Analyse des algorithmes de tris

Christopher BIRD
Quentin RONDEAU
Matisse SENECHAL

29 mars 2024

UNIVERSITÉ
CAEN
NORMANDIE

Table des matières

1 Introduction

Suite à la présentation des différents projets par Grégory BONNET, nous avons décide de choisir le projet se basant sur **"l'Analyse des Algorithmes de tris"**.

1.1 Description générale du projet

Ce projet vise à explorer et à comparer la performance d'une variété d'algorithmes de tri en fonction du degré de désordre des données en entrée. En implémentant plusieurs algorithmes de tri et en utilisant un générateur de données paramétré, l'objectif est de comprendre comment différents algorithmes se comportent dans des conditions de données désordonnées, en termes de nombre de comparaisons, de temps d'exécution, ...

1.2 Présentation du plan du rapport

Dans ce rapport, nous commencerons par exposer les objectifs essentiels du projet, suivis d'une analyse approfondie de la problématique proposée. Ensuite, nous détaillerons les points-clés et les principales étapes du projet, avant de passer en revue les travaux existants sur le même sujet pour citer nos différentes sources d'inspiration. Nous décrirons après les fonctionnalités implémentées, en mettant en avant ce que le projet permet de faire, ainsi que l'organisation du travail au sein de l'équipe. Nous fournirons également des informations techniques sur les algorithmes. Nous présenterons de plus l'architecture du projet, en discutant de l'utilisation de paquetages non-standards, en fournissant des diagrammes des modules et des classes, ainsi qu'en expliquant les interactions entre les différentes composantes. Les expérimentations seront également abordées, avec des cas d'utilisation illustrés par des captures d'écran, des résultats obtenus et une analyse approfondie de ces résultats. Enfin, la conclusion récapitulera la problématique et la réalisation du projet, synthétisera les résultats obtenus et proposera des pistes d'amélioration dans l'éventualité où nous continuerions le projet.

2 Objectif du projet

2.1 Problématique du projet

Notre projet se penche sur la question de savoir comment les performances des algorithmes de tri sont affectées par le degré de désordre des données en entrée. L'objectif est d'analyser si certains algorithmes sont plus adaptés que d'autres à trier des ensembles de données présentant différentes formes de désordre, et si oui, dans quelles mesures.

2.2 Description des points-clés et des grandes étapes

Pour résumer ce projet à quelques points-clés, il faudrait tout d'abord, il faudrait développer un générateur de tableaux non triés, avec une fonctionnalité permettant de spécifier le niveau de désordre souhaité en termes de quantité et de répartition des données. Ensuite, implémenter une multitude d'algorithmes de tri, avec un objectif minimal de dix algorithmes différents, afin de fournir une base solide pour l'analyse. En parallèle, i faudrait développer une interface graphique pour la visualisation des algorithmes de tri, offrant une représentation visuelle du processus de tri. Enfin, il serait nécessaire d'améliorer le générateur, en introduisant des fonctionnalités permettant un contrôle plus fin du niveau de désordre des données générées, ce qui enrichirait notre capacité à étudier l'impact de cette caractéristique sur les performances des algorithmes de tri.

2.3 Description de travaux existants sur le même sujet

Quatre liens nous ont été proposés :

— le premier renvoyé vers une vidéo, intitulée : "15 algorithms in 6 minutes" [1] montrant l'exécution de différents algorithmes de tri sur des listes choisies aléatoirement. Cette vidéo nous a servies d'inspiration pour la visualisation graphique lors du tri des tableaux désordonnés.

1. Youtube : `https://www.youtube.com/watch?v=kPRA0W1kECg`

— le second, était un site, nommé : "Sorting Algorithms" [2], qui nous a donné nos premières idées quant aux algorithmes à implémenter. Nous avons compléter notre liste avec une page Wikipédia [3] référençant plusieurs algorithmes de tri,
— le troisième, amenait vers une liste d'algorithmes de tri [4], où l'on retrouvait pour chacun de ces algorithmes une courte description et leurs implémentations en C. De plus, on pouvait aussi y retrouver des exemples de graphique en barres, ce qui nous a donné certaines idées pour la représentation de nos expérimentations,
— enfin, le dernier lien [5] renvoyait vers des exemples de générateurs de tableaux en C, dont nous ne sommes pas inspirés.

3 Fonctionnalités implémentées

3.1 Organisation du projet

Nous avons divisé le travail en trois parties distinctes :
— Durant la première semaine, Matisse et Quentin se sont chargés de l'implémentation de la liste des algorithmes sélectionnés, tandis que Christopher a entamé la réflexion sur les générateurs et leur potentiel désordre.
— Les semaines suivantes, Quentin a finalisé les derniers algorithmes et a également écrit les tests associés. Pendant ce temps, Matisse et Christopher ont collaboré à l'implémentation des générateurs, en ajoutant progressivement divers paramètres de désordre et mesures à expérimenter.
— Au fil des semaines, Matisse a automatisé plusieurs tâches en créant des scripts shell pour simplifier la compilation et l'exécution des programmes. Il a également conçu les interfaces graphiques, incluant un formulaire pour les paramètres ainsi que des interfaces pour comparer les algorithmes à l'aide d'histogrammes ou d'écart-types. De plus, il a développé les classes nécessaires pour l'exécution des programmes selon la méthode choisie et pour le calcul des résultats.
— Pendant les dernières semaines, Matisse et Christopher ont mené les tests sur les générateurs déjà en cours de développement par Quentin. Celui-ci s'est concentré sur la visualisation en temps réel d'un algorithme sélectionné sur le générateur correspondant. En parallèle, Matisse a également pris en charge le script pour les expérimentations.

3.2 Description des fonctionnalités

3.2.1 Comparaison des algorithmes de tri

Afin de pouvoir comparer les tris entre eux, nous avons fait un programme python qui permet de visualiser sous forme d'histogramme les résultats que l'on peut obtenir qui sont : le temps d'exécution, le nombre de comparaisons et le nombre d'assignations.
Ces résultats sont obtenus :

1. soit en exécutant le script runSansInterface et en écrivant directement les paramètres, comme expliqué dans le fichier README.md, et en suivant les indications du programme,
2. soit en exécutant le script runAvecInterface. Cela lance un autre programme python qui sert à rentrer les différents paramètres nécessaires, qui sont le générateur à utiliser, le pourcentage de désordre et le nombre d'intervalles s'il y en a. Après avoir validé ces paramètres, une deuxième fenêtre s'ouvre et on peut indiquer quels résultats on veut obtenir.

Ces deux méthodes exécutent le programme python précédemment mentionné et affichent l'histogramme avec la moyenne des résultats de tous les tris sur 50 exécutions avec les mêmes paramètres, afin de rendre les résultats les plus précis possible tout en gardant un temps d'exécution rapide.
Voici un exemple de résultats que l'on peut obtenir :

2. Sorting Algorithms : `https://www.geeksforgeeks.org/sorting-algorithms/`
3. page Wikipédia : `https://fr.wikipedia.org/wiki/Algorithme_de_tri`
4. Comparison of several sorting algorithms : `http://warp.povusers.org/SortComparison/`
5. Sort Benchmark Data Generator and Output Validator : `http://www.ordinal.com/gensort.html`

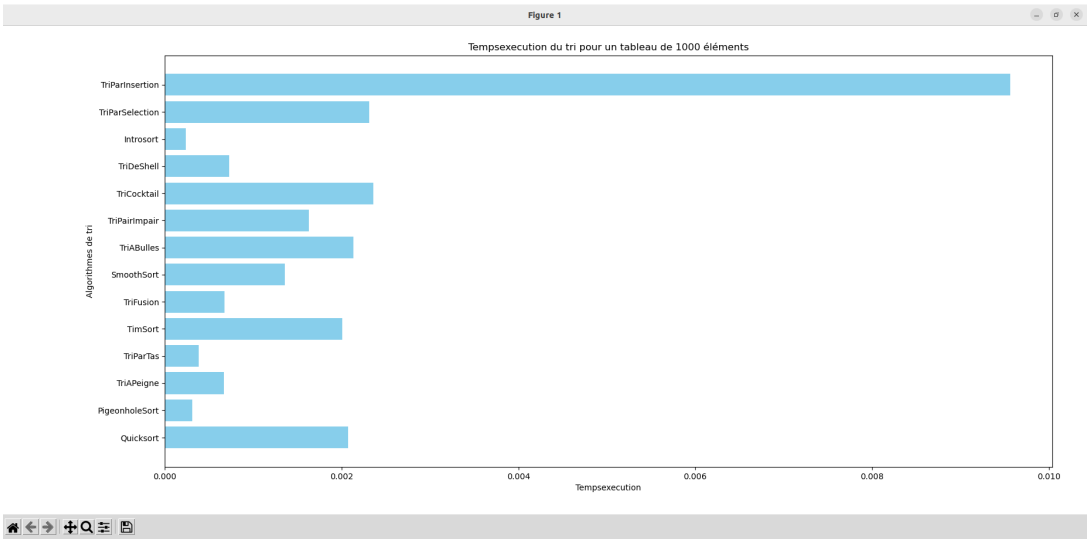


FIGURE 1 – Histogramme montrant le temps d'execution pour chaque tri

3.2.2 Comparaisons des écart-types des algorithmes de tri

De plus, nous avons aussi implémenté une interface graphique permettant d'observer l'écart-type entre tous les algorithmes de tri pour un générateur donné avec des paramètres définis.

