TCS for Machine Learning Scientists

Colin de la Higuera

Barcelona July 2007

Outline

1. Strings
2. Order
3. Distances
4. Kernels
5. Trees
6. Graphs
7. Some algorithmic notions and complexity theory for machine learning
8. Complexity of algorithms
9. Complexity of problems
10. Complexity classes
11. Stochastic classes
12. Stochastic algorithms
13. A hardness proof using RP≠NP

Disclaimer

- The view is that the essential bits of linear algebra and statistics are taught elsewhere. If not they should also be in a lecture on basic TCS for ML.
- There are not always fixed name for mathematical objects in TCS. This is one choice.

1 Alphabet and strings

- An alphabet Σ is a finite nonempty set of symbols called letters.
- A string *w* over Σ is a finite sequence $a_1 \ldots a_n$ of letters.
- Let $|w|$ denote the length of *w*. In this case we have $|w| = |a_1 \ldots a_n| = n$.
- The empty string is denoted by λ (in certain books notation ε is used for the empty string).

- Alternatively a string w of length n can be viewed as a mapping $[n] \rightarrow \Sigma$:
- if $w = a_1a_2 \ldots a_n$ we have $w(1) = a_1$, $w(2) = a_2$ …, $w(n) = a_n$.
- Given $a \in \Sigma$, and w a string over Σ, $|w|_a$ denotes the number of occurrences of letter a in w.
- Note that $[n]=\{1,\ldots,n\}$ with $[0]=\emptyset$

Letters of the alphabet will be indicated by *a*, *b*, *c*,…, strings over the alphabet by *u*, *v*,… , *z*

- Let Σ^* be the set of all finite strings over alphabet.
- Given a string w, x is a substring of w if there are two strings l and r such that $w = lxr$. In that case we will also say that w is a superstring of x.

- We can count the number of occurrences of a given string *u* as a substring of a string *w* and denote this value by $|w|_u = |\{l \in \Sigma^* : \exists r \in \Sigma^* \wedge w = lur\}|$.

- x is a subsequence of w if it can be obtained from w by erasing letters from w. Alternatively: $\forall x, y, z, x_1, x_2 \in \Sigma^*, \forall a \in \Sigma$:
  - x is a subsequence of x,
  - x_1x_2 is a subsequence of x_1ax_2
  - if x is a subsequence of y and y is a subsequence of z then x is a subsequence of z.

cdlh 2007

Basic combinatorics on strings

- Let $n=|w|$ and $p=|\Sigma|$
- Then the number of…

At least		At most
$n+1$	Prefixes of *w*	$n+1$
$n+1$	Substrings of w	$n(n+1)/2+1$
$n+1$	Subsequences of w	2^n

Algorithmics

- There are many algorithms to compute the maximal subsequence of 2 strings
- But computing the maximal subsequence of *n* strings is NP-hard.
- Yet in the case of substrings this is easy.

Knuth-Morris-Pratt algorithm

- Does string *s* appear as substring of string *u*?
- **Step 1** compute T[*i*] the table indicating the longest correct prefix if things go wrong.
- $T[i]=k \Leftrightarrow s_1 \ldots s_k = s_{i-k} \ldots s_{i-1}$.
- Complexity is O(|*s*|)

T[7]=2 means that if we fail when parsing *d*, we can still count on the first 2 characters been parsed.

i	1	2	3	4	5	6	7
s[*i*]	*a*	*b*	*c*	*d*	*a*	*b*	*d*
***T*[*i*]**	0	0	0	0	0	1	2

KMP (**Step 2**)

$m \leftarrow 0$; *m *position where s starts**\

$i \leftarrow 1$; **i is over s and u**\

while ($m + i \leq |u|$ & $i \leq |s|$)

  if ($u[m + i] = s[i]$) $++i$ **matches**\

  else *doesn't *match*\

    $m \leftarrow m + i - T[i] - 1$; *go back $T[i]$ *in u*\

    $i \leftarrow T[i]+1$

if ($i > |s|$) **return** $m+1$ **found s**\

else **return** $m + i$ *not *found**\

A run with abac in aaabcacabacac

i	1	2	3	4
s[*i*]	*a*	*b*	*a*	*c*
***T*[*i*]**	0	0	0	1

aaabcacabacac

m	0	0	0	1		2	2	5	7	7	7	7
i	1	2	1	2	1	2	3	2	1	2	3	4
s	a	b	a	b	a	b	a	b	a	b	a	c
u	a	a	a	a	a	b	c	c	a	b	a	c

Conclusion

- Many algorithms and data structures (tries).
- Complexity of KMP=$O(|s|+|u|)$
- Research is often about constants…

2 Order! Order!

- Suppose we have a total order relation over the letters of an alphabet Σ. We denote by $\leq_{alpha}$ this order, which is usually called the alphabetical order.
- $a \leq_{alpha} b \leq_{alpha} c\ldots$

Different orders can be defined over Σ:

- the prefix order: $x \leq_{pref} y$ if
  - $\exists w \in \Sigma^* : y = xw$;
- the lexicographic order: $x \leq_{lex} y$ if
  - either $x \leq_{pref} y$ or
  - $x = uaw \wedge y = ubz \wedge a \leq_{alpha} b$.

- A more interesting order for grammatical inference is the hierarchical order (also sometimes called the length-lexicographic or length-lex order):
- If x and y belong to Σ^*, $x \leq_{\text{length-lex}} y$ if
  - $|x| < |y| \vee (|x| = |y| \wedge x \leq_{\text{lex}} y)$.
- The first strings, according to the hierarchical order, with $\Sigma = \{a, b\}$ will be $\{\lambda, a, b, aa, ab, ba, bb, aaa, \ldots\}$.

Example

- Let $= \{a, b, c\}$ with $a <_{alpha} b <_{alpha} c$. Then $aab \leq_{lex} ab$,
- but $ab \leq_{length\text{-}lex} aab$. And the two strings are incomparable for $\leq_{pref}$.

3 Distances

- What is the issue?
- 4 types of distances
- The edit distance

The problem

- A class of objects or representations *C*
- A function $C^2 \rightarrow R^+$
- Such that the closer *x* and *y* are one to each other, the smaller is $d(x,y)$.

The problem

- A class of objects/representations C
- A function $C^2 \to \mathbb{R}$
- which has the following properties:
  - $d(x,x)=0$
  - $d(x,y)=d(y,x)$
  - $d(x,y)\geq 0$
- And sometimes
  - $d(x,y)=0 \Rightarrow x=y$
  - $d(x,y)+d(y,z)\geq d(x,z)$

A metric space

Summarizing

A metric is a function $C^2 \to \mathrm{R}$

which has the following properties:

- $d(x,y)=0 \Leftrightarrow x=y$
- $d(x,y)=d(y,x)$
- $d(x,y)+d(y,z) \geq d(x,z)$

Pros and cons

- A distance is more flexible
- A metric gives us extra properties that we can use in an algorithm

Four types of distances (1)

- Compute the number of modifications of some type allowing to change *A* to *B.*
- Perhaps normalize this distance according to the sizes of *A* and *B* or to the number of possible paths
- Typically, the edit distance

Four types of distances (2)

- Compute a similarity between *A* and *B*. This is a positive measure *s*(*A*,*B*).
- Convert it into a metric by one of at least 2 methods.

Method 1

- Let $d(A,B)=2^{-s(A,B)}$
- If $A=B$, then $d(A,B)=0$
- Typically the prefix distance, or the distance on trees:
- $S(t_1,t_2)=\min\{|x|: t_1(x)\neq t_2(x)\}$

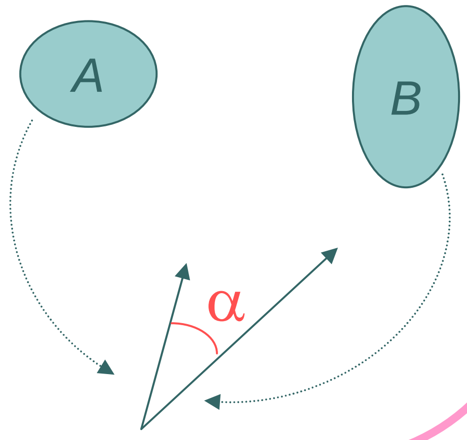
Method 2

- $d(A,B) = s(A,A) - s(A,B) - s(B,A) + s(B,B)$
- Conditions
  - $d(x,y)=0 \Rightarrow x=y$
  - $d(x,y)+d(y,z) \geq d(x,z)$

only hold for some special conditions on *s*.

Four types of distances (3)

- Find a finite set of measurable features
- Compute a numerical vector for *A* and *B* (v_A and v_B). These vectors are elements of R^n.
- Use some distance d_v over R^n
- $d(A,B)=d_v(v_A, v_B)$



Four types of distances (4)

- Find an infinite (enumerable) set of measurable features
- Compute a numerical vector for A and B (v_A and v_B). These vectors are elements of R^∞.
- Use some distance d_v over R^∞
- $d(A,B)=d_v(v_A, v_B)$

The edit distance

- Defined by Levens(h)tein, 1966
- Algorithm proposed by Wagner and Fisher, 1974
- Many variants, studies, extensions, since

Basic operations

- Insertion
- Deletion
- Substitution
- Other operations:
  - inversion

- Given two strings w and w' in Σ^*, w rewrites into w' in one step if one of the following correction rules holds:
- $w=uav$, $w'=uv$ and u, $v\in\Sigma^*$, $a\in\Sigma$ (single symbol deletion)
- $w=uv$, $w'=uav$ and u, $v\in\Sigma^*$, $a\in\Sigma$ (single symbol insertion)
- $w=uav$, $w'=ubv$ and u, $v\in\Sigma^*$, $a,b\in\Sigma$, (single symbol substitution)

Examples

- $abc \rightarrow ac$
- $ac \rightarrow abc$
- $abc \rightarrow aec$

cdlh 2007

- We will consider the reflexive and transitive closure of this derivation, and denote $w \xrightarrow{k} w'$ if and only if w rewrites into w' by k operations of single symbol deletion, single symbol insertion and single symbol substitution.

- Given 2 strings w and w', the *Levenshtein distance* between w and w' denoted $d(w,w')$ is the smallest k such that $w \xrightarrow{k} w'$.
- **Example:** $d(abaa, aab) = 2$. *abaa* rewrites into *aab* via (for instance) a deletion of the *b* and a substitution of the last *a* by a *b*.

