` is proportional to

$$2^{0.694n} \approx 1.6^n$$

Thus, it takes 1.6 times longer to compute F_{n+1} than F_n.

So if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}, and so on ...

Just one more number every year!

Such is the curse of exponential time.

Three Questions

1. Is it correct?
2. How much time does it take, as a function of n?
3. Can we do better?

Now we know `FIB1`(n) is correct and inefficient, so can we do better?

An Polynomial Algorithm


```
FIBO2(n)
a nature number n;
if n = 0 then return(0);
create an array f[0...n];
f[0] = 0; f[1] = 1;
for i = 2 to n do
    f[i] = f[i − 1] + f[i − 2];
end
return(f[n]);
```

The correctness of `FIBO2` is trivial.

How long does it take?

The inner loop consists of a single computer step and is executed $n - 1$ times. Therefore the number of computer steps used by `FIBO2` is linear in n.

A More Careful Analysis

We count the number of basic computer steps executed by each algorithm and regard these basic steps as taking a constant amount of time.

It is reasonable to treat addition as a single computer step if small numbers are being added, e.g., 32-bit numbers.

The n-th Fibonacci number is about $0.694n$ bits long, and this can far exceed 32 as n grows.

Arithmetic operations on arbitrarily large numbers cannot possibly be performed in a single, constant-time step.

The addition of two n-bit numbers takes time roughly proportional to n (next lecture).

`FIBO1`, which performs about F_n additions, uses a number of basic step roughly proportional to nF_n.

The number of steps taken by `FIBO2` is proportional to n^2, and still polynomial in n.

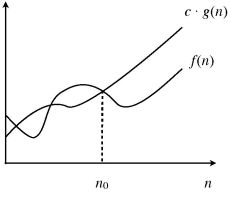
Q: Can we do better?

- Exercise 0.4

Big-O Notation

Big O notation

Upper bounds. $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.



Example

Let $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is $O(n^2)$.
- $f(n)$ is neither $O(n)$ nor $O(n \log n)$.

Typical usage. Insertion sort makes $O(n^2)$ compares to sort n elements.

Let $f(n) = 3n^2 + 17n \log_2 n + 1000$. Which of the following are true?

A $f(n)$ is $O(n^2)$.

B $f(n)$ is $O(n^3)$.

C Both A and B.

D Neither A nor B.

Big O notational abuses

One-way "equality". $O(g(n))$ is a set of functions, but computer scientists often write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.

Example

Consider $g_1(n) = 5n^3$ and $g_2(n) = 3n^2$.

- We have $g_1(n) = O(n^3)$ and $g_2(n) = O(n^3)$.
- But, do not conclude $g_1(n) = g_2(n)$.

Big O notation: properties

Reflexivity. f is $O(f)$.

Constants. If f is $O(g)$ and $c > 0$, then $c\,f$ is $O(g)$.

Products. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1 g_2)$.

Proof.

- $\exists c_1 > 0$ and $n_1 \geq 0$ such that $0 \leq f_1(n) \leq c_1 \cdot g_1(n)$ for all $n \geq n_1$.
- $\exists c_2 > 0$ and $n_2 \geq 0$ such that $0 \leq f_2(n) \leq c_2 \cdot g_2(n)$ for all $n \geq n_2$.
- Then, $0 \leq f_1(n) \cdot f_2(n) \leq c_1 \cdot c_2 \cdot g_1(n) \cdot g_2(n)$ for all $n \geq \max\{n_1, n_2\}$.

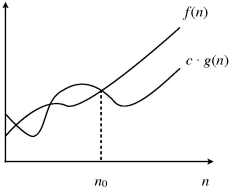
Sums. If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max\{g_1, g_2\})$.

Transitivity. If f is $O(g)$ and g is $O(h)$, then f is $O(h)$.

Ex. $f(n) = 5n^3 + 3n^2 + n + 1234$ is $O(n^3)$.

Big Ω notation

Lower bounds. $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n) \geq 0$ for all $n \geq n_0$.



Example

Let $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is both $\Omega(n^2)$ and $\Omega(n)$.
- $f(n)$ is not $\Omega(n^3)$.

Typical usage. Any compare-based sorting algorithm requires $\Omega(n \log n)$ compares in the worst case.

Quiz

Which is an equivalent definition of big Omega notation?

A $f(n)$ is $\Omega(g(n))$ iff $g(n)$ is $O(f(n))$.

B $f(n)$ is $\Omega(g(n))$ iff there exist constants $c > 0$ such that

$$f(n) \geq c \cdot g(n) \geq 0$$

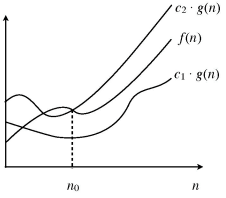
for infinitely many n.

C Both A and B.

D Neither A nor B.

Big Θ notation

Tight bounds. $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0, c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0$.



Example

Let $f(n) = 32n^2 + 17n + 1$.

- $f(n)$ is $\Theta(n^2)$.
- $f(n)$ is neither $\Theta(n^3)$ nor $\Omega(n)$.

Typical usage. Mergesort makes $\Theta(n \log n)$ compares to sort n elements.

Which is an equivalent definition of big Theta notation?

A $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is both $O(g(n))$ and $\Omega(g(n))$.

B $f(n)$ is $\Theta(g(n))$ iff $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant $0 < c < +\infty$.

C Both A and B.

D Neither A nor B.

Proposition

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ *for some constant* $0 < c < \infty$ *then* $f(n)$ *is* $\Theta(g(n))$.

Proof.

By definition of the limit, for any $\varepsilon > 0$, there exists n_0 such that

$$c - \varepsilon \leq \frac{f(n)}{g(n)} \leq c + \varepsilon$$

for all $n \geq n_0$.

Choose $\varepsilon = 1/2c > 0$.

Multiplying by $g(n)$ yields $1/2c \cdot g(n) \leq f(n) \leq 3/2c \cdot g(n)$ for all $n \geq n_0$.

Thus, $f(n)$ is $\Theta(g(n))$ by definition, with $c_1 = 1/2c$ and $c_2 = 3/2c$.

Proposition

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, *then* $f(n)$ *is* $O(g(n))$ *but not* $\Omega(g(n))$.

Proposition

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, *then* $f(n)$ *is* $\Omega(g(n))$ *but not* $O(g(n))$.

Asymptotic bounds for some common functions

Polynomials. Let $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then, $f(n)$ is $\Theta(n^d)$.

$$\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$$

Logarithms and polynomials. $\log_a n$ is $O(n^d)$ for every $a > 1$ and every $d > 0$.

$$\lim_{n \to \infty} \frac{\log_a n}{n^d} = 0$$

Exponentials and polynomials. n^d is $O(r^n)$ for every $r > 1$ and every $d > 0$.

$$\lim_{n \to \infty} \frac{n^d}{r^n} = 0$$

Asymptotic bounds for some common functions

Factorials. $n!$ is $2^{\Theta(n \log n)}$.

Stirling's formula:

$$n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$$

Big O notation with multiple variables

Upper bounds. $f(m, n)$ is $O(g(m, n))$ if there exist constants $c > 0$, $m_0 \geq 0$, and$n_0 \geq 0$ such that $f(m, n) \leq c \cdot g(m, n)$ for all $n \geq n_0$ and $m \geq m_0$.

Example

$f(m, n) = 32mn^2 + 17mn + 32n^3$.

- $f(m, n)$ is both $O(mn^2 + n^3)$ and $O(mn^3)$.
- $f(m, n)$ is neither $O(n^3)$ nor $O(mn^2)$.

Typical usage. Breadth-first search takes $O(m + n)$ time to find a shortest path from s to t in a digraph with n nodes and m edges.

Algorithms Design II

Algorithms with Numbers I

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Two Seemingly Similar Problems

Factoring: Given a number N, express it as a product of its prime factors.

Primality: Given a number N, determine whether it is a prime.

We believe that Factoring is hard and much of the electronic commerce is built on this assumption.

There are efficient algorithms for Primality, e.g., AKS test by Manindra Agrawal, Neeraj Kayal, and Nitin Saxena.

A Notable Result

The AKS primality test is a deterministic primality-proving algorithm created and published by Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, computer scientists at the Indian Institute of Technology Kanpur, on August 6, 2002, The algorithm was the first to determine whether any given number is prime or composite within polynomial time. The authors received the 2006 Gödel Prize and the 2006 Fulkerson Prize for this work.

Preliminaries

How to Represent Numbers

We are most familiar with decimal representation:

- 1024

But computers use binary representation:

- $1\underbrace{0\ldots0}_{10 \text{ times}}$

The bigger the base is, the shorter the representation is. But how much do we really gain by choosing large base?

Q: How many digits are needed to represent the number $N \geq 0$ in base b?

$$\lceil \log_b (N+1) \rceil$$

Q: How much does the size of a number change when we change bases?

$$\log_b N = \frac{\log_a N}{\log_a b}$$

In O notation, the base is irrelevant, and thus we write the size simply as $O(\log N)$

Roles of $Log\ N$

$log\ N$ is the power to which you need to raise 2 in order to obtain N.

It can also be seen as the number of times you must halve N to get down to 1. (More precisely: $\lceil log\ N \rceil$.)

It is the number of bits in the binary representation of N. (More precisely: $\lceil log\ (N+1) \rceil$.)

It is also the depth of a complete binary tree with N nodes. (More precisely: $\lfloor log\ N \rfloor$.)

It is even the sum $1 + 1/2 + 1/3 + \ldots + 1/n$, to within a constant factor.

Basics Arithmetic

Lemma

The sum of any three single-digit number is at most two digits long.

This rule holds not just in decimal but in any $b \geq 2$.

In binary, the maximum possible sum of three single-bit numbers is 3, which is a 2-bit number.

This simple rule gives us a way to add two numbers in any bases.

$$\begin{array}{cccccccc} 1 & & & 1 & 1 & 1 & & \\ & & & & & & & \\ & 1 & 1 & 0 & 1 & 0 & 1 \\ & 1 & 0 & 0 & 0 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}$$

Q: Given two binary number x and y, how long does our algorithm take to add them?

The answer expressed as a function of the size of the input: the number of bits of x and y (suppose they are n bit long).

The sum of x and y is $n + 1$ bits at most. Each individual bit of this sum gets computed in a fixed amount of time.

The total running time for the addition is of form $c_0 + c_1\, n$, where c_0 and c_1 are some constants, i.e., $O(n)$.

Addition

Q: Can we do better?

In order to add two n-bit numbers, we must read them and write down the answer, and even that requires n operations.

So the addition algorithm is optimal.

Perform Addition in One Step?

A single instruction we can add integers whose size in bits is within the word length of today's computer - 64 perhaps.

It is often useful and necessary to handle numbers much larger than this, perhaps several thousand bits long.

To study the basic algorithms encoded in the hardware of today's computers, we shall focus on the bit complexity of the algorithm, the number of elementary operations on individual bits.

Multiplication

The grade-school algorithm for multiplying two number x and y is to create an array of intermediate sums.

If x and y are both n bit, then there are n intermediate rows with length of up to $2n$ bit. (Q: why?)

$$
\begin{array}{cccccccc}
 & & & & 1 & 1 & 0 & 1 \\
 & & & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 1 & 0 & 1 \\
 & & & 1 & 1 & 0 & 1 & \\
 & & 0 & 0 & 0 & 0 & & \\
+ & 1 & 1 & 0 & 1 & & & \\
\hline
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}
$$

$$
\underbrace{O(n) + \ldots + O(n)}_{n-1}
$$

$$
O(n^2)
$$

Quiz

What is the complexity of a number times 2?

Multiplication by Al Khwarizmi

- write them next to each other.
- halve the first number by 2, dropping the .5, and double the second number.
- keep going till the first number gets down to 1.
- strike out all the rows where the first number is even.
- add up the remains in the second columns.

11	13
5	26
1	104
	143

- The left is to calculate the binary number.
- The right is to shift the row!

Multiplication á la Françis

MULTIPLY(x, y)
Two n-bit integers x and y, where $y \geq 0$;

if $y = 0$ **then** return 0;
z=MULTIPLY(x, $\lfloor y/2 \rfloor$);
if y *is even* **then**
    return $2z$;
    else return $x + 2z$;
end

Another formulation:

$$x \cdot y = \begin{cases} 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is even} \\ x + 2(x \cdot \lfloor y/2 \rfloor) & \text{if } y \text{ is odd} \end{cases}$$

Multiplication á la Françis

Q: How long does the algorithm take?

It will terminate after n recursive calls, since at each call y is halved.

At each call requires these operations:

- a division by 2 (right shift);
- a test for odd/even (looking up the last bit);
- a multiplication by 2 (left shift);
- and a possibly one addition.

A total operations are $O(n)$, The total time taken is thus $O(n^2)$.

Q: Can we do better?

- Yes!

Division

DIVIDE(x, y)
Two n-bit integers x and y, where $y \geq 1$;

if $x = 0$ **then** return$(0, 0)$;
(q, r)=DIVIDE$(\lfloor x/2 \rfloor, y)$;
$q = 2 \cdot q$, $r = 2 \cdot r$;
if x *is odd* **then** $r = r + 1$;
if $r \geq y$ **then** $r = r - y$, $q = q + 1$;
return(q, r);

Q: How long does the algorithm take?

- Exercise 1.8!

Modular Arithmetic

What Is Modular

Modular arithmetic is a system for dealing with restricted ranges of integers.

x modulo N is the remainder when x is divided by N; that is, if $x = qN + r$ with $0 \leq r < N$, then x modulo N is equal to r.

x and y are congruent modulo N if they differ by a multiple of N, i.e.

$$x \equiv y \mod N \quad \Leftrightarrow \quad N \texttt{ divides } (x - y)$$

Two Interpretations

1. It limits numbers to a predefined range $\{0, 1, \ldots, N\}$ and wraps around whenever you try to leave this range - like the hand of a clock.
2. Modular arithmetic deals with all the integers, but divides them into N equivalence classes, each of the form $\{i + k \cdot N \mid k \in \mathbb{Z}\}$ for some i between 0 and $N - 1$.

Two's Complement

Modular arithmetic is nicely illustrated in two's complement, the most common format for storing signed integers.

It uses n bits to represent numbers in the range

$$-2^{n-1}, 2^{n-1} - 1$$

and is usually described as follows:

- Positive integers, in the range 0 to $2^{n-1} - 1$, are stored in regular binary and have a leading bit of 0.
- Negative integers $-x$, with $1 \leq x \leq 2^{n-1}$, are stored by first constructing x in binary, then flipping all the bits, and finally adding 1. The leading bit in this case is 1.

Two's Complement

0	1	1	1	1	1	1	1	=	127
0	0	0	0	0	0	1	0	=	2
0	0	0	0	0	0	0	1	=	1
0	0	0	0	0	0	0	0	=	0
1	1	1	1	1	1	1	1	=	−1
1	1	1	1	1	1	1	0	=	−2
1	0	0	0	0	0	0	1	=	−127
1	0	0	0	0	0	0	0	=	−128

(from wiki)

Rules

Substitution rules: if $x \equiv x' \mod N$ and $y \equiv y' \mod N$, then

$$x + y \equiv x' + y' \mod N$$
$$xy \equiv x'y' \mod N$$

$$x + (y + z) \equiv (x + y) + z \mod N \quad \texttt{Associativity}$$
$$xy \equiv yx \mod N \quad \texttt{Commutativity}$$
$$x(y + z) \equiv xy + xz \mod N \quad \texttt{Distributivity}$$

It is legal to reduce intermediate results to their remainders modulo N at any stage.

$$2^{345} \equiv (2^5)^{69} \equiv 32^{69} \equiv 1^{69} \equiv 1 \mod 31$$

Modular Addition

Since x and y are each in the range 0 to $N-1$,their sum is between 0 and $2(N-1)$.

If the sum exceeds $N-1$, we merely need to subtract off N to bring it back into the required range.

The overall computation therefore consists of an addition, and possibly a subtraction, of numbers that never exceed $2N$.

Its running time is $O(n)$, where $n = \lceil \log N \rceil$.

The product of x and y can be as large as $(N-1)^2$, but this is still at most $2n$ bits long since

$$log\,(N-1)^2 = 2log\,(N-1) \leq 2n$$

To reduce the answer $\mod N$, we compute the remainder upon dividing it by N. ($O(n^2)$)

Multiplication thus remains a quadratic operation.

Not quite so easy!

In ordinary arithmetic there is just one tricky case - division by zero.

It turns out that in modular arithmetic there are potentially other such cases as well.

Whenever division is legal, however, it can be managed in cubic time, $O(n^3)$.

In the cyptosystem, it is necessary to compute $x^y \pmod N$ for values of x, y, and N that are several hundred bits long.

The result is some number $\bmod N$ and is therefore a few hundred bits long. However, the raw value x^y could be much, much longer.

When x and y are just 20-bit numbers, x^y is at least

$$(2^{19})^{(2^{19})} = 2^{(19)(524288)}$$

about 10 million bits long!

To make sure the numbers never grow too large, we need to perform all intermediate computations modulo N.

First idea: calculate $x^y \mod N$ by repeatedly multiplying by x modulo N.

The resulting sequence of intermediate products,

$$x \mod N \to x^2 \mod N \to x^3 \mod N \to \ldots \to x^y \mod N$$

consists of numbers that are smaller than N, and so the individual multiplications do not take too long.

But imagine if y is 500 bits long . . .

Second idea: starting with x and squaring repeatedly modulo N, we get

$$x \mod N \to x^2 \mod N \to x^4 \mod N \to x^8 \mod N \to \ldots x^{2^{\lfloor \log y \rfloor}} \mod N$$

Each takes just $O(\log^2 N)$ time to compute, and in this case there are only $\log y$ multiplications.

To determine $x^y \mod N$, multiply together an appropriate subset of these powers, those corresponding to 1's in the binary representation of y.

For instance,

$$x^{25} = x^{11001_2} = x^{10000_2} \cdot x^{1000_2} \cdot x^{1_2} = x^{16} \cdot x^8 \cdot x^1$$

Modular Exponentiation

MODEXP (x, y, N)
Two n-bit integers x and N, and an integer exponent y;

if $y = 0$ **then** return 1;
z=MODEXP $(x, \lfloor y/2 \rfloor, N)$;
if y *is even* **then**
  return $z^2 \mod N$;
  else return $x \cdot z^2 \mod N$;
end

Another formulation:

$$x^y \mod N = \begin{cases} (x^{\lfloor y/2 \rfloor})^2 \mod N & \text{if } y \text{ is even} \\ x \cdot (x^{\lfloor y/2 \rfloor})^2 \mod N & \text{if } y \text{ is odd} \end{cases}$$

MODEXP (x, y, N)
Two n-bit integers x and N, and an integer exponent y;

if $y = 0$ **then** return 1;
z=MODEXP (x, $\lfloor y/2 \rfloor$, N);
if y *is even* **then**
  return $z^2 \mod N$;
  else return $x \cdot z^2 \mod N$;
end

The algorithm will halt after at most n recursive calls, and during each call it multiplies n-bit numbers. for a total running time of $O(n^3)$

Q: Given two integers x and y, how to find their greatest common divisor ($gcd(x, y)$)?

Euclid's rule

If x and y are positive integers with $x \geq y$, then $gcd(x, y) = gcd(x \pmod y, y)$.

Proof:

It is enough to show the rule $gcd(x, y) = gcd(x - y, y)$. Result can be derived by repeatedly subtracting y from x.

Euclid's Algorithm for Greatest Common Divisor


```
EUCLID(x, y)
Two integers x and y with x ≥ y;
if y = 0 then return x;
return(EUCLID(y, x mod y));
```

Lemma

If $a \geq b \geq 0$, *then* $a \mod b < a/2$

Proof:

- if $b \leq a/2$, $a \mod b < b \leq a/2$;
- if $b > a/2$, $a \mod b = a - b < a/2$.

Euclid's Algorithm for Greatest Common Divisor

EUCLID(x, y)
Two integers x and y with $x \geq y$;

if $y = 0$ **then** return x;
return(EUCLID$(y, x \mod y)$);

Lemma

If $a \geq b \geq 0$, then $a \mod b < a/2$

This means that after any two consecutive rounds, both arguments, x and y are at the very least halved in value, i.e., the length of each decreases at least one bit.

Euclid's Algorithm for Greatest Common Divisor

EUCLID(x, y)
Two integers x and y with $x \geq y$;

if $y = 0$ **then** return x;
return(EUCLID$(y, x \mod y)$);

Lemma

If $a \geq b \geq 0$, then $a \mod b < a/2$

If they are initially n-bit integers, then the base case will be reached within $2n$ recursive calls. Since each call involves a quadratic-time division, the total time is $O(n^3)$.

Q: Suppose someone claims that d is the greatest common divisor of x and y, how can we check this?

It is not enough to verify that d divides both x and y...

Lemma

If d divides both x and y, and $d = ax + by$ for some integers a and b, then necessarily $d = gcd(x, y)$.

Proof:

$d \leq gcd(x, y)$, obviously;

$d \geq gcd(x, y)$, since $gcd(x, y)$ can divide x and y, it must also divide $ax + by = d$.

An Extension of Euclid's Algorithm

EXTENDED-EUCLID(x, y)
Two integers x and y with $x \geq y \geq 0$;

if $y = 0$ **then** return $(1, 0, x)$;
(x', y', d)=EXTENDED-EUCLID$(y, x \pmod y)$;
return $(y', x' - \lfloor x/y \rfloor y', d)$;

Correctness of the algorithm?
DIY!

Modular Inverse

We say x is the multiplicative inverse of $a \mod N$ if

$$ax \equiv 1 \mod N$$

There can be at most one such $x \mod N$, denoted a^{-1}.

Remark: The inverse does not always exists! for instance, 2 is not invertible modulo 6.

Lemma

If $gcd(a, N) > 1$*, then* $ax \not\equiv 1 \mod N$.

Proof:

$ax \mod N = ax + kN$, then $gcd(a, N)$ divides $ax \mod N$

If $gcd(a, N) = 1$, then extended Euclid algorithm gives us integers x and y such that $ax + Ny = 1$, which means $ax \equiv 1 \mod N$. Thus x is a's sought inverse.

Theorem (Modular Division Theorem)

For any $a \mod N$, a has a multiplicative inverse modulo N if and only if it is relatively prime to N.

When the inverse exists, it can be found in time $O(n^3)$ by running the extended Euclid algorithm.

This resolves the issues of modular division: when working modulo N, can divide by numbers relatively prime to N. And to actually carry out the division, multiply by the inverse.

Algorithms Design III

Algorithms with Numbers II

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Primality

Fermat's Little Theorem

Theorem

If p is a prime, then for every $1 \leq a < p$,

$$a^{p-1} \equiv 1 \pmod{p}$$

Proof:

Let $S = \{1, 2, \ldots, p-1\}$, then multiplying these numbers by $a \pmod{p}$ is to permute them.

$a.i \pmod{p}$ are distinct for $i \in S$, and all the values are nonzero.

multiplying all numbers in each representation, then gives $(p-1)! \equiv a^{(p-1)}.(p-1)! \pmod{p}$, and thus

$$1 \equiv a^{(p-1)} \pmod{p}$$

A (Problematic) Algorithm for Testing Primality

PRIMALITY(N)
Positive integer N;

Pick a positive integer $a < N$ at random;
if $a^{N-1} \equiv 1 \pmod{N}$ **then**
    return yes;
    else return no;
end

A (Problematic) Algorithm for Testing Primality

The problem is that Fermat's theorem is not an if-and-only-if condition.

- e.g. $341 = 11 \cdot 31$, and $2^{340} \equiv 1 \pmod{341}$

Our best hope: for composite N, most values of a will fail the test.

Rather than fixing an arbitrary value of a, we should choose it randomly from $\{1, \ldots, N-1\}$.

Carmichael Number

Theorem

There are composite numbers N such that for every $a < N$ relatively prime to N,

$$a^{N-1} \equiv 1 \pmod{N}$$

Example:

$$561 = 3 \cdot 11 \cdot 17$$

Lemma

If $a^{N-1} \not\equiv 1 \pmod{N}$ for some a relatively prime to N, then it must hold for at least half the choices of $a < N$.

Proof:

Fix some value of a for which $a^{N-1} \not\equiv 1(\mod N)$.

Assume some $b < N$ satisfies $b^{N-1} \equiv 1(\mod N)$, then

$$(a \cdot b)^{N-1} \equiv a^{N-1} \cdot b^{N-1} \equiv a^{N-1} \not\equiv 1(\mod N)$$

For $b \neq b'$, we have

$$a \cdot b \not\equiv a \cdot b' \mod N$$

The one-to-one function $b \mapsto a \cdot b(\mod N)$ shows that at least as many elements fail the test as pass it.

Primality Testing without Carmichael Numbers

We are ignoring Carmichael numbers, so we can assert,

- If N is prime, then $a^{N-1} \equiv 1 \mod N$ for all $a < N$
- If N is not prime, then $a^{N-1} \equiv 1 \mod N$ for at most half the values of $a < N$.

Therefore, (for non-Carmichael numbers)

- $Pr(\texttt{PRIMALITY}$ returns yes when N is prime$) = 1$
- $Pr(\texttt{PRIMALITY}$ returns yes when N is not prime$) \leq 1/2$

Primality Testing with Low Error Probability

PRIMALITY2(N)
Positive integer N;

Pick positive integers $a_1, \ldots, a_k < N$ at random;
if $a_i^{N-1} \equiv 1 \mod N$ *for all* $1 \leq i \leq k$ **then**
    return yes;
    else return no;
end

- Pr(PRIMALITY2 returns yes when N is prime)$= 1$
- Pr(PRIMALITY2 returns yes when N is not prime)$\leq 1/2^k$

Lagrange's Prime Number Theorem

Let $\pi(x)$ be the number of primes $\leq x$. Then $\pi(x) \approx x/ln(x)$, or more precisely,

$$\lim_{x \to \infty} \frac{\pi(x)}{(x/ln\, x)} = 1$$

Such abundance makes it simple to generate a random n-bit prime:

- Pick a random n-bit number N.
- Run a primality test on N.
- If it passes the test, output N; else repeat the process.

Q: How fast is this algorithm?

If the randomly chosen N is truly prime, which happens with probability at least $1/n$, then it will certainly pass the test.

On each iteration, this procedure has at least a $1/n$ chance of halting.

Therefore on average it will halt within $O(n)$ rounds.

- Exercise 1.34!

Tips: Randomized Algorithm

Monte Carlo Algorithm (MC):

- Always bounded in runtime
- Correctness is random
- Examples: Primary Testing

Las Vegas Algorithm (LV):

- Always correct
- Runtime is random (small time with good probability)
- Examples: Quicksort, Hashing

Cryptography

The Typical Setting for Cryptography

Alice and Bob, who wish to communicate in private.

Eve, an eavesdropper, will go to great lengths to find out what Alice and Bob are saying.

Even Ida, an intruder, will break the rules of communications positively.

The Typical Setting for Cryptography

Alice wants to send a specific message x, written in binary, to her friend Bob.

- Alice encodes it as $e(x)$, sends it over.
- Bob applies his decryption function $d(.)$ to decode it: $d(e(x)) = x$.
- Eve, will intercept $e(x)$: for instance, she might be a sniffer on the network.
- Ida, can do anything Eve does, he may also be able to pretend to be Alice or Bob.

Ideally, $e(x)$ is chosen that without knowing $d(.)$, Eve cannot do anything with the information she has picked up.

IOW, knowing $e(x)$ tells her little or nothing about what x might be.

Private VS. Public Schemes

For centuries, cryptography was based on what we now call private-key protocols. Alice and Bob meet beforehand and choose a secret codebook.

Public-key schemes allow Alice to send Bob a message without having met him before.

Bob is able to implement a digital lock, to which only he has the key. Now by making this digital lock public, he gives Alice a way to send him a secure message.

An encryption function:

$$e : \langle messages \rangle \rightarrow \langle encoded \ \ messages \rangle$$

e must be invertible, and is therefore a bijection.

- Alice and Bob secretly choose a binary string r of the same length as the message x that Alice will later send.
- Alice's encryption function is then a bitwise exclusive-or

$$e_r(x) = x \oplus r$$

- The function e_r is a bijection, and it is its own inverse:

$$e_r(e_r(x)) = (x \oplus r) \oplus r = x \oplus 0 = x$$

Why Secure?

Alice and Bob pick r at random.

This will ensure that if Eve intercepts the encoded message $y = e_r(x)$, she gets no information about x.

Why One-Time Pad

One-time pad is impractical and unsafe when r is repeatedly used.

Any one can get $x \oplus z$ when they know $x \oplus r$ and $z \oplus r$.

- it reveals whether the two messages begin or end the same;
- if one message contains a long sequence of zeros, then the corresponding part of the other message will be exposed.

If Ida is powerful enough that pretends to be Bob. . .

Therefore the random string that Alice and Bob share has to be the combined length of all the messages they will need to exchange.

- Random strings are costly!

AES (advanced encryption standard)

- 128-bit fixed size.
- repeatedly use

Public-Key Schemes

Anybody can send a message to anybody else using publicly available information, rather like addresses or phone numbers.

Each person has a public key known to the whole world and a secret key known only to himself.

When Alice wants to send message x to Bob, she encodes it using his public key.

Bob decrypts it using his secret key, to retrieve x.

Eve is welcome to see as many encrypted messages, but she will not be able to decode them, under certain assumptions.

The RSA Cryptosystem: Fundamental Property

Pick up two primes p and q and let $N = pq$.

For any e relatively prime to $(p-1)(q-1)$:

- The mapping $x \mapsto x^e \mod N$ is a bijection on $\{0, 1, \ldots N-1\}$.
- The inverse mapping is easily realized: let d be the inverse of e modulo $(p-1)(q-1)$. Then for all $x \in \{0, 1, \ldots, N-1\}$,

$$(x^e)^d \equiv x \mod N$$

The mapping $x \mapsto x^e \mod N$ is a reasonable way to encode messages x. If Bob publishes (N, e) as his public key, everyone else can use it to send him encrypted messages.

Bob retain the value d as his secret key, with which he can decode all messages that come to him by simply raising them to the d-th power modulo N.

Proof of the Property

Proof:

If the mapping $x \to x^e \mod N$ is invertible, it must be a bijection; hence statement 2 implies statement 1.

To prove statement 2, observe that e is invertible modulo $(p-1)(q-1)$ because it is relatively prime to this number.

To show that $(x^e)^d \equiv x \mod N$: Since $ed \equiv 1 \mod (p-1)(q-1)$, can write $ed = 1 + k(p-1)(q-1)$ for some k.

Then

$$(x^e)^d - x = x^{ed} - x = x^{1+k(p-1)(q-1)} - x$$

$x^{1+k(p-1)(q-1)} - x$ is divisible by p (since $x^{p-1} \equiv 1 \mod p$) and likewise by q. Since p and q are primes, this expression must be divisible by $N = pq$.

Bob chooses his public and secret keys:

- He starts by picking two large (n-bit) random primes p and q.
- His public key is (N, e) where $N = pq$ and e is a $2n$-bit number relatively prime to $(p-1)(q-1)$.
- his secret key is d, the inverse of e modulo $(p-1)(q-1)$.

Alice wishes to send message x to Bob

- She looks up his public key (N, e) and sends him $y = (x^e \mod N)$.
- He decodes the message by computing $y^d \mod N$.

Security Assumption of RSA

The security of RSA hinges upon a simple assumption

Given N, e and $y = x^e \mod N$, it is computationally intractable to determine x.

How might Eve try to guess x

She could experiment with all possible values of x, each time checking whether $x^e \equiv y \mod N$, but this would take exponential time.

How might Eve try to guess x

she could try to factor N to retrieve p and q, and then figure out d by inverting e modulo $(p-1)(q-1)$, but we believe factoring to be hard.

Digital Signature

A digital signature scheme is a mathematical scheme for demonstrating the authenticity of a digital message or document.

In a digital signature scheme, there are two algorithms, signing and verifying.

A signing algorithm that, given a message and a private key, produces a signature.

A signature verifying algorithm that, given a message, public key and a signature, either accepts or rejects the message's claim to authenticity.

Is Communication Safe?

Is a communication safe in the internet when cryptography is unbreakable?

- No!

The NSPK Protocol

$$
\begin{aligned}
A \longrightarrow B: &\quad \{A, N_A\}_{+K_B} \\
B \longrightarrow A: &\quad \{N_A, N_B\}_{+K_A} \\
A \longrightarrow B: &\quad \{N_B\}_{+K_B}
\end{aligned}
$$

An Attack

$$
\begin{array}{rcll}
A & \longrightarrow & I: & \{A, N_A\}_{+K_I} \\
I(A) & \longrightarrow & B: & \{A, N_A\}_{+K_B} \\
B & \longrightarrow & I(A): & \{N_A, N_B\}_{+K_A} \\
I & \longrightarrow & A: & \{N_A, N_B\}_{+K_A} \\
A & \longrightarrow & I: & \{N_B\}_{+K_I} \\
I(A) & \longrightarrow & B: & \{N_B\}_{+K_B}
\end{array}
$$

The Fixed NSPK Protocol

$$
\begin{aligned}
A &\longrightarrow B: && \{A, N_A\}_{+K_B} \\
B &\longrightarrow A: && \{B, N_A, N_B\}_{+K_A} \\
A &\longrightarrow B: && \{N_B\}_{+K_B}
\end{aligned}
$$

$$
\begin{aligned}
A &\longrightarrow I: && \{A, N_A\}_{+K_I} \\
I(A) &\longrightarrow B: && \{A, N_A\}_{+K_B} \\
B &\longrightarrow I(A): && \{B, N_A, N_B\}_{+K_A} \\
I &\not\longrightarrow A: && \{I, N_A, N_B\}_{+K_A}
\end{aligned}
$$

Divide-and-Conquer

The divide-and-conquer strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.

Product of Complex Numbers

Carl Friedrich Gauss(1777-1855) noticed that although the product of two complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and $(a + b)(c + d)$, since

$$bc + ad = (a + b)(c + d) - ac - bd$$

- In big O way of thinking, reducing the number of multiplications from four to three seems wasted ingenuity.
- But this modest improvement becomes very significant when applied recursively.

Multiplication

Suppose x and y are two n-integers, and assume for convenience that n is a power of 2.

[Hints: For every n there exists an n' with $n \leq n' \leq 2n$ such that n' a power of 2.]

As a first step toward multiplying x and y, we split each of them into their left and right halves, which are $n/2$ bits long

$$x = \boxed{x_L}\ \boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L}\ \boxed{y_R} = 2^{n/2}y_L + y_R$$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

Additions and multiplications by powers of 2 take linear time.

The additions take linear time, as do multiplications by powers of 2 (that is, $O(n)$).

The significant operations are the four $n/2$-bit multiplications: these can be handled by four recursive calls.

Writing $T(n)$ for the overall running time on n-bit inputs, we get the recurrence relations:

$$T(n) = 4T(n/2) + O(n)$$

Solution: $O(n^2)$

By Gauss's trick, three multiplications $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$ suffice.

Algorithm for Integer Multiplication

`MULTIPLY(`x, y`)`
Two positive integers x and y, in binary;

n=`max` (size of x, size of y) rounded as a power of 2;
if $n = 1$ **then** `return(`xy`)`;
x_L, x_R= leftmost $n/2$, rightmost $n/2$ bits of x;
y_L, y_R= leftmost $n/2$, rightmost $n/2$ bits of y;
$P1$=`MULTIPLY(`x_L, y_L`)`;
$P2$=`MULTIPLY(`x_R, y_R`)`;
$P3$=`MULTIPLY(`$x_L + x_R, y_L + y_R$`)`;
`return(`$P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$`)`

The recurrence relation

$$T(n) = 3T(n/2) + O(n)$$

The algorithm's recursive calls form a tree structure.

At each successive level of recursion the subproblems get halved in size.

At the $(log_2\, n)^{th}$ level, the subproblems get down to size 1, and so the recursion ends.

The height of the tree is $\log_2 n$.

The branch factor is 3: each problem produces three smaller ones, with the result that at depth k there are 3^k subproblems, each of size $n/2^k$.

For each subproblem, a linear amount of work is done in combining their answers.

Time Analysis

The total time spent at depth k in the tree is

$$3^k \times O(\frac{n}{2^k}) = (\frac{3}{2})^k \times O(n)$$

At the top level, when $k = 0$, we need $O(n)$.

At the bottom, when $k = \log_2 n$, it is $O(3^{\log_2 n}) = O(n^{\log_2 3})$

The work done increases geometrically from $O(n)$ to $O(n^{\log_2 3})$, by a factor of $3/2$ per level.

The sum of any increasing geometric series is, within a constant factor, the last term of the series.

Therefore, the overall running time is

$$O(n^{\log_2 3}) \approx O(n^{1.59})$$

Q: Can we do better?

- Yes!

Recurrence Relations

Master Theorem

If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$ and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

The Proof of the Theorem

Proof:

Assume that n is a power of b.

The size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels - the the height of the recursion tree.

Its branching factor is a, so the k-th level of the tree is made up of a^k subproblems, each of size n/b^k.

$$a^k \times O(\frac{n}{b^k})^d = O(n^d) \times (\frac{a}{b^d})^k$$

k goes from 0 to $\log_b n$, these numbers form a geometric series with ratio a/b^d, comes down to three cases.

The Proof of the Theorem

The ratio is less than 1.

Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.

The ratio is greater than 1.

The series is increasing and its sum is given by its last term, $O(n^{\log_b a})$

The ratio is exactly 1.

In this case all $O(\log n)$ terms of the series are equal to $O(n^d)$.

Quiz

$T(n) = 4 \cdot T(\sqrt{n}) + 1$

Merge Sort

The Algorithm


```
MERGESORT(a[1...n])
An array of numbers a[1...n];
if n > 1 then
    return(MERGE(MERGESORT(a[1...⌊n/2⌋]),
     MERGESORT(a[⌊n/2⌋ + 1..., n])));
    else return(a);
end
```

```
MERGE(x[1...k], y[1...l])
if k = 0 then return y[1...l];
if l = 0 then return x[1...k];
if x[1] ≤ y[1] then return( x[1]∘MERGE(x[2...k], y[1...l]));
else return( y[1]∘MERGE(x[1...k], y[2...l]));
```

An Iterative Version

ITERTIVE-MERGESORT($a[1 \dots n]$)
An array of numbers $a[1 \dots n]$;

$Q = [\ ]$ *empty queue*;
for $i = 1$ *to* n **do**
  Inject(Q, $[a[i]]$);
end
while $|Q| > 1$ **do**
  Inject (Q,MERGE (Eject (Q),Eject (Q)));
end
return(Eject *(Q)*);

The Time Analysis

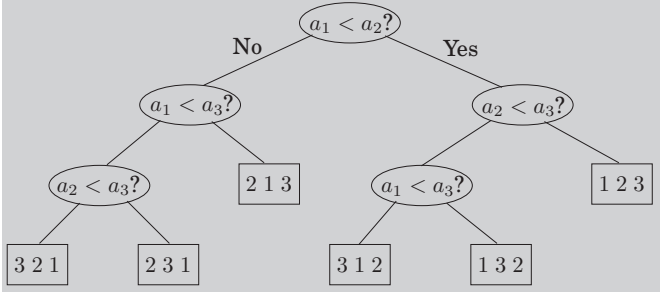
The recurrence relation:

$$T(n) = 2T(n/2) + O(n)$$

By Master Theorem:

$$T(n) = O(n \log n)$$

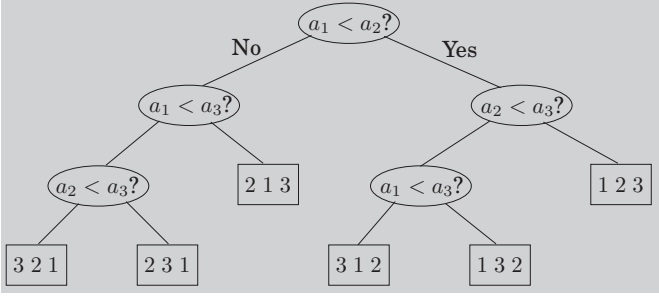
Q: Can we do better?



Sorting algorithms can be depicted as trees.

The depth of the tree - the number of comparisons on the longest path from root to leaf, is the worst-case time complexity of the algorithm.

Assume n elements. Each of its leaves is labeled by a permutation of $\{1, 2, \ldots, n\}$.



Every permutation must appear as the label of a leaf.

This is a binary tree with $n!$ leaves.

So, the depth of the tree - and the complexity of the algorithm - must be at least

$$\log(n!) \approx \log(\sqrt{\pi(2n+1/3)} \cdot n^n \cdot e^{-n}) = \Omega(n \log n)$$

Median

Median

The median of a list of numbers is its $50th$ percentile: half the number are bigger than it, and half are smaller.

If the list has even length, we pick the smaller one of the two.

The purpose of the median is to summarize a set of numbers by a single typical value.

Computing the median of n numbers is easy, just sort them. ($O(n \log n)$).

Q: Can we do better?

Selection

Input: A list of number S; an integer k.
Output: The $k\,th$ smallest element of S.

A Randomized Selection

For any number v, imagine splitting list S into three categories:

- elements smaller than v, i.e., S_L;
- those equal to v, i.e., S_v (there might be duplicates);
- and those greater than v, i.e., S_R; respectively.

$$selection(S,k) = \begin{cases} selection(S_L,k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ selection(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v| \end{cases}$$

How to Choose v?

It should be picked quickly, and it should shrink the array substantially, the ideal situation being

$$| S_L |, | S_R | \approx \frac{| S |}{2}$$

If we could always guarantee this situation, we would get a running time of

$$T(n) = T(n/2) + O(n) = O(n)$$

But this requires picking v to be the median, which is our ultimate goal!

Instead, we pick v randomly from S!

How to Choose v?

Worst-case scenario would force our selection algorithm to perform

$$n + (n-1) + (n-2) + \ldots + \frac{n}{2} = \Theta(n^2)$$

Best-case scenario $O(n)$

v is good if it lies within the $25th$ to $75th$ percentile of the array that it is chosen from.

A randomly chosen v has a 50% chance of being good.

Lemma

On average a fair coin needs to be tossed two times before a heads is seen.

Proof:

Let E be the expected number of tosses before heads is seen.

$$E = 1 + \frac{1}{2}E$$

Therefore, $E = 2$.

Let $T(n)$ be the expected running time on the array of size n, we get

$$T(n) \leq T(3n/4) + O(n) = O(n)$$

Matrix Multiplication

The product of two $n \times n$ matrices X and Y is a $n \times n$ matrix $Z = XY$, with which $(i, j)th$ entry

$$Z_{ij} = \sum_{i=1}^{n} X_{ik} Y_{kj}$$

In general, matrix multiplication is not commutative, say, $XY \neq YX$

The running time for matrix multiplication is $O(n^3)$

- There are n^2 entries to be computed, and each takes $O(n)$ time.

Matrix multiplication can be performed blockwise.

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

$$T(n) = 8T(n/2) + O(n^2)$$

$$T(n) = O(n^3)$$

Strassen Algorithm

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

$$T(n) = 7T(n/2) + O(n^2)$$

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

Algorithm Design V

Divide and Conquer II

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Counting Inversions

Counting Inversions

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: $1, 2, \ldots, n$.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs i and j are inverted if $i < j$, but $a_i > a_j$.

	A	B	C	D	E
me	1	2	3	4	5
you	1	3	4	2	5

2 **inversions**: 3-2, 4-2

Divide-and-Conquer

Divide: separate list into two halves A and B.

Conquer: recursively count inversions in each list.

Combine: count inversions (a, b) with $a \in A$ and $b \in B$.

Return sum of three counts.

input

1	5	4	8	10	2	6	9	3	7

count inversions in left half A

1	5	4	8	10

5-4

count inversions in right half B

2	6	9	3	7

6-3 9-3 9-7

count inversions (a, b) with $a \in A$ and $b \in B$

1	5	4	8	10

2	6	9	3	7

4-2 4-3 5-2 5-3 8-2 8-3 8-6 8-7 10-2 10- 3 10-6 10-7 10-9

output 1+3+13=17

How to Combine Two Subproblems?

