Université de Montréal

De nouveaux algorithmes de tri par transpositions

par
Maxime Benoît-Gagné

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

juin, 2007

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

De nouveaux algorithmes de tri par transpositions

présenté par:

Maxime Benoît-Gagné

a été évalué par un jury composé des personnes suivantes:

Nadia El-Mabrouk
président-rapporteur

Sylvie Hamel
directeur de recherche

Yann-Gaël Guéhéneuc
membre du jury

Mémoire accepté le 20 septembre 2007

RÉSUMÉ

L'opération qui coupe un segment d'une permutation et le colle ailleurs s'appelle une transposition. Trier une permutation par transpositions est chercher la séquence minimale de transpositions qui la transforme en la permutation identité. Nous expliquons, dans ce mémoire, pourquoi ce problème peut avoir des applications en phylogénie quand on modélise une molécule d'ADN par une permutation.

Aujourd'hui, nous ignorons si un algorithme en temps polynomial existe pour trier une permutation par transpositions. Dans ce mémoire, nous présentons des algorithmes heuristiques de tri par transpositions provenant de la littérature. Nous décrivons ensuite de nouveaux algorithmes heuristiques en temps polynomial qui proviennent de l'auteur. Ces algorithmes utilisent des séquences de nombres, appelées "codes", calculées à partir de la permutation. Ils utilisent certains "motifs" caractéristiques de ces codes que nous nommons "objets". Pour chaque objet défini, nous expliquons comment celui-ci peut être traité pour décider d'une transposition. Nous prouvons aussi la complexité en temps de ce traitement. Nous expliquons ensuite comment nous agençons ces objets pour créer nos algorithmes.

Notre algorithme le plus simple a une garantie de performance de 3 et une complexité en temps dans $O(n^2)$. Nous expliquons comment nous avons testé certains de nos algorithmes sur des permutations de différentes longueurs et concluons que nos algorithmes sont plus rapides que ceux déjà existants mais approximent moins bien la distance de transposition.

Mots clés : Bio-informatique, réarrangement génomique, distance de tranposition, permutation, algorithme heuristique, complexité algorithmique.

ABSTRACT

The operation that cuts a segment of a permutation and pastes it elsewhere is called transposition. Sorting a permutation by transpositions is searching for the minimal sequence of transpositions that transforms it into the identity permutation. We explain, in this thesis, why this problem can have applications in phylogeny when we model genomic rearrangements on a DNA molecule by operations on a permutation.

At this time, it is not known if a polynomial time algorithm exists to sort a permutation by transpositions. In this thesis, we present several already known approximative algorithms to sort by transpositions. We then describe our new polynomial time approximative algorithms. These algorithms use sequences of numbers, that we name "codes", computed from the permutation and use some characteristic "patterns" of these codes that we name "objects". For each defined object, we explain how it can be processed in order to decide of an efficient transposition. We then prove the time complexity of this processing and explain how we fit together these objects in order to create our algorithms.

Our simplest algorithm have an approximation ratio of 3 and a time complexity in $O(n^2)$. We explain how we tested some of our algorithms on permutations of diverse lengths and conclude that our algorithms are faster than those already existing but give worse results in terms of the transposition distance.

Keywords: Bioinformatics, genome rearrangement, transposition distance, permutation, approximative algorithm, algorithmic complexity.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES APPENDICES

LISTE DES SIGLES

ADN	Acide DésoxyriboNucléique
LIS	Sous-séquence non contiguë croissante la plus longue
$\mathrm{TMAF_AF_SDMP}$	*TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau*
TMP	*TrieMontagnePlateau*
TP	*TriePlateau*
TPG	*TriePlateauGauche*

NOTATION

À mes parents

REMERCIEMENTS

J'aimerais d'abord remercier ma directrice de recherche, Sylvie Hamel, de m'avoir guidé le long de cette maîtrise. Il m'a été agréable d'apprendre à faire de la recherche avec elle grâce à sa gentillesse et à sa disponibilité.

Je tiens aussi à remercier mes parents de leur soutien.

Merci à tous les membres du laboratoire LBIT de l'atmosphère qu'ils ont su créée. Je remercie spécialement la stagiaire d'été du CRSNG, Caroline Quinn, qui a participé à mon projet au tout début.

Je remercie finalement le CRSNG et ma directrice de recherche de leur soutien financier.

AVANT-PROPOS

Le but de ce projet de recherche était de concevoir un algorithme plus simple que ceux déjà existants pour le problème du tri par transpositions. Pendant que je rédigeais ce mémoire de maîtrise, Sylvie Hamel et moi avons écrit le manuscrit *A new and faster method of sorting by transpositions* [BGH07]. Nous y présentons nos résultats plus succintement que dans le présent document.

CHAPITRE 1

INTRODUCTION

1.1 Présentation du problème

Nous débutons notre mémoire par une mise en situation. Jean est content aujourd'hui. C'est son anniversaire. Il a six ans. Il a reçu une belle boîte pleine de cubes. Les cubes sont de toutes les couleurs. Une lettre ou un chiffre est écrit sur chaque face de chaque cube. Jean choisit neuf cubes et les tourne de manière à avoir un chiffre sur la face du dessus. Jean a choisi neuf cubes de manière à avoir les chiffres de 1 à 9. Il aligne les cubes devant lui sans les ordonner (voir Figure 1.1 a).

