L'abus de comparaisons est mauvais pour la santé *

Emma Caizergues[1,2] **François Durand**[1] **Fabien Mathieu**[3]

[1] Nokia Bell Labs, Massy, France
[2] Université Paris-Dauphine, Paris, France
[3] Swapcard, Paris, France
emma.caizergues@nokia.com
francois.durand@nokia.com
fabien@swapcard.com

Résumé

Un *algorithme interruptible* (*anytime algorithm* en anglais) est un algorithme capable de renvoyer une estimation du résultat à chaque étape de son exécution. Dans cet article, nous nous intéressons au problème du *tri interruptible*. Nous considérons que chaque comparaison est une étape d'exécution de l'algorithme, et nous mesurons la proximité entre l'estimation et la liste triée à l'aide de la distance tau de Kendall. Nous présentons *Corsort*, une famille d'algorithmes de tris interruptibles reposant sur des estimateurs. Par simulation, nous montrons qu'un Corsort bien configuré a un temps de terminaison quasi-optimal, et fournit de meilleures estimations intermédiaires que les meilleurs tris dont nous avons connaissance.

Abstract

An *anytime algorithm* (*algorithme interruptible* in French) is an algorithm that is able to give an estimation of the result after each step of execution. In this article, we study the problem of *anytime sorting*. We consider that each comparison is a step of execution, and we measure the proximity between the estimation and the sorted list with the Kendall tau distance. We present *Corsort*, a family of anytime sorting algorithms using estimators. By simulation, we show that a well-configured Corsort has a quasi-optimal termination time, and gives better estimations than the other algorithms of our benchmark.

1 Introduction

Agathe, viticultrice dans le Haut-Rhin, possède n crus de Riesling qu'elle aimerait classer afin de mieux satisfaire les demandes de ses acheteurs et acheteuses. Chaque cru a une qualité distincte dont Agathe peut juger car elle est une experte : quand deux vins lui sont proposés, elle est capable d'identifier le meilleur, sans jamais faire d'erreur. Cependant, comme la perception de la qualité d'un vin est aussi complexe qu'éphémère, Agathe ne peut déguster que deux vins à la fois et ne peut pas comparer des vins d'une dégustation à l'autre. Autrement dit, Agathe ne peut pas insérer chaque nouveau vin qu'elle goûte dans une liste triée des vins déjà dégustés, mais doit comparer les crus deux à deux.

Pour éviter un excès de dégustations, Agathe souhaite effectuer un minimum de comparaisons. A minima, il faut qu'elle n'effectue pas deux fois la même comparaison. Également, elle est consciente que son palais peut être saturé à tout moment, rendant toute nouvelle dégustation impossible. Si elle doit s'arrêter avant d'avoir fini de trier tous les crus, elle souhaite avoir une bonne estimation du classement des vins. Elle cherche donc un algorithme de tri qui minimise le nombre de comparaisons tout en donnant une bonne estimation du résultat après chaque comparaison.

Contributions

Afin d'aider Agathe dans son classement, nous apportons les contributions suivantes :

— Nous formalisons le problème de la recherche d'un bon *tri interruptible par comparaisons*, dont la performance est mesurée par la distance tau de Kendall entre résultat provisoire et liste parfaitement triée ;
— Nous revisitons les algorithmes de tri classiques du point de vue *interruptible* ;
— Nous proposons des heuristiques simples pour estimer, à partir de l'ordre partiel correspondant à l'étape d'exécution en cours, le résultat final du tri (encore inconnu) ;

*Ce travail a été effectué au LINCS (https://www.lincs.fr/).

— Nous introduisons *Corsort*, une famille de tris interruptibles dont la logique repose sur des estimateurs ;
— Nous publions un paquet Python dédié à l'analyse de performance des tris interruptible par comparaisons ;
— Par simulation, nous montrons qu'un Corsort bien configuré a un temps de terminaison quasi-optimal, et fournit de meilleures estimations intermédiaires que les meilleurs tris dont nous avons connaissance.

Travaux connexes

Le classement de crus de Riesling s'inscrit dans le problème plus général du classement d'une liste par intervention humaine. Dans ce contexte, on considère généralement qu'attribuer une utilité à chaque élément de la liste n'est pas une technique fiable et qu'il vaut mieux procéder en comparant des paires [9]. Par exemple, le site web `https://www.pubmeeple.com/ranking-engine` propose une interface pour réaliser des classements par comparaisons successives. La comparaison humaine est beaucoup plus coûteuse que toute opération de base réalisée par ordinateur, et justifie que l'on sépare la complexité œnologique (en nombre de comparaisons) de la complexité globale [5, 17]. Des problèmes similaires apparaissent en dehors de toute intervention humaine dès que le coût de comparaison entre deux objets est prohibitifs. C'est par exemple le cas si les objets sont constitués de données massives et que chaque comparaison nécessite un transfert de données [15].

Si Agathe était sûre d'arriver au bout de la dégustation, on retrouverait un problème classique : trier en minimisant le nombre de comparaisons [6]. En plus d'algorithmes célèbres tels que le tri rapide, le tri fusion et le tri par tas (parmi bien d'autres), on peut citer l'algorithme de Ford-Johnson [8], extrêmement proche de la borne théorique en nombre de comparaisons, et même optimal pour certaines valeurs [16].

Une problème proche est celui *tri sous information partielle* : étant donné un ordre total qu'on ne connaît que partiellement (au sens où un ordre partiel compatible est connu), comment retrouver l'ordre total en un minimum de comparaisons [5] ? La plupart des variantes de problèmes de tri se déclinent *sous information partielle*, par exemple le classement des k meilleures valeurs [7]. Le problème du tri sous information partielle est à l'origine du principe de fonctionnement des tris *Corsort* proposés dans cet article.

La possible interruption de la dégustation nous place dans le domaine des *algorithmes interruptibles* (*anytime algorithms*), qui maintiennent à tout instant une estimation du résultat [18]. Étonnamment, les tris ont été peu étudiés dans cette littérature : les études ne concernent que le tri par sélection, le tri de Shell ou le tri rapide, sans introduire d'algorithme plus adapté, et les mesures d'écart au résultat final ne sont pas des distances [10, 12].