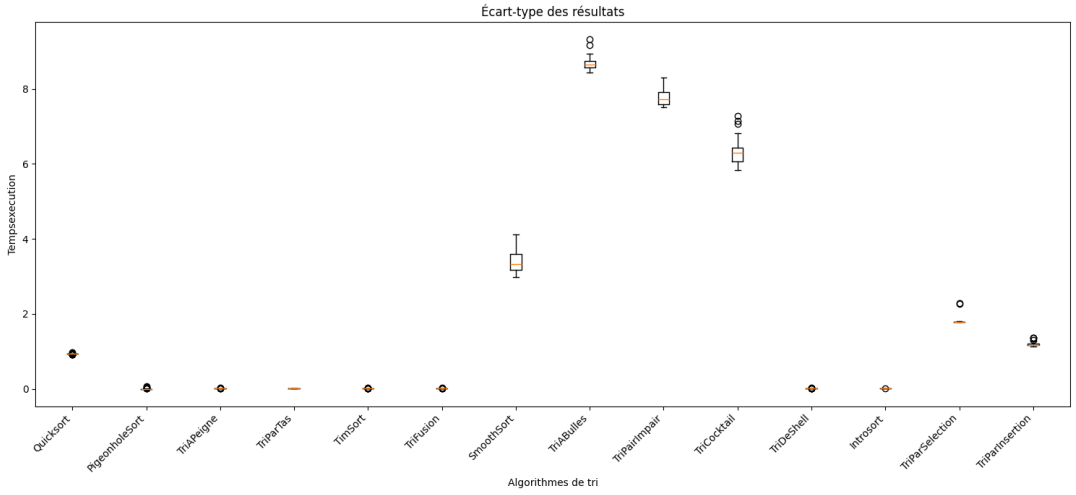


FIGURE 2 – Graphique représentant l'écart-type de chacun des algorithmes de tri, représenté par des boîtes à moustache

L'écart-type des algorithmes de tri sur nos générateurs, en fonction de diverses mesures telles que le nombre de comparaisons, d'assignations ou le temps d'exécution, offre une perspective sur la dispersion des performances de ces algorithmes. En considérant plusieurs générateurs produisant des ensembles de données variés, tels que partiellement triés, inversés ou aléatoires, nous pouvons évaluer les performances des algorithmes de tri. L'écart-type de ces performances révèle la variation des performances d'un ensemble de données à un autre. Une variation plus élevée indique une sensibilité accrue des algorithmes aux caractéristiques des données, tandis qu'une variation plus faible reflète une stabilité des performances. Ainsi, l'écart-type offre un aperçu de la cohérence et de la robustesse des algorithmes de tri face à la diversité des ensembles de données, facilitant ainsi le choix de l'algorithme optimal selon les besoins spécifiques de l'application.

3.2.3 Visualisation des algorithmes de tri

Nous pouvons également visualiser graphiquement le déroulé de chacun des tris implémentés.

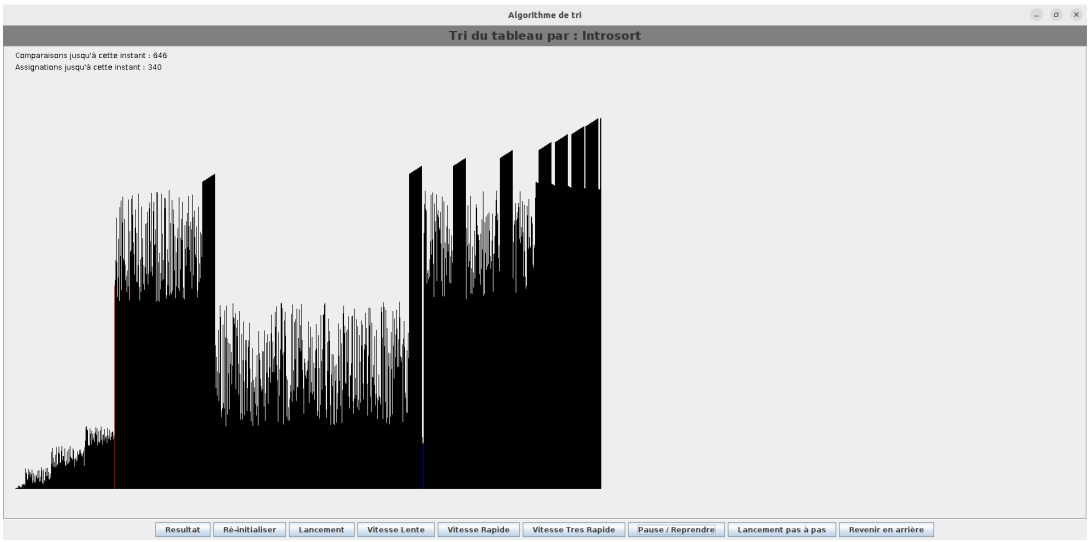


FIGURE 3 – Exemple de visualisation

Voici comment nous pouvons utliser cette interface :

— le bouton "Lancement" : lance un thread qui effectue une boucle while, afin d'afficher tous les états de la liste à trier, un par un, jusqu'à ce que le tri soit effectué.

— les boutons "Vitesse Lente / Vitesse Rapide / Vitesse Très Rapide" : permettent de changer la vitesse du tri, que ce soit avant ou bien pendant le lancement du tri.

— le bouton "Pause/Reprendre" : permet d'arrêter le tri ou de le relancer. Ce bouton modifie une variable qui rend la condition du "while" dans le "thread" de false à true pour reprendre l'exécution.

— le bouton "Lancement pas à pas" : affiche l'état de la liste à l'instant suivant (à chaque fois qu'il y a une modification durant le tri, un nouvel état est ajouté à cet "instant" dans la liste des états présents dans la classe du tri utilisé). Ce bouton peut être utilisé pendant que la boucle du bouton "Lancement" tourne, mais ce bouton est surtout utile quand cette boucle est mise en pause. Si l'état est le dernier de la liste, le bouton ne fait rien.

— les boutons "Revenir en arrière" : affiche l'état de la liste à l'instant précédent, il a les mêmes propriétés que le bouton précédent, à l'exception que celui-ci, ne fait rien quand l'état est le premier état de la liste et non le dernier.

4 Element techniques

4.1 Descriptions des algorithmes

4.1.1 Génération des tableaux

Dans notre démarche, pour la génération de tableaux désordonnés, nous avons dû prendre en compte le fait qu'il pouvait y avoir plusieurs types de générateurs. Pour cela, nous avons déclaré une méthode dans l'interface "Generateur" qui se nomme : "generer", et qui selon le type de générateur allé adopté

une stratégie différente. Voici son pseudo-code :

Algorithme 1 : Génère des valeurs ordonnées et désordonnées et remplit un tableau avec

```
1  intervallesDesordonnes = calculDesordre()
2  si le nom du generateur contient : "Alt" alors
       // Si le désordre consiste en des valeurs alternées
3      genererTableauAlt(intervallesDesordonnes)
4  sinon
       // Si le désordre consiste en des valeurs décroissantes
5      genererTableauDec(intervallesDesordonnes)
       // Si le désordre consiste en des valeurs mélangées
6      si le nom du générateur contient : "Mel" alors
7          répéter
8              melangerIntervallesDesordonnes(intervallesDesordonnes)
9          jusqu'à ce que la valeur la plus petite ne soit pas en première position
10     fin
11 fin
```

Pour le calcul du nombre de valeurs dans l'/les intervalle(s) nous avons décidé d'écrire une méthode abstraite : "calculDesordre". Cette méthode est implémentée ensuite, dans deux classes abstraites : "GenerateurDesordreIntervalle" et "GenerateurDesordreIntervalles". Elle est implémentée par deux classes, car son implémentation est différente, selon si l'on possède un ou plusieurs intervalles de désordre, voici respectivement les deux implémentations possibles :

Algorithme 2 : Calcule les bornes de l'intervalle des éléments à désordonner en fonction du pourcentage de désordre et de la répartition spécifiée

Sortie : La borne de l'intervalle de désordre

```
1  suivant repartition faire
2      cas où debut faire
3          debutDesordre = 0
4          finDesordre = le nombre d'éléments à désordonner−1
5      fin
6      cas où milieu faire
7          debutDesordre =(taille du tableau − le nombre d'éléments à désordonner)/2
8          finDesordre = debutDesordre+ le nombre d'éléments à désordonner−1
9      fin
10     cas où fin faire
11         debutDesordre = taille du tableau − le nombre d'éléments à désordonner
12         finDesordre = taille du tableau−1
13     fin
14 fin
15 retourner le tableau contenant debutDesordre et finDesordre
```

Algorithme 3 : Calcule les bornes des intervalles des éléments à désordonner en fonction du pourcentage de désordre

Sortie : Le tableau contenant les bornes des intervalles de désordre

```
1  longueurIntervallesDesordonnes = tableau contenant les longueurs de chaque intervalle
2  nbIntervallesDesordonnes = le nombre d'intervalles
3  longueurIntervallesOrdonnes = (taille du tableau - nombre d'éléments à désordonner) /
     nbIntervallesDesordonnes
4  intervalles = tableau vide
5
6  pour chaque longueur de longueurIntervallesDesordonnes faire
7  |  finIntervalle = debutIntervalle + longueur
8  |  on ajoute debutIntervalle et finIntervalle à intervalles
9  |  debutIntervalle = finIntervalle + longueurIntervallesOrdonnes
10 fin
11
12 retourner intervalles
```

Pour finir, comme vu à la page précédente nous possédons plusieurs méthodes pour la création des générateurs. Par exemple, "melangerIntervallesDesordonnes" consiste à mélanger les valeurs entre le début et la fin de chaque intervalle de désordre. Cependant, les deux méthodes qui vont surtout nous intéresser sont : "genererTableauDec" et "genererTableauAlt".