Jean choisit une suite de trois cubes collés les uns aux autres dans la ligne de cubes. Il les pousse devant lui. Cela fait un trou dans la ligne de cubes (voir Figure 1.1 b). Il pousse ensuite les deux cubes qui sont à droite du trou pour qu'ils touchent l'extrémité gauche du trou. Jean a ainsi déplacé le trou vers la droite (voir Figure 1.1 c). Il décide de boucher ce trou en ramenant vers lui la suite de trois cubes qu'il avait mise de côté en prenant soin de ne pas modifier leur ordre respectif (voir Figure 1.1 d).

Jean répète cette opération plusieurs fois en prenant n'importe quelle suite de cubes de n'importe quelle longueur dans la ligne de cubes. Jean a découvert l'opération nommée transposition.

Jean essaie de replacer les cubes en ordre croissant de 1 à 9 en ne faisant que des transpositions. Il tente de trouver les transpositions qui permettent de replacer les cubes en ordre à partir d'un alignement de départ. Jean essaie de trier par transpositions.

Pour comprendre l'opération de transposition, il suffit de remplacer le mot cube par le mot élément, l'expression ligne de cubes par le mot permutation et l'expression suite de cubes par le mot segment. Évidemment, il peut y avoir plus que neuf cubes (éléments).

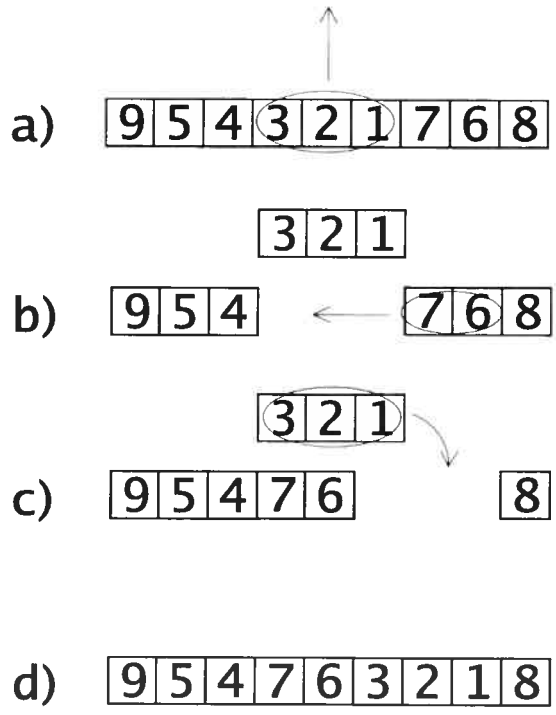


FIG. 1.1 – Cubes de Jean.

a) Les cubes alignés. Une suite de cubes sera mise de côté. b) Le nouvel alignement des cubes. Une suite de cubes à droite du trou sera déplacée à gauche du trou. c) Le nouvel alignement des cubes. La suite de cubes mise de côté sera réinsérée. d) L'alignement final des cubes.

Le concept de permutation peut servir de modèle en biologie. Par exemple, deux bouts d'ADN contenant les mêmes gènes dans des ordres différents peuvent être modélisés par deux permutations distinctes. Si les bouts d'ADN sont bicaténaires, c'est-à-dire s'ils contiennent deux brins, le brin sur lequel le gène se trouve est représenté par le signe, + ou -, devant l'élément de la permutation correspondant au gène. La transposition se fait sur une permutation non signée. On peut essayer de transformer un brin en l'autre (une permutation en l'autre) en triant par transpositions. Cela modélise un scénario évolutionnaire possible. Ce problème spécifique du tri par transpositions est le sujet de ce mémoire. Un joueur de cartes qui classe sa main de cartes fait aussi du tri par transpositions [EEK+01].

1.2 Reconstruction d'une phylogénie

Un scénario possible pour retracer l'évolution de certaines espèces s'appelle une phylogénie. Celle-ci est souvent représentée sous la forme d'un arbre de phylogénie. L'étude de la construction de tels scénarios s'appelle aussi la phylogénie.

Les hypothèses simplificatrices principales à la base de l'étude de la phylogénie sont regroupées sous le nom de théorie de l'évolution. Dans une espèce, chaque cellule contient les mêmes molécules d'ADN. Toutes les espèces actuelles proviennent d'un même ancêtre. La diversité entre les espèces est due à la spéciation (la séparation entre deux espèces). Les caractères sont transmis d'une génération à l'autre. Ils subissent une série de mutations au cours de l'évolution. Les variations observées proviennent d'événements appelés mutations ponctuelles, réarrangements génomiques et transferts latéraux. Une mutation ponctuelle touche une séquence de nucléotides ou d'acides aminés. Nous appelons réarrangement génomique une mutation qui touche l'ordre des gènes. Un gène qui passe directement d'une espèce A à une espèce B est un transfert latéral de gènes [SM97]. Dans une situation typique d'étude de phylogénie, on veut construire un arbre de phylogénie à partir des "distances" entre des séquences génomiques appartenant à plusieurs espèces. Plusieurs méthodes de calcul de distances entre séquences génomiques ont été développées.

A	a	c	_	g	t
B	c	c	c	g	_
	1	2	3	4	5

FIG. 1.2 – Types de mutations ponctuelles.
A et B sont deux séquences de caractères. Un gap ('_') signifie qu'il n'y a pas de caractère à cette position. a) Changement à la position 1. b) Insertion à la position 3. c) Suppression à la position 5.

1.2.1 Méthodes de mutations ponctuelles

Les méthodes dites de mutations ponctuelles représentent une séquence génétique par une séquence de caractères (une séquence de nucléotides ou d'acides aminés). Ces méthodes comparent les séquences en cherchant des différences au niveau de la position et de la nature des caractères.

