Algoritmos para códigos separadores

Marcel Fernandez

Sebastià Martín Molleví

Universitat Politecnica de Catalunya

John Livieratos

Department of Mathematics,
National & Kapodistrian University of Athens

Contents

Separating codes.

There are almost no (2,2)-separating codes in the literature

Best known
(comes after several names)

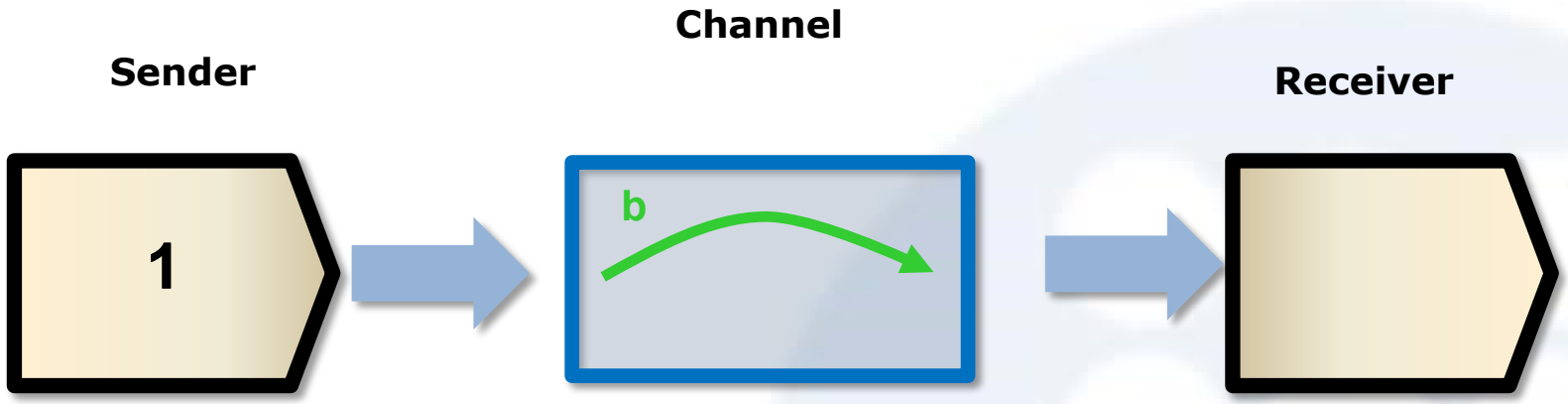
Dual Hamming code
Haddamard code
Simplex code
(rate -> 0)

The purpose of this work is to construct (2,2)-separating codes

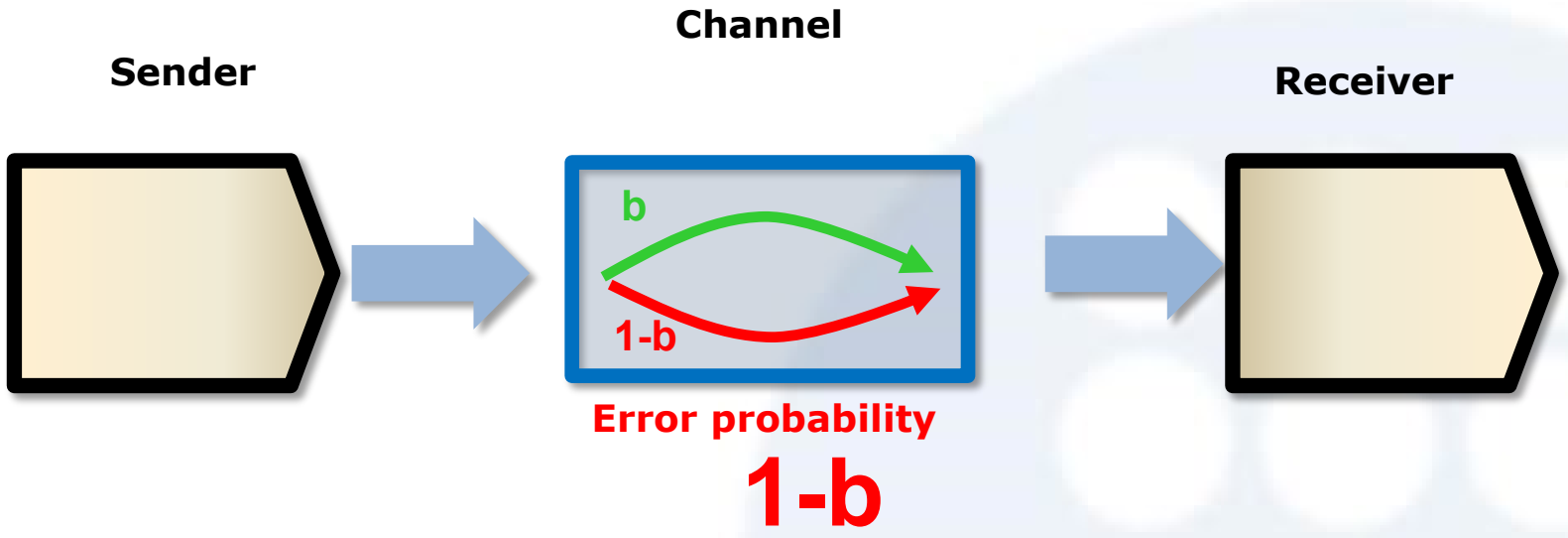
of positive rate

Error correcting codes

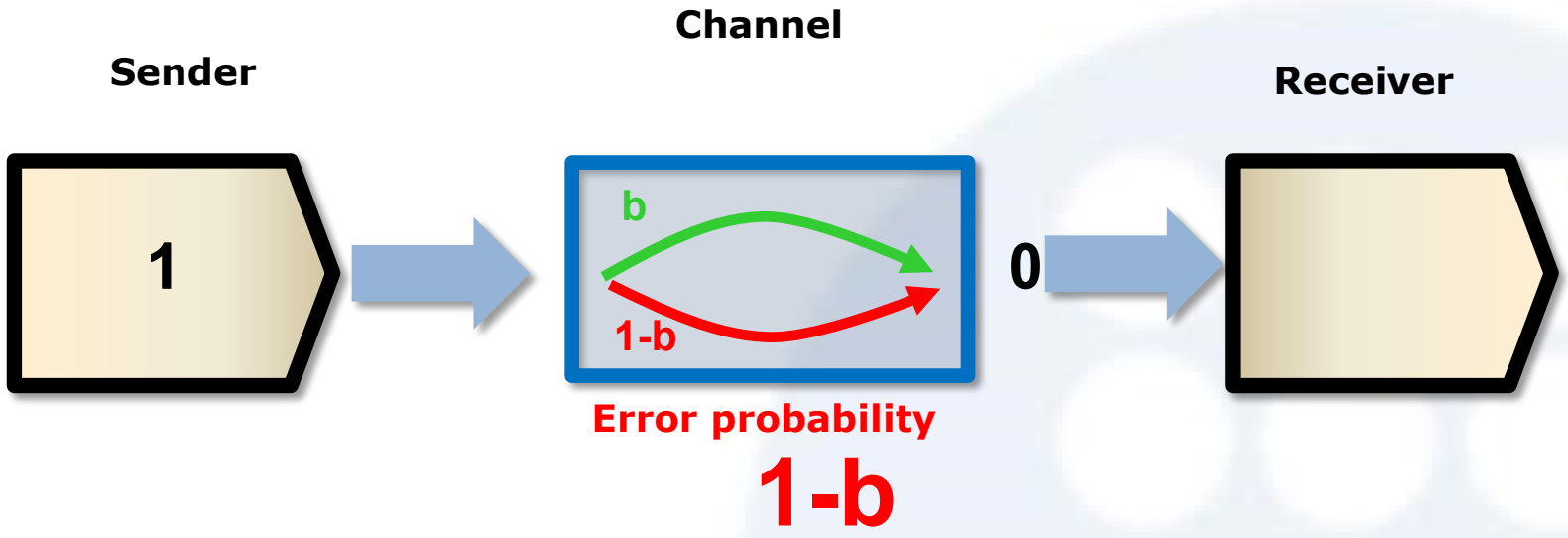
Error Correcting Codes. Code rate.



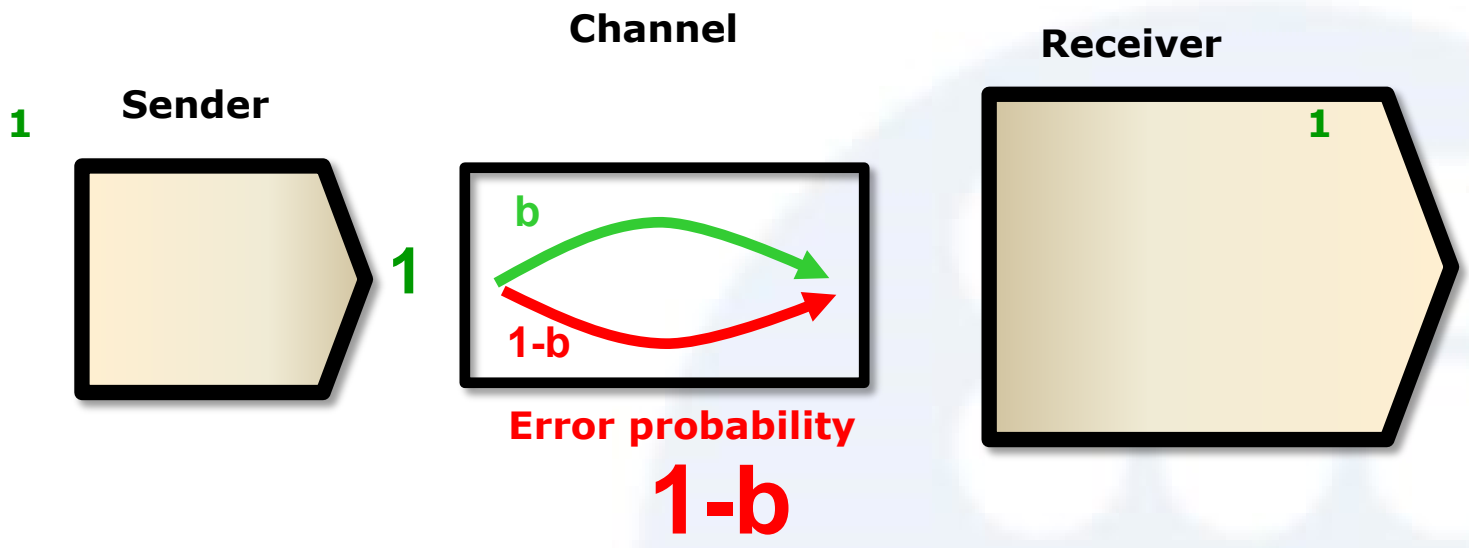
Error Correcting Codes. Code rate.



Error Correcting Codes. Code rate.

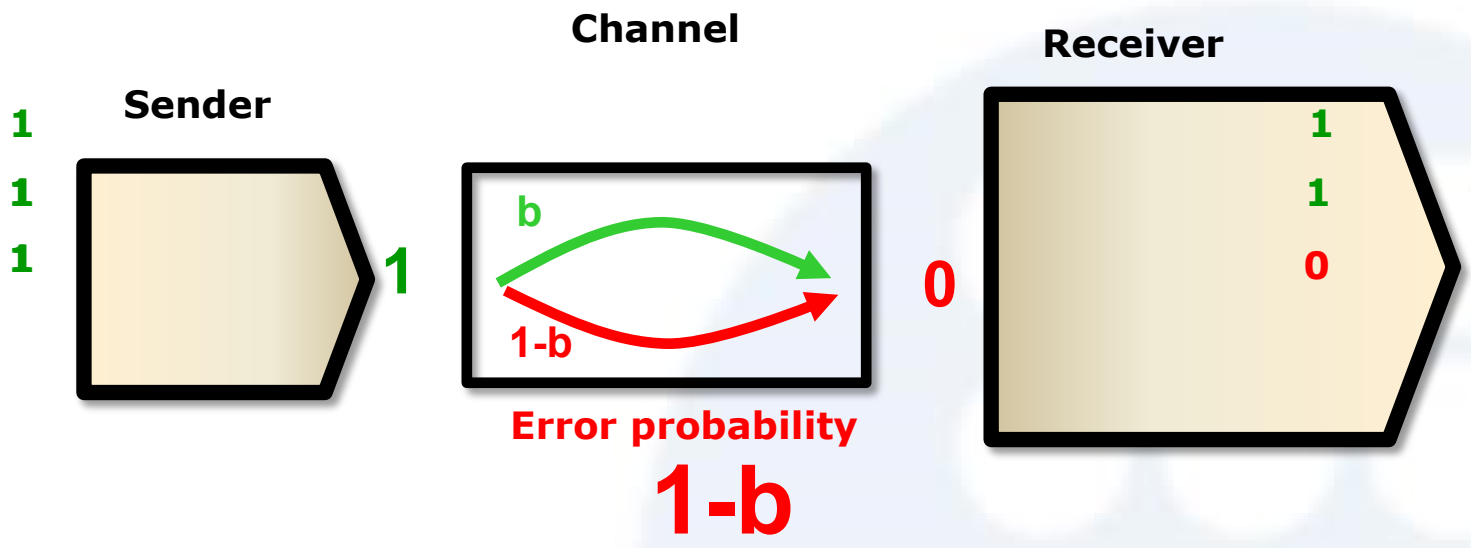


Error Correcting Codes. Code rate.



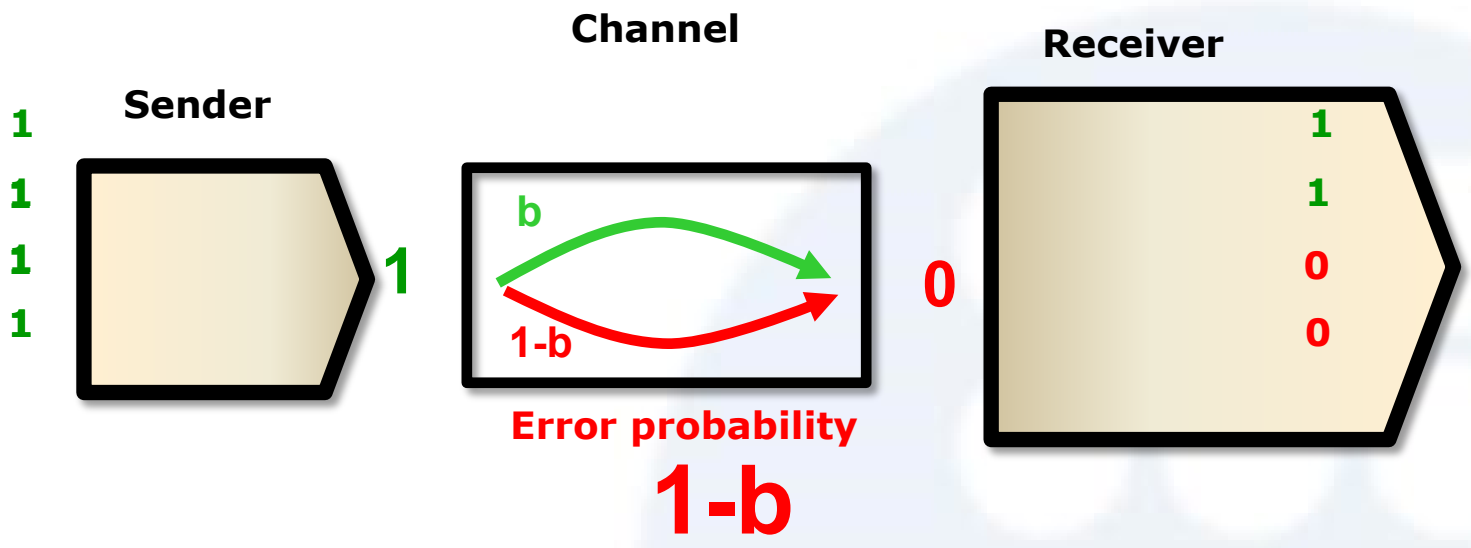
Can we correct errors?

Error Correcting Codes. Code rate.



