UNIVERSITÉ LIBRE DE BRUXELLES

BA2-INFO

INFO F-203

Projet Algorithmique 2

Berthion Antoine

Jean Cardinal

2024

Table des matières

1 Introduction **1**

1.1 Introduction 1

1.2 Problématique 1

1.3 Objectifs 2

2 Phase 1 **3**

2.1 Idée Générale 3

2.2 Implémentation 4

2.3 Complexité de l'Algorithme 6

3 Phase 2 **8**

3.1 Idée Générale 8

3.1.1 Discussion de la nature de la classe du problème 8

3.2 Implémentation 9

3.2.1 Optimisations de l'algorithme 10

1 Introduction

1.1 Introduction

En informatique, il est fréquent de rencontrer des problèmes nouveaux qui appartiennent toutefois à des catégories de problèmes déjà étudiées. L'objectif de l'algorithmique est de développer des solutions générales pour des ensembles de problèmes similaires. Dans ce projet, bien que nous soyons confrontés à une situation nouvelle et unique, elle appartient néanmoins à une catégorie de problèmes bien établie et étudiée.

1.2 Problématique

Nous sommes confrontés à une grille où des lampes sont disposées, chacune ayant des modes d'allumage distincts. Chaque lampe est connectée à la grille par **deux entrées**, qui sont à leur tour liées à des interrupteurs pouvant être en position *ouverte* ou *fermée* **(Fig 1.1)**. L'interrogation principale porte, pour une configuration donnée des lampes avec leurs modes d'allumage respectifs, à déterminer **s'il existe** une disposition des interrupteurs permettant d'allumer **toutes** les lampes simultanément.
Dans un autre aspect du problème, nous cherchons à déterminer le nombre *maximal* d'ampoules pouvant être allumées **simultanément**. Cette question revêt une portée plus générale, car elle ne se limite pas à déterminer si toutes les ampoules s'allumeront, mais plutôt à identifier celles qu'il faudrait retirer pour maximiser le nombre d'ampoules activées.

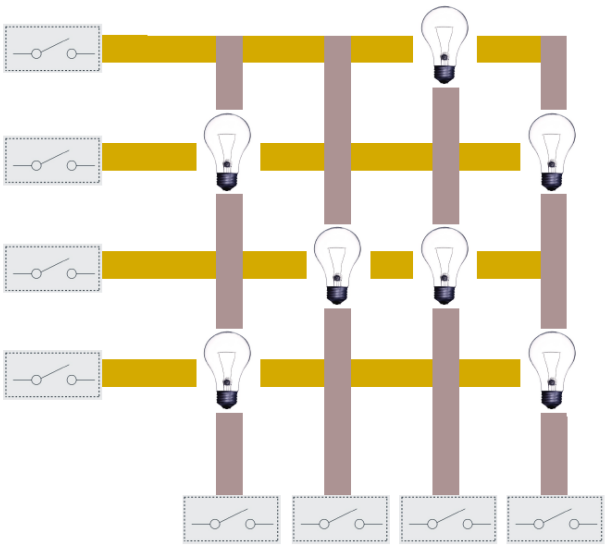

FIGURE 1.1: Grille avec des lampes et des interrupteurs

1.3 Objectifs

Dans la première phase de ce projet, notre objectif est de concevoir un algorithme **polynomial** pour répondre à la question suivante : est-il possible, par une permutation des états des interrupteurs, d'allumer entièrement une grille de lampes donnée ? Ce premier algorithme sera ensuite affiné afin d'améliorer son efficacité dans la borne de complexité à laquelle il appartient.

Concernant la seconde phase, notre objectif est de déterminer l'existence d'un tel algorithme en temps polynomial. Dans le cas ou il ne serait pas possible d'en trouver un, nous chercherons à démontrer qu'**aucune solution de ce type n'existe**. Ensuite, nous prévoyons d'implémenter une solution théorique sous la forme d'un algorithme inefficace (en temps *exponentiel*) dans les cas où il n'est pas possible de trouver une approche efficace.