A confusion matrix

	a	*b*	*c*	λ
a	0	1	1	1
b	1	0	1	1
c	1	1	0	1
λ	1	1	1	0

Another confusion matrix

	a	*b*	*c*	λ
a	0	0.7	0.4	1
b	0.7	0	0.6	0.8
c	0.4	0.6	0	0.7
λ	1	0.8	0.7	0

A similarity matrix using an evolution model

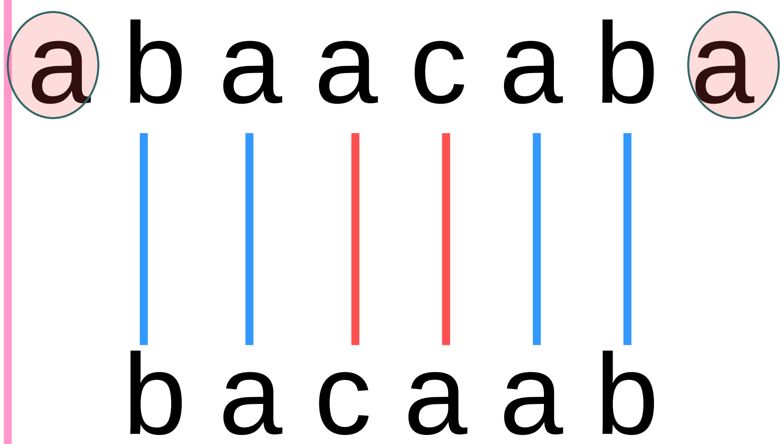
BLOSUM62 matrix

C	9																			
S	-1	4																		
T	-1	1	5																	
P	-3	-1	-1	7																
A	0	1	0	-1	4															
G	-3	0	-2	-2	0	6														
N	-3	1	0	-2	-2	0	6													
D	-3	0	-1	-1	-2	-1	1	6												
E	-4	0	-1	-1	-1	-2	0	2	5											
Q	-3	0	-1	-1	-1	-2	0	0	2	5										
H	-3	-1	-2	-2	-2	-2	1	-1	0	0	8									
R	-3	-1	-1	-2	-1	-2	0	-2	0	1	0	5								
K	-3	0	-1	-1	-1	-2	0	-1	1	1	-1	2	5							
M	-1	-1	-1	-2	-1	-3	-2	-3	-2	0	-2	-1	-1	5						
I	-1	-2	-1	-3	-1	-4	-3	-3	-3	-3	-3	-3	-3	1	4					
L	-1	-2	-1	-3	-1	-4	-3	-4	-3	-2	-3	-2	-2	2	2	4				
V	-1	-2	0	-2	0	-3	-3	-3	-2	-2	-3	-3	-2	1	3	1	4			
F	-2	-2	-2	-4	-2	-3	-3	-3	-3	-3	-1	-3	-3	0	0	0	-1	6		
Y	-2	-2	-2	-3	-2	-3	-2	-3	-2	-1	2	-2	-2	-1	-1	-1	-1	3	7	
W	-2	-3	-2	-4	-3	-2	-4	-4	-3	-2	-2	-3	-3	-1	-3	-2	-3	1	2	11
	C	S	T	P	A	G	N	D	E	Q	H	R	K	M	I	L	V	F	Y	W

Conditions

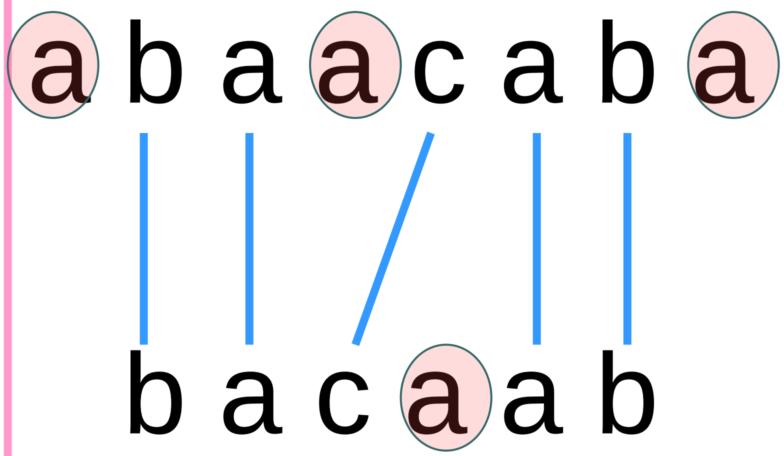
- $C(a,b) < C(a,\lambda)+C(\lambda,b)$
- $C(a,b) = C(b,a)$
- Basically *C* has to respect the triangle inequality

Aligning



d=2+2+0=4

Aligning



d=3+0+1=4

General algorithm

- What does not work:
  - Compute all possible sequences of modifications, recursively.
- Something like:

$$d(ua,vb)=1+\min(d(ua,v),\ d(u,vb),\ d(u,v))$$

The formula for dynamic programming

$d(ua,vb)=$

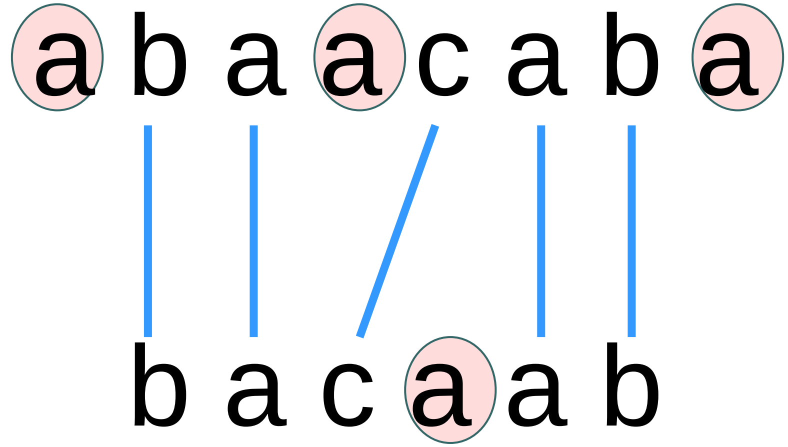
- if $a=b$, $d(u,v)$
- if $a\neq b$,

$$\min \begin{cases} d(u,vb)+C(a,\lambda) \\ d(u,v)+C(a,b) \\ d(ua,v)+C(\lambda,b) \end{cases}$$

b	6	5	4	4	3	3	4	3	4
a	5	4	4	3	2	3	3	3	3
a	4	3	3	2	2	3	2	3	4
c	3	2	2	2	2	2	3	4	5
a	2	1	2	1	2	3	4	5	6
b	1	1	1	2	3	4	5	6	7
λ	0	1	2	3	4	5	6	7	8
	λ	a	b	a	a	c	a	b	a

cdlh 2007

b	6	5	4	4	3	3	4	3	4
a	5	4	4	3	2	3	3	3	3
a	4	3	3	2	2	3	2	3	4
c	3	2	2	2	2	2	3	4	5
a	2	1	2	1	2	3	4	5	6
b	1	1	1	2	3	4	5	6	7
λ	0	1	2	3	4	5	6	7	8
	λ	a	b	a	a	c	a	b	a



b	6	5	4	4	3	3	4	3	4
a	5	4	4	3	2	3	3	3	3
a	4	3	3	2	2	3	2	3	4
c	3	2	2	2	2	2	3	4	5
a	2	1	2	1	2	3	4	5	6
b	1	1	1	2	3	4	5	6	7
λ	0	1	2	3	4	5	6	7	8
	λ	a	b	a	a	c	a	b	a

Complexity

- Time and space $O(|u|.|v|)$
- Note that if normalizing by dividing by the sum of lengths $[d_N(u,v)=d_e(u,v) / (|u|+|v|)]$ you end up with something that is not a distance:
  - $d_N(ab,aba)=0.2$
  - $d_N(aba,ba)=0.2$
  - $d_N(ab,ba)=0.5$

Extensions

- Can add other operations such as inversion *uabv*$\rightarrow$*ubav*
- Can work on circular strings
- Can work on languages

- A. V. Aho, Algorithms for Finding Patterns in Strings, in: *Handbook of Theoretical Computer Science* (Elsevier, Amsterdam, 1990) 290-300.
- L. Miclet, *Méthodes Structurelles pour la Reconnaissance des Formes* (Eyrolles, Paris, 1984).
- R. Wagner and M. Fisher, The string-to-string Correction Problem, *Journal of the ACM* **21** (1974) 168-178.

Note (recent (?) idea, re Bunke *et al.*)

- Another possibility is to choose *n* strings, and given another string *w*, associate the feature vector <$d(w,w_1),d(w,w_2),\ldots$>.
- How do we choose the strings?
- Has this been tried?

4 Kernels

- A kernel is a function $\kappa : A \times A \to R$ such that there exists a feature mapping $\phi : A \to R^n$, and $\kappa(x,y) = < \phi(x), \phi(y) >$.
- $<\phi(x), \phi(y)> = \phi_1(x)\cdot\phi_1(y) + \phi_2(x)\cdot\phi_2(y) + \ldots + \phi_n(x)\cdot\phi_n(y)$
- (dot product)

Some important points

- The κ function is explicit, the feature mapping ϕ may only be implicit.
- Instead of taking R^n any Hilbert space will do.
- If the kernel function is built from a feature mapping ϕ, this respects the kernel conditions.

Crucial points

- Function ϕ should have a meaning.
- The computation of $\kappa(x,y)$, should be inexpensive: we are going to be doing this computation many times. Typically $O(|x|+|y|)$ or $O(|x|.|y|)$.
- But notice that $\kappa(x,y)=\sum_{i\in I} = \phi_i(x)\cdot\phi_i(y)$
- With I that can be infinite!

Some string kernels (1)

- The Parikh kernel:
  - $\phi(u)=(|u|_{a1}, |u|_{a2}, |u|_{a3},\ldots, |u|_{a|\Sigma|})$
    $\kappa(aaba, bbac)=|aaba|_a*|bbac|_a+ |aaba|_b*|bbac|_b + |aaba|_c*|bbac|_c= 3*1+1*2+0*1=5$

Some string kernels (2)

- The spectrum kernel:
- Take a length p. Let $s_1, s_2, \ldots, s_k$ be an enumeration of all strings in Σ^p
  - $\phi(u)=(|u|_{s1}, |u|_{s2}, |u|_{s3},\ldots, |u|_{sk})$
  - $\kappa(aaba, bbac)=1$ (for $p=2$)

  (only *ba* in common!)
  - In other fields *n*-grams !
  - Computation time $O(p\ |x|\ |y|)$

Some string kernels (3)

- The all-subsequences kernel:
- Let s_1, s_2, …, s_n,… be an enumeration of all strings in Σ^+
- Denote by $\phi^A(u)_s$ the number of times s appears as a subsequence in u.
  - $\phi^A(u)=(\phi^A(u)_{s1}, \phi^A(u)_{s2}, \phi^A(u)_{s3},\ldots, \phi^A(u)_{sn}, \ldots)$
  - $\kappa(aaba, bbac)=6$
  - $\kappa(aaba, abac)=7+3+2+1=13$

Some string kernels (4)

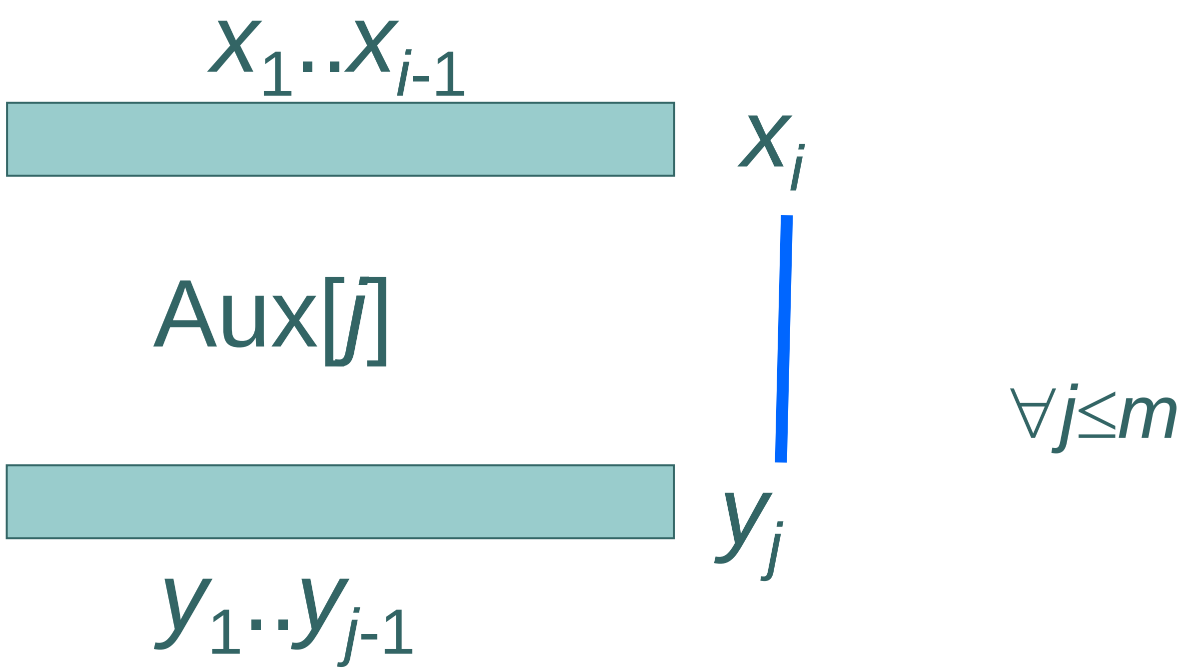
- The gap-weighted subsequences kernel:
- Let $s_1, s_2, \ldots, s_n, \ldots$ be an enumeration of all strings in Σ^+
- Let λ be a constant > 0
- Denote by $\phi_j(u)_{s,i}$ be the number of times s appears as a subsequence in u of length i
- Then $\phi_j(u)$ is the sum of all $\phi_j(u)_{sj,l}$
- Example: u=‘caat’, let s_j=‘at’, then $\phi_j(u) = \lambda^2 + \lambda^3$

- Curiously a typical value, for theoretical proofs, of λ is 2. But a value between 0 and 1 is more meaningful.
- $O(|x|\,|y|)$ computation time.

