

# Extended Combinatorial Testing

*With Graph Algorithms and Apache Spark*

**By**

**Edmond La Chance**

**Director: Sylvain Hallé**

**UQAC**

Université du Québec  
à Chicoutimi





# About me

I was born in Montreal, and I now live in Chicoutimi.

I'm a fan of RPG games like D&D, Pathfinder and Call of Cthulhu.

I used to be a very active foosball player... I've been playing less for the last 2 years, but I hope to compete again one day!

# Introduction

# THE CISQ report

CISQ estimates in their report that in 2018, the total cost related to software bugs is \$1.53 trillion, or 7% of the US GDP in 2018.

CISQ estimates, with their 2020 report, that this cost has increased by 26% to \$2.08 trillion.

Catégorie	%
Losses from software failures	37.46
Legacy system problems	21.42
Technical debt	18.22
Finding/fixing defects	16.87
Troubled/canceled projects	6.01

# The case of the Therac-25



# Problem statement

Why do we do all this work to find bugs, and make our software more reliable, and predictable?

Bugs and unexpected behavior can be catastrophic to people's safety.

Software problems have a very high cost to the economy. If we save on this, we can invest money elsewhere.

**How do we prevent these problems?**

# Techniques of Software Verification

1. Manual Code Inspection
2. Static analysis, Runtime analysis
  - Code Smells
  - Model Checking
  - Runtime Monitoring
3. Test case generation
  - Black-box Fuzzing
  - Concolic Testing
  - **Combinatorial tests**







```
1 RandomFuzzing(input seed) {
2   int numWrites = random(len(seed)/1000)+1;
3   input newInput = seed;
4   for (int i=1; i<=numWrites; i++) {
5     int loc = random(len(seed));
6     byte value = (byte)random(255);
7     newInput[loc] = value;
8   }
9   result = ExecuteAppWith(newInput);
10  if (result == crash) print("bug found!");
11 }
```

Godefroid, P. (2020). Fuzzing: Hack, art, and science. Commun. ACM, 63(2), 70–76.



# Combinatorial testing



 a	1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
 b	1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3
 c	1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3
 a	3 2 1 2 1 3 1 3 2
 b	1 2 3 1 2 3 1 2 3
 c	1 1 1 2 2 2 3 3 3

# t-way testing

- A sequence of  $t=2$  tests is a sequence in which each parameter-value pair appears at least once.
- We can call these combinations of values "combos".
- Empirical studies by Kuhn et al. have shown that t-way tests are very effective than exhaustive tests in finding bugs.

```
Type:           Single, Spanned, Striped, Mirror, RAID-5
Size:           10, 100, 1000, 10000, 40000
Format method: Quick, Slow
File system:    FAT, FAT32, NTFS
Cluster size:  512, 1024, 2048, 4096, 8192, 16384
Compression:   On, Off
```

```
#Constraints
```

```
IF [File system] = "FAT" THEN [Size] <= 4096;
IF [File system] = "FAT32" THEN [Size] <= 32000;
```

```
IF [File system] <> "NTFS" OR
  ( [File system] = "NTFS" AND [Cluster size] > 4096 )
THEN [Compression] = "Off";
```

```
IF NOT ( [File system] = "NTFS" OR
  ( [File system] = "NTFS" AND NOT [Cluster size] <= 4096 ))
THEN [Compression] = "Off";
```

Un SUT avec 6 paramètres  
et deux contraintes

Source: [Manuel de PICT](#)

# Empirical studies on t-way testing

Wallace and Kuhn (2001) study 15 years of bug reports from embedded medical applications.

t=2 finds 97% of the bugs, t=3 finds 99% and t=4 100%.

Kuhn and Reilly (2002) study Mozilla Web Browser and Apache Web Server bug reports.

t=2 finds 76% of the bugs, t=3 finds 95% of the bugs, then t=4, t=5 and t=6 further increase the performance, up to 100% of the bugs found with t=6

A study by Kuhn et al. (2004) of software used by NASA.

t=2 and t=3 find the majority of problems, and t=4 through t=6 seem to provide pseudoexhaustive coverage.

# Initial observations

# Initial observations

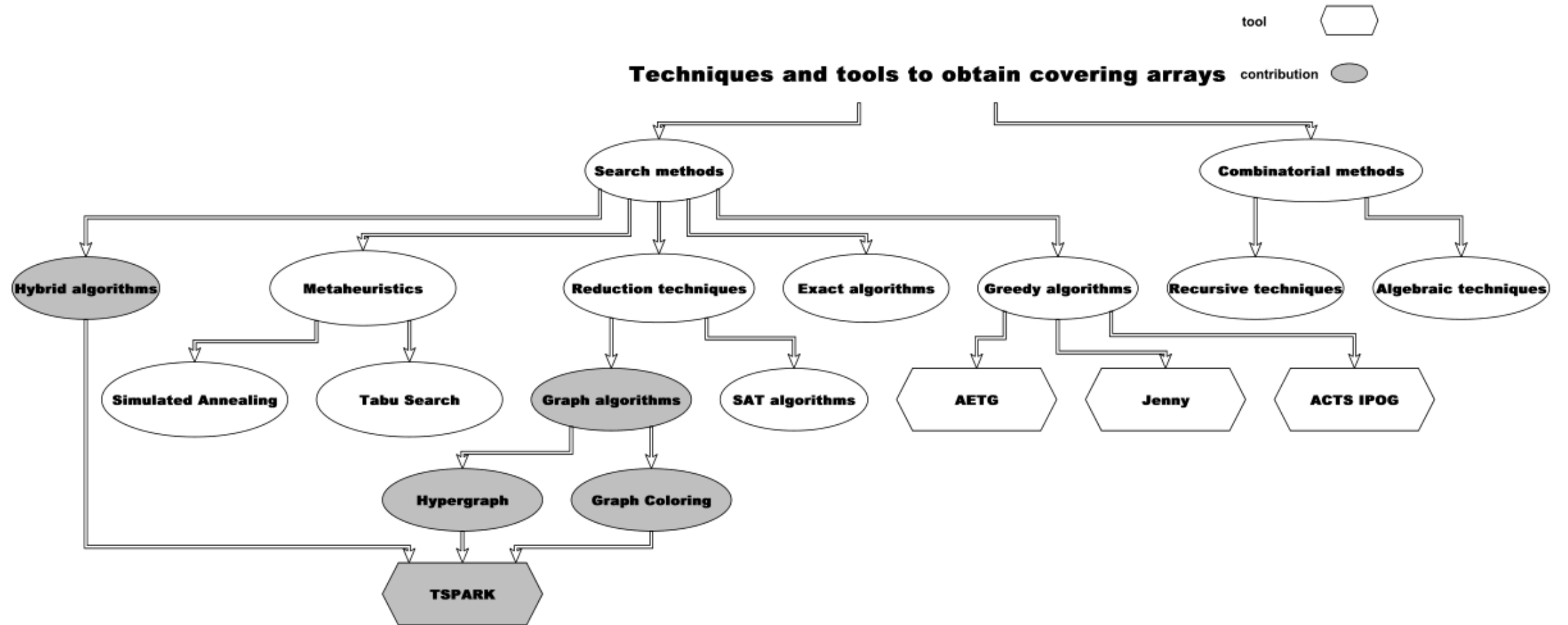
1. Useful tools to generate combinatorial tests already exist (PICT, ACTS, Jenny etc). However, these tools do not necessarily have all the desired features. For example, the support for existential constraints is very basic.
2. Existing tools have a single-threaded design, which can lead to scalability problems. No existing tool works with distributed computing.
3. A large-scale study to compare these solutions, in terms of quality, time, and how they handle the constraints is missing.

