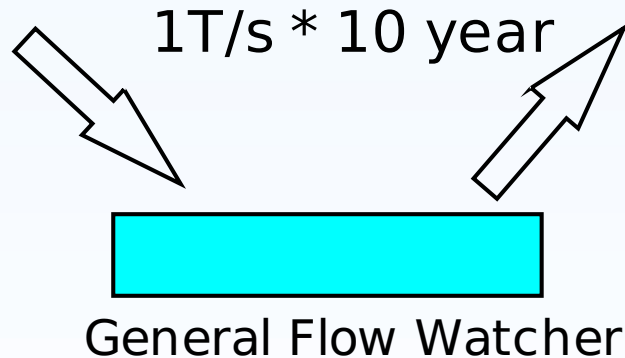


Randomization, Approximization and Optimization

Fan Haoqiang
Tsinghua, IIS

a motivating example

- suppose you are facing a large volume of network flow.
- your memory is very limited. Only $O(\log^k(N))$



and you want to count

- and you want to count the number of **distinct** packages.
- maybe counting the number of visitors to a particular site.

and MLE

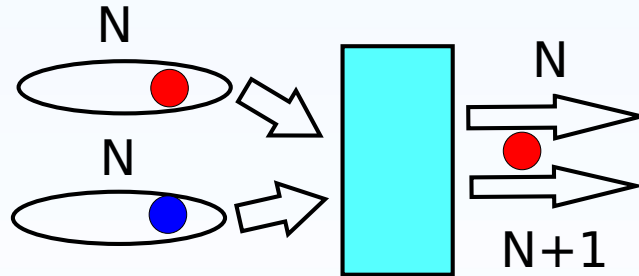
- is it ever possible given such limited memory?

negative result

- it is **not** possible to **precisely** count the number of distinct elements if you cannot store them.

negative result

- proof sketch: there must be two different sets of packages that lead the device to the same state (pigeon-hole principle). And it will fail because of its failure to distinguish between the two sets.



how to make things work?

- what if we only care about an **approximation** of the number of distinct packages?
- useful in a fistful of cases.

still negative

- it is even not possible to **approximate** the number of packages up to a constant factor.
- ϵ -approximation means for every input sequence, at any time,

$$\frac{D}{1 + \epsilon} \leq Z \leq (1 + \epsilon)D$$

where D is the number of distinct elements, and Z is the estimated number.

proof sketch

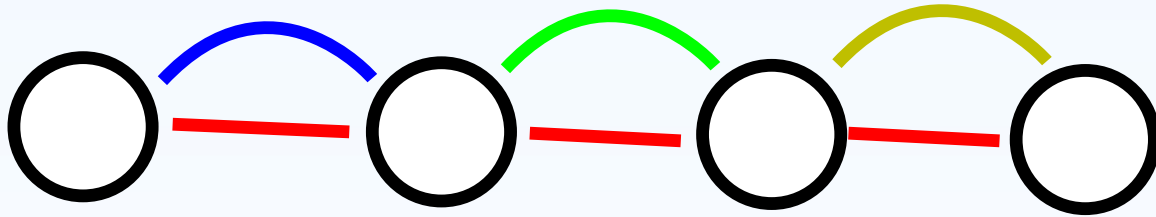
- there is no ϵ -approximation algorithm that uses $O(\log(N))$ space.
- key idea: the number of internal states of the algorithm is too small.

proof sketch

$n/2$ -dim subspaces in $\text{GF}(p)^n$

heavy collision

sparse subsets



$\text{poly}(N)$ states

unbounded approx. ratio

how to make it possible?

- here randomization comes into play.
- we can accept small **probability** of failure. The probability should only depend on the coin tosses made by us.

randomized approximation

- a randomized algorithm is said to be (ϵ, δ) approx. iff

$$\Pr\left[\frac{D}{1 + \epsilon} \leq Z \leq (1 + \epsilon)D\right] \leq 1 - \delta$$

- is it possible to construct a **randmoized** algorithm?

our first attempt

- let's play with our first idea: Bitmap Algorithm.
- a good hash function $h : \{1 \dots N\} \rightarrow \{1 \dots n\}$



record all hashes encountered

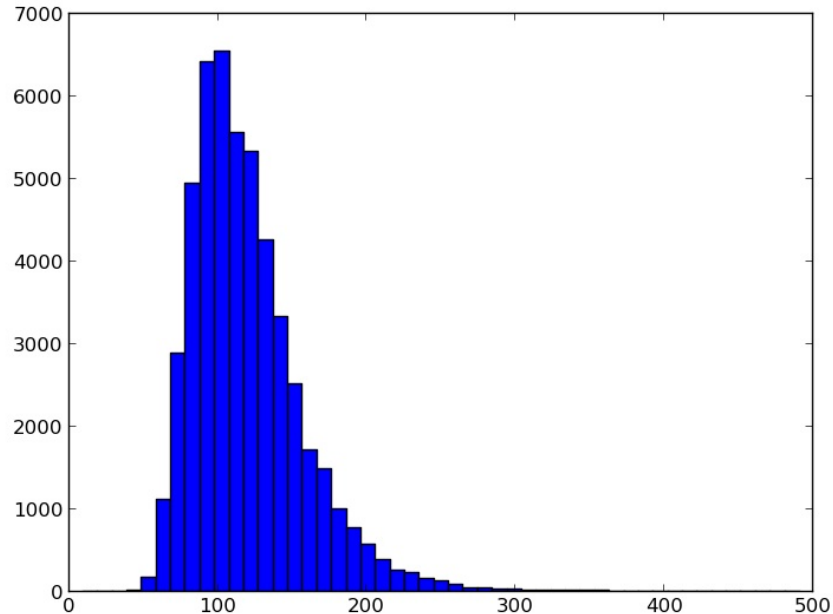
coupon collector problem

- how many coupons do you have to collect before having z out of n different types of coupons?
- easy calculation of expectation gives:

$$\sum_{i=1}^z \frac{n}{n+1-i} \approx n \ln\left(\frac{n}{n-z}\right)$$

coupon collection

- how good is the approximation?
- not so sharp, but good enough

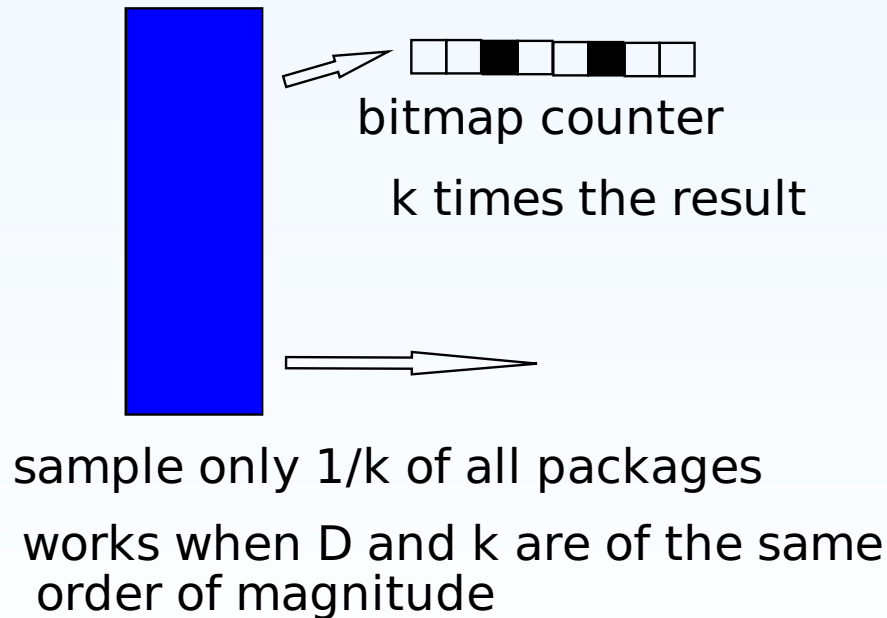


coupon collector problem

- only works when z is small compared to n
- so, we only saved $O(\log(N))$ space instead of the desired $O(\frac{N}{\log(N)})$
- anyway to improve this idea?

