Compressibility and probabilistic proofs

alexander.shen@lirmm.fr, www.lirmm.fr/~ashen

LIRMM CNRS & University of Montpellier

CiE 2017

Probabilistic existence proofs

Probabilistic existence proofs

- An object with some properties exists. . .

Probabilistic existence proofs

- An object with some properties exists... because a random object has these properties (with positive probability)

Probabilistic existence proofs

- An object with some properties exists... because a random object has these properties (with positive probability)
- A noncomputable binary sequence exists...

Probabilistic existence proofs

- An object with some properties exists. . . because a random object has these properties (with positive probability)
- A noncomputable binary sequence exists. . .
  . . . because the probability for a random sequence of fair coin tossings to be computable is 0

Probabilistic existence proofs

- An object with some properties exists... because a random object has these properties (with positive probability)
- A noncomputable binary sequence exists...
  ...because the probability for a random sequence of fair coin tossings to be computable is 0
  ...because the probability of a random sequence to be computed *by a given algorithm* is 0 and we have countably many algorithms.

Probabilistic existence proofs

- An object with some properties exists... because a random object has these properties (with positive probability)
- A noncomputable binary sequence exists...
  ...because the probability for a random sequence of fair coin tossings to be computable is 0
  ...because the probability of a random sequence to be computed *by a given algorithm* is 0 and we have countably many algorithms.
- cardinality argument in disguise, but we immediately get...

Probabilistic existence proofs

- An object with some properties exists… because a random object has these properties (with positive probability)
- A noncomputable binary sequence exists…
  …because the probability for a random sequence of fair coin tossings to be computable is 0
  …because the probability of a random sequence to be computed *by a given algorithm* is 0 and we have countably many algorithms.
- cardinality argument in disguise, but we immediately get…
- Kleene, Post: *there are non-comparable Turing degrees*, i.e., two binary sequences that do not compute each other (being used as oracles)

Probabilistic existence proofs

- An object with some properties exists… because a random object has these properties (with positive probability)
- A noncomputable binary sequence exists…
  …because the probability for a random sequence of fair coin tossings to be computable is 0
  …because the probability of a random sequence to be computed *by a given algorithm* is 0 and we have countably many algorithms.
- cardinality argument in disguise, but we immediately get…
- Kleene, Post: *there are non-comparable Turing degrees*, i.e., two binary sequences that do not compute each other (being used as oracles)
- The probability of "β computes α" for random independent α and β is zero (fixed β + Fubini's theorem), and vice versa

Finite probabilistic existence proof

Finite probabilistic existence proof

- Boolean matrices $n \times n$

Finite probabilistic existence proof

- Boolean matrices $n \times n$
- $k \times k$ minor: fix arbitrary k rows and k columns

Finite probabilistic existence proof

- Boolean matrices $n \times n$
- $k \times k$ minor: fix arbitrary k rows and k columns
- monochromatic minor: all zeros/all ones

Finite probabilistic existence proof

- Boolean matrices $n \times n$
- $k \times k$ minor: fix arbitrary k rows and k columns
- monochromatic minor: all zeros/all ones
- Theorem: for $k = 3 \log n$ and large n there exists a matrix without $k \times k$ monochromatic minors

Finite probabilistic existence proof

- Boolean matrices $n \times n$
- $k \times k$ minor: fix arbitrary k rows and k columns
- monochromatic minor: all zeros/all ones
- Theorem: for $k = 3\log n$ and large n there exists a matrix without $k \times k$ monochromatic minors
- Proof: for a random matrix the probability to have a large monochromatic minor is small (and therefore < 1)

Finite probabilistic existence proof

- Boolean matrices $n \times n$
- $k \times k$ minor: fix arbitrary k rows and k columns
- monochromatic minor: all zeros/all ones
- Theorem: for $k = 3 \log n$ and large n there exists a matrix without $k \times k$ monochromatic minors
- Proof: for a random matrix the probability to have a large monochromatic minor is small (and therefore < 1)
- the probability to have a $k \times k$ monochromatic minor at a given position: $2 \times 2^{-k^2}$

Finite probabilistic existence proof

- Boolean matrices $n \times n$
- $k \times k$ minor: fix arbitrary k rows and k columns
- monochromatic minor: all zeros/all ones
- Theorem: for $k = 3\log n$ and large n there exists a matrix without $k \times k$ monochromatic minors
- Proof: for a random matrix the probability to have a large monochromatic minor is small (and therefore < 1)
- the probability to have a $k \times k$ monochromatic minor at a given position: $2 \times 2^{-k^2}$
- number of possible positions: $\le n^k \times n^k = 2^{2k\log n}$

Finite probabilistic existence proof

- Boolean matrices $n \times n$
- $k \times k$ minor: fix arbitrary k rows and k columns
- monochromatic minor: all zeros/all ones
- Theorem: for $k = 3\log n$ and large n there exists a matrix without $k \times k$ monochromatic minors
- Proof: for a random matrix the probability to have a large monochromatic minor is small (and therefore < 1)
- the probability to have a $k \times k$ monochromatic minor at a given position: $2 \times 2^{-k^2}$
- number of possible positions: $\leq n^k \times n^k = 2^{2k\log n}$
- $2k\log n \ll k^2$ if $k \gg 2\log n$, so the union bound works