2 Phase 1

2.1 Idée Générale

Pour aborder cette partie du problème, examinons les modes d'allumage de nos lampes. Considérons chaque possibilité d'allumer une lampe donnée comme une valeur booléenne, vrai ou faux. Ensuite, formons une série de *disjonctions* (**OR**) entre ces conditions. Par exemple, pour la lampe numéro 1, positionnée en **(2, 1)** dans l'exemple **(Fig 1.1)** :

$$L_1 := c_1 \vee c_2 \quad | \quad c_1 = \neg s_2 \wedge \neg s_5 \text{ et } c_2 = s2 \wedge \neg s_5 \tag{2.1}$$

FIGURE 2.1: Expression de L_1

Par conséquent, notre problème peut être exprimé comme une longue expression booléenne, nécessitant la détermination des valeurs booléennes des interrupteurs s_i pour satisfaire cette expression. Cette classe de problèmes est connue sous le nom de **2-SAT**, ou *problème de satisfiabilité à 2 variables*.

Vérifier l'existence de valeurs de s_i pour satisfaire l'expression peut être reformulé en se demandant s'il existe des **contradictions inévitables** dans l'expression booléenne donnée. Si tel est le cas, alors il n'existe pas de permutation des valeurs de s_i permettant de satisfaire l'expression. Concrètement, nous recherchons des séries d'*implications* conduisant à un résultat du type :

$$a \Rightarrow b \Rightarrow c \Rightarrow ... \Rightarrow \neg a$$

Il est évidemment impossible qu'un interrupteur s_i soit dans ses deux états (*ouvert* et *fermé*) simultanément. Cette approche offre une nouvelle perspective sur le problème : si nous identifions une contradiction, alors la grille ne peut pas s'allumer. Sinon, elle le peut.

Pour détecter les contradictions dans un problème 2-SAT, une approche courante consiste à utili-

ser un **graphe d'implication**. L'idée est de représenter chaque interrupteur s_i dans son état fermé et dans son état ouvert par des nœuds dans le graphe. Dans ce graphe **dirigé**, chaque arête représente une **implication** d'un nœud à un autre.

2.2 Implémentation

En premier lieu, nous décidons d'une structure pour une lampe. Le mode correspond à la traduction des 4 bits **x y z t** en un entier représentant la manière dont la lampe peut s'allumer.

```
public class Lamp {
    private int x;  // -> switch sur la ligne
    private int y;  // -> y + nbr_ligne = switch sur la colonne
    private int mode;
}
```

Listing 2.1: Classe Lampe

Pour décrire une relation, nous utilisons une liste de disjonctions (OR) des c_i pour chaque lampe L_i. Chaque condition c_i est une conjonction (AND) de l'état des deux interrupteurs, en commençant par la ligne puis la colonne. Chaque interrupteur est identifié par son numéro, allant de la ligne 1 jusqu'à la colonne m. Un interrupteur ouvert est représenté par l'opposé de son numéro d'interrupteur. Ainsi, la représentation des clauses de la lampe L_i décrite précédemment dans la **Figure 2.1** se présente comme suit : [[-2, -5], [2, -5]].

La première étape de l'algorithme consiste donc à créer ces listes d'implications à partir du mode d'allumage de chacune des lampes. La méthode choisie est la suivante :

```
function createClauses(L)

    clauses = List()
    numberOfLines = max(l.x, l.y for l in L)

    Pour chaque lampe l dans L:
        rowSid = l.x
        columnSid = numberOfLines + l.y
        si l.mode est 0: clauses.add([[]])   // 0000
        si l.mode est 1: clauses.add([[rowSid, columnSid]])   // 0001
        si l.mode est 2: clauses.add([[rowSid, NOT columnSid]]) // 0010
        ... // On fait ceci pour chacun des modes
    return clauses;
```

Listing 2.2: Création des clauses

Chacune des clauses est donc une liste contenant des disjonctions de c_i, selon le format discuté précédemment.

Pour la deuxième étape de l'algorithme, nous passons cette liste de clauses dans une méthode qui permet de les simplifier. Intuitivement, si une clause ne contient qu'un seul c_i, alors elle devient

un prérequis. Ainsi, nous pouvons retirer $\neg c_i$ de toutes les autres clauses.

Enfin, si après simplification nous trouvons une clause vide, cela signifie qu'elle ne peut pas être remplie ⇒ La grille ne peut pas s'allumer. Il est également possible de détecter une contradiction dans l'ensemble des prérequis. De manière similaire, si une telle contradiction est présente, alors la grille ne peut pas s'allumer.

Un exemple de simplification valide d'une expression booléenne peut être trouvé ci-dessous dans la **Figure 2.2**.

