

# Extended Combinatorial Testing

*With Graph Algorithms and Apache Spark*

**Par**

**Edmond La Chance**

**Directeur: Sylvain Hallé**

**UQAC**

Université du Québec  
à Chicoutimi





# À propos de moi

Je suis né à Montréal, et je vis maintenant à Chicoutimi.

Je suis un amateur de jeux RPG comme D&D, Pathfinder et Call of Cthulhu.

Anciennement un joueur très actif de babyfoot... Je joue moins depuis 2 ans, mais j'espère refaire une compétition un jour!

# Introduction

# Le rapport du CISQ

Le CISQ estime dans son rapport qu'en 2018, le coût total relié aux bugs logiciels est de **1.53 billions** de dollars, soit 7% du PIB des É.U en 2018.

Le CISQ estime, avec leur rapport de 2020, que ce coût à augmenté de 26% jusqu'à **2.08 billions** de dollars.

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

Source: Rapport du CISQ de 2018

Note: 1 billion = 1 trillion en anglais

CISQ = Consortium for Information&Software Quality

# Le cas du Therac-25



# Problématique

Pourquoi fait-on tout ce travail pour trouver des bugs, et rendre nos logiciels plus fiables, et prévisibles?

1. Les bogues et les comportements imprévus peuvent être catastrophiques pour la **sécurité** des gens.
2. Les problèmes logiciels ont un coût très important pour **l'économie**. Si on économise à ce niveau là, on peut investir de l'argent ailleurs.

**Comment peut-on  
prévenir ces problèmes?**

# Techniques de vérification

1. Inspection manuelle du code
2. Analyse Statique, Analyse Runtime
  - Code Smells
  - Model Checking
  - Runtime Monitoring
3. Génération de tests
  - Black-box Fuzzing
  - Concolic Testing
  - **Tests combinatoires**







```
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.



# Tests combinatoires



	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	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	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	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

- Une suite de tests  $t=2$  est une suite dans laquelle chaque paire paramètre-valeur apparait au moins une fois.
- On peut appeler ces combinaisons de valeurs des “combos”.
- Des études empiriques par Kuhn et al. ont montré que les tests t-way sont très efficaces que les tests exhaustifs pour trouver les bogues.

```
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](#)

# Études empiriques sur le t-way testing

Étude de Wallace et Kuhn (2001) sur 15 ans de rapports de bugs provenant d'applications médicales embarquées.

t=2 trouve 97% des problèmes, t=3 trouve 99% et t=4 100%

Étude de Kuhn et Reilly (2002) sur les rapports de bogues de Mozilla Web Browser et Apache Web Server.

t=2 trouve 76% des bugs, t=3 trouve 95% des bugs, ensuite t=4, t=5 et t=6 augmentent encore la performance, jusqu'à 100% des bugs trouvés avec t=6

Étude de Kuhn et al. (2004) sur des logiciels utilisés par la NASA.

t=2 et t=3 trouvent la majorité des problèmes, et t=4 jusqu'à t=6 semblent assurer une couverture pseudoexhaustive.

# Constat

# Constat initial

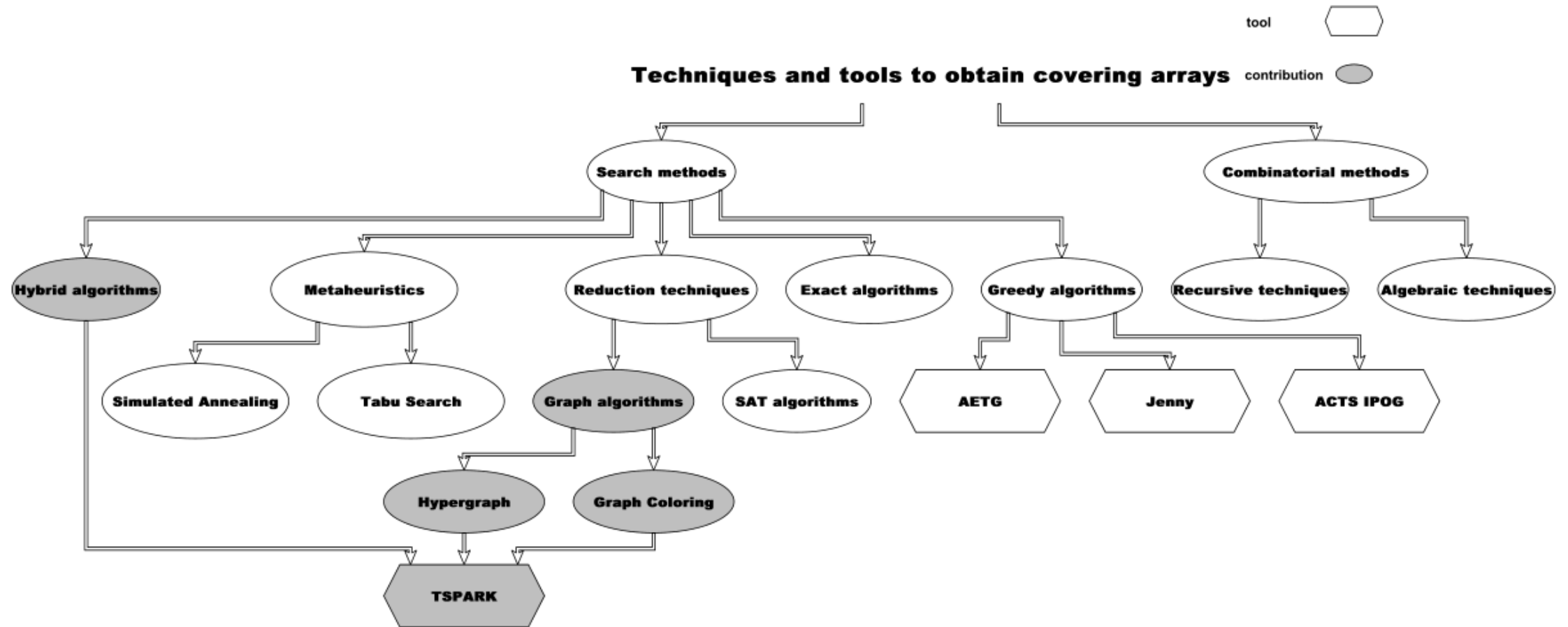
1. Des outils pour générer des tests combinatoires existent déjà (PICT, ACTS, Jenny etc). Cependant, ces outils n'ont pas forcément toutes les fonctionnalités désirées. Par exemple, le support pour les **contraintes existentielles** est très basique.
2. Les outils existants ont un design mono-thread, ce qui peut entraîner des problèmes de **scalabilité**. Aucun outil existant fonctionne en calcul distribué, c'est à dire que le problème est découpé pour être exécuté par un cluster de  $n$  machines.
3. Il manque une étude de grande envergure pour comparer ces solutions, en terme de qualité, de temps, et comment celles-ci supportent les contraintes.