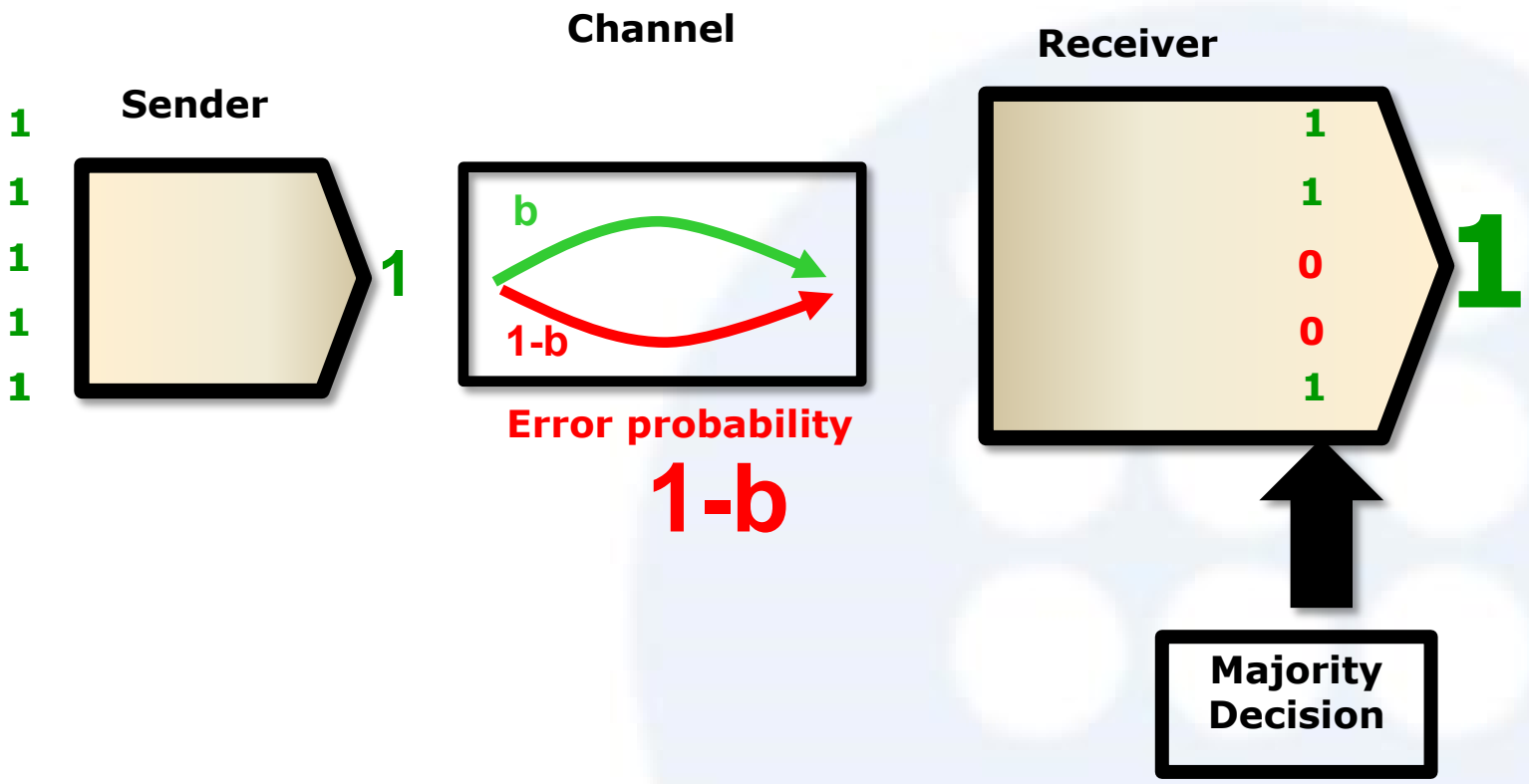
Can we correct errors?

Error Correcting Codes. Code rate.



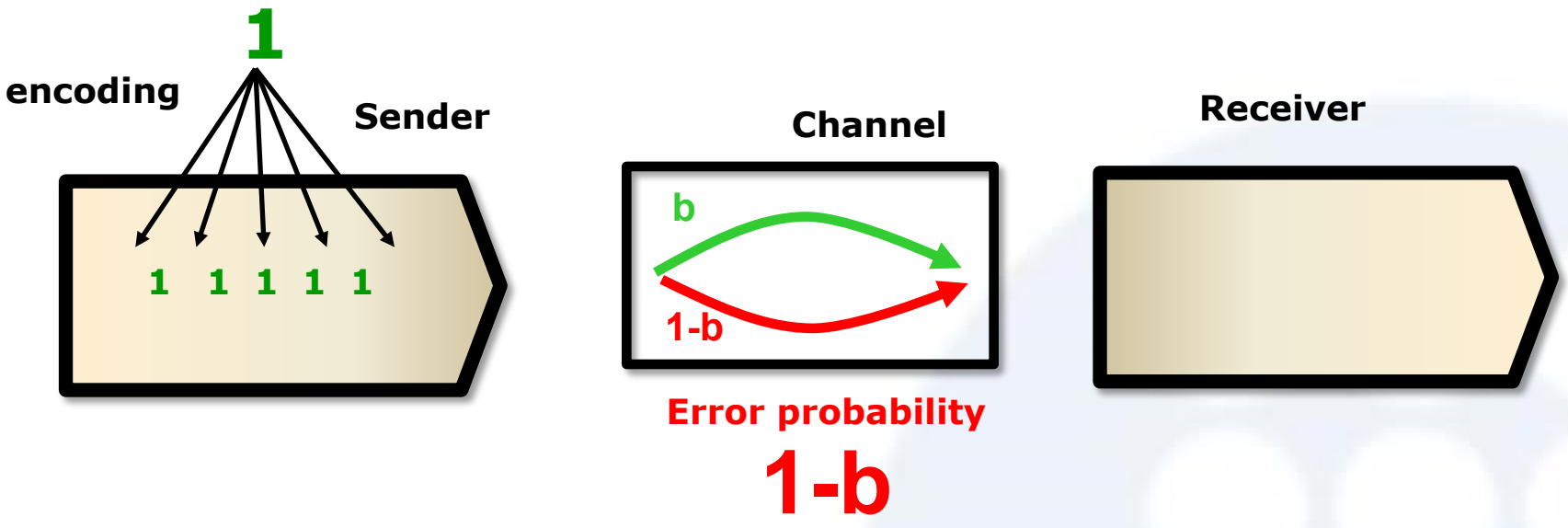
Can we correct errors?

Error Correcting Codes. Code rate.



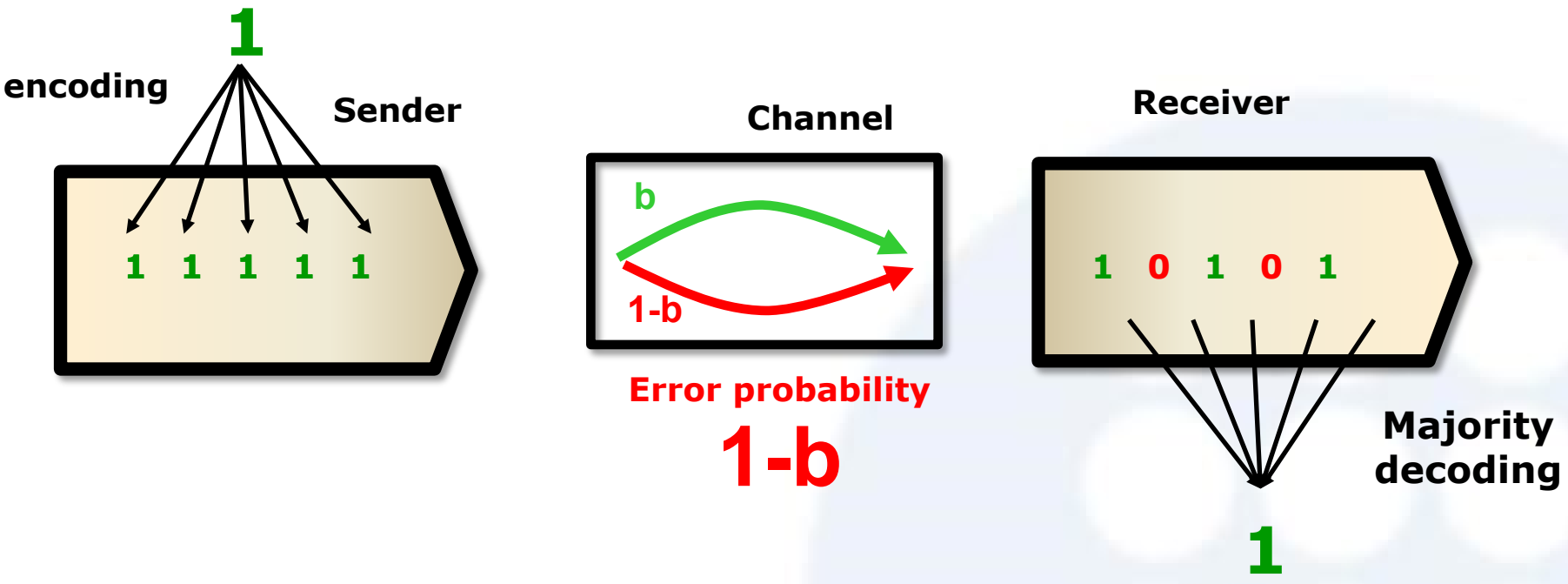
Can we correct errors?

Error Correcting Codes. Code rate.



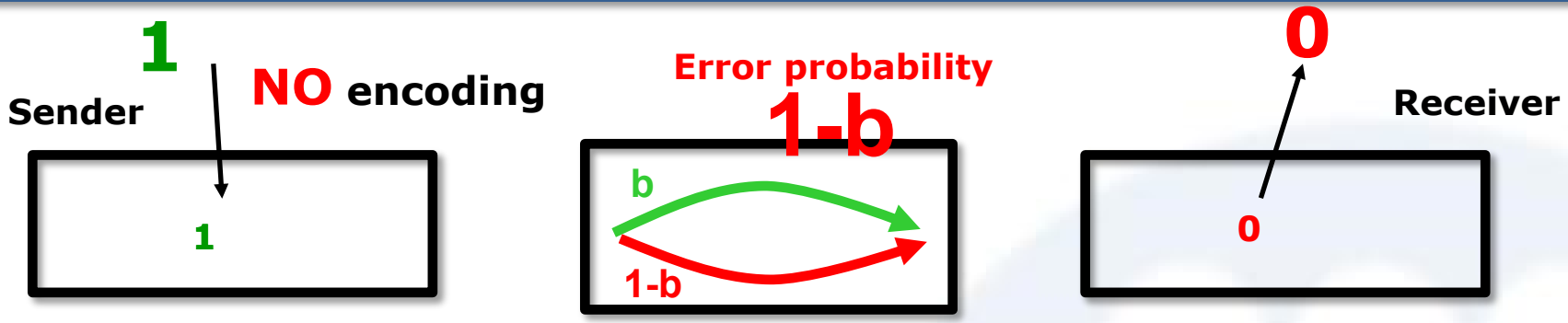
Can we correct errors?

Error Correcting Codes. Code rate.



Message		Repetition code
1	→ encoding	1111111 ··· 11
0	→ encoding	0000000 ··· 00

Error Correcting Codes. Code rate.



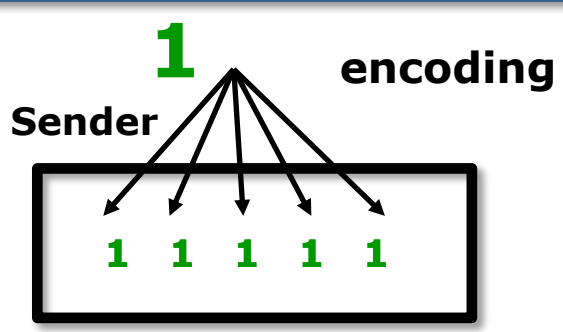
Send
1

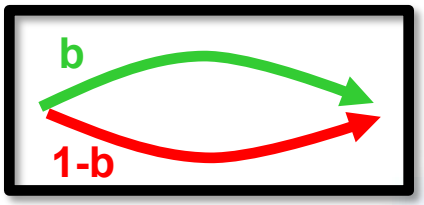
Error control
NO

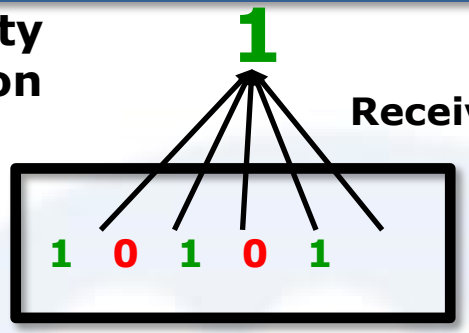
$$\frac{\text{information bits}}{\text{sent bits}}$$

1

Error Correcting Codes. Code rate.

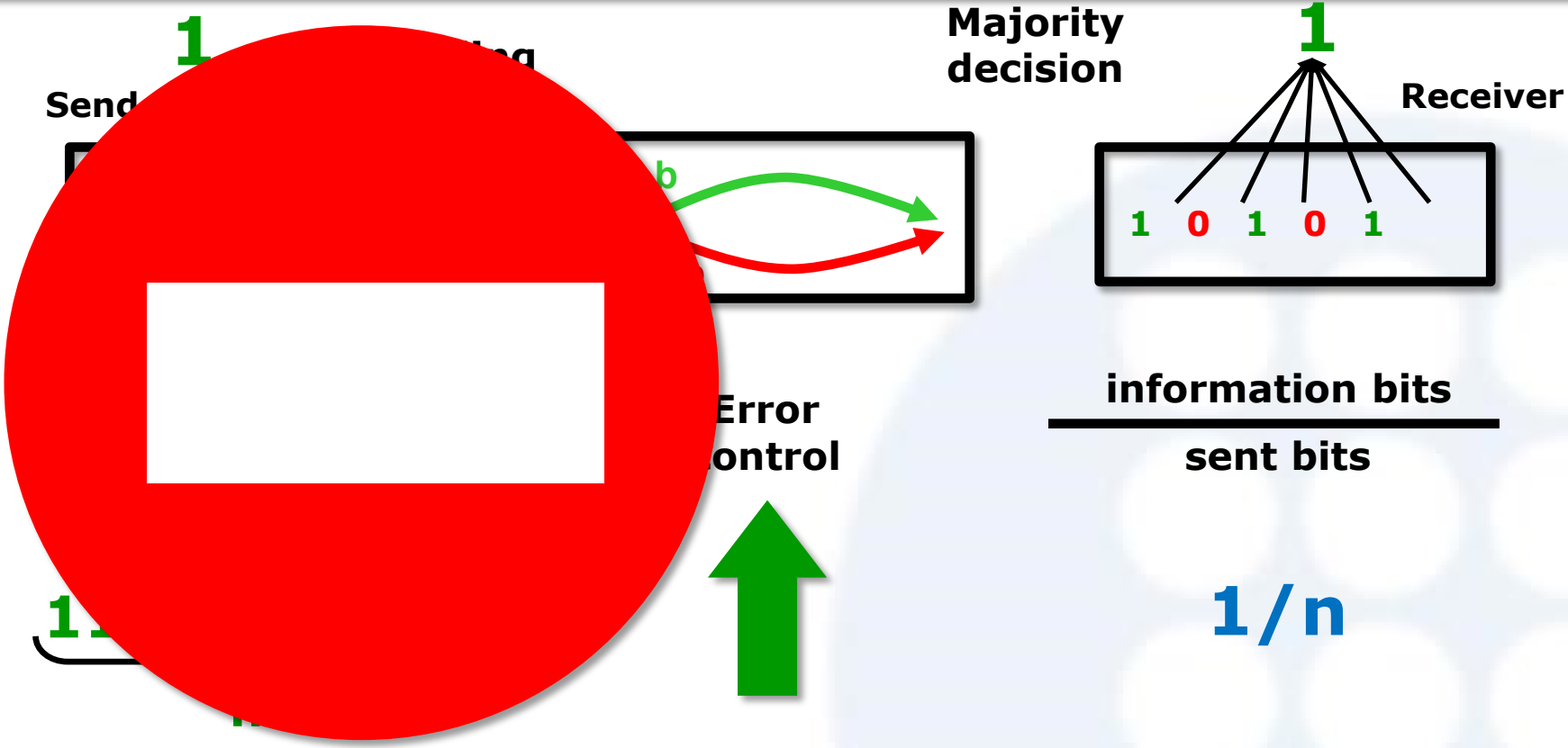






Send	Error control	$\frac{\text{information bits}}{\text{sent bits}}$
1	NO	1
11111	↑	1/5 = 0.2
$\underbrace{111111\cdots11}_{n}$	↑	1/n

Error Correcting Codes. Code rate.



