ALGORITHMIQUE AVANCÉE

DEVOIR DE PROGRAMMATION
GESTION DES DIAGRAMMES DE DÉCISION BINAIRES

BENAKCHA Mebarek Raid

AMNACHE Miassa

Sous la supervision de : M. Antoine GENITRINI

Echauffement

1. Choix du langage de programmation

Comme nous l'avons vu au cours, " Il est toujours préférable de réfléchir aux contraintes avant de choisir les structures de données " Mr.GENITRINI.

Pour implémenter l'ensemble de fonctionnalités demandés dans l'énoncé du projet, nous avons opté pour JavaScript pour les raisons suivantes :

• La maîtrise du langage

• Exécution rapide

• Dynamiquement typé

• Facile à manipuler

• Les méthodes et les structures de données intégrées (toString(2), les objets, les classes, les tableaux de taille dynamiques...)

• Support des entiers de tailles arbitraires avec la classe BigInt, ce qui répond essentiellement à la contrainte de cette question.

2. Décomposition binaire des entiers naturels

```
const decomposition = (x) => {
  const result = []
  const x2 = x.toString(2)
  for (let i = x2.length - 1; i >= 0; i--) result.push(x2[i] == 1)
  return result
}
```

Le JavaScript comprend plusieurs méthodes intégrées qui facilitent l'obtention des informations supplémentaires d'une donnée telle que dans notre cas, dans le projet, on a besoin de trouver la représentation binaire d'un entier. La méthode toString(2) offerte par JavaScript retourne directement la représentation binaire d'un entier sous forme de chaîne

de caractères tel que le bit du poids fort est en tête de la chaîne. On boucle à partir du dernier caractère de la chaîne jusqu'au premier pour construire le tableau correspondant à la question.

3. Complétion du tableau

```
const completion = (l, n) => {
  if (n <= l.length) return l.slice(0, n)
  else {
    for (; l.length < n; ) l.push(false)
    return l
  }
}
```

La fonction completion prend une représentation binaire d'un entier ***l*** et un entier ***n***, puis retourne la liste tronquée de ***l*** ne contenant que ses n premiers éléments, sinon elle retourne ***l*** complétée par false jusqu'à n éléments.

4. Construction de la table de vérité

La fonction table retourne la table de vérité obtenue à partir de la représentation binaire d'un entier n. Nous avons combiné les deux fonctions précédentes pour la définir.

```
const table = (x, n) => {
  let dec = decomposition(x)
  return completion(dec, n)
}
```

5. Représentation des arbres binaires de décision

Un arbre binaire de décision est composé de nœuds qui ont exactement deux successeurs et il est défini par sa racine root, initialement ***null***. Il contient également l'objet

(table de hachage/dictionnaire) ***table*** qui va servir à compresser l'arbre dans les questions suivantes, ainsi que les méthodes qui aident à le construire et à le compresser.

```
export class Arbre {
  constructor() {
    this.root = null
    this.table = {}
  }
> cons_arbre(liste) {···
  }
> luka() {···
  }
> compression() {···
  }
> dot() {···
  }
> compression_bdd() {···
  }
}
```

```
class Node {
  constructor(data, left, right) {
    this.data = data
    this.left = left
    this.right = right
  }
}
```

Un noeud de l'arbre est un objet qui contient initialement trois champs :

- data : comporte la valeur du noeud (soit Xi s'il s'agit d'un noeud interne, sinon true/false s'il s'agit d'une feuille)

- left : comporte une référence vers le fils gauche du nœud.
- right : comporte une référence vers le fils droit du nœud.

6. Construction de l'arbre binaire de décision

Pour construire l'arbre binaire de décision, nous avons implémenté la fonction cons_arbre, qui prend une table de vérité en argument et construit l'arbre binaire de décision en suivant les étapes suivantes :

- Si la table est de la taille 1, la racine est une feuille qui porte la valeur unique de la table.