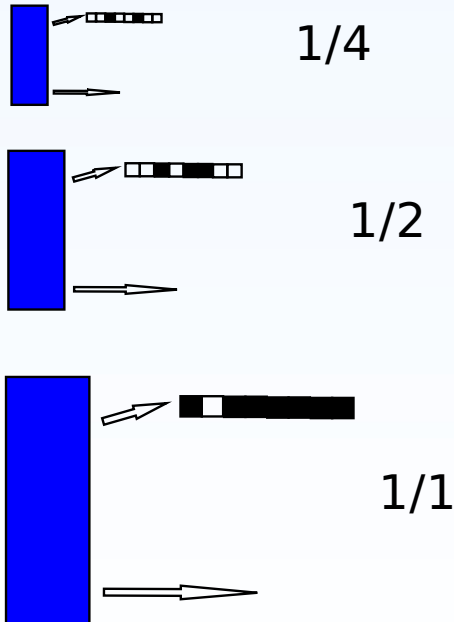
reduce the number of elements

- add a random sampler to make the number of elements bounded.



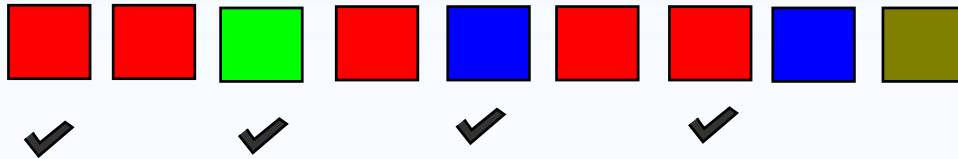
multi-resolution

- what if we don't know how big D is in advance?
- build filters with different sampling factors and use the most accurate one.



sampling

- Does the sampling technique really works?
- Negative: sampling half of the data does not mean getting half of the distinct elements.



how to make things work?

- the crux: if an element is seen multiple times, its probability of being chosen accumulates.



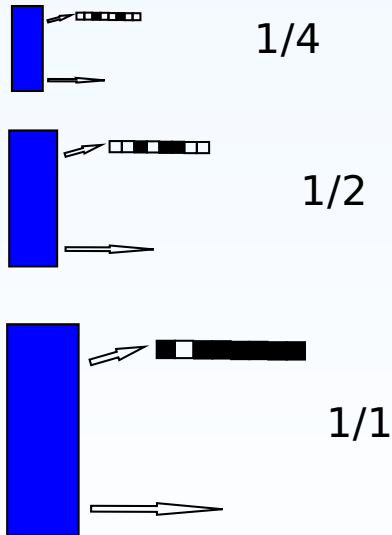
these guys should not have any effect

another way of sampling

- Base our decision on the element itself.
- Choose x iff $\text{hash}(x) < 1/k$
- Now, the number of distinct elements scales linearly with k

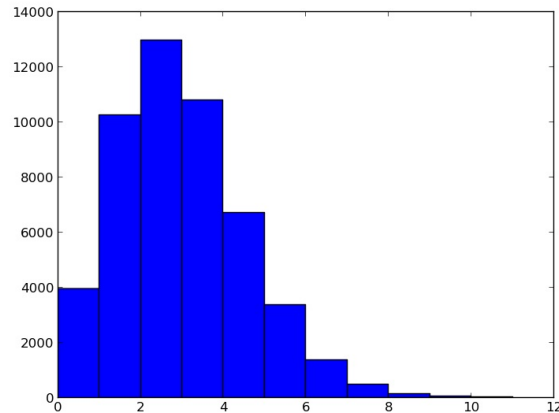
and it works

- multi-resolution sampling + bitmap algorithm
- $O(\log^2(N))$ space with $O(1)$ approximation ratio. nearly reaches our goal.



law of small numbers

- if something happens with prob. $\frac{1}{k}$, and you have nk of them, what's the distribution of the number of events really happened?
- Poisson distribution



wait a minute

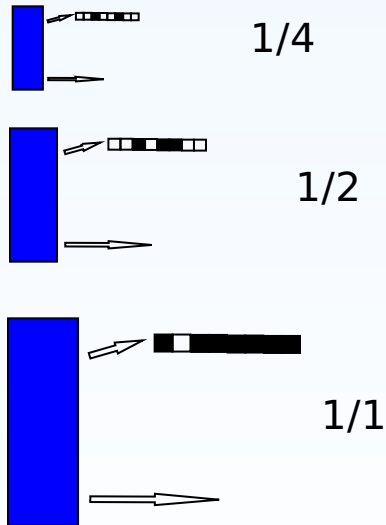
- when you have a strong sense that your algorithm will work, it is time to analyze it formally.

the bitmap

- how large should the bitmap be?
- what's the roll of bitmaps in our algorithm?

simplified model

- the part that really works is not the bitmap; the multi-resolution idea is.



make it simple

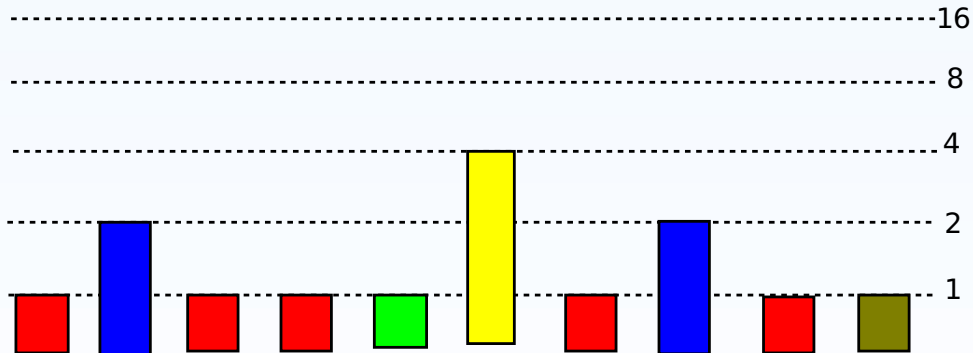
- what if the bitmap has only one bin?
- another algorithm

simplified algorithm

- for each element encountered, compute $\text{hash}(x)$
- the filter at level n accepts the element if its hash is less than 2^{-n}
- find the highest level z that an element ever reaches. output 2^z
- uses only $O(\log(N))$ space now!

intuition

- intuitively, this algorithm should give an $O(1)$ approximation ratio.



intuition

- an element have 2^{-n} chance of passing the n th filter
- if we really see one such element, it should be backed up by 2^n other elements.
- how to make this argument rigid

Bernoulli trail

- suppose you have done D experiments, each of which has 2^{-n} probability of giving positive outcome.
- what's your chance of getting at least one positive outcome? what's your change of getting no positive outcome?
- $(1 - 2^{-n})^D \approx e^{-\frac{D}{2^n}}$

our first analysis

- if the (exact) number of distinct elements is D
- the chance that $2^{(z+1)} < D$ is
$$e^{-D/2^z} \leq 0.14$$
- the chance that $2^{(z-1)} > D$ is
$$1 - e^{-D/2^z} \leq 0.4$$
- so we have a very good chance of giving an output in the range $[D/2, 2D]$

amplify the probabilities

- it there a way to improve our chance of giving good estimation (without changing the space complexity)?
- repeat (in parallel) many instances of our algorithm

and take the median

- take the median (instead of the average).
- the probability that the median is less than $D/2$

is

$$\sum_{i=0}^{T/2} \binom{T}{i} 0.14^{T-i} 0.86^i \leq e^{-0.075 T}$$

- as T increases, the probability approaches zero.
- the same applies to $Z > 2D$

probability amplification

- so we only have to make sure that the one-side failure rates are smaller than 0.5 by a constant term.
- works well in practice.

wait a minute

- there is a bug in our analysis (and also in the analysis of many other algorithms).
- our source of randomness

hash functions

- we require that the hash values are

independently **i**dentically **d**istributed

- ideally, the hash should be something like:

- ```
if x is not encountered
 fetch a number from /dev/random
else
 return the value we have assigned to x
```

# however

- but we don't have enough space to remember all previously assigned values.
- so how things really work is:
- randomly choose some parameters at the beginning  
use these parameters to compute the hash function

# information theoretic

- so long as the length of the parameters is  $o(N \log N)$ , the hash values are not independent

# take an example

- say our hash is

$$\text{hash}(x) = x * a \bmod N$$

- then  $\text{hash}(0)$  is always 0.
- $\text{hash}(0)$  and  $\text{hash}(N/2)$  has a very good chance of colliding.



