Projet de programmation par contraintes

Cyclic Bandwidth

Sullivan Bitho - Théo Hermegil - Duc Anh Le

Université de Nantes — UFR Sciences et Techniques
Master informatique parcours "optimisation en recherche opérationnelle" (ORO)
Année académique 2021-2022

Avril 2022

Table des matières

1 Introduction

Le problème de la cyclic bandwidth est un problème $\mathcal{NP}$-dure de la théorie des graphes. Il consiste à trouver un étiquetage de sommets de manière a minimiser la *distance* maximale entre ceux-ci. Il existe des applications de ce problème en informatique des réseaux, pour envoyer des messages à des clients/serveurs en un minimum d'étapes, ou dans l'ingénierie des circuits intégrés pour n'en citer que quelques unes.

Soit un graphe G et son étiquetage associé f. La *distance* d entre 2 sommets u et v est une distance cyclique. Elle se calcul comme suit : $d(u,v) = min\{|f(v) - f(u)|,\ n - |f(v) - f(u)|\}$. La cyclic bandwidth donne la distance maximale résultant de l'étiquetage choisit. Elle se formule de la manière suivante :

$$CB(G,f) = max_{(u,v)\in(G)}\{min\{d(u,v),\ n - d(u,v)\}\}$$

Le problème consiste à minimiser la valeur de la CB du graphe :

$$CB(G,g) = min_{f\in\varepsilon}\{CB(G,f)\}$$

avec ε l'ensemble des étiquetages possibles.

Inventaire des fichiers

Vous trouverez dans cette archive :

`Readme.txt` : contient la liste des commandes possibles

docs > : divers documents nous ayant aidé pour le projet
instances > : les dossiers d'instances
— `ann` *instances du groupe Adrien Nicolas Nicolas (*20)*
— `compare` *instances utilisées pour la comp des modèles (*10)*
— `didactique` *instances didactiques (*4)*
res > : les résultats d'experimentations
— `graphs` *contient les resultats graphiques des runs (plots)*
— `csv` *contient les résultats en format text et csv*
src > : codes sources
— `modeles` *contient les 5 modèles de M1 à M3_bis*
— `compare.py`
— `Run_M_ann.py`
— `solv_adaptatif.py`

Acronymes

Vous trouverez dans ce rapport les acronymes suivants :
— *PPP* : Partie Poste Présentation
— *AL*, *AM*, *dist*, *sym2* : AtLeast, AtMost, distance, symetrie 2 resp.
— *ann*, *k_ann* : resp. instances et $CB <= k$ utilisés par le groupe Adrien Nicolas Nicolas pour leurs comparaisons

Nous allons présenter dans ce document les différentes approches de résolutions que nous avons expérimentées. Une première approche (modèle 1) consiste à résoudre le problème d'optimisation tel quel. Les deux suivantes (modèles 2 et 3) cherche une approche par satisfaction dichotomique : l'une par des contraintes table, l'autre par des clauses booléennes.

2 Modélisation

2.1 Modèle 1

Soit un graphe $G = (V, E)$. On cherche un étiquetage g de taille $n = |V|$ pour le problème :

$$\begin{array}{ll} Min & max_{(u,v)\in E}\{min\{|g_u - g_v|,\ n - |g_u - g_v|\}\} \\ s.t. & AllDifferent(g_i) \\ & g_i \in \{1, \ldots, n\},\ \forall i \in V \end{array}$$

2.2 Modèle 2

Soit un graphe $G = (V, E)$. Soit $n = |V|$ et k un nombre entier tel que $k \leq n$. Les paires d'étiquettes qui ont une distance inférieure ou égale à k sont stockées dans une table :

$$Table = \{(u, v) \mid min(|u - v|,\ n - |u - v|) \leq k \text{ et } u \neq v\}$$

On cherche un étiquetage g de taille n tel que :

$$\begin{array}{ll} \text{SAT} & max_{(u,v)\in E}\{min\{|g_u - g_v|,\ n - |g_u - g_v|\}\} \leq k \\ s.t. & AllDifferent(g_i),\ g_i \in \{1, \ldots, n\}\ \forall i \in V \\ & (g_i, g_j) \in Table\ \forall (i, j) \in E \end{array}$$

2.3 Modèle 3

Le modèle 3 modélise les contraintes du problème de façon purement booléenne. Alors que nous travaillions sur la modélisation des contraintes *AtMost* et *AtLeast*, une propriété intéressante nous est apparue : contrairement aux exemples de placement de rennes sur un échiquier comme vu en TD, il s'agit ici d'une affectation bijective entre sommets et étiquettes.

Dans l'exemple du placement des rennes, nous avions besoin d'un ensemble de contraintes dit *AtLeast* pour que chaque renne soit affectée à au moins une case de l'échiquier. Deux ensembles de contraintes dit *AtMost* se charge de 1) imposer une renne maximum par ligne et 2) une renne maximum par colonne.
Dans notre problème d'étiquetage, étant donnée que chaque sommet doit être etiqueté, et que chaque étiquette doit étiqueter, nous avons une bijection d'affectation. Cette dernière forme une matrice des solutions carré. Cela nous permet d'économiser un des deux AtMost. En effet le second AtMost se déduit du AtLeast et du premier AtMost. Une petite démonstration par l'absurde permet de s'en convaincre.

Soit un graphe $G = (V, E)$. Soit $n = |V|$ et k un nombre entier tel que $k \leq n$. Soit $x_{i,j}$ la variable booléenne qui représente l'affectation de l'étiquette j au sommet i avec $i, j \in I$, $I = \{1, \ldots, n\}$. Soit les fonctions :

$a : V^2 \rightarrow \{TRUE, FALSE\}$ qui retourne vrai s'il existe une arête entre deux sommets.

$d : I^2 \rightarrow \{TRUE, FALSE\}$ qui retourne vrai si la distance entre deux étiquettes est inférieur ou égale à k.

Voici le modèle que nous avons finalement obtenu :

$$
\begin{aligned}
\text{SAT} \quad & max_{(u,v)\in E}\{min\{|f_u - f_v|,\ n - |f_u - f_v|\}\} \leq k \\
s.t. \quad & \bigwedge_{i\in V} \bigvee_{j\in I} x_{i,j} = TRUE && \text{(AtLeast)} \\
& \bigwedge_{j\in I} \bigwedge_{i,p\in V\,:\,i\neq p} \neg\left(x_{i,j} \wedge x_{p,j}\right) = TRUE && \text{(AtMost)} \\
& \bigwedge_{i,p\in V\,;\,j,q\in I} x_{i,j} \wedge x_{p,q} \wedge a(i,p) \implies d(j,q) = TRUE \\
& f_i \in I,\ \forall i \in V
\end{aligned}
$$

3 Symétries

Par "étiquetage symétrique" nous comprenons ici un étiquetage équivalent par symétrie. Ainsi, deux étiquetages ayant la même valeur de CB ne seront pas nécessairement symétriques. Nous partons du principe que les instances sont connexes. Si ce n'est pas le cas, un découpage du graphes en sous graphes connexes est nécessaire. On pourra alors calculer les CBs de chaque morceaux connexe, et calculer la CB maximum parmi ceux-ci.

Nous avons identifié deux symétries pour ce problème :

- Une symétrie sur le cercle, lorsque nous trouvons un étiquetage, on peut tourner l'étiquetage dans un sens ou dans l'autre on conservant la même configuration.
  Pour casser cette symétrie, il suffit de fixer préalablement un sommet.
- Une symétrie miroir, on peut imaginer une ligne qui sépare le cercle en deux parties que l'on peut inverser.
  Pour briser cette symétrie, on s'assure que le sommet à droite du sommet fixé est inférieur au sommet à gauche du sommet fixé. Cette astuce fonctionne car le calcul de la bandwidth est cyclique, on peut donc inverser l'ordre des étiquetages et retomber sur un étiquetage symétrique.

Il est intéressant de noter la simplification d'énumération de solutions, en effet pour pour un problème à 10 sommets avec une cyclic bandwidth de 3, nous sommes passé de 4800 à 240 étiquetages, soit 20 fois moins.

Briser les symétries n'a de sens que pour le modèle 1 qui a besoin d'explorer tous les espaces de recherches pour trouver la valeur optimale. Les modèles 2 et 3 eux s'arrête dès que pySAT a trouvé une affectation de littéraux qui satisfie les clauses.

4 Implémentation

Chaque implémentation des modèles présentée ici a été dûment commenté dans les codes sources correspondants (dossier `/src`)

4.1 Implémentation de M1

L'implémentation du modèle 1 consiste en une simple réécriture du problème en pyCSP3. Le modèle 1 était très simple à implémenter mais son exécution est très lente. En fait, rien que la fonction minimize() coûte déjà a elle seule un temps de calcul déraisonnable pour un nombre

d'arêtes m > 300.