Sinon

- On calcule la taille de l'arbre avec la relation suivante height = $\lceil log_2(liste.length) \rceil$
- On instancie la racine avec un noeud qui porte la valeur X_{height}
- On initialise deux compteurs à 0, count qui représente l'index de la valeur booléenne dans la table de vérité et level qui représente le niveau du noeud dans l'arbre

Pour chaque noeud de l'arbre à partir de la racine, tant que level < height -1

  - On instancie son fils gauche et droit à 'height - level - 1'
  - On incrémente level par 1 et on fait le même traitement récursivement pour les fils instanciés.
  - Si level = height - 1, ça veut dire le noeud a 2 fils qui sont des feuilles, on les instancie avec les valeurs de l'indice count et count+1 (gauche et droit respectivement) dans la table de vérité.
  - On incrémente count par 2 et on décrémente i par 1

```
cons_arbre(liste) {
  if (liste.length == 1) {
    this.root = new Node(liste[0], null, null)
    return
  }
  const height = Math.ceil(Math.log2(liste.length))
  this.root = new Node(`x${height}`, null, null)
  const node = this.root
  let count = 0,
    level = 0
  let helper = (node) => {
    if (level < height - 1) {
      node.left = new Node(`x${height - 1 - level}`, null, null)
      node.left.id = Math.random()
      node.right = new Node(`x${height - 1 - level}`, null, null)
      node.right.id = Math.random()
      level++
      helper(node.left)
      helper(node.right)
      level--
    } else {
      node.left = new Node(liste[count], null, null)
      node.right = new Node(liste[count + 1], null, null)
      count = count + 2
    }
  }
  helper(node)
}
```

7. Mot de Łukasiewicz

```
cons_arbre(liste) {···
}
luka() {
  if (!this.root) return
  const node = this.root
  let helper = (node) => {
    if (!node.left) {
      node.luka = `${node.data}`
      node.id = Math.random()
      return `${node.data}`
    }
    if (!node.right) {
      node.luka = `${node.data}(${helper(node.left)})`
      return node.luka
    }
    node.luka = `${node.data}(${helper(node.left)})(${helper(node.right)})`
    return node.luka
  }
  helper(node)
}
```

Pour construire le mot de Lukasiewicz d'un arbre binaire de décision, il suffit de faire un parcours postfixe de l'arbre et pour chaque noeud :

Si le noeud est une feuille, son mot de Lukasiewicz est sa valeur data

Sinon ça sera 'data **(**luka(fils gauche)**) (**luka (fils droit)**)**'

8. Compression vers un DAG

Pour construire le graphe orienté acyclique à partir d'un arbre binaire de décision, on fait un parcours en profondeur, et pour chaque noeud, si le mot de Lukasiewicz n'existe pas dans la table de compression, ça veut dire c'est sa première occurrence, on le rajoute dans la table s'il s'agit d'une feuille. Sinon, on traite récursivement son fils gauche et droit. En dernier on met sa valeur à la référence du premier nœud qui a le même mot de Lukasiewicz dans la table de compression et de même pour ses fils gauche et droit.

```
compression() {
  const table = this.table
  const helper = (node) => {
    if (!table[node.luka]) {
      if (!node.left) {
        table[node.luka] = node
      } else {
        helper(node.left)
        helper(node.right)
        table[node.luka] = node
        table[node.luka].left = table[node.left.luka]
        table[node.luka].right = table[node.right.luka]
      }
    } else return
  }
  helper(this.root)
  return table
}
```

9. Génération du graphe DOT

La création du fichier dot se fait en itérant sur la table de compression. On crée les noeuds selon les mots de Lukasiewicz dans la table, les arêtes remplies selon l'existence du fils droit du noeud et les arêtes pointillées selon les fils gauches des noeuds.

```
dot() {
  const table = this.table
  let code = 'graph {'
  for (let node in table) {
    code = code + `"${table[node].luka} " [label="${table[node].data}"] `
    if (table[node].left) {
      code =
        code +
        `
      edge [style="filled"]
      "${table[node].luka} " -- "${table[node].right.luka} ";
      edge [style="dashed"]
      "${table[node].luka} "-- "${table[node].left.luka} ";
      `
    }
  }
  code = code + '}'
  return code
}
```