# fix the bug

- $\text{hash}(x) = a * x + b \mod P$
- $P$  is a (big) prime number.  $a$  and  $b$  are randomly chosen.
- given  $\text{hash}(0)$  and  $\text{hash}(1)$ , the hash function is uniquely determined.

# what to do

- now that the hash function is not purely random. how to base our analysis on such highly correlated hash values?
- find some guarantees that hold.

# pairwise independent

- a nice property of our (linear) hash function:
- if we only look at two hash values  
 $\text{hash}(x_1)$  and  
 $\text{hash}(x_2)$ ,  
they are i.i.d. for any choices of  $x_1$  and  $x_2$ .
- pairwise independence does not mean global independence, but it has been good enough.

# weaker bound

- suppose you conducted  $D$  experiments, each of which has probability  $2^{-z}$  of giving positive outcome, and they are **pairwise** independent.
- what's your chance of getting at least one positive output?
- what's your chance of not getting any positive output?

# Chebyshev bound

- base our analysis on expectation and variation.
- $\Pr[\sum_i X_i \geq 1] \leq \sum_i E[X_i]$
- $\Pr[|X - E[X]| \geq \delta \text{std}[X]] \leq \frac{1}{\delta}$

# the conclusion

- let's forget about the math
- $\Pr[3 \cdot 2^{(z+0.5)} < D] \leq \text{var}[\sum X_i] / E[\sum X_i]^2 \leq \sqrt{2}/3 \leq 0.472$
- $\Pr[2^{(z+0.5)} / 3 > D] \leq E[\sum X_i] \leq \sqrt{2}/3 \leq 0.472$
- so the conclusion is that we have at least 0.057 chance of getting an output in  $[D/3, 3D]$
- the probability can be amplified to arbitrarily high value.
- but the constant 3 cannot be improved by sheer repeating.

# improve the accuracy

- is there a way to improve the accuracy?
- finer grained filters?
- $1, 1+a, (1+a)^2, \dots, (1+a)^n$

# oh, no

- not working.
- our previous approach works because  $\sqrt{2}/3 < 0.5$
- 2 is a magic number that cannot be improved.



# another view

- only the element with the minimum hash value matters.
- in expectation, the minimum of  $D$  i.i.d. variables is  $1/(1 + D)$

# another way

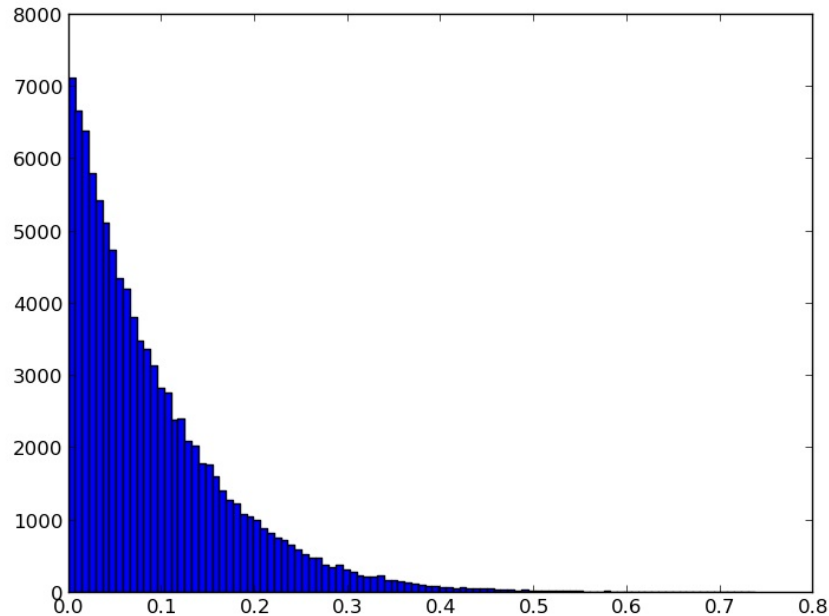
- so here comes our next algorithm
- maintain  $z$  the minimum of the hash values encountered  
output  $1/z - 1$
- repeat many times, take the median

# does it work?

- a somehow enhanced version of our first algorithm
- it is guaranteed to be not worse than the previous algorithm by a constant factor.
- does the accuracy scales with number of instances?

# however

- no, if we only take the median
- the distribution is skew



# do not give up

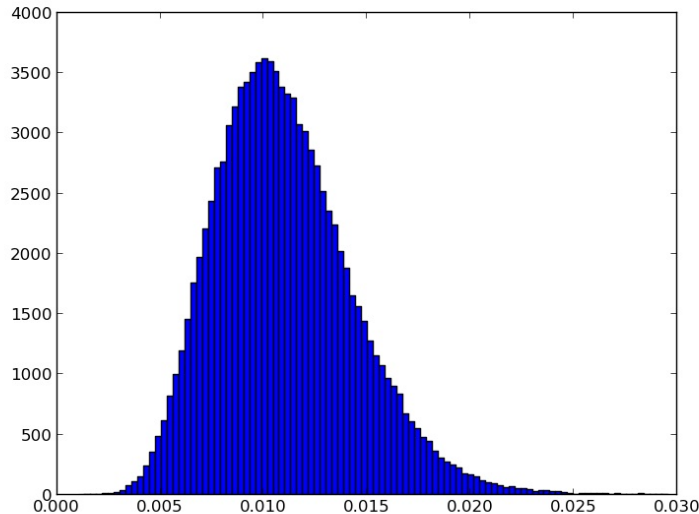
- repeat in another way
- what if we keep the  $t$ th smallest hash value?

# and it works!

- let  $\epsilon$  be the desired relative accuracy
- set  $t = \left\lceil \frac{\Theta(1)}{\epsilon^2} \right\rceil$
- $D$  events, each occurs with prob.  $Dt/(1 + \epsilon)$ .
- chance that no more than  $D$  events occurs is (using Chebyshev):  $O(1)/(\epsilon^2 t) = O(1)$
- the same applies to the other side

# much better distribution

- at the expense of increased space
- but the space complexity remains  $O(\log(N))$  for any given  $\epsilon$  and  $\delta$



# the end of the story?

- what's the intuition behind the  $t$ th smallest number?
- bitmap with  $O(t)$  bins
- these two ideas lead to the same algorithm



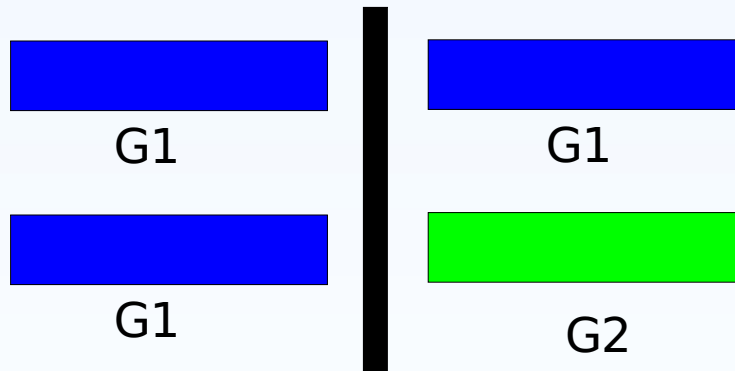
# what to do

- what else do we have to do after devising an algorithm?
- show that it is optimal

# the lower bound

- communicate from the past to the present

enough space to distinguish between G1 and G2



# coding theory

- There exists  $2^{\Omega(N)}$  subsets of  $\{1 \dots N\}$ , each with cardinality  $N/4$ , such that any two subsets have at most  $N/8$  elements in common.
- counting argument: consider how many subsets can one subset rule out.