Q. How to count inversions (a, b) with $a \in A$ and $b \in B$?

A. Easy if A and B are sorted!

Warmup algorithm.

- Sort A and B.
- For each element $b \in B$,
  - binary search in A to find how many elements in A are greater than b.

list A

7	10	18	3	14

list B

20	23	2	11	16

sort A

3	7	10	14	18

sort B

2	11	16	20	23

binary search to count inversions (a, b) with $a \in A$ and $b \in B$

3	7	10	14	18

2	11	16	20	23
5	2	1	0	0

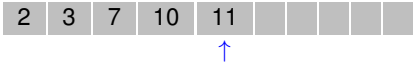
How to Combine Two Subproblems?

Count inversions (a, b) with $a \in A$ and $b \in B$, assuming A and B are sorted.

- Scan A and B from left to right.
- Compare a_i and b_j.
- If $a_i < b_j$, then a_i is not inverted with any element left in B.
- If $a_i > b_j$, then b_j is inverted with every element left in A.
- Append smaller element to sorted list C.

count inversions (a, b) with $a \in A$ and $b \in B$

merge to form sorted list C



Algorithm Implementation

SHANGHAI JIAO TONG UNIVERSITY

SORT-AND-COUNT(L) ;
input : List L
output: Number of inversions in L and L in sorted order

if *List L has one element* **then**
  RETURN $(0, L)$;
end
Divide the list into two halves A and B;
$(r_A, A) \leftarrow$ SORT-AND-COUNT (A);
$(r_B, B) \leftarrow$ SORT-AND-COUNT (B);
$(r_{AB}, L) \leftarrow$ MERGE-AND-COUNT (A,B);
RETURN $(r_A + r_B + r_{AB}, L)$;

Proposition

The sort-and-count algorithm counts the number of inversions in a permutation of size n in $O(n \log n)$ time.

Proof.

$$T(n) = 2 \cdot T(\lceil n/2 \rceil) + \Theta(n)$$

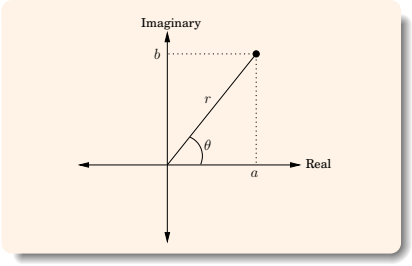
Complex Number

Complex Number

$z = a + bi$ is plotted at position (a, b).

In its polar coordinates, denoted (r, θ), rewrite as

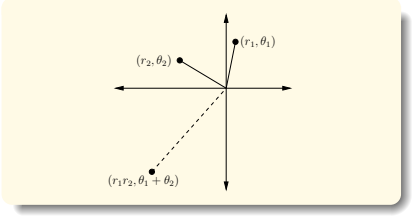
$$z = r(\cos\theta + i\sin\theta) = re^{i\theta}$$



- length: $r = \sqrt{a^2 + b^2}$.
- angle: $\theta \in [0, 2\pi)$.
- θ can always be reduced modulo 2π.

Basic arithmetic:

- $-z = (r, \theta + \pi)$.
- $(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.
- If z is on the unit circle (i.e., $r = 1$), then $z^n = (1, n\theta)$.



The n-th Complex Roots of Unity

Solutions to the equation $z^n = 1$

- by the multiplication rules: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$.
- It can be represented as

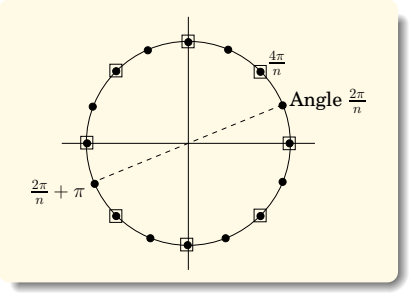
$$1, \omega, \omega^2, \ldots, \omega^{n-1}$$

where

$$\omega = e^{2\pi i/n}$$

For n is even:

- These numbers are plus-minus paired.
- Their squares are the $(n/2)$-nd roots of unity.



Complex Conjugate

The complex conjugate of a complex number $z = re^{i\theta}$ is $z^* = re^{-i\theta}$.

The complex conjugate of a vector (or a matrix) is obtained by taking the complex conjugates of all its entries.

The angle between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v(v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their inner product

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \ldots + u_{n-1} v_{n-1}^*$$

The above quantity is maximized when the vectors lie in the same direction and is zero when the vectors are orthogonal to each other.

The Fast Fourier Transform

Polynomial multiplication

If $A(x) = a_0 + a_1 x + \ldots + a_d x^d$ and $B(x) = b_0 + b_1 x + \ldots + b_d x^d$, their product

$$C(x) = c_0 + c_1 x + \ldots + c_{2d} x^{2d}$$

has coefficients

$$c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_k b_0 = \sum_{i=0}^{k} a_i b_{k-i}$$

where for $i > d$, take a_i and b_i to be zero.

Computing c_k from this formula take $O(k)$ step, and finding all $2d + 1$ coefficients would therefore seem to require $\Theta(d^2)$ time.

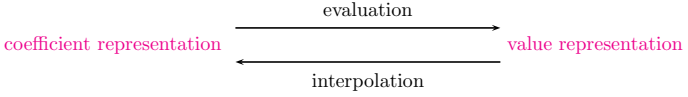
Q: Can we do better?

An alternative representation

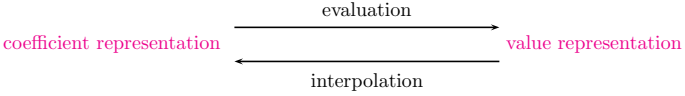
Fact: A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points.

We can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \ldots + a_d x^d$ by either of the following:

- Its coefficients $a_0, a_1, \ldots, a_d$. (coefficient representation).
- The values $A(x_0), A(x_1), \ldots A(x_d)$ (value representation).



An alternative representation



The product $C(x)$ has degree $2d$, it is determined by its value at any $2d + 1$ points.

Its value at any given point z is just $A(z)$ times $B(z)$.

Therefore, polynomial multiplication takes linear time in the value representation.

The algorithm

Input: Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree d
Output: Their product $C = A \cdot B$

Selection
Pick some points $x_0, x_1, \ldots, x_{n-1}$, where $n \geq 2d + 1$.

Evaluation
Compute $A(x_0), A(x_1), \ldots, A(x_{n-1})$ and $B(x_0), B(x_1), \ldots, B(x_{n-1})$.

Multiplication
Compute $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \ldots, n-1$.

Interpolation
Recover $C(x) = c_0 + c_1 x + \ldots + c_{2d} x^{2d}$

The selection step and the multiplications are just linear time:

- In a typical setting for polynomial multiplication, the coefficients of the polynomials are real number.
- Moreover, are small enough that basic arithmetic operations take unit time.

Evaluating a polynomial of degree $d \leq n$ at a single point takes $O(n)$, and so the baseline for n points is $\Theta(n^2)$.

The Fast Fourier Transform (FFT) does it in just $O(n \log n)$ time, for a particularly clever choice of $x_0, \ldots, x_{n-1}$.

Evaluation by divide-and-conquer

Q: How to make it efficient?

First idea, we pick the n points,

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1}$$

then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, because the even power of x_i coincide with those of $-x_i$.

We need to split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 = (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4)$$

More generally

$$A(x) = A_e(x^2) + xA_o(x^2)$$

where $A_e(\cdot)$, with the even-numbered coefficients, and $A_o(\cdot)$, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$.

Evaluation by divide-and-conquer

Given paired points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled toward computing $A(-x_i)$:

$$A(x_i) = A_e(x_i^2) + x_i A_o(x_i^2)$$

$$A(-x_i) = A_e(x_i^2) - x_i A_o(x_i^2)$$

Evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.

If we could recurse, we would get a divide-and-conquer procedure with running time

$$T(n) = 2T(n/2) + O(n) = O(n \log n)$$

How to choose n points?

Aim: To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be themselves plus-minus pairs.

Q: How can a square be negative?

- We use complex numbers.

At the very bottom of the recursion, we have a single point, 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$.

The next level up then has $\pm\sqrt{+1} = \pm 1$, as well as the complex numbers $\pm\sqrt{-1} = \pm i$.

By continuing in this manner, we eventually reach the initial set of n points: the complex $n\,th$ roots of unity, that is the n complex solutions of the equation

$$z^n = 1$$

The n-th complex roots of unity

Solutions to the equation $z^n = 1$

- by the multiplication rules: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$.
- It can be represented as

$$1, \omega, \omega^2, \ldots, \omega^{n-1}$$

where

$$\omega = e^{2\pi i/n}$$

For n is even:

- These numbers are plus-minus paired.
- Their squares are the $(n/2)$-nd roots of unity.

The FFT algorithm

`FFT(`A, ω`)`

input : coefficient reprentation of a polynomial $A(x)$ of degree $\leq n-1$, where n is a power of 2;
ω, an n-th root of unity

output: value representation $A(\omega^0), \ldots, A(\omega^{n-1})$

if $\omega = 1$ **then** `return` $A(1)$;
express $A(x)$ in the form $A_e(x^2) + xA_o(x^2)$;
call `FFT(`A_e,ω^2`)` to evaluate A_e at even powers of ω;
call `FFT(`A_o,ω^2`)` to evaluate A_o at even powers of ω;
for $j = 0$ *to* $n-1$ **do**
  compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$;
end
`return(`$A(\omega^0), \ldots, A(\omega^{n-1})$`)`;

`FFT` moves from coefficients to values in time just $O(n \log n)$, when the points $\{x_i\}$ are complex n-th roots of unity $(1, \omega, \omega^2, \ldots, \omega^{n-1})$.

That is,

$$\langle value \rangle = \texttt{FFT}(\langle coefficients \rangle, \omega)$$

We will see that the interpolation can be computed by

$$\langle coefficients \rangle = \frac{1}{n}\texttt{FFT}(\langle values \rangle, \omega^{-1})$$

A matrix reformation

Let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$.

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Let M be the matrix in the middle, which is a Vandermonde matrix.

- If $x_0, x_1, \dots, x_{n-1}$ are distinct numbers, then M is invertible.
- evaluation is multiplication by M, while interpolation is multiplication by M^{-1}.

A matrix reformation

This reformulation of our polynomial operations reveals their essential nature more clearly.

It justifies an assumption that $A(x)$ is uniquely characterized by its values at any n points.

Vandermonde matrices also have the distinction of being quicker to invert than more general matrices, in $O(n^2)$ time instead of $O(n^3)$.

However, using this for interpolation would still not be fast enough for us..

Interpolation resolved

In linear algebra terms, the `FFT` multiplies an arbitrary n-dimensional vector, which we have been calling the coefficient representation, by the $n \times n$ matrix.

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \dots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & x^{(n-1)(n-1)} \end{bmatrix}$$

Its (j, k)-th entry (starting row- and column-count at zero) is ω^{jk}

Interpolation resolved

The columns of M are orthogonal to each other, which is often called the Fourier basis.

The FFT is thus a change of basis, a rigid rotation. The inverse of M is the opposite rotation, from the Fourier basis back into the standard basis.

Inversion formula

$$M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$$

Interpolation resolved

Take ω to be $e^{2\pi i/n}$, and think of M as vectors in $\mathbb{C}^n$.

Recall that the angle between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v(v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their inner product

$$u \cdot v^* = u_0 v_0^* + u_1 v_1^* + \ldots + u_{n-1} v_{n-1}^*$$

where z^* denotes the complex conjugate of z.

The above quantity is maximized when the vectors lie in the same direction and is zero when the vectors are orthogonal to each other.

Lemma

The columns of matrix M are orthogonal to each other.

Proof.

- Take the inner product of of any columns j and k of matrix M,

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \ldots + \omega^{(n-1)(j-k)}$$

  This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio ω^{j-k}.
- Therefore, if $j \neq k$, it evaluates to

$$\frac{1 - \omega^{n(j-k)}}{1 - \omega^{(j-k)}} = 0$$

- If $j = k$, then it evaluates to n.

Interpolation resolved

Corollary

$MM^* = nI$, *i.e.*,

$$M_n^{-1} = \frac{1}{n} M_n^*$$

The definitive FFT algorithm

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex n-th roots of unity.

It multiplies vector a by the $n \times n$ matrix $M_n(\omega)$, which has (j, k)-th entry ω^{jk}.

The potential for using divide-and-conquer in this matrix-vector multiplication becomes apparent when M's columns are segregated into evens and odds.

The product of $M_n(\omega)$ with vector $a = (a_0, \ldots, a_{n-1})$, a size-$n$ problem, can be expressed in terms of two size-$n/2$ problems: the product of $M_{n/2}(\omega^2)$ with $(a_0, a_2, \ldots, a_{n-2})$ and with $(a_1, a_3, \ldots, a_{n-1})$.

This divide-and-conquer strategy leads to the definitive FFT algorithm, whose running time is $T(n) = 2T(n/2) + O(n) = O(nlogn)$.

The general FFT algorithm

`FFT` (a, ω)

input : An array $a = (a_0, a_1, \ldots, a_{n-1})$ for n is a power of 2; ω, an n-th root of unity

output: $M_n(\omega)a$

if $\omega = 1$ **then** `return` a;

$(s_0, s_1, \ldots, s_{n/2-1})$=`FFT` $((a_0, a_2, \ldots, a_{n-2}), \omega^2)$;

$(s'_0, s'_1, \ldots, s'_{n/2-1})$=`FFT` $((a_1, a_3, \ldots, a_{n-1}), \omega^2)$;

for $j = 0$ *to* $n/2 - 1$ **do**

  r_j=$s_j + \omega^j s'_j$;

  $r_{j+n/2}$=$s_j - \omega^j s'_j$;

end

`return` $(r_0, r_1, \ldots, r_{n-1})$;

Top 10 algorithms of the 20th century

1946: The Metropolis Algorithm

1947: Simplex Method

1950: Krylov Subspace Method

1951: The Decompositional Approach to Matrix Computations

1957: The Fortran Optimizing Compiler

1959: QR Algorithm

1962: Quicksort

1965: Fast Fourier Transform

1977: Integer Relation Detection

1987: Fast Multipole Method

Top 10 algorithms of the 20th century

1946: The Metropolis Algorithm

1947: Simplex Method

1950: Krylov Subspace Method

1951: The Decompositional Approach to Matrix Computations

1957: The Fortran Optimizing Compiler

1959: QR Algorithm

1962: Quicksort

1965: Fast Fourier Transform

1977: Integer Relation Detection

1987: Fast Multipole Method

Algorithm Design VI

Decompositions of Graphs

Guoqiang Li
School of Software, Shanghai Jiao Tong University

An Exercise

Let B be an $n \times n$ chessboard, where n is a power of 2. Use a divide-and-conquer argument to describe how to cover all squares of B except one with L-shaped tiles. For example, if $n = 2$, then there are four squares three of which can be covered by one L-shaped tile, and if $n = 4$, then there are 16 squares of which 15 can be covered by 5 L-shaped tiles.

Decompositions of Graphs

EXPLORE (G, v)
input : $G = (V, E)$ is a graph; $v \in V$
output: $visited(u)$ to $true$ for all nodes u reachable from v

$visited(v) = true$;
PREVISIT (v);
for *each edge* $(v, u) \in E$ **do**
  if *not* $visited(u)$ **then** EXPLORE (G, u);
end
POSTVISIT (v);

Depth-First Search

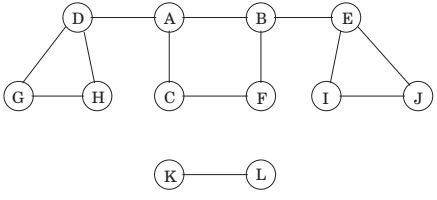

```
DFS(G)
for all v ∈ V do
    visited(v) = false;
end
for all v ∈ V do
    if not visited(v) then Explore(G, v);
end
```

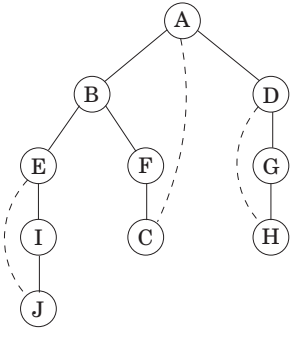
Connectivity in Undirected Graphs

Types of Edges in Undirected Graphs

Those edges in G that are traversed by `EXPLORE` are tree edges.

The rest are back edges.





Connectivity in Undirected Graphs

Definition

An undirected graph is connected, if there is a path between any pair of vertices.

Definition

A connected component is a subgraph that is internally connected but has no edges to the remaining vertices.

When `EXPLORE` is started at a particular vertex, it identifies precisely the connected component containing that vertex.

Each time the DFS outer loop calls `EXPLORE`, a new connected component is picked out.

DFS is trivially adapted to check if a graph is connected.

More generally, to assign each node v an integer $ccnum[v]$ identifying the connected component to which it belongs.

```
PREVISIT(v)
ccnum[v] = cc;
```

where cc needs to be initialized to zero and to be incremented each time the `DFS` procedure calls `EXPLORE`.

Previsit and Postvisit Orderings

For each node, we will note down the times of two important events:

- the moment of first discovery (corresponding to `PREVISIT`);
- and the moment of final departure (`POSTVISIT`).

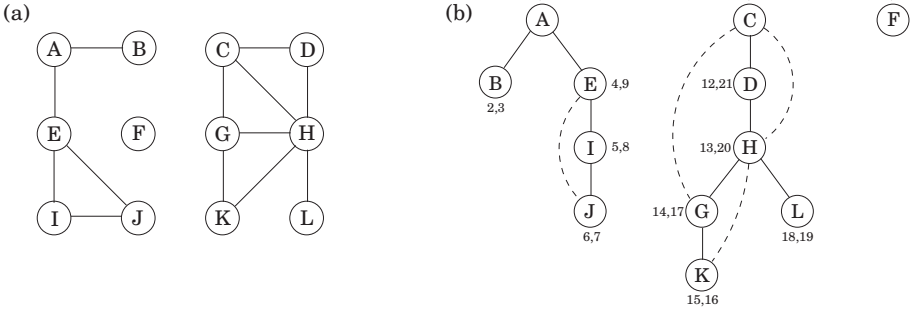
```
PREVISIT(v)
pre[v] = clock;
clock++;
```

```
POSTVISIT(v)
post[v] = clock;
clock++;
```

Lemma

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

Previsit and Postvisit Orderings



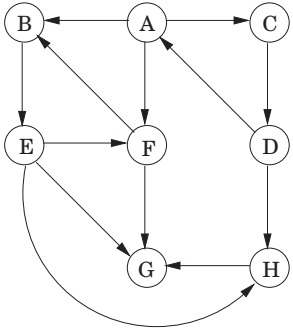
Connectivity in Directed Graphs

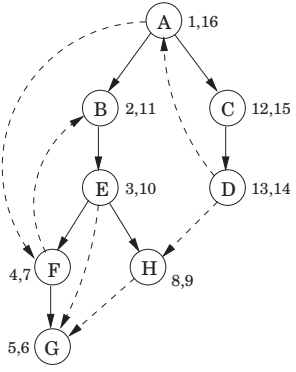
Types of Edges in Directed Graphs

DFS yields a search tree/forests.

- root.
- descendant and ancestor.
- parent and child.
- Tree edges are actually part of the DFS forest.
- Forward edges lead from a node to a nonchild descendant in the DFS tree.
- Back edges lead to an ancestor in the DFS tree.
- Cross edges lead to neither descendant nor ancestor.

Directed Graphs





Types of Edges

pre/post ordering for (u, v)				Edge type
$[_u$	$[_v$	$]_v$	$]_u$	Tree/forward
$[_v$	$[_u$	$]_u$	$]_v$	Back
$[_v$	$]_v$	$[_u$	$]_u$	Cross

Q: Is that all?

Definition

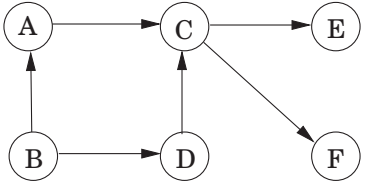
A cycle in a directed graph is a circular path

$$v_0 \to v_1 \to v_2 \to \ldots v_k \to v_0$$

Lemma

A directed graph has a cycle if and only if its depth-first search reveals a back edge.

Directed Acyclic Graphs (DAG)



Linearization/Topologically Sort: Order the vertices such that every edge goes from a earlier vertex to a later one.

Q: What types of dags can be linearized?

A: All of them.

DFS tells us exactly how to do it: perform tasks in decreasing order of their post numbers.

The only edges (u, v) in a graph for which $post(u) < post(v)$ are back edges, and we have seen that a DAG cannot have back edges.

Lemma

In a DAG, every edge leads to a vertex with a lower post number.

There is a linear-time algorithm for ordering the nodes of a DAG.

Acyclicity, linearizability, and the absence of back edges during a depth-first search - are the same thing.

The vertex with the smallest post number comes last in this linearization, and it must be a sink - no outgoing edges.

Symmetrically, the one with the highest post is a source, a node with no incoming edges.

Lemma

Every DAG has at least one source and at least one sink.

The guaranteed existence of a source suggests an alternative approach to linearization:

1. Find a source, output it, and delete it from the graph.
2. Repeat until the graph is empty.

Strongly Connected Components

Defining Connectivity for Directed Graphs

Definition

Two nodes u and v of a directed graph are connected if there is a path from u to v and a path from v to u.

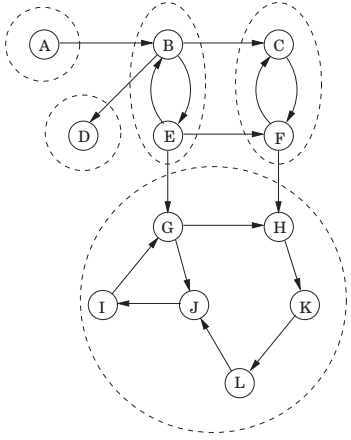
This relation partitions V into disjoint sets that we call strongly connected components (SCC).

Lemma

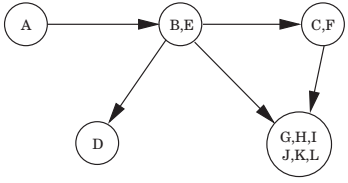
Every directed graph is a DAG of its SCC.

Strongly Connected Components

(a)



(b)



Lemma

If the `EXPLORE` *subroutine at node* u*, then it will terminate precisely when all nodes reachable from* u *have been visited.*

If we call explore on a node that lies somewhere in a sink SCC, then we will retrieve exactly that component.

We have two problems:

1. How do we find a node that we know for sure lies in a sink SCC?
2. How do we continue once this first component has been discovered?

Lemma

The node that receives the highest post number in a depth-first search must lie in a source SCC.

Lemma

If C and C' are SCC, and there is an edge from a node in C to a node in C', then the highest post number in C is bigger than the highest post number in C'.

Hence the SCCs can be linearized by arranging them in decreasing order of their highest post numbers.

Solving Problem A

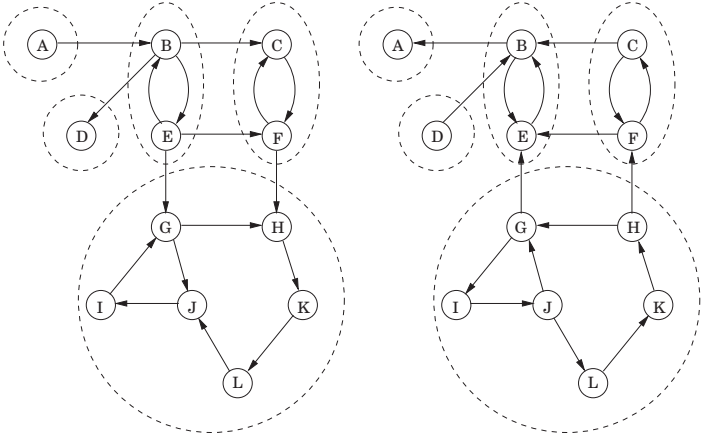
Consider the reverse graph G^R, the same as G but with all edges reversed.

G^R has exactly the same SCCs as G.

If we do a depth-first search of G^R, the node with the highest post number will come from a source SCC in G^R.

It is a sink SCC in G.

Strongly Connected Components



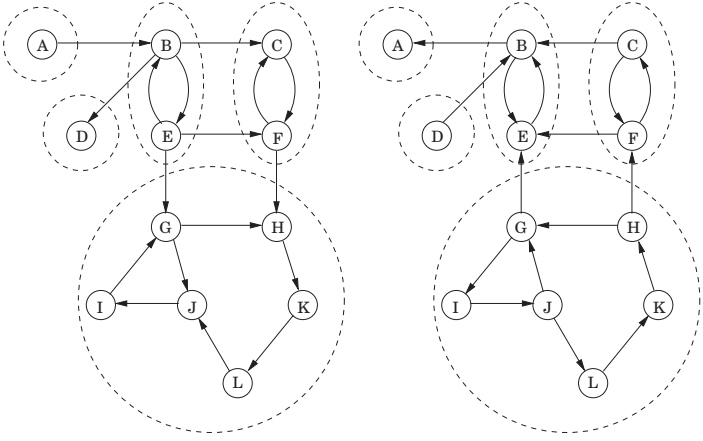
Once we have found the first SCC and deleted it from the graph, the node with the highest post number among those remaining will belong to a sink SCC of whatever remains of G.

Therefore we can keep using the post numbering from our initial depth-first search on G^R to successively output the second strongly connected component, the third SCC, and so on.

The Linear-Time Algorithm

1. Run depth-first search on G^R.
2. Run the EXPLORE algorithm on G, and during the depth-first search, process the vertices in decreasing order of their post numbers from step 1.

Strongly Connected Components



Think About

How the SCC algorithm works when the graph is very, very huge?

Think About

How about edges instead of paths?

Exercises

Exercises 1

Suppose a CS curriculum consists of n courses, all of them mandatory. The prerequisite graph G has a node for each course, and an edge from course v to course w if and only if v is a prerequisite for w. Find an algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum (assume that a student can take any number of courses in one semester). The running time of your algorithm should be linear.

Exercises 2

Give an efficient algorithm which takes as input a directed graph $G = (V, E)$, and determines whether or not there is a vertex $s \in V$ from which all other vertices are reachable.

Algorithm Design VII

Path in Graphs

Guoqiang Li
School of Software

Shanghai Jiao Tong University

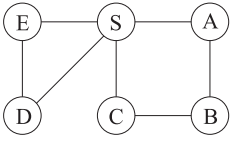
Distances

Distances

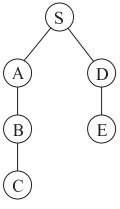
Definition

The distance between two nodes is the length of the shortest path between them.

(a)



(b)



Breadth-First Search

`BFS` (G, s)
input : Graph $G = (V, E)$, directed or undirected; Vertex $s \in V$
output: For all vertices u reachable from s, $dist(u)$ is the set to the distance from s to s

for *all* $v \in V$ **do**
  $dist(v) = \infty$;
end
$dist[s] = 0$;
$Q = [s]$ *queue containing just* v;
while Q *is not empty* **do**
  u=`Eject` (Q);
  for *all edge* $(u, v) \in E$ **do**
    if $dist(v) = \infty$ **then**
      `Inject` (Q,v); $dist[v] = dist[u] + 1$;
    end
  end
end

Correctness

Lemma

For each $d = 0, 1, 2, \ldots$ there is a moment at which,

1. *all nodes at distance $\leq d$ from s have their distances correctly set;*
2. *all other nodes have their distances set to ∞; and*
3. *the queue contains exactly the nodes at distance d.*

Lengths on Edges

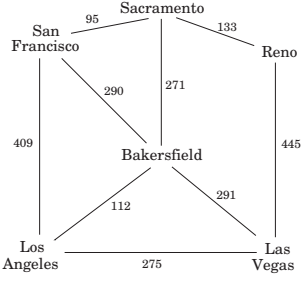
BFS treats all edges as having the same length.

It is rarely true in applications where shortest paths are to be found.

Every edge $e \in E$ with a length l_e.

If $e = (u, v)$, we will sometimes also write

$$l(u, v) \qquad \text{or} \qquad l_{uv}$$



Dijkstra's Algorithm

An Adaption of Breadth-First Search

BFS finds shortest paths in any graph whose edges have unit length.

Q: Can we adapt it to a more general graph $G = (V, E)$ whose edge lengths l_e are positive integers?

A simple trick: For any edge $e = (u, v)$ of E, replace it by l_e edges of length 1, by adding $l_e - 1$ dummy nodes between u and v. It might take time

$$O(|V| + \sum_{e \in E} l_e)$$

It is bad in case we have edges with high length.

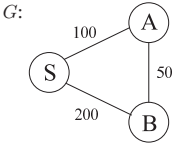
Alarm Clocks

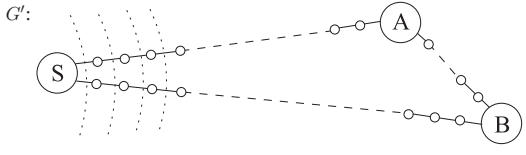
Set an alarm clock for node s at time 0.

Repeat until there are no more alarms:

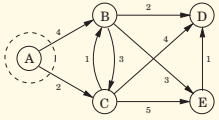
The next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:
  - If there is no alarm yet for v, set one for time $T + l(u, v)$.
  - If v's alarm is set for later than $T + l(u, v)$, then reset it to this earlier time.

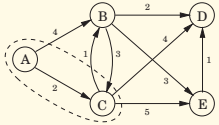




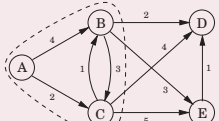
An Example



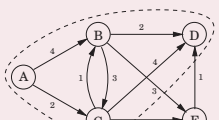
A: 0	D: ∞
B: 4	E: ∞
C: 2	



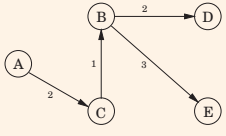
A: 0	D: 6
B: 3	E: 7
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	



Dijkstra's Shortest-Path Algorithm

DIJKSTRA(G, l, s)
input : Graph $G = (V, E)$, directed or undirected; positive edge length $\{l_e \mid e \in E\}$; Vertex $s \in V$
output: For all vertices u reachable from s, $dist(u)$ is the set to the distance from s to u

for *all* $u \in V$ **do**
  $dist(u) = \infty$; $prev(u) = nil$;
end
$dist(s) = 0$;
H =makequeue (V) \\ *using dist-values as keys*;
while H *is not empty* **do**
  u=deletemin (H);
  for *all edge* $(u, v) \in E$ **do**
    if $dist(v) > dist(u) + l(u, v)$ **then**
      $dist(v) = dist(u) + l(u, v)$; $prev(v) = u$;
      decreasekey (H,v);
    end
  end
end

Data Structure Trailer: Priority Queue

Priority queue is a data structure usually implemented by heap.

- Insert: Add a new element to the set.
- Decrease-key: Accommodate the decrease in key value of a particular element.
- Delete-min: Return the element with the smallest key, and remove it from the set.
- Make-queue: Build a priority queue out of the given elements, with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

The first two let us set alarms, and the third tells us which alarm is next to go off.

Running Time

Since `makequeue` takes at most as long as $|V|$ insert operations, we get a total of $|V|$ `deletemin` and $|V| + |E|$ `insert/decreasekey` operations.

Which Heap is Best

SHANGHAI JIAO TONG UNIVERSITY

Implementation	deletemin	insert/decreasekey	$\lvert V\rvert \times$ deletemin$+(\lvert V\rvert+\lvert E\rvert)\times$ insert
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert+\lvert E\rvert)\log \lvert V\rvert)$
d-ary heap	$O(\frac{d \log \lvert V\rvert}{\log d})$	$O(\frac{\log \lvert V\rvert}{\log d})$	$O(\frac{(d\lvert V\rvert+\lvert E\rvert)\log \lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V\rvert)$	$O(1)$ (amortized)	$O(\lvert V\rvert \log \lvert V\rvert+\lvert E\rvert)$

Which heap is Best

A naive array implementation gives a respectable time complexity of $O(|V|^2)$, whereas with a binary heap we get $O((|V| + |E|) \log |V|)$. Which is preferable?

This depends on whether the graph is sparse or dense.

- $|E|$ is less than $|V|^2$. If it is $\Omega(|V|^2)$, then clearly the array implementation is the faster.
- On the other hand, the binary heap becomes preferable as soon as $|E|$ dips below $|V|^2 / \log |V|$.
- The d-ary heap is a generalization of the binary heap and leads to a running time that is a function of d. The optimal choice is $d \approx |E|/|V|$;

Proof of Correctness

For each node $u \in S$, where S is the set of vertex with the $dist$ being set.

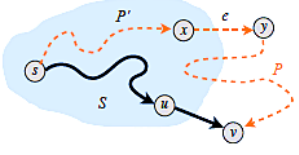
Proof. [by induction on $|S|$]

Base case: $|S| = 1$ is easy since $S = \{s\}$ and $dist[s] = 0$.

Inductive hypothesis: Assume true for $|S| \geq 1$.

- Let v be next node added to S, and let (u, v) be the final edge.
- A shortest $s \rightsquigarrow u$ path plus (u, v) is an $s \rightsquigarrow v$ path of length $\pi(v)$.
- Consider any other $s \rightsquigarrow v$ path P. We show that it is no shorter than $\pi(v)$.
- Let $e = (x, y)$ be the first edge in P that leaves S, and let P' be the subpath from s to x.
- The length of P is already $\geq \pi(v)$ as soon as it reaches y:

$$\begin{aligned} l(P) &\geq l(P') + \ell_e \\ &\geq dist[x] + \ell_e \\ &\geq \pi(y) \geq \pi(v). \end{aligned}$$



Shortest Paths in the Presence of Negative Edges

Negative Edges

Dijkstra's algorithm works because the shortest path from the starting point s to any node v must pass exclusively through nodes that are closer than v.

This no longer holds when edge lengths can be negative.

Q: What needs to be changed in order to accommodate this new complication?

A crucial invariant of Dijkstra's algorithm is that the $dist$ values it maintains are always either overestimates or exactly correct.

They start off at ∞, and the only way they ever change is by updating along an edge:

UPDATE $((u, v) \in E)$

$dist(v) = min\{dist(v), dist(u) + l(u, v)\}$;

Update

UPDATE $((u, v) \in E)$

$dist(v) = min\{dist(v), dist(u) + l(u, v)\};$

This UPDATE operation expresses that the distance to v cannot possibly be more than the distance to u, plus $l(u, v)$. It has the following properties,

1. It gives the correct distance to v in the particular case where u is the second-last node in the shortest path to v, and $dist(u)$ is correctly set.
2. It will never make $dist(v)$ too small, and in this sense it is safe.

Update

UPDATE $((u, v) \in E)$

$dist(v) = min\{dist(v), dist(u) + l(u, v)\}$;

Let

$$s \to u_1 \to u_2 \to u_3 \to \ldots \to u_k \to t$$

be a shortest path from s to t.

This path can have at most $|V| - 1$ edges (why?).

If the sequence of updates performed includes $(s, u_1), (u_1, u_2), \ldots, (u_k, t)$, in that order, then by rule 1 the distance to t will be correctly computed.

It doesn't matter what other updates occur on these edges, or what happens in the rest of the graph, because updates are safe (by rule 2).

But still, if we don't know all the shortest paths beforehand, how can we be sure to update the right edges in the right order?

We simply update all the edges, $|V| - 1$ times!

Bellman-Ford Algorithm

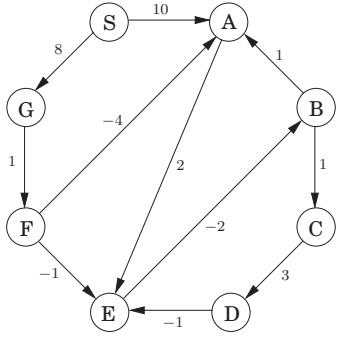
SHANGHAI JIAO TONG UNIVERSITY

SHORTEST-PATHS(G, l, s)
input : Graph $G = (V, E)$, edge length$\{l_e \mid e \in E\}$; Vertex $s \in V$
output: For all vertices u reachable from s, $dist(u)$ is the set to the distance from s to u

for *all* $u \in V$ **do**
  $dist(u) = \infty$;
end
$dist[s] = 0$;
repeat $|V| - 1$ times: **for** $e \in E$ **do**
  UPDATE(e);
end

Running time: $O(|V| \cdot |E|)$

Bellman-Ford Algorithm



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

Negative Cycles

If the graph has a negative cycle, then it doesn't make sense to even ask about shortest path.

Q: How to detect the existence of negative cycles:

Instead of stopping after $|V| - 1$, iterations, perform one extra round.

There is a negative cycle if and only if some $dist$ value is reduced during this final round.

Shortest Paths in DAGs

Graphs without Negative Edges

There are two subclasses of graphs that automatically exclude the possibility of negative cycles:

- graphs without negative edges,
- and graphs without cycles.

We will now see how the single-source shortest-path problem can be solved in just linear time on directed acyclic graphs.

As before, we need to perform a sequence of updates that includes every shortest path as a subsequence.

- In any path of a DAG, the vertices appear in increasing linearized order.

A Shortest-Path Algorithm for DAG

DAG-SHORTEST-PATHS(G, l, s)
input : Graph $G = (V, E)$, edge length$\{l_e \mid e \in E\}$; Vertex $s \in V$
output: For all vertices u reachable from s, $dist(u)$ is the set to the distance from s to u

for *all* $u \in V$ **do**
  $dist(u) = \infty$;
end
$dist[s] = 0$;
linearize G;
for *each* $v \in V$ *in linearized order* **do**
  for *all* $(u, v) \in E$ **do**
    UPDATE((u, v));
  end
end

The scheme does not require edges to be positive.

Even can find longest paths in a DAG by the same algorithm: just negate all edge lengths.

Think About

How about finding simple longest paths in a general directed graph with all positive lengths?

Exercises

Exercises 1

Professor Fake suggests the following algorithm for finding the shortest path from node s to node t in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra's algorithm starting at node s, and return the shortest path found to node t.

Exercises 2

You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between *all pairs of nodes*, with the one restriction that these paths must all pass through v_0.

Algorithm Design VIII

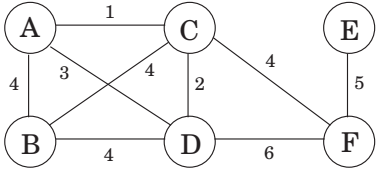
Greedy Algorithms

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Minimum Spanning Trees

Build a Network

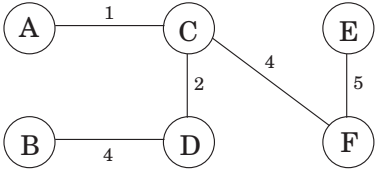


Suppose you are asked to network a collection of computers by linking selected pairs of them.

This translates into a graph problem in which

- nodes are computers,
- undirected edges are potential links, each with a maintenance cost.

Build a Network



The goal is to

- pick enough of these edges that the nodes are connected,
- the total maintenance cost is minimum.

One immediate observation is that the optimal set of edges cannot contain a cycle.

Properties of the Optimal Solutions

Lemma (1)

Removing a cycle edge cannot disconnect a graph.

So the solution must be connected and acyclic: undirected graphs of this kind are called trees.

A tree with minimum total weight, is a minimum spanning tree, MST.

Input: An undirected graph $G = (V, E)$; edge weights w_e

Output: A tree $T = (V, E')$ with $E' \subseteq E$ that minimizes

$$\texttt{weight}(T) = \sum_{e \in E'} w_e$$

Lemma (2)

A tree on n nodes has $n - 1$ edges.

To build the tree one edge at a time, starting from an empty graph.

Each of the n nodes is disconnected from the others, in a connected component by itself.

As edges are added, these components merge. Since each edge unites two different components, exactly $n - 1$ edges are added by the time the tree is fully formed.

When a particular edge (u, v) comes up, we can be sure that u and v lie in separate connected components, for otherwise there would already be a path between them and this edge would create a cycle.

Lemma (3)

Any connected, undirected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree.

It is the converse of Lemma (2). We just need to show that G is acyclic.

While the graph contains a cycle, remove one edge from this cycle.

The process terminates with some graph $G' = (V, E'), E' \subseteq E$, which is acyclic and, by Lemma (1), is also connected.

Therefore G' is a tree, whereupon $|E'| = |V| - 1$ by Lemma (2). So $E' = E$, no edges were removed, and G was acyclic to start with.

Lemma (4)

An undirected graph is a tree if and only if there is a unique path between any pair of nodes.

In a tree, any two nodes can only have one path between them; for if there were two paths, the union of these paths would contain a cycle.

On the other hand, if a graph has a path between any two nodes, then it is connected. If these paths are unique, then the graph is also acyclic.

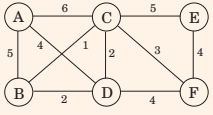
Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule.

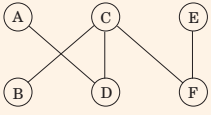
Repeatedly add the next lightest edge that doesn't produce a cycle.

Example

Starting with an empty graph and then attempt to add edges in increasing order of weight

$$B-C; C-D; B-D; C-F; D-F; E-F; A-D; A-B; C-E; A-C$$





The Cut Property

Lemma

Suppose edges X are part of a MST of $G = (V, E)$. Pick any subset of nodes S for which X does not cross between S and $V \backslash S$, and let e be the lightest edge across this partition. Then

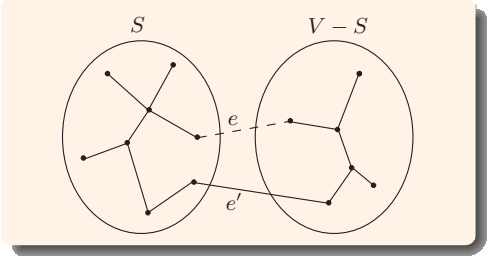
$$X \cup \{e\}$$

is part of some MST.

The Cut Property

A cut is any partition of the vertices into two groups, S and $V \backslash S$.

It is safe to add the lightest edge across any cut, provided X has no edges across the cut.



Proof of the Cut Property

Proof:

Edges X are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove.

So assume e is not in T. We will construct a different MST T' containing $X \cup \{e\}$ by altering T slightly, changing just one of its edges.

Add edge e to T. Since T is connected, it already has a path between the endpoints of e, so adding e creates a cycle.

This cycle must also have some other edge e' across the cut $(S, V \backslash S)$. If we now remove e'

$$T' = T \cup \{e\} \backslash \{e'\}$$

which we will show to be a tree.

T' is connected by Lemma (1), since e' is a cycle edge. And it has the same number of edges as T; so by Lemma (2) and Lemma (3), it is also a tree.

Proof of the Cut Property

Proof:

T' is a minimum spanning tree, since

$$weight(T') = weight(T) + w(e) - w(e')$$

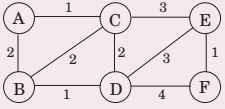
Both e and e' cross between S and $V \backslash S$, and e is the lightest edge of this type. Therefore $w(e) \leq w(e')$, and

$$weight(T') \leq weight(T)$$

Since T is an MST, it must be the case that $weight(T') = weight(T)$ and that T' is also an MST.

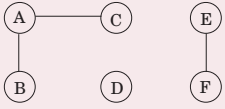
An Example of Cut Property

(a)

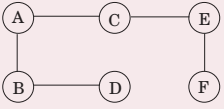


(b)

Edges X:

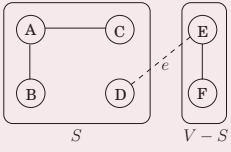


MST T:

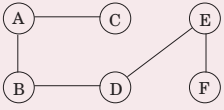


(c)

The cut:



MST T':



Kruskal's Algorithm

KRUSKAL (G, w)
input : A connected undirected graph $G = (V, E)$, with edge weight w_e
output: A minimum spanning tree defined by the edges X

for *all* $u \in V$ **do**
  `makeset` (u);
end
$X = \{\}$;
Sort the edges E by weight;
for *all* $(u, v) \in E$ *in increasing order of weight* **do**
  if `find` *(u)*$\neq$`find` *(v)* **then**
    add (u, v) to X;
    `union` (u,v)
  end
end

Data Structure Retailer: Disjoint Sets

`makeset`(x)	create a singleton set containing x	$\lvert V\rvert$
`find`(x)	find the set that x belong to	$2 \cdot \lvert E\rvert$
`union`(x, y)	merge the sets containing x and y	$\lvert V\rvert - 1$

Prim's Algorithm

A General Kruskal's Algorithm

$X = \{\ \}$;
repeat until $|X| = |V| - 1$;
    pick a set $S \subset V$ for which X has no edges between S and $V - S$;
    let $e \in E$ be the minimum-weight edge between S and $V - S$;
    $X = X \cup \{e\}$;

A popular alternative to Kruskal's algorithm is Prim's, in which the intermediate set of edges X always forms a subtree, and S is chosen to be the set of this tree's vertices.