10. Génération du ROBDD

À partir d'un arbre binaire de décision, on génère un DAG (Merging Rule), puis pour chaque noeud de l'arbre, s'il s'agit d'une feuille, on renvoie un noeud portant la valeur booléenne de la feuille (Terminal Rule). Sinon, s'il s'agit d'un nœud interne, on teste si son fils gauche est le même nœud de son fils droit, on le supprime de la table de compression et on retourne l'un de ses fils (Deletion Rule).

```
compression_bdd() {
  const node = this.root
  const F = new Node(false, null, null)
  const T = new Node(true, null, null)
  F.luka = 'false'
  T.luka = 'true'
  T.id = Math.random()
  F.id = Math.random()
  const helper = (node) => {
    if (!node.left) {
      if (node.data == false) return F
      return T
    } else {
      node.left = helper(node.left)
      node.right = helper(node.right)
      if (node.left == node.right) {
        delete this.table[node.luka]
        const left = node.left

        delete table[node.luka]
        node = null
        return left
      } else {
        return node
      }
    }
  }
  this.compression()
  this.root = helper(this.root)
}
}
```

Preuves Sur Les ROBDD

Longueur du mot de Lukasiewicz

1. Nombre de noeuds et de feuilles dans un arbre binaire de décision

On montre par récurrence qu'un arbre binaire de décision de la hauteur h possède $2^h - 1$ noeuds internes et 2^h feuilles.

Cas de base : Un arbre réduit à une feuille contient 1 feuille et 0 nœud interne.

Supposons que la propriété est vraie pour chaque h, un arbre binaire de décision est composé d'une racine et deux sous-arbres. Par hypothèse de récurrence, cet arbre possède $1 + 2 \times (2^h - 1)\ noeuds\ internes + 2 \times 2^h$ feuilles, donc $2^{h+1} - 1\ noeuds\ internes$ et 2^{h+1} feuilles. Donc la propriété est vraie pour tout h naturel.

2. Longueur du mot de Lukasiewicz de la racine d'un arbre de hauteur h entre 0 et 9.

La longueur du mot de Lukasiewicz d'un arbre binaire de décision est calculée comme suit :

$2^h \times 5 + (2^h - 1) \times (4 + 2)$ tel que :

Au pire cas, les valeurs des feuilles sont à false => taille = 5 * nombre de feuilles

La taille des mots de Lukasiewicz des noeuds internes est le nombre de parenthèses (4) plus la taille de la valeur Xi (2) => taille = 6

Donc $l_h = 2^h \times 5 + (2^h - 1) \times (4 + 2) = 11 \times 2^h - 6$

Complexité de l'algorithme de compression

En utilisant un algorithme qui utilise la structure d'un lexicographique comme exemple pour la démonstration, telles que les branches sont les mots de Lukasiewicz et les feuilles sont les références des nœuds. Le nombre de comparaisons total est égal à la

sommes des nombres de noeuds dans chaque niveau dans l'arbre multiplié par la longueur du mot de Lukasiewicz à ce niveau.

$$= \sum_{i=0}^{h} 2^{h-i} \times (11 \times 2^{i} - 6)$$

$$= \sum_{i=0}^{h} 2^{h-i} \times (11 \times 2^{i}) - 6 \sum_{i=0}^{h} 2^{h-i}$$

$$= \sum_{i=0}^{h} 2^{h} \times 11 - 6 \times (2^{h+1} - 1)$$

$$= 2^{h} \times 11(h + 1) - 12 \times 2^{h} + 6 = (11h - 1) \times 2^{h} + 6$$

La complexité en fonction du nombre des noeuds

Soit n le nombre de nœuds de l'arbre et h sa hauteur. On peut trouver h à partir du n par la relation suivante : $h = log_2(n + 1) - 1$. En remplaçant dans l'équation de la question précédente on trouve :