4.2 Implémentation de M2

Pour le modèle 2, nous nous servons d'un distancier précalculé pour un nombre de sommet donné, et de la matrice des arêtes du graphe en entrée. Ces deux matrices réunies nous permettent d'avoir une liste d'étiquetages possibles, que nous insérons dans une contrainte en extension dit contrainte table. Vérifier si un étiquetage est autorisé revient à vérifié si la paire d'étiquetage d'une paire de sommet est présent dans la table.

En remaniant le problème pour le transformer en problème de satisfiabilité, nous avons drastiquement réduit la complexité. Le nombre d'étiquetage possible pour un nombre n de sommets est de $n!$ ($\in \mathcal{NP}$-complet). Le distancier se calcule seulement en $\mathcal{O}(n^2)$. Le modèle 3 transforme également le problème d'optimisation en problème de décision mais implémente des contraintes purement booléennes.

4.3 Implémentation de M3

Notre première implémentation "naïve" de M3 consiste a vérifier pour chaque sommet s'il est possible de l'étiqueter e en respectant les contraintes de distances et de connexités du graphe. Cette approche résulte en une complexité exponentielle de résolution. En effet, pySAT n'a pas d'information sur notre problème et semble résoudre les clauses en utilisant de la force brute. Pour alléger ces calculs[1], nous nous sommes inspiré de ce qu'on nous avons vu en cours, à savoir la propagation de contrainte et la cohérences d'arcs (AC). Après quelques recherches, nous avons constaté que notre algorithme effectue des propagations de contraintes dit à gros grain (*coarse-grained*). Il s'agit des techniques utilisés par les solveurs traditionnels comme ILOG ou CHOCO. [1]

Illustrons notre idée sur un graphe didactique où l'on cherche si $\exists$ CB <= 1 :

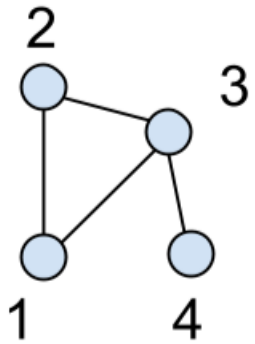

FIGURE 1 – Graphe didactique a 4 sommets

Dans cet exemple, notre première implémentation a une complexité en $n!$.
L'idée de notre version AC est d'initialiser un pré-traitement qui va propager une cohérence de contraintes.
Ici, au départ, le domaine des $x[i]$ est $\{1, 2, 3, 4\}, \forall i \in \{1..4\}$
Nous commençons par fixer $x[1] =$ "1" (ce qui brise la symétrie rotative). Les domaines de $x[2]$,

$x[3]$ et $x[4]$ passe à $\{2, 3, 4\}$.
La table des étiquetages possibles pour $k = 1$ est $T = [(1, 2), (2, 1), (2, 3), (3, 2), (3, 4), (4, 3), (1, 4), (4, 1)]$.
Vu qu'on a fixé $x[1] = 1$, les domaines de $x[2]$ et $x[3]$ passe à $\{2, 4\}$ ($(1, 3) \notin T$). Nous obtenons alors 2 variables qui ont un domaine de cardinalité 2 : on peut supprimer $\{2, 4\}$ du domaine de $x[4]$. Finalement $x[4] = \{3\}$.
Ainsi par propagation de contraintes on a $x[1] = \{1\}$, $x[2] = \{2, 4\}$, $x[3] = \{4, 2\}$ et $x[4] = \{3\}$.
On passe de 4! contraintes à 2 contraintes.

Voici l'algorithme utilisé pour *M3_bis* :

Algorithm 1 M3_BIS

```
1: function M3_BIS(Graphe G, k)
2:     M_dist ← pre_trait_M(G, k)
3:     domaines_cour ← {1,..,n} pour chaque sommet s de G
4:     domaines_prec ← [{}]
5:     while domaines_prec ≠ domaines_cour do    ▷ tant que l'ensemble des domaines évolue
6:         domaines_prec ← domaines_courant
7:         domaines_cour ← AC(domaines_cour) ▷ on calcul la cohérence d'arc des domaines
8:         if domaines_cour = "infaisable" then
9:             return "infaisable"
10:        else
11:            domaines_cour ← FiltrageDist(G, domaines, M_dist)        ▷ puis on filtre les
    domaines de leurs etiquetages infaisables par distance
12:        end if
13:    end while
14:    modele3 ← ConstructionClauses(G, domaines, M_dist)    ▷ on construit le modèle
    avec ses contraintes purement booléennes
15:    return solvePySAT(modele3)                ▷ on retourne sa résolution par PySAT
16: end function
```

Algorithm 2 AC

```
1: function AC(domaines)
2:     for d ∈ domaines do
3:         domaines_p[d] ← sommets ayant pour domaine d
4:         if len(d) < len(domaines_p[d]) then        ▷ si trop de sommet partage le domaine
5:             return "infaisable"
6:         else if |d| = |domaines_p[d]| then                    ▷ si égalité des cardinalités
7:             for d' ∈ domaines\{d} do        ▷ on retire cet ensemble des autres domaines
8:                 domaines ← domaines - {d'}
9:             end for
10:        end if
11:    end for
12:    return domaines                          ▷ on retourne les domaines arc-cohérents
13: end function
```

Algorithm 3 FiltrageDist

```
1: function FiltrageDist(G, domaines, M_dist)
2:     for arête (s1,s2) ∈ G do
3:         dom1 = domaines[s1]
4:         dom2 = domaines[s2]
5:         etiq_susp1 = dom1 \dom2 ▷ les étiquetages suspects sont ceux n'appartenant pas au domaine de l'autre
6:         etiq_susp2 = dom2 \dom1
7:         for e1 ∈ etiq_susp1 do                ▷ on ne vérifie que les distances suspectes
8:             if e1 < min(dom2) then
9:                 e2 = min(dom2)
10:            else
11:                e2 = max(dom2)
12:            end if
13:            if M_dist[e1][e2] = False then    ▷ si la distance est non vérifiée, on retire l'etiquetage suspect de son domaine
14:                domaines[s1] -= {e1}
15:            end if
16:        end for
17:        for e2 ∈ etiq_susp2 do
18:            if e2 < min(dom1) then
19:                e1 = min(dom1)
20:            else
21:                e1 = max(dom1)
22:            end if
23:            if M_dist[e1][e2] = False then
24:                domaines[s2] -= {e2}
25:            end if
26:        end for
27:    end for
28:    return domaines
29: end function
```

Le modèle `M3_bis`, comme les autres, a été commenté avec soin.

PPP : [1] Après expérimentation, ce qui prend le plus de temps de calcul n'est pas la résolution de pySAT mais bien la construction des clauses. Notre modèle $M3\_bis$ a certes un impact positif sur le temps de résolution de pySAT mais ce dernier est négligeable comparé à celui sur les temps de construction des clauses. (voir partie 6 : Comparaison des modèles SAT)

5 Résolution du problème d'optimisation avec M2 et M3

5.1 Résolution dichotomique

PPP : La dichotomie est possible car un graphe ayant un CB = k a aussi un CB = k+1.

Voici l'algorithme de recherche dichotomique de la cyclic bandwidth optimale :

Algorithm 4 DICHOSAT

```
1: function DICHOSAT(Modele M, Graphe G)
2:     borne_inf ← calcul_borne_inf_theorique(G)
3:     borne_sup ← calcul_borne_sup_theorique(G)
4:     k ← ⌈(borne_inf + borne_sup) / 2⌉
5:     k_best ← borne_sup
6:     while borne_inf ≠ borne_sup do
7:         if |borne_sup − borne_inf| = 1 then
8:             k ← borne non traitée
9:         end if
10:        decision = M(G, k)
11:        if decision = True then
12:            borne_sup ← k
13:        else
14:            borne_inf ← k
15:        end if
16:    end while
17:    if decision = True then
18:        k_best ← k
19:    else
20:        k_best ← borne_sup
21:    end if
22:    return k_best
23: end function
```

5.2 Autres approches de résolutions

Nous avons pris connaissance d'autres types d'approches pour la résolution du Cyclic Bandwidth via la littérature, notamment l'utilisation de méta heuristics comme GRASP et TABU, publié sur le site de Rafael Marti [2] et mentionnées dans l'article d'Eduardo Rodriguez-Tello et al. [3]. Une exploration de ces méthodes pourrait être faite dans le cadre d'une nouvelle étude.

6 Expérimentations et analyses

Pour reproduire les expérimentations ci-dessous, veuillez entrer les *commande utilisée*, précisées pour chaque tableau.