# Contribution de la thèse

1. Généralisation du t-way testing en  $\Phi$ -way testing.
2. Génération des tests en utilisant une réduction en coloriage de graphes, et couverture par ensembles.
3. Implémentation et conception d'algorithmes distribués pour résoudre des problèmes de  $\Phi$ -way testing et t-way testing.
4. Expérimentation locale, et expérimentation avec Calcul Canada pour produire nos résultats distribués.

# **Génération des tests combinatoires**

# Taxonomie des méthodes de génération





# Algorithme IPOG

```
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.

# Algorithme IPOG

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

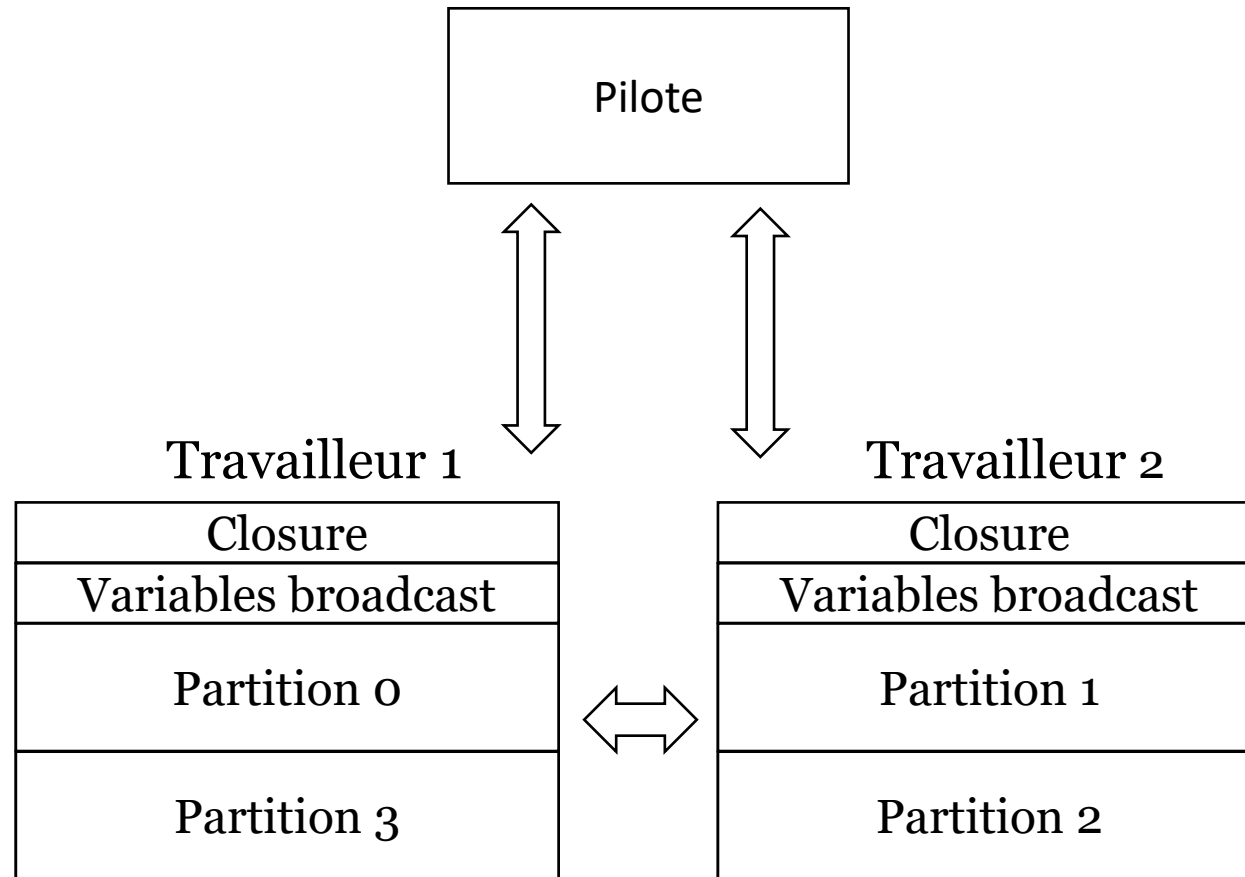
Apache Spark est un framework pour programmer des clusters (une grappe de serveurs). Il est très utile pour le traitement de données Big Data.

Lorsqu'on programme avec Spark, on décrit des transformations à faire sur une collection de données qui est partitionnée sur plusieurs ordinateurs.

Les partitions de données existent sur les ordinateurs qui sont des *travailleurs*. Ces ordinateurs communiquent entre eux avec des agrégations de type MapReduce. Le *programme pilote* lui, s'occupe d'orchestrer tout le calcul et de donner du travail aux travailleurs.

# Organisation du cluster

Extended Combinatorial Testing



# Vocabulaire

**RDD:** Resilient Distributed Dataset. Le RDD est une collection d'éléments ou d'objets qui est distribuée sur le cluster. Un RDD est divisé en partitions.

**Pattern:** une transformation qu'on effectue sur un RDD. Les principales transformations utilisées sont **flatMap** et **reduceByKey**.

**Driver:** Le programme qui orchestre le calcul distribué.

**Worker:** Le worker est un processus qui effectue le travail demandé par les transformations du RDD (flatMap, reduceByKey etc).

# **Contribution I: Généralisation du t-way testing en $\Phi$ -way testing**

# Une généralisation du t-way testing

- Notre généralisation appelée  $\Phi$ -way (Phi-way) testing remplace la notion de force d'interaction ( $t$ ) sur un système par un ensemble arbitraire de conditions booléennes sur les paramètres-valeurs. On appelle cet ensemble  $\Phi$ . Ces conditions booléennes peuvent avoir des opérateurs comme  $! < > =$
- Exemple: Un système avec trois paramètres  $a, b, c$  ; chacun avec deux valeurs 0 ou 1

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



# Contraintes universelles et existentielles

Une contrainte dite **universelle** est une contrainte qui s'applique à **tous** les tests dans la suite de tests.

Une contrainte dite **existentielle** est une contrainte qui s'applique seulement pour un seul test.

# Contraintes universelles

La plupart des outils existants (ACTS, PICT, Jenny) supportent les contraintes **universelles**. Au moment de finaliser un test, on vérifie si celui-ci respecte toutes les contraintes universelles.

