Computer Science/Mathematics 3804

Design and Analysis of Algorithms I

Course Outline

Class Schedule

Classroom:	Tory Building 447
Class Times:	Wednesdays and Fridays 4:05 pm - 5:25 pm
Course Website:	http://scs.carleton.ca/~sack/teaching/3804

Instructor Information

Professor	Office	Telephone	Email	Office Hours
Dr. J.-R. Sack	HP 5350	N/A	sack@scs.carleton.ca	Wednesdays 10:30-12:00

Teaching Assistants

T.A.	Office Hours	Room Number
Amin Gheibi	Thursdays 3-5 p.m.	1170 HP
Sarah Liske	Every Wednesday 3-4 after an assignment has been handed back	1170 HP

Course Outline cont'd

Course Description

An introduction to the design and analysis of algorithms.

Topics Covered

Topics include: recurrence relations, sorting and searching, divide-and-conquer, dynamic programming, greedy algorithms, NP-completeness.

Prerequisites

COMP 2002 or COMP 2402, and either COMP 1805 or both of MATH 2007 and MATH 2108, or equivalents.

Textbook (s)

Algorithms by S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani, (2007) Published by McGrawHill also available online from the first author's web-page

alternate reference – more detailed:

Introduction to Algorithms (3rd Edition) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein (2009). ISBN 0-262-03384-4. Published by MIT Press.

The Appendices A,B, and C provide a useful summary of material (much of it) should be known.

Course Outline cont'd

Component	Weight	Due Date
Assignment 1	10%	Sept 28th before class
Assignment 2	10%	Oct 19th before class
Assignment 3	10%	Nov. 9th before class
Assignment 4	10%	Nov. 30th before class
Midterm Exam	20%	October 31st during class
Final Exam	40%	tba

Course Outline cont’d

Excerpts from Calendar; refer to Calendar

September 6 classes begin. **September 19** Last day for registration. Last day to change courses or sections for fall/winter and fall term courses. **September 30** Last day to withdraw from fall term and fall/winter courses with a full fee adjustment. **October 5, 2012** University Day at Carleton. Undergraduate classes suspended. **November 9, 2012** Last day to submit, to the Paul Menton Centre for Students with Disabilities, Formal Examination Accommodation Forms for December examinations. **December 3, 2012** Fall term ends. Last day for academic withdrawal from fall term courses

Final Exam Note

The final exam will be scheduled by the University.

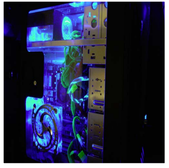
Assignments

Late assignments will not be accepted.

Collaboration Policy

Students are encouraged to collaborate on assignments, but at the level of discussion only. When writing down the solutions, students must do so in their own words.

Computer Science

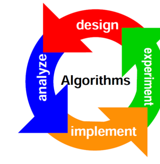


Hardware

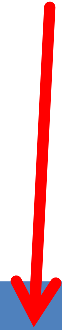
- Hardware
- Communication Infra-structure

Software

- Programs
- Software Infra-structure



Efficient algs+data str.



- Algorithms
- Data Structures

Introduction

- "Algorithms change/d the world"
- Precise instructions of how to add, subtract, multiply, divide, extract square roots, get digits of π, … [Al Khwarizmi, AD 600, Baghdad]
- Unambiguous, precise, mechanical, efficient and correct

Three questions to ask - algorithm

After we understand what it does or is supposed to do.

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

A Sequence

- The first 21 numbers are:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765

What can we say about this sequence?

Many things

- Non-negative
- Increasing
- Rapidly increasing almost as fast as 2^n , in fact a good approximation is: $2^{0.694n}$
- If we call F_n the nth number then
  $F_n = F_{n-2} + F_{n-1}$ where n>1 and
  $F_1 = 0$ and $F_0 =1$.

Fibonacci Numbers

F_n is called the nth Fibonacci number.

Fibonacci was living between 1170 and 1250.

Fibonacci Occurrences in Nature

Number of Petals	Flower
3 petals (or 2 sets of 3)	lily (usually in 2 sets of 3 for 6 total), iris
5 petals	buttercup, wild rose, larkspur, columbine (aquilegia), vinca
8 petals	delphinium, coreopsis
13 petals	ragwort, marigold, cineraria
21 petals	aster, black-eyed susan, chicory
34 petals	plantain, daisy, pyrethrum
55 petals	daisy, the asteraceae family
89 petals	daisy, the asteraceae family

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

Pineapple

The number of spirals going in each direction is a Fibonacci Number.

Here, there are 13 spirals that turn clockwise and 21 curving counterclockwise.

In sunflowers, the number of clockwise and counterclockwise spirals will always be consecutive Fibonacci Numbers like 21 and 34 or 55 and 34.

Algorithm I for Fibonacci

- Function fib1(n)

```
if n = 0 return 0
if n = 1return 1
return fib1(n-1) + fib1(n-2)
```

The Three Questions

1. Is the algorithm correct?
2. How much time does it take to compute?
3. Can we do better?

1. Is easy that is simply the definition of the Fibonacci sequence.

2. How much time does it take to compute?

- The time is a function of n, let us call it T(n)
- $T(n) \leq 2$, for $n \leq 1$ why?
- For larger values of n, i.e., $n>1$

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

so, we can see that $T(n) \geq F(n)$

bad news?!

- to compute T(200) ≥ F(200) ≥ 2^{138} steps would be required
- this is huge the NEC Earth Simulator can do 40 trillion steps per second. Thus, to compute F(200) would take at least 2^{92} seconds that is about 78509642268225537474 years.
- The earth is expected to live to 5 billion (5000000000) years only.

So, can we do better?

3. Can we do better?

Why is fib1 so bad?

```
Function fib1(n)
    if n = 0 return 0
    if n = 1return 1
    return fib1(n-1) + fib1(n-2)
```

a second attempt fib2

Function fib2(n)

```
if n = 0 return 0
create an array f[0 .. n]
f[0] = 0, f[1] = 1
for i = 2 ... n
      f[i] = f[i-1] + f[i-2]
return f[n]
```

The three questions

1. Is the algorithm correct?
2. How much time does it take to compute?
3. Can we do better?

1. Is the algorithm correct?
   again, here the correctness is self-evident
   as it follows from the definition of Fibonacci.

2. How much time does it take to compute?

- Except of the small number of constant operations the main work is carried out in the loop.
  - the loop body is one addition
  - and is executed n-1 times

So, the total number of operations taken by fib2 is linear in n.

Careful

- We get that fib2 executes a linear number of additions. However, the numbers are huge.
- Our standard model of analysis assumes that the numbers have a constant number of bits (32 or 64 …).
- Here the numbers are about 0.694n bits long!
- Such numbers cannot be added in one step!
- Adding 2 n-bit numbers takes time proportional to n.

Conclusion

- Lemma: The running time of fib2(n) is proportional to n^2.

- What is the running time of fib1(n) then?

3. Can we do better? YES, see Exercise 0.4 in book

Recall Big-O

- Let $f(n)$ and $g(n)$ be functions from the positive integers to the reals. We say that

  $f = O(g)$ if there is a constant $c>0$ such that

  $f(n) \leq c*g(n)$.
- This means that f grows no faster than g.
- Also, the constant c allows us to ignore small values of n.

Some examples

- $f(x) = x^2 + \log(x) + 7x + 100000000$

Course requirement

- I will assume that you are familiar with big-O.

Divide&Conquer Algorithm

- Recall D&C
- Break problem into subproblems that are smaller instances of the same type of problem
- Recursively solve the subproblems
- Combine the answers to the subproblems into an answer to the original problem

Multiplication "*"

<u>Gauss</u>   <u>Multiplication of complex number</u>

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

4 "*"

but, he saw it can be done with 3 "*"

ac, bd and $(a+b)(c+d)$

Since

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

Seems insignificant
3 "*" instead of 4 "*"

but, when applied recursively, it becomes significant!

$X = x_1 \dots x_{\lfloor n/2 \rfloor} \dots x_n$ 2 n-bit number

$Y_1 = y_1 \dots y_{\lfloor n/2 \rfloor} \dots y_n$

assume n is even

$X =$ [X_L] [X_R] $= 2^{n/2} X_L + X_R$

$x_1 \dots x_{\lfloor n/2 \rfloor} \dots x_n$

similarly,

$Y =$ [Y_L] [Y_R] $= 2^{n/2} Y_L + Y_R$

$$xy = (2^{n/2} X_L + X_R)(2^{n/2} Y_L + Y_R)$$

$$= 2^n X_L Y_L + 2^{n/2} (X_L Y_R + X_R Y_L) + X_R Y_R$$

↑ ↑ ↑ ↑

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

↑

4 "x" of 2 $n/2$-bit numbers

$O(n) \leftarrow$ [2^n or $2^{n/2}$ · is a shift each
⇒ linear
addition linear]

$$T(n) = 4T(n/2) + O(n) \overset{?}{=} O(n^2)$$

(see later)

Now, Gauss $\Rightarrow$ 3 "*" only

$X_L Y_L$, $X_R Y_R$ and $(X_L + X_R)(Y_L + Y_R)$

$\Rightarrow$

$$T(n) = 3T(n/2) + O(n) = O(n^{1.59})$$

Recursion tree

# of problems $1 = 3^0$

$3 = 3^1$

$9 = 3^2$

level k: problemsize $n/2^k$

\# of problems 3^k

(root at level 0)

\# of levels $\log_2 n$

How much work is done?

level 0 $O(n)$

1 $\frac{3}{2} n$

⋮

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

⋮

$\log_2 n$ $3^{\log_2 n} * O(1) = n^{\log_2 3}$

geometrically increasing series

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

Thus, the run-time of multiplying 2 n-bit numbers is not quadratic, but using Gauss + D&C $O(n^{1.59})$

can still be improved using Fast Fourier transform (D&C)

Note:

$$3^{\log_2 n} = n^{\log_2 3}$$

Why?

$$\Updownarrow$$

$$\log_3 (3^{\log_2 n}) = \log_3 (n^{\log_2 3})$$

$$\Updownarrow$$

$$\log_2 n = \log_2 3 \log_3 n$$

$$\Updownarrow$$

$$\frac{\log_2 n}{\log_2 3} = \log_3 n$$

The last equation holds base transform

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

Proof:

$$b^{\log_b x} = x$$

$$\log_a b \cdot \log_b x = \log_a x \qquad \square$$

—

Recurrence Relations

Many D&C algorithms can be analyzed as follows:

Let a = # of subproblems
$\lceil n/b \rceil$ = size of subproblems
n = total size

$$T(n) = a \cdot T(\lceil n/b \rceil) + O(n^d)$$

then,

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

Master Theorem

Proof (sketch)

assume w.l.o.g. $n = b^j$, for some j.

work on level k of recurrence tree

$$a^k \cdot O\left(\frac{n}{b^k}\right)^d$$

- a^k ← # of sub-problems on level k
- $\frac{n}{b^k}$ ← their sizes
- d ← work due to $O(n^d)$

$$a^k \cdot O\left(\frac{n}{b^k}\right)^d = O(n^d)\left(\frac{a}{b^d}\right)^k$$

$k = 0, \ldots, \log_b n$

$\Rightarrow$ geometric series with ratio a/b^d

Cases:

1. $\frac{a}{b^d} < 1 \Rightarrow$ sum is the first term i.e. $O(n^d)$
2. $\frac{a}{b^d} = 1 \Rightarrow$ all $O(\log n)$ terms are equal to $O(n^d)$

③ $\frac{a}{b^d} > 1 \Rightarrow$ Series is increasing and its sum = last term $O(n^{\log_b a})$

$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right)$$

$$= a^{\log_b n} = a^{\log_a n \log_b a}$$

$$= n^{\log_b a}$$

□

Recale Geometric Series

$$1 + r + r^2 + \cdots$$

$$= \lim_{n\to\infty} (1 + r + \cdots + r^n)$$

$$= \lim_{n\to\infty} \frac{1 - r^{n+1}}{1 - r} \qquad r^{n+1} \to 0, \text{ for } |r| < 1$$

$$a + ar + ar^2 + \cdots + ar^{n-1} = a\frac{1 - r^n}{1 - r}$$

Examples Master Theorem

I. Binary Search

$$T(n) = T(\lceil n/2 \rceil) + O(1)$$

$a = 1 \qquad b = 2 \qquad d = 0$

$\log_b a = \log_2 1 = 0$

$d = 0 = \log_2 1 \Rightarrow$

Case B:

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

$$= O(\log n) \qquad d = 0$$

Binary Search: $O(\log n)$

II. Merge Sort

```
function mergesort (a[1..n])
input:  An array of numbers a[1..n]
output: A sorted array of those numbers

   if n > 1
      return merge(mergesort(a[1..⌊n/2⌋]),
                   mergesort(a[⌊n/2⌋+1, ... n]))

   else
      return a
```

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

∘ = concatenation

Observation 1) function merge is linear in $k+l$, i.e. the run time is $O(k+l)$

2) function mergesort's run time is given by

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

Theorem MergeSort sorts n numbers in $O(n \log n)$ time.

"The three questions"

① Is the algorithm correct?

< see class >

② Time see above $O(n \log n)$

③ Can we do better?

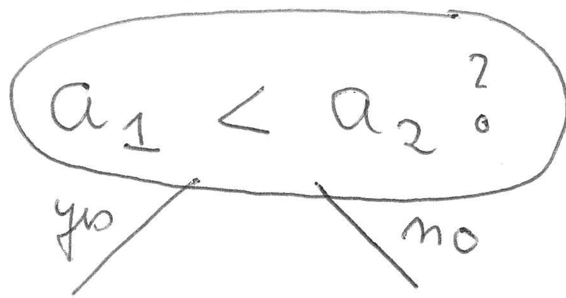
A lower bound for Sorting

We will show that no algorithm can exist that sorts n numbers with "fewer than "$n \log n$" comparisons".

So, $\Omega(n \log n)$ is a lower bound on the number of comparisons required to sort n numbers.

(numbers or elements we can compare)

comparison



$(a_1 \geq a_2)$
if elements are distinct $a_1 > a_2$

Example Sorting 3 element
1, 2, 3 stored somehow in a_1, a_2, a_3

$a_1 < a_2$?
yes → $a_2 < a_3$?
- yes → 123
- no → $a_1 < a_3$
  - yes → 132
  - no → 312

no → $a_1 < a_3$?
- yes
- no

fill in yourself

Consider now sorting n comparable elements.

Any algorithm that is based on comparisons that sorts these elements has an execution that can be described by such a comparison tree.

Every permutation of the n elements is possible & the algorithm must be able to sort them.

- Any permutation must appear at a leaf.
- There are $n!$ leaves.

Comparison trees are binary trees.
Binary tree of depth d has at most 2^d leaves.

$\Rightarrow$ depth of tree $\geq \log(n!)$

$\log(n!) \geq c \cdot n \log n$, for some $c>0$

<u>Theorem:</u> Any comparison-based sorting algorithm has an $\Omega(n \log n)$ lower bound, to sort n element.

<u>Corollary:</u> Mergesort is an optimal comparison-based sorting algorithm.

Why is $\log(n!) \geq c \cdot n \log n$?

$$n! \geq \left(n/2\right)^{n/2}$$

$$\log n! \geq n/2 \log n/2$$

---

a better bound is derived from Stirling's approximation

$$n! \approx \sqrt{\pi\left(2n + \frac{1}{3}\right)} \cdot n^{n} \cdot e^{-n}$$

Median(s)

Median (informal def)
of a list of numbers is a
number s. t. half the numbers
are larger than it, and half
are smaller.

More precisely (and correctly)
The median (lower median)
of a set of n numbers, is
the element in position $\lfloor n/2 \rfloor$
after sorting the elements in
increasing order.

Motivation:

see class

Algorithms for Median

① < brute force > see class

② A randomized D&C algorithm for selection

Selection is more general than median finding

Selection (S, k)

<u>input</u>. S: set of n elements (orderable)

k: input parameter

<u>Output</u>: kth smallest element of S

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

$S_L = \{x \in S \mid x < v\}$

$S_v = \{x \in S \mid x = v\}$ (duplicates of v in S)

$S_R = \{x \in S \mid x > v\}$

<u>Example</u>.

see class

Run time: S_L, S_v, S_R can be computed in linear time + space.

After the split the runtime is proportional to $\max\{S_L, S_R\}$

① we are lucky
$$|S_L| = |S_R|$$
$$T(n) = T(n/2) + O(n)$$
$$= \qquad ?$$
(see class)

② we are unlucky
one of $|S_L|, |S_R| = 0$ ~~(or 1)~~
e.g. for median
$$n + (n-1) + \dots + (n/2) = \Theta(n^2)$$

③ we are sort of lucky

v is called good if it
is between the $n/4$ ... and $n\,3/4$
element every time (n changes)

then $T(n) \leq T(\frac{3}{4}n) + O(n)$

still good! $O(\log n)$

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

Proof. Let E = exp # of ~~exp~~ tosses before
heads is seen

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

1 ↑ 1 toss if heads done

$\frac{1}{2}E$ ↑ in half the cases we recurse

Therefore, after 2 split operations on average, the array will shrink to at most 3/4 of its size

(a randomly chosen v has a probability of 0.5 to be good so 2 random choices of v on average suffice)

Theorem. The *expected* run-time of the Selection Algorithm is $O(n)$

NOT in text book

Instead of an *expected* running time, we can achieve a guaranteed linear running time for selection.

Select (S, k)

1. Divide the n-element input array into $\lfloor n/5 \rfloor$ groups of 5 elements each + one group of the remaining $n \bmod 5$ elements. (illustration see class)

2. Find the median of each of the 5 element group by sorting them (say insertion sort; $O(1)$)

3. Recursively find the median x of the $\lceil n/5 \rceil$ medians (from 2.)

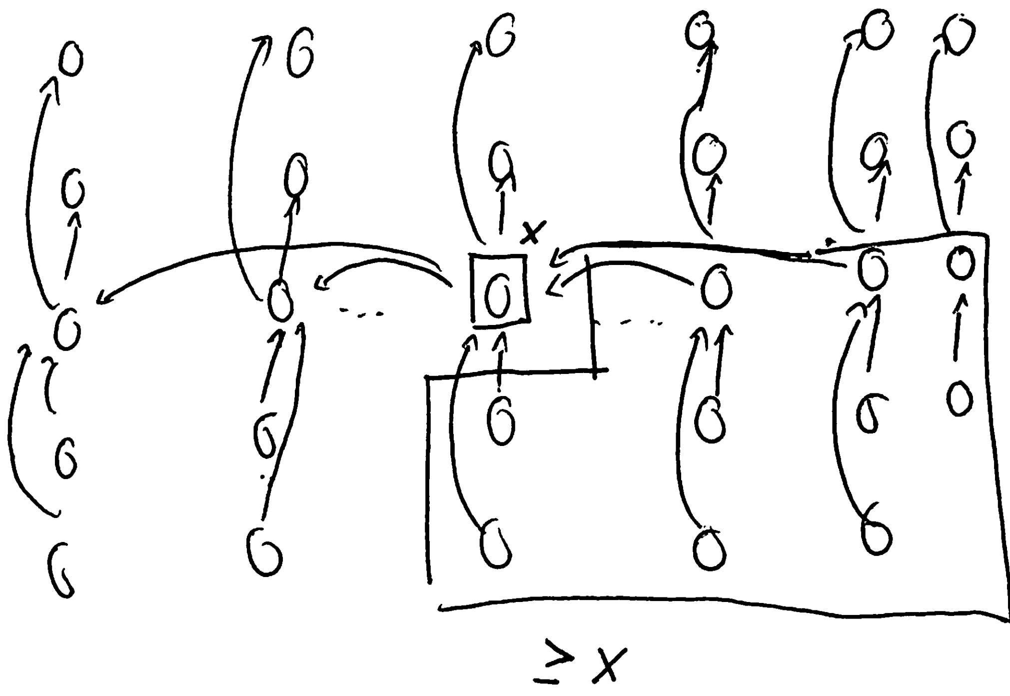
4. Partition the input array around x (from 3.)

Let $q :=$ # of elements $\leq x$; call the set S_L

$n-q :=$ the remaining elements; call this set S_R

5. if $q < k$ recursively call Select(S_R , $k-q$)

$q \geq k$ (S_L , k)



$$3 \cdot \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$$

- 3 ↑ elements
- $\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil$: half of the group
- -2 ↑ two groups not counted that have fewer than 5 elements or contain x

# of elements guaranteed to be $\geq x$ is $\frac{3n}{10} - 6$

analogously, for the # of elements guaranteed to be $\leq x$

Analysis:

Steps 1, 2 and 4 : $O(n)$

Step 3 : ~~O~~ $T(\lceil n/5 \rceil)$

Step 5 : $T(\underbrace{\frac{7n}{10}+6}_{\text{why?}})$

Notes:

$\frac{7n}{10} + 6 < n$ for $n > 20$

Sorting say 80 numbers $O(1)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 80 \\ T(\lceil n/5 \rceil) + T(\frac{7n}{10} + 6) + O(n), & \text{if } n > 80 \end{cases}$$

$$T(n) \leq c\lceil n/5 \rceil + c(7n/10 + 6) + O(n)$$

$$\leq cn/5 + c + 7cn/10 + 6c + O(n)$$

$$\leq 9cn/10 + 7c + O(n)$$

$$\leq cn$$

by def. of "O"

$$T(n) = O(n)$$

Theorem:

Selecting the kth element in a set of n elements takes $O(n)$ time.

Question: Why does this method not work with groups of 3 elements?

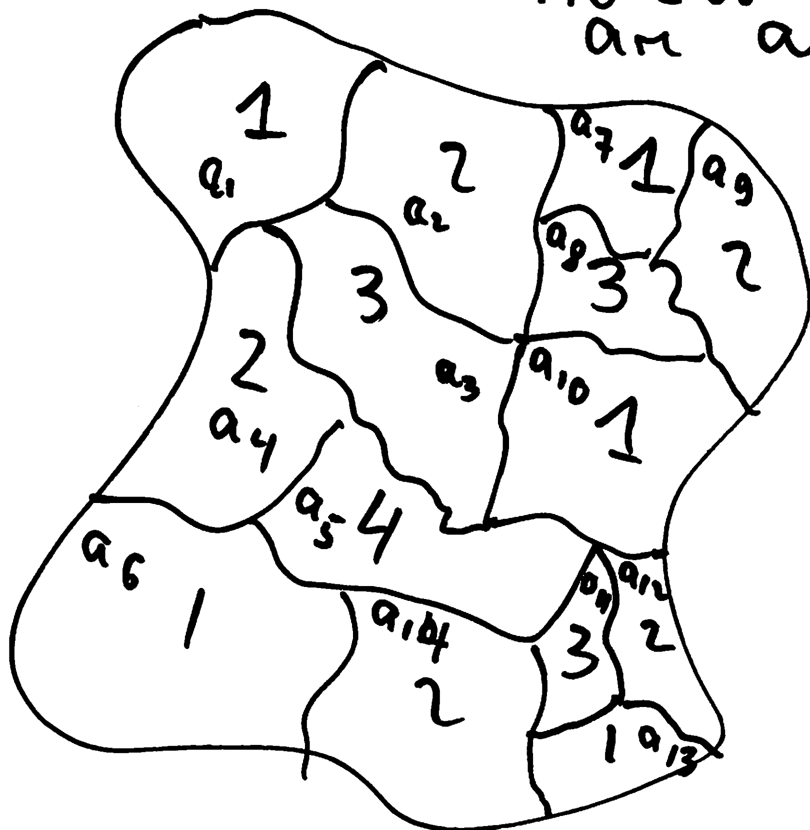
Chapter 3 [in your book]

Graphs have many... many applications.

1. Colouring a map
2. Scheduling an examination set
3. Describing industrial processes

⋮

1. Colouring a map

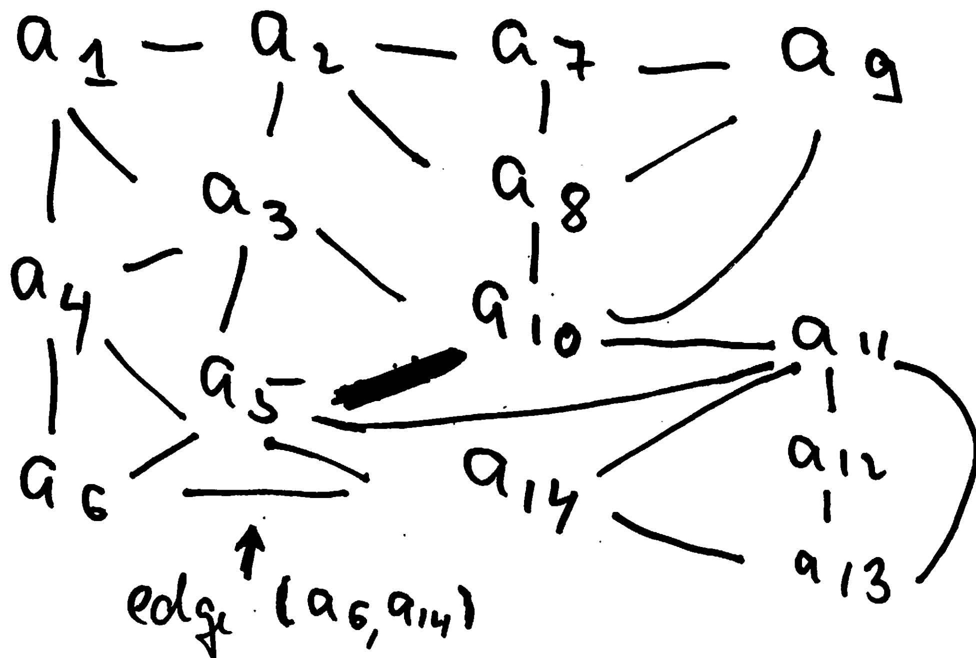


no two countries are allowed to share a border (except at a point) and have the same colour

This is best do with graph theory.

1) Vertices represent : countries
$a_1, \dots, a_{13}$

2) edges represent : 2 countries sharing a boundary



Graph $G = (V, E)$

$V :=$ set of vertices

$E :=$ set of edges
each edge connects a pair of vertices

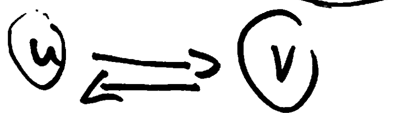
Directed Graphs

- (u) → (v) directed edges

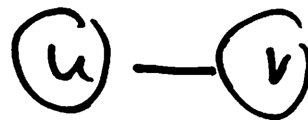
  (u,v) from u to v

  e.g., travel in one-way street

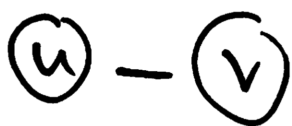
- Undirected edges



$\hat{=}$



directed in both directions — same as — undirected



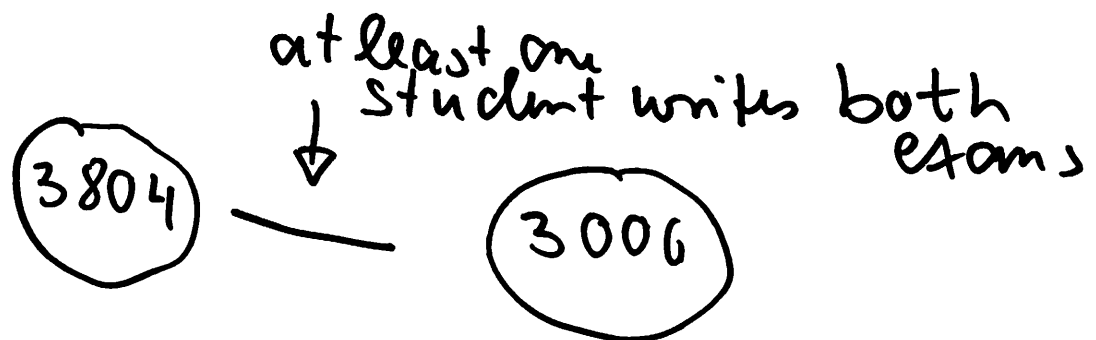
: you can go from u to v or from v to u

2. Scheduling Examinations

constraint: no student can write an exam at the same time

Vertices: exams

Edges: a student is taking both exams



Assigning time slots for the exams corresponds to graph colouring

3. Describing industrial Processes

<see class>

Representations of Graph

It is critical to represent a graph correctly.

Size of the graph can be huge.

Ex: Facebook
June 2012 955 M
active users

to represent the facebook graph

Vertices: users

$|V| = 955M$

Edges:

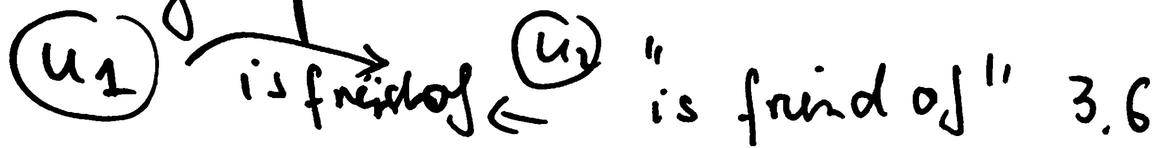
different possibilities of defining this
say friends!

$$|E| \le \frac{|V| \cdot |V-1|}{2}$$

if friends are mutual
i.e. undirected graph

$$|E| \le |V| \cdot |V-1|$$

if it is a directed graph



So, how many edges does the facebook graph have maximally?

$|E| \leq |V|\,|V|-1 = 955000000 * 954999999$

$\approx 9.1 * 10^{17}$

(9.1 20249990450000000)

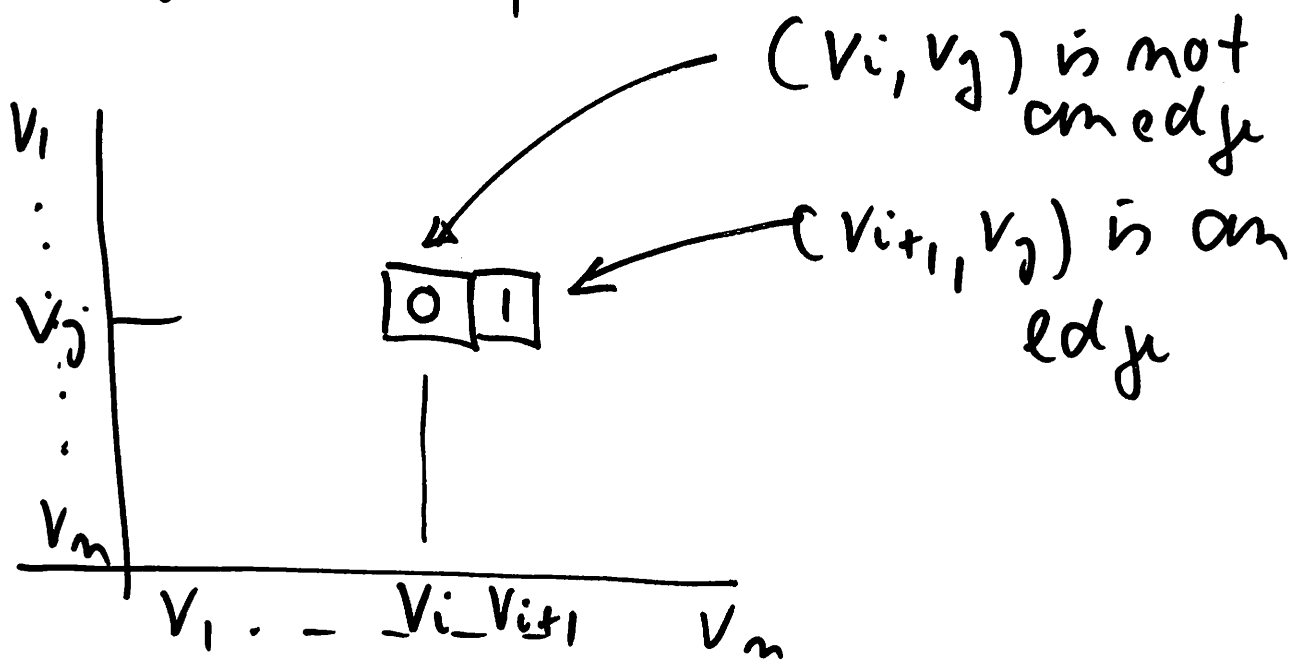
How to store graphs?

For this, we need to know what operations we want to carry out using the graph.

Why?.

because the cost of an operation could depend critically on the representation.

I. Bit-array



Storage cost: $|V|^2$ bits

access to an edge $O(1)$

For small graphs this is an efficient representation

For large graph storage cost is prohibitive, esp. if the number of edges is small compared to $|V|^2$.

<u>operations</u> (Part I)

- is (v_i, v_j) an edge of the graph?
- give me all edges $(v_i, -)$
  (so to which vertices is v_i adjacent?)

these need to be executed fast

Representation as Lists

For each vertex v_i: list of all vertices v_j s.t. $(v_i, v_j) \in E$

v_1 : $(v_5) \to (v_6) \to (v_{100}) \to (v_{205})$

.
.
.
.
.

v_i :

.
.
.

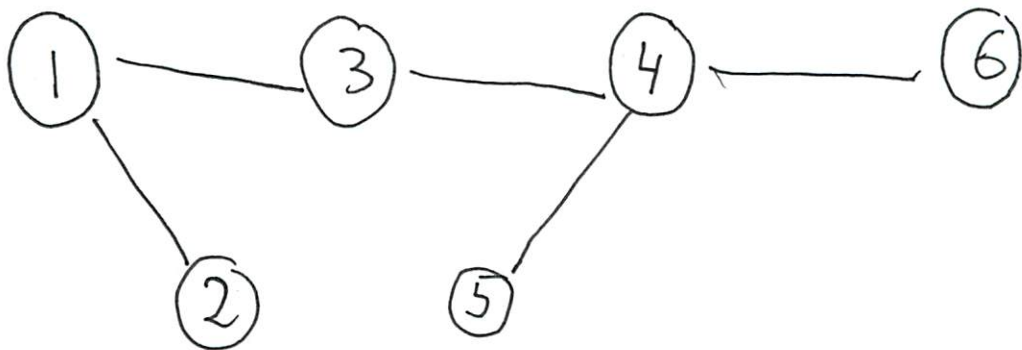
v_n :

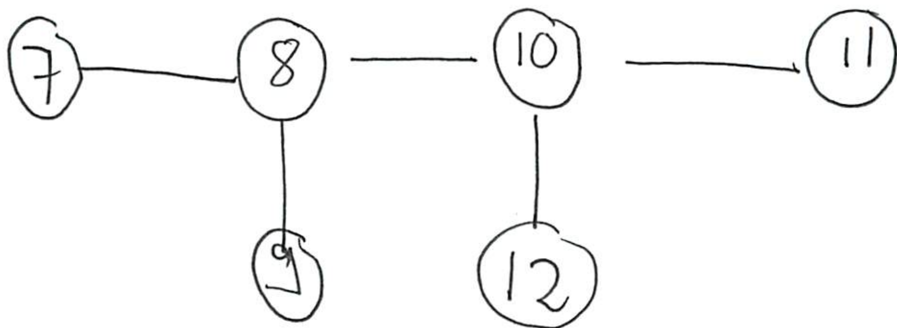
Storage : minimal, as required

however to see if say $(v_1, v_{100}) \in E$ takes linear time in the length of the list for v_1

Depth-first Search in Undirected Graph

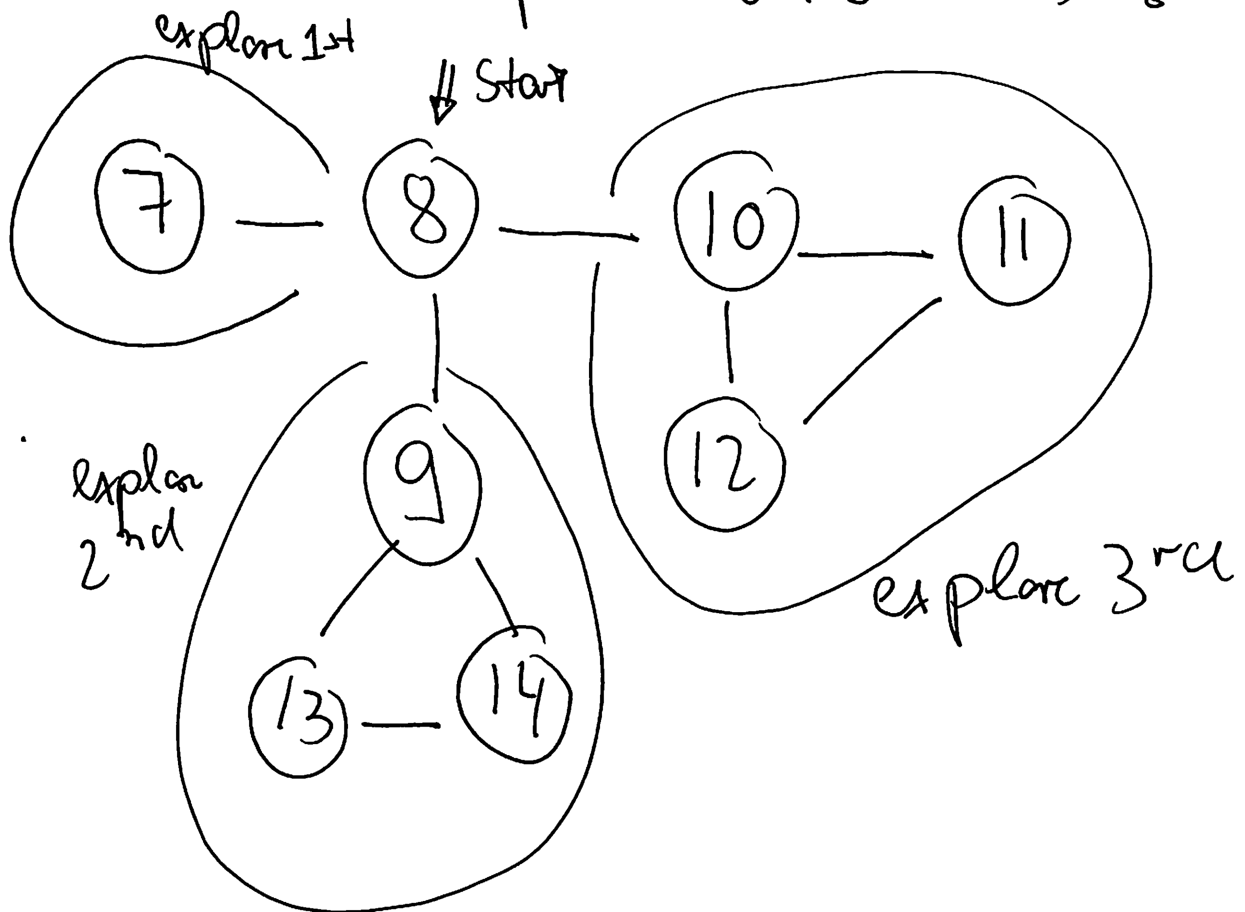
Q: Which parts of a graph are reachable from a given vertex





reachable from say 8
are nodes 7, 10, 11, 12 and 9

How to implement this?

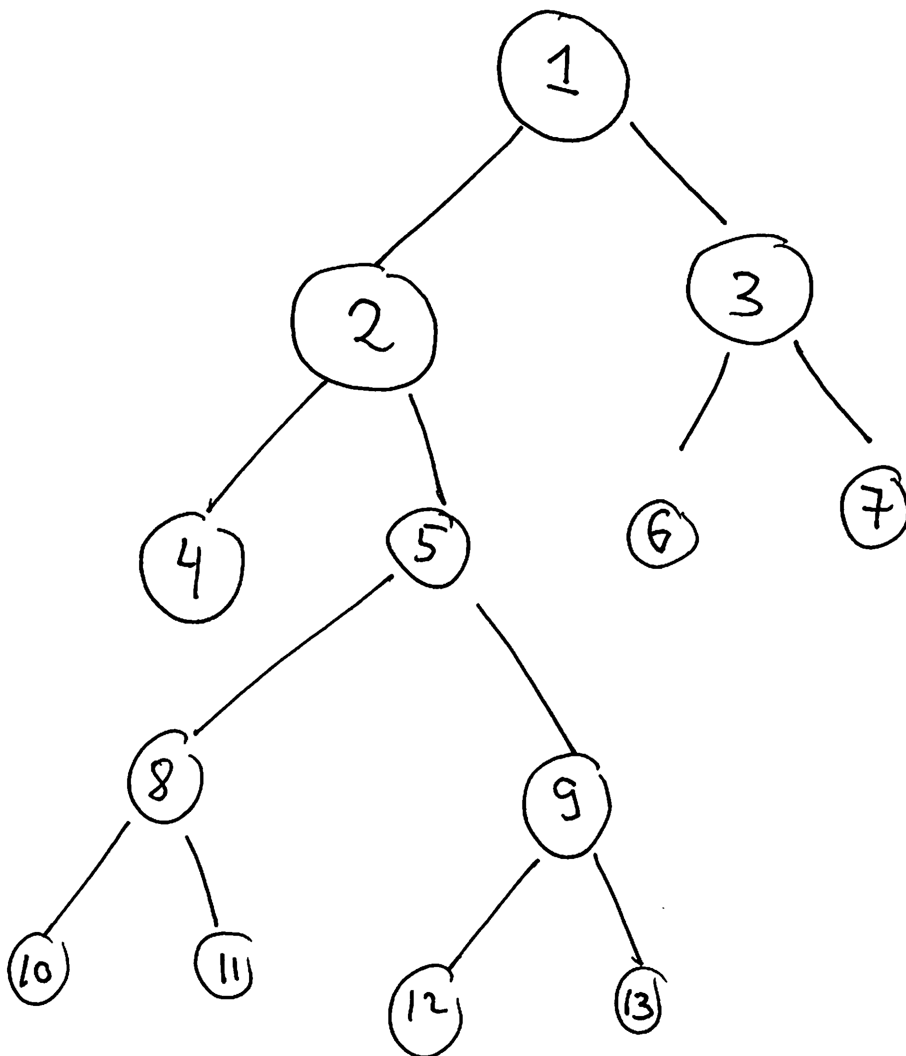


① we could explicitly use a stack

② we can use recursion

One difficulty are cycles because we need to know that we already visited nodes.