How is a kernel computed?

- Through dynamic programming
- We do not compute function ϕ
- Example of the all-subsequences kernel
  - $K[i][j] = \kappa(x_1,\ldots x_i, y_1 \ldots y_j)$
  - Aux[j] (at step i): number of alignments where x_i is paired with y_j.

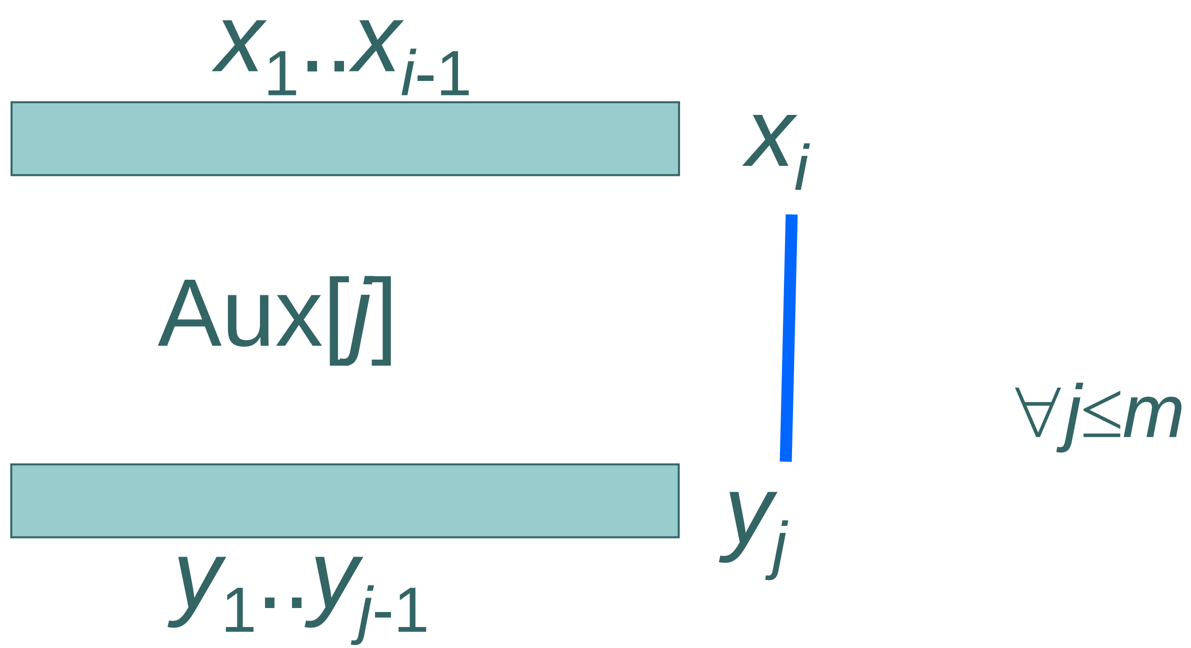
General idea (1) Suppose we know (at step *i*)



The number of alignments of $x_1..x_i$ with $y_1..y_j$ *where* x_i *is matched with* y_j

cdlh 2007

General idea (2)



Notice that Aux[j] =K[i-1][j-1]

General idea (3)

An alignment between $x_1..x_i$ and $y_1..y_m$ is either an alignment where x_i is matched with one of the y_j (and the number of these is Aux[*m*]), or an alignment where x_i is not matched with anyone (so that is K[*i*-1][*m*].

$\kappa(x_1,\ldots x_n, y_1\ldots y_m)$

λ always matches

For $j \in [1,m]$ K[0][j]=1

All matchings of x_i with earlier y

For $i \in [1,n]$

last $\leftarrow$ 0; Aux[0] $\leftarrow$ 0;

Match x_i with y_j

For $j \in [1,m]$

Aux [k] $\leftarrow$ Aux[last]

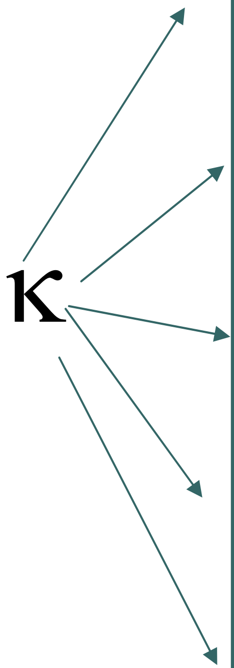
if ($x_i = y_j$) then Aux[j] $\leftarrow$ Aux[last]+K[i-1][j-1]

last $\leftarrow$ k;

For $j \in [1,m]$

K[i][j] $\leftarrow$ K[i-1][j]+Aux[j]

The arrays K and Aux for cata and gatta



	λ	g	a	t	t	a
λ	1	1	1	1	1	1
Aux	0	0	0	0	0	0
c	1	1	1	1	1	1
Aux	0	0	1	1	1	2
a	1	1	2	2	2	3
Aux	0	0	0	2	4	4
t	1	1	2	4	6	7
Aux	0	0	1	1	1	7
a	1	1	3	5	7	14

Ref: Shawe Taylor and Christianini

Why not try something else ?

- The all-substrings kernel:
- Let s_1, s_2, …, s_n,… be an enumeration of all strings in Σ^+
  - $\phi(u)=(|u|_{s1}, |u|_{s2}, |u|_{s3},\ldots, |u|_{sn}, \ldots)$
  - $\kappa(aaba, bbac)=7$ (1+3+2+0+0..+1+0…)
- No formula ?

Or an alternative edit kernel

- $\kappa(x,y)$ is the number of possible matchings in a best alignment between *x* and *y.*
- Is this positive definite (Mercer's conditions)?

Or counting substrings only once?

- $\phi_u(x)$ is the maximum n such that u^n is a subsequence of x.
- No nice way of computing things…

Bibliography

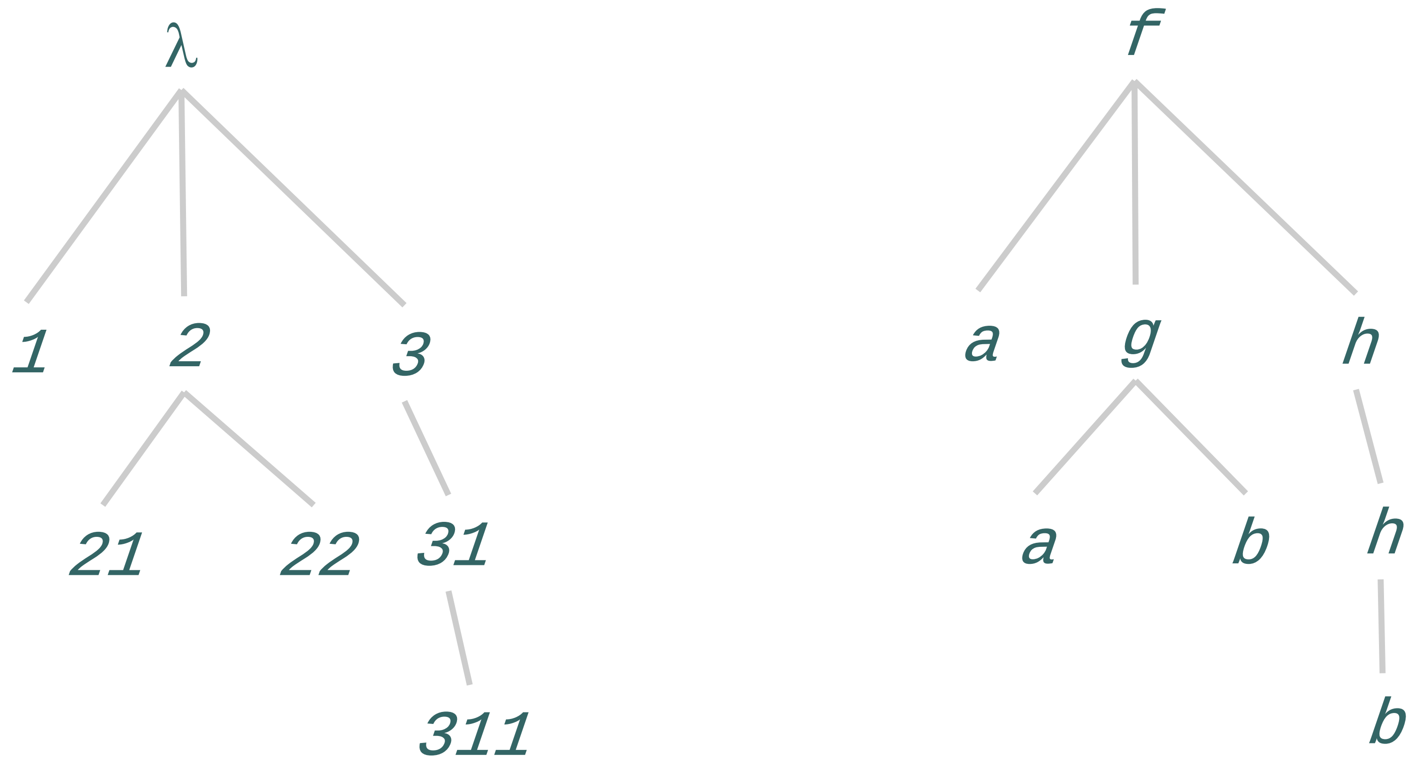
- Kernel Methods for Pattern Analysis.
  J. Shawe Taylor and N. Christianini. CUP
- Articles by A. Clark and C. Watkins (et al.) (2006-2007)

5 Trees

- A tree domain (or Dewey tree) is a set of strings over alphabet $\{1,2,\ldots,n\}$ which is prefix closed:
- $uv \in \mathrm{Dom}(t) \Rightarrow u \in \mathrm{Dom}(t)$.
- Example: $\{\lambda, 1, 2, 3, 21, 22, 31, 311\}$
- Note: often start counting from 0 (*sic*)