Il y a trois sortes de différences entre une même position pour deux séquences de caractères. Il peut y avoir un changement d'un caractère (aussi appelé substitution) (voir Figure 1.2 a). Il peut y avoir une insertion d'un caractère (aussi appelée ajout) (voir Figure 1.2 b). Le contraire d'une insertion est une suppression d'un caractère (aussi appelé délétion) (voir Figure 1.2 c).

Les mutations ponctuelles surviennent relativement fréquemment au cours de l'évolution. C'est pourquoi deux génomes provenant d'une ancienne spéciation peuvent devenir très différents avec le temps. Des méthodes ponctuelles pour construire un arbre de phylogénie risquent alors de donner des résultats peu fiables.

1.2.2 Génomique comparée

Les méthodes de génomique comparée peuvent être la solution lorsque les méthodes de mutations ponctuelles sont moins adéquates. La génomique comparée compare le contenu, l'organisation et la fonction des gènes de deux génomes [Mou01]. Des mutations affectant l'ordre des gènes sont plus rares que des mutations ponctuelles. C'est pourquoi des mutations étudiées par la génomique comparée peuvent donner des indices plus robustes sur l'évolution [Har03].

Un problème de la génomique comparée est la recherche de gènes qui ont une fonction identique. Comment déterminer si un gène a du génome A possède une fonction identique au gène b du génome B ?

Les concepts de gènes orthologues et paralogues sont utiles pour résoudre ce problème. Deux gènes sont orthologues si l'événement le plus récent qui les réunis est un événement de spéciation. Deux gènes sont paralogues si l'événement le plus récent qui les réunis est une duplication. En simplifiant, on considère que les gènes orthologues ont une fonction identique alors que les gènes paralogues n'en ont pas [DEKM98]. Les questions concernant le caractère identique de la fonction des gènes et de la détermination des gènes orthologues sont vastes et débordent du sujet de ce mémoire. Le problème que ce mémoire analyse nécessite que ces problèmes aient déjà été résolus. Nous allons supposer que c'est le cas à partir de maintenant.

La génomique comparée peut servir à tenter de reconstruire la séquence d'événements évolutionnaires qui ont transformé un génome en un autre. L'inversion signée, la transposition et la transposition inverse sont des opérations de réarrangement pouvant survenir dans les génomes monochromosomaux, c'est-à-dire qui ne contiennent qu'un seul chromosome. Les virus, les bactéries, les mitochondries et les chloroplastes ont des génomes à un seul chromosome. L'inversion signée (voir Figure 1.3 a) consiste à renverser la séquence de gènes et à changer leur signe pour indiquer que cette séquence de gènes est transportée sur le brin complémentaire. La transposition (voir Figure 1.3 b) consiste à couper une séquence de gènes et à la coller ailleurs. Il existe toujours 3 inversions qui permettent de faire la même chose que la transposition (voir Figure 1.4 a). Une transposition inverse (voir Figure 1.3 c) consiste en une transposition suivie par une inversion signée sur la même séquence de gènes. Il existe toujours 2 inversions qui permettent de faire la même chose que la transposition inverse (voir Figure 1.4 b).

Les génomes à plusieurs chromosomes sont appelés génomes multichromosomaux. Les génomes des mammifères sont multichromosomaux, par exemple. Certaines opérations telles que la translocation, la fusion et la fission sont spécifiques aux génomes multichromosomaux. Lors d'une translocation, les extrémités des

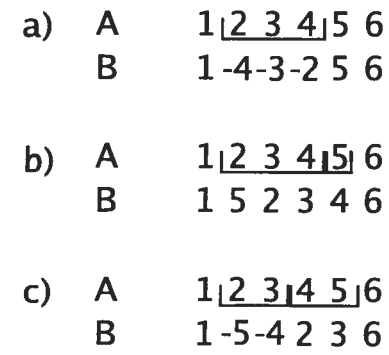


FIG. 1.3 – Opérations de réarrangement dans un génome à un seul chromosome. A et B sont deux génomes différents. a) Inversion signée. b) Transposition. c) Transposition inverse.

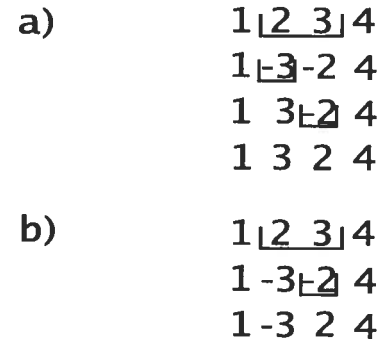


FIG. 1.4 – Séquence d'inversions signées permettant d'atteindre le même résultat qu'une autre opération.

a) 1 transposition faite avec 3 inversion signées. b) 1 transposition inverse faite avec 2 inversions signées.

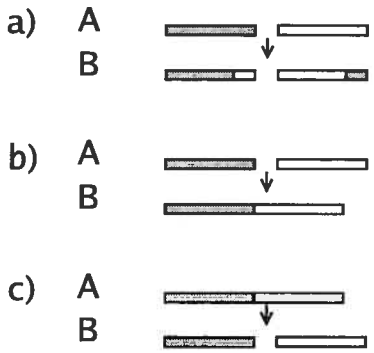


FIG. 1.5 – Opérations de réarrangement dans un génome multichromosomal. A et B sont deux génomes différents. a) Translocation. b) Fusion. c) Fission.

chromosomes A et B sont échangées (voir Figure 1.5 a). La fusion consiste en deux chromosomes qui se collent bout à bout (voir Figure 1.5 b). La fission est un chromosome qui se coupe en deux (voir Figure 1.5 c).