Finite probabilistic existence proof

- Boolean matrices $n \times n$
- $k \times k$ minor: fix arbitrary k rows and k columns
- monochromatic minor: all zeros/all ones
- Theorem: for $k = 3\log n$ and large n there exists a matrix without $k \times k$ monochromatic minors
- Proof: for a random matrix the probability to have a large monochromatic minor is small (and therefore < 1)
- the probability to have a $k \times k$ monochromatic minor at a given position: $2 \times 2^{-k^2}$
- number of possible positions: $\le n^k \times n^k = 2^{2k\log n}$
- $2k\log n \ll k^2$ if $k \gg 2\log n$, so the union bound works
- Just counting (of course)

Same proof using the compression language

Same proof using the compression language

- $n \times n$ matrix can be encoded as a n^2-bit string

Same proof using the compression language

- $n \times n$ matrix can be encoded as a n^2-bit string
- most strings are incompressible (cannot be described by fewer bits)

Same proof using the compression language

- $n \times n$ matrix can be encoded as a n^2-bit string
- most strings are incompressible (cannot be described by fewer bits)
- if matrix with a $k \times k$ monochromatic minor for $k \gg 2\log n$ is compressible

Same proof using the compression language

- $n \times n$ matrix can be encoded as a n^2-bit string
- most strings are incompressible (cannot be described by fewer bits)
- if matrix with a $k \times k$ monochromatic minor for $k \gg 2 \log n$ is compressible
- why? it has a short description:

Same proof using the compression language

- $n \times n$ matrix can be encoded as a n^2-bit string
- most strings are incompressible (cannot be described by fewer bits)
- if matrix with a $k \times k$ monochromatic minor for $k \gg 2\log n$ is compressible
- why? it has a short description:
- each of $2k$ rows/columns of the minor requires $\log n$ bits, $2k \log n$ in total

Same proof using the compression language

- $n \times n$ matrix can be encoded as a n^2-bit string
- most strings are incompressible (cannot be described by fewer bits)
- if matrix with a $k \times k$ monochromatic minor for $k \gg 2 \log n$ is compressible
- why? it has a short description:
- each of $2k$ rows/columns of the minor requires $\log n$ bits, $2k \log n$ in total
- one bit for the color of the minor

Same proof using the compression language

- $n \times n$ matrix can be encoded as a n^2-bit string
- most strings are incompressible (cannot be described by fewer bits)
- if matrix with a $k \times k$ monochromatic minor for $k \gg 2\log n$ is compressible
- why? it has a short description:
- each of $2k$ rows/columns of the minor requires $\log n$ bits, $2k \log n$ in total
- one bit for the color of the minor
- the rest of the matrix ($n^2 - k^2$ bits)

Same proof using the compression language

- $n \times n$ matrix can be encoded as a n^2-bit string
- most strings are incompressible (cannot be described by fewer bits)
- if matrix with a $k \times k$ monochromatic minor for $k \gg 2\log n$ is compressible
- why? it has a short description:
- each of $2k$ rows/columns of the minor requires $\log n$ bits, $2k\log n$ in total
- one bit for the color of the minor
- the rest of the matrix ($n^2 - k^2$ bits)
- replacing k^2 by $2k\log n + 1$: compression if $k \gg 2\log n$

So what?

So what?

- may be the compression language is more intuitive

So what?

- may be the compression language is more intuitive
- but not very impressive. . .

So what?

- may be the compression language is more intuitive
- but not very impressive...
- more interesting examples

So what?

- may be the compression language is more intuitive
- but not very impressive. . .
- more interesting examples
- Lovasz local lemma instead of the union bound

So what?

- may be the compression language is more intuitive
- but not very impressive. . .
- more interesting examples
- Lovasz local lemma instead of the union bound
- algorithmic version due to Moses-Tardos

So what?

- may be the compression language is more intuitive
- but not very impressive. . .
- more interesting examples
- Lovasz local lemma instead of the union bound
- algorithmic version due to Moses-Tardos
- do not need to know what is LL and MT algorithm

So what?

- may be the compression language is more intuitive
- but not very impressive...
- more interesting examples
- Lovasz local lemma instead of the union bound
- algorithmic version due to Moses-Tardos
- do not need to know what is LL and MT algorithm
- scheme: we try to most natural randomized algorithm

So what?

- may be the compression language is more intuitive
- but not very impressive. . .
- more interesting examples
- Lovasz local lemma instead of the union bound
- algorithmic version due to Moses-Tardos
- do not need to know what is LL and MT algorithm
- scheme: we try to most natural randomized algorithm
- it succeeds with high probability. . .

So what?

- may be the compression language is more intuitive
- but not very impressive...
- more interesting examples
- Lovasz local lemma instead of the union bound
- algorithmic version due to Moses-Tardos
- do not need to know what is LL and MT algorithm
- scheme: we try to most natural randomized algorithm
- it succeeds with high probability...
- because if it fails, the random bits used are compressible

So what?

- may be the compression language is more intuitive
- but not very impressive. . .
- more interesting examples
- Lovasz local lemma instead of the union bound
- algorithmic version due to Moses-Tardos
- do not need to know what is LL and MT algorithm
- scheme: we try to most natural randomized algorithm
- it succeeds with high probability. . .
- because if it fails, the random bits used are compressible
- A: forbidden factors (Ochem, Gonçalves)

So what?

