Deux algorithmes pour l'identification d'un axe gauche-droite sur les candidats à une élection

P-ANDROIDE

Thibault Gigant
Laura Greige

Encadrant :
Olivier Spanjaard

Année 2015 - 2016

Table des matières

Introduction

Habituellement, lors d'un vote, l'électeur est amené à choisir un unique candidat parmi une multitude. Lorsqu'il est donné la possibilité aux électeurs de ne plus voter pour un seul candidat, mais pour un sous-ensemble d'entre eux qu'il approuverait, on parle de vote par « approbation », ou vote par « assentiment ». Avec cette procédure de vote, on peut étudier les différents votes formulés pour tenter d'en extraire un axe « gauche-droite » classant les candidats les uns par rapport aux autres en fonction de leur proximité dans les bulletins récoltés.

Afin de résoudre ce problème, deux principales méthodes seront utilisées dans ce projet :

— Un algorithme de sériation permettant de calculer un axe en fonction des similarités entre les candidats, calculées à partir des bulletins de vote.

— Un algorithme de branch & bound identifiant un sous-ensemble le plus large possible de bulletins cohérents avec un axe. Un bulletin est dit cohérent avec un axe s'il correspond à un sous-ensemble de candidats contigus sur cet axe.

Ces deux approches seront plus amplement détaillées dans la seconde partie de ce rapport. La première partie quant à elle détaillera les différentes tâches qui ont été demandées par l'encadrant. La troisième partie concernera l'implémentation en elle-même, le logiciel créé pour répondre à ces tâches. Enfin, une dernière partie donnera les résultats de tests numériques sur des données réelles.

1 Cahier des Charges

Deux approches ont donc été demandées, la première étant une approche par sériation et la seconde par branch & bound. De plus, une interface graphique pour lancer les algorithmes et visualiser les résultats a été requise.

1.1 Sériation

La première méthode qui sera utilisée appliquera un algorithme de sériation. Le principe est de calculer une matrice dont l'élément en position ij représente la dissimilarité entre le candidat i et le candidat j. L'indice de dissimilarité peut être calculé de plusieurs manières, et chacune sera étudiée et comparée aux autres, pour déterminer laquelle correspond le plus à la recherche de l'axe voulu. L'indice est un réel compris entre 0 et 1, tel que 0 représente une similarité parfaite, et 1 une dissimilarité totale. Bien entendu, la matrice de dissimilarité est parfaitement symétrique et a pour diagonale principale des 0. Elle peut donc être considérée comme une matrice triangulaire supérieure stricte, puisque la diagonale ne changera pas et la partie inférieure n'est que le miroir de la partie supérieure.

A partir de cette matrice, pour trouver un axe sur les candidats, il suffit alors de trouver une permutation des candidats (donc des lignes et des colonnes) telle que les éléments sur les lignes et les colonnes soient croissants au fur et à mesure que l'on s'éloigne de la diagonale. Du moins, on cherche à être le plus proche possible de cette propriété. On peut procéder de deux manières pour ce faire :

— **la force brute** considère toutes les permutations possibles, donc tous les axes possibles, calcule un score à partir de la matrice de dissimilarité de chaque permutation, et renvoie celle donnant le score minimal. Cette méthode naïve sera implémentée, mais étant donnée sa complexité exponentielle, puisqu'elle demande de calculer le score des $n!$ axes possibles (n étant le nombre de candidats), elle ne sera utilisée que pour vérifier la validité de la solution suivante, avec un nombre de candidats restreint.

— **la procédure de programmation dynamique** est inspirée des travaux de HUBERT [1]. Elle permet d'identifier une permutation optimale en $O(2^n)$, où 2^n est le nombre de sous-ensembles possibles de candidats. Cela reste conséquent, mais largement inférieur à $n!$.

1.2 Branch & bound

Pour déterminer un axe compatible avec un nombre maximum de votants, on fait appel à une procédure de branch & bound. Supposons qu'on ait n bulletins distincts. Un algorithme d'énumération complète des solutions peut être illustré par une arborescence de hauteur n, où en chaque nœud on décide de sélectionner ou non un bulletin. On obtient donc un arbre à 2^n feuilles, et en chacune de ces feuilles, on a une solution possible qui correspond à un sous-ensemble de bulletins. On cherche s'il existe un axe compatible avec l'ensemble des bulletins sélectionnés et on retient la meilleure solution obtenue, c'est-à-dire l'ensemble le plus large de bulletins cohérents avec un même axe.

Afin d'améliorer l'efficacité du branch & bound, seules les solutions potentiellement de bonne qualité seront énumérées, les solutions ne pouvant pas conduire à améliorer la solution courante ne seront pas explorées.

Le branch & bound est basé sur trois principes : le principe de séparation qui consiste à séparer un ensemble de solutions en sous-ensembles, le principe d'évaluation qui permet de connaître la qualité des nœuds à traiter, et le principe du parcours de l'arbre qui permet de choisir le prochain nœud à séparer.

1.3 Interface graphique

L'interface demandée est une interface simple, permettant d'abord de choisir l'algorithme à appliquer, avec quelles caractéristiques, et sur quels fichiers. Deux modes ont dû être implémentés. L'un permet de réaliser des bancs d'essai, donnant des résultats détaillés pour tous les fichiers sélectionnés. L'autre est plus interactif, et affichera pour chaque instance un graphe reflétant les différences, s'il y en a, entre le ou les axes trouvés par l'algorithme et l'axe « réel », récupéré manuellement depuis un site tel que Wikipedia.

2 Analyse et conception

Nous détaillons dans cette partie les deux algorithmes implémentés : la sériation et le branch & bound.

2.1 Sériation

Le premier algorithme à implémenter est un algorithme par sériation. La sériation est la capacité à comparer deux entités pour les ranger dans un ordre précis. Par exemple dire qu'un objet est plus grand qu'un autre ou inversement. Dans le cas du problème posé, le but est de positionner les candidats les uns par rapport aux autres sur un axe gauche-droite en fonction des bulletins exprimés. En effet, il paraît normal que deux candidats souvent présents dans un même bulletin ne soient pas trop éloignés l'un de l'autre dans l'axe.

2.1.1 Principe

La méthode emploie une matrice symbolisant la dissimilarité entre les candidats. Comme expliqué précédemment (1.1) on a donc une matrice carrée, où chaque ligne et chaque colonne représente un candidat. Le calcul de la dissimilarité peut se faire de différentes façons. Nous noterons, pour chaque

bulletin b_k, le nombre de personnes n_k ayant voté de cette manière. De plus, nous noterons $b_{i\wedge j}$ les bulletins contenant les candidats i et j, $b_{i\vee j}$ les bulletins contenant le candidat i ou le candidat j, et d_{ij} la dissimilarité entre ces deux candidats. Pour ce projet, 3 grandes fonctions ont été utilisées pour modéliser la dissimilarité entre deux candidats :

— En comptant le nombre de bulletins sur lesquels les deux candidats sont présents, divisé par le nombre de bulletins totaux. Pour obtenir un indice de dissimilarité, il suffit de retrancher ce résultat à 1 :

$$\forall i, j \text{ candidats}, d_{ij} = 1 - \frac{\sum\limits_{b_{i\wedge j}} n_{i\wedge j}}{\sum\limits_{b_k} n_k}$$

Nous appellerons cette fonction de dissimilarité `dissimilarity_and_n` par la suite.

— En comptant le nombre de bulletins sur lesquels les deux candidats sont présents, divisé par le nombre de bulletins où l'un ou l'autre des candidats est présent. De même, il suffit de retrancher ce résultat à 1 pour obtenir la dissimilarité entre les deux candidats :

$$\forall i, j \text{ candidats}, d_{ij} = 1 - \frac{\sum\limits_{b_{i\wedge j}} n_{i\wedge j}}{\sum\limits_{b_{i\vee j}} n_{i\vee j}}$$

Nous appellerons cette fonction de dissimilarité `dissimilarity_and_or` par la suite.

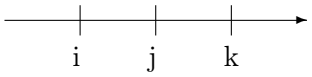
— Il est aussi possible de prendre en compte la taille des bulletins. En effet, il semble naturel de donner plus de poids à un bulletin ou seulement deux candidats sont présents, plutôt qu'à un bulletin où ces deux candidats sont noyés sous beaucoup d'autres. Ainsi, en s'inspirant de la formule précédente, et en notant $|b_k|$ le nombre de candidats approuvés dans le bulletin b_k :

$$\forall i, j \text{ candidats}, d_{ij} = 1 - \frac{\sum\limits_{b_{i\wedge j}} \frac{1}{|b_{i\wedge j}|} \times n_{i\wedge j}}{\sum\limits_{b_{i\vee j}} \frac{1}{|b_{i\vee j}|} \times n_{i\vee j}}$$

Nous appellerons cette fonction de dissimilarité `dissimilarity_over_over` par la suite.

Pour obtenir l'axe gauche-droite qui nous intéresse, il faut trouver une permutation sur les lignes et les colonnes (la même permutation sur les lignes et les colonnes), telle que les valeurs des dissimilarités soient croissantes lorsqu'on s'éloigne de la diagonale. On appelle une matrice respectant cette propriété une matrice « Anti-Robinson ».

On cherche à transformer la matrice ainsi pour la raison suivante. Prenons par exemple 3 candidats i, j, k, positionnés dans ce même ordre dans l'axe gauche-droite trouvé. La dissimilarité entre le candidat i et le candidat k, comme la matrice possède cette propriété, sera supérieure à celle entre les candidats i et j, ou celle entre les candidats j et k. Voici un petit schéma pour illustrer ces propos :



$$\iff \qquad d_{ik} \geq \max\{d_{ij}, d_{jk}\}$$

On aura donc ainsi trouvé l'axe qui positionne les candidats les plus similaires les uns à côté des autres.

En pratique, il n'est que rarement possible d'obtenir une matrice de dissimilarité parfaitement Anti-Robinson. C'est pourquoi on se ramène à un problème d'optimisation en définissant une « distance » à la propriété souhaitée. On attribue alors pour chaque permutation des lignes et colonnes un score à cette matrice. Dans ce projet, nous avons envisagé deux manières de calculer ce score.

Pour chacune, on regarde chaque élément d_{ij} $(j > i)$, de la partie supérieure droite de la matrice. Pour rappel, il est inutile de travailler sur la partie inférieure gauche, car elle est symétrique à la partie supérieure. On s'éloigne alors de la propriété voulue si il existe des éléments d_{ik} et d_{ij} $(k > j)$ sur la même ligne tels que $d_{ik} < d_{ij}$. De même, il faut regarder sur les colonnes s'il existe des éléments d_{kj} et d_{ij} $(k < i)$ tels que $d_{kj} < d_{ij}$. En notant n le nombre de candidats, les deux manières de calculer le score s d'une matrice de dissimilarité sont les suivantes :

— **Score non-pondérée :** on ajoute 1 au score à chaque fois qu'on s'éloigne de la propriété. On a donc :

$$s = \sum_{i=0}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} \mathbb{1}_{d_{ik}<d_{ij}} + \sum_{i=1}^{n} \sum_{j=0}^{i-1} \sum_{k=0}^{j-1} \mathbb{1}_{d_{kj}<d_{ij}}$$

— **Score pondéré :** cette fois, on prend en compte l'écart qu'il y a entre les éléments qui ne vérifient pas la propriété :

$$s = \sum_{i=0}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} [\mathbb{1}_{d_{ik}<d_{ij}} \times (d_{ij} - d_{ik})] + \sum_{i=1}^{n} \sum_{j=0}^{i-1} \sum_{k=0}^{j-1} [\mathbb{1}_{d_{kj}<d_{ij}} \times (d_{ij} - d_{kj})]$$

Plus le score est proche de 0, plus on est proche d'une matrice Anti-Robinson. On a alors une permutation donnant un axe gauche-droite compatible avec les bulletins exprimés. Bien entendu, lorsqu'on trouve une permutation donnant un score minimal, la permutation avec les candidats dans l'ordre inverse donnera un score identique. En effet, cette permutation inverse donnera la matrice transposée à la première, et donc les incohérences repérées dans le colonnes de l'une seront repérées dans les lignes de l'autre, et inversement. C'est pourquoi il y a toujours un nombre pair d'axes compatibles avec les données.