Error Correcting Codes. Code rate.

We would like

Code rate

$n \to \infty$

Error control

$$\frac{\text{information bits}}{\text{sent bits}} > 0$$

Separating codes

Separating codes.

Code *C*

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1

Separating codes.

Code *C*

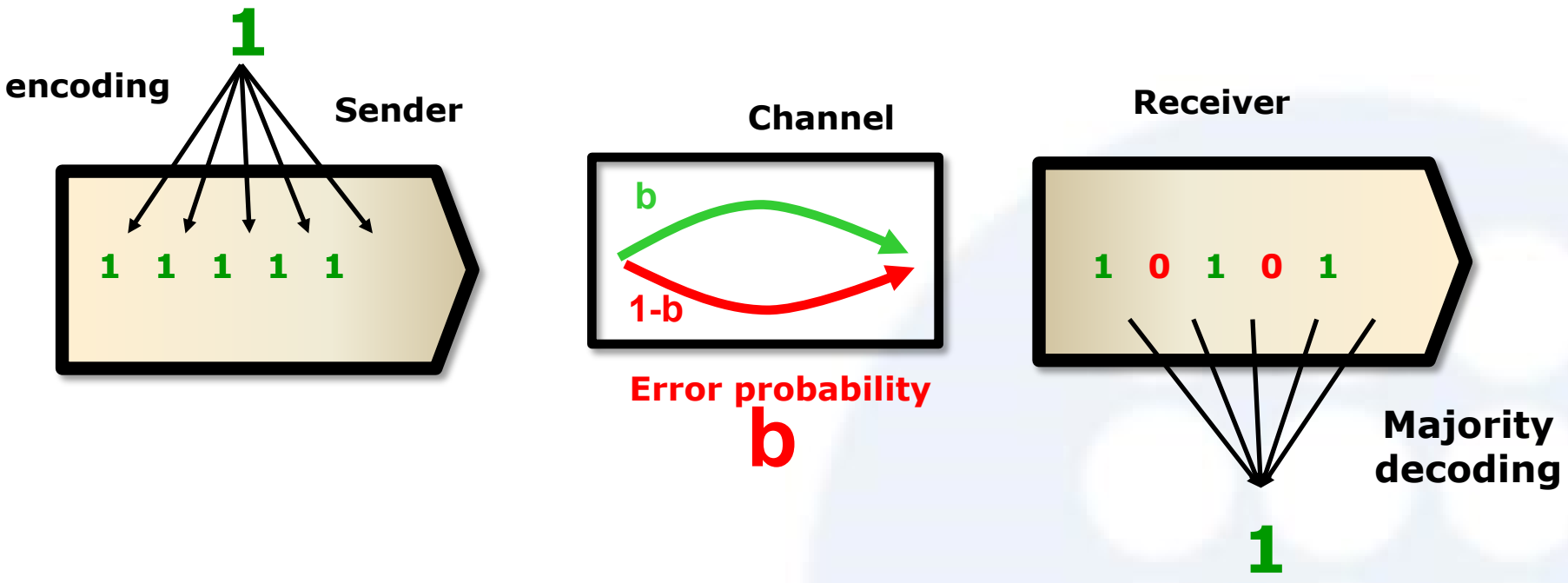
1	0	0	1	1	1	0	1
2	0	1	0	1	0	1	1
3	0	1	1	0	1	1	0
4	1	0	0	0	1	1	1
5	1	0	1	1	0	1	0
6	1	1	0	1	1	0	0
7	1	1	1	0	0	0	1

Separating codes.

Code C

001	0 0 1 1 1 0 1
010	0 1 0 1 0 1 1
011	0 1 1 0 1 1 0
100	1 0 0 0 1 1 1
101	1 0 1 1 0 1 0
110	1 1 0 1 1 0 0
111	1 1 1 0 0 0 1

Error Correcting Codes. Code rate.



Message		Repetition code
1	encoding →	1111111 ··· 11
0	encoding →	0000000 ··· 00

Separating codes.

Code *C*

001	0	0	1	1	1	0	1
010	0	1	0	1	0	1	1
011	0	1	1	0	1	1	0
100	1	0	0	0	1	1	1
101	1	0	1	1	0	1	0
110	1	1	0	1	1	0	0
111	1	1	1	0	0	0	1

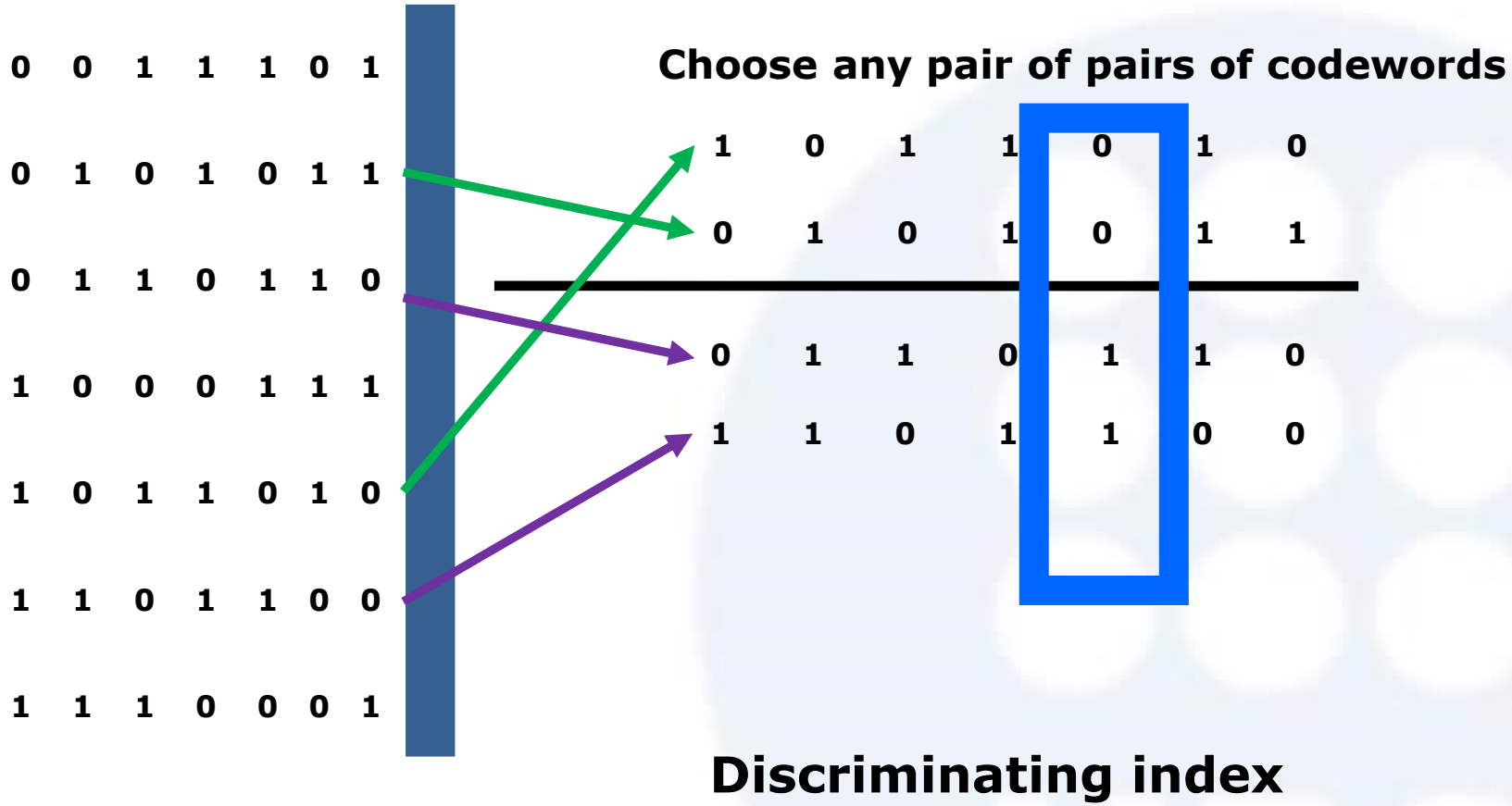
Separating codes.

Properties of C ?

001	0	0	1	1	1	0	1
010	0	1	0	1	0	1	1
011	0	1	1	0	1	1	0
100	1	0	0	0	1	1	1
101	1	0	1	1	0	1	0
110	1	1	0	1	1	0	0
111	1	1	1	0	0	0	1

Separating codes.

Properties of C ?



Separating codes.

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
1	0	1	1	0	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1

(2,2)-separating code

Discriminating index

Separating codes.

There are almost no (2,2)-separating codes in the literature

Best known
(comes after several names)

Dual Hamming code
Haddamard code
Simplex code
(rate -> 0)

The purpose of this work is to construct (2,2)-separating codes

of positive rate

3

Lovász Local Lemma

Lovasz Local Lemma

Lovász local lemma

Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p

- is independent of all the other events except for at most d of them.

If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

"Nothing bad happens"

It can be used to proof existence results

4

Existence of separating codes of positive rate

Existence of (2,2)-separating codes.

Is there a (2,2)-separating code such

$$\frac{\text{information bits}}{\text{sent bits}} > 0$$

?

Existence of (2,2)-separating codes.

Lovász local lemma

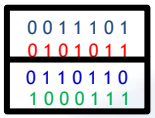
Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

sets of disjoint pairs of pairs



Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

Existence of (2,2)-separating codes.

Lovász local lemma

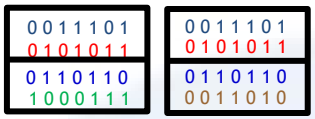
Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

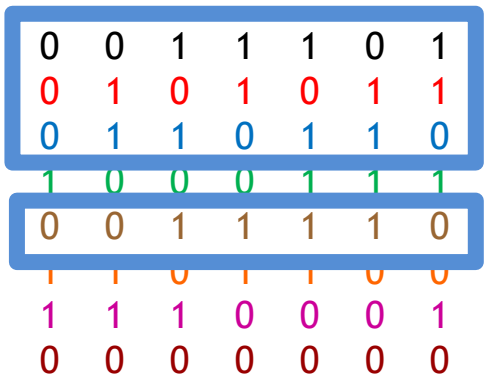
If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

sets of disjoint pairs of pairs



Code



Existence of (2,2)-separating codes.

Lovász local lemma

Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

sets of disjoint pairs of pairs

Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

Existence of (2,2)-separating codes.

Lovász local lemma

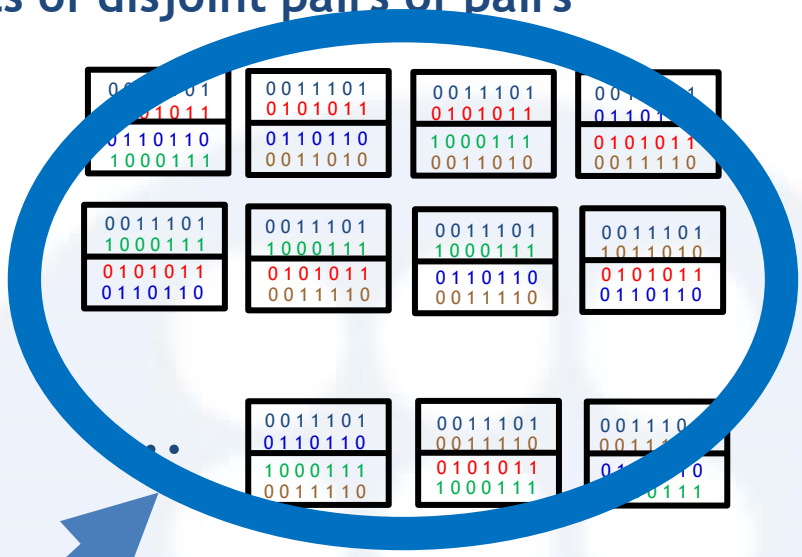
Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

sets of disjoint pairs of pairs



Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

We would like all pairs of pairs separated

Existence of (2,2)-separating codes.

Lovász local lemma

Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

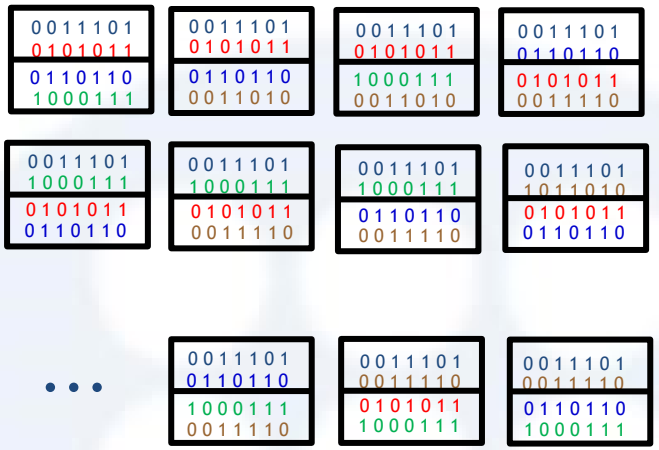
If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

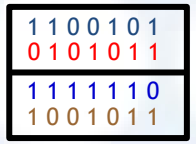
sets of disjoint pairs of pairs



Define the Event

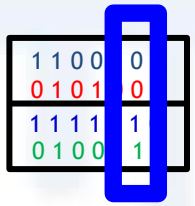
E(i,j)

‘Bad event’



Not Separated : No Discriminating index

‘Good event’



Separated :

Discriminating index

Existence of (2,2)-separating codes.

Lovász local lemma

Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

PROBABILITY OF A BAD EVENT

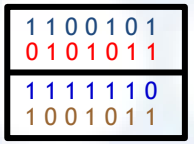
Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

size code

length code

'Bad event'



Not Separated : No Discriminating index

probability = (7/8)^{length code}

Existence of (2,2)-separating codes.

Lovász local lemma

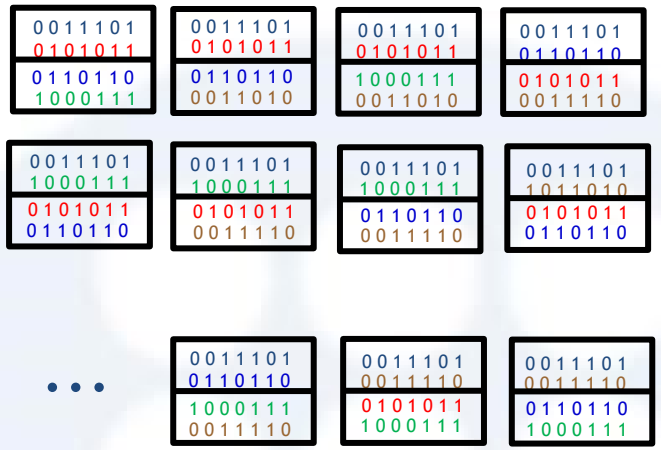
Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **ep(d+1) <= 1**

then there is a nonzero probability that none of the events occurs.

sets of disjoint pairs of pairs



Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

size code

length code

DEPENDENCE OF BAD EVENTS

Existence of (2,2)-separating codes.

Lovász local lemma

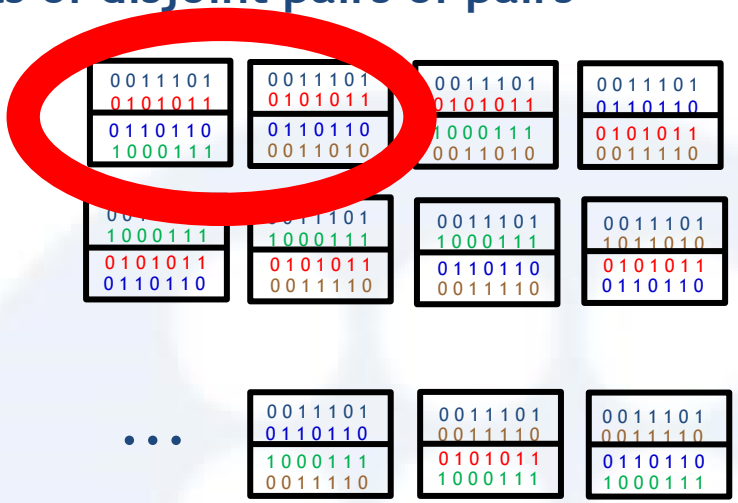
Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

sets of disjoint pairs of pairs



Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

size code

length code

Dependent

(share at least a row)

Existence of (2,2)-separating codes.