```
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";
```

# Limite des contraintes universelles

Les contraintes universelles ne sont pas capables de gérer tous les scénarios. Par exemple, on ne peut pas écrire de contrainte universelle qui demande que les paramètres d'une même clause aient des valeurs différentes.

Par contre, en  $\Phi$ -way testing, on peut exprimer ce genre de situation sans problème:

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

# Contraintes existentielles

La plupart des outils mentionnés auparavant ont un support basique pour les contraintes existentielles. Ils supportent un mode « seeding », une forme limitée de contraintes existentielles qui permet d'étendre une suite de tests existante.

Cependant, ces outils ne peuvent pas exprimer une situation comme des classes d'équivalences. Les contraintes universelles ne fonctionnent pas également.

## Exemple:

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

On sait que quand  $a < 5$  et  $b = 0$ , le logiciel va avoir le même comportement.

▣ On peut donc écrire deux clauses:

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

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

# **Contribution II: Réductions en coloriage de graphe, et couverture par ensemble**

# Réduction en coloriage de graphes

La réduction en coloriage de graphe permet, une fois le graphe colorié, d'obtenir la suite des tests.

Si on trouve le nombre chromatique du graphe, on est également assuré d'avoir **la solution optimale**. Cependant, trouver le nombre chromatique est NP-Difficile.

La réduction des clauses  $\Phi$ -way en coloriage de graphes permet de supporter les **contraintes existentielles** (mais pas les contraintes universelles).

# Informellement,

## Construction du graphe

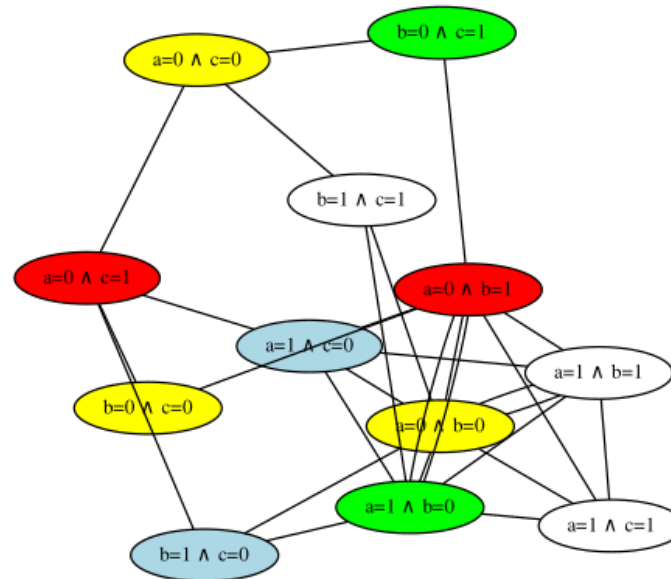
- Chaque clause devient un sommet de graphe
- On met une arête entre deux clauses qui ne peuvent pas être vraies en même temps

## Coloriage du graphe

- On colorie le graphe avec un algorithme
- On crée un test par couleur; le test devient la conjonction des clauses d'une même couleur

# Exemple

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





# Réduction en couverture par ensembles

La réduction en couverture par ensembles permet d'utiliser un algorithme glouton d'une grande efficacité (Feige, 1998)

Cette réduction permet de supporter les **contraintes universelles et existentielles**.

# Informellement,

## Construction de l'hypergraphe

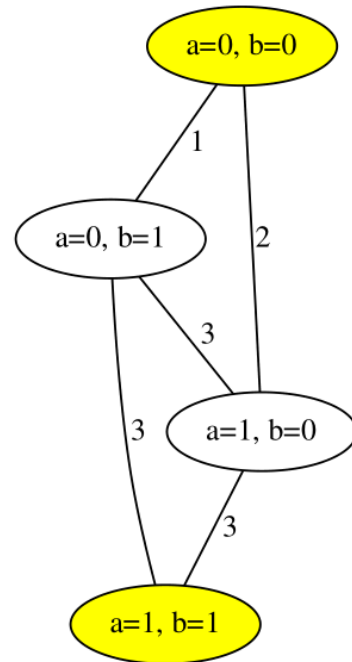