- may be the compression language is more intuitive
- but not very impressive...
- more interesting examples
- Lovasz local lemma instead of the union bound
- algorithmic version due to Moses-Tardos
- do not need to know what is LL and MT algorithm
- scheme: we try to most natural randomized algorithm
- it succeeds with high probability...
- because if it fails, the random bits used are compressible
- A: forbidden factors (Ochem, Gonçalves)
- B: CNF with bounded neighborhood (Moser, Fortnow)

Forbidden factors

Forbidden factors

- $F_1, \ldots, F_k$: binary strings ("forbidden strings")

Forbidden factors

- $F_1, \ldots, F_k$: binary strings ("forbidden strings")
- is there an infinite bit sequence that does not have any of F_i as a substring?

Forbidden factors

- $F_1, \ldots, F_k$: binary strings ("forbidden strings")
- is there an infinite bit sequence that does not have any of F_i as a substring?
- infinite $\Leftrightarrow$ arbitrarily long

Forbidden factors

- $F_1, \ldots, F_k$: binary strings ("forbidden strings")
- is there an infinite bit sequence that does not have any of F_i as a substring?
- infinite $\Leftrightarrow$ arbitrarily long
- the answer depends on the list: $0, 11$ does not exist;

Forbidden factors

- $F_1, \ldots, F_k$: binary strings ("forbidden strings")
- is there an infinite bit sequence that does not have any of F_i as a substring?
- infinite $\Leftrightarrow$ arbitrarily long
- the answer depends on the list: $0, 11$ does not exist; $0, 00$ does exist

Forbidden factors

- $F_1, \ldots, F_k$: binary strings ("forbidden strings")
- is there an infinite bit sequence that does not have any of F_i as a substring?
- infinite $\Leftrightarrow$ arbitrarily long
- the answer depends on the list: $0, 11$ does not exist; $0, 00$ does exist
- for a fixed list we get a regular expression / finite automaton

Forbidden factors

- $F_1, \ldots, F_k$: binary strings ("forbidden strings")
- is there an infinite bit sequence that does not have any of F_i as a substring?
- infinite $\Leftrightarrow$ arbitrarily long
- the answer depends on the list: $0, 11$ does not exist; $0, 00$ does exist
- for a fixed list we get a regular expression / finite automaton
- quantitative results: "if there are not too many forbidden strings of each length, then there are long sequences without forbidden strings"

Forbidden factors

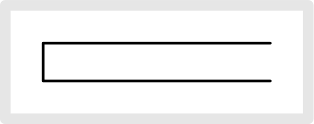
- $F_1, \dots, F_k$: binary strings ("forbidden strings")
- is there an infinite bit sequence that does not have any of F_i as a substring?
- infinite $\Leftrightarrow$ arbitrarily long
- the answer depends on the list: $0, 11$ does not exist; $0, 00$ does exist
- for a fixed list we get a regular expression / finite automaton
- quantitative results: "if there are not too many forbidden strings of each length, then there are long sequences without forbidden strings"
- *Let a_i be the number of forbidden strings of length i. If*

$$\sum a_i t^i < mt - 1 \textit{ for some } t > 0$$

then there exist arbitrarily long strings without forbidden factors. (For the case of m letters)