Rappelons que la force brute requerrait le calcul du score de $n!$ matrices, n étant le nombre de candidats du scrutin. Il faut de plus prendre en compte le temps nécessaire pour appliquer les permutations à la matrice.

Nous avons donc implémenté une procédure de programmation dynamique proposée par Hubert et Golledge [1]. Elle repose sur le postulat suivant : en partant de n candidats, si on arrive à classer de façon certaine $n-1$ candidats à gauche dans l'axe, il ne reste plus qu'à placer le dernier candidat à droite des autres. On réalise donc d'abord un appel récursif pour placer correctement les $n-1$ candidats, puis on place le dernier.

Pour expliquer cela plus clairement, on notera s la fonction donnant le score minimal d'un sous-ensemble de candidats, c'est à dire le score retourné par la fonction sériation correspondant à ce sous-ensemble. De même, on notera h la fonction donnant la distance entre un candidat c_i et un sous-ensemble. Avec C l'ensemble des n candidats, la relation de récurrence s'écrit :

$$s(C) = \min_{c_i \in C} [s(C \setminus c_i) + d(C \setminus c_i, c_i)]$$

2.1.2 Pseudo-Code

Voici ci-dessous le pseudo-code pour la programmation dynamique. L'algorithme prend en entrée la matrice de dissimilarité calculée de la manière souhaitée, l'ensemble entier des candidats, et un dic-

tionnaire vide qui sera rempli par l'algorithme :

Algorithme 1 : Sériation

Entrées :
Matrice de similarité *mat*
Ensemble des candidats impliqués *candidates_set*
Dictionnaire des résultats déjà calculés $function\_map$ dont les clés sont les sous-ensembles de candidats et les valeurs sont des tuples donnant le score et les permutations optimales associées.

Sorties :
Dictionnaire des résultats pour chaque sous-ensemble de candidats : $function\_map$

```
1  début
2      si candidates_set a déjà été calculé alors
3          retourner function_map
4      fin
5      n := cardinalité de candidates_set
6      si n = 1 alors
7          function_map[candidates_set] := (0, candidates_set)
8          retourner function_map
9      fin
10     temp := []
11     pour chaque combinaison de n-1 candidats dans candidates_set faire
12         candidat_courant := candidates_set \ combinaison
13         function_map := Sériation(mat, combinaison, function_map)
14         scores :=
           distance(similarity_matrix, combinaison, candidat_courant, function_map)
15         # La fonction distance renvoie une liste de tuples scores
16         pour chaque score dans scores faire
17             Ajout de (score[0] + function_map[combinaison][0], score[1]) à temp
18         fin
19     fin
20     On ne garde dans temp que les tuples avec le score minimum
21     score_min := score minimal dans temp
22     liste_combinaisons := liste des combinaisons donnant le score minimum dans temp
23     function_map[candidates_set] := (score_min, liste_combinaisons)
24     retourner function_map
25 fin
```

Dans cet algorithme, la fonction distance a été introduite. Cette fonction calcule la distance de la matrice au fait d'être Anti-Robinson, du moins en partie, en fonction du positionnement du candidat courant après les candidats présents dans *candidates_set*. En effet, une fois qu'on a positionné le candidat le plus à gauche, quelle que soit la permutation des candidats suivants, on peut déjà calculer l'impact de ce positionnement sur le score de la matrice. La colonne et la ligne de ce candidat étant positionnées de façon certaine, on peut regarder les éléments à droite de cette colonne et en haut de cette ligne qui ne permettraient pas à la matrice d'être Anti-Robinson. C'est donc bien le score de la matrice qui est calculé, mais uniquement par rapport à une ligne et une colonne qu'on suppose bien placée. Il faut donc réitérer le processus une fois qu'on aura placé le second candidat en rajoutant le score obtenu après placement du premier, puis le troisième, et ainsi de suite. Au final, on aura bien le

score final de la matrice, donnant son éloignement à la propriété recherchée.

Une fois l'algorithme de sériation appliqué, il ne reste plus qu'à retourner les axes les plus cohérents avec les bulletins, ainsi que le score de la matrice engendrée par ces axes. Ces deux sorties sont stockées dans le dictionnaire $function\_map$.

Voici un exemple trivial, à trois candidats C_1, C_2, C_3, décrivant comment trouver une permutation donnant un score minimal. On notera s la fonction donnant le score minimal d'un sous-ensemble de candidats, c'est à dire le score retourné par la fonction sériation correspondant à ce sous-ensemble. De même, on notera h la fonction donnant la distance entre un candidat et un sous-ensemble, distance calculée de façon pondérée pour cet exemple. Pour ce faire, nous donnerons une matrice de dissimilarité fictive simple, constituée d'entiers en lieu et place des flottants entre 0 et 1, pour faciliter les calculs :

$$\begin{array}{c} \\ C_1 \\ C_2 \\ C_3 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} C_1 & C_2 & C_3 \end{array} \\ \begin{pmatrix} 0 & 3 & 1 \\ 3 & 0 & 2 \\ 1 & 2 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \\ C_1 \\ C_3 \\ C_2 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} C_1 & C_3 & C_2 \end{array} \\ \begin{pmatrix} 0 & 1 & 3 \\ 1 & 0 & 2 \\ 3 & 2 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \\ C_2 \\ C_1 \\ C_3 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} C_2 & C_1 & C_3 \end{array} \\ \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 3 \\ 2 & 3 & 0 \end{pmatrix} \end{array}$$

$$\begin{array}{c} \\ C_2 \\ C_3 \\ C_1 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} C_2 & C_3 & C_1 \end{array} \\ \begin{pmatrix} 0 & 2 & 3 \\ 2 & 0 & 1 \\ 3 & 1 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \\ C_3 \\ C_1 \\ C_2 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} C_3 & C_1 & C_2 \end{array} \\ \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 3 \\ 2 & 3 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \\ C_3 \\ C_2 \\ C_1 \end{array}\!\!\begin{array}{c} \begin{array}{ccc} C_3 & C_2 & C_1 \end{array} \\ \begin{pmatrix} 0 & 2 & 1 \\ 2 & 0 & 3 \\ 1 & 3 & 0 \end{pmatrix} \end{array}$$

La recherche d'un axe gauche droite sur cet exemple donnerait :

$$\begin{aligned} s(\{C_1\}) &= 0 \\ s(\{C_2\}) &= 0 \\ s(\{C_3\}) &= 0 \end{aligned}$$

$$\begin{aligned} s(\{C_1, C_2\}) &= \min\{s(\{C_1\}) + d(\{C_1\}, C_2), s(\{C_2\}) + d(\{C_2\}, C_1)\} \\ &= \min\{0 + ((3-1) + (2-1)), 0 + (3-2)\} = 1 \end{aligned}$$

$$\begin{aligned} s(\{C_1, C_3\}) &= \min\{s(\{C_1\}) + d(\{C_1\}, C_3), s(\{C_3\}) + d(\{C_3\}, C_1)\} \\ &= \min\{0 + 0, 0 + (3-2)\} = 0 \end{aligned}$$

$$\begin{aligned} s(\{C_2, C_3\}) &= \min\{s(\{C_2\}) + d(\{C_2\}, C_3), s(\{C_3\}) + d(\{C_3\}, C_2)\} \\ &= \min\{0 + 0, 0 + ((2-1) + (3-1))\} = 0 \end{aligned}$$

$$\begin{aligned} s(\{C_1, C_2, C_3\}) &= \min\{s(\{C_1, C_2\}) + d(\{C_1, C_2\}, C_3), \\ &\qquad s(\{C_1, C_3\}) + d(\{C_1, C_3\}, C_2), \\ &\qquad s(\{C_2, C_3\}) + d(\{C_2, C_3\}, C_1)\} \\ &= \min\{1 + 0,\ 0 + 0,\ 0 + 0\} = 0 \end{aligned}$$

On peut alors retrouver les permutations optimales par induction arrière depuis le résultat donné par $s(\{C_1, C_2, C_3\})$. Ici il y a deux possibilités, donnant deux axes symétriques, ce qui est normal, comme expliqué précédemment. On a déterminé que les permutations optimales sont les suivantes : $C_1C_3C_2$ et $C_2C_3C_1$.

2.2 Branch & bound

Le deuxième algorithme implémenté est un algorithme par séparation et évaluation. Le Branch & bound est une méthode arborescente qui consiste à énumérer les solutions réalisables en éliminant les solutions partielles qui ne mènent pas à la solution que l'on recherche. En effet, on cherche un sous-ensemble de bulletins cohérents, il est donc nécessaire de formaliser cette cohérence avant de passer à l'algorithme de Branch & Bound.

2.2.1 Formalisation

Pour trouver un sous-ensemble S de bulletins cohérents, il suffit de trouver un axe compatible avec les bulletins présents dans S. En effet, s'il existe un tel axe, alors pour tout bulletin b appartenant à S, b figure dans l'axe et est un sous-ensemble contigu de l'axe. C'est la propriété des 1-consécutifs : soit une matrice binaire dont chaque ligne représente un bulletin et chaque colonne représente un candidat. La matrice M est alors composée de 0 et de 1, $m_{ij} = 1$ si le votant i approuve le candidat j, $m_{ij} = 0$ sinon. Si l'on trouve une permutation des colonnes telle que les 1 de la matrice sont consécutifs sur chaque ligne, alors on a trouvé un axe compatible.

Exemple : Prenons comme exemple un ensemble de bulletins $S = \{\{1,2,3\},\{1,3\},\{3,4\},\{1,3,4\}\}$ pour les candidats 1, 2, 3 et 4. On trouve la matrice suivante :

$$\begin{array}{c} \\ \{1,2,3\} \\ \{1,3\} \\ \{3,4\} \\ \{1,3,4\} \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \end{array} \Longrightarrow \begin{array}{c} \\ \{1,2,3\} \\ \{1,3\} \\ \{3,4\} \\ \{1,3,4\} \end{array} \begin{array}{c} \begin{array}{cccc} 2 & 1 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} \end{array}$$

Les 1 de la matrice de droite sont tous consécutifs sur chacune des lignes avec la permutation $p = [2,1,3,4]$, on a donc trouvé un axe compatible qui est $a = 2-1-3-4$. On peut donc dire que les bulletins sont cohérents.

D'autre part, pour tester si un bulletin est cohérent avec un sous-ensemble de bulletins, il faut vérifier qu'il est compatible avec au moins un des axes trouvés pour le sous-ensemble. Dans l'exemple précédent, un seul axe était compatible avec le sous-ensemble S, donc pour tester la cohérence d'un nouveau bulletin avec ce sous-ensemble, il suffit de tester la cohérence entre le nouveau bulletin et l'axe trouvé. Soit deux bulletins $b_1 = \{2,1\}$ et $b_2 = \{2,3\}$: b_1 est compatible avec l'axe a puisque 1 et 2 se trouvent côte à côte dans l'axe, alors que le bulletin b_2 ne l'est pas.