1.3 Hypothèses

Le sujet de ce mémoire est restreint à la transposition. Nous émettons des hypothèses pour simplifier les concepts reliés au tri par transpositions et trouver des solutions algorithmiques au problème de la distance de transposition.

1. Les deux génomes ont le même ensemble de gènes et il n'y a pas de duplications.
2. Toutes les transpositions ont le même coût.
3. Il n'y a que des transpositions.
4. Parcimonie. La séquence de transpositions la plus probable est celle qui implique le nombre minimal de transpositions.

1.3.1 Limites de l'utilisation de la distance de transposition en génomique comparée

Les hypothèses précédentes limitent l'utilisation de la distance de transposition en génomique comparée parce qu'elles peuvent ne pas correspondre à la réalité. Par exemple, l'hypothèse 1 n'est vraie que pour certaines espèces. Voir [HP94, PH86] pour des exemples de telles espèces. Cependant, ce n'est pas le cas pour plusieurs espèces qui ont des ensembles de gènes différents et possiblement plusieurs copies de ces gènes. On peut alors satisfaire la première hypothèse en restreignant notre ensemble de gènes à ceux présents dans les deux espèces et en choisissant, arbitrairement, une copie de chacun d'eux.

La distance de transposition n'est alors qu'un indice parmi d'autres pour retracer les événements évolutionnaires ayant transformé un génome en un autre. Évaluer la pertinence biologique de tels indices est un autre problème beaucoup plus compliqué pour lequel il existe, à notre connaissance, peu d'outils à ce jour.

1.4 Définitions

Nous allons maintenant présenter les définitions formelles qui seront utilisées tout au long de ce mémoire. Une **permutation** π est une bijection de l'ensemble $[n] = \{1, 2 \ldots, n\}$ sur lui-même. Une permutation π de $[n]$ sera dénotée par $\pi = \pi_1 \pi_2 \ldots \pi_n$.

Une **transposition** est une opération sur les permutations qui déplace un bloc d'éléments contigus et le place ailleurs. Plus formellement, pour $1 \leq i \leq j \leq n+1$ et $j < k \leq n+1$ (ou $1 \leq k < i$), on définit la **transposition** $\rho(i, j, k)$ sur π par

$$\rho(i, j, k) \cdot \pi = \pi_1 \ldots \pi_{i-1} \pi_j \ldots \pi_{k-1} \pi_i \ldots \pi_{j-1} \pi_k \ldots \pi_n$$

c'est-à-dire que le bloc allant de π_i à π_{j-1} (inclusivement) a été enlevé et placé juste avant π_k. La figure 1.6 en présente un exemple.

Trier par transpositions revient à transformer π en la permutation identité notée

$$\pi^1 = 6\ 3\ 2\ 1\ 4\ 5$$

$$\pi^1 \cdot \rho(2, 7, 1) = 3\ 2\ 1\ 4\ 5\ 6$$

FIG. 1.6 – Exemple d'une transposition.

Id en n'utilisant que des transpositions. À partir de n'importe quelles permutations de départ π^1 et π^2, on peut renommer les éléments de π^1 et π^2 de manière à obtenir $\pi^{1'}$ et $\pi^{2'}$ où $\pi^{2'} = \text{Id}$. Transformer π^1 en π^2 revient alors à trier $\pi^{1'}$.

La **distance de transposition** de π, notée $d(\pi)$, est le nombre minimal de transpositions nécessaires pour trier π. Notre but est de calculer $d(\pi)$ à partir de π en trouvant une séquence optimale de transpositions. Le statut de complexité du tri par transpositions est un problème ouvert. Il n'existe pas d'algorithme connu capable de calculer la distance de transposition exacte entre deux permutations en temps polynomial. Il n'existe pas non plus de preuve que c'est un problème NP-difficile.

L'ensemble de toutes les permutations possibles de longueur n est le groupe symétrique S_n. Chaque permutation π de S_n est à une certaine distance $d(\pi)$ de Id. Par exemple, Id est à une distance 0 de Id. Le **diamètre de transposition** de n, d(n) est le maximum parmi les $d(\pi)$ telles que $\pi \in S_n$.

1.5 Présentation du mémoire

Le mémoire est organisé comme suit. Le chapitre 2 revoit la littérature sur le sujet du tri par transpositions. Le chapitre 3 présente les concepts à la base de nos algorithmes. Il présente aussi des exemples de nos algorithmes qui ont différentes complexités en temps. Il contient une preuve d'un facteur d'approximation théorique en fonction de la permutation pour notre algorithme le plus rapide. Le chapitre 4 présente les expériences que nous avons menées et leurs résultats. Nous avons comparé certains de nos algorithmes entre eux et avec des algorithmes déjà existants. Une discussion au sujet de l'exactitude de l'approximation et des

temps clos le chapitre. Finalement, le chapitre 5 présente la conclusion du mémoire ainsi que quelques idées de travaux futurs possibles.

CHAPITRE 2

REVUE DE LA LITTÉRATURE

2.1 Introduction

Nous présentons une revue de la littérature sur le sujet du tri par transpositions dans ce chapitre. Nous voulons insister sur le fait qu'il n'existe pas d'algorithme en temps polynomial qui calcule la distance de transposition exacte entre deux permutations à ce jour. Les algorithmes que nous présentons dans cette revue de la littérature sont donc des algorithmes approximatifs qui ont différentes garanties de performance et complexités en temps.