Parmi les problèmes connexes, les *algorithmes progressifs* peuvent aussi être interrompus à tout moment, mais l'accent est mis sur des bornes prouvables de performance en pire cas plutôt que sur l'efficacité moyenne empirique [1]. Les *algorithmes par contrat* opèrent également un compromis entre temps et précision, mais supposent que le temps disponible est connu à l'avance [18]. À l'inverse, le *tri approximatif* fixe un objectif en terme d'erreur maximale, et essaie de borner le nombre d'opérations nécessaire pour l'atteindre. L'algorithme ASort, sur lequel nous reviendrons dans la suite de cet article, donne des garanties de cette nature [9].

Le reste de l'article est organisé comme suit : la section 2 formalise les notions de tri interruptible et d'estimateur de classement ; la section 3 présente notre principale contribution, les tris *Corsort* ; la section 4 évalue la qualité des différentes solutions considérées au moyen de simulations ; la section 5 conclut.

2 Tris interruptibles

Formellement, nous voulons trier une liste $X = (X[1], \ldots, X[n])$, où $n > 0$, en effectuant des comparaisons du type : *est-ce que* $X[i] < X[j]$ *?* Un *tri interruptible* (*anytime sorting algorithm*) est un algorithme capable, à chaque étape k de son exécution, de renvoyer une estimation X_k du résultat. Dans notre modèle, chaque comparaison constitue une étape de l'algorithme [1]. On mesure la qualité de X_k par la distance tau de Kendall [14] entre X_k et la liste triée, c'est-à-dire par le nombre de paires d'éléments que l'estimation classe dans le mauvais sens : $\tau_k = |\{(i, j) : i < j, X_k[i] > X_k[j]\}|$. Idéalement, nous cherchons un tri interruptible dont le *profil de performance* $k \to \tau_k$, qui représente l'évolution de l'erreur commise au fur et à mesure de l'exécution de l'algorithme, est constamment plus bas que celui des autres algorithmes testés.

2.1 Tris classiques

Certains algorithmes classiques peuvent être vus comme interruptibles car ils maintiennent une liste courante qui converge vers la liste triée et peut servir d'estimation X_k. C'est par exemple le cas du tri rapide et du tri fusion, que nous avons tous deux implantés d'une manière favorable à l'esprit de l'algorithme initial : par exemple, pour le tri rapide, la position du pivot est mise à jour dans la liste après chaque comparaison.

Modifier l'ordre des comparaisons effectuées peut améliorer les estimations intermédiaires X_k. Pour le tri fusion, il est naturel de parcourir l'arbre de récursion en profondeur (DFS), c'est-à-dire en triant récursivement des parties gauche, puis droite de chaque sous-liste considérée. Cependant, on peut aussi le parcourir en largeur (BFS), en

1. Par convention, si l'algorithme termine en moins de k comparaisons, alors X_k est le résultat final, c'est-à-dire la liste triée.

DFS	BFS
(ab)	(ab)
(cd)	(cd)
$(abcd)$	(ef)
(ef)	(gh)
(gh)	$(abcd)$
$(efgh)$	$(efgh)$
$(abcdefgh)$	$(abcdefgh)$

TABLE 1 – Déroulement d'un tri fusion pour une liste de taille 8 ($abcdefg$), avec parcours récursif *classique* (DFS) ou *en largeur* (BFS). Chaque ligne représente une sous-liste que l'algorithme doit trier.

triant des sous-listes de taille croissante. La table 1 illustre la différence sur un exemple simple [2]. Les mêmes comparaisons sont faites dans les deux cas mais dans un ordre différent. L'ordre du parcours en largeur, en maintenant une répartition des comparaisons équilibrée sur l'ensemble des éléments de la liste, semble a priori plus favorable pour les estimations intermédiaires X_k (et nous le vérifierons expérimentalement).

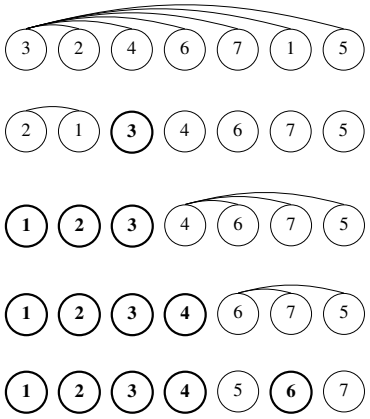


FIGURE 1 – Exécution du tri rapide sur la liste $X = (3246715)$. Chaque étape représente l'application complète d'un pivot. Les arêtes représentent les comparaisons effectuées. Un nœud apparaît en gras s'il a déjà été utilisé comme pivot : il partitionne alors la liste en éléments plus petits à gauche et plus grands à droite. On a omis les étapes sans aucune comparaison.

Pour le tri rapide, notre implémentation améliorée est équivalente à l'algorithme ASort [9], en utilisant la sélection rapide [11] comme sous-algorithme d'identification de la médiane. Le principe d'ASort est d'identifier la médiane de la liste et de séparer les éléments plus petits et les éléments plus grands, qu'on va ensuite trier récursivement de

2. Par concision, nous omettons les virgules pour noter les listes prises en exemples dans les tables et les figures. Par exemple, la liste (a, b, c, d) est notée $(abcd)$.

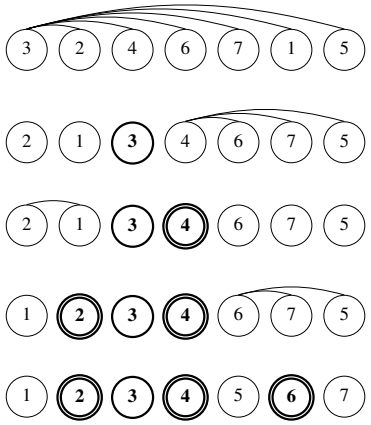


FIGURE 2 – Exécution d'ASort sur la liste $X = (3246715)$, avec la sélection rapide comme sous-algorithme de médiane. Chaque étape représente l'application complète d'un pivot par la sélection rapide. Un nœud apparaît en gras s'il a déjà été utilisé comme pivot, et doublement entouré si c'est une des médianes déjà trouvées. On a omis les étapes sans aucune comparaison. Lors de la troisième étape, ayant trouvé la médiane 4, on doit trouver la médiane de la sous-liste de gauche, (213) ; mais il est inutile d'effectuer toute comparaison avec l'élément 3 car celui-ci, précédemment utilisé comme pivot, est déjà à sa place définitive.

la même façon. Reste à décider comment on identifie la médiane. L'algorithme de sélection rapide, similaire au tri rapide, permet de la trouver par l'application de pivots successifs. Des exemples d'exécution du tri rapide et d'ASort sont donnés dans les figures 1 et 2. On constate que les comparaisons effectuées sont les mêmes, quoique dans un ordre différent, et il est facile de se convaincre que c'est toujours le cas.