Prenons un autre exemple où plusieurs axes différents sont compatibles avec un même sous-ensemble de bulletins. Soit $S' = \{\{1,2,3\},\{2,3,4\}\}$ un sous-ensemble de bulletins. Les axes compatibles avec S' sont $a_1 = 1-2-3-4$ et $a_2 = 4-2-3-1$. Soit un bulletin $b_3 = \{1,3\}$: b_3 n'est pas cohérent avec a_1, mais il l'est avec a_2. Il est donc cohérent avec le sous-ensemble de bulletins S'.

On en déduit qu'il est bien nécéssaire d'énumérer tous les axes compatibles avec un sous-ensemble pour tester la cohérence d'un nouveau bulletin avec celui-ci.

2.2.2 Algorithme

On note $C = \{c_1, c_2, ..., c_m\}$ l'ensemble des m candidats impliqués et $L = \{(x_1, b_1), (x_2, b_2), ...(x_n, b_n)\}$ l'ensemble des n bulletins de vote uniques, avec x_i le nombre de fois que figure le bulletin b_i dans l'urne.

On définit un « sous-ensemble de bulletins compatibles » par l'existence d'un axe tel que pour tout bulletin $b \in$ sous-ensemble, b figure dans l'axe et est un sous-ensemble contigu de l'axe.

On cherche à identifier le plus large sous-ensemble de bulletins cohérents avec un axe. Un même bulletin peut figurer plusieurs fois dans l'urne, il est donc avantageux d'examiner les bulletins dans l'ordre décroissant de leur nombre de votants x_i. On sélectionne ainsi en priorité les bulletins les plus nombreux.

Avant d'ajouter un bulletin dans un sous-ensemble, il faut d'abord tester s'il est compatible avec les bulletins déjà sélectionnés figurant dans ce sous-ensemble. Il faut donc tester sa cohérence avec les axes trouvés. S'il est compatible, on l'ajoute au sous-ensemble en mettant à jour éventuellement la liste des axes compatibles et on continue le branchement. Sinon, on arrête l'exploration du sous-arbre. Pour améliorer l'efficacité du branch & bound, lorsqu'un seul et unique axe est trouvé à un nœud de l'arbre, on arrête l'exploration du sous-arbre et on insère directement dans le sous-ensemble les bulletins restants qui sont compatibles avec cet axe.

Le branch & bound est basé sur trois principes : le principe de séparation, le principe d'évaluation et le parcours de l'arbre.

Principe de séparation

Le principe de séparation consiste à diviser le problème en un certain nombre de sous-problèmes qui ont chacun leur ensemble de solutions réalisables. En résolvant tous les sous-problèmes et en prenant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial.

On branche de tel sorte qu'à chaque nœud on sélectionne (et on l'ajoute dans le sous-ensembles de bulletins sélectionnés) ou non un nouveau bulletin b_i de L. Ce principe de séparation est appliqué de manière récursive à chacun des sous-ensembles tant que celui-ci contient plusieurs solutions.

La procédure de séparation d'un ensemble s'arrête donc lorsqu'une des conditions suivantes est vérifiée :
– on connaît une solution meilleure que toutes celles de l'ensemble ;
– on sait que l'ensemble ne contient aucune solution admissible (cas où les bulletins sélectionnés ne sont pas cohérents).

Principe d'évaluation

L'exploration de l'arbre se fait bien sûr en un temps qui croît exponentiellement au pire cas avec le nombre n de bulletins uniques. Le principe d'évaluation a pour objectif de connaître la qualité des nœuds à traiter afin d'explorer le moins de nœuds possible. Pour cela, on introduit deux types de bornes : une borne inférieure de la fonction d'utilité du problème initial et une borne supérieure de la fonction d'utilité. La connaissance d'une borne inférieure du problème et d'une borne supérieure de la fonction d'utilité de chaque sous-problème permet d'arrêter l'exploration d'un sous-ensemble de solutions dont on peut prouver qu'il ne comporte pas de solution optimale.

Supposons qu'on soit à un noeud p de l'arbre. La borne inférieure de ce nœud est définie comme la somme du nombre de bulletins sélectionnés. Elle nous permet d'obtenir la valeur de la solution partielle du nœud p de l'arbre, notée $v(s_p)$. Soit x_i le nombre de fois que figure le bulletin b_i. La borne inférieure b_{inf} s'écrit :

$$b_{inf} = v(s_p) = \sum_{b_i \ sélectionné} x_i$$

Si la borne inférieure trouvée à une feuille de l'arbre (c'est-à-dire après avoir décidé de séléctionner ou non chacun des n bulletins) est supérieure à la valeur de la solution courante notée *best*, alors on a trouvé une meilleure solution.

La borne supérieure est définie comme la somme de la valeur de la solution partielle et du nombre de bulletins restant qui sont compatibles avec les bulletins déjà sélectionnés.

$$b_{sup} = v(s_p) + \sum_{b_i \in B} x_i$$

avec $B = \{$bulletins b restant : b compatible avec les bulletins de $s_p\}$.

Si cette borne supérieure est inférieure à la meilleure solution courante, il est inutile de continuer à explorer ce nœud puisque cette solution partielle ne pourrait pas conduire à améliorer la solution courante. La connaissance de ces deux bornes nous permet donc d'arrêter l'exploration d'un sous-ensemble de solutions avant d'atteindre les feuilles de l'arbre.

Parcours de l'arbre

Nous explorons en profondeur d'abord l'arborescence des sous-problèmes. Cela présente l'avantage de permettre d'obtenir rapidement des premières solutions réalisables, qui permettront ensuite d'élaguer la recherche.

2.2.3 Pseudo-Code

L'algorithme de branch & bound fait intervenir :

- la liste des candidats impliqués *candidates*
- la liste des bulletins uniques $preferences$: chaque bulletin est de la forme $(nb\_voters, ballot)$, nb_voters étant le nombre de fois que figure le bulletin *ballot*
- la fonction $find\_axes(node)$: fonction qui retourne tous les axes compatibles avec le sous-ensemble de bulletins contenus dans *node*
- la fonction $add\_coherent\_ballots(node, preferences)$: fonction qui insère directement dans *node* les bulletins restants compatibles
- les fonctions $borne\_inf(node)$ et $borne\_sup(node)$ calculent respectivement la borne inférieure et la borne supérieure d'un nœud *node*

Voici le pseudo-code correspondant à l'algorithme de branch & bound :

Algorithme 2 : branch_and_bound

Entrées :
Ensemble des bulletins de vote *preferences*
Ensemble des candidats impliqués *candidates*
Nœud courant *node*

Sorties :
Plus large sous-ensemble de bulletins cohérents *best*

```
1  début
2      si preferences n'est pas vide alors
3          (nb, ballot) := preferences[0]
4          preferences := preferences \ preferences[0]
5          nodeL := node + (nb, ballot)
6          nodeR := node

7          si find_ axes(nodeL) n'est pas vide alors
8              b_sup := borne_sup(nodeL) // calcul de la borne supérieure
9              si length(axes) = 2 alors
10                 nodeL := add_coherent_ballots(nodeL, preferences)
11             fin
12             si length(axes) > 2 alors
13                 v := borne_inf(nodeL) // calcul de la borne inférieure
14                 si v > best alors // best = meilleure solution courante
15                     best := v
16                 fin
17                 si b_sup > best alors
18                     branch_and_bound(nodeL)
19                 fin
20             fin
21         fin
22         branch_and_bound(nodeR)
23     fin
24     retourner best
25 fin
```

3 Logiciel

3.1 Choix du langage

Le problème des 1-consécutifs décrit précédemment (2.2.1) peut être résolu en modifiant un peu le problème. En effet, cela revient à regrouper tous les bulletins contenant chaque candidat c_i dans des ensembles différents et de trouver une permutation de ces ensembles telle que les bulletins soient contigus. Donc en trouvant une telle permutation, on aura résolu le problème des 1-consécutifs. Or pour représenter facilement et sur un espace mémoire réduit des permutations, il existe une structure appelée PQ-Tree. Cette structure arborescente peut représenter toutes les permutations possibles d'un ensemble. Chaque noeud intermédiaire est soit un noeud « P », soit un noeud « Q ». Un noeud P possède au moins 2 fils et signifie que l'on peut prendre toutes les permutations possibles de ses fils. Un noeud Q a au moins 3 fils et signifie que l'on peut prendre ses fils dans l'ordre indiqué ou dans le

sens contraire.

Prenons le PQ-Tree suivant :

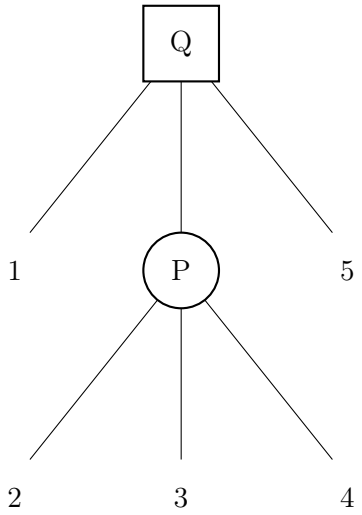


FIGURE 1 – Exemple de PQ-Tree

Celui-ci représente pour l'ensemble $\{1, 2, 3, 4, 5\}$ les permutations qui ont pour premier et dernier éléments 1 ou 5, et dont la partie centrale est l'ensemble des permutations des chiffres 2, 3 et 4. En clair, voici l'ensemble des permutations représentées par ce PQ-Tree :
12345, 12435, 13245, 13425, 14235, 14325, 52341, 52431, 53241, 53421, 54231, 54321.

Nous allons donc utiliser cette structure pour résoudre notre problème des 1-consécutifs. Pour trouver un axe compatible avec notre sous-ensemble de bulletins, nous allons utiliser la bibliothèque Sage, écrite en Python 2, qui possède des outils implémentant cette structure. Sage dispose également de fonctionnalités avancées comme la fonction *set_contiguous* qui sera utilisée dans le branch & bound pour trouver tous les axes compatibles. Si aucun axe compatible n'existe, *set_contiguous* lève une exception.

De plus, Python est un langage que nous utilisons souvent. Il permet d'écrire nos algorithmes assez rapidement et en un nombre de lignes réduit, tout en utilisant des fonctionnalités avancées offertes par le langage ou disponibles dans les nombreuses bibliothèques à disposition.

3.2 Installation de Sage et lancement

Malheureusement, il n'est pas toujours aisé d'installer cette bibliothèque qu'est Sage. Pour cela, nous vous recommandons d'aller dans la partie « Download » du site officiel de Sage : http://www.sagemath.org/. Normalement le site détecte de lui-même votre système d'exploitation, et vous demande de choisir un serveur proche de votre localisation pour télécharger les binaires qui correspondent à votre machine. Ensuite, nous vous conseillons de suivre les instructions à cette adresse pour être sûr d'avoir une installation parfaitement fonctionnelle : http://doc.sagemath.org/html/en/installation/binary.html.