Nous avons choisi d'expliquer plus en détails les algorithmes de Bafna et Pevzner [BP98] à la section 2.2 parce qu'ils sont parmi les premiers à avoir été publiés. De plus, plusieurs algorithmes ultérieurs de tri par transpositions se basent sur les résultats de Bafna et Pevzner. Les algorithmes de Bafna et Pevzner (et ceux qui s'en inspirent) se basent sur une construction appelée graphe de cycles et sur des analyses de cas exhaustives. Nous présentons seulement quelques unes des preuves inclues dans l'article pour éviter d'alourdir le texte. Nous présentons ensuite rapidement d'autres algorithmes de tri par transpositions qui existent dans la littérature à la section suivante. Les auteurs dans le domaine des algorithmes de tri par transpositions publient souvent leur algorithme sans l'avoir implémenté. Ce sont souvent d'autres auteurs qui publient les implémentations et les résultats expérimentaux de ces algorithmes. C'est pourquoi nous consacrons la dernière section de ce chapitre à des publications qui présentent des implémentations des algorithmes que nous aurons préalablement décrits.

2.2 Algorithmes de Bafna et Pevzner

2.2.1 Introduction

Les auteurs de [BP98] furent, à notre connaissance, les premiers à présenter un algorithme de tri par transpositions possédant une garantie de performance aussi bonne que 1,5. Ils présentent trois algorithmes dans leur article. Le premier a une garantie de performance de 2 (voir définition 2.1). Il n'a pas de nom dans l'article. Nous le désignerons dorénavant par *BP2*. Le deuxième est nommé *TSort* par les auteurs et a une garantie de performance de 1,75. *TransSort* est le troisième algorithme et a une garantie de performance de 1,5.

2.2.2 BP2

Nous commencerons par présenter *BP2*. C'est l'algorithme à la base de *TSort* et *TransSort*. Les auteurs ajoutent des cas à analyser à *BP2* pour obtenir ces nouveaux algorithmes approximatifs qui ont une meilleure garantie de performance mais qui sont aussi plus compliqués.

2.2.2.1 L'algorithme

La structure de données qui représente la permutation π de longueur n à la base de l'algorithme est le **graphe de cycles** (*cycle graph*) noté $G(\pi)$. L'ensemble des sommets de $G(\pi)$ est $\{0, 1 \ldots, n+1\}$. Les arêtes sont coloriées et dirigées. Les **arêtes grises** vont de $i-1$ à i et les **arêtes noires** vont de π_i à π_{i-1}, $\forall i$ tel que $1 \leq i \leq n+1$. On numérote les arêtes noires par la position du sommet qui est leur origine.

Un **cycle** est un chemin orienté commençant et se terminant dans le même sommet et dans lequel les arêtes alternent de couleur. On identifie un cycle par les arêtes noires qu'il contient en commençant par celle la plus à droite. On définit $c(\pi)$ comme étant le nombre de cycles présents dans $G(\pi)$.

Les figures 2.1 et 2.2 présentent plusieurs exemples de graphes de cycles. La

permutation identité est celle qui a le nombre maximal de cycles. Le but de l'algorithme sera donc de garantir une certaine augmentation du nombre de cycles.

Nous rappelons que $\rho(i,j,k)$ est une transposition qui insère $\pi_i \dots \pi_{j-1}$ entre π_{k-1} et π_k. Le symbole $\Delta c(\rho)$ représente la variation du nombre de cycles liée à ρ. On a que $\Delta c(\rho) = c(\pi \cdot \rho) - c(\pi)$. Un **x-coup** (*x-move*) est une transposition ρ telle que $\Delta c(\rho) = x$. Un **x-y-coup** est une série de transposition, ρ_1 suivie de ρ_2, où ρ_1 est un x-coup et ρ_2 est un y-coup. Voir les figures 2.1 et 2.2 pour des exemples de 0-coups et de 2-coups.

Les auteurs prouvent le lemme suivant.

Lemme 2.1. $\Delta c(\rho) \in \{-2, 0, 2\}$ ∎

Rappelons que $d(\pi)$ est le nombre minimal de transpositions nécessaires pour trier π. Les auteurs trouvent une borne inférieure à $d(\pi)$.

Théorème 2.2. $d(\pi) \geq \frac{n+1-c(\pi)}{2}$

Preuve. La permutation identité, Id, a $n+1$ cycles. La permutation π de départ a $c(\pi)$ cycles. Une séquence de transpositions qui trient π, c'est-à-dire qui transforment π en Id, doit donc créer $n+1-c(\pi)$ cycles. Selon le lemme 2.1, on peut créer au plus deux cycles par transposition. C'est pourquoi on a besoin d'au moins $\frac{n+1-c(\pi)}{2}$ transpositions pour trier π. ∎

Cependant, il n'est pas toujours possible de faire un 2-coup. Nous allons maintenant introduire des définitions. Un cycle est **orienté** lorsque la séquence des numéros d'arêtes lui correspondant n'est pas décroissante. Sinon, le cycle est **non-orienté**. Le lecteur est encore dirigé vers les figures 2.1 et 2.2 pour voir des exemples de cycles orientés et non-orientés. Nous pouvons maintenant énoncer l'algorithme *BP2*. Un pseudo-code de l'algorithme est donné à l'annexe I.

```
Algorithme BP2(π)
Tant que π ≠ Id
    Si G(π) a un cycle orienté
        Faire un 2-coup (théorème 2.3).
    Sinon
        Faire un 0-2-coup (théorème 2.4).
```

La figure 2.1 présente une trace de l'exécution de cet algorithme.