# Contributions of this thesis

1. Generalization of t-way testing to  $\Phi$ -way testing.
2. A reduction to  $\Phi$ -way testing to Graph Coloring and Hypergraph Vertex Covering
3. Implementation and design of distributed algorithms to solve  $\Phi$ -way testing or t-way testing problems.
4. Local experimentation, and experimentation with Compute Canada to produce our distributed results.

# **Combinatorial tests generation**

# Taxonomy of methods





# IPOG Algorithm

```
Algorithm IPOG-Test (int  $t$ , ParameterSet  $ps$ )
{
1. initialize test set  $ts$  to be an empty set
2. denote the parameters in  $ps$ , in an arbitrary order, as  $P_1, P_2, \dots$ , and  $P_n$ 
3. add into test set  $ts$  a test for each combination of values of the first  $t$  parameters
4. for (int  $i = t + 1; i \leq n; i ++$ ) {
5.   let  $\pi$  be the set of  $t$ -way combinations of values involving parameter  $P_i$ 
      and  $t - 1$  parameters among the first  $i - 1$  parameters
6.   // horizontal extension for parameter  $P_i$ 
7.   for (each test  $\tau = (v_1, v_2, \dots, v_{i-1})$  in test set  $ts$ ) {
8.     choose a value  $v_i$  of  $P_i$  and replace  $\tau$  with  $\tau' = (v_1, v_2, \dots, v_{i-1}, v_i)$  so that  $\tau'$  covers the
      most number of combinations of values in  $\pi$ 
9.     remove from  $\pi$  the combinations of values covered by  $\tau'$ 
10.  }
11.  // vertical extension for parameter  $P_i$ 
12.  for (each combination  $\sigma$  in set  $\pi$ ) {
13.    if (there exists a test that already covers  $\sigma$ ) {
14.      remove  $\sigma$  from  $\pi$ 
15.    } else {
16.      change an existing test, if possible, or otherwise add a new test
      to cover  $\sigma$  and remove it from  $\pi$ 
17.    }
18.  }
19.}
20.return  $ts$ ;
}
```

Source:

Lei, Yu, et al. "IPOG: A general strategy for  $t$ -way software testing." *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE, 2007.

# IPOG Algorithm

P1	P2	P3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

(a)

P1	P2	P3	P4
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	0
1	0	0	1
1	0	1	2
1	1	0	0
1	1	1	1

(b)

P1	P2	P3	P4
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	0
1	0	0	1
1	0	1	2
1	1	0	0
1	1	1	1
1	0	1	0
0	1	0	1
0	0	1	2
1	1	0	2
-	0	0	2
-	1	1	2

Source:

Lei, Yu, et al. "IPOG: A general strategy for t-way software testing." *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'07)*. IEEE, 2007.

# Apache Spark

# Apache Spark

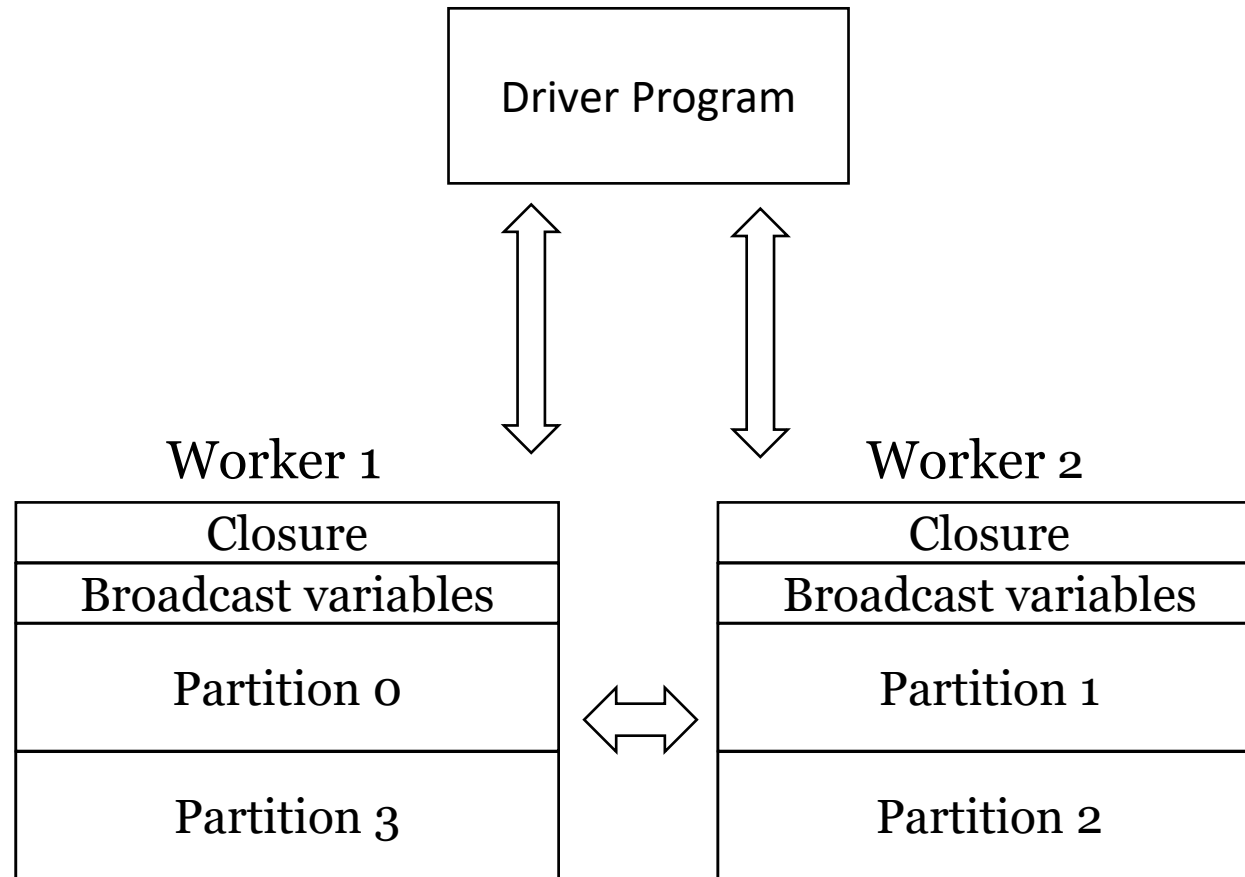
Apache Spark is a framework for programming clusters (a cluster of servers). It is mostly used for Big Data processing.

When programming with Spark, we describe transformations to be done on a collection of data that is partitioned on several computers.

Data partitions exist on computers that are *workers*. These computers communicate with each other with MapReduce type aggregations. The *pilot program* takes care of orchestrating all the computation and giving work to the workers.

# Organisation of the cluster

Extended Combinatorial Testing



# Vocabulary

**RDD:** Resilient Distributed Dataset. The RDD is a collection of elements or objects that is distributed over the cluster. A RDD is divided into partitions.

**Pattern:** a transformation performed on a RDD. The main transformations used are **flatMap** et **reduceByKey**.

**Driver:** The program that orchestrates the distributed computing.

**Worker:** The worker is a process that performs the work required by the RDD transformations (flatMap, reduceByKey etc).