Après l'installation, pour vérifier que tout s'est bien déroulé, lancez dans votre terminal la commande suivante :

```
sage -python
```

Si vous obtenez un interpréteur Python (2.7.10 normalement), tout s'est bien passé, et vous pourrez lancer l'interface que nous avons créée depuis le dossier principal de notre projet simplement en lançant la commande :

```
sage -python run.py
```

3.3 Interface

L'interface permet de lancer les algorithmes branch & bound et la sériation sur plusieurs fichiers. Ceux-ci ont été récupérés sur le site PrefLib. Nous avons traité les données issues d'élections effectuées en Irlande en 2002, ce sont les fichiers ayant un nom de cette forme : `ED-00001-0000000X.toc` (où X est un chiffre de 1 à 3). D'autres données proviennent d'élections menées à Glasgow en 2007, ce sont les fichiers ayant un nom de la forme : `ED-00008-000000XX.toc` (où XX représente les nombres de 01 à 21). Enfin, des données provenant de l'élection présidentielle française de 2002 ont aussi été utilisées. Ce sont les fichiers ayant un nom de la forme `ED-00026-0000000X.toc` (où X est un chiffre entre 1 et 6). Il suffit de cocher les fichiers sur lesquels on veut lancer les algorithmes directement depuis l'interface. Voici donc l'affichage de base, au lancement du programme :

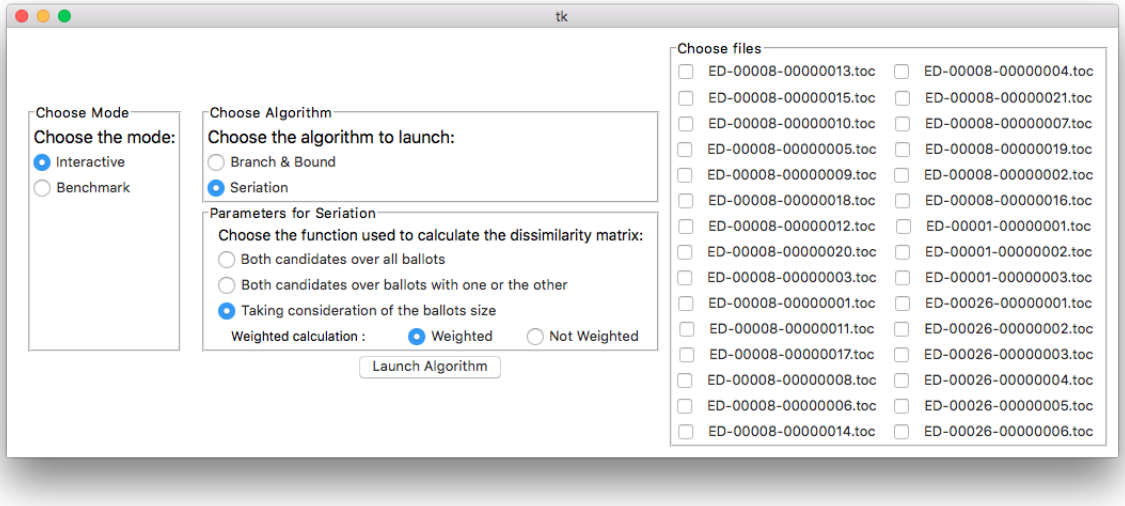


FIGURE 2 – Affichage à l'ouverture du programme

Elle comporte deux modes : un mode interactif et un mode « benchmark ». Quel que soit le mode choisi, on a toujours le choix entre les deux algorithmes présentés précédemment. Pour la sériation, un panneau apparaît pour donner le choix entre les 3 fonctions implémentées pour modéliser la dissimilarité entre deux candidats, ainsi que celui du calcul pondéré ou non de la distance à une matrice Anti-Robinson. Pour l'algorithme de branch & bound, une case apparaît pour donner le choix d'écarter les bulletins que l'on juge non pertinents car ils n'ont recueilli qu'un seul suffrage. Si cette case est cochée, on ne prendra pas en compte les bulletins ayant un nombre de votants égal à 1, et on appliquera l'algorithme sur le reste des bulletins. Enfin, le bouton « Launch Algorithm » permet le lancement de l'algorithme choisi sur les fichiers choisis.

Dans les deux modes, on choisit toutes les instances sur lesquelles on veut appliquer l'algorithme voulu dans la partie droite de la fenêtre. Cependant l'affichage qui en découle est différent.

Dans le mode « interactif », des graphes simples sont affichés, dans le but de comparer les axes retournés par l'algorithme lancé avec les axes réels. Ils vont permettre de visualiser les erreurs de positionnement rapidement. Il faut tout d'abord récupérer l'axe réel d'une manière ou d'une autre, ici nous avons récupéré les données sur Wikipedia. Chaque candidat est assimilé à son groupe d'appartenance : extrême gauche, gauche, centre, droite, et extrême droite. En numérotant ces groupes de gauche à

droite de 1 à 5, le graphe qui nous permettra de comparer les axes fonctionne comme suit. Il suffit de positionner les candidats sur l'axe des abscisses, et le point correspondant sera l'identifiant du groupe auquel il appartient réellement. Ainsi, si la courbe obtenue est monotone (croissante ou décroissante), l'algorithme aura retourné un axe qui correspond parfaitement à l'axe réel.

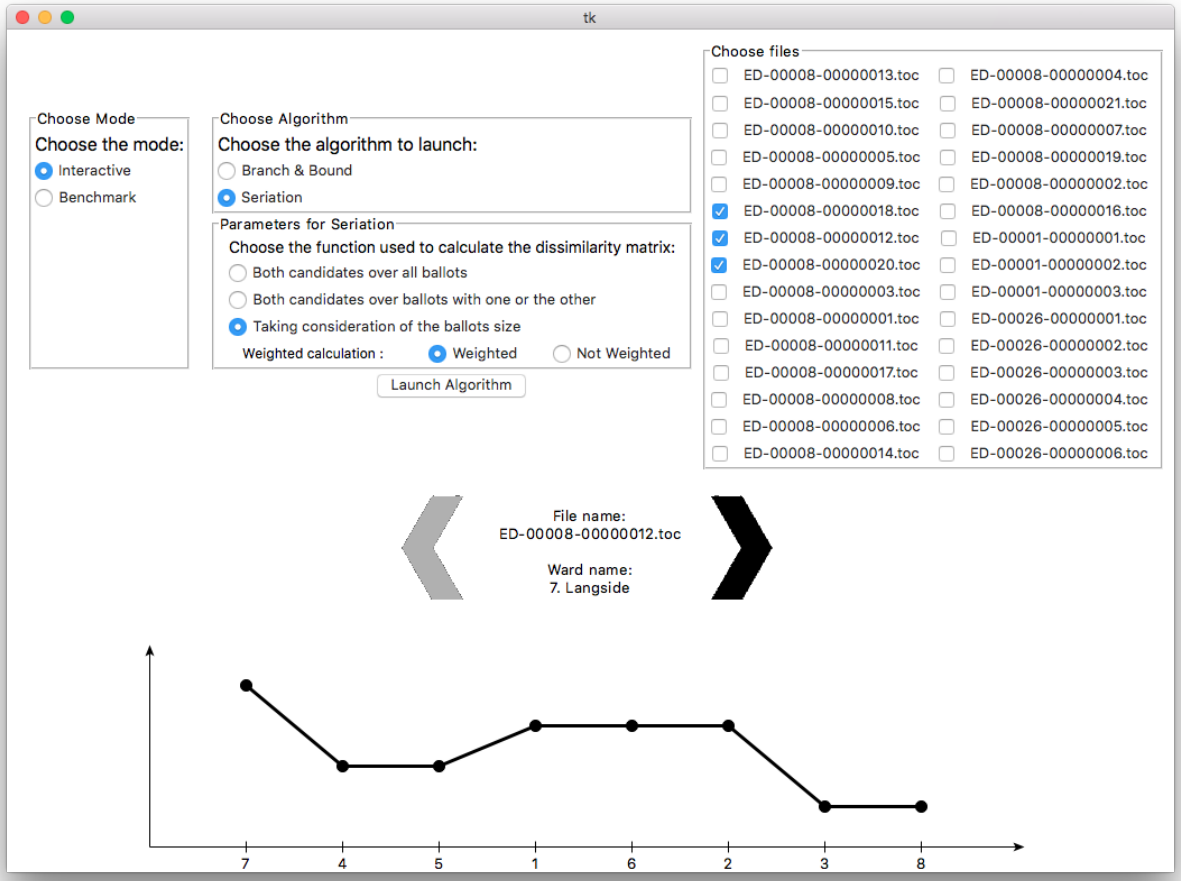
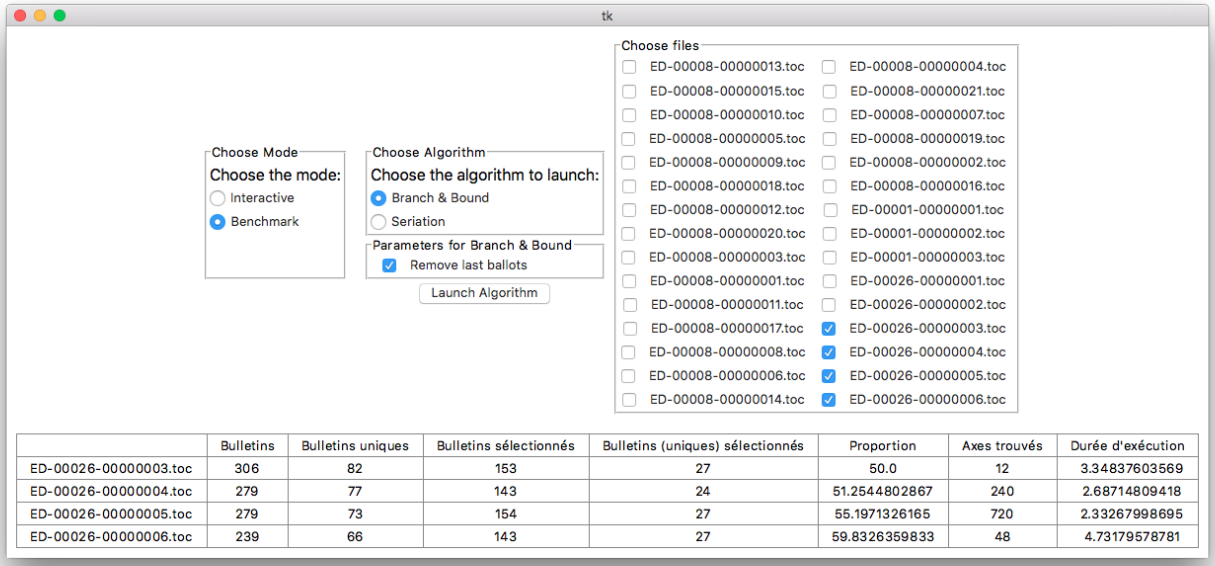


FIGURE 3 – Mode « interactif » du programme

D'autre part dans le mode « benchmark », on peut choisir plusieurs fichier sur lesquels lancer l'algorithme choisi. Une fois que l'exécution est terminée sur toutes les instances, un tableau affiche les donneées (nombre total de bulletins et bulletins uniques dans le fichier), les résultats (le nombre d'axes retournés par l'algorithme, ainsi que le nombre total de bulletins sélectionnés et le nombre de bulletins uniques sélectionnés par l'algorithme de branch & bound) et enfin le temps d'exécution.



	Bulletins	Bulletins uniques	Bulletins sélectionnés	Bulletins (uniques) sélectionnés	Proportion	Axes trouvés	Durée d'exécution
ED-00026-00000003.toc	306	82	153	27	50.0	12	3.34837603569
ED-00026-00000004.toc	279	77	143	24	51.2544802867	240	2.68714809418
ED-00026-00000005.toc	279	73	154	27	55.1971326165	720	2.33267998695
ED-00026-00000006.toc	239	66	143	27	59.8326359833	48	4.73179578781

FIGURE 4 – Mode « benchmark » du programme

Malheureusement, lorsque l'on clique sur le bouton « Launch Algorithm », comme les algorithmes peuvent prendre un temps assez considérable, toute l'interface se bloque le temps de la résolution. Nous n'avons malheureusement pas eu le temps d'implémenter un système permettant à l'utilisateur de connaître l'avancement des calculs, et de ne pas bloquer entièrement la fenêtre. C'est pourquoi il n'est pas recommandé de lancer le programme sur de nombreux fichiers d'un seul coup.