D'autres algorithmes classiques permettent d'obtenir une estimation X_k par une transformation simple et naturelle de l'état courant. C'est le cas du tri par tas, qui conserve en mémoire le tas associé à la liste partiellement triée. Pour obtenir une estimation juste à l'égard de l'algorithme, il suffit donc de parcourir le tas à l'envers, puis les éléments déjà triés à l'endroit.

Enfin, pour certains algorithmes comme Ford-Johnson, il est difficile d'associer une estimation simple respectant « l'esprit » de l'algorithme. En effet, l'état courant dans l'algorithme de Ford-Johnson n'a pas une structure naturellement proche d'une liste : il n'existe aucune transformation simple qui puisse donner une estimation raisonnable. Comme il n'est pas toujours trivial de trouver une transposition naturelle de *l'idée* d'un algorithme en estimateurs intermédiaires, nous montrons dans la section suivante comment produire des X_k pour n'importe quel algorithme de tri.

2.2 Estimateurs

Pour rendre n'importe quel tri interruptible, nous proposons d'utiliser un estimateur qui ignore l'algorithme de tri utilisé et repose uniquement sur l'historique des comparaisons effectuées.

Notons $C_k = \{X[i_1] < X[j_1], \ldots, X[i_k] < X[j_k]\}$ le résultat de k comparaisons. C_k définit par clôture transitive un ordre partiel $\preceq_k$ sur les éléments de la liste [3]. Un *estimateur* est une fonction qui associe à tout ordre partiel un ordre total compatible.

Il existe un estimateur optimal qu'on peut résumer en deux étapes. D'abord, on considère l'ensemble des ordres totaux compatibles avec $\preceq_k$, appelés ses *extensions linéaires*. Ensuite, on trouve l'ordre qui minimise l'espérance de la distance tau de Kendall avec une extension linéaire aléatoire uniforme : autrement dit, on applique la *méthode de Kemeny* [13] au *profil de vote* constitué des extensions linéaires. Malheureusement, énumérer les extensions linéaires est un problème #P-complet [3], et appliquer la méthode de Kemeny à un profil de vote est aussi un problème NP-difficile [2]. Le coût de cette approche semble donc absolument prohibitif.

En pratique, pour construire un estimateur, nous allons dorénavant utiliser une fonction de *score* qui associe à chaque élément une valeur qui reflète sa position estimée dans la liste. L'estimation renvoyée est la liste associée au tri des scores. Rappelons que les complexités globale et œnologique sont distinctes : il est bien moins coûteux de trier n scores que de comparer deux crus. En cas d'égalité des scores, on renvoie les éléments dans leur ordre d'origine. Par un léger abus de langage, nous appelons aussi estimateur la fonction de score utilisée.

Une première idée est d'associer à chaque élément x un score correspondant à sa position moyenne $p_k(x)$ dans les extensions linéaires de $\preceq_k$. C'est bien un estimateur : si un élément est plus grand qu'un autre dans l'ordre partiel $\preceq_k$, alors il le sera dans toutes ses extensions linéaires, donc aussi selon la position moyenne p_k. Autrement dit, p_k renvoie toujours un ordre total cohérent avec l'ordre partiel. En outre, renvoyer une extension linéaire « moyenne » assure une distance raisonnable à la liste triée. Malheureusement, le coût de l'estimateur p_k reste prohibitif en complexité globale, puisque c'est également un problème #P-complet [3].

Nous proposons donc d'introduire des fonctions de score heuristiques plus simples à calculer pour produire des estimations X_k raisonnables. Pour cela, si x est un élément de la liste, on note $d_k(x) = |\{y \in X : y \preceq_k x\}|$ et $a_k(x) = |\{y \in X : x \preceq_k y\}|$ respectivement le nombre de descendants et d'ancêtres connus de x (lequel est inclus dans les deux ensembles par convention). Une manière simple de calculer d_k et a_k est de partir de C_k et de construire sa clôture transitive, pour un coût en $O(n^2)$.

Nous avons considéré les fonctions de scores suivantes :

— Δ_k, définie par $\Delta_k(x) = d_k(x) - a_k(x)$. Δ_k attribue à chaque x un score qui reflète la moyenne entre sa plus basse et sa plus haute positions possibles.

— ρ_k, définie par $\rho_k(x) = d_k(x)/(d_k(x) + a_k(x))$. Cela revient à positionner x comme si ses descendants et ses ancêtres avaient en moyenne des positions régulièrement espacées au sein de l'ensemble de la liste.

S'il n'y a pas d'ambiguïté, nous omettrons par la suite l'indice k. Tout comme p, Δ et ρ sont des estimateurs : il est facile de vérifier que si un élément est plus grand qu'un autre dans l'ordre partiel, alors il aura aussi un plus grand score [4].

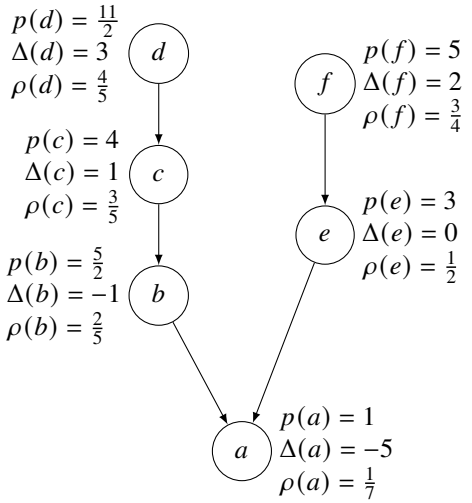


FIGURE 3 – Exemple de sélection d'extension linéaire. Ici, les estimateurs p, Δ et ρ renvoient la même estimation $(abecfd)$, en accord avec la méthode de Kemeny.