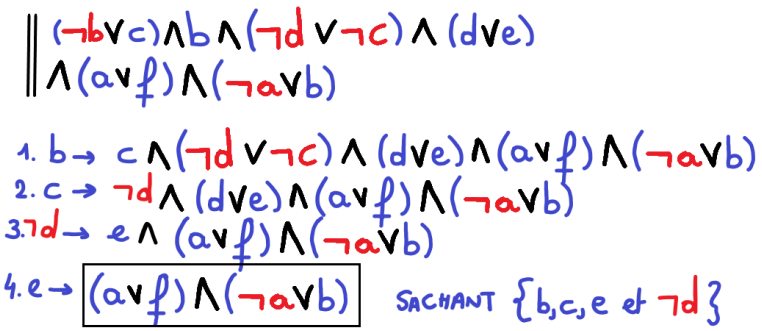

FIGURE 2.2: Simplification d'une expression booléenne

On implémente un tel algorithme en pseudocode de la manière suivante :

```
function SimplifierClauses (clauses)
    boolean possibleSimplification = true;
    boolean contradiction = false;
    prerequisites = {}
    tant que (possibleSimplification) do:
        possibleSimplification = false;
        pour chaque clause dans clauses:
            si clause.size == 0:
                contradiction = True
                break
            si clause.size == 1:
                prerequisites.add(each elements of clause)
                clauses.remove(clause)
            sinon
                pour chaque sous_clause de clause:
                    pour chaque elem de sous_clause
                        si NOT elem in prerequisites:
                            possibleSimplification = true
                            clause.remove(sous_clause)
                            break
    pour chaque elem de prerequisites:
        si NOT elem in prerequisites:
            contradiction = True
            break
    return clauses, contradiction
```

Listing 2.3: Simplification de clauses

Une fois cette étape réalisée, nous disposons d'une expression réduite au maximum et d'un booléen indiquant une contradiction. Nous pouvons utiliser ce booléen pour effectuer un retour anticipé (early return) dans le cas où sa valeur serait True.

La dernière étape de l'algorithme consiste à vérifier si notre expression simplifiée peut être satisfaite. Pour ce faire, nous utilisons le graphe d'implication mentionné précédemment.

```
int nbVar = findMax(clauses);

for (int i = 1; i <= nbVar; i++) {
    graph.put(i, new ArrayList<>());
    graph.put(-i, new ArrayList<>());
}

for (List<Integer> c : clauses) {
    for (int i = 0; i < c.size(); i++) {
        int a = c.get(i);
        int b = c.get((i + 1) % c.size());
        graph.get(a).add(b);
        graph.get(b).add(a);
    }
}
```

Listing 2.4: Déclaration du graphe d'implications

Là où il serait possible d'appliquer un BFS ou un DFS sur chaque composante du graphe pour vérifier les contradictions, nous préférons utiliser la propriété selon laquelle le graphe est dirigé pour trouver toutes ses composantes fortement connexes. Si un nœud et son opposé appartiennent à la même composante connexe, alors il existe un chemin qui les relie ⇒ contradiction, la grille ne peut pas s'allumer. Nous utilisons un algorithme tel que celui de Tarjan pour cela, mais notons qu'un algorithme comme Kosaraju aurait aussi fait l'affaire.

```
List<List<Integer>> scc = tarjan(graph);

for (List<Integer> cc : scc) {
    for (int n : cc) {
        if (cc.contains(n * -1)) {
            return false;
    }
}
}
return true;
```

Listing 2.5: Vérification de la validité de l'expression

2.3 Complexité de l'Algorithme

L'algorithme décrit, il nous reste à déterminer son efficacité, se caractérisant évidemment par sa complexité. Puisque ce problème appartient à la classe de problème 2-SAT, nous devrions obtenir une complexité en temps polynomial, car **2-SAT** ∈ **P**. [Par19]

En premier lieu, intéressons-nous à la création et à la simplification des clauses représentant l'expression booléenne. Pour chacune des lampes, une clause est ajoutée à la liste des clauses. Puisqu'une lampe peut avoir au plus 4 sous-clauses (mode 15, 1111), la taille de la liste résultante ne peut être au maximum que de 4n. Le nombre d'opérations reste également linéaire puisque nous

allons ajouter les clauses pour chacune des n lampes une et une seule fois. Nous avons donc une complexité pour la création de clauses en $\Theta(n)$.