3.4 GitHub

Pour gérer le développement de ce projet en groupe, nous avons utilisé le gestionnaire de version `git`, avec un dépôt distant sur GitHub. Ce gestionnaire de version est vraiment très pratique pour la gestion des éventuels conflits. De plus, il est très souvent nativement intégré dans les IDE modernes. Enfin, grâce à GitHub, il est très facile de partager le code de notre projet, pour que quiconque y ait accès et puisse suggérer des améliorations.

Notre projet est disponible à cette adresse : https://github.com/lauragreige/P-ANDROIDE.

4 Expérimentation

Comme expliqué précédemment, les données utilisées pour réaliser nos tests sont issues du site PrefLib. Dans cette section, nous allons donner et interpréter certains résultats obtenus avec les différents algorithmes implémentés.

4.1 Sériation

Rappelons que la procédure de programmation dynamique est exponentielle en fonction du nombre de candidats qui se présentent à l'élection. En effet, on effectue un appel à cette procédure sur chaque sous-ensemble possible de l'ensemble total de candidat, c'est-à-dire le cardinal de l'ensemble des parties : 2^n, avec n le cardinal de l'ensemble des candidats. Les différentes circonscriptions et les différentes

élections n'ayant pas le même nombre de candidats, ces données vont nous permettre de comparer le temps d'exécution des algorithmes en fonction de ce critère. Voici l'évolution exponentielle du temps de calcul en fonction du nombre de candidats sur la figure suivante :

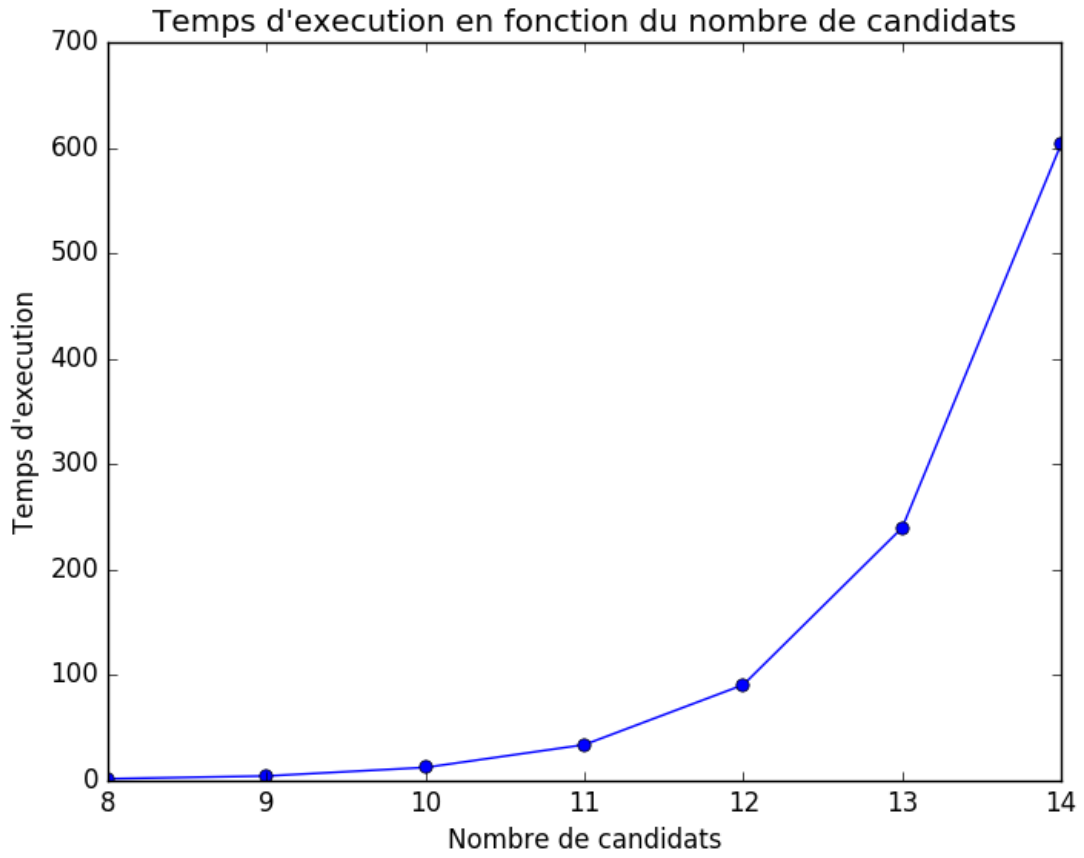


FIGURE 5 – Evolution du temps d'exécution en fonction du nombre de candidats

Le temps d'exécution de l'algorithme de programmation dynamique avec 14 candidats approche les 10 minutes. Au regard de cette évolution très exponentielle, il est donc recommandé de ne pas lancer cet algorithme sur plus d'une quinzaine de candidats. C'est pourquoi, dans ce projet, lorsque nous avons lancé cet algorithme sur les données de l'élection présidentielle française de 2002, y compris depuis l'interface, nous avons retiré les 4 candidats ayant recueilli le moins de suffrage : Corinne Lepage, Daniel Gluckstein, Christiane Taubira, et Christine Boutin. De cette façon, on passe à un problème à 12 candidats, qui se résout en un temps limité.

De plus, le calcul de la matrice de similarité est assez rapide comparé à cette procédure, il est en $O(n^2 \times m)$ où m est le nombre de bulletins uniques.

Comparons maintenant les axes retournés par cette procédure avec les axes réels. Pour cela, nous allons utiliser les graphes décrits dans la section à propos de l'interface (3.3). La monotonie de ces graphes nous informera sur la justesse de notre algorithme.

On peut donc, grâce à ces graphes, comparer les résultats obtenus en fonction de la façon dont on mesure la dissimilarité entre les candidats. Les trois graphes suivants sont les résultats obtenus avec les données issues de l'élection présidentielle française de 2002. Ainsi, les résultats seront encore plus parlants. Ces données recueillies dans 6 bureaux de vote ont été fusionnées pour obtenir des données

moins biaisées. Cela dit, toutes ces données proviennent d'une même zone géographique, donc un manque de brassage social peut malheureusement venir biaiser malgré tout nos données. Les graphes décrivant les résultats en fonction des mesures de dissimilarité sont en annexe A.1.

On remarque que globalement, quelle que soit la méthode pour mesurer la dissimilarité, les axes trouvés ne sont pas si éloignés de l'axe réel. En effet, à une ou deux inversions près, les courbes sont monotones. Cela s'observe pour toutes les données utilisées lors de ce projet : les données issues des élections irlandaises et de Glasgow. Il y a la plupart du temps de rares inversions dans l'axe, mais les groupes politiques sont le plus souvent assez bien trouvés par l'algorithme. Il arrive même que ce ne soient pas des inversions entre candidats qui soient repérées mais des inversions entre deux groupes entiers. Cela montre bien que la procédure de programmation dynamique implémentée repère bien les candidats qui sont souvent sur le même bulletin. En revanche, entre groupes distincts, comme les votes regroupent rarement des personnes d'affiliation politique très différentes, il est plus difficile de placer ces groupes les uns par rapport aux autres.

De plus, il est intéressant de noter que le plus souvent, c'est en ne pondérant pas le calcul de la distance à une matrice Anti-Robinson qu'on obtient le plus d'axes. En effet, cela est dû au fait qu'en ne pondérant pas nos calculs, il est plus probable que deux matrices donnent le même score final. En revanche si on prend en compte les écarts entre les éléments, donc en pondérant nos calculs, il est fort probable que ces deux matrices aient un score différent l'une de l'autre. Par exemple, pour les données de l'élection présidentielle française de 2002, lors de l'utilisation de la méthode de calcul de dissimilarité `dissimilarity_and_or`, la version pondérée n'a renvoyé qu'un seul axe, alors que la version non-pondérée en a renvoyé 4.

4.2 Branch & bound

L'algorithme de branch & bound a été appliqué sur les données provenant de l'élection présidentielle française de 2002 seulement (ED-000026-000000XX.toc). En effet, dans le branch & bound on explore tous les nœuds de l'arbre dans le pire des cas, c'est-à-dire $\sum_{i=1}^{n} 2^i$ nœuds, n étant le nombre de bulletins uniques. Les données provenant des élections menées en Irlande et à Glasgow contenaient un grand nombre de bulletins uniques (plus de 19000 bulletins uniques pour l'Irlande et plus de 2000 pour Glasgow), il était donc impossible d'appliquer l'algorithme de branch & bound sur ces données.

Dans la suite, on représente chacun des 16 candidats des élections présidentielles françaises par un numéro différent : 1 : Megret, 2 : Lepage, 3 : Gluckstein, 4 : Bayrou, 5 : Chirac, 6 : Le Pen, 7 : Taubira, 8 : Saint-Josse, 9 : Mamere, 10 : Jospin, 11 : Boutin, 12 : Hue, 13 : Chevenement, 14 : Madelin, 15 : Laguillier, 16 : Besancenot.

Une fois l'algorithme appliqué sur les données françaises, on affiche les résultats dans le tableau suivant.

Fichier	Bulletins (dont uniques)	Sélectionnés	Proportion (%)	Axes trouvés	Durée (en s.)
03.toc	476 (252)	171 (45)	35.9	2	115.1
04.toc	460 (258)	162 (43)	35.2	1	145.9
05.toc	472 (266)	174 (47)	36.8	2	908.2
06.toc	415 (242)	161 (45)	38.8	1	279.5

FIGURE 6 – Tableau représentant les résultats de l'algorithme de branch & bound sur les données françaises

La première colonne indique le nombre total de bulletins ainsi que le nombre de bulletins uniques (inscrit entre parenthèses), la deuxième colonne indique le nombre de bulletins sélectionnés ainsi que

le nombre de bulletins uniques sélectionnés (inscrit également entre parenthèses). La troisième colonne indique la proportion de bulletins sélectionnés par rapport au nombre total de bulletins et enfin les quatrième et cinquième colonne indiquent respectivement le nombre d'axes trouvés, cohérents avec les bulletins sélectionnés, et la durée d'exécution de l'algorithme.