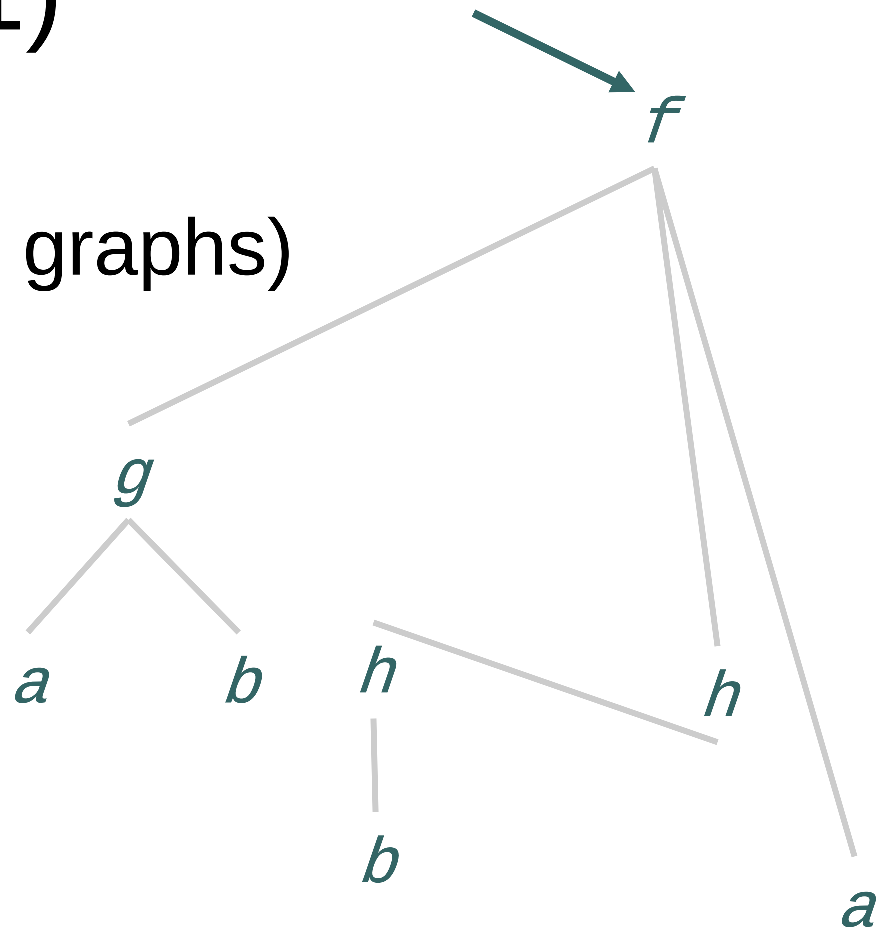
- A ranked alphabet is an alphabet Σ, with a rank (arity) function $\rho: \Sigma \rightarrow \{0,..,n\}$
- A tree is a function from a tree domain to a ranked alphabet, which respects $\rho(u)=k \Rightarrow uk \in \mathrm{Dom}(t)$ and $u(k+1) \notin \mathrm{Dom}(t)$

An example



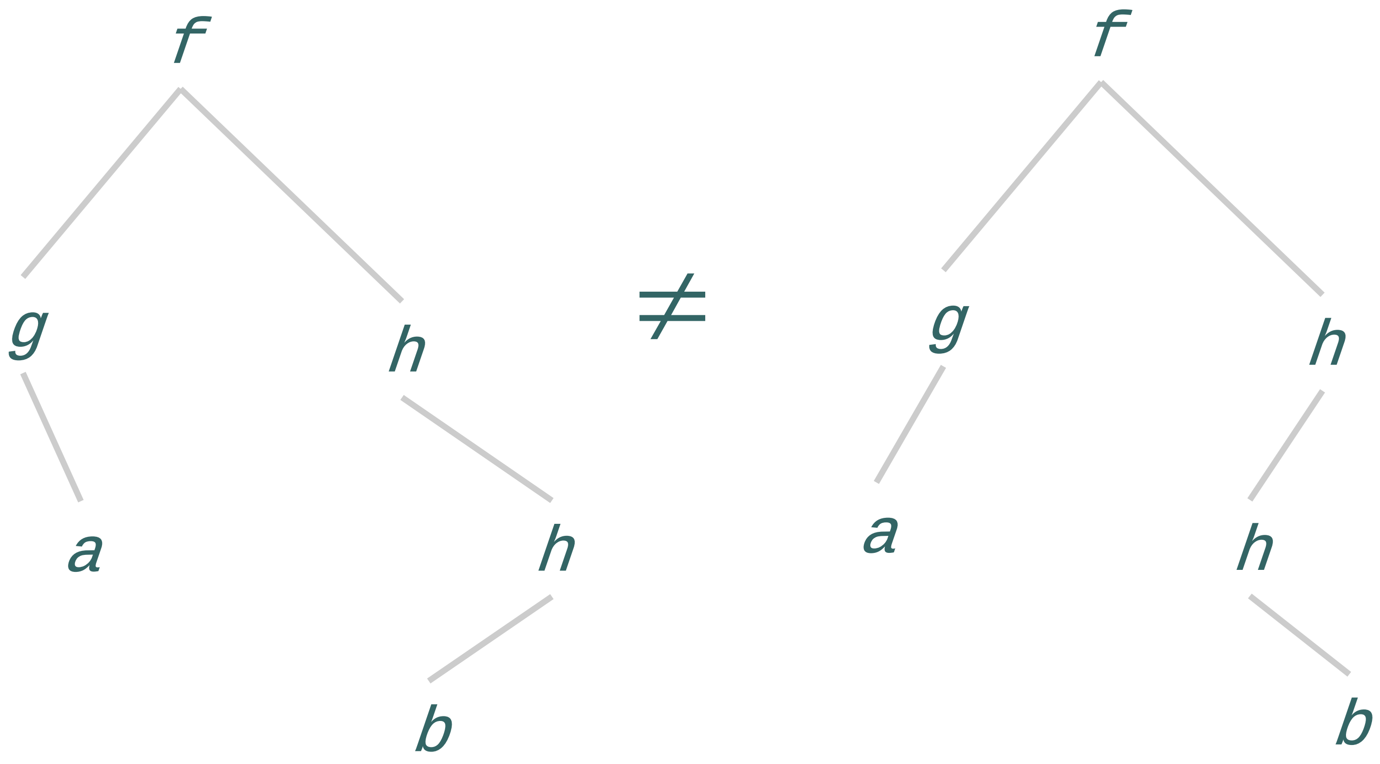
Variants (1)

- Rooted trees (as graphs)



But also unrooted…

Binary trees

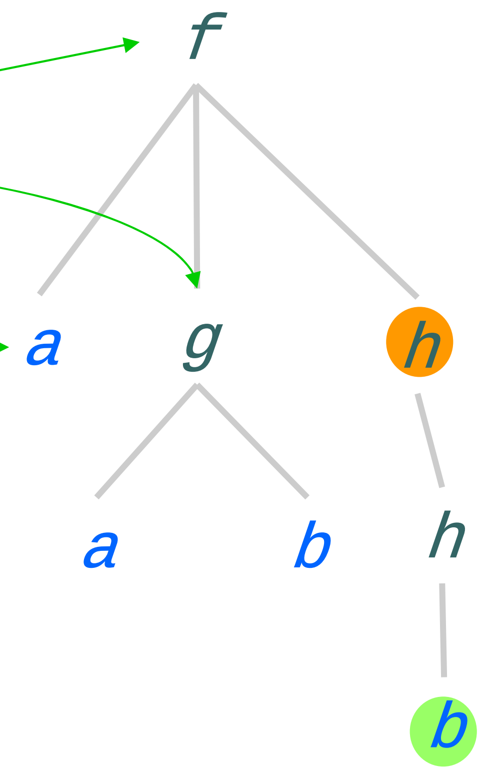


Exercises

- Some combinatorics on trees…
- How many
  - Dewey trees are there with 2, 3, *n* nodes?
  - binary trees are there with 2, 3, *n* nodes?

Some vocabulary

- The root of a tree
- Internal node
- Leaf in a tree
- The frontier of a tree
- The siblings
- The ancestor (of)
- The descendant (of)
- Father-son…Mother daughter !



About binary trees

full binary tree → every node has zero or two children.

perfect (**complete**) **binary tree** → *full binary tree* + *leaves* are at the same *depth*.

About algorithms

- An edit distance can be computed
- Tree kernels exist
- Finding patterns is possible
- General rule: we can do on trees what we can do on strings, at least in the ordered case!
- But it is usually more difficult to describe.

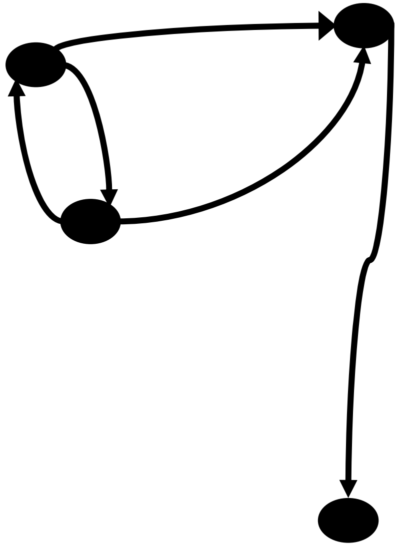
Set of trees…

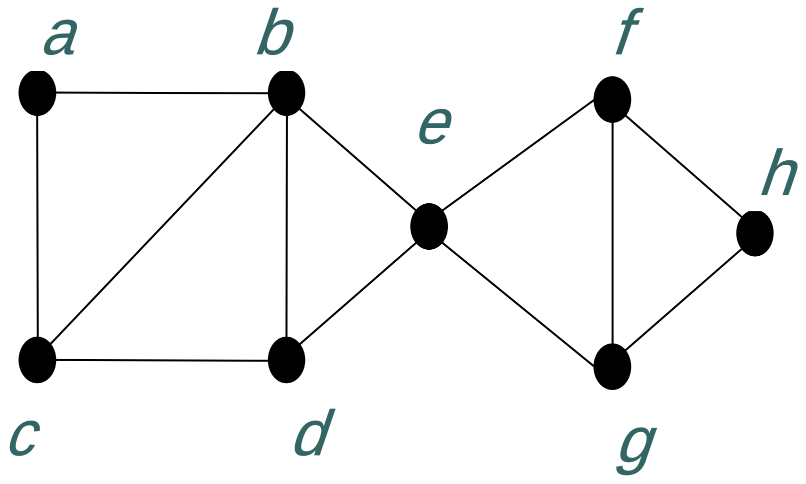
is a forest

- Sequence of trees…

is a hedge!

6 Graphs





A graph

is undirected, (*V*,*E*), where *V* is the set of vertices (a vertex), and *E* the set of edges.

- You may have loops.
- An edge is undirected, so a set of 2 vertices {*a*,*b*} or of 1 vertex {*a*} (for a loop). An edge is incident to 2 vertices. It has 2 extremities.

A digraph

is a $G=(V,A)$ where V is a set of vertices and A is a set of arcs. An arc is directed and has a start and an end.

Some vocabulary

Undirected graphs

- an edge
- a chain
- a cycle
- connected

Di-graphs

- an arc
- a path
- a circuit
- strongly connected

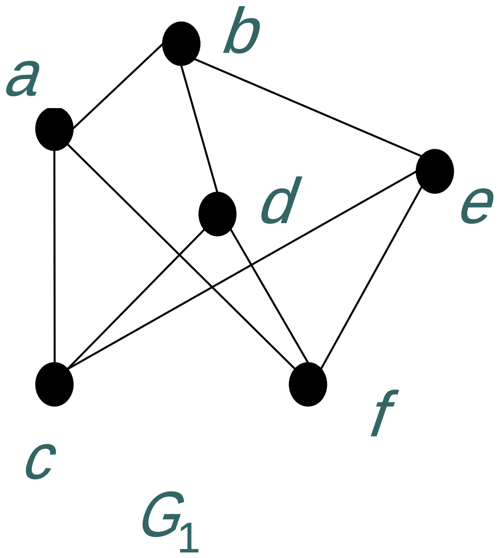
What makes graphs so attractive?