Algorithme 4 : Génère un tableau d'entiers en suivant des intervalles de désordre décroissant spécifiés

Entrées : intervallesDesordonnes, le tableau de l'intervalle de désordre sous la forme : [début, fin]

```
1  valeurMax ← −∞
2  valeurMin ← ∞
3  pour i < taille du tableau faire
   |  // Récupération de la valeur précédente, initialisée à 0 pour le premier élément
4  |  si i > 0 alors
5  |  |  precedenteValeur = récupère la valeur du tableau à la position i-1
6  |  sinon
7  |  |  precedenteValeur = 0
8  |  fin
9  |
10 |  si i est compris dans l'un des intervalles de intervallesDesordonnes alors
11 |  |  valeurMin = min(precedenteValeur-1, valeurMin)
12 |  |  ajoute au tableau valeurMin
13 |  |  valeurMax = max(precedenteValeur, valeurMax)
14 |  sinon
15 |  |  valeurMax = max(precedenteValeur+1, valeurMax)
16 |  |  ajoute au tableau valeurMax
17 |  |  valeurMin = min(precedenteValeur, valeurMin)
18 |  fin
19 fin
```

Algorithme 5 : Génère un tableau d'entiers en suivant des intervalles de désordre alterné spécifiés

Entrées : intervallesDesordonnes, le tableau des intervalles de désordre sous la forme : [début, fin]

```
1  valeurMax ← taille du tableau /2
2  valeurMinDesordonnee = (−taille du tableau/2) + 1 ;
   valeurMinOrdonnee = valeurMinDesordonnee
3
4  indexDerniereValeurOrdonnee = récupère la valeur de la dernière position ne faisant pas partie
   d'un intervalle de désordre
5  pour i < taille du tableau faire
   |  // Récupération de la valeur précédente, initialisée à 0 pour le premier élément
6  |  si i > 0 alors
7  |  |  precedenteValeur = récupère la valeur du tableau à la position i-1
8  |  sinon
9  |  |  precedenteValeur = 0
10 |  fin
11 |
12 |  si i est compris dans l'un des intervalles de intervallesDesordonnes alors
13 |  |  si precedenteValeur est positive alors
14 |  |  |  on ajoute au tableau valeurMinDesordonnee-1
15 |  |  sinon
   |  |  |  /* pour s'assurer que si un intervalle de désordre se trouve à la fin du tableau alors
   |  |  |  toutes ses valeurs positives ne doivent pas déjà être à la bonne position */
16 |  |  |  si un intervalle de désordre se trouve à la fin du tableau alors
17 |  |  |  |  on ajoute au tableau valeurMax − (la valeur se trouvant à
   |  |  |  |  indexDerniereValeurOrdonnee−2)
18 |  |  |  sinon
19 |  |  |  |  on ajoute au tableau valMax+1
20 |  |  |  fin
21 |  |  fin
22 |  sinon
23 |  |  on ajoute au tableau valeurMinOrdonnee+1
24 |  fin
25 fin
```

4.2 Description des données

Pour pouvoir récupérer nos données nous utilisons une méthode de la classe *"ResultatMesure"* (cf. 5.1.3), qui s'appelle *"mesureResultats"*, dont voici son pseudo-code :

Algorithme 6 : Mesure les résultats obtenus d'un algorithme de tri sur un tableau donné

Entrées : tri, l'algorithme de tri à mesurer et tableau, le tableau à trier
Sortie : Un objet "ResultatMesure" contenant les résultats des mesures

```
1  tempsAvantTri ← horloge actuelle
2  comparaisonsAvantTri ← le nombre de comparaisons du tri
3  assignationsAvantTri ← le nombre d'assignations du tri
4
5  triage du tableau
6
7  tempsApresTri ← horloge actuelle
8  comparaisonsApresTri ← le nombre de comparaisons du tri
9  assignationsApresTri ← le nombre d'assignations du tri
10
11 tempsExecution = (tempsApresTri − tempsAvantTri)/ convertit en secondes
12 comparaisonsEffectuees = comparaisonsApresTri − comparaisonsAvantTri
13 assignationsEffectuees = assignationsApresTri − assignationsAvantTri
14
15 retourner les mesures obtenus sous la forme d'un objet ResultatMesure
```

4.3 Codage de l'interface

Nous avons créé une interface graphique pour représenter l'exécution des algorithmes de tri sur une liste générée par l'un de nos générateurs. Elle nous permet de visualiser ce que font les tris pour obtenir une liste triée. À chaque fois qu'un élément est échangé avec un autre, le tri ajoute un état à sa liste, et quand nous voulons le visualiser, nous pouvons voir l'état du tableau à cet instant ainsi que le nombre de comparaisons et d'assignations effectués jusqu'à cet instant. Il y a également 2 positions du tableau qui n'ont pas la même couleur que les autres éléments, un est rouge et l'autre est bleu, il s'agit des 2 éléments qui ont interverti leur position. Il est aussi possible qu'il n'y ait qu'un seul élément qui ait changé de couleur, cela signifie qu'il n'y a pas eu d'échange à cet instant, mais que le tri change les valeurs un par un au lieu d'échanger les valeurs du tableau. L'interface est découpée en plusieurs parties :

1. La zone du haut contient un Label contenant le tri sélectionné.
2. La zone centrale contient le tableau à l'instant sélectionné ainsi que le nombre d'assignations et de comparaisons qui sont placées dans des labels et fixées à un emplacement qui ne devrait pas gêner l'affichage du tableau.
3. La zone du bas contient les boutons décrits dans la partie "3.2 Description des fonctionnalités".

5 Architecture du projet

Nous avons choisi de séparer notre projet en deux paquetages principaux :
— "ComparaisonTri",
— "Execution".

Le premier contenant l'essentiel de notre code et le second ne contenant uniquement les paquetages servant à l'exécution de notre programme selon le choix de l'utilisateur, ainsi que l'exécution des tests et des expérimentations.
À la racine de notre projet, on retrouve différents scripts shells (cf. page suivante) :
— *"compil.sh"*, qui comme son nom l'indique sert à faire la compilation de tous nos fichiers,
— *"runAvecInterface.sh"*, sert à lancer notre programme à l'aide d'une interface graphique,
— *"runSansInterface.sh"*, sert à lancer notre programme en ligne de commandes,
— *"runTest.sh"*, sert à lancer les tests sur les générateurs et les algorithmes de tri,
— *"runExperimentation.sh"*, sert à lancer les expérimentations.

5.1 Description des paquetages

Pour revenir à notre paquetage principal, *"ComparaisonTri"*, nous l'avons divisé en trois sous-paquetages, afin de respecter l'architecture MVC.

5.1.1 Contrôleur

Ainsi, dans le contrôleur, nous avons uniquement deux classes :
- *"VerifParams"*, qui se charge de vérifier que les paramètres rentrés pour un générateur soient bien valides, dans le cas où on décide d'exécuter le programme en ligne de commandes,
- *"VerifParamsVue"*, qui se charge de faire la même chose que la classe précédente, sauf que, ici elle ne vérifie que les paramètres dans le cas où on lancerait le programme avec une interface graphique.

5.1.2 Vue

Dans la vue, nous avons plusieurs classes servant à afficher sous la forme de graphique la comparaison entre les algorithmes de tri sur un générateur choisie et des paramètres rentrés par l'utilisateur. Il y a aussi la possibilité de visualiser un algorithme triant le générateur choisi précédemment (cf. 3.2.3).

5.1.3 Modèle

Qui plus est pour la partie modèle, nous l'avons également divisé elle aussi en plusieurs sous-paquetages :
- *"algoTris"* , qui contient tous les algorithmes de tri implémentés, ainsi que leurs tests associés,
- *"donnees"*, qui contient un sous-paquetage *"experimentation"*, dans lequel si l'on décidait de lancer les expérimentations, il posséderait alors tous les fichiers JSON créés. Il contient de plus, une classe *"ResultatMesure"*, qui est chargée de calculer les résultats et de les exporter sous format JSON dans les deux autres fichiers qui composent ce paquetage, qui sont : *"resultatsMoyensTris"* et *"resultatsTris"*,
- *"generateur"*, qui comme son nom l'indique contient tous les types de générateurs implémentés qui sont regroupés dans des sous-paquetages à l'intérieur de celui-ci selon si ces générateurs contiennent un ou plusieurs intervalle(s) de désordre.

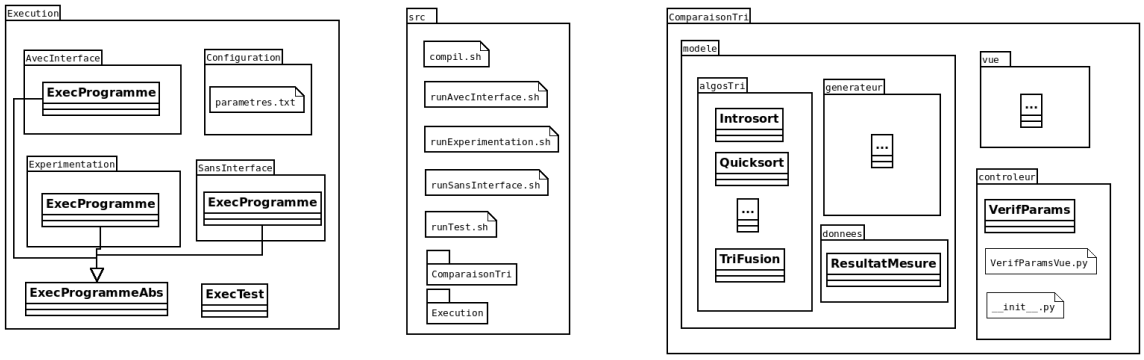


FIGURE 4 – Structure globale du projet

5.2 Interaction entre les classes/paquetages

Comme nous l'avons vu, page 11, nos trois paquetages principaux de notre projet interagissent bien entre eux. Mais maintenant, intéressons nous plus précisément aux sous-paquetages de *"modèle"*.

Tout d'abord, comme dit à la page précédente, le dossier *"algosTris"* contient tous les tris que nous avons implémenté. Ils héritent tous de la classe abstraite *"TriImplementation"* qui elle-même implémente l'interface *"Tri"*. Cela permet de factoriser le code afin d'éviter la répétition et d'améliorer la lisibilité du code et sa maintenabilité. Certains tris utilisent d'autres tris, comme Timsort qui utilise le tri par insertion ou IntroSort qui utilise le tri par tas. Les tests de ces tris se trouvant dans le sous-paquetage : *"algosTrisTests"* testent que les méthodes implémentées de l'interface soient correctement implémentées.