Pour les instances ED-000026-00000001.toc et ED-000026-00000002.toc, le temps d'exécution était trop important par rapport aux autres instances. Nous avons donc implémenté une fonction qui permet d'écarter les bulletins que l'on juge non pertinents (dont le nombre de votants est égal à 1). Ceci nous permet de réduire la taille de l'instance en gardant les bulletins important, afin de diminuer le temps d'exécution. En appliquant cette méthode sur toutes les instances, on obtient les résultats suivants :

Fichier	Bulletins (dont uniques)	Sélectionnés	Proportion (%)	Axes trouvés	Durée (en s.)
01.toc	204 (55)	129 (25)	63.2	24	27.5
02.toc	234 (65)	127 (22)	54.3	720	229.9
03.toc	306 (82)	153 (27)	50.0	12	3.4
04.toc	279 (77)	143 (24)	51.3	240	2.7
05.toc	279 (73)	154 (27)	55.2	720	2.3
06.toc	239 (66)	143 (27)	59.9	48	4.8

FIGURE 7 – Tableau représentant les résultats de l'algorithme de branch & bound sur les données françaises en écartant les derniers bulletins

Malheureusement, en écartant les bulletins, on trouve un grand nombre d'axes correspondant au plus large sous-ensemble de bulletins cohérents. Par exemple, avec les données du fichier ED-000026-00000001.toc, on trouve 24 axes possibles : ceci est dû au fait que certains candidats ne figurent pas dans les bulletins non écartés. En effet, si on compare les 24 axes retournés par l'algorithme, on trouve toujours une même séquence de candidats dans les axes, et les candidats ne figurant pas dans les bulletins non écartés se retrouvent de part et d'autre de cette séquence :

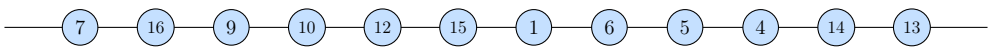


FIGURE 8 – Séquence de l'axe

Les candidats Lepage (2), Gluckstein (3), Saint-Josse (8), Boutin (11) ne figurant pas dans les bulletins ayant servi à construire cet axe, on trouve toutes les permutations possibles de ces candidats de part et d'autre de l'axe. Un exemple d'axe complet obtenu en écartant les bulletins de cette instance :

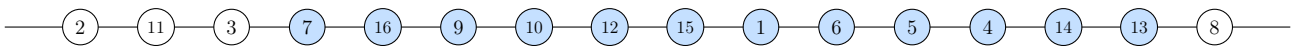


FIGURE 9 – Axe complet

On remarque que lorsqu'on écarte les bulletins, l'axe qu'on obtient n'est pas un axe gauche-droite. En effet, on retrouve Megret (1) et Le Pen (6) vers le centre de l'axe, alors qu'ils font tous les deux partie de l'extrême droite. De plus, si on regarde de plus près les résultats des élections présidentielle de 2002 au 1er tour, on remarque que Chirac (5) et Le Pen (6) figurent tous les deux sur un même bulletin plusieurs fois, ce qui explique le fait qu'ils soient à côté dans les axes.

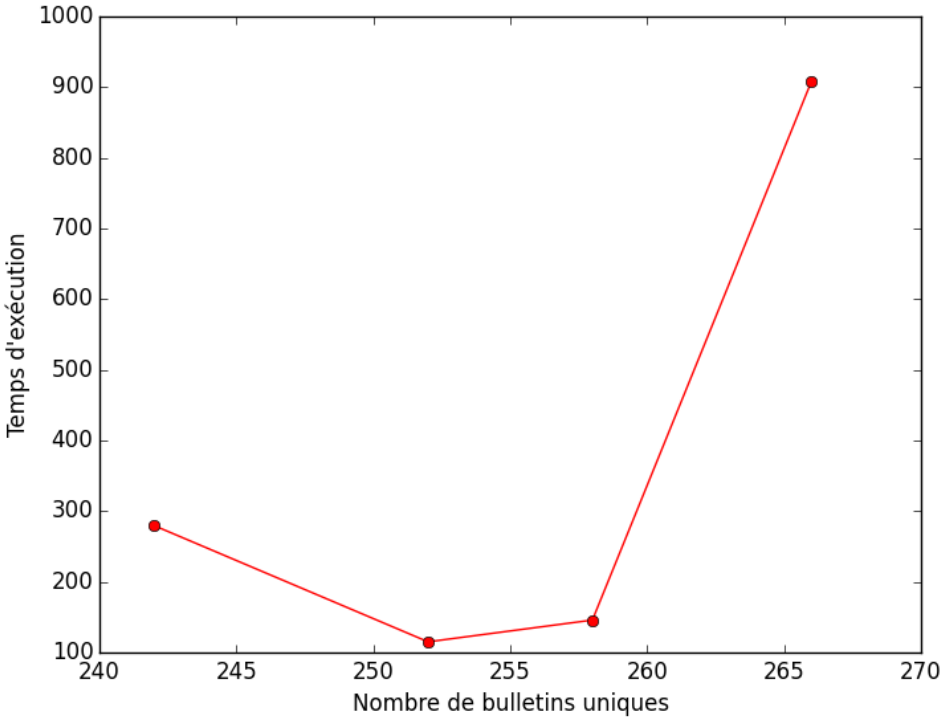


FIGURE 10 – Évolution du temps d'exécution du branche & bound en fonction du nombre de bulletins uniques

Le nombre de nœuds de l'arbre exhaustif des solutions possibles croit très vite lorsque le nombre de bulletins uniques augmente et avec nos tests, on remarque bien que la performance de notre algorithme diffère selon l'instance. La méthode de branch & bound n'est qu'une construction et un parcours astucieux de cet arbre, qui permet de ne pas explorer certaines branches grâce à un système de minoration et majoration (les bornes inférieures et supérieures) des solutions que cette branche peut produire.

Sur certaines instances le temps d'exécution dépendait plus de la position de la solution optimale dans l'arbre que du nombre de bulletins uniques. En effet, si une solution se trouvait dans la première branche d'un arbre de grande taille, on pourrait obtenir un temps d'exécution qui est inférieur au temps d'exécution d'un arbre de petite taille, dont la solution optimale se trouvait dans une de ses dernières branches. Ceci entraîne donc le parcours de plus de branches de l'arbre, ce qui est tout simplement dû au fait que l'on n'ait pas de « chance » sur l'ordre de parcours, et par conséquent, notre algorithme de branch & bound explorera un très grand ou très petit nombre de nœuds avant de renvoyer la solution optimale quel que soit le nombre de bulletins uniques figurant dans l'instance.

Dans la suite, nous étudierons les axes retournés par l'algorithme de branch & bound sur les instances sans écarter les bulletins. Prenons comme exemple l'axe correspondant aux données du fichier ED-000026-00000003.toc, et comparons celui-ci à l'axe réel.

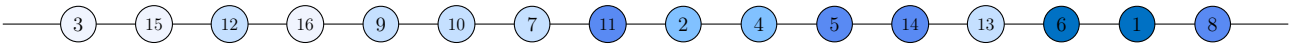


FIGURE 11 – Axe branch & bound

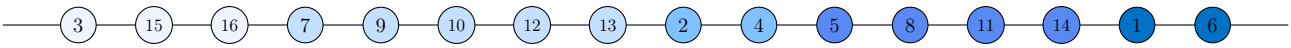


FIGURE 12 – Axe gauche-droite

Chaque candidat est représenté par une différente nuance de bleu selon son groupe d'appartenance (extrême gauche, gauche, centre, droite et extrême droite) afin de mieux visualiser les similitudes entre les deux axes ainsi que les erreurs de positionnement.

En effet, on retrouve bien les candidats Gluckstein (3), Laguiller (15) et Besancenot (16) à l'extrême gauche, ainsi que Megret (1) et Le Pen (6) à l'extrême droite. De plus, les candidats Taubira (7), Mamere (9), Jospin (10) et Hue (12) sont bien regroupés du même côté de l'axe. De même, les candidats Lepage (2) Bayrou (4), Chirac (5) et Boutin (11) sont regroupés de l'autre côté de l'axe et à la bonne position. Comme pour la sériation, on peut utiliser les graphes pour visualiser les erreurs de positionnement ainsi que les similitudes, pour mieux comparer ces axes à l'axe réel. Les graphes décrivant ces résultats sont en annexe A.2.

Conclusion

Dans ce projet, deux approches ont été étudiées et implémentées pour identifier un axe politique gauche-droite à partir de données de votes par approbation. Les deux approches retournent des axes assez proches de l'axe réel et sont globalement correctes. Cependant, en écartant certains bulletins avec la méthode de branch & bound, on obtient des axes qui ne correspondent pas du tout à la réalité. Le fait de ne pas prendre en compte quelques données que l'on juge non pertinentes et de négliger certains candidats, modifie tout l'axe. Cela dit, on gagne énormément en temps de calcul.

Il est important de noter que ces deux méthodes sont adaptées à deux situations différentes. En effet, le temps d'exécution de la sériation augmente en fonction du nombre de candidats, tandis que celle du branch & bound augmente en fonction du nombre de bulletins uniques. De plus, le branch & bound indiquera quels sont les bulletins cohérents ou non avec les axes trouvés. On peut ainsi observer ces incohérences, et déterminer si elles sont le fruit du hasard ou d'une erreur, ou si elles dénotent des idées politiques communes malgré une apparente dissension entre les groupes d'appartenance des candidats impliqués.

Dans notre cas, l'algorithme de sériation a donné des résultats avec des graphes plus monotones, et donc plus proches de la réalité, que branch & bound. Cependant, quelle que soit l'algorithme employé, les groupes politiques sont assez bien reconnus et les candidats souvent proches dans l'axe. Les groupes entre eux sont malheureusement parfois interchangés, créant de petites excentricités dans la monotonie. Une étude plus approfondie des différences entre ces deux algorithmes aurait pu être effectuée.

Pour ce projet, il a été nécessaire de réaliser une interface utilisateur pour permettre de lancer les algorithmes et visualiser les axes et les résultats. Notre plus grand regret a été notre incapacité à produire une interface donnant des informations sur le déroulement de l'algorithme lorsqu'il tourne.

A Annexes

A.1 Sériation : Résultats

Ci-dessous sont retranscrits les résultats de l'élection présidentielle française de 2002. Dans les données récupérées, les candidats ont été identifiés de la manière suivante :

1 - Bruno Megret
2 - Corinne Lepage
3 - Daniel Gluckstein
4 - François Bayrou
5 - Jacques Chirac
6 - LePen
7 - Christiane Taubira
8 - Jean Saint-Josse
9 - Noël Mamere
10 - Lionel Jospin
11 - Christine Boutin
12 - Robert Hue
13 - Jean-Pierre Chevenement
14 - Alain Madelin
15 - Arlette Laguiller
16 - Olivier Besancenot