(A) Assume we deal with binary trees, i.e. graphs without cycles and outdegree at most 2



Assume a travel on the bicycle path

3 Methods

① explore each west-window and print the elements

② explore each south window and print the elements

③ explore each east window and print the elements

1
2 3
4 5 6 7
8 9
10 11 12 13

bicycle path

each node has 3 windows

① produces

1, 2, 4, 5, 8, 10, 11, 9, 12, 13, 3, 6, 7

② 4, 2, 10, 8, 11, 5, 12, 9, 13, 1, 6, 3, 7

③ 4, 10, 11, 8, 12, 13, 9, 5, 6, 7, 3, 1

3.16

Algorithms

① preordertravel (T, a)
Visit (a)
preordertraversal (T, leftchild(a))
Preordertraversal(T, rightchild(a))

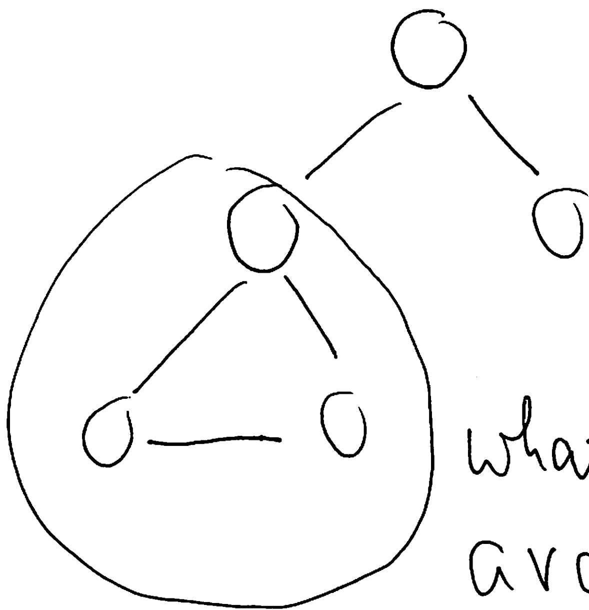
② inordertraversal (T, a)
inordertraversal (T, leftchild(a))
Visit (a)
inordertraversal (T, rightchild(a)

③ Postordertraversal (T, a)
postordertraversal (T, leftchild(a))
Postordertraversal (T, rightchild(a))
Visit (a)

3, 17

This generalizes to trees that have a larger than 2 Outdegree. [How?]

<u>Problem:</u> cycles



what to do here to avoid multiple visits

Idea: Mark what you have already seen

Procedure Explore (G, v)

<u>input</u>: a Graph $G = (V, E)$ and a node (vertex) $v \in V$

<u>output</u>: visited(u) is set to true for all nodes u reachable from v

```
visited(v) := true
previsit(v)
for each edge (v,u) ∈ E :
    if not visited(u) : Explore(u)
postvisit(v)
```

<u>Note</u>: at this point previsit and postvisit are not required

Why is the algorithm correct?

assume that a vertex w is reachable from v but the algorithm does not set visited(w) to true

Reachability from v means that there is a sequence of edges

$$(v, x_1)\,(x_1, x_2)\,\ldots\,(x_m, w)$$

assume that w is the first node in that which has visited(w) $\neq$ true

(otherwise, an inductive argument can be made)

then x_m is reachable + visited(x_m) = true

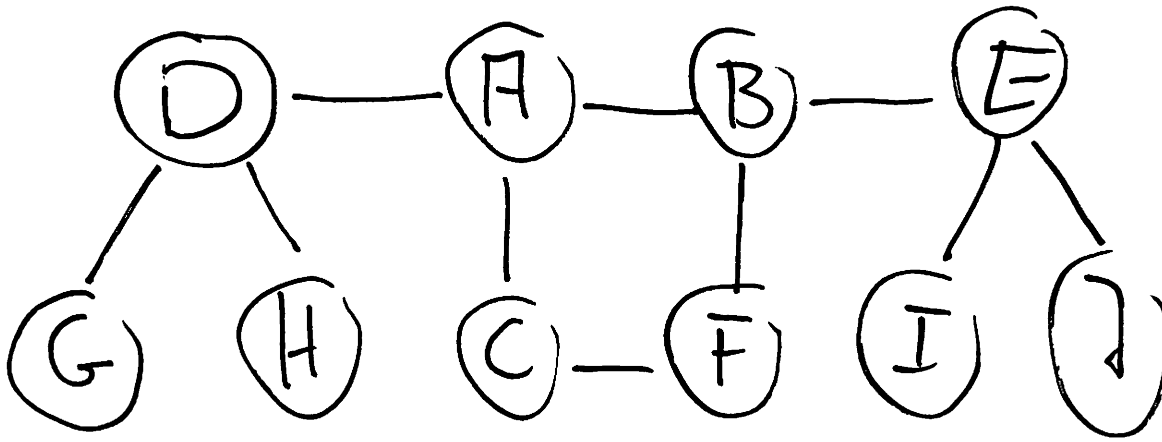
But $(x_m, w) \in E$ and the loop body would then explore (w)

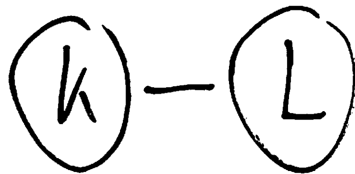
How fast is the algorithm?

$O(|E|)$

Can it be improved?

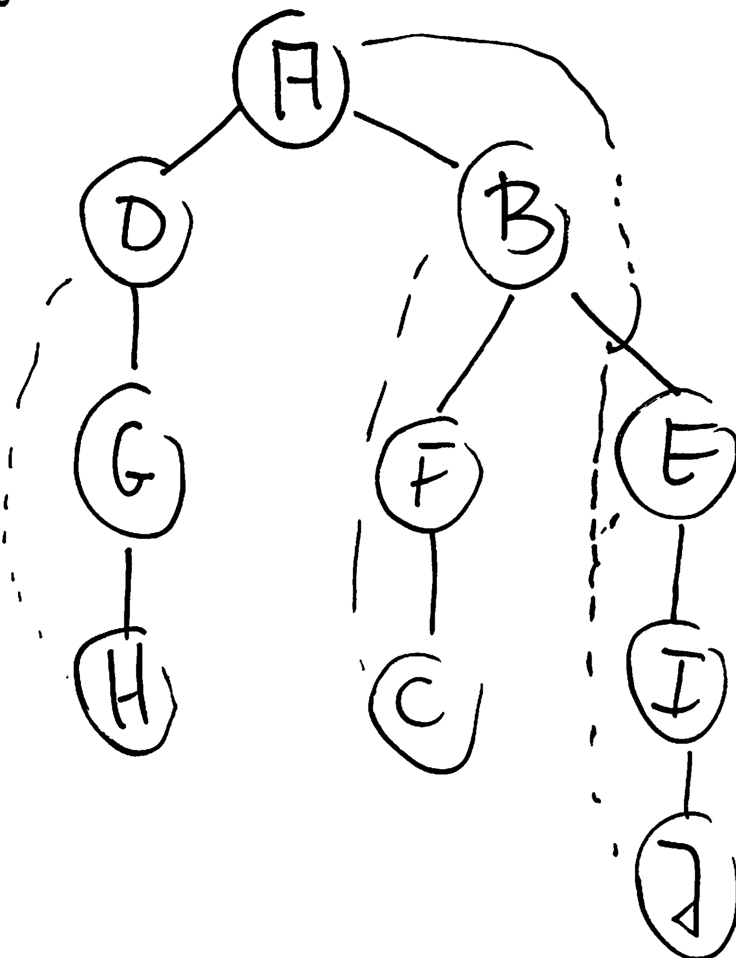
no we need to examine all edges





explore (G, A)

yields



3.22

Depth-first search

Task: visit all vertices (nodes) of a graph

Procedure dfs(G)

for all $v \in V$
    visited(v) = false
for all $v \in V$
    if not visited(v): Explore(v)

Is the algorithm correct?: Yes
Each vertex is explored and it is explored only once (due to visited array)

Complexity:

The loop goes over all vertices : $O(V)$

Each edge is examined twice
once as (x, y) and
once as (y, x)
(again as in Explore)

So, $O(|V| + |E|)$

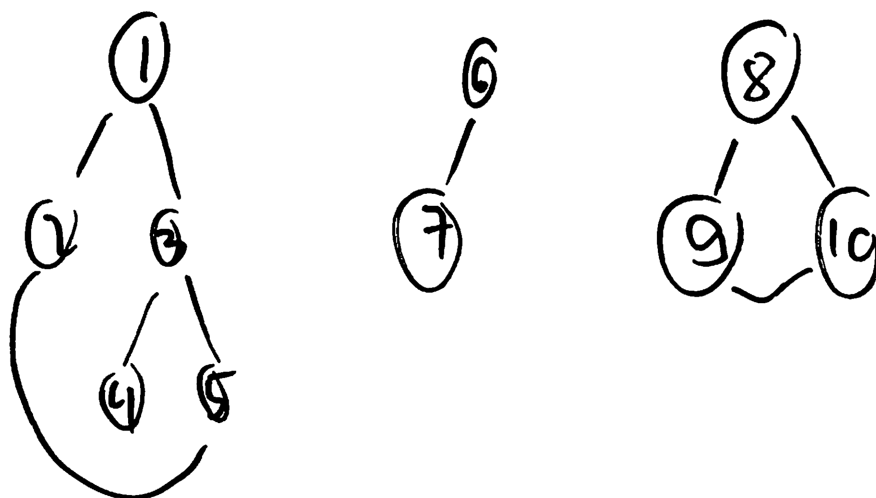
Note: the time per vertex differs in loop

Can we do better? No

$O(|V| + |E|)$ is optimal

Connected Components

in undirected graphs



3 connected components

$\{1,2,3,4,5\}$, $\{6,7\}$, $\{8,9,10\}$

Try to define yourself the connected components of a graph!

Observation:

The DFS procedure finds all connected components of a graph.

Each time the algorithm starts to examine a new ~~ver~~ vertex in the loop-body (for all $v \in V$) that has not been visited before. I.e.. Explore (v) is called, a new connected component is built.

procedure previsit(v)

ccnum[v] = cc

---

with cc initialized to 0
and then incremented each
time the loop calls explore

---

previsit will give each vertex
the number of the connected
component it belongs to

Previsit and postvisit Orderings

```
Procedure previsit(v)
  pre[v] = clock
  clock = clock+1

Procedure postvisit(v)
  post[v] = clock
  clock = clock+1
```

---

previsit captures the "time" of first discovery of a vertex
postvisit captures the "time" of final departure.

Can you link this to our discussion of tree traversals?

Master theorem: (general form)

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \geq 1,\ b > 1$$

Case 1: $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for $\varepsilon > 0$

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Case 2: $f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$$

Case 3: $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$

and it's also true that $a f\left(\frac{n}{b}\right) \leq c f(n)$ for $c < 1$ and sufficiently large n

$$\Rightarrow T(n) = \Theta(f(n))$$

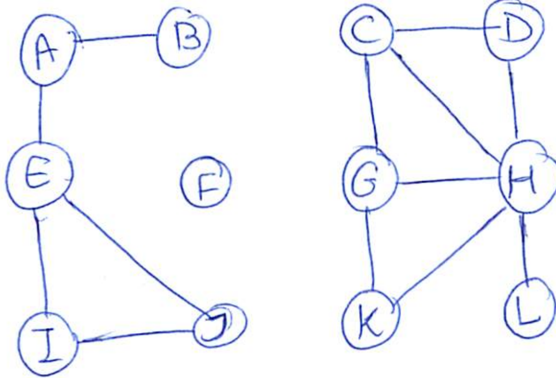
---

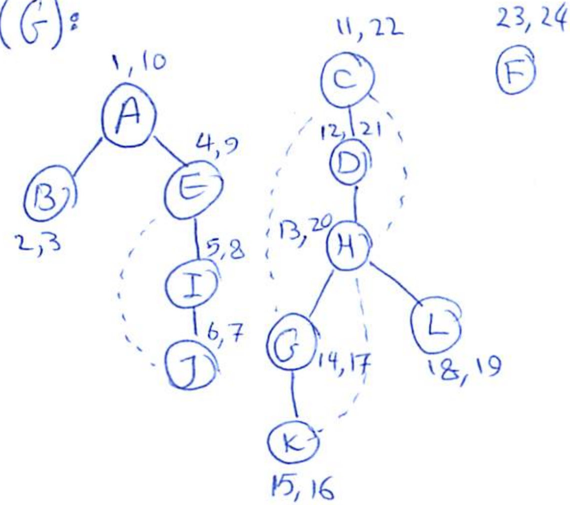
ex: $T(n) = 2T\left(\frac{n}{2}\right) + n^2$ Case 3: $T(n) = \Theta(n^2)$

Announcement: Errata for the text book:

cseweb.ucsd.edu/~dasgupta/book/errata.pdf

Review DFS by an example:





---

to collect more information about the graph, we record two type of events:

- First discovery (Pre visit)
- Final departure (Post visit)

[Pre, Post]

(node)

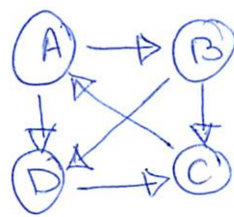
Property 1: $\forall u, v \in V$, two intervals $[Pre(u), Post(u)]$ and $[Pre(v), Post(v)]$ are either disjoint or one is contained in the other one.

why? last-in, first-out behaviour of stack

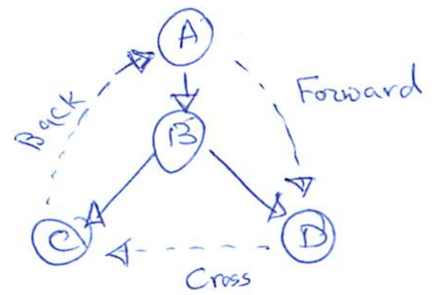
DFS for directed graphs: - What's a directed graph?

Algorithm is the same but movements by edges are in the direction of those edges.

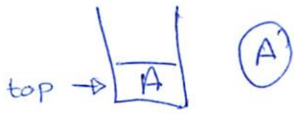
Example:

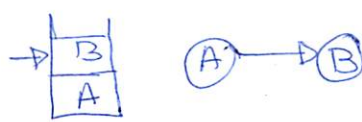


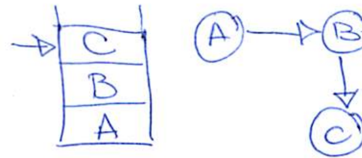
DFS ⟿

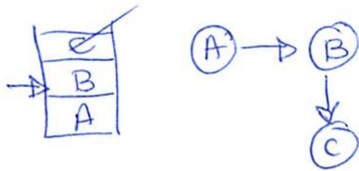


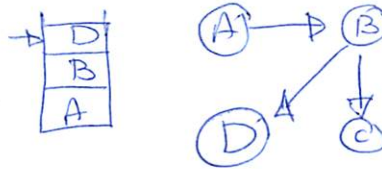
DFS steps with stack:

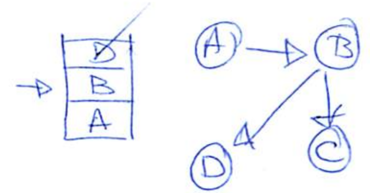


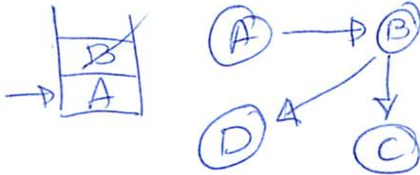












Terminology:

- "A" is the root of DFS tree, everything else is its descendant, and conversely, "A" is their ancestor.

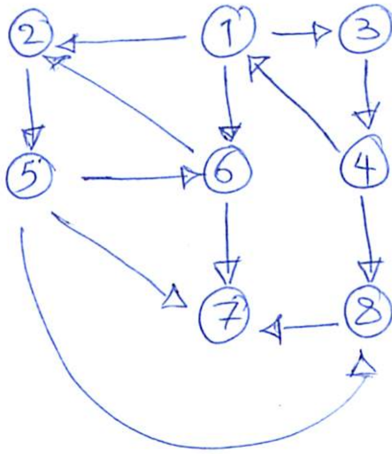
- Parent: direct ancestor (immediate)
- Child: 〃 descendant (〃)

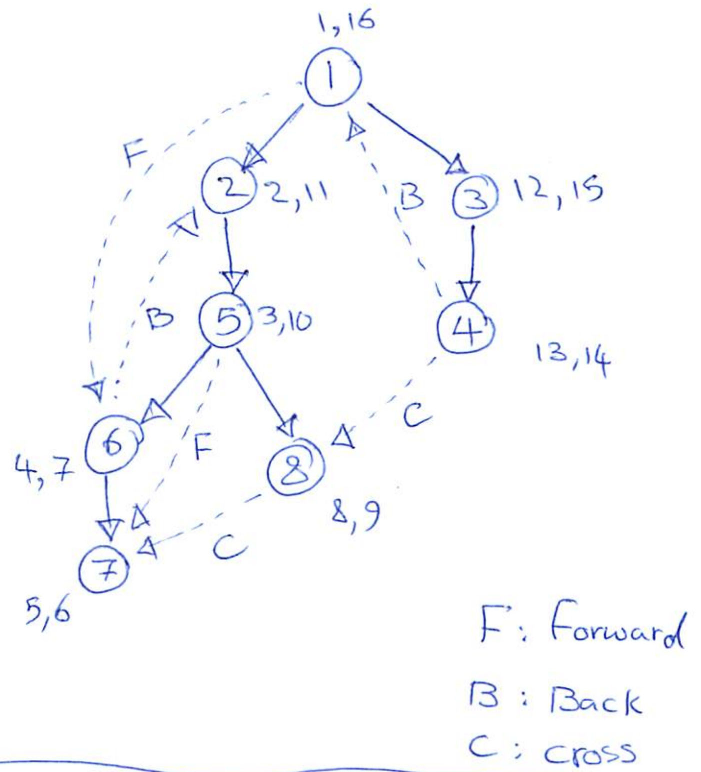
{ A is parent of B; B is a child of A }

- Source: if a node has only outgoing edges is called source
- Sink: 〃 〃 〃 〃 〃 incoming 〃 〃 〃 Sink

- Types of edges in DFS forest:
  - Tree edges (u,v) ⟶ [[ ]] with labels u v v u
  - Forward 〃 ⟶
  - Back 〃 ⟶ [[ ]] with labels v u u v
  - Cross 〃 ⟶ [ ] [ ] with labels v v u u

Example:





F: Forward

B: Back

C: cross

Q: why this pattern [[ ]] is not possible?
u v u v

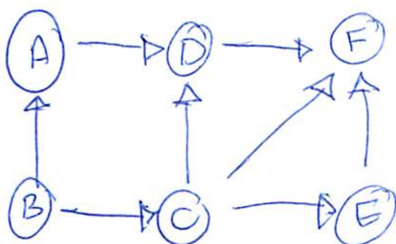
Q: What we can infer from [ ] [ ] ?
u u v v

we infer that v is not accesible from u but the reverse is unknown.

Property 2: A directed graph has a cycle iff its DFS forest has a back edge. Proof? (both directions of proof)

Directed Acyclic Graph (DAG) : good for modeling dependencies.

We are interested in an ordering of nodes, that dependencies constraints are satisfied ⤳ Topological Sort



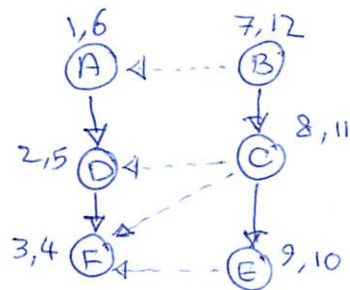
One order: B, C, A, D, E, F

not unique

B, A, C, D, E, F
B, C, E, A, D, F

One simple algorithm (in linear time) is to run DFS on DAG and keep nodes in a decreasing order of their post value.

Note: to get a linear-time complexity you should not sort them after running DFS, because we know sorting is $\Omega(n \log n)$. figure out how?



post values: 12, 11, 10, 6, 5, 4
order : B, C, E, A, D, F

Why this algorithm is correct and give us a true order?

- Because only Back edges has a pattern like $post(u) < post(v)$ for (u,v) as an edge. Since it's a DAG $\Rightarrow$ no cycle $\xrightarrow{\text{Property 2}}$ no Back edge ▨

Since we can linearize a DAG, always exist a highest value for post (that's the source) and a lowest value (that's the sink).

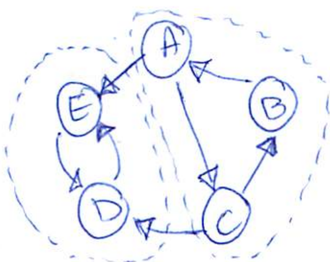
Alternative algorithm for topological Sort:
- Find a source, output it & delete it
- Repeat until the graph is empty

Strongly Connected Components: (SCC)

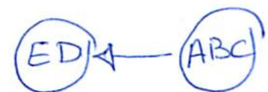
Relation: $\forall u, v \in V$, they are connected iff $\exists$ a path from u to v and from v to u $(u R v)$

This relation partition vertices into disjoint sets, called SCC.

Ex.



meta-graph:



It's always a DAG
why?

That's a very nice property that we have two-tiered Connectivity structure for directed graphs. It's very helpful to analyze the graph, for example, when you are analyzing web pages graph.

An efficient algorithm to find SCCs:

Observation: If we call explore( ) on a node that lies in a sink SCC, then that component will be retrieved

Q1: How to find a node that is inside a sink SCC? Note that that node is not a sink itself, the Containing SCC in meta-graph is a sink.

Q2: How to Continue when first SCC is discovered?

To answer Q1: there is no easy way, but we have the following property:

Property 3: If C and C' are two SCCs and there is an edge from C to C' (in meta-graph), then highest post value in C is bigger than highest post value in C'.

Proof,

If C is visited first by DFS, then first node in C will have highest value among all nodes in C and C'.
If C' is visited first, it's clear.

Note: - A source in G, is a sink in G^R (reverse graph)
- G^R has exactly the same SCCs as G

From property 3, it follows that the node with highest post value is inside a source SCC.

Algorithm:

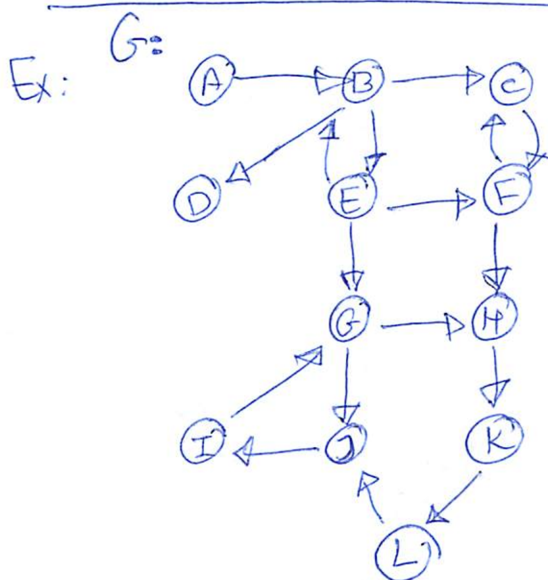
1) Run depth-first search on G^R, get a decreasing order of post values (highest is inside a source SCC in G^R, that is a sink in G).

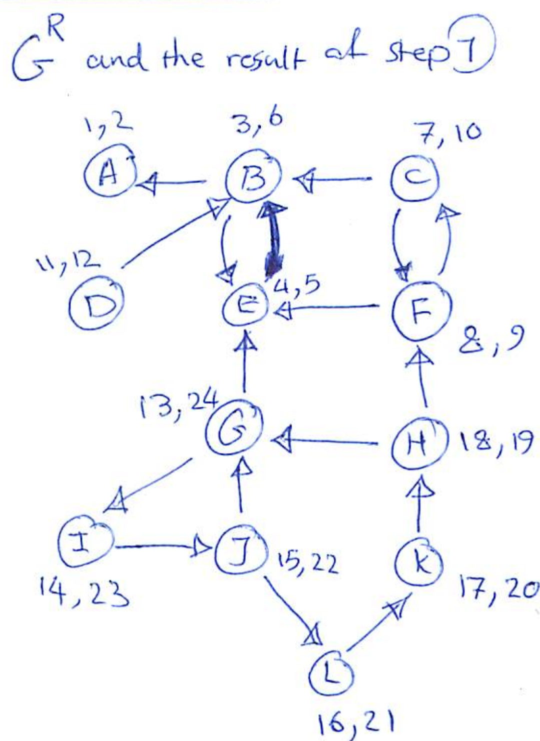
2) Run DFS on G based on this order:

- you have a counter that shows component's number, initial value is zero.

- plus one this counter on each explore() call

Ex:





Order based on post: G, I, J, L, K, H, D, C, F, B, E, A

SCC-num[], from Step2: {1 1 1 1 1 1}{2}{3, 3}{4, 4}{5}

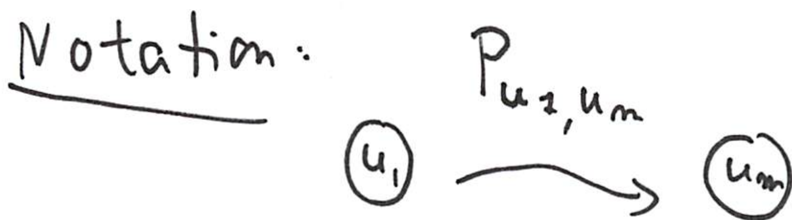
Chapter 4 [textbook]

Paths in Graphs

Let $u_1, \ldots, u_m$ be vertices (all distinct) of a graph $G=(V, E)$ so that $(u_i, u_{i+1}) \in E$ $i=1,\ldots,m-1$

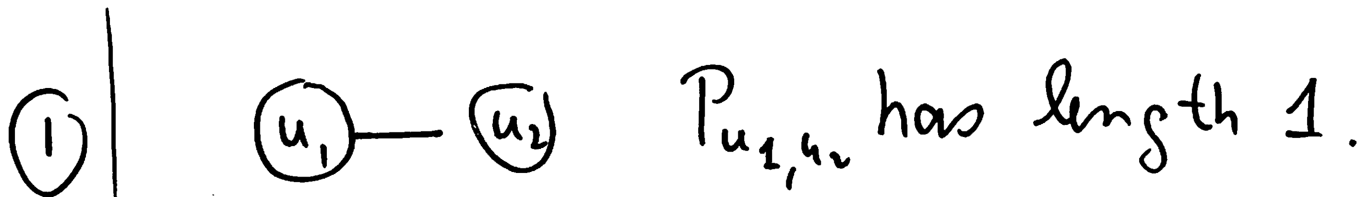
we say that there is a path P_{u_1,u_m} (or $P(u_1,u_m)$) from u_1 to u_m in G.

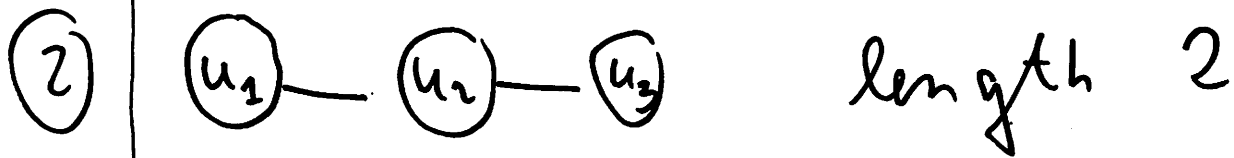
Notation:



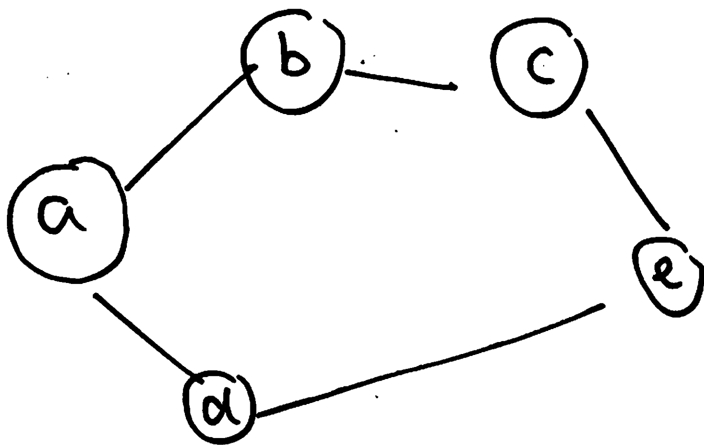
Def The Length of a Path is the number of its edges.

Example:





In general there are many paths between a pair of vertices in a graph.



One path is (a) — (b) — (c) — (e)

another path (a) — (d) — (e)

(a)–(b)–(c)–(e) has length: 3

(a)–(d)–(e) " " : 2

Def. The distance between two vertices in a graph is the length of the shortest path connecting them.

Notation. $|P_{a,b}|$ = length of Path $a \overset{P_{a,b}}{\rightsquigarrow} b$

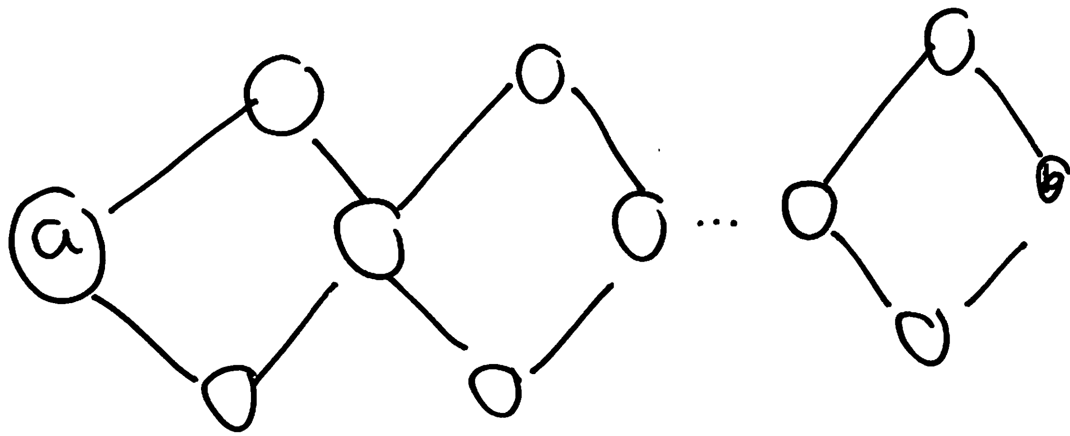
$d(a,b)$ = distance between a and b

$(= \min\{|P_{a,b}| \mid$ all paths $P_{a,b}$ in $G\}$

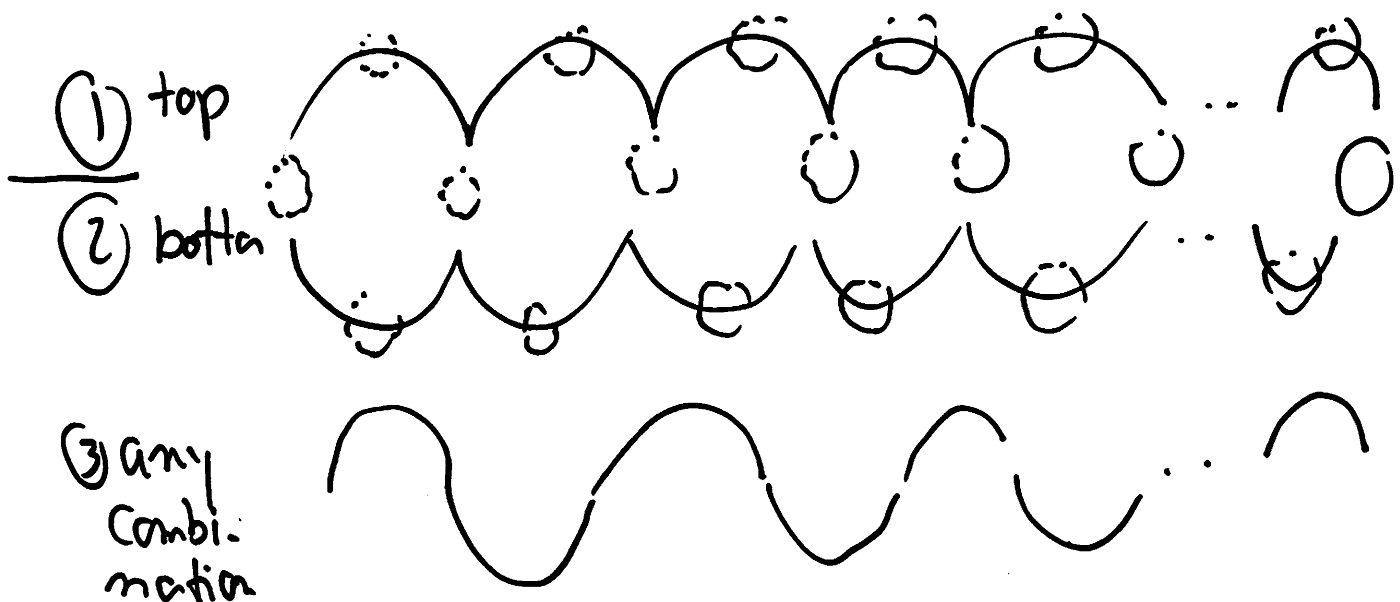
For the above: $|P_{a,b}| = 3$
where $P_{a,e}$ = (a)–(b)–(c)–(e)
but distance $d(a,e) = 2$
because the shortest path $P_{a,e}$ has

length = 2, (a)-(d)-(e)

How many path could there be, in a graph G, connect two vertices a and b?



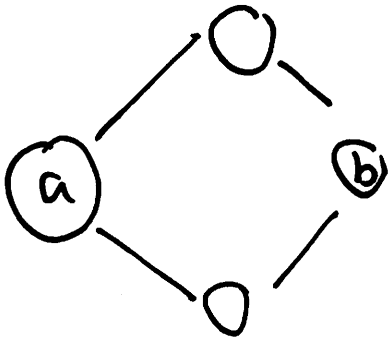
Exponentially many!



How many exactly?

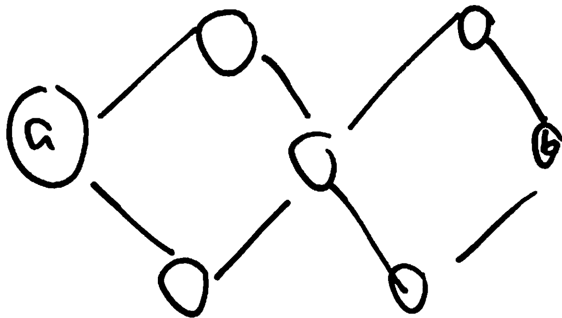
assume # of vertices $n = 1 + k \cdot 3$

Let $k = 1$ then $n = 4$



: 2 paths (top, bottom)

(for $k = 2$ $n = 7$



4 paths

top top

top bottom

bottom top

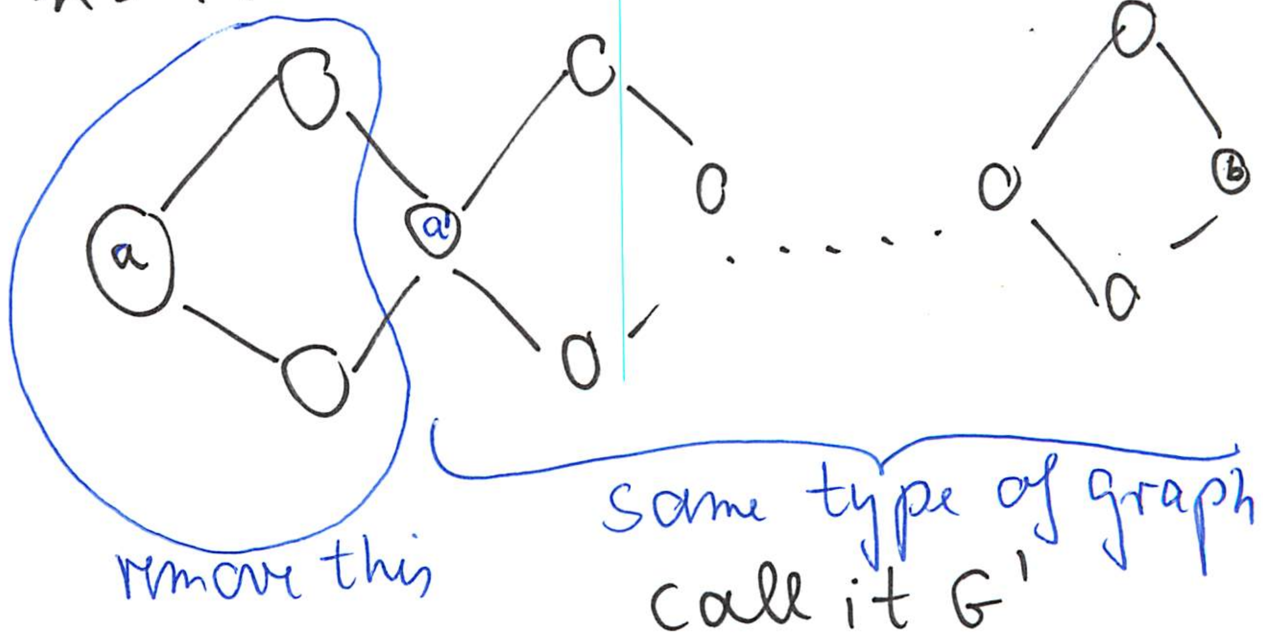
bottom bottom

Claim: Let $n = 1 + k \cdot 3$ be the number of vertices in this type of graph. Then there are 2^k different path from a to b.

Proof: $k = 1$ we got $2^1 = 2$ paths

Assume that for $k-1$ we get 2^{k-1} different path.

Consider now the graph G with $n = 1 + k \cdot 3$ vertices.



Any path from a to b in G must go through a'.

Now by induction, there are 2^{k-1} path in G' from a' to b.

There are 2 paths from a to a' in G. Any of these 2 paths can be concatenated with a path from a' ⇝ b in G' but also in G.

$a \rightsquigarrow a' \longrightarrow b$

these are paths from a ⇝ b in G.

So, the number of path in G from a to b $= 2 \cdot 2^{k-1} = 2^k$ ∎

So, there are graphs with an exponential number of paths between pairs of vertices.

So how to find the distance between 2 vertices?

Algorithm "STOPID" or "BRUTE FORCE"

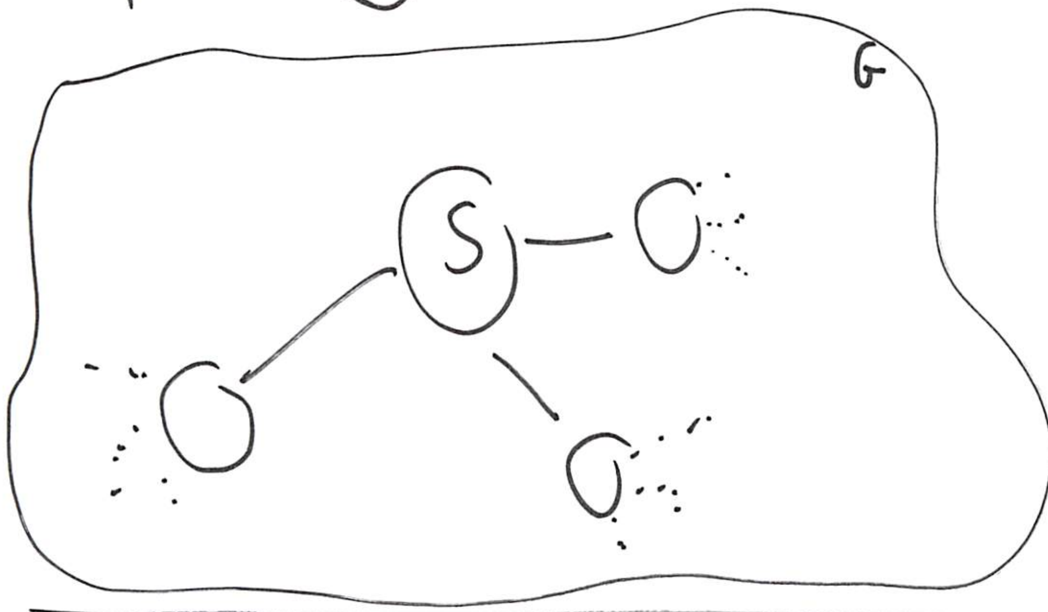
1. Generate all paths
2. Determine their lengths and compute the minimum

Time Complexity: exponential

can we do better?

Consider a vertex, s, called source.

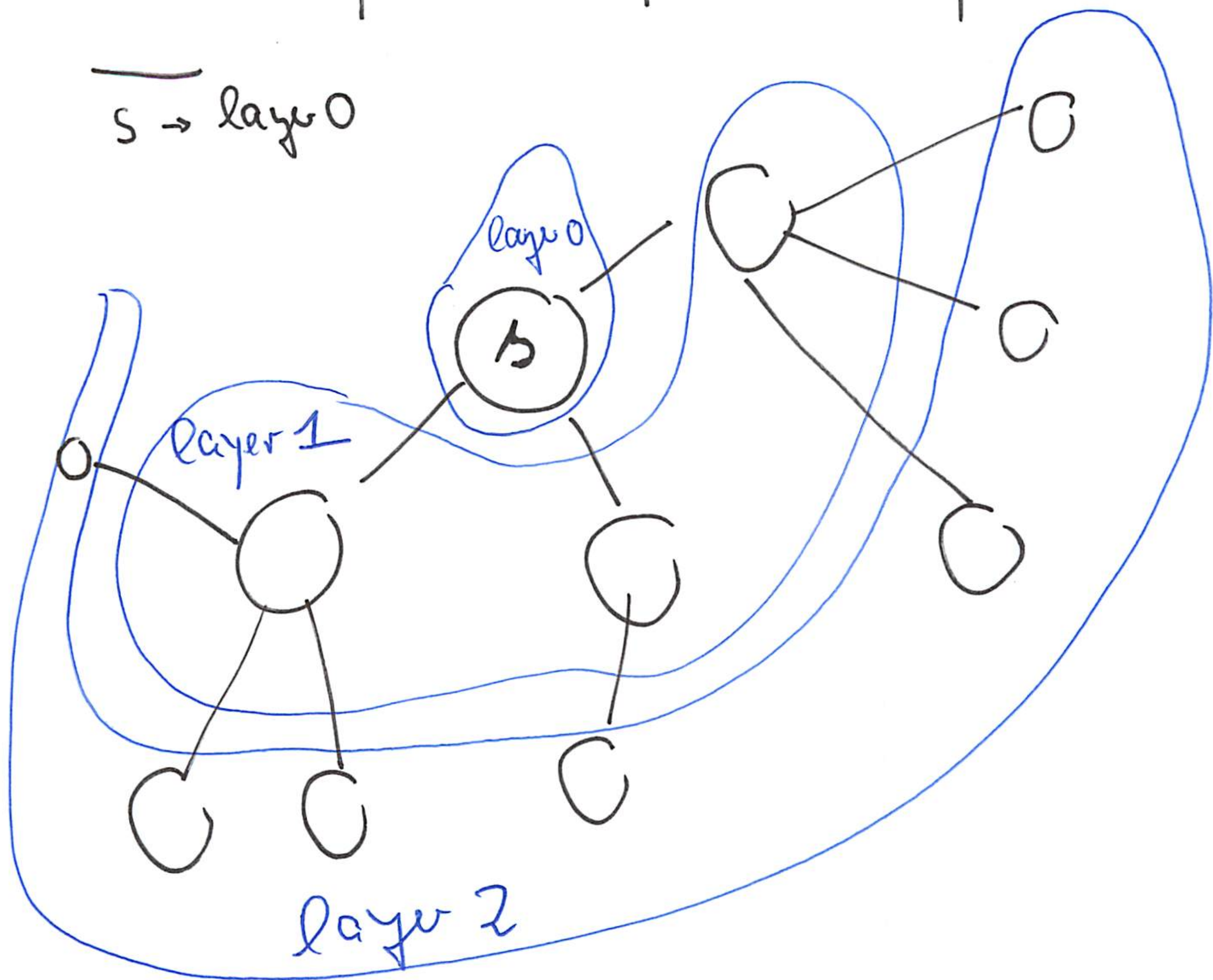
some vertices are at distance 1 from (s).



let us call these vertices layer 1 vertices.