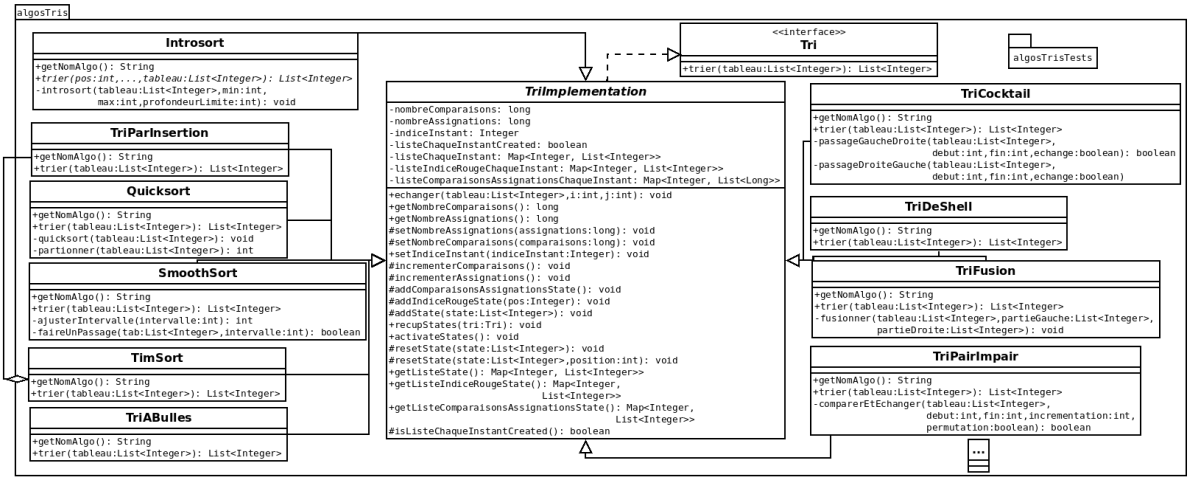


FIGURE 5 – Diagramme de classe du dossier algosTris

Ensuite, le paquetage *"generateur"* contient deux sous-dossiers séparant les générateurs ayant un intervalle de désordre et ceux qui en possèdent plusieurs. Les deux dossiers contiennent trois sous-paquetages où se trouvent les implémentations des générateurs. Ils héritent tous d'une classe abstraite qui hérite elle-même d'une autre classe abstraite *"GenerateurImpl"* qui implémente l'interface *"Generateur"*. Il est quasiment construit comme le dossier *"algosTris"* (cf. 5.2) et contient également un paquetage contenant tous les tests sur les méthodes de l'interface.

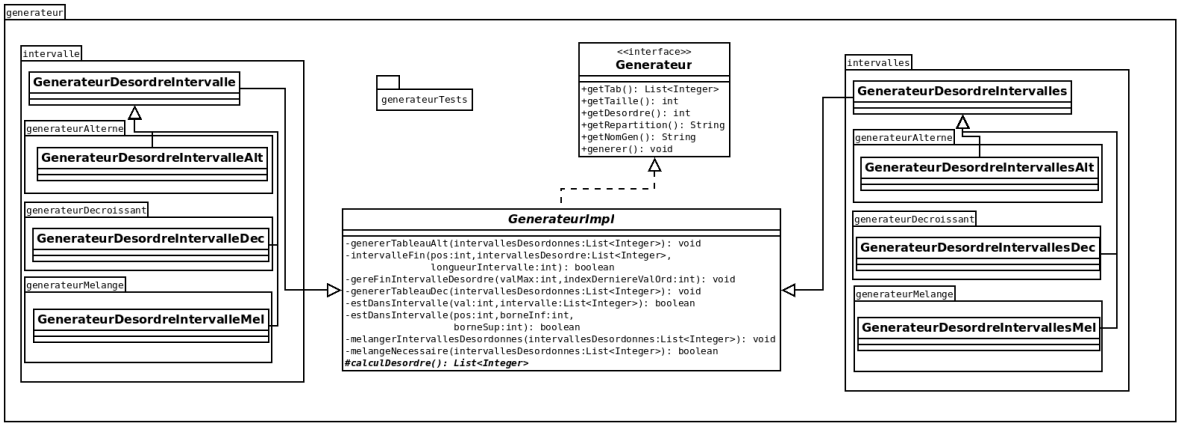


FIGURE 6 – Diagramme de classe du dossier generateur

Comme expliqué à la page 11, les paquetages *"vue"* et *"controleur"* sont mis en relations par le biais de l'architecture MVC.
Cependant, dans le paquetage *"vue"*, on voit que plusieurs de ses classes sont mises en relation. Ici, par exemple, nous avons représenté les classes servant à la visualisation du tri (cf. 3.2.3) d'un générateur en temps réel.

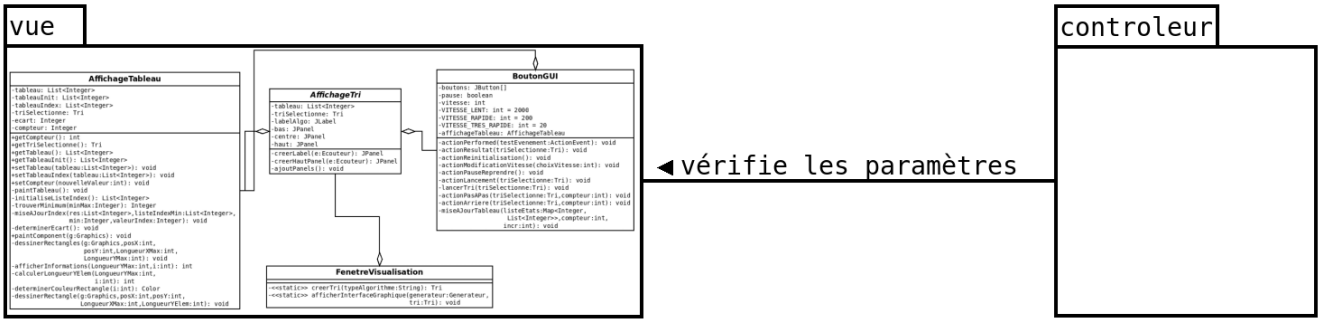


FIGURE 7 – Diagramme de classe du dossier vue

6 Expérimentations

Pour pouvoir réaliser nos expérimentations, nous avons décidé de varier tous les paramètres des générateurs, pour ainsi obtenir les résultats les plus qualitatifs possibles. En effet, nous avons utilisé plusieurs tailles pour les générateurs, les tailles sélectionnées sont :

- 1000 éléments,
- 5000 éléments,
- 10000 éléments,
- 50000 éléments.

Ce qui nous a permis de visualiser les expériences sur des tableaux de petites, moyennes et grandes tailles. De plus, nous avons aussi varié toutes les répartitions possibles, autrement dit :

- début,
- milieu,
- fin.

De plus, nous avons décidé de prendre trois cas pour le pourcentage de désordre, qui selon nous représenter au mieux les situations les plus particulières :

- 20% pour un tableau peu désordonné,
- 50% pour un tableau moyennement désordonné,
- 80% pour un tableau grandement désordonné,

Enfin, nous avons dupliqué toutes ces combinaisons de paramètres par deux. Car nous avons décidé de comparer les résultats entre des générateurs n'utilisant qu'un seul intervalle de désordre (l'intervalle des positions dans lequel on y retrouve toutes les valeurs désordonnées), que nous appellerons : G_1 et des générateurs qui en utilisent plusieurs, que nous noterons : G^+.

6.1 Cas d'utilisation

Pour pouvoir réaliser nos expérimentations, nous avons écrit un script shell : *"./runExperimentation.sh"* qui réalise les expériences avec toutes les possibilités énoncées ci-dessus.

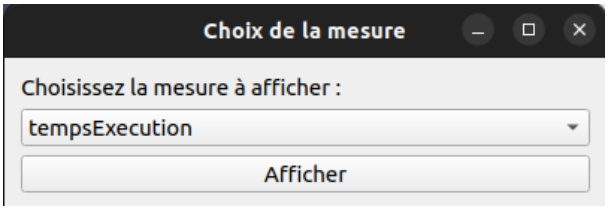
```
matisse@matisse-MS-7C83:~/bird_rondeau_senechal/src$ ./runExperimentation.sh
Voulez-vous effectuer les expérimentations (e) ou les afficher (a) ?
```

Cas d'utilisation

Nous avons ainsi donc deux possibilités :

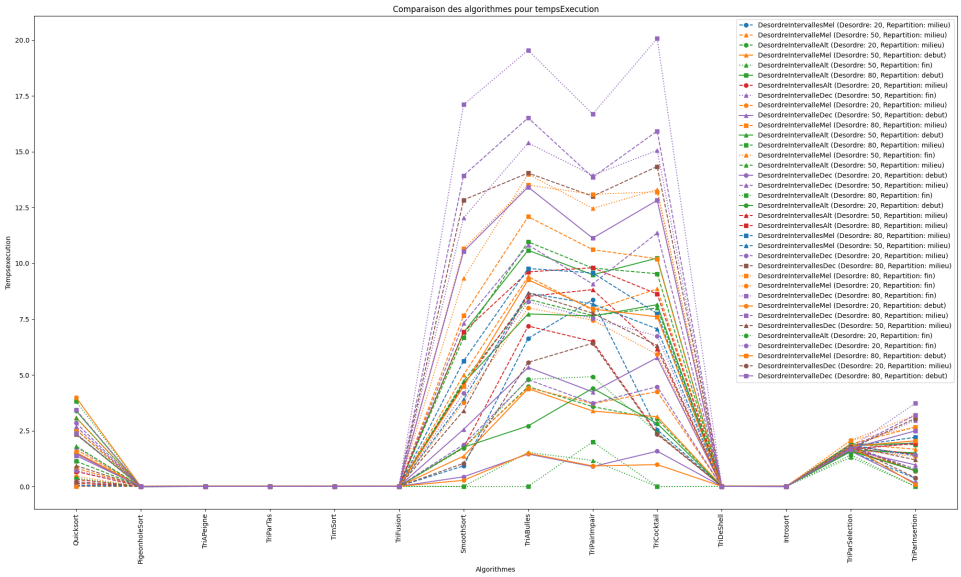
- soit nous décidons de lancer les expérimentations,
- soit nous décidons d'afficher les résultats obtenus (à condition que les expérimentations aient déjà été effectuées et que le nom des fichiers à afficher contiennent bien dans leurs noms "Resultats-MoyensTris").