Protocole d'expérimentation :

- Environnement matériel :
  - **Mémoire** : 8GB (DDR4)
  - **CPU** : Intel(R) Core(TM) i3-6300 CPU @ 3.80GHz 3.79 GHz
  - **SSD** Kingston 112Go
- Solveurs :
  - **ACE** (pyCSP3) pour $M1$, $M1_{bis}$, $M2$
  - **Glucose3** (pySAT) pour $M3$, $M3_{bis}$
- Budget :
  - **200 secondes** pour les résolutions de ACE et Glucose 3 par instance et par modèle. Le temps total pour une instance peut donc monter encore plus selon les autres calculs (pré traitements et construction des clauses du modèle).

6.1 Comparaison et analyse des 3 modèles

Nous avons choisi les 10 premières instances classées par nombre de sommets croissant.

Légende :
— Les cellules coloriées en bleu clair indique que briser la symétrie a eu un impact sur le temps de résolution. (dans le tableau suivant il semblerait qu'un problème soit survenue (codage de la brise symétrie incorrecte ?))
— Les cellules en vert indique que le modèle a eu la meilleure performance sur cette attribut
— Les cellules en vert claire indique un second meilleure score pour l'attribut inf.
— Les cellules en rouge indique une pire performance parmi les modèles `SAT`.
$CB*$ pour CB optimale. $t(s)$ pour temps d'execution en seconde. inf pour meilleure borne inférieure trouvée dans le budget imparti.

commande utilisée : `python3 compare.py compare v [M1,M1_bis,M2,M3,M3_bis] 200`

Instance	$M1$		$M1\_bis$		$M2$		$M3$		$M3\_bis$	
	$CB*$	$t(s)$	$CB*$	$t(s)$	inf	$t(s)$	inf	$t(s)$	inf	$t(s)$
pores_1	inconnue	> 200	193	17.8	7	5.1	7	7.0	7	2.7
ibm32	inconnue	> 200	inconnue	> 200	9	5.1	9	201.3	9	201.7
bcspwr01	193	10.1	193	4.8	4	5.8	4	2.8	4	1.4
bcsstk01	inconnue	> 200	inconnue	> 200	12	11.3	13	430.3	13	562.4
bcspwr02	inconnue	> 200	inconnue	> 200	7	8.9	7	202.2	7	201.3
curtis54	inconnue	> 200	inconnue	> 200	8	8.1	8	93.5	8	29.4
will57	193	55.2	193	31.4	6	8.2	6	51.7	6	25.7
ash85	inconnue	> 200	inconnue	> 200	9	12.6	11	233.8	10	486.7
dwt_234	inconnue	> 200	inconnue	> 200	11	26.0	14	736.7	33	284.5
bcspwr03	inconnue	> 200	inconnue	> 200	10	29.2	14	518.7	12	584.5

TABLE 1 – comparaison des temps de résolution dichotomique avec et sans symétrie

Analyses :
— Nous constatons une dominance générale du modèle M2.
— Néanmoins, pour les petites instance (n $<$ 40 & m $<$ 100), les modèle $M3$ et $M3\_bis$ proposent de bonnes voire de meilleures performances que le modèle $M2$.
— le modèle $M3\_bis$ domine presque systématiquement le modèle $M3$ sur ces instances. Cela lui permet parfois de trouver de meilleures bornes inf que le modèle $M3$
— Briser les symétries a généralement un impact sur les modèles M1 et M3. (non visible lors de cette expérience).
— Le modèle M1 ne semble pas viable pour résoudre le problème de Cyclic Bandwidth

Hypothèses :
Nous pensons que la dominance du modèle M2 peut s'expliquer par au moins deux facteurs :

1. La transformation du problème d'optimisation en problème de décision résolu par dichotomie explique la dominance de $M2$ sur $M1$
2. L'utilisation du solveur `ACE` est plus performant que notre utilisation de pySAT dans les modèles $M3$ et $M3\_bis$.

Comparaisons graphiques :
Graphique généré dans `res/graphs/compare/`.
Les dominances relatives par instance se constatent graphiquement.

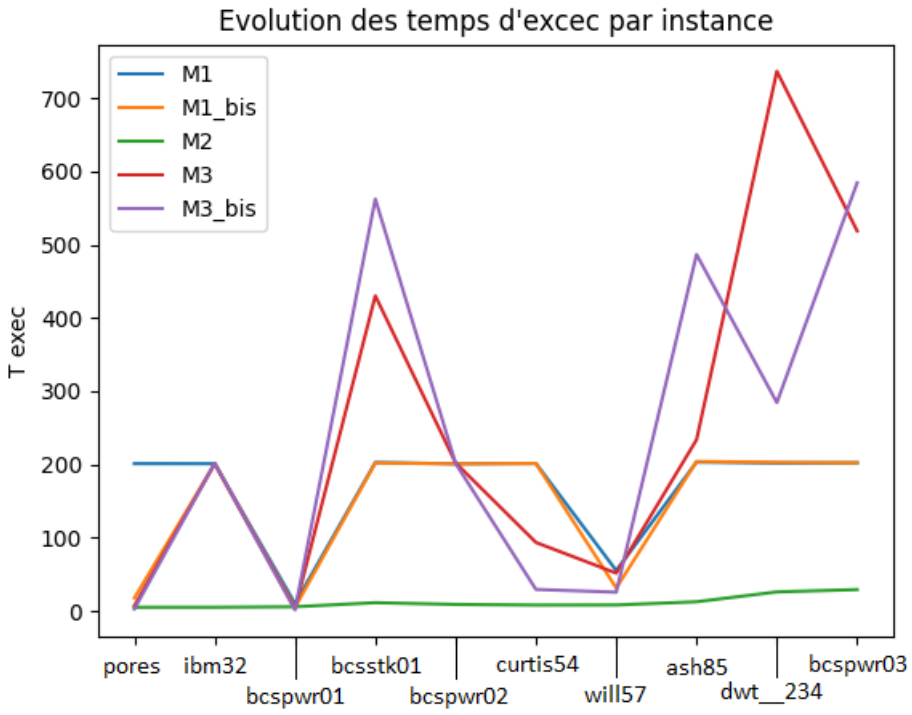

FIGURE 2 – comparaison des modèles $M1$ à $M3\_bis$ sur les instances `compare`

Nous avons créer des graphiques afin d'essayer de saisir d'où viennent les différences de temps de calcul pour les modèles 3. Ci-dessous un graphique montrant l'évolution des temps de calculs en fonction de la valeur k demandée (question $CB <= k$?) sur l'instance bcspwr01 :

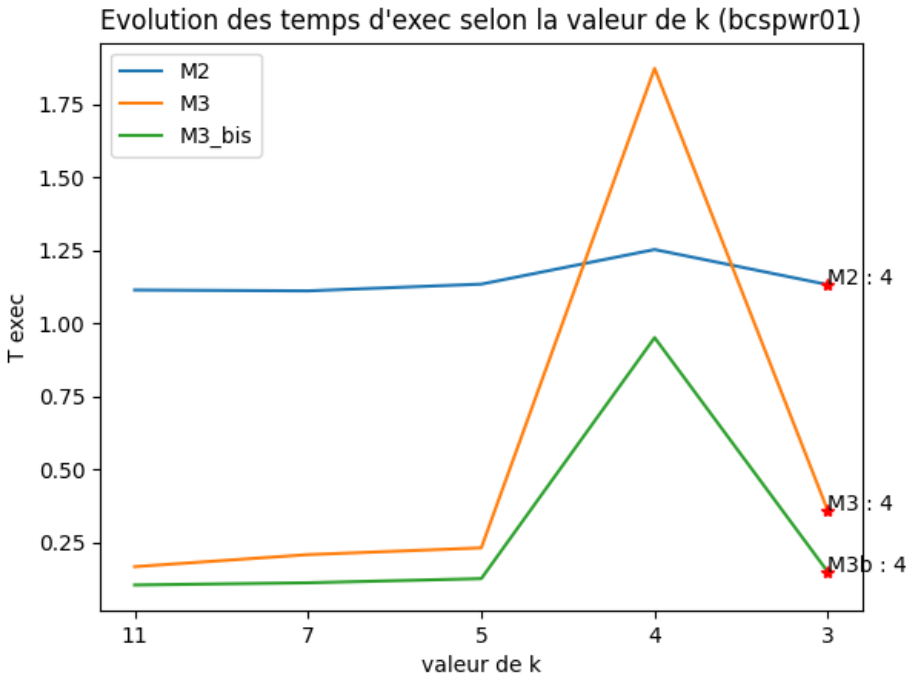

FIGURE 3 – les temps de calculs augmente pour les 3 modèles SAT $k = CB*$

Nous constatons que les temps de calculs subissent un soubresaut pour $k = CB*$. Afin d'avoir une meilleure visions sur les temps de calculs autour de $CB*$, prenons l'exemple de l'execution de bcspwr03 :