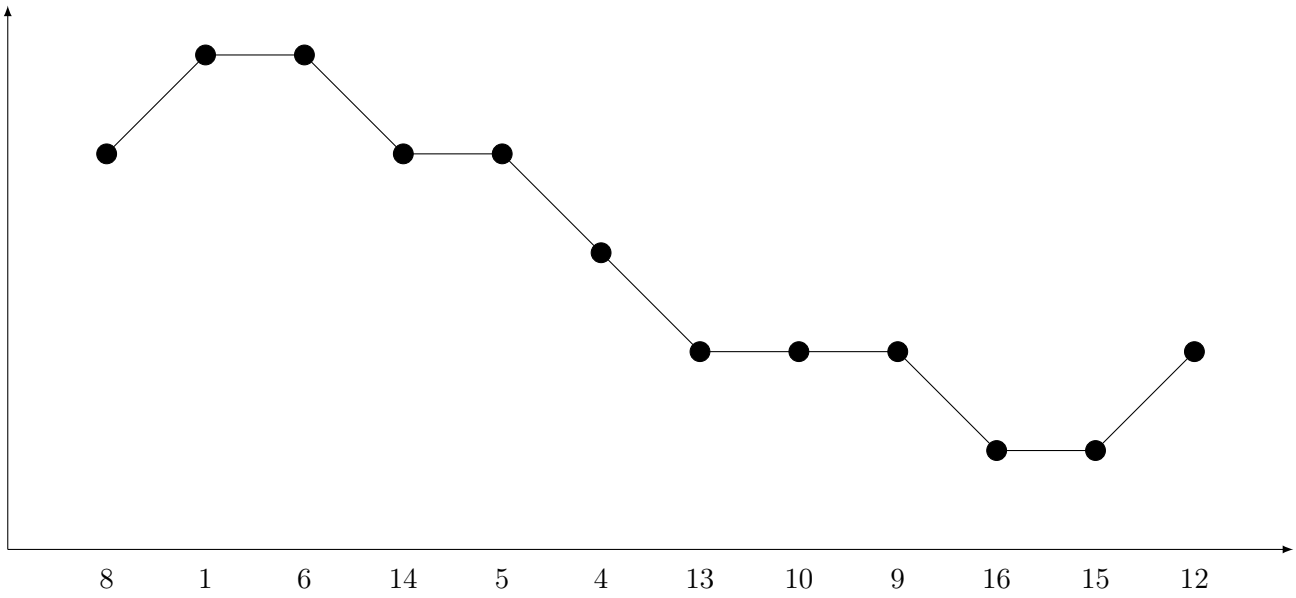


FIGURE 13 – Axe trouvé avec dissimilarity_and_n et un calcul pondéré

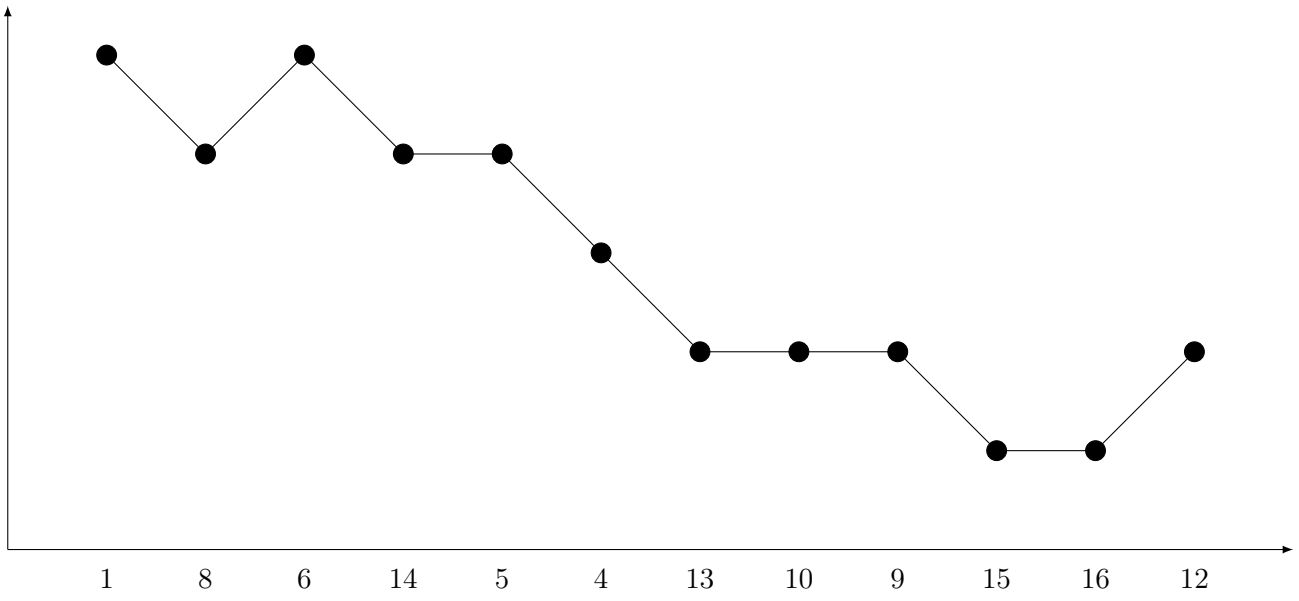


FIGURE 14 – Axe trouvé avec dissimilarity_and_n et un calcul non-pondéré

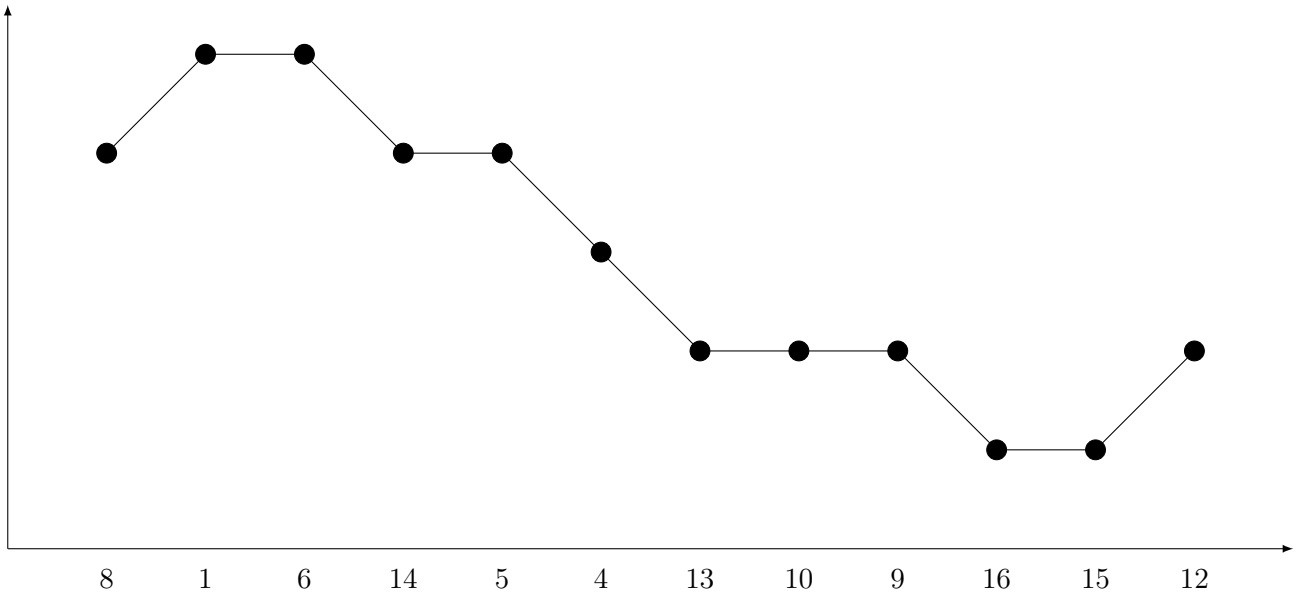


FIGURE 15 – Axe trouvé avec dissimilarity_and_or et un calcul pondéré

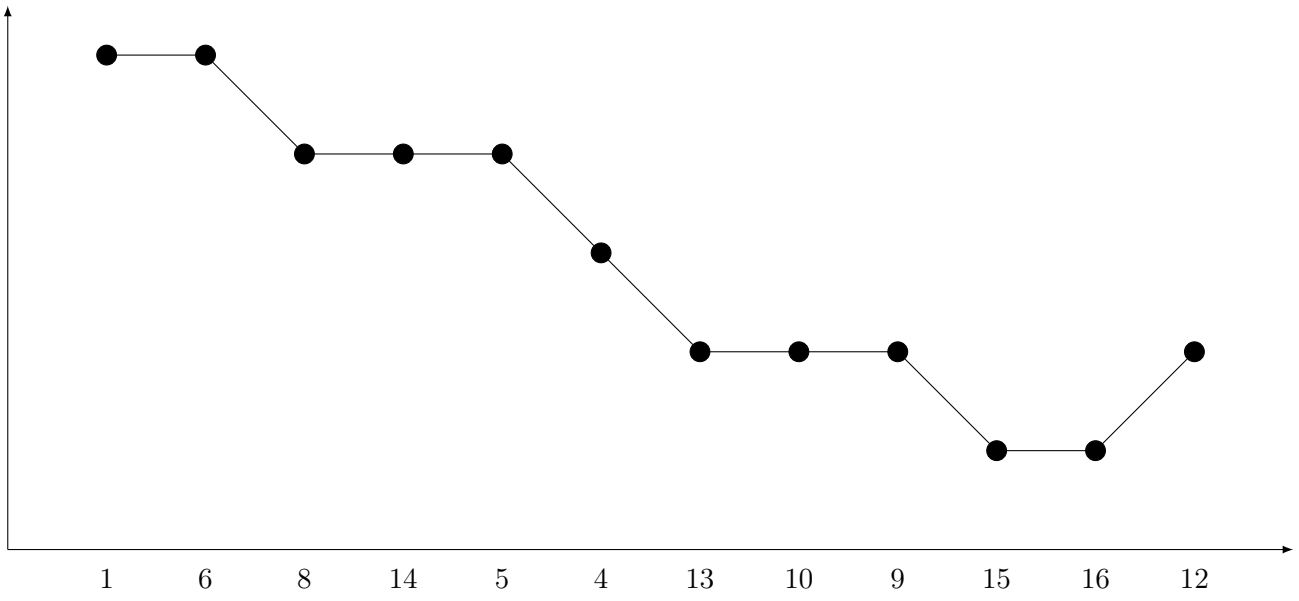


FIGURE 16 – Axe trouvé avec dissimilarity_and_or et un calcul non-pondéré

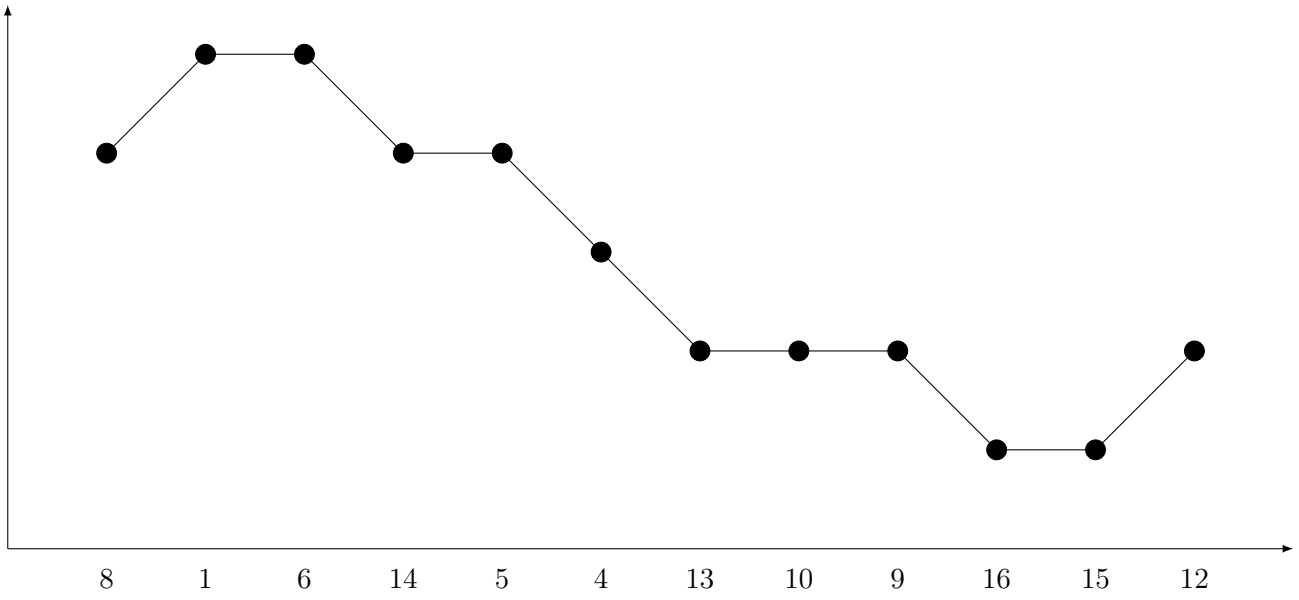


FIGURE 17 – Axe trouvé avec dissimilarity_over_over et un calcul pondéré

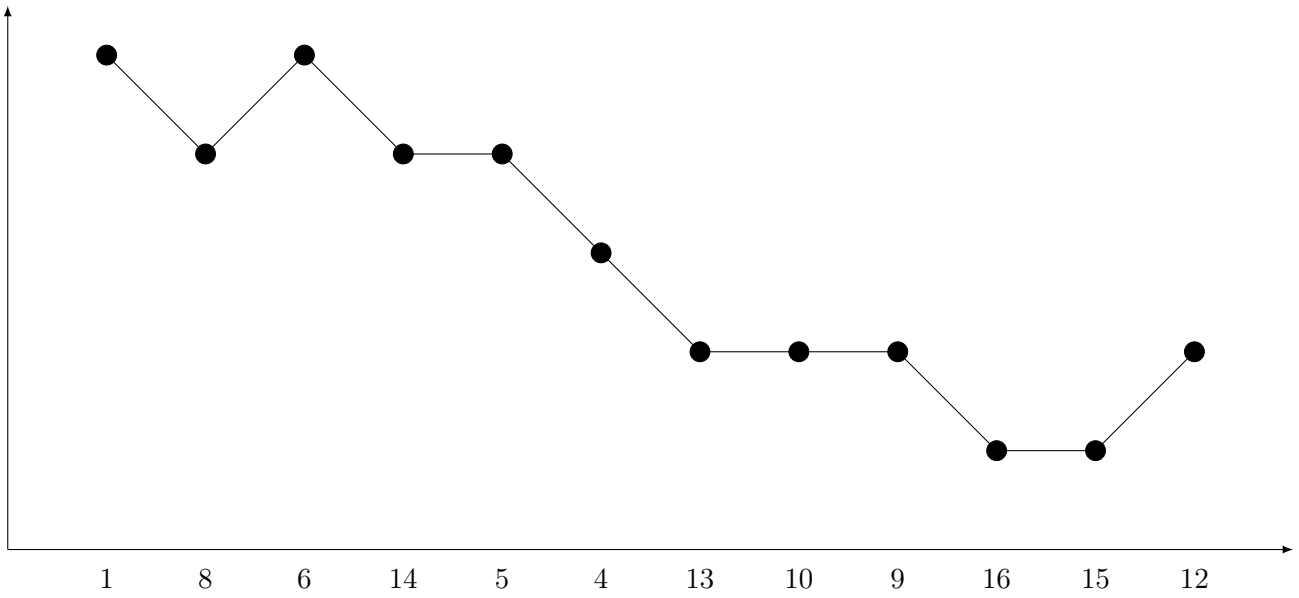


FIGURE 18 – Axe trouvé avec dissimilarity_over_over et un calcul pondéré

Tous les résultats de l'algorithme de sériation pour les données françaises, irlandaises, et de Glasgow sont disponibles dans notre projet sous `Data/TeX/pdf`. Nous rappelons que le projet est disponible à cette adresse : https://github.com/lauragreige/P-ANDROIDE.