Une fois les expériences terminées, si l'utilisateur décide de vouloir les afficher, il obtiendra ceci :



Choix de la mesure à afficher

Il s'affiche alors une interface graphique demandant à l'utilisateur quelle "mesure" (données) il désire afficher. Admettons qu'il choisisse d'afficher les résultats obtenus pour le temps d'exécution. Il obtiendra alors ce type de graphique :



Graphique d'exemple pour le temps d'exécution

Nous pouvons alors voir un graphique, avec en abscisse le nom de tous les algorithmes sur lesquels on a réalisé les expériences et en ordonnée les valeurs obtenues. Chaque courbe désigne les résultats obtenus pour un type de générateur et qui pour un type de générateur, on a toutes les combinaisons possibles (G_+ étant des cas particuliers puisque la répartition n'a pas d'influence). Cependant, dans les graphiques que nous vous montrons par la suite, la seule taille considérée sera 50 000, pour pouvoir mettre en exergue les disparités des valeurs obtenues, car dans la majorité des cas, or exceptions, les résultats obtenus ressemblaient à ceux que nous allons vous présenter ci-dessous.

6.2 Résultats quantifiables

Nous allons donc séparer nos résultats obtenus en trois parties, dans le cas où vous voudriez directement comparer toutes les combinaisons de générateurs entre elles, voir la section Annexes (cf. 8). Dans chacune de ces parties, nous diviserons les résultats entre les G_1 et les G_+.