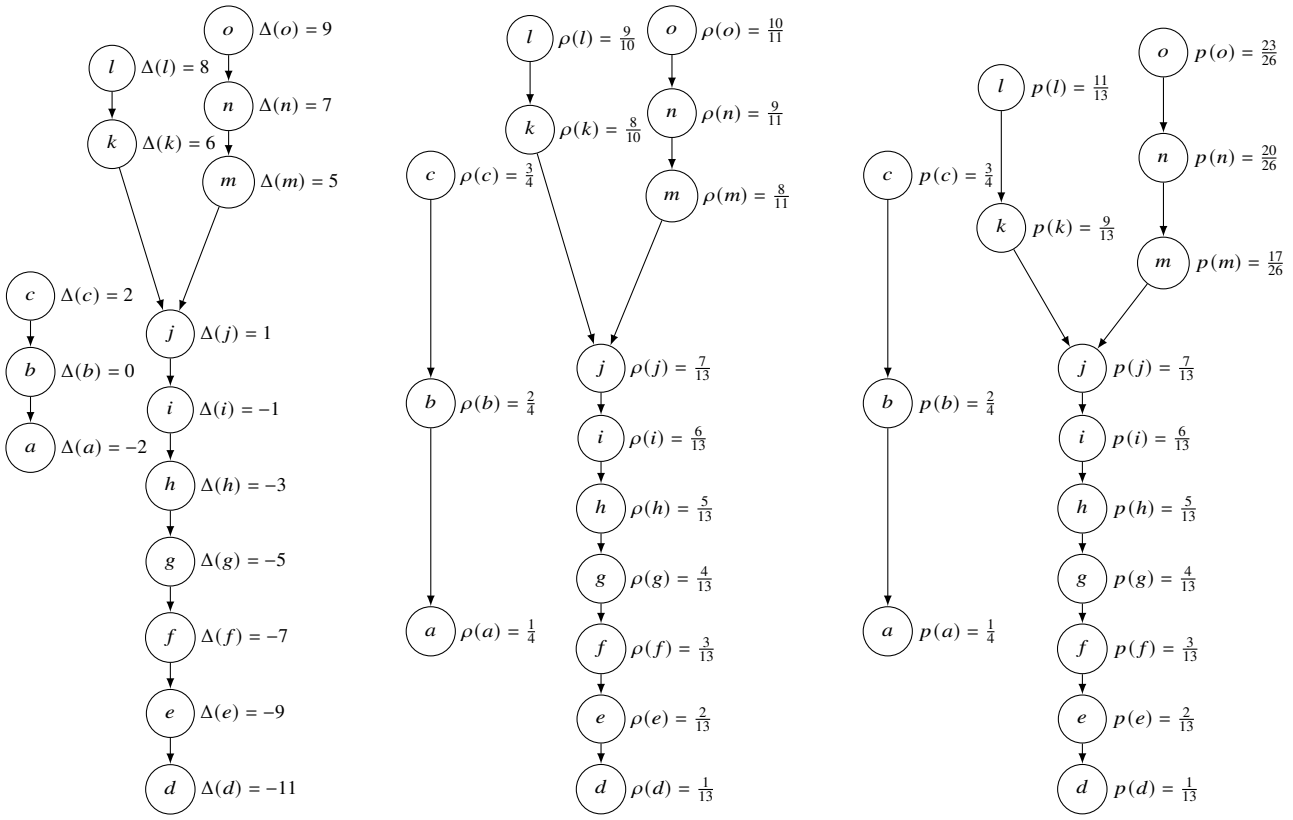
La figure 3 montre sur un exemple simple le résultat des différents estimateurs. Dans ce cas, les trois estimateurs Δ, ρ et p sont optimaux, au sens où ils renvoient le même ordre total que la méthode de Kemeny.

La figure 4 exhibe un cas plus complexe où les quatre estimateurs donnent des résultats distincts. Ici, ρ est meilleur que Δ, notamment car il positionne mieux les éléments de la petite composante connexe (nœuds a à c) par rapport à ceux de la grande (nœuds d à o). Il reste cependant surpassé par l'indicateur p, lui-même moins performant que l'optimum donné par la méthode de Kemeny.

En Section 3, lors de l'optimisation de notre tri Corsort, nous verrons que de manière générale, ρ semble être un

3. D'un point de vue purement formel, quitte à étiqueter chaque élément x par son indice i dans la liste initiale et à noter le résultat obtenu (x, i), on peut supposer que tous les éléments sont distincts. Ainsi, lors du tri de la liste $(17, 42, 42)$, il est possible de se trouver dans une situation où on sait déjà que $(17, 1) \preceq_k (42, 2)$ mais où on ne sait pas encore comparer $(17, 1)$ et $(42, 3)$.

4. En particulier, quand on connaît le résultat de toutes les comparaisons, classer les éléments selon l'estimateur Δ ou ρ renvoie bien la liste triée.



(a) Scores Δ. (b) Scores ρ. (c) Scores p.

Estimateur	Estimation renvoyée	$\overline{\tau}$
Δ	$(defghaibjcmknlo)$	10,0
ρ	$(defaghibjmcknlo)$	8,91
p	$(defaghibjmkcnlo)$	8,66
Kemeny (optimal)	$(deafghibjmknclo)$	8,61

FIGURE 4 – Exemple de sélection d'extension linéaire par attribution de score. L'estimateur p (à droite) est normalisé par $n+1 = 16$ afin de faciliter la comparaison avec ρ (au centre). Le tableau regroupe les estimations renvoyées et l'espérance $\overline{\tau}$ de l'erreur τ. La méthode de Kemeny (optimale) n'attribue pas de score aux éléments et n'a pas de figure associée, mais est également donnée pour comparaison. La position verticale de chaque nœud est proportionnelle à la valeur de l'estimateur considéré (Δ, ρ ou p).

meilleur estimateur que Δ. Mais nous verrons également que Δ présente un autre type d'intérêt.