Tetris algorithm

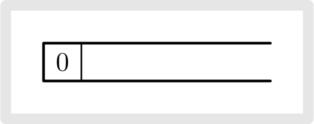
Forbidden strings: 01, 110



Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

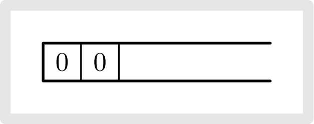


0

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

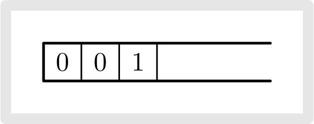


00

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

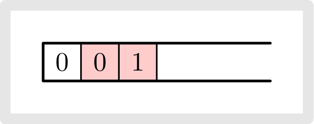


001

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

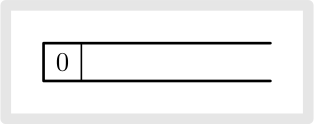


001

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

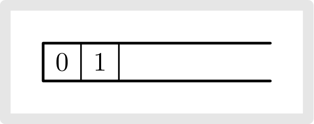


001

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

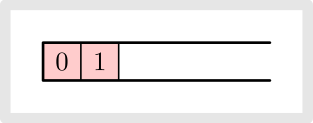


0011

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

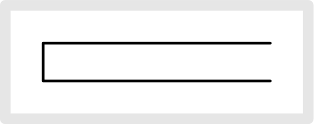


0011

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

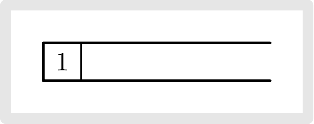


0011

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

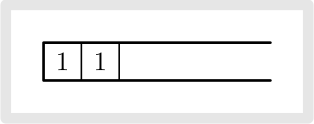


00111

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

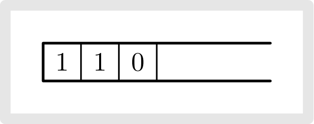


001111

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

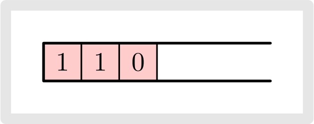


0011110

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

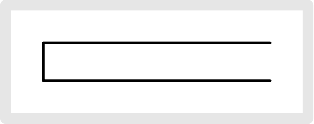


0011110

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

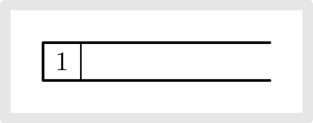


0011110

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

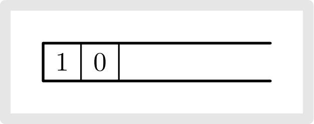


00111101

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

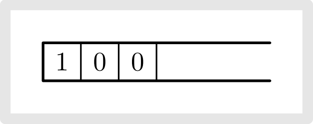


001111010

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

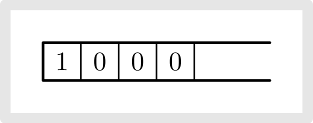


0011110100

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

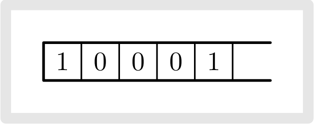


00111101000

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

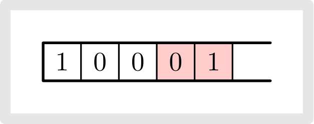


001111010001

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

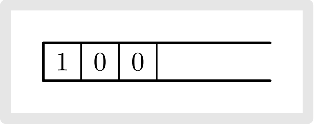


001111010001

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110

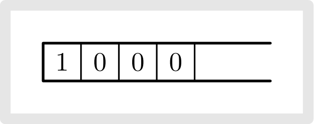


001111010001

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Tetris algorithm

Forbidden strings: 01, 110



0011110100010

Random bits are added one by one; if a forbidden string appears (at the end), it vanishes, and the process continues

Will it grow indefinitely?

Will it grow indefinitely?

- alphabet size m

Will it grow indefinitely?

- alphabet size m
- a_n is the number of forbidden strings of length $n \geq 2$

Will it grow indefinitely?

- alphabet size m
- a_n is the number of forbidden strings of length $n \geq 2$
- assume that $\sum_n a_n t^n < mt - 1$ for some $t > 0$

Will it grow indefinitely?

- alphabet size m
- a_n is the number of forbidden strings of length $n \geq 2$
- assume that $\sum_n a_n t^n < mt - 1$ for some $t > 0$
- Claim: *if the string remains short forever, then the sequence of random bits is compressible*

Will it grow indefinitely?

- alphabet size m
- a_n is the number of forbidden strings of length $n \geq 2$
- assume that $\sum_n a_n t^n < mt - 1$ for some $t > 0$
- Claim: *if the string remains short forever, then the sequence of random bits is compressible*
- log file: sequence of signs like $+$, $+_{01}$, $+_{110}$ for adding the new bit (not indicated) without or with cancelled string

Will it grow indefinitely?

- alphabet size m
- a_n is the number of forbidden strings of length $n \geq 2$
- assume that $\sum_n a_n t^n < mt - 1$ for some $t > 0$
- Claim: *if the string remains short forever, then the sequence of random bits is compressible*
- log file: sequence of signs like $+$, $+_{01}$, $+_{110}$ for adding the new bit (not indicated) without or with cancelled string
- going backwards: $+$ means deletion of the last bit, $+_u$ means adding u and then deleting the last bit

Will it grow indefinitely?

- alphabet size m
- a_n is the number of forbidden strings of length $n \geq 2$
- assume that $\sum_n a_n t^n < mt - 1$ for some $t > 0$
- Claim: *if the string remains short forever, then the sequence of random bits is compressible*
- log file: sequence of signs like $+$, $+_{01}$, $+_{110}$ for adding the new bit (not indicated) without or with cancelled string
- going backwards: $+$ means deletion of the last bit, $+_u$ means adding u and then deleting the last bit
- current sequence $+$ log file $\rightarrow$ random bits used

Will it grow indefinitely?

- alphabet size m
- a_n is the number of forbidden strings of length $n \geq 2$
- assume that $\sum_n a_n t^n < mt - 1$ for some $t > 0$
- Claim: *if the string remains short forever, then the sequence of random bits is compressible*
- log file: sequence of signs like $+$, $+_{01}$, $+_{110}$ for adding the new bit (not indicated) without or with cancelled string
- going backwards: $+$ means deletion of the last bit, $+_u$ means adding u and then deleting the last bit
- current sequence + log file $\rightarrow$ random bits used
- few forbidden strings $\Rightarrow$ few symbols in log file $\Rightarrow$ efficient encoding

More details

More details

- current string + log file $\to$ sequence of random bits

More details

- current string + log file $\to$ sequence of random bits
- current string is $O(1)$ if it doesn't grow indefinitely

More details

- current string $+$ log file $\to$ sequence of random bits
- current string is $O(1)$ if it doesn't grow indefinitely
- the length of log file is the number of random bits

More details

- current string + log file → sequence of random bits
- current string is $O(1)$ if it doesn't grow indefinitely
- the length of log file is the number of random bits
- so we need to encode the log file efficiently (< 1 bit/symbol)

More details

- current string $+$ log file $\to$ sequence of random bits
- current string is $O(1)$ if it doesn't grow indefinitely
- the length of log file is the number of random bits
- so we need to encode the log file efficiently (< 1 bit/symbol)
- large alphabet $+_x$ but most symbols are $+$