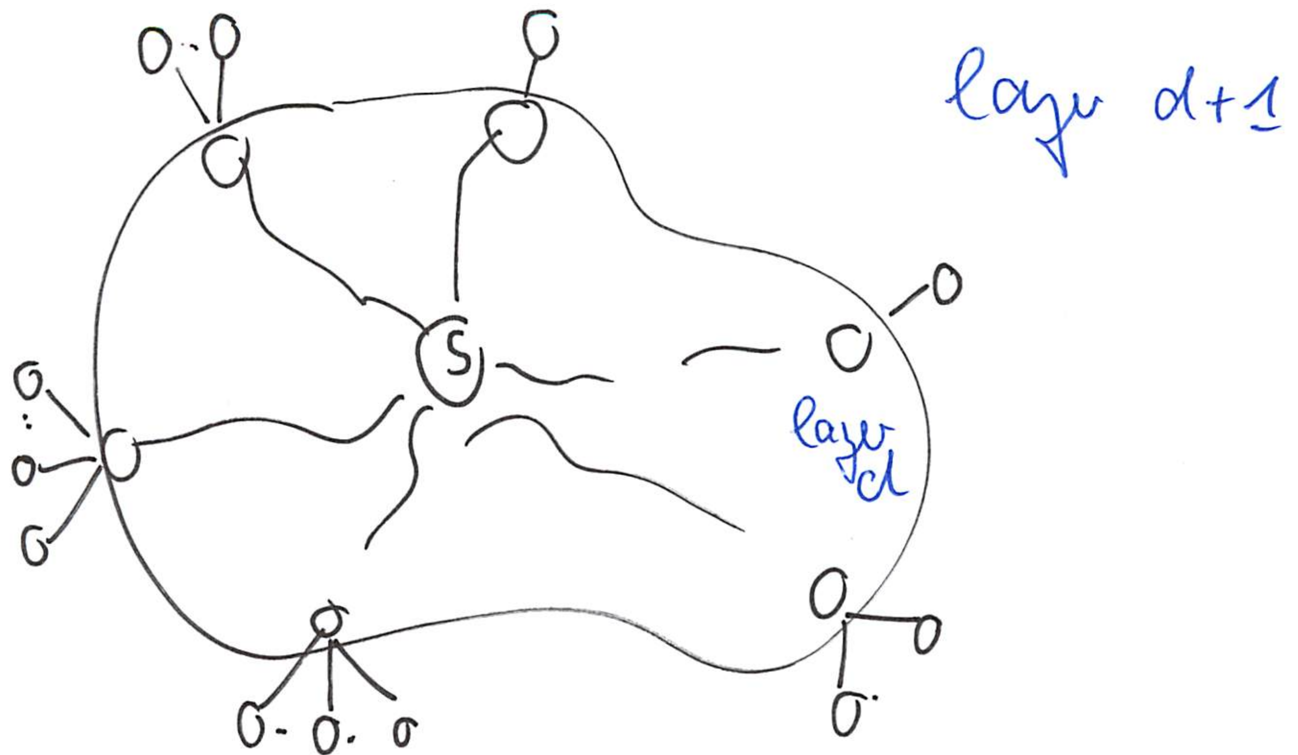
What are the layer 2 vertices?

All those "new" vertices that can be reached via an edge (just one not a path) from layer 1 vertices



We say that 2 vertices are adjacent if there is an edge connecting them.

in general, layer d+1 are all vertices that are adjacent to layer d vertices (but not already on layer 0, 1, ..., d)



Note: A vertex v is at distance d from s if and only if it is on layer d for vertex s.

So to compute the distance from s to a vertex t, we can determine the layer on which t appears.

Note: We get more information than just the distance $d(s,t)$.

Procedure bfs (G, s)

bfs: breadth first search

Input: Graph G = (V, E) directed or undirected; vertex $s \in V$

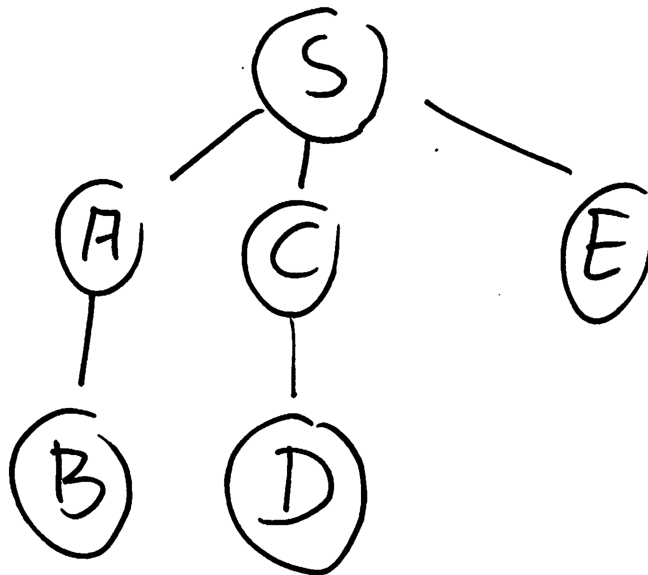
Output: For all vertices u reachable from s, dist(u) is set to the distance from s to u, i.e., d(s, u).

for all $u \in V$
  dist(u) = "∞"

dist(s) = 0
Q = [s]   (queue containing just s)

while Q is not empty
  u = eject(Q)
  for all edges $(u, v) \in E$
    if dist(v) = ∞
      inject(Q, v)
      dist(v) = dist(u) + 1

Example



Order of Visitation	Layer dist( )	Queue contents after processing vertex
		[S]
S	0	[A C E]
A	1	[C E B]
C	1	[E B D]
E	1	[B D]
B	2	[D]
D	2	[ ]

Correctness

Inductive Hypothesis
for each $d = 0, 1, ..$ 1) all nodes at distance $\leq d$ have correct. distance value

2) all other node have distances ∞

3) the queue contains exactly the nodes at distance d (once all distance $d-1$ nodes have been processed)

The induction Hypothesis is easily turned into a proof by induction.

Why does this prove the correctness?

Complexity:

The overall running time of the algorithm is $O(|E| + |V|)$ for a graph $G = (V, E)$.

Argument: Each vertex is added exactly once to the Queue (if it is reachable from s). ⇒

There are $2|V|$ queue operations.

The inner loop executes for each edge a constant amount of operations whose cost is a constant.

We cannot improve in terms of "big oh".

dfs Depth-First Search and
bfs Breadth-First Search
are 2 alternate ways in
which to explore graphs.

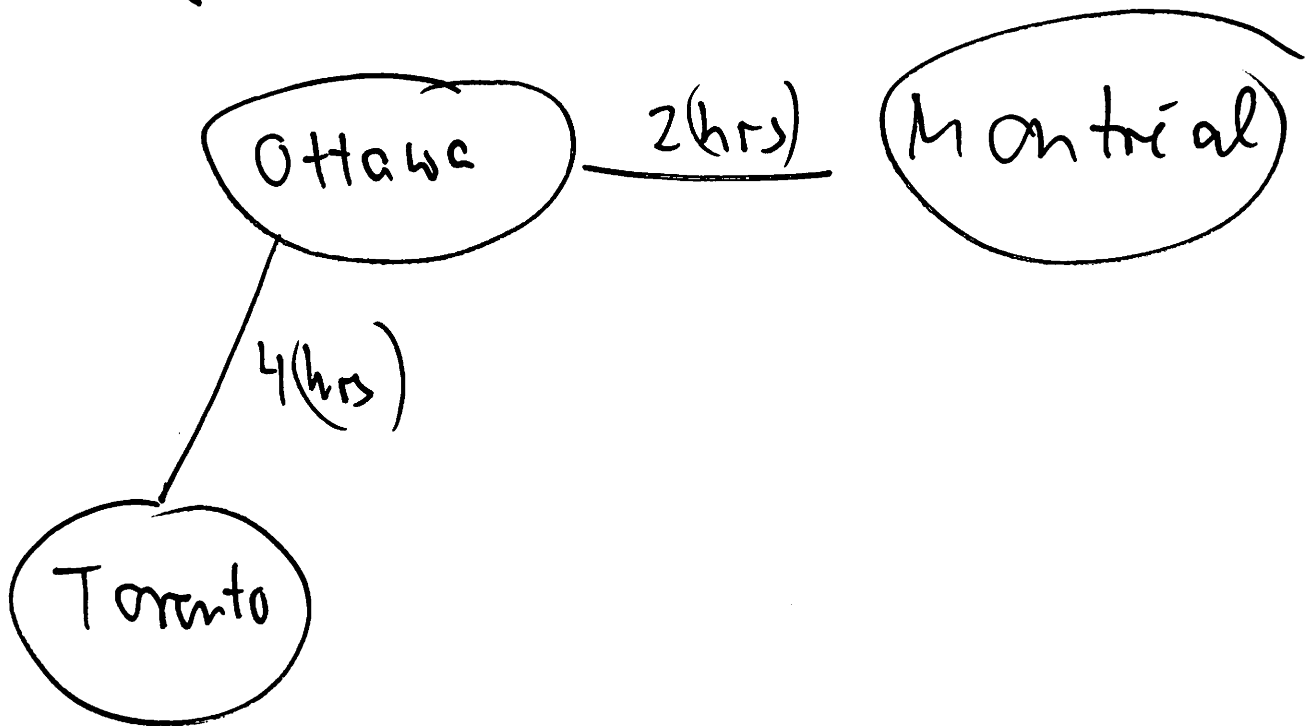
Find examples for each method.

e.g., Chess is suited for
which strategy?

Lengths / weights on edges

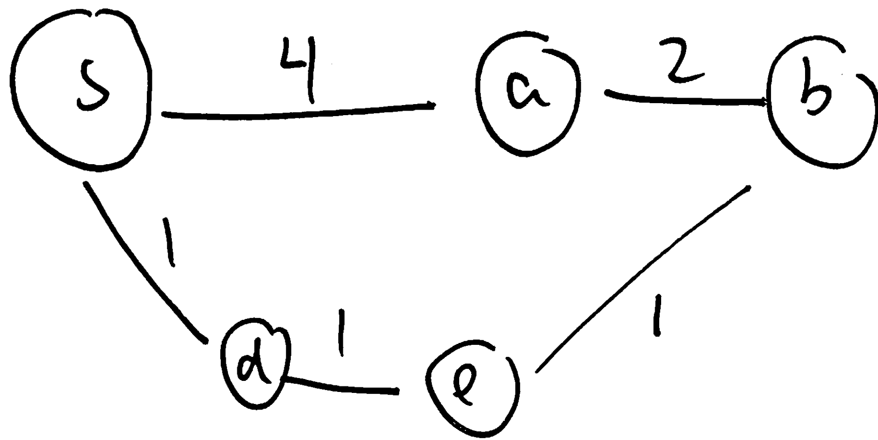
bfs considers all edges to have the same length or weight

If you drive though you have different lengths



Now the distance d(s, u) is still the min cost of all path from s to u.

But the cost of a path = sum of the weights of each edge on the path.



cost (s-a-b) = 6

cost (s-d-e-b) = 3

<u>distance</u> d(s, b) = 3

number of edges | s-d-e-b | = 3 | s-a-b | = 2

How to find the distance in this setting?

Dijkstra's algorithm (G, l, s)

Input: Graph $G = (V, E)$, $s \in V$
positive edge length $\{ l_e : e \in E \}$

Output: for all vertices u reachable from s;
distance, dist(u), is $d(s,u)$

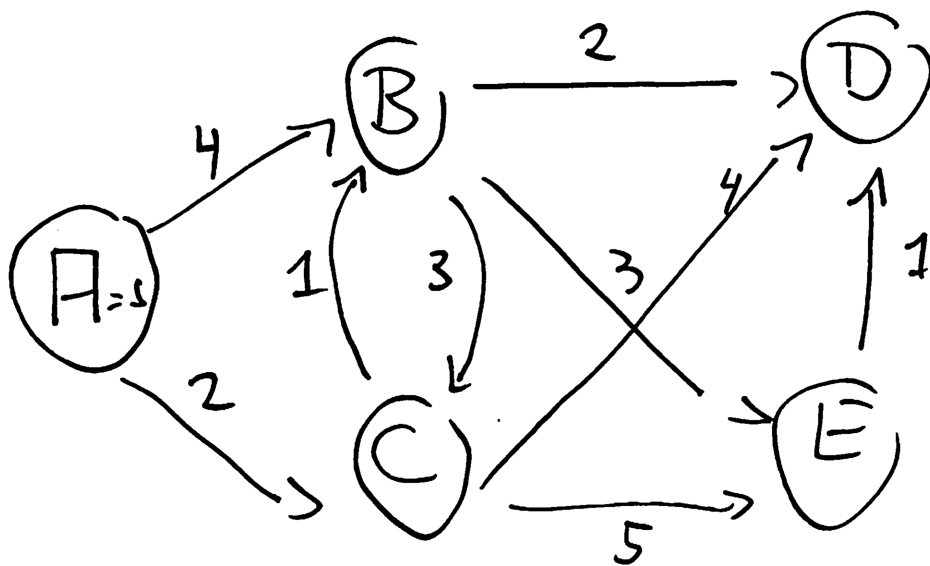
```
for all u ∈ V
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = makequeue(V)    (using dist-values as keys)
while H is not empty
    u = deletemin(H)
    for all edges (u,v) ∈ E
        if dist(v) > dist(u) + l(u,v)
            dist(v) = dist(u) + l(u,v)
            prev(v) = u
            decreasekey(H, v)
```

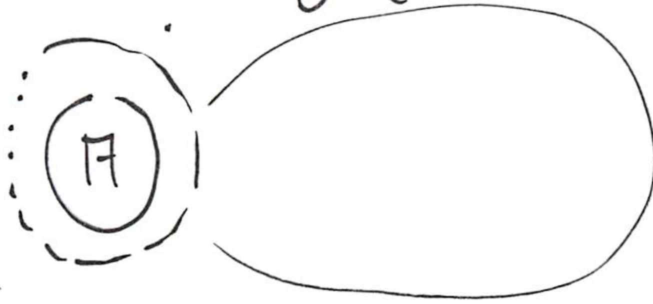
$dist(u) = \infty$ means no value found for u yet

when u is removed from H the value of dist (u) is d(s, u) and prev(u) is the node on the shortest path from s to u just before u.

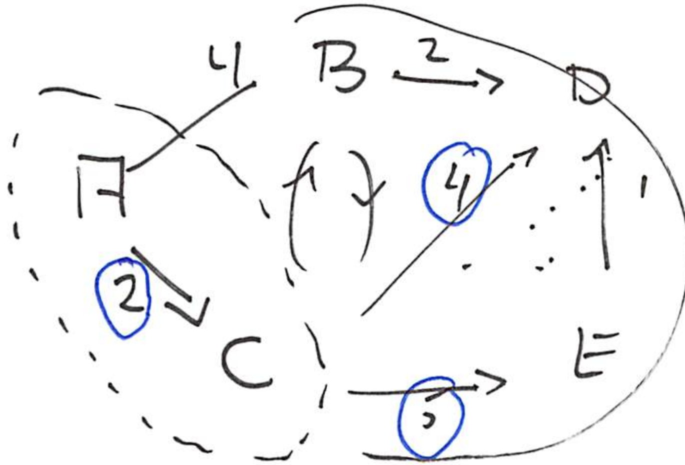
Example: (see book)



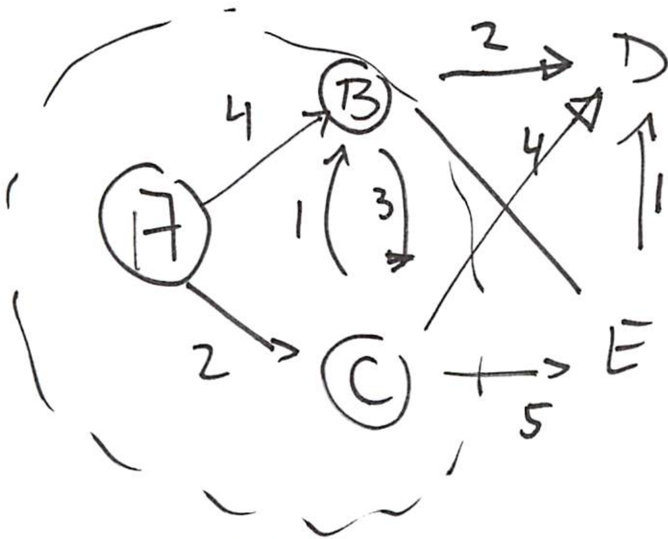
See textbook



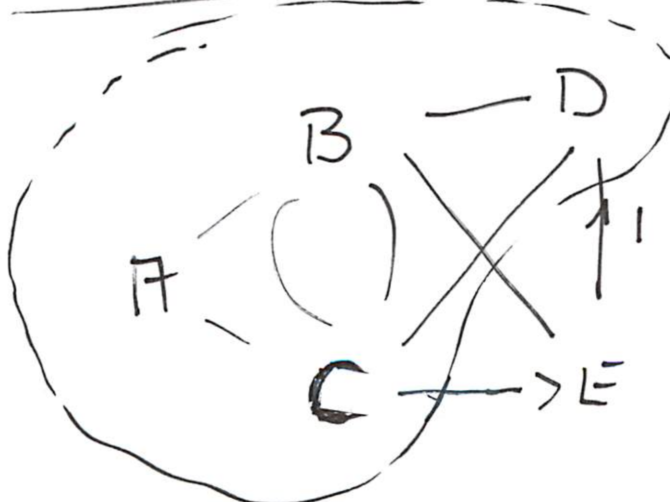
A : 0
B : 4
C : 2
D : ∞
E : ∞



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

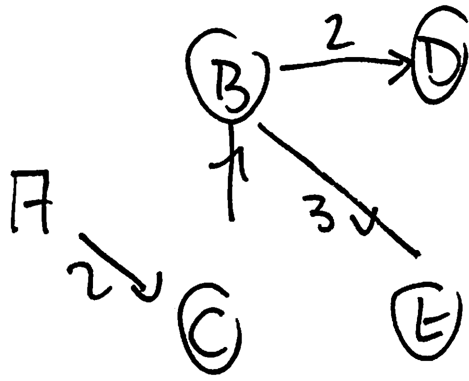


A : 0
B : 3
C : 2
D : 5
E : 6



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

4.22



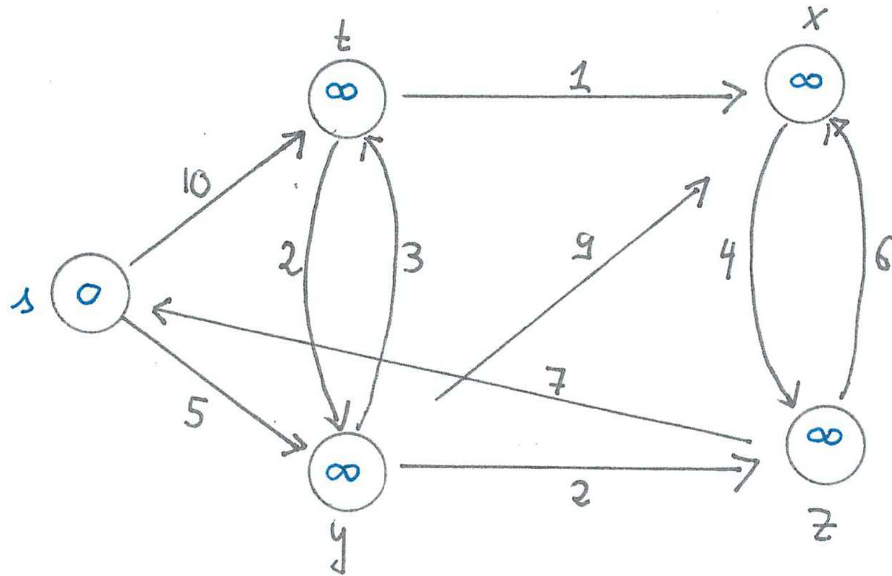
using the pointers

① We have seen an example (pp. 4.2 – 4.7) of a graph with an exponential, 2^k, $k = \theta(n)$, paths. This example was chosen so that it demonstrates that there can be an exponential number of shortest paths.

Think: how can Dijkstra's algorithm run in $O(|E| + |V| \lg |V|)$ time if there can be an exponential number of shortest paths?

② If we did not want shortest paths, but (simple) paths. Is there a very simple graph with many many paths? What is the graph(s) and how many paths are there?

D.O



Initialization

all distances (except from s to s) are ∞

$dist(u) = \infty$ for all $u \neq s$

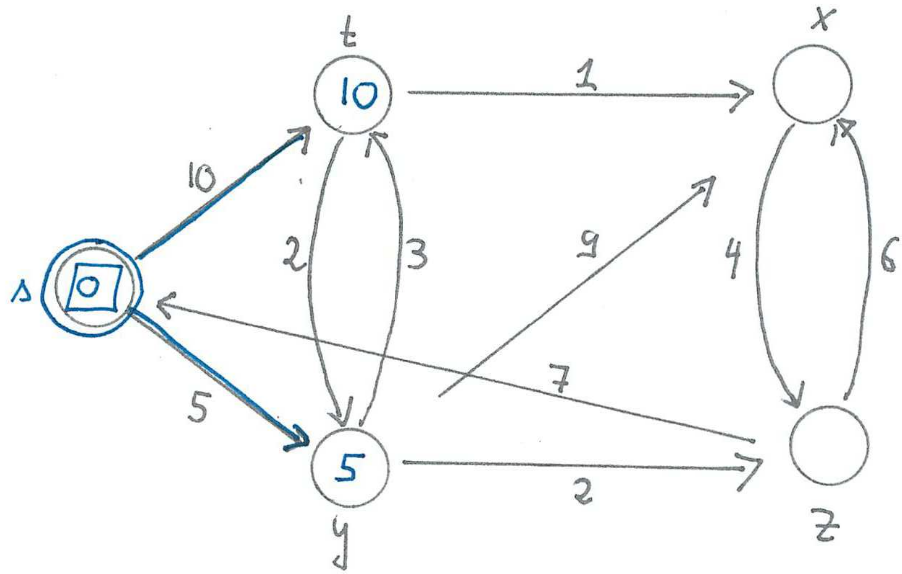
$dist(s) = 0$

$prev(u) = nil$ for all $u \neq s$

D.1

u = s

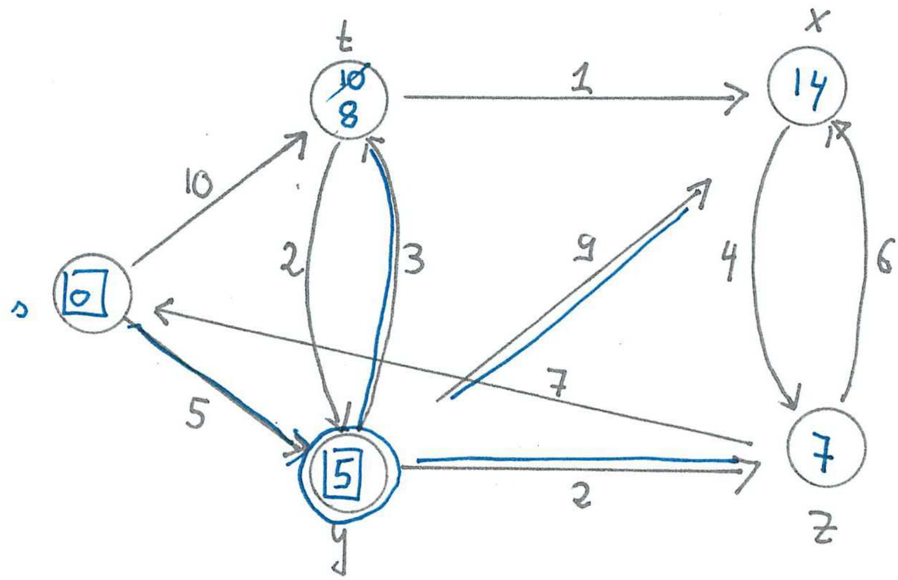
dist( )		Prev
t	10	s
x	∞	nil
y	5	s
z	∞	nil



d(s) = 0 has been found

$u = y$

	dist()	prev
t	8	y
x	14	y
z	7	y

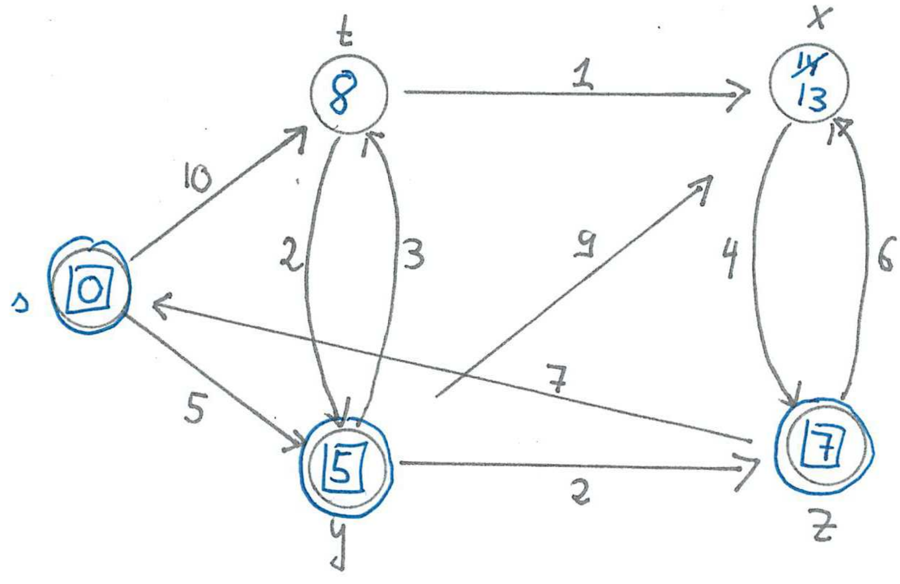


$d(y) = 5$ has been found

D.3

u = z

	dist()	prev
t	8	y
x	13	z

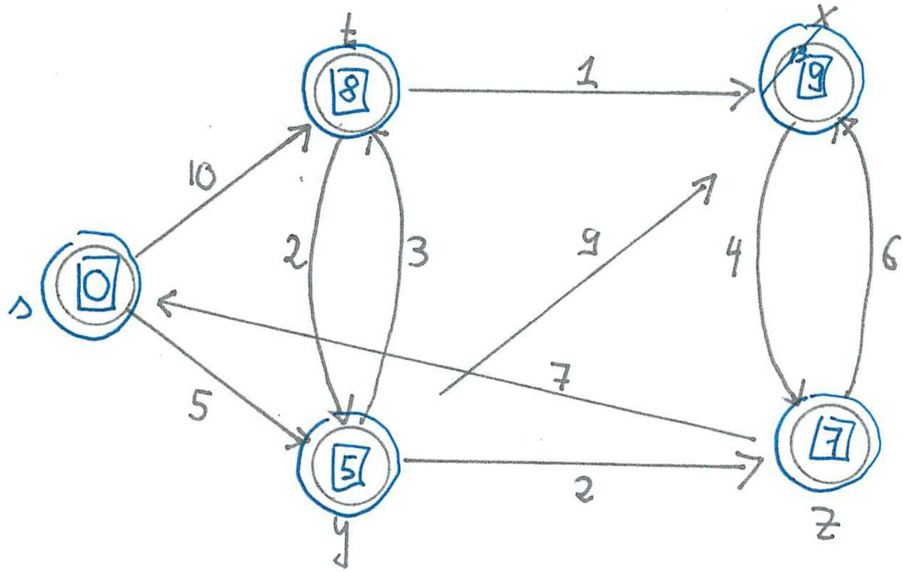


d(z) = 7

D.4

$v = t$

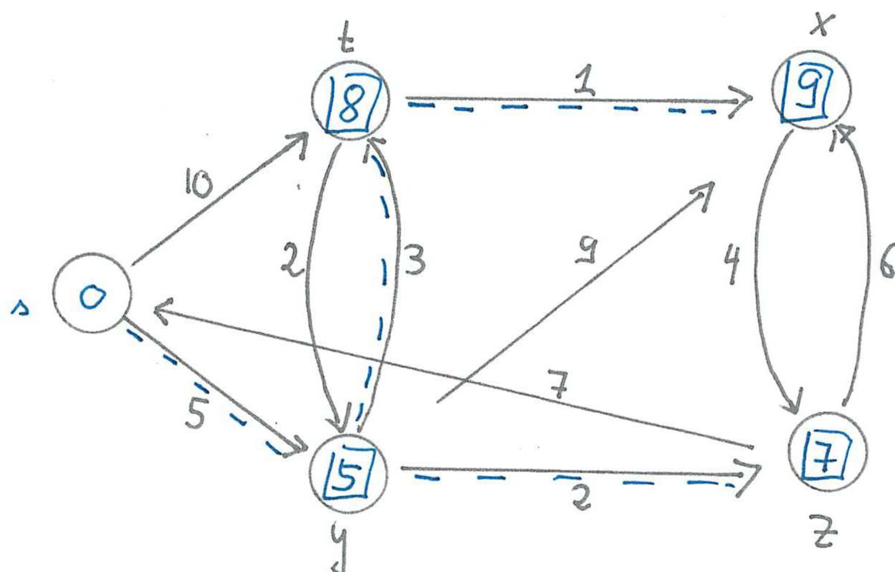
x	dist	prev
x	9	t



$d(t) = 8$

D.5

$U = X$



- - - : previous relationship

$d(x) = 9$

D. 6

Theorem Given a weighted directed Graph $G = (V, E)$ with non-negative weights on each edge of G, source s, the algorithm, Dijkstra, terminates with $d(u)$ = distance from s to u for all $u \in V$.

Correctness

Loop invariant: at the start of each iteration of the while loop for each vertex $v \in V - H =: S$ (i.e., all vertices no longer in the queue) the correct distance from s to v has been found.

Initialization:

initially all vertices are in H, thus $S = V - H = \emptyset$. The statement is therefore trivially true.

Termination:

$H = \emptyset$, thus if we can show the invariant to hold. The algorithm is correct

Maintenance:

We want to show that when u, the vertex with minimium key value is added, the invariant remains true.

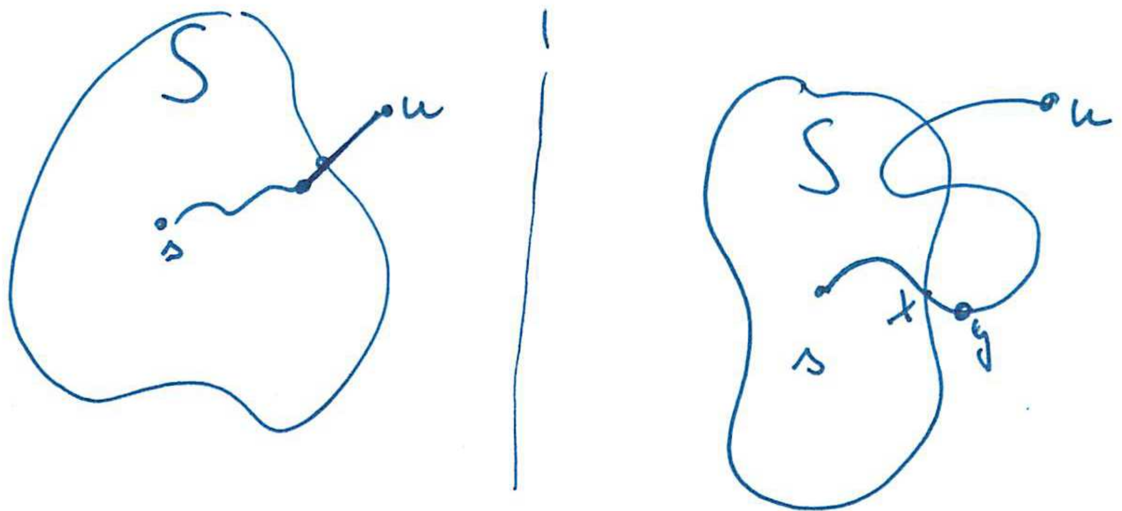
Proof by contradiction.

Assume that u is the first vertex for which $d(u)$ is incorrectly computed.

Consider the moment u is taken of the queue. $u \neq s$ as $d(s) = 0$ is correctly determined.

So, $S \neq \emptyset$, when u is added.

There is a path from s to u. Which means there is also a shortest path from s to u.



Let y be the first vertex on that path outside S and x the predecessor of y.
(Note the part outside S before u could be empty.)
Also $x = s$ is possible

D.G

By loop invariant, $d(x)$ is correct. The edge (x,y) was used to give an upper bound on $d(y)$, i.e. $dist(y) = d(x) + cost(x,y)$

$$(cost(x,y) = \ell(x,y))$$

$$dist(y) = d(s,y) \leq d(s,u) \leq dist(u)$$

But, both u and y were in H when u was chosen

$$dist(u) \leq dist(y)$$

thus $dist(y) = dist(u)$

and $dist(y) = d(s,y) = d(s,u) = dist(u)$

This is a contradiction to our assumption.

$\Rightarrow$ the loop invariant is correct.

D.10

We can also show the correctness of the predecessor trivially.

<u>Corollary</u>: The algorithm also computes the shortest path tree with roots by using the predecessor relation.

<u>Analysis</u>:

The important factor determining the run-time, is the time for executing the priority queue operations.

- insert well we create the entire queue
- ~~eject~~ deletemin
- decrease key

- There are $|V|$ deletemin operations
- we create the entire queue
- decreasekey is often done by removing the element and reinserting it.

  There are $|V|+|E|$ such operations

If we use a binary (min) heap for n elements

Create takes $O(n)$

insert " $O(\log n)$

deletemin " $O(\log n)$

Thus, Dijkstra's algorithm runs in $O((|V|+|E|)\log(|V|))$ time using a binary heap.

Using a more complex priority queue organization known as Fibonacci heap, the time is reduced to

$$O(|E| + |V| \log |V|)$$

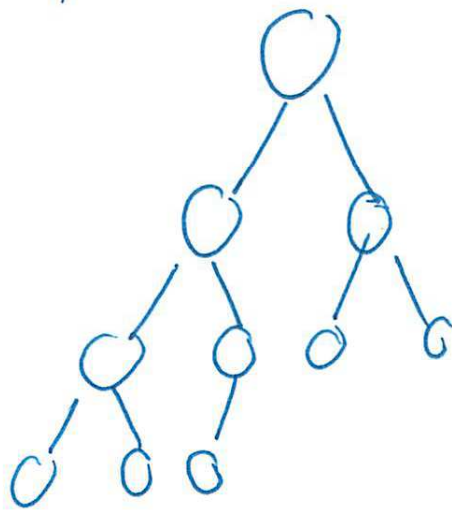
[insert + decrease key are $O(1)$ amortised]

What is the time complexity of Dijkstra's algorithm using an array implementation for the PQ.

Heaps (binary)

① Elements are stored in a complete binary tree, *(Structure)*

i.e. each level is filled from left to right and must be full before the next level is started. Except for the last level all levels are thus full. The last level may not be

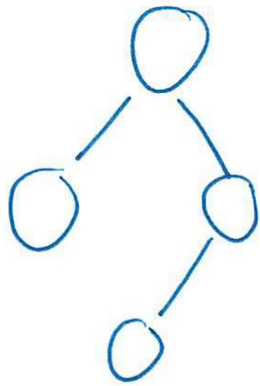


$n = 10$

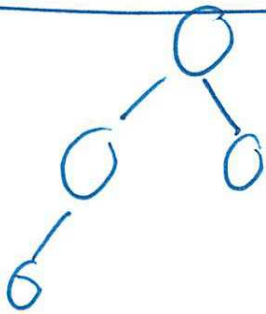
② Order

The key value of a parent is ≤ that of its children.

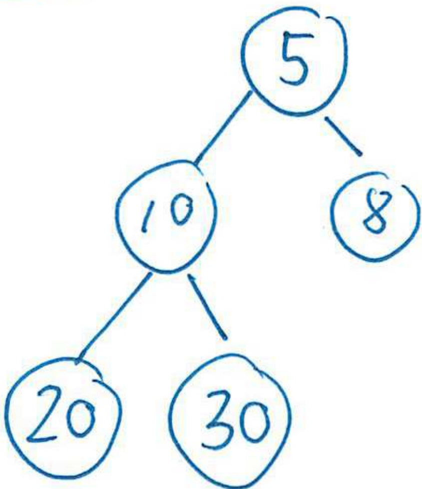
Ex:



– not a heap violate the structure

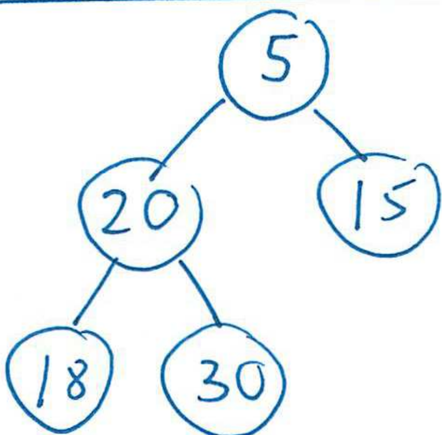


✓ correct structure



✓ correct structure

✓ correct order



✓ correct structure

– incorrect order

20 > 18

Fact: The global minimum is at the root.

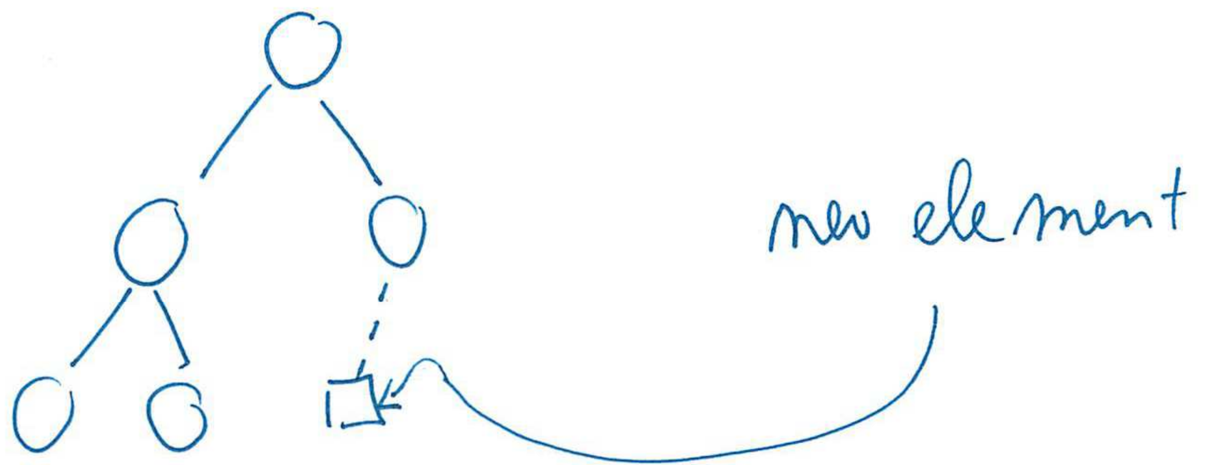
Question: • Where is the maximum?

• How many comparison are required to find the maximum in a min-ordered heap?

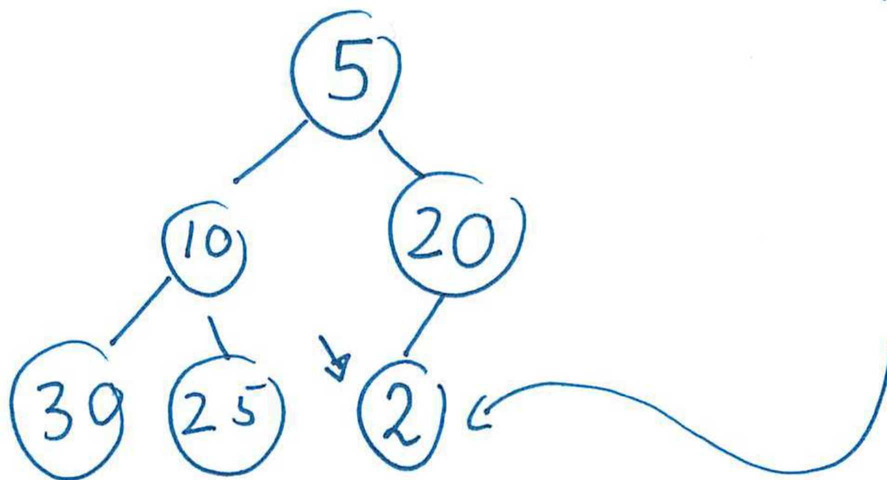
see Class

Insertion:

a new element is inserted into a heap by first getting the structure right then the order.



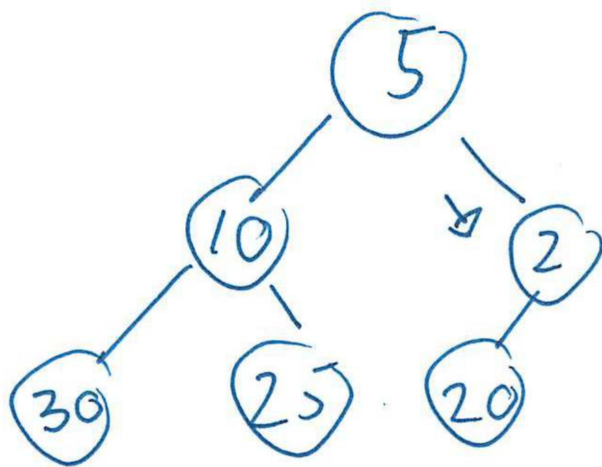
Then, to get the order right we bubble the element up.



heap order is violated.

⇒ compare element to parent
if smaller swap
and move one level up

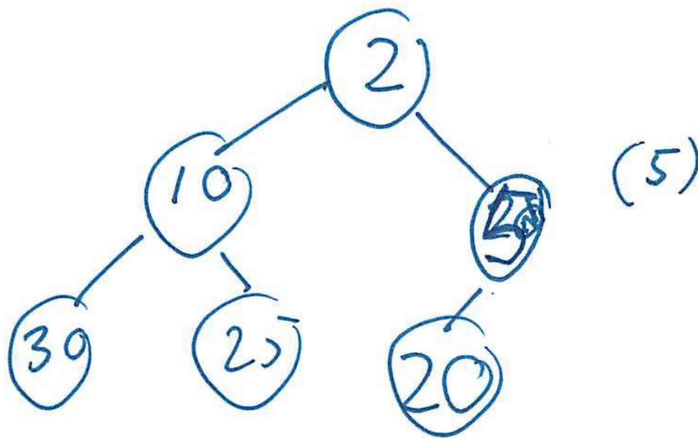
repeat



after swap

---

again, parent ⑤ > ② ⇒ swap



Why is it correct?