**Contribution I:  
Generalization of t-way  
testing to  $\Phi$ -way testing**

# A generalization of t-way testing

- Our generalization called  $\Phi$ -way (Phi-way) testing replaces the notion of interaction strength ( $t$ ) on a system with an arbitrary set of boolean conditions on the parameter-values. We call this set  $\Phi$ . These boolean conditions can have operators like  $! < > =$
- Example: A system with three parameters  $a, b, c$ ; each with two values 0 or 1

<u>Strength</u>	<u><math>\Phi</math>-way formulas</u>
$t = 2$	$a = 0 \wedge b = 0, a = 0 \wedge b = 1, a = 1 \wedge b = 0, a = 1 \wedge b = 1, a = 0 \wedge c = 0, a = 0 \wedge c = 1, a = 1 \wedge c = 0, a = 1 \wedge c = 1, b = 0 \wedge c = 0, b = 0 \wedge c = 1, b = 1 \wedge c = 0, b = 1 \wedge c = 1$



# Universal and existential constraints

A **universal constraint** is a constraint that applies to all tests in the test suite.

An **existential constraint** is a constraint that applies only to a single test.

# Universal constraints

Most existing tools (ACTS, PICT, Jenny) support universal constraints.  
When finalizing a test, we check if it respects all the universal constraints.

```
Type:          Single, Spanned, Striped, Mirror, RAID-5
Size:          10, 100, 1000, 10000, 40000
Format method: Quick, Slow
File system:   FAT, FAT32, NTFS
Cluster size:  512, 1024, 2048, 4096, 8192, 16384
Compression:  On, Off

#Constraints

IF [File system] = "FAT" THEN [Size] <= 4096;
IF [File system] = "FAT32" THEN [Size] <= 32000;

IF [File system] <> "NTFS" OR
  ( [File system] = "NTFS" AND [Cluster size] > 4096 )
THEN [Compression] = "Off";

IF NOT ( [File system] = "NTFS" OR
  ( [File system] = "NTFS" AND NOT [Cluster size] <= 4096 ) )
THEN [Compression] = "Off";
```

# Limits of universal constraints

Universal constraints are not able to handle every scenario. For example, one cannot write a universal constraint that requires parameters of the same clause to have different values.

However, in  $\Phi$ -way testing, we can express this kind of situation without problems:

$$\{a = 0 \wedge b = 1, a = 0 \wedge b = 1, a = 0 \wedge c = 1, a = 1 \wedge c = 0, b = 0 \wedge c = 1, b = 1 \wedge c = 0\}$$

# Existential constraints

Most of the tools mentioned earlier have basic support for existential constraints. They support a "seeding" mode, a limited form of existential constraints that allows to extend an existing test suite.

However, these tools cannot express a situation as equivalence classes. Universal constraints do not work either.

## Example:

$a = \{0, \dots, 9\}$  et  $c = \{0, 1\}$

We know that when  $a < 5$  and  $b = 0$ , the software will have the same behavior.

☛ So we can write two clauses:

$$a < 5 \wedge b = 0 \wedge c = 0$$

$$a < 5 \wedge b = 0 \wedge c = 1$$

# **Contribution II: Reductions to Graph Coloring and Hypergraph Vertex Covering**

# Reduction to Graph Coloring

The reduction in graph coloring allows, once the graph is colored, to obtain the test suite.

If we find the chromatic number of the graph, we can find the optimal test suite. However, finding the chromatic number is NP-hard.

The reduction of  $\Phi$ -way clauses in graph coloring supports existential constraints (but not universal constraints).

# Informally,

## Graph Construction Phase

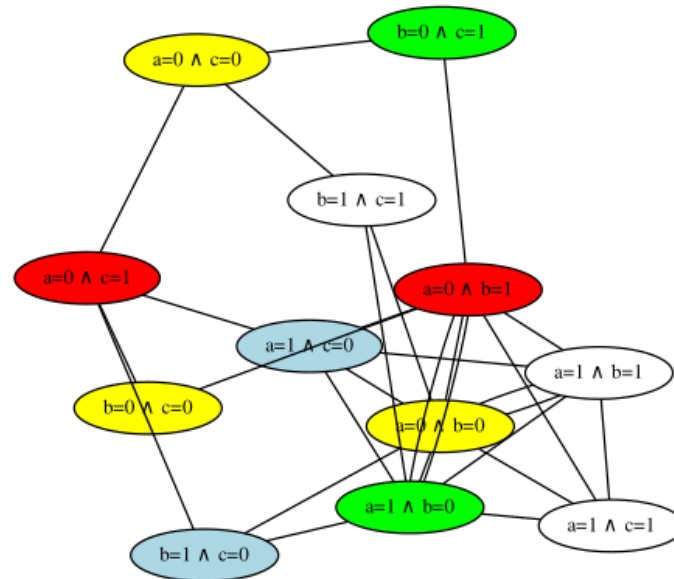
- Each clause becomes a graph vertex
- We put an edge between two clauses that cannot be true at the same time

## Graph Coloring Phase

- We color the graph with any graph coloring algorithm
- We create a test from each color; the test becomes the conjunction of the clauses of the same color

# Example

Strength	$\Phi$ -way formulas
$t = 2$	$a = 0 \wedge b = 0, a = 0 \wedge b = 1, a = 1 \wedge b = 0, a = 1 \wedge b = 1, a = 0 \wedge c = 0, a = 0 \wedge c = 1, a = 1 \wedge c = 0, a = 1 \wedge c = 1, b = 0 \wedge c = 0, b = 0 \wedge c = 1, b = 1 \wedge c = 0, b = 1 \wedge c = 1$





# Reduction to Hypergraph Vertex Covering

The reduction to Hypergraph Vertex Covering allows the use of a highly efficient greedy algorithm (Feige, 1998)

This reduction supports universal and existential constraints.

# Informally,

## Hypergraph Construction Phase

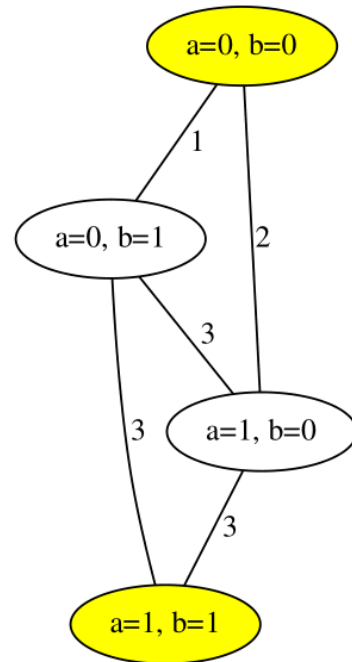
- Each clause becomes an edge
- An edge visits all the vertices with which a conjunction is possible
- We eliminate the vertices which do not satisfy the universal constraints

## Hypergraph Covering Phase

- We apply the greedy algorithm. We find the set of vertices that cover all the hyperedges
- Each vertex becomes a test