3 Tris orientés comparaisons (Corsort)

Nous appelons tri *Corsort* (*Comparison-ORiented Sort*) un tri qui, à chaque étape de son exécution, sélectionne la comparaison suivante en fonction de l'ordre partiel courant $\preceq_k$. On suppose qu'on choisit toujours des paires non comparables selon $\preceq_k$. Cela assure de ne jamais effectuer deux fois la même comparaison, donc de terminer en au plus $n(n-1)/2$ étapes, et plus généralement de ne jamais effectuer des comparaisons déductibles par transitivité.

Le cœur d'un corsort, c'est-à-dire le choix de la prochaine comparaison, vise deux objectifs. D'une part, il doit assurer une terminaison rapide, c'est-à-dire minimiser le nombre total de comparaisons. C'est un problème de *tri sous information partielle*, qui revient à choisir une comparaison dont les deux issues sont aussi équiprobables que possible [5]. À cette fin, nous proposons d'utiliser un estimateur basé sur une notion de score et de comparer des éléments dont les scores sont proches. En effet, cela indique une grande incertitude quant au résultat de la comparaison, ce qui rapproche de l'équiprobabilité souhaitée. D'autre part, pour améliorer

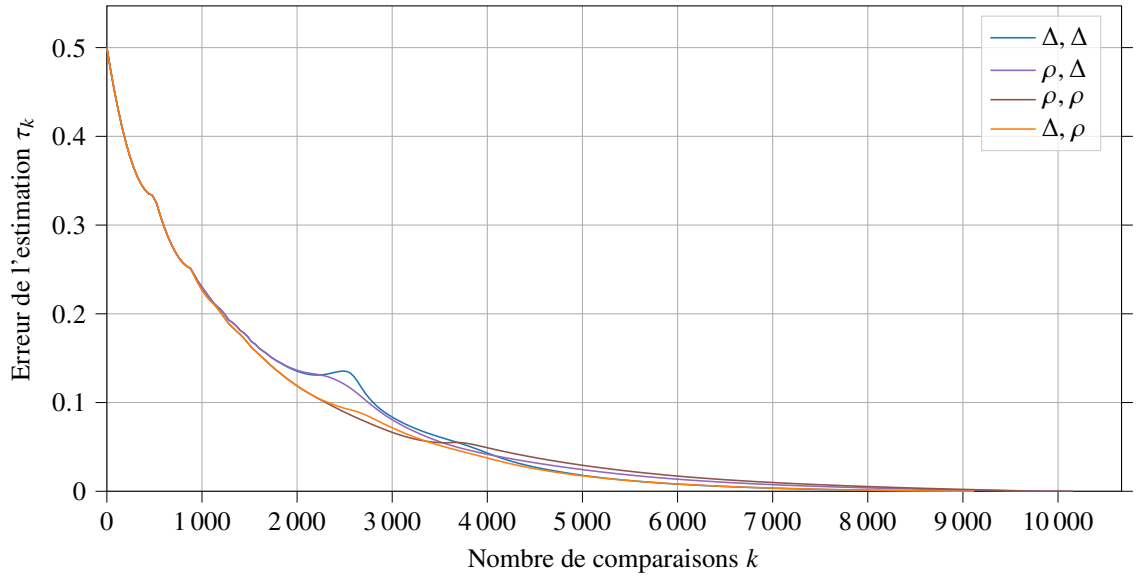


FIGURE 5 – Profils de performances de tris Corsort pour $n = 1\,000$. Chaque courbe est obtenue en triant 10 000 listes aléatoires. Pour chaque valeur de k, on calcule l'erreur τ_k médiane, normalisée par $n(n-1)/2$. En légende, chaque Corsort est défini par le critère principal à minimiser pour déterminer la prochaine comparaison, puis l'estimateur utilisé pour renvoyer X_k.

Algorithme 1 : Corsort sélectionné

$X \leftarrow$ une liste de n éléments munie d'un ordre total
$k \leftarrow 0$
$\preceq_k \leftarrow$ l'ordre partiel vide sur X
tant que $\preceq_k$ *est incomplet et pas d'interruption* **faire**
  $i, j \leftarrow \arg\min_{i,j \text{ non comparables dans } \preceq_k} \left(|\Delta_k(i) - \Delta_k(j)|, \max(I_k(i), I_k(j))\right)$
  $\preceq_{k+1} \leftarrow$ clôture transitive de $\preceq_k$ augmenté de la comparaison i, j
  $k \leftarrow k + 1$
retourner X *trié selon l'estimateur* ρ_k

la qualité des estimations intermédiaires X_k, il faut acquérir de l'information sur les éléments pour lesquels on en a peu, car ces derniers créent une incertitude qui va se répercuter sur τ_k. Pour représenter la quantité d'information acquise sur un sommet x, on introduit donc $I_k(x) = a_k(x) + d_k(x)$, et on souhaite comparer des éléments pour lesquels I_k est faible. Empiriquement, nous avons constaté que le premier objectif devait être prioritaire sur le second et avons opté pour une sélection lexicographique. En résumé :

— Le choix de la prochaine comparaison se fait en cherchant à minimiser l'écart des scores selon un estimateur (Δ ou ρ dans nos expériences).
— En cas d'égalité, on cherche à comparer une paire (x, y) pour laquelle $I_k(x)$ et $I_k(y)$ sont faibles. À cette fin, nous cherchons une paire qui minimise $\max(I_k(x), I_k(y))$.
— Les valeurs de X_k sont données par un estimateur, Δ ou ρ, qui n'est pas nécessairement le même que pour choisir la prochaine comparaison.

Afin de déterminer la meilleure combinaison possible, nous avons développé un paquet Python pour créer des tris interruptibles et mesurer leurs performances [4]. Ce paquet permet, entre autres choses, d'étudier le déroulement d'un algorithme de tri. Nous avons ensuite testé les différentes variantes de Corsort par simulation. La figure 5 montre les profils de performance médians obtenus pour $n = 1\,000$. On observe que tous les Corsorts utilisant l'estimateur Δ présentent une « bosse » entre $k = 2\,000$ et $k = 3\,000$, en particulier quand la prochaine comparaison est aussi déterminée par Δ. Cela nous amène à choisir ρ pour les estimations X_k (nous avons vérifié que ρ est aussi globalement plus performant pour les algorithmes de tris classiques). À l'inverse, Δ est presque toujours plus intéressant que ρ pour choisir la prochaine comparaison, en particulier dans la deuxième moitié de l'exécution du tri [5].