For the heap-order to hold every parent node key ≤ child node key

---

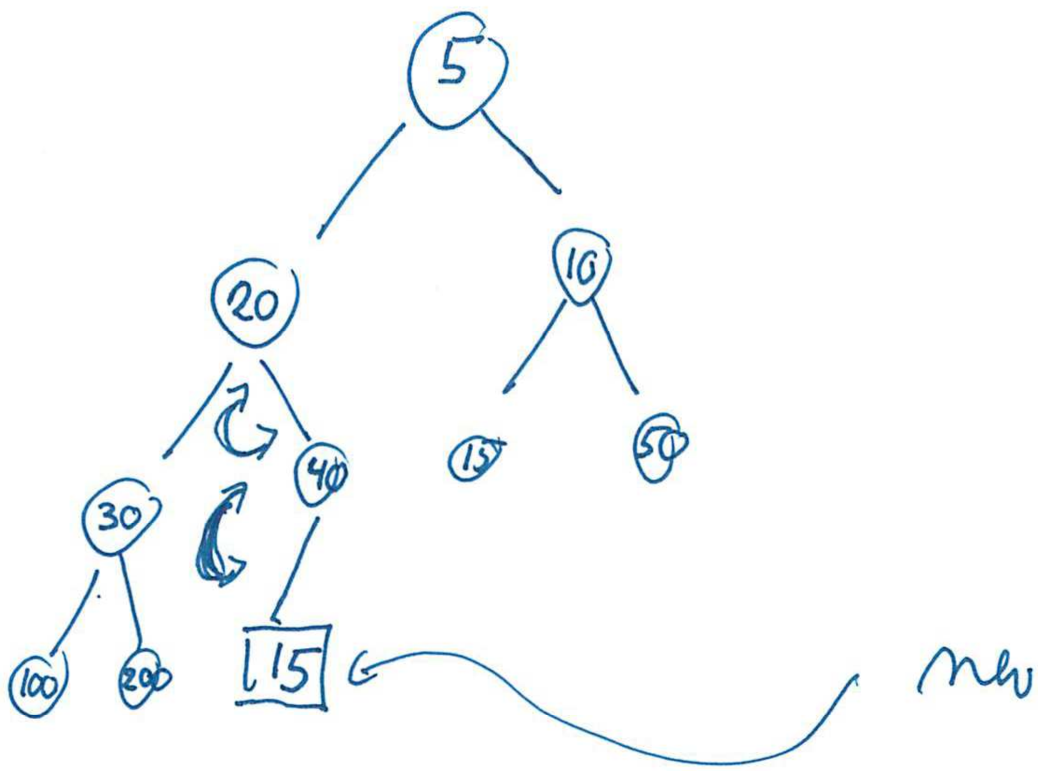
all nodes not on the path
from the new leaf to the root
remain unchanged.

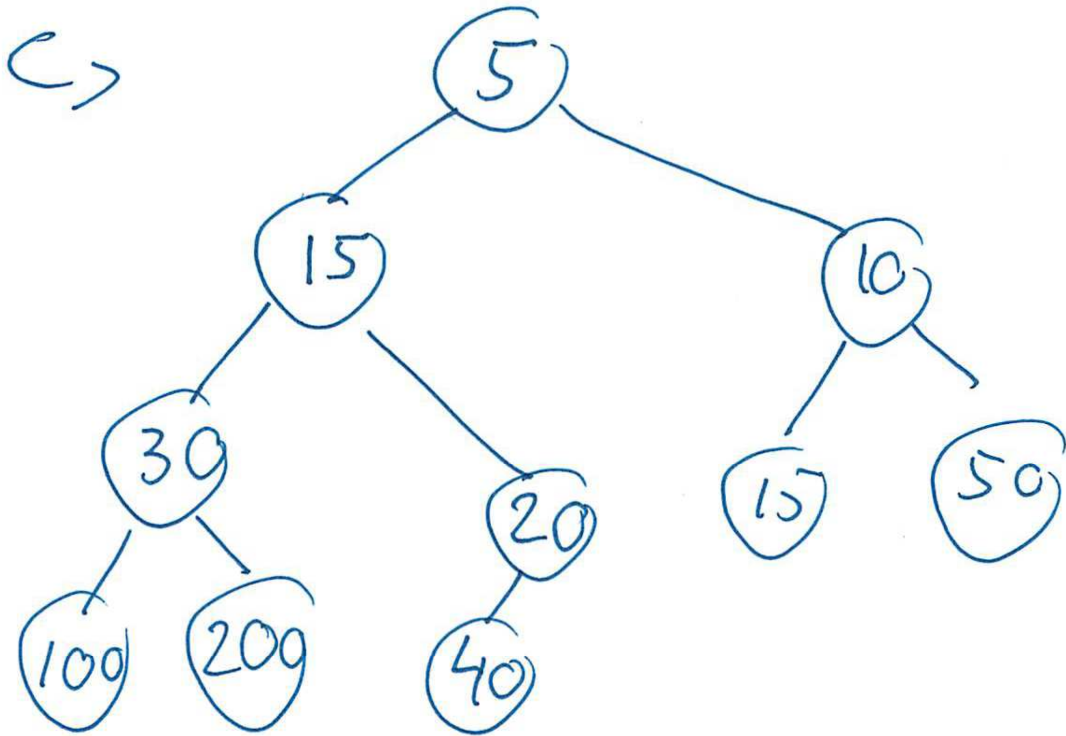
The new leaf is after possible
swap ≥ parent key.

Every time an element pair
is swapped it results in a
parent key that is < the original
so the parent is replaced by a
smaller key. The heap-order
to the other child is thus correct.

If a parent, child relation is
correct on the path no further
swaps are required

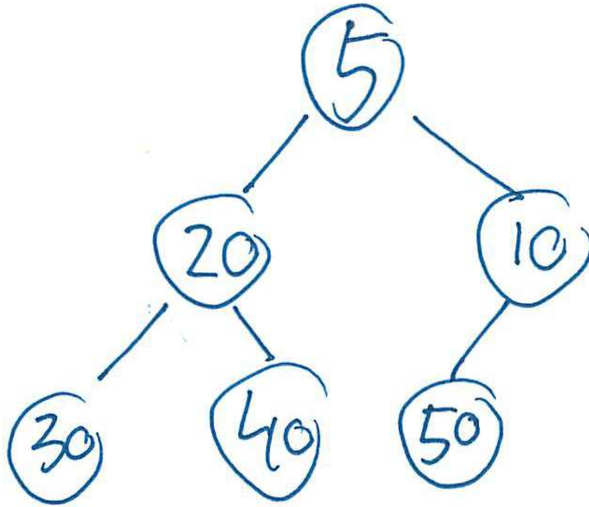
Time: $O(\log n)$: path length



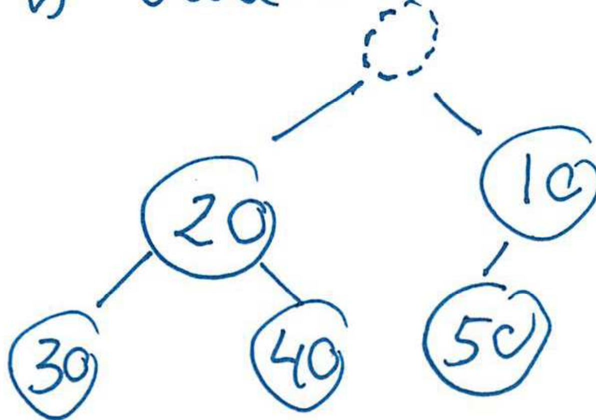


D.20

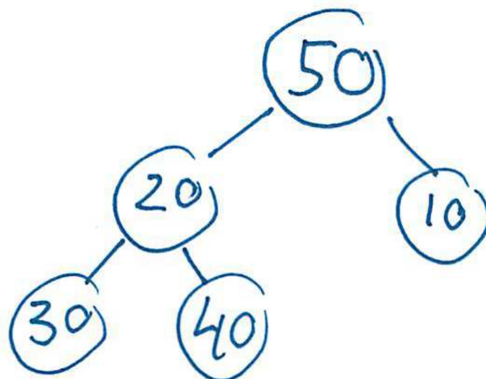
Delete Min



5 is deleted



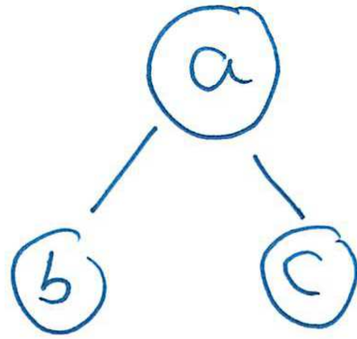
To get the structure right move (50) into root position



now get the heap order fixed

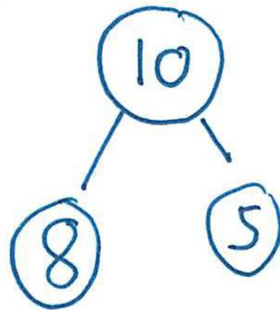
50 > 20
> 10

if $a > \min(b,c)$

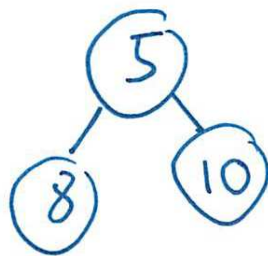


replace a by $\min(b,c)$

<u>Example</u>:



↳

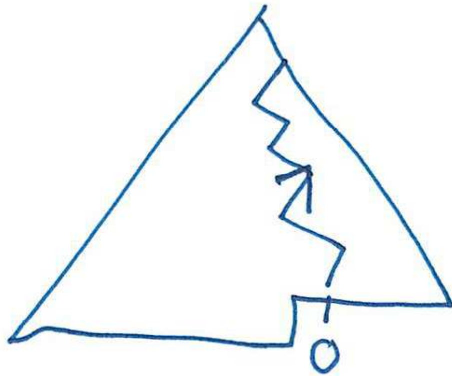


recurse to the side of the swapped element. Here to the right child.

constant work per level on a path to a leaf.

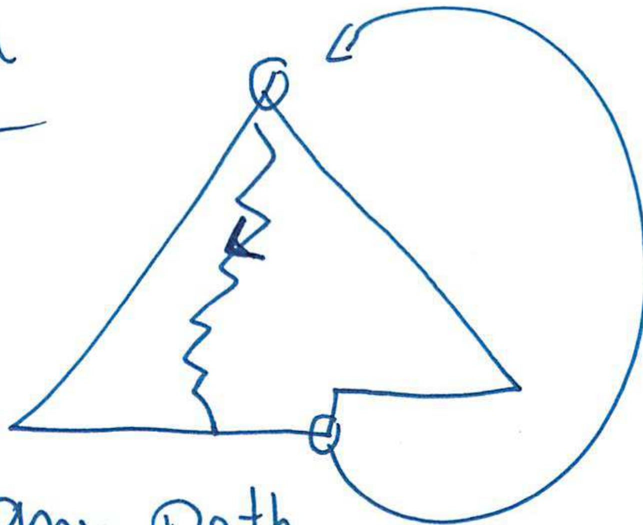
Notes: For a given n the structure of the heap is unique.

Insertion



The path is unique "bubble-up"

Delete



"trickle down"

any path is possible

How to create a heap on n elements:

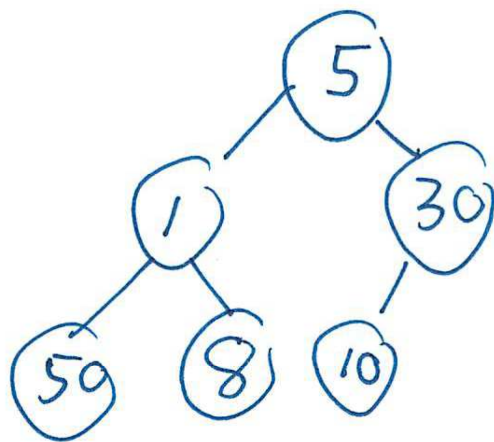
I. $n \times$ insert element

$\Rightarrow O(n \log n)$

II. an $O(n)$ method

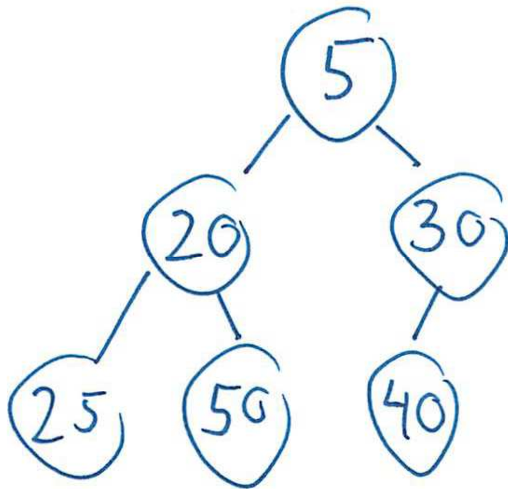
Take the n elements and put them into a heap-structured tree (possibly violating the order)

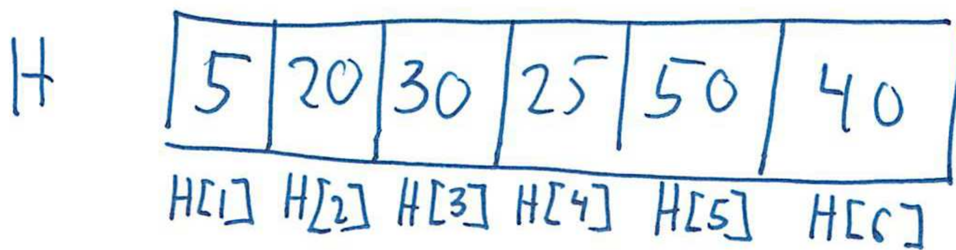
5, 1, 30, 50, 8, 10



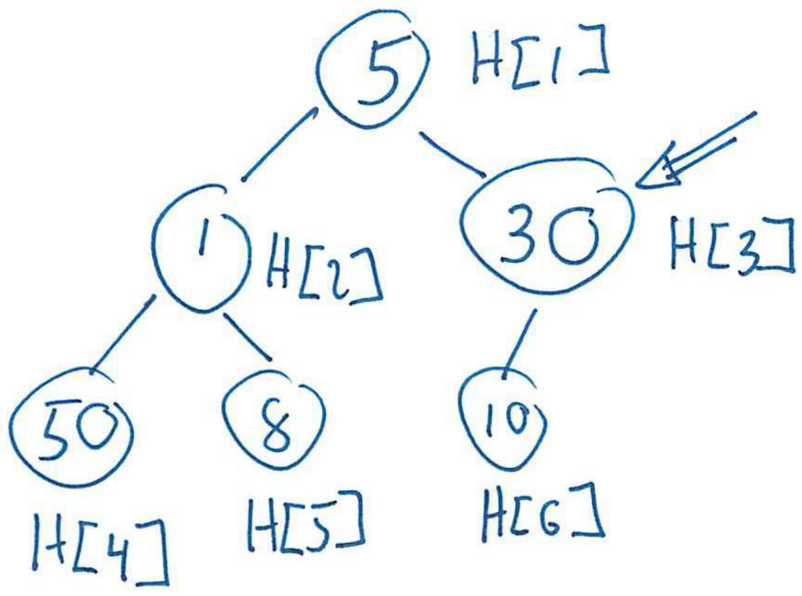
Now 10, 8, and 50 have no children so their heap-order is trivially correct.

heaps are most efficiently stored in an array.

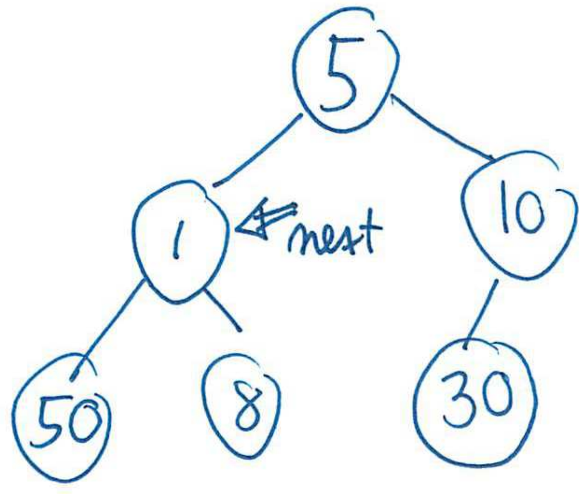




parent H[i] has children
H[2i], H[2i+1]
(if it exists)

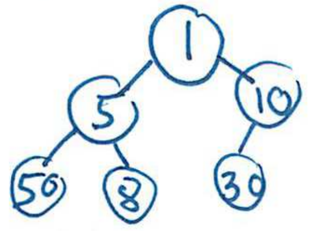


H[3] is rightmost non leaf in array.
Check if heap order is correct for H[3]
If not fix (like in deletemin operation)



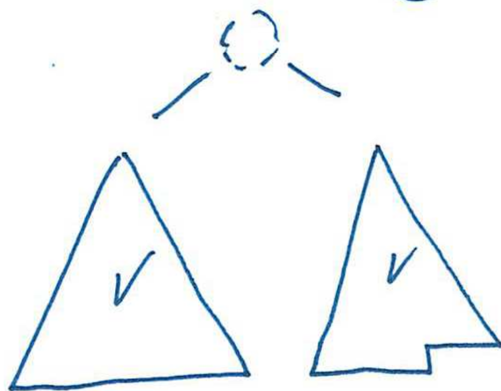
H[2] is correct

The H[1] fix it.



Correctness

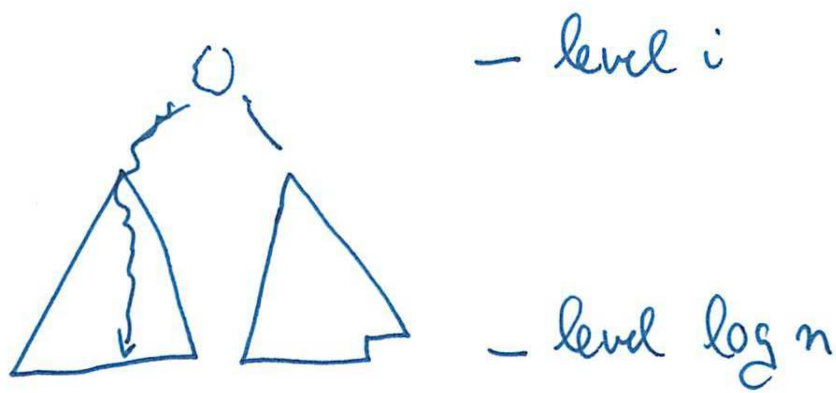
recursive argument



possibly incorrect
↳ gets heap-ordered.

Analysis

(bottom)
intuition: at the lower levels
we have many elements, but little work per element (height are small)
at the top levels (near root) we have few elements with a lot of work.



there are 2^i elements on level i
(except for possibly the last level)

each element may need to go to the bottom (trickle-down)
work $O(\log n - i)$

$$\sum_{i=0}^{\log n} 2^i \cdot (\log n - i) = \sum_{i=0}^{\log n} 2^{\log n - i} \cdot i$$

$$= \sum_{i=0}^{\log n} \frac{2^{\log n}}{2^i} \cdot i = \sum_{i=0}^{\log n} \frac{n}{2^i} \cdot i$$

$$= n \sum_{i=0}^{\log n} \frac{i}{2^i} \leq 2n$$

$\Rightarrow$ Heap construction takes $O(n)$ time.

Bellman-Ford Alg.

better description than in textbook

Bellman-Ford (G, w, s)

Input: Graph G, weight function w, source s
$G = (V, E)$ ↑ also known as "l"

output: If there is a negative cycle reachable from s the algorithm returns "false" i.e. no solution exists
Otherwise, the algorithm produces the shortest path distances from s to all reachable vertices

```
1  Initialize-Single-Source (G, s)
2  for i = 1 to |V| - 1 do
3       for each edge (u, v) ∈ E
4            do Relax (u, v, w)
5  for each edge (u, v) ∈ E
6       do if d[v] > d(u) + w(u, v)
7            then return "false"
8  return "true"
```

with

(A) Initialize-Single-Source(G, s)

for each vertex $v \in V$

do $d[v] = \infty$

$\pi[v] = nil$

$d[s] = 0$

(B) Relax(u, v, w)

if $d[v] > d[u] + w[u, v)$

then $d[v] = d[u] + w(u, v)$

$\pi[v] = u$

In many textbooks the Relax procedure is used.

The complexity of Bellman-Ford's algorithm is $O(|V| \cdot |E|)$.

```
dominates [ for each vertex do
              for each edge do
                  [constant work]
```

Correctness not covered here but easy.

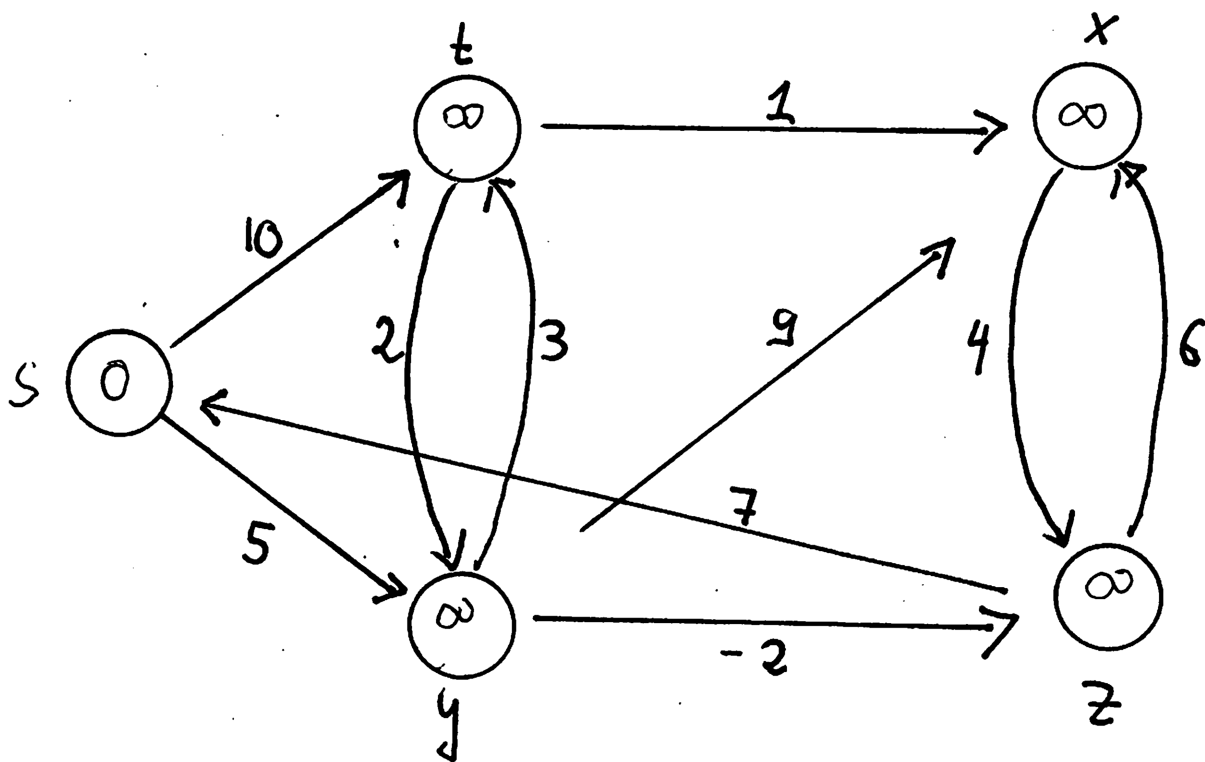
A path has at most $|V|-1$ vertices unless it has cycles.

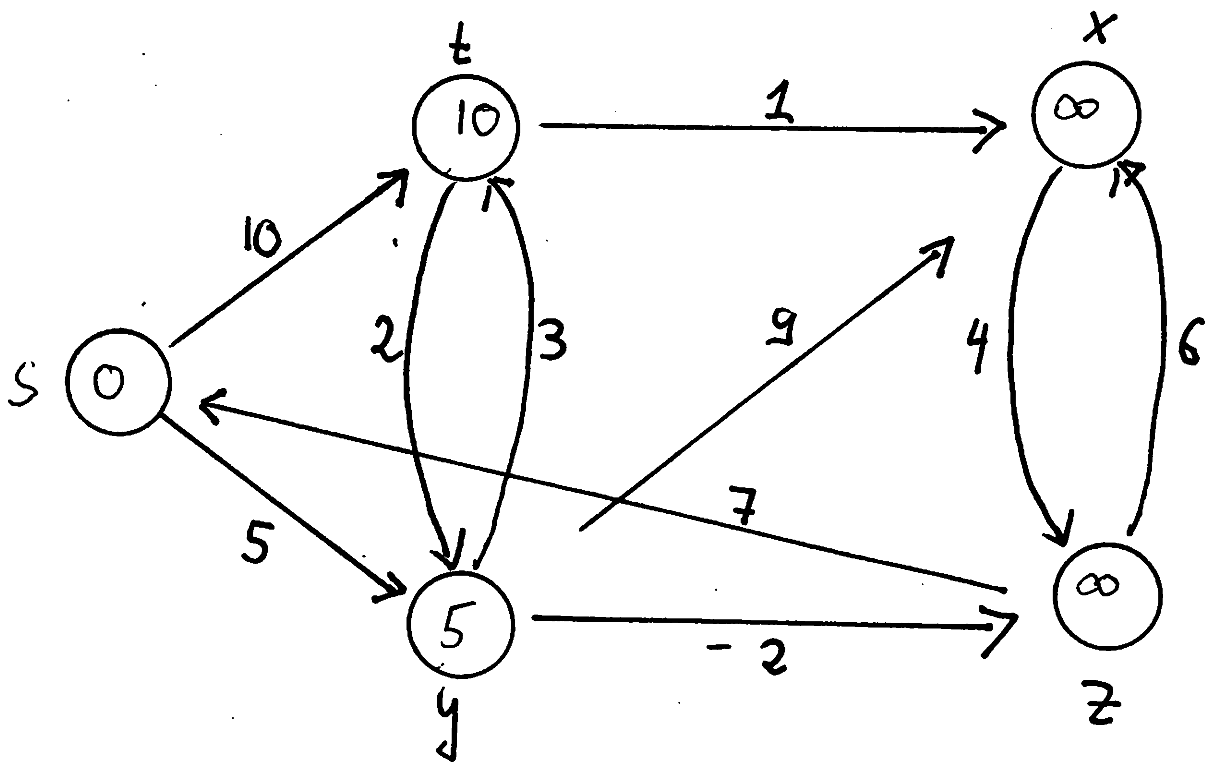
So, the lines 5-7 test if after executing the loop (2) one can still gain $\Rightarrow$ cycle.

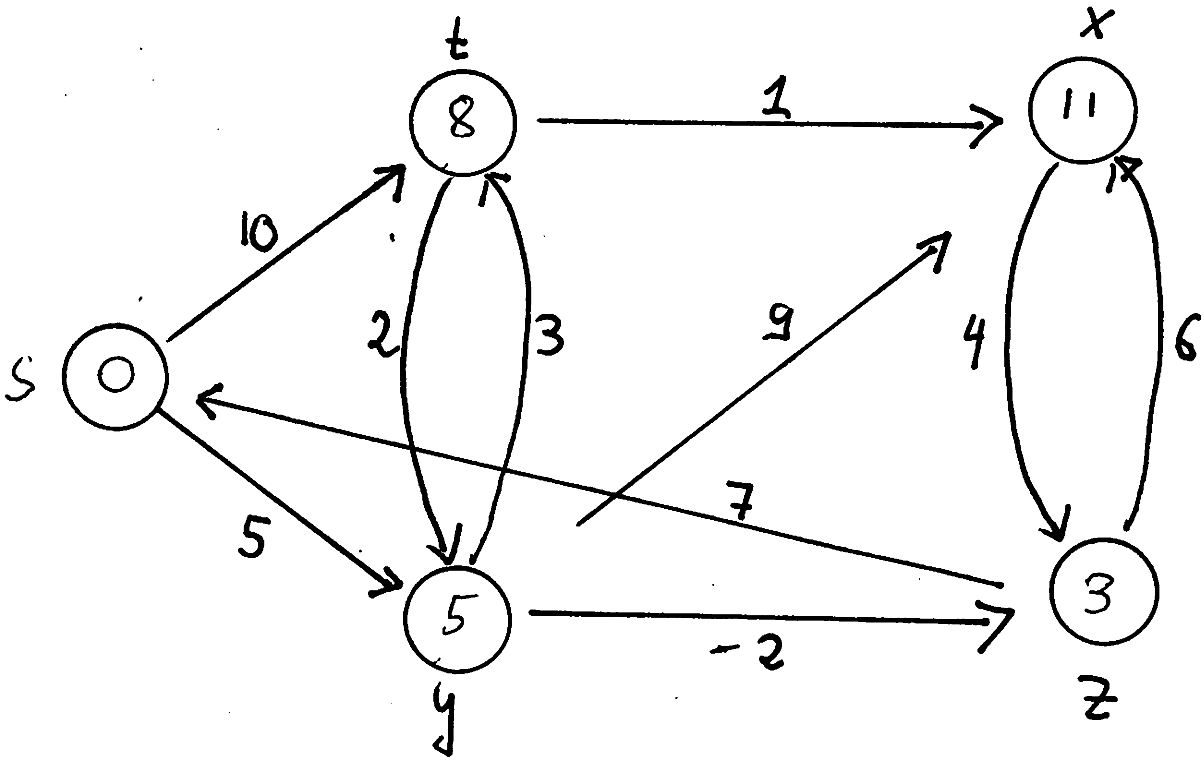
If Bellman-Ford's algorithm "sees" no changes between 2 iterations of the main loop (2) then it can terminate.
This is frequently, an easily implementable, improvement in actual run-time.

Construct for yourself 2 examples when this is a
(a) <u>major</u> improvement and
(b) <u>no</u> improvement.

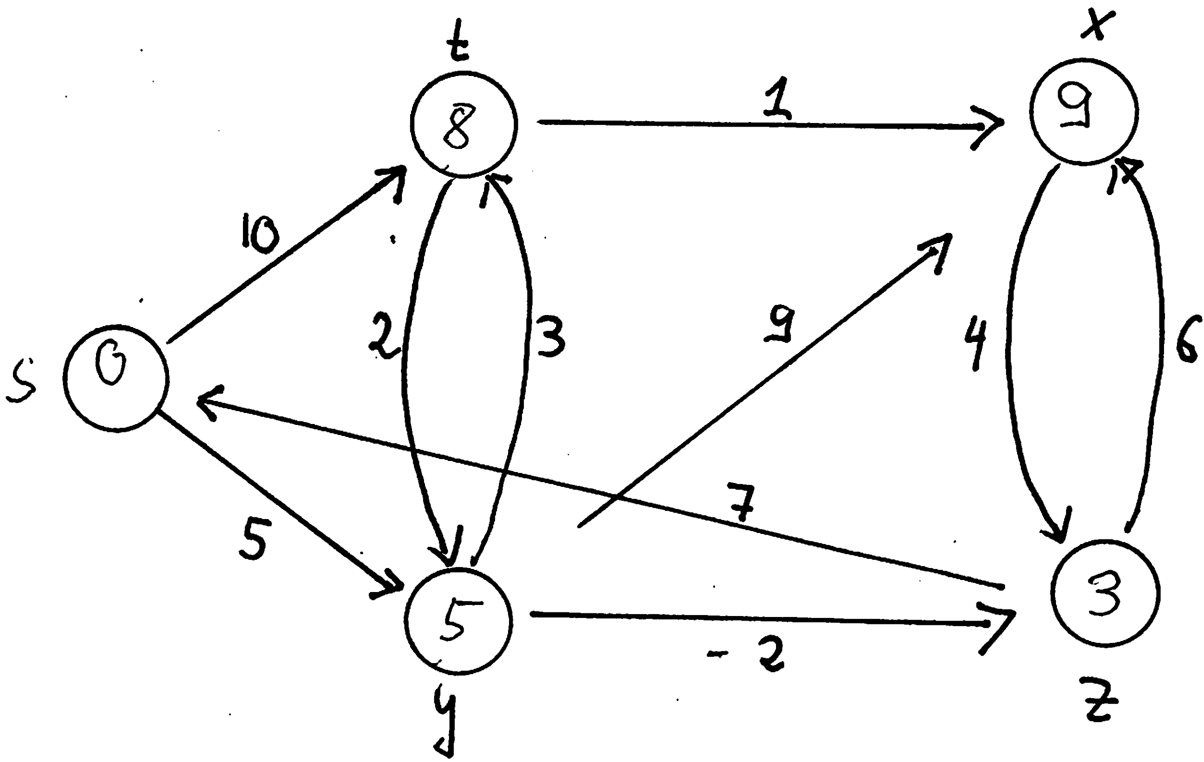
Example




t
10
x
∞
1
10
2
3
9
4
6
s
0
7
5
5
-2
∞
y
z



D34



D35

Chapter 5

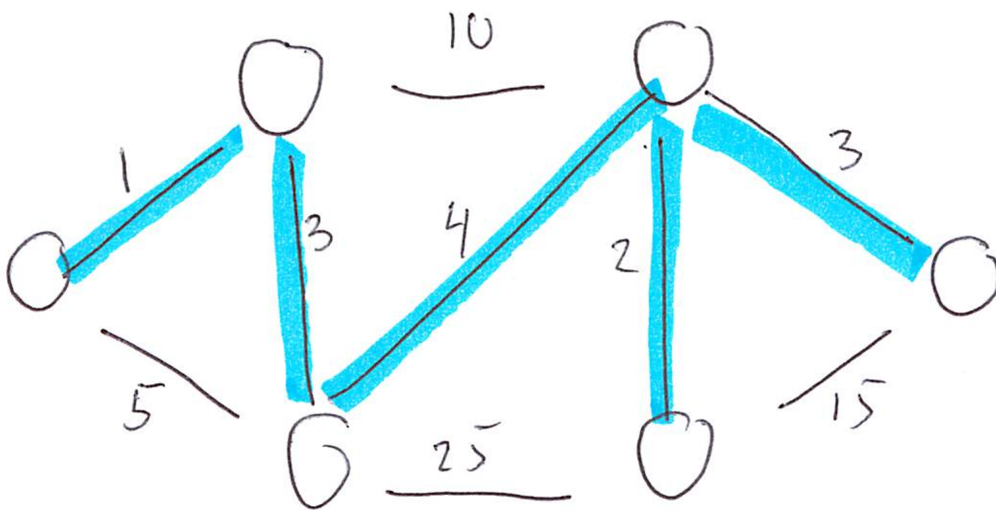
Greedy Algorithms

Informally, an algorithm is greedy if it builds a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit.

Think about real-life examples when this greedy approach succeeds and where not.

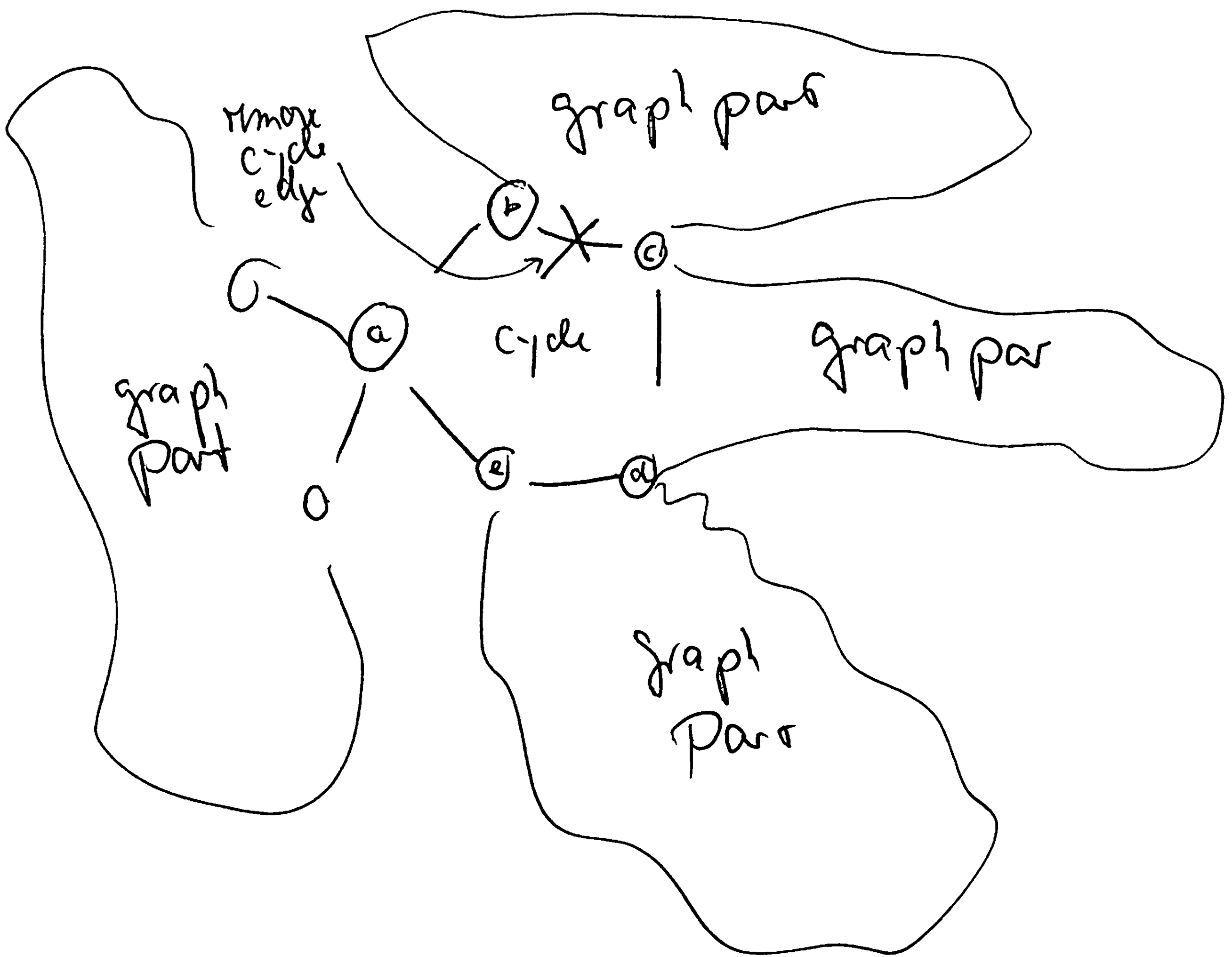
Minimum Spanning Trees

Example: Computer Network with min amount of wire between them.



First some facts / Properties

1. Removing a cycle edge cannot disconnect a graph.



a – b – c – d – e is a cycle.

if we remove say, b–c, and

there is a path

x, . . . , b, c, . . . , y

we can always replace this path b

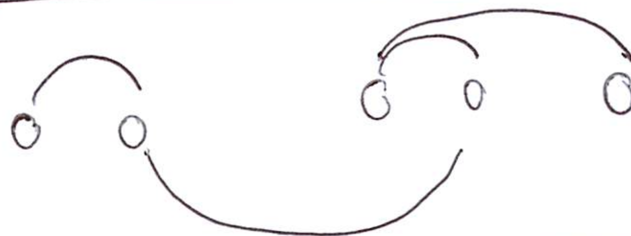
x, . . . , b, a, e, d, c, . . . y

when the parts ... stay.

Properties cont'd

(2) A tree on n nodes has n-1 edges.

Build the tree adding one edge at a time.



each time we add an edge 2 components that were disconnected are connected.

after n-1 edges all initially singleton nodes are in one connected component

(Adding an edge inside a component creates a cycle.)

---

③ Also, the converse is true:

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

Assume that G has a cycle.
Then remove one of the cycle edges.
After all cycles have been removed a graph $G' = (V, E')$, $E' \subseteq E$ is created that is acyclic.
Using Property 1 (repeatedly)
G' is connected $\Rightarrow$ $|E'| = |V| - 1$
by Property 2. (if $E' = E$, G was already acyclic.

Property

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

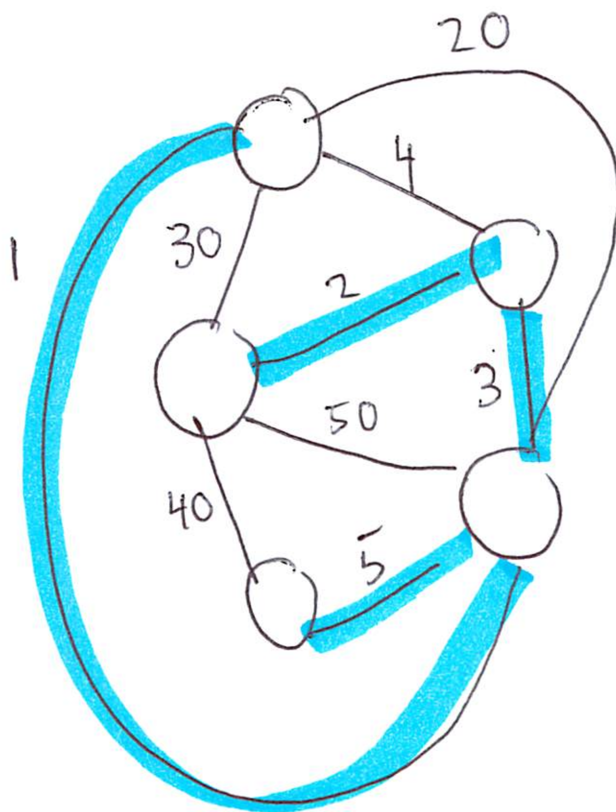
"⇒" If G is a tree there is a unique path as G is connected and if there were 2 paths a cycle would result.

"⇐" if there is a unique path between any pair of vertices. Then G is connected and acyclic ⇒ G is a tree.

Definition A minimum spanning tree for a graph $G = (V, E, w)$ is a tree $T = (V, E')$ $E' \subseteq E$ which minimizes

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

We call the minimum spanning tree MST



Minimum Spanning Tree

Kruskal's Algorithm

(Greedy approach)

Procedure Kruskal (G, w)

Input: A connected, undirected graph $G = (V, E)$ with edge weights w_e

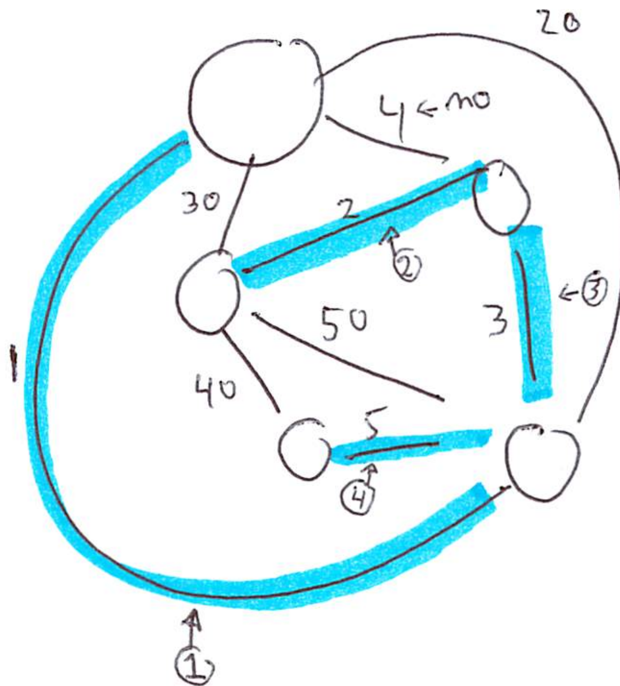
Output: A minimum spanning tree for G

```
for all u ∈ V do
    makeset(u)

X = {}
Sort the edges E by weight
for all edges {u,v} ∈ E, in increasing order (weight)
  do  if find(u) ≠ find(v)
          add edge {u,v} to X
          union(u,v)
```

"Algorithm" for MST

repeatedly add the next lightest edge that does not produce a cycle.