On each iteration, the subtree defined by X grows by one edge.

The lightest edge between a vertex in S and a vertex outside S. We can equivalently think of S as growing to include the vertex $v \notin S$ of smallest `cost`:

$$\texttt{cost}(v) = \min_{u \in S} w(u, v)$$

The Algorithm

PRIM(G, w)
input : A connected undirected graph $G = (V, E)$, with edge weights w_e
output: A minimum spanning tree defined by the array $prev$

for *all* $u \in V$ **do**
    $cost(u) = \infty$;
    $prev(u) = nil$;
end
pick any initial node u_0;
$cost(u_0) = 0$;
H =makequeue(V) \\ *using cost-values as keys*;
while H *is not empty* **do**
    v=deletemin(H);
    for *each* $(v, z) \in E$ **do**
        if $cost(z) > w(v, z)$ **then**
            $cost(v) = w(v, z)$; $prev(z) = v$;
            decreasekey (H,z);
        end
    end
end

Dijkstra's Algorithm

DIJKSTRA(G, l, s)
input : Graph $G = (V, E)$, directed or undirected; positive edge length $\{l_e \mid e \in E\}$; Vertex $s \in V$
output: For all vertices u reachable from s, $dist(u)$ is the set to the distance from s to u

for *all* $u \in V$ **do**
    $dist(u) = \infty$;
    $prev(u) = nil$;
end
$dist(s) = 0$;
H =makequeue(V) \\ *using dist-values as keys*;
while H *is not empty* **do**
    u=deletemin(H);
    for *all edge* $(u, v) \in E$ **do**
        if $dist(v) > dist(u) + l(u, v)$ **then**
            $dist(v) = dist(u) + l(u, v)$; $prev(v) = u$;
            decreasekey (H,v);
        end
    end
end

Think About

Let C be a cycle with no red edges, and select an uncolored edge of C of max cost and color it red.

Let D be a edge set crossing a cut with no blue edges, and select an uncolored edge in D of min cost and color it blue.

Apply the red and blue rules nondeterministically until all edges are colored.

The blue edges form an MST.

Set Cover

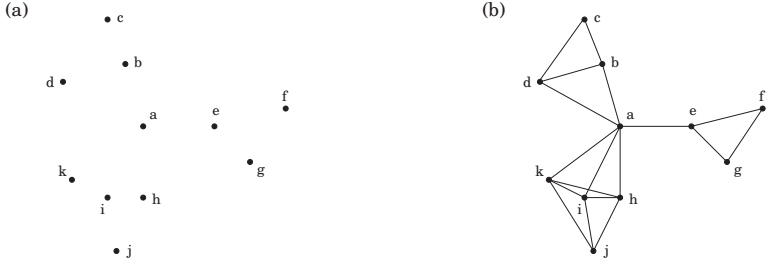
The Problem

A county is in its early stages of planning and is deciding where to put schools.

There are only two constraints:

- each school should be in a town,
- and no one should have to travel more than 30 miles to reach one of them.

Q: What is the minimum number of schools needed?



The Problem

This is a typical (cardinality) set cover problem.

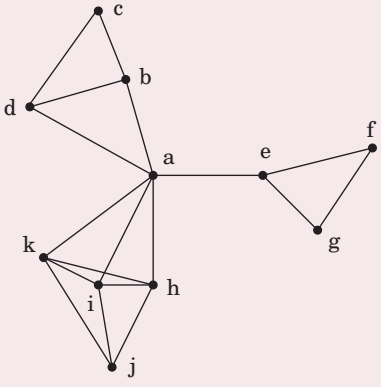
- For each town x, let S_x be the set of towns within 30 miles of it.
- A school at x will essentially "cover" these other towns.
- The question is then, how many sets S_x must be picked in order to cover all the towns in the county?

Set Cover Problem

Set Cover

- Input: A set of elements B, sets $S_1, \ldots, S_m \subseteq B$
- Output: A selection of the S_i whose union is B.
- Cost: Number of sets picked.

The Example



Performance Ratio

Lemma

Suppose B contains n elements and that the optimal cover consists of OPT sets. Then the greedy algorithm will use at most $\ln n \cdot OPT$ sets.

Proof.

Let n_t be the number of elements still not covered after t iterations of the greedy algorithm (so $n_0 = n$).

Since these remaining elements are covered by the optimal OPT sets, there must be some set with at least n_t/OPT of them.

Therefore, the greedy strategy will ensure that

$$n_{t+1} \leq n_t - \frac{n_t}{OPT} = n_t(1 - \frac{1}{OPT})$$

which by repeated application implies

$$n_t \leq n_0(1 - \frac{1}{OPT})^t$$

A more convenient bound can be obtained from the useful inequality

$$1 - x \leq e^{-x} \text{ for all } x$$

with equality if and only if $x = 0$,

Thus

$$n_t \leq n_0(1 - \frac{1}{OPT})^t < n_0(e^{-\frac{1}{OPT}})^t = ne^{-\frac{t}{OPT}}$$

At $t = \ln n \cdot OPT$, therefore, n_t is strictly less than $ne^{-\ln n} = 1$, which means no elements remain to be covered.

Why Greedy Does Not Work: Coin Changing

Coin changing

Goal. Given U. S. currency denominations $\{1, 5, 10, 25, 100\}$, devise a method to pay amount to customer using fewest coins.

Example $\$34$.

Cashier's algorithm. At each iteration, add coin of the largest value that does not take us past the amount to be paid.

Example $\$2.89$.

Cashier's algorithm

CASHIERS-ALGORITHM($x, c_1, c_2, \ldots, c_n$)

SORT n coin denominations so that $0 < c_1 < c_2 < \ldots < c_n$;
$S \leftarrow \emptyset$;
while $x > 0$ **do**
    $k \leftarrow$ largest coin denomination c_k such that $c_k \leq x$;
    if *no such* k **then** RETURN no solution;
    else
        $x \leftarrow x - c_k$;
        $S \leftarrow S \cup \{k\}$;
    end
end
RETURN S;

Quiz

Is the cashier's algorithm optimal?

A Yes, greedy algorithms are always optimal.

B Yes, for any set of coin denominations $c_1 < c_2 < \ldots < c_n$ provided $c_1 = 1$.

C Yes, because of special properties of U.S. coin denominations.

D No.

Cashier's algorithm (for arbitrary coin denominations)

Q. Is cashier's algorithm optimal for any set of denominations?

A. No. Consider U.S. postage: $1, 10, 21, 34, 70, 100, 350, 1225, 1500$.

- Cashier's algorithm: $\$140 = 100 + 34 + 1 + 1 + 1 + 1 + 1$.
- Optimal: $\$140 = 70 + 70$.

A. No. It may not even lead to a feasible solution if $c_1 > 1$: $7, 8, 9$.

- Cashier's algorithm: $\$15 = 9 + ?$.
- Optimal: $\$15 = 7 + 8$.

Properties of any optimal solution (for U.S. coin denominations)

Property. Number of pennies ≤ 4.

Proof. Replace 5 pennies with 1 nickel.

Property. Number of nickels ≤ 1.

Property. Number of quarters ≤ 3.

Property. Number of nickels + number of dimes ≤ 2.

Proof.

- Recall: ≤ 1 nickel.
- Replace 3 dimes and 0 nickels with 1 quarter and 1 nickel;
- Replace 2 dimes and 1 nickel with 1 quarter.

Optimality of cashier's algorithm (for U.S. coin denominations)

Theorem

Cashier's algorithm is optimal for U.S. coins $\{1, 5, 10, 25, 100\}$.

A rather formal proof

Proof. by induction on amount to be paid x

Consider optimal way to change $c_k \leq x \leq c_{k+1}$: greedy takes coin k.

Claim that any optimal solution must take coin k.

- if not, it needs enough coins of type $c_1, \ldots, c_{k-1}$ to add up to x.
- table below indicates no optimal solution can do this

Problem reduces to coin-changing $x - c_k$ cents, which, by induction, is optimally solved by cashier's algorithm.

k	c_k	all optimal solutions must satisfy	max value of $c_1, c_2, \ldots c_{k-1}$ in any optimal solution
1	1	$P \leq 4$	none
2	5	$N \leq 1$	4
3	10	$N + D \leq 2$	$4 + 5 = 9$
4	25	$Q \leq 3$	$20 + 4 = 24$
5	100	no limit	$75 + 24 = 99$

Algorithm Design IX

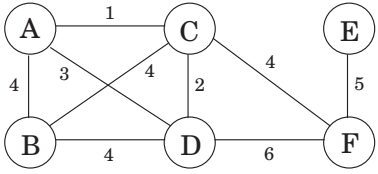
Advanced Data Structures

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Review of Previous Lectures

Build a Network

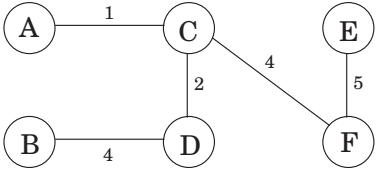


Suppose you are asked to network a collection of computers by linking selected pairs of them.

This translates into a graph problem in which

- nodes are computers,
- undirected edges are potential links, each with a maintenance cost.

Build a Network



The goal is to

- pick enough of these edges that the nodes are connected,
- the total maintenance cost is minimum.

One immediate observation is that the optimal set of edges cannot contain a cycle.

The Prim Algorithm

SHANGHAI JIAO TONG UNIVERSITY

PRIM(G, w)
input : A connected undirected graph $G = (V, E)$, with edge weights w_e
output: A minimum spanning tree defined by the array $prev$

for *all* $u \in V$ **do**
  $cost(u) = \infty$; $prev(u) = nil$;
end
pick any initial node u_0; $cost(u_0) = 0$;
H =makequeue(V) \\ *using cost-values as keys*;
while H *is not empty* **do**
  v=deletemin(H);
  for *each* $(v, z) \in E$ **do**
    if $cost(z) > w(v, z)$ **then**
      $cost(v) = w(v, z)$; $prev(z) = v$; decreasekey (H,z);
    end
  end
end

The Kruskal's Algorithm

KRUSKAL (G, w)
input : A connected undirected graph $G = (V, E)$, with edge weight w_e
output: A minimum spanning tree defined by the edges X

for *all* $u \in V$ **do**
  makeset (u);
end
$X = \{\}$;
Sort the edges E by weight;
for *all* $(u, v) \in E$ *in increasing order of weight* **do**
  if find $(u) \neq$ find (v) **then**
    add (u, v) to X;
    union (u,v)
  end
end

Priority Queue

Priority Queue

Priority queue is a data structure usually implemented by heap.

- Insert: Add a new element to the set.
- Decrease-key: Accommodate the decrease in key value of a particular element.
- Delete-min: Return the element with the smallest key, and remove it from the set.
- Make-queue: Build a priority queue out of the given elements, with the given key values. (In many implementations, this is significantly faster than inserting the elements one by one.)

The Prim Algorithm

SHANGHAI JIAO TONG UNIVERSITY

PRIM(G, w)
input : A connected undirected graph $G = (V, E)$, with edge weights w_e
output: A minimum spanning tree defined by the array $prev$

for *all* $u \in V$ **do**
  $cost(u) = \infty$; $prev(u) = nil$;
end
pick any initial node u_0; $cost(u_0) = 0$;
H =makequeue(V) \\ *using cost-values as keys*;
while H *is not empty* **do**
  v=deletemin(H);
  for *each* $(v, z) \in E$ **do**
    if $cost(z) > w(v, z)$ **then**
      $cost(v) = w(v, z)$; $prev(z) = v$; decreasekey (H,z);
    end
  end
end

Which Heap is Best

SHANGHAI JIAO TONG UNIVERSITY

Implementation	deletemin	insert/decreasekey	$\lvert V\rvert \times$ deletemin$+(\lvert V\rvert+\lvert E\rvert)\times$ insert
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert+\lvert E\rvert)\log \lvert V\rvert)$
d-ary heap	$O(\frac{d \log \lvert V\rvert}{\log d})$	$O(\frac{\log \lvert V\rvert}{\log d})$	$O(\frac{(d\lvert V\rvert+\lvert E\rvert)\log \lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V\rvert)$	$O(1)$ (amortized)	$O(\lvert V\rvert \log \lvert V\rvert+\lvert E\rvert)$

Which heap is Best

A naive array implementation gives a respectable time complexity of $O(|V|^2)$, whereas with a binary heap we get $O((|V| + |E|) \log |V|)$. Which is preferable?

This depends on whether the graph is sparse or dense.

- $|E|$ is less than $|V|^2$. If it is $\Omega(|V|^2)$, then clearly the array implementation is the faster.
- On the other hand, the binary heap becomes preferable as soon as $|E|$ dips below $|V|^2 / \log |V|$.
- The d-ary heap is a generalization of the binary heap and leads to a running time that is a function of d. The optimal choice is $d \approx |E|/|V|$;

Array

The simplest implementation is as an unordered array of key values for all potential elements.

An insert or `decreasekey` is fast, because it just involves adjusting a key value, an $O(1)$ operation.

To `deletemin`, on the other hand, requires a linear-time scan of the list.

Elements are stored in a complete binary tree.

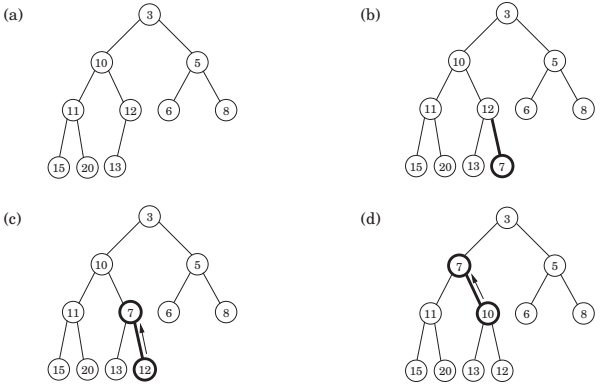
In addition, The key value of any node of the tree is less than or equal to that of its children.

In particular, therefore, the root always contains the smallest element.

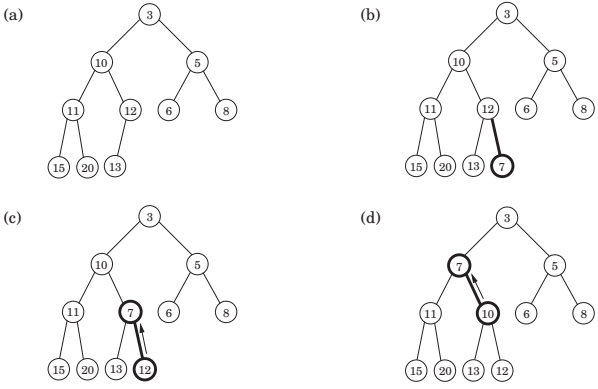
Binary Heap

To insert, place the new element at the bottom of the tree (in the first available position), and let it "bubble up".

The number of swaps is at most the height of the tree $\lfloor \log_2 n \rfloor$, when there are n elements.

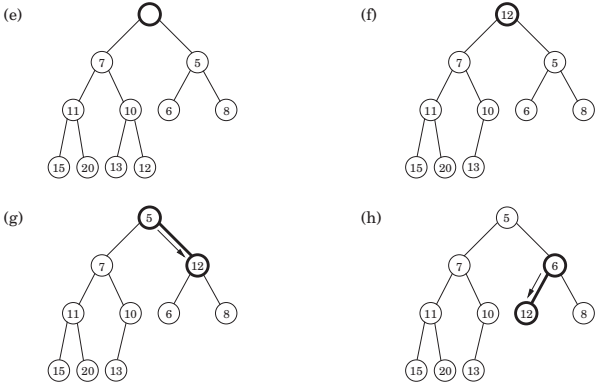


A `decreasekey` is similar, except the element is already in the tree, so we let it bubble up from its current position.



To `deletemin`, return the root value, and remove it from the heap, take the last node in the tree (in the rightmost position in the bottom row) and place it at the root.

Then let it "shift down". Again this takes $O(\log n)$ time.



The regularity of a complete binary tree makes it easy to represent using an array.

The tree nodes have a natural ordering: row by row, starting at the root and moving left to right within each row.

If there are n nodes, this ordering specifies their positions $1, 2, \ldots, n$ within the array.

Moving up and down the tree is easily simulated on the array, using the fact that node number j has parent $\lfloor j/2 \rfloor$ and children $2j$ and $2j + 1$.

d-ary heap

A d-ary heap is identical to a binary heap, except that nodes have d children.

This reduces the height of a tree with n elements to

$$\Theta(\log_d n) = \Theta((\log n)/(\log d))$$

`Inserts` are therefore speeded up by a factor of $\Theta(\log d)$.

`Deletemin` operations, however, take a little longer, namely $O(d \log_d n)$.

A Fibonacci heap is a collection of min-heap-ordered trees. Trees within Fibonacci heaps are rooted but unordered linked by a circular, doubly linked list.

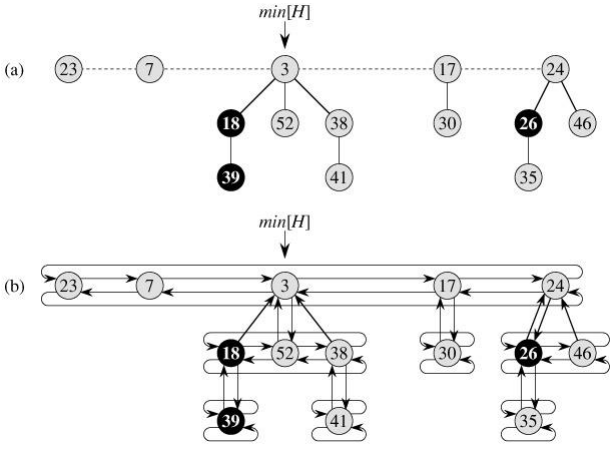
Each node x contains a pointer $p[x]$ to its parent and a pointer $child[x]$ to any one of its children.

The children of x are linked together in a circular, doubly linked list, which we call the child list of x.

Each child y in a child list has pointers $left[y]$ and $right[y]$ that point to y's left and right siblings, respectively.

If node y is an only child, then $left[y] = right[y] = y$. The order in which siblings appear in a child list is arbitrary.

Fibonacci Heaps



Priority Queue Review

Implementation	deletemin	insert/decreasekey	$\lvert V\rvert \times$ `deletemin` $+(\lvert V\rvert+\lvert E\rvert)\times$ `insert`
Array	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert^2)$
Binary heap	$O(\log \lvert V\rvert)$	$O(\log \lvert V\rvert)$	$O((\lvert V\rvert+\lvert E\rvert)\log \lvert V\rvert)$
d-ary heap	$O(\frac{d \log \lvert V\rvert}{\log d})$	$O(\frac{\log \lvert V\rvert}{\log d})$	$O(\frac{(d\lvert V\rvert+\lvert E\rvert)\log \lvert V\rvert}{\log d})$
Fibonacci heap	$O(\log \lvert V\rvert)$	$O(1)$ (amortized)	$O(\lvert V\rvert \log \lvert V\rvert+\lvert E\rvert)$

Disjoint Set

The Kruskal's Algorithm

SHANGHAI JIAO TONG UNIVERSITY

`KRUSKAL` (G, w)
input : A connected undirected graph $G = (V, E)$, with edge weight w_e
output: A minimum spanning tree defined by the edges X

for *all* $u \in V$ **do**
  `makeset` (u);
end
$X = \{\}$;
Sort the edges E by weight;
for *all* $(u, v) \in E$ *in increasing order of weight* **do**
  if `find` *(u)*$\neq$`find` *(v)* **then**
    add (u, v) to X;
    `union` (u,v)
  end
end

Disjoint Sets

`makeset`(x)	create a singleton set containing x	$\|V\|$
`find`(x)	find the set that x belong to	$2 \cdot \|E\|$
`union`(x, y)	merge the sets containing x and y	$\|V\| - 1$

Union by rank

We store a set is by a directed tree. Nodes of the tree are elements of the set, arranged in no particular order, and each has parent pointers that eventually lead up to the root of the tree.

This root element is a convenient representative, or name, for the set. It is distinguished from the other elements by the fact that its parent pointer is a self-loop.

In addition to a parent pointer π, each node also has a rank that, for the time being, should be interpreted as the height of the subtree hanging from that node.

Union by Rank


```
MAKESET(x)
π(x) = x;
rank (x)=0;
```

```
FIND(x)
while x ≠ π(x) do
    x = π(x);
end
return(x);
```

MAKESET is a constant-time operation.

FIND follows parent pointers to the root of the tree and therefore takes time proportional to the height of the tree.

The tree actually gets built via the third operation, UNION, and so we must make sure that this procedure keeps trees shallow.

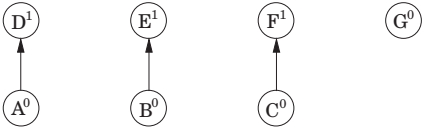
Union


```
UNION(x, y)
r_x=find(x);
r_y=find(y);
if r_x = r_y then return;
if rank(r_x)>rank(r_y) then
    π(r_y) = r_x;
end
else
    π(r_x) = r_y;
    if rank(r_x)=rank(r_y) then rank(r_y)=rank(r_y)+1;
end
```

An Example

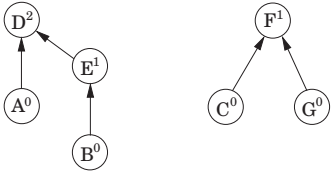
After `makeset`(A), `makeset`(B), $\ldots$, `makeset`(G):

After `union`(A, D), `union`(B, E), `union`(C, F):

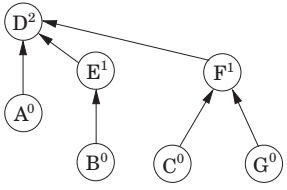


An Example

After $\texttt{union}(C, G), \texttt{union}(E, A)$:



After $\texttt{union}(B, G)$:



Lemma (1)

For any non-root x, $\texttt{rank}(x) < \texttt{rank}(\pi(x))$.

Proof Sketch:

By design, the rank of a node is exactly the height of the subtree rooted at that node. This means, for instance, that as you move up a path toward a root node, the rank values along the way are strictly increasing.

Lemma (2)

Any root node of `rank` k has least 2^k nodes in its tree.

Proof Sketch:

A root node with rank k is created by the merger of two trees with roots of rank $k-1$. By induction to get the results.

Lemma (3)

If there are n elements overall, there can be at most $n/2^k$ nodes of `rank` k.

Proof Sketch:

A node of rank k has at least 2^k descendants.

Any internal node was once a root, and neither its rank nor its set of descendants has changed since then.

Different rank-k nodes cannot have common descendants. Any element has at most one ancestor of rank k.

The Analysis

With the data structure as presented so far, the total time for Kruskal's algorithm becomes

- $O(|E| \log |V|)$ for sorting the edges ($\log |V| \approx \log |E|$),
- $O(|E| \log |V|)$ for the `union` and `find` operations that dominate the rest of the algorithm.

But what if the edges are given to us sorted? Or if the weights are small (say, $O(|E|)$) so that sorting can be done in linear time?

THEN THE DATA STRUCTURE PART BECOMES THE BOTTLENECK!

The main question:

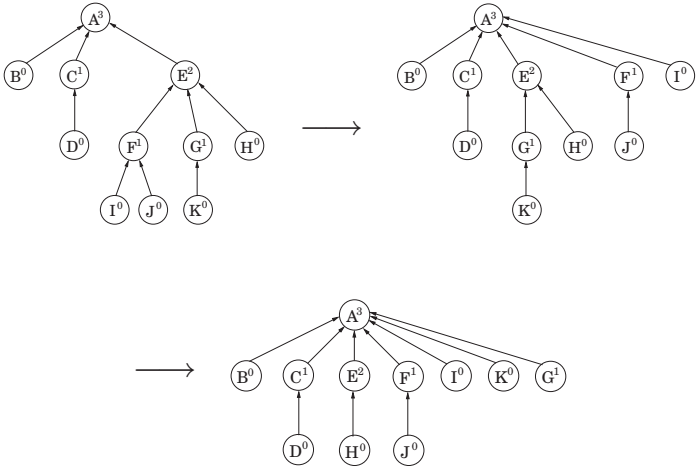
How can we perform `union`'s and `find`'s faster than $\log n$?

Path Compression

FIND(x)

if $x \neq \pi(x)$ **then** $\pi(x)$=FIND($\pi(x)$);
return($\pi(x)$);

Path Compression



Path Compression

The benefit of this simple alteration is long-term rather than instantaneous and thus necessitates a particular kind of analysis.

We need to look at sequences of `find` and `union` operations, starting from an empty data structure, and determine the average time per operation.

This amortized cost turns out to be just barely more than $O(1)$, down from the earlier $O(\log n)$.

Time Analysis

Think of the data structure as having a “top level” consisting of the root nodes, and below it, the insides of the trees.

There is a division of labor:

- `find` operations (with or without path compression) only touch the insides of trees,
- `union`’s only look at the top level.

Thus path compression has no effect on `union` operations and leaves the top level unchanged.

We know that the ranks of root nodes are unaltered, but what about non-root nodes?

The key point is that once a node ceases to be a root, it never resurfaces, and its rank is forever fixed.

Therefore the ranks of all nodes are unchanged by path compression, even though these numbers can no longer be interpreted as tree heights.

In particular,

- For any x that is not a root, $rank(x) < rank(\pi(x))$
- Any root node of `rank` k has least 2^k nodes in its tree.
- If there are n elements overall, there can be at most $n/2^k$ nodes of `rank` k.

If there are n elements, their rank values can range from 0 to $\log n$.

Divide the nonzero part of this range into the following intervals:

$$\{1\}, \{2\}, \{3, 4\}, \{5, 6, \ldots, 16\}, \{17, 18, \ldots, 2^{16} = 65536\},$$
$$\{65537, 65538, ..., 2^{65536}\}, \ldots$$

Each group is of the form $\{k + 1, k + 2, \ldots, 2^k\}$ where k is a power of 2.

The number of groups is $\log^* n$, which is defined to be the number of successive $\log$ operations that need to be applied to n to bring it down to 1 (or below 1).

- For instance, $\log^* 1000 = 4$ since $\log \log \log \log 1000 \leq 1$.
- In practice there will just be the first five of the intervals shown; more are needed only if $n > 2^{65536}$, in other words never.

Time Analysis

In a sequence of `find` operations, some may take longer than others.

We'll bound the overall running time using some creative accounting.

We will give each node a certain amount of pocket money, such that the total money doled out is at most $n \log^* n$ dollars.

We will then show that each `find` takes $O(\log^* n)$ steps, plus some additional amount of time that can be paid for using the pocket money of the nodes involved - one dollar per unit of time.

Thus the overall time for m `find`'s is $O(m \log^* n)$ plus at most $O(n \log^* n)$.

A node receives its allowance as soon as it ceases to be a root, at which point its rank is fixed.

If this rank lies in the interval $\{k+1, \ldots, 2^k\}$, the node receives 2^k dollars.

By Lemma 3, the number of nodes with $\texttt{rank} > k$ is bounded by

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \ldots \leq \frac{n}{2^k}$$

Therefore the total money given to nodes in this particular interval is at most n dollars, and since there are $\log^* n$ intervals, the total money disbursed to all nodes is $n \log^* n$.

Now, the time taken by a specific `find` is simply the number of pointers followed.

Consider the ascending rank values along this chain of nodes up to the root.

Nodes x on the chain fall into two categories:

- either the rank of $\pi(x)$ is in a higher interval than the rank of x,
- or else it lies in the same interval.

There are at most $\log^* n$ nodes of the first type, so the work done on them takes $O(\log^* n)$ time.

The remaining nodes - whose parents' ranks are in the same interval as theirs - have to pay a dollar out of their pocket money for their processing time.

Time Analysis

This only works if the initial allowance of each node x is enough to cover all of its payments in the sequence of find operations.

Here's the crucial observation: each time x pays a dollar, its parent changes to one of higher rank.

Therefore, if x's rank lies in the interval $\{k+1, \ldots, 2^k\}$, it has to pay at most 2^k dollars before its parent's rank is in a higher interval; whereupon it never has to pay again.

Algorithm Design X

Dynamic Programming I

Guoqiang Li
School of Software

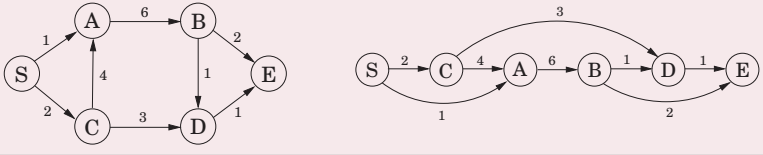
Shanghai Jiao Tong University

Shortest Paths in DAGs, Revisited

Shortest Paths in Dags, Revisited

The special distinguishing feature of a DAG is that its nodes can be linearized.

If compute `dist` values in the left-to-right order, by the time get to a node v, we already have all the information to compute `dist`(v).



Shortest Paths in Dags, Revisited

Initialize all $dist(.)$ value to ∞;
$dist(s)$=0;
for *each* $v \in V \backslash \{s\}$, *in linearized order* **do**
  $dist(v) = min_{(u,v) \in E}\{dist(u) + l(u,v)\}$;
end

This algorithm is solving a collection of subproblems, $\{dist(u) \mid u \in V\}$

This algorithm is solving a collection of subproblems, $\{\texttt{dist}(u) : u \in V\}$.

Start with the smallest of them, $\texttt{dist}(s)$.Then proceed with progressively "larger" subproblems, where a subproblem is large if a lot of other subproblems is solved before get to it.

Dynamic programming is a powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling smallest first, using the answers to small problems to figure out larger ones, until the whole lot of them is solved.

In dynamic programming we are not given a DAG; the DAG is implicit.

Longest Increasing Subsequences

The Problem

The input of the longest increasing subsequence problem is a sequence of numbers $a_1, \ldots, a_n$.

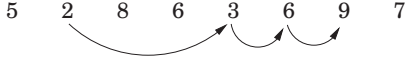
A subsequence is any subset of these numbers taken in order, of the form

$$a_{i_1}, a_{i_2}, \ldots, a_{i_k}$$

where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$.

An increasing subsequence is one in which the numbers are getting strictly larger.

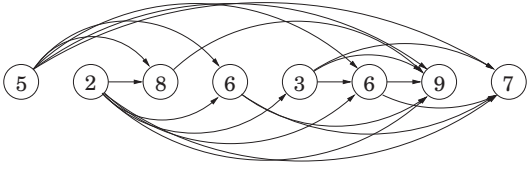
The task is to find the increasing subsequence of greatest length.



Graph Reformulation

Create a graph of all permissible transitions:

- a node i for each element a_i,
- a directed edge (i, j) if possible for a_i and a_j to be consecutive elements in an increasing subsequence: $i < j$ and $a_i < a_j$.



This graph $G = (V, E)$ is a DAG, since all edges (i, j) have $i < j$.

There is a one-to-one correspondence between increasing subsequences and paths in this DAG.

Therefore, the goal is to find the longest path in the DAG!

for $j = 1$ *to* n **do**
  $L(j) = 1 + \max\{L(i) \mid (i, j) \in E\}$;
end
`return(`$\max_j L(j)$`);`

$L(j)$ is the length of the longest path - the longest increasing subsequence - ending at j (plus 1).

If there are no edges into j , we take zero.

The final answer is the largest $L(j)$, since any ending position is allowed.

Q: How long does this step take?

It requires the predecessors of j to be known; for this the adjacency list of the reverse graph G^R, constructible in linear time is handy.

The computation of $L(j)$ then takes time proportional to the indegree of j, giving an overall running time linear in $|E|$.

This is at most $O(n^2)$, the maximum being when the input array is sorted in increasing order.

This Is Dynamic Programming

In order to solve our original problem, we have defined a collection of subproblems $\{L(j) \mid 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

In our case, each subproblem is solved using the relation

$$L(j) = 1 + \max\{L(i) | (i, j) \in E\}$$

Why Not Recursion

The formula for $L(j)$ also suggests recursive algorithm.

Recursion is a very bad idea: the resulting procedure would require exponential time!

- e.g. (i, j) for all $i < j$. The formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

Why did recursion work so well with divide-and-conquer? In divide-and-conquer, a problem is expressed in terms of subproblems that are substantially smaller, say, half the size.

In a dynamic programming, a problem is reduced to subproblems that are slightly smaller. Thus the full recursion tree has polynomial depth and an exponential number of nodes.

Edit Distance

The problem

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by.

Q: What is the appropriate notion of closeness in this case?

A natural measure of the distance between two strings is the extent to which they can be aligned, or matched up.

Technically, an alignment is simply a way of writing the strings one above the other.

S	–	N	O	W		Y
S	U	N	N	–		Y

Cost: 3

–	S	N	O	W	–	Y
S	U	N	–	–	N	Y

Cost: 5

The problem

S	–	N	O	W		Y
S	U	N	N	–		Y

Cost: 3

–	S	N	O	W	–	Y
S	U	N	–	–	N	Y

Cost: 5

The cost of an alignment is the number of columns in which the letters differ.

The edit distance between two strings is the cost of their best possible alignment.

Edit distance is so named because it can also be thought of as the minimum number of edits.

The number of insertions, deletions, and substitutions of characters needed to transform the first string into the second.

A Dynamic Programming Solution

What are the subproblems?

The goal is to find the edit distance between two strings, $x[1, \ldots, m]$ and $y[1, \ldots, n]$.

For every i, j with $1 \leq i \leq m$ and $1 \leq j \leq n$, let $E(i, j)$: the edit distance between some prefix of the first string, $x[1, \ldots, i]$, and some prefix of the second, $y[1, \ldots, j]$.

$$E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \texttt{diff}(i, j) + E(i-1, j-1)\}$$

where $\texttt{diff}(i, j)$ is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

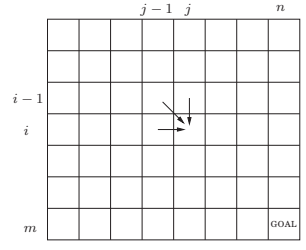
An Example

Edit distance between EXPONENTIAL and POLYNOMIAL, subproblem $E(4,3)$ corresponds to the prefixes EXPO and POL. The rightmost column of their best alignment must be one of the following:

$$\begin{matrix}\mathtt{O}\\ \mathtt{-}\end{matrix} \quad \text{or} \quad \begin{matrix}\mathtt{-}\\ \mathtt{L}\end{matrix} \quad \text{or} \quad \begin{matrix}\mathtt{O}\\ \mathtt{L}\end{matrix}$$

Thus, $E(4,3) = \min\{1 + E(3,3), 1 + E(4,2); 1 + E(3,2)\}$.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

The Algorithm

for $i = 0$ *to* m **do**
  $E(i, 0) = i$;
end
for $j = 1$ *to* n **do**
  $E(0, j) = j$;
end
for $i = 1$ *to* m **do**
  for $j = 1$ *to* m **do**
    $E(i, j) = \min\{1 + E(i-1, j), 1 + E(i, j-1), \texttt{diff}(i, j) + E(i-1, j-1)\}$;
  end
end
`return(`$E(m, n)$`);`

The over running time is $O(m \cdot n)$.

Subproblems of Dynamic Programming

Finding the right subproblem takes creativity and experimentation.

There are a few standard choices that seem to arise repeatedly in dynamic programming.

- The input is $x_1, x_2, \ldots x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.The number of subproblems is therefore linear.
- The input is $x_1, \ldots x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ldots, x_i$ and $y_1, \ldots y_j$. The number of subproblems is $O(mn)$.
- The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$. The number of subproblems is $O(n^2)$.
- The input is a rooted tree. A subproblem is a rooted subtree.

Knapsack

The Problem

In a robbery, a burglar finds much more loot than he had expected and has to decide what to take.

His bag (or "knapsack") will hold a total weight of at most W pounds.

There are n items to pick from, of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$.

Q: What's the most valuable combination of items he can put into his bag?

There are two versions of this problem:

- there are unlimited quantities of each item available;
- there is one of each item.

Example: Knapsack with Repetition

Take $W = 10$, and

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

and there are unlimited quantities of each item available.

Knapsack with Repetition

For every $w \leq W$ let

$$K(w) = \text{ maximum value achievable with a knapsack of capacity } w$$

We express this in terms of smaller subproblems:

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

$K(0) = 0$;
for $w = 1$ *to* W **do**
  $K(w) = \max_{i:w_i \leq w}\{K(w - w_i) + v_i\}$;
end
`return` $(K(W))$;

The over running time is $O(n \cdot W)$.

Example: Knapsack without Repetition

Take $W = 9$, and

Item	Weight	Value
1	2	\$3
2	3	\$4
3	4	\$5
4	5	\$7

and there is only one of each item available.

Knapsack without Repetition

For every $w \leq W$ and $0 \leq j \leq n$, let

$$K(w, j) = \text{ maximum value achievable with a knapsack of capacity } w \text{ and items } 1, \ldots, j$$

We express this in terms of smaller subproblems:

$$K(w, j) = \max\{K(w - w_j, j - 1) + v_j, K(w, j - 1)\}$$

Knapsack without Repetition

Initialize all $K(0, j) = 0$ and all $K(w, 0) = 0$;
for $j = 1$ *to* n **do**
  for $w = 1$ *to* n **do**
    if $w_j > w$ **then** $K(w, j) = K(w, j-1)$;
    $K(w, j) = \max\{K(w - w_j, j-1) + v_j, K(w, j-1)\}$;
  end
end
`return` $(K(W, n))$;

The over running time is $O(n \cdot W)$.

Longest Common Subsequence

Problem. Given two strings $x_1 x_2 \dots x_m$ and $y_1 y_2 \dots y_n$, find a common subsequence that is as long as possible.

Alternative viewpoint. Delete some characters from x; delete some character from y; a common subsequence if it results in the same string.

Example. **LCS(GGCACCACG, ACGGCGGATACG) = GGCAACG**.

Solution for LCS

$$LCS[i,j] = \begin{cases} 0, & \text{if } i = 0 \vee j = 0 \\ LCS[i-1, j-1] + 1, & \text{if } a[i] == b[j] \\ \max\{LCS[i-1, j], LCS[i, j-1]\}, & \text{otherwise.} \end{cases}$$

How about the longest common substring?

Solution for LCS(substring)

$$LCS[i,j] = \begin{cases} 0, & \text{if } i = 0 \vee j = 0 \\ LCS[i-1, j-1] + 1, & \text{if } a[i] == b[j] \\ 0, & \text{otherwise.} \end{cases}$$

Chain Matrix Multiplication

The Problem

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes $m \cdot n \cdot p$ multiplications.

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	$60,200$
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	$7,000$

Q: How do we determine the optimal order, if we want to compute $A_1 \times A_2 \times \ldots \times A_n$, where the A_i's are matrices with dimensions $m_0 \times m_1, m_1 \times m_2, \ldots, m_{n-1} \times m_n$, respectively?

Binary Tree

A particular parenthesization can be represented by a binary tree in which

- the individual matrices correspond to the leaves,
- the root is the final product, and
- interior nodes are intermediate products.

The possible orders in which to do the multiplication correspond to the various full binary trees with n leaves.

Subproblems

For $1 \leq i \leq j \leq n$, let

$$C(i,j) = \text{ minimum cost of multiplying } A_i \times A_{i+1} \times \ldots \times A_j$$

$$C(i,j) = \min_{i \leq k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

The Program

for $i = 1$ *to* n **do**
  $C(i, i) = 0$;
end
for $s = 1$ *to* $n - 1$ **do**
  for $i = 1$ *to* $n - s$ **do**
    $j = i + s$;
    $C(i, j) = \min_{i \le k < j}\{C(i, k) + C(k + 1, j) + m_{i-1} \cdot m_k \cdot m_j\}$;
  end
end
`return` $(C(1, n))$;

The over running time is $O(n^3)$.

The Example

Suppose that we want to multiply four matrices, A, B, C, D, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively.

Algorithm Design XI

Dynamic Programming II

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Past Exam

2. （15 分）考虑**三分问题（3-Partition problem）**：给定整数集合$\{a_1, a_2, \ldots, a_n\}$，判断是否可以将其分割为三个相互不相交的子集$I, J, K$，使得

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_i = \frac{1}{3} \sum_{i=1}^{n} a_i$$

6. (20 分)在一条河上有一座独木桥，长度为 L，上面分布着一些石子，为了简单起见，我们假设桥为 *0-L* 的一段线段，而石子都分布在整数坐标上，也就是有一个函数 *stone(x)*，表示在坐标 x 上是否有石子，比如 *stone(0)= 1* 表示在桥头有一个石子，*stone(2) = 0* 表示在坐标为 *2* 的位置没有石子。

现在有一个小朋友站在桥头的位置想要过桥（站在桥尾或者跨过桥尾均为过了桥），但他不想踩到石子，他每跨出一步的步长是*[S,T]*区间中的任何整数（包括 S 和 T）。设计算法求小朋友要过河，必须踩到的最少的石子数。

（1）（15 分）写出动态规划范式，注意边界条件，并详细说明动态规划范式所代表的意思。

（2）（5 分）根据动态规划范式，给出时间空间复杂度分析。

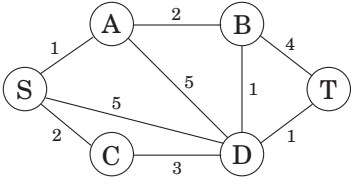
Shortest Reliable Paths

Shortest Reliable Paths

In a network, even if edge lengths faithfully reflect transmission delays, there may be other considerations involved in choosing a path.

For instance, each extra edge in the path might be an extra “hop” fraught with uncertainties and dangers of packet loss.

We would like to avoid paths with too many edges.



Shortest Reliable Paths

Suppose then that we are given a graph G with lengths on the edges, along with two nodes s and t and an integer k, and we want the shortest path from s to t that uses at most k edges.

Dynamic programming will work!

For each vertex v and each integer $i \leq k$, let

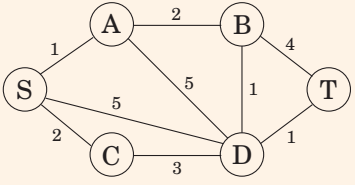
$$dist(v, i) = \text{ the length of the shortest path from } s \text{ to } v \text{ that uses } i \text{ edges}$$

The starting values $dist(v, 0)$ are ∞ for all vertices except s, for which it is 0.

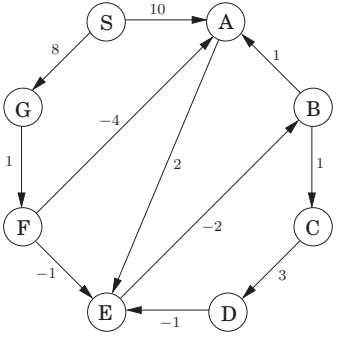
$$dist(v, i) = \min_{(u,v) \in E} \{dist(u, i-1) + l(u, v)\}$$

Shortest Reliable Paths

Find out the shortest reliable path from S to T, when $k = 3$.



Bellman-Ford Algorithm



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

All-Pairs Shortest Path

All-Pairs Shortest Path

What if we want to find the shortest path not just between s and t but between all pairs of vertices?

One approach would be to execute Bellman-Ford-Moore algorithm $|V|$ times, once for each starting node.

The total running time would then be $O(|V|^2|E|)$.

We'll now see a better alternative, the $O(|V|^3)$, named Floyd-Warshall algorithm.

Floyd-Warshall Algorithm

Dynamic programming again!

The Subproblems

Number the vertices in V as $\{1, 2, \ldots, n\}$, and let

$dist(i, j, k) =$ the length of the shortest path from i to j in which only nodes $\{1, 2, \ldots, k\}$ can be used as intermediates.