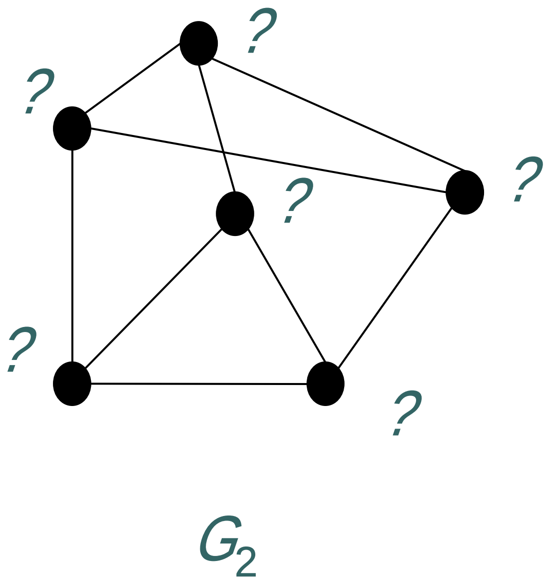
- We can represent many situations with graphs.
- From the modelling point of view, graphs are great.

Why not use them more?

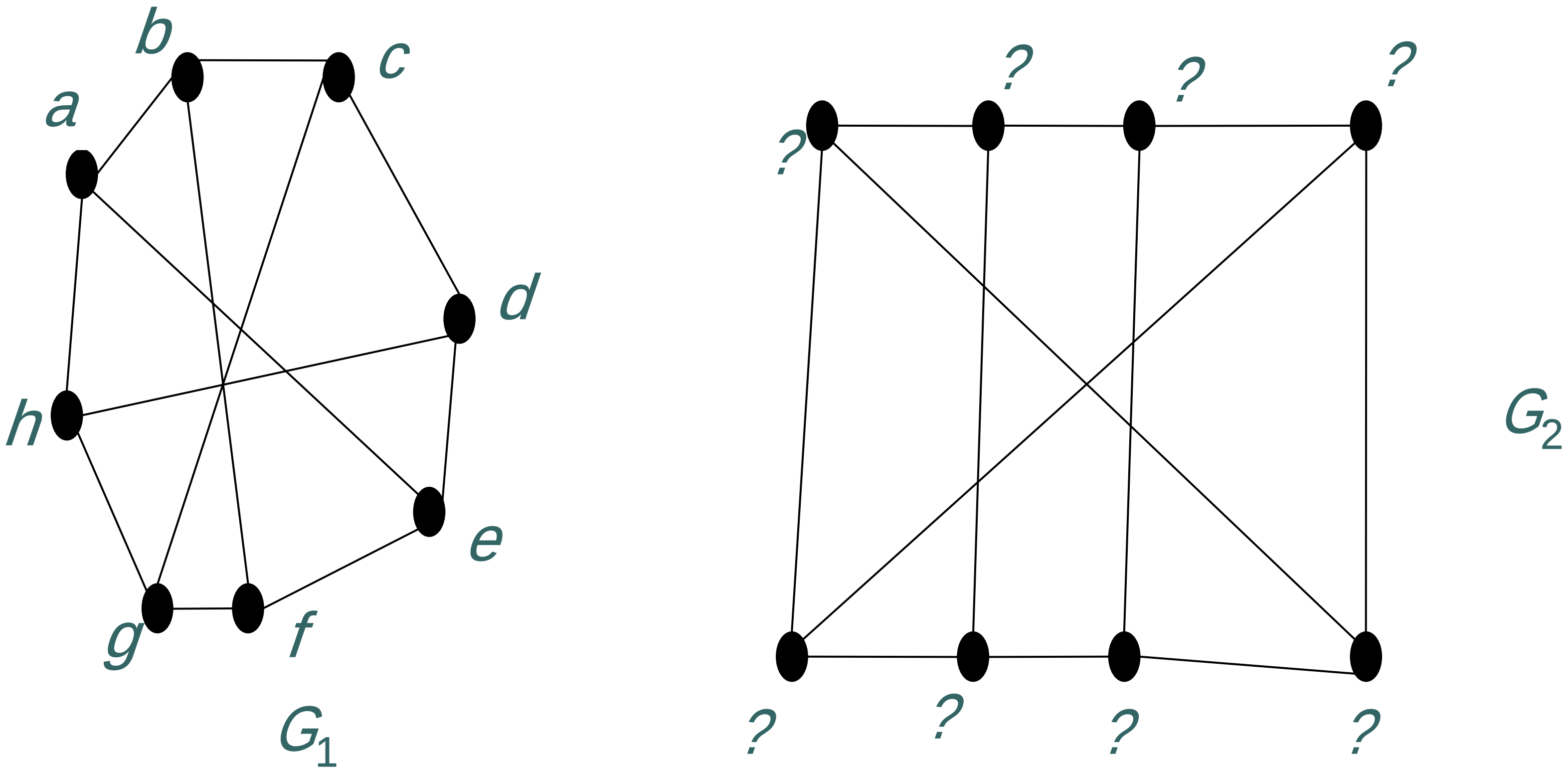
- Because the combinatorics are really hard.
- Key problem: graph isomorphism.
- Are graphs G1 and G2 isomorphic?
- Why is it a key problem?
  - For matching
  - For a good distance (metric)
  - For a good kernel

Isomorphic?





Isomorphic?



Conclusion

- Algorithms matter.
- In machine learning, some basic operations are performed an enormous number of times. One should look out for the definitions algorithmically reasonable.

7 Some algorithmic notions and complexity theory for machine learning

- Concrete complexity (or complexity of the algorithms
- Complexity of the problems

Why are complexity issues going to be important?

- Because the volumes of data for ML are very large
- Because since we can learn with randomized algorithms we might be able to solve combinatorially hard problems thanks to a learning problem
- Because mastering complexity theory is one key to successful ML applications.

8 Complexity of algorithms

- Goal is to say some thing about how fast an algorithm is.
- Alternatives are:
  - Testing (stopwatch)
  - Maths

Maths

- We could test on
  - A best case
  - An average case
  - A worse case

Best case

- We can encode detection of the best case in the algorithm, so this is meaningless

Average case

- Appealing
- Where is the distribution over which we average?
- But sometimes we can use Monte-Carlo algorithms to have average complexity

Worse case

- Gives us an upper bound
- Can sometimes transform the worse case to average case through randomisation

Notation O($f(n)$)

- This is the set of all functions asymptotically bounded (by above) by $f(n)$
- So for example in O(n^2) we find

$n \to n^2$, $n \to n \log n$, $n \to n$, $n \to 1$,
$n \to 7$, $n \to 5n^2+317n+423017$

- Exists $\exists n_0$, $\exists\, k > 0$, $\forall n \geq n_0$, $g(n) \leq k \cdot f(n)$

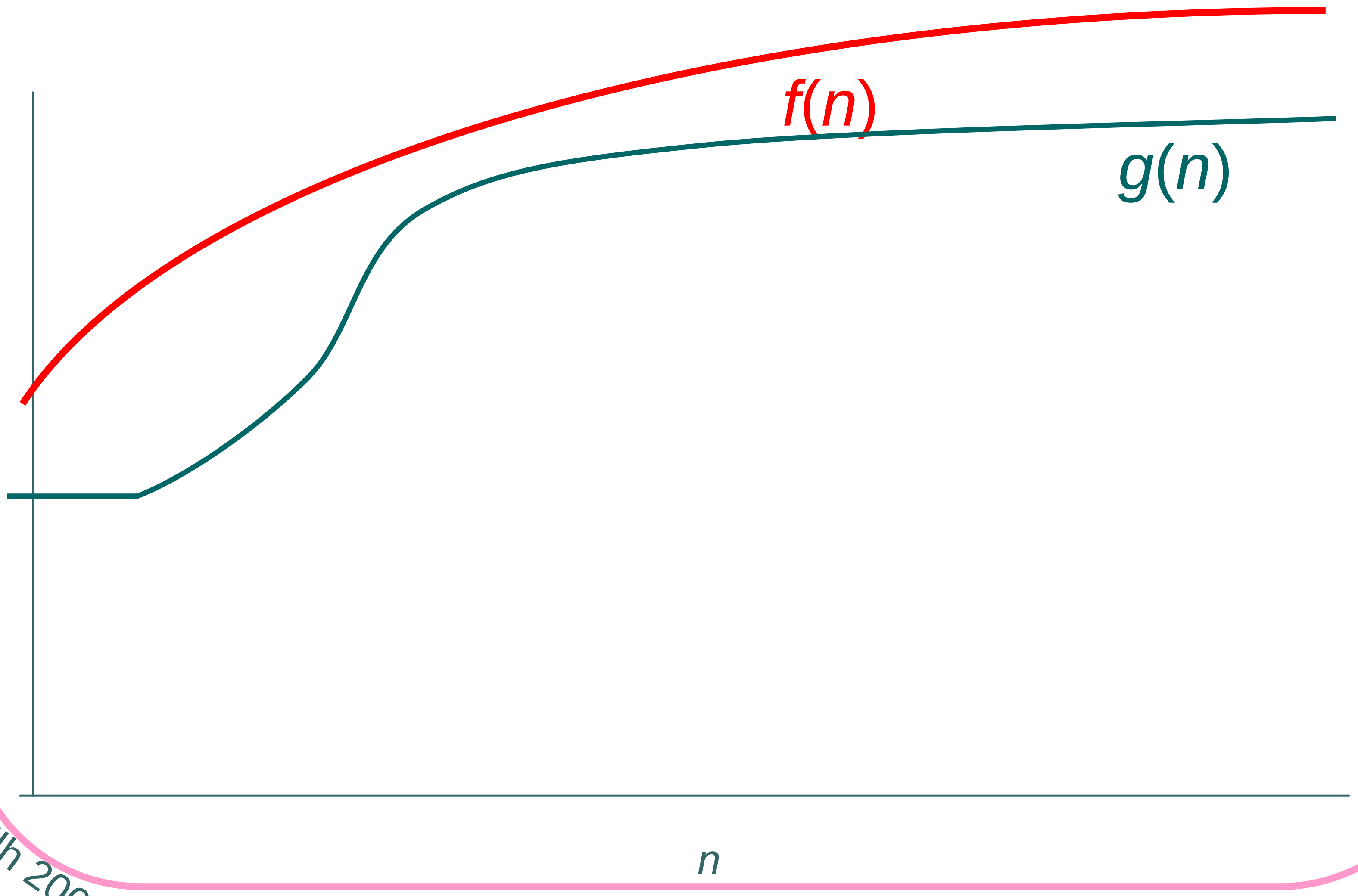
Alternative notations

- $\Omega(f(n))$

  This is the set of all functions asymptotically bounded (by underneath) by $f(n)$
- $\Theta(f(n))$

  This is the set of all functions· asymptotically bounded (by both sides) by $f(n)$

$$\exists n_0, \exists\ k_1,k_2>0, \forall n\geq n_0, k_1 \cdot f(n) \leq g(n) \leq k_2 \cdot f(n)$$


f(n)
g(n)
n

Some remarks

- This model is known as the RAM model. It is nowadays attacked, specifically for large masses of data.
- It is usually accepted that an algorithm whose complexity is polynomial is OK. If we are in $\Omega(2^n)$, no.