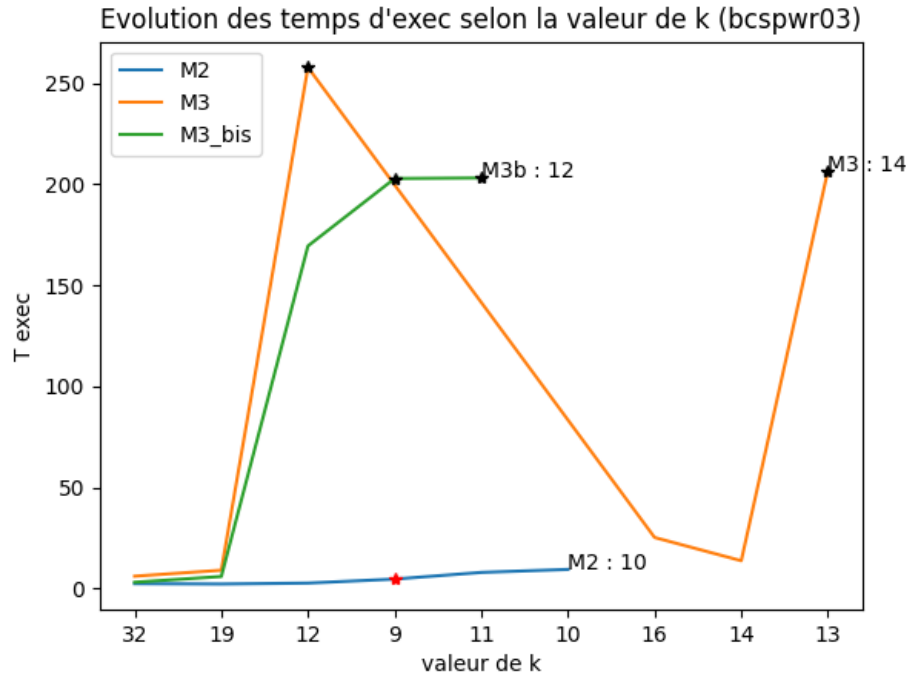

FIGURE 4 – les temps de calculs augmente autour de $CB*$

On confirme en effet que les temps de calculs juste en-dessous et juste au-dessus de $CB* = 10$ ($k = 12$, $k = 9$, $k = 11$) augmentent fortement pour $M3$ et $M3\_bis$, et diminue au fur et à

mesure que k s'éloigne de $CB*$. Veuillez trouver en annexe les autres graphes correspondant à ces comparaisons.

Conclusion de l'analyse :
Nous avons montré que sur des petites instances, $M3\_bis$ pouvait dominer M2. Bien que généralement meilleur que $M3$, $M3\_bis$ souffre de performance par rapport à M2 sur les grosses instances. Afin de faire valoir $M3\_bis$, nous pensons qu'une poursuite de développement de ce pré-solveur dédié pourrait montrer des résultats intéressants.

Dans la partie 5.2, nous allons confronter nos modèles 2 et 3 aux instances *ann*, et nous intéresser aux créations de clauses booléennes des modèles 3.

6.2 Comparaison des modèles `SAT`

Les cellules coloriées en vert correspondent au meilleur temps pour l'instance, en blanc le second meilleur temps et en rouge le pire temps.

commande utilisée : `python3 Run_M_ann.py ann v [M2,M3,M3_bis] 200`

Instance	n	m	k	temps $M2$	temps $M3$	temps $M3_{bis}$
pores_1	30	103	10	1.1789	0.2487	0.0829
ibm32	32	90	11	1.1582	0.1866	0.0948
bcspwr01	39	46	11	1.1332	0.1768	0.1063
bcsstk01	48	176	15	1.6038	0.8298	1.9642
bcspwr02	49	59	13	1.1998	0.3571	0.2064
curtis54	54	124	17	1.3832	0.8643	0.3314
will57	57	127	16	1.389	0.8934	0.439
impcol_b	59	281	19	2.0473	3.1166	7.2408
steam3	80	424	23	4.681	9.4847	3.0279
ash85	85	219	23	3.0938	4.1268	1.7348
nos4	100	247	26	3.3939	6.4251	3.4443
bcsstk22	110	254	29	3.5583	9.1142	3.6882
gre_115	115	267	31	4.2464	10.6475	6.6947
dwt_234	117	162	31	2.9951	17.6358	3.0273
bcspwr03	118	179	32	3.2134	7.6683	3.3202
lns_131	123	275	33	4.7581	10.7883	11.3845
west0132	132	404	38	171.3952	196.6222	157.9116
west0156	156	371	41	10.245	37.7669	56.8889
nos1	158	312	40	6.4184	21.2835	12.3545
saylr1	238	445	60	13.9646	68.2313	34.9708

TABLE 2 – performances des modèles $M2$, $M3$ et $M3_{bis}$

6.2.1 Comparaison du nombre de clauses

Par soucis de lisibilité, vous trouverez ci-joint les nombres des clauses en pourcentages du nombre total de clauses. Pour voir les nombres de clauses exacts, vous pouvez lancer la commande utilisée (avec l'option v).

— les lignes en bleu clair représentent les résultats pour le modèle $M3$

— les lignes en bleu foncé représentent les résultats pour le modèle $M3_{bis}$
— les cellules coloriées en rouge indique un temps pySAT plus important pour $M3\_bis$

commande utilisée : `python3 Run_M_ann.py ann v` $[M3, M3\_bis]$ `200`

Instance	nb AL	nb AM	nb dist	sym2	nb total	t pySAT (s)
pores_1	0.0%	18.9%	80.5%	0.6%	69106	0.077
pores_1	0.1%	31.8%	66.9%	1.3%	32091	0.0035
ibm32	0.0%	23.3%	76.0%	0.7%	68209	0.0005
ibm32	0.1%	37.5%	61.2%	1.3%	36395	0.0025
bcspwr01	0.0%	33.2%	65.9%	0.8%	87049	0.0007
bcspwr01	0.1%	49.7%	48.8%	1.4%	51376	0.0005
bcsstk01	0.0%	15.8%	83.9%	0.3%	342505	0.0432
bcsstk01	0.0%	27.1%	72.2%	0.6%	168381	1.9863
bcspwr02	0.0%	31.0%	68.4%	0.6%	186005	0.0082
bcspwr02	0.0%	46.9%	52.1%	1.0%	113335	0.0014
curtis54	0.0%	23.2%	76.4%	0.4%	333154	0.1335
curtis54	0.0%	39.6%	59.6%	0.8%	163828	0.003
will57	0.0%	20.7%	79.0%	0.3%	440041	0.0093
will57	0.0%	33.9%	65.5%	0.6%	250629	0.0035
impcol_b	0.0%	13.2%	86.6%	0.2%	765821	1.4765
impcol_b	0.0%	23.0%	76.6%	0.4%	402026	4.6045
steam3	0.0%	10.1%	89.7%	0.1%	2494681	0.7223
steam3	0.0%	18.5%	81.2%	0.3%	1170473	0.5024
ash85	0.0%	17.6%	82.2%	0.2%	1721761	0.1192
ash85	0.0%	29.9%	69.7%	0.4%	928891	0.0553
nos4	0.0%	17.5%	82.3%	0.2%	2821751	0.3974
nos4	0.0%	29.9%	69.7%	0.3%	1514685	0.5642
bcsstk22	0.0%	18.8%	81.1%	0.2%	3515326	0.8426
bcsstk22	0.0%	31.5%	68.1%	0.3%	1947980	0.0755
gre__115	0.0%	19.1%	80.8%	0.2%	3953701	1.1672
gre__115	0.0%	32.0%	67.7%	0.3%	2203746	2.4888
dwt__234	0.0%	27.9%	71.9%	0.2%	2847781	11.9115
dwt__234	0.0%	43.6%	56.0%	0.4%	1665783	0.0194
bcspwr03	0.0%	26.6%	73.2%	0.2%	3060390	0.0649
bcspwr03	0.0%	42.0%	57.7%	0.4%	1863039	0.0355
lns__131	0.0%	19.6%	80.3%	0.2%	4718773	0.0864
lns__131	0.0%	32.8%	67.0%	0.3%	2707884	7.4147
west0132	0.0%	16.3%	83.6%	0.1%	7015999	177.5854
west0132	0.0%	28.3%	71.4%	0.2%	3837586	137.7349
west0156	0.0%	18.2%	81.7%	0.1%	10348027	12.5331
west0156	0.0%	30.9%	68.9%	0.2%	5876213	40.6595
nos1	0.0%	20.5%	79.4%	0.1%	9563662	0.0644
nos1	0.0%	34.0%	65.8%	0.2%	5516912	2.4655
saylr1	0.0%	21.3%	78.6%	0.1%	31523458	2.3642
saylr1	0.0%	35.1%	64.7%	0.2%	18560428	2.1231

TABLE 3 – nombre de clauses par type pour $M3_{bis}$ et $M3$

Analyse :
— $M3\_bis$ bénéficie d'une division de son nombre de clause total par un facteur $\tilde{2}$ par rapport à $M3$.
— Le gain se fait principalement sur le nombre de clause des contraintes de distances.
— L'économie du nombre de clause ne semble pas être corrélé à un gain de temps pour la résolution PySAT.