Ⓐ Does it work?

Ⓑ What is the time complexity?

Ⓒ Can we do better?

Ⓑ depends on how fast we find the lightest edge

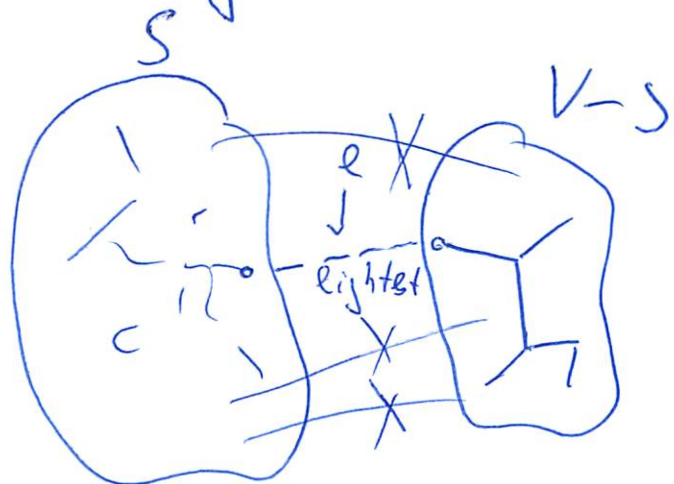
(how fast we can test if we create a cycle) ← easy!

which component does a vertex belong to

CUT Property (MST)

Minimum spanning tree

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



X : not in MST

lightest = minimum weight edge

So, we have 3 operations

makeset (x): create singleton set for x

find (x): to which set does x belong?

union (x, y): merge the sets containing x and y

For Kruskal's algorithm:

|V| makeset operations

2|E| find

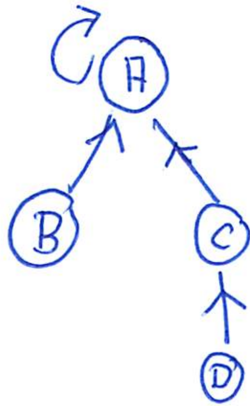
|V| - 1 union

so, how to implement these?

The operations are also used in other algorithms.

Disjoint Sets Problem

Store a set in a tree,



← the root is the representative of the set

i.e. the name of the set

procedure makeset (x)

$\pi(x) = x$

rank(x) = 0 [height of subtree]

function find(x)

while $x \neq \pi(x)$ do $x = \pi(x)$

return x

(follows the up pointers to the root and return that root value)

Procedure union(x, y)

$r_x = find(x)$ — representative finding

$r_y = find(y)$

if $r_x = r_y$: return — are the equal! sets are equal

if $rank(r_x) > rank(r_y)$ — link the smaller tree as child of larger

then $\pi(r_y) = r_x$

else

$\pi(r_x) = r_y$

if $rank(r_x) = rank(r_y)$ then

$rank(r_y) = rank(r_y) + 1$

rank (height) goes up when the trees have equal heights.

Example of sequence of operations in class

also question:

What would happen if we were to hang the larger tree under the smaller i.e. reverse the inequality in procedure union?

Property 1 :

For any x, $rank(x) < rank(\pi(x))$

Proof obvious

Property 2 : Any root node of rank k has at least 2^k nodes in its tree.

Corollary : analogously to Property 2, the result holds for any node not only the root.

Property 3 If there are n elements overall, there are at most $n/2^k$ nodes at rank k.

What does this imply for our analysis?

Property 3 $\Rightarrow$ <u>maximum rank is $\log n$</u>

$\Rightarrow$ height of each tree $\leq \log n$

$\Rightarrow$ find(x) is $O(\log n)$

Since union(x, y) calls find(x) and find(y) and otherwise is $O(1)$

Union(x, y) is $O(\log n)$

Note: $O(\log |E|) = O(\log |V|)$
because $|E| \le |V|^2$ and
$\log(|E|) \le \log(|V|^2) = 2 \log |V|$

Kruskal's algorithm

sort the edges by weight $O(|E| \log |V|)$

then $|V|$ makeset ops $O(|V|)$

$|E|$ union/find ops $O(|E| \log |V|)$

$O(|E| \log |V|)$

Can this be improved?

assume that the weight are sorted or are sorted faster than $O(|E| \log |V|)$ e.g., if weights are bounded.

Then, we need to improve the complexity of find (x).

This can only be done if the height is better than log n.

This may not be possible w.c. but is possible over a sequence of operations. Some operations will take longer, some will be very fast. Let us sum the costs up for all operations.

Amortized cost per operation is this sum divided by the number of operations.

With a nice "trick" the amortized cost per find operation can be shown to be <u>almost</u> $O(1)$, it is $O(\log^* n)$.

$\log^* n$ is the number of time the log can be taken before 1 or < 1 is reached.

<u>Example</u>: $\log^* 1000 = 4$

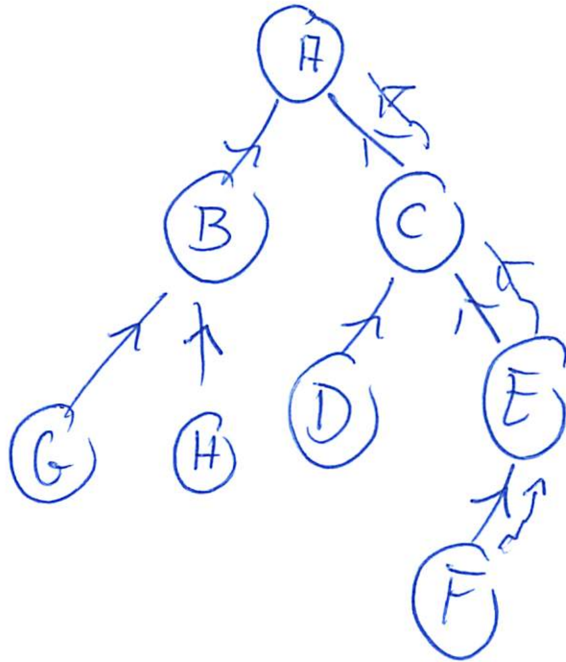
$\log \log \log \log 1000 \leq 1$

$\log^* n \leq 5$ if $n \leq 2^{65536}$

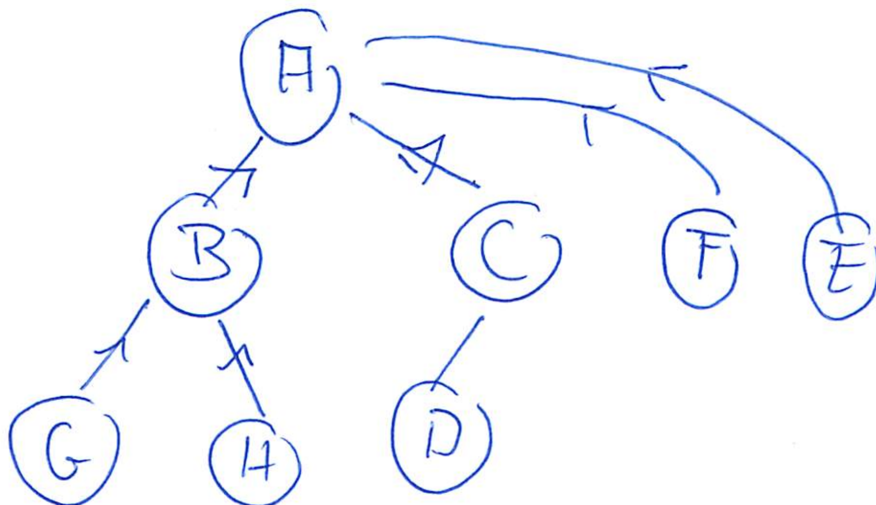
for all practical problems this will hold.

What is the "trick"?

When a find(x) operation is done, we go up the tree to the root.



Why not reduce this for next time?



hang the node on the path directly under the root.

function find(x)

  if $x \neq \pi(x)$ then

    $\pi(x) = find(\pi(x))$

  return $\pi(x)$

This path compression reduces the amortized cost per find/union operation to

$O(\log^* n)$

$\Rightarrow$ $O(|E|)$ union/find operation can be carried out in

$O(|E| \log^* n)$ time.

The idea of the analysis in the "amortized" way is to give operations some money. The cheap operations are not paying. The other operations are then shown to "pay" no more than made available - i.e., the amortized cost * # of operations.

Details not covered here but are in text book.

PRIM's Algorithm

$X = \{\}$ (edges picked so far)

<u>repeat</u> 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 $V-S$

$X = X \cup \{e\}$.

somewhat similar in flavour to Dijkstra's algorithm.

The subtree defind by X alwaysgrowth by one edge, which is the lightest edge between S and V - S.

it is the smallest cost vertex

$$Cost(v) := \min_{u \in S} w(u, v)$$

The correctness follows from the Cut Property.

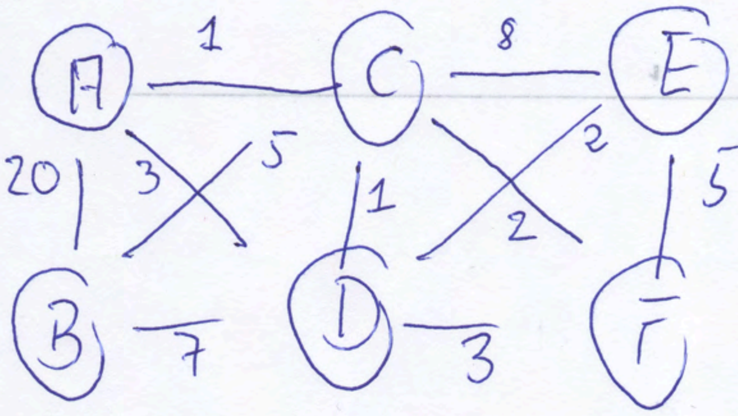
Detailed Prim's Algorithm

Procedure Prim (G, w)

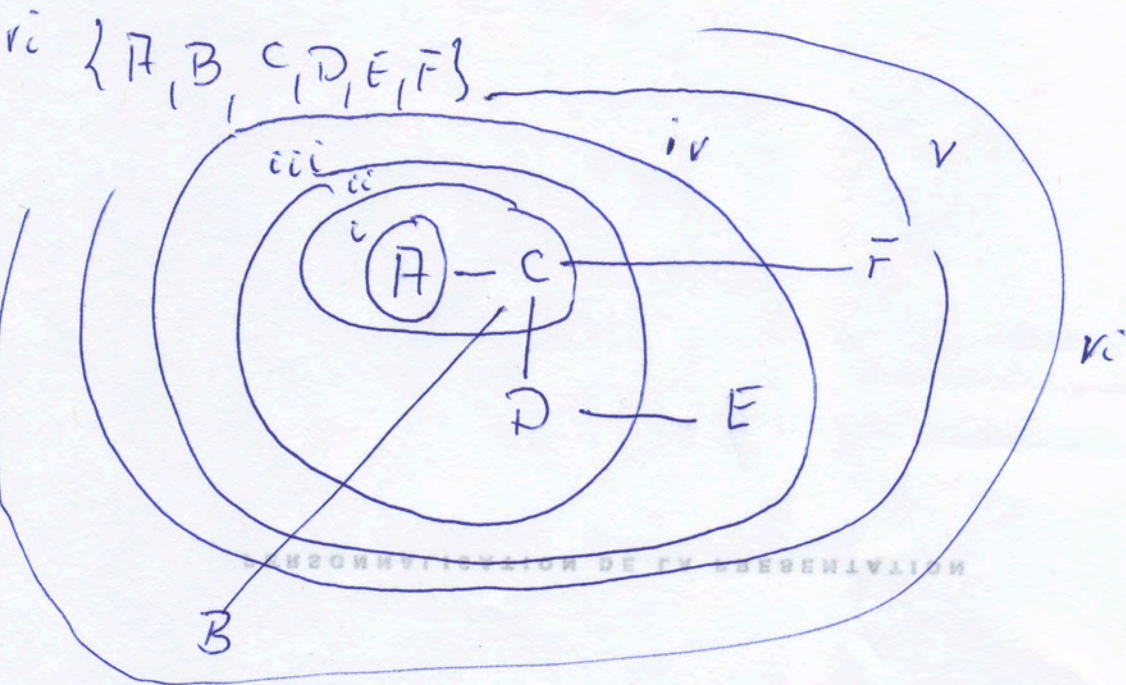
input: A connected undirected graph $G = (V, E)$ with edge weights w_e

Output: A minimum spanning tree for G

```
for all u ∈ V
    cost(u) = ∞
    prev(u) = nil
pick any initial node u₀
cost(u₀) = 0
H = makequeue(V)   [PQ : keys = costs]
while H is not empty do
    v = deletemin(H)
    for each (v, z) ∈ E
        if cost(z) > w(v, z)
            then cost(z) = w(v, z)      }
            prev(z) = v                 }  note errors in textbook
            decreasekey(H, z)           }
```



Set S	A	B	C	D	E	F
{ }	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
i {A}		20/A	1/A	3/A	∞/nil	∞/nil
ii {A, C}		5/C		1/C	8/C	2/C
iii {A, C, D}		5/C			2/D	2/C
iv {A, C, D, E}		5/C				2/C
v {A, C, D, E, F}		5/C				
vi {A, B, C, D, E, F}						



Dynamic Programming

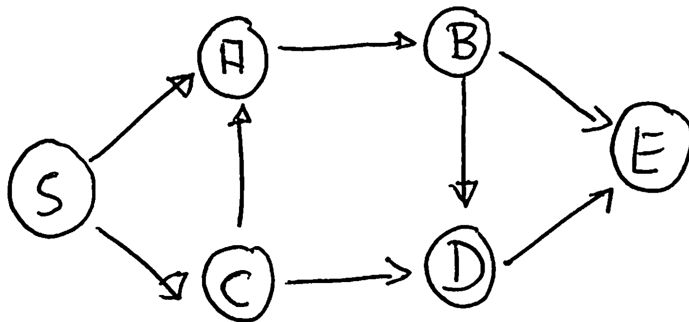
Chapter 6 Textbook

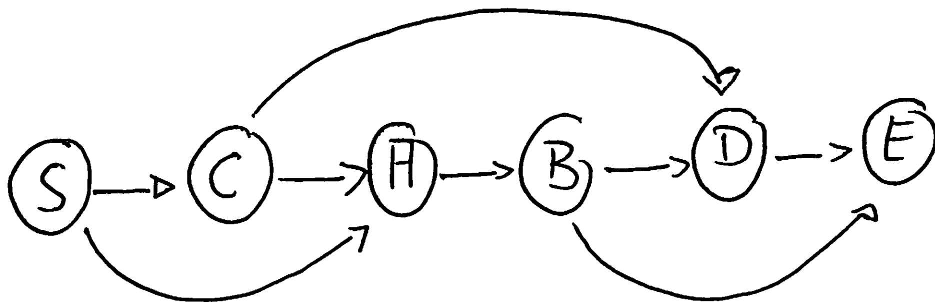
A useful, albeit sometimes costly, tool.

<u>An example</u> Shortest Paths in DAGs
(DAG = directed acyclic graph)

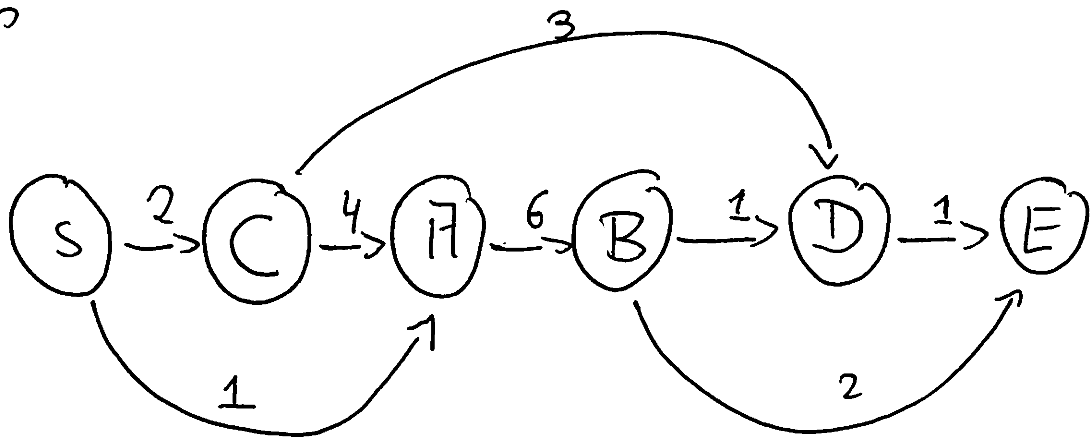
<u>Property</u> DAGs can be linearized, i.e., arranged on a line and edges are oriented left to right.

<u>Reason</u>: topological sort





Now consider the shortest path problem on DAGs. So, introduce costs on the edges



Now realize that, in order to compute the distance to a node further right we need to know the distance to its left vertices. We start from S, as always.

ex.: 1) $dist(D) = \min(dist(C) + 3, dist(B) + 1)$

2) $dist(E) = \min(dist(B) + 2, dist(D) + 1)$

This holds for all vertices analogously.

A single pass over the linearized graph gives all distances from S.
$D(v) = dist(v)$

<u>Procedure</u> SP(DAG)

<u>input</u>: linearized DAG $G=(V,E)$, weights l
<u>output</u>: all distances $D(v)$, $v \in V$
$D(v)$ = distance s to v in G

initialize $D(\cdot)$ to ∞
$D(s) = 0$
<u>for</u> each $v \in V \setminus \{s\}$ <u>do</u>
in linearized order

$$D(v) = \min_{(u,v) \in E} \{D(u) + l(u,v)\}$$

Analogously, we can compute the longest path. Note that longest path is <u>not</u> always as easy as shortest path.

<u>Dynamic programming</u> computes the value at a node using a function (min or max) of values of predecessor node.

A problem is solved by solving (many) subproblems & combine their solutions via minimization (maximization) to get a solution to the original problem.

Another example, LIS
(longest increasing subsequence)

Input: A sequence of numbers $a_1, \dots, a_n$

Output: A subsequence $a_{i_1}, a_{i_2}, \dots, a_{i_k}$
$i_1 \le i_2 \le \dots \le i_k \le n$

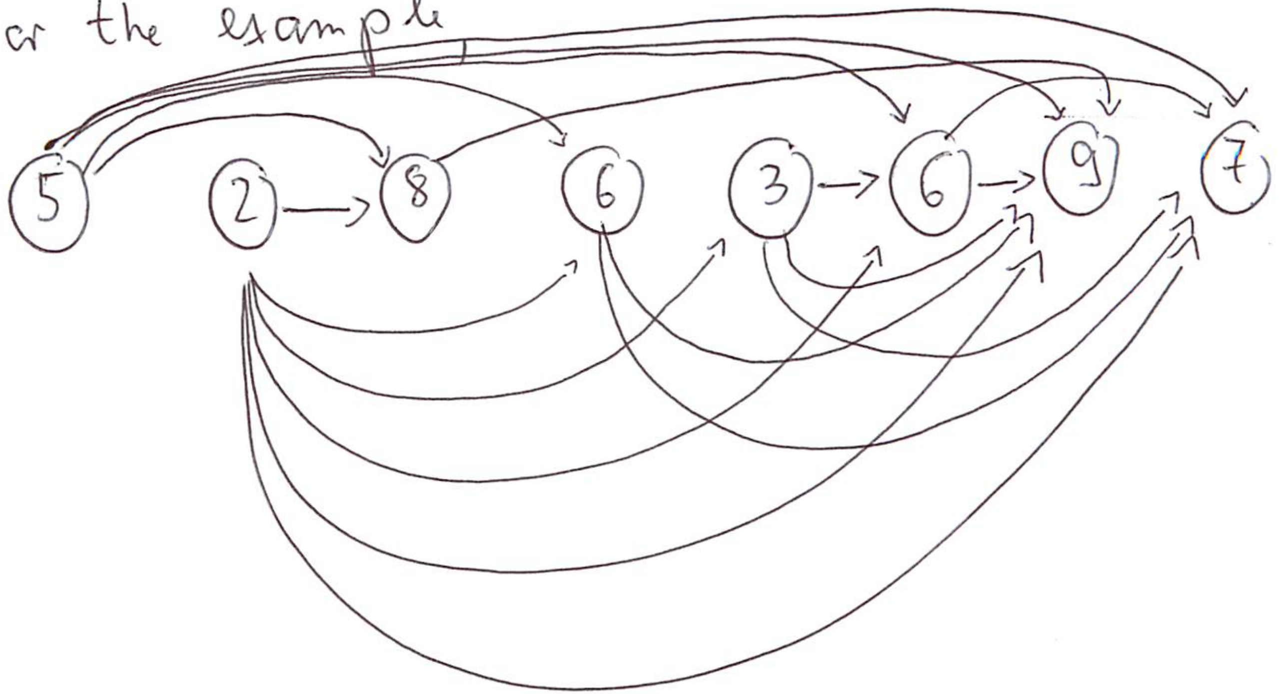
s.t. $a_{i_1} \le a_{i_2} \le \dots \le a_{i_k}$ and
k is the largest integer for which this holds.

Example. 5 2 8 6 3 6 9 7
a_1 a_2 a_3 a_4 a_5

5 8 9 is an increasing sequence $k=3$
but
2 3 6 9 is both increasing and longer $k=4$

Let us restate this in terms of a graph. The vertices are linearized by the input order, the edges $a_i \to a_j$ are $(a_i, a_j) \in E \Leftrightarrow i < j$ and $a_i < a_j$

For the example



<u>Observation</u> : This is a DAG. G

LIS problem $\hat{=}$ longest path in G

<u>for</u> $j = 1, \ldots, n$ <u>do</u>

$\quad L(j) = 1 + \max\{L(i) : (i,j) \in E\}$

<u>return</u> $\max_j L(j)$

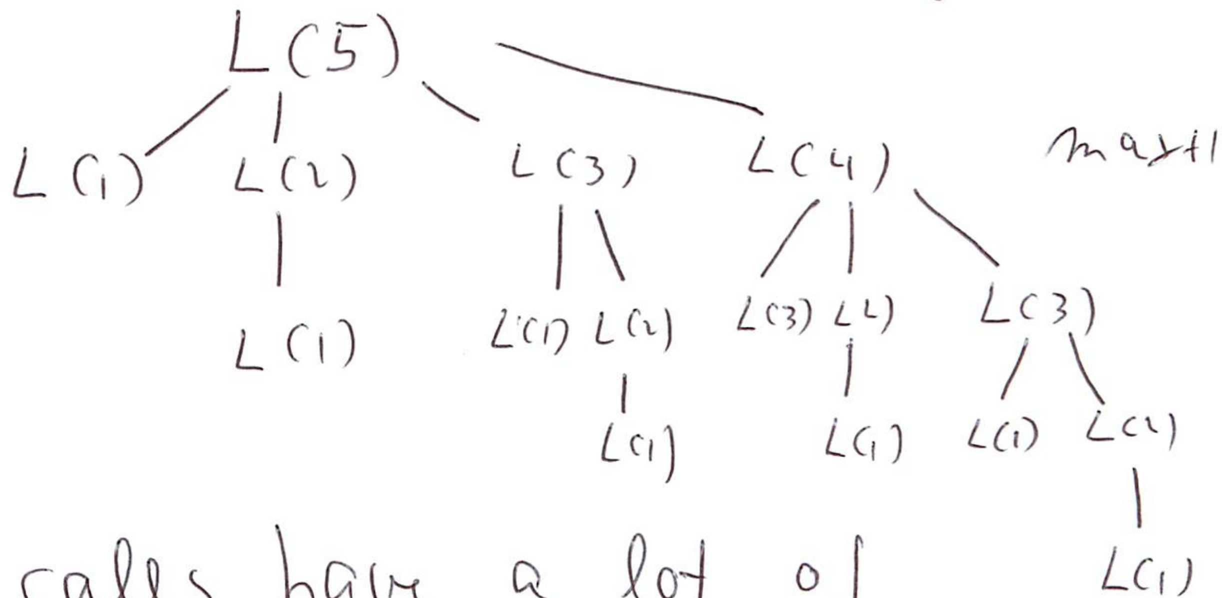
Why is dynamic programming sometimes good?

Consider the LIS problem.
Now, solve it via a <u>recursive algorithm</u>. [bad here but we want to understand why]

Say that the a_i's happen to be sorted. Then

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$$

Visualize the recursion, say $j=5$



The calls have a lot of redundancies. Dynamic Programming stores the results so that they are available.

Question how many calls are made to $L(1), L(2), \ldots$?

Divide & Conquer (recursion) does well when subproblem sizes are dramatically reduced.

Dynamic Programming builds up solutions from all smaller sized problems.

A geometric problem for dynamic Programming.

Min Weight Triangulation

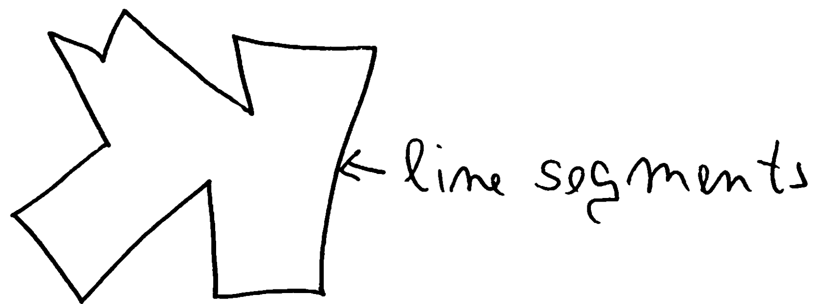
Motivation: e.g. 1) CAD
2) estimation of ice thickness

See class for details

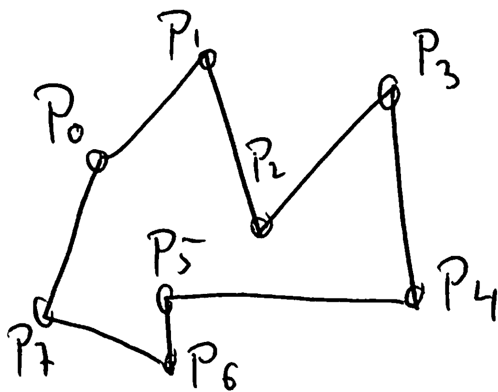
Triangulation:

Problem setting for polygons in fact let us look at convex polygons

A polygon is a closed plane figure bounded by line segments



We order the vertices of the polygon clockwise around its boundary.

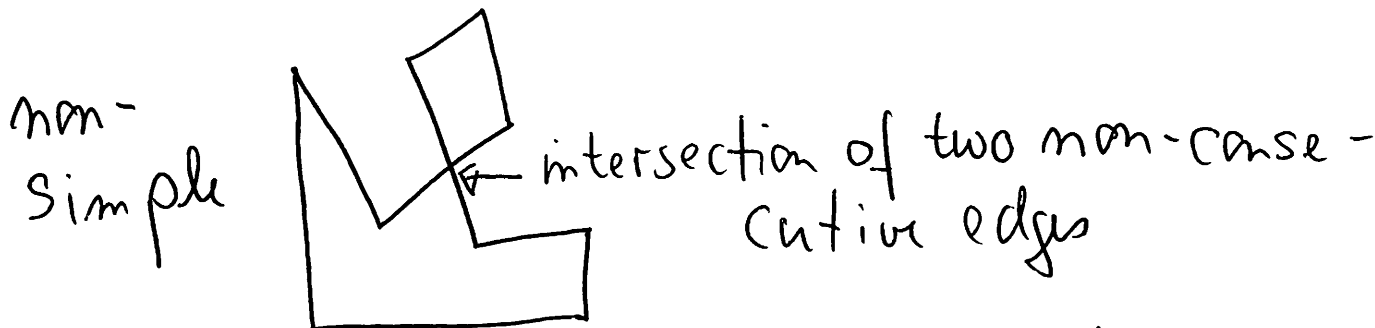


polygon $= (P_0, P_1, \ldots, P_n)$

P_{i-1}, P_i is a line segment $i = 1, \ldots, n$.

Typically one distinguishes between simple polygons and non-simple or arbitrary polygons.

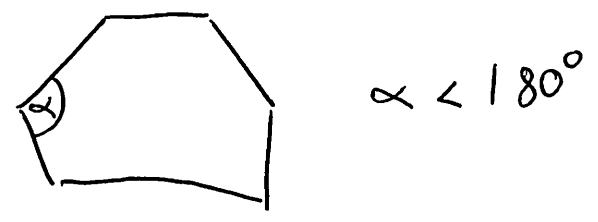
A polygon is non-simple if there are two edges e_i, e_j, $j \neq i+1, j \neq i-1$ which intersect.



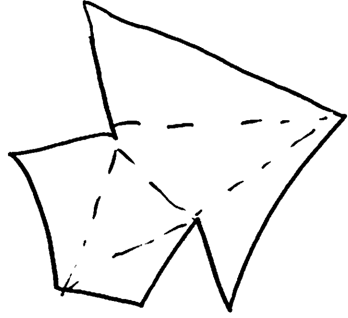
Consecutive edges are only allowed to intersect at their common end point.

(*) We consider from now on only simple polygons

A polygon is convex if each internal angle $\leq 180°$.



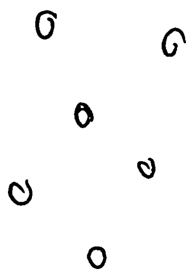
A triangulation of a polygon is a partition of the polygon into triangles. The partition edges must join vertices of the polygon.

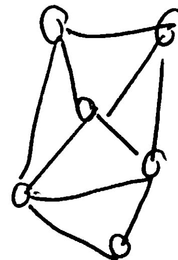


each face is a triangle
additional edges join vertices

An alternate definition is that a triangulation is a planar graph with the maximal # of edges build on the vertices (+ edges of the polygon).

The latter definition extends to triangulation of point sets





Triangulations are important intermediate steps for solving many geometric problems.

There are many different triangulations of the same input polygon. They differ substantially.

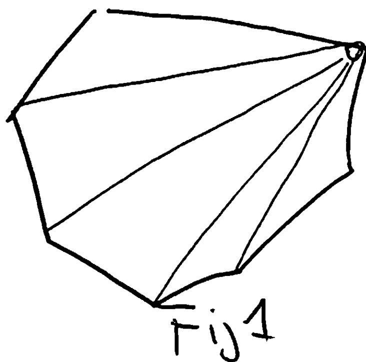


Fig 1

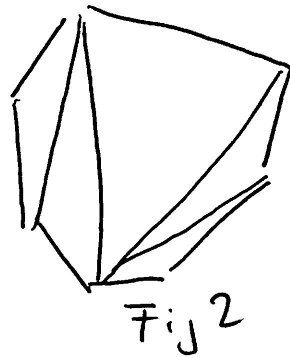


Fig 2

Assume you had to draw these with the min ink possible.

$$\min_{T} \sum_{e \in T} d(e) \quad , \text{ where } T \text{ is a triangulation}$$

$d(e)$ = distance between the 2 vertices of e

We minimize the sum of all distances of the added diagonals over all possible triangulations.

We call $\sum_{e \in T} d(e)$ the weight of triangulation T.

Min Weight triangulation

Find a triangulation of a given polygon with min weight over all possible triangulations.

Solution: Dynamic Programming

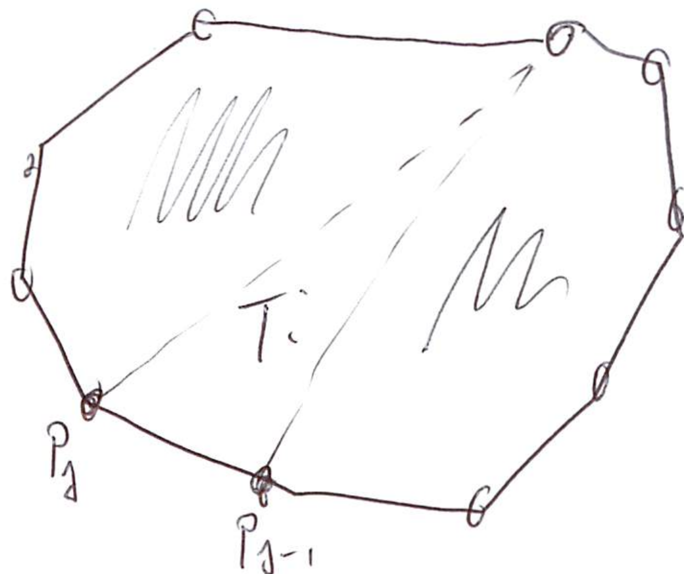
Key insight

there is a triangle T_i which partitions the problem into smaller subproblems of the same type

T_i

Then, the weight of T_i + that of the subproblems is the weight of the entire triangulation.

Can we make it 2 subproblems?



Yes! because the edge $P_{j-1}P_j$ must be in one triangle.

This is a better insight as we have less subproblems to combine and only need to iterate over all vertices with whom $P_{j-1}P_j$ can be in a triangle.

1. Let $v_0, \dots, v_n$ be the $n+1$ vertices of an input polygon.
2. Call $C_{0,n}$ the cost (weight) of the minimum weight triangulation

   $C_{i,k}$ analogously defined for subpolygon with vertices $v_i, \dots, v_k$, $k \geq i+2$

Case 1: $k < i+2$ — no triangulation possible

Case 2: $k \geq i+2$ — then there are choices for a j

$i < j < k$.

For each choice of j, calculate

$\min c_{i,j} + c_{j,k} + w(i,j,k) =: c_{i,k}$

the minimum over all appropriate j

$$c_{i,k} = \begin{cases} 0 & k < i+2 \\ \min_{i<j<k} \{c_{ij} + c_{j,k} + w(i,j,k)\} & \text{otherwise.} \end{cases}$$

Record which index j gives the minimum, call that table m ($m_{i,k} = j$)

This gives the basis for the dynamic programming.

Time complexity $O(n^3)$

Knapsack

We wish to put items in a knapsack. Each item has a weight and a value. The knapsack has a particular max. capacity. The task is to maximize the total value of the items that fit into the knapsack.

Ex:

Item	Weight	Value
1	6	$30
2	3	$14
3	4	$16
4	2	$9

max capacity = 10

Two variants

① each item is unique

② every item is available in unlimited quantity

Solution to ①

item 1 + item 3

capacity reached $6+4=10$

value $30 + $16 = $46

Solution to ②

item 1 + 2 * item 4 value $30 + $18 = $48

capacity: $6+2*2=10$

General solution to (2)

Let $k(w)$ = max value achievable with a knapsack of capacity w

then,

$$k(w) = \max_{i:\, w_i \le w} \{ k(w - w_i) + v_i \}$$

Interpretation:

if the optimal solution with $k(w)$ contain item i, then removing v_i leaves an optimal solution to $k(w - w_i)$

Algorithm

input: capacity W, weights, values
output: optimal solution for $k(W)$

$k(0) = 0$
for $w = 1$ to W
  $k(w) = \max\{k(w - w_i) + v_i \mid w_i \le w\}$
return $k(W)$

1-d table

Complexity: $O(nW)$

General Solution to (1)

So, if no repetitions are allowed once we remove an item w_i we cannot just take $k(w-w_i)$ as the subsolution might contain that element. Since it is unique we cannot use it twice.

Add a second parameter j, $0 \leq j \leq n$

define $k(w,j)$ = max value achievable using a knapsack of capacity w and items $1..j$

Final answer $k(W,n)$

Subproblems

$$k(w, j) = \max\{k(w - w_j, j-1) + v_j, \; k(w, j-1)\}$$

so either v_j is in or not.
at step j

Initialize all $k(0, j) = 0$ and
all $k(w, 0) = 0$

for $j = 1$ to n **do**
  for $w = 1$ to W **do**
    if $w_j > w$ **then** $k(w, j) = k(w, j-1)$
    else $k(w, j) = \max\{k(w, j-1),$
      ~~$k(w - w_j) + v_j\}$~~
      $k(w - w_j, j-1) + v_j\}$

return $k(W, n)$

2-d table required
still $O(nW)$ complexity

6.20

Chain Matrix Multiplication

<u>Task</u>: 4 matrices to multiply

$$A \times B \times C \times D$$

The matrices may have different dimensions, say

A : 50×20

B : 20×1

C : 1×10

D : 10×100

Matrix multiplication is associative.

$A \times B \times C \times D$ can be bracket in different ways

$(A \times B) \times (C \times D)$ $\quad ((A \times B) \times C) \times D$

$A \times (B \times C) \times D$ $\quad A \times ((B \times C(\times D)))$

e.g., $A \cdot B$ has dim $50 \times \underline{1}$
which reduces subsequent computations

Parenthesization	Cost computation	total 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
$(A \times B)(C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7,000

So, with a clever parenthesization one can save MANY computations.

Greedy fails here.

Question: Let $A_1, \ldots, A_m$ be 2-d matrices, A_i has dimension $(m_{i-1}, m_i)_{i=1,\ldots,m}$

Find a parenthesization of

$$A_1 \cdots A_m$$

that minimizes the total cost.

How to find this?

<u>Fact</u>: Valid bracket sequences are in 1-1 correspondence with full binary trees.

A bracket sequence is well-formed if

1) • # "(" = # ")" equal number of opening and closing brackets

2) • # "(" ≥ # ")" when reading the sequence left to right (at any point)

<u>Ex</u>:
- (( )) ( ) is well-formed
- ( )) (is not

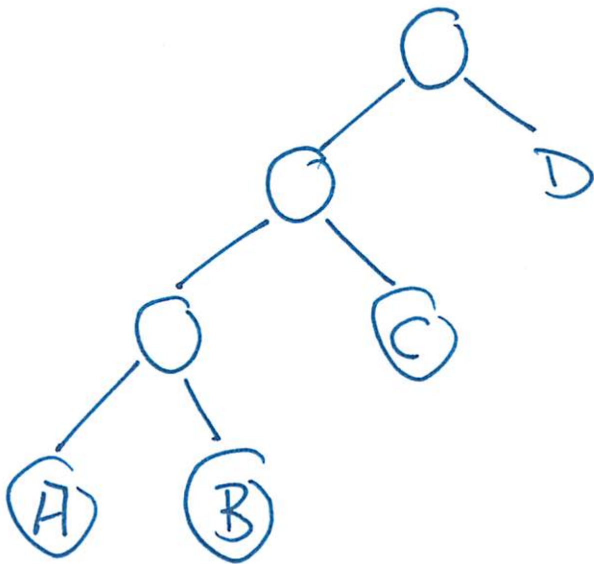
  ↑ # "(" = 1, # ")" = 2 — violates (2)

- ( ) ( ) (is not

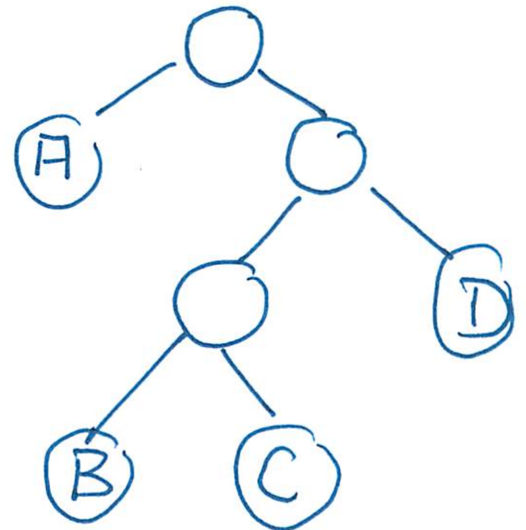
  # "(" = 3, # ")" = 2 — violates (1)

Here, we can see the multiplication arranged in a tree

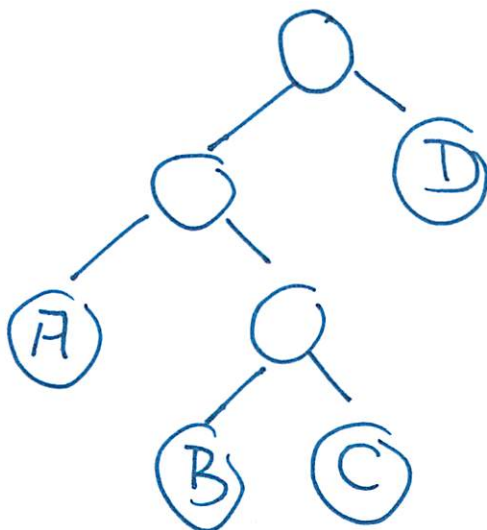
$((A \times B) \times C) \times D$



$A \times ((B \times C) \times D)$



$(A \times (B \times C)) \times D$



Fact: Every subtree corresponds to a particular way of bracketing a subproblem

$$A_i \cdots A_j$$

Fact: For the entire tree to be optimal each subtree must be optimal. (i.e. have optimal cost)

$C(i,j)$:= min cost of multiplying $A_i \times \cdots \times A_j$

$C(i,i) = 0$

$$C(i,j) = \min_{i \le k \le j} \{ C(i,k) + C(k+1,j) + m_{i-1} m_k m_j \}$$

this is

$(\underbrace{A_i \times \cdots \times A_k}_{\min}) \cdot (\underbrace{A_{k+1} \times \cdots \times A_j}_{\min})$ — cost of multiplying ×

Dynamic Program

for $i = 1$ **to** n **do** $C(i,i) = 0$

for $s = 1$ **to** $n-1$ **do**

  for $i = 1$ **to** $n-s$ **do**

    $j = i+s$

    $C(i,j) = \min\{ C(i,k) + C(k+1,j) + m_{i-1} m_k m_j$ ovr $i \le k < j \}$

return $C(1,n)$

2-d table each entry cost $O(n)$
to compute: $O(n^3)$

The dynamic program prevents us from having to construct all binary trees for the bracketings. This would be exponential.