$$(11(log_2(n + 1) - 1) - 1) \times 2^{log_2(n+1)-1} + 6$$

$$= (11log_2(n + 1) - 12) \times (n + 1)/2 + 6$$

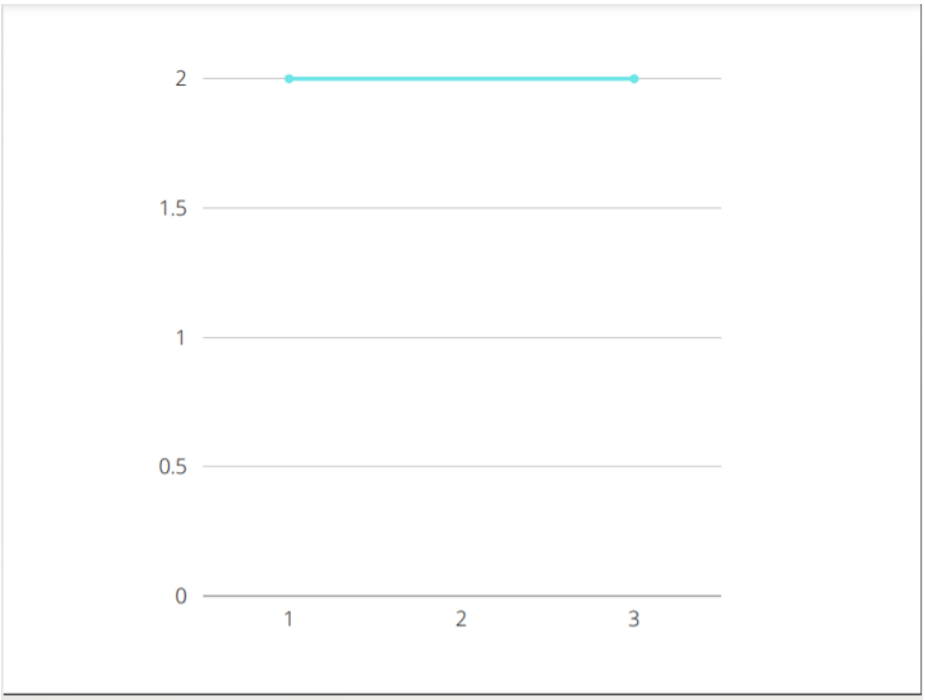
$$= 11/2\, n\, log2(n + 1) - 6n + 11/2\, log2(n + 1)$$

C'est une complexité en O(nlog n).