Hypothèses :
— Contrairement à notre hypothèse initiale, l'économie du nombre de clause ne semble pas entraîner une réduction du temps de résolution pour pySAT. Nous emettons une nouvelle hypothèse : cet exemple ne contient que des cas SAT ; avoir plus de clauses et plus de litéraux peut être avantageux en terme de temps de réponse quand la réponse est SAT, car la probabilité de proposer des certificats valident est plus élevé ? A l'inverse pour un problème UNSAT, dans le pire des cas on vérifie toutes les clauses (seule la dernière est fausse pour chaque certificat), ce qui devrait à priori avantager $M3\_bis$ le problème. Une analyse de l'algorithme de résolution de PySAT devrait nous éclairer sur cette question.
— Le gain de temps vient de la génération des clauses ?

6.2.2 Comparaison des temps de construction des clauses

Par soucis de lisibilité, vous trouverez ci-joint les temps de construction des clauses en pourcentages par rapport au temps total que prend la construction de toutes les clauses du modèle (indépendamment des temps de préparation ou préfiltrage)
— les lignes en bleu clair représentent les résultats pour le modèle $M3$
— les lignes en bleu foncé représentent les résultats pour le modèle $M3_{bis}$
— t total représente le temps total des calculs de t AL, t AM, t dist et t sym2 seulement.

— les modèles ayant un temps total dominant sont coloriées en vert
— les arrondis sur les temps entraînent des pourcentages peu précis. Pour consulter les temps on pourra lance la *commande utilisée*.
commande utilisée : `python3 Run_M_ann.py ann v` $[M3, M3\_bis]$ `200`

Instance	t AL (s)	t AM (s)	t dist (s)	t sym2 (s)	t total (s)
pores_1	0.1%	3.4%	81.4%	0.2%	0.2
pores_1	0.3%	11.9%	52.5%	0.3%	0.1
ibm32	0.2%	4.3%	84.8%	0.1%	0.2
ibm32	0.2%	14.3%	60.2%	0.3%	0.1
bcspwr01	0.3%	7.1%	84.2%	0.2%	0.2
bcspwr01	0.4%	33.9%	82.8%	1.0%	0.1
bcsstk01	0.2%	3.2%	94.8%	0.1%	1.1
bcsstk01	0.1%	15.3%	85.5%	0.2%	0.3
bcspwr02	0.2%	6.6%	96.7%	0.4%	0.6
bcspwr02	0.3%	26.3%	79.1%	0.5%	0.2
curtis54	0.1%	7.4%	94.5%	0.2%	1.1
curtis54	0.2%	25.2%	83.0%	0.7%	0.3
will57	0.1%	4.1%	93.2%	0.1%	1.2
will57	0.1%	19.8%	89.2%	0.3%	0.4
impcol_b	0.0%	2.4%	97.9%	0.0%	2.3
impcol_b	0.1%	11.5%	88.9%	0.1%	0.8
steam3	0.0%	2.3%	97.6%	0.0%	5.7
steam3	0.1%	10.1%	91.7%	0.1%	2.0
ash85	0.0%	5.9%	94.8%	0.1%	3.9
ash85	0.1%	17.8%	82.8%	0.1%	1.6
nos4	0.0%	6.5%	93.6%	0.0%	6.1
nos4	0.1%	16.7%	81.4%	0.2%	2.4
bcsstk22	0.0%	4.4%	95.7%	0.0%	7.3
bcsstk22	0.1%	18.5%	80.3%	0.1%	3.3
gre__115	0.0%	4.4%	95.0%	0.0%	8.2
gre__115	0.1%	16.1%	82.6%	0.1%	3.8
dwt__234	0.0%	7.4%	92.1%	0.1%	5.7
dwt__234	0.1%	22.7%	75.5%	0.1%	2.8
bcspwr03	0.0%	6.4%	94.2%	0.1%	6.0
bcspwr03	0.1%	23.5%	76.5%	0.1%	3.2
lns__131	0.0%	4.4%	95.4%	0.0%	10.4
lns__131	0.1%	21.5%	78.0%	0.1%	5.8
west0132	0.0%	3.4%	96.5%	0.0%	16.2
west0132	0.0%	15.5%	84.7%	0.1%	6.7
west0156	0.0%	4.4%	95.7%	0.0%	20.0
west0156	0.0%	20.1%	80.2%	0.1%	9.2
nos1	0.0%	5.2%	95.0%	0.0%	17.9
nos1	0.0%	21.1%	78.9%	0.1%	8.2
saylr1	0.0%	5.9%	94.0%	0.0%	57.5
saylr1	0.0%	22.4%	77.4%	0.1%	28.0

TABLE 4 – temps de construction des clauses pour $M3_{bis}$ et $M3$

Analyse :

— Comme attendu, le modèle $M3\_bis$ calcul ses clauses plus vite que $M3$ (facteur $\tilde{2}$) et ce temps gagner vient principalement des clauses dist.

Conclusion des comparaisons des modèles SAT :

— Réduire le nombre de clauses a bien un effet positif sur les temps de résolution.
— Pour réduire davantage ce dernier, on pourra chercher à filtrer davantage en utilisant d'autres propriétés des graphes, et améliorer les algorithmes de génération de clauses.
— Dans une future étude, il serait intéressant d'explorer la question "existe-t-il un solveur dédié à la résolution du Cyclic Bandwidth plus que rapide les solveurs traditionnels ACE, CHOCO etc."

7 Améliorations

7.1 Résolution adaptative

Nos expérimentations ont montré quel modèle dominait les autres selon les instances en entrée. Nous en avons tirer qu'en vérifiant au préalable les caractéristiques des instances à résoudre, nous pouvons décider quel modèle est le plus adapté pour le résoudre. Nous avons fixer empiriquement que :

— Les instances ayant un nombre d'arête $m < 50$ seront résolus avec M3.
— Les instances ayant un nombre d'arête $m >= 50$ seront résolus avec M2.

Dans l'idéal, l'utilisation d'une heuristique estimant la proximité potentielle de la CB optimale pourrait nous aider à changer de modèle au fur et à mesure que l'on s'approche de celle-ci. En effet, d'après nos analyses, si k est suffisamment éloigné de CB*, les modèles sous contraintes booléennes semblent plus efficaces.
Des idées de *relabelliser* les étiquetages en cours de route nous semblent également être intéressantes, d'après la littérature sur le sujet [4].

7.2 Réduction du temps de calcul de la CB optimale

Au moins deux actions peuvent réduire le temps de calcul de la CB optimale :

— Réduire le temps de décision
— Resserrer les bornes qui encadre la valeur optimale

1) Réduire le temps de décision :

La réduction du temps de calcul de `sat(k)` a le potentiel de gain le plus élevé car cette fonction est appelé $\log_2(n)$ fois par notre méthode dichotomique.
Utiliser des techniques de filtrage réduit considérablement la complexité. Si de plus, l'objectif du problème consiste seulement à rechercher la CB minimale, sans se soucier de conserver les arrangements d'étiquetages possibles non symétriques qui donne un même CB, alors on doit pouvoir réduire les temps de calcul pour M2 et M3. En effet la décision `UNSAT` apparaîtra plus rapidement car la résolution n'ira pas explorer des parties de l'arbre donnant un même CB.

2) Resserrage des bornes :

Une fois qu'on a un temps de calcul minimum pour sat(k), on peut chercher à trouver de meilleurs bornes pour CB*. Le resserrage des bornes peut s'effectuer en exploitant les propriétés mathématiques du problème [5] [6], et en faisant appelle aux méta-heuristiques [3].