Ensuite, nous appliquons la simplification de clauses. Nous allons tourner dans la boucle principale un nombre de fois qui n'est pas tout de suite trivial étant donné un nombre w de clauses. En effet, les clauses de taille 0 entraînent un arrêt immédiat de la boucle, les clauses de taille 1 sont supprimées des clauses et les clauses de taille supérieure sont simplifiées à chaque tour de boucle. On peut cependant discuter de la complexité dans le pire des cas. Imaginons que dans le pire des cas, à chaque tour de boucle, une et une seule nouvelle clause de taille 1 soit trouvée et simplifie une et une seule clause en clause de taille 1, traitée au tour de boucle suivant. Alors, nous faisons un nombre de tours égal à $\sum_{i=1}^{w} i$, c'est-à-dire une complexité bornée en $O(w^2)$. Cependant, comme nous travaillons dans un contexte où le nombre de sous-clauses est au plus 4, on peut considérer la simplification comme s'exécutant en temps virtuellement linéaire par rapport aux n lampes. Notons tout de même que pour un nombre petit de lampes, l'algorithme sera borné par un temps quadratique

Ensuite, nous exécutons l'algorithme de Tarjan sur le graphe d'implication créé à partir de nos clauses. Puisque nous sommes dans le contexte d'un problème 2-SAT, le nombre maximum d'arrêtes est de **2w**, pour w clauses. On obtient par la complexité de Tarjan que déterminer le nombre de compostantes fortement connexes de notre graphe d'implication se fait en temps O($2c + 2w$), avec c le nombre de switchs impliqués dans nos clauses. On peut donc dire que Tarjan s'exécute en temps linéaire dans le contexte de notre problème 2-SAT.

Enfin, pour chaque interrupteur, nous vérifions si lui et son opposé appartiennent tous les deux à la même composante fortement connexe. Cette vérification s'exécute en temps linéaire dans le pire des cas.

Nous avons donc un algorithme borné par une complexité en $O(n^2)$ (mais virtuellement $O(n)$) pour les instances 2-SAT relatives au problème des lampes du projet. Il est important de noter cependant que le nombre de modes d'allumage influe largement sur cette complexité, car si le nombre de clauses devient très grand, alors nous ne pouvons plus considérer la simplification des clauses comme une opération en temps linéaire.

3 Phase 2

3.1 Idée Générale

Dans la phase 2 de ce projet, nous sommes amenés à résoudre un problème plus général que le premier : est-il possible de connaitre le nombre maximal de lampes pouvant être allumées simultanément?
Une solution naturelle et intuitive serait de faire un backtracking en partant d'une liste de lampes vide. Pour chaque ajout de lampe, nous vérifions avec l'algorithme de la phase 1 s'il est possible d'allumer la grille. Si tel est le cas, alors nous continuons à essayer d'ajouter des lampes. Sinon, nous coupons la branche et nous enlevons cette lampe de la permutation actuelle. Le but de ce backtracking est de retenir le nombre maximal de lampes pouvant être allumées simultanément sur la grille, sans que cela ne cause de contradiction dans l'expression booléenne caractéristique.
Nous sommes cependant gênés par la complexité de la mise en oeuvre d'un tel algorithme. En effet, nous sommes face à un algorithme en temps non polynomial (backtracking). Il s'agit donc d'une approche inefficace.

3.1.1 Discussion de la nature de la classe du problème

Le problème de la phase 2 constitue en réalité d'un problème de type **MAX 2-SAT**, ou *problème de satisfiabilité maximale à 2 variables*. Nous savons qu'il s'agit d'un problème NP-complet [Rya04]. Démontrons cependant ce résultat pour se convaincre qu'il ne sera pas possible de trouver un algorithme polynomial pour résoudre le problème, à moins que P = NP :

Prenons une instance d'un problème 3-SAT, tel que chaque clause $C_i = (l_1 \wedge l_2 \wedge l_3)$. Pour chacune de ces clauses, créons les 10 sous-clauses suivantes :

$$(l_1), (l_2), (l_3), (d_i), (\neg l_1 \wedge \neg l_2), (\neg l_2 \wedge \neg l_3), (\neg l_1 \wedge \neg l_3), (l_1 \wedge \neg d_i), (l_2 \wedge \neg d_i), (l_3 \wedge \neg d_i) \tag{3.1}$$

Il y'a exactement 8 états possible pour (L_1, L_2, L_3).

— (0, 0, 0) ⇒ La clause C_i n'est pas respectée, et l'assignement respecte au plus 6 des 10 clauses 3.1.1, peu importe la valeur de d_i.

— (0, 0, 1), (0, 1, 0) et (1, 0, 0) ⇒ La clause C_i est respectée. Avec $d_i = 0$, exactement 7 des 10 clauses 3.1.1 sont remplies simultanément.