MATRIX-CHAIN-ORDER(p)

```
1  n = p.length - 1
2  let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
3  for i = 1 to n
4      m[i,i] = 0
5  for l = 2 to n              // l is the chain length
6      for i = 1 to n - l + 1
7          j = i + l - 1
8          m[i,j] = ∞
9          for k = i to j - 1
10             q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j
11             if q < m[i,j]
12                 m[i,j] = q
13                 s[i,j] = k
14 return m and s
```

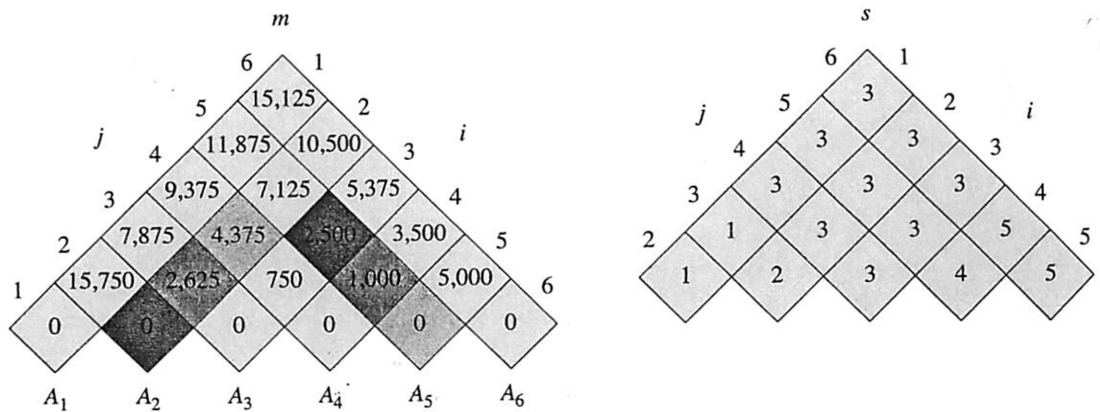


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125\,.$$

7. Linear Programming

(Some material presented uses:
"Formulating Integer Linear Programs:
A Rogues' Gallery"
G. Brown, R. Dell)

Many problems ask to minimize or maximize an objective function given a linear function of certain variables and (possibly) competing constraints.

Linear-programming Problem:

- objective function is linear
- constraints equalities or inequalities (linear)

Example Profit Maximization

2 products P_1 and P_2

1. x_1 boxes of P_1 can be produced per day
2. x_2 " " P_2

3. Profit from P_1: \$1
4. Profit from P_2: \$6

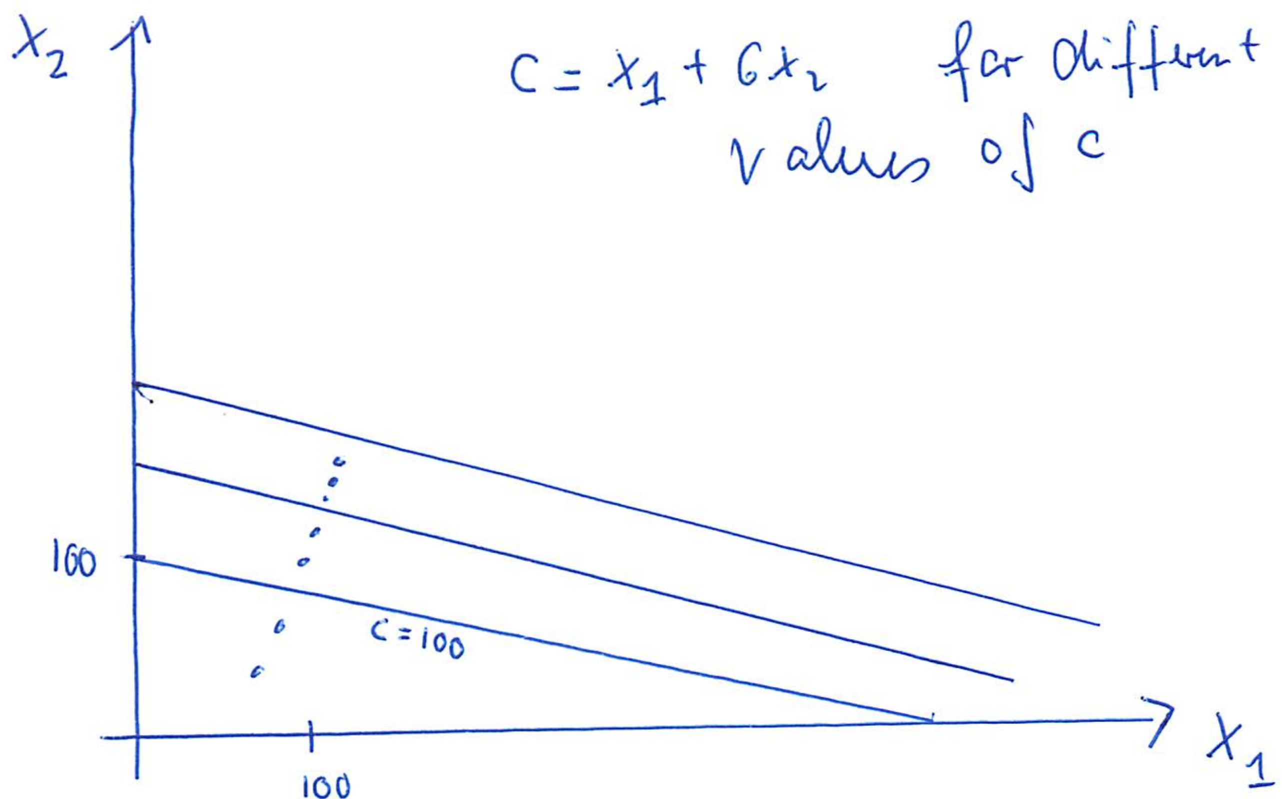
5. daily demand for P_1: ≤ 200
6. " " " P_2: ≤ 300

7. at most a total of 400 boxes can be produced per day

Q: How many boxes of P_1 and P_2 should we produce to maximize our profit?

Objective function $\max \; 1 \cdot x_1 + 6 \cdot x_2$

it is linear in x_1, x_2 in 2-d space



for different values of c, we get different parallel lines.

The higher up the line is, the better the profit.

But, we have constraints preventing us to shift the line arbitrarily high up.

It is obvious $x_1 \geq 0$, $x_2 \geq 0$

5. $\Rightarrow$ $x_1 \leq 200$

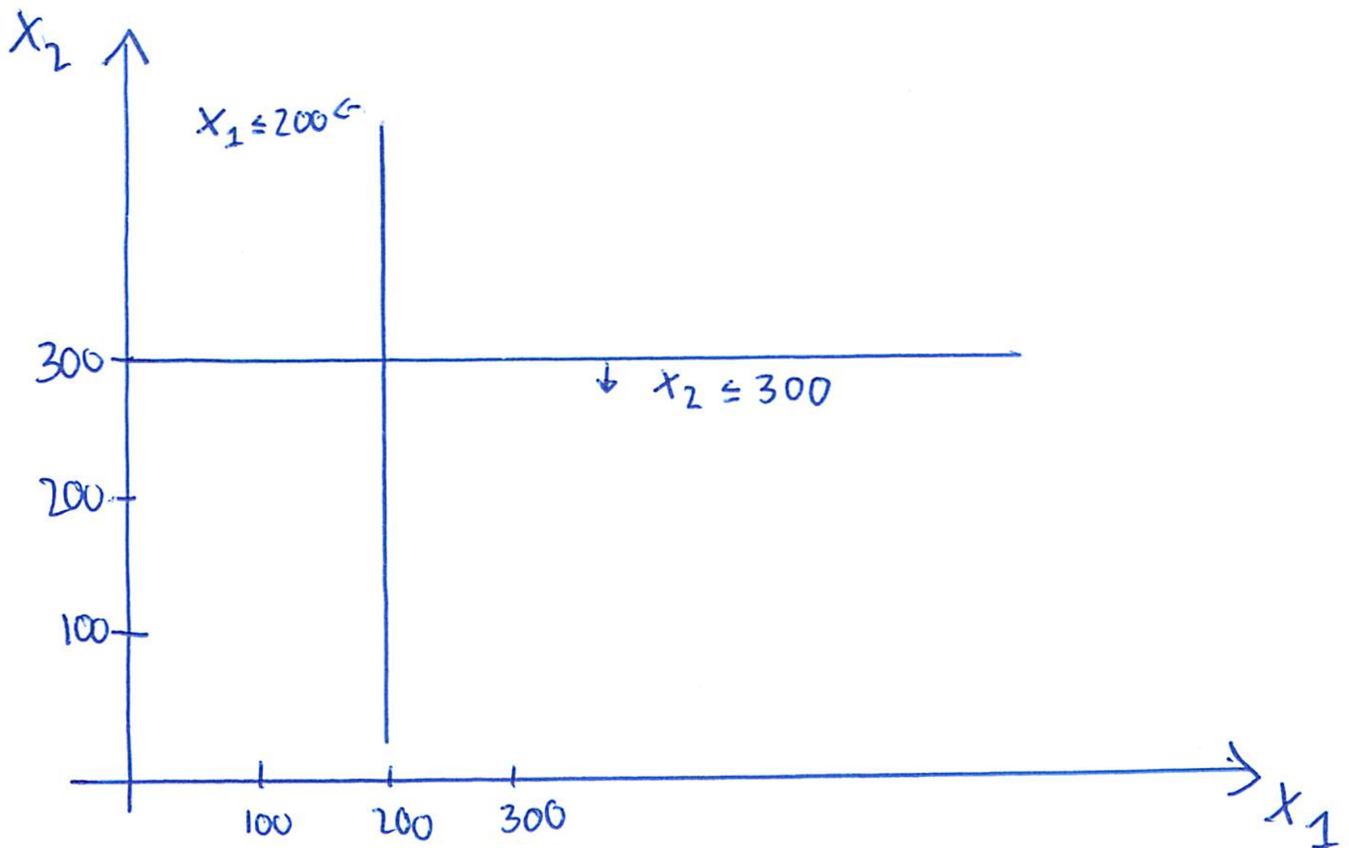
6. $\Rightarrow$ $x_2 \leq 300$

7. $\Rightarrow$ $x_1 + x_2 \leq 400$

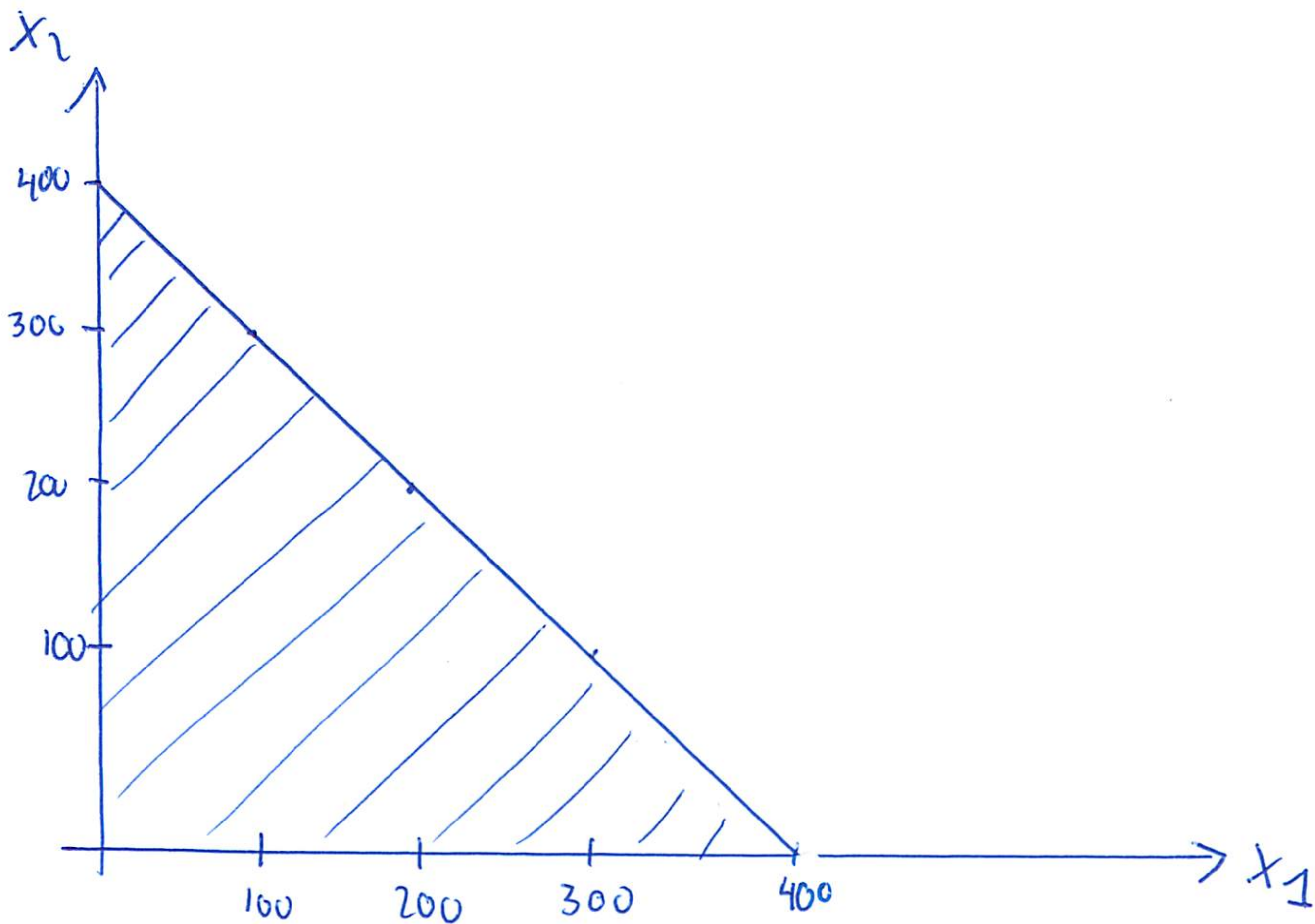
these are the constraints

linear constraints.

How do they look like graphically?

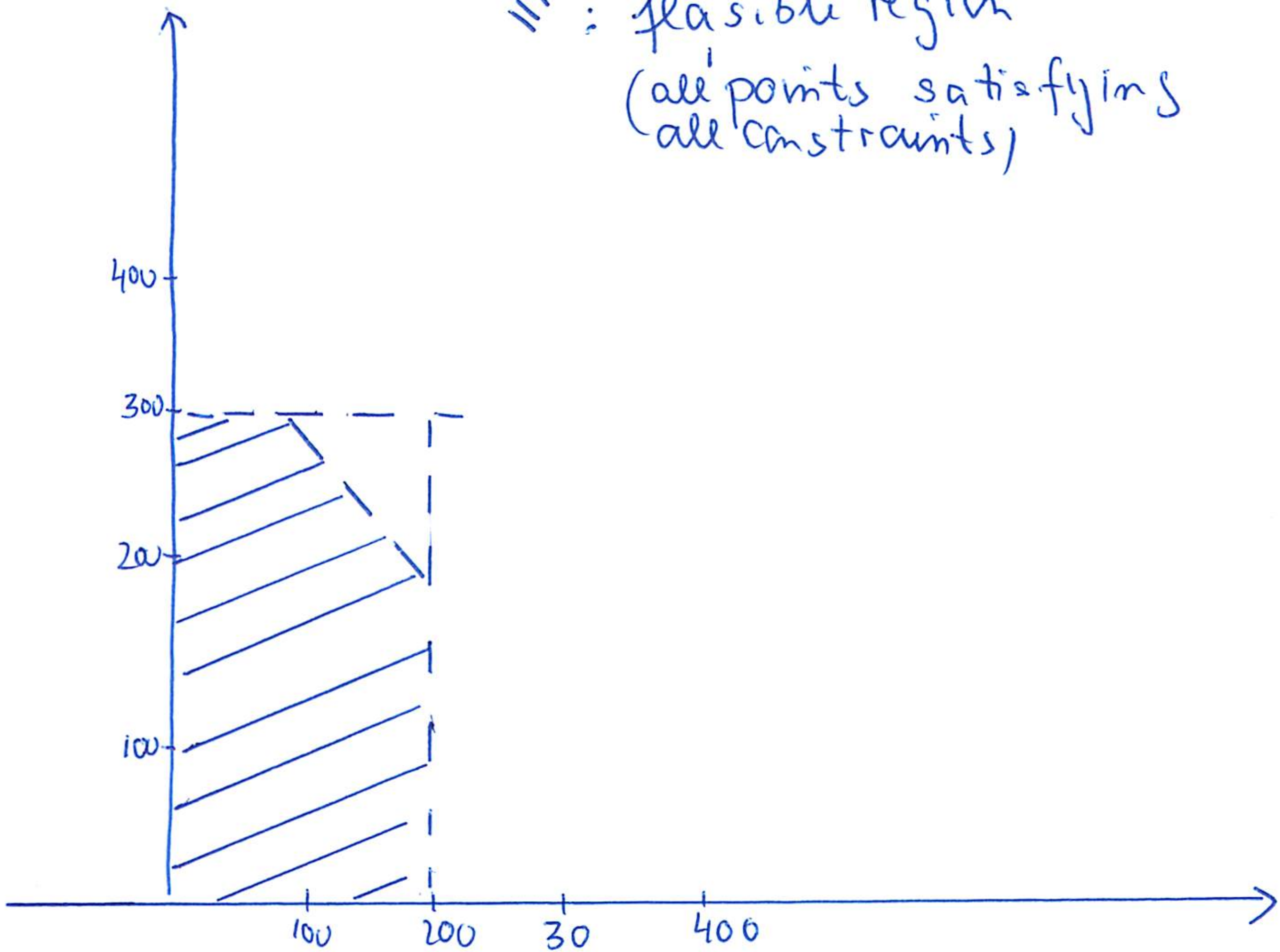


Constraints 5. + 6.

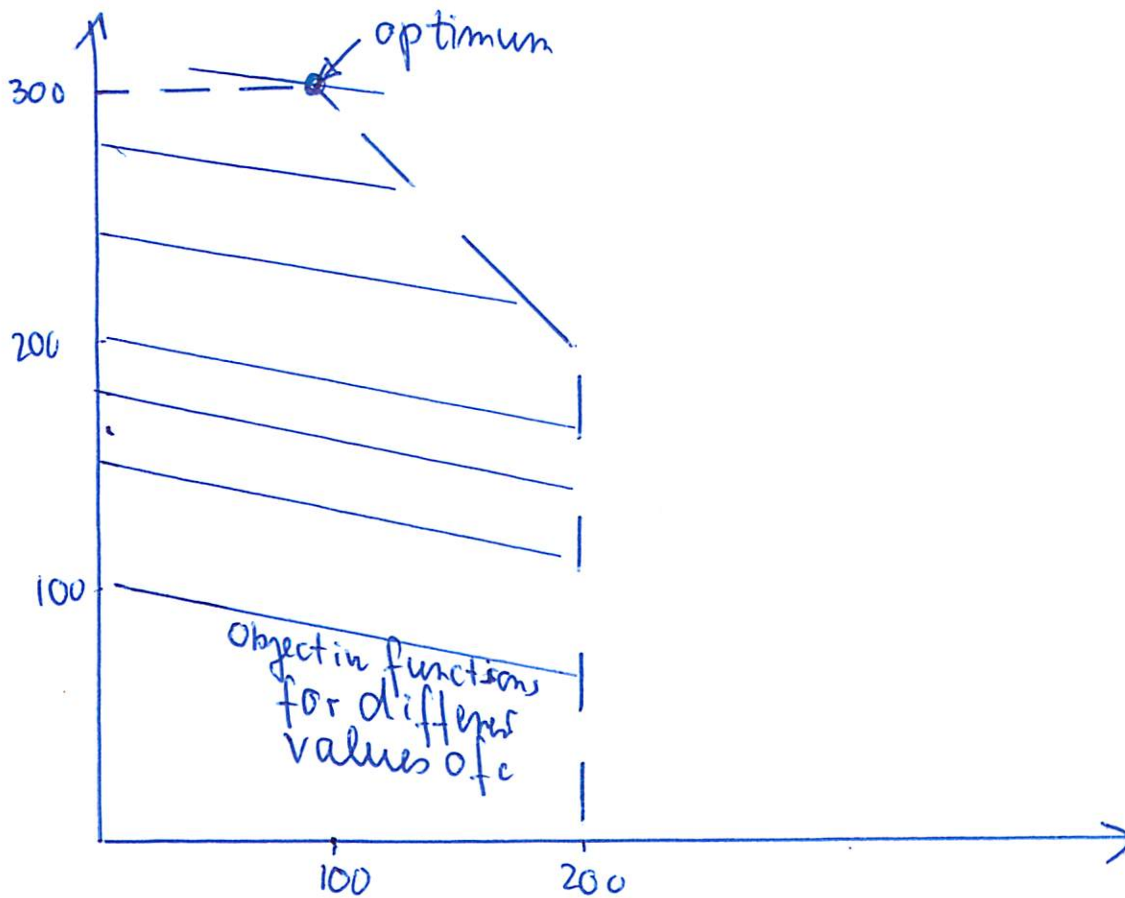


Constraint 7.

Now let us put the constraints into one graph



Now we wish to find a point which maximizes the profit.



the function maximitation means going up & right until we are about leave the feasible region. This happens at a vertex of the feasible region.

7.7

Two cases where the optimum is not at a vertex

① the linear program is infeasible

② there is an unbounded feasible region

①

$x_1 \leq 1$, and $x_1 \geq 2$

these are (obviously) conflicting / contradictory constraints.

$\Rightarrow$ no solution

②

$$\left.\begin{array}{ll} \max & x_1 + x_2 \\ x_1 \geq 0 & x_2 \geq 0 \end{array}\right\} \text{unbounded}$$

Solving linear programs (LP)

Here is the good news:

there are package!

We don't solve, but simply call the LP solver.

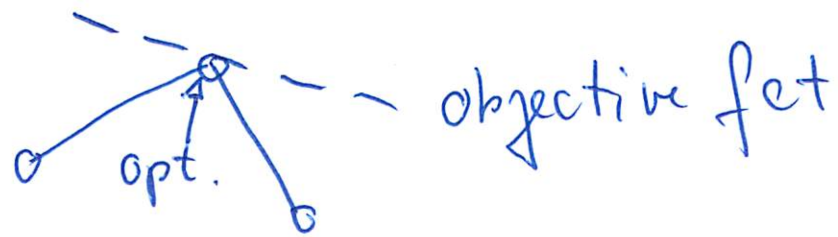
These solvers are hot commercial commodities.

First solution by George Dantzig.

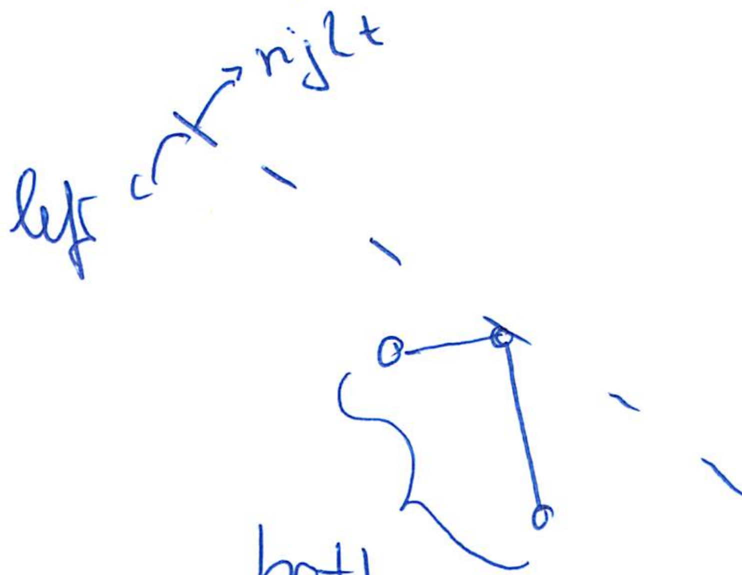
Method is based on hill climbing.
Start at a vertex of the feasible region. Calculate value of objective function.

Look for an adjacent vertex of better value ⇒ hill climbing.

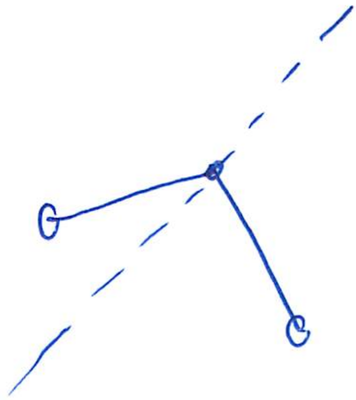
done when a vertex is reached with no better neighbor.



geometrically, both vertices lie on the same side of the objective function



both lie of the left of the line (objective function)



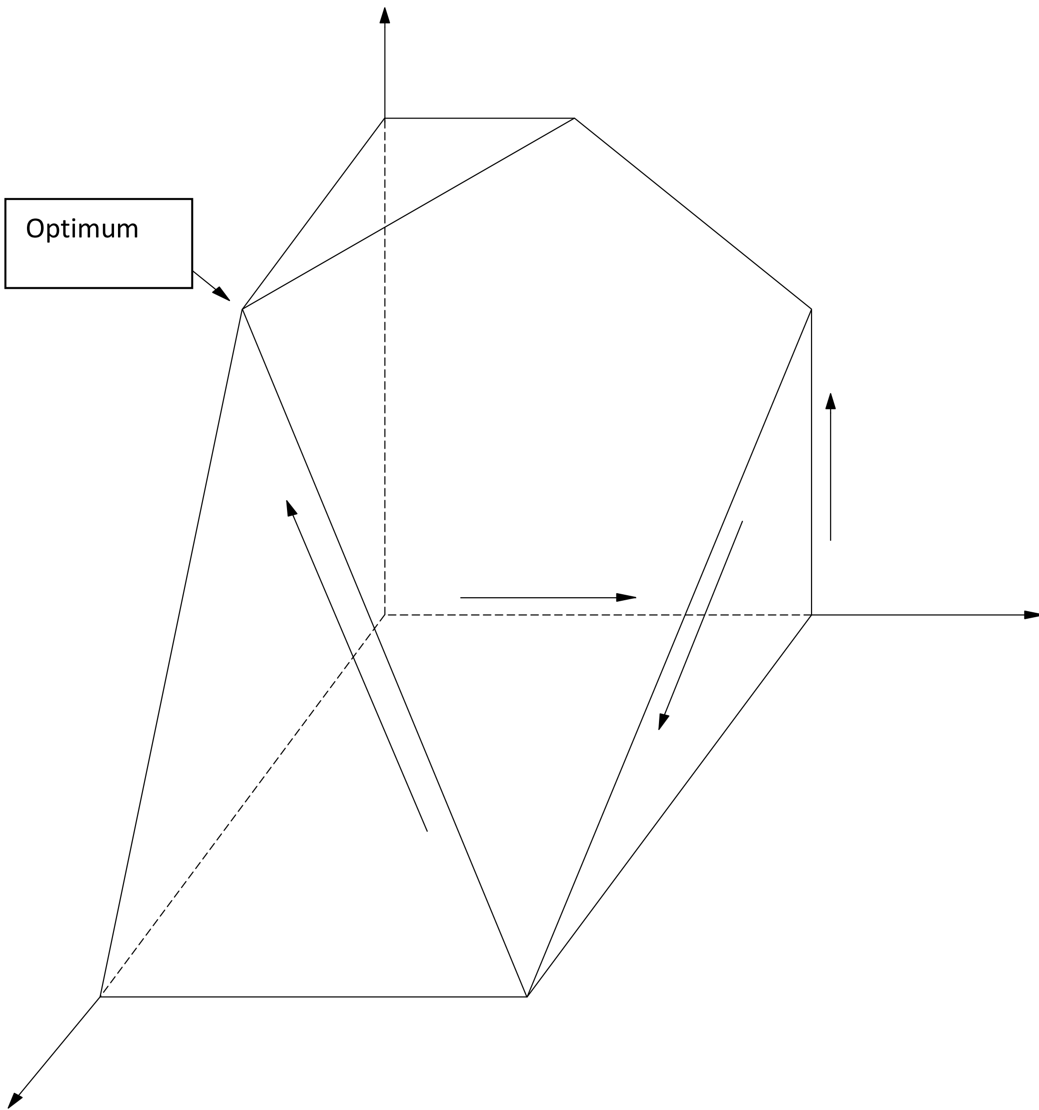
a better solution
is possible as
the 2 vertices
are on opposite
sides

Example 2

1. x_1 boxes of P_1 can be produced per day
2. x_2 " " P_2 " " " " "
3. x_3 " " P_3 " " " " "
4. Profit from P_1 : \$1
5. " " P_2 : \$6
6. " " P_3 : \$13
7. daily demand for P_1 : ≤ 200
8. " " " P_2 : ≤ 300
9. at most a total of 400 boxes can be produced per day
10. P_3 requires 3* more packaging than P_2 and together $3 \cdot P_3 + P_2 \leq 600$

Q: How many boxes of P_1, P_2, P_3 should we produce to maximize our profit?

7.11


Optimum

Objective function

max : $1 \cdot x_1 + 6 \cdot x_2 + 13 x_3$

subject to:

- $x_1, x_2, x_3 \geq 0$
- $x_1 \leq 200$
- $x_3 \leq 300$
- $x_1 + x_2 + x_3 \leq 400$
- $x_2 + 3x_3 \leq 600$

} constraints

Optimal : $(0, 300, 100)$

total profit \$3100

Exercise: check how simplex would examine the vertices.

Main Elements of LP

(in fact of any constrained optimization problem)

- Variables
  values to be found by the solution
  they find/determine the best values for the objective function
- objective function
  expression defined on the variables to express the goal for the optimization
- constraints
  expresses limits on the possible solutions via expressions using the variables
- variable bounds
  typically variables are bounded

In our 1st example:

- variables x_1, x_2
- objective function $\max x_1 + 6x_2$
- constraints $x_1 + x_2 \leq 400$
- variable bounds $0 \leq x_1 \leq 200$
  $0 \leq x_2 \leq 300$

We will group constraints + variable bounds under heading constraints

---

Brown et al. us the term "formulattes"

LP Formulattes

For each formulatte, we write constraints in terms of non-negative, continuous variables $X_1, \ldots$

L1) "For each unit of X_1, there must be at least 5 units of X_2"

$$\boxed{5 X_1 \leq X_2}$$

Frequent wrong answer: $X_1 \leq 5 X_2$

When in doubt make an example.

$X_1 = 3 \quad \Rightarrow X_2 \geq 15$

L2) "A port can load 11 x_1's per week, or 45 x_2's or 30 x_3's. What combination of x_1, x_2 and x_3 can be loaded in 10 weeks?"

$$\frac{1}{11} x_1 + \frac{1}{45} x_2 + \frac{1}{30} x_3 \leq 10$$

Most frequent wrong answer:

$$11 x_1 + 45 x_2 + 30 x_3 \leq 10$$

Sanity check immediate shows that this is wrong.

$$11 x_1 + 45 x_2 + 30 x_3 = 10$$

is also wrong: $x_1 = x_2 = x_3 = 0$ should be a feasible solution

L3) "There must be exactly 7 units of x_1 for every 9 units of x_2"

$$\boxed{\frac{1}{7} x_1 = \frac{1}{9} x_2}$$

(or $9x_1 = 7x_2$)

Again a simple example will tell you if you got it wrong.

L4) "Elements of type x_1 must constitute at most 33% of all elements of type x_1, x_2 and x_3."

$$0.67\, x_1 \leq 0.33\, x_2 + 0.33\, x_3$$

How to obtain this?

$$\frac{x_1}{x_1 + x_2 + x_3} \leq \frac{33}{100}$$

But what is wrong with this constraint?

it is not a linear expression (Linear Programming)

So, we need to convert it.

$$100x_1 \le 33x_1 + 33x_2 + 33x_3$$

$$100x_1 - 33x_1 \le 33x_2 + 33x_3$$

$$67x_1 \le 33x_2 + 33x_3$$

25) "What mixtures of punch from 80-proof x_1 100-proof x_2, and 0-proof x_3 are at least 30-proof?"

again, to derive it we state a non-linear inequality first.

$$80x_1 + 100x_2 + 0x_3 \ge 30(x_1 + x_2 + x_3)$$

$$\boxed{(80-30)x_1 + (100-30)x_2 + (0-30)x_3 \ge 0}$$

L6) "Process x_1 produces 24.5 units of x_2 and 73.1 units of x_3 per hour"

$$x_2 = 24.5\, x_1$$
$$x_3 = 73.1\, x_1$$

Process x_1 produces simultaneously at fixed rates 2 outputs.

These formulations are the basic building blocks for the LP solver input.

PRACTICE

LP solvers

① **Simplex Method**

In this method the algorithm go from corner to corner (of feasible region) always improving for an exponential number of steps.
In practice it runs fast on almost all inputs.

② 1975 a young Soviet mathematician Leonid Khachiyan designed the **ellipsoid method.**
to solve any LP in polynomial time. Theoretically better than the Simplex Method, but not practically.

(3) 1984 Narendra Karmarkar

grad student Uni. of California, Berkley

interior point method

Goes to corners through polyhedron in a clever way.

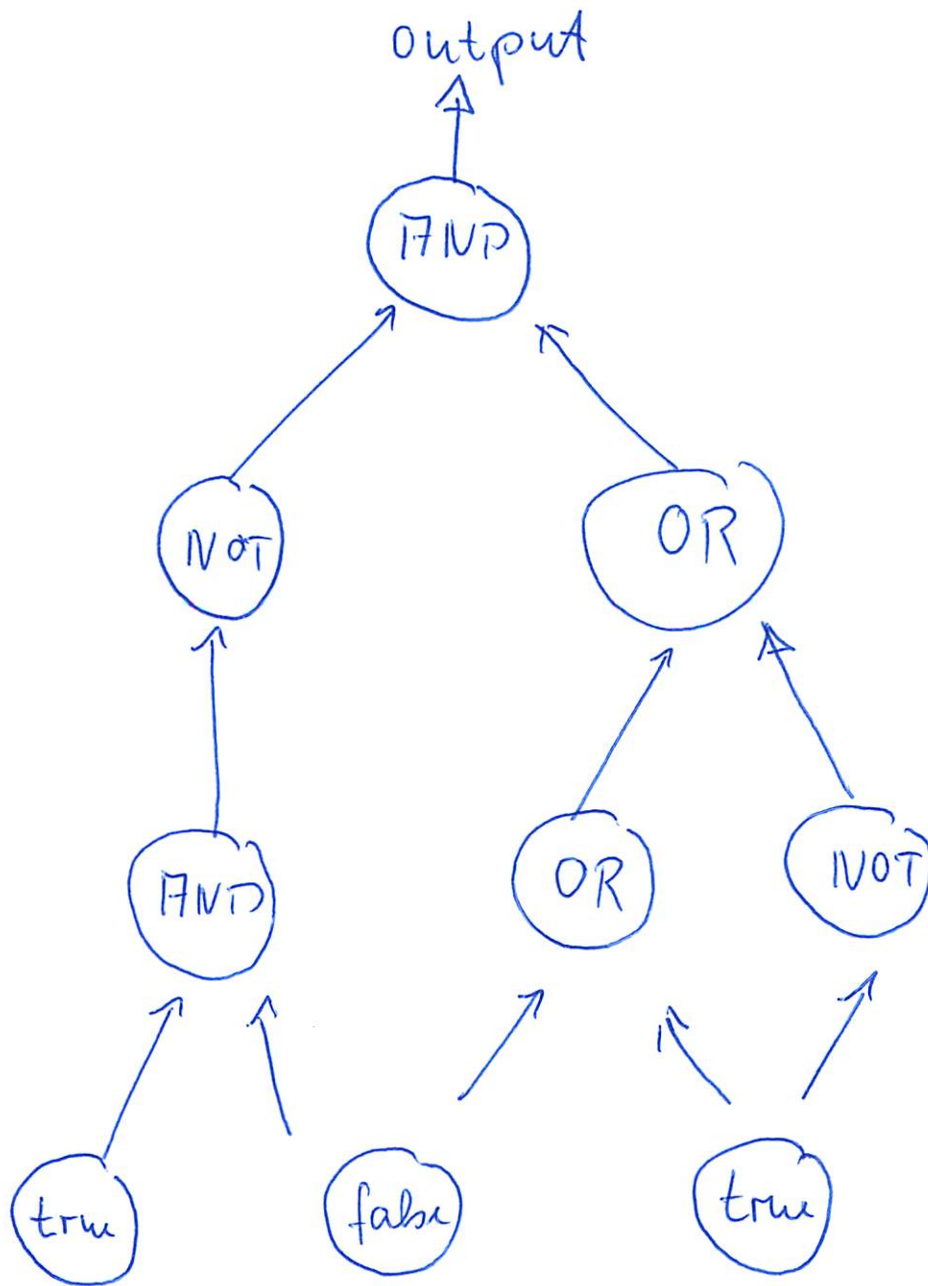
Behaves well in practice.

Postscript: circuit evaluation

"The ultimate application"

Given a boolean circuit, i.e., a dag of gates of the following types.

- INPUT GATES
  - indegree = 0
  - values < true / false
- AND GATES } indegree 2
- OR GATES } indegree 2
- NOT GATES indegree 1
- One OUTPUT GATE
  which is a designated gate


output
AND
NOT
OR
AND
OR
NOT
true
false
true

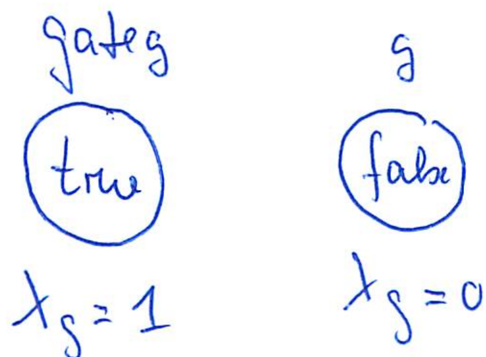
CIRCUIT Value Problem:

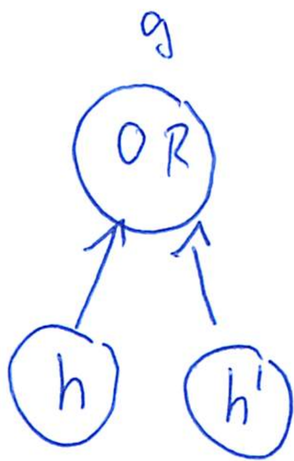
when laws of Boolean logic are applied to the gates in topological order, does the output gate "report" true?

$\hookrightarrow$ LP

For each gate g, create variable x_g, with constraints $0 \le x_g \le 1$.

add additional constraints for each gate according to:

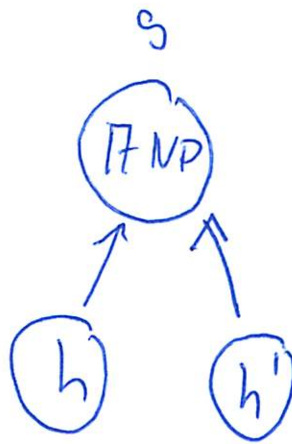




$x_g \geq x_h$

$x_g \geq x_{h'}$

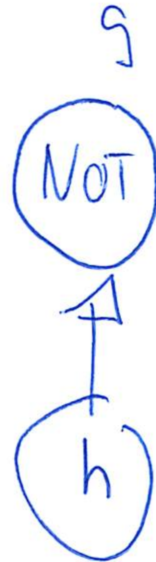
$x_g \leq x_h + x_{h'}$



$x_g \leq x_h$

$x_g \leq x_{h'}$

$x_g \geq x_h + x_{h'} - 1$



$x_g = 1 - x_h$

No max or min function required.

The circuit value problem is (in a sense) the most general problem solvable in polynomial time.

Any algorithm will be executed on gates.

If an algorithm runs in polynomial time, take the computer's gates copy them a polynomial times and use the gate values layer by layer.

This together with:
CIRCUIT value reduces to LP

$\Rightarrow$ All problems that can be solved in polynomial time do.

In the final chapter of this course, NP-completeness, we will see that many "hard" problems reduce (in a similar way) to INTEGER PROGRAMMING.
(your textbook calls Integer Programming the difficult twin of LP.)

Chapter 8

NP-complete Problems

Towards the biggest open problem in Computer Science !

We have seen polynomial-time solutions for problems that could easily be solved via an exponential algorithm.

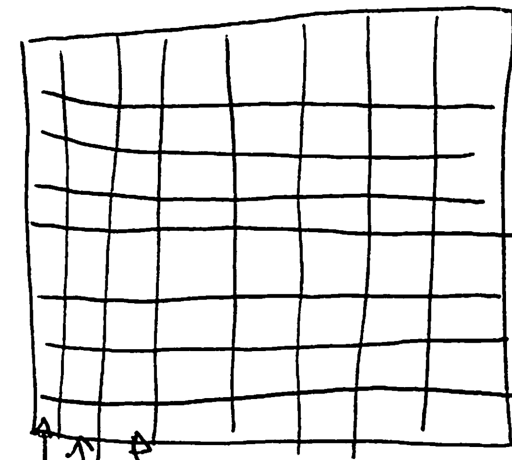
E.g., Dijkstra's shortest path algorithm
min spanning tree
max increasing subsequences

These are polynomial-time solvable.
But e.g. - there are an exponential number of paths between a source and a destination node
- an exponential # of spanning trees
- subsequences

Is this always possible?

Is there always a clever technique like dynamic programming, LP, ... to get a polynomial-time algorithm for any computable problem?

How bad is exponential?



chessboard
Brahmin Sissa

always double the previous #

1 grain, 2 grains, 4 grains of rice

$2^0, 2^1, 2^2, \ldots, 2^{64-1}$

$2^{64-1} = 18,446,744,073,709,551,615$

quintillions, quadrillions, trillions, billions, millions

Thus far, we designed algorithm whose run-time is polynomial,

i.e., $O(n^k)$ for some constant k and input size n.

The class of all problems for which a polynomial time algorithm exists is called:

P

Now, we look at problems for which no polynomial-time algorithm has been discovered as yet.

However, nobody has proven that no such algorithm exists either for these problems.

We first study a particular class of problems.

$\mathcal{NP}$

defined informally as those problems that can be verified in polynomial time.

Verified means that a certificate of a solution can be verified to be correct in polynomial time in the input size to the problem.

(There is a little bit more to the NP-class, but good enough for now.)

Clearly,

$$P \subseteq NP$$

Take a problem in P, solve it in polynomial time even without a certificate.