5. Non montré par souci de lisibilité de la figure 5 : pour le critère secondaire de choix de la prochaine comparaison, à la place de $\max(I_k(x), I_k(y))$, on peut utiliser $I_k(x) + I_k(y)$. Notre choix d'uti-

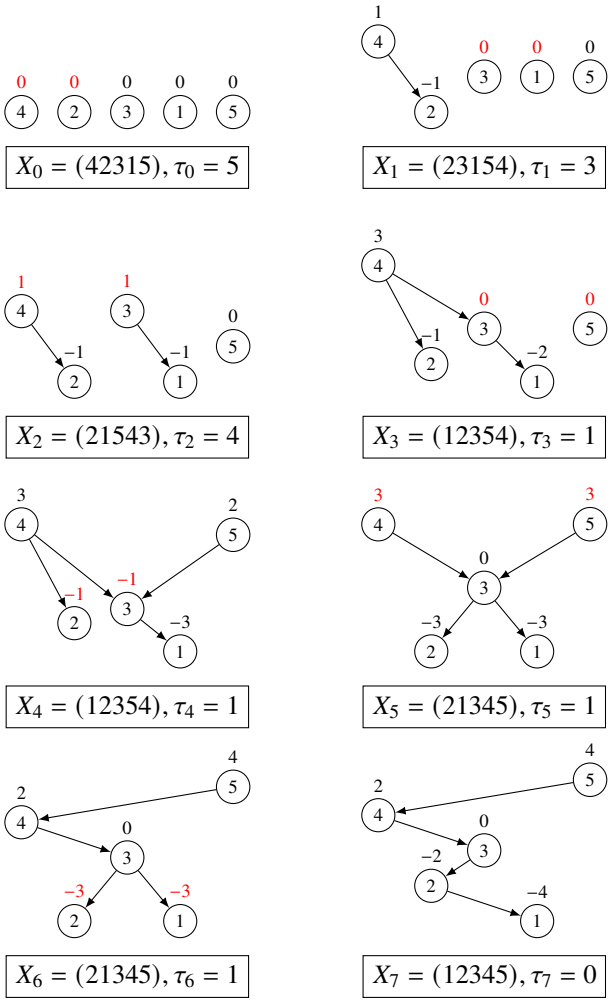


FIGURE 6 – Exécution du Corsort sélectionné sur la liste $X = (42315)$. Chaque étape k montre l'ordre partiel courant après k comparaisons, Δ_k (indiqué au-dessus de chaque élément), ρ_k (représenté par sa hauteur), l'estimation X_k renvoyée et l'erreur τ_k. À chaque étape, la prochaine comparaison se fera entre les deux sommets dont les valeurs de Δ_k sont en rouge.

Nous avons donc choisi le Corsort décrit par l'algorithme 1 : parmi les paires encore incomparables, choisir la paire (x, y) qui minimise lexicographiquement $\big(|\Delta_k(x) - \Delta_k(y)|, \max(I_k(x), I_k(y))\big)$; et utiliser ρ comme estimateur. À titre d'exemple, la figure 6 montre l'exécution du Corsort sélectionné sur la liste $X = (42315)$. En l'occurrence, on retrouve la quasi-monotonie du profil de performance déjà constaté de manière plus générale dans la figure 5.

liser le maximum améliore la performance mais de manière marginale.

4 Évaluation

À présent, nous souhaitons comparer notre tri Corsort avec les algorithmes classiques présentés en section 2.1. Notre méthodologie d'évaluation est la suivante : pour une valeur de n donnée, nous tirons 10 000 listes aléatoires et calculons le nombre de comparaisons nécessaires pour un tri complet, ainsi que les profils de performance $k \longrightarrow \tau_k$. Pour donner un aperçu de la distribution des résultats, nous traçons pour chaque algorithme la médiane (courbe foncée), les quantiles de 25% à 75% (zone claire), et les quantiles de 2,5% à 97,5%, qui représentent un intervalle de confiance à 95% (zone très claire).

La figure 7 montre le temps de terminaison (en nombre de comparaisons) pour des valeurs de n allant de 8 à 1024 et les tris suivants : par tas, rapide, Corsort, fusion et Ford-Johnson. L'axe des ordonnées montre l'écart relatif par rapport à la borne inférieure théorique $n \log_2(n) - n/\ln(2) + \log_2(2\pi n)/2$ [6] : plus une courbe est proche de 0, plus elle est proche de l'optimum.

Nous observons que le tri par tas effectue presque deux fois plus de comparaisons que nécessaire. Le tri rapide est meilleur (moins de 30% de surcoût) mais avec une grande variance : pour $n = 1024$, l'intervalle de confiance à 95% va de 17% à 46%. Cette grande variance est due au choix du pivot, dont la valeur influe grandement sur la rapidité de l'algorithme. À chaque étape, plus le pivot est proche de la médiane (de la sous-liste courante considérée), plus l'algorithme sera rapide. Les trois tris restants ont un surcoût encore plus faible (5% pour Corsort, 2% pour le tri fusion, 0,3% pour Ford-Johnson) et une variance négligeable. Nous concluons que Corsort est un bon candidat puisqu'il n'est battu que par des algorithmes dont la terminaison est asymptotiquement optimale, i.e. équivalente à $n \log_2(n)$ [8].

La figure 8 montre les profils de performance des quatre meilleurs tris de la figure 7 : tri rapide, Corsort, tri fusion, et algorithme de Ford-Johnson. Ce dernier, tout comme Corsort, utilise l'estimateur ρ. Pour les tris rapide et fusion, deux variantes sont utilisées : la version de base avec son estimation naturelle (l'état courant de la liste), et une version améliorée (respectivement fusion-BFS et ASort) munie de l'estimateur ρ.

Premièrement, on voit que le tri rapide et Asort ont une très grande variance. Ceci s'explique encore par le système de pivot, qui influe sur la structure des comparaisons effectuées, et donc sur la qualité de l'estimation. Les variances des tris fusion, fusion-BFS muni de ρ et Ford-Johnson muni de ρ sont assez faibles, tout comme Corsort, dont la variance est quasiment négligeable. Ces tris sont donc plus robustes que le tri rapide et Asort.