Lovász local lemma

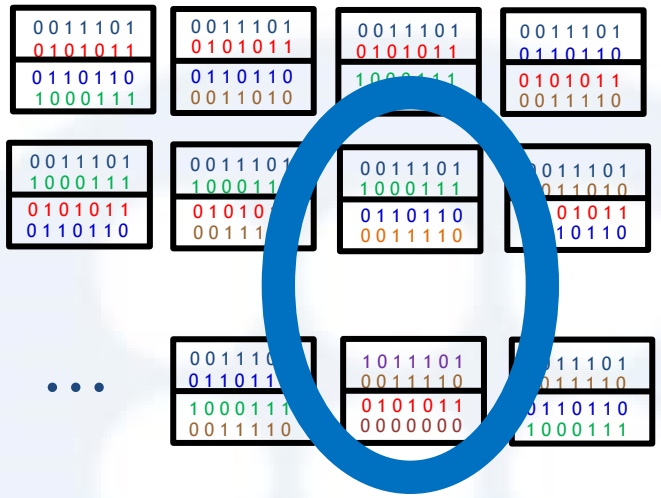
Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

sets of disjoint pairs of pairs



Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

size code

length code

Independent!!

do not share rows

Existence of (2,2)-separating codes.

Lovász local lemma

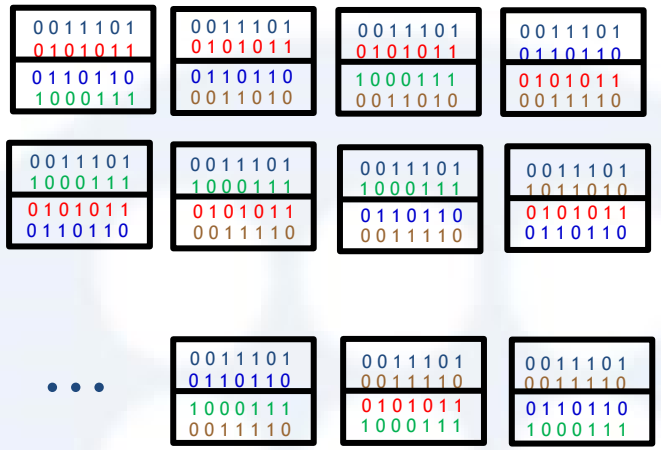
Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

sets of disjoint pairs of pairs



Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

size code

length code

dependence < 5{size code}^3

Existence of (2,2)-separating codes.

Lovász local lemma

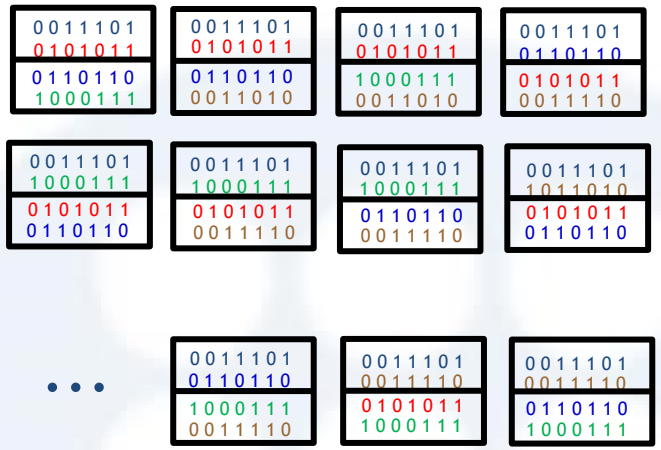
Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **ep(d+1) <= 1**

then there is a nonzero probability that none of the events occurs.

sets of disjoint pairs of pairs



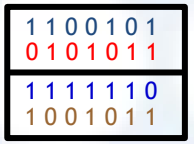
Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

size code

length code

‘Bad event’



Not Separated : No Discriminating index

prob = (7/8)^{length code}

dependence < 5{size code}^3

Existence of (2,2)-separating codes.

Lovász local lemma

Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other even
except for at most d of them.

If **ep(d+1) <= 1**

then there is a nonzero proba
that none of the events occu

Theorem:

There exist 2-separating codes of size

size code = CONST · (8/7)$^{\text{code length}/3}$

Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

size code

length code

'Bad event'

1	1	0	0	1	0	1
0	1	0	1	0	1	1
1	1	1	1	1	1	0
1	0	0	1	0	1	1

Not Separated : No Discriminating index

prob = (7/8)^{length code}

dependence < 5{size code}^3

Existence of (2,2)-separating codes.

Lovász local lemma

Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **ep(d+1) <= 1**

then there is a nonzero probability that none of the events occurs.

Theorem:

There exist 2-separating codes of size

size code = CONST · (8/7)$^{\text{code length}/3}$

Proof:

e·((7/8)^{length code})(5{size code}^3)<= 1

Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

size code

length code

'Bad event'

1 1 0 0 1 0 1
0 1 0 1 0 1 1

1 1 1 1 1 1 0
1 0 0 1 0 1 1

Not Separated : No Discriminating index

prob = (7/8)^{length code}

dependence < 5{size code}^3

Existence of (2,2)-separating codes.

Lovász local lemma

Let $A_1, A_2, \ldots, A_k$ be a sequence of **bad** events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **$ep(d+1) <= 1$**

then there is a nonzero probability that none of the events occurs.

Theorem:

There exist 2-separating codes of size

$$\text{size code} = \text{CONST} \cdot (8/7)^{\text{code length}/3}$$

with

$$\frac{\text{information bits}}{\text{sent bits}} > 0$$

positive rate

Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

size code

length code

Algorithmic construction

Algorithmic Construction.

Bernoulli random variables

x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17}
x_{21} . . .

SAMPLE

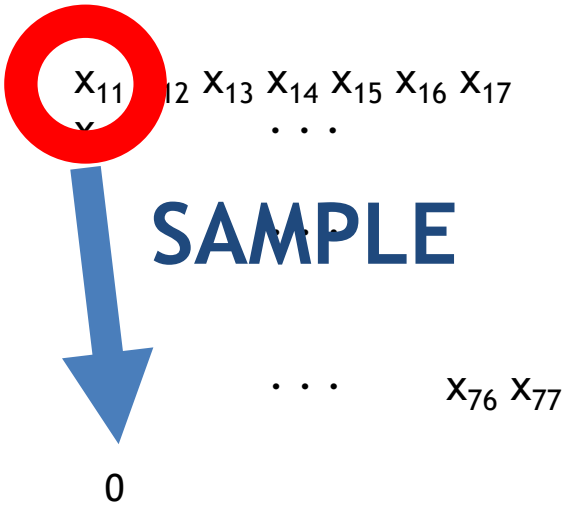
. . . x_{76} x_{77}

$$\Pr(x_{ij} = 0) = \frac{1}{2}$$

$$\Pr(x_{ij} = 1) = \frac{1}{2}$$

Algorithmic Construction.

Bernoulli random variables

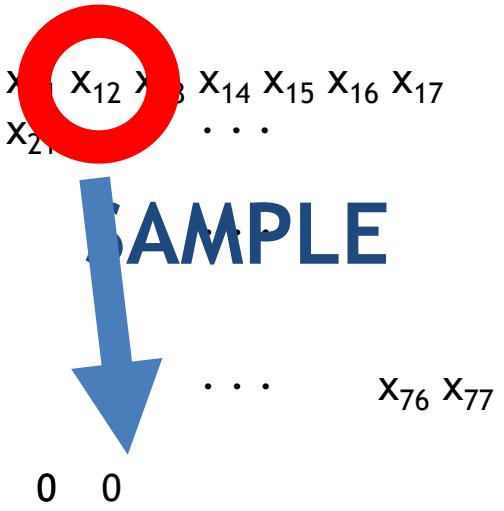


$$\Pr(x_{ij} = 0) = \tfrac{1}{2}$$

$$\Pr(x_{ij} = 1) = \tfrac{1}{2}$$

Algorithmic Construction.

Bernoulli random variables

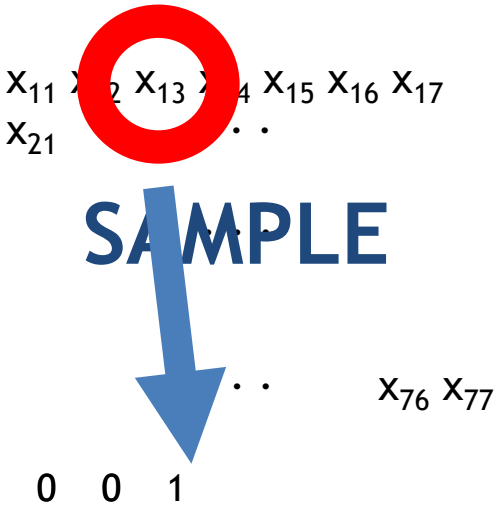


$$\Pr(x_{ij} = 0) = \tfrac{1}{2}$$

$$\Pr(x_{ij} = 1) = \tfrac{1}{2}$$

Algorithmic Construction.

Bernoulli random variables



$$\Pr(x_{ij} = 0) = \tfrac{1}{2}$$

$$\Pr(x_{ij} = 1) = \tfrac{1}{2}$$

Algorithmic Construction.

Bernoulli random variables

x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17}
x_{21} . . .

SAMPLE

. . . x_{76} x_{77}

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

sets of disjoint pairs of pairs

Algorithmic Construction.

Bernoulli random variables

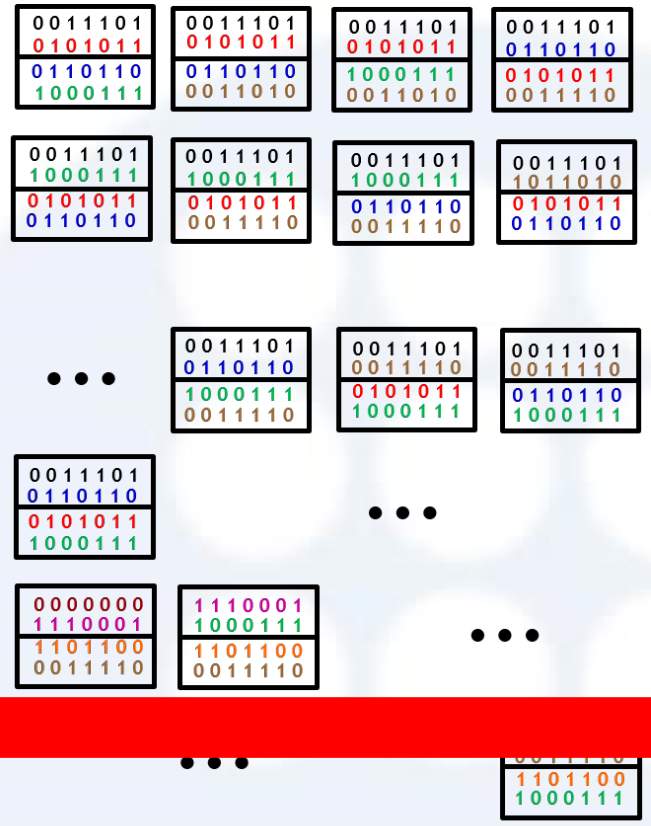
$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}$
x_{21} . . .

SAMPLE

. . . $x_{76}\ x_{77}$

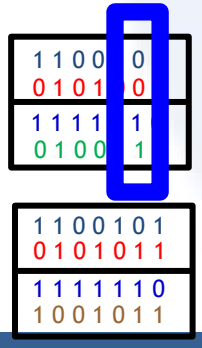
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0

sets of disjoint pairs of pairs



‘Good event’

‘Bad event’

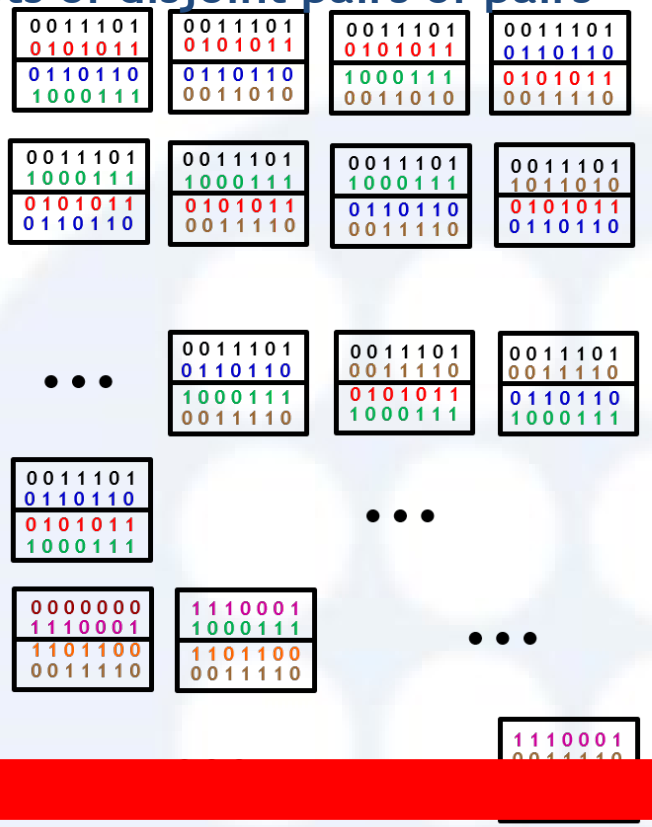


Separated : Discriminating index

Algorithmic Construction.

Bernoulli random variables

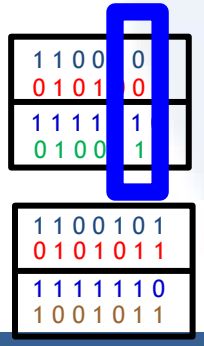
sets of disjoint pairs of pairs



Avoid all bad events

‘Good event’

‘Bad event’