- Chaque clause devient une arête
- Une arête visite tous les sommets avec lesquels une conjonction est possible
- On élimine les sommets qui ne satisfont pas les contraintes universelles

## Couverture par ensembles

- On applique l'algorithme glouton. On trouve l'ensemble des sommets qui couvrent toutes les arêtes
- Chaque sommet devient un test

# Exemple

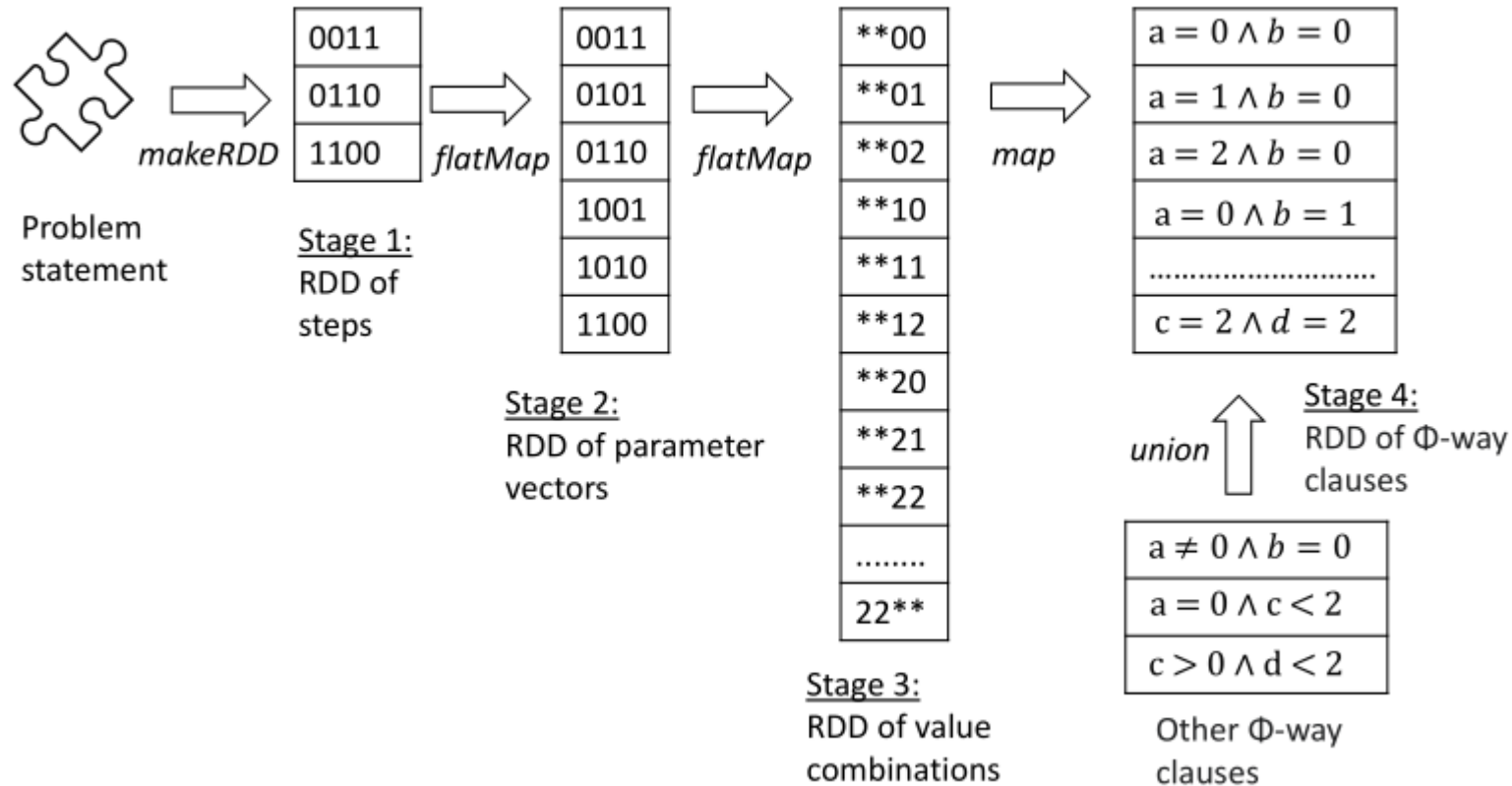
$$\Phi = \left\{ \overbrace{a = 0, b = 0}^{\text{Arête 1}}, \overbrace{a \neq 0 \vee b \neq 0}^{\text{Arête 2}}, \overbrace{a = 1, b = 1}^{\text{Arête 3}} \right\}$$



Les noeuds choisis  
deviennent la suite de tests

# **Contribution III: Algorithmes distribués**

# Génération des clauses $\Phi$ -way + union



# Coloriage de graphes en calcul distribué

*Avec Apache Spark*

# Construction du graphe en distribué

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**

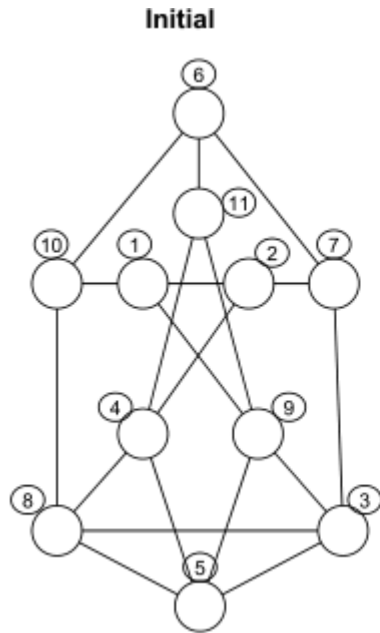
ID	Adjlist
4	1011
6	110011
2	00

**Partition 0**

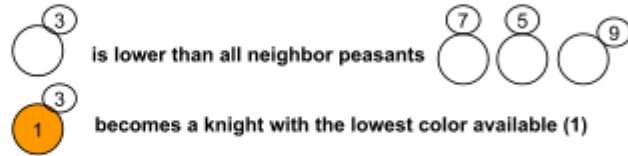
ID	Adjlist
1	0
3	111
5	01111

**Partition 1**

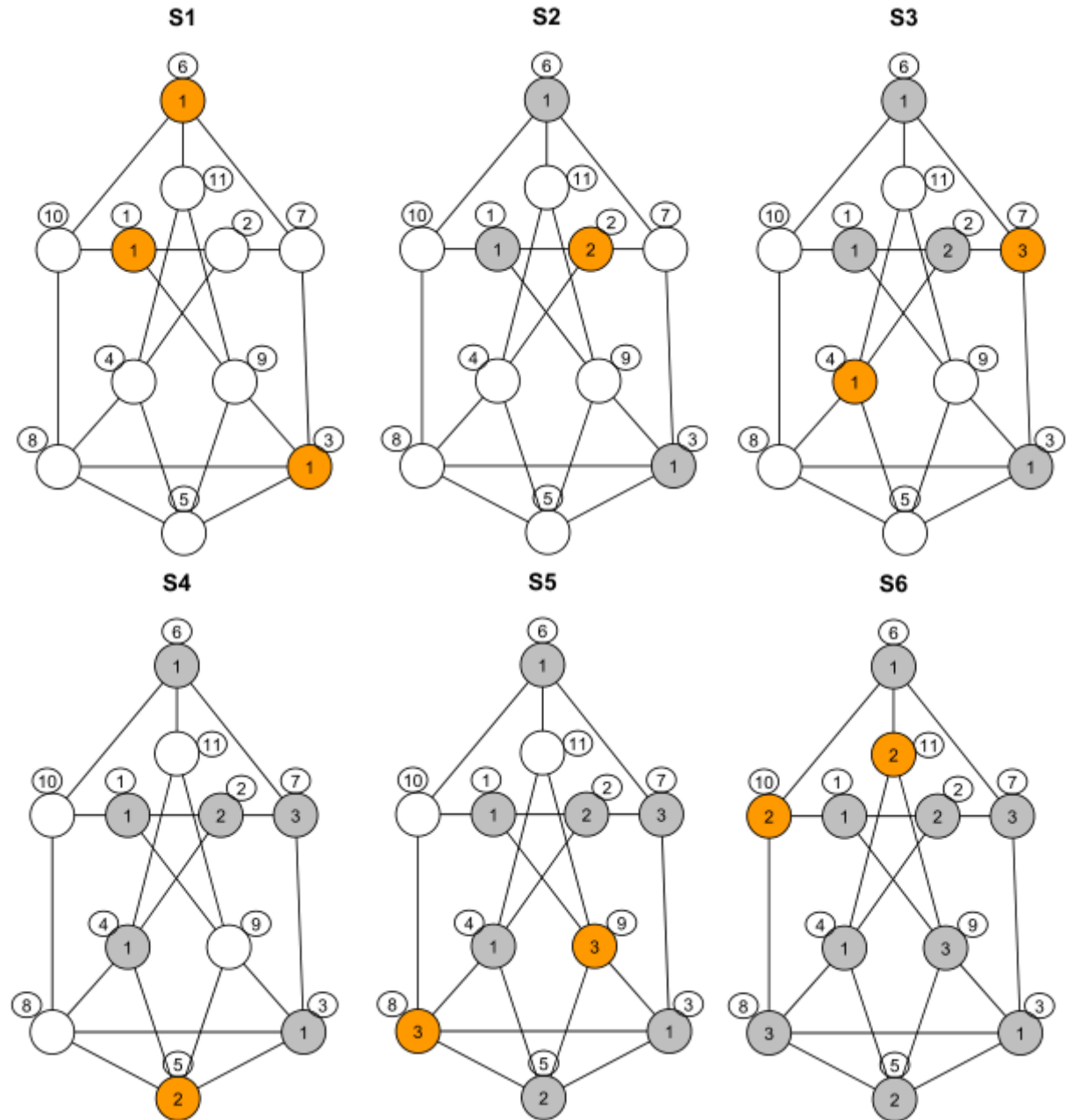
# Algorithme Knights & Peasants



Peasant tiebreaker war example



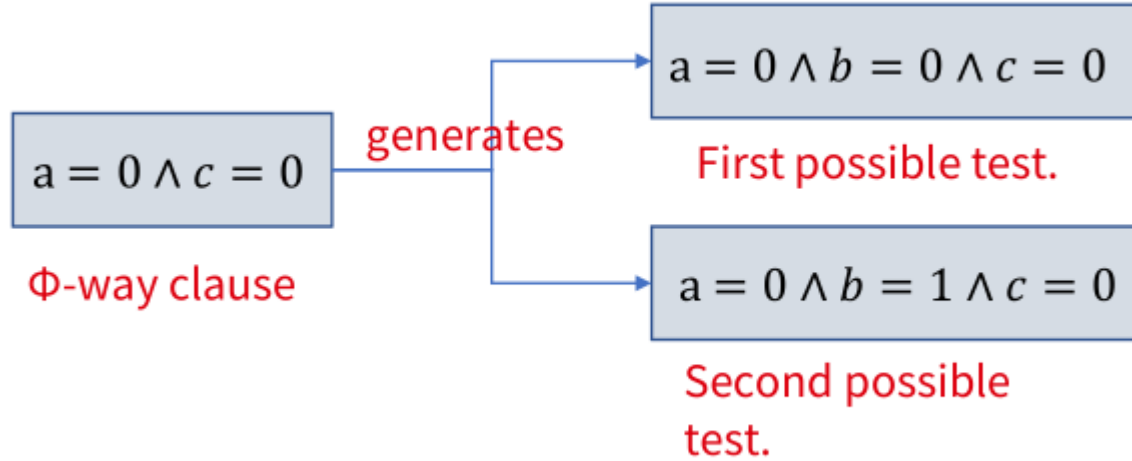