Le théorème 2.3 prouve l'existence d'un 2-coup à condition que $G(\pi)$ contienne un cycle orienté. Dans le cas contraire, le théorème 2.4 prouve l'existence d'un 0-2-coup. Les transpositions ρ_2 et ρ_4 de la figure 2.1 et les transpositions ρ_6 et ρ_7 de la figure 2.2 sont des exemples de 2-coups sur un cycle orienté.

Théorème 2.3. *Si C est un cycle orienté, alors il existe un 2-coup agissant sur C.* ■

Les séquences de transpositions $\rho_1 \cdot \rho_2$ et $\rho_3 \cdot \rho_4$ de la figure 2.1 et la séquence de transpositions $\rho_5 \cdot \rho_6$ de la figure 2.2 sont des exemples de 0-2-coups sur un cycle non-orienté.

Théorème 2.4. *Pour une permutation π qui n'est pas la permutation identité et qui ne contient pas de cycles orientés, il existe un 0-2-coup.* ■

Ces deux théorèmes permettent aux auteurs de prouver le théorème suivant.

Théorème 2.5. *N'importe quelle permutation π peut être triée en $n + 1 - c(\pi)$ transpositions.*

Preuve. Selon *BP2*, on peut toujours faire soit un 2-coup soit un 0-2-coup. Dans le pire cas, on ne pourra faire que des 0-2-coups. Un 0-2-coup fait augmenter le nombre de cycles de deux en deux transpositions. En moyenne, il crée un cycle par transposition. On a $n + 1 - c(\pi)$ cycles à créer (voir la preuve du théorème 2.2). Dans le pire cas, on pourra donc les créer par *BP2* en $n + 1 - c(\pi)$ transpositions. ■

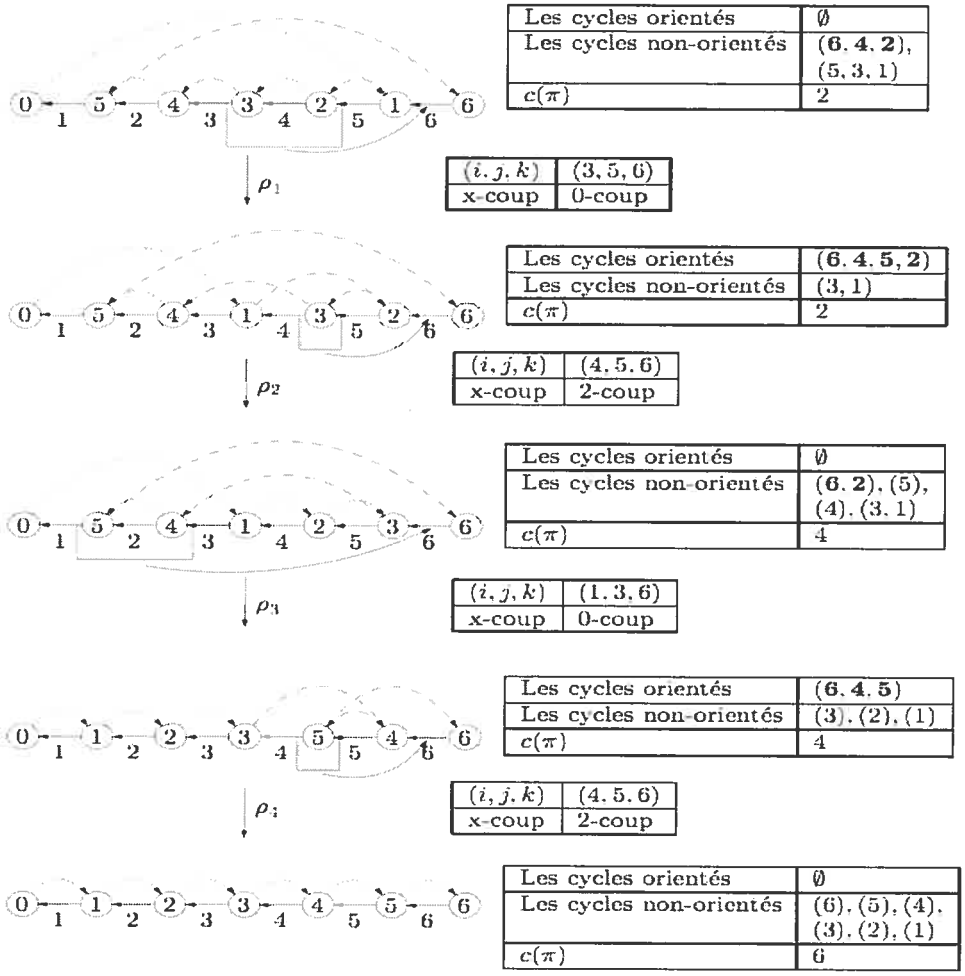


FIG. 2.1 – Exemple de tri par transpositions en utilisant l'algorithme *BP2* à partir du graphe de cycles de la permutation inverse de longueur 5.

Les sommets sont représentés par des cercles. Le nombre à l'intérieur de chacun d'eux les identifie. Les arêtes noires sont représentées par des flèches pleines. Les arêtes noires sont numérotées par les nombres sous chacune d'elles. Les arêtes grises sont représentées par des flèches pointillées. Des cycles distincts sont représentés par différents formats de flèches pointillées. Les arêtes grises des cycles constitués d'une seule arête noire et d'une seule arête grise sont cependant identiques pour plus de clarté.