9 Complexity of problems

- A problem has to be well defined, *ie* different experts will agree about what a correct solution is.
- For example 'learn a formula from this data' is ill defined, as is 'where are the interest points in this image?'.
- For a problem to be well defined we need a description of the instances of the problem and of the solution.

Typology of problems (1)

- Counting problems
- How many ***x*** in ***I*** such that ***f***(***x***)

Typology of problems (2)

- Search/optimisation problems
- Find ***x*** minimising ***f***

Typology of problems (3)

- Decision problems
- Is ***x*** (in ***I***) such that ***f***(***x***)?

About the parameters

- We need to encode the instances in a fair and reasonable way.
- Then we consider the parameters that define the size of the encoding
- Typically
  - Size(n)=log n
  - Size(w)=$|w|$ (when $|\Sigma|\geq 2$)
  - Size($G=(V,E)$)=$|V|^2$ or $|V| \cdot |E|$

What is a good encoding?

- An encoding is reasonable if it encodes sufficient different objects.
- *Ie* with n bits you have 2^{n+1} encodings so optimally you should have 2^{n+1} different objects.
- Allow for redundancy and syntactic sugar, so $\Omega(p(2^{n+1}))$ different languages.

Simplifying

- Only decision problems !
  - Answer is YES or NO
- A problem is a Π, and the size of an instance is *n*.
- With a problem Π, we associate the co-problem co-Π
- The set of positive instances for Π is denoted I+(Π,)

10 Complexity Classes

- $\mathcal{P}$: *deterministic polynomial time*
- $\mathcal{NP}$: *non deterministic polynomial time*

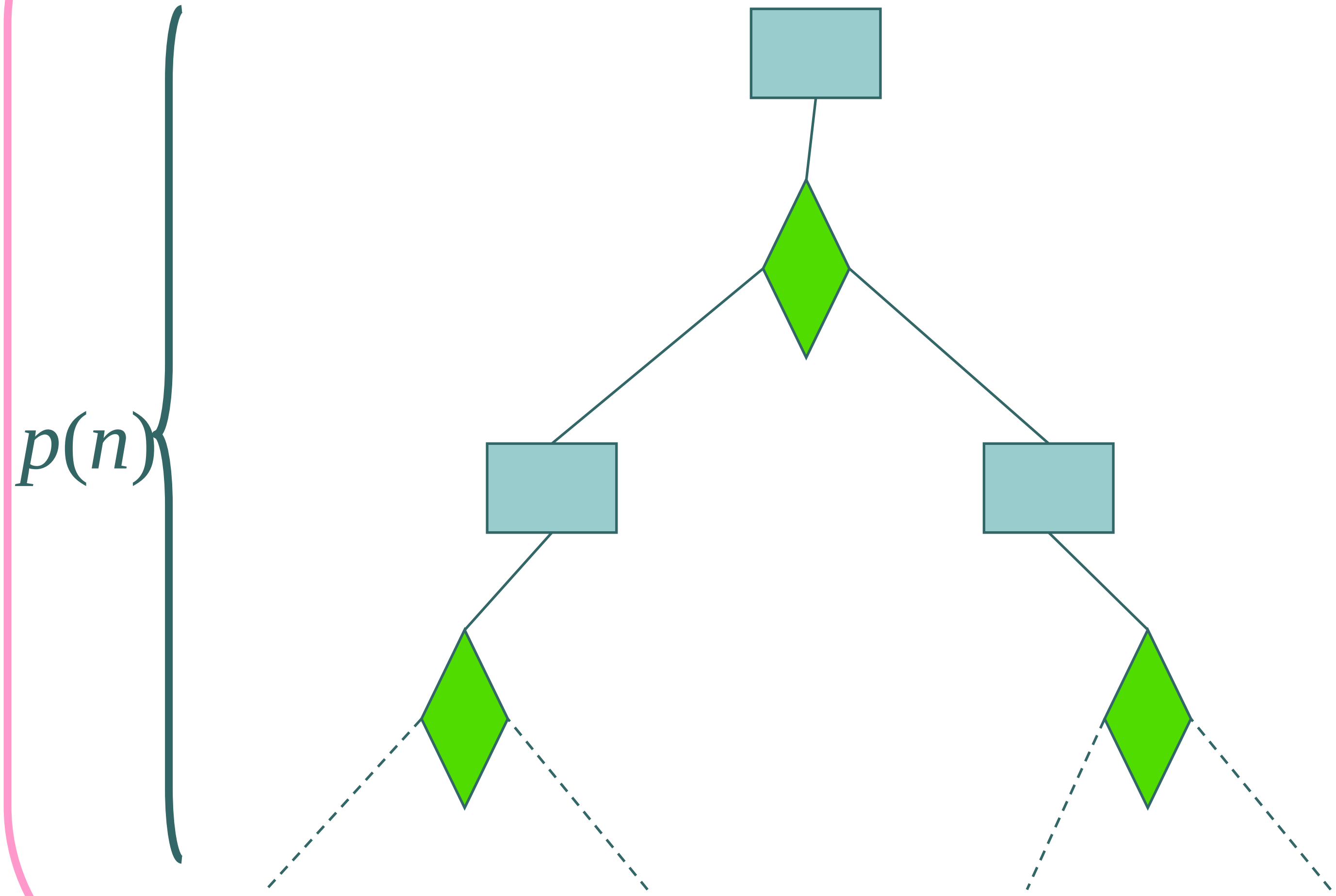
Turing machines

- Only one tape
- Alphabet of 2 symbols
- An *input* of length *n*
- We can count:
  - number of steps till halting
  - size of tape used for computation

Determinism and non determinism

- Determinism: at each moment, only one rule can be applied.
- Non determinism: various rules can be applied “in parallel”. The language recognised is that of the (positive) instances where there is at least one accepting computation.

Computation tree for non determinism



$\mathcal{P}$ and $\mathcal{NP}$

- $\Pi \in \mathcal{P} \Leftrightarrow \exists\, M_D\ \exists\, p()\ \forall i \in I(\Pi)$: #steps $(M_D(i)) \leq p(\text{size}(i))$
- $\Pi \in \mathcal{NP} \Leftrightarrow \exists\, M_N\ \exists\, p()\ \forall i \in I{+}(\Pi)$: #steps $(M_N(i)) \leq p(\text{size}(i))$

Programming point of view

- $\mathcal{P}$: the program works in polynomial time
- $\mathcal{NP}$: the program takes wild guesses, and if guesses were correct will find the solution in polynomial time.

Turing Reduction

- $\Pi_1 \leq^{\mathcal{P}}{}_{T} \Pi_2$ (Π_1 reduces to Π_2) if there exists a polynomial algorithm solving Π_1 using an oracle that consults Π_2 .

- There is another type of reduction, usually called ‘polynomial’

Reduction

- $\Pi_1 \leq^{\mathcal{P}} \Pi_2$ (Π_1 reduces to Π_2) if there exists a polynomial transformation ψ of the instances of Π_1 into those of Π_2 such that

$$i \in \Pi_1 \Leftrightarrow \psi(i) \in \Pi_2 .$$

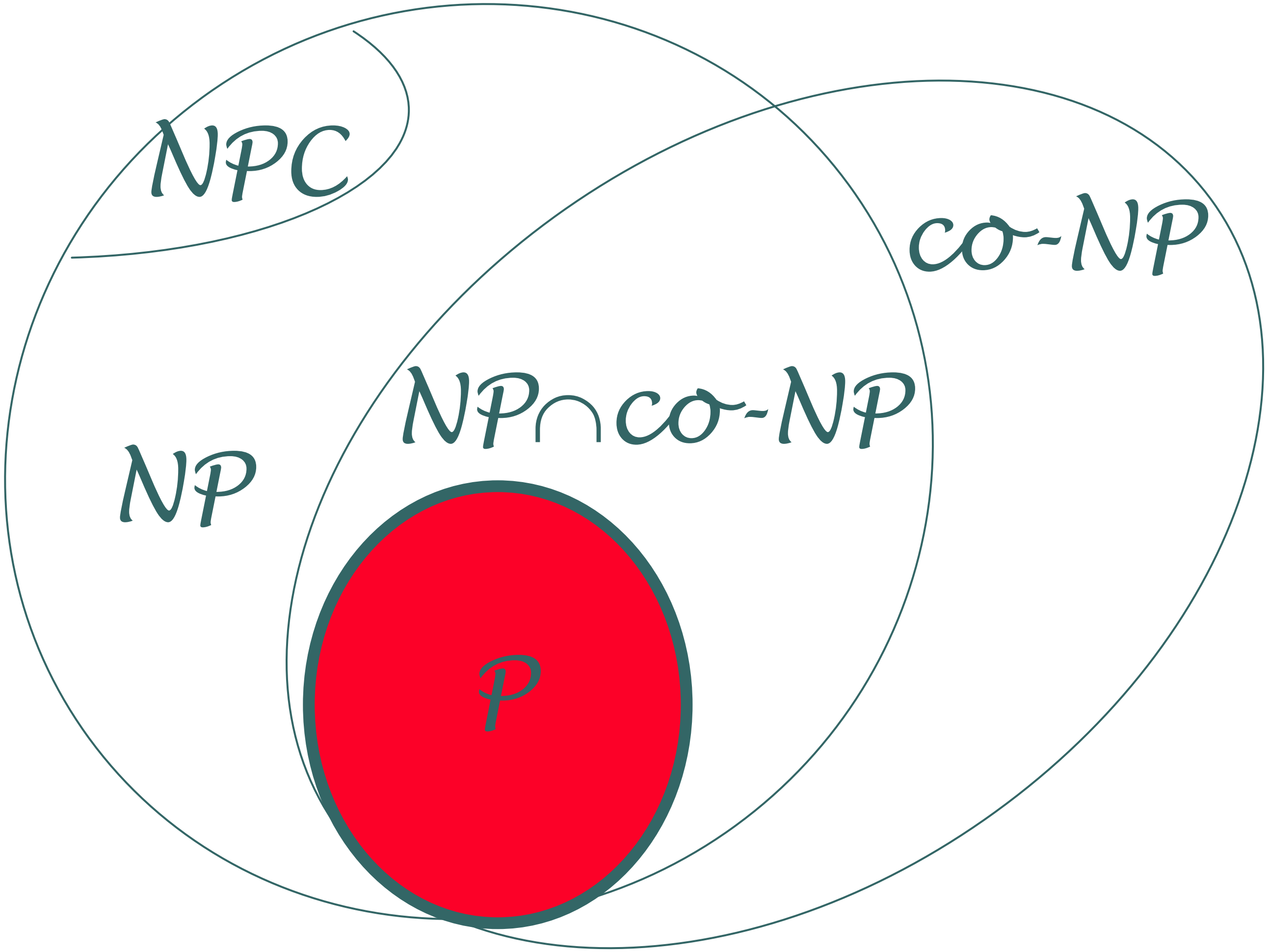
Then Π_2 is at least as hard as Π_1 (polynomially speaking)

Complete problems

- A problem Π is C-complete if any other problem from C reduces to Π
- A complete problem is ‘the hardest’ of its class.
- Nearly all classes have complete problems.

Example of complete problems

- SAT is $\mathcal{NP}$-complete
- 'Is there a path from *x* to *y* in graph *G?*' is $\mathcal{P}$-complete
- SAT of a Boolean quantified closed formula is $\mathcal{P}$-$\mathcal{SPACE}$ complete
- Equivalence between two NFA is $\mathcal{P}$-$\mathcal{SPACE}$ complete



cdlh 2007

SPACE Classes

We want to measure how much tape is needed, without taking into account the computation time.

$\mathcal{P}$-$\mathcal{SPACE}$

is the class of problems solvable by a deterministic Turing machine that uses only polynomial space.