Initially, $dist(i, j, 0)$ is the length of the direct edge between i and j, if it exists, and is ∞ otherwise.

For $k \geq 1$

$$dist(i, j, k) = \min\{dist(i, j, k-1), dist(i, k, k-1) + dist(k, j, k-1)\}$$

The Program

for $i = 1$ *to* n **do**
  for $j = 1$ *to* n **do**
    $dist(i, j, 0) = \infty$;
  end
end
for *all* $(i, j) \in E$ **do**
  $dist(i, j, 0) = l(i, j)$;
end
for $k = 1$ *to* n **do**
  for $i = 1$ *to* n **do**
    for $j = 1$ *to* n **do**
      $dist(i, j, k) = \min\{dist(i, j, k-1), dist(i, k, k-1) + dist(k, j, k-1)\}$;
    end
  end
end

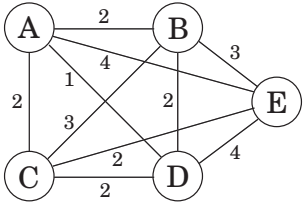
Traveling Salesman Problem

The Traveling Salesman Problem

A traveling salesman is getting ready for a big sales tour. Starting at his hometown, he will conduct a journey in which each of his target cities is visited exactly once before he returns home.

Q: Given the pairwise distances between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

The brute-force approach is to evaluate every possible tour and return the best one. Since there are $(n-1)!$ possibilities, this strategy takes $O(n!)$ time.



The Traveling Salesman Problem

Denote the cities by $1, \ldots, n$, the salesman's hometown being 1, and let $D = (d_{ij})$ be the matrix of intercity distances.

The goal is to design a tour that starts and ends at 1, includes all other cities exactly once, and has minimum total length.

The Subproblems

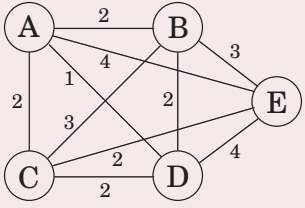
For a subset of cities $S \subseteq \{1, 2, \ldots, n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When $|S| > 1$, we define $C(S, 1) = \infty$.

For $j \neq 1$ with $j \in S$ we have

$$C(S, j) = \min_{i \in S : i \neq j} C(S \backslash \{j\}, i) + d_{ij}$$

Exercise



The Program

$C(1,1) = 0$;
for $s = 2$ *to* n **do**
  for *all subsets* $S \subseteq \{1, 2, \ldots, n\}$ **do**
    $C(S, 1) = \infty$;
    for *all* $j \in S$ *and* $j \neq 1$ **do**
      $C(S, j) = \min_{i \in S: i \neq j} C(S \backslash \{j\}, i) + d_{ij}$;
    end
  end
end
`return` $(\min_j C(\{1, 2, \ldots, n\}, j) + d_{j1})$;

There are at most $2^n \cdot n$ subproblems, and each one takes linear time.

The total running time is therefore $O(n^2 \cdot 2^n)$.

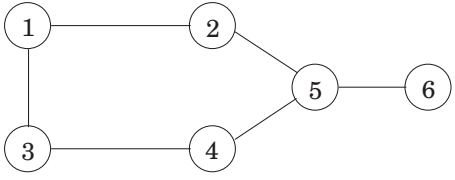
Independent Sets in Trees

The Problem

A subset of nodes $S \subseteq V$ is an independent set of graph $G = (V, E)$ if there are no edges between them.

Finding the largest independent set in a graph is believed to be intractable.

However, when the graph happens to be a tree, the problem can be solved in linear time, using dynamic programming.



The Subproblems

$I(u) =$ size of largest independent set of subtree hanging from u.

$$I(u) = \max\{1 + \sum_{\text{grandchildren } w \text{ of } u} I(w), \sum_{\text{children } w \text{ of } u} I(w)\}$$

Algorithm Design XII

Linear Programming I

Guoqiang Li
School of Software

Shanghai Jiao Tong University

An Introduction to Linear Programming

A linear programming problem gives a set of variables, and assigns real values to them so as to

1. satisfy a set of linear equations and/or linear inequalities involving these variables, and
2. maximize or minimize a given linear objective function.

Example: Profit Maximization

A boutique chocolatier has two products:

- triangular chocolates, called Pyramide,
- and the more decadent and deluxe Pyramide Nuit.

Q: How much of each should it produce to maximize profits?

- Every box of Pyramide has a a profit of $\$1$.
- Every box of Nuit has a profit of $\$6$.
- The daily demand is limited to at most 200 boxes of Pyramide and 300 boxes of Nuit.
- The current workforce can produce a total of at most 400 boxes of chocolate per day.

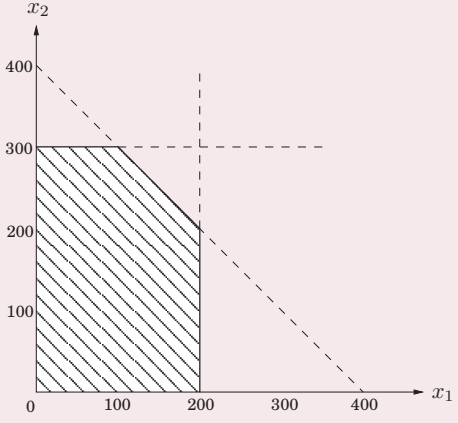
$$
\begin{array}{ll}
\texttt{Objective function} & \max x_1 + 6x_2 \\
\texttt{Constraints} & x_1 \leq 200 \\
& x_2 \leq 300 \\
& x_1 + x_2 \leq 400 \\
& x_1, x_2 \geq 0
\end{array}
$$

A linear equation in x_1 and x_2 defines a line in the two-dimensional (2D) plane, and a linear inequality designates a half-space, the region on one side of the line.

The set of all feasible solutions of this linear program is the intersection of five half-spaces.

It is a convex polygon.

The Convex Polygon



The Optimal Solution

We want to find the point in this polygon at which the objective function is maximized.

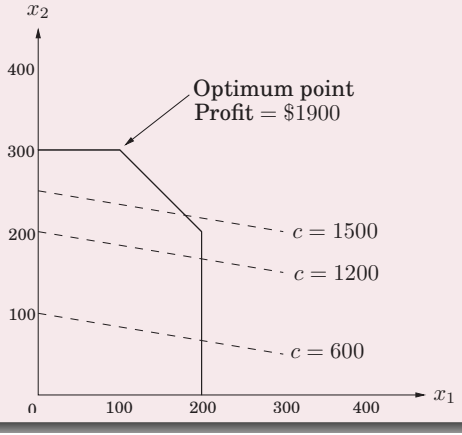
The points with a profit of c dollars lie on the line $x_1 + 6x_2 = c$, which has a slope of $-1/6$.

As c increases, this "profit line" moves parallel to itself, up and to the right.

Since the goal is to maximize c, we must move the line as far up as possible, while still touching the feasible region.

The optimum solution will be the very last feasible point that the profit line sees and must therefore be a vertex of the polygon.

The Convex Polygon



The Optimal Solution

It is a general rule of linear programs that the optimum is achieved at a vertex of the feasible region.

The only exceptions are cases in which there is no optimum; this can happen in two ways:

1. The linear program is infeasible; that is, the constraints are so tight that it is impossible to satisfy all of them.
   - For instance, $x \leq 1$, $x \geq 2$.
2. The constraints are so loose that the feasible region is unbounded, and it is possible to achieve arbitrarily high objective values.
   - For instance, $\max x_1 + x_2$
   - $x_1, x_2 \geq 0$

Solving Linear Programs

Linear programs (LPs) can be solved by the simplex method, devised by George Dantzig in 1947.

This algorithm starts at a vertex, and repeatedly looks for an adjacent vertex of better objective value.

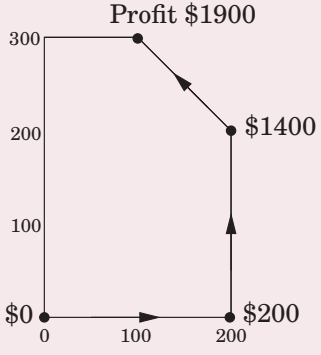
It does hill-climbing on the vertices of the polygon, walking from neighbor to neighbor so as to steadily increase profit along the way.

Upon reaching a vertex that has no better neighbor, simplex declares it to be optimal and halts.

Q: Why does this local test imply global optimality?

By simple geometry. Since all the vertex's neighbors lie below the line, the rest of the feasible polygon must also lie below this line.

The Example



More Products

The chocolatier introduces a third and even more exclusive chocolates, called Pyramide Luxe. One box of these will bring in a profit of $\$13$.

Let x_1, x_2, x_3 denote the number of boxes of each chocolate produced daily, with x_3 referring to Luxe.

The old constraints on x_1 and x_2 persist. The labor restriction now extends to x_3 as well: the sum of all three variables is at most 400.

Nuit and Luxe require the same packaging machinery. Luxe uses it three times as much, which imposes another constraint $x_2 + 3x_3 \leq 600$.

$$\max x_1 + 6x_2 + 13x_3$$
$$x_1 \leq 200$$
$$x_2 \leq 300$$
$$x_1 + x_2 + x_3 \leq 400$$
$$x_2 + 3x_3 \leq 600$$
$$x_1, x_2, x_3 \geq 0$$

The space of solutions is now three-dimensional.

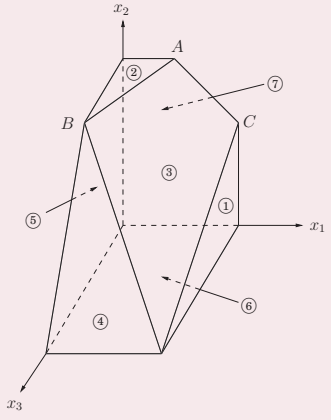
Each linear equation defines a 3D plane, and each inequality a half-space on one side of the plane.

The feasible region is an intersection of seven half-spaces, a polyhedron.

A profit of c corresponds to the plane $x_1 + 6x_2 + 13x_3 = c$.

As c increases, this profit-plane moves parallel to itself, further into the positive orthant until it no longer touches the feasible region.

The Example



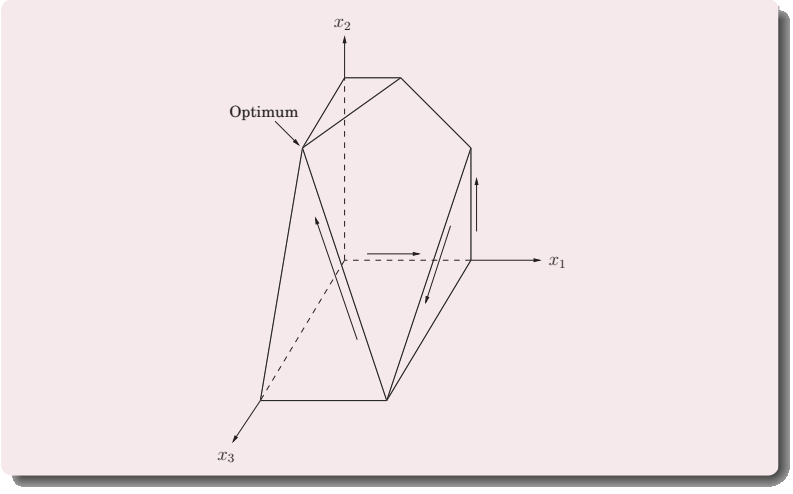
The point of final contact is the optimal vertex: $(0, 300, 100)$, with total profit $\$3100$.

Q: How would the simplex algorithm behave on this modified problem?

A possible trajectory

$$\underset{\$0}{(0,0,0)} \to \underset{\$200}{(200,0,0)} \to \underset{\$1400}{(200,200,0)} \to \underset{\$2800}{(200,0,200)} \to \underset{\$3100}{(0,300,100)}$$

The Example



Integer Linear Programming and Rounding

Example: Production Planning

The company makes handwoven carpets, a product for which the demand is extremely seasonal.

Our analyst has just obtained demand estimates for all months of the next calendar year: $d_1, d_2, \ldots, d_{12}$, ranging from 440 to 920.

Currently with 30 employees, each of whom makes 20 carpets per month and gets a monthly salary of $\$2000$.

With no initial surplus of carpets.

Example: Production Planning

Q: How can we handle the fluctuations in demand? There are three ways:

1. Overtime. Overtime pay is 80% more than regular pay. Workers can put in at most 30% overtime.
2. Hiring and firing, costing $\$320$ and $\$400$, respectively, per worker.
3. Storing surplus production, costing $\$8$ per carpet per month. Currently without stored carpets on hand, and without any carpets stored at the end of year.

w_i	$=$	number of workers during i-th month; $w_0 = 30$.
x_i	$=$	number of carpets made during i-th month.
o_i	$=$	number of carpets made by overtime in month i.
h_i, f_i	$=$	number of workers hired and fired, respectively, at beginning of month i.
s_i	$=$	number of carpets stored at end of month i; $s_0 = 0$.

All variables must be nonnegative:

$$w_i, x_i, o_i, h_i, f_i, s_i \geq 0, i = 1, \ldots, 12$$

The total number of carpets made per month consists of regular production plus overtime:

$$x_i = 20w_i + o_i$$

$i = 1, \ldots, 12$.

The number of workers can potentially change at the start of each month:

$$w_i = w_{i-1} + h_i - f_i$$

The number of carpets stored at the end of each month is what we started with, plus the number we made, minus the demand for the month:

$$s_i = s_{i-1} + x_i - d_i$$

And overtime is limited:

$$o_i \leq 6w_i$$

The objective function is to minimize the total cost:

$$\min 2000 \sum_i w_i + 320 \sum_i h_i + 400 \sum_i f_i + 8 \sum_i s_i + 180 \sum_i o_i$$

The optimum solution might turn out to be fractional; for instance, it might involve hiring 10.6 workers in the month of March.

This number would have to be rounded to either 10 or 11 in order to make sense, and the overall cost would then increase correspondingly.

In the example, most of the variables take on fairly large values, and thus rounding is unlikely to affect things too much.

There are other LPs, in which rounding decisions have to be made very carefully to end up with an integer solution of reasonable quality.

There is a tension in linear programming between the ease of obtaining fractional solutions and the desirability of integer ones.

In NP problems, finding the optimum integer solution of an LP is an important but very hard problem, called integer linear programming.

Duality

Recall:

$$\begin{aligned} \max\, & x_1 + 6x_2 \\ & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1 + x_2 \leq 400 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Simplex declares the optimum solution to be $(x_1, x_2) = (100, 300)$, with objective value 1900.

Can this answer be checked somehow?

We take the first inequality and add it to six times the second inequality:

$$x_1 + 6x_2 \leq 2000$$

Multiplying the three inequalities by 0, 5, and 1, respectively, and adding them up yields

$$x_1 + 6x_2 \leq 1900$$

Multipliers

Let's investigate the issue by describing what we expect of these three multipliers, call them y_1, y_2, y_3.

Multiplier	Inequality				
y_1	x_1			$\leq$	200
y_2			x_2	$\leq$	300
y_3	x_1	$+$	x_2	$\leq$	400

These y_i's must be nonnegative, otherwise they are unqualified to multiply inequalities.

After the multiplication and addition steps, we get the bound:

$$(y_1 + y_3)x_1 + (y_2 + y_3)x_2 \leq 200y_1 + 300y_2 + 400y_3$$

We want the left-hand side to look like the objective function $x_1 + 6x_2$ so that the right-hand side is an upper bound on the optimum solution.

$$x_1 + 6x_2 \leq 200y_1 + 300y_2 + 400y_3$$

if

$$y_1, y_2, y_3 \geq 0$$
$$y_1 + y_3 \geq 1$$
$$y_2 + y_3 \geq 6$$

We can easily find y's that satisfy the inequalities on the right by simply making them large enough, for example $(y_1, y_2, y_3) = (5, 3, 6)$.

These particular multipliers tell us that the optimum solution of the LP is at most

$$200 \cdot 5 + 300 \cdot 3 + 400 \cdot 6 = 4300$$

What we want is a bound as tight as possible, so we minimize

$$200y_1 + 300y_2 + 400y_3$$

subject to the preceding inequalities. This is a new linear program!

$$\begin{aligned} \min\ & 200y_1 + 300y_2 + 400y_3 \\ & y_1 + y_3 \geq 1 \\ & y_2 + y_3 \geq 6 \\ & y_1, y_2, y_3 \geq 0 \end{aligned}$$

Any feasible value of this dual LP is an upper bound on the original primal LP.

If we find a pair of primal and dual feasible values that are equal, then they must both be optimal.

Here is just such a pair:

- Primal: $(x_1, x_2) = (100, 300)$;
- Dual: $(y_1, y_2, y_3) = (0, 5, 1)$.

They both have value 1900 and certify each other's optimality.

Matrix-Vector Form and Its Dual

Primal LP	Dual LP
$\max c^T \mathbf{x}$	$\min \mathbf{y}^T b$
$A\mathbf{x} \leq b$	$\mathbf{y}^T A \geq c^T$
$\mathbf{x} \geq 0$	$\mathbf{y} \geq 0$

Primal LP:

$$\begin{aligned} &\max\ c_1 x_1 + \cdots + c_n x_n \\ &a_{i1} x_1 + \cdots + a_{in} x_n \leq b_i \quad \text{for } i \in I \\ &a_{i1} x_1 + \cdots + a_{in} x_n = b_i \quad \text{for } i \in E \\ &x_j \geq 0 \quad \text{for } j \in N \end{aligned}$$

Dual LP:

$$\begin{aligned} &\min\ b_1 y_1 + \cdots + b_m y_m \\ &a_{1j} y_1 + \cdots + a_{mj} y_m \geq c_j \quad \text{for } j \in N \\ &a_{1j} y_1 + \cdots + a_{mj} y_m = c_j \quad \text{for } j \notin N \\ &y_i \geq 0 \quad \text{for } i \in I \end{aligned}$$

Matrix-Vector Form and Its Dual

$$\begin{aligned} &\max x_1 + 6x_2 \\ &x_1 \le 200 \\ &x_2 \le 300 \\ &x_1 + x_2 \le 400 \\ &x_1, x_2 \ge 0 \end{aligned}$$

$$\begin{aligned} &\min 200y_1 + 300y_2 + 400y_3 \\ &y_1 + y_3 \ge 1 \\ &y_2 + y_3 \ge 6 \\ &y_1, y_2, y_3 \ge 0 \end{aligned}$$

Theorem (Duality)

If a linear program has a bounded optimum, then so does its dual, and the two optimum values coincide.

Complementary Slackness

The number of variables in the dual is equal to that of constraints in the primal and the number of constraints in the dual is equal to that of variables in the primal.

An inequality constraint has slack if the slack variable is positive.

The complementary slackness refers to a relationship between the slackness in a primal constraint and the associated dual variable.

$$\max x_1 + 6x_2$$
$$x_1 \leq 200$$
$$x_2 \leq 300$$
$$x_1 + x_2 \leq 400$$
$$x_1, x_2 \geq 0$$

$$x_1 = 100, x_2 = 300$$

$$\min 200y_1 + 300y_2 + 400y_3$$
$$y_1 + y_3 \geq 1$$
$$y_2 + y_3 \geq 6$$
$$y_1, y_2, y_3 \geq 0$$

$$y_1 = 0, y_2 = 5, y_3 = 1$$

Complementary Slackness

Theorem

Assume LP problem (P) has a solution x^* and its dual problem (D) has a solution y^*.

1. *If $x_j^* > 0$, then the j-th constraint in (D) is binding.*
2. *If the j-th constraint in (D) is not binding, then $x_j^* = 0$.*
3. *If $y_i^* > 0$, then the i-th constraint in (P) is binding.*
4. *If the i-th constraint in (P) is not binding, then $y_i^* = 0$.*

A Concrete Example for Duality

Brewery Problem

Small brewery produces ale and beer.

- Production limited by scarce resources: corn, hops, barley malt.
- Recipes for ale and beer require different proportions of resources.

Beverage	Corn(pounds)	Hops(ounces)	Malt(pounds)	Profit($)
Ale (barrel)	5	4	35	13
Beer (barrel)	15	4	20	23
constraint	480	160	1190	

LP and its Dual

$$\max 13x_1 + 23x_2$$
$$5x_1 + 15x_2 \leq 480$$
$$4x_1 + 4x_2 \leq 160$$
$$35x_1 + 20x_2 \leq 1190$$
$$x_1, x_2 \geq 0$$

$x_1^* = 12, x_2^* = 28$

Brewer: find optimal mix of beer and ale to maximize profits.

$$\min 480y_1 + 160y_2 + 1190y_3$$
$$5y_1 + 4y_2 + 35y_3 \geq 13$$
$$15y_1 + 4y_2 + 20y_3 \geq 23$$
$$y_1, y_2, y_3 \geq 0$$

$y_1^* = 1, y_2^* = 2, y_3^* = 0$

Entrepreneur: buy individual resources from brewer at min cost.

Q. How much should brewer be willing to pay (marginal price) for additional supplies of scarce resources?

A. corn $1, hops $2, malt $0.

Q. Suppose a new product “light beer” is proposed. It requires 2 corn, 5 hops, 24 malt. How much profit must be obtained from light beer to justify diverting resources from production of beer and ale?

A. At least 2 ($1) + 5 ($2) + 24 ($0) = $12 / barrel.

Algorithm Design XIII

Linear Programming II

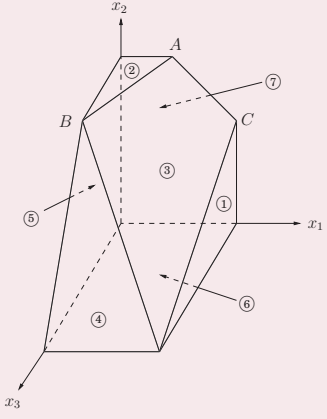
Guoqiang Li
School of Software

Shanghai Jiao Tong University

Review of Previous Lecture

$$\max x_1 + 6x_2 + 13x_3$$
$$x_1 \leq 200$$
$$x_2 \leq 300$$
$$x_1 + x_2 + x_3 \leq 400$$
$$x_2 + 3x_3 \leq 600$$
$$x_1, x_2, x_3 \geq 0$$

The Example



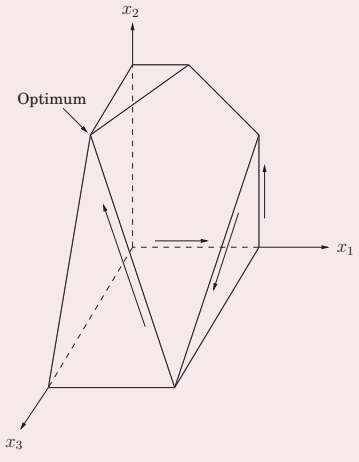
The point of final contact is the optimal vertex: $(0, 300, 100)$, with total profit $\$3100$.

Q: How would the simplex algorithm behave on this modified problem?

A possible trajectory

$$\underset{\$0}{(0,0,0)} \to \underset{\$200}{(200,0,0)} \to \underset{\$1400}{(200,200,0)} \to \underset{\$2800}{(200,0,200)} \to \underset{\$3100}{(0,300,100)}$$

The Example



LP and Its Dual

Primal LP	Dual LP
$\max c^T \mathbf{x}$	$\min \mathbf{y}^T b$
$A\mathbf{x} \leq b$	$\mathbf{y}^T A \geq c^T$
$\mathbf{x} \geq 0$	$\mathbf{y} \geq 0$

Primal LP:

$$\begin{aligned} \max\ & c_1 x_1 + \cdots + c_n x_n \\ a_{i1} x_1 + \cdots + a_{in} x_n &\leq b_i \quad \text{for } i \in I \\ a_{i1} x_1 + \cdots + a_{in} x_n &= b_i \quad \text{for } i \in E \\ x_j &\geq 0 \quad \text{for } j \in N \end{aligned}$$

Dual LP:

$$\begin{aligned} \min\ & b_1 y_1 + \cdots + b_m y_m \\ a_{1j} y_1 + \cdots + a_{mj} y_m &\geq c_j \quad \text{for } j \in N \\ a_{1j} y_1 + \cdots + a_{mj} y_m &= c_j \quad \text{for } j \notin N \\ y_i &\geq 0 \quad \text{for } i \in I \end{aligned}$$

LP and Its Dual

$$\begin{aligned} \max\ & x_1 + 6x_2 \\ & x_1 \le 200 \\ & x_2 \le 300 \\ & x_1 + x_2 \le 400 \\ & x_1, x_2 \ge 0 \end{aligned}$$

$$\begin{aligned} \min\ & 200y_1 + 300y_2 + 400y_3 \\ & y_1 + y_3 \ge 1 \\ & y_2 + y_3 \ge 6 \\ & y_1, y_2, y_3 \ge 0 \end{aligned}$$

Complementary Slackness

Theorem

Assume LP problem (P) has a solution x^* and its dual problem (D) has a solution y^*.

1. *If $x_j^* > 0$, then the j-th constraint in (D) is binding.*
2. *If the j-th constraint in (D) is not binding, then $x_j^* = 0$.*
3. *If $y_i^* > 0$, then the i-th constraint in (P) is binding.*
4. *If the i-th constraint in (P) is not binding, then $y_i^* = 0$.*

Shortest Path

Shortest path problem gives a weighted, directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{Q}^+$ mapping edges to real-valued weights, a source vertex s, and destination vertex t. We wish to compute the weight of a shortest path from s to t.

Shortest Path in LP

$$\max d_t$$

$$\begin{aligned} d_v &\le d_u + w(u,v) && (u,v) \in E \\ d_s &= 0 \\ d_i &\ge 0 && i \in V \end{aligned}$$

Q: Another formalization?

Let $\mathcal{S} = \{S \subseteq V : s \in S, t \notin S\}$; that is, $\mathcal{S}$ is the set of all s-t cuts in the graph. Then we can model the shortest s-t path problem with the following integer program,

$$\min \sum_{e \in E} w_e x_e$$

$$\sum_{e \in \delta(S)} x_e \geq 1 \qquad S \in \mathcal{S}$$
$$x_e \in \{0, 1\} \qquad e \in E$$

where $\delta(S)$ is the set of all edges that have one endpoint in S and the other endpoint not in S.

- Can we relax the restriction $x_e \in \{0, 1\}$ to $0 \leq x_e \leq 1$?
- How about $x_e \geq 0$?

Shortest Path in LP

$$\min \sum_{e \in E} w_e x_e$$

$$\sum_{e \in \delta(S)} x_e \geq 1 \qquad S \in \mathcal{S}$$

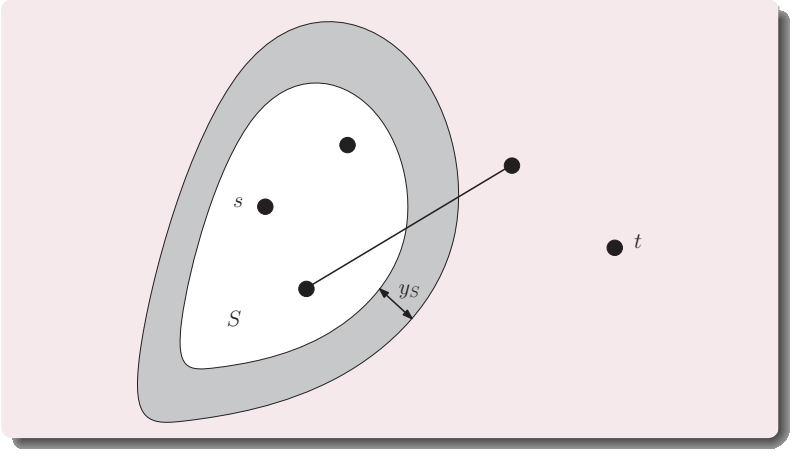
$$x_e \geq 0 \qquad e \in E$$

$$\max \sum_{S \in \mathcal{S}} y_S$$

$$\sum_{S \in \mathcal{S}, e \in \delta(S)} y_S \leq w_e \qquad e \in E$$

$$y_S \geq 0 \qquad S \in \mathcal{S}$$

The Moat



Standard Linear Programming

A general linear program has many degrees of freedom:

1. It can be either a maximization or a minimization problem.
2. Its constraints can be equations and/or inequalities.
3. The variables are often restricted to be nonnegative, but they can also be unrestricted in sign.

We will now show that these various LP options can all be reduced to one another via simple transformations.

To turn a maximization problem into a minimization (or vice versa), multiply the coefficients of the objective function by -1.

To turn an inequality constraint like $\sum_{i=1}^{n} a_i x_i \leq b$ into an equation, introduce a new variable s and use

$$\sum_{i=1}^{n} a_i x_i + s = b$$

$$s \geq 0$$

This s is called the slack variable for the inequality.

To change an equality constraint into inequalities is easy: rewrite $ax = b$ as the equivalent pair of constraints $ax \leq b$ and $ax \geq b$.

Finally, to deal with a variable x that is unrestricted in sign, do the following:

- Introduce two nonnegative variables, $x^+, x^- \geq 0$.
- Replace x, wherever it occurs in the constraints or the objective function, by $x^+ - x^-$.

Standard Form

We can reduce any LP into an LP of a much more constrained kind that we call the standard form:

- the variables are all nonnegative.
- the constraints are all equations.
- and the objective function is to be minimized.

$$\begin{array}{c} \max x_1 + 6x_2 \\ x_1 \leq 200 \\ x_2 \leq 300 \\ x_1 + x_2 \leq 400 \\ x_1, x_2 \geq 0 \end{array} \quad \Longrightarrow \quad \begin{array}{c} \min -x_1 - 6x_2 \\ x_1 + s_1 = 200 \\ x_2 + s_2 = 300 \\ x_1 + x_2 + s_3 = 400 \\ x_1, x_2, s_1, s_2, s_3 \geq 0 \end{array}$$

The Simplex Algorithm

Simplex

let v be any *vertex* of the feasible region, while there is a *neighbor* v' of v with better objective value:
set $v = v'$

Vertices and Neighbors

Definition (Vertex)

Each vertex is the unique point at which some subset of hyperplanes meet.

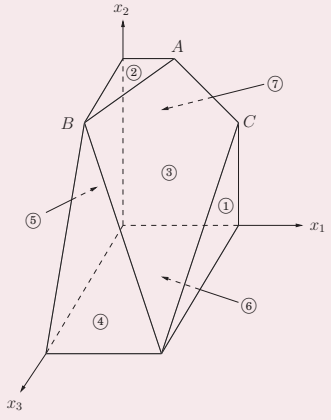
Pick a subset of the inequalities. If there is a unique point that satisfies them with equality, and this point happens to be feasible, then it is a vertex.

Each vertex is specified by a set of n inequalities (say there are n variables).

Definition (Neighbors)

Two vertices are neighbors if they have $n - 1$ defining inequalities in common.

The Example



Algorithm

On each iteration, simplex has two tasks:

1. Check whether the current vertex is optimal (and if so, halt).
2. Determine where to move next.

Both tasks are easy if the vertex is at the origin. If the vertex is elsewhere, we transform the coordinate system to move it to the origin.

The Convenience for the Origin

Suppose we have some generic LP:

$$\begin{aligned} &\max c^T \mathbf{x} \\ &A\mathbf{x} \leq b \\ &\mathbf{x} \geq 0 \end{aligned}$$

where $\mathbf{x}$ is the vector of variables, $\mathbf{x} = (x_1, \ldots, x_n)$.

Suppose the origin is feasible. Then it is certainly a vertex, since it is the unique point at which the n inequalities

$$\{x_1 \geq 0, \ldots, x_n \geq 0\}$$

are tight.

Lemma

The origin is optimal if and only if all $c_i \leq 0$.

Proof.

If all $c_i \leq 0$, then considering the constraints $x \geq 0$, we can't hope for a better objective value.

Conversely, if some $c_i > 0$, then the origin is not optimal, since we can increase the objective function by raising x_i.

Task 2 in the Origin

We can move by increasing some x_i for which $c_i > 0$.

Q: How much can we increase it?

A: Until we hit some other constraint.

We release the tight constraint $x_i \geq 0$ and increase x_i until some other inequality, previously loose, now becomes tight.

We have exactly n tight inequalities, so we are at a new vertex.

An Example

$$\max 2x_1 + 5x_2$$

$$\begin{aligned} 2x_1 - x_2 &\leq 4 \\ x_1 + 2x_2 &\leq 9 \\ -x_1 + x_2 &\leq 3 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

The trick is to transform u into the origin, by shifting the coordinate system from the usual $(x_1, \ldots, x_n)$ to the "local view" from u.

These local coordinates consist of distances $y_1, \ldots, y_n$ to the n hyperplanes (inequalities) that define and enclose u.

If one of these enclosing inequalities is $a_i \cdot x \leq b_i$, then the distance from a point x to that particular "wall" is

$$y_i = b_i - a_i \cdot x$$

The n equations of this type, one per wall, define the y_i's as linear functions of the x_i's, and this relationship can be inverted to express the x_i's as a linear function of the y_i's.

An Example

$$\max 2x_1 + 5x_2$$

$$\begin{aligned} 2x_1 - x_2 &\leq 4 \\ x_1 + 2x_2 &\leq 9 \\ -x_1 + x_2 &\leq 3 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

$$\max 15 + 7y_1 - 5y_2$$

$$\begin{aligned} y_1 + y_2 &\leq 7 \\ 3y_1 - 2y_2 &\leq 3 \\ y_2 &\geq 0 \\ y_1 &\geq 0 \\ -y_1 + y_2 &\leq 3 \end{aligned}$$

An Example

$$\max 15 + 7y_1 - 5y_2$$

$$\begin{aligned} y_1 + y_2 &\leq 7 \\ 3y_1 - 2y_2 &\leq 3 \\ y_2 &\geq 0 \\ y_1 &\geq 0 \\ -y_1 + y_2 &\leq 3 \end{aligned}$$

$$\max 22 - 7/3z_1 - 1/3z_2$$

$$\begin{aligned} -1/3z_1 + 5/3z_2 &\leq 6 \\ z_1 &\geq 0 \\ z_2 &\geq 0 \\ 1/3z_1 - 2/3z_2 &\leq 1 \\ 1/3z_1 + 1/3z_2 &\leq 4 \end{aligned}$$

Rewriting the LP

We can rewrite the entire LP in terms of the $\mathbf{y}$'s.

This doesn't fundamentally change, but expresses it in a different coordinate frame.

The revised "local" LP has the following three properties:

1. It includes the inequalities $\mathbf{y} \geq 0$, which are simply the transformed versions of the inequalities defining u.
2. u itself is the origin in $\mathbf{y}$-space.
3. The cost function becomes $\max c_u + \tilde{c}^T \mathbf{y}$, where c_u is the value of the objective function at u and $\tilde{c}$ is a transformed cost vector.

Loose End

The Starting Vertex

In a general LP, the origin might not be feasible and thus not a vertex.

It turns out that finding a starting vertex can be reduced to an LP and solved by simplex!

Start with any linear program in standard form:

$$\min c^T \mathbf{x} \text{ such that } \mathbb{A}\mathbf{x} = b \text{ and } x \geq 0.$$

We make sure that the right-hand sides of the equations are all nonnegative: if $b_i < 0$, multiply both sides of the i-th equation by -1.

The Starting Vertex

Then we create a new LP as follows:

- Create m new artificial variables $z_1, \ldots, z_m \geq 0$, where m is the number of equations.
- Add z_i to the left-hand side of the i-th equation.
- Let the objective, to be minimized, be $z_1 + z_2 + \ldots + z_m$.

An Example

$$
\begin{aligned}
&\min -x_1 - 6x_2 \\
&x_1 + s_1 = 200 \\
&x_2 + s_2 = 300 \\
&x_1 + x_2 + x_3 = 400 \\
&x_1, x_2, x_3 \geq 0
\end{aligned}
$$

$$
\begin{aligned}
&\min z_1 + z_2 + z_3 \\
&x_1 + s_1 + z_1 = 200 \\
&x_2 + s_2 + z_2 = 300 \\
&x_1 + x_2 + x_3 + z_3 = 400 \\
&x_1, x_2, x_3 \geq 0 \\
&z_1, z_2, z_3 \geq 0
\end{aligned}
$$

The Starting Vertex

For this new LP, it's easy to come up with a starting vertex, namely, the one with $z_i = b_i$ for all i and all other variables zero.

Therefore we can solve it by simplex, to obtain the optimum solution.

There are two cases:

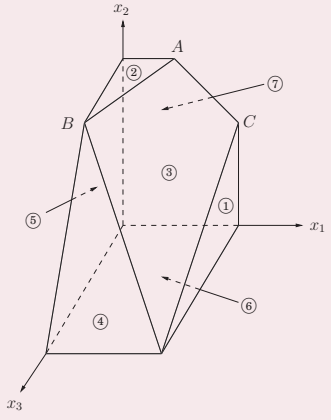
1. If the optimum value of $z_1 + \ldots + z_m$ is zero,then all z_i's obtained by simplex are zero, and hence from the optimum vertex of the new LP we get a starting feasible vertex of the original LP.
2. If the optimum objective turns out to be positive: We tried to minimize the sum of the z_i's, but it cannot be zero. This means that the original linear program is infeasible.

Degeneracy

A vertex is degenerate if it is the intersection of more than n faces of the polyhedron, say $n + 1$.

It means that if we choose any one of n sets of $n + 1$ inequalities and solve the corresponding system of these linear equations in n unknowns, we'll get the same solution in all $n + 1$ cases.

An Example



Degeneracy

This is a serious problem: simplex may return a suboptimal degenerate vertex simply because all its neighbors are identical to it and thus have no better objective.

If we modify simplex so that it detects degeneracy and continues to hop from vertex to vertex despite lack of any improvement in the cost, it may end up looping forever.

One way to fix this is by a perturbation:

change each b_i by a tiny random amount to $b_i \pm \varepsilon_i$.

This doesn't change the essence of the LP, but it has the effect of differentiating between the solutions of the linear systems.

Unboundedness

In some cases an LP is unbounded, in that its objective function can be made arbitrarily large (or small, if it's a minimization problem).

If this is the case, simplex will discover it:

- In exploring the neighborhood, it will notice that taking out an inequality and adding another leads to an underdetermined system of equations.
- The space of solutions contains a whole line across which the objective can become larger and larger, all the way to ∞.

In this case simplex halts and complains.

An Example

$$\max x_1 + x_2$$
$$x_1 - x_2 \geq 0$$
$$x_1, x_2 \geq 0$$

The Running Time

Q: What is the running time of simplex, for a generic linear program:

$$\max c^T \mathbf{x} \text{ such that } \mathbb{A}\mathbf{x} \leq 0 \text{ and } \mathbf{x} \geq 0$$

where there are n variables and $\mathbb{A}$ contains m inequality constraints?

It is an iterative algorithm that proceeds from vertex to vertex. Let u be the current vertex.

Each of its neighbors shares $n-1$ of these inequalities, so u can have at most $n \cdot m$ neighbors.

A naive way for an iteration:

1. check each potential neighbor to see whether it really is a vertex of the polyhedron,
2. determine its cost.

Finding the cost is quick, just a dot product.

Checking whether it is a true vertex involves: solve a system of n equations and check whether the result is feasible.

By Gaussian elimination this takes $O(n^3)$ time, giving total $O(mn^4)$ per iteration.

A much better way: the mn^4 can be improved to mn.

Recall the local view from vertex u. The per-iteration overhead of rewriting the LP in terms of the current local coordinates is just $O((m+n)n)$.

The local view changes only slightly between iterations, in just one of its defining inequalities.

To select the best neighbor, we recall that the objective function is of the form

$$\max c_u + \tilde{c} \cdot \mathbf{y}$$

where c_u is the value of the objective function at u.

This immediately identifies a promising direction to move: we pick any $\tilde{c}_i > 0$.

Since the rest of the LP has now been rewritten in terms of the $\mathbf{y}$-coordinates, it is easy to determine how much y_i can be increased before some other inequality is violated.

The Running Time of Simplex

Q: How many iterations could there be?

A: At most $\binom{m+n}{n}$, i.e., the number of vertices.

It is exponential in n.

And in fact, there are examples of LPs for which simplex does indeed take an exponential number of iterations.

Simplex is an exponential-time algorithm.

However, such exponential examples do not occur in practice, and it is this fact that makes simplex so valuable and so widely used.

A Notable Result

Smoothed analysis proposed by Daniel Spielman and Shanghua Teng is a way of measuring the complexity of an algorithm. It gives a more realistic analysis of the practical performance of the algorithm. It was used to explain that the simplex algorithm runs in exponential-time in the worst-case and yet in practice it is a very efficient algorithm, which was one of the main motivations for developing smoothed analysis. The authors received the 2008 Gödel Prize and the 2009 Fulkerson Prize.

Algorithm Design XIV

Linear Programming III

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Max-Flow Min-Cut in LP

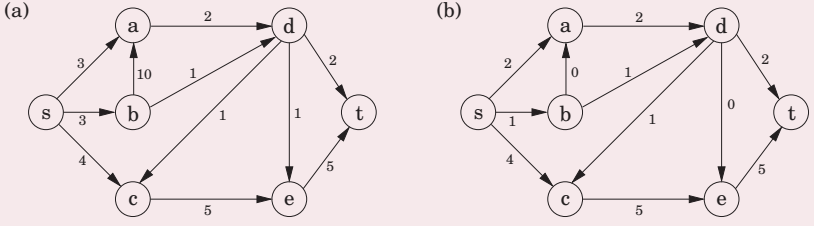
Shipping Oil

We have a network of pipelines along which oil can be sent.

The goal is to ship as much oil as possible from the source to the sink.

Each pipeline has a maximum capacity it can handle, and there are no opportunities for storing oil en route.

A Flow Example



The networks consist of a directed graph $G = (V, E)$; two special nodes $s, t \in V$, a source and sink of G; and capacities $c_e > 0$ on the edges.

Aim to send as much oil as possible from s to t without exceeding the capacities of any of the edges.

A flow consists of a variable f_e for each edge e of the network, satisfying the following two properties:

1. It doesn't violate edge capacities: $0 \leq f_e \leq c_e$ for all $e \in E$.
2. For all nodes u except s and t, the amount of flow entering u equals the amount leaving

$$\sum_{(w,v)\in E} f_{wu} = \sum_{(u,z)\in E} f_{uz}$$

In other words, flow is conserved.

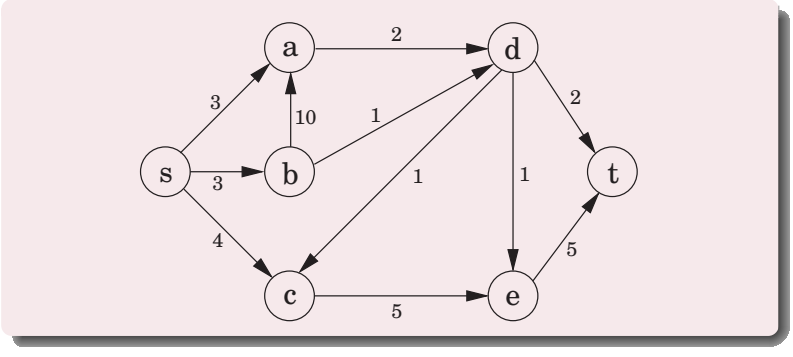
The value of a flow is the total quantity sent from s to t and, by the conservation principle, is equal to the quantity leaving s:

$$\texttt{val}(f) = \sum_{(s,u)\in E} f_{su}$$

Our goal is to assign values to $\{f_e | e \in E\}$ that will satisfy a set of linear constraints and maximize a linear objective function.

This is a linear program. The maximum-flow problem reduces to linear programming.

The Example



11 variables, one per edge.

maximize $f_{sa} + f_{sb} + f_{sc}$

27 constraints:

- 11 for nonnegativity (such as $f_{sa} \geq 0$),
- 11 for capacity (such as $f_{sa} \leq 3$),
- 5 for flow conservation (one for each node of the graph other than s and t, such as $f_{sc} + f_{dc} = f_{ce}$).

Another Representation

First, introduce a fictitious edge of infinite capacity from t to s thus converting the flow to a circulation;

The objective is to maximize the flow on this edge, denoted by f_{ts}.