# communication complexity

- if you can approximately count the number of distinct elements to arbitrarily good precision, you can differentiate  $G1$  from  $G2$
- results from communication complexity says that at least  $\Omega(\log(N))$  space is required.

# comments

- hash functions are far from perfect. find solid base for our analysis
- hash-based data-structures?

# treap

- requires that the chance of one node having the smallest weight among  $t$  nodes is  $1/t$
- it is not hashing. `/dev/random` will do the job

# hashing strings

- polynomial hashing  $\text{hash}(A) = \sum_i A_i x^i$
- if the computation is done in a field (mod  $p$ ),  $\text{hash}(A) = \text{hash}(B)$  has at most  $\max\{\text{len}(A), \text{len}(B)\}$  solutions.
- choosing  $p > m^3$  suffices to guarantee that all of  $m$  strings have different hash values.
- if we require that all sub-strings of a string of length  $m$  do not collide,  $p$  should be even greater.

# over a ring

- what if hashing is done over a ring?
- mod  $2^{32}$ , a common practice



# counterexample

- 01101001100101101001011001101001...
- collide with the inverse of itself for all odd bases

# counterexample

- if  $h(u)$  and  $h(\bar{u})$  collided at the lowest  $n$  bits
- $h(u)p^k + h(\bar{u})$  and  $h(\bar{u})p^{2^k} + h(u)$  collided  $\Theta(k^2)$  more bits.
- string of modest length is able to fail mod  $2^{64}$  hash.

# more on randomized algorithms

- Monte Carlo and Las Vegas
- examples

# polynomial identity testing

- is  $a^3 + b^3 + c^3 - 3abc = (a + b + c)(a^2 + b^2 + c^2 - ab - bc - ca)$ ?
- a polynomial expression of degree at most  $d$
- see if it is equal to zero (after expanding)

# Monte Carlo

- choose a sufficiently **large** field.
- randomly sample numbers
- if the polynomial is non-zero, the chance that it evaluates to zero is at most  $\frac{d}{p}$
- proved by induction on the number of variables and the degree of the polynomial.

# Monte Carlo

- no deterministic algorithms known yet

# associativity test

- given an operator  $f : S \times S \rightarrow S$ , see if it has associativity.
- brute force algorithm runs  $O(|S|^3)$

# randomized algorithm

- let  $X_\alpha$  be numbers in a field.
- define the convolution  $Z = X \oplus Y$

$$Z_\alpha = \sum_{f(\beta, \gamma) = \alpha} X_\beta Y_\gamma$$

- if  $f$  has associativity,  
 $(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$
- and with high probability, if  $X, Y, Z$  are randomly sampled, vice versa.



# give me a proof

- look closely at what we are doing.
- reduction to PIT

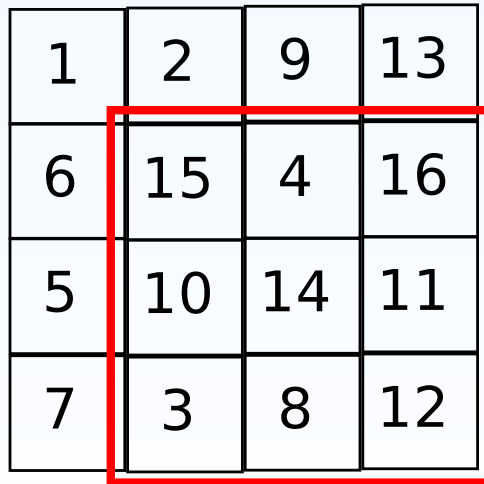
# and more?

- distributive test?
- test if a subring is an ideal?
- normal subgroup?

# median in rectangle

- in an  $H \times W$  number matrix, find one  $R \times R$  sub-matrix with the greatest median.

|   |    |    |    |
|---|----|----|----|
| 1 | 2  | 9  | 13 |
| 6 | 15 | 4  | 16 |
| 5 | 10 | 14 | 11 |
| 7 | 3  | 8  | 12 |

A 4x4 matrix of numbers. A 2x2 sub-matrix is highlighted with a red border. The sub-matrix consists of the elements 15, 4, 10, and 14, which are located at the intersection of the second and third rows and the second and third columns of the main matrix.

# baseline

- $O(HW \log(HW))$  algorithm based on binary search.

# linear time algorithm

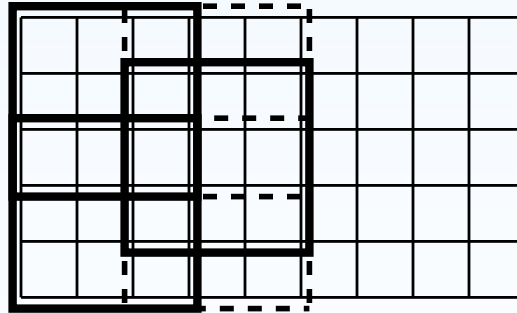
- algorithm that runs in linear time
- based on random sampling
- very clever idea

# recall

- the minimum of  $n$  i.i.d.  $0 \sim 1$  random variables is  $1/(1 + n)$  in expectation.
- we have an efficient way of filtering out rectangles whose median is smaller than a threshold.
- need a way to quickly handle small number of rectangles.

# compression

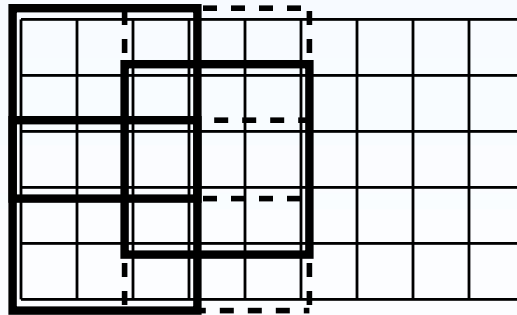
- $K$  rectangles can be done in  $O(K^2 \log(HW) + HW)$  time.



# compression

- use a list to maintain the *HW* numbers.
- during each iteration of the binary search, the list halves.

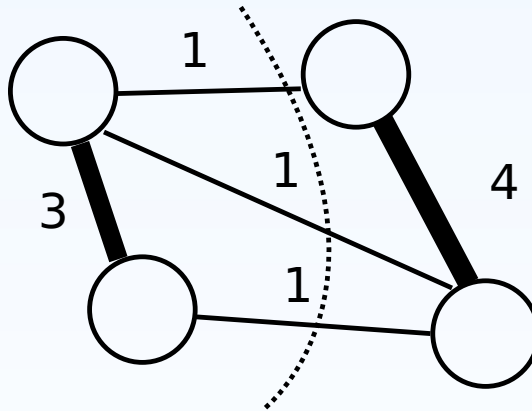
1 2 3 4 5 6 7 8 9  
1 2 3 4 5 6 7 8 9





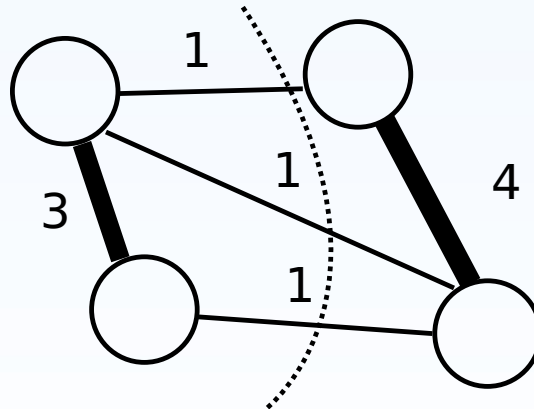
# Las Vegas

- randomized global min-cut



# min cut

- choose an edge with prob. proportional to its cost, merge its two endpoints
- repeat until there is only one edge.