A.2 Branch & bound : Résultats

Ci-dessous les résultats de l'élection présidentielle française de 2002 en utilisant l'algorithme de branch & bound.

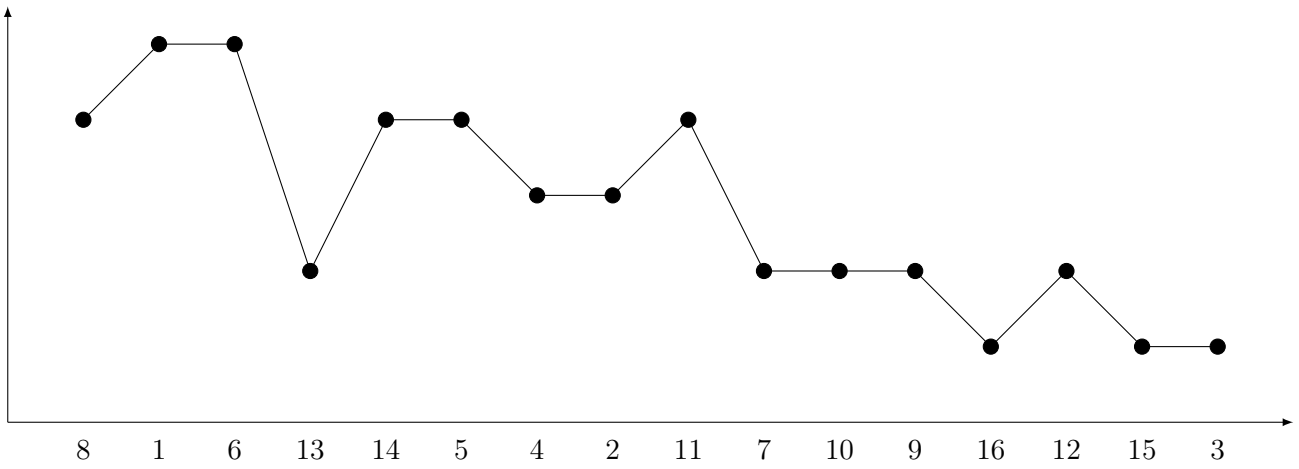


FIGURE 19 – Axe trouvé avec les données de l'instance ED-00026-00000003.toc

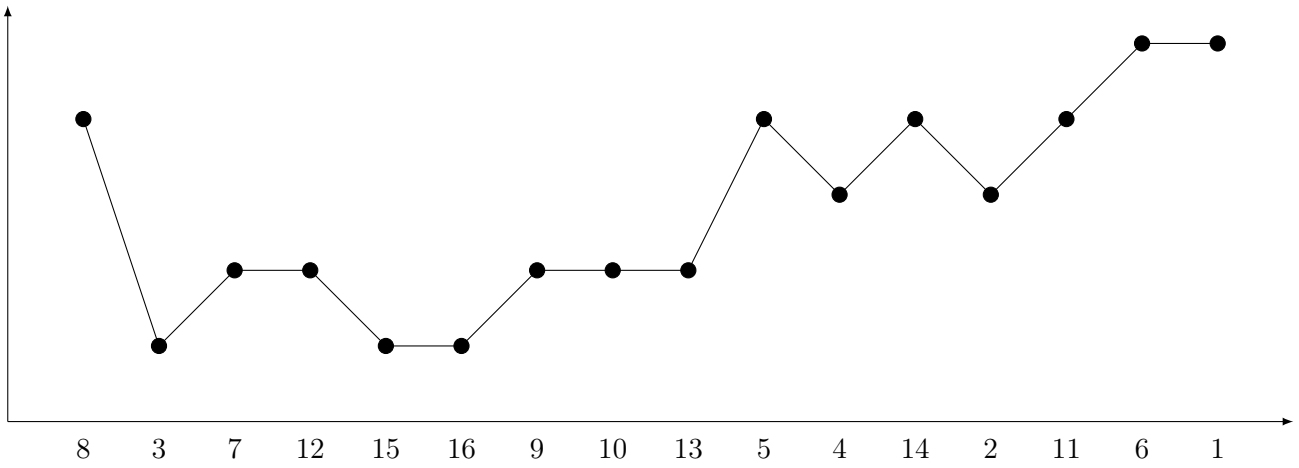


FIGURE 20 – Axe trouvé avec les données de l'instance ED-00026-00000004.toc

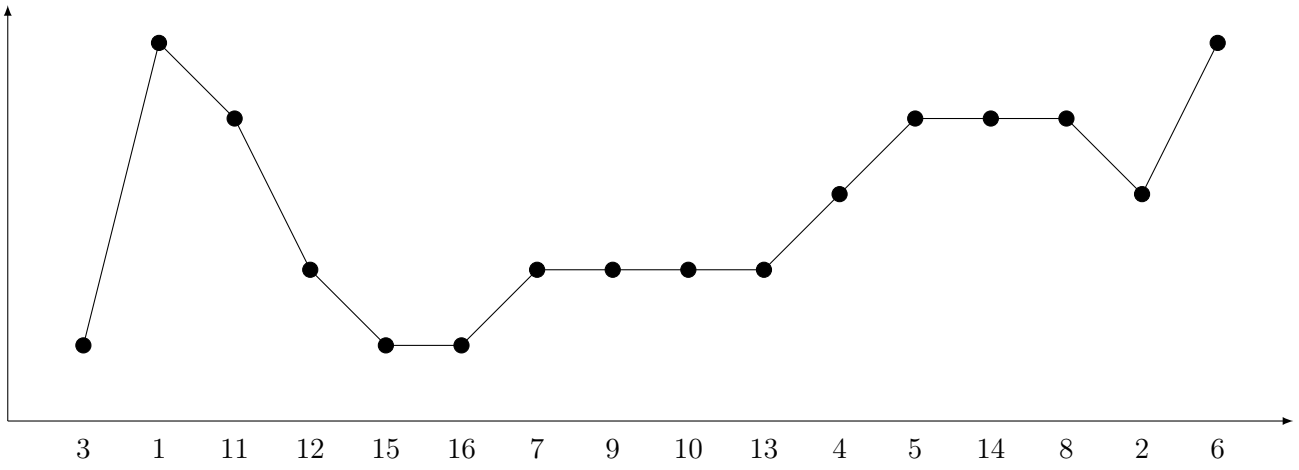


FIGURE 21 – Axe trouvé avec les données de l'instance ED-00026-00000005.toc

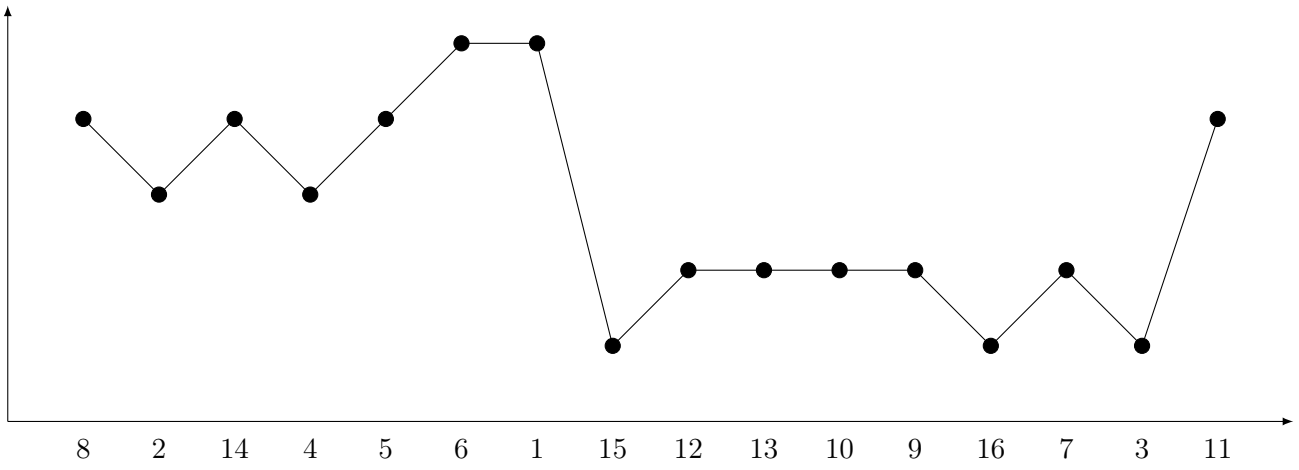


FIGURE 22 – Axe trouvé avec les données de l'instance ED-00026-00000006.toc

Références

[1] L.J. Hubert and R.G. Golledge. Matrix reorganization and dynamic programming : Applications to paired comparisons and unidimensional seriation. *Psychometrika-Vol.46*, pages 429–441, December 1981.