The advantage of making this modification is that we can now require flow conservation at s and t as well.

Another Representation

$$\max f_{ts}$$

$$f_{ij} \leq c_{ij} \qquad (i,j) \in E$$

$$\sum_{(w,i)\in E} f_{wi} - \sum_{(i,z)\in E} f_{iz} \leq 0 \qquad i \in V$$

$$f_{ij} \geq 0 \qquad (i,j) \in E$$

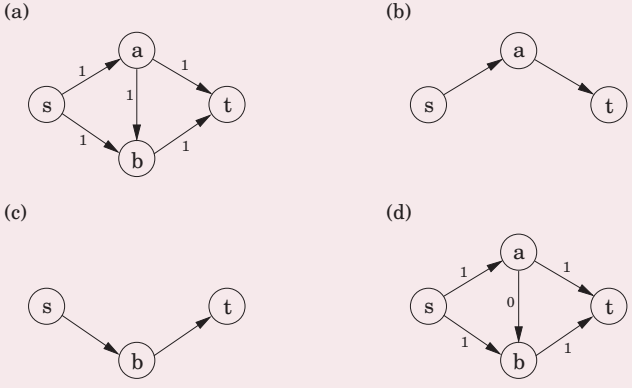
A Closer Look at the Algorithm

Simplex algorithm keeps making local moves on the surface of a convex feasible region, successively improving the objective function until reaches the optimal solution.

The behavior of the simplex has an elementary interpretation:

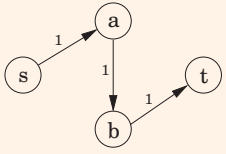
- Start with zero flow.
- Repeat: choose an appropriate path from s to t, and increase flow along the edges of this path as much as possible.

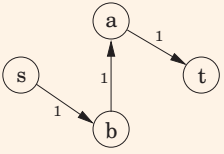
A Flow Example



What if we choose a path that blocks all other paths?

Simplex gets around this problem by also allowing paths to cancel existing flow.





A Closer Look at the Algorithm

To summarize, in each iteration simplex looks for an $s - t$ path whose edges (u, v) can be of two types:

1. (u, v) is in the original network, and is not yet at full capacity.
2. The reverse edge (v, u) is in the original network, and there is some flow along it.

If the current flow is f, then in the first case, edge (u, v) can handle up to $c_{uv} - f_{uv}$ additional units of flow;

in the second case, up to f_{vu} additional units (canceling all or part of the existing flow on (v, u)).

A Closer Look at the Algorithm

These flow-increasing opportunities can be captured in a residual network $G^f = (V, E^f)$, which has exactly the two types of edges listed, with residual capacities c^f:

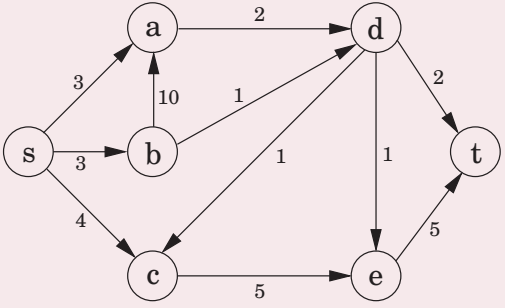
$$\begin{cases} c_{uv} - f_{uv} & \text{if } (u,v) \in E \text{ and } f_{uv} < c_{uv} \\ f_{vu} & \text{if } (v,u) \in E \text{ and } f_{vu} > 0 \end{cases}$$

Thus we can equivalently think of simplex as choosing an $s - t$ path in the residual network.

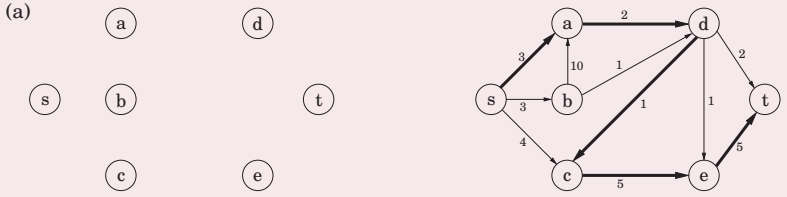
By simulating the behavior of simplex, we get a direct algorithm for solving max-flow.

It proceeds in iterations, each time explicitly constructing G^f, finding a suitable $s - t$ path in G^f by the breadth-first search, and halting if there is no longer any such path along which flow can be increased.

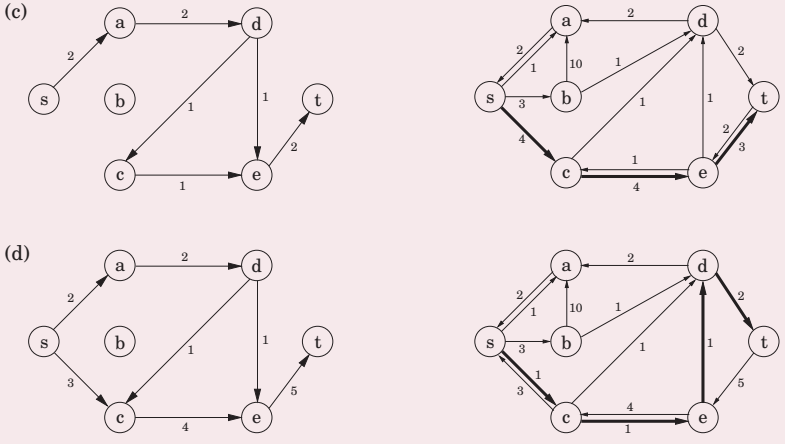
The Example



A Flow Example

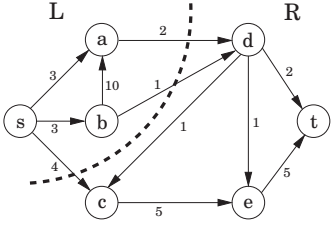


A Flow Example



A truly remarkable fact:

Not only does simplex correctly compute a maximum flow, but it also generates a short proof of the optimality of this flow!



An (s,t)-cut partitions the vertices into two disjoint groups L and R, such that $s \in L$ and $t \in R$. Its capacity is the total capacity of the edges from L to R, and as argued previously, is an upper bound on any flow:

Pick any flow f and any (s,t)-cut (L,R). Then $\texttt{size}(f) \leq \texttt{capacity}(L,R)$.

A Certificate of Optimality

Theorem (Max-flow min-cut)

The size of the maximum flow in a network equals the capacity of the smallest (s, t)-cut.

Proof:

Suppose f is the final flow when the algorithm terminates.

We know that node t is no longer reachable from s in the residual network G^f.

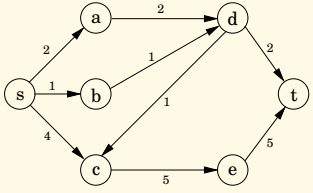
Let L be the nodes that are reachable from s in G^f, and let $R = V \backslash L$ be the rest of the nodes.

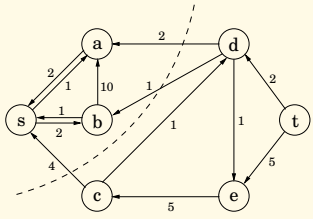
We claim that $\texttt{size}(f) = \texttt{capacity}(L, R)$.

To see this, observe that by the way L is defined, any edge going from L to R must be at full capacity (in the current flow f), and any edge from R to L must have zero flow.

Therefore the net flow across (L, R) is exactly the capacity of the cut.

An Example of Max-Flow Min-Cut





Efficiency

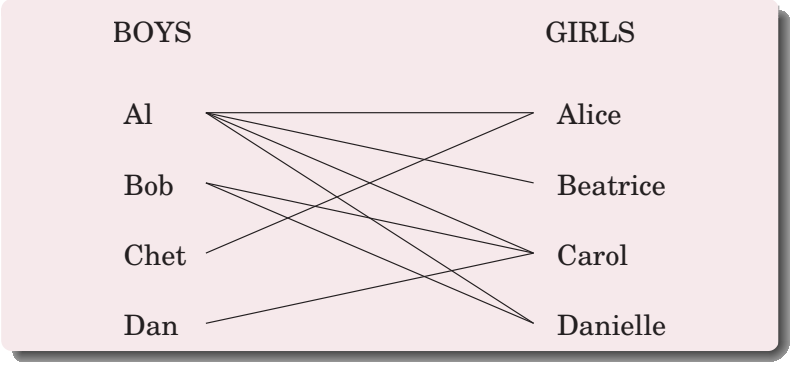
Each iteration is efficient, requiring $O(|E|)$ time if a DFS or BFS is used to find an $s-t$ path.

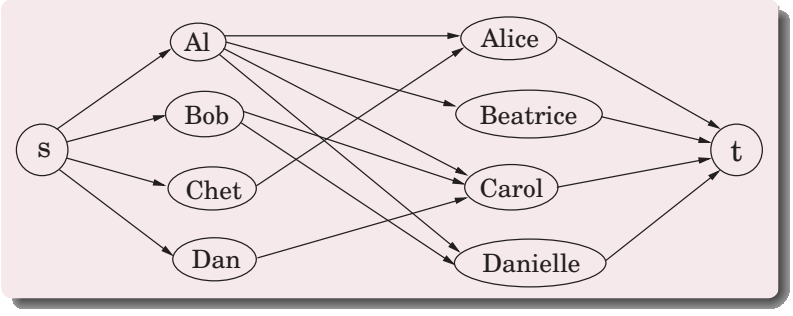
But how many iterations are there?

Suppose all edges in the original network have integer capacities $\leq C$. Then on each iteration of the algorithm, the flow is always an integer and increases by an integer amount. Therefore, since the maximum flow is at most $C|E|$.

If paths are chosen by using a BFS, which finds the path with the fewest edges, then the number of iterations is at most $O(|V| \cdot |E|)$. *Edmonds-Karp algorithm*

This latter bound gives an overall running time of $O(|V| \cdot |E|^2)$ for maximum flow.


BOYS
GIRLS
Al
Bob
Chet
Dan
Alice
Beatrice
Carol
Danielle


Al
Bob
s
Chet
Dan
Alice
Beatrice
Carol
Danielle
t

Min-Max Relations in LP

LP for Max Flow

$$\max f_{ts}$$

$$f_{ij} \leq c_{ij} \qquad (i,j) \in E$$

$$\sum_{(w,i)\in E} f_{wi} - \sum_{(i,z)\in E} f_{iz} \leq 0 \qquad i \in V$$

$$f_{ij} \geq 0 \qquad (i,j) \in E$$

$$\max f_{ts}$$

$$f_{ij} \le c_{ij} \qquad (i,j) \in E$$

$$\sum_{(w,i)\in E} f_{wi} - \sum_{(i,z)\in E} f_{iz} \le 0 \qquad i \in V$$

$$f_{ij} \ge 0 \qquad (i,j) \in E$$

$$\min \sum_{(i,j)\in E} c_{ij} d_{ij}$$

$$\begin{aligned} d_{ij} - p_i + p_j &\ge 0 && (i,j) \in E \\ p_s - p_t &\ge 1 \\ d_{ij} &\ge 0 && (i,j) \in E \\ p_i &\ge 0 && i \in V \end{aligned}$$

Explanation of the Dual

$$\min \sum_{(i,j)\in E} c_{ij} d_{ij}$$

$$\begin{aligned} d_{ij} - p_i + p_j &\geq 0 && (i,j) \in E \\ p_s - p_t &\geq 1 && \\ d_{ij} &\in \{0,1\} && (i,j) \in E \\ p_i &\in \{0,1\} && i \in V \end{aligned}$$

To obtain the dual program we introduce variables d_{ij} and p_i corresponding to the two types of inequalities in the primal.

- d_{ij}: distance labels on edges;
- p_i: potentials on nodes.

$$\min \sum_{(i,j)\in E} c_{ij} d_{ij}$$

$$\begin{aligned} d_{ij} - p_i + p_j &\geq 0 \quad && (i,j) \in E \\ p_s - p_t &\geq 1 \\ d_{ij} &\in \{0,1\} && (i,j) \in E \\ p_i &\in \{0,1\} && i \in V \end{aligned}$$

Let $(\mathbf{d}^*, \mathbf{p}^*)$ be an optimal solution to this integer program.

The only way to satisfy the inequality $p_s^* - p_t^* \geq 1$ with a $0/1$ substitution is to set $p_s^* = 1$ and $p_t^* = 0$.

This solution defines an $s-t$ cut $(X, \overline{X})$, where X is the set of potential 1 nodes, and $\overline{X}$ the set of potential 0 nodes.

$$\min \sum_{(i,j)\in E} c_{ij} d_{ij}$$

$$\begin{aligned} d_{ij} - p_i + p_j &\geq 0 \quad && (i,j) \in E \\ p_s - p_t &\geq 1 \\ d_{ij} &\in \{0,1\} && (i,j) \in E \\ p_i &\in \{0,1\} && i \in V \end{aligned}$$

Consider an edge (i,j) with $i \in X$ and $j \in \overline{X}$, Since $p_i^* = 1$ and $p_j^* = 0$, and thus $d_{ij}^* = 1$.

The distance label for each of the remaining edges can be set to either 0 or 1 without violating the first constraints.

The objective function value is precisely the capacity of the cut $(X, \overline{X})$, and hence $(X, \overline{X})$ must be a minimum $s-t$ cut.

Relaxation of the Integer Program

The integer program is a formulation of the minimum $s - t$ cut problem.

The dual program can be viewed as a relaxation of the integer program where the integrality constraint on the variables is dropped.

This leads to the constraints $1 \geq d_{ij} \geq 0$ for $(i, j) \in E$ and $1 \geq p_i \geq 0$ for $i \in V$.

The upper bound constraints on the variables are redundant; their omission cannot give a better solution.

We will say that this program is the LP relaxation of the integer program.

Relaxation of the Integer Program

The best fractional $s-t$ cut could have lower capacity than the best integral cut. This does not happen here.

Now, it can be proven that each vertex solution is integral, with each coordinate being 0 or 1.

The constraint matrix of this program is totally unimodular, Thus, the dual program always has an integral optimal solution.

More Examples

Set Cover

- Input: A set of elements U, sets $S_1, \ldots, S_m \subseteq U$
- Output: A selection of the S_i whose union is U.
- Cost: Number of sets picked.

Set Cover

$$\begin{aligned} \min \quad & \sum_{S \in \mathcal{S}} x_S & \\ & \sum_{S: e \in S} x_S \geq 1, & e \in U \\ & x_S \geq 0, & S \in \mathcal{S} \end{aligned}$$

$$\begin{aligned} \max \quad & \sum_{e \in U} y_e & \\ & \sum_{e: e \in S} y_e \leq 1, & S \in \mathcal{S} \\ & y_e \geq 0, & e \in U \end{aligned}$$

Set Cover

- Input: A set of elements U, sets $S_1, \ldots, S_m \subseteq U$, and a cost function $c : \mathcal{S} \to \mathbb{Q}^+$.
- Output: A selection of the S_i whose union is U.
- Cost: Sum of costs of set picked.

The special case, in which all subsets are of unit cost, will be called the cardinality set cover problem.

Set Cover

$$\begin{aligned} \min \quad & \sum_{S \in \mathcal{S}} c(S) x_S & \\ & \sum_{S: e \in S} x_S \geq 1, & e \in U \\ & x_S \geq 0, & S \in \mathcal{S} \end{aligned}$$

$$\begin{aligned} \max \quad & \sum_{e \in U} y_e & \\ & \sum_{e: e \in S} y_e \leq c(S), & S \in \mathcal{S} \\ & y_e \geq 0, & e \in U \end{aligned}$$

Quiz: Set Multicover

Each element, e, needs to be covered a specified integer number, r_e, of times.

The objective again is to cover all elements up to their coverage requirements at minimum cost.

Each set can be picked at most once.

Let $r_e \in \mathbb{Z}_+$ be the coverage requirement for each element $e \in U$.

$$\begin{aligned} \min \quad & \sum_{S \in \mathcal{S}} c(S) x_S & \\ & \sum_{S: e \in S} x_S \geq r_e, & e \in U \\ & x_S \in \{0,1\}, & S \in \mathcal{S} \end{aligned}$$

In the LP-relaxation, the constraints $x_S \leq 1$ are no longer redundant.

$$
\begin{aligned}
\min \quad & \sum_{S \in \mathcal{S}} c(S) x_S & & \\
& \sum_{S : e \in S} x_S \geq r_e, & & e \in U \\
& -x_S \geq -1, & & S \in \mathcal{S} \\
& x_S \geq 0, & & S \in \mathcal{S}
\end{aligned}
$$

Dual Program

The additional constraints in the primal lead to new variables, z_S, in the dual.

$$
\begin{aligned}
\max \quad & \sum_{e \in U} r_e y_e - \sum_{S \in \mathcal{S}} z_S \\
& (\sum_{e: e \in S} y_e) - z_S \leq c(S), && S \in \mathcal{S} \\
& y_e \geq 0, && e \in U \\
& z_S \geq 0, && S \in \mathcal{S}
\end{aligned}
$$

Algorithm Design XV

NP Problem I

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Efficient Problems, Difficult Problems

Efficient Algorithms

We have developed algorithms for

- Finding shortest paths in graphs,
- Minimum spanning trees in graphs,
- Matchings in bipartite graphs,
- Maximum increasing subsequences,
- Maximum flows in networks,
- .....

All these algorithms are efficient, since their time requirement grows as a polynomial function (such as n, n^2, or n^3) of the size of the input.

Exponential Search Space

In these problems we are searching for a solution (path, tree, matching) from among an exponential population of possibilities.

All these problems could in principle be solved in exponential time by checking through all candidate solutions, one by one.

An algorithm with running time 2^n, or worse, is useless in practice.

The efficient algorithms is to find clever ways to bypass exhaustive search, using clues from the input to narrow down the search space.

Are there "search problems" in which seeking a solution among an exponential chaos, and the fastest algorithms for them are exponential?

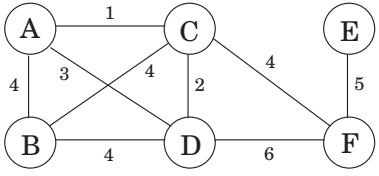
Minimum Spanning Trees

Build a Network

Suppose you are asked to network a collection of computers by linking selected pairs of them.

This translates into a graph problem in which

- nodes are computers,
- undirected edges are potential links, each with a maintenance cost.



A General Kruskal's Algorithm

$X = \{\}$;
repeat until $|X| = |V| - 1$;
    pick a set $S \subset V$ for which X has no edges between S and $V - S$;
    let $e \in E$ be the minimum-weight edge between S and $V - S$;
    $X = X \cup \{e\}$;

A Little Change of the MST

What if the tree is not allowed to branch?

SATISFIABILITY PROBLEM

Satisfiability

The instances of SATISFIABILITY or SAT:

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

a Boolean formula in conjunctive normal form (CNF).

It is a collection of clauses (the parentheses),

- each consisting of the disjunction of several literals;
- a literal is either a Boolean variable or the negation of one.

A satisfying truth assignment is an assignment of `false` or `true` to each variable so that every clause contains a literal of `true`.

Given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

Given a set of clauses, where each clause is the disjunction of two literals, looking for an assignment so that all clauses are satisfied.

$$(x_1 \vee x_2) \wedge (\overline{x}_1 \vee x_3) \wedge (x_1 \vee \overline{x}_2) \wedge (x_3 \vee x_4) \wedge (\overline{x}_1 \vee \overline{x}_4)$$

Given an instance I of 2-SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes, one for each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$ of I, G_I has an edge from the negation of α to β, and one from the negation of β to α.

Show that if G_I has a strongly connected component containing both x and $\overline{x}$ for some variable x, then I has no satisfying assignment.

If none of G_I's strongly connected components contain both a literal and its negation, then the instance I must be satisfiable.

Conclude that there is a linear-time algorithm for solving 2-SAT.

3-SAT, SAT?

Various Problem Definitions and Reductions

Satisfiability

The instances of SATISFIABILITY or SAT:

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

a Boolean formula in conjunctive normal form (CNF).

It is a collection of clauses (the parentheses),

- each consisting of the disjunction of several literals;
- a literal is either a Boolean variable or the negation of one.

A satisfying truth assignment is an assignment of `false` or `true` to each variable so that every clause contains a literal of `true`.

Given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

Decision Problem and Search Problem

Decision problem. Does there exist an assignment to satisfy the formula in CNF?

Search problem. Find an assignment to satisfy the formula.

Decision VS. Search

Turning an decision problem into a search problem does not change its difficulty, because the two versions reduce to one another.

The decision problem is a special version of the search problem.

The solution of the search problem is on the output tape of the Turing machine that encodes the decision problem.

For the SAT, we can ask the assignment of each variable one by one until all variables are finally assigned. The total execution time is bounded by the number of variables times the execution time of the decision problems.

Search Problems

SAT is a typical search problem.

We are given an `instance` I

- Some input data specifying the problem
- A Boolean formula in conjunctive normal form

we are asked to find a `solution` S

- An object that meets a particular specification
- An assignment that satisfies each clause

If no such solution exists, we must say so.

A search problem must have the property that any proposed solution S to an instance I can be quickly checked for correctness.

S must be concise, with length polynomially bounded by that of I.

- This is true for SAT, where S is an assignment to the variables.

There is a polynomial-time algorithm that takes as input I and S and decides whether or not S is a solution of I.

- For SAT, it is easy to check whether the assignment specified by S satisfies every clause in I.

A search problem is specified by an algorithm C that takes two inputs, an instance I and a proposed solution S, and runs in time polynomial in $|I|$.

We say S is a solution to I if and only if $C(I, S) = \texttt{true}$.

Researchers over the past 80 years have tried to find efficient ways to solve the SAT, but without success.

The fastest algorithms we have are still exponential on their worst-case inputs.

There are two natural variants of SAT with good algorithms.

- 2-SAT can be solved in linear time.
- All clauses contain at most one positive literal, say Horn formula, can be found by the greedy algorithm.

Traveling Salesman Problem

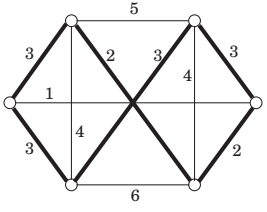
The Traveling Salesman Problem

In the traveling salesman problem(TSP) we are given n vertices and all $n(n-1)/2$ distances between them, and a budget b.

To find a cycle that passes through every vertex exactly once, of total cost b or less - or to report that no such cycle.

A permutation $\tau(1), \ldots, \tau(n)$ of the vertices such that when they are toured in this order, the total distance covered is at most b:

$$d_{\tau(1),\tau(2)} + d_{\tau(2),\tau(3)} + \ldots + d_{\tau(n),\tau(1)} \leq b$$



Optimization Problem and Search Problem

Optimization problem. given an instance of TSP, find the minimum cost tour.

Search problem. given an instance of TSP, find a tour within the budget (or report that none exists).

Search VS. Optimization

Turning an optimization problem into a search problem does not change its difficulty, because the two versions reduce to one another.

Any algorithm that solves the optimization also solves the search problem:

- find the optimum tour and if it is within budget, return it; if not, there is no solution.

Conversely, an algorithm for the search problem can also be used to solve the optimization problem:

- First suppose that we knew the cost of the optimum tour; then we could find this tour by calling the algorithm for the search problem, using the optimum cost as the budget.
- We can find the optimum cost by binary search.

Search Instead of Optimization

Isn't any optimization problem also a search problem in the sense that we are searching for a solution that has the property of being optimal?

The solution to a search problem should be easy to recognize, or as we put it earlier, polynomial-time checkable.

Given a potential solution to the TSP, it is easy to check the properties "is a tour" (just check that each vertex is visited exactly once) and "has total length $\leq b$."

But how could one check the property "is optimal"?

There are no known polynomial-time algorithms for the TSP, despite much effort by researchers over nearly a century.

There exists a faster, yet still exponential, dynamic programming algorithm.

The Minimum spanning tree (MST) problem, for which we do have efficient algorithms, provides a stark contrast here.

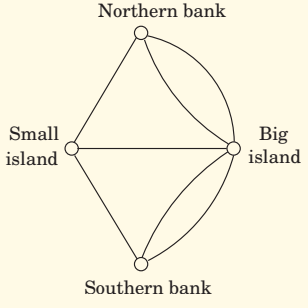
The TSP can be thought of as a tough cousin of the MST problem, in which the tree is not allowed to branch and is therefore a path.

This extra restriction on the structure of the tree results in a much harder problem.

Euler and Rudrata

Euler path:

Given a graph, find a path that contains each edge exactly once

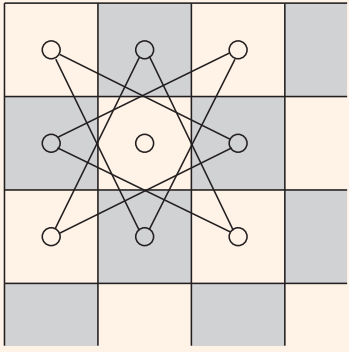


The answer is yes if and only if

1. the graph is connected and
2. every vertex, with the possible exception of two vertices (the start and final vertices of the walk), has even degree.

There is a polynomial time algorithm for EULER PATH.

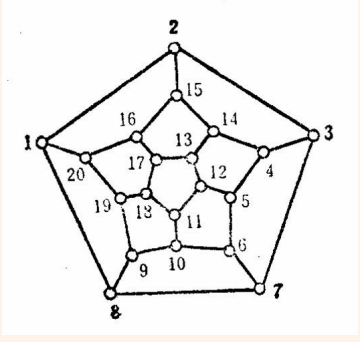
Rudrata Cycle



Rudrata Cycle:

Given a graph, find a cycle that visits each vertex exactly once.

In the literature this problem is known as the Hamilton cycle problem.

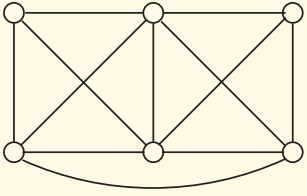


Cuts and Bisections

Minimum Cut

A cut is a set of edges whose removal leaves a graph disconnected.

MINIMUM CUT: given a graph and a budget b, find a cut with at most b edges.



This problem can be solved in polynomial time by $n - 1$ max-flow computations:

- give each edge a capacity of 1,
- and find the maximum flow between some fixed node and every single other node.

The smallest such flow will correspond (via the max-flow min-cut theorem) to the smallest cut.

Balanced Cut

In many graphs, the smallest cut leaves just a singleton vertex on one side - it consists of all edges adjacent to this vertex.

Far more interesting are small cuts that partition the vertices of the graph into nearly equal-sized sets.

BALANCED CUT: Given a graph with n vertices and a budget b, partition the vertices into two sets S and T such that $|S|, |T| \geq n/3$ and such that there are at most b edges between S and T.

Integer Linear Programming

In a LINEAR PROGRAMMING problem we are given a set of variables, and to assign real values to them so as to

1. satisfy a set of linear equations and/or linear inequalities involving these variables, and
2. maximize or minimize a given linear objective function.

$$\begin{aligned}
&\max x_1 + 6x_2 + 13x_3 \\
&x_1 \leq 200 \\
&x_2 \leq 300 \\
&x_1 + x_2 + x_3 \leq 400 \\
&x_2 + 3x_3 \leq 600 \\
&x_1, x_2, x_3 \geq 0
\end{aligned}$$

INTEGER LINEAR PROGRAMMING (ILP): We are given a set of linear inequalities $A\mathbf{x} \leq b$, where

- A is an $m \times n$ matrix and
- b is an m-vector;
- an objective function specified by an n-vector c;
- a goal g.

We want to find a nonnegative integer n-vector x such that $A\mathbf{x} \leq b$ and $c \cdot \mathbf{x} \geq g$.

$$\begin{aligned} \max\ & 2x_1 + 5x_2 \\ & 2x_1 - x_2 \leq 4 \\ & x_1 + 2x_2 \leq 9 \\ & -x_1 + x_2 \leq 3 \\ & x_1, x_2 \geq 0 \end{aligned}$$

$$\begin{aligned} & 2x_1 + 5x_2 \leq g \\ & 2x_1 - x_2 \leq 4 \\ & x_1 + 2x_2 \leq 9 \\ & -x_1 + x_2 \leq 3 \\ & x_1, x_2 \geq 0 \end{aligned}$$

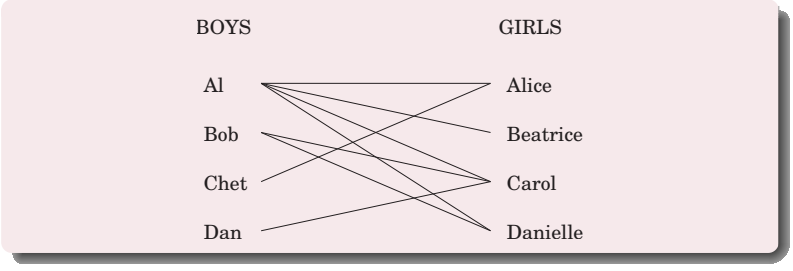
But there is a redundancy here:

- the last constraint $c \cdot \mathbf{x} \geq g$ is itself a linear inequality and
- can be absorbed into $A\mathbf{x} \leq b$.

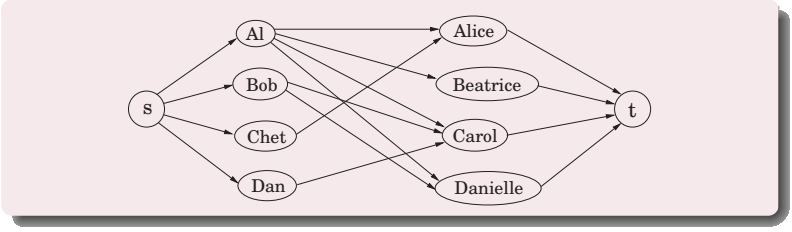
So, we define ILP to be following search problem:

Given A and b, find a nonnegative integer vector $\mathbf{x}$ satisfying the inequalities $A\mathbf{x} \leq b$.

3D-MATCHING


BOYS
GIRLS
Al
Bob
Chet
Dan
Alice
Beatrice
Carol
Danielle

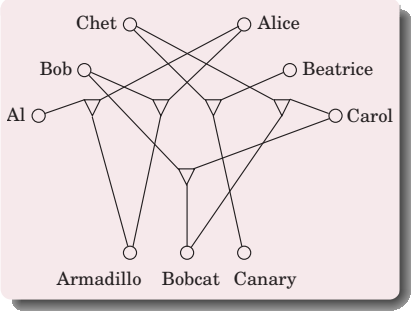
Bipartite Matching



3D MATCHING: There are n boys, n girls, and n pets. The compatibilities are specified by a set of triples, each containing a boy, a girl, and a pet.

A triple (b, g, p) means that boy b, girl g, and pet p get along well together.

To find n disjoint triples and thereby create n harmonious households.



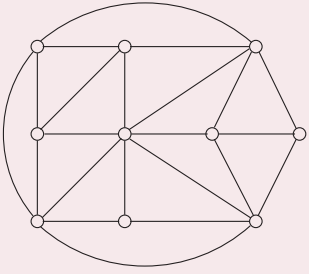
Independent Set, Vertex Cover, and Clique

Independent Set, Vertex Cover, and Clique

INDEPENDENT SET: Given a graph and an integer g, find g vertices, no two of which have an edge between them.

VERTEX COVER: Given a graph and an integer b, find b vertices cover (touch) every edge.

CLIQUE: Given a graph and an integer g, find g vertices such that all possible edges between them are present.



LONGEST PATH

Longest Path

LONGEST PATH: Given a graph G with nonnegative edge weights and two distinguished vertices s and t, along with a goal g.

To find a path from s to t with total weight at least g.

To avoid trivial solutions we require that the path be simple, containing no repeated vertices.

KNAPSACK

Knapsack

KNAPSACK: Given integer weights $w_1, \ldots, w_n$ and integer values $v_1, \ldots, v_n$ for n items. We are also given a weight capacity W and a goal g.

Seek a set of items whose total weight is at most W and whose total value is at least g.

The problem is solvable in time $O(nW)$ by dynamic programming.

Knapsack

Is there a polynomial algorithm for KNAPSACK? Nobody knows of one.

A variant of the KNAPSACK problem is that the unary integers.

- by writing $IIIIIIIIIII$ for 12.
- It defines a legitimate problem, which we could call UNARY KNAPSACK.
- It has a polynomial algorithm.

A different variation:

- Suppose now that each item's value is equal to its weight, the goal g is the same as the capacity W.
- This special case is tantamount to finding a subset of a given set of integers that adds up to exactly W.
- Q: Could it be polynomial?

Subset Sum

SUBSET SUM: Find a subset of a given set of integers that adds up to exactly W.

Algorithm Design XVI

NP Problem II

Guoqiang Li
School of Software

Shanghai Jiao Tong University

P and NP Problems

Set Cover

Set Cover

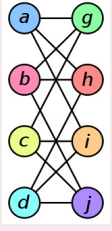
- Input: A set of elements B, sets $S_1, \ldots, S_m \subseteq B$
- Output: A selection of the S_i whose union is B.
- Cost: Number of sets picked.

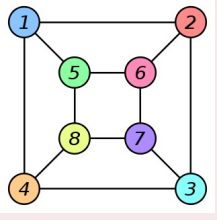
Graph Isomorphism

An isomorphism of graphs G and H is a bijection between the vertex sets of G and H

$$f : V(G) \to V(H)$$

such that any two vertices u and v of G are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in H.





Hard Problems, Easy Problems

Hard problems (NP-complete)	Easy problems (in P)
3SAT	2SAT, Horn SAT
Traveling salesman problem	Minimum spanning tree
Longest path	Shortest path
3D matching	Bipartite matching
Knapsack	Unary knapsack
Independent set	Independent set on trees
Integer linear programming	Linear programming
Rudrata path	Euler path
Balanced cut	Minimum cut

if A search problem satisfies:

1. there exists an efficient checking algorithm C, taking as input the given `instance` I, a `solution` S, and outputs `true` iff S is a solution I.
2. The running time of $C(I, S)$ is bounded by a polynomial in $|I|$.

We denote the class of all such problems by NP.

An algorithm that takes as input an `instance` I and has a running time polynomial in $|I|$.

- I has a solution, the algorithm returns such a solution;
- I has no solution, the algorithm correctly reports so.

The class of all search problems that can be solved in polynomial time is denoted P.

Why P and NP

P: polynomial time

NP: nondeterministic polynomial time

Complementation

A class of problems $\mathcal{C}$ is closed under complementation if for any problem in $\mathcal{C}$, its complement is also in $\mathcal{C}$.

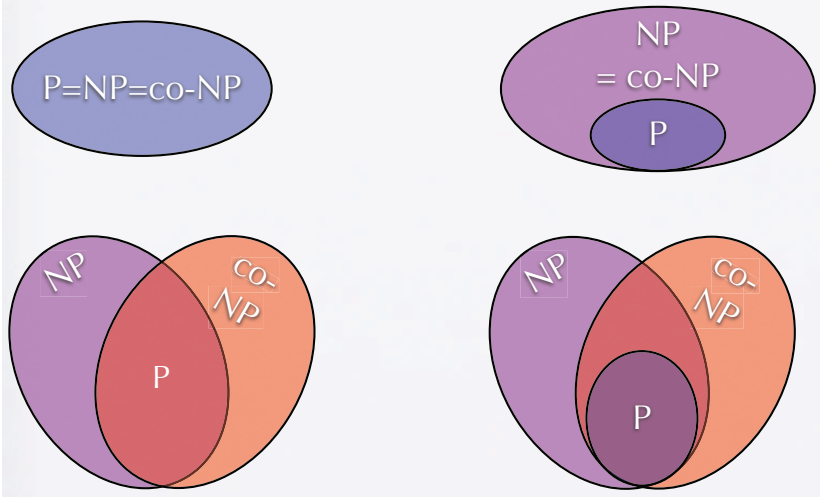
P: is closed under complementation.

NP ?

Example (Complementation of TSP)

Given n cities with their intercity distances, is it the case that there does not exist any tour length k or less?

Conjectures



Theorem Proving

- Input: A mathematical statement φ and n.
- Problem: Find a proof of φ of length $\leq n$ if there is one.

A formal proof of a mathematical assertion is written out in excruciating detail, it can be checked mechanically, by an efficient algorithm and is therefore in NP.

So if P $=$ NP, there would be an efficient method to prove any theorem, thus eliminating the need for mathematicians!

Solve One and All Solved

Even if we believe P $\neq$ NP, can we find an evidence that these particular problems have no efficient algorithm?

Such evidence is provided by reductions, which translate one search problem into another.

We will show that the hard problems in previous lecture exactly the same problem, the hardest search problems in NP.

If one of them has a polynomial time algorithm, then every problem in NP has a polynomial time algorithm.

Reduction

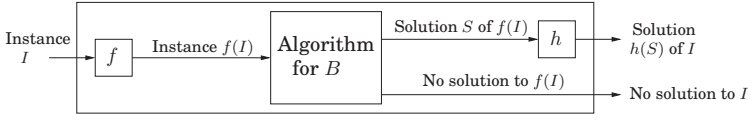
Reduction Between Search Problems

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Together with another polynomial time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I.

If $f(I)$ has no solution, then neither does I.

These two translation procedures f and h imply that any algorithm for B can be converted into an algorithm for A.



The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

$$A \to B$$

- If we can solve B efficiently, then we can also solve A efficiently.
- If we know A is hard, then B must be hard too.

If $A \to B$ and $B \to C$, then $A \to C$.

NP-Completeness

Definition

A NP problem is NP-complete if all other NP problems reduce to it.

NP-complete problems are hard: all other search problems reduce to them.

For a problem to be NP-complete, it can solve every NP problem in the world.

If even one NP-complete problem is in P, then P = NP.

If a problem A is NP-complete, a new NP problem B is proved to be NP-complete, by reducing A to B.

Co-NP-Completeness

Definition

A co-NP problem is co-NP-complete if all other co-NP problems reduce to it.

A problem is NP-complete if and only if its complement is co-NP-complete.

If a problem and its complement are NP-complete then co-NP = NP.

TAUTOLOGY

A CNF formula f is unsatisfiable if and only if its negation is a TAUTOLOGY. The negation of a CNF formula can be converted into a DNF formula. The resulting DNF formula is a TAUTOLOGY if and only if the negation of the CNF formula is a tautology.

The problem TAUTOLOGY: Given a formula f in DNF, is it a tautology?

- TAUTOLOGY is in P if and only if co-NP = P, and
- TAUTOLOGY is in NP if and only if co-NP = NP.

Factoring

The difficulty of FACTORING is of a different nature than that of the other hard search problems we have just seen.

Nobody believes that FACTORING is NP-complete.

One evidence is that a number can always be factored into primes.

Another difference: FACTORING succumbs to the power of quantum computation, while SAT, TSP and the other NPC problems do not seem to.

Primarily and Composite

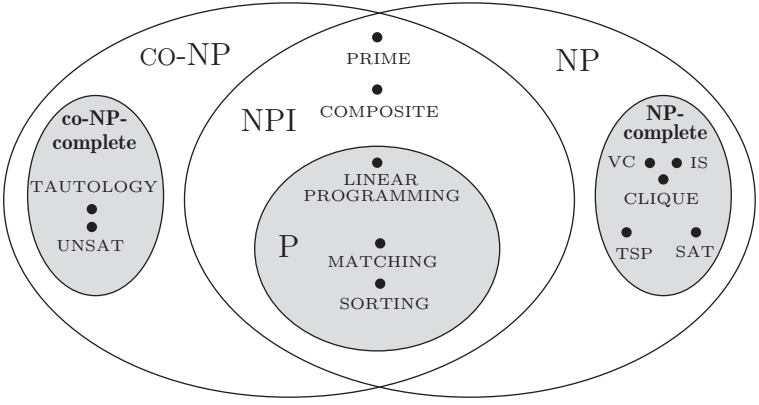
Primarily

Given an integer $k \geq 2$, is k a prime number?

Composite

Given an integer $k \geq 4$, are there two integers $p, q \geq 2$ such that $k = pq$?

NPI (A Problematic Category)



NP-Intermediate

Definition (NPI)

Problems that are in the complexity class NP but are neither in the class P nor NP-complete are called NP-intermediate, and the class of such problems is called NPI.

Theorem (Lander Theorem)

If $P \neq NP$, then NPI is not empty; that is, NP contains problems that are neither in P nor NP-complete.

Quiz

Prove that if NP $\neq$ Co-NP, then P $\neq$ NP.

Algorithm Design XVII

NP Problem III

Guoqiang Li
School of Software

Shanghai Jiao Tong University

The Reductions

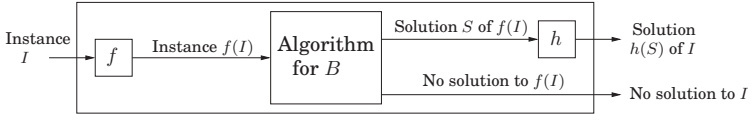
Reduction Between Search Problems

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Together with another polynomial time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I.

If $f(I)$ has no solution, then neither does I.

These two translation procedures f and h imply that any algorithm for B can be converted into an algorithm for A.

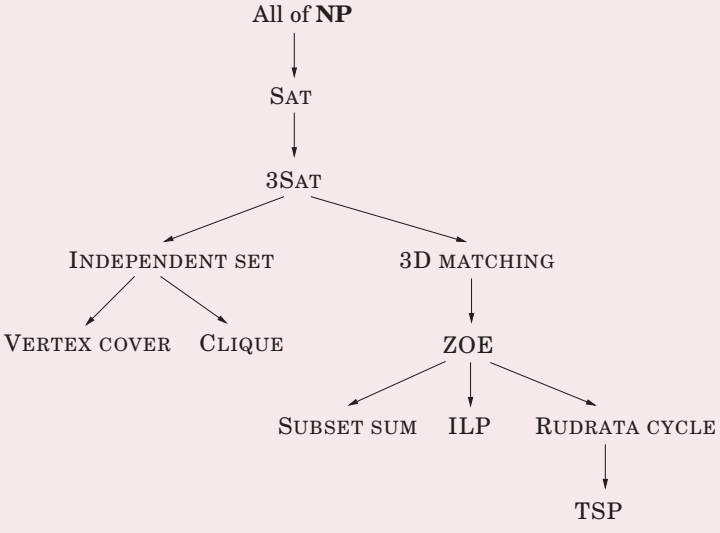


Assume there is a reduction from a problem A to a problem B.

$$A \to B$$

- If we can solve B efficiently, then we can also solve A efficiently.
- If we know A is hard, then B must be hard too.

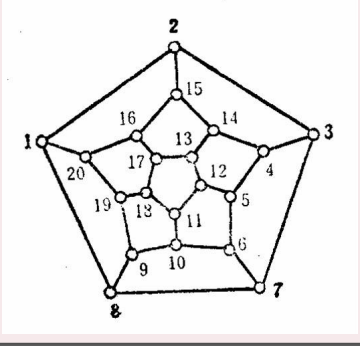
If $A \to B$ and $B \to C$, then $A \to C$.


All of NP
SAT
3SAT
INDEPENDENT SET
3D MATCHING
VERTEX COVER
CLIQUE
ZOE
SUBSET SUM
ILP
RUDRATA CYCLE
TSP

RUDRATA PATH $\rightarrow$ RUDRATA CYCLE

Rudrata Cycle

Given a graph, find a cycle that visits each vertex exactly once.

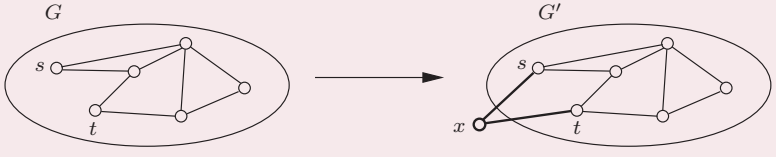


A RUDRATA (s,t)-PATH problem specifies two vertices s and t and wants a path starting at s and ending at t that goes through each vertex exactly once.