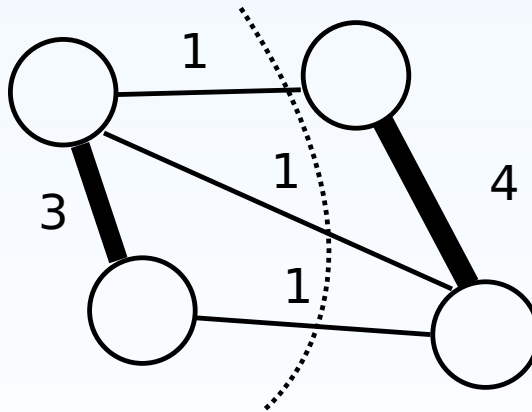


# analyzing the algorithm

- intuitively, this algorithm doesn't seem to have a good chance of success
- even if we repeated it multiple times.
- how to bound on the probability of giving the correct output in a single run?

# analyzing the algorithm

- look at one minimum cut
- compute its chance of surviving



# key observation

- the probability of destroying a cut in one round is proportional to its cost.
- $\Pr[\text{destroyed}] = \frac{\text{min cut}}{\text{cost of all edges}} \leq \frac{2}{|V|}$
- min-cut is smaller than the average cost of cutting a node.

# telescoping product

- so, the chance that the cut will survive till the end is:

- $\frac{1}{3} \cdot \frac{2}{4} \cdot \frac{3}{5} \cdot \dots \cdot \frac{|V| - 2}{|V|} = \Theta\left(\frac{1}{|V|^2}\right)$

- so, we need to repeat it  $O(|V|^2)$  times.
- this lead to an  $O(|V|^4)$  algorithm

# implementation detail

- how to run one instance of the algorithm in  $O(|V|^2)$  time?
- use adjacency matrix to store the graph
- maintain the total weight of edges incident to an edge
- two step sampling: choose a node, then choose an edge

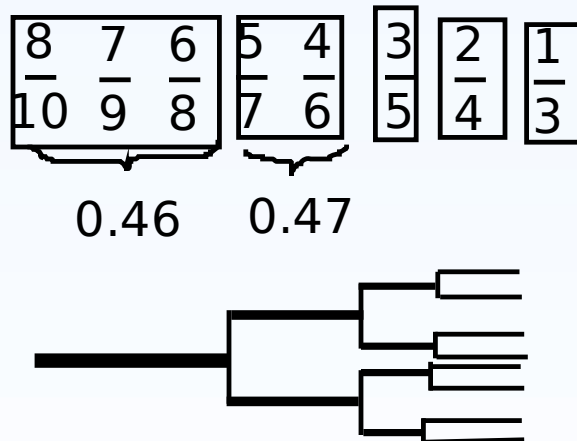
# beat the baseline

- our baseline algorithm is to run network flow  $O(|V|)$  times.
- the current state-of-art network flow algorithm runs in  $O(|V| \cdot |E|)$
- is there a way to improve the randomized algorithm?



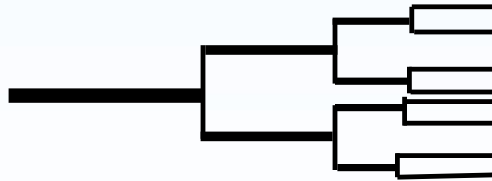
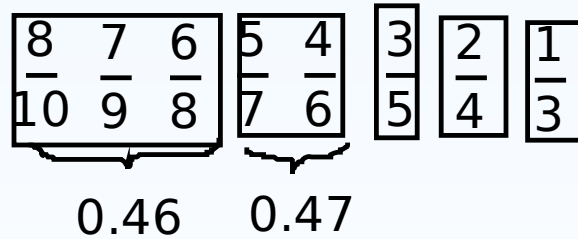
# heuristic that works

- we are more likely to fail at the second half of the execution.



# recursive

- when the graph's size is reduced by  $\sqrt{2}$ , run two instances of the rest of the algorithm
- and choose the better one



# the overall complexity

- $T(N) = 2T(N/\sqrt{2}) + O(N^2)$
- $P(N) = \frac{1}{2}(1 - (1 - P(N/\sqrt{2}))^2)$
- $O(N^2 \log^{O(1)} N)$  in general

# yet another example

- the MST
- linear algorithm

# MST

- idea: what if we randomly sample the edges?
- recursive application of the algorithm.
- and use MST of the subgraph to prune remaining edges

# after pruning

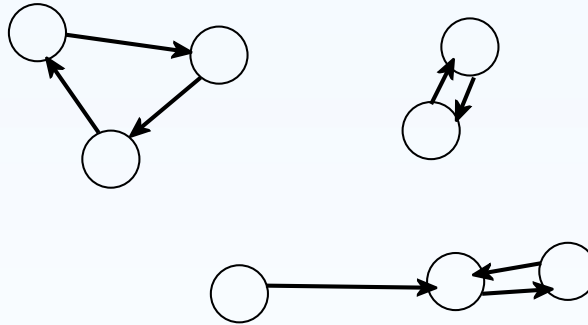
- consider Kruskal's algorithm
- add edges in order of weight from lightest to heaviest
- $N - 1$  edges will be actually added
- $2(N - 1)$  edges will be considered (and discarded with prob. 0.5)

# make it work

- after pruning, we have only  $O(N)$  edges left.
- the pruning step can be implemented in linear time using some data-structures
- work needed to make it really work
- how to reduce the number of nodes?

# Boruvka's step

- find the lightest out going edge for each node
- form some cycles. reduce the number of nodes by half in the worst case





# Boruvka's step

- $T(V, E) \leq T(V, E/2) + T(V, 3V) + O(V + E)$   
 $T(V, E) \leq T(kV, E) + O(V + E)$
- $T(V, E) \leq T(V/4, E/2) + T(V/4, 3V/4) + O(V + E)$

# Optimization

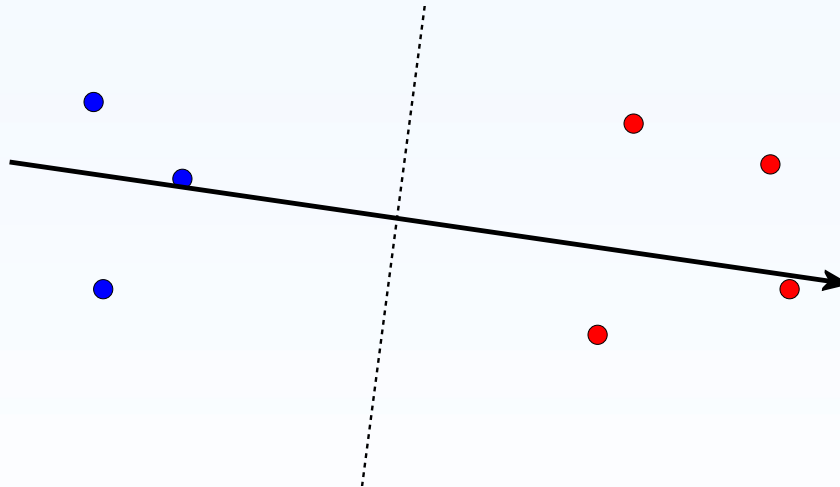
- now it is time to turn to another topic

# (numeric) optimization

- maximize  $f(x)$
- subject to some constraints

# motivation

- find a direction along which two sets of points are furthest separated.
- separating hyperplane with the largest margin