- $\mathcal{NP} \subseteq \mathcal{P}$-$\mathcal{SPACE}$

General opinion is that the inclusion is strict.

NP-SPACE

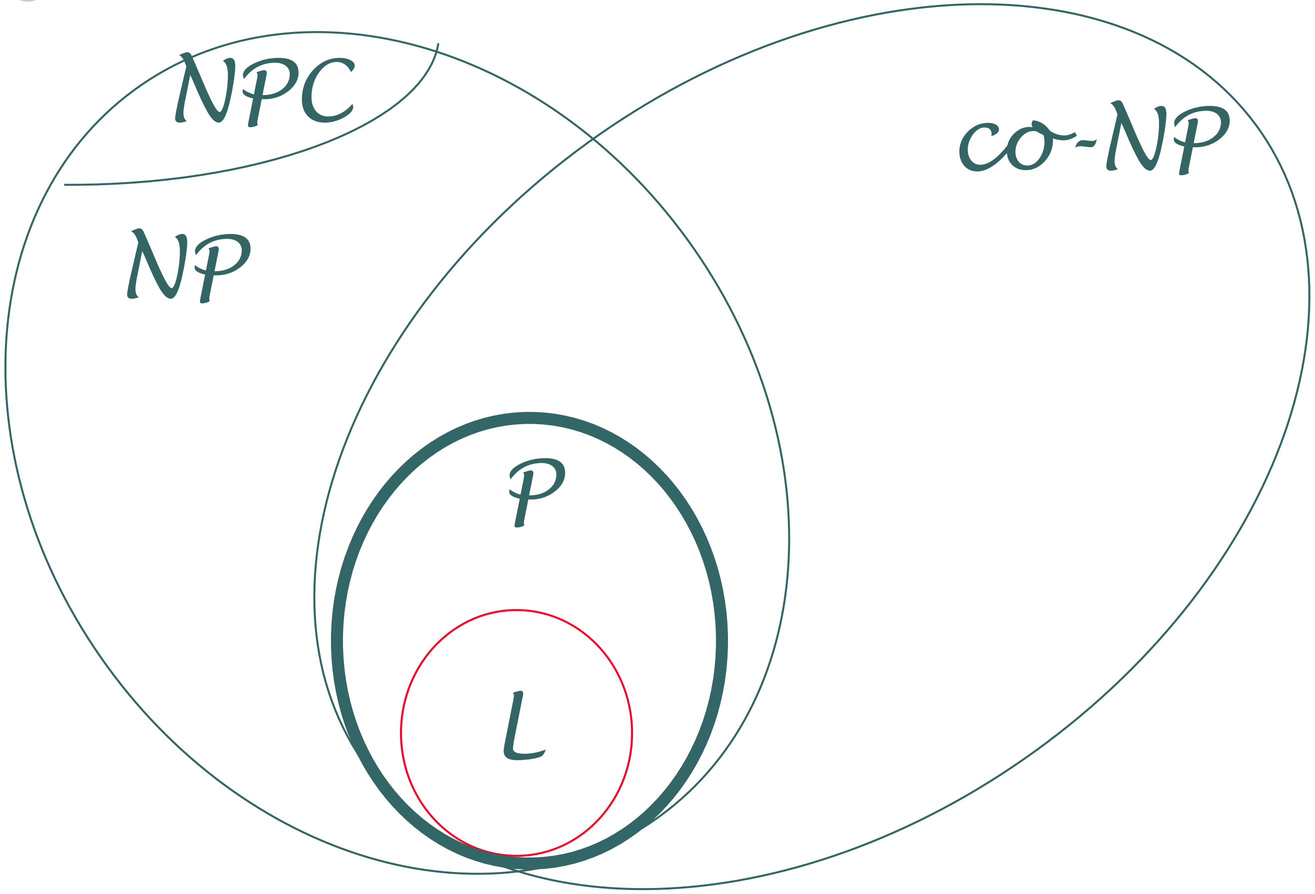
- is the class of problems solvable by a nondeterministic Turing machine that uses only polynomial space.
- Savitch theorem

  P-SPACE=NP-SPACE

log-SPACE

L=log-SPACE

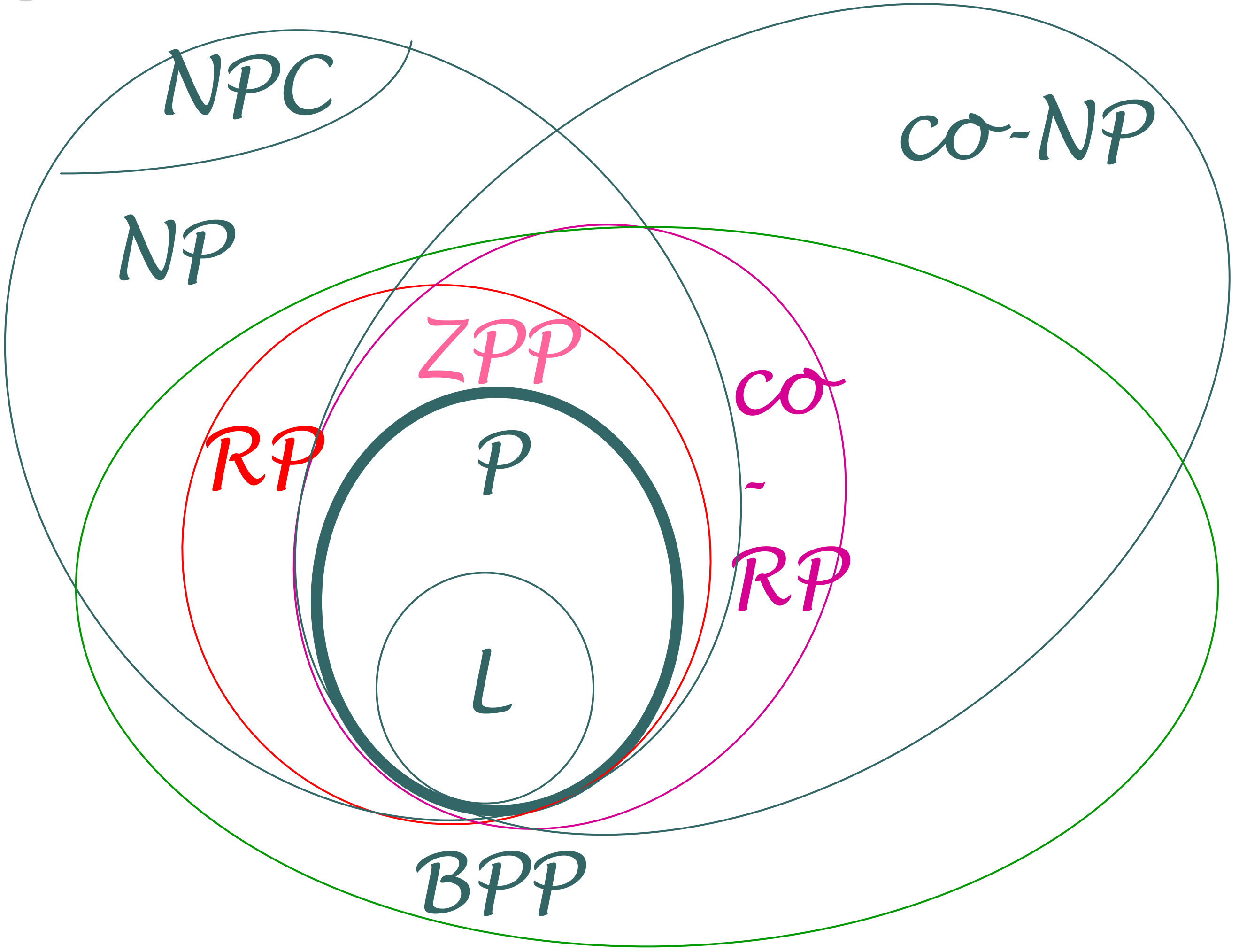
- L is the class of problems that use only poly-logarithmic space.
  - Obviously reading the *input* does not get counted.
- $L \subseteq \mathcal{P}$
  - General opinion is that the inclusion is strict.



$L \neq$ P-SPACE P-SPACE = NP-SPACE

cdlh 2007

P-SPACE= NP-SPACE



11 Stochastic classes

- Algorithms that use function `random()`
- Are there problems that deterministic machines cannot solve but that probabilistic ones can?

11.1 Probabilistic Turing machines (PTM)

- These are non deterministic machines that answer YES when the majority of computations answer YES;
- The accepted set is that of those instances for which the majority of computations give YES.
- $\mathcal{PP}$ is the class of those decision problems solvable by polynomial PTMs

$\mathcal{PP}$ is a useless class…

If probability of correctness is only $\left(\frac{1}{2}+\frac{1}{2^n}\right)$

an exponential (in *n*) number of iterations is needed to do better than random choice.

$\mathcal{BPP}$: *Bounded away from* $\mathcal{P}$

- $\mathcal{BPP}$ is the class of decision problems solvable by a PTM for which the probability of being correct is at least $1/2+\delta$, with δ a constant>0.
- It is believed that $\mathcal{NP}$ and $\mathcal{BPP}$ are incomparable, with the $\mathcal{NP}$-complete in $\mathcal{NP}\backslash\mathcal{BPP}$, and some symmetrical problems in $\mathcal{BPP}\backslash\mathcal{NP}$.

Hierarchy

- $\mathcal{P} \subseteq \mathcal{BPP} \subseteq \mathcal{BQP}$
- $\mathcal{NP}\text{-}\mathit{complete} \cap \mathcal{BQP} = \emptyset$
- Quantic machines should not be able to solve $\mathcal{NP}$-hard problems

11.2 Randomized Turing Machines (RTM)

These are non deterministic machines such that

- either no computation accepts
- either half of them do

(instead of half, any fraction >0 is OK)

$\mathcal{RP}$

- $\mathcal{RP}$ is the class of decision problems solvable by a RTM
- $\mathcal{P} \subseteq \mathcal{RP} \subseteq \mathcal{NP}$
- Inclusions are believed to be strict
- Example: *Composite* $\in \mathcal{RP}$

An example of a problem in $\mathcal{RP}$

Product Polynomial Inequivalence

- 2 sets of rational polynomials

$$P_1 \ldots P_m$$

$$Q_1 \ldots Q_n$$

- Answer : YES when $\prod_{i \leq m} P_i \neq \prod_{i \leq n} Q_i$

This problem seems neither to be in $\mathcal{P}$ nor in $co\text{-}\mathcal{NP}$.

Example

- $(x-2)(x^2+x-21)(x^3-4)$
- $(x^2-x+6)(x+14)(x+1)(x-2)(x+1)$
- Notice that developing both polynomials is too expensive.

$\mathcal{ZPP}=\mathcal{RP}\cap$ *co*-$\mathcal{RP}$

- *$\mathcal{ZPP}$: Zero error probabilistic polynomial time*
- Use in parallel the algorithm for $\mathcal{RP}$ and the one for *co*-$\mathcal{RP}$
- These algorithms are called 'Las Vegas'
- They are always right but the complexity is in average polynomial.

12 Stochastic Algorithms

‘Monte-Carlo’ Algorithms

- Negative instance $\Rightarrow$ answer is NO
- Positive instance $\Rightarrow$ Pr(answer is YES) > 0.5
- They can be wrong, but by iterating we can get the error arbitrarily small.
- Solve problems from $\mathcal{RP}$

‘Las Vegas’ algorithms

- Always correct
- In the worse case too slow
- In average case, polynomial time.

Another example of 'Monte-Carlo' algorithm

Checking the product of matrices.

Consider 3 matrices A, B and C

Question $AB \neq C$?