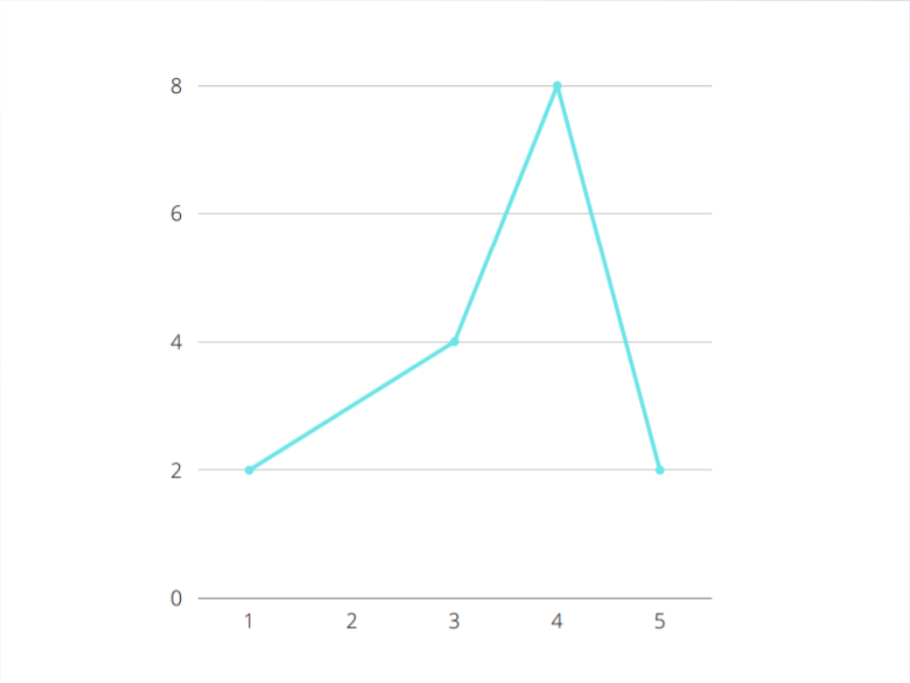
Étude expérimentale

1. Courbes de la Figure 9

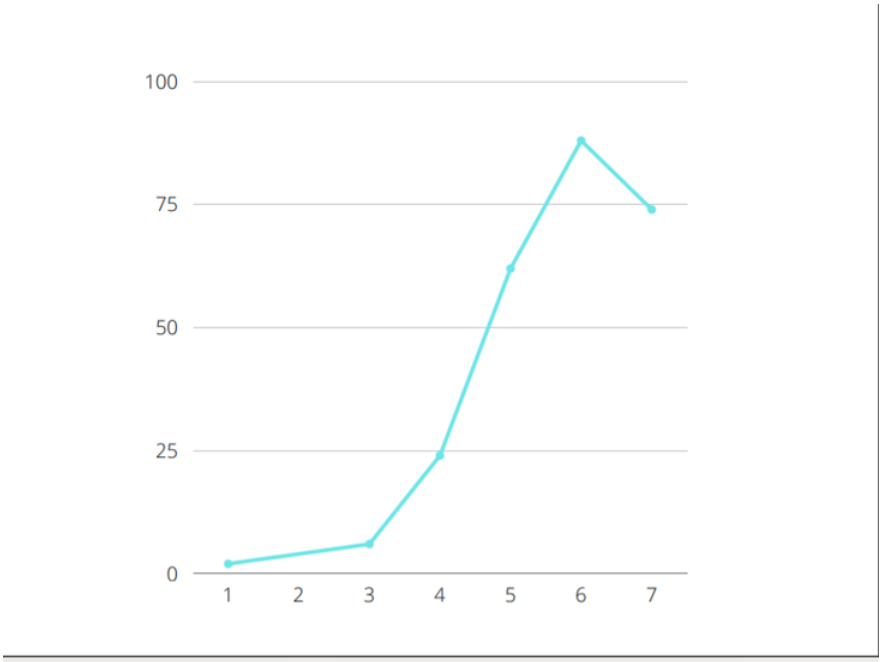
Nous avons calculé le nombre de nœuds des ROBDDs de une jusqu'à 4 variables en faisant un traitement exhaustif de toutes les fonctions booléennes qui permettent leurs générations. Nous avons obtenu les mêmes résultats :



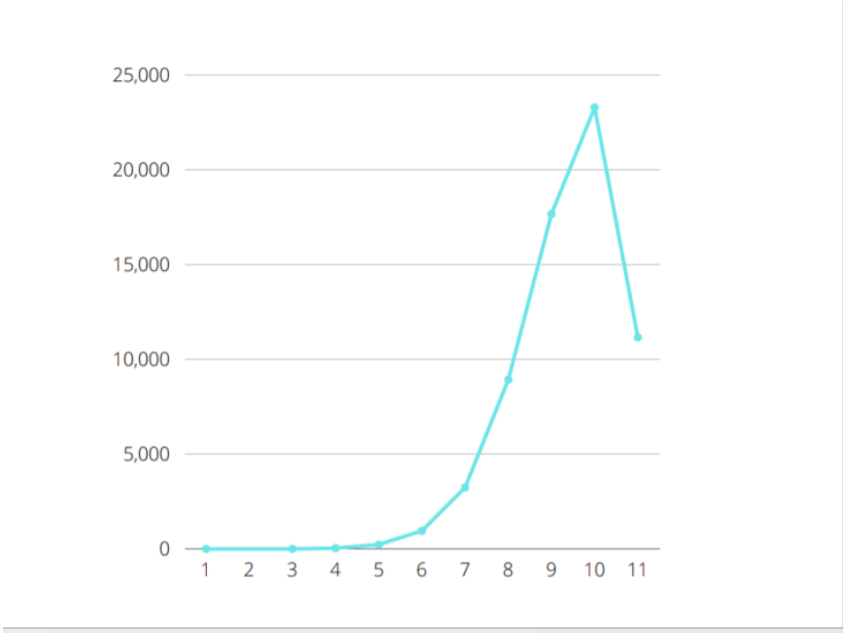
Nombre de fonctions booléennes en fonction du nombre de nœuds du ROBDD à 1 variable



Nombre de fonctions booléennes en fonction du nombre de nœuds du ROBDD à 2 variables