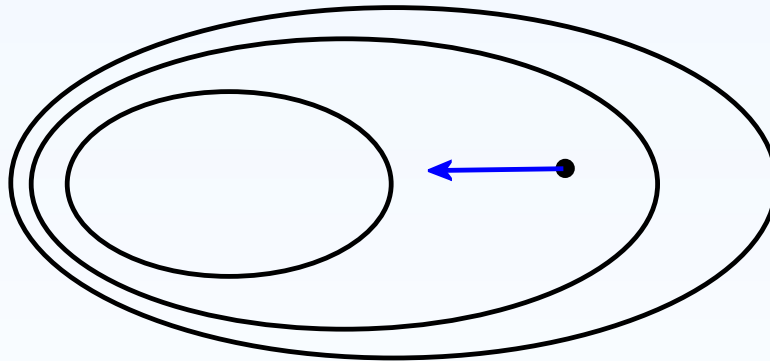


# quadratic programming

- $f(\vec{x}) = \min_{\vec{a} \in A} \vec{a} \cdot \vec{x} - \max_{\vec{b} \in B} \vec{b} \cdot \vec{x}$
- subject to  $\vec{x} \cdot \vec{x} = 1$
- convex problem

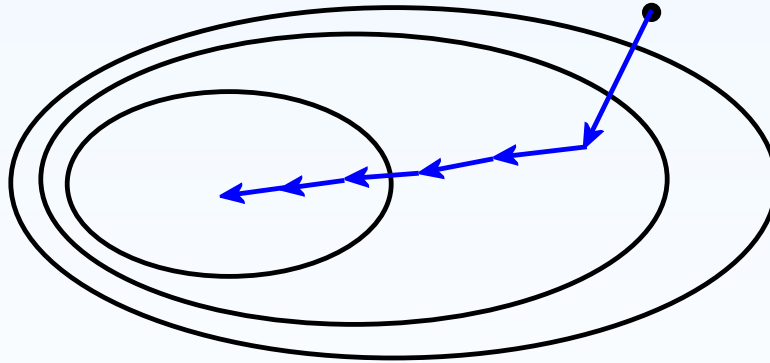
# general attack

- gradient descent



# general attack

- iterative algorithm



# some guarantees

- when the algorithm terminates, the gradient is zero, hence a local minimum is reached
- $\vec{x} \leftarrow \vec{x} + \omega \nabla f(\vec{x})$
- if  $\omega$  is small enough, the objective function is guaranteed to be improved

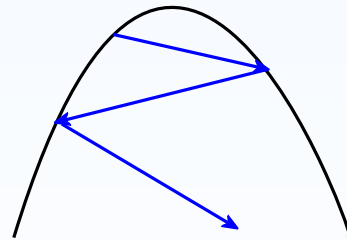
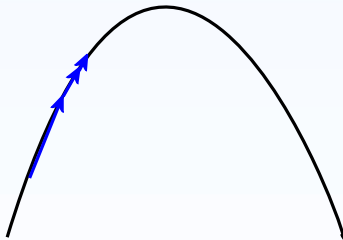


# practical issues

- how to choose the step size?
- how fast will it converge?
- how to handle constraints?

# step size

- it should be large enough so that progress is made
- it should be small enough to avoid divergence



# step size

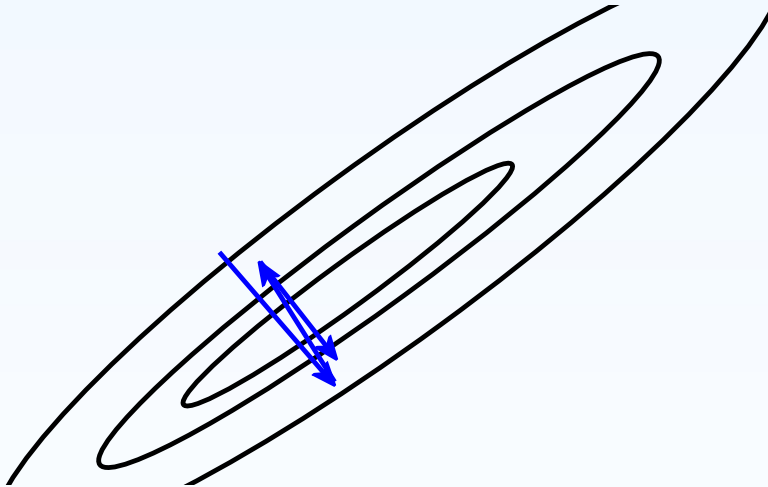
- mathematically, setting  $\omega$  inversely proportional to the current number of iterations will do the job
- in practice, extensive hand-tuning is required

# step size and speed

- consider minimizing  $f(x, y) = 100x^2 + y^2$
- start at  $(2, 2)$
- $x \leftarrow x - 100\omega x$   
 $y \leftarrow y - \omega y$
- $\omega$  has to be lower than 0.02 or  $x$  will blow up
- but convergence on  $y$  will be very slow

# step size and speed

- the eigen-value gap problem

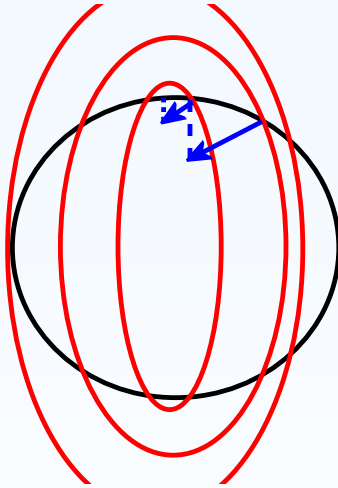


# second order methods

- use (estimated) Hessian to find the best direction
- conjugate gradient, line search

# constraints

- descent and project



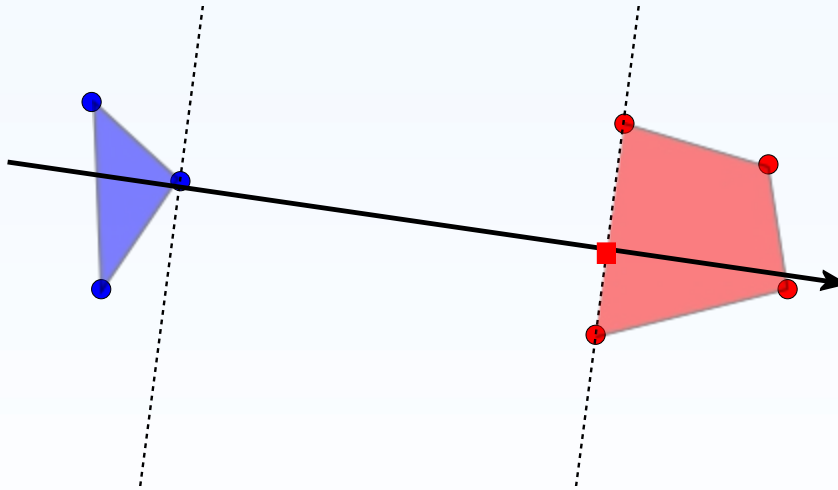
# duality

- how to get rid of the constraints?
- Lagrange multipliers
- duality



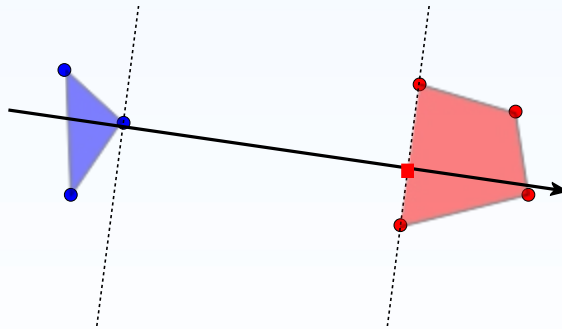
# duality

- meditate on the following fact:
- the maximum margin is the same with the minimum distance between the two polytopes



# Lagrange multipliers

- $L(\vec{x}) = f(\vec{x}) - \lambda \vec{x} \cdot \vec{x}$
- if  $\vec{x}$  is the solution to the original problem, then there is a  $\lambda$  such that  $\vec{x}$  is also the maximizer of  $L$
- consider the gradient