Ces idées d'améliorations pourront être poussées plus loin dans un potentiel projet futur. (semestre prochain ? :))

8 Conclusion

Ce projet nous aura permit d'avoir un avant goût passionnant de la puissance et des possibilités offertes par la programmation par contraintes. Beaucoup de recherches, de réflexions, et d'intenses émotions ont été traversées pour aboutir à cette première étude. Le problème de recherche de Cyclic Bandwidth est passionnant, et recèle bien plus de propriétés qu'il n'y paraît au premier abord. Nous sommes désormais très curieux de poursuivre l'aventure. La prochaine étape pour nous serait d'utiliser des méta heuristiques pour la résolution, et optimiser nos algorithmes de constructions de clauses. Aussi comme vous nous l'avez suggérez lors de notre présentation, utiliser le paramétrage du solveur PyCSP3 pour y incruster nos filtrages est à essayer.

Références

[1] Christian Bessiere et al. « An optimal coarse-grained arc consistency algorithm ». In : *Artificial Intelligence* 165.2 (2005), p. 165-185.

[2] Rafael Marti. *Good solutions and lower bounds for this problem.* 2016. url : `https://www.uv.es/rmarti/paper/bmp.html` (visité le 10/04/2022).

[3] Eduardo Rodriguez-Tello et al. « Tabu search for the cyclic bandwidth problem ». In : *Computers & Operations Research* 57 (2015), p. 17-32.

[4] Ronan Hamon et al. « Relabelling vertices according to the network structure by minimizing the cyclic bandwidth sum ». In : *Journal of Complex Networks* 4.4 (2016), p. 534-560.

[5] Hugues Déprés, Guillaume Fertin et Eric Monfroy. « Improved Lower Bounds for the Cyclic Bandwidth Problem ». In : *International Conference on Computational Science.* Springer. 2021, p. 555-569.

[6] Sanming Zhou. « Bounding the bandwidths for graphs ». In : *Theoretical computer science* 249.2 (2000), p. 357-368.