Ensuite, on remarque que l'utilisation de l'estimateur ρ améliore le profil de performance [6]. Pour le tri fusion, cela

6. Non montré par souci de lisibilité de la figure : Δ améliore également le profil des tris classiques, mais est généralement moins performant que ρ.

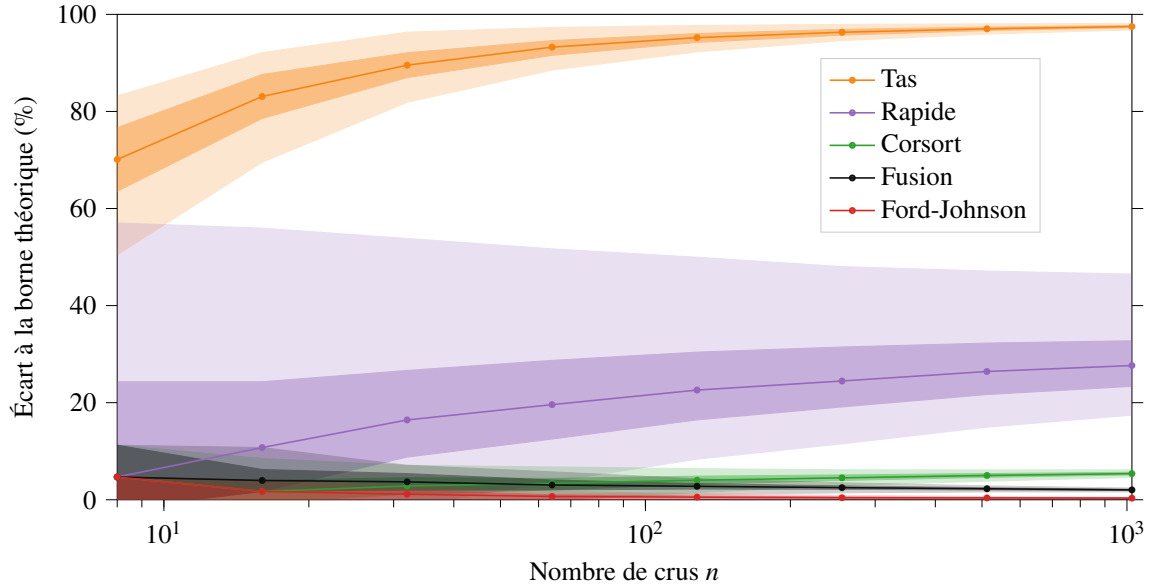


FIGURE 7 – Nombre de comparaisons pour terminer l'algorithme, exprimé en écart par rapport à la borne théorique. Chaque point est obtenu en triant 10 000 listes aléatoires.

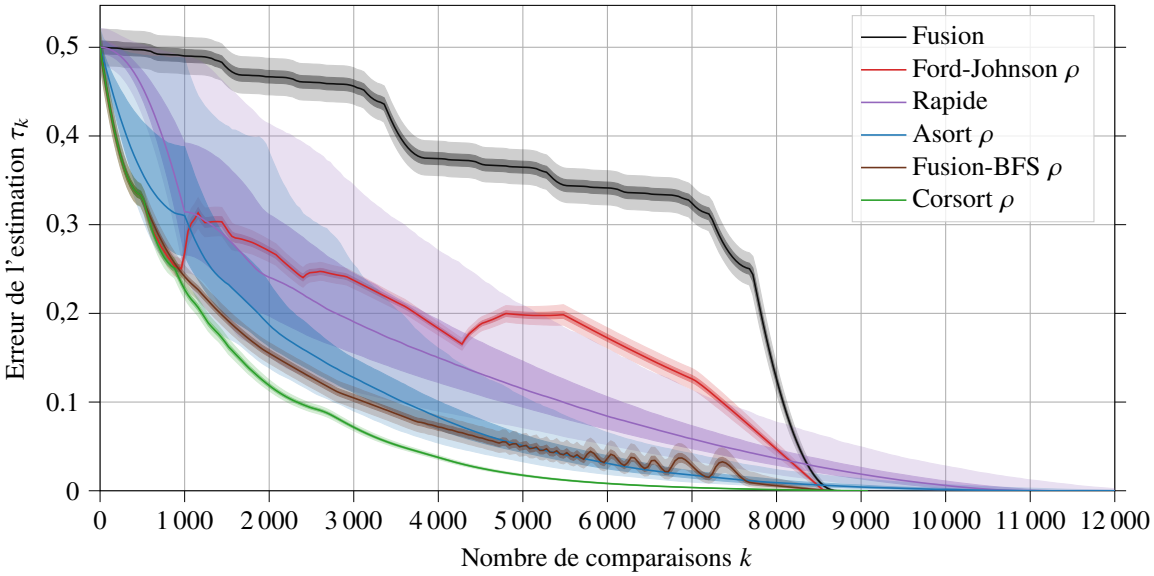


FIGURE 8 – Profils de performance de tris interruptibles pour $n = 1000$. Chaque courbe est obtenue en triant 10 000 listes aléatoires. L'erreur τ est normalisée par $n(n-1)/2$.

se fait au prix d'un comportement non-monotone en fin de parcours [7]. Cependant, grâce à une implémentation BFS et à notre estimateur, le profil de performance du tri fusion amélioré reste presque constamment en dessous du tri rapide, que celui-ci soit en version naïve ou améliorée (Asort). En plus d'améliorer les estimations, l'estimateur ρ réduit aussi la variance (qui reste tout de même conséquente pour Asort).

Enfin, il ressort la supériorité du profil de Corsort : il est presque tout le temps monotone, avec une très faible variance, et il est constamment sous les autres à part en terminaison (il termine un peu plus tard que fusion ou Ford-

7. La non-monotonie du tri fusion-BFS muni de ρ se produit au moment de la fusion des plus grandes sous-listes, ce qui donne lieu à des ordres partiels similaires à l'exemple de la figure 4, avec des composantes connexes en forme de chaîne et d'autre en forme de Y. Les deux dernières « bosses » de la figure 8 correspondent aux fusions des quarts de liste ; les quatre précédentes, aux huitièmes de listes. La dernière fusion n'occasionne pas de non-monotonie car l'estimateur ρ est performant sur une configuration en Y si celle-ci n'est pas accompagnée d'une chaîne à côté.