More details

- current string $+$ log file $\to$ sequence of random bits
- current string is $O(1)$ if it doesn't grow indefinitely
- the length of log file is the number of random bits
- so we need to encode the log file efficiently (< 1 bit/symbol)
- large alphabet $+_x$ but most symbols are $+$
- arithmetic coding: use less that 1 bit for $+$

More details

- current string + log file $\to$ sequence of random bits
- current string is $O(1)$ if it doesn't grow indefinitely
- the length of log file is the number of random bits
- so we need to encode the log file efficiently (< 1 bit/symbol)
- large alphabet $+_x$ but most symbols are $+$
- arithmetic coding: use less that 1 bit for $+$
- the savings due to $+$'s are used for encoding $+_x$ letters

More details

- current string $+$ log file $\to$ sequence of random bits
- current string is $O(1)$ if it doesn't grow indefinitely
- the length of log file is the number of random bits
- so we need to encode the log file efficiently (< 1 bit/symbol)
- large alphabet $+_x$ but most symbols are $+$
- arithmetic coding: use less that 1 bit for $+$
- the savings due to $+$'s are used for encoding $+_x$ letters
- amortized analysis: $+$ increases the length by 1 and $+_x$ decreases the length by $|x| - 1$

More details

- current string + log file $\to$ sequence of random bits
- current string is $O(1)$ if it doesn't grow indefinitely
- the length of log file is the number of random bits
- so we need to encode the log file efficiently (< 1 bit/symbol)
- large alphabet $+_x$ but most symbols are $+$
- arithmetic coding: use less that 1 bit for $+$
- the savings due to $+$'s are used for encoding $+_x$ letters
- amortized analysis: $+$ increases the length by 1 and $+_x$ decreases the length by $|x| - 1$
- so there couldn't be many $+_x$ unless there are many $+$

More details

- current string + log file $\to$ sequence of random bits
- current string is $O(1)$ if it doesn't grow indefinitely
- the length of log file is the number of random bits
- so we need to encode the log file efficiently (<1 bit/symbol)
- large alphabet $+_x$ but most symbols are $+$
- arithmetic coding: use less that 1 bit for $+$
- the savings due to $+$'s are used for encoding $+_x$ letters
- amortized analysis: $+$ increases the length by 1 and $+_x$ decreases the length by $|x|-1$
- so there couldn't be many $+_x$ unless there are many $+$
- role of t: parameter for amortized analysis of the encoding efficiency

Technical details

Technical details

- arithmetic coding: each symbol z has some weight $p_z > 0$

Technical details

- arithmetic coding: each symbol z has some weight $p_z > 0$
- $\sum_z p_z \leq 1$

Technical details

- arithmetic coding: each symbol z has some weight $p_z > 0$
- $\sum_z p_z \leq 1$
- encoding z by $\log(1/p_z)$ bits

Technical details

- arithmetic coding: each symbol z has some weight $p_z > 0$
- $\sum_z p_z \leq 1$
- encoding z by $\log(1/p_z)$ bits
- allocate weight q_0 for $+$ and total weight q_n for all $+_u$ where u are forbidden strings of length n.

Technical details

- arithmetic coding: each symbol z has some weight $p_z > 0$
- $\sum_z p_z \leq 1$
- encoding z by $\log(1/p_z)$ bits
- allocate weight q_0 for $+$ and total weight q_n for all $+_u$ where u are forbidden strings of length n.
- code lengths: $-\log q_0$, for "$+$"
  $-\log q_n + \log a_n$, for each "$+_u$" with $|u| = n$

Technical details

- arithmetic coding: each symbol z has some weight $p_z > 0$
- $\sum_z p_z \leq 1$
- encoding z by $\log(1/p_z)$ bits
- allocate weight q_0 for $+$ and total weight q_n for all $+_u$ where u are forbidden strings of length n.
- code lengths: $-\log q_0$, for "$+$"

  $-\log q_n + \log a_n$, for each "$+_u$" with $|u| = n$
- each log symbol corresponds to one random symbol, so we want to encode log symbols with less than $\log m$ bits

Technical details

- arithmetic coding: each symbol z has some weight $p_z > 0$
- $\sum_z p_z \le 1$
- encoding z by $\log(1/p_z)$ bits
- allocate weight q_0 for $+$ and total weight q_n for all $+_u$ where u are forbidden strings of length n.
- code lengths: $-\log q_0$, for "$+$"

  $-\log q_n + \log a_n$, for each "$+_u$" with $|u| = n$
- each log symbol corresponds to one random symbol, so we want to encode log symbols with less than $\log m$ bits
- $+$ increases the length by 1, and $+_u$ decreases the length by $n-1$ for $|u| = n$

Technical details

- arithmetic coding: each symbol z has some weight $p_z > 0$
- $\sum_z p_z \leq 1$
- encoding z by $\log(1/p_z)$ bits
- allocate weight q_0 for $+$ and total weight q_n for all $+_u$ where u are forbidden strings of length n.
- code lengths: $-\log q_0$, for "$+$"
  $-\log q_n + \log a_n$, for each "$+_u$" with $|u| = n$
- each log symbol corresponds to one random symbol, so we want to encode log symbols with less than $\log m$ bits
- $+$ increases the length by 1, and $+_u$ decreases the length by $n-1$ for $|u| = n$
- amortized analysis: when increasing length $(+)$, reserve δ; when decreasing length $n-1$, use $\delta(n-1)$ from reserves.