## Legend



# **La couverture par ensembles, en distribué**

*Avec Apache Spark*

# Couverture par ensembles



# Heuristique: Greedy Picker

---

**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 essaie de choisir des tests qui sont tous différents les uns des autres. Pour faire cela, on calcule un pourcentage de similarité entre chaque test.

On sélectionne un nouveau test parmi les candidats seulement si ce test est 60% différent de tous les autres.

# 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.

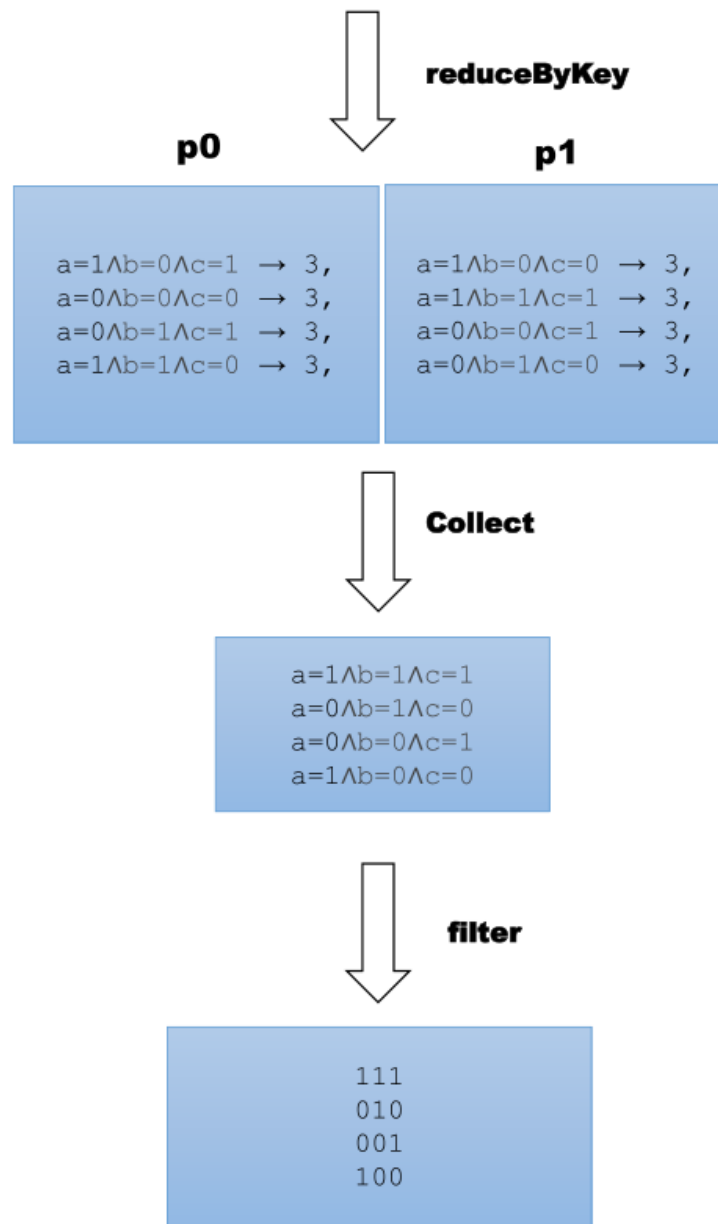
## **Note:**

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.

# **In-Parameter-Order en distribué**



# Processus complet

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

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.

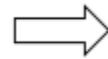
# Processus complet

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

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.



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

Horizontal growth

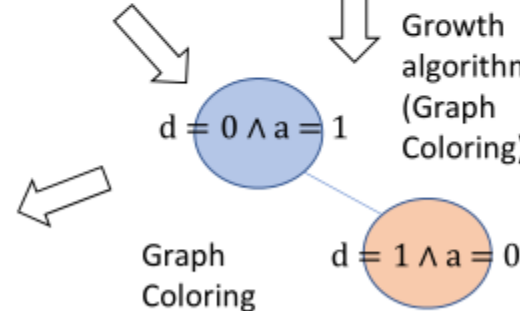
Extended test suite

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: Évaluation Expérimentale**

# L'outil TSPARK

☰ 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
```