<u>OPEN:</u> $P \stackrel{?}{=} NP$

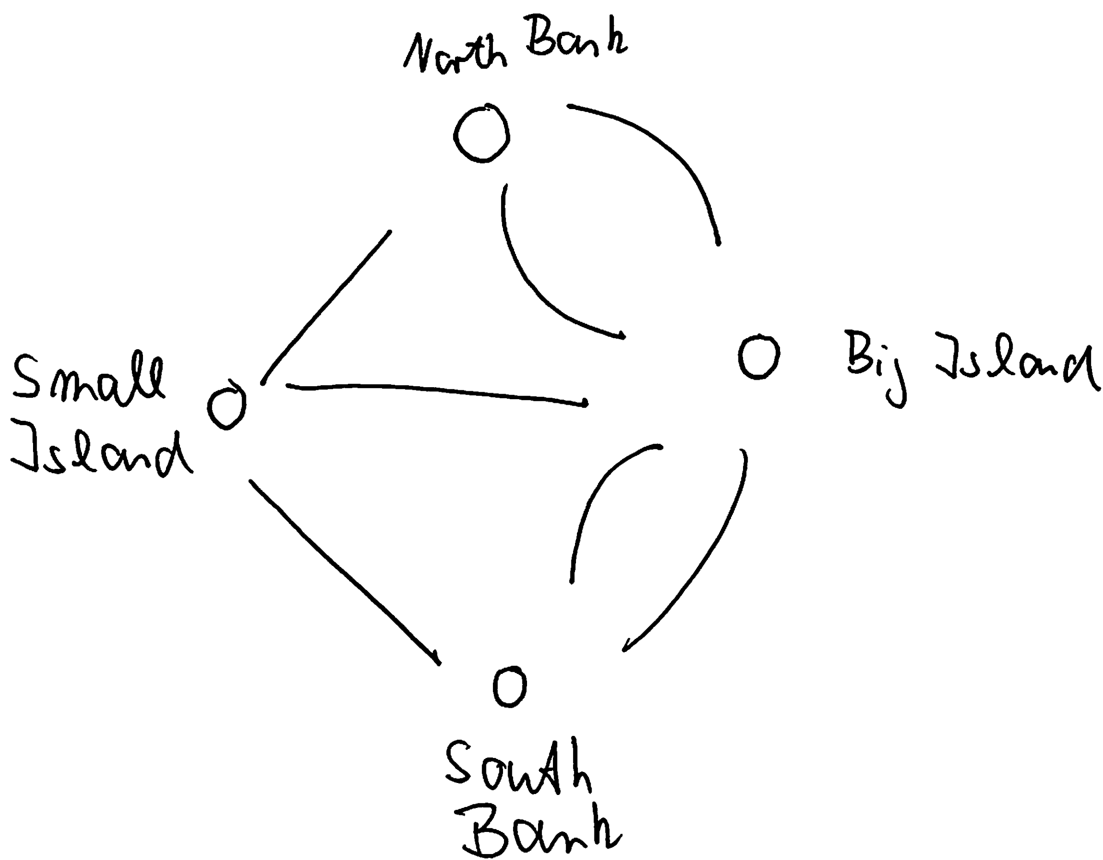
biggest CS open problem.

Examples:

P	NP
① **Shortest path** in a (weighted) directed graph with or without negative weights.	**Longest simple path** (even determining whether a graph contains a simple path with at least a given # of edges is NP-complete)
② **Eulerian Tour** An Eulerian tour of a connected, directed graph $G=(V,E)$ is a cycle that traverses each edge of G exactly once. (Vertices may be visited more than once.)	**Hamiltonian cycle** (your book: Rudrata cycle) A Hamiltonian cycle of a directed graph $G=(V,E)$ is a simple cycle that contains each vertex in G.

Leonhard Euler

Swiss Mathematician studied a problem called, Königsberg Bridge Problem, 1735, which lead to Eulerian (or Euler) tours.



edges are bridge | vertices land

Can one find a start location to visit on a tour each bridge exactly once?

Note: the graph has multiple edges between nodes (or vertices).

The Euler tour through Königsberg can be stated as:
Can the graph be drawn without lifting the pencil from the paper (and drawing edges twice)

Euler, being a mathematician, solved a more general problem than just for the Königsberg problem instance,

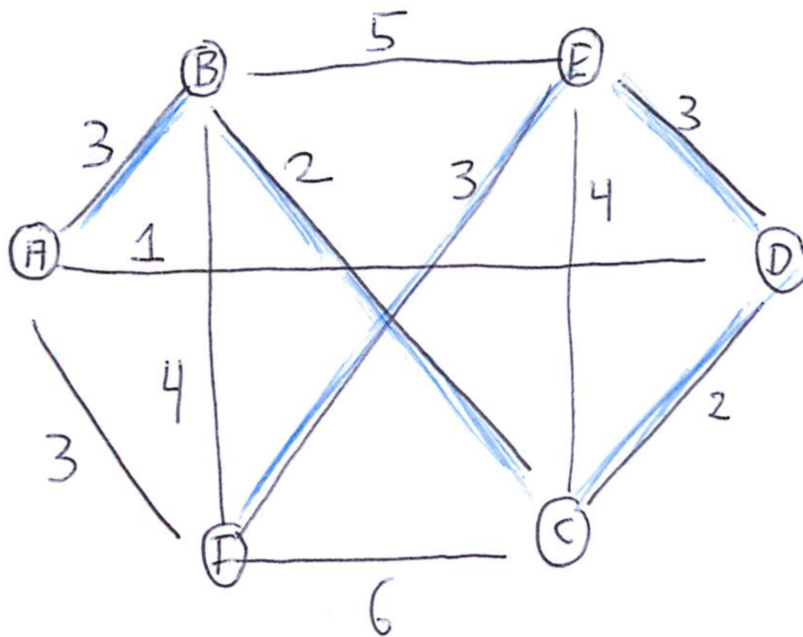
Theorem: There is an Eulerian tour for G

$\Leftrightarrow$ (1) G is connected
(2) every vertex, with the possible exception of two vertices (start and end) has even degree.

In the Königsberg Bridge Problem all 4 vertices have odd degree $\Rightarrow$ no solution

P	NP
(3) **Linear Programming**	**Integer Linear Programming**
We saw that, although the Simplex Method was not polynomial-time, there were polynomial-time solutions.	Here we require the variables of the Linear Program solutions to be integers.
(4) **Minimum Spanning Tree**	**Traveling Salesman**
Find a tree (spanning G) of minimum cost	(the problem remains to be called salesman not salesperson, sorry) Here, we find a special tree, i.e., a tour through every vertex exactly once of cost b, for a Graph in which every pair of vertices has a connecting edge of a particular cost. The cost of the tour is the sum of the individual costs per edge.

Note, we are not asking for the minimum cost. (see later for a discussion on this)



Optimal tour: A, B, C, D, E, F

Cost: 13
(length)

There are $(n-1)!$ tours.

Checking if a tour cost $\leq b$
is clearly polynomial time.
(in fact, it is linear)
So, it is in $\mathcal{NP}$.

A boolean formula is in
Conjunctive **N**ormal **F**orm
CNF
if it is given as a collection of clauses (parentheses), each consisting of disjunctions (logical or, $\vee$) of several literals.

Literals: boolean variables or negations of boolean variables.

ex: $(x \vee y \vee z)(x \vee \bar{y})(z \vee \bar{x})(y \vee \bar{z})(\bar{x} \vee \bar{y} \vee \bar{z})$

literal → x; or → $\vee$; (and $\wedge$) → between clauses; negated → $\bar{z}$

say x = false y = true z = true then
the expression is false

The question is: Satisfiability

Is there a true/false assignment to the literals so that the boolean formula given in CNF is true?

For this example:

Here, no

Why?

$(x \vee \bar{y})(z \vee \bar{x})(y \vee \bar{z})$

force all three to have the same value

either $x = y = z =$ true

or $x = y = z =$ false

but, then

$(x \vee y \vee z)(\bar{x} \vee \bar{y} \vee \bar{z}) =$ false

$\Rightarrow$ the entire formula = false

We call the problem CNF - Satisfiability.

When each clause has exactly k clauses it is called

k - CNF Satisfiability

For $k=2$, we obtain

2-CNF satisfiability

for $k=3$, we obtain

3-CNF satisfiability.

---

ex: $(x_1 \vee \bar{x}_2)(\bar{x}_1 \vee x_3)(\bar{x}_2 \vee \bar{x}_3)$

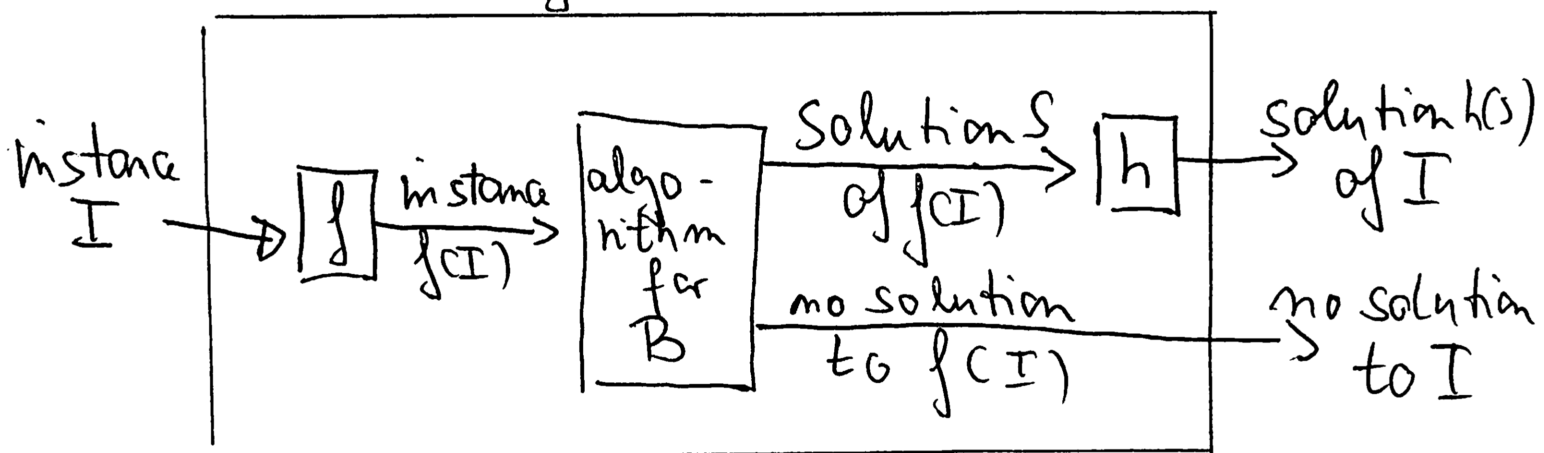
is 2-CNF

$(x_1 \vee \bar{x}_2 \vee x_3)(\bar{x}_1 \vee x_2 \vee x_3)$

is 3-CNF

P	NP
2 - CNF Satisfiability	3 - CNF Satisfiability

A *reduction* from a search problem A to a search problem B is a polynomial time algorithm that transforms any instance I of A into an instance $f(I)$ of B, together with a polynomial time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I.



Let us look at 2 problems

Hamiltonian (s,t)-path and
Hamiltonian Cycle problem

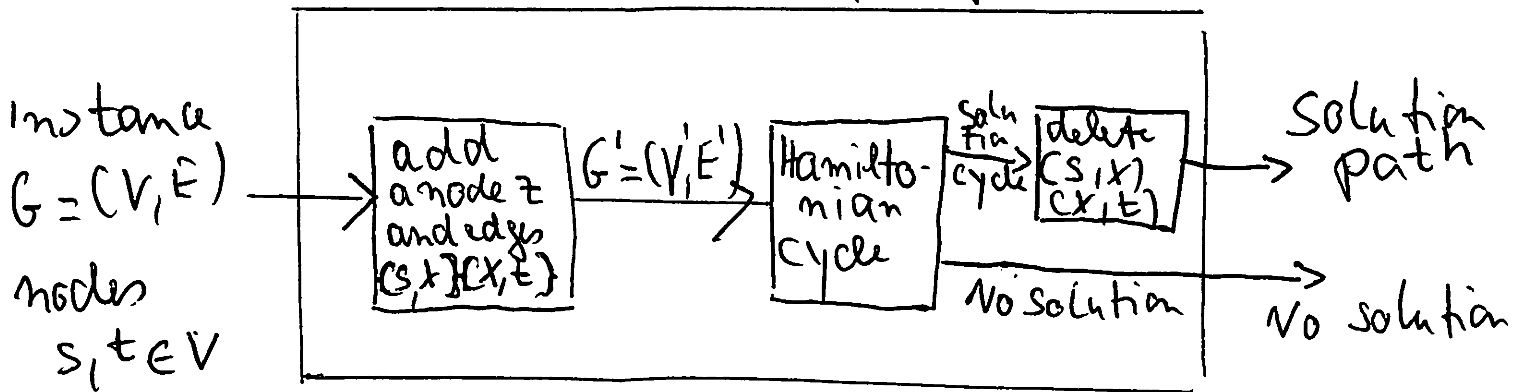
Hamiltonian (s,t)-path problem

Given a graph and 2 vertices s, t.
$\exists$ then a path starting at s
and ending at t that goes
through each vertex exactly once?

Is it possible that Hamiltonian
cycle is easier than Hamilton
(s,t) - path problem?

NO

Hamiltonian (s,t)-path



Case

1. The instance of Hamiltonian Cycle has a Solution

Vertex x has exactly 2 neighbours s and t.
Any Hamiltonian cycle c' in G' must therefore traverse $\{x, s\}$ and $\{x, t\}$
(undirected $\{x, s\} = \{s, x\}$)

The remainder from s to t must traverse all other vertices in G'.
So deleting $\{x, s\}$ and $\{x, t\}$ from the cycle leaves an Hamiltonian path for G connecting s to t.

2. Hamiltonian Cycle does not exist

It is easy to show that in this case the original Hamiltonian path problem between s and t cannot have a solution either.
Assume it did have a solution.
Then just add $\{x, s\}$ and $\{t, x\}$ and we get a Hamiltonian cycle for G'.
A contradiction.

Important!
we need to check that pre and post-processing can be done in polynomial time!
Here, easy as we just add 2 edges or delete 2 edges.

Reduction from A to B

$$A \rightarrow B$$

A search problem is **NP-complete** if all other search problems reduce to it.

These problems are in some sense the hardest problems in NP.

Assume A reduces to B

A → B

"the difficulty" flows in the direction of the arrow.

while
"efficient algorithms" move the other way.

---

if [A → B] and [B → C]

then [A → C]

"composition"
of the input of output functions

<u>Fact</u>: Once we know that a problem, A, is NP-complete, we can use it to prove that another search problem, B, is also NP-complete by reducing A to B.

Follows from composition.

Formal definitions: (p.244 your book)

P: set of search problems that have a polynomial time solution.

NP: Set of search problems that can be

(1) found

and

(2) verified

in polynomial time by a non-deterministic (sort of) algorithm.

[Guesses correctly at every time.]

I do not like the definition. Original def. via formal languages, as decision problems $w \in L$ or $w \notin L$, L some language.

Decision problems vs. optimization problems

In optimization problems, feasible solutions have values and we want to determine the optimal value.

In decision problems, answers are yes or no.

Usually, we can state an optimization problems as a decision problem.

E.g., is there a shortest path with at most k edges, between s and t.
(k here is an integer)
(Q: Find shortest path with fewest edges.)

If a decision Problem is hard
it related optimization problem
is then hard too.

Reductions cont'd

3SAT → Independent Set

↑

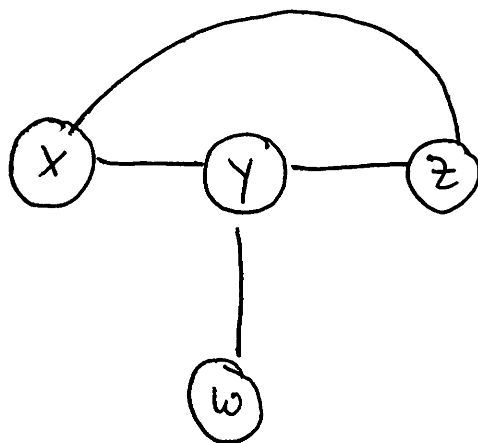
$(\bar{x} \vee y \vee \bar{z})(x \vee \bar{y} \vee z)(x \vee y \vee z)(\bar{x} \vee \bar{y})$

Independent Set Problem:

input: a graph G and a number g

output: find a set of g pairwise non-adjacent vertices in G or report that no such set exists

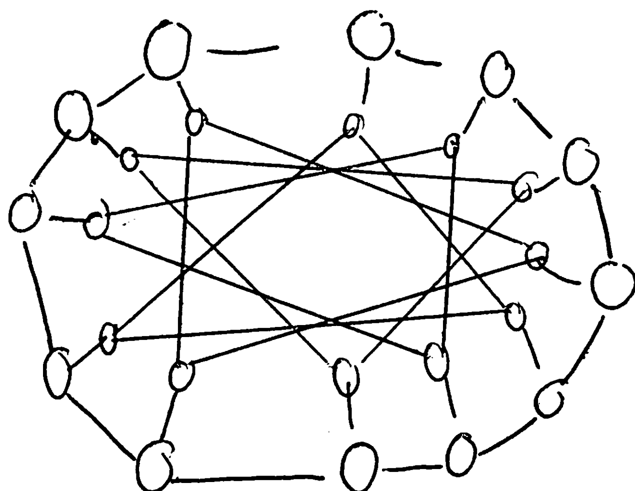
Ex:



$g = 2$

indep. set $\{x, w\}$

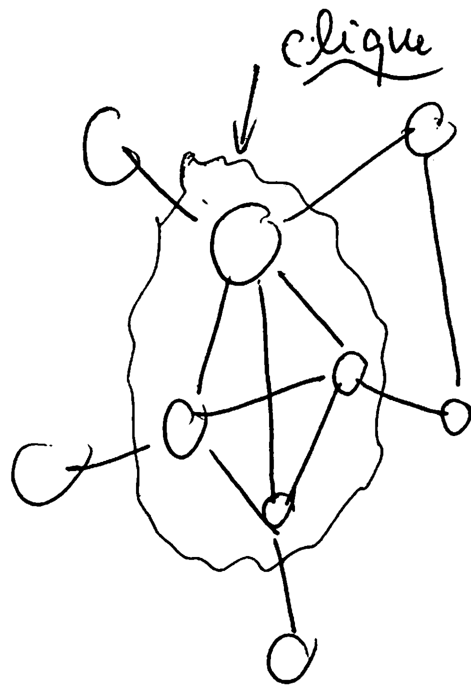
The optimization problem corresponding to this is maximum independent set



What is the size of the maximum independent set? 9

Find them!

A clique in a graph G is a subset of G's vertices in which every two vertices are connected by an edge in G.



A set in G is independent $\iff$ it forms a clique in the complement of G.

Complement: if $(x,y) \in E$ then $(x,y) \notin E$

$G = V, E \quad G^c = (V^c, E^c)$ if $(x,y) \notin E$ then $(x,y) \in E$

8.2?

Sufficiently large graphs with no large cliques have large independent sets
(→ Ramsey theory)

Cliques are useful as they have very strong (optimal) connectivity.

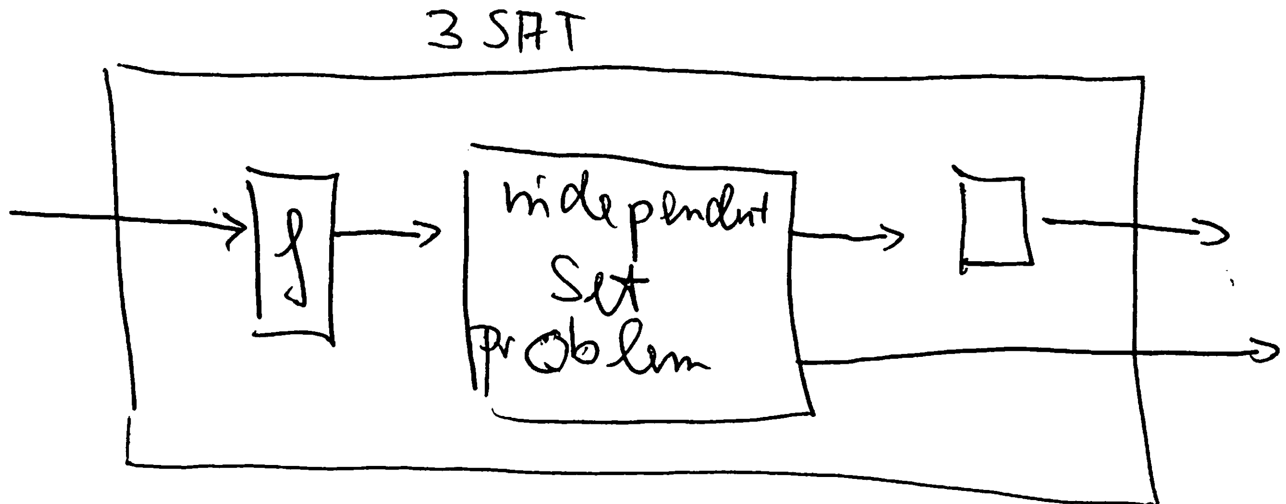
---

back to the reduction

3SAT → Independent Set

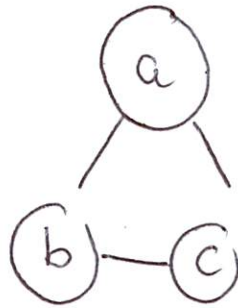
So, we need to find [f]

[f]: Instance I (3-SAT) into an instance f(I) for independent set



Construction: f

For each clause: $(a \vee b \vee c)$
construct a triangle



vertices: a, b, c

edges: a - b
a - c
b - c

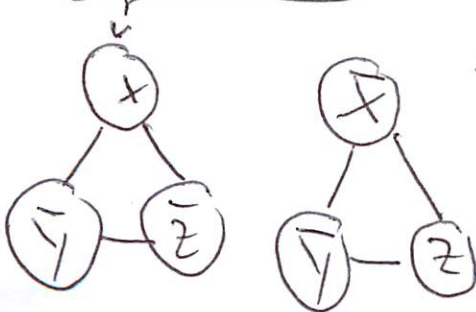
(a, b, c can be negated e.g. $b = \bar{y}$)

For each literal: x

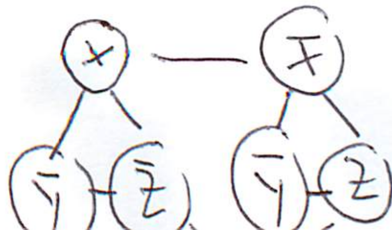
connect $x - \bar{x}$ in all occurances

⇒ The construction take polynomial time.

Ex: $(x \vee \bar{y} \vee \bar{z})(\bar{x} \vee \bar{y} \vee z)$

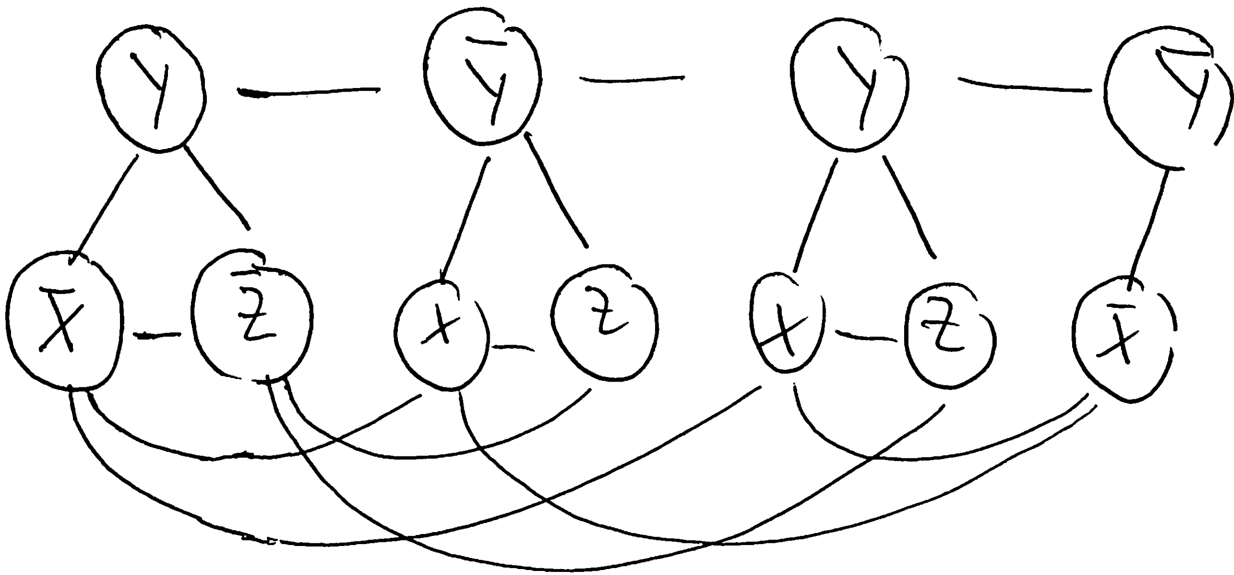


now add
$x - \bar{x}$
$y - \bar{y}$
$z - \bar{z}$



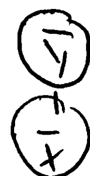
Ex:

$(\bar{x} \vee y \vee \bar{z})(x \vee \bar{y} \vee z)(x \vee y \vee z)(\bar{x} \vee \bar{y} \vee z)$



(Note, your book allows clauses with 2-literals)

The construction is then simply an edge $(\bar{y} \vee \bar{x})$



The goal g = # of clauses

in our example $g = 4$.

The "output", i.e. second conversion function h.

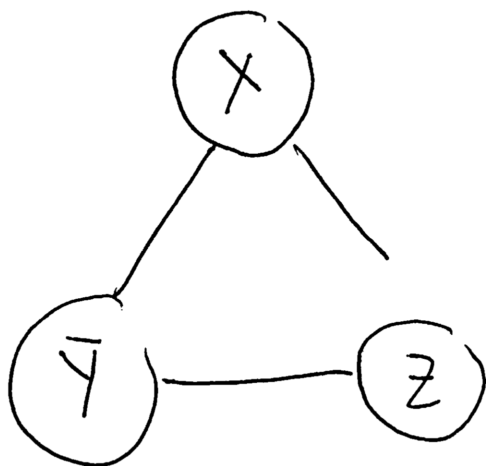
Because in G (x) — $(\bar{x})$ for any literal x, and any occurrance of $x, \bar{x}$,

$\Rightarrow$ either x or $\bar{x}$ is in the independent set, S, not both.

---

h : assign: x = true if $x \in S$
x = false if $\bar{x} \in S$

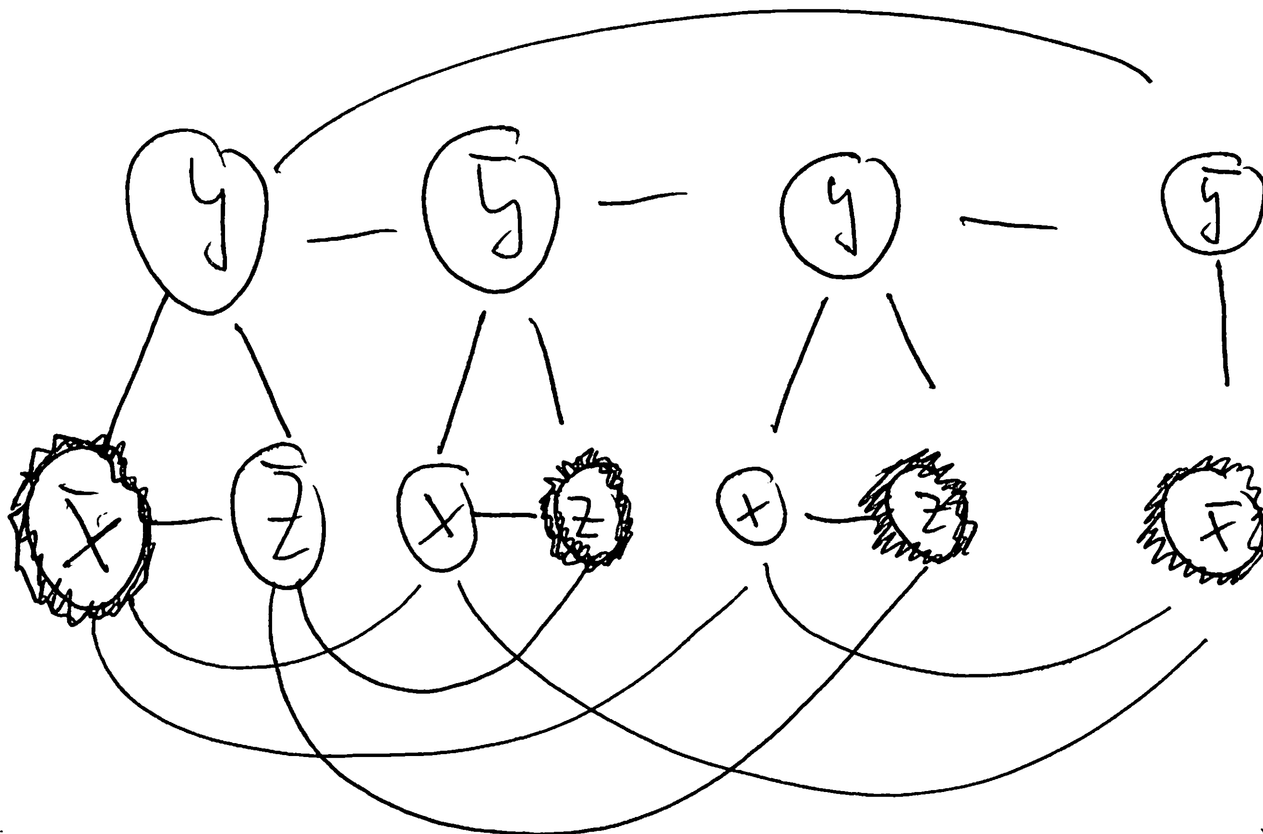
Why does this work?



in an independent set, we can select at most one vertex per triangle. But, since g = # of clauses = # triangles we must select one per triangle

We cannot choose both x and $\bar{x}$.

Choose the vertices as ind.
Set as indicated



(in each triangle (or $\overset{\circ}{\underset{\circ}{|}}$) exactly one vertex is chosen.
No literal can be chosen as both negated and unnegated.)

chosen $\bar{x}$: $\Rightarrow$ x = false

z $\Rightarrow$ z = true

y arbitrary

Fact If Graph G has no ind. set of size g, then the Boolean formula I is unsatisfiable.

Proof: Assume the contrary, i.e. I is satisfiable. Then, for each clause pick any literal whose value under the satisfying assignment is true.
Clearly, in each clause there is one.
They must be an independent set. because we never pick a literal and its negation. ◇

Independent Set -> Clique

We observed already

Lemma: A set of nodes S is an independent set of G if and only if S is a clique of G^c. (G^c is the complement Graph of G)

Thus, the above reduction is easy.

SAT -> 3-SAT

This is a reduction from a general problem to a special case of itself. So, the problem is just as hard.

How do we shorten clauses with more than 3 literals?

Here is the "trick":

$(a_1 \vee a_2 \vee a_3 \vee \dots \vee a_k)$ is satisfied $\Leftrightarrow$ there is a setting of the y_i's for which

$(a_1 \vee a_2 \vee y_1)(\bar{y}_1 \vee a_3 \vee y_2) \dots (\bar{y}_{k-3} \vee a_{k-1} \vee a_k)$ are all satisfied

Why is this true?

(1) Assume all clauses on right are satisfied

<u>Claim</u>: at least one of $a_1, \dots, a_k$ is true.

Assume otherwise. Then,

$(a_1 \vee a_2 \vee y_1) = t$ forces $y_1 = \text{true}$

$(\bar{y}_1 \vee a_3 \vee y_2) = t$ forces $y_2 = t$

($\bar{y}_1$ ← false, a_3 ← false)

$\vdots$ $\qquad\qquad\qquad\qquad \vdots$

$y_{k-3} = t$

$(\bar{y}_{k-3} \vee a_{k-1} \vee a_k) = \text{false}$ ↯

Thus claim holds. $\Rightarrow$ one of $a_1, \dots, a_k$ is true $\Rightarrow (a_1 \vee \dots \vee a_k) = \text{true}$.

(2) If $(a_1 \vee a_2 \vee \dots a_k)$ is satisfied $\Rightarrow$ one of $a_1, \dots, a_k = t$, say $a_i = t$

Then set $y_1, \dots, y_{i-2} = t$

$y_{i-1}, \dots, y_{k-3} = f$

$(a_1 \vee a_2 \vee y_1) = t$

$\uparrow t$ (under y_1)

$\vdots$

$(\bar{y}_{i-3} \vee a_{i-1} \vee y_{i-2}) = t$

$\uparrow t$ (under y_{i-2})

$(\bar{y}_{i-2} \vee a_i \vee y_{i-1}) = t$

$\uparrow t$ (under a_i)

$(\bar{y}_{i-1} \vee a_{i+1} \vee y_i) = t$

$\uparrow t$ (under $\bar{y}_{i-1}$)

$\vdots$

$(\bar{y}_{k-3} \vee a_{k-1} \vee a_k) = t$

$\uparrow t$ (under $\bar{y}_{k-3}$)

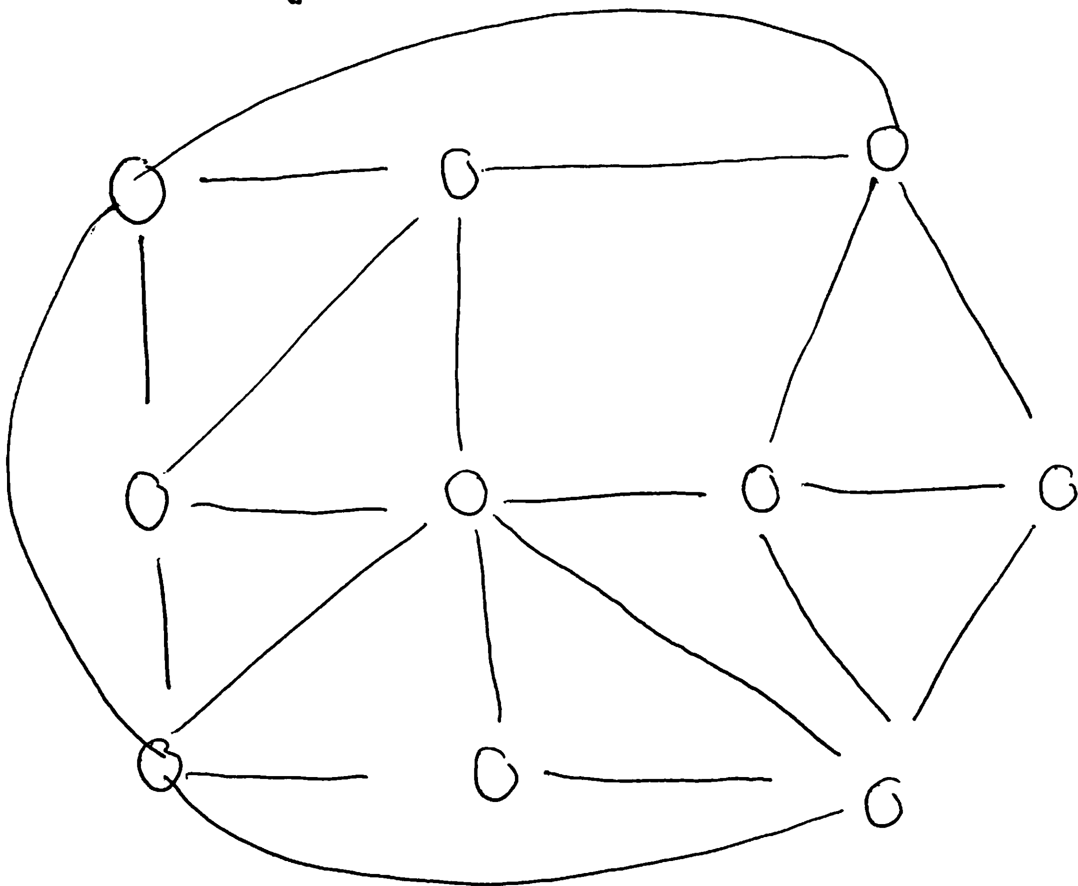
So what are $\boxed{f}$ and $\boxed{h}$?

Is it polynomial time for f, h ?

Vertex Cover Problem

Input: graph and budget b

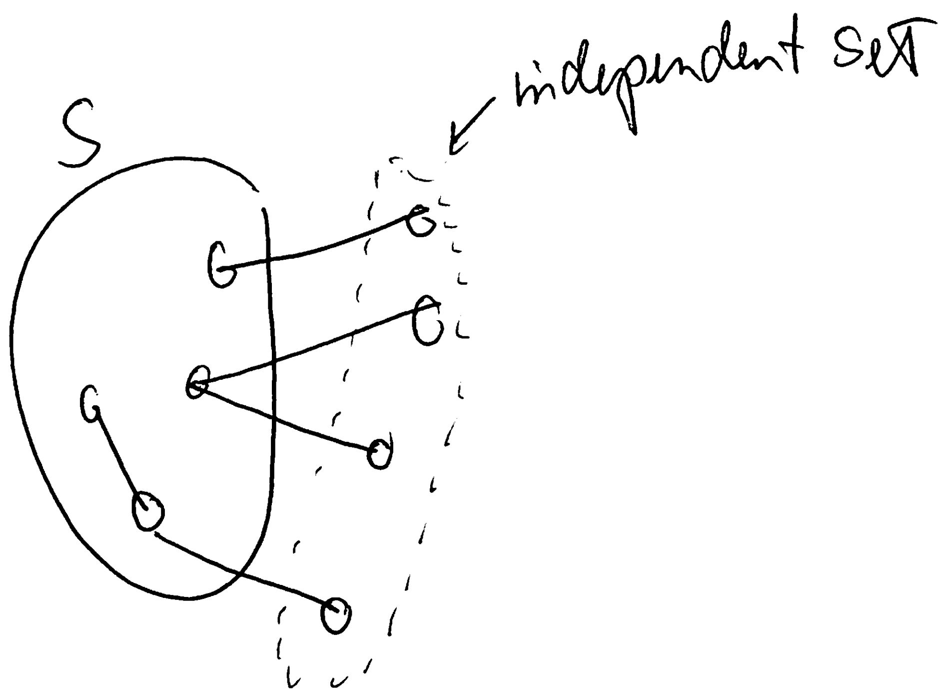
Task. Find b vertices that touch every edge.



Independent Set → Vertex Cover

Observe,

S is a vertex cover of G $\iff$
V − S is an independent set of G.



To solve the independent set problem for (G, g), look for a vertex cover of G with $|V| - g$ nodes.

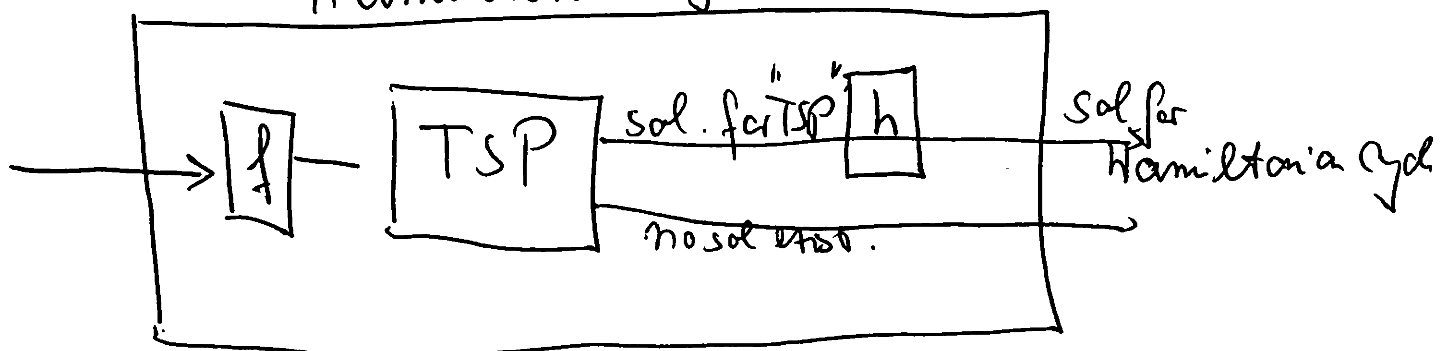
If such a cover exists $\Rightarrow$ take all nodes NOT in it.

Otherwise, G cannot have an ind. set of size g.

Reductions cont'd

Hamiltonian Cycle → TSP

Given a graph $G=(V,E)$ construct an instance of a TSP:



[f] : we construct a special TSP

$V \to$ vertices of the TSP

distances: $(u,v) \in E \Rightarrow dist(u,v) = 1$

$(u,v) \notin E \Rightarrow dist(u,v) = 1 + \alpha$ for some $\alpha \geq 1$ to be determined

budget for TSP $:= |V|$ $(|V| = n)$

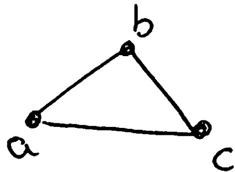
<u>Note</u>: If G has a Hamiltonian cycle then the same cycle can be made a tour within the budget.

If G has no Hamiltonian cycle then there is no solution.
Cheapest TSP cost $\geq n + \alpha$
(at least one edge not in E would have to be used, its cost is $1+\alpha$)

Why α? Interesting special cases!

$\boxed{\alpha = 1}$ $\Rightarrow \text{dist}(u,v) = \begin{cases} 1 & (u,v) \in E \\ 2 & (u,v) \notin E \end{cases}$

in this case, the distances of TSP satisfy the triangle inequality



$d(a,c) \leq d(a,b) + d(b,c)$

This is the case for Euclidean distances

Why satisfied here? if $1 \leq i, j, k \leq 2$ then $i + j \geq k$

TSP can in this case be well <u>approximated</u>.

α large $\Rightarrow$ TSP may not satisfy the triangle inequality

Then: either there is a solution of cost n (or less)
or all its solutions have cost at least $n+\alpha$
but α can be arbitrarily large. Compared to n

$$n \xrightarrow[\text{huge gap}]{} n+\alpha$$

This gap property implies (not shown here) that unless $P = NP$, <u>no approximation for TSP can exist.</u>

ZOE - Problem

Let A be an $m \times n$ matrix

$$A[i,j] = \begin{cases} 0 & \text{or} \\ 1 \end{cases}$$

Find a $(0,1)$-vector $x = (x_1, \dots, x_n)$ so that

$$Ax = \mathbb{1} \qquad \mathbb{1} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

Ex: $A = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ 3 equations $\quad \begin{aligned} x_1 + x_3 &= 1 \\ x_1 + x_2 + x_3 &= 1 \\ x_3 &= 1 \end{aligned}$

Many combinatorial problems can be expressed as a ZOE.

E.g., matching problems.