6.2.1 Comparaisons

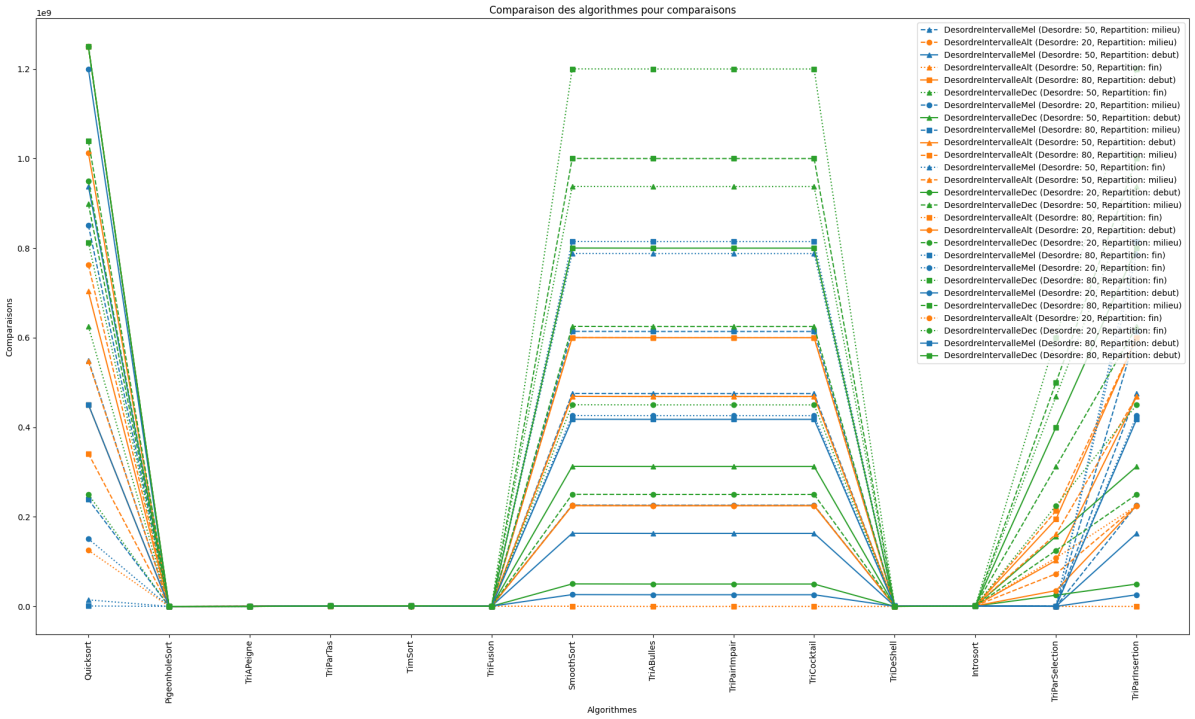


FIGURE 8 – Nombre de comparaisons pour chaque algorithme de tri, pour G_1

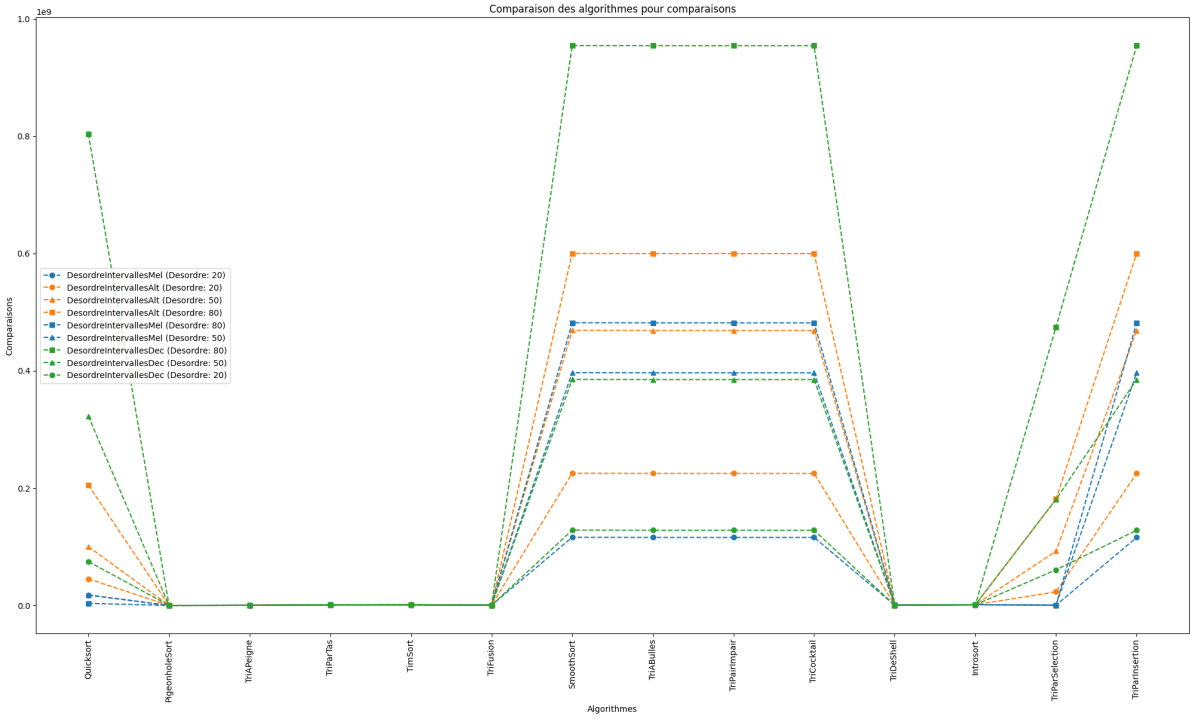


FIGURE 9 – Nombre de comparaisons pour chaque algorithme de tri, pour G_+

6.2.2 Assignations

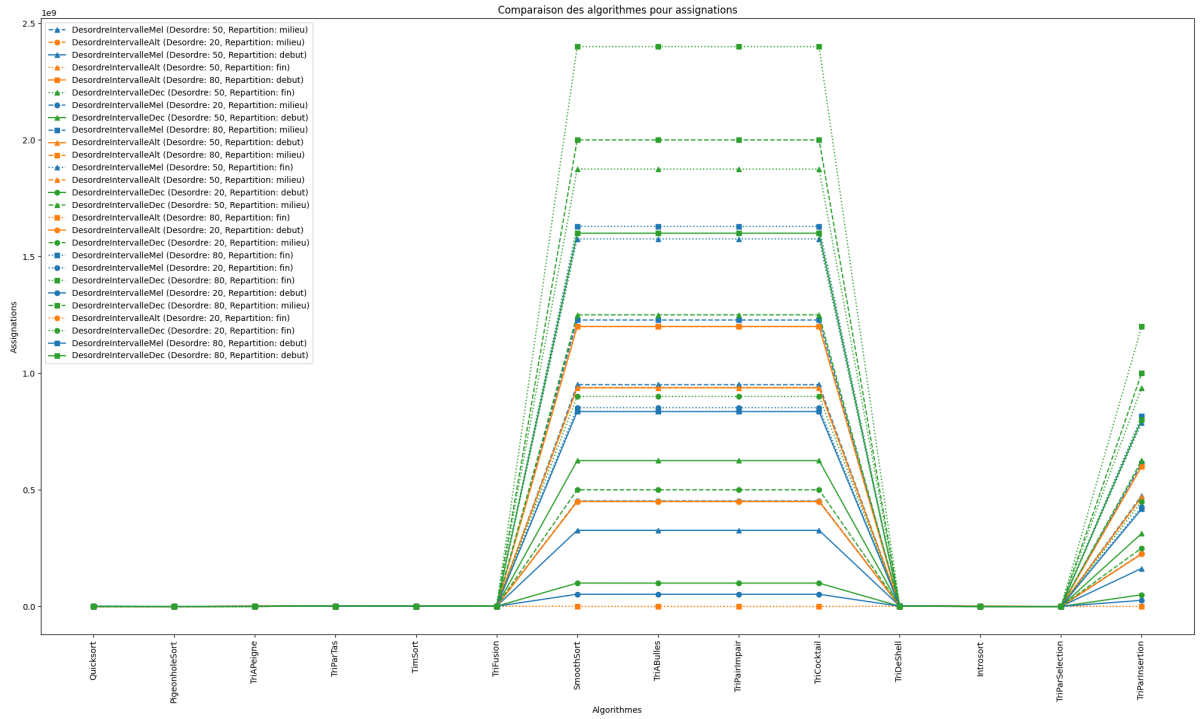


FIGURE 10 – Nombre d'assignations pour chaque algorithme de tri, pour G_1

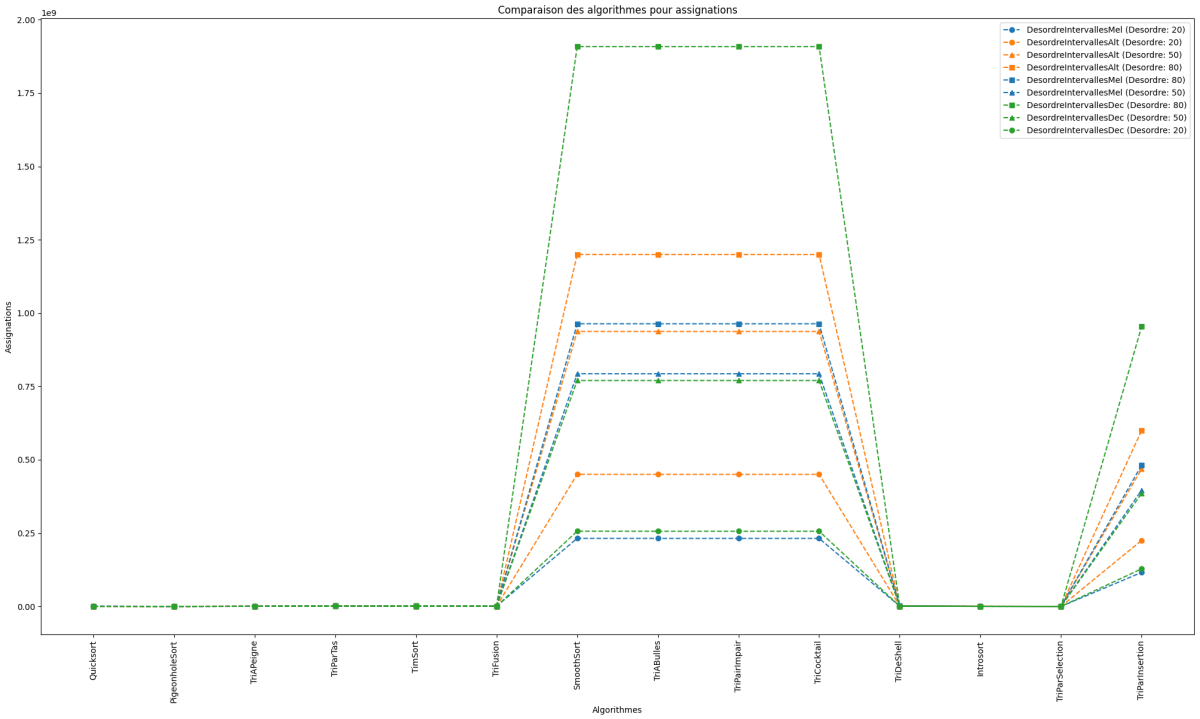


FIGURE 11 – Nombre d'assignations pour chaque algorithme de tri, pour G_+

6.2.3 Temps d'exécution

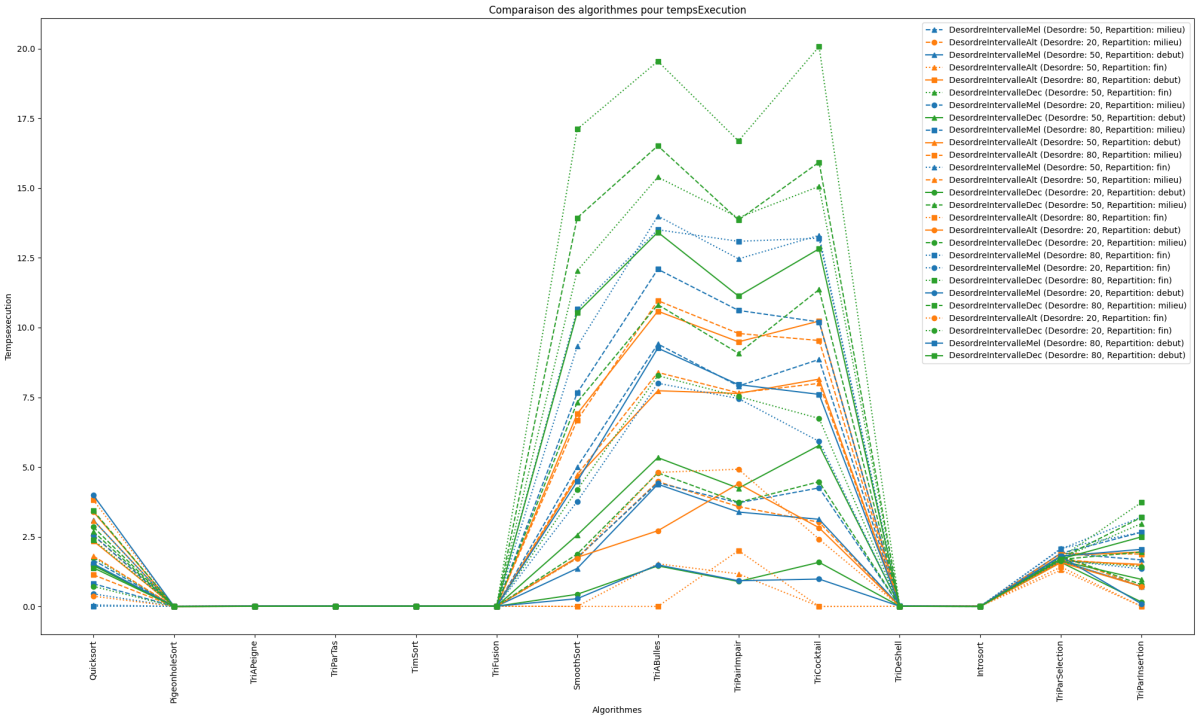


FIGURE 12 – Temps d'exécution pour chaque algorithme de tri, pour G_1

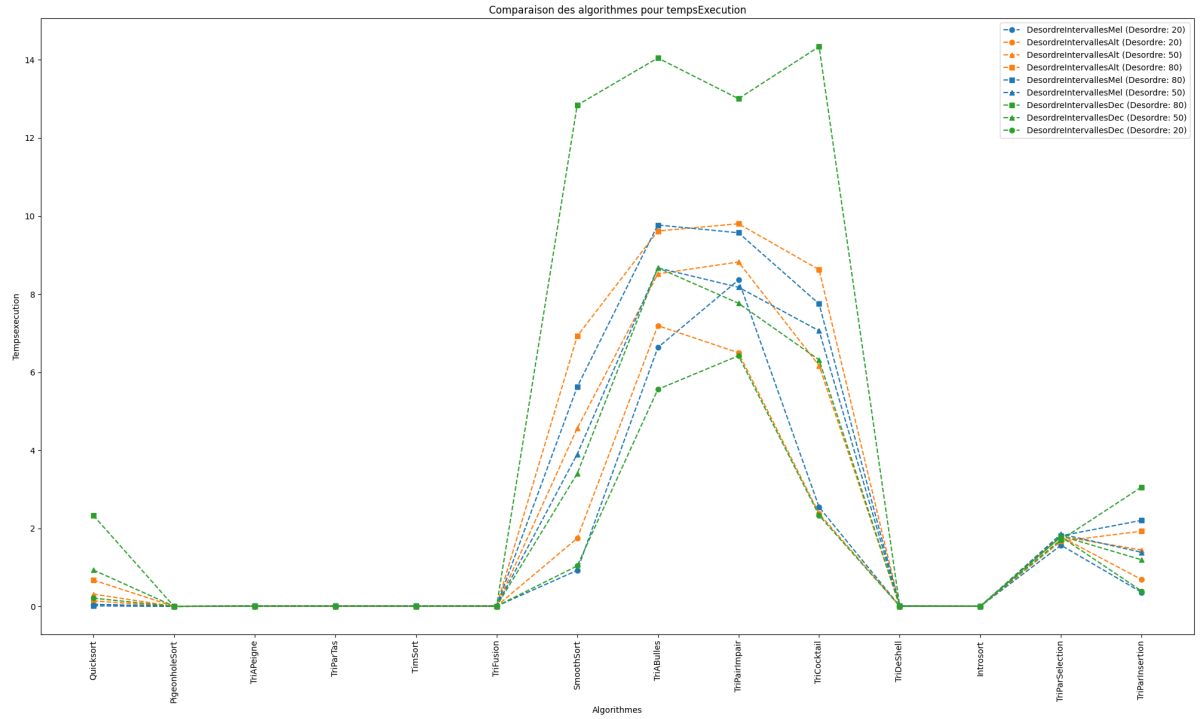


FIGURE 13 – Temps d'exécution pour chaque algorithme de tri, pour G_+

6.3 Analyse des résultats

Après avoir réalisé l'ensemble de nos analyses nous avons pu observer des ressemblances entre certains algorithmes. En effet, on décèle parmi les résultats quatre types de familles d'algorithmes.

6.3.1 1ère famille

Dans cette première famille, nous rangerons ces algorithmes :
- PigeonholeSort,
- Tri à peigne,
- Tri par tas,
- Timsort,
- Tri Fusion,
- Tri de Shell,
- Introsort.