# Cluster Calcul Canada



```
#!/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
```

# Le cluster Graham

Nous avons utilisé le cluster [Graham](#) pour nos premiers résultats. Sur ce cluster, nous avons demandé 16 machines en utilisation partielle des ressources, ce qui nous a permis d'avoir 128 cœurs et 1 téraoctets de mémoire vive.



## 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	-

# Taille des problèmes

Instance	Nombre de sommets	Taille du graphe*
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

\* En utilisant une matrice triangle et des tableaux de bits.

\*\*Formule utilisée:

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



# Résultats du Cluster Graham

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

# Résultats du Cluster Graham

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

# Forces de l'approche

Le  $\Phi$ -way testing est un système plus riche pour exprimer les conditions d'un système à couvrir. Son support complet pour les contraintes existentielles n'a pas d'équivalent dans les outils existants.

L'algorithme de couverture par ensembles obtient tout le temps d'excellentes solutions, si la structure du problème s'y prête bien.

Le coloriage de graphe performe bien, et donne des bonnes solutions lorsque le graphe est parsemé; ce qui intéressant dans de nombreuses situations de t-way testing.

Toutes les solutions proposées vont devenir plus intéressantes dans le futur, avec des avancements technologiques comme:

- Un réseau plus rapide
- L'augmentation de la mémoire vive (des barrettes DDR5 de 512go par exemple)
- L'augmentation du nombre de cœurs dans les machines

# Faiblesses de l'approche

Les itérations distribuées avec Apache Spark+Cluster sont assez lentes ☹️

Pour le moment, l'approche "avoir/louer un cluster d'ordinateurs" est assez coûteuse! Ce qui fait en sorte que l'approche cluster n'est pas à la portée de tous.

# Pistes de travaux futurs

Implémentation des algorithmes décrits dans la thèse sur une carte vidéo. On pense notamment à la programmation CUDA sur des cartes vidéos dotées de 10 000 cœurs. On peut également faire du multi-GPU sur la même machine, pour davantage de puissance. La programmation GPU permet d'avoir beaucoup de muscle, avec moins de latence qu'un cluster.

Implémentation des algorithmes avec C++. Ce qui permettrait de profiter des opérations SIMD dans certains endroits clés des algorithmes. Il y a peut-être également moyen de faire plus performant qu'Apache Spark en C++.

Ajout de nouveaux algorithmes pour TSPARK

**Merci**

# **Annexe**



# Développements récents

# Le cluster Niagara

Pour nos résultats sur Niagara, nous avons utilisé 20 ordinateurs, ce qui nous donne un total de 800 cœurs, et 4 téraoctets de mémoire vive.