Le tableau à droite du graphe de cycles indique les cycles orientés et non-orientés qu'il contient. $c(\pi)$ est le nombre total de cycles dans $G(\pi)$. Le cycle choisi pour calculer la transposition à faire est en caractère gras. Le tableau à droite de la transposition la décrit sous la forme (i, j, k) et indique le genre de transposition que c'est (-2-coup, 0-coup ou 2-coup).

Le théorème 2.5 permet de déduire le corollaire 2.6. Celui-ci nous apprend la valeur de la borne supérieure de $BP2(\pi)$.

Corollaire 2.6. $\text{BP2}(\pi) \leq n + 1 - c(\pi)$

Définition 2.1. *La* **garantie de performance** *d'un algorithme est le facteur par lequel il faut multiplier la réponse exacte pour obtenir la pire approximation possible par l'algorithme.*

On obtient la garantie de performance en faisant le quotient de la borne supérieure par la borne inférieure. Le théoreme 2.7 indique la garantie de performance de *BP2*.

Théorème 2.7. *La garantie de performance de* BP2 *est 2.*

Preuve.

$$\frac{\text{Borne supérieure de } BP2(\pi)}{\text{Borne inférieure de } d(\pi)}$$

Par le corollaire 2.6 et le théorème 2.2

$$= \frac{n + 1 - c(\pi)}{\frac{n+1-c(\pi)}{2}} = 2$$

■

Nous avons implémenté l'algorithme *BP2* (voir la section 4.1.2). Des résultats sont présentés aux tableaux 4.1 à 4.6.

2.2.3 TSort

Nous présentons maintenant un autre algorithme du même article [BP98]. Il s'agit de *TSort* et il a une garantie de performance de 1,75.

Introduisons quelques définitions. Un **k-cycle** est un cycle de longueur 2k. C'est aussi un cycle contenant k arêtes noires. Un cycle est dit **court** s'il contient une ou deux arêtes noires, sinon il est dit **long**.

Un **triplet** est un ensemble de trois arêtes noires x, y, z appartenant au même cycle C de $G(\pi)$. C induit un ordre cyclique sur x, y, z. Parmi les trois représentations

possibles de cet ordre, on choisit celle commençant par l'arête noire la plus à droite. Les séquences d'entiers ordonnés $(v_1, \ldots, v_k)$ et $(w_1, \ldots, w_k)$ **s'entrecoupent** si soit $v_1 < w_1 < v_2 < w_2 < \ldots < v_k < w_k$ soit $w_1 < v_1 < w_2 < v_2 < \ldots < w_k < v_k$. Les ensembles d'entiers V et W **s'entrecoupent** si l'ordonnancement de V et W s'entrecoupent. Un triplet (x, y, z) et un triplet (x', y', z') **s'entrecoupent** si $\{x, y, z\}$ et $\{x', y', z'\}$ s'entrecoupent.

Les cycles C et C' **se croisent** s'il existe un triplet orienté dans C et un triplet non-orienté dans C' qui s'entrecoupent. On a donc deux cycles C et C' tels que $(x, y, z) \in C, (x, y, z)$ est un triplet orienté, $(x', y', z') \in C', (x', y', z')$ est un triplet non-orienté et (x, y, z) entrecoupe (x', y', z'). Les auteurs prouvent qu'une transposition $\rho(y, z, x)$ est un 2-coup agissant sur C. Ils prouvent aussi que cette transposition transforme C' en un cycle orienté. Il sera alors possible de faire un second 2-coup (théorème 2.3).

Théorème 2.8. *Si $G(\pi)$ a deux cycles qui se croisent, alors il existe un 2-2-coup.* ■

La séquence de transpositions $\rho_6 \cdot \rho_7$ de la figure 2.2 est un exemple d'un tel 2-2-coup.

Les cycles C et C' sont **disjoints** (*noninterfering*) lorsqu'il existe un triplet orienté dans C et un autre triplet orienté dans C' qui ne s'entrecoupent pas. On a donc deux cycles C et C' tels que $(x, y, z) \in C, (x', y', z') \in C', (x, y, z)$ et (x', y', z') sont des triplets orientés qui ne s'entrecoupent pas. Les auteurs prouvent qu'une transposition $\rho(y, z, x)$ est un 2-coup agissant sur C. Ils prouvent aussi que ρ laisse C' orienté. Il sera alors possible de faire un 2-coup (théorème 2.3).

Théorème 2.9. *Si $G(\pi)$ a deux cycles qui sont disjoints, alors il existe un 2-2-coup.* ■

Les résultats précédents permettent aux auteurs de présenter le théorème central de *TSort*.

Théorème 2.10. *S'il existe un long cycle dans $G(\pi)$, alors il existe soit un 2-coup soit un 0-2-2-coup.* ■

Nous allons maintenant expliquer les grandes lignes du théorème 2.10. Les auteurs prouvent qu'il existe une transposition qui est un 0-coup créant soit des cycles qui se croisent soit des cycles disjoints lorsqu'il existe un long cycle dans $G(\pi)$ mais aucun cycle orienté. La preuve est une analyse exhaustive de cas. Deux cycles qui se croisent ou deux cycles disjoints permettent de faire un 2-2-coup (théorèmes 2.8 et 2.9 respectivement). Nous pouvons donc conclure que s'il existe un long cycle dans $G(\pi)$ mais aucun cycle orienté, alors il existe un 0-2-2-coup. Lorsque $G(\pi)$ a un long cycle, soit il a un cycle orienté permettant un 2-coup (théorème 2.3) soit il n'en a pas et nous venons de voir qu'il est alors possible de faire un 0-2-2-coup.