Technical details-2

- code lengths: $-\log q_0$, for "$+$"
  $-\log q_n + \log a_n$, for each "$+_u$" with $|u| = n$
- each log symbol corresponds to one random symbol, so we want to encode log symbols with less than $\log m$ bits
- $+$ increases the length by 1, and $+_u$ decreases the length by $n-1$ for $|u| = n$
- amortized analysis: when increasing length ($+$), reserve δ; when decreasing length $n-1$, use $\delta(n-1)$ from reserves.

Technical details-2

- code lengths: $-\log q_0$, for "$+$"
  $-\log q_n + \log a_n$, for each "$+_u$" with $|u| = n$
- each log symbol corresponds to one random symbol, so we want to encode log symbols with less than $\log m$ bits
- $+$ increases the length by 1, and $+_u$ decreases the length by $n-1$ for $|u| = n$
- amortized analysis: when increasing length ($+$), reserve δ; when decreasing length $n-1$, use $\delta(n-1)$ from reserves.
- $-\log q_0 \leq \log m - \delta$

Technical details-2

- code lengths: $-\log q_0$, for "$+$"
  $-\log q_n + \log a_n$, for each "$+_u$" with $|u| = n$
- each log symbol corresponds to one random symbol, so we want to encode log symbols with less than $\log m$ bits
- $+$ increases the length by 1, and $+_u$ decreases the length by $n-1$ for $|u| = n$
- amortized analysis: when increasing length ($+$), reserve δ; when decreasing length $n-1$, use $\delta(n-1)$ from reserves.
- $-\log q_0 \le \log m - \delta$
- $-\log q_n + \log a_n \le \log m + (n-1)\delta$

Technical details-2

- code lengths: $-\log q_0$, for "$+$"
  $-\log q_n + \log a_n$, for each "$+_u$" with $|u| = n$
- each log symbol corresponds to one random symbol, so we want to encode log symbols with less than $\log m$ bits
- $+$ increases the length by 1, and $+_u$ decreases the length by $n-1$ for $|u| = n$
- amortized analysis: when increasing length $(+)$, reserve δ; when decreasing length $n-1$, use $\delta(n-1)$ from reserves.
- $-\log q_0 \le \log m - \delta$
- $-\log q_n + \log a_n \le \log m + (n-1)\delta$
- $\sum_n q_n < 1$

Technical details-3

- $-\log q_0 \leq \log m - \delta$
- $-\log q_n + \log a_n \leq \log m + (n-1)\delta$
- $\sum_n q_n < 1$

Technical details-3

- $-\log q_0 = \log m - \delta$
- $-\log q_n + \log a_n = \log m + (n-1)\delta$
- $\sum_n q_n < 1$

Technical details-3

- $-\log q_0 = \log m - \delta$; $q_0 = (1/m)2^\delta$
- $-\log q_n + \log a_n = \log m + (n-1)\delta$; $q_n = (1/m)a_n 2^\delta 2^{-n\delta}$
- $\sum_n q_n < 1$

Technical details-3

- $-\log q_0 = \log m - \delta$; $q_0 = (1/m)2^{\delta}$
- $-\log q_n + \log a_n = \log m + (n-1)\delta$; $q_n = (1/m)a_n 2^{\delta} 2^{-n\delta}$
- $\sum_n q_n < 1$
- $(1/m)2^{\delta} + (1/m)2^{\delta} \sum_n a_n (2^{-\delta})^n < 1$

Technical details-3

- $-\log q_0 = \log m - \delta$; $q_0 = (1/m)2^\delta$
- $-\log q_n + \log a_n = \log m + (n-1)\delta$; $q_n = (1/m)a_n 2^\delta 2^{-n\delta}$
- $\sum_n q_n < 1$
- $(1/m)2^\delta + (1/m)2^\delta \sum_n a_n (2^{-\delta})^n < 1$
- $1 + \sum_n a_n (2^{-\delta})^n < m(2^{-\delta})$

Technical details-3

- $-\log q_0 = \log m - \delta$; $q_0 = (1/m)2^\delta$
- $-\log q_n + \log a_n = \log m + (n-1)\delta$; $q_n = (1/m)a_n 2^\delta 2^{-n\delta}$
- $\sum_n q_n < 1$
- $(1/m)2^\delta + (1/m)2^\delta \sum_n a_n(2^{-\delta})^n < 1$
- $1 + \sum_n a_n(2^{-\delta})^n < m(2^{-\delta})$
- let $t = 2^{-\delta}$

Technical details-3

- $-\log q_0 = \log m - \delta$; $q_0 = (1/m)2^\delta$
- $-\log q_n + \log a_n = \log m + (n-1)\delta$; $q_n = (1/m)a_n 2^\delta 2^{-n\delta}$
- $\sum_n q_n < 1$
- $(1/m)2^\delta + (1/m)2^\delta \sum_n a_n(2^{-\delta})^n < 1$
- $1 + \sum_n a_n(2^{-\delta})^n < m(2^{-\delta})$
- let $t = 2^{-\delta}$
- $\sum_n a^n t^n < mt - 1$, as stated