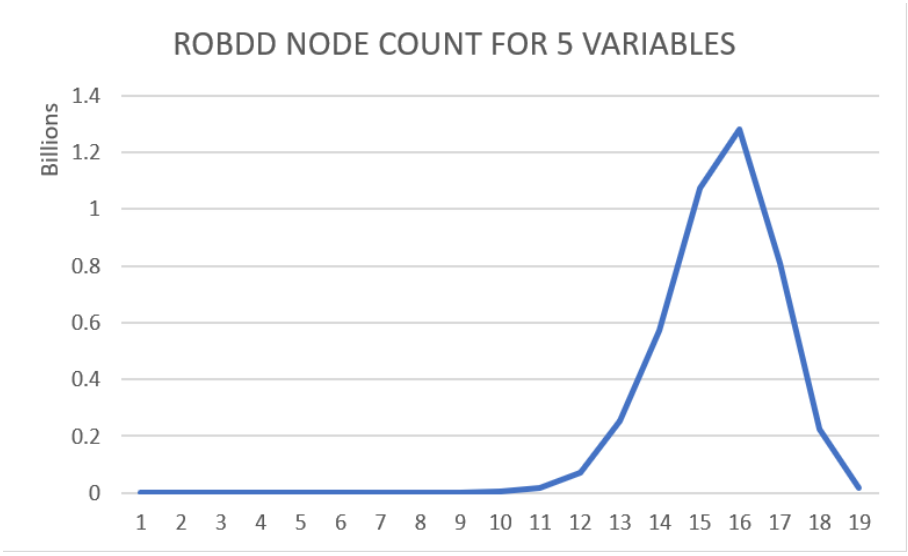
Nombre de fonctions booléennes en fonction du nombre de nœuds du ROBDD à 3 variables