Natural idea

- Multiply *A* by *B* and compare with *C*
- Complexity
  - $O(n^3)$ brute force algorithm
  - $O(n^{2.37})$ Strassen algorithm
- But we can do better!

Algorithm

generate S, bit vector	$O(n)$
compute $X=(SA)B$	$O(n^2)$
compute $Y=SC$	$O(n^2)$
If $X \neq Y$ return TRUE else return FALSE	$O(n)$

Example

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad B = \begin{pmatrix} 3 & 1 & 4 \\ 1 & 5 & 9 \\ 2 & 6 & 5 \end{pmatrix}$$

$$C = \begin{pmatrix} 11 & 29 & 37 \\ 29 & 65 & 91 \\ 47 & 99 & 45 \end{pmatrix}$$

$$(1,1,0)\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{pmatrix}=(5,7,9)$$

$$(5,7,9)\begin{pmatrix}3 & 1 & 4\\ 1 & 5 & 9\\ 2 & 6 & 5\end{pmatrix}=(40,94,128)$$

$$(1,1,0)\begin{pmatrix}11 & 29 & 37\\ 29 & 65 & 91\\ 47 & 99 & 45\end{pmatrix}=(40,94,128)$$

$$(0,1,1)\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{pmatrix} = (11, 13, 15)$$

$$\begin{pmatrix}3 & 1 & 4\\ 1 & 5 & 9\\ 2 & 6 & 5\end{pmatrix} = (76,166,236)$$

$$(0,1,1)\begin{pmatrix}11 & 29 & 37\\ 29 & 65 & 91\\ 47 & 99 & 45\end{pmatrix} =(76,164,{\color{red}1}36)$$

Proof

- Let $D = C - AB \neq 0$
- Let V be a wrong column of D
- Consider a bit vector S,

if $SV=0$, then $S'V \neq 0$ with

$S' = S$ **xor** $(\underbrace{0\ldots0, 1, 0\ldots0}_{i-1})$

- Pr(*S*)=Pr(*S'*)
- Choosing a random *S*, we have $SD \neq 0$ with probability at least 1/2
- Repeating the experiment...

Error

- If $C=AB$ the answer is always NO
- if $C \neq AB$ the error made (when answering NO instead of YES) is $(1/2)^k$ *(if k experiments)*

Quicksort: an example of ‘Las Vegas’ algorithm

- Complexity of Quicksort=O(n^2)

This is the worse case, being unlucky with the pivot choice.

- If we choose it randomly we have an average complexity O(n `log` n)

13 The hardness of learning 3-term-DNF by 3-term-DNF

- references:
  - Pitt & Valiant 1988, Computational Limitations on learning from examples 1, JACM 35 965-984.
  - Examples and Proofs: Kearns & Vazirani, An Introduction to Computational Learning Theory, MIT press, 1994

- A formula in disjunctive normal form:
- $X=\{u_1,..,u_n\}$
- $F=T_1 \vee T_2 \vee T_3$
- each T_i is a conjunction of literals

sizes

- An example: $\underbrace{<0,1,\ldots.0,1>}_{n}$
- a formula: max 9*n*
- To efficiently learn a 3-term-DNF, you have to be polynomial in: $1/\varepsilon$, $1/\delta$, and *n.*

Theorem:

If $\mathcal{RP} \neq \mathcal{NP}$ the class of 3-term-DNF is not polynomially learnable by 3-term-DNF.

Definition:

A hypothesis h is consistent with a set of labelled examples $S=\{<x_1,b_1>,\ldots<x_p,b_p>\}$, if

$$\forall x_i \in S \; h(x_i)=b_i$$

3-colouring

- Instances: a graph $G=(V, A)$
- Question: does there exist a way to colour V in 3 colours such that 2 adjacent nodes have different colours?

- Remember: 3-colouring is $\mathcal{NP}$-complete

Our problem

- Name: 3-term-DNF consistent
- Instances: a set of positive examples *S*+ and a set of negative examples *S*-
- Question: does there exist a 3-term-DNF consistent with *S*+ and *S*-?

Reduce 3-colouring to « consistent hypothesis »

Remember:

- Have to transform an instance of 3-colouring to an instance of « consistent hypothesis »
- And that the graph is 3 colourable ***iff*** the set of examples admits a consistent 3-term-DNF

Reduction

build from $G=(V, A)$: $S_G+ \cup S_G-$

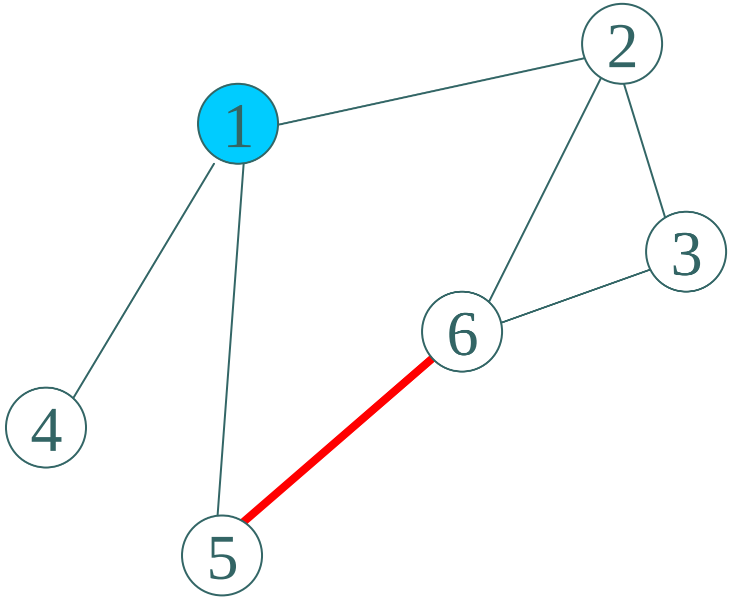
$\forall i \in [n]$ $<v(i),1> \in S_G+$ where $v(i)=(1,1,..,1,0,1,..1)$

i

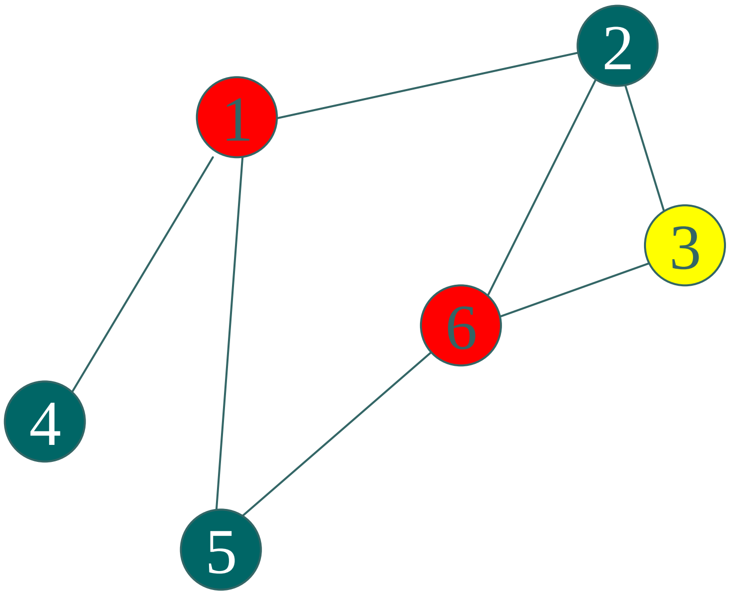
$\forall (i, j) \in A$ $<a(i, j),0> \in S_G-$

where $a(i, j)=(1,..,1,0,...,0,1,..,1)$

i j

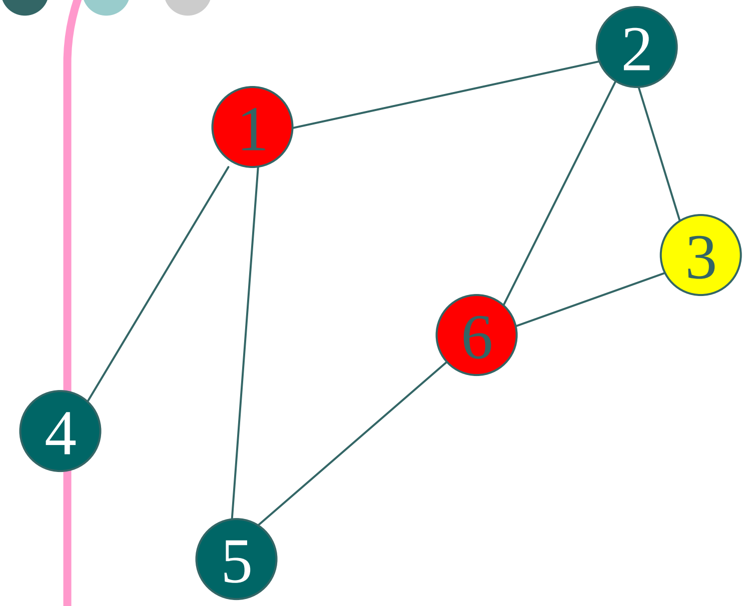


S_G+	S_G-
(011111, 1)	(001111, 0)
(101111, 1)	(011011, 0)
(110111, 1)	(011101, 0)
(111011, 1)	(100111, 0)
(111101, 1)	(101110, 0)
(111110, 1)	(110110, 0)
	(111100, 0)



S_G+	S_G-
(011111, 1)	(001111, 0)
(101111, 1)	(011011, 0)
(110111, 1)	(011101, 0)
(111011, 1)	(100111, 0)
(111101, 1)	(101110, 0)
(111110, 1)	(110110, 0)
	(111100, 0)

$T_{yellow}=x_1\wedge x_2\wedge x_4 \wedge x_5\wedge x_6$
$T_{blue}=x_1\wedge x_3\wedge x_6$
$T_{red}=x_2\wedge x_3\wedge x_4\wedge x_5$



S_G+	S_G-
(011111, 1)	(001111, 0)
(101111, 1)	(011011, 0)
(110111, 1)	(011101, 0)
(111011, 1)	(100111, 0)
(111101, 1)	(101110, 0)
(111110, 1)	(110110, 0)
	(111100, 0)

$T_{yellow}=x_1 \wedge x_2 \wedge x_4 \wedge x_5 \wedge x_6$

$T_{blue}=x_1 \wedge x_3 \wedge x_6$

$T_{red}=x_2 \wedge x_3 \wedge x_4 \wedge x_5$

Where did we win?

- Finding a 3-term-DNF consistent is exactly PAC-learning 3-term DNF
- Suppose we have a polynomial learning algorithm *L* that learns 3-term-DNF PAC.
- Let *S* be a set of examples
- Take $\varepsilon = 1/(2|S|)$

- We learn with the uniform distribution over *S* with an algorithm *L*.
- If there exists a consistent 3-term-DNF, then with probability at least $1-\delta$ the error is less than ε: so there is in fact no error !
- If there exists no consistent 3-term-DNF, *L* will not find anything.
- So just by looking at the results we know in which case we are.

Therefore:

- *L* is a randomized learner that checks in polynomial time if a sample *S* admits a consistent 3-term-DNF.
- If *S* does not admit a consistent 3-term-DNF *L* answers « no » with probability 1.
- If *S* admit a consistent 3-term-DNF *L* answers« yes », with probability 1-δ.
- In this case we have 3-colouring $\in \mathcal{RP}$.

Careful

- The class 3-term-DNF is polynomially PAC learnable by 3-*CNF !*

General conclusion

Lots of other TCS topics in ML.

Logics (decision trees, ILP)

Higher graph theory (graphical models, clustering, HMMs and DFA)

Formal language theory

… and there never is enough algorithmics !