CNF with bounded neighborhood

CNF with bounded neighborhood

- CNF: clause $\wedge$ clause $\wedge \ldots \wedge$ clause

CNF with bounded neighborhood

- CNF: clause $\wedge$ clause $\wedge \ldots \wedge$ clause
- clause: literal $\vee$ literal $\vee \ldots \vee$ literal

CNF with bounded neighborhood

- CNF: clause $\wedge$ clause $\wedge \ldots \wedge$ clause
- clause: literal $\vee$ literal $\vee \ldots \vee$ literal
- literal: propositional variable or its negation

CNF with bounded neighborhood

- CNF: clause $\wedge$ clause $\wedge \ldots \wedge$ clause
- clause: literal $\vee$ literal $\vee \ldots \vee$ literal
- literal: propositional variable or its negation
- $(p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$

CNF with bounded neighborhood

- CNF: clause $\wedge$ clause $\wedge \ldots \wedge$ clause
- clause: literal $\vee$ literal $\vee \ldots \vee$ literal
- literal: propositional variable or its negation
- $(p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$
- each clause prohibits some combination of values

CNF with bounded neighborhood

- CNF: clause $\wedge$ clause $\wedge \ldots \wedge$ clause
- clause: literal $\vee$ literal $\vee \ldots \vee$ literal
- literal: propositional variable or its negation
- $(p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$
- each clause prohibits some combination of values
- here all four combinations are prohibited, unsatisfiable

CNF with bounded neighborhood

- CNF: clause $\wedge$ clause $\wedge \ldots \wedge$ clause
- clause: literal $\vee$ literal $\vee \ldots \vee$ literal
- literal: propositional variable or its negation
- $(p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$
- each clause prohibits some combination of values
- here all four combinations are prohibited, unsatisfiable
- Assume all clauses are with n literals, thus prohibiting one combination for some n variables.

CNF with bounded neighborhood

- CNF: clause $\wedge$ clause $\wedge \ldots \wedge$ clause
- clause: literal $\vee$ literal $\vee \ldots \vee$ literal
- literal: propositional variable or its negation
- $(p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$
- each clause prohibits some combination of values
- here all four combinations are prohibited, unsatisfiable
- Assume all clauses are with n literals, thus prohibiting one combination for some n variables. To make the CNF unsatisfiable, we need about 2^n of them and they should have more or less the same variables:

CNF with bounded neighborhood

- CNF: clause $\wedge$ clause $\wedge \ldots \wedge$ clause
- clause: literal $\vee$ literal $\vee \ldots \vee$ literal
- literal: propositional variable or its negation
- $(p \vee q) \wedge (p \vee \neg q) \wedge (\neg p \vee q) \wedge (\neg p \vee \neg q)$
- each clause prohibits some combination of values
- here all four combinations are prohibited, unsatisfiable
- Assume all clauses are with n literals, thus prohibiting one combination for some n variables. To make the CNF unsatisfiable, we need about 2^n of them and they should have more or less the same variables:
- Claim: *If each clause has n literals and has at most* 2^{n-3} *neighbors (=clauses that have common variable), then CNF is satisfiable.*

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
    $\textsc{Resample}(C)$
    for all C' that are neighbors of C (including C):
        if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
  $\textsc{Resample}(C)$
  for all C' that are neighbors of C (including C):
    if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- this is enough (fixing clauses one by one)

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
    $\textsc{Resample}(C)$
    for all C' that are neighbors of C (including C):
        if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- this is enough (fixing clauses one by one)
- conditional correctness

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
    $\textsc{Resample}(C)$
    for all C' that are neighbors of C (including C):
        if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- this is enough (fixing clauses one by one)
- conditional correctness
- termination?

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
    $\textsc{Resample}(C)$
    for all C' that are neighbors of C (including C):
        if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- this is enough (fixing clauses one by one)
- conditional correctness
- termination?
- Claim: *if no termination after a long time, the sequence of random bits used for resampling is compressible*

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
    $\textsc{Resample}(C)$
    for all C' that are neighbors of C (including C):
        if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
  $\textsc{Resample}(C)$
  for all C' that are neighbors of C (including C):
    if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- long (unfinished) execution of $\textsc{Fix}(C)$

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
    $\textsc{Resample}(C)$
    for all C' that are neighbors of C (including C):
        if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- long (unfinished) execution of $\textsc{Fix}(C)$
- log file: list of clauses for all calls $\textsc{Fix}(C')$

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
    $\textsc{Resample}(C)$
    for all C' that are neighbors of C (including C):
        if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- long (unfinished) execution of $\textsc{Fix}(C)$
- log file: list of clauses for all calls $\textsc{Fix}(C')$
- only false clauses are fixed

Fixing clauses

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
  $\textsc{Resample}(C)$
  for all C' that are neighbors of C (including C):
    if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- long (unfinished) execution of $\textsc{Fix}(C)$
- log file: list of clauses for all calls $\textsc{Fix}(C')$
- only false clauses are fixed
- knowing this list, and the current values of variable we can go backwards and reconstruct the values of variables and bits used for resampling — *and the log file is the compressed encoding of the bits used for the resampling*

Tree traversal

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
  $\textsc{Resample}(C)$
  for all C' that are neighbors of C (including C):
    if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