# Example

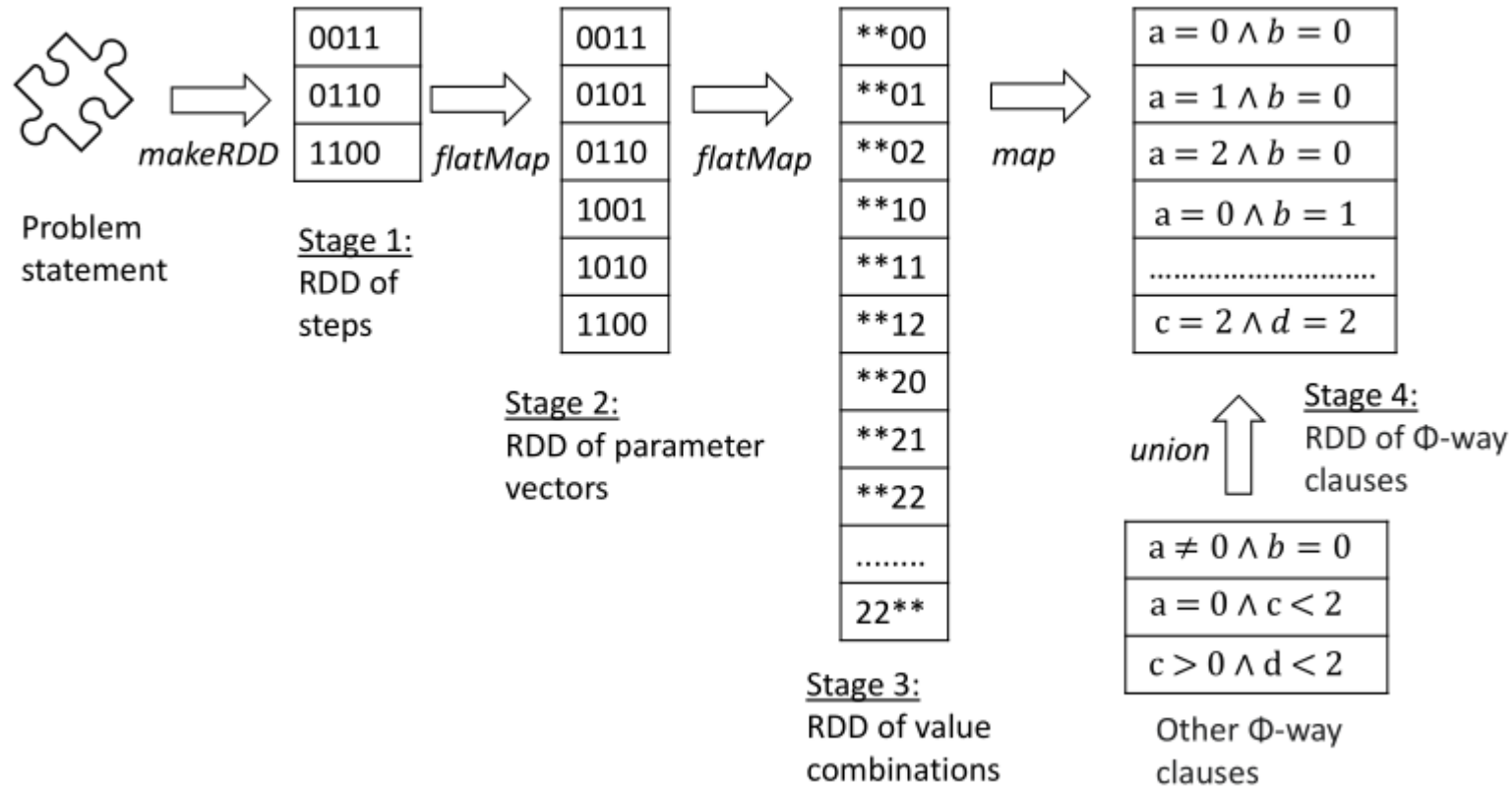
$$\Phi = \left\{ \overbrace{a = 0, b = 0}^{\text{Arête 1}}, \overbrace{a \neq 0 \vee b \neq 0}^{\text{Arête 2}}, \overbrace{a = 1, b = 1}^{\text{Arête 3}} \right\}$$



The chosen nodes become the test suite

# **Contribution III: Distributed Algorithms**

# Generation of $\Phi$ -way clauses + union



# Distributed Graph Coloring

With Apache Spark

# Building the graph

ID	Clause
0	$b=1 \wedge c=0$
2	$a=1 \wedge c=0$
4	$a=1 \wedge c=1$
6	$b=0 \wedge c=0$
8	$b=1 \wedge c=1$
9	$b=0 \wedge c=1$
10	$a=0 \wedge c=0$
11	$a=0 \wedge c=1$

**Partition 0**

ID	Clause
1	$a=1 \wedge b=1$
3	$a=0 \wedge b=0$
5	$a=0 \wedge b=1$
7	$a=1 \wedge b=0$