— (0, 1, 1), (1, 0, 1), (1, 1, 0), (1, 1, 1) ⇒ La clause C_i est également respectée. Exactement 7 des 10 clauses 3.1.1 sont remplies simultanément, peu importe la valeur de d_i.

Introduisons la réduction du problème 3-SAT en problème MAX 2-SAT :

1. Pour chaque clause dans notre 3-SAT $C_1, C_2, ..., C_m$, déclarons les 10 clauses correspondantes. Nous avons donc un total de *10m* clauses au total pour un problème MAX 2-SAT.
2. Tout état possible qui satisfait l'expression du problème 3-SAT doit également satisfaire 7 des 10 clauses du problème MAX 2-SAT. On en déduit logiquement que satisfaire toutes les *m* clauses du problème 3-SAT implique de satisfaire *7m* clauses du problème MAX 2-SAT.
3. On écrit alors k = 7m, dans le problème MAX 2-SAT.

Puisque nous savons que le problème 3-SAT est un problème NP-complet et que nous avons montré que le problème MAX 2-SAT était réductible à ce même problème, nous en concluons que, à moins que P = NP, le problème MAX 2-SAT est NP-complet. Il n'existe donc pas d'algorithme en temps polynomial pour résoudre le problème de satisfiabilité maximale à 2 variables.

3.2 Implémentation

Comme discuté dans la section précédente, nous souhaitons implémenter un algorithme non polynomial de backtracking pour vérifier les permutations de lampes. Cet algorithme se base sur la fonction canBeTurnedOn implémentée dans la partie 1 ainsi que sur le concept du backtracking, pour générer les permutations. Une implémentation en pseudo-code est proposée ci-dessous :

```
function maxLampThatCanBeTurnedOn(lamps)

    si canBeTurnedOn(lamps):
        return lamps.size
    sinon:
        return backtracking([], 0)


function backtracking(currentConfig, currentMax):

    si currentConfig:
        si canBeTurnedOn(currentConfig):
            currentMax = max(currentMax, currentConfig.size)

    pour toutes les possibleLamp dans lamps:
        si possibleLamp not in currentConfig:
            currentConfig.add(possibleLamp)
            maxLamps = backtracking(currentConfig, maxLamps)
            currentConfig.remove(possibleLamp)

    return maxLamps
```

Listing 3.1: Résolution du problème MAX 2-SAT

3.2.1 Optimisations de l'algorithme

Si on peut remarquer quelque chose sur l'algorithme proposé dans la partie implémentation, c'est qu'il n'est pas très rapide. En réalité, utiliser un tel algorithme est très peu efficace, car borné en n! soit $O(n^n)$ via l'approximation de Stirling.

La première amélioration directe évidente est de changer la façon dont on génère les permutations. Il serait par exemple possible de générer toutes les permutations de positions des switchs concernés par les clauses de l'expression du problème MAX 2-SAT et de vérifier le nombre de lampes que cette même permutation activera. Ainsi, pour C switchs **différents** impliqués dans les clauses, on aura un nombre de permutation égal à 2^C, car deux états possibles pour chacun des switchs.

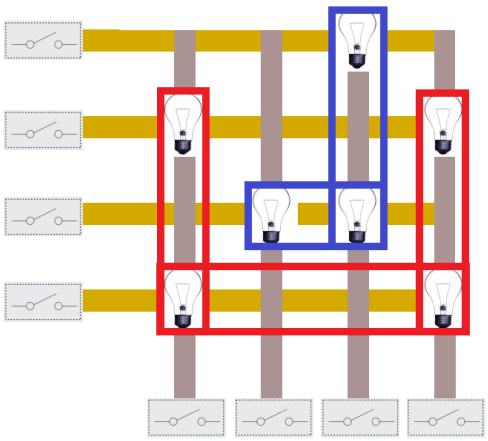

FIGURE 3.1: Les 2 clusters dans l'exemple 1.1

Une deuxième amélioration possible et extrêmement efficace consiste à séparer les lampes en clusters. L'idée serait de trouver (via une exploration avec un algorithme comme le BFS) les sous-groupes de lampes ayant une influence mutuelle sur les autres. La figure 3.1 ci-dessus représente les 2 clusters de l'exemple donné dans le cadre du projet.
Chaque clusters d_i serait ensuite analysé avec la méthode présenté dans la première optimisation. Le résultat du MAX 2-SAT est donc la somme du maximum de clauses satisfaites dans chacun des clusters d_i. De ce fait, la complexité devient borné par quelque chose de plus petit que 2^C, à savoir $d \times 2^{\frac{C}{d}}$, pour d clusters de taille à peu près équivalente.
Pour se convaincre de l'efficacité de ces optimisations se trouve ci-dessous 3.2 une comparaison du temps d'exécution de l'algorithme classique sans clustering et l'algorithme avec, sur une grille avec un nombre de lampes variant de 1 à 30 :