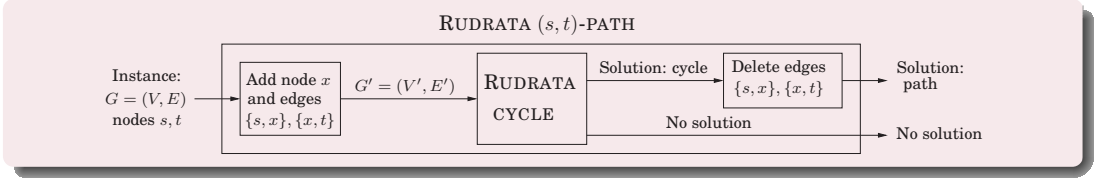
Q: Is it possible that RUDRATA CYCLE is easier than RUDRATA (s,t)-PATH?

The reduction maps an `instance` G of RUDRATA (s,t)-PATH into an `instance` G' of RUDRATA CYCLE as follows: G' is G with an additional vertex x and two new edges $\{s,x\}$ and $\{x,t\}$.

RUDRATA (s,t)-PATH $\rightarrow$ RUDRATA CYCLE



RUDRATA (s,t)-PATH $\rightarrow$ RUDRATA CYCLE

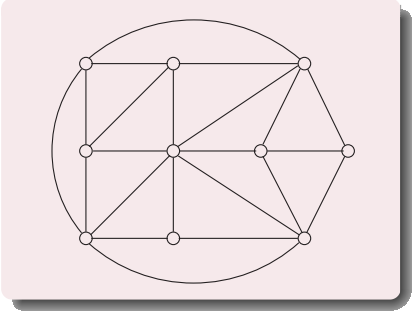


3SAT $\rightarrow$ INDEPENDENT SET

The instances of 3SAT, is set of clauses, each with three or fewer literals.

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

Independent Set



INDEPENDENT SET: Given a graph G and an integer g, find g vertices, no two of which have an edge between them.

True Assignment

To form a satisfying truth assignment we must pick one literal from each clause and give it the value `true`.

The choices must be consistent, if we choose $\overline{x}$ in one clause, we cannot choose x in another.

Solution: put an edge between any two vertices that correspond to opposite literals.

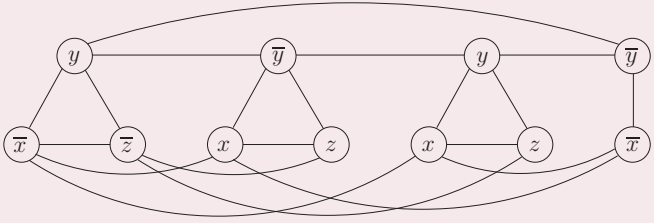
Represent a clause, say $(x \vee \overline{y} \vee z)$, by a triangle, with vertices labeled $x, \overline{y}, z$.

Because a triangle has its three vertices maximally connected, and thus forces to pick only one of them for the independent set.

Given an instance I of 3SAT, create an instance (G, g) of INDEPENDENT SET as follows,

- A triangle for each clause, with vertices labeled by the clause's literals.
- Additional edges between any two vertices that represent opposite literals.
- The goal g is set to the number of clauses.

3SAT → Independent set



$$(\overline{x} \vee y \vee \overline{z})(x \vee \overline{y} \vee z)(x \vee y \vee z)(\overline{x} \vee \overline{y})$$

$\text{SAT} \rightarrow \text{3SAT}$

This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an `instance` I of SAT, use exactly the same `instance` for 3SAT, except that any clause with more than three literals,

$$(a_1 \vee a_2 \vee \ldots \vee a_k)$$

is replaced by a set of clauses,

$$(a_1 \vee a_2 \vee y_1)(\overline{y_1} \vee a_3 \vee y_2)(\overline{y_2} \vee a_4 \vee y_3)\ldots(\overline{y_{k-3}} \vee a_{k-1} \vee a_k)$$

where the y_i's are new variables.

The reduction is in polynomial and I' is equivalent to I in terms of satisfiability.

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Suppose that the clauses on the right are all satisfied. Then at least one of the literals $a_1, \ldots, a_k$ must be `true`. Otherwise y_1 would have to be `true`, which would in turn force y_2 to be `true`, and so on.

Conversely, if $(a_1 \vee a_2 \vee \ldots \vee a_k)$ is satisfied, then some a_i must be `true`. Set $y_1, \ldots, y_{i-2}$ to `true` and the rest to `false`.

3SAT remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the 3SAT `instance`, variable x appears in $k > 3$ clauses. Then replace its first appearance by x_1, its second by x_2, and so on, replacing each of its k appearances by a different new variable.

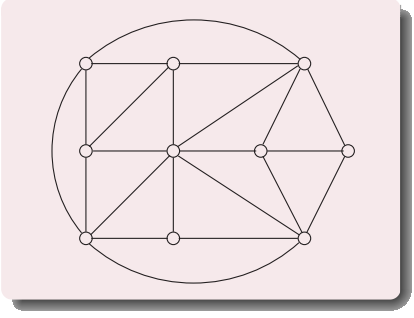
Finally, add the clauses

$$(\overline{x_1} \vee x_2)(\overline{x_2} \vee x_3) \ldots (\overline{x_k} \vee x_1)$$

In the new formula no variable appears more than three times (and in fact, no literal appears more than twice).

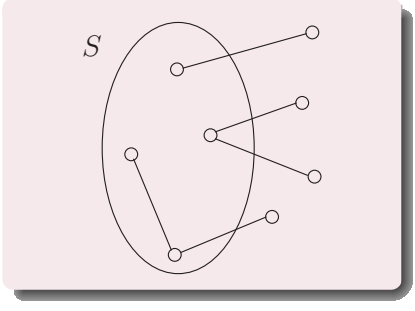
INDEPENDENT SET $\rightarrow$ VERTEX COVER

Vertex Cover



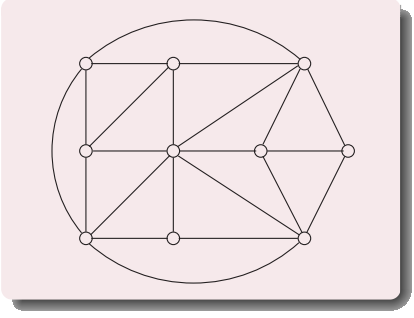
VERTEX COVER: Given a graph G and an integer b, find b vertices cover (touch) every edge.

A set of nodes S is a vertex cover of graph $G = (V, E)$ iff the remaining nodes, $V - S$, are an independent set of G.



INDEPENDENT SET $\rightarrow$ CLIQUE

Clique



CLIQUE: Given a graph G and an integer g, find g vertices such that all possible edges between them are present.

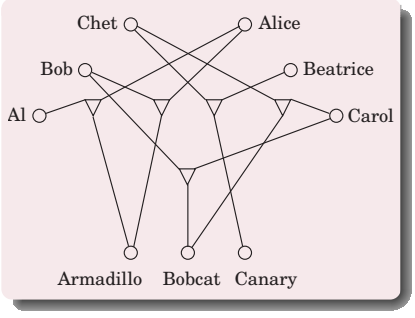
The complement of a graph $G = (V, E)$ is $\overline{G} = (V, \overline{E})$, where $\overline{E}$ contains precisely those unordered pairs of vertices that are not in E. A set of nodes S is an independent set of G iff S is a clique of $\overline{G}$.

Therefore, we can reduce INDEPENDENT SET to CLIQUE by mapping an instance (G, g) of INDEPENDENT SET to the corresponding instance $(\overline{G}, g)$ of CLIQUE.

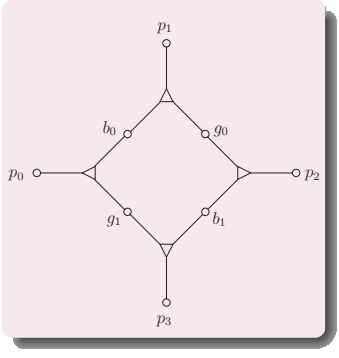
3SAT $\rightarrow$ 3D MATCHING

Three-Dimensional Matching

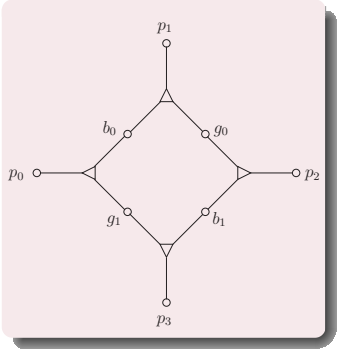
3D MATCHING: There are n boys, n girls, and n pets. The compatibilities are specified by a set of triples, each containing a boy, a girl, and a pet.
A triple (b, g, p) means that boy b, girl g, and pet p get along well together.
To find n disjoint triples and thereby create n harmonious households.



Consider a set of four triples, each represented by a triangular node joining a boy, girl, and pet.
Any matching must contain either the two triples $(b_0, g_1, p_0), (b_1, g_0, p_2)$ or $(b_0, g_0, p_1), (b_1, g_1, p_3)$.



Therefore, this “gadget” has two possible states: it behaves like a Boolean variable.
Transform an `instance` of 3SAT to one of 3D MATCHING, by creating a gadget for each variable x.



For each clause c introduce a new boy b_c and a new girl g_c.

E.g., $c = (x \vee \overline{y} \vee z)$, b_c, g_c will be involved in three triples, one for each literal in the clause.

And the pets in these triples must reflect the three ways whereby the clause can be satisfied:

1. $x = \texttt{true}$,
2. $y = \texttt{false}$,
3. $z = \texttt{true}$.

For $x = \texttt{true}$, we have the triple (b_c, g_c, p_{x1}), where p_{x1} is the pet p_1 in the gadget for x.

- If $x = \texttt{true}$, then b_{x0} is matched with g_{x1} and b_{x1} with g_{x0}, and so pets p_{x0} and p_{x2} are taken.
- If $x = \texttt{false}$, then p_{x1} and p_{x3} are taken, and so g_c and b_c cannot be accommodated.

We do the same thing for the other two literals, which yield triples involving b_c and g_c with either p_{y0} or p_{y2} and with either p_{z1} or p_{z3}.

We have to make sure that for every occurrence of a literal in a clause c there is a different pet to match with b_c and g_c.

This is easy: an earlier reduction guarantees that no literal appears more than twice, and so each variable gadget has enough pets, two for negated occurrences and two for positive.

The last problem remains: in the matching defined so far, some pets may be left unmatched.

If there are n variables and m clauses, then $2n - m$ pets will be left unmatched.

Add $2n - m$ new boy-girl couples that are “generic animal-lovers”, and match them by triples with all the pets!

3D MATCHING $\rightarrow$ ZOE

Zero-One Equations

ZOE

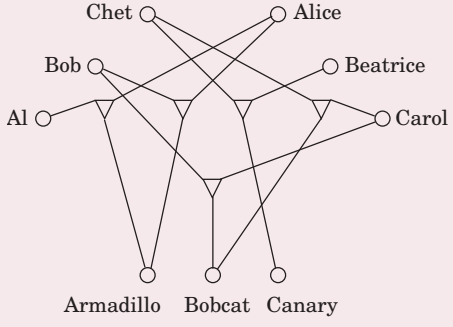
Given an $m \times n$ matrix A with $0-1$ entries, and find a $0-1$ vector $\mathbf{x} = (x_1, \ldots, x_n)$ such that the m equations $A\mathbf{x} = 1$; are satisfied.

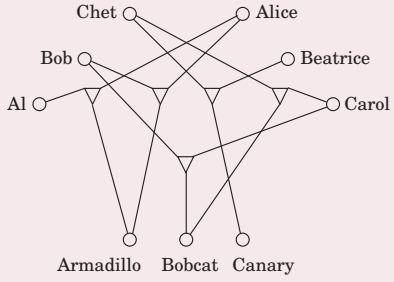
Assume in 3D MATCHING, there are m boys, m girls, m pets, and n boy-girl-pet triples.

We have $0-1$ variables, $x_1, \ldots, x_n$, one per triple, where $x_i = 1$ means that the i-th triple is chosen for the matching, and $x_i = 0$ means that it is not chosen.

For each boy, girl, or pet, suppose that the triples containing him (or her, or it) are those numbered $j_1, j_2, \ldots, j_k$; the appropriate equation is then

$$x_{j_1} + x_{j_2} + \ldots + x_{j_k} = 1$$


Chet
Alice
Bob
Beatrice
Al
Carol
Armadillo
Bobcat
Canary



$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

ZOE $\rightarrow$ SUBSET SUM

Subset sum

Subset sum: Find a subset of a given set of integers that adds up to exactly W.

This is a reduction between two special cases of ILP:

- One with many equations but only $0-1$ coefficients;
- The other with a single equation but arbitrary integer coefficients.

The reduction is based on a simple and time-honored idea: $0-1$ vectors can encode numbers!

If the columns is regarded as binary integers, a subset of the integers corresponds to the columns of A that add up to the binary integer $11\ldots1$.

This is an `instance` of SUBSET SUM. The reduction seems complete!

An Example

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Except for one detail: carry.

E.g., 5-bit binary integers can add up to $11111 = 31$, for example, $5 + 6 + 20 = 31$ or, in binary,

$$00101 + 00110 + 10100 = 11111$$

even when the sum of the corresponding vectors is not $(1, 1, 1, 1, 1)$.

Solution: The column vectors not as integers in base 2, but as integers in base $n + 1$, one more than the number of columns.

At most n integers are added, and all their digits are 0 and 1 There is no carry anymore.

ZOE $\rightarrow$ ILP

Special Cases

3SAT is a special case of SAT, or, SAT is a generalization of 3SAT.

By special case we mean that the `instances` of 3SAT are a subset of the `instances` of SAT.

There is a reduction from 3SAT to SAT, where the input has no transformation, and the solution to the target instance also kept unchanged.

A useful and common way of establishing that a problem is NP-complete: it is a generalization of a known NP-complete problem.

E.g., the SET COVER problem is NP-complete because it is a generalization of VERTEX COVER.

In ILP we are looking for an integer vector $\mathbf{x}$ that satisfies $A\mathbf{x} \leq b$, for given matrix A and vector b.

To write an `instance` of ZOE in this precise form, we need to rewrite each equation of the ZOE `instance` as two inequalities, and to add for each variable x_i the inequalities $x_i \leq 1$ and $-x_i \leq 0$.

ZOE $\rightarrow$ RUDRATA CYCLE

In RUDRATA CYCLE, seek a cycle in a graph that visits every vertex exactly once.

In ZOE, given an $m \times n$ matrix A with $0-1$ entries, and find a $0-1$ vector $\mathbf{x} = (x_1, \ldots, x_n)$ such that the m equations $A\mathbf{x} = 1$; are satisfied.

We will prove it NP-complete in two stages:

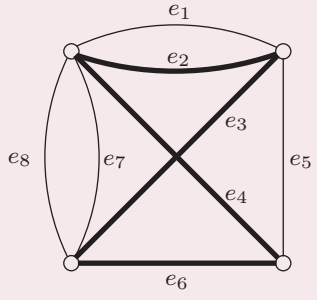
1. Firstly, reduce ZOE to a generalization of RUDRATA CYCLE, called RUDRATA CYCLE WITH PAIRED EDGES.
2. Secondly, get rid of the extra features of that problem and reduce it to the plain RUDRATA CYCLE.

Given a graph $G = (V, E)$ and a set $C \subseteq E \times E$ of pairs of edges. Find a cycle that,

1. visits all vertices once,
2. for every pair of edges (e, e') in C, traverses either edge e or edge e' exactly one of them.

Notice that two or more parallel edges between two nodes are allowed.

An Example



$$C = \{(e_1, e_3), (e_5, e_6), (e_4, e_5), (e_3, e_7), (e_3, e_8)\}$$

ZOE $\rightarrow$ RUDRATA CYCLE WITH PAIRED EDGES

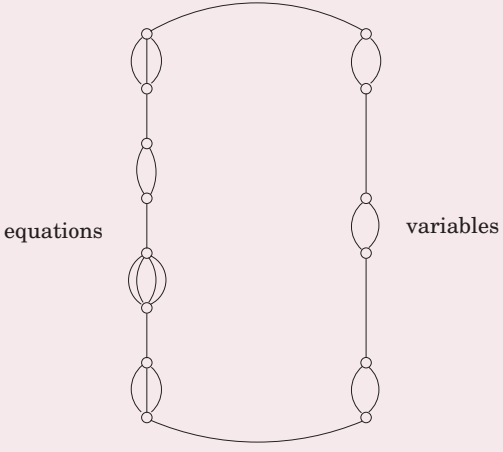
Given an `instance` of ZOE, $A\mathbf{x} = 1$, where A is an $m \times n$ matrix with $0 - 1$ entries, the graph is as follows

- A cycle that connects $m + n$ collections of parallel edges.
- Each variable x_i has two parallel edges, for $x_i = 1$ and $x_i = 0$).
- Each equation $x_{j_1} + \ldots + x_{j_k} = 1$ involving k variables has k parallel edges, one for every variable appearing in the equation.

Any RUDRATA CYCLE traverses the $m + n$ collections of parallel edges one by one, choosing one edge from each collection.

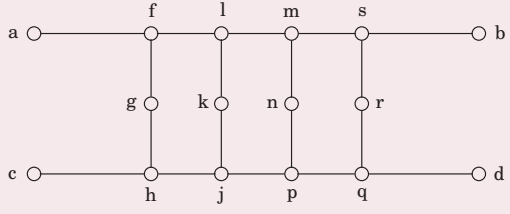
The cycle "chooses" for each variable a value 0 or 1 and, for each equation, a variable appearing in it.

ZOE $\rightarrow$ Rudrata Cycle with Paired Edges



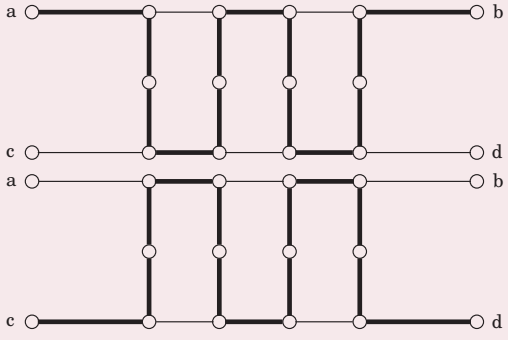
Get Rid of Edge Pairs

Consider the graph, and suppose it is a part of a larger graph G in such a way that only the four endpoints a, b, c, d touch the rest of the graph.



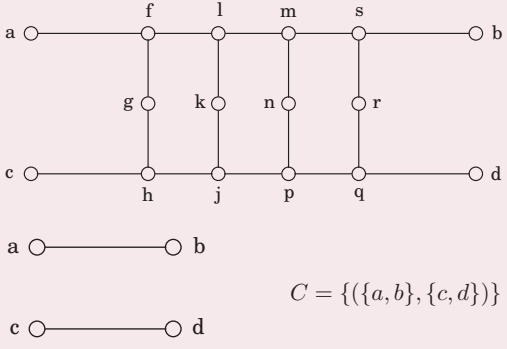
Get Rid of Edge Pairs

We claim that this graph has the following important property: in any Rudrata cycle of G the subgraph shown must be traversed in one of the two ways.



Get Rid of Edge Pairs

This gadget behaves just like two edges $\{a, b\}$ and $\{c, d\}$ that are paired up in the Rudrata cycle with paired edges.



$$C = \{(\{a, b\}, \{c, d\})\}$$

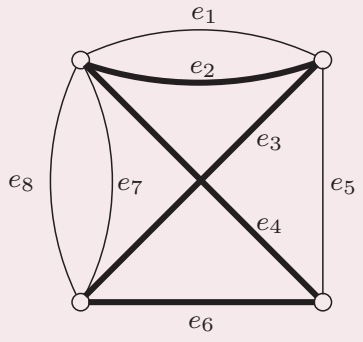
Go through the pairs in C one by one. To get rid of each pair $(\{a, b\}, \{c, d\})$ by replacing the two edges with the gadget.

For any other pair in C that involves $\{a, b\}$, replace the edge $\{a, b\}$ with the new edge $\{a, f\}$, where f is from the gadget.

Similarly, $\{c, h\}$ replaces $\{c, d\}$.

The RUDRATA CYCLES in the resulting graph will be in one-to-one correspondence with the RUDRATA CYCLES in the original graph that conform to the constraints in C.

An Example



$$C = \{(e_1, e_3), (e_5, e_6), (e_4, e_5), (e_3, e_7), (e_3, e_8)\}$$

RUDRATA CYCLE $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the `instance` of the TSP:

- The set of nodes is the same as V.
- The distance between cities u and v is 1 if $\{u, v\}$ is an edge of G and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP `instance` is $|V|$.

If G has a RUDRATA CYCLE, then the same cycle is also a tour within the budget of the TSP `instance`.

If G has no RUDRATA CYCLE, then there is no solution: the cheapest possible TSP tour has cost at least $n + \alpha$.

If $\alpha = 1$, then all distances are either 1 or 2, and so this `instance` of the TSP satisfies the triangle inequality: if i, j, k are cities, then

$$d_{ij} + d_{jk} \geq d_{ik}$$

This is a special case of the TSP which is in a certain sense easier, since it can be efficiently approximated.

If α is large, then the resulting `instance` of the TSP may not satisfy the triangle inequality, and has another important property.

This important gap property implies that, unless P = NP, no approximation algorithm is possible.

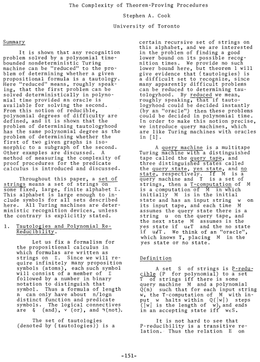
ANY PROBLEM $\rightarrow$ SAT

home reading!

The “first” NP-complete problem

Theorem (Cook 1971, Levin 1973)

*SAT is **NP**-complete.*



The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto

Summary

It is shown that any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be "reduced" to the problem of determining whether a given propositional formula is a tautology. Here "reduced" means, roughly speaking, that the first problem can be solved deterministically in polynomial time provided an oracle is available for solving the second. From this notion of reducible, polynomial degrees of difficulty are defined, and it is shown that the problem of determining tautologyhood has the same polynomial degree as the problem of determining whether the first of two given graphs is isomorphic to a subgraph of the second. Other examples are discussed. A method of measuring the complexity of proof procedures for the predicate calculus is introduced and discussed.

Throughout this paper, a set of strings means a set of strings on some fixed, large, finite alphabet Σ. This alphabet is large enough to include symbols for all sets described here. All Turing machines are deterministic recognition devices, unless the contrary is explicitly stated.

1. Tautologies and Polynomial Re-Reducibility.

Let us fix a formalism for the propositional calculus in which formulas are written as strings on Σ. Since we will require infinitely many proposition symbols (atoms), each such symbol will consist of a member of Σ followed by a number in binary notation to distinguish that symbol. Thus a formula of length n can only have about n/logn distinct function and predicate symbols. The logical connectives are & (and), ∨ (or), and ¬(not).

The set of tautologies (denoted by {tautologies}) is a certain recursive set of strings on this alphabet, and we are interested in the problem of finding a good lower bound on its possible recognition times. We provide no such lower bound here, but theorem 1 will give evidence that {tautologies} is a difficult set to recognize, since many apparently difficult problems can be reduced to determining tautologyhood. By reduced we mean, roughly speaking, that if tautologyhood could be decided instantly (by an "oracle") then these problems could be decided in polynomial time. In order to make this notion precise, we introduce query machines, which are like Turing machines with oracles in [1].

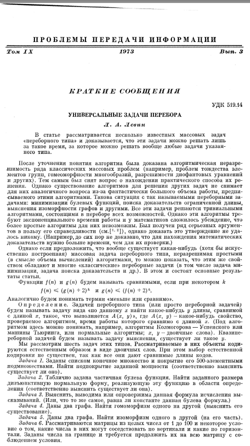
A query machine is a multitape Turing machine with a distinguished tape called the query tape, and three distinguished states called the query state, yes state, and no state, respectively. If M is a query machine and T is a set of strings, then a T-computation of M is a computation of M in which initially M is in the initial state and has an input string w on its input tape, and each time M assumes the query state there is a string u on the query tape, and the next state M assumes is the yes state if u∈T and the no state if u∉T. We think of an "oracle", which knows T, placing M in the yes state or no state.

Definition

A set S of strings is P-reducible (P for polynomial) to a set T of strings iff there is some query machine M and a polynomial Q(n) such that for each input string w, the T-computation of M with input w halts within Q(|w|) steps (|w| is the length of w), and ends in an accepting state iff w∈S.

It is not hard to see that P-reducibility is a transitive relation. Thus the relation E on

-151-



ПРОБЛЕМЫ ПЕРЕДАЧИ ИНФОРМАЦИИ

Том IX — 1973 — Вып. 3

КРАТКИЕ СООБЩЕНИЯ

УДК 519.14

УНИВЕРСАЛЬНЫЕ ЗАДАЧИ ПЕРЕБОРА

Л. А. Левин

В статье рассматривается несколько известных массовых задач «переборного типа» и доказывается, что эти задачи можно решать лишь за такое время, за которое можно решать вообще любые задачи указанного типа.

После уточнения понятия алгоритма была доказана алгоритмическая неразрешимость ряда классических массовых проблем (например, проблем тождества элементов групп, гомеоморфности многообразий, разрешимости диофантовых уравнений и других). Тем самым был снят вопрос о нахождении практического способа их решения. Однако существование алгоритмов для решения других задач не снимает для них аналогичного вопроса из-за фантастически большого объема работы, предписываемого этими алгоритмами. Такова ситуация с так называемыми переборными задачами: минимизации булевых функций, поиска доказательств ограниченной длины, выяснения изоморфности графов и другими. Все эти задачи решаются тривиальными алгоритмами, состоящими в переборе всех возможностей. Однако эти алгоритмы требуют экспоненциального времени работы и у математиков сложилось убеждение, что более простые алгоритмы для них невозможны. Был получен ряд серьезных аргументов в пользу его справедливости (см.[1, 2]), однако доказать это утверждение не удалось никому. (Например, до сих пор не доказано, что для нахождения математических доказательств нужно больше времени, чем для их проверки.)

Однако если предположить, что вообще существует какая-нибудь (хотя бы искусственно построенная) массовая задача переборного типа, неразрешимая простыми (в смысле объема вычислений) алгоритмами, то можно показать, что этим же свойством обладают и многие «классические» переборные задачи (в том числе задача минимизации, задача поиска доказательств и др.). В этом и состоят основные результаты статьи.

Функции $f(n)$ и $g(n)$ будем называть сравнимыми, если при некотором k

$$f(n) \leqslant (g(n)+2)^k \text{ и } g(n) \leqslant (f(n)+2)^k.$$

Аналогично будем понимать термин «меньше или сравнимо».

Определение. Задачей переборного типа (или просто переборной задачей) будем называть задачу вида «по данному x найти какое-нибудь y длины, сравнимой с длиной x, такое, что выполняется $A(x, y)$», где $A(x, y)$ – какое-нибудь свойство, проверяемое алгоритмом, время работы которого сравнимо с длиной x. (Под алгоритмом здесь можно понимать, например, алгоритмы Колмогорова — Успенского или машины Тьюринга, или нормальные алгоритмы; x, y – двоичные слова). Квазипереборной задачей будем называть задачу выяснения, существует ли такое y.

Мы рассмотрим шесть задач этих типов. Рассматриваемые в них объекты кодируются естественным образом в виде двоичных слов. При этом выбор естественной кодировки не существен, так как все они дают сравнимые длины кодов.

Задача 1. Заданы списком конечное множество и покрытие его 500-элементными подмножествами. Найти подпокрытие заданной мощности (соответственно выяснить существует ли оно).

Задача 2. Таблично задана частичная булева функция. Найти заданного размера дизъюнктивную нормальную форму, реализующую эту функцию в области определения (соответственно выяснить существует ли она).

Задача 3. Выяснить, выводима или опровержима данная формула исчисления высказываний. (Или, что то же самое, равна ли константе данная булева формула.)

Задача 4. Даны два графа. Найти гомоморфизм одного на другой (выяснить его существование).

Задача 5. Даны два графа. Найти изоморфизм одного в другой (на его часть).

Задача 6. Рассматриваются матрицы из целых чисел от 1 до 100 и некоторое условие о том, какие числа в них могут соседствовать по вертикали и какие по горизонтали. Заданы числа на границе и требуется продолжить их на всю матрицу с соблюдением условия.

Algorithm Design XVIII

Coping with NP-Completeness I: Approximation Algorithm

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-complete problem. What should I do?

A. Theory says you're unlikely to find polynomial time algorithm.

Must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

Think About: What features have been sacrificed in today's topic?

Approximation Algorithms

Optimization Problems

In an optimization problem we are given an instance I and are asked to find the optimum solution:

- The one with the maximum gain if we have a maximization problem like Independent Set,or
- The minimum cost if we are dealing with a minimization problem such as the Tsp.

For every instance I , $OPT(I)$ denotes the value (benefit or cost) of the optimum solution.

We always assume $OPT(I)$ is a positive integer (we may write as OPT when context is clear).

Approximation Ratio

We have seen the greedy algorithm for Set Cover:

For any instance I of size n, this greedy algorithm quickly finds a set cover of cardinality at most $OPT(I) \cdot \log n$.

This $\log n$ factor is known as the approximation guarantee of the algorithm.

Suppose now that we have an algorithm A for a minimization problem which, given an instance I, returns a solution with value $\mathcal{A}(I)$.

The approximation guarantee of $\mathcal{A}$ is defined to be

$$\alpha_{\mathcal{A}} = \max_{I} \frac{\mathcal{A}(I)}{OPT(I)}$$

To establish the approximation guarantee, the cost of the solution produced by the algorithm needs to compare with an optimal solution.

For such problems, not only is it NP-hard to find an optimal solution, but it is also NP-hard to compute the cost of an optimal solution.

In fact, computing the cost of an optimal solution is precisely the difficult core of such problems.

How do we establish the approximation guarantee? The answer provides a key step in the design of approximation algorithms.

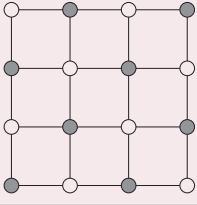
Vertex Cover

VERTEX COVER

Input: An undirected graph $G = (V, E)$.

Output: A subset of the vertices $S \subseteq V$ that touches every edge.

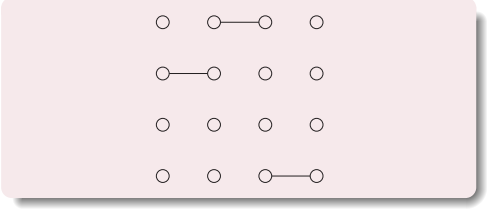
Goal: Minimize $|S|$.

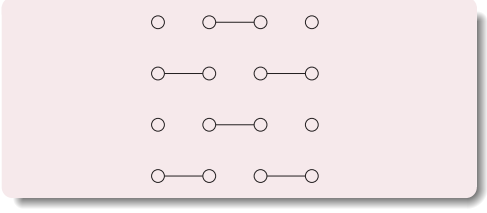


Given a graph $G = (V, E)$, a subset of the edges $M \subseteq E$ is said to be a matching if no two edges of M share an endpoint.

A matching of maximum cardinality in G is called a maximum matching.

A matching that is maximal under inclusion is called a maximal matching.





Matching

Given a graph $G = (V, E)$, a subset of the edges $M \subseteq E$ is said to be a matching if no two edges of M share an endpoint.

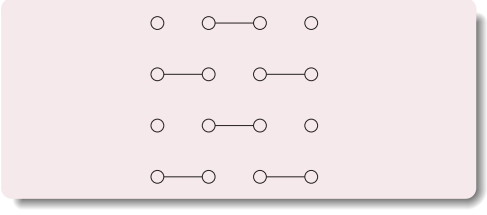
A matching of maximum cardinality in G is called a maximum matching.

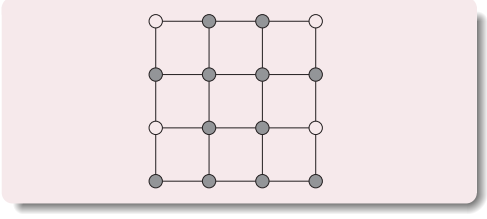
A matching that is maximal under inclusion is called a maximal matching.

A maximal matching can clearly be computed in polynomial time by simply greedily picking edges and removing endpoints of picked edges. More sophisticated means lead to polynomial time algorithms for finding a maximum matching as well.

Algorithm

Find a maximal matching in G and output the set of matched vertices.





Approximation Guarantee Factor

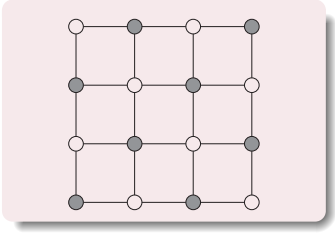
The Algorithm is a factor 2 approximation algorithm for the vertex cover problem.

Proof.

- No edge can be left uncovered by the set of vertices picked.
- Let M be the matching picked. As argued above,

$$|M| \leq OPT$$

- The approximation factor is at most $2 \cdot OPT$.



Lower Bounding OPT

The approximation algorithm for vertex cover was very much related to, and followed naturally from, the lower bounding scheme. This is in fact typical in the design of approximation algorithms.

Can the Guarantee be Improved?

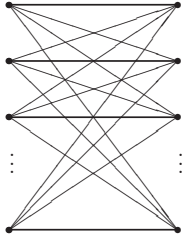
Can the approximation guarantee of Algorithm be improved by a better analysis?

Can an approximation algorithm with a better guarantee be designed using the lower bounding scheme of Algorithm?

Is there some other lower bounding method that can lead to an improved approximation guarantee for VERTEX COVER?

A Better Analysis?

Consider the infinite family of instances given by the complete bipartite graphs $K_{n,n}$.



When run on $K_{n,n}$, Algorithm will pick all $2n$ vertices, whereas picking one side of the bipartition gives a cover of size n.

A Better Guarantee?

The lower bound, of size of a maximal matching, is half the size of an optimal vertex cover for the following infinite family of instances. Consider the complete graph K_n, where n is odd. The size of any maximal matching is $(n-1)/2$, whereas the size of an optimal cover is $n-1$.

A Better Algorithm?

Still Open!

On the hardness of approximating minimum vertex cover

By Irit Dinur and Samuel Safra*

Abstract

We prove the Minimum Vertex Cover problem to be NP-hard to approximate to within a factor of 1.3606, extending on previous PCP and hardness of approximation technique. To that end, one needs to develop a new proof framework, and to borrow and extend ideas from several fields.

Clustering

Euclidean Distance $d(x, y)$

1. $d(x, y) \geq 0$ for all x, y.
2. $d(x, y) = 0$ if and only if $x = y$.
3. $d(x, y) = d(y, x)$.
4. (Triangle inequality) $d(x, y) \leq d(x, z) + d(z, y)$.

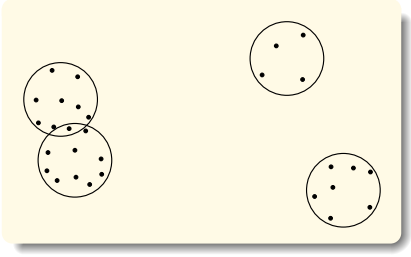
Definition (k-Cluster)

Input: Points $X = \{x_1, \ldots, x_n\}$ with underlying distance metric $d(\cdot, \cdot)$, integer k.

Output: A partition of the points into k clusters $C_1, \ldots, C_k$.

Goal: Minimize the diameter of the clusters,

$$\min_j \max_{x_a, x_b \in C_j} d(x_a, x_b)$$



Theorem

k-clustering is NP-hard.

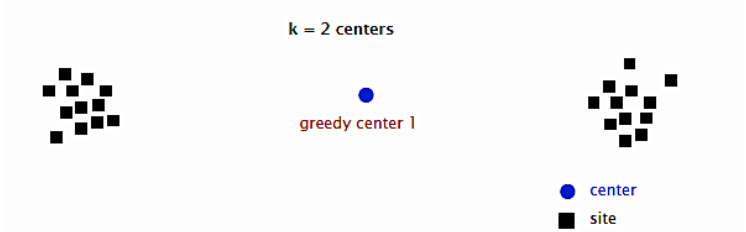
Remark

Search can be infinite!

Greedy algorithm. Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.

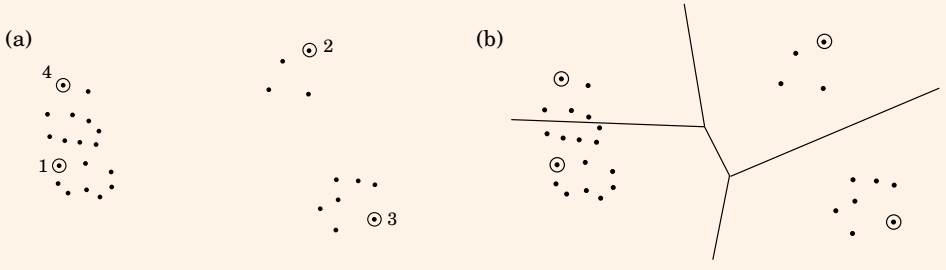
Remark

Greedy algorithm is arbitrarily bad!



Approximation Algorithm

Pick any point $\mu_1 \in X$ as the first cluster center
for $i = 2$ to k:
  Let μ_i be the point in X that is farthest from $\mu_1, \ldots, \mu_{i-1}$
  (i.e., that maximizes $\min_{j<i} d(\cdot, \mu_j)$)
Create k clusters: $C_i = \{$all $x \in X$ whose closest center is $\mu_i\}$



Proof for the Approximation Ratio 2

Let $x \in X$ be the point farthest from $\mu_1, \ldots, \mu_k$, and r be its distance to its closest center.

Then every point in X must be within distance r of its cluster center. By the triangle inequality, this means that every cluster has diameter at most $2r$.

We have identified $k+1$ points $\{\mu_1, \mu_2, \ldots, \mu_k, x\}$ that are all at a distance at least r from each other.

Any partition into k clusters must put two of these points in the same cluster and must therefore have diameter at least r.

Remark

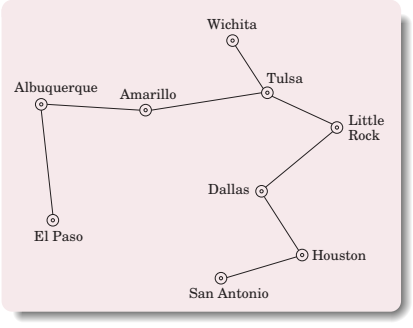
No better approximation algorithm for this problem so far.

TSP

TSP on Metric Space

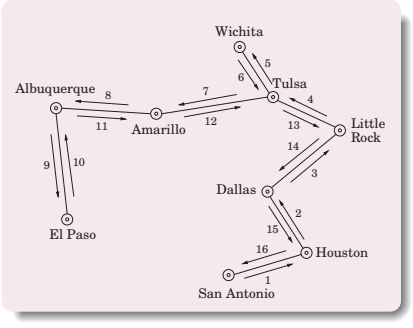
Removing any edge from a traveling salesman tour leaves a path through all the vertices, which is a spanning tree.

Therefore, TSP cost $\geq$ cost of this path $\geq$ MST cost



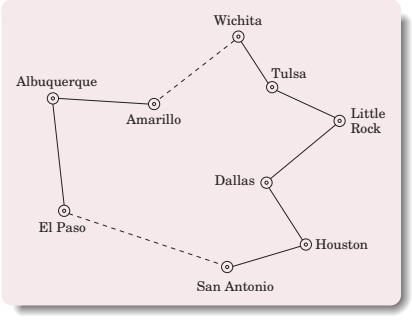
TSP on Metric Space

If we can use each edge twice, then by following the shape of the MST we end up with a tour that visits all the cities, some of them more than once.



To fix the problem, the tour should simply skip any city it is about to revisit, and instead move directly to the next new city in its list.

By the triangle inequality, these bypasses can only make the overall tour shorter.



Consider the following algorithm:

1. Find an MST, T of G
2. Double every edge of the MST to obtain an Eulerian graph.
3. Find an Eulerian tour, $\mathcal{T}$, on this graph.
4. Output the tour that visits vertices of G in the order of their first appearance in T. Let $\mathcal{C}$ be this tour.

The above algorithm is a factor 2 approximation algorithm for metric TSP.

Analysis

$cost(T) \leq$OPT.

Since $\mathcal{T}$ contains each edge of T twice, $\mathrm{cost}(\mathcal{T}) = 2 \cdot \mathrm{cost}(T)$.

Because of triangle inequality, after the short-cutting (step 4) step, $\mathrm{cost}(\mathcal{C}) \leq \mathrm{cost}(\mathcal{T})$.

Combining these inequalities we get that

$$\mathrm{cost}(\mathcal{C}) \leq 2 \cdot \mathrm{OPT}.$$

What if we are interested in instances of TSP that do not satisfy the triangle inequality?

It turns out that this is a much harder problem to approximate.

Recall we gave a polynomial-time reduction which given any graph G and integer any $C > 0$ produces an instance $I(G, C)$ of the TSP such that:

1. If G has a Rudrata cycle then $OPT(I(G, C)) = n$, the number of vertices in G.
2. If G has no Rudrata cycle, then $OPT(I(G, C)) \geq n + C$.

This means that even an approximate solution to TSP would enable us to solve RUDRATA CYCLE.

Consider an approximation algorithm $\mathcal{A}$ for TSP and let $\alpha_{\mathcal{A}}$ denote its approximation ratio.

From any instance G of Rudrata Cycle, we will create an instance $I(G, C)$ of Tsp using the specific constant

$$C = n \cdot \alpha_{\mathcal{A}}$$

```
Given any graph G:
   compute I(G,C) (with C = n·α_A) and run algorithm A on it
   if the resulting tour has length ≤ nα_A:
      conclude that G has a Rudrata path
   else:  conclude that G has no Rudrata path
```

Knapsack

Recall the KNAPSACK problem: There are n items, with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ (all positive integers), and the goal is to pick the most valuable combination of items subject to the constraint that their total weight is at most W.

Earlier we saw a dynamic programming solution to this problem with running time $O(nW)$. Using a similar technique, a running time of $O(nV)$ can also be achieved, where V is the sum of the values.

Neither of these running times is polynomial, because W and V can be very large, exponential in the size of the input.

Let's consider the $O(nV)$ algorithm.

In the bad case when V is large, what if we simply scale down all the values in some way?

For instance, if

$$v_1 = 117,586,003, \qquad v_2 = 738,493,291, \qquad v_3 = 238,827,453$$

we could simply knock off some precision and instead use 117, 738, and 238.

This doesn't change the problem all that much and will make the algorithm much, much faster!

Along with the input, the user is assumed to have specified some approximation factor $\epsilon > 0$.

```
Discard any item with weight > W
Let v_max = max_i v_i
Rescale values v̂_i = ⌊v_i · n/(ε v_max)⌋
Run the dynamic programming algorithm with values {v̂_i}
Output the resulting choice of items
```

Since the rescaled values $\hat{v}_i$ are all at most n/ϵ, the dynamic program is efficient, running in time

$$O(n^3/\epsilon)$$

Suppose the optimal solution to the original problem is to pick some subset of items S, with total value K^*.

The rescaled value of this same assignment is

$$\sum_{i \in S} \hat{v}_i = \sum_{i \in S} \lfloor v_i \cdot \frac{n}{\epsilon \cdot v_{max}} \rfloor \geq \sum_{i \in S} (v_i \cdot \frac{n}{\epsilon \cdot v_{max}} - 1) \geq K^* \cdot \frac{n}{\epsilon \cdot v_{max}} - n$$

Therefore, the optimal assignment for the shrunken problem, call it $\hat{S}$, has a rescaled value of at least this much.

In terms of the original values, assignment $\hat{S}$ has a value of at least

$$\sum_{i \in \hat{S}} v_i \geq \sum_{i \in \hat{S}} \hat{v}_i \cdot \frac{\epsilon \cdot v_{max}}{n} \geq (K^* \cdot \frac{n}{\epsilon \cdot v_{max}} - n) \cdot \frac{\epsilon \cdot v_{max}}{n} = K^* - \epsilon \cdot v_{max} \geq K^*(1 - \epsilon)$$

The Approximability Hierarchy

The Approximability Hierarchy

All NP-complete optimization problems are classified as follows:

- Those for which, like the TSP, no finite approximation ratio is possible.
- Those for which an approximation ratio is possible, but there are limits to how small this can be: Vertex Cover, k-Cluster, and the TSP with triangle inequality.
- Down below we have a more fortunate class of NP-complete problems for which approximability has no limits, and polynomial approximation algorithms with error ratios arbitrarily close to zero exist: Knapsack.
- Finally, there is another class of problems, between the first two given here, for which the approximation ratio is about $\log n$: Set Cover.