On peut observer sur chacun des graphiques qu'ils possèdent tous les mêmes résultats pour n'importe quel type de générateur, et pour n'importe quelles combinaisons de paramètres rentrées dans ces dits générateurs.
On peut donc en conclure que le désordre quelle que soit sa répartition, son pourcentage, son nombre d'intervalles n'a aucune influence sur cette famille d'algorithmes.

6.3.2 2ème famille

Ensuite, dans cette seconde, nous allons y placer les algorithmes suivants :
- Smoothsort,
- Tri à bulles,
- Tri pair-impair,
- Tri Cocktail.

On observe des caractéristiques communes telles que les résultats soient similaires entre les différents algorithmes. Cependant, on observe aussi des réactions différentes dans certains cas, c'est ce que nous expliquons ci-dessous :
- Comparaisons :
  - Générateurs avec un seul intervalle de désordre (G_1) :
    - Pour G_1^{dec}, le nombre de comparaisons augmente à mesure que la répartition se rapproche de "fin" et que le pourcentage de valeurs désordonnées augmente.
    - Pour G_1^{mel} et G_1^{alt}, le nombre de comparaisons diminue à mesure que le pourcentage de désordre augmente.
  - Générateurs avec plusieurs intervalles de désordre (G_+) :
    - Dans tous les cas, le nombre de comparaisons augmente avec l'augmentation du pourcentage de désordre.
- Assignations :
  - Pour G^{dec} et G_+, les résultats sont similaires, mais inverses pour G^{mel} et G^{alt} par rapport aux résultats précédents.
  - Pour G_1^{dec} et G_1^{alt}, pour ces générateurs il y a moins d'assignations effectuées lorsque la répartition se rapproche de "fin".
- Temps d'exécution :
  - Générateurs avec un seul intervalle de désordre (G_1) :
    - G_1^{dec} suit une logique cohérente.
    - G_1^{alt} ne suit pas de logique claire, mais le temps d'exécution est généralement plus faible pour la répartition "fin".
    - G_1^{mel} ressemble à G_1^{dec}, mais avec des exceptions pour certains pourcentages de désordre.

En regroupant ces algorithmes dans cette deuxième famille, nous pouvons conclure que leurs performances varient en fonction du type de générateur et des paramètres de désordre. Cependant, des tendances communes peuvent être observées pour les comparaisons, les assignations et le temps d'exécution en fonction de ces facteurs.

6.3.3 3ème famille

Pour conclure, dans cette dernière famille, nous y retrouvons tous les cas particuliers, qui sont :
— Tri par sélection
— Tri par insertion
— Quicksort

G_1 :
— Quicksort :
  — Comparaisons : diminution du nombre de comparaisons lorsque la répartition se rapproche de "fin". Augmentation du nombre de comparaisons avec l'augmentation du pourcentage de désordre. À cela, on peut rajouter ces deux cas particuliers, que sont : G^{alt} et G^{mel}, qui ont tendance à réduire les comparaisons lorsque le pourcentage augmente.
  — Assignations : faibles quels que soient le type de générateur et les paramètres.
  — Temps d'exécution : aucune logique apparente concernant la répartition ou le pourcentage de désordre.
— Tri par insertion :
  — Comparaisons : augmentation du nombre de comparaisons lorsque la répartition se rapproche de "fin" et que le pourcentage de désordre augmente ; sauf pour G^{alt} et G^{mel} pour qui plus le pourcentage augmente moins ce tri fait de comparaisons.
  — Assignations : augmentation des assignations avec l'augmentation du pourcentage de désordre et lorsque la répartition se rapproche de "fin". En revanche, pour G^{alt} quand la répartition se rapproche de "fin" il fait moins d'assignations.
  — Temps d'exécution : augmentation des assignations avec l'augmentation du pourcentage de désordre et lorsque la répartition se rapproche de "fin". Il existe uniquement un cas particulier qui est : G^{alt}, pour qui on remarque que 20% et 50% ont globalement les mêmes résultats et que plus la répartition du désordre se rapproche de fin moins le temps d'exécution est élevé.
— Tri par sélection :
  — Comparaisons : augmentation du nombre de comparaisons lorsque la répartition se rapproche de "fin" et que le pourcentage de désordre augmente. De plus, on distingue clairement une hiérarchie entre les trois générateurs. En effet, G^{dec} est celui qui en fait faire le plus, vient ensuite G^{alt} et enfin G^{mel}, pour qui il semble nul.
  — Assignations : aucune assignation, peu influencée par la répartition ou le pourcentage de désordre.
  — Aucune influence significative de la répartition ou du pourcentage de désordre.

G_+ :
— Quicksort, Tri par insertion :
  — Comparaisons et Assignations : augmentation du nombre de comparaisons et d'assignations lorsque le pourcentage de désordre augmente.
  — Temps d'exécution : plus le pourcentage de désordre est grand plus ils prennent de temps.
— Tri par sélection :
  — Comparaisons : augmentation du nombre de comparaisons avec l'augmentation du pourcentage de désordre.
  — Assignations : aucune assignation, peu influencée par la répartition ou le pourcentage de désordre.
  — Aucune influence significative de la répartition ou du pourcentage de désordre.

7 Conclusion

7.1 Récapitulatif de la problématique et des résultats

Dans cette étude, notre objectif était d'évaluer l'efficacité des algorithmes de tri face au désordre des données en entrée. Pour ce faire, nous avons analysé plusieurs familles d'algorithmes, chacune présentant des réactions différentes en fonction du type de générateur et des paramètres de désordre. Tout d'abord, une première famille d'algorithmes, comprenant PigeonholeSort, Tri à peigne, Tri par tas, Timsort, Tri Fusion, Tri de Shell et Introsort, s'est révélée insensible au désordre des données. Ces algorithmes ont

maintenu une efficacité constante, quel que soit le niveau de désordre, montrant ainsi une robustesse face aux variations de données.

En revanche, une deuxième famille, regroupant Smoothsort, Tri à bulles, Tri pair-impair et Tri Cocktail, a présenté des réactions plus nuancées. Ces algorithmes ont montré des tendances variables en fonction du type de générateur et des paramètres de désordre. Par exemple, le nombre de comparaisons et d'assignations a fluctué en réponse aux variations de désordre, avec des comportements différents selon l'algorithme et le type de désordre. De même, une troisième famille, composée de Quicksort, Tri par insertion et Tri par sélection, a également montré une sensibilité au désordre des données. Les performances de ces algorithmes ont varié en fonction de la répartition et du pourcentage de désordre, avec des réactions spécifiques pour chaque algorithme et type de générateur.

Ces observations soulèvent des questions importantes sur l'optimisation des performances des algorithmes de tri dans des contextes réels. Comment choisir le meilleur algorithme en fonction des caractéristiques spécifiques des données ? Quelles stratégies de prétraitement des données peuvent être mises en œuvre pour maximiser l'efficacité des algorithmes de tri dans des situations de désordre ? Ces interrogations ouvrent la voie à des réflexions sur les futures innovations et développements dans le domaine des algorithmes de tri, visant à améliorer leur adaptabilité et leur efficacité dans des environnements de données diversifiés et complexes.