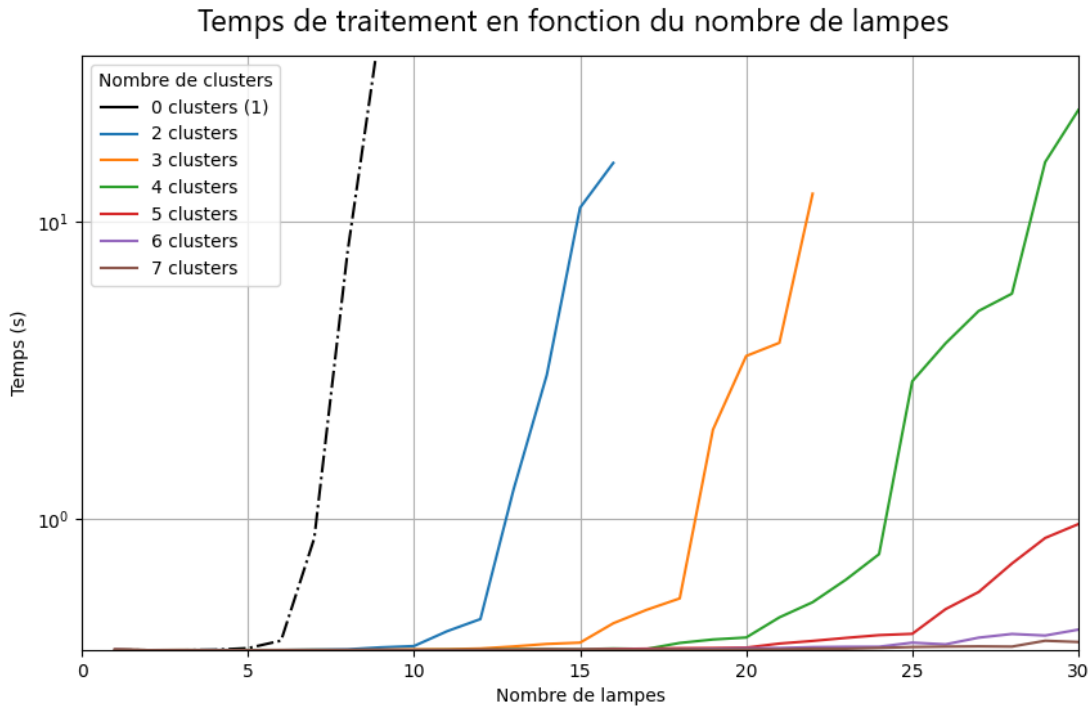

FIGURE 3.2: Benchmark en fonction du nombre de clusters

Il est clair que lorsque le nombre de clusters augmente, l'algorithme peut facilement segmenter le problème en plusieurs sous-problèmes plus gérables. En général, l'algorithme reste efficace pour un nombre de clusters représentant environ 15 à 20 % du nombre total de lampes. Il est également important que les clusters soient répartis de manière équilibrée. L'objectif réel est de limiter chaque cluster à 7 à 8 lampes maximum. Au-delà, cela équivaut en pratique à ne pas utiliser de clustering du tout, ce qui se traduit par des coûts en temps et en calculs considérables, comme le montre le graphique.

Articles

[oub23] OUBELLI, Syphax Ait (2023). « How to solve the 2-SAT problem in POLYNOMIAL TIME? » In : URL : https://www.youtube.com/watch?v=Ku-jJ0G4tIc.

[Par19] PARREAUX, Julie (2019). « Le problème 2-SAT ». In : URL : https://perso.eleves.ens-rennes.fr/people/julie.parreaux/fichiers_agreg/info_dev/2SAT.pdf.

[Rya04] RYAN WILLIAMS, Carnegie Mellon University (2004). « Maximum 2-satisfiability ». In : URL : https://people.csail.mit.edu/rrw/williams-max2sat-encyc.pdf.