# dual of the program

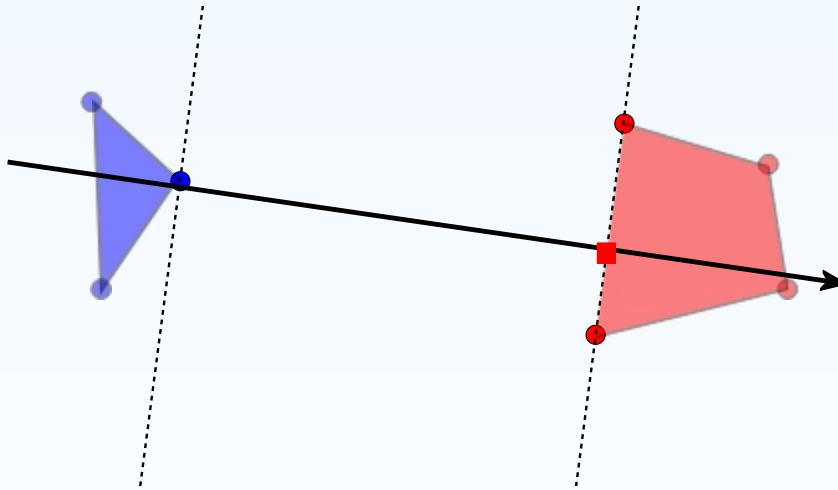
- minimize  $|A\vec{\alpha} - B\vec{\beta}|$
- subject to  $\alpha, \beta$  are non-negative, and  $\text{sum}(\alpha)=1$ ,  $\text{sum}(\beta)=1$
- these constraints are easier to handle

# coordinate descent

- if all but two variables are fixed, the optimization problem becomes trivial
- optimize two at a time

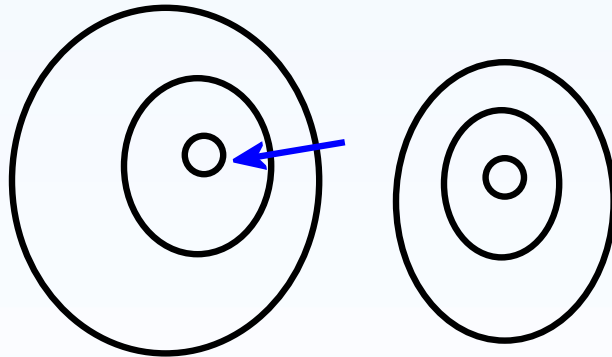
# coordinate descent

- active set



# another issue

- before applying coordinate descent to all optimization problems, be aware of its pitfall
- local minimum

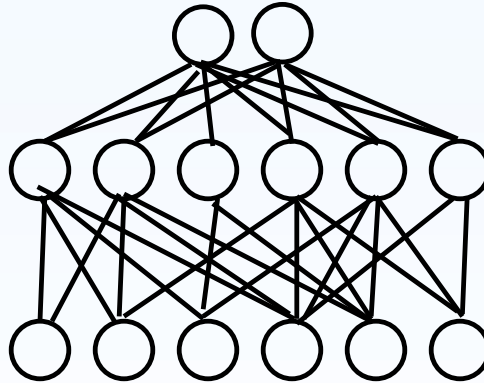


# neural networks

- paramount to the success of gradient descent algorithms

# what's a neural network

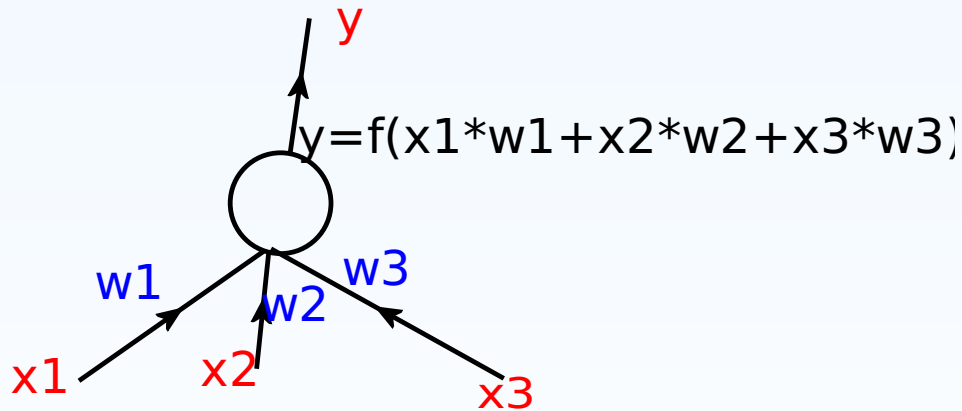
- a non-linear parametrized function family
- layered structure





# neuron

- non-linear computing unit



# training neural networks

- $y = F(\vec{x}, \vec{w})$
- given some input-output pairs, find proper values for  $\vec{w}$  such that the network's output is closest to the desired output
- $\min_{\vec{w}} \sum_{\vec{x}, y \in X, Y} (y - F(\vec{x}, \vec{w}))^2$
- a (numeric) optimization problem

# very hard

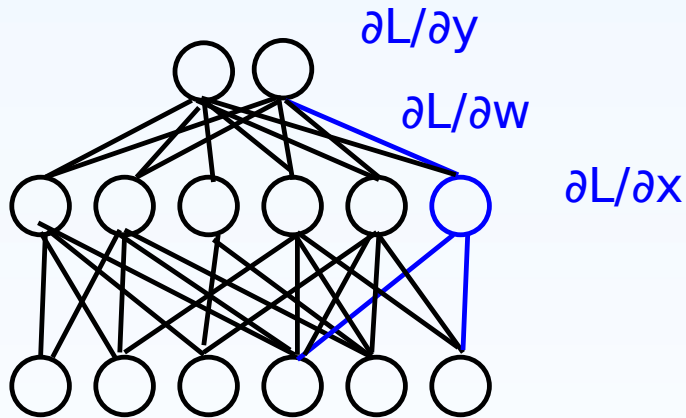
- full of local-minima due to its non-linearity
- NP-hard for a wide range of networks
- virtually impossible to compute the optimal solution

# gradient descent

- the way neural networks are trained
- randomly choose  $\vec{w}$
- use gradient descent
- $\vec{w} \leftarrow \vec{w} - \alpha \Delta \vec{w}$

# back propagation

- chain rule to compute the derivatives



# very hard

- (hopefully) converges to a local minimum
- good enough in practice
- massive computation

# google's style

- neural network with millions of neurons and one billion trainable parameters
- optimization is done by a 1000 machine cluster (16000 cores in total)
- would have run for 10 years on one PC

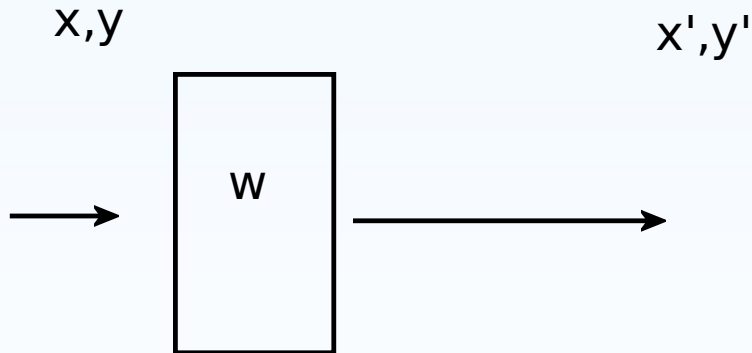
# optimization

- (paralleled) gradient descent
- surely not (or even close to) a (local) minimum
- but has been good enough to give surprising result