Annexe

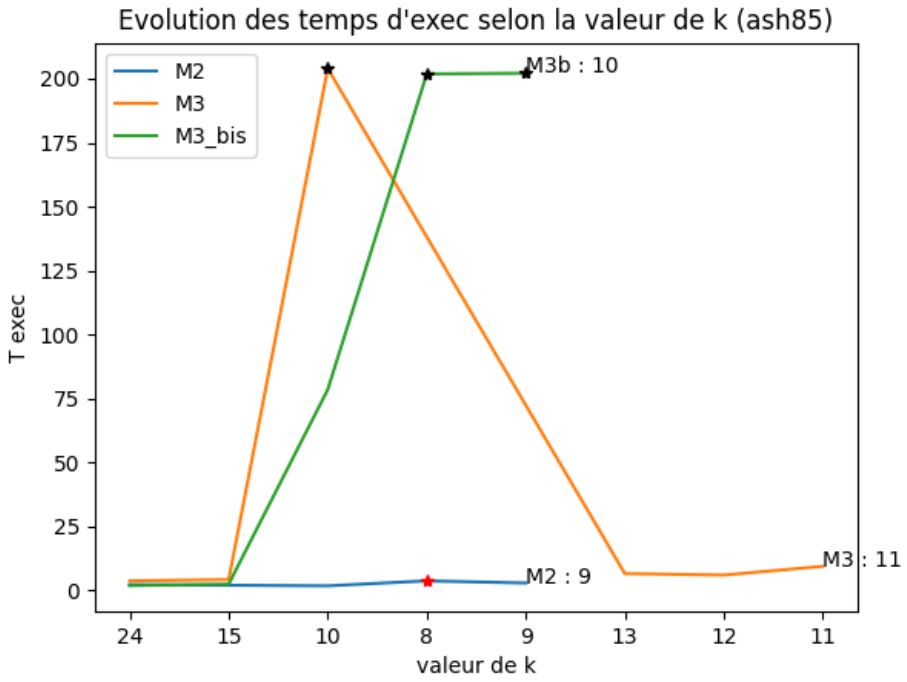

FIGURE 5 – ash85_M1_M1_bis_M2_M3_M3_bis

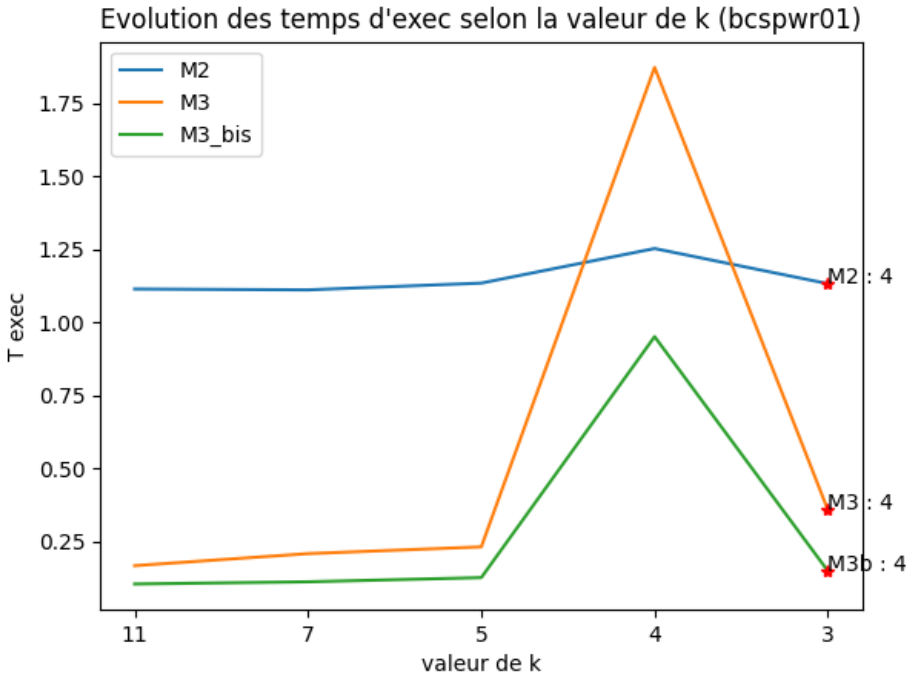

FIGURE 6 – bcspwr01_M1_M1_bis_M2_M3_M3_bis

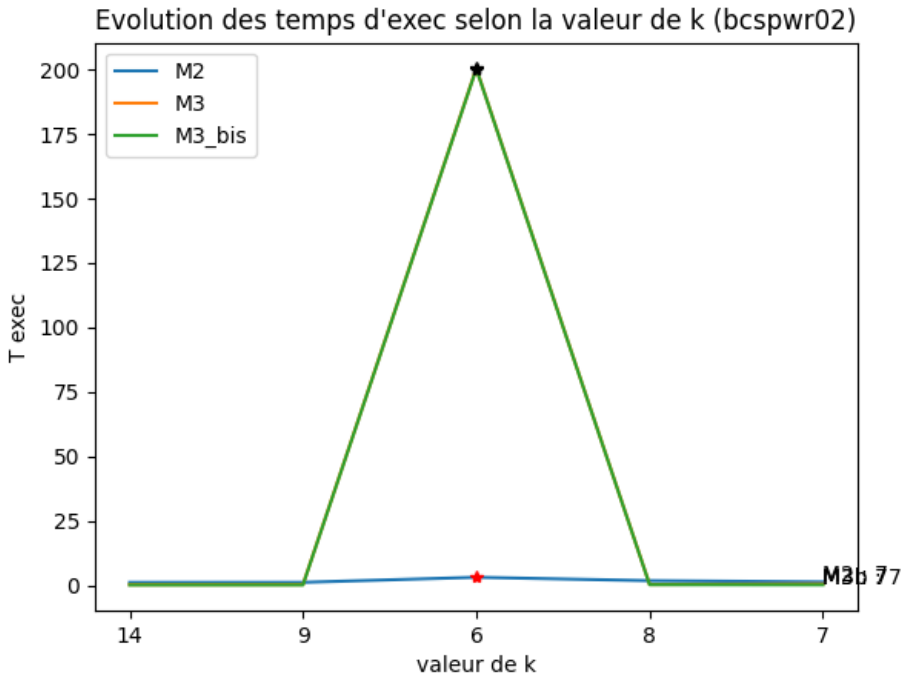

FIGURE 7 – bcspwr02_M1_M1_bis_M2_M3_M3_bis

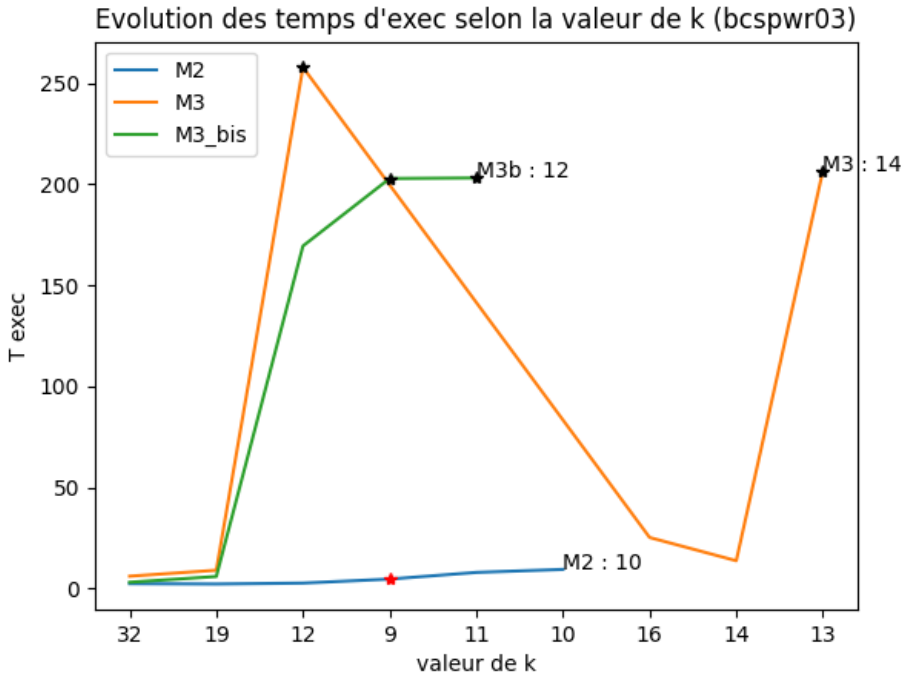

FIGURE 8 – bcspwr03_M1_M1_bis_M2_M3_M3_bis

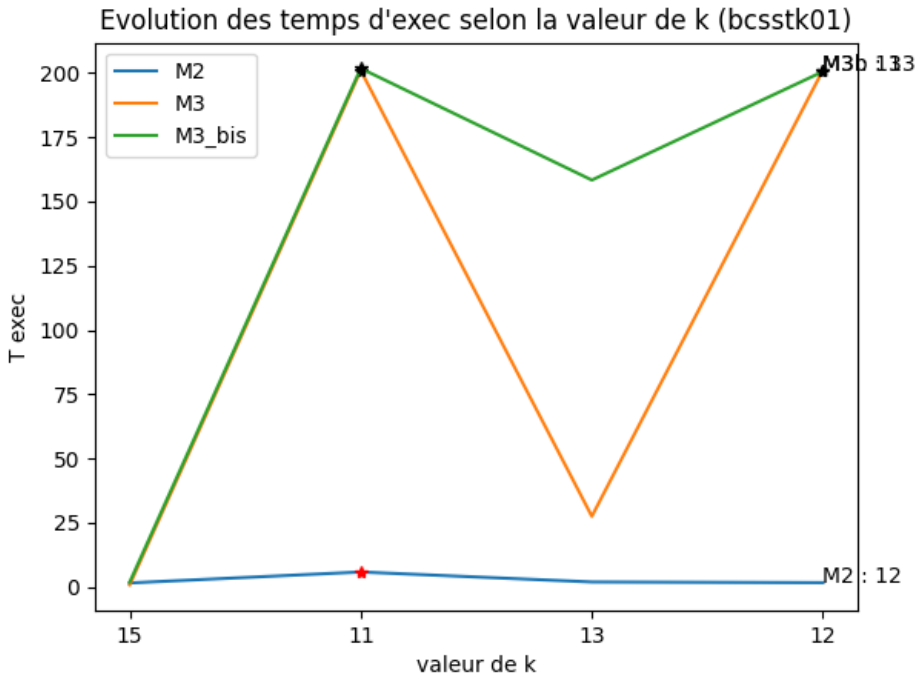

FIGURE 9 – bcsstk01_M1_M1_bis_M2_M3_M3_bis

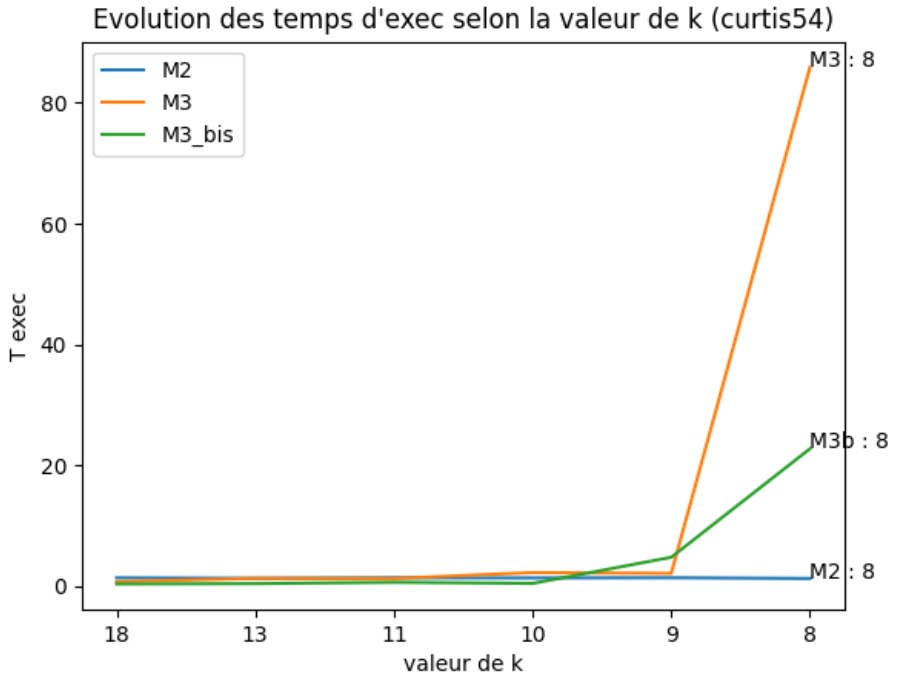

FIGURE 10 – curtis54_M1_M1_bis_M2_M3_M3_bis