Separated : Discriminating index

Not Separated : No Discriminating index

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

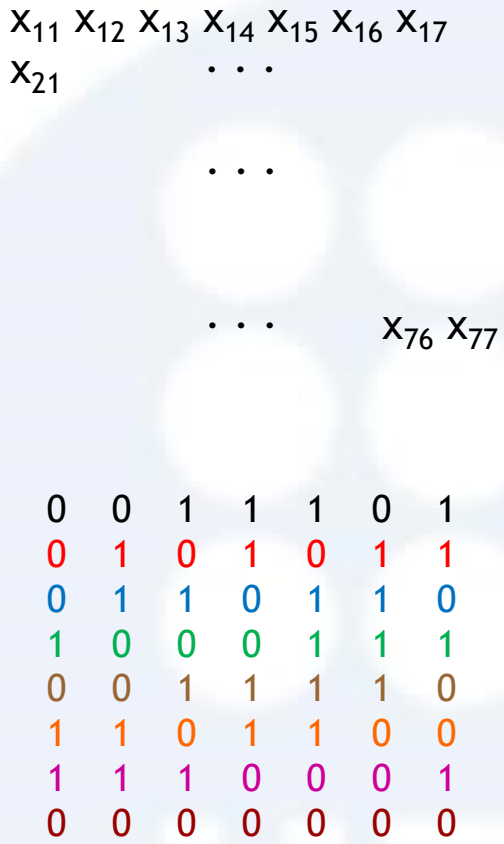
Algorithmic Lovász Local Lemma

Moser-Tardos

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

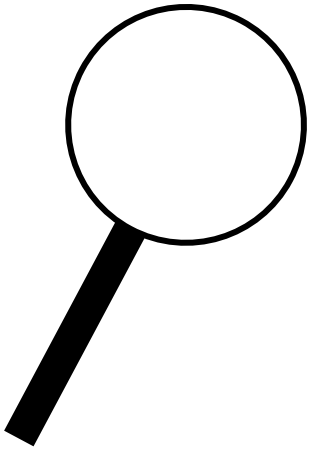


Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {



$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}$ 0 0 1 1 1 0 1

x_{21} . . . 0 1 0 1 0 1 1

0 1 1 0 1 1 0

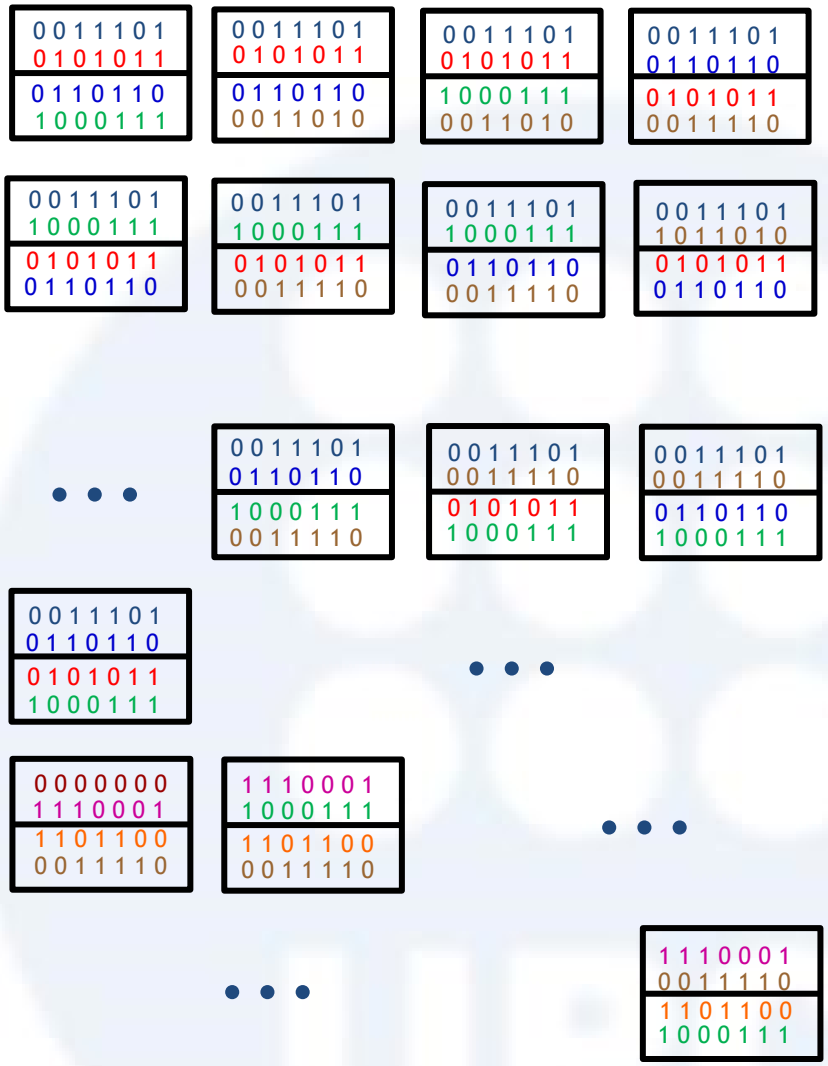
. . . 1 0 0 0 1 1 1

0 0 1 1 1 1 0

1 1 0 1 1 0 0

1 1 1 0 0 0 1

. . . $x_{76}\ x_{77}$ 0 0 0 0 0 0 0

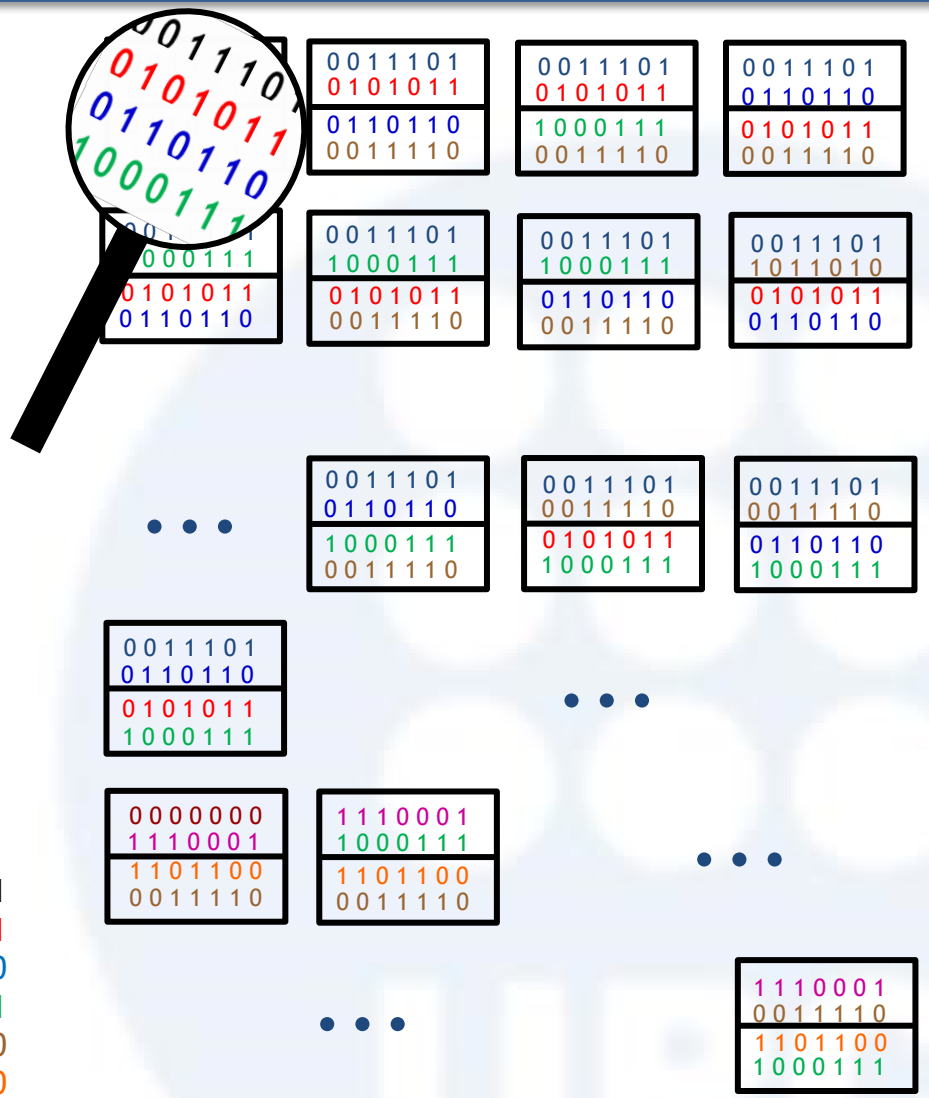


Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {



$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}$ 0 0 1 1 1 0 1

x_{21} . . . 0 1 0 1 0 1 1

0 1 1 0 1 1 0

. . . 1 0 0 0 1 1 1

0 0 1 1 1 1 0

1 1 0 1 1 0 0

1 1 1 0 0 0 1

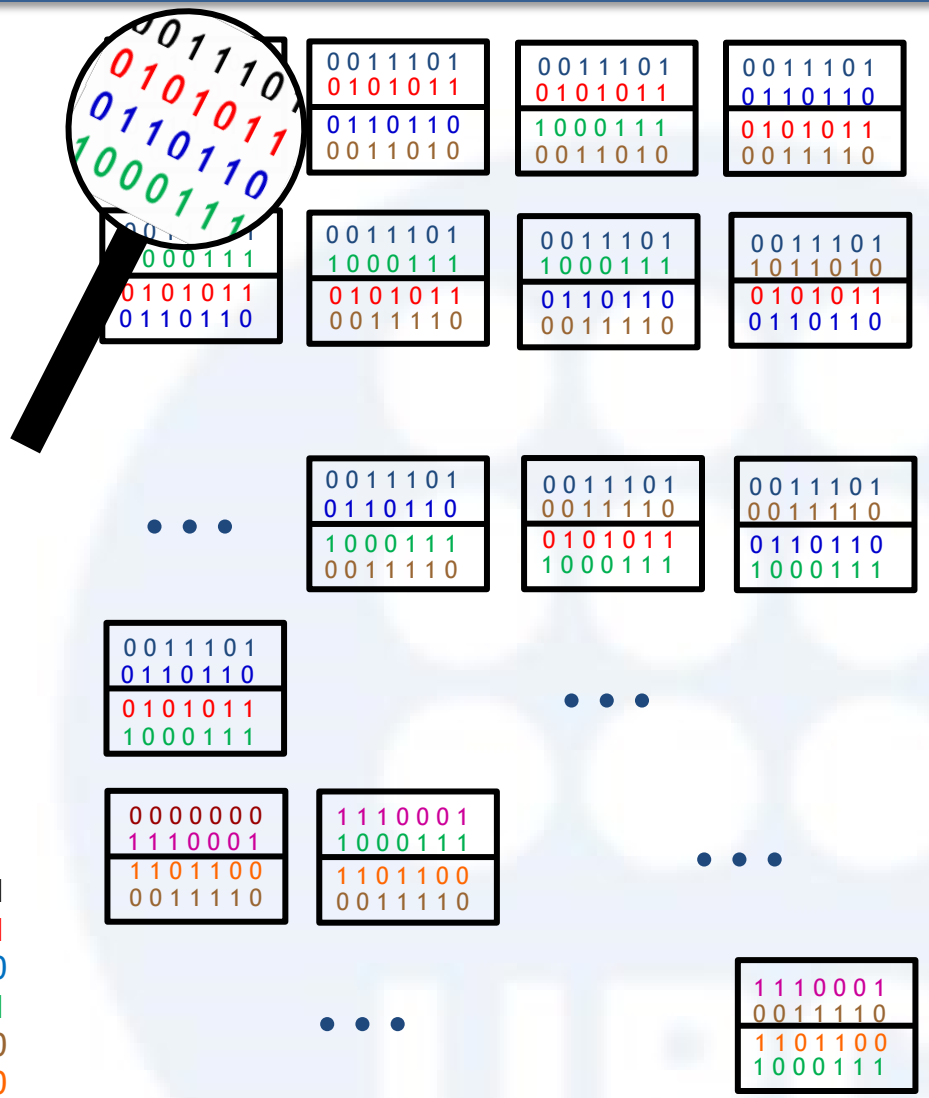
. . . $x_{76}\ x_{77}$ 0 0 0 0 0 0 0

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {



$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}$ 0 0 1 1 1 0 1

x_{21} . . . 0 1 0 1 0 1 1

0 1 1 0 1 1 0

. . . 1 0 0 0 1 1 1

0 0 1 1 1 1 0

1 1 0 1 1 0 0

1 1 1 0 0 0 1

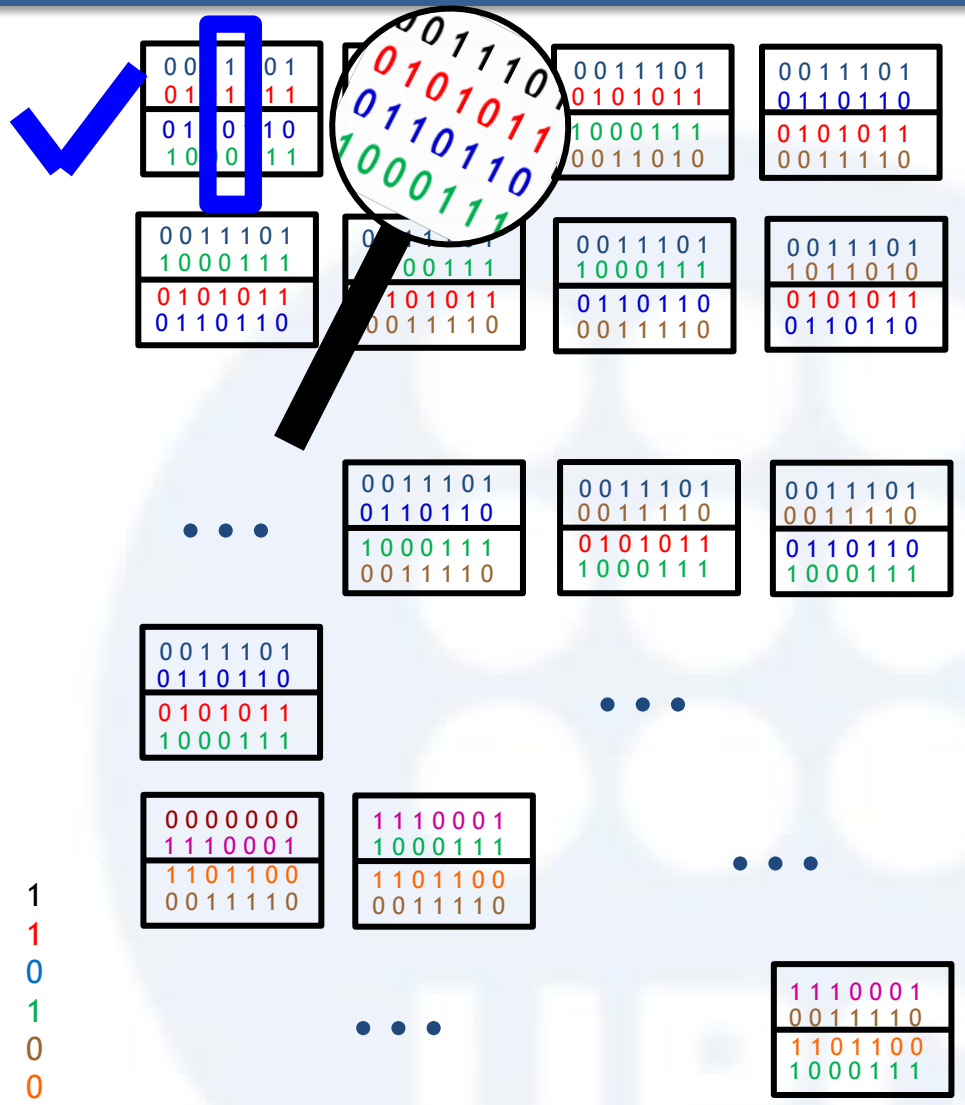
. . . $x_{76}\ x_{77}$ 0 0 0 0 0 0 0

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {



$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}$ 0 0 1 1 1 0 1

x_{21} . . . 0 1 0 1 0 1 1

0 1 1 0 1 1 0

. . . 1 0 0 0 1 1 1

0 0 1 1 1 1 0

1 1 0 1 1 0 0

1 1 1 0 0 0 1

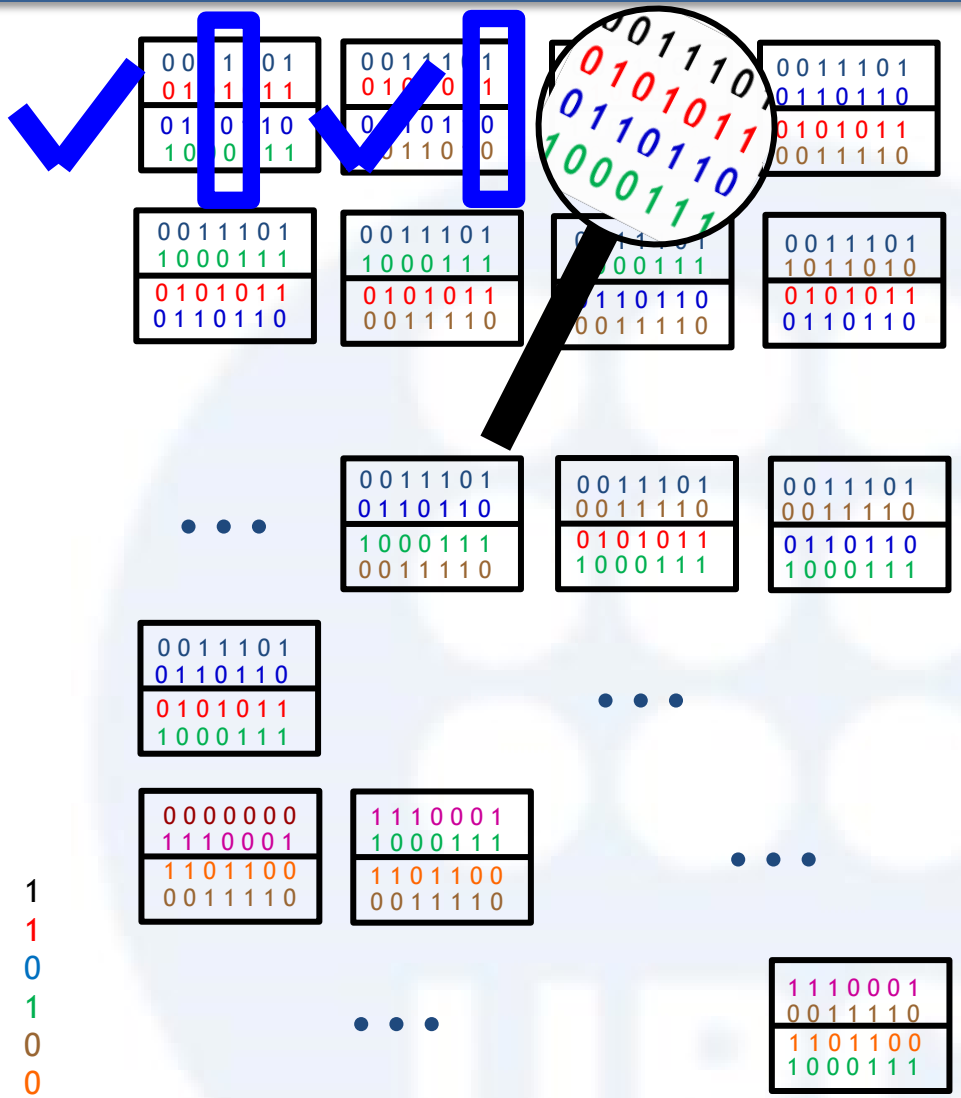
. . . $x_{76}\ x_{77}$ 0 0 0 0 0 0 0

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {



$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}$ 0 0 1 1 1 0 1

x_{21} . . . 0 1 0 1 0 1 1

0 1 1 0 1 1 0

. . . 1 0 0 0 1 1 1

0 0 1 1 1 1 0

1 1 0 1 1 0 0

1 1 1 0 0 0 1

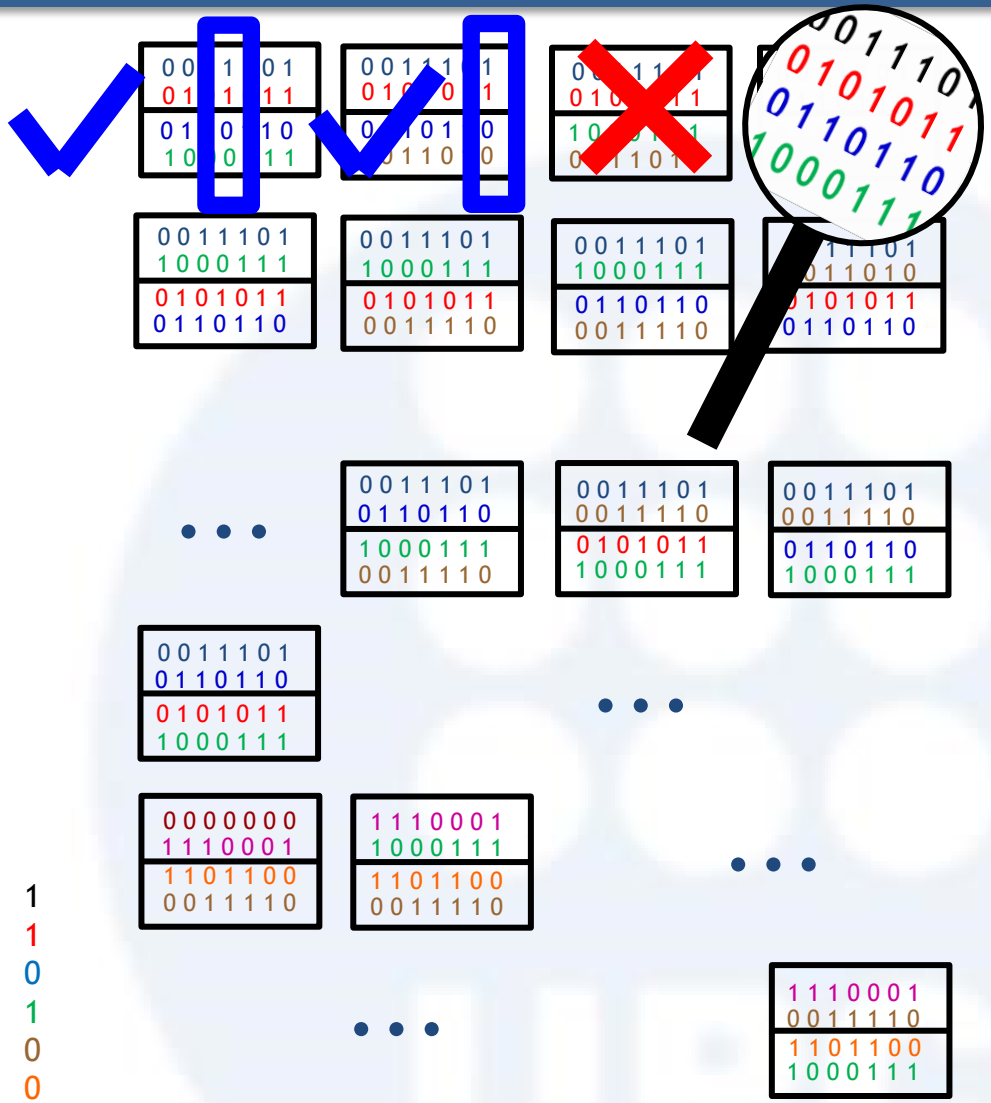
. . . $x_{76}\ x_{77}$ 0 0 0 0 0 0 0

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {