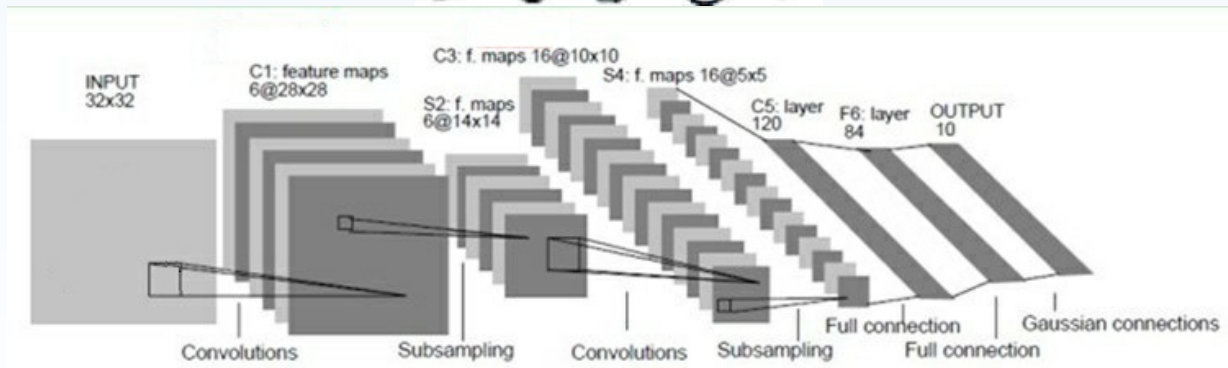
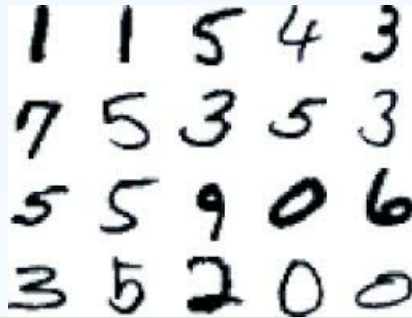
# the idea of machine learning

- traditional view of supervised learning
- training sample, model, prediction



# example

- MNIST

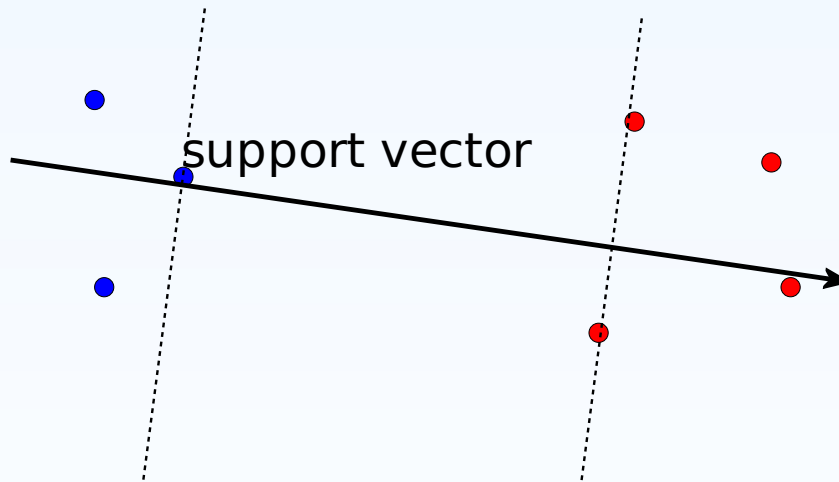


# the theorem

- **the** most important result in statistical learning
- Given sufficient training samples, under some conditions (tons of technical details omitted), if the model fits the training samples well, it is **mathematically guaranteed** that the model will give good prediction on **unseen** samples

# a tangible example

- SVM

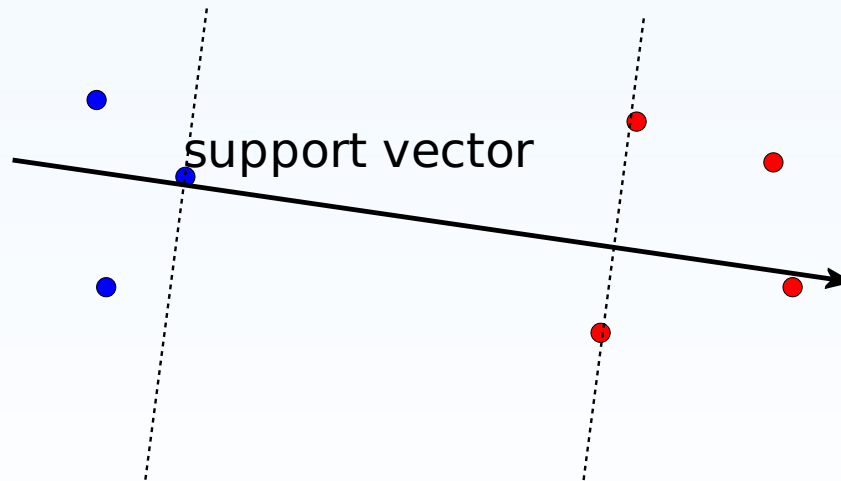


# theorem

- let  $v$  be the number of support vectors
- $Pr[Pr[\text{incorrect prediction}] \geq \Theta(\frac{v}{n}) + \epsilon] \geq 1 - o(1)$
- how to reason about unseen examples?

# unseen example

- assume the sample are obtained from a (fixed) distribution
- only the support vectors mattered



# leave one out

- assume the model is trained on all but one (randomly chosen) training sample
- $Pr[\text{incorrect prediction}] = Pr[y_i \neq M_{\text{trained on all but } (x_i, y_i)}(x_i)]$
- unbiased estimator
- the model will not change so long as the support vectors are not selected

# reinforcement learning

- neural network that plays game. backgammon
- on par with human champion





# learning algorithm

- TD-lambda
- play with itself and learn from the game

# game playing

- estimate the probability of winning for each arrangement of the board
- $f(x,w)$
- search for one step and greedily choose the best action

# TD-lambda

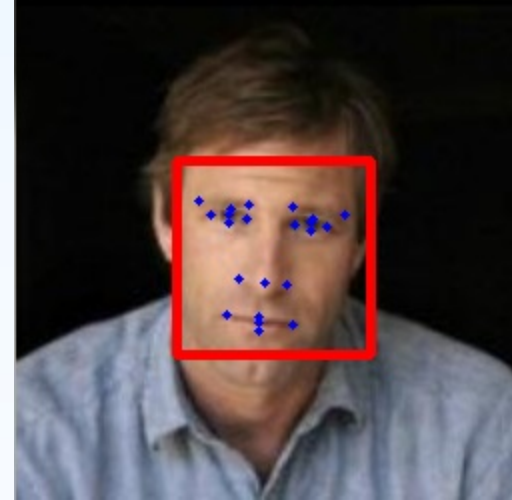
- loss function
- $f$  should be in line with the final result
- $f$  should not change too rapidly during the game

# reinforcement learning

- play the game with itself, and gradient descent on the parameters of  $f$
- Gerald Tesauro, 1994, *TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play*
- human master performance after 1500000 rounds of training

# neural networks at work

- facial keypoint detection
- trained on tens of thousands of images





# neural networks at work

[SIGN IN](#)[LANGUAGE ▾](#)[Home](#)[Tech • Service](#)[Examples](#)[Demo](#)[Dev Center](#)

used in face++, a leading face recognition system built by megvii

## Demo

- Face Detection >
- Face Search >
- Face Landmark >
- Face Mask >
- Interactive Demo >

## Tips:

Select sample image, paste picture URL, or upload local

5

25

83



## REST URL:

```
http://apius.faceplusplus.com/v2/detection/landmark?api_key=DEMO_KEY&api_secret=DEMO_SECRET&face_id=7c3d05e04b722fb5c3e617c9d1792900&type=25
```

## RESPONSE JSON:

```
{
 "result": [
 {
 "face_id": "7c3d05e04b722fb5c3e617c9d1792900",
 "landmark": {
 "left_eye_bottom": {
 "x": 37.113171,
 "y": 30.080244
 }
 }
 }
]
}
```

<mailto:career@megvii.com> if you are interested in an internship