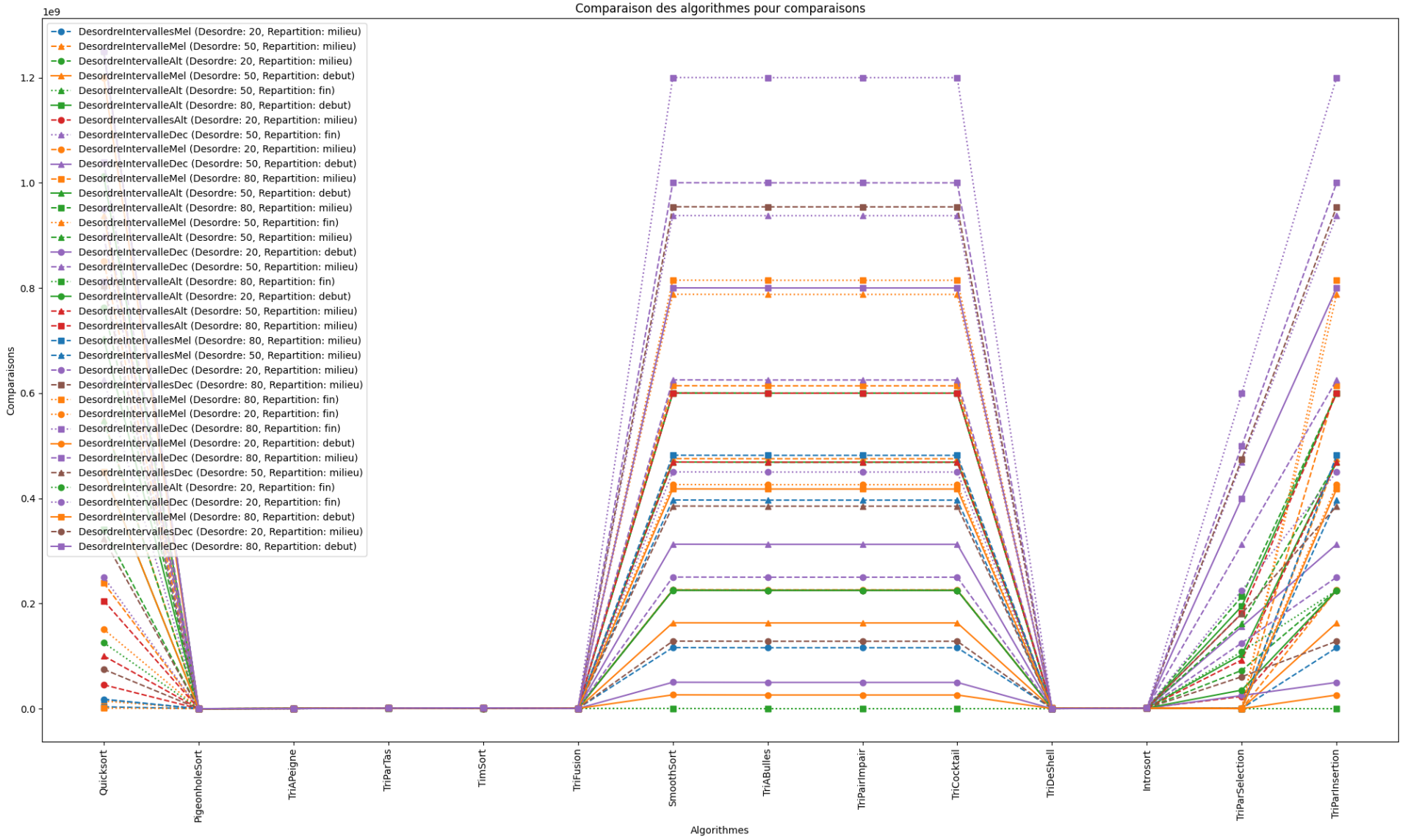
7.2 Propositions d'améliorations

1. Variation des paramètres expérimentaux : modifier le nombre d'intervalles de désordre pour observer leur impact sur les performances des algorithmes, ce que nous n'avons pas eu le temps d'implémenter.
2. Analyse de la complexité spatiale : examiner la complexité spatiale des algorithmes dans différentes situations de désordre pour obtenir une vision complète de leur efficacité.
3. Exploration de nouvelles familles d'algorithmes : en plus des familles d'algorithmes déjà étudiées, nous pourrions explorer d'autres types d'algorithmes de tri moins conventionnels ou plus récents.

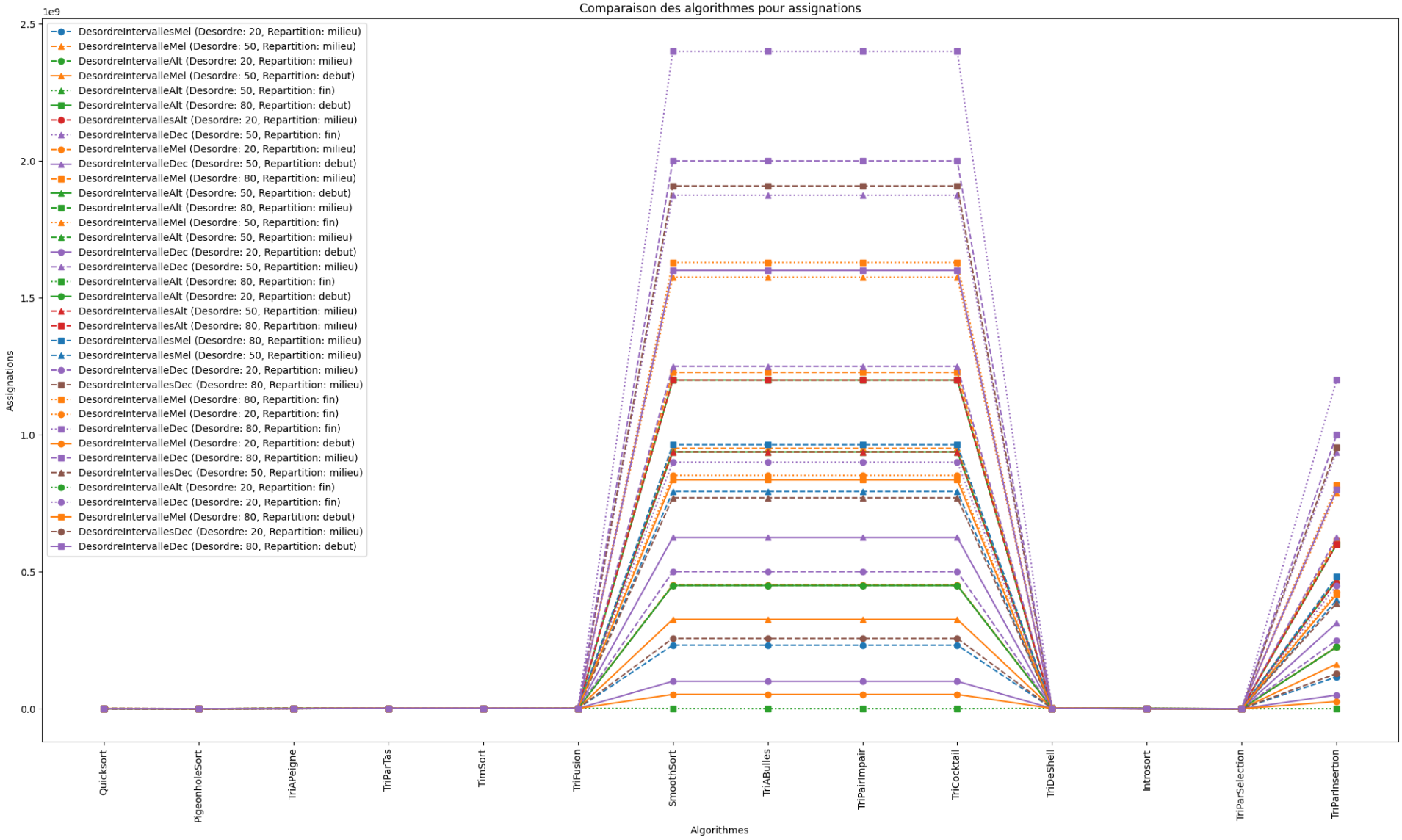
7.3 Remerciement

Un sincère remerciement à nos professeurs qui nous ont guidés. En particulier, nous tenons à remercier Monsieur BONNET Gregory pour nous avoir présenté les projets et fourni les connaissances théoriques nécessaires à ce projet. De même, nous exprimons notre gratitude envers Monsieur ZANUTTINI Bruno, le professeur de notre groupe de TP, qui nous a aidés à mieux comprendre notre projet et à rectifier nos erreurs de compréhension afin de respecter au mieux les objectifs du projet "Analyse des Algorithmes de tris".

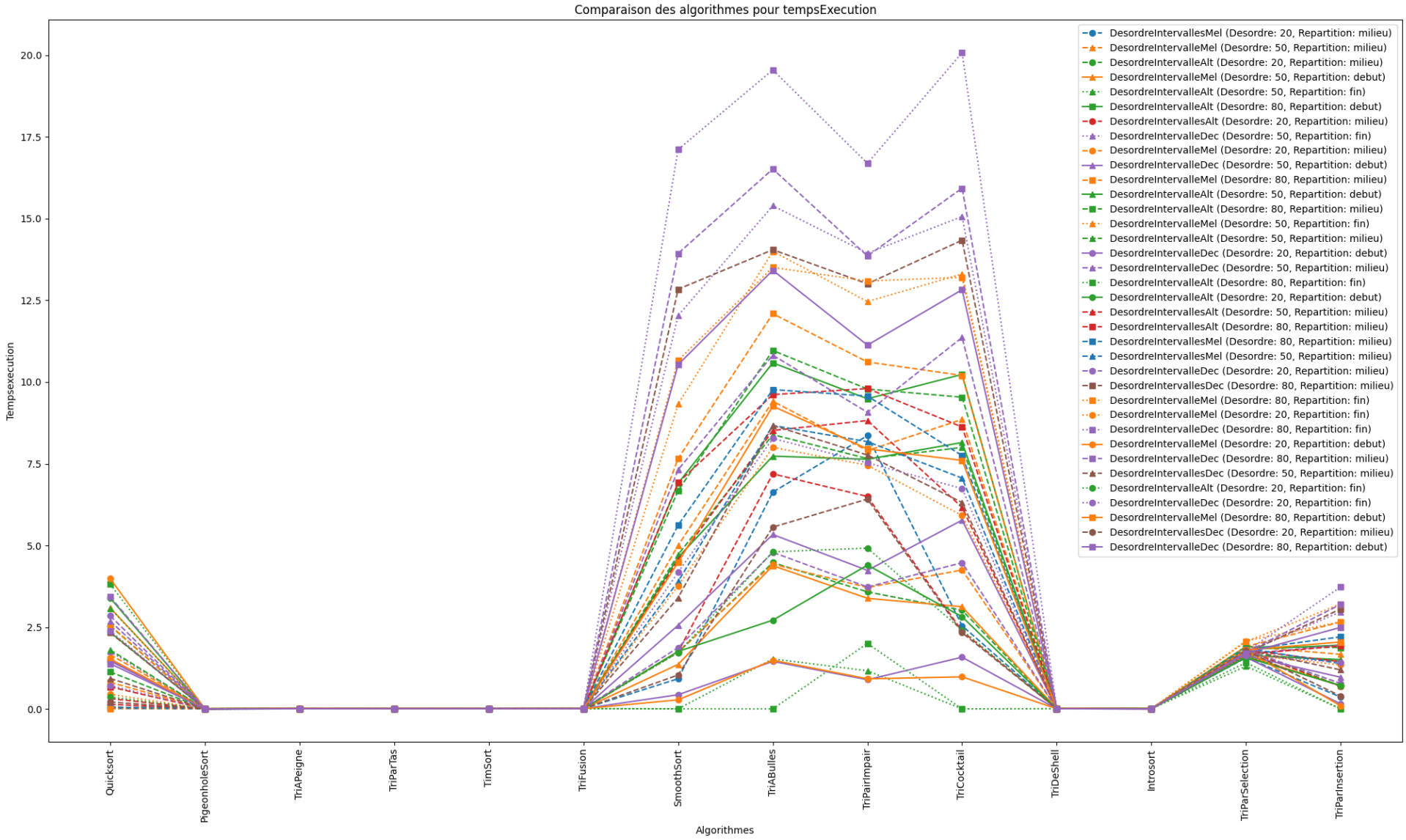
8 Annexes



Graphique regroupant le nombre de comparaisons pour tous les types de générateurs



Graphique regroupant le nombre d'assignations pour tous les types de générateurs



Graphique regroupant le temps d'exécution pour tous les types de générateurs