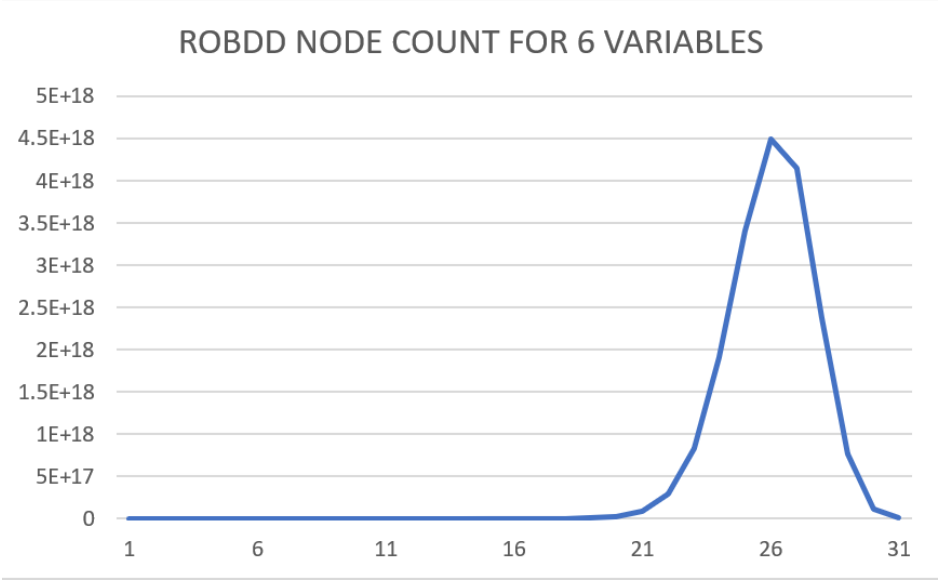


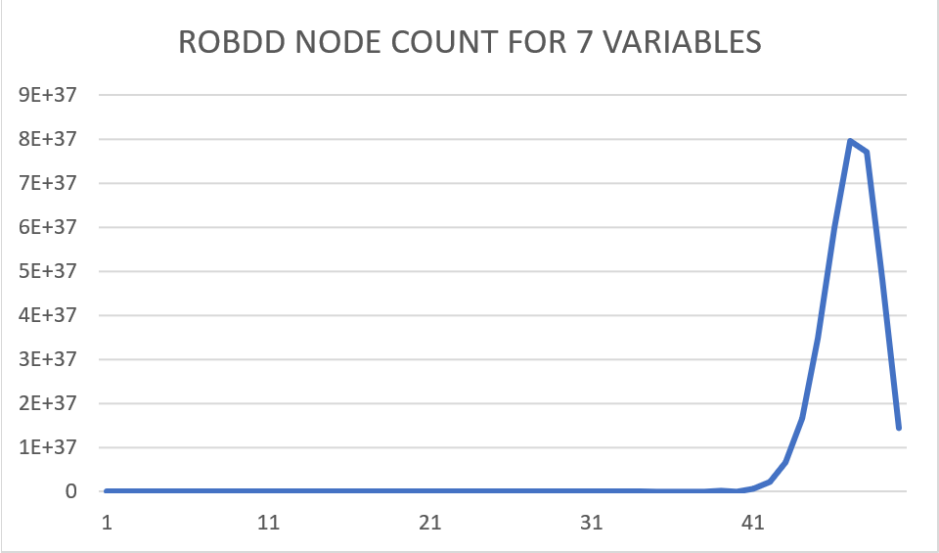
Nombre de fonctions booléennes en fonction du nombre de nœuds du ROBDD à 4 variables

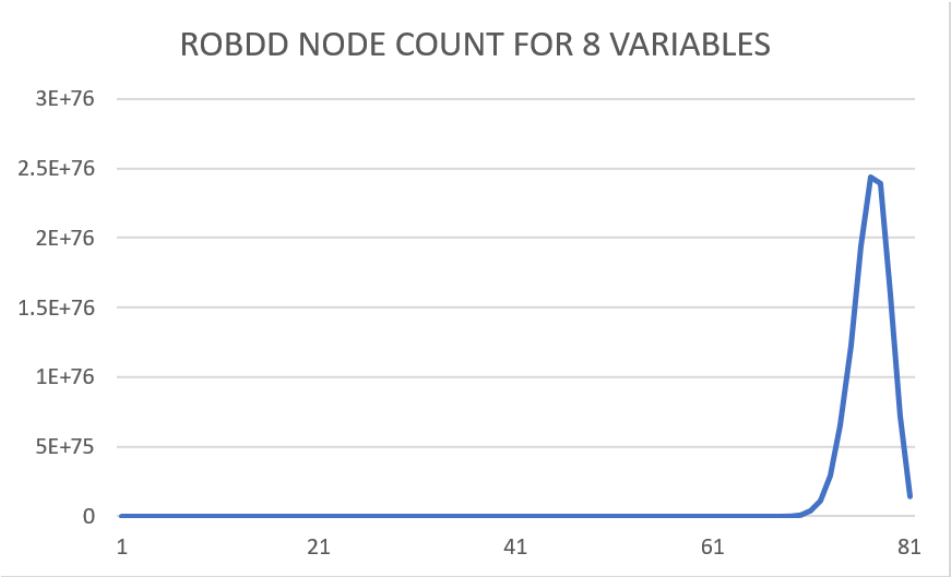
Nous n'avons pas pu calculer le nombre de noeuds pour les ROBDDs de 5 variables avec la même méthode exhaustive car le nombre de fonctions booléennes différentes permettant de générer des arbres de 5 variables est énorme (4 294 967 296 fonctions booléennes différentes), donc nous avons procédé comme suggéré dans l'article de Newton et Verna en faisant l'extrapolation linéaire.