**Partition 1**

**mapPartitions**

**flatMap +  
collect**

ID	Clause
1	a=1∧b=1
2	a=1∧c=0
3	a=0∧b=0
4	a=1∧c=1
5	a=0∧b=1
6	b=0∧c=0

**Local  
Array**

**Broadcast  
chunk to  
cluster**

ID	Adjlist
2	00
5	00101
4	1010
1	0
3	101
6	100010

**Partition 0**

ID	Adjlist
2	00
5	01010
4	0001
3	010
6	010001

**Partition 1**





**reduceByKey**

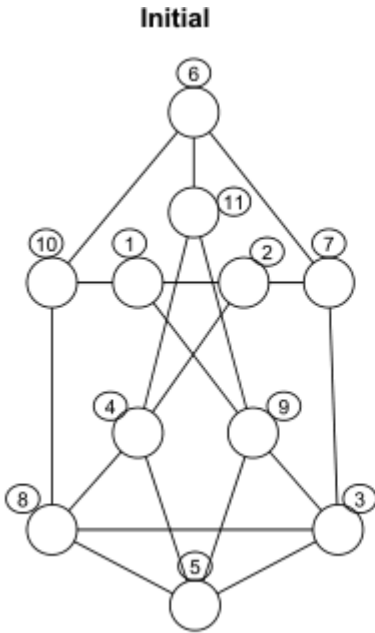
ID	Adjlist
4	1011
6	110011
2	00

**Partition 0**

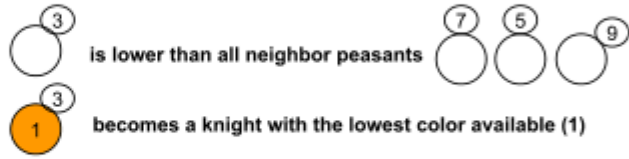
ID	Adjlist
1	0
3	111
5	01111

**Partition 1**

# Knights & Peasants algorithm

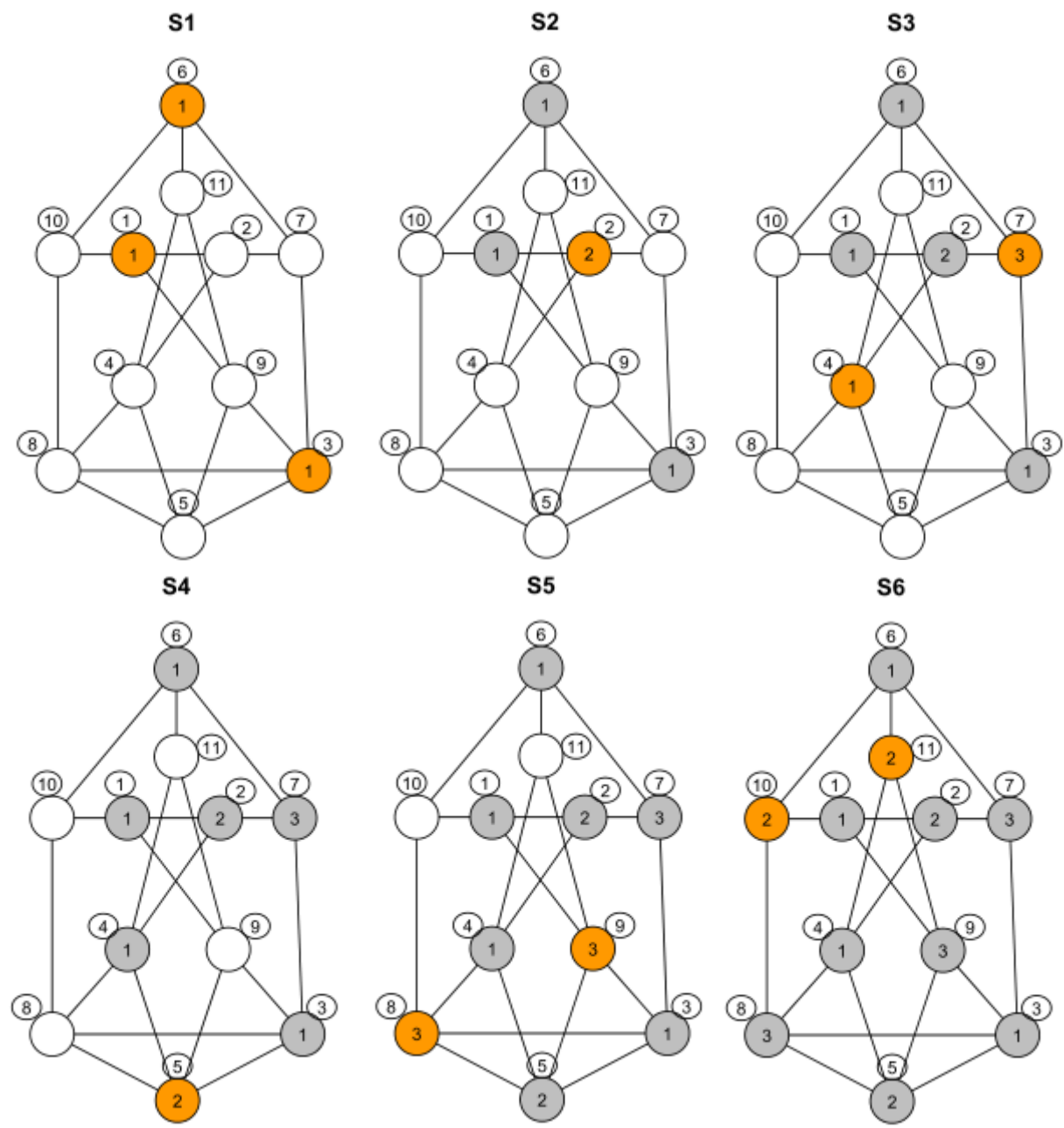


Peasant tiebreaker war example



## Legend

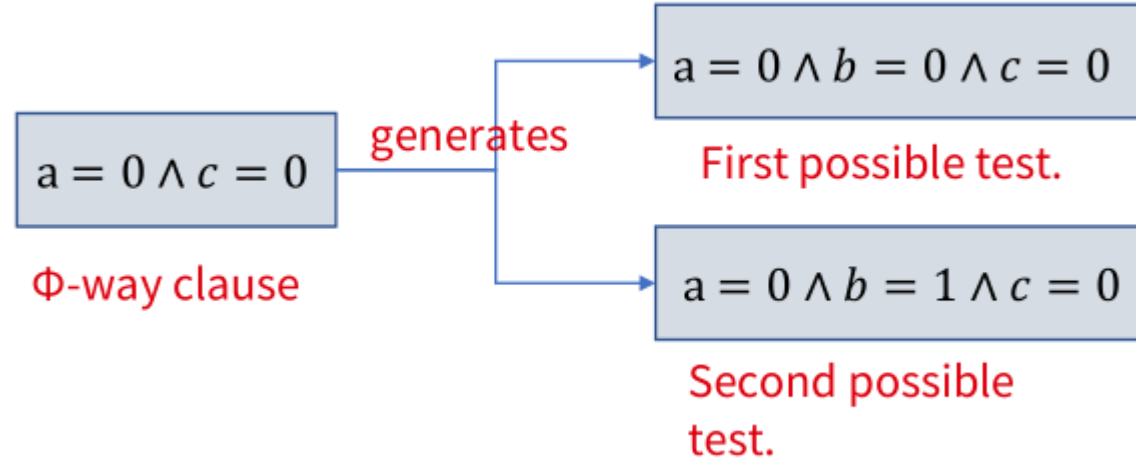
- Tie breaker
- Peasant node
- Knight node with color = 2
- New knight



# Distributed Hypergraph Vertex Covering

With Apache Spark

# An overview



# The Greedy Picker heuristic

---

**Algorithm 13:** Greedy test picker.

---

```
input  : list, a list of the best tests
output : chosenTests, a list of chosen tests

1 Let sizeOfTests be the size of a test, in length of characters. Let differenceScore be a function that compares two tests and
  returns a score of difference
2 if list contains only one test then
3   | return list(0)
4 end
5 chosenTests+ = list(0)
6 for every element i of the list, starting from the second element do
7   | thisTest ← list(i)
8   | for every element j of the chosenTests do
9     | oneofthechosen ← chosenTests(j)
10    | difference ← differenceScore(thisTest, oneofthechosen)
11    | diff ← difference/sizeOfTests * 100
12    | if diff < 40.0 then
13      | found ← false
14      | return
15      | end
16    | end
17    | if found = true then
18      | chosenTests+ = thisTest
19      | end
20 end
21 return chosenTests
```

---

Greedy picker tries to choose tests that are all different from each other. To do this, we calculate a percentage of similarity between each test.

We select a new test among the candidates only if this test is 60% different from all the others.

# Example

**p0**

```
b=0∧c=0  
b=0∧c=1  
b=1∧c=0  
b=1∧c=1  
a=0∧c=0  
a=0∧c=1  
a=1∧c=0  
a=1∧c=1
```

**p1**

```
a=0∧b=0  
a=0∧b=1  
a=1∧b=0  
a=1∧b=1
```

## **Initial state.**

RDD of  $\phi$ -way clauses, two partitions. Every clause will become a hyperedge.



**mapPartitions**

**p0**

```
a=1∧b=1∧c=1 → 2,  
a=0∧b=1∧c=1 → 2,  
a=1∧b=1∧c=0 → 2,  
a=1∧b=0∧c=1 → 2,  
a=0∧b=1∧c=0 → 2,  
a=0∧b=0∧c=1 → 2,  
a=1∧b=0∧c=0 → 2,  
a=0∧b=0∧c=0 → 2,
```

**p1**

```
a=1∧b=1∧c=1 → 1,  
a=1∧b=1∧c=0 → 1,  
a=0∧b=1∧c=1 → 1,  
a=1∧b=0∧c=1 → 1,  
a=0∧b=1∧c=0 → 1,  
a=1∧b=0∧c=0 → 1,  
a=0∧b=0∧c=1 → 1,  
a=0∧b=0∧c=0 → 1,
```

## **Step 1.**

Initial aggregation by building one hash table per partition. Pictured are the final hash tables.