## 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.



# Algorithme OR-FLIP

- Se base sur l'observation suivante : On peut directement construire l'ensemble des éléments qui sont non-compatibles avec une clause en construisant pour chaque paramètre, l'ensemble des non-valides. L'ensemble final des non-valides, les arêtes, est l'union de tous les sous-ensembles de non-valides.
- Construit le graphe beaucoup plus rapidement en faisant des manipulations d'ensembles avec des structures de données optimisées (Bitset ou RoaringBitmap)
- Mise à jour de l'état plus rapide dans D-IPOG et D-Hypergraph
- Très peu de branchements dans l'algorithme. Opérations bit à bit sans branchement.
- Plus optimal pour Spark: Remplace MapReduce par une simple transformation flatMap

# Résultats Hypergraph

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

# Résultats K&P sur problèmes parsemés

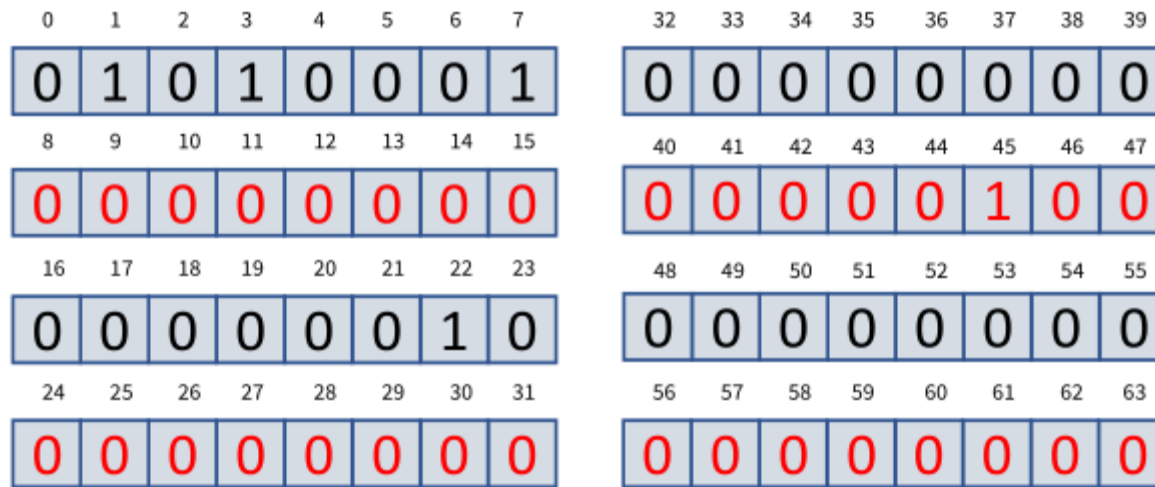
Instance	Nombre de sommets	Taille du graphe *	Nombre de tests trouvés	Temps	% des iterations effectuées**	Taille & Temps ACTS	Différence % ***
N=100	19 800	0.02g	<b>14</b>	58s	3.681%	16 & 0s	13%
N=200	79 600	0.39g	<b>15</b>	116s	1.90%	18 & 0s	17%
N=400	319 200	6.36g	<b>17</b>	253s	0.972%	20 & 0s	15%
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%

- On utilise ici le cluster Niagara, avec les RoaringBitmaps pour représenter les morceaux de graphes
- t=2, v=2 sur tous les problèmes
- Taille des chunks = 200 000 sommets
- \* Taille calculée avec le triangle de la matrice + Bitsets
- \*\* On calcule le % des itérations en comptant le nombre d'itérations distribuées, divisé par le nombre de sommets
- \*\*\* Différence de qualité entre ACTS et K&P Coloring

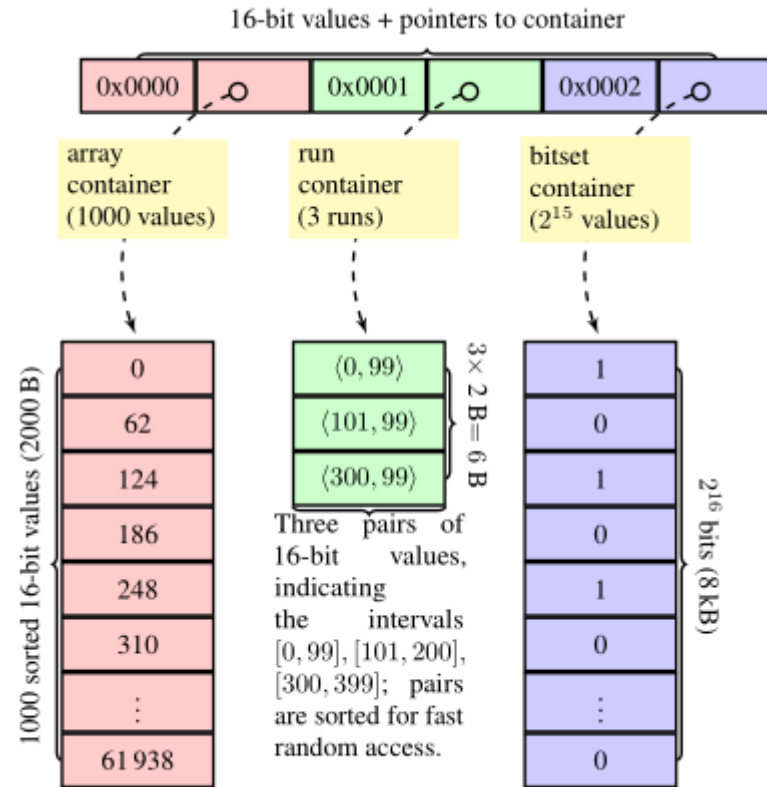
# Amélioration D-IPOG

Instance	ACTS IPOG	D-IPOG-C	D-IPOG-C nouveau	Amélioration D-IPOG
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

# Le 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.*