Algorithm Design XIX

Coping with NP-Completeness II: Backtracking

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-complete problem. What should I do?

A. Theory says you're unlikely to find polynomial time algorithm.

Must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

Think About: What features have been sacrificed in today's topic?

Why SAT Solving

The First Example

Let $S = \{s_1, \ldots, s_n\}$ be a set of radio stations, each of which has to be allocated one of k transmission frequencies, for some $k < n$. Two stations that are too close to each other cannot have the same frequency. The set of pairs having this constraint is denoted by E, satisfying

- Every station is assigned at least one frequency.
- Every station is assigned not more than one frequency.
- Close stations are not assigned the same frequency.

Give solution to work out that whether k is enough for a given situation.

The Solution

Define a set of propositional variables

$$\{x_{ij} | i \in \{1, \ldots, n\}, j \in \{1, \ldots, k\}\}$$

Intuitively, variable x_{ij} is set to true if and only if station i is assigned the frequency j.

The Solution

Every station is assigned at least one frequency:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} x_{ij}$$

Every station is assigned not more than one frequency:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{k-1} (x_{ij} \rightarrow \wedge_{j<t\leq k} \neg x_{it})$$

Close stations are not assigned the same frequency. For each $(i, j) \in E$,

$$\bigwedge_{t=1}^{k} x_{it} \rightarrow \neg x_{jt}$$

The Second Example

Consider the two code fragments. The fragment on the right-hand side might have been generated from the fragment on the left-hand side by an optimizing compiler. We would like to check if the two programs are equivalent.

```
if(!a && !b) h();
else
    if(!a) g();
    else f();
```

```
if(a) f();
else
    if(b) g();
    else h();
```

The Solution

(if x then y else z) $\equiv (x \wedge y) \vee (\neg x \wedge z)$

$$(\neg a \wedge \neg b) \wedge h \vee \neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g \vee a \wedge f)$$
$$\leftrightarrow a \wedge f \vee \neg a \wedge (b \wedge g \vee \neg b \wedge h)$$

Q: Can a proportional formula be transformed into an equivalent CNF formula effectively?

A: It can, however, while potentially increasing the size of the formula exponentially.

The propositional formula can be transformed into an equisatisfiable CNF formula with only a linear increase in the size of the formula.

The price to be paid is n new Boolean variables, known as Tseitin's encoding.

Tseitin's Encoding

The Exponential Way

$CNF(\phi)\{$

case

- ϕ is a literal: return ϕ
- ϕ is $\varphi_1 \wedge \varphi_2$: return $CNF(\varphi_1) \wedge CNF(\varphi_2)$
- ϕ is $\varphi_1 \vee \varphi_2$: return $Dist(CNF(\varphi_1), CNF(\varphi_2))$

$\}$

$Dist(\varphi_1, \varphi_2)\{$

case

- φ_1 is $\psi_{11} \wedge \psi_{12}$: return $Dist(\psi_{11}, \varphi_2) \wedge Dist(\psi_{12}, \varphi_2)$
- φ_2 is $\psi_{21} \wedge \psi_{22}$: return $Dist(\varphi_1, \psi_{21}) \wedge Dist(\varphi_1, \psi_{22})$

$\}$

Consider the formula $\phi = (x_1 \wedge y_1) \vee (x_2 \wedge y_2)$

$$CNF(\phi) = (x_1 \vee x_2) \wedge (x_1 \vee y_2) \wedge (y_1 \vee x_2) \wedge (y_1 \vee y_2)$$

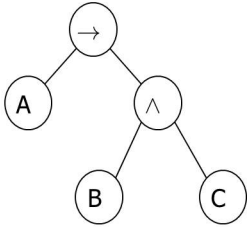
Now consider: $\phi_n = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \ldots \vee (x_n \wedge y_n)$

Q: How many clauses $CNF(\phi_n)$ returns?

A: 2^n

Consider the formula $(A \rightarrow (B \wedge C))$

The parse tree:

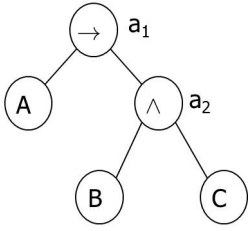


Associate a new auxiliary variable with each gate.

Add constraints that define these new variables.

Finally, enforce the root node.

$(a_1 \leftrightarrow (A \rightarrow a_2)) \wedge (a_2 \leftrightarrow (B \wedge C)) \wedge (a_1)$



Each such constraint has a CNF representation with 3 or 4 clauses.

First: $(a_1 \vee A) \wedge (a_1 \vee \neg a_2) \wedge (\neg a_1 \vee \neg A \vee a_2)$

Second: $(\neg a_2 \vee B) \wedge (\neg a_2 \vee C) \wedge (a_2 \vee \neg B \vee \neg C)$

Tseitin's Encoding

$$\phi_n = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \ldots \vee (x_n \wedge y_n)$$

With Tseitin's encoding we need:

- n auxiliary variables $a_1, \ldots, a_n$.
- Each adds 3 constraints.
- Top clause: $(a_1 \vee \ldots \vee a_n)$

Hence, we have

- $3n + 1$ clauses, instead of 2^n.
- $3n$ variables rather than $2n$.

Methodologies

Two Usual Ways to Implement

Exhaustive Search (DPLL Algorithm): traversing and backtracking on a binary tree.

Stochastic Search: guessing a full assignment, and flipping values of variables according to some heuristic.

A Brief History

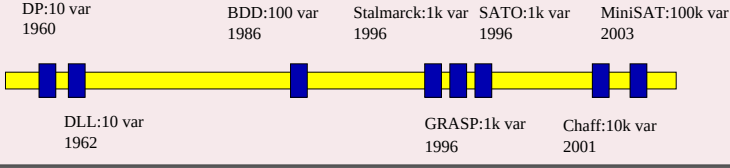
Originally, DPLL was incomplete method for SAT in FO logic

First paper (Davis and Putnam) in 1960: memory problems

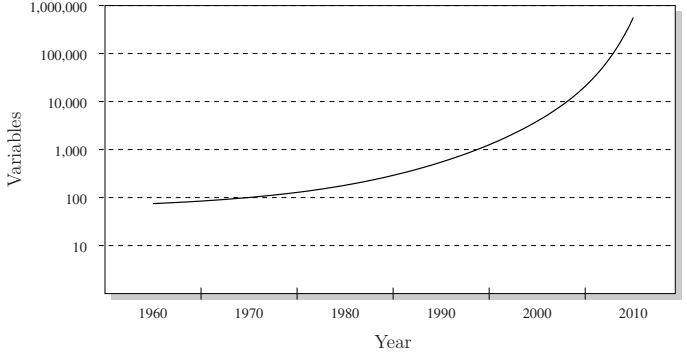
Second paper (Davis, Logemann and Loveland) in 1962: Depth-first-search with backtracking

Late 90's and early 00's improvements make DPLL efficient:

Break-through systems: GRASP, SATO, zChaff, MiniSAT, Z3



A Brief History



Backtracking

It is often possible to reject a solution by looking at just a small portion of it.

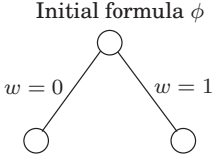
For example, if an `instance` of SAT contains the clause $(x_1 \vee x_2)$, then all assignments with $x_1 = x_2 =$ `false` can be instantly eliminated.

To put it differently, by quickly checking and discrediting this partial assignment, we are able to prune a quarter of the entire search space.

A promising direction, but can it be systematically exploited?

An Example

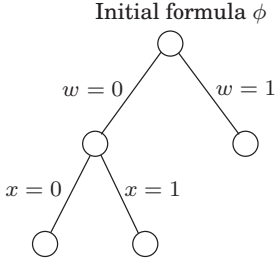
$$(w \vee x \vee y \vee z)(w \vee \overline{x})(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{w})(\overline{w} \vee \overline{z})$$



Plugging $w = 0$ and $w = 1$ into Φ, we find that no clause is immediately violated and thus neither of these two partial assignments can be eliminated outright.

An Example

$$\Phi = (w \vee x \vee y \vee z)(w \vee \overline{x})(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{w})(\overline{w} \vee \overline{z})$$



The partial assignment $w = 0, x = 1$ violates the clause $(w \vee \overline{x})$ and can be terminated, thereby pruning a good chunk of the search space.

An Example

$$\Phi = (w \vee x \vee y \vee z)(w \vee \overline{x})(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{w})(\overline{w} \vee \overline{z})$$

Backtracking explores the space of assignments, only growing the tree only at nodes where there is uncertainty.

Each node of the search tree can be described either by a partial assignment or by the clauses that remain.

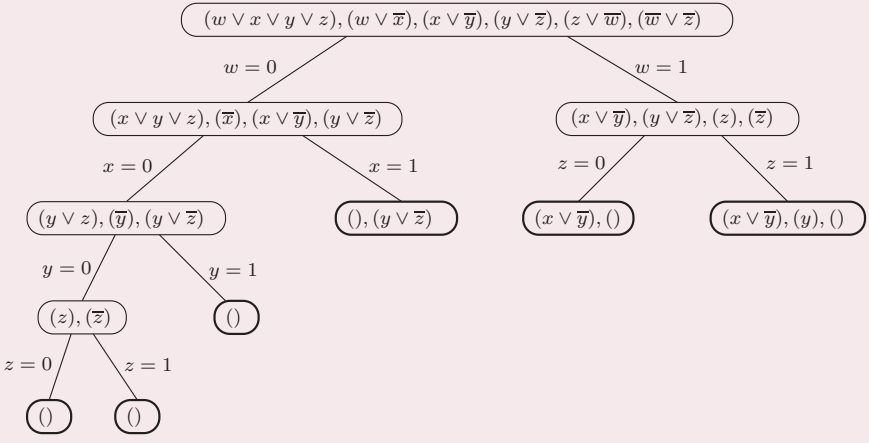
If $w = 0$ and $x = 0$ then any clause with w or x is instantly satisfied and any literal $\overline{w}$ or $\overline{x}$ is not satisfied and can be removed.

What's left is

$$(y \vee z)(\overline{y})(y \vee \overline{z})$$

Thus the nodes of the search tree, representing partial assignments, are themselves SAT subproblems.

An Example


$(w \vee x \vee y \vee z), (w \vee \overline{x}), (x \vee \overline{y}), (y \vee \overline{z}), (z \vee \overline{w}), (\overline{w} \vee \overline{z})$
$w = 0$
$w = 1$
$(x \vee y \vee z), (\overline{x}), (x \vee \overline{y}), (y \vee \overline{z})$
$(x \vee \overline{y}), (y \vee \overline{z}), (z), (\overline{z})$
$x = 0$
$x = 1$
$z = 0$
$z = 1$
$(y \vee z), (\overline{y}), (y \vee \overline{z})$
$(), (y \vee \overline{z})$
$(x \vee \overline{y}), ()$
$(x \vee \overline{y}), (y), ()$
$y = 0$
$y = 1$
$(z), (\overline{z})$
$()$
$z = 0$
$z = 1$
$()$
$()$

Basic Functions

`Decide()`: Choose the next variable and value. Return False if all variables are assigned.

`BCP()`: Apply repeatedly the unit clause rule. Return False if reached a conflict.

`Resolve-conflict()`: Backtrack until no conflict. Return False if impossible.

Algorithm

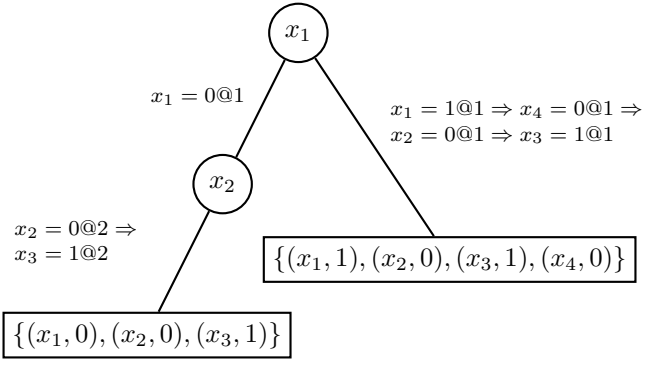

```
SAT()
while true do
    if ¬ Decide () then
        return true ;
    end
    else
        while ¬ BCP () do
            if ¬ Resolve-conflict () then return false;
        end
    end
end
```

Organize the search in the form of a decision tree

- Each node corresponds to a decision.
- Definition: Decision Level (DL) is the depth of the node in the decision tree.
- Notation: $x = v@d$, where $x \in \{0, 1\}$ is assigned to v at decision level d.

Backtracking Search in Action

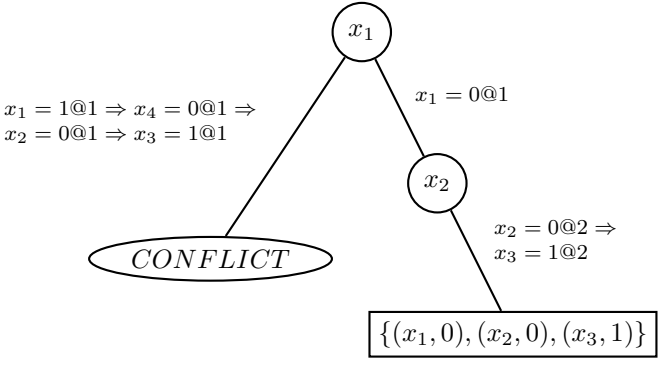
$$(x_2 \vee x_3), (\neg x_1 \vee, \neg x_4), (\neg x_2 \vee x_4)$$



No backtrack in this example, regardless of the decision!

Backtracking Search in Action

$$(x_2 \vee x_3), (\neg x_1 \vee, \neg x_4), (\neg x_2 \vee x_4), (\neg x_1 \vee x_2 \vee \neg x_3)$$



Status of a Clause

A clause can be

- Satisfied: at least one literal is satisfied
- Unsatisfied: all literals are assigned but non are satisfied
- Unit: all but one literals are assigned but none are satisfied
- Unresolved: all other cases

Example: $C = (x_1 \vee x_2 \vee x_3)$

x_1	x_2	x_3	C
1	0		Satisfied
0	0	0	Unsatisfied
0	0		Unit
	0		Unresolved

DLIS (Dynamic Largest Individual Sum)

Choose the assignment that increases the most the number of satisfied clauses.

For a given variable x:

- C_{xp}: ♯ unresolved clauses in which x appears positively
- C_{xn}: ♯ unresolved clauses in which x appears negatively
- Let x be the literal for which C_{xp} is maximal
- Let y be the literal for which C_{yn} is maximal
- If $C_{xp} > C_{yn}$ choose x and assign it TRUE
- Otherwise choose y and assign it FALSE

Requires l (♯ literals) queries for each decision.

Jeroslow-Wang

Compute for every clause w and every literal l in each phase

$$J(l) = \sum_{l \in w, w \in \varphi} 2^{-|w|}$$

where $|w|$ the length.

Choose the literal l that maximizes $J(l)$.

This gives an exponentially higher weight to literals in shorter clauses.

Next

We will see other (more advanced) decision Heuristics soon.

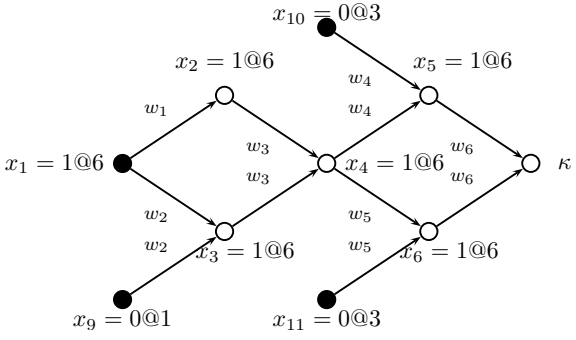
These heuristics are integrated with a mechanism called Learning with Conflict-Clauses, which we will learn next.

Learning New Clause

Current truth assignment $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2\}$

Current decision assignment $\{x_1 = 1@6\}$

$w_1 = \neg x_1 \vee x_2$
$w_2 = \neg x_1 \vee x_3 \vee x_9$
$w_3 = \neg x_2 \vee \neg x_3 \vee x_4$
$w_4 = \neg x_4 \vee x_5 \vee x_{10}$
$w_5 = \neg x_4 \vee x_6 \vee x_{11}$
$w_6 = \neg x_5 \vee \neg x_6$
$w_7 = x_1 \vee x_7 \vee \neg x_{12}$
$w_8 = x_1 \vee x_8$
$w_9 = \neg x_7 \vee \neg x_8 \vee \neg x_{13}$

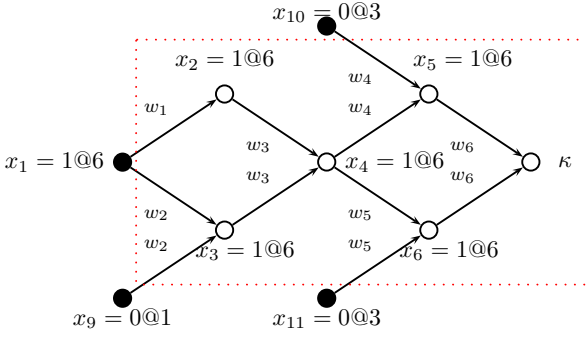


Implication Graphs and Learning

Current truth assignment $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2\}$

Current decision assignment $\{x_1 = 1@6\}$

$w_1 = \neg x_1 \vee x_2$
$w_2 = \neg x_1 \vee x_3 \vee x_9$
$w_3 = \neg x_2 \vee \neg x_3 \vee x_4$
$w_4 = \neg x_4 \vee x_5 \vee x_{10}$
$w_5 = \neg x_4 \vee x_6 \vee x_{11}$
$w_6 = \neg x_5 \vee \neg x_6$
$w_7 = x_1 \vee x_7 \vee \neg x_{12}$
$w_8 = x_1 \vee x_8$
$w_9 = \neg x_7 \vee \neg x_8 \vee \neg x_{13}$



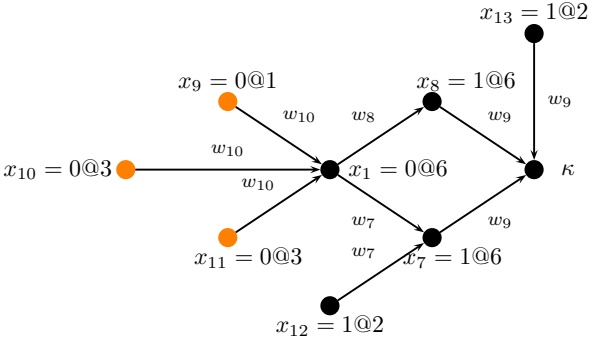
We learn the conflict clause $w_{10} : (\neg x_1 \vee x_9 \vee x_{11} \vee x_{10})$

Flipped Assignment

Current truth assignment $\{x_9 = 0@1, x_{10} = 0@3, x_{11} = 0@3, x_{12} = 1@2, x_{13} = 1@2\}$

Current flipped assignment $\{x_1 = 0@6\}$

$w_1 = \neg x_1 \vee x_2$
$w_2 = \neg x_1 \vee x_3 \vee x_9$
$w_3 = \neg x_2 \vee \neg x_3 \vee x_4$
$w_4 = \neg x_4 \vee x_5 \vee x_{10}$
$w_5 = \neg x_4 \vee x_6 \vee x_{11}$
$w_6 = \neg x_5 \vee \neg x_6$
$w_7 = x_1 \vee x_7 \vee \neg x_{12}$
$w_8 = x_1 \vee x_8$
$w_9 = \neg x_7 \vee \neg x_8 \vee \neg x_{13}$
$w_{10} = \neg x_1 \vee x_9 \vee x_{11} \vee x_{10}$



Another conflict clause: $w_{11} : (\neg x_{13} \vee \neg x_{12} \vee x_{11} \vee x_{10} \vee x_9)$

Where should we backtrack to now?

Which assignments caused the conflicts?

- $x_9 = 0@1$
- $x_{10} = 0@3$
- $x_{11} = 0@3$
- $x_{12} = 1@2$
- $x_{13} = 1@2$

These assignments are sufficient for causing a conflict.

Backtrack to $DL = 3$

So the rule is: backtrack to the largest decision level in the conflict clause.

This works for both the initial conflict and the conflict after the flip.

Q: What if the flipped assignment works?

A: Change the decision retroactively.

$$x_1 = 0$$
$$x_2 = 0$$
$$x_3 = 1$$
$$x_4 = 0$$
$$x_5 = 0$$

$$x_1 = 0$$
$$x_2 = 0$$
$$x_3 = 1$$
$$x_4 = 0$$
$$x_5 = 0$$

Non-Chronological Backtracking

$$x_1 = 0$$
$$x_2 = 0$$
$$x_3 = 1$$
$$x_4 = 0$$
$$x_5 = 0$$

$$x_5 = 1$$
$$x_7 = 0$$
$$x_9 = 1$$

$x_1 = 0$
$x_2 = 0$
$x_3 = 1$
$x_4 = 0$
$x_5 = 0$

$x_5 = 1$
$x_7 = 0$
$x_9 = 1$

$x_9 = 0$

Non-Chronological Backtracking

$x_1 = 0$
$x_2 = 0$
$x_3 = 1$
$x_4 = 0$
$x_5 = 0$

$x_5 = 1$
$x_7 = 0$
$x_9 = 1$

$x_3 = 0$
$x_9 = 0$

$x_1 = 0$
$x_2 = 0$

$x_3 = 0$
$x_6 = 0$
$\ldots$
$\ldots$

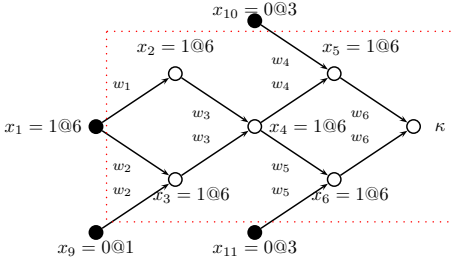
More Conflict Clauses

Definition

A Conflict Clause is any clause implied by the formula.

Let L be a set of literals labeling nodes that form a cut in the implication graph, separating the conflict node from the roots.

Claim: $\bigvee_{l \in L} \neg l$ is a Conflict Clause.



$x_{10} \vee \neg x_1 \vee x_9 \vee x_{11}$

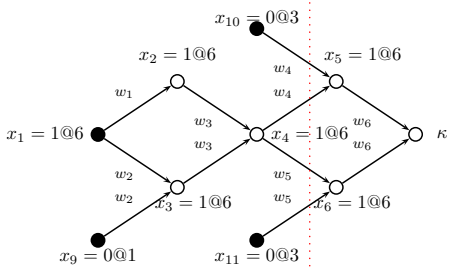
More Conflict Clauses

Definition

A Conflict Clause is any clause implied by the formula.

Let L be a set of literals labeling nodes that form a cut in the implication graph, separating the conflict node from the roots.

Claim: $\bigvee_{l \in L} \neg l$ is a Conflict Clause.



$x_{10} \vee \neg x_1 \vee x_9 \vee x_{11}$

$x_{10} \vee \neg x_4 \vee x_{11}$

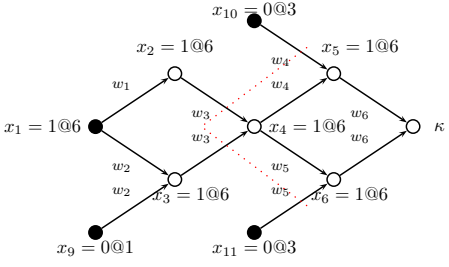
More Conflict Clauses

Definition

A Conflict Clause is any clause implied by the formula.

Let L be a set of literals labeling nodes that form a cut in the implication graph, separating the conflict node from the roots.

Claim: $\bigvee_{l \in L} \neg l$ is a Conflict Clause.



$x_{10} \vee \neg x_1 \vee x_9 \vee x_{11}$

$x_{10} \vee \neg x_4 \vee x_{11}$

$x_{10} \vee \neg x_2 \vee \neg x_3 \vee x_{11}$

Conflict Clause

How many clauses should we add?

If not all, then which ones?

- Shorter ones?
- Check their influence on the backtracking level?
- The most “influential”?

Conflict Clause

Definition

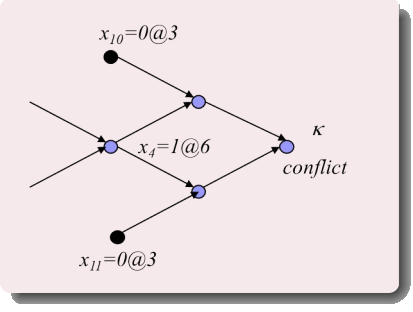
An Asserting Clause is a Conflict Clause with a single literal from the current decision level.
Backtracking (to the right level) makes it a Unit clause.

Asserting clauses are those that force an immediate change in the search path.

Modern solvers only consider Asserting Clauses.

Alternative Backtracking

Conflict-Driven Backtracking



Conflict clause: $(x_{10} \vee \neg x_4 \vee \neg x_{11})$

With standard Non-Chronological Backtracking we backtracked to $DL = 6$.

Conflict-driven Backtrack: backtrack to the second highest decision level in the clause (without erasing it).

In this case, to $DL = 3$.

Q: why?

$$x_1 = 0$$
$$x_2 = 0$$
$$x_3 = 1$$
$$x_4 = 0$$
$$x_5 = 0$$

$$x_1 = 0$$
$$x_2 = 0$$
$$x_3 = 1$$
$$x_4 = 0$$
$$x_5 = 0$$

Conflict-Driven Backtracking

$$x_1 = 0$$
$$x_2 = 0$$
$$x_3 = 1$$
$$x_4 = 0$$
$$x_5 = 0$$

$$x_1 = 0$$
$$x_2 = 0$$
$$x_5 = 1$$
$$x_7 = 0$$
$$x_9 = 1$$

$x_1 = 0$
$x_2 = 0$

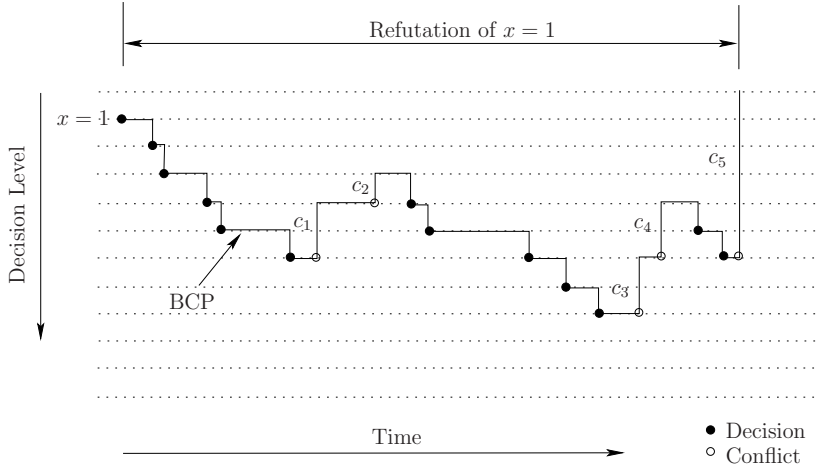
$x_5 = 1$
$x_7 = 0$
$x_9 = 1$

$x_1 = 0$
$x_2 = 0$

$x_5 = 1$
$x_7 = 0$
$x_9 = 1$

$x_1 = 0$
$x_2 = 0$
$x_5 = 1$
$x_9 = 0$
$x_6 = 0$

Conflict-Driven Backtracking



So the rule is: backtrack to the second highest decision level dl, but do not erase it.

This way the literal with the currently highest decision level will be implied in $DL = dl$.

Q: what if the conflict clause has a single literal?

For example, from $(x \vee \neg y) \wedge (x \vee y)$ and decision $x = 0$, we learn the conflict clause (x).

Resolution

The binary resolution is a sound inference rule:

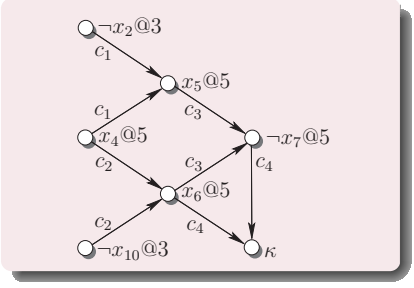
$$\frac{(a_1 \vee \ldots \vee a_n \vee \beta) \quad (b_1 \vee \ldots \vee b_m \vee \neg\beta)}{(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)}$$ Binary Resolution

Example

$$\frac{x_1 \vee x_2 \quad \neg x_1 \vee x_3 \vee x_4}{x_2 \vee x_3 \vee x_4}$$

Example

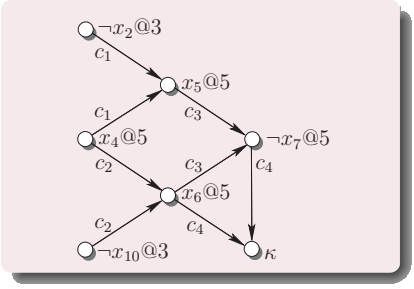
$$c_1 = (\neg x_4 \vee x_2 \vee x_5)$$
$$c_2 = (\neg x_4 \vee x_{10} \vee x_6)$$
$$c_3 = (\neg x_5 \vee \neg x_6 \vee \neg x_7)$$
$$c_4 = (\neg x_6 \vee x_7)$$



Conflict Clause : $c_5 = (\neg x_4 \vee x_2 \vee x_{10})$

Example

$c_1 = (\neg x_4 \vee x_2 \vee x_5)$
$c_2 = (\neg x_4 \vee x_{10} \vee x_6)$
$c_3 = (\neg x_5 \vee \neg x_6 \vee \neg x_7)$
$c_4 = (\neg x_6 \vee x_7)$



Assume that the implication order in the BCP was x_4, x_5, x_6, x_7.

name	cl	lit	var	$ante$
c_4	$(\neg x_6 \vee x_7)$	x_7	x_7	c_3
	$(\neg x_5 \vee \neg x_6)$	$\neg x_6$	x_6	c_2
	$(\neg x_4 \vee x_{10} \vee \neg x_5)$	$\neg x_5$	x_5	c_1
c_5	$(\neg x_4 \vee x_2 \vee x_{10})$			

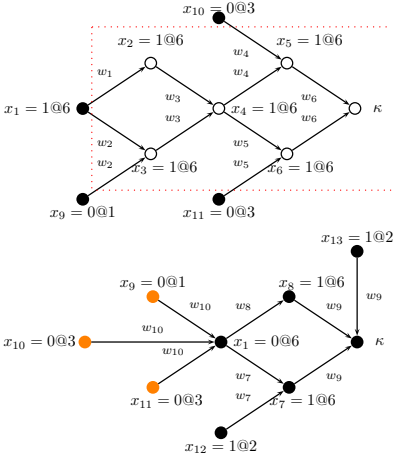
The Algorithm

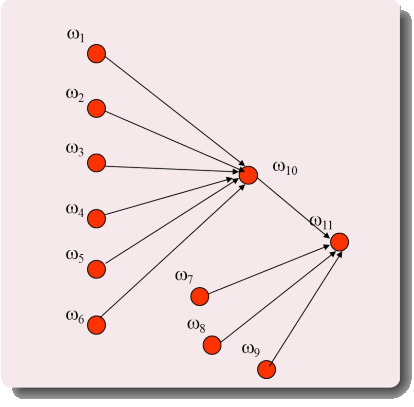
```
ANALYZE-CONFLICT()
if current_desicion_level = 0 then return False;
;
while ¬ STOP-CRITERION-MET (cl) do
    lit := LAST-ASSIGNED-LITERAL (cl);
    var := VARIABLE-OF-LITERAL (lit);
    ante := Antecedent(lit);
    cl := RESOLVE (cl, ante, var);
end
ADD-CLAUSE-TO-DATABASE (cl);
```

name	cl	lit	var	$ante$
c_4	$(\neg x_6 \vee x_7)$	x_7	x_7	c_3
	$(\neg x_5 \vee \neg x_6)$	$\neg x_6$	x_6	c_2
	$(\neg x_4 \vee x_{10} \vee \neg x_5)$	$\neg x_5$	x_5	c_1
c_5	$(\neg x_4 \vee x_2 \vee x_{10})$			

Resolution Graph

The resolution graph keeps track of the inference relation.

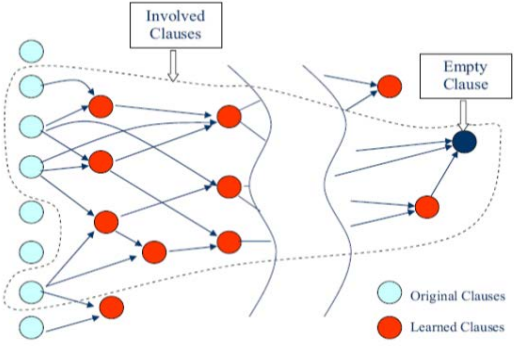




Resolution Graph

What is it good for?

Example: for computing an unsatisfiable core



from SAT'03

VSIDS (Variable State Independent Decaying Sum)

Each literal has a counter initialized to 0.

When a clause is added, the counters are updated.

The unassigned variable with the highest counter is chosen.

Periodically, all the counters are divided by a constant.

firstly implemented in **Chaff**

Chaff holds a list of unassigned variables sorted by the counter value.

Updates are needed only when adding conflict clauses.

Thus, decision is made in constant time.

Decision Heuristics - VSIDS

VSIDS is a quasi-static strategy:

- static because it does not depend on current assignment
- dynamic because it gradually changes. Variables that appear in recent conflicts have higher priority.

This strategy is a conflict-driven decision strategy, which dramatically improves performance.

Decision Heuristics - Berkmin

Keep conflict clauses in a stack

Choose the first unresolved clause in the stack (If there is no such clause, use VSIDS)

Choose from this clause a variable + value according to some scoring (e.g. VSIDS)

This gives absolute priority to conflicts.

SAT Solver

SAT solver is to be said as the ”most successful formal tools”.

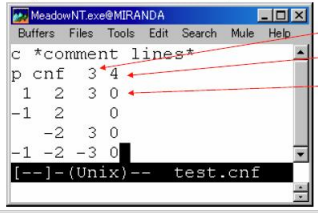
There are a SAT Competitions every one or two years.

- http://www.satcompetition.org/

Zchaff(The champion of 2004) can handle 100,000 variables with millions of clauses (Experiments: 800 variables with 9,000 clauses in 0.0sec).

Using zChaff

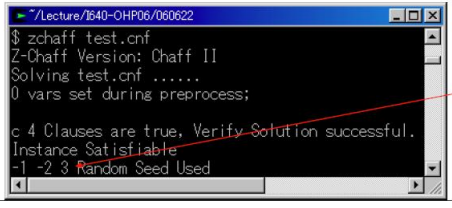
- Input file format (CNF in the *suggested form*)



Number of variables

Number of clauses

Clause means:
$x_1 \vee x_2 \vee x_3$
$\neg x_1 \vee x_2$
$\neg x_2 \vee x_3$
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
where 0 is terminator.



Results:
$x_1 = 0$
$x_2 = 0$
$x_3 = 1$.

More on SAT Society

SMT solver, string solver.

Coping with NP-completeness

Q. Suppose I need to solve an **NP**-complete problem. What should I do?

A. Theory says you're unlikely to find polynomial time algorithm.

Must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

Think About: What features have been sacrificed in today's topic?

Gradient Descent

Gradient Descent: Vertex Cover

VERTEX COVER. Given a graph $G = (V, E)$, find a subset of nodes S of minimal cardinality such that for each $(u, v) \in E$, either u or v are in S.

Neighbor relation. $S \sim S'$ if S' can be obtained from S by adding or deleting a single node.

Note. Each vertex cover S has at most n neighbors.

Gradient descent. Start with $S = V$. If there is a neighbor S' that is a vertex cover and has lower cardinality, replace S with S'.

Remark

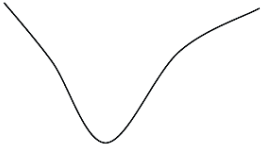
Algorithm terminates after at most n steps since each update decreases the size of the cover by one.

Local Search

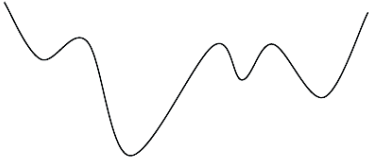
Local search. Algorithm that explores the space of possible solutions in sequential fashion, moving from a current solution to a "nearby" one.

Neighbor relation. Let $S \sim S'$ be a neighbor relation for the problem.

Gradient descent. Let S denote current solution. If there is a neighbor S' of S with strictly lower cost, replace S with the neighbor whose cost is as small as possible. Otherwise, terminate the algorithm.



A funnel



a jagged funnel

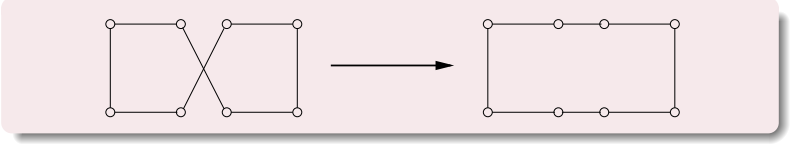
```
let s be any initial solution
while there is some solution s' in the neighborhood of s
   for which cost(s') < cost(s):  replace s by s'
return s
```

Further Example: TSP

Assume we have all interpoint distances between n cities, giving a search space of $(n-1)!$ different tours. What is a good notion of neighborhood?

Two tours are close if they differ in just a few edges. They can't differ in just one edge, so we will consider differences of two edges.

We define the 2-change neighborhood of tour s as being the set of tours that can be obtained by removing two edges of s and then putting in two other edges.



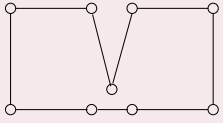
Evaluation of 2-Change Neighborhood

Q: What is its overall running time, and does it always return the best solution?

Neither of these questions has a satisfactory answer:

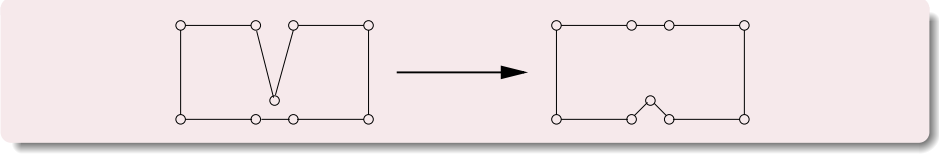
Each iteration is certainly fast, because a tour has only $O(n^2)$ neighbors. There might be an exponential number of them.

We can easily say about the final tour is that it is locally optimal. There might be better solutions further away.



3-Change Neighborhood

We may try a more generous neighborhood, for instance 3-change, consisting of tours that differ on up to three edges.



The size of a neighborhood becomes $O(n^3)$, making each iteration more expensive. Moreover, there may still be suboptimal local minima, although fewer than before.

To avoid these, we would have to go up to 4-change, or higher.

Efficiency demands neighborhoods that can be searched quickly, but smaller neighborhoods can increase the abundance of low-quality local optima.

The appropriate compromise is typically determined by experimentation.

Metropolis Algorithm

Metropolis Algorithm

Metropolis algorithm.

- Simulate behavior of a physical system according to principles of statistical mechanics.
- Globally biased toward "downhill" steps, but occasionally makes "uphill" steps to break out of local minima.

Gibbs-Boltzmann Function

The probability of finding a physical system in a state with energy E is proportional to $e^{-E/(kT)}$, where $T > 0$ is temperature and k is a constant.

- For any temperature $T > 0$, function is monotone decreasing function of energy E.
- System more likely to be in a lower energy state than higher one.
  - T large: high and low energy states have roughly same probability
  - T small: low energy states are much more probable

Metropolis Algorithm

Metropolis algorithm.

- Given a fixed temperature T, maintain current state S.
- Randomly perturb current state S to new state $S' \in N(S)$.
- If $E(S') \leq E(S)$, update current state to S'.
- Otherwise, update current state to S' with probability $e^{-\Delta E/(kT)}$, where $\Delta E = E(S') - E(S) > 0$.

Theorem

Let $f_S(t)$ be fraction of first t steps in which simulation is in state S. Then, assuming some technical conditions, with probability 1:

$$\lim_{t \to \infty} f_S(t) = \frac{1}{Z} e^{-E(S)/(kT)} \text{ where } Z = \sum_{S \in N(S)} e^{-E(S)/(kT)}$$

Intuition. Simulation spends roughly the right amount of time in each state, according to Gibbs-Boltzmann equation.

Metropolis Algorithm in Minimum Problems

Start with an initial solution S_0, and constants k and T
In one step:
  Let S be the current solution
  Let S' be chosen uniformly at random from the neighbors of S
  If $c(S') \leq c(S)$ then
    Update $S \leftarrow S'$
  Else
    With probability $e^{-(c(S')-c(S))/(kT)}$
      Update $S \leftarrow S'$
    Otherwise
      Leave S unchanged
  EndIf

Weakness of Metropolis Algorithm

Consider a graph G with no edges, gradient descent solves this instance with no trouble, deleting nodes in sequence until none are left.

While the Metropolis algorithm will start out this way, it begins to go astray as it nears the global optimum.

Consider the situation in which the current solution contains only c nodes, where c is much smaller than the total number of nodes, n.

With very high probability, the neighboring solution generated by the Metropolis Algorithm will have size $c + 1$, rather than $c - 1$, Thus it gets harder and harder to shrink the size of the vertex cover as the algorithm proceeds.

It is exhibiting a sort of flinching reaction near the bottom of the funnel.

Simulated annealing.

- T large $\Rightarrow$ probability of accepting an uphill move is large.
- T small $\Rightarrow$ uphill moves are almost never accepted.
- Idea: turn knob to control T.
- Cooling schedule: $T = T(i)$ at iteration i.

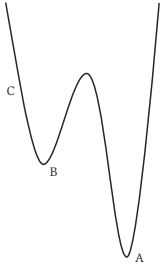
Physical analog.

- Take solid and raise it to high temperature, we do not expect it to maintain a nice crystal structure.
- Take a molten solid and freeze it very abruptly, we do not expect to get a perfect crystal either.
- Annealing: cool material gradually from high temperature, allowing it to reach equilibrium at succession of intermediate lower temperatures.

Simulated Annealing

Simulated annealing works by running the Metropolis algorithm while gradually decreasing the value of T over the course of the execution.

The exact way in which T is updated is called a cooling schedule,which is a function τ from $\{1, 2, 3, \ldots\}$ to the positive real numbers; in iteration i, we use the temperature $T = \tau(i)$.



Physical systems reach a minimum energy state via annealing, the simulated annealing has no guarantee to find an optimal solution.

If the two funnels take equal area, then at high temperatures the system is essentially equally likely to be in either funnel.

Once cooling the temperature, it will become harder and harder to switch between the two funnels.

Hopfield Neural Networks

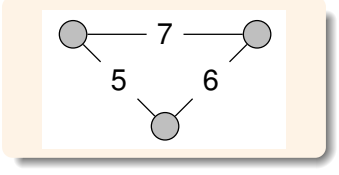
Hopfield networks. Simple model of an associative memory, in which a large collection of units are connected by an underlying network, and neighboring units try to correlate their states.

Input: Graph $G = (V, E)$ with integer (positive or negative) edge weights w.

Configuration. Node assignment $s_u = \pm 1$.

Intuition. If $w_{uv} < 0$, then u and v want to have the same state; if $w_{uv} > 0$ then u and v want different states.

Note. In general, no configuration respects all constraints.



Definition (Good edge)

With respect to a configuration S, edge $e = (u, v)$ is good if $w_e \times s_u \times s_v < 0$. That is, if $w_e < 0$ then $s_u = s_v$; if $w_e > 0$, then $s_u \neq s_v$.

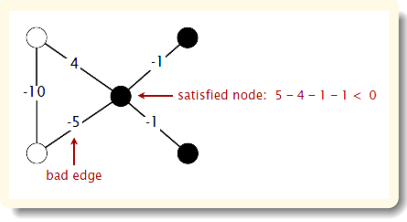
Definition (Satisfied node)

With respect to a configuration S, a node u is satisfied if the weight of incident good edges $\geq$ weight of incident bad edges.

$$\sum_{v:e=(u,v)\in E} w_e s_u s_v \leq 0$$

Definition (Stable configuration)

A configuration is stable if all nodes are satisfied.