x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17}	0	0	1	1	1	0	1
x_{21} . . .	0	1	0	1	0	1	1
	0	1	1	0	1	1	0
. . .	1	0	0	0	1	1	1
	0	0	1	1	1	1	0
	1	1	0	1	1	0	0
	1	1	1	0	0	0	1
. . . x_{76} x_{77}	0	0	0	0	0	0	0

Algorithmic Construction.

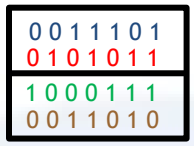
Algorithm:

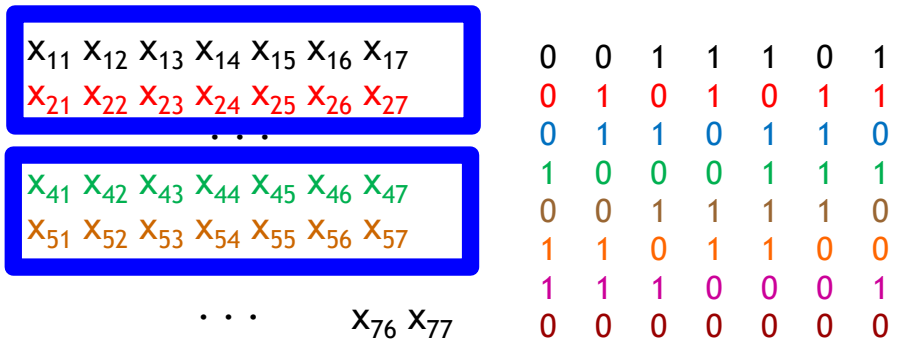
sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j





0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

Algorithmic Construction.

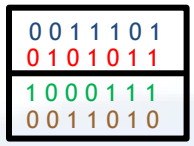
Algorithm:

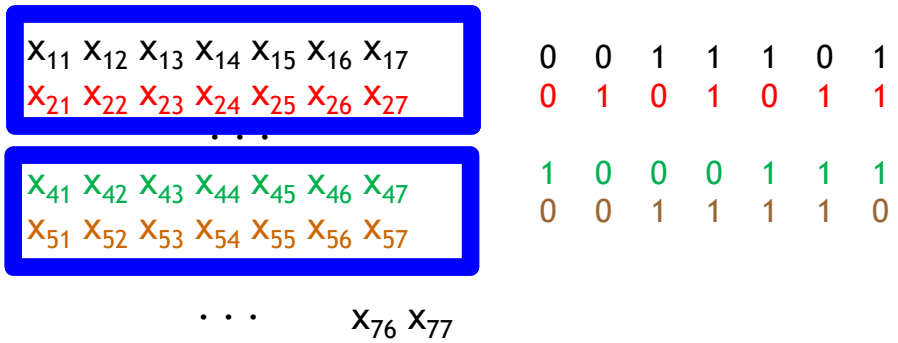
sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j





0 0 1 1 1 0 1
0 1 0 1 0 1 1

1 0 0 0 1 1 1
0 0 1 1 1 1 0

Algorithmic Construction.

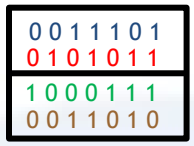
Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j



x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17}
x_{21} x_{22} x_{23} x_{24} x_{25} x_{26} x_{27}
. . .
x_{41} x_{42} x_{43} x_{44} x_{45} x_{46} x_{47}
x_{51} x_{52} x_{53} x_{54} x_{55} x_{56} x_{57}
. . . x_{76} x_{77}

0 0 1 1 1 0 1
0 1 0 1 0 1 1

1 0 0 0 1 1 1
0 0 1 1 1 1 0

$\Pr(x_{ij} = 0) = ½$

$\Pr(x_{ij} = 1) = ½$

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

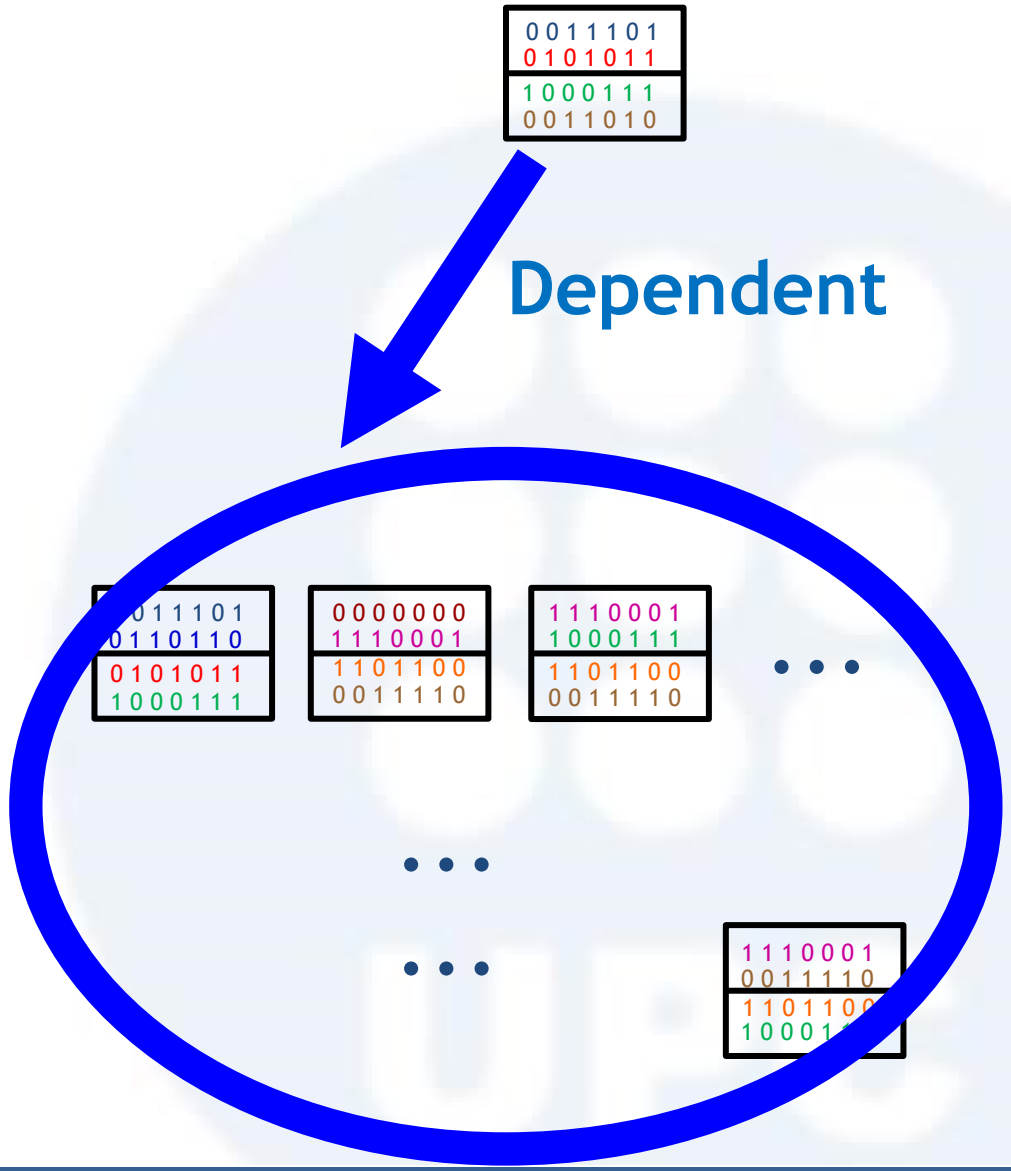
resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}



x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17}			0	0	1	1	1	0	1
x_{21}	. . .		0	1	0	1	0	1	1
			0	1	1	0	1	1	0
	. . .		1	0	0	0	1	1	1
			1	0	1	1	0	1	0
			1	1	0	1	1	0	0
			1	1	1	0	0	0	1
	. . .	x_{76} x_{77}	0	0	0	0	0	0	0

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

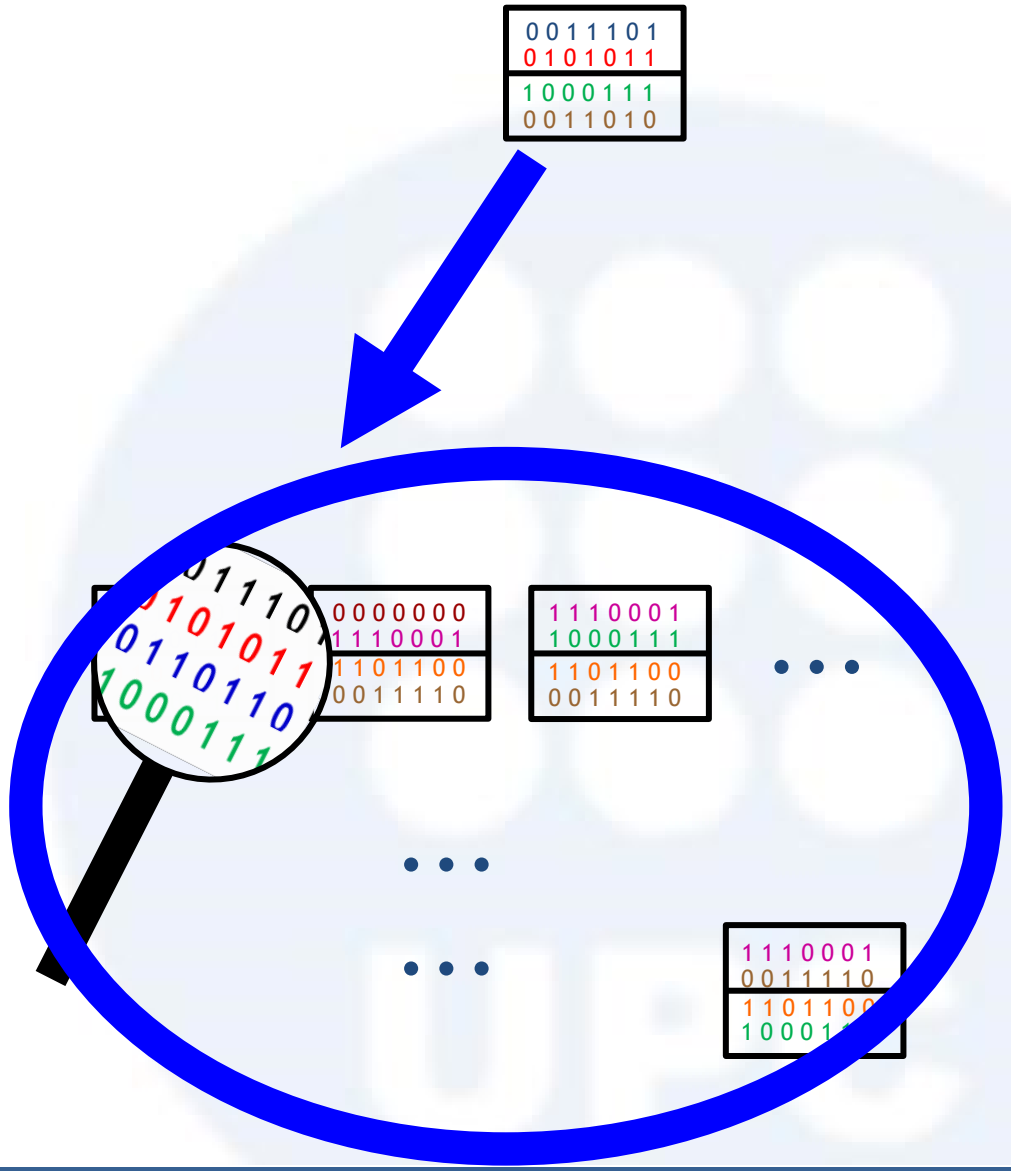
resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}



x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17}							0 0 1 1 1 0 1
x_{21}	. . .						0 1 0 1 0 1 1
							0 1 1 0 1 1 0
	. . .						1 0 0 0 1 1 1
							1 0 1 1 0 1 0
							1 1 0 1 1 0 0
							1 1 1 0 0 0 1
	. . .	x_{76} x_{77}					0 0 0 0 0 0 0