Johnson). Corsort est donc un excellent tri interruptible que nous recommandons à Agathe pour trier ses crus de Riesling. Enfin, on peut constater la relative bonne performance du tri fusion-BFS muni de ρ, qui peut être un choix intéressant si l'on désire un tri interruptible qui soit également rapide, avec une terminaison, hors estimateur, en $O(n\log(n))$ opérations totales (pas seulement en nombre de comparaisons).

5 Conclusion et travaux futurs

Nous avons étudié des tris interruptibles générant un minimum de comparaisons. Nous avons proposé une méthode pour rendre tout tri interruptible, avec interruption possible après chaque comparaison. Nous avons introduit Corsort, une famille de tris à base d'estimateurs. Par simulation, nous avons montré qu'un tri Corsort bien configuré a un temps de terminaison (en nombre de comparaisons) quasi-optimal et possède un profil de performance meilleur que les meilleurs tris dont nous avons connaissance. Nous avons aussi montré que munir les tris classiques de nos estimateurs améliore leurs profils de performance.

La relative nouveauté de notre approche laisse place à plusieurs pistes de réflexion pour des travaux futurs.

D'abord, il sera intéressant de réaliser des nouvelles simulations avec d'autres fonctions de score pour essayer d'améliorer la terminaison de notre tri Corsort et/ou son profil de performance. Il sera aussi souhaitable de trouver des fonctions de score qui améliorent encore les profils de performance des tris classiques, en particulier qui tendent à rendre monotones l'algorithme de Ford-Johnson et le tri fusion.

À terme, on voudra essayer de répondre à plusieurs questions encore ouvertes. Le comportement en pire cas du tri Corsort choisi est-il en $O(n\log(n))$? Quelle est la borne théorique du profil de performance ? Peut-on trouver d'autres estimateurs raisonnables pour qu'un tri Corsort s'en approche ?

Références

[1] Alewijnse, S. P. A., T. M. Bagautdinov, M. de Berg, Q. W. Bouts, A. P. ten Brink, K. Buchin et M. A. Westenberg: *Progressive Geometric Algorithms*. Dans *Proceedings of the thirtieth annual symposium on Computational geometry*, pages 50–59, Kyoto Japan, juin 2014. ACM, ISBN 978-1-4503-2594-3. https://dl.acm.org/doi/10.1145/2582112.2582156.

[2] Bartholdi, John, Craig A Tovey et Michael A Trick: *Voting schemes for which it can be difficult to tell who won the election*. Social Choice and welfare, 6 :157–165, 1989.

[3] Brightwell, Graham et Peter Winkler: *Counting linear extensions is #P-complete*. Dans *Proceedings of the twenty-third annual ACM symposium on Theory of computing - STOC '91*, pages 175–181, New Orleans, Louisiana, United States, 1991. ACM Press, ISBN 978-0-89791-397-3. http://portal.acm.org/citation.cfm?doid=103418.103441.

[4] Caizergues, Emma, François Durand et Fabien Mathieu: *Corsort : Comparaison ORiented Sort*. https://emczg.github.io/corsort/, 2023.

[5] Cardinal, Jean, Samuel Fiorini, Gwenaël Joret, Raphaël M. Jungers et J. Ian Munro: *Sorting under partial information (without the ellipsoid algorithm)*. Dans *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 359–368, Cambridge Massachusetts USA, juin 2010. ACM, ISBN 978-1-4503-0050-6. https://dl.acm.org/doi/10.1145/1806689.1806740.

[6] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest et Clifford Stein: *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009, ISBN 9780262533058.

[7] Dushkin, Eyal et Tova Milo: *Top-k Sorting Under Partial Order Information*. Dans *Proceedings of the 2018 International Conference on Management of Data*, pages 1007–1019, Houston TX USA, mai 2018. ACM, ISBN 978-1-4503-4703-7. https://dl.acm.org/doi/10.1145/3183713.3199672.

[8] Ford, Lester R. et Selmer M. Johnson: *A Tournament Problem*. The American Mathematical Monthly, 66(5) :387–389, mai 1959, ISSN 0002-9890, 1930-0972. https://www.tandfonline.com/doi/full/10.1080/00029890.1959.11989306.

[9] Giesen, Joachim, Eva Schuberth et Miloš Stojaković: *Approximate Sorting*. Fundam. Inf., 90(1–2) :67–72, jan 2009, ISSN 0169-2968.

[10] Grass, Joshua et Shlomo Zilberstein: *Anytime Algorithm Development Tools*. SIGART Bulletin Special Issue on Anytime Algorithms and Deliberation Scheduling, 7(2) :20–27, 1996. http://rbr.cs.umass.edu/shlomo/papers/GZsigart96.html.

[11] Hoare, C. A. R.: *Algorithm 65 : find*. Communications of the ACM, 4(7) :321–322, 1961.

[12] Horvitz, Eric: *Reasoning Under Varying and Uncertain Resource Constraints*. Morgan Kaufmann Publishers, janvier 1988. https://www.microsoft.com/en-us/research/publication/reasoning-under-varying-and-uncertain-resource-

[13] Kemeny, John G: *Mathematics without numbers*. Daedalus, 88(4) :577–591, 1959.

[14] Kendall, M. G.: *A New Measure of Rank Correlation*. Biometrika, 30(1/2) :81, juin

1938, ISSN 00063444. https://www.jstor.org/stable/2332226?origin=crossref.

[15] Mesrikhani, Amir et Mohammad Farshi: *Progressive sorting in the external memory model*. The CSI Journal on Computer Science and Engineering, 15(2), 2018.

[16] Peczarski, Marcin: *New Results in Minimum-Comparison Sorting*. Algorithmica, 40(2) :133–145, octobre 2004, ISSN 0178-4617, 1432-0541. http://link.springer.com/10.1007/s00453-004-1100-7.

[17] Peters, Dominik et Ariel D Procaccia: *Preference Elicitation as Average-Case Sorting*. Dans *Proceedings of the AAAI Conference on Artificial Intelligence*, tome 35, pages 5647–5655, 2021.

[18] Zilberstein, Shlomo: *Using Anytime Algorithms in Intelligent Systems*. AI Magazine, 17(3) :73, mars 1996. https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/1232, Section : Articles.