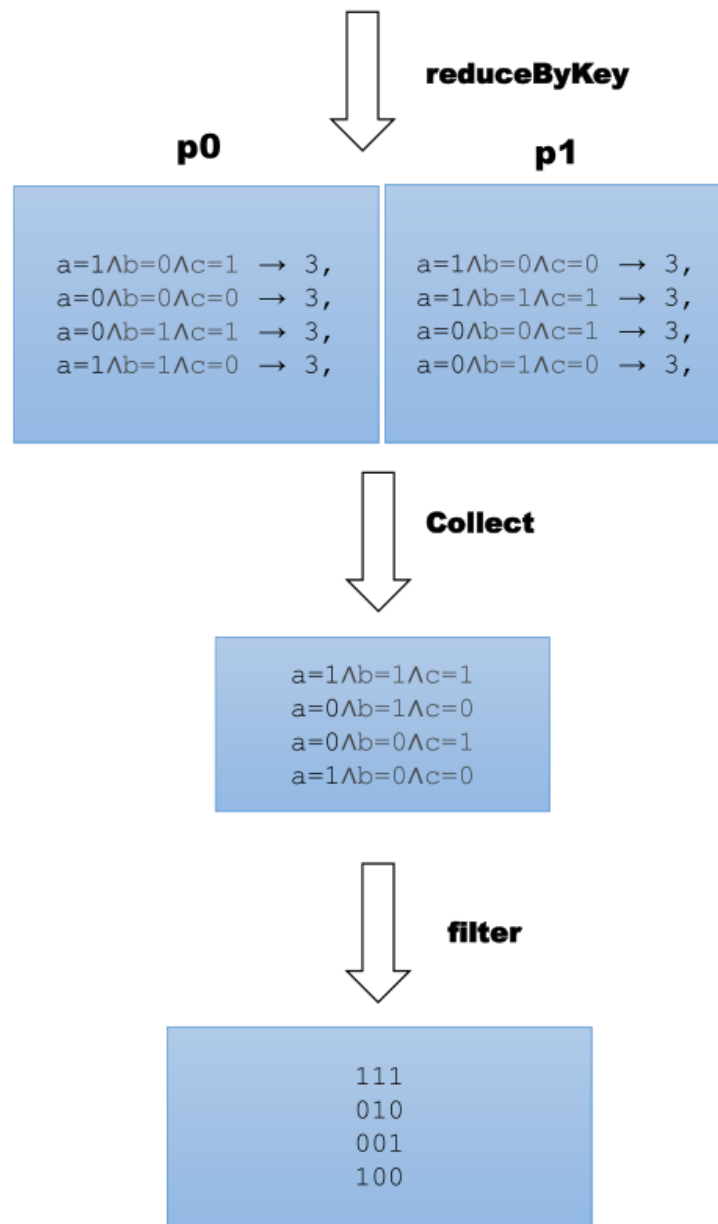
## **Note:**

We iterate over all hyperedges, we generate their covered vertices on the fly (their tests), and we add them to the hash table of their partition.

## **Note 2:**

This is also the moment where we can check the tests against **universal constraints**. If a test fails the constraint, it is not added to the hash table.

# Example



## **Step 2.**

Final aggregation of the best tests using the `reduceByKey` pattern.

## **Step 3.**

We collect the results to the driver program. Then, we use the algorithm called **greedy picker** to pick a diverse subset of the best tests. Here, the greedy picker algorithm selects all four tests.

## **Step 4.**

Delete hyperedges using the tests that were picked. Here, the four picked tests are enough to cover every hyperedge. The algorithm does not go into a second iteration because the RDD is now empty.

# Distributed In- Parameter-Order



# Complete process

$c = 0 \wedge a = 0$
$c = 0 \wedge a = 1$
$c = 1 \wedge a = 0$
$c = 1 \wedge a = 1$
$c = 0 \wedge b = 0$
$c = 0 \wedge b = 1$
$c = 1 \wedge b = 0$
$c = 1 \wedge b = 1$

Clauses that need to be covered for this parameter

$a = 0 \wedge b = 0$
$a = 0 \wedge b = 1$
$a = 1 \wedge b = 0$
$a = 1 \wedge b = 1$

Current test suite.



Four iterations of horizontal growth

$a = 0 \wedge b = 0 \wedge c = 0$
$a = 0 \wedge b = 1 \wedge c = 1$
$a = 1 \wedge b = 0 \wedge c = 1$
$a = 1 \wedge b = 1 \wedge c = 0$

Extended test suite


All clauses are covered. The vertical growth algorithm is not needed.

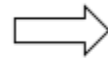
# Complete process

$d = 0 \wedge a = 0$
$d = 0 \wedge a = 1$
$d = 1 \wedge a = 0$
$d = 1 \wedge a = 1$
$d = 0 \wedge b = 0$
$d = 0 \wedge b = 1$
$d = 1 \wedge b = 0$
$d = 1 \wedge b = 1$
$d = 0 \wedge c = 0$
$d = 0 \wedge c = 1$
$d = 1 \wedge c = 0$
$d = 1 \wedge c = 1$

Clauses that need to be covered for this parameter

$a = 0 \wedge b = 0 \wedge c = 0$
$a = 0 \wedge b = 1 \wedge c = 1$
$a = 1 \wedge b = 0 \wedge c = 1$
$a = 1 \wedge b = 1 \wedge c = 0$

Current test suite.



Horizontal growth

$a = 0 \wedge b = 0 \wedge c = 0 \wedge d = 0$
$a = 0 \wedge b = 1 \wedge c = 1 \wedge d = 1$
$a = 1 \wedge b = 0 \wedge c = 1 \wedge d = 0$
$a = 1 \wedge b = 1 \wedge c = 0 \wedge d = 1$

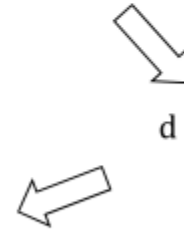
Extended test suite

Remaining clauses

$d = 0 \wedge a = 1$
$d = 1 \wedge a = 0$

Add incomplete tests.

Vertical Growth algorithm (Graph Coloring)



Graph Coloring reduction. 2 colors = 2 tests.

$a = 0 \wedge b = 0 \wedge c = 0 \wedge d = 0$
$a = 0 \wedge b = 1 \wedge c = 1 \wedge d = 1$
$a = 1 \wedge b = 0 \wedge c = 1 \wedge d = 0$
$a = 1 \wedge b = 1 \wedge c = 0 \wedge d = 1$
$d = 0 \wedge a = 1$
$d = 1 \wedge a = 0$

Final test suite

# **Contribution IV: Experimental Evaluation**

# The TSPARK tool

☰ README.md



By Edmond La Chance and Sylvain Hallé, Laboratoire d'informatique formelle (LIF)

- [Introduction](#)
- [Download](#)
- [Included Algorithms](#)
- [Command line usage](#)
- [SLURM cluster manager script](#)



# TSPARK

## Results

File Type	Files	Lines of Code	Total lines
Text	5	0	5763
Scala	51	17848	26956
Java	2	64	80

Source: <https://line-count.herokuapp.com/mitchi/TSPARK>

```
C:\WINDOWS\system32\cmd.exe

TSPARK - a distributed testing tool

Usage

TSPARK [options] command [command options]

Commands

color [command options] <t> <n> <v> : Single Threaded Graph Coloring
--colorings=NUM : Number of parallel graph colorings to run
-v, --verify : verify the test suite
<t> : interaction strength
<n> : number of parameters
<v> : domain size

dcoloring [command options] <t> <n> <v> : Distributed Graph Coloring
--algorithm=STRING : Which algorithm to use (KP or OC)
--chunksize=NUM : Chunk size, in vertices. Default is 20k
-c, --compressRuns : Activate run compression with Roaring Bitmaps
-v, --verify : verify the test suite
<t> : interaction strength
<n> : number of parameters
<v> : domain size of a parameter

dhgraph [command options] <t> <n> <v> : distributed hypergraph covering
-v, --verify : verify the test suite
--vstep=NUM : Covering speed (optional)
<t> : interaction strength
<n> : number of parameters
<v> : domain size of a parameter

dic [command options] <t> <n> <v> : Distributed Ipog-Coloring
--colorings=NUM : Number of graph colorings to run
--hstep=NUM : Number of parameters of tests to extend in par
-v, --save : Save the test suite to a file
--seeding=STRING : Seeding at param,file
-st, --singlethreaded : use single threaded coloring
-v, --verify : verify the test suite
<t> : interaction strength
<n> : number of parameters
<v> : domain size

dicr [command options] <t> <n> <v> : Distributed IPOG-Coloring using a co
```