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

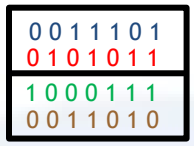
resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

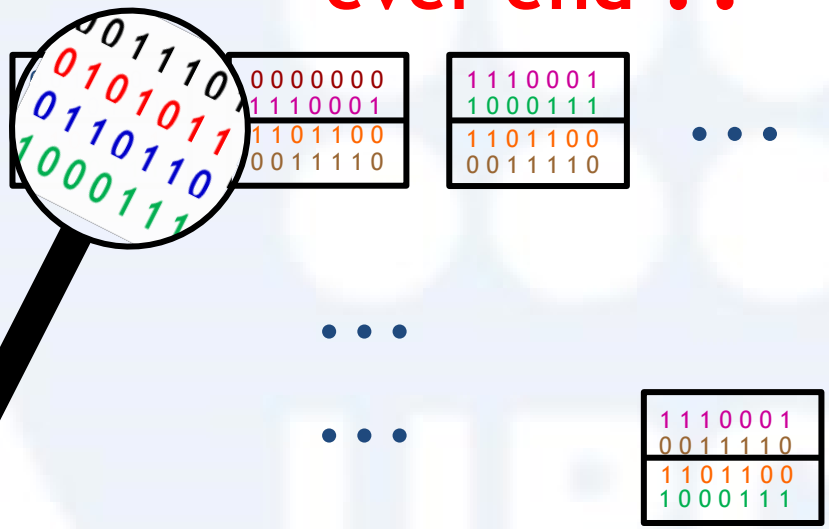
RESAMPLE E_k

}

}



Does all this looking up and resampling ever end ??



x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17}			0	0	1	1	1	0	1
x_{21}	. . .		0	1	0	1	0	1	1
			0	1	1	0	1	1	0
	. . .		1	0	0	0	1	1	1
			1	0	1	1	0	1	0
			1	1	0	1	1	0	0
			1	1	1	0	0	0	1
	. . .	x_{76} x_{77}	0	0	0	0	0	0	0

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

Theorem

For every **code length** > 0,

there is a randomized algorithm,

such that

the probability of it lasting for at least N rounds

is inverse exponential in N, and
that outputs a 2-separating code of size

$$\text{size code} = \text{CONST} \cdot (8/7)^{\text{code length}/3}$$

Sketch of the proof

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

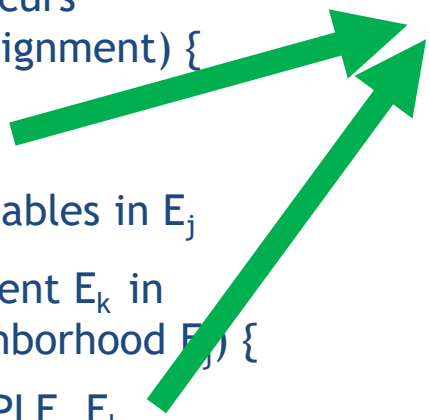
resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}



✓ **Directly compute the probability of having at least n RESAMPLE phases.**

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

✓ **Directly compute the probability of having at least n RESAMPLE phases.**

✓ **In the literature most of the approaches compute the expectation of the number of RESAMPLE phases.**

Algorithmic Construction.

Algorithm:

```
sample X_ij  <- current assignment
while (bad event E_j occurs
        in current assignment) {
    RESAMPLE  E_j
      resample variables in E_j
      while (bad event E_k in
            neighborhood E_j) {
          RESAMPLE  E_k
      }
}
```

- ✓ **Directly compute the probability of having at least n RESAMPLE phases.**
- ✓ **In the literature most of the approaches compute the expectation of the number of RESAMPLE phases.**
- ✓ **Probability of at least n RESAMPLE phases is inverse exponential to n**

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

We want to show this

✓ **Directly compute the probability of having at least n RESAMPLE phases.**

✓ **In the literature most of the approaches compute the expectation of the number of RESAMPLE phases.**

✓ **Probability of at least n RESAMPLE phases is inverse exponential to n**

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

We want to show this

- ✓ **Directly compute the probability of having at least n RESAMPLE phases.**
- ✓ **In the literature most of the approaches compute the expectation of the number of RESAMPLE phases.**
- ✓ **Probability of at least n RESAMPLE phases is inverse exponential to n**

We want to upper bound this probability

Algorithmic Construction.

Algorithm:

```
sample X_ij  <- current assignment
while (bad event E_j occurs
        in current assignment) {
    RESAMPLE  E_j
      resample variables in E_j
      while (bad event E_k in
            neighborhood E_j) {
          RESAMPLE  E_k
      }
}
```

✓ Directly compute the probability of having at least n RESAMPLE phases.

✓ In the literature most of the approaches compute the expectation of the number of RESAMPLE phases.

✓ Probability of at least n RESAMPLE phases is exponential to n

We want to upper bound this probability

we draw a picture of how the resamplings occur

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in neighborhood {

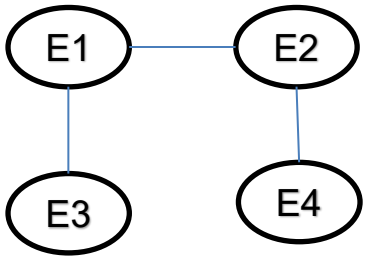
RESAMPLE E_k

}

}

we want to draw a picture of how the resamplings occur

We need to know how the events depend on each other



Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in neighborhood {

RESAMPLE E_k

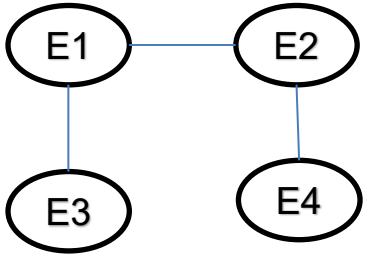
}

}

we want to draw a picture of how the resamplings occur

Example of an execution:

We need to know how the events depend on each other



Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in neighborhood {

RESAMPLE E_k

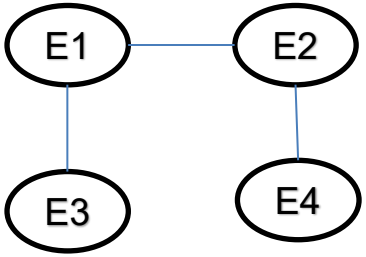
}

}

we want to draw a picture of how the resamplings occur

Example of an execution:

We need to know how the events depend on each other



Algorithmic Construction.

Algorithm:

sample X_i <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in neighborhood {

RESAMPLE E_k

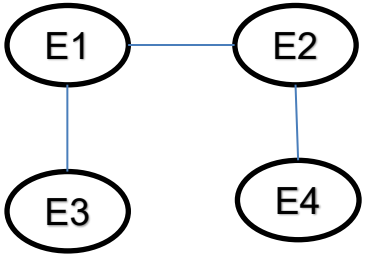
}

}

we want to draw a picture of how the resamplings occur

Example of an execution:

We need to know how the events depend on each other



Algorithmic Construction.

Algorithm:

sample X_i <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in neighborhood of E_j)

RESAMPLE E_k

}

}

we want to draw a picture of how the resamplings occur

Example of an execution:

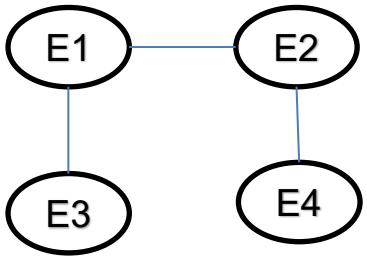
Phase 1 E1

$Event_1$ occurs (fail)

RESAMPLE E_1

time

We need to know how the events depend on each other



Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

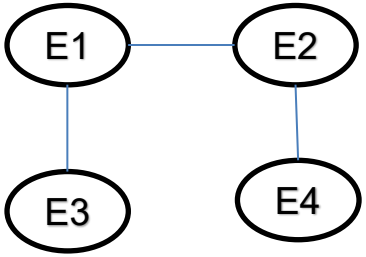
while (bad event E_k in neighborhood {

RESAMPLE E_k

}

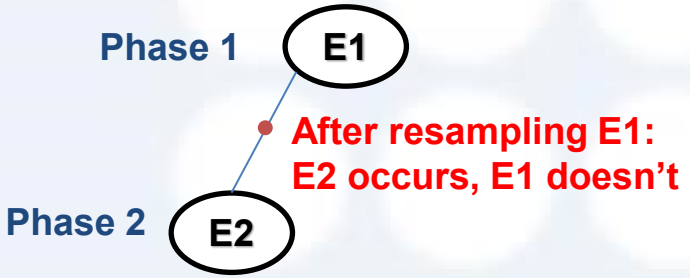
}

We need to know how the events depend on each other



we want to draw a picture of how the resamplings occur

Example of an execution:



Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

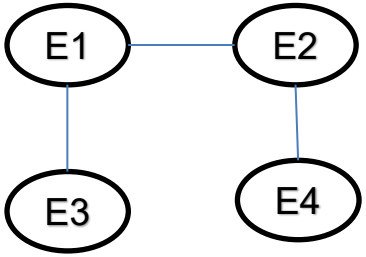
while (bad event E_k in neighborhood {

RESAMPLE E_k

}

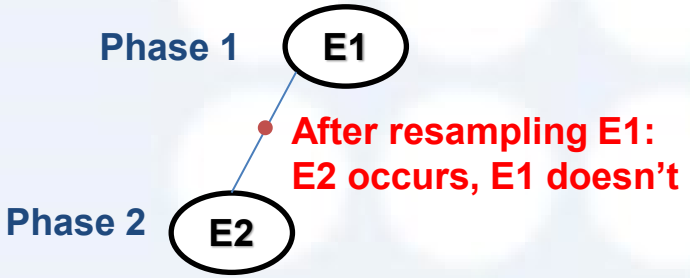
}

We need to know how the events depend on each other



we want to draw a picture of how the resamplings occur

Example of an execution:



Do as before !!
Resample E2

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

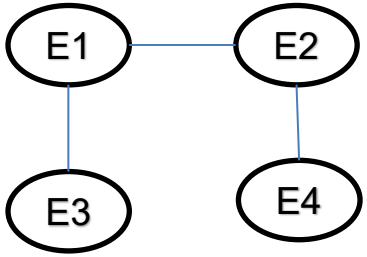
while (bad event E_k in neighborhood {

RESAMPLE E_k

}

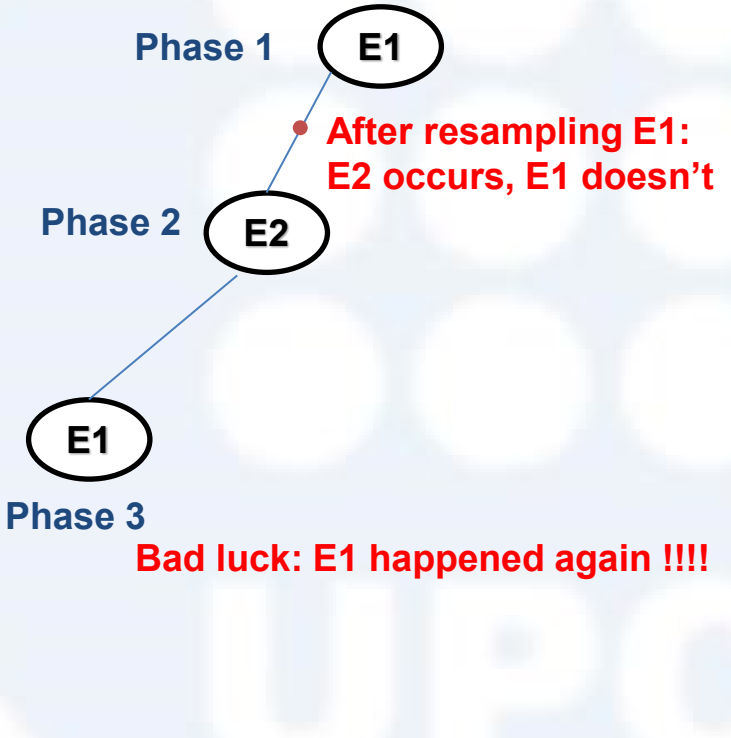
}

We need to know how the events depend on each other



we want to draw a picture of how the resamplings occur

Example of an execution:



Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

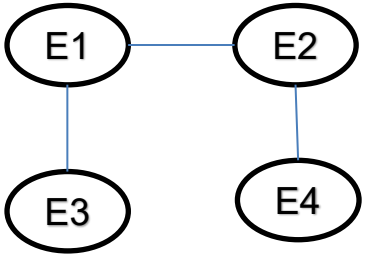
while (bad event E_k in neighborhood {

RESAMPLE E_k

}

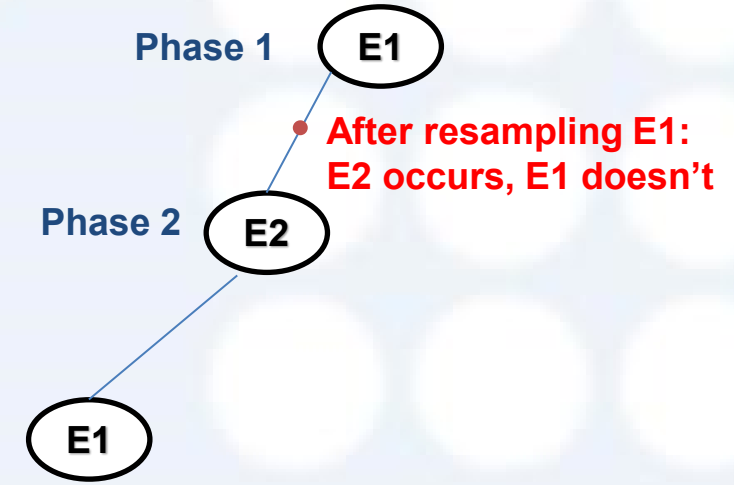
}

We need to know how the events depend on each other



we want to draw a picture of how the resamplings occur

Example of an execution:



time

Resample E1 again!!!!!!

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

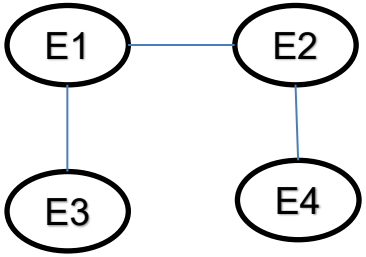
while (bad event E_k in neighborhood {

RESAMPLE E_k

}

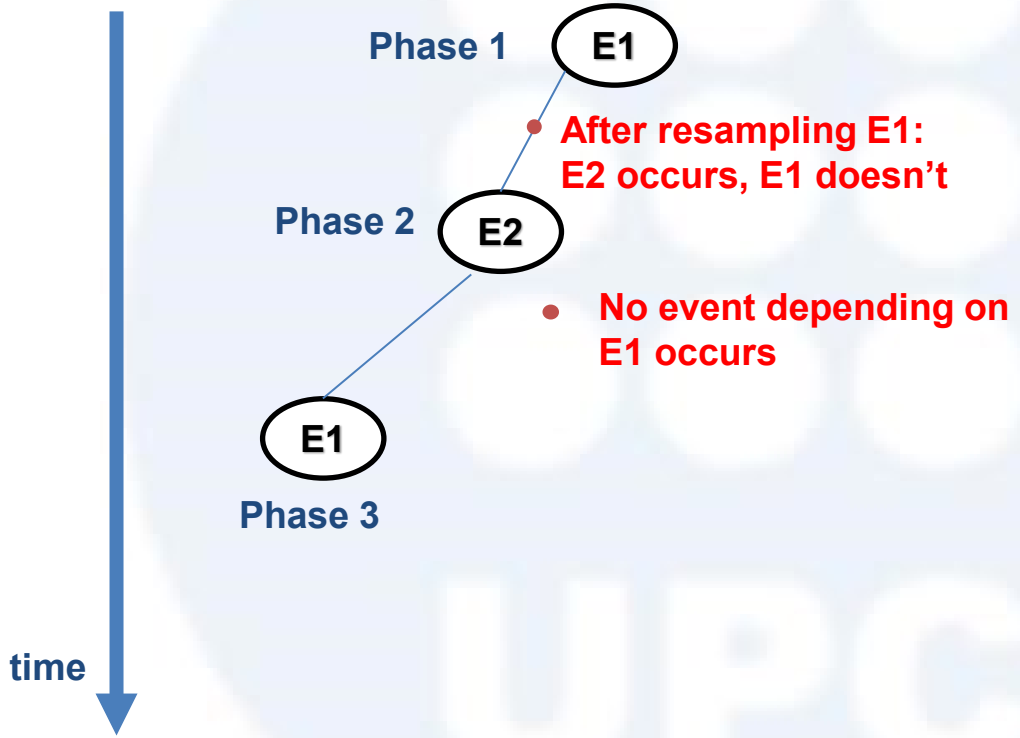
}

We need to know how the events depend on each other



we want to draw a picture of how the resamplings occur

Example of an execution:



Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in neighborhood {

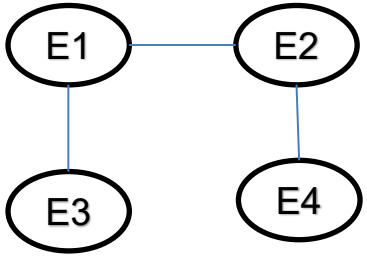
RESAMPLE E_k

}

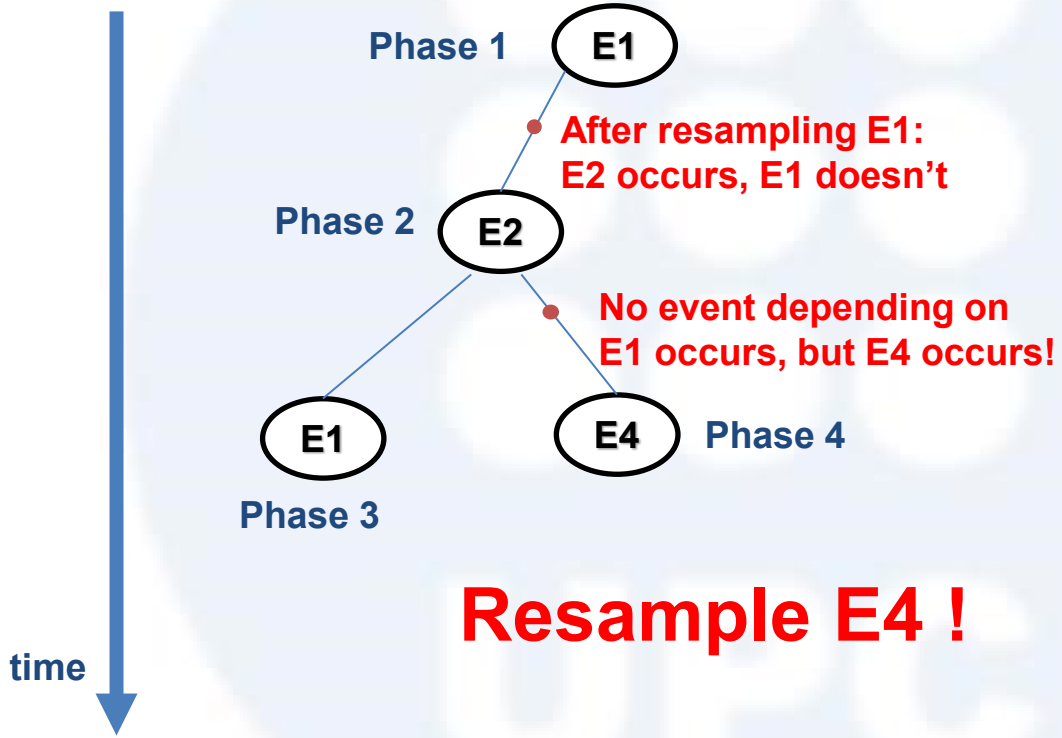
}

we want to draw a picture of how the resamplings occur

We need to know how the events depend on each other



Example of an execution:



Resample E4 !

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in neighborhood {

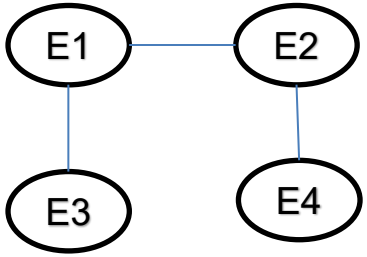
RESAMPLE E_k

}

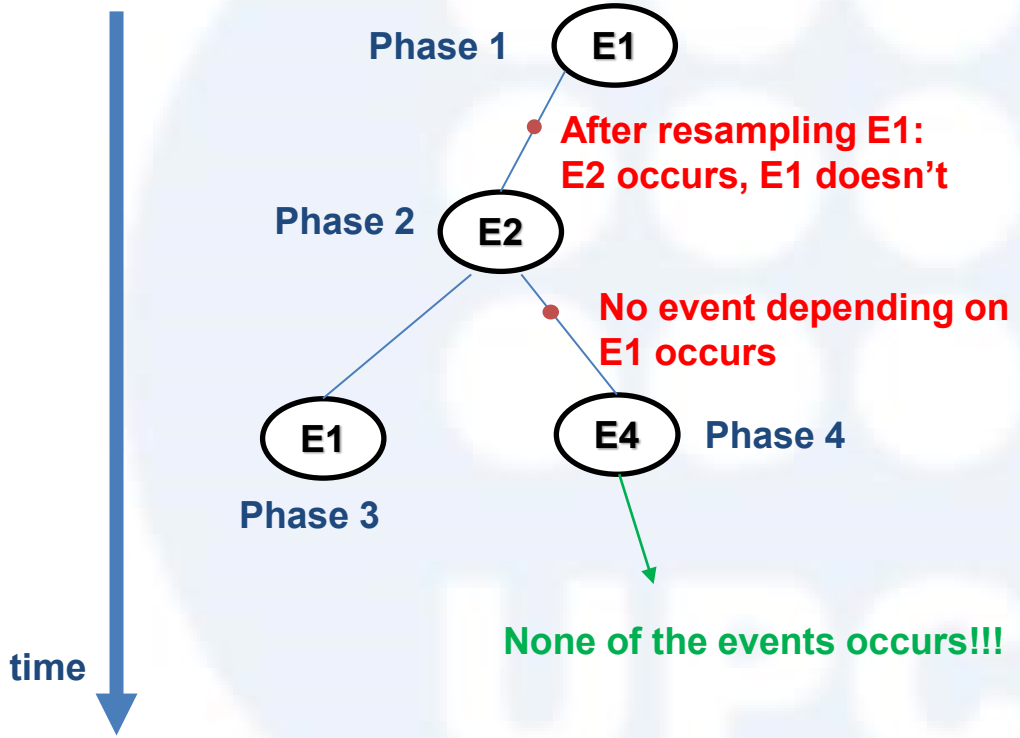
}

we want to draw a picture of how the resamplings occur

We need to know how the events depend on each other



Example of an execution:



Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in neighborhood {

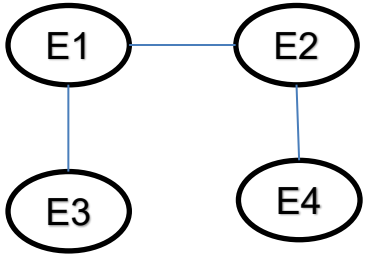
RESAMPLE E_k

}

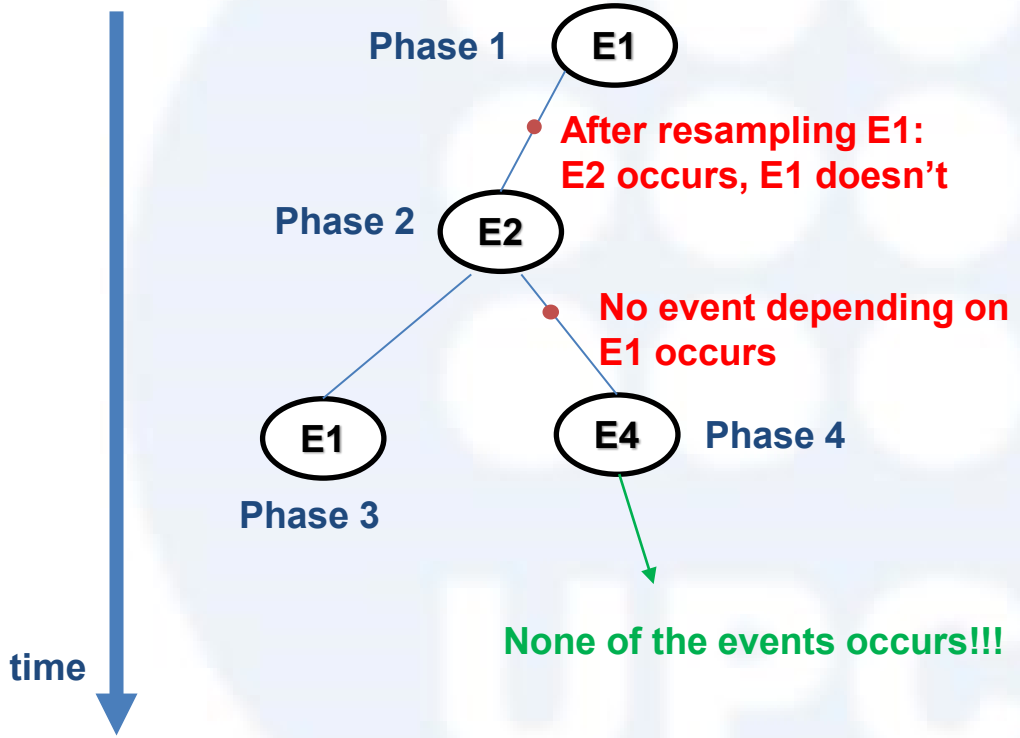
}

we want to draw a picture of how the resamplings occur

We need to know how the events depend on each other



Example of an execution:



Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs in current assignment) {

RESAMPLE E_j

resample variables in E_j

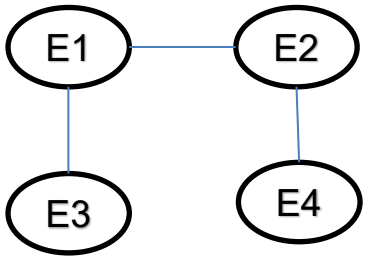
while (bad event E_k in neighborhood {

RESAMPLE E_k

}

}

We need to know how the events depend on each other



we want to draw a picture of how the resamplings occur

Example of an execution:

The proof follows because the number of forests with such trees is finite

Phase 3

None of the events occurs!!!

time

Algorithmic Construction.

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

Theorem

For every **code length** > 0,

there is a randomized algorithm,

such that

the probability of it lasting for at least N rounds

is inverse exponential in N, and
that outputs a 2-separating code of size

size code = CONST · $(8/7)^{\text{code length}/3}$

Summary

1. Given a lower bound for separating codes (using the Lovasz Local Lemma)
2. The lower bound shows there exist codes of positive rate
3. Given algorithmic constructions (best rate known).

Algoritmos para códigos separadores

Marcel Fernandez

Sebastià Martín Molleví

Universitat Politecnica de Catalunya

John Livieratos

Department of Mathematics,
National & Kapodistrian University of Athens

Algoritmos para códigos separadores

Marcel Fernandez

Sebastià Martín Molleví

Universitat Politecnica de Catalunya

John Livieratos

Department of Mathematics,
National & Kapodistrian University of Athens

Algoritmos para códigos separadores

Marcel Fernandez

Sebastià Martín Molleví

Universitat Politecnica de Catalunya

John Livieratos

Department of Mathematics,
National & Kapodistrian University of Athens

Algoritmos para códigos separadores

Marcel Fernandez

Sebastià Martín Monllevi

Universitat Politecnica de Catalunya

John Livieratos

Department of Mathematics,
National & Kapodistrian University of Athens

Algoritmos para códigos separadores

Marcel Fernandez

Sebastià Martín Monllevi

Universitat Politecnica de Catalunya

John Livieratos

Department of Mathematics,
National & Kapodistrian University of Athens

Algorithmic Aspects On The Construction Of Separating Codes

Marcel Fernandez

Department of Network Engineering
Universitat Politecnica de Catalunya

John Livieratos

Department of Mathematics,
National & Kapodistrian University of Athens

Algorithmic Aspects On The Construction Of Separating Codes

Marcel Fernandez

Department of Network Engineering
Universitat Politecnica de Catalunya

John Livieratos

Department of Mathematics,
National & Kapodistrian University of Athens

Contents

Construction of polynomial complexity

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

If we want a code with positive rate

go over the approximately $2\{\text{size code}\}^4$ events

check all the approximately $5\{\text{size code}\}^3$ events

$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}$ 0 0 1 1 1 0 1

x_{21} . . . 0 1 0 1 0 1 1

0 1 1 0 1 1 0

. . . 1 0 0 0 1 1 1

0 0 1 1 1 1 0

1 1 0 1 1 0 0

1 1 1 0 0 0 1

. . . $x_{76}\ x_{77}$ 0 0 0 0 0 0 0

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

If we want a code with positive rate

go over the approximately $2\{\text{size code}\}^4$ events

check all the approximately $5\{\text{size code}\}^3$ events

$$\text{size code} = \text{CONST} \cdot (8/7)^{\text{code length}/3}$$

$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}$ 0 0 1 1 1 0 1

x_{21} . . . 0 1 0 1 0 1 1

0 1 1 0 1 1 0

. . . 1 0 0 0 1 1 1

0 0 1 1 1 1 0

1 1 0 1 1 0 0

1 1 1 0 0 0 1

. . . $x_{76}\ x_{77}$ 0 0 0 0 0 0 0

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

If we want a code with positive rate

go over the approximately $2\{CONST\}^{\text{code length}^{4/3}}$ events

check all the approximately $5\{CONST\}^{\text{code length}}$ events

$$\text{size code} = \text{CONST} \cdot (8/7)^{\text{code length}/3}$$

x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} 0 0 1 1 1 0 1

x_{21} . . . 0 1 0 1 0 1 1

0 1 1 0 1 1 0

. . . 1 0 0 0 1 1 1

0 0 1 1 1 1 0

1 1 0 1 1 0 0

1 1 1 0 0 0 1

. . . x_{76} x_{77} 0 0 0 0 0 0 0

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

If we want a code with positive rate

go over the approximately $2\{CONST\}^{\text{code length}^{4/3}}$ events

check all the approximately $5\{CONST\}^{\text{code length}}$ events

$$\text{size code} = \text{CONST} \cdot (8/7)^{\text{code length}/3}$$

$x_{11}\ x_{12}\ x_{13}\ x_{14}\ x_{15}\ x_{16}\ x_{17}$ 0 0 1 1 1 0 1

x_{21} . . . 0 1 0 1 0 1 1

0 1 1 0 1 1 0

. . . 1 0 0 0 1 1 1

0 0 1 1 1 1 0

1 1 0 1 1 0 0

1 1 1 0 0 0 1

. . . $x_{76}\ x_{77}$ 0 0 0 0 0 0 0

The number of checkings is exponential in the input size

!!

Lovász local lemma

Let $A_1, A_2, \ldots, A_k$ be a sequence of events such that each event

- occurs with probability at most p
- is independent of all the other events except for at most d of them.

If **ep(d+1) <= 1**

then there is a nonzero probability that none of the events occurs.

We want constructions of polynomial complexity in the input size (code length)

Theorem:

For every **code_length > 0** and every **a > 0**
There exist 2-separating codes of size

size code = CONST · (code_length)$^{a/3}$

Code

0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
0	0	1	1	1	1	0
1	1	0	1	1	0	0
1	1	1	0	0	0	1
0	0	0	0	0	0	0

Algorithm:

sample X_{ij} <- current assignment

while (bad event E_j occurs
in current assignment) {

RESAMPLE E_j

resample variables in E_j

while (bad event E_k in
neighborhood E_j) {

RESAMPLE E_k

}

}

We want constructions of polynomial complexity in the input size (code length)

go over approximately **polynomial number of** events

check approximately **polynomial number of** events

size code = CONST · $(code\_length)^{a/3}$

x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} 0 0 1 1 1 0 1
x_{21} . . . 0 1 0 1 0 1 1
0 1 1 0 1 1 0
. . . 1 0 0 0 1 1 1
0 0 1 1 1 1 0
1 1 0 1 1 0 0
1 1 1 0 0 0 1
. . . x_{76} x_{77} 0 0 0 0 0 0 0

The number of checkings is polynomial in the input size

rate tending to zero

go over approximately **polynomial number of** events

but

check approximately **polynomial number of** events

With better rate than

$$\text{size code} = \text{CONST} \cdot (\text{code\_length})^{a/3}$$

Dual Hamming code
Haddamard code
Simplex code

0	1
1	1
1	0
1	1
1	0
0	0
0	1
0	0

The number of checkings is polynomial in the input size