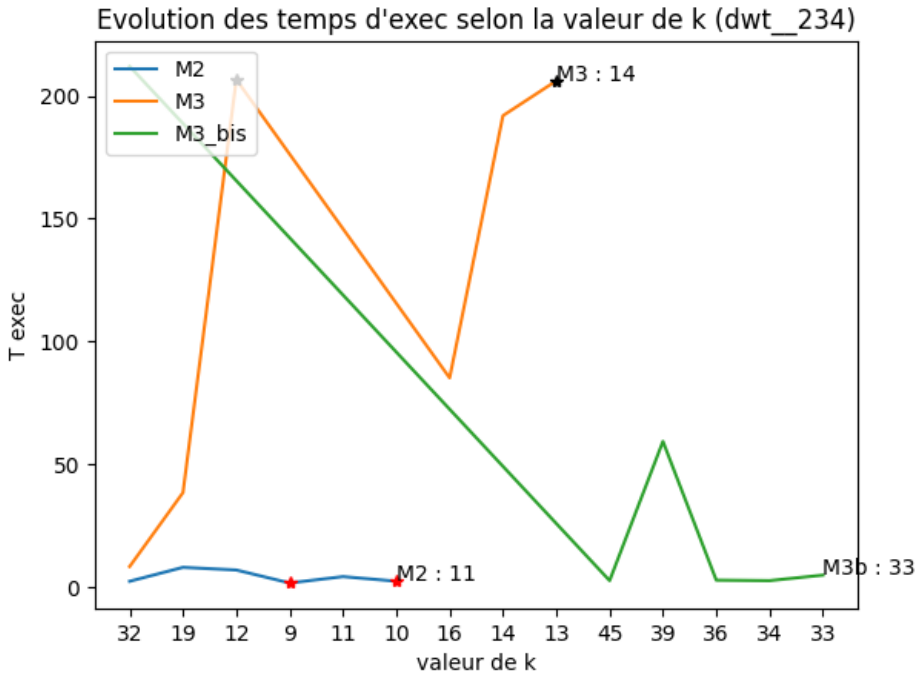

FIGURE 11 – dwt__234_M1_M1_bis_M2_M3_M3_bis

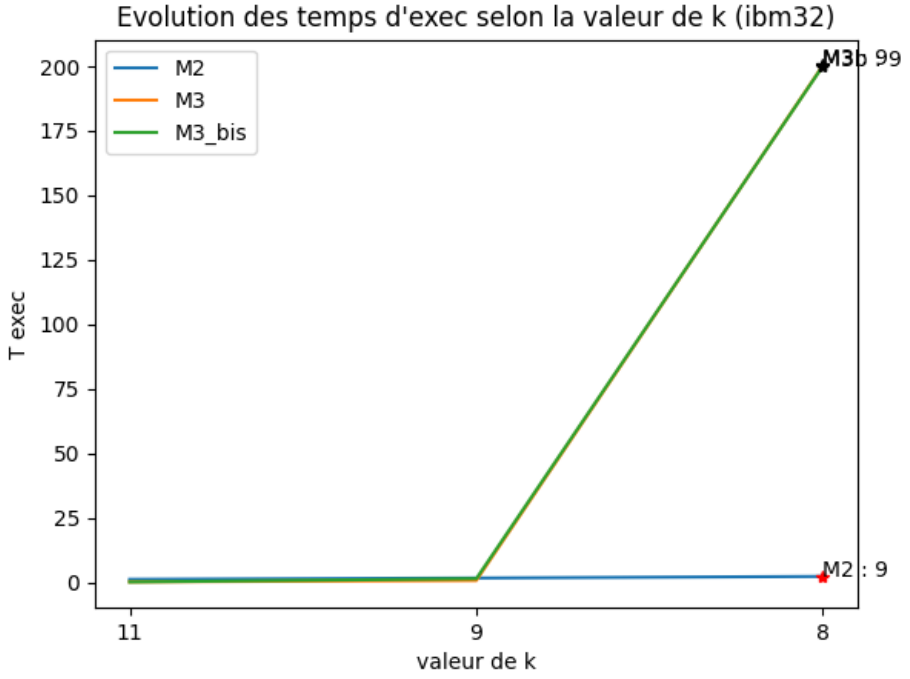

FIGURE 12 – ibm32_M1_M1_bis_M2_M3_M3_bis

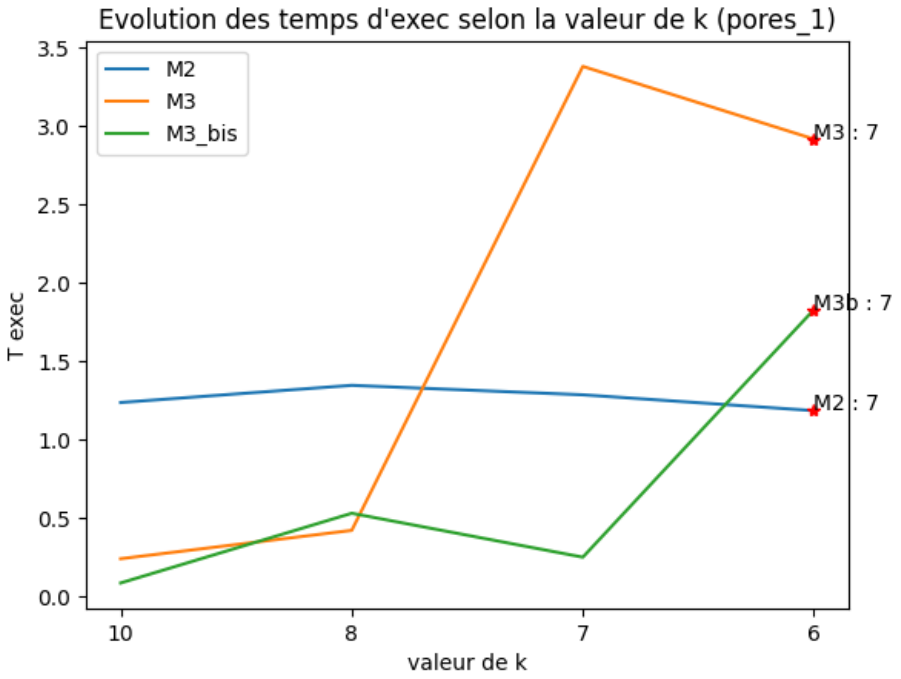

FIGURE 13 – pores_1_M1_M1_bis_M2_M3_M3_bis

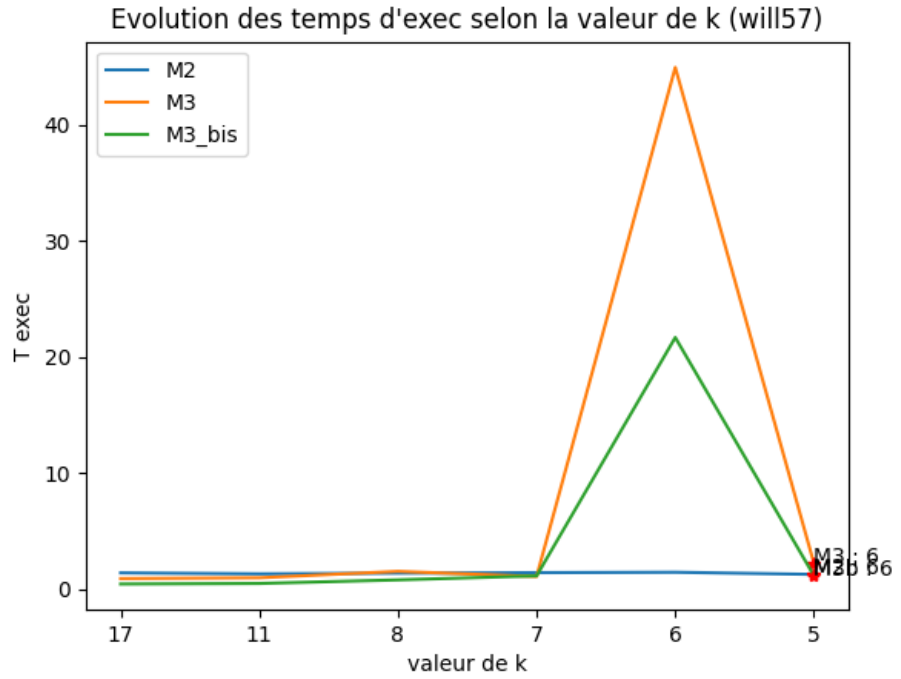

FIGURE 14 – will57_M1_M1_bis_M2_M3_M3_bis

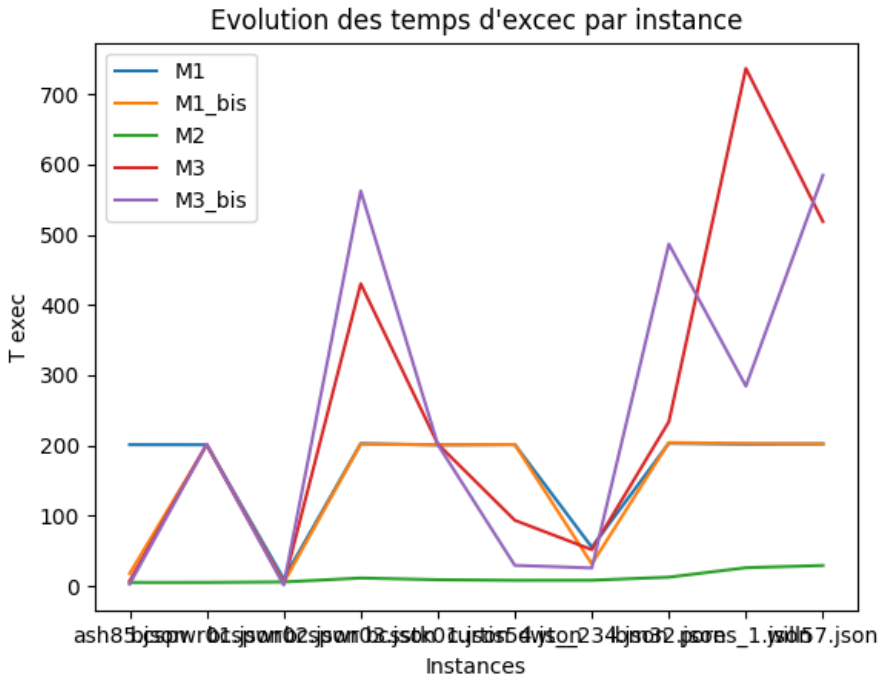

FIGURE 15 – M1_M1_bis_M2_M3_M3_bis

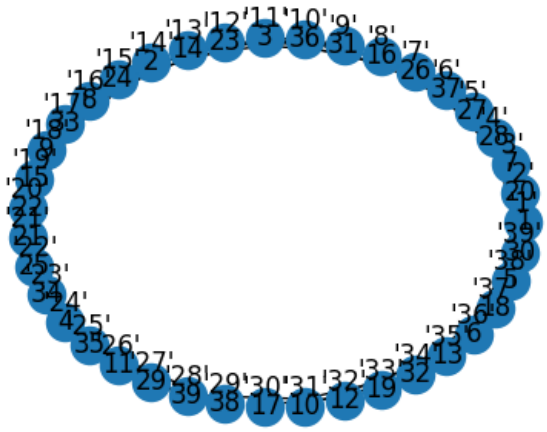

FIGURE 16 – graphique cyclique de bcspwr01 pour $k = 4$