# Compute Canada Cluster



```
#!/bin/bash
#SBATCH --account=*****
#SBATCH --time=24:00:00
#SBATCH --nodes=16
#SBATCH --mem=64G
#SBATCH --cpus-per-task=8
#SBATCH --ntasks-per-node=1

module load spark/2.4.4

# Recommended settings for calling Intel MKL routines from multi-threaded applications
# https://software.intel.com/en-us/articles/recommended-settings-for-calling-intel-mkl-
routines-from-multi-threaded-applications
export MKL_NUM_THREADS=1
export SPARK_IDENT_STRING=$SLURM_JOBID
export SPARK_WORKER_DIR=$SLURM_TMPDIR
export SLURM_SPARK_MEM=$(printf "%.0f" $(( ${SLURM_MEM_PER_NODE} * 95 / 100 )))

start-master.sh
sleep 5
MASTER_URL=$(grep -Po '(?=spark://).*' $SPARK_LOG_DIR/spark-{$SPARK_IDENT_STRING}-
org.apache.spark.deploy.master*.out)

NWORKERS=$((SLURM_NTASKS - 1))
#SPARK_NO_DAEMONIZE=1 srun -n {$NWORKERS} -N {$NWORKERS} --label --
output=$SPARK_LOG_DIR/spark-{$j}-workers.out start-slave.sh -m {$SLURM_SPARK_MEM}M -c
{$SLURM_CPUS_PER_TASK} {$MASTER_URL} &
slaves_pid=$!

srun -n 1 -N 1 spark-submit --master {$MASTER_URL} --executor-memory
```

# The Graham Cluster

We used the [Graham](#) cluster for our initial results. On this cluster, we requested 16 machines in partial resource utilization, which gave us 128 cores and 1 terabyte of RAM.



## Node characteristics [\[edit\]](#)

A total of 41,548 cores and 520 GPU devices, spread across 1,185 nodes of different types; note that Turbo Boost is activated for the ensemble of Graham nodes.

nodes ↕	cores ↕	available memory ↕	CPU ↕	storage ↕	GPU ↕
903	32	125G or 128000M	2 x Intel E5-2683 v4 Broadwell @ 2.1GHz	960GB SATA SSD	-
24	32	502G or 514500M	2 x Intel E5-2683 v4 Broadwell @ 2.1GHz	960GB SATA SSD	-
56	32	250G or 256500M	2 x Intel E5-2683 v4 Broadwell @ 2.1GHz	960GB SATA SSD	-
3	64	3022G or 3095000M	4 x Intel E7-4850 v4 Broadwell @ 2.1GHz	960GB SATA SSD	-
160	32	124G or 127518M	2 x Intel E5-2683 v4 Broadwell @ 2.1GHz	1.6TB NVMe SSD	2 x NVIDIA P100 Pascal (12GB HBM2 memory)
7	28	178G or 183105M	2 x Intel Xeon Gold 5120 Skylake @ 2.2GHz	4.0TB NVMe SSD	8 x NVIDIA V100 Volta (16GB HBM2 memory)
2	40	377G or 386048M	2 x Intel Xeon Gold 6248 Cascade Lake @ 2.5GHz	5.0TB NVMe SSD	8 x NVIDIA V100 Volta (32GB HBM2 memory), NVLINK
6	16	192G or 196608M	2 x Intel Xeon Silver 4110 Skylake @ 2.10GHz	11.0TB SATA SSD	4 x NVIDIA T4 Turing (16GB GDDR6 memory)
30	44	192G or 196608M	2 x Intel Xeon Gold 6238 Cascade Lake @ 2.10GHz	5.8TB NVMe SSD	4 x NVIDIA T4 Turing (16GB GDDR6 memory)
72	44	192G or 196608M	2 x Intel Xeon Gold 6238 Cascade Lake @ 2.10GHz	879GB SATA SSD	-

# Problem sizes

Problem	Number of vertices	Size of graph*
N=8	131 072	1g**
N=9	589 824	21g
N=10	1 966 080	241g
N=11	5 406 720	1.8t
N=12	12 976 128	10t
N=13	28 114 944	49t
N=14	56 229 888	197t
N=15	105 431 040	694t
N=16	187 432 960	2.19pb
N=17	318 636 032	6.34pb

\* Using bit arrays and half of the matrix

\*\*The formula:

$$\frac{((\binom{8}{7}) * 4^7)^2}{1000 / 1000 / 1000 / 8 / 2}$$



# Graham results

Instance	PICT	ACTS	Jenny	D-Hypergraph	Order Coloring	D-IPOG-Coloring	D-IPOG-Hypergraph	K&P Coloring
N=8	24391	<b>16384</b>	25659	23255	28593	26152	25249	28811
N=9	35351	39296	36293	<b>33462</b>	42198	38938	35825	Time limit
N=10	45320	51636	46355	<b>41387</b>	56153	50242	45595	Time limit
N=11	55143	58952	55892	<b>49059</b>	Time limit	61079	54904	Time limit
N=12	64555	65882	64844	Time limit	Time limit	71134	<b>63726</b>	Time limit
N=13	73588	<b>72941</b>	Error	Time limit	Time limit	80463	Time limit	Time limit
N=14	Time Limit	<b>81412</b>	Error	Time limit	Time limit	89121	Time limit	Time limit
N=15	Time Limit	<b>88885</b>	Error	Time limit	Time limit	97190	Time limit	Time limit
N=16	Time Limit	<b>95700</b>	Error	Time limit	Time limit	104752	Time limit	Time limit
N=17	Time Limit	<b>102430</b>	Error	Time limit	Time limit	111833	Time limit	Time limit

# Graham results

Instance	PICT	ACTS	Jenny	D-Hypergraph	Order Coloring	D-IPOG-Coloring	D-IPOG-Hypergraph	K&P Coloring
N=8	112s	<b>0s</b>	58s	47s	96s	34s	231s	7590s
N=9	662s	3s	341s	<b>336s</b>	4472s	167s	1501s	Time limit
N=10	2680s	6s	1494s	<b>3322s</b>	68937s	524s	7454s	Time limit
N=11	8560s	10s	5107s	<b>93294s</b>	Time limit	1343s	17135s	Time limit
N=12	23393s	16s	13968s	Time limit	Time limit	3139s	<b>95176s</b>	Time limit
N=13	56586s	<b>33s</b>	Error	Time limit	Time limit	6597s	Time limit	Time limit
N=14	Time Limit	<b>63s</b>	Error	Time limit	Time limit	12796s	Time limit	Time limit
N=15	Time Limit	<b>131s</b>	Error	Time limit	Time limit	22971s	Time limit	Time limit
N=16	Time Limit	<b>280s</b>	Error	Time limit	Time limit	39370s	Time limit	Time limit
N=17	Time Limit	<b>483s</b>	Error	Time limit	Time limit	64578s	Time limit	Time limit

# Conclusion

# Strengths of this approach

$\Phi$ -way testing is a richer system for expressing the conditions of a system to be covered. Its comprehensive support for existential constraints has no equivalent in existing tools. The Hypergraph algorithm obtains excellent solutions all the time, if the problem structure is suitable.

Graph coloring performs well and gives good solutions when the graph is sparse; this is interesting in many t-way testing situations.

All the proposed solutions will become more interesting in the future, with technological advances like:

- A faster network
- Increasing the amount of RAM (e.g. 512GB DDR5)
- Increasing the number of cores in the machines

# Drawbacks of this approach

Distributed iterations with Apache Spark+Cluster are quite slow!

At the moment, the "have or rent a computer cluster" approach is quite expensive.

# Future work

- Implementation of the algorithms described in the thesis on a video card. Newer video cards, such as the RTX 3090 have 10 000 cores and can be programmed using CUDA.
- We can also do multi-GPU on the same machine, for more power. GPU programming allows to have a lot of muscle, with less latency than a cluster.
- Implementation of the algorithms with C++. This would allow to take advantage of SIMD operations in some key places.
- Addition of new algorithms for TSPARK

**Thank you**

# Annex



# Recent developments

# The Niagara Cluster

For our results on Niagara, we used 20 computers, giving us a total of 800 cores, and 4 terabytes of RAM.