We will show

ZOE $\rightarrow$ (Rudrata Cycle)
Hamiltonian Cycle

The reduction goes in 2 stages

① ZOE → special Hamiltonian Cycle problem

② remove "special"

---

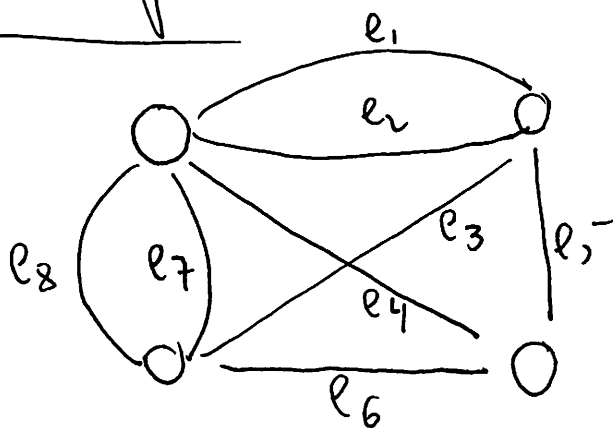
① ZOE → special Hamiltonian Cycle
"pair-edges"

Paired-edges Hamiltonian Cycle

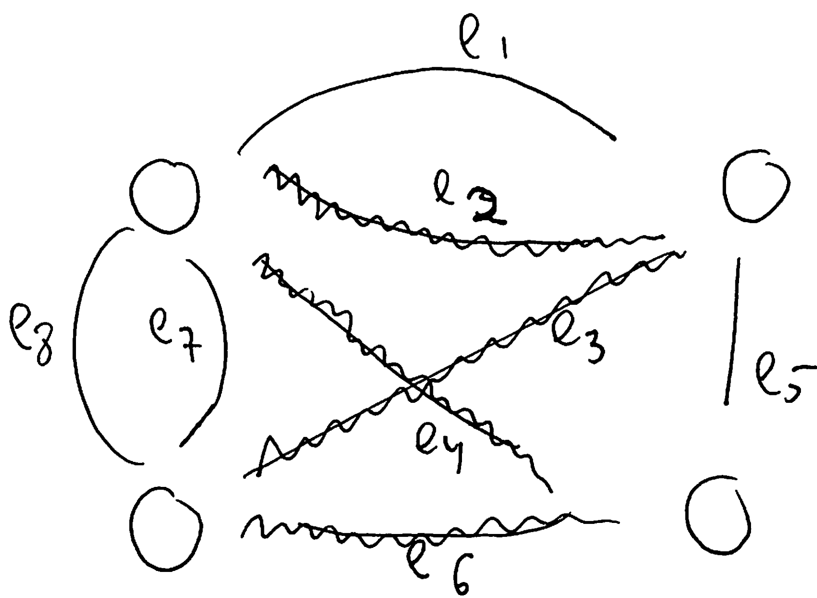
Example: $G = (V, E)$

$C \subseteq E \times E$

$C = \{(e_1, e_3), (e_5, e_6), (e_4, e_5), (e_3, e_7), (e_3, e_8)\}$



for each pair (e_i, e_j) exactly one edge is traversed by the cycle (visiting all nodes)



$(e_1, e_3) \in C$

So, if we pick e_1 we cannot pick e_3, but we must pick one of e_1, e_3

e_2, e_3, e_6, e_4 is a pair-edge Hamiltonian cycle in G.

e_2 is not paired

e_3 is paired with e_1, e_7 and e_8 — they are not in the cycle

e_4 is paired with e_5 ← not on the cycle

e_6 " " " " " " "

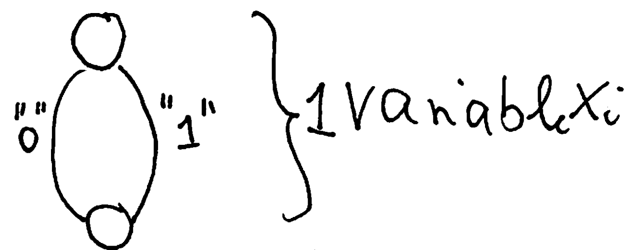
w.r.t. the vertices we have a cycle.

7.OE

instance is a matrix A $(n \times m)$
with 0,1 entries
$\Rightarrow$ m equations in n 0-1 variables

We need to create an input instance for paired Hamiltonian cycle problem.
2 STEPS

STEP 1: For each variable x_i
Create 2 edges



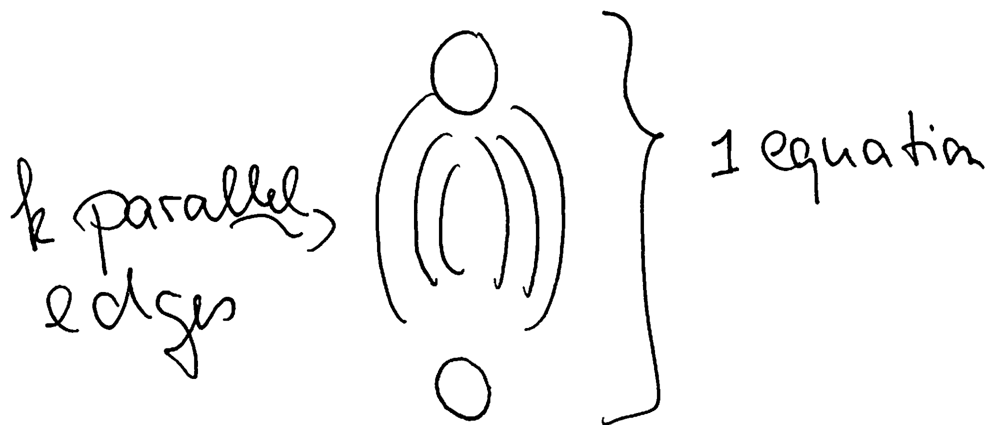
corresponding to
$x_i = 0$ and $x_i = 1$

For each equation

$$x_{j_1} + \dots + x_{j_k} = 1$$

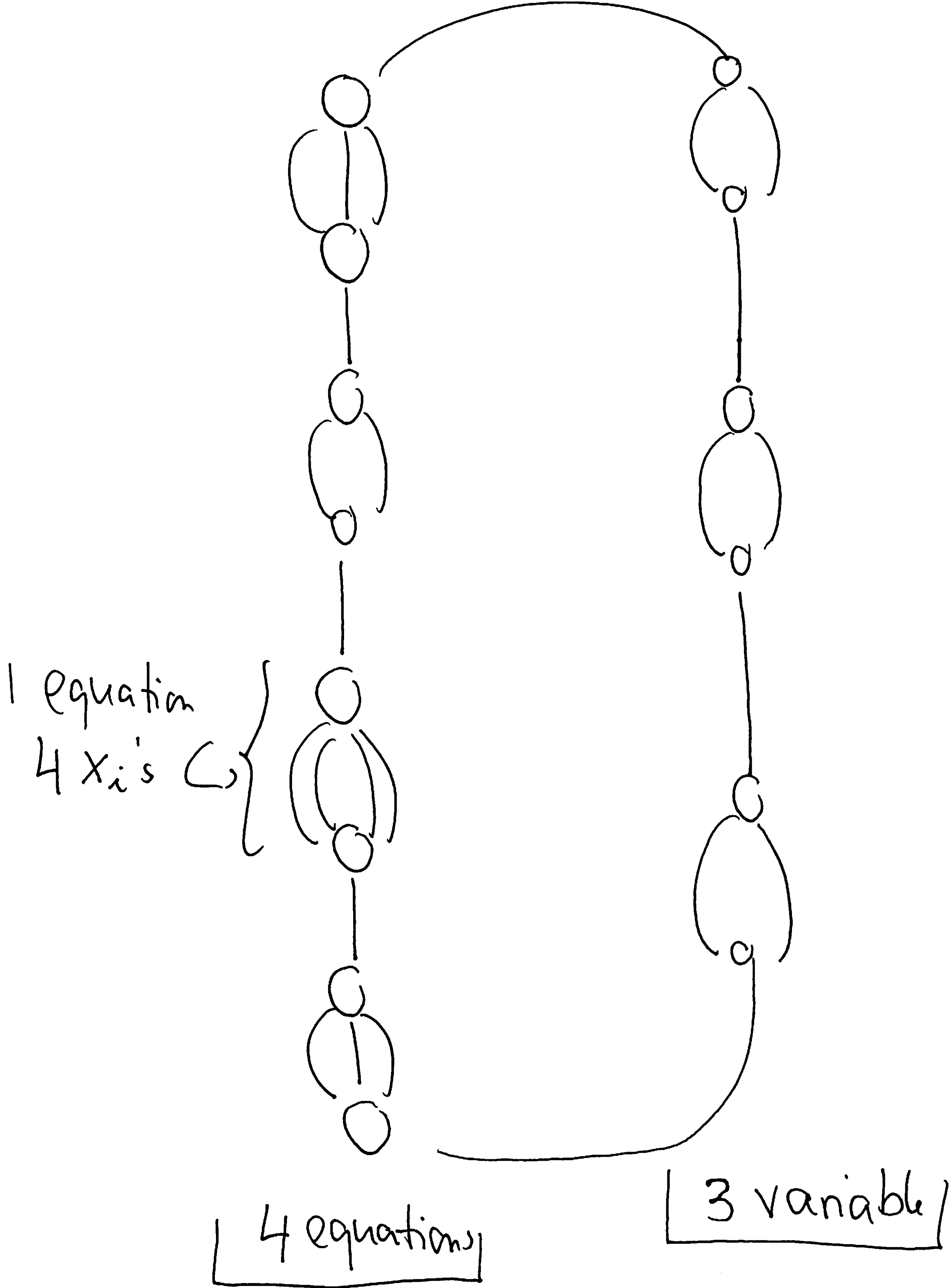
if this involves k variables

Create

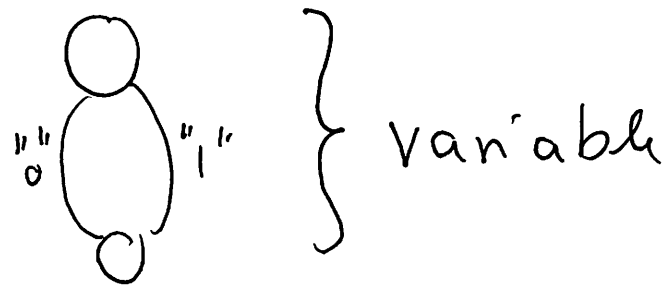


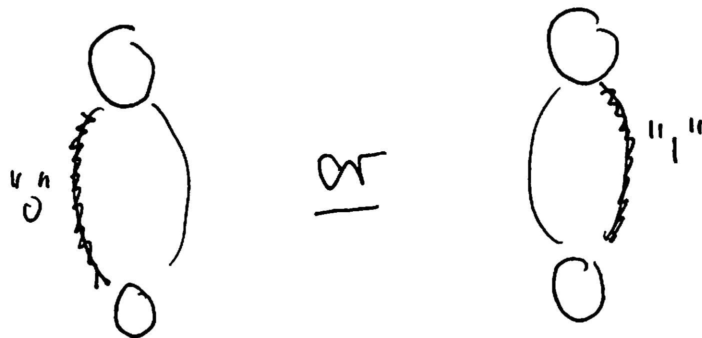
Balance. STEP 1

We created a graph with $m + n$ parallel edges


1 equation
4 x_i's
4 equations
3 variable

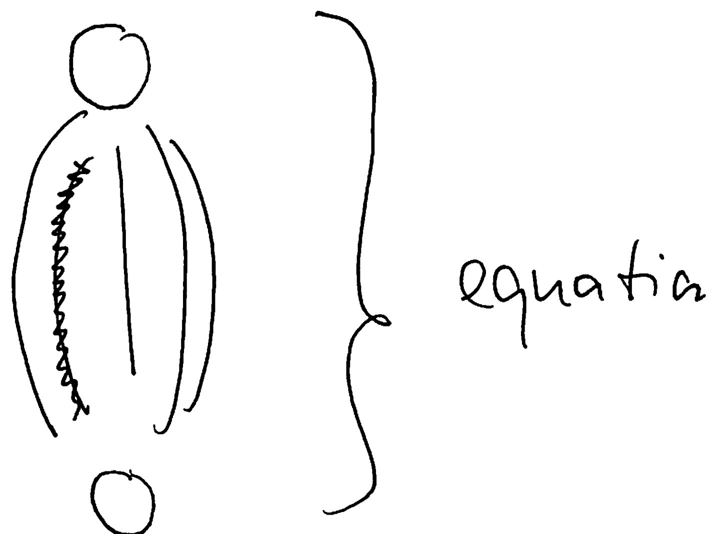
In any Hamiltonian cycle solution the solution uses one of $x_i = \begin{cases} 0 \\ 1 \end{cases}$ or not both





---

and picks one variable per equation



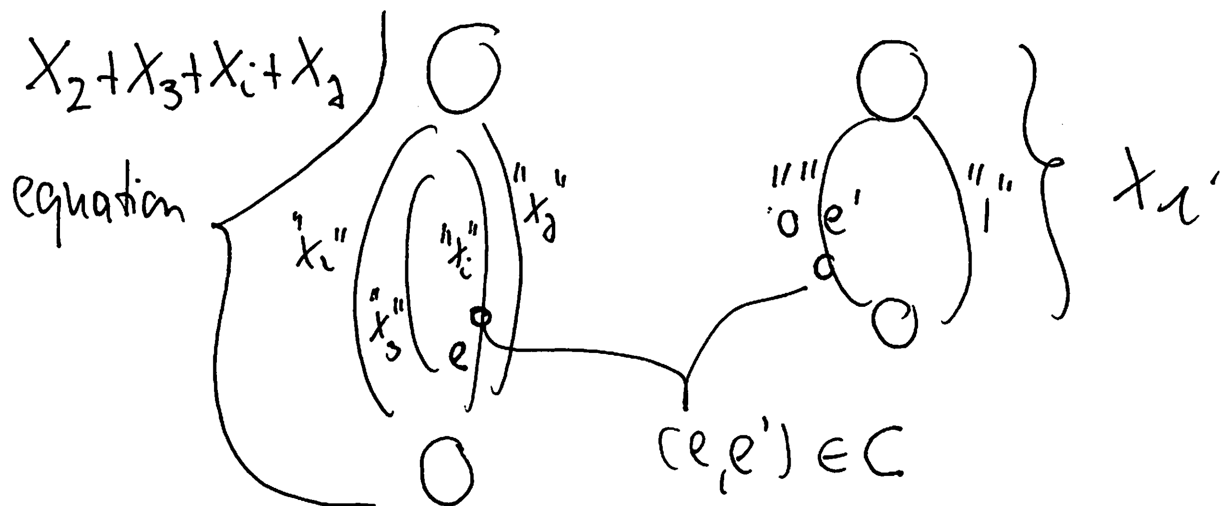
STEP 2:

For every equation and every variable x_i appearing in the equation

add (e, e') to C

where e is edge corresponding to x_i in the equation

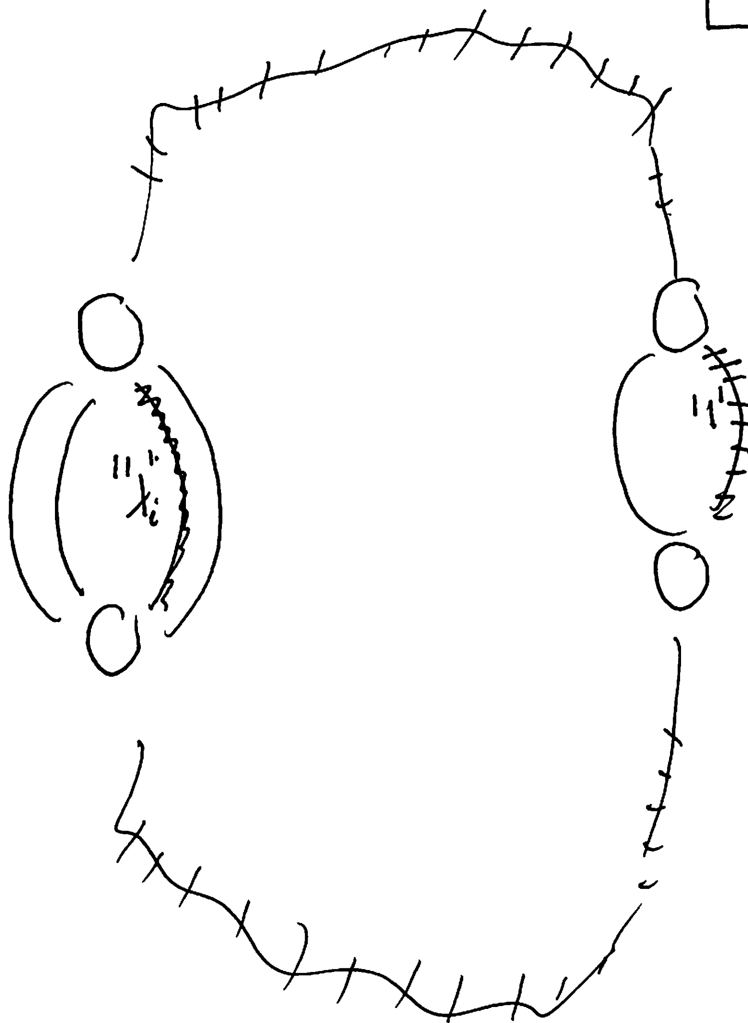
and e' is the edge corresponding to $x_i = 0$



What does this do?

if the cycle picks/uses the edge corresponding to x_i it **cannot** pick ("$x_i = 0$")

$\Rightarrow$ "$x_i = 1$" picked



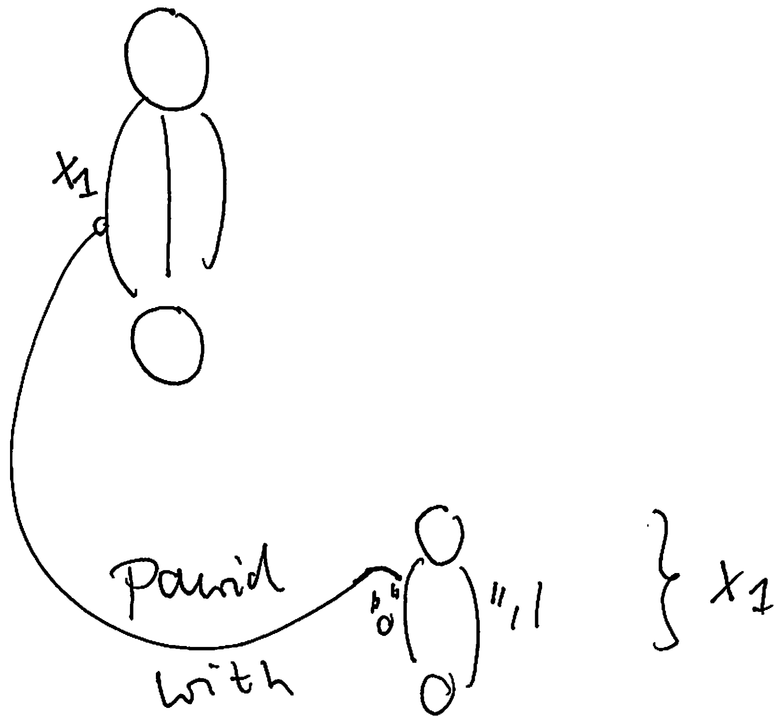
but then in every equation x_i is picked (if it appears) because they are paired with "$x_i = 0$".

On the "left", the equation side, always one edge is picked (i.e, is on the cycle).

For this variable x_i the pairedness ensures that "$x_i = 1$" on the variable side.

and "$x_i = 1$" is then ensuring that x_i is always chosen

Every where

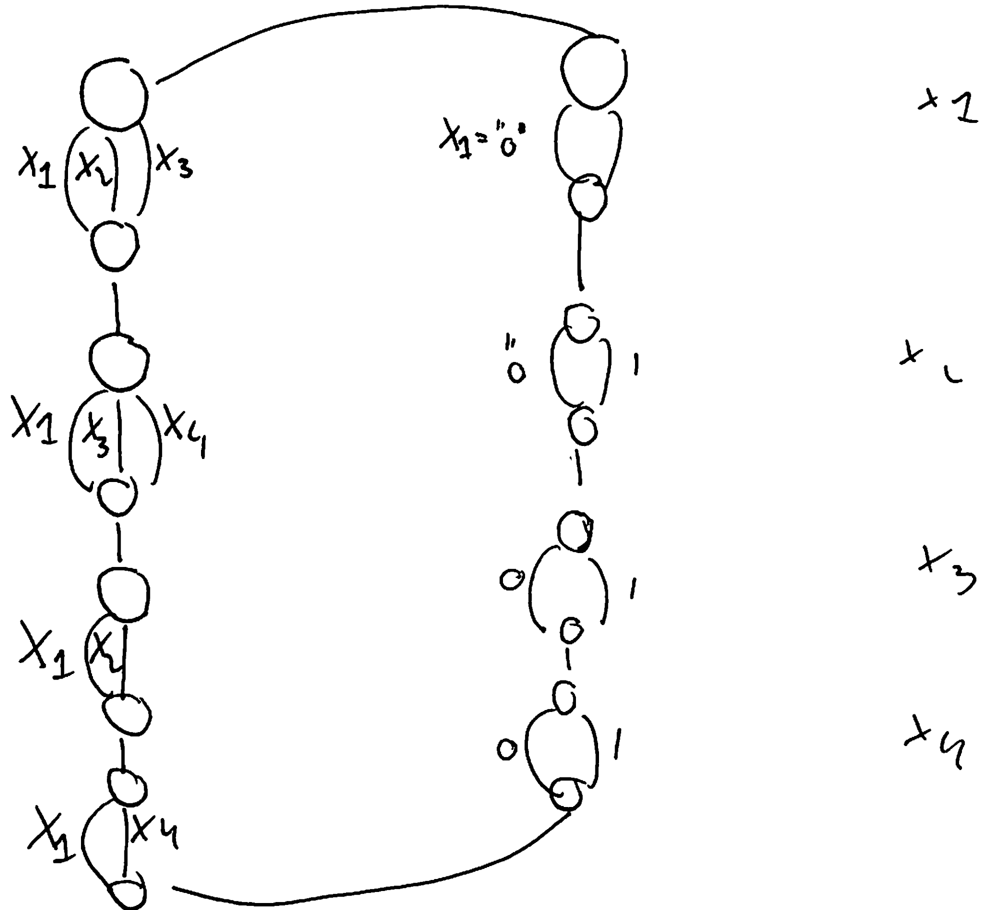


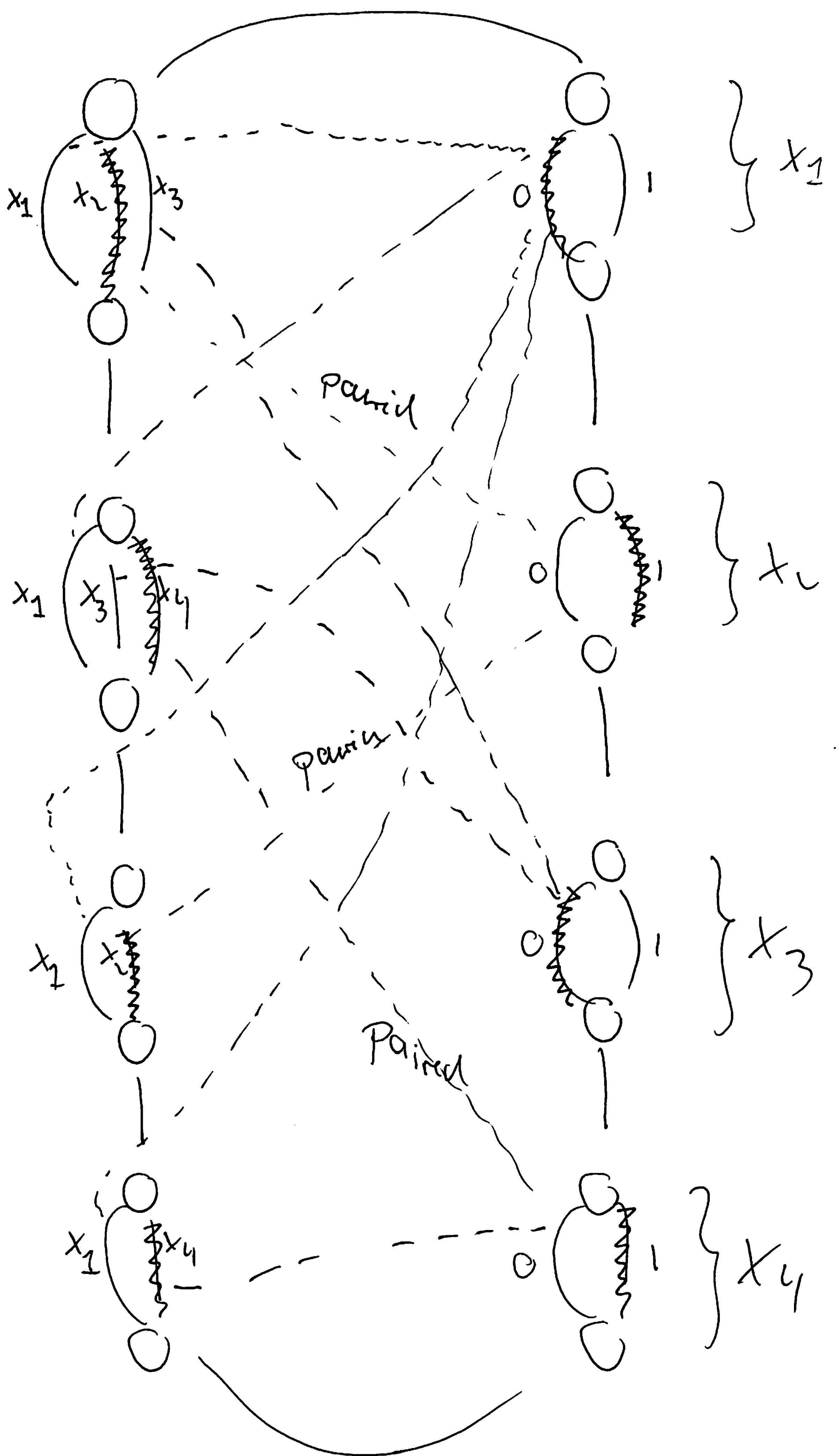
Example:

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = 1$$

Equation

$$x_1 + x_2 + x_3 = 1$$
$$x_1 + x_3 + x_4 = 1$$
$$x_1 + x_2 = 1$$
$$x_1 + x_4 = 1$$





x1
x2
x3
0
1
X1
paired
x1
x3
x4
0
1
x3
0
1
X3
Paired
x1
x4
0
1
X4

$x_2 = 1$ $\quad$ $x_4 = 1$

$x_1 = 0$ $\quad$ $x_3 = 0$

paired Ham. Cycle $\Rightarrow$ correct solution to $\mathbb{A}x = \mathbb{1}$

Also, correct solution to $\mathbb{A}x = \mathbb{1}$ gives a solution to the paired Hamiltonian cycle problem.

---

(2) We wanted

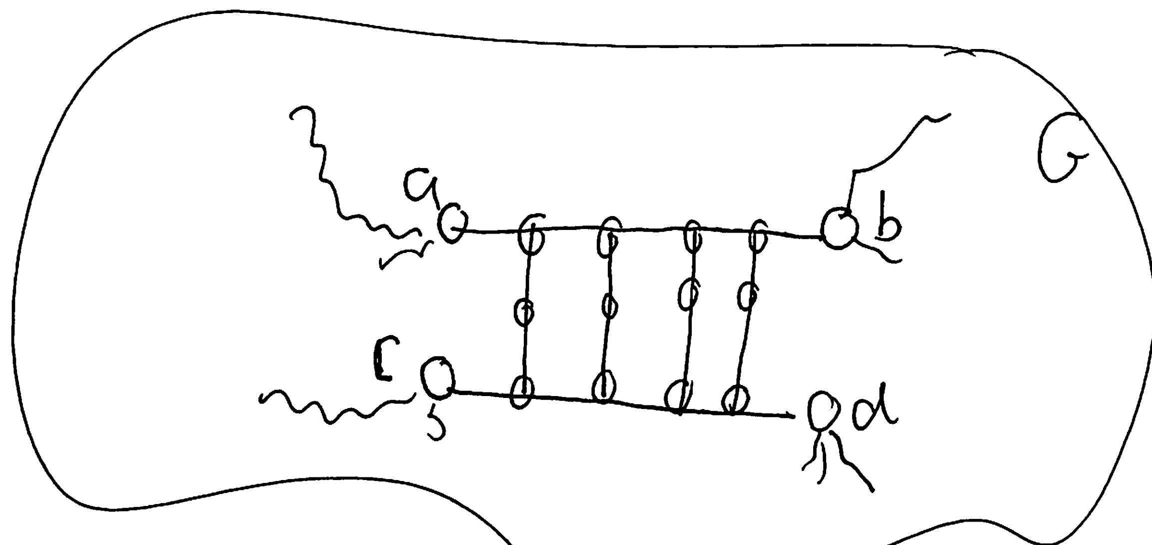
ZOE → Hamiltonian Cycle

So far, we have

ZOE → Paired-edges Hamiltonian Cycle.

Hamiltonian cycle is a special case of paired-edges version for $C = \emptyset$.
But, this is not good enough
(wrong way special case)

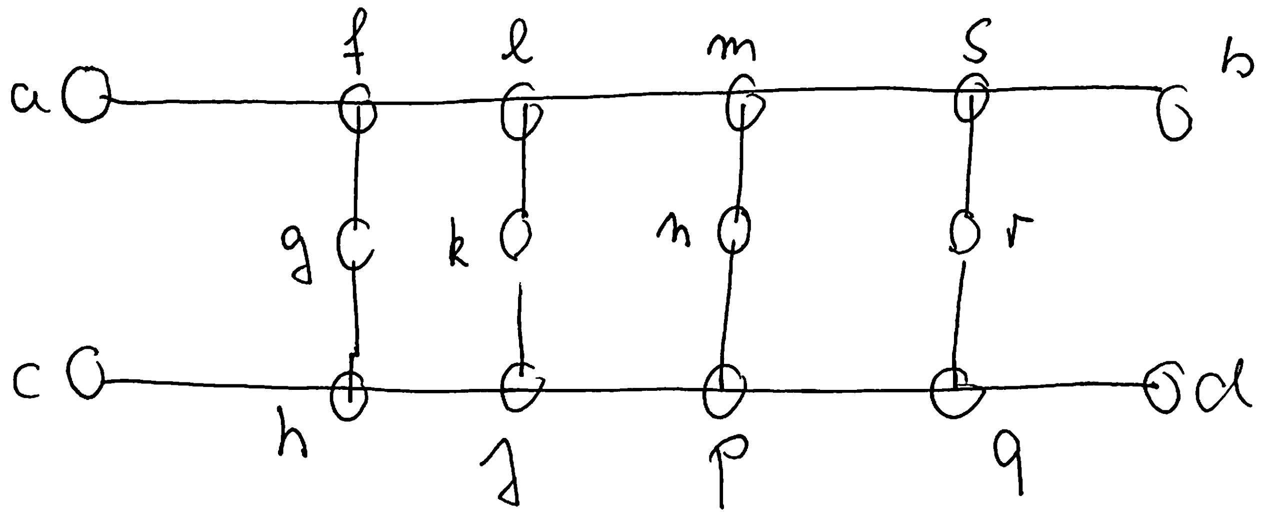
A gadget for enforcing Paired behavior



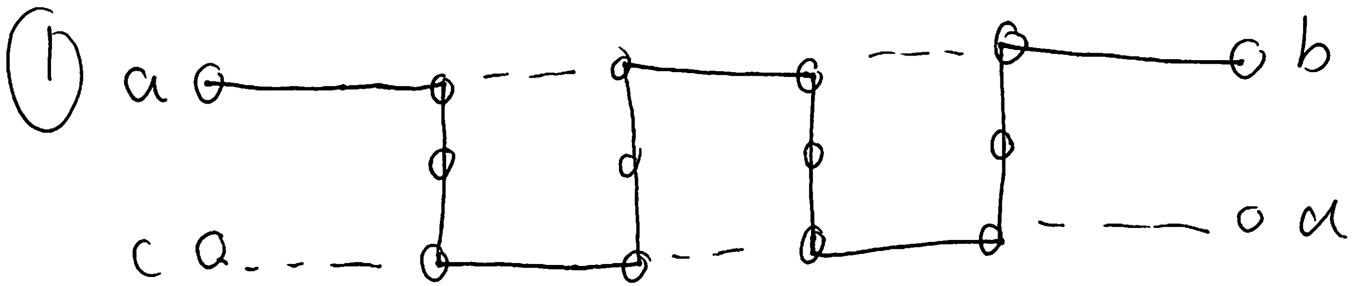
a, b, c, d connect the gadget to G; gadget no other connections.

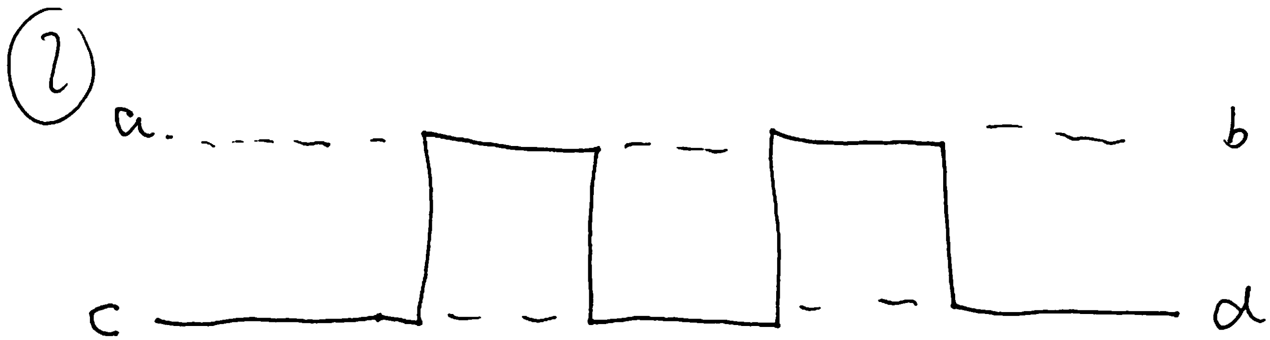
Now study the gadget in more detail.

We will see that there are exactly 2 ways to traverse it.



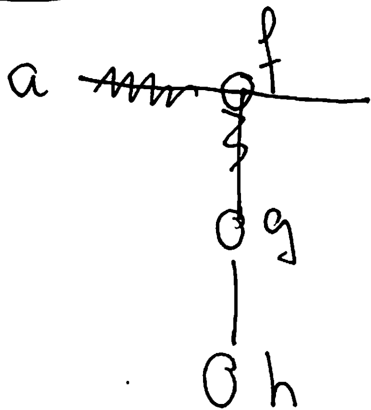
The 2 ways are



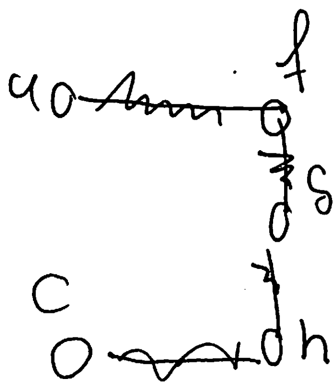


Why? see class

2 observations



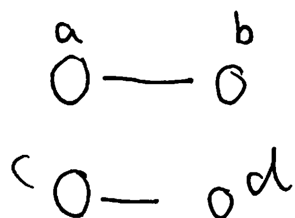
if at f must go to g at degree(g) = 2



if a, f, g, h, c is chosen the rest of the gadget cannot be traversed by a Hamiltonian cycle.

paired-edges Hamiltonian cycle → Hamiltonian cycle

For each pair of paired-edges
({a,b}, {c,d})



Construct the gadget

For any other pair that uses
({a,b}, {x,y}) replace by

({a,f}, {x,y})

if {a,f} is traversed in the gadget then in the old graph {a,b} would be traversed.

analogously {c,h} replaces {c,d}

Each gadget adds 12 vertices to the graph. + Polynomial time.

P is a Hamiltonian path in the paired-edge problem instance

$\Longleftrightarrow$

P' is a Hamiltonian path in the so construct graph

(P' as described above)

Concludes reduction ZOE → Hamiltonian Path $\square$

Recall, SAT → 3-SAT

Now

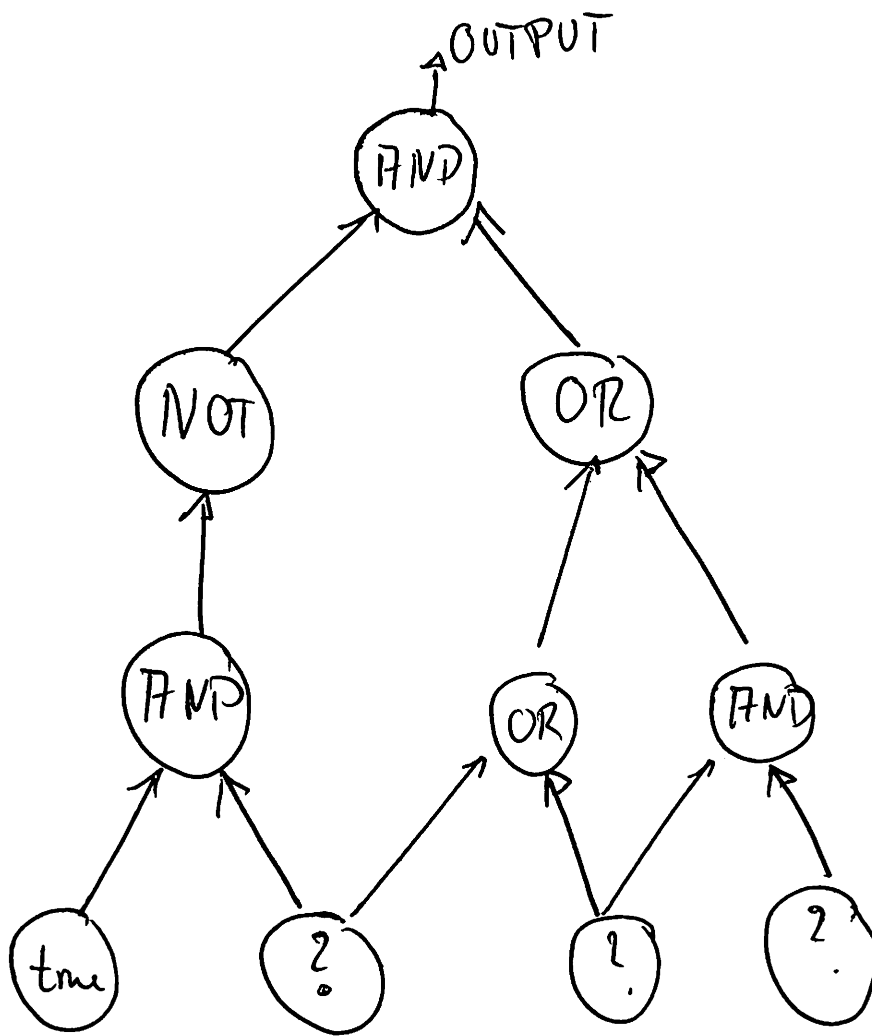
Any problem in NP → SAT

We study <u>Circuit sat</u> defined as

given a Boolean circuit,
a dag with <u>vertices</u> : <u>gates</u>
of five different types

1) AND gates
2) OR gates
3) NOT gates
4) Known input gates (true/false)
5) Unknown input gates "?"

One of the DAG's sink is called "OUTPUT" gate.



After assigning true/false values to the "?"s the gates can be evaluated in topological order.

CIRCUIT SAT

Given a circuit, find a truth assignment for the unknown gates so that the circuit outputs "true".

- CIRCUIT SAT is a generalization of SAT

a SAT formula is really a special circuit

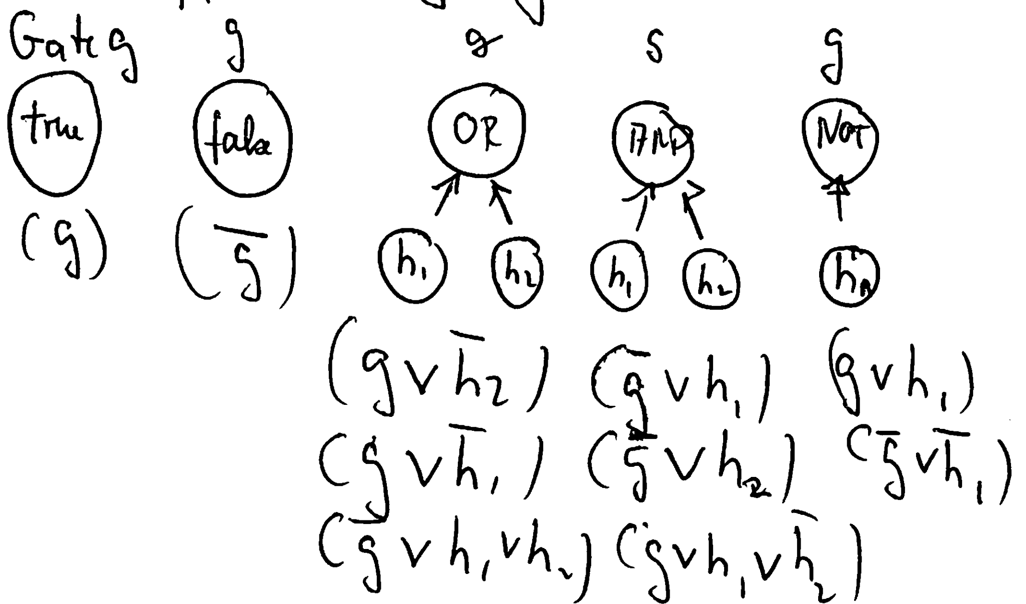
- ( ) ( )
  ↑ and gate
- (v v) or gate
- output is the value of the formula
- each literal is an unknown input gate or its negation
- no known input gates

CIRCUIT SAT → SAT

take a circuit → write it in CNF

for each gate g in the circuit
↳ create a variable g

- model effects of gates as:



[recall this construction in LP formulations]

- add ~~on~~ to output gate the clause (g)
value true is forced

So, CIRCUIT-SAT → SAT

Show that all search <u>problems in NP</u> → CIRCUIT-SAT

Any search problem's (in NP) solution can be checked in polynomial time.
(def. search problem in NP)

⇒ there is a checking algorithm which takes an instance of the search problem, I, and a solution, S, and tests if S is a solution for I.

For the <u>checking algorithm</u>

I : is encoded as a binary string of polynomial length of I

Recall any polynomial algorithm can be expressed (in a computer) as a circuit, input gates are the input to the algorithm

# of gates polynomial in the input.

Checking algorithm :

S is a solution to I → yes

S is not a " " " → no

The output gate then gives true / false correspondingly.

Conclusion :

Instance I of problem A

$\hookrightarrow$ construct in polynomial time circuit

known inputs → bits encoding I

Unknown input → bits " S

Output: true S is a solution to I

false S is not a solution to I

NP - complete problems
can be solved by "some" algorithm.
(run time likely exponential)

Some problems cannot be solved at all!
There is NO ALGORITHM that solves the problem.

Example:
are there integer values x, y, z that satisfy

$$x^3yz + 2y^4z^2 - 7xy^5z = 6 ?$$

No algorithm at all can solve this polynomial, exponential, whatever

Alan M. Turing 1936

Let P be a program
 X an input to P

Can we write a Program ?
 Terminates(P, X)
 input: any program P
 X: input to P
 Output: yes if P terminates
 on input X
 no if P does not
 terminate on X

Answer we cannot
write such a program!

Theorem No algorithm can exist which tests if any program terminates on a given input

Proof. Suppose there is such an algorithm

terminates(P, x)

Then write this:

```
function paradox (z: file)
1. if terminates (z, z) goto 1
```

You see the "infinite loop"

if z applied on itself as input terminates
then paradox does not
if it does not then paradox will.

Now apply paradox to itself
(it is a program so that is legal!

Paradox stops when it does not ??.

Contradiction

So, no algorithm can exist to decide if any algorithm stops.

↳ unsolvable