Goal. Find a stable configuration, if such a configuration exists.

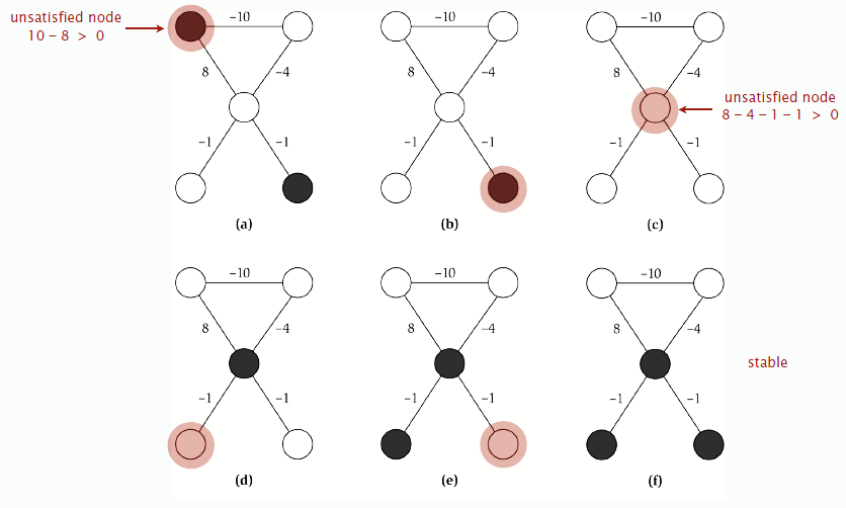
State-Flipping Algorithm

State-flipping algorithm. Repeated flip state of an unsatisfied node.

HOPFIELD-FLIP(G, w)

$S \leftarrow$ arbitrary configuration;
while *current configuration is not stable* **do**
  $u \leftarrow$ unsatisfied node;
  $s_u \leftarrow -s_u$;
end
return S;

State-Flipping Algorithm Example



State-Flipping Algorithm: Proof of Correctness

Theorem

The state-flipping algorithm terminates with a stable configuration after at most $W = \sum_e |w_e|$ *iterations.*

Proof [Hint.] Consider measure of progress $\Phi(S) = \#$ satisfied nodes.

Theorem

The state-flipping algorithm terminates with a stable configuration after at most $W = \sum_e |w_e|$ *iterations.*

Proof. Consider measure of progress $\Phi(\mathrm{S}) = \sum_{\mathrm{e\ good}} |\mathrm{w_e}|$.

- Clearly $0 \leq \Phi(\mathrm{S}) \leq \mathrm{W}$.
- We show $\Phi(\mathrm{S})$ increase by at least 1 after each flip.
  When u flips state:
  - all good edges incident to u become bad
  - all bad edges incident to u become good
  - all other edges remain the same

$$\Phi\left(\mathrm{S}'\right) = \Phi(\mathrm{S}) - \sum_{\substack{e\,:\,e=(u,v)\in E \\ e \text{ is bad}}} |\mathrm{w_e}| + \sum_{\substack{e\,:\,e=(u,v)\in E \\ e \text{ is good}}} |\mathrm{w_e}| \geq \Phi(\mathrm{S}) + 1$$

Maximum Cut

Maximum Cut

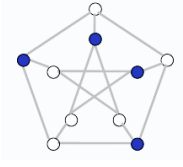
Maximum cut. Given an undirected graph $G = (V, E)$ with positive integer edge weights w_e, find a $cut(A, B)$ such that the total weight of edges crossing the cut is maximized.

$$w(A, B) := \sum_{u \in A, v \in B} w_{uv}$$

Toy application.

- n activities, m people.
- Each person wants to participate in two of the activities.
- Schedule each activity in the morning or afternoon to maximize number of people that can enjoy both activities.

Real applications. Circuit layout, statistical physics.



Single-flip neighborhood. Given a cut (A, B), move one node from A to B, or one from B to A if it improves the solution.

Greedy algorithm.

```
MAX-CUT-LOCAL(G, w)
while there exists an improving node v do
    if v ∉ A then
        A ← A ∪ {v}; B ← B − {v};
    end
    else
        B ← B ∪ {v}; A ← A − {v};
    end
end
return(A,B)
```

Maximum Cut: Big Improvement Flips

Local search. Within a factor of 2 for MAX-CUT, but not polynomial time!

Big-improvement-flip algorithm. Only choose a node which, when flipped, increases the cut value by at least $\frac{2\varepsilon}{n} w(A, B)$

Claim

Upon termination, big-improvement-flip algorithm returns a cut (A, B) such that $(2 + \varepsilon)w(A, B) \geq w(A^*, B^*)$

Proof idea. Add $\frac{2\varepsilon}{n} w(A, B)$ to each inequality in original proof.

Maximum Cut: Big Improvement Flips

Claim

Big-improvement-flip algorithm terminates after $O\left(\varepsilon^{-1} n \log W\right)$ *flips, where* $W = \sum_e w_e$.

Proof sketch.

Each flip improves cut value by at least a factor of $(1 + \varepsilon/n)$.

After n/ε iterations the cut value improves by a factor of 2.

- $(1 + 1/x)^x \geq 2$ for $x \geq 1$.

Cut value can be doubled at most $\log_2 W$ times.

Maximum Cut: Context

Theorem (Sahni-Gonzales 1976)

There exists a $1/2$-approximation algorithm for MAX-CUT.

Theorem

There exists an 0.878-approximation algorithm for MAX-CUT.

Theorem

*Unless **P = NP**, no 0.942-approximation algorithm for MAX-CUT.*

Neighbor Relations for Max Cut

1-flip neighborhood. Cuts (A, B) and (A', B') differ in exactly one node.

k-flip neighborhood. Cuts (A, B) and (A', B') differ in at most k nodes.

KL-neighborhood.[Kernighan-Lin 1970]

- To form neighborhood of (A, B):
  - Iteration 1: flip node from (A, B) that results in best cut value (A_1, B_1), and mark that node
  - Iteration i: flip node from (A_{i-1}, B_{i-1}) that results in best cut value (A_i, B_i) among all nodes not yet marked.
- Neighborhood of $(A, B) = (A_1, B_1), \ldots, (A_{n-1}, B_{n-1})$.
- Neighborhood includes some very long sequences of flips, but without the computational overhead of a k-flip neighborhood.
- Practice: powerful and useful framework.
- Theory: explain and understand its success in practice.

Nash Equilibria

Best response dynamics. Each agent is continually prepared to improve its solution in response to changes made by other agents.

Nash equilibrium. Solution where no agent has an incentive to switch.

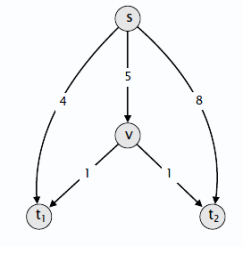
Fundamental question. When do Nash equilibria exist?

Two agents start with outer paths.

Agent 1 has no incentive to switch paths, since $4 < 5 + 1$. But agent 2 does since $8 > 5 + 1$.

Once this happen, agent 1 prefers middle path (since $4 > 5/2 + 1$)

Both agents using middle path is a Nash equilibrium.



Local search algorithm. Each agent is continually prepared to improve its solution in response to changes made by other agents.

Analogies.

- Nash equilibrium : local search.
- Best response dynamics : local search algorithm.
- Unilateral move by single agent : local neighborhood.

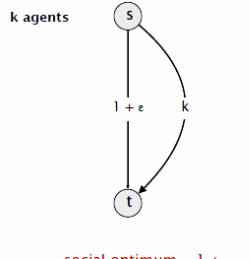
Contrast. Best-response dynamics need not terminate since no single objective function is being optimized.

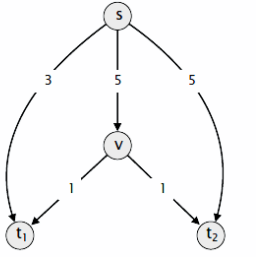
Socially Optimum

Social optimum. Minimizes total cost to all agent.

Observation. In general, there can be many Nash equilibria.

Even when its unique, it does not necessarily equal the social optimum.



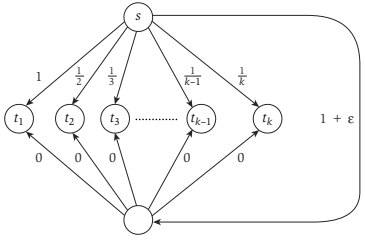


Price of Stability

Price of stability. Ratio of best Nash equilibrium to social optimum.

Fundamental question. What is price of stability?

Example. Price of stability = $\Theta(\log k)$.



Social optimum. Everyone takes bottom paths.

Unique Nash equilibrium. Everyone takes top paths.

Price of stability. $H(k)/(1+\varepsilon)$

Theorem

The following algorithm terminates with a Nash equilibrium.

BEST-RESPONSE-DYNAMICS (G, c, k)

for $j = 1$ *to* k **do**
  $P_j \leftarrow$ any path for agent j;
end
while *not a Nash equilibrium* **do**
  $j \leftarrow$ some agent who can improve by switching paths;
  $P_j \leftarrow$ better path for agent i;
end
return $P_1, P_2, \ldots P_k$

Proof. Consider a set of $P_1, \ldots, P_k$

- Let x_e denote the number of paths that use edge e.
- Let $\Phi(P_1, P_2, \ldots P_k) = \sum_{e \in E} c_e \cdot H(x_e)$ be a potential function, where

$$H(0) = 0$$
$$H(k) = \sum_{i=1}^{k} \frac{1}{i}$$

- Since there are only finitely many sets of paths, it suffices to show that Φ strictly decreases in each step.

Proof. [continued]

- Consider agent j switching from path P_j to path P'_j.
- Agent j switches because

$$\underbrace{\sum_{f \in P'_j - P_j} \frac{c_f}{x_f + 1}}_{\text{newly incurred cost}} < \underbrace{\sum_{e \in P_j - P'_j} \frac{c_e}{x_e}}_{\text{cost saved}}$$

- Φ increase by $\sum_{f \in P'_j - P_j} c_f \left[H\left(x_f + 1\right) - H\left(x_f\right)\right] = \sum_{f \in P'_j - P_j} \frac{c_f}{x_f + 1}$.
- Φ decrease by $\sum_{e \in P_j - P'_j} c_e \left[H\left(x_e\right) - H\left(x_e - 1\right)\right] = \sum_{e \in P_j - P'_j} \frac{c_c}{x_e}$
- Thus, net change in Φ is negative.

Lemma

Let $C(P_1, \ldots, P_k)$ denote the total cost of selecting paths $P_1, \ldots, P_k$. For any set of paths $P_1, \ldots, P_k$, we have

$$C\left(P_1, \ldots, P_k\right) \leq \Phi\left(P_1, \ldots, P_k\right) \leq H(k) \cdot C\left(P_1, \ldots, P_k\right)$$

Proof.

Let x_e denote the number of paths containing edge e.

- Let E^+ denote set of edges that belong to at least one of the paths. Then,

$$C\left(P_1, \ldots, P_k\right) = \sum_{e \in E^+} c_e \leq \underbrace{\sum_{e \in E^+} c_e H\left(x_e\right)}_{\Phi(P_1, \ldots, P_k)} \leq \sum_{e \in E^+} c_e H(k) = H(k) C\left(P_1, \ldots, P_k\right)$$

Theorem

There is a Nash equilibrium for which the total cost to all agents exceeds that of the social optimum by at most a factor of $H(k)$.

Proof.

- Let $(P_1^*, \ldots, P_k^*)$ denote a set of socially optimal paths.
- Run best-response dynamics algorithm starting from P^*.
- Since Φ is monotone decreasing $\Phi(P_1, \ldots, P_k) \leq \Phi(P_1^*, \ldots, P_k^*)$,

$$C(P_1, \ldots, P_k) \leq \Phi(P_1, \ldots, P_k) \leq \Phi(P_1^*, \ldots, P_k^*) \leq H(k) \cdot C(P_1^*, \ldots, P_k^*)$$

Summary

Existence. Nash equilibria always exist for k-agent multicast routing with fair sharing.

Price of stability. Best Nash equilibrium is never more than a factor of $H(k)$ worse than the social optimum.

Fundamental open problem. Find any Nash equilibria in polynomial time.

Algorithm Design XXI

Quantum Algorithms

Guoqiang Li
School of Software

Shanghai Jiao Tong University

Qubits, Superpositions, and Measurement

A Quote from Richard Feynman

I think I can safely say that no one understands quantum physics.

Chips

In ordinary computer chips, bits are physically represented by low and high voltages on wires.

But there are many other ways a bit could be stored, for instance, in the state of a hydrogen atom. The single electron in this atom can

- either be in the ground state (the lowest energy configuration),
- or it can be in an excited state(a high energy configuration).

We can use these two states to encode for bit values 0 and 1, respectively.

Notation

Ground state: $|0\rangle$

Excited state: $|1\rangle$

Superpositions

If a quantum system can be in one of two states, then it can also be in any linear superposition of those two states.

For instance,

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \text{ or } \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

or an infinite number of other combination of the form

$$\alpha_0|0\rangle + \alpha_1|1\rangle$$

The α's can be even complex numbers, provided

$$|\alpha_0|^2 + |\alpha_1|^2 = 1$$

i.e., they are normalized.

Superpositions

The whole concept of a superposition suggests that the electron does not make up its mind about whether it is in the ground or excited state, and the amplitude α_0 is a measure of its inclination toward the ground state.

Continuing along this line of thought, it is tempting to think of α_0 as the probability that the electron is in the ground state.

But then how are we to make sense of the fact that α_0 can be negative, or even worse, imaginary?

WE DON'T UNDERSTAND THIS, BUT GET USED TO IT.

This linear superposition is the private world of the electron.

For us to get a glimpse of the electron's state we must make a measurement to get a single bit of information - 0 or 1.

If the state of the electron is $\alpha_0 |0\rangle + \alpha_1 |1\rangle$, then the outcome of the measurement is 0 with probability $|\alpha_0|^2$ and 1 with probability $|\alpha_1|^2$.

Moreover, the act of measurement causes the system to change its state:

if the outcome of the measurement is 0, then the new state of the system is $|0\rangle$ (the ground state), and if the outcome is 1, the new state is $|1\rangle$ (the excited state).

k-level systems

The superposition principle holds not just for 2-level systems, but in general for k-level systems.

In reality the electron in the hydrogen atom can be in one of many energy levels, starting with the ground state, the first excited state, the second excited state, and so on.

A k-level systems consists of the ground state and the first $\mathbf{k}-1$ excited states denoted by

$$|0\rangle, |1\rangle, |2\rangle, \ldots, |\mathbf{k}-1\rangle$$

The general quantum state of the system is

$$\alpha_0 |0\rangle + \alpha_1 |1\rangle + \alpha_{k-1} |k-1\rangle$$

where $\sum_{j=0}^{k-1} |\alpha_j|^2 = 1$

Measuring the state of the system would now reveal a number between 0 and $k-1$, and outcome j would occur with probability $|\alpha_j|^2$.

The measurement would disturb the system, and the new state would actually become $|j\rangle$ or the jth excited state.

Encoding n bits

We could choose $\mathtt{k} = 2^n$ levels of the hydrogen atoms.Or it is more promising to use n qubits.

Considering two qubits, that is, the state of the electrons of two hydrogen atoms.

Since each electron can be in either the ground or excited state, in classical physics the two electrons have a total of four possible states 00, 01, 10, or 11, and are therefore suitable for storing 2 bits of information.

But in quantum physics, the superposition principle tells us that the quantum state of the two electrons is a linear combination of the four classical states,

$$|\alpha\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle$$

where $\sum_{x\in\{0,1\}^2} |\alpha_x|^2 = 1$.

Measuring the state of the system now reveals 2 bits of information, and the probability of outcome $x \in \{0,1\}^2$ is $|\alpha_x|^2$.

If the outcome of measurement is jk, then the new state of the system is $|jk\rangle$.

What if we make a partial measurement?

If we measure just the first qubit, what is the probability that the outcome is 0?

$$\text{Prob}\{1stbit = 0\} = \text{Prob}\{00\} + \text{Prob}\{01\} = |\alpha_{00}|^2 + |\alpha_{01}|^2$$

How much does this partial measurement disturb the state of the system?

If the outcome of measuring the first qubit is 0, then the new superposition is obtained by crossing out all terms of $|\alpha\rangle$ that are inconsistent with this outcome (that is, whose first bit is 1).

The new state would be

$$|\alpha_{new}\rangle = \frac{\alpha_{00}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} |00\rangle + \frac{\alpha_{01}}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} |01\rangle$$

n hydrogen atoms

Classically the states of the n electrons could be used to store n bits of information in the obvious way.

But the quantum state of the n qubits is a linear superposition of all 2^n possible classical states:

$$\sum_{x \in \{0,1\}^n} \alpha_x |x\rangle$$

For $n = 500$, the number 2^n s much larger than estimates of the number of elementary particles in the universe.

- Where does Nature store this information?
- How could microscopic quantum systems of a few hundred atoms contain more information than we can possibly store in the entire classical universe?

WE DON'T UNDERSTAND THIS, BUT GET USED TO IT.

In this phenomenon lies the basic motivation for quantum computation. Why not tap into this massive amount of effort being expended at the quantum level?

There is a fundamental problem: this exponentially large linear superposition is the private world of the electrons.

Measuring the system only reveals n bits of information. The probability that the outcome is a particular n-bit string x is $|\alpha_x|^2$. And the new state after measurement is just $|x\rangle$.

The Plan

The structure of a quantum algorithm

The structure of quantum algorithm reflects the tension between

- The exponential private workspace of an n-qubit system
- and the mere n bits that can be obtained through measurement.

The input to a quantum algorithm consists of n classical bits, and the output also consists of n classical bits.

It is while the quantum system is not being watched that the quantum effects take over and we have the benefit of Nature working exponentially hard on our behalf.

The structure of a quantum algorithm

If the input is an n-bit string x, then the quantum computer takes as input n qubits in state $|x\rangle$.

Then a series of quantum operations are performed, by the end of which the state of the n qubits has been transformed to some superposition

$$\sum_y \alpha_y |y\rangle$$

Finally, a measurement is made, and the output is the n-bit string y with probability $|\alpha_y|^2$.

Observe that this output is random. As long as y corresponds to the right answer with high enough probability, we can repeat the whole process a few times to make the chance of failure miniscule.

Quantum factoring algorithm

The algorithm to factor a large integer N can be viewed as a sequence of reductions:

- Factoring is reduced to finding a nontrivial square root of $1 \text{ modulo } N$.
- Finding such a root is reduced to computing the order of a random integer modulo N.
- The order of an integer is precisely the period of a particular periodic superposition.
- Finally, periods of superpositions can be found by the quantum FFT.

The Quantum Fourier Transform

Interpolation resolved

FFT takes as input an M-dimensional, complex-valued vector α (where $M = 2^m$), and outputs an M-dimensional complex-valued vector β:

$$\begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_{M-1} \end{bmatrix} = \frac{1}{\sqrt{M}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \dots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & x^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_{M-1} \end{bmatrix}$$

The new factor $\sqrt{M}$ is to ensure that if the $|\alpha_i|^2$ add up to 1, then so do the $|\beta_i|^2$.

The quantum fast Fourier transform

Input: A superposition of $m = \log M$ qubits, $|\alpha\rangle = \sum_{j=1}^{M-1} \alpha_j |r\rangle$

Method: Using $O(m^2) = O(\log^2 M)$ quantum operation perform the quantum FFT to obtain the superposition $|\beta\rangle = \sum_{j=1}^{M-1} \beta_j |r\rangle$.

Output: A random m-bit number j (i.e., $0 \leq j < M$) from the probability distribution $\mathrm{Prob}[j] = |\beta_2|^2$.

Periodicity

Periodicity

Suppose $|\alpha\rangle = (\alpha_0, \ldots, \alpha_{M-1})$ is such that

$$\alpha_i = \alpha_j \qquad \text{if } i \equiv j \mod k$$

Moreover, suppose that exactly one of the $\mathtt{k}$ numbers $\alpha_0, \ldots, \alpha_{\mathtt{k}-1}$ is nonzero, say α_j. Then we say that $|\alpha\rangle$ is periodic with period $\mathtt{k}$ and offset j.

Quantum FFT for periodicity

Theorem

Suppose the input to quantum Fourier sampling is periodic with period $\mathtt{k}$*, for some* $\mathtt{k}$ *that divides* M*. Then the output will be a multiple of* $M/\mathtt{k}$*, and it is equally likely to be any of the* $\mathtt{k}$ *multiples of* $M/\mathtt{k}$*.*

Computing $M/\mathtt{k}$

Lemma

Suppose s independent samples are drawn uniformly from

$$0, \frac{M}{\mathtt{k}}, \frac{2M}{\mathtt{k}}, \ldots, \frac{(k-1)M}{\mathtt{k}}$$

with probability at least $1 - k/2^s$, the greatest common divisor of these samples is $M/\mathtt{k}$.

Factoring as Periodicity

Nontrivial square root

Fix an integer N. A nontrivial square root of $1 \mod N$ is any integer $x \not\equiv \pm 1 \mod N$ such that $x^2 \equiv 1 \mod N$.

Lemma

If x is a nontrivial square root of $1 \mod N$, then $\gcd(x+1, N)$ is a nontrivial factor of N.

Order

The order of $x \mod N$ is the smallest positive integer r such that $x^r \equiv 1 \mod N$.

Lemma

Let N be an odd composite, with at least two distinct prime factors, and let x be chosen uniformly at random between 0 and $N-1$. If $\gcd(x, N) = 1$, then with probability at least $1/2$, the order r of $x \mod N$ is even, and moreover $x^{r/2}$ is a nontrivial square root of $1 \mod N$.

The Quantum Algorithm for Factoring

The algorithm

1. Choose x uniformly at random in the range $1 \leq x \leq N-1$.
2. Let M be a power of 2 near N.
3. Repeat $s = 2\log N$ times:
   - 3.1. Start with two quantum registers, both initially 0, the first large enough to store a number $\text{modulo } M$ and the second $\text{modulo } N$.
   - 3.2. Use the periodic function $f(a) \equiv x^a \mod N$ to create a periodic superposition $|\alpha\rangle$ of length M as follows:
     - 3.2.1. Apply the QFT to the first register to obtain the superposition $\sum_{a=0}^{M-1} \frac{1}{\sqrt{M}} |a, 0\rangle$
     - 3.2.2. Compute $f(a) = x^a \mod N$ using a quantum circuit, to get the $\sum_{a=0}^{M-1} \frac{1}{\sqrt{M}} |a, x^a \mod N\rangle$
     - 3.2.3. Measure the second register. Now the first register contains the periodic superposition $|\alpha\rangle = \sum_{j=0}^{M/r-1} \sqrt{\frac{r}{M}} |jr+k\rangle$ where k is a random offset between 0 and $r-1$ (recall that r is the order of $x \mod N$).
   - 3.3. Fourier sample the superposition $|\alpha\rangle$ to obtain an index between 0 and $M-1$.

   Let g be the $\gcd$ of the resulting indices $j_1, \ldots, j_s$.
4. If M/g is even, then compute $\gcd(N, x^{M/2g}+1)$ and output it if it is a nontrivial factor of N; otherwise return to 1.

"algorithms which employ a degree of randomness as part of its logic"

—Wikipedia

algorithms that flip coins

Basis

Randomized algorithms are a basis of

- Online algorithms
- Approximation algorithms
- Quantum algorithms
- Massive data algorithms

Pro and Con of Randomization

Cost of Randomness

- chance the answer is wrong.
- chance the algorithm takes a long time.

Benefits of Randomness

- on average, with high probability gives a faster or simpler algorithm.
- some problems require randomness.

Types of Randomized Algorithms

Las Vegas Algorithm (LV):

- Always correct
- Runtime is random (small time with good probability)
- Examples: Quicksort, Hashing

Monte Carlo Algorithm (MC):

- Always bounded in runtime
- Correctness is random
- Examples: Karger's min-cut algorithm

LV implies MC:

Fix a time T and let the algorithm run for T steps. If the algorithm terminates before T, we output the answer, otherwise we output 0.

MC does not always imply LV:

The implication holds when verifying a solution can be done much faster than finding one.

Test the output of MC algorithm and stop only when a correct solution is found.

Las Vegas Algorithm: Quicksort

Quick Sort

`QuickSort(`x`);`

if $x == []$ **then** `return []`;
Choose pivot $t \in [n]$;
`return`
`QuickSort`$([x_i | x_i < x_t]) + [x_t] +$ `QuickSort`$([x_i | x_i \geq x_t])$;

Why Random?

If $t = 1$ always, then the complexity is $\Theta(n^2)$.

Note that, we do not need an adversary for bad inputs.

In practice, lists that need to be sorted will consist of a mostly sorted list with a few new entries.

Z_{ij} := event that ith largest element is compared to the jth largest element at any time during the algorithm.

Each comparison can happen at most once. Time is proportional to the number of comparisons $= \sum_{i<j} Z_{ij}$

$Z_{ij} = 1$ iff the first pivot in $\{i, i+1, \ldots, j\}$ is i or j.

- if the pivot is $< i$ or $> j$, then i and j are not compared;
- if the pivot is $> i$ and $< j$, then the pivot splits i and j into two different recursive branches so they will never be compared.

Thus, we have

$$Pr[Z_{ij}] = \frac{2}{j-i+1}$$

.

$$
\begin{aligned}
E[Time] &\leq E\left[\sum_{i<j} Z_{ij}\right] \\
&= \sum_{i<j} E\left[Z_{ij}\right] \\
&= \sum_{i} \sum_{i<j} \left(\frac{2}{j-i+1}\right) \\
&= 2 \cdot \sum_{i} \left(\frac{1}{2} + \cdots + \frac{1}{n-i+1}\right) \\
&\leq 2 \cdot n \cdot (H_n - 1) \\
&\leq 2 \cdot n \cdot \log n
\end{aligned}
$$

Monte Carlo: Min Cuts

Min Cuts

The (s,t) cut is the set $S \subseteq V$ with $s \in S$, $t \notin S$.

The cost of a cut is the number of edges e with one vertex in S and the other vertex not in S.

The min (s,t) cut is the (s,t) cut of minimum cost.

The global min cut is the min (s,t) cut over all $s,t \in V$.

Complexity

Iterate over all choices of s and t and pick the smallest (s, t) cut, by simply running $O(n^2)$ the Ford-Fulkersons algorithms over all pairs.

The complexity can be reduced by a factor of n by noting that each node must be in one of the two partitioned subsets.

Select any node s and compute a $\min(s, t)$ cut for all other vertices and return the smallest cut. This results in $O(n)$ the Ford-Fulkersons algorithm, resulting in complexity $O(n \cdot nm) = O(n^2 m)$.

The First Randomized Algorithm


```
MinCut1(G);
while n > 1 do
    Choose a random edge;
    Contract into a single vertex;
end
```

Contracting an edge means that removing the edge and combining the two vertices into a super-node.

Analysis

Lemma

The chance the algorithm fails in step 1 is $\leq \frac{2}{n}$

Proof.

The chance of failure in the first step is $\frac{OPT}{m}$ where

$$OPT = \text{cost of true min cut} = \mid E(S, \bar{S}) \mid$$

and m is the number of edge.

For all $u \in V$, let $d(u)$ be the degree of u.

$$OPT = \mid E(S, \bar{S}) \mid \leq \min_u d(u) \leq \frac{1}{n} \sum_{u \in V} d(u) \leq \frac{2 \cdot m}{n}$$

Dividing both sides by m we have

$$\frac{\mid E(S, \bar{S}) \mid}{m} \leq \frac{2}{n}$$

Continue the analysis in each subsequent round, we have

$$Pr(\text{fail in } 1^{st} \text{ step}) \leq \frac{2}{n}$$

$$Pr(\text{fail in } 2^{nd} \text{ step} \mid \text{success in } 1^{st} \text{ step}) \leq \frac{2}{n-1}$$

$$\vdots$$

$$Pr(\text{fail in } i^{th} \text{ step} \mid \text{success till } (i-1)^{th} \text{ step}) \leq \frac{2}{n+1-i}$$

Analysis

Lemma

`MinCut1` succeeds with $\geq \frac{2}{n^2}$ probability.

Proof.

$Z_i :=$ the event that an edge from the cut set is picked in round i.

$$Pr[Z_i | \bar{Z}_1 \cap \bar{Z}_2 \cap \cdots \cap \bar{Z}_{i-1}] \leq \frac{2}{n+1-i}$$

Thus the probability of success is given by

$$\begin{aligned} Pr(Succ) &= Pr[\bar{Z}_1 \cap \bar{Z}_2 \cap \cdots \cap \bar{Z}_{n-2}] \geq (1-\frac{2}{n})(1-\frac{2}{n-1})\ldots(1-\frac{2}{3}) \\ &\geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdot \frac{n-5}{n-3} \cdot \ldots \cdot \frac{2}{4} \cdot \frac{1}{3} = \frac{1}{n} \cdot \frac{1}{n-1} \cdot \frac{2}{1} \cdot \frac{1}{1} \\ &= \frac{2}{n(n-1)} \geq \frac{2}{n^2} \end{aligned}$$

A Lemma

Suppose that the `MinCut1` is terminated when the number of vertices remaining in the contracted graph is exactly t. Then any specific min cut survives in the resulting contracted graph with probability at least

$$\frac{\binom{t}{2}}{\binom{n}{2}} = \Omega(\frac{t}{n})^2$$

The Second Randomized Algorithm

`MinCut2(`G, k`);`

$i = 0$;
while $i > k$ **do**
    `MinCut1(`G`);`
    $i++$;
end
pick the optimization;

If we run `MinCut1` k times and pick the set of min cost, then

$$Pr(failure) \leq (1 - \frac{2}{n^2})^k \leq e^{\frac{-2k}{n^2}}$$

A Useful Bound

$$(1-a) \leq e^{-a} \qquad \forall a > 0$$

How to Choose k

Set $k = \frac{n^2}{2} \log(\frac{1}{\delta})$ to get $1 - \delta$ success probability.

Observation

Initial stages of the algorithm are very likely to be correct. In particular, the first step is wrong with probability at most $2/n$.

As contracting more edges, failure probability goes up.

Since earlier ones are more accurate and slower, why not do less of them at the beginning, and more as the number of edges decreases?

The Third Randomized Algorithm

`MinCut3(`G`);`

Repeat twice{
  take $n - \frac{n}{\sqrt{2}}$ steps of contraction;
  recursively apply this algorithm;
}
take better result;

$$T(n) = 2T(\frac{n}{\sqrt{2}}) + O(n^2) = n^2 \log n$$

Success Probability

$$
\begin{aligned}
pr(n) &= 1 - (\text{failure probability of one branch})^2 \\
&= 1 - (1 - \text{success in one branch})^2 \\
&\geq 1 - (1 - (\frac{\frac{n}{\sqrt{2}}}{n})^2 \cdot pr(\frac{n}{\sqrt{2}}))^2 \\
&= 1 - \left(1 - \frac{1}{2}\, pr\left(\frac{n}{\sqrt{2}}\right)\right)^2 \\
&= pr\left(\frac{n}{\sqrt{2}}\right) - \frac{1}{4}\, pr\left(\frac{n}{\sqrt{2}}\right)^2
\end{aligned}
$$

Let $x = \log_{\sqrt{2}} n$, by setting $f(x) = pr(n)$, we get

$$f(x) = f(x-1) - f(x-1)^2$$

$f(x) = \frac{1}{x}$ gives:

$$f(x-1) - f(x) = \frac{1}{x-1} - \frac{1}{x} = \frac{1}{x(x-1)} \approx \frac{1}{(x-1)^2} = f(x-1)^2$$

Thus, $pr(n) = O\left(\frac{1}{\log n}\right)$.

The Fourth Randomized Algorithm

Repeat `MinCut3` $O(\log n \log \frac{1}{\delta})$ times.

Complexity & Success Analysis: DIY!

Linearity of Expectation

Linearity of Expectation

$$E[\sum_i X_i] = \sum_i E[X_i]$$

Sailors Problem

A ship arrives at a port, and the 40 sailors on board go ashore for revelry. Later at night, the 40 sailors return to the ship and, in their state of inebriation, each chooses a random cabin to sleep in.

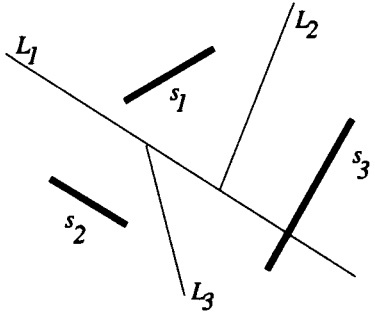
What is the expected number of sailors sleeping in their own cabins.

$$E[\sum_{i=1}^{40} X_i] = \sum_{i=1}^{40} E[X_i] = 1$$

Binary Planar Partition

Given a set $S = \{S_1, S_2, \ldots, S_n\}$ of non-intersecting line segments in the plane, we wish to find a binary planar partition such that every region in the partition contains at most one line segment (or a portion of one line segment).

An Example



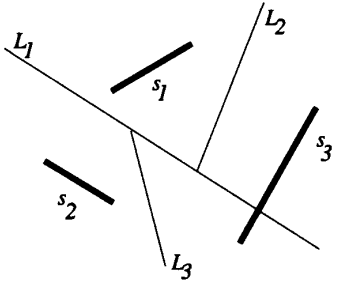
Binary Partition Tree

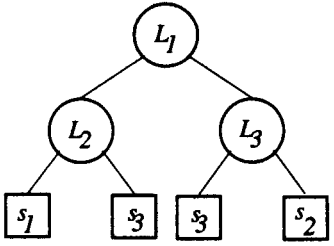
A binary planar partition consists of a binary tree together with some additional information.

Associated with each node v is a region $r(v)$ of the plane, and with each internal node v is a line $l(v)$ that intersects $r(v)$.

The region corresponding to the root is the entire plane. The region $r(v)$ is partitioned by $l(v)$ into two regions $r_1(v)$ and $r_2(v)$, which are the regions associated with the two children of v.

The Example





Remark

Because the construction of the partition can break some of the input segments S_i into smaller pieces, the size of the partition need not be n.

it is not clear that a partition of size $O(n)$ always exists.

Autopartition

For a line segment s, let $l(s)$ denote the line obtained by extending s on both sides to infinity.

For the set $S = \{s_1, s_2, \ldots, s_n\}$ of line segments, a simple and natural class of partitions is the set of autopartitions, which are formed by only using lines from the set $\{l(s_1), l(s_2), \ldots, l(s_n)\}$

An Algorithm

`AutoPartition(`$\{s_1, s_2, \ldots, s_n\}$`)`

Pick a permutation π of $\{1, 2, \ldots, n\}$ uniformly at random from the $n!$ possible permutations;

while *a region contains more than one segment* **do**

  cut it with $l(s_i)$ where i is first in the ordering π such that s_i cuts that region;

end

Theorem

The expected size of the autopartition produced by `AutoPartition` *is* $O(n \log n)$.

$index(u, v) = i$ if $l(u)$ intersects $i - 1$ other segments before hitting v.

$u \dashv v$ denotes the event that $l(u)$ cuts v in the constructed partition.

The probability of $index(u, v) = i$ and $u \dashv v$ is $1/(i + 1)$.

Let $C_{uv} = 1$ if $u \dashv v$ and 0 otherwise,

$$E[C_{uv}] = Pr[u \dashv v] \leq \frac{1}{index(u, v) + 1}$$

$$
\begin{aligned}
Pr(\text{partition numbers}) &= n + E[\sum_u \sum_v C_{uv}] \\
&= n + \sum_u \sum_v E[C_{uv}] \\
&= n + \sum_u \sum_{v \neq u} Pr[u \dashv v] \\
&\leq n + \sum_u \sum_{v \neq u} \frac{1}{index(u,v)+1} \\
&\leq n + \sum_u \sum_{i=1}^{n-1} \frac{2}{i+1} \\
&\leq n + 2nH_n
\end{aligned}
$$

Probability Basics and Fundamental Inequalities

Let $X_1 ... X_n$ be i.i.d. random variables. Define $X = \sum_{i=1}^{n} X_i$. Then as $n \to +\infty$, we have $X \sim N(n\,E(X_i), n\,Var(X_i))$.

This implies that the average of n i.i.d. random variables approaches $X \sim N(E(X_i), \frac{\mathsf{Var}(X_i)}{n})$.

We have $\mathsf{Var}(X_i) = E((X_i - E(X_i))^2) = 1/4$. Therefore, for $n = 1000$ we have

$$\sigma = \sqrt{\mathsf{Var} \sum_i X_i} = \sqrt{\sum_i \mathsf{Var} X_i} = \sqrt{n/4} = \sqrt{250} \approx 16$$

We can expect to be within 2σ (95% chance), so we probably get 470 to 530 heads. 600 is a big surprise!

Markov's inequality

Let X be a non-negative random variable. It is the case that: $E(X) \geq tPr[y \geq t]$, therefore:

$$Pr[y \geq t] \leq \frac{E(X)}{t}$$

Chebyshev's inequality:

Let X a random variable of variance σ^2 that can now take negative values. It is the case that:

$$Pr\left[|X - E(X)| > t\sigma\right] \leq \frac{1}{t^2}$$

Applying to Coins

Applying Chebyshev's inequality to n coins such that $E(X) = np$ and $Var(X) = \frac{n}{4}$, we have:

$$Pr\left[|X - E(X)| > t\sigma\right] \leq \frac{1}{t^2}$$

For $n = 1000$ coins, the probability of $|X - 500| > 2\sigma$ is less than $\frac{1}{4}$. In other words, we expect with probability greater than $3/4$, the number of heads to be between $[468, 516]$.

Randomized Complexity Classes

Common Complexity Class

P: Deterministic Polynomial Time. We have $L \in \mathbf{P}$ iff there exists a polynomial-time algorithm that decides L.

NP: Nondeterministic Polynomial-Time. We have $L \in \mathbf{NP}$ iff for every input $x \in L$ there exists some solution string y such that a polynomial-time algorithm can accept x if $x \in L$ given the solution y.

PP: Probabilistic Polynomial. $L \in \mathbf{PP}$ iff there exists a polynomial-time algorithm A s.t.

- if $x \in L$ then A accepts with probability $\geq 1/2$.
- if $x \notin L$ then A rejects with probability $> 1/2$.

BPP: Bounded Probabilistic Polynomial. $L \in \mathbf{BPP}$ iff there exists a poly-time algorithm A s.t.

- if $x \in L$ then A accepts with probability $\geq 2/3$
- if $x \notin L$ then A rejects with probability $\geq 2/3$.

Relations

$$\mathbf{P} \subseteq \mathbf{ZPP} \subseteq \mathbf{RP} \subseteq \mathbf{NP} \subseteq \mathbf{PP}$$

$$\mathbf{RP} \subseteq \mathbf{BPP} \subseteq \mathbf{PP}$$

Conjecture

$$\mathbf{P} = \mathbf{BPP} \subseteq \mathbf{NP}$$

Algorithm Design (XXIV)

Conclusion

Guoqiang Li
School of Software

Shanghai Jiao Tong University

What Is Algorithm

Algorithm Design

Basic algorithms:

- Recursion
- Algorithms on Lists, Trees and Graphs

Advanced strategies:

- Divide and Conquer
  - Master Theorem
- Dynamic Programming
- Greedy
- Duality
- Reduction
- Approximation
- Randomization
- Computational Geometry
- Algorithms on Massive Data
- ...

Graphs

- undirected graphs, directed graphs.
- DAG.
- bipartite.
- graphs with weights.
- ...

Network flows

- Ford-Fulkerson algorithm, Edmonds-Karp algorithm
- ...

Computational Geometry

Big-O Notation (Ω, Θ)

Advanced Methodology:

- Probability Analysis
- Amortized Analysis
- Competition Analysis

Standard Algorithms

- Sorting
- Searching & Hashing
- Strongly connected components
- Finding shortest paths in graphs
- Edit distances
- Minimum spanning trees in graphs
- Matchings in bipartite graphs
- Maximum flows in networks

Data Structure

- Balance trees, red-and-black trees
- Kripke structure, automata
- Priority queue
- Disjoint set
- Ordered binary decision diagrams (OBDD)
- ...

Computational Complexity

Church-Turing Thesis

Complexity class

- P, NP, Co-NP, NPI, NP-complete
- PSPACE
- RP, ZPP

Handling hard problems

- Simplex, DPLL(CDCL)(backtracking)
- Approximation,
- local search
- treewidth

The Door of Algorithms Will Open!

Roadmap

SHANGHAI JIAO TONG UNIVERSITY

	算法策略					算法结构		
	分治法	动态规划	贪婪	规约	对偶	图	流	数
基本问题与算法	排序问题、中位数等	最长公共子序列、编辑距离等	最小生成树、哈夫曼编码等	图、树上的常规算法	最大流最小割、最大匹配最小顶点覆盖等	深度搜索、广度搜索、DAG图、最短路径	福特弗格森算法	大数问题、模问题
理论分析方法	大师定理、Akra-Bazzi定理	树宽、时空转化，自顶向下，自底向上	—	难易问题划分	原问题-对偶问题	优先队列、并查集	良结构系统证明方法	概率分析、大数分析
高级问题与算法	快速傅里叶变换	马尔科夫链、序列比对、树宽等	近似算法	复杂性类问题Karp规约、图灵规约	拉格朗日对偶	强连通子图、Bellman-Ford算法、图同构	Dinitz算法	公钥加密、一次一密
工程算法与具体应用	工程快速傅里叶变换算法	动态规划中的空间压缩、马尔科夫链	Boruvka算法、簇聚类算法	DPLL/CDCL算法、不变量生成	神经网络的验证方法	Kosaraju算法、Tarjan算法、形式验证算法	前项流推动算法	Miller-Rabin算法

Guidelines of This Exam

Algorithms in This Lecture

Algorithm Strategies

- divide and conquer
- dynamic programming
- greedy algorithms
- duality
- reduction

Specific algorithms

- algorithm with numbers
- graph algorithms
- network flows

NP problems

- NP, Co-NP, NPC
- reduction
- handling NPH problem

First of ALL

Hand in ALL homework!

Languages

The exam is given in Chinese,

with translation sheet for international students.

Types of the Exam Problems

M1. Show modelling ability, proof ability, and algorithm analysis ability (20')

M2. Adopt algorithmic strategies to solve and analyze problems (greedy, D&C, DP, etc.) (30')

M3. Design algorithms and analysis on numbers, graphs, and flows. (25')

M4. Prove a NPC problem (15')

M5. Cope with NPH problem (10')

Types of the Exam Problems

M1. Show modelling ability, proof ability, algorithm analysis ability (20')

- given an problem, try to model it formally.
- proof the correctness of a simple algorithm.
- give an analysis to a piece of Pseudo codes.
- given a linear programming, figure out its duality, and find out the optimization solution.

Types of the Exam Problems

M2. Adopt algorithmic strategies to solve and analyze problems (greedy, D&C, DP, etc.) (30')

- Divide and conquer (master theorem)
- Dynamic programming (design, border conditions, complexity)
- Greedy
- Reduction
- Duality

Types of the Exam Problems

M3. Design algorithms and analysis on graphs, numbers and flow (25')

- DFS, BFS
- Shortest path, MST
- Algorithms on DAG
- Algorithms on numbers (modular)
- Applications of network flows

Types of the Exam Problems

M4. Prove an NPC problem (15')

- Prove an NP problem
- Prove an NPC problem

Types of the Exam Problems

M5. Cope with NPC problem (10')

- Approximation algorithm
- Backtracking
- Local search

Exam This Year

	1	2	3	4	5	6	7	Total
M1: Modeling, proof and analysis (20')	5	5				10		20
M2: Strategies (30')	10		15				5	30
M3: Graph, flow, and number (25')		10					15	25
M4: Prove NPC (15')					15			15
M5: Handle NPH (10')				10				10
Total	15	15	15	10	15	10	20	100