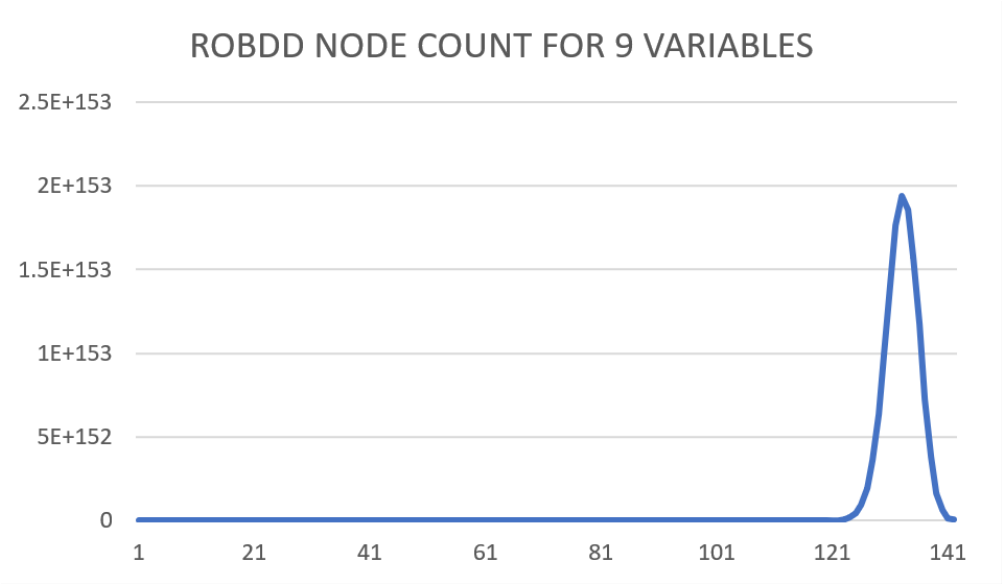
Nous avons donc pu obtenir les résultats suivants :

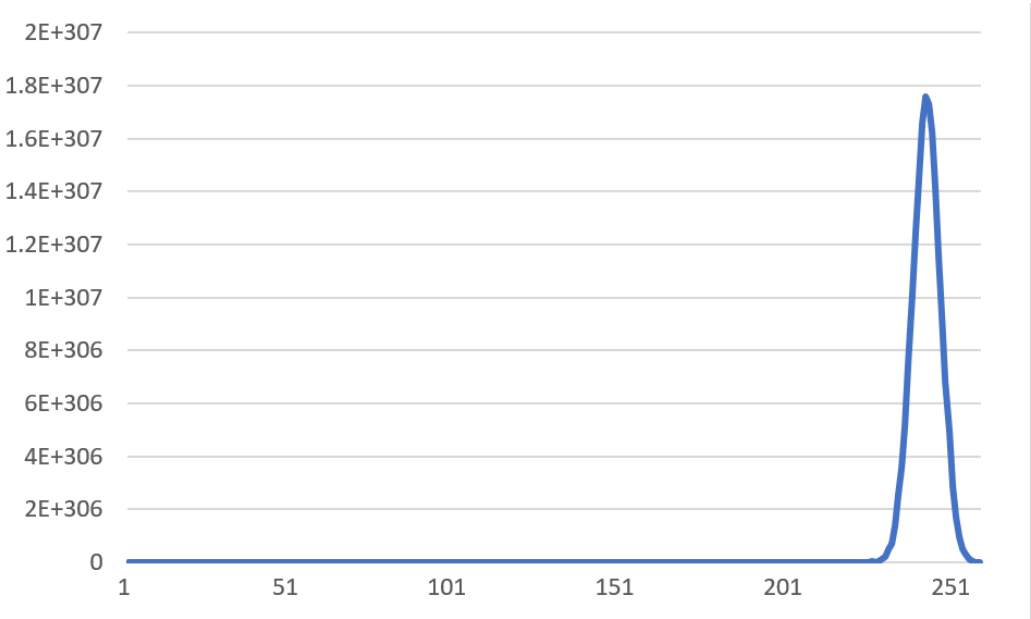





ROBDD NODE COUNT FOR 7 VARIABLES
9E+37
8E+37
7E+37
6E+37
5E+37
4E+37
3E+37
2E+37
1E+37
0
1
11
21
31
41


ROBDD NODE COUNT FOR 8 VARIABLES
3E+76
2.5E+76
2E+76
1.5E+76
1E+76
5E+75
0
1
21
41
61
81





ROBDD NODE COUNT FOR 10 VARIABLES

No. Variables	No. Samples	No. Unique Sizes	Compute Time (s)	Seconds per ROBDD
5	500003	13	64	0.00012
6	400003	17	89	0.00022
7	486892	16	192	0.00039
8	56343	13	42	0.00074
9	94999	22	152	0.0016
10	17975	33	49	0.00272

La méthode utilisée pour calculer les temps d'exécution est Date.now() offerte par JavaScript, on sauvegardant le temps dans l’instant du début de la génération des ROBDD ensuite en soustrayant le temps de l’instant le comptage des nœuds termine.