## Niagara hardware specifications [\[edit\]](#)

---

- 2024 nodes, each with 40 Intel Skylake or Cascadelake cores at 2.4GHz, for a total of 80,640 cores.
- 202 GB (188 GiB) of RAM per node.
- EDR Infiniband network in a so-called 'Dragonfly+' topology.
- 12.5PB of scratch, 3.5PB of project space (parallel filesystem: IBM Spectrum Scale, formerly known as GPFS).
- 256 TB burst buffer (Excelero + IBM Spectrum Scale).
- No local disks.
- No GPUs.
- Theoretical peak performance ("Rpeak") of 6.25 PF.
- Measured delivered performance ("Rmax") of 3.6 PF.
- 685 kW power consumption.



# The Database Graph Construction algorithm

- A **new algorithm** for building the graph. It is based on the following observation: One can directly construct the set of elements that are non-compatible with a clause by constructing for each parameter, the set of non-valids. The final set of non-valids, the edges, is the union of all the sub-sets of non-valids.
- This algorithm builds the graph much faster by doing set manipulations with optimized data structures (Bitset or RoaringBitmap)
- Faster state update in D-IPOG and D-Hypergraph
- Very few branches in the algorithm. Bitwise operations without branching.
- More optimal for Spark: Replaces MapReduce with a simple flatMap transformation

# Hypergraph results

Instance	Hypergraph-Graham	Hypergraph-Niagara	Speedup
N=10	3322s	485s	6.84x
N=11	93294s	5540s	16.84x

# K&P results on sparse graphs

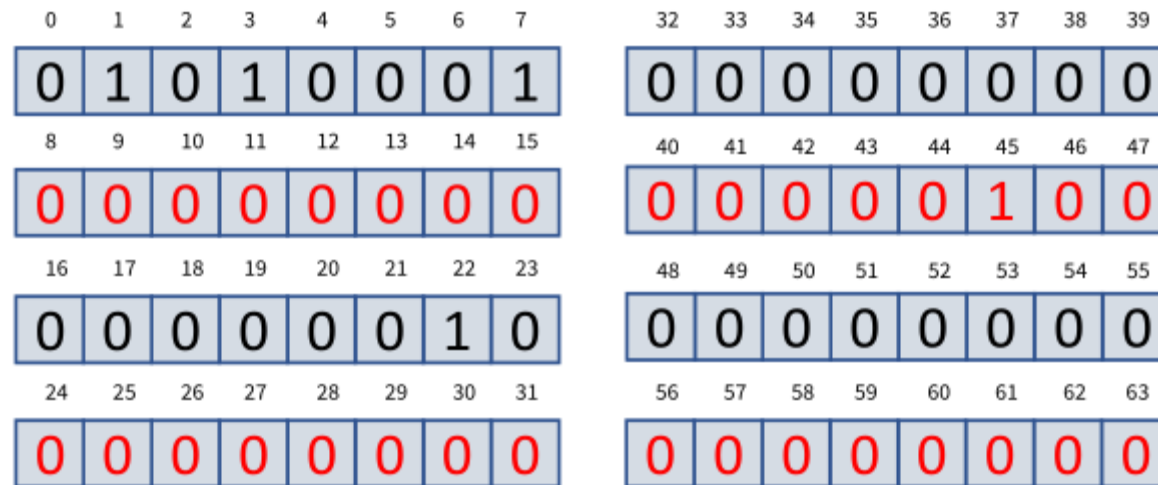
Instance	Number of vertices	Size of graph*	Number of tests found	Time	% of iterations**	Size&Time for ACTS	Difference %***
N=100	19 800	0.02g	<b>14</b>	58s	3.681%	16 & 0s	13%
N=200	79 600	0.39g	<b>15</b>	116s	1.90%	18 & 0s	17%
N=400	319 200	6.36g	<b>17</b>	253s	0.972%	20 & 0s	15%
N=800	1 278 400	102g	<b>18</b>	747s	0.50%	22 & 1.6s	19%
N=1600	5 116 800	1.636t	<b>19</b>	3358s	0.288%	24 & 5.73s	21%
N=3200	20 473 600	26t	<b>21</b>	23278s	0.15%	26 & 34s	20%

- We use here the Niagara cluster, with the RoaringBitmaps to represent the graph pieces
- t=2, v=2 on all problems
- Size of the chunks = 200 000 vertices
- \* Size computed with the triangle of the matrix + Bitsets
- \*\* The % of iterations is computed by counting the number of iterations distributed, divided by the number of vertices
- \*\*\* Quality difference between ACTS and K&P Coloring

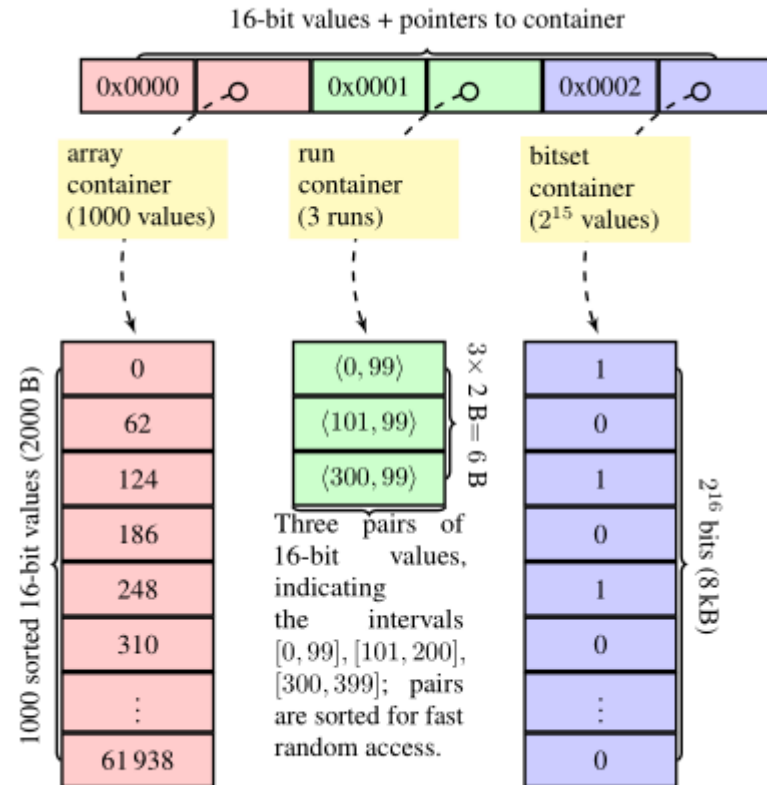
# D-IPOG improvements

Instance	ACTS IPOG	D-IPOG-C	D-IPOG-C new	Speedup
N=8	0s	34s	44s	0.75x
N=9	3s	167s	112s	1.49x
N=10	6s	524s	282s	1.85x
N=11	10s	1343s	614s	2.18x
N=12	16s	3139s	1217s	2.57x
N=13	33s	6597s	1744s	3.78x
N=14	63s	12796s	3489s	3.66x
N=15	131s	22971s	4830s	4.75x
N=16	280s	39370s	9389s	4.19x
N=17	483s	64578s	16646s	3.87x

# Bitset



# RoaringBitmap



Source:

*Lemire, D., Kaser, O., Kurz, N., Deri, L., O'Hara, C., Saint-Jacques, F. and Ssi-Yan-Kai, G. (2018). Roaring bitmaps: Implementation of an optimized software library. Software: Practice and Experience, 48(4), 867–895.*