Tree traversal

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
  $\textsc{Resample}(C)$
  for all C' that are neighbors of C (including C):
    if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- main observation: C' is a neighbor of C

Tree traversal

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
  $\textsc{Resample}(C)$
  for all C' that are neighbors of C (including C):
    if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- main observation: C' is a neighbor of C
- in the tree of recursive calls sons are neighbors

Tree traversal

{ C is false }
$\textsc{Fix}(C : \text{clause})$:
    $\textsc{Resample}(C)$
    for all C' that are neighbors of C (including C):
        if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

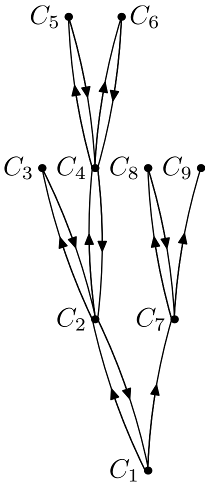
- main observation: C' is a neighbor of C
- in the tree of recursive calls sons are neighbors
- we specify a neighbor using $n-3$ bits instead of n needed to specify resampling bits: compression

Tree traversal

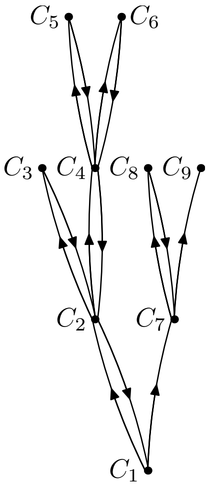
{ C is false }
$\textsc{Fix}(C : \text{clause})$:
    $\textsc{Resample}(C)$
    for all C' that are neighbors of C (including C):
        if C' is false **then** $\textsc{Fix}(C')$
{ C is true; all clauses that were true remain true }

- main observation: C' is a neighbor of C
- in the tree of recursive calls sons are neighbors
- we specify a neighbor using $n - 3$ bits instead of n needed to specify resampling bits: compression
- techically incorrect, since we also go down the tree (return from recursive calls) - we need to reserve two more bits $((n-3)+2<n)$

Tree traversal: counting bits

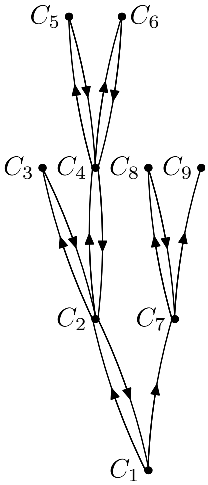


Tree traversal: counting bits



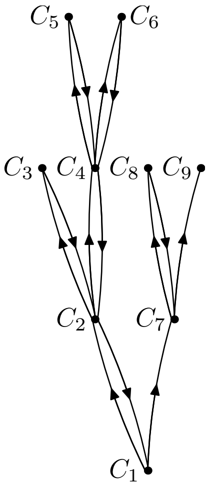
- $n-3$ bits: neighbor number

Tree traversal: counting bits

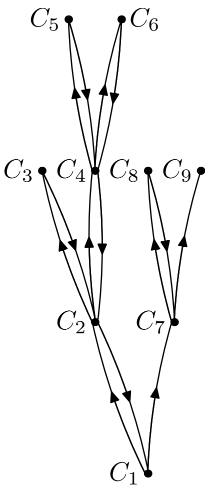


- $n-3$ bits: neighbor number
- plus 1 direction bit "up" (when going up)

Tree traversal: counting bits



- $n-3$ bits: neighbor number
- plus 1 direction bit "up" (when going up)
- 1 direction bit (when going down)



- $n-3$ bits: neighbor number
- plus 1 direction bit “up” (when going up)
- 1 direction bit (when going down)
- $(n-3)+1+1$ per one move up (n sampling bits): $(n-1)$ instead of n

History and references

History and references

- Probabilistic/averaging arguments — Littlewood (Mathematical miscellany?)

History and references

- Probabilistic/averaging arguments — Littlewood (Mathematical miscellany?)
- Lovasz local lemma (1975)

History and references

- Probabilistic/averaging arguments — Littlewood (Mathematical miscellany?)
- Lovasz local lemma (1975)
- Moser (2008)–Tardos (2009)

History and references

- Probabilistic/averaging arguments — Littlewood (Mathematical miscellany?)
- Lovasz local lemma (1975)
- Moser (2008)–Tardos (2009)
- Miller (potential, ≤ 2011)

History and references

- Probabilistic/averaging arguments — Littlewood (Mathematical miscellany?)
- Lovasz local lemma (1975)
- Moser (2008)–Tardos (2009)
- Miller (potential, $\leq$ 2011)
- Golod–Shafarevich (1964)

History and references

- Probabilistic/averaging arguments — Littlewood (Mathematical miscellany?)
- Lovasz local lemma (1975)
- Moser (2008)–Tardos (2009)
- Miller (potential, ≤ 2011)
- Golod–Shafarevich (1964)
- Ochem, Gonçalves (2014)

History and references

- Probabilistic/averaging arguments — Littlewood (Mathematical miscellany?)
- Lovasz local lemma (1975)
- Moser (2008)–Tardos (2009)
- Miller (potential, $\leq$ 2011)
- Golod–Shafarevich (1964)
- Ochem, Gonçalves (2014)

Thanks for the attention!