Nous avons maintenant tous les outils pour présenter l'algorithme *TSort*.

```
Algorithme TSort(π)
Tant que π ≠ Id
        S'il existe un long cycle dans G(π)
                Faire un 2-coup ou un 0-2-2-coup (théorème 2.10).
        Sinon
                Faire un 0-2-coup (théorème 2.4).
```

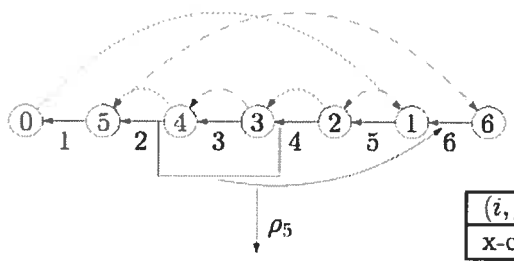
La figure 2.2 présente une trace de l'exécution de cet algorithme.

Il est démontré par les auteurs que la garantie de performance de *TSort* est 1.75.

2.2.4 TransSort

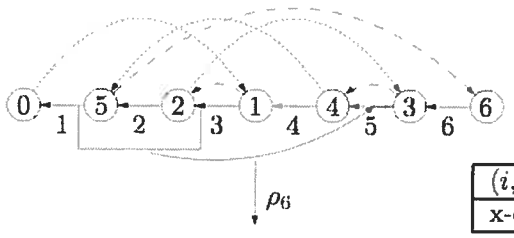
Nous présentons maintenant le plus compliqué des algorithmes de [BP98]. C'est aussi celui qui a la meilleure garantie de performance. Elle est de 1,5. Sa complexité en temps est dans $O(n^2)$ où n est la longueur de π. TransSort utilise la parité des cycles.

Un cycle de $G(\pi)$ est **impair** s'il a un nombre impair d'arêtes noires et **pair** sinon. On voit que le graphe de cycles de la permutation identité ne contient que des cycles impairs et contient un maximum de cycles impairs (voir Figures 2.1 et 2.2).



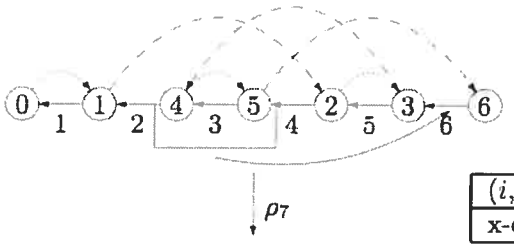
Les cycles orientés	$\emptyset$
Les cycles non-orientés	**(6, 4, 2)**, (5, 3, 1)
$c(\pi)$	2

(i, j, k)	(2, 4, 6)
x-coup	0-coup



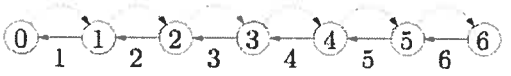
Les cycles orientés	**(5, 1, 3)**
Les cycles non-orientés	(6, 4, 2)
$c(\pi)$	2

(i, j, k)	(1, 3, 5)
x-coup	2-coup



Les cycles orientés	**(6, 2, 4)**
Les cycles non-orientés	(5), (3), (1)
$c(\pi)$	4

(i, j, k)	(2, 4, 6)
x-coup	2-coup



Les cycles orientés	$\emptyset$
Les cycles non-orientés	(6), (5), (4), (3), (2), (1)
$c(\pi)$	6

FIG. 2.2 – Exemple de tri par transpositions en utilisant l'algorithme *TSort* à partir du graphe de cycles de la permutation inverse de longueur 5.
Voir la légende de la figure 2.1

Trier une permutation revient donc à augmenter le nombre de cycles impairs dans son graphe de cycles.

Soit $c_{odd}(\pi)$ le nombre de cycles impairs dans $G(\pi)$ et soit $\Delta c_{odd}(\rho)$ la variation du nombre de cycles impairs à la suite de la transposition ρ. Les auteurs prouvent les équivalents du lemme 2.1 et du théorème 2.2 dans le cadre des cycles impairs et obtiennent les résultats suivants :

Lemme 2.11. $\Delta c_{odd}(\rho) \in \{-2, 0, 2\}$ ∎

Théorème 2.12. $d(\pi) \geq \frac{n+1-c_{odd}(\pi)}{2}$

Preuve. Le graphe de cycles de la permutation identité a $n+1$ cycles impairs. Il faut donc créer $n+1-c_{odd}(\pi)$ cycles impairs. On peut en créer au plus deux par transposition selon le lemme 2.11. ∎

Une transposition ρ est **valide** si $\Delta c(\rho) = \Delta c_{odd}(\rho)$.

Les auteurs modifient le théorème 2.3 pour obtenir le théorème 2.13.

Théorème 2.13. *S'il existe un cycle orienté dans $G(\pi)$, alors il existe soit un 2-coup valide soit un 0-coup valide suivi de deux 2-coups valides.* ∎

Les auteurs prouvent ensuite le théorème 2.14 qui peut être considéré comme l'équivalent du théorème 2.10. La différence avec le théorème 2.10 est que l'on s'assure que toutes les transpositions soient valides.

Théorème 2.14. *S'il existe un long cycle dans $G(\pi)$ mais pas de cycles orientés, alors il existe soit un 2-coup valide soit un 0-coup valide suivi de deux 2-coups valides.* ∎

La transposition ρ est un **bon 0-coup** si $\Delta c(\rho) = 0$ et $\Delta c_{odd}(\rho) = 2$.

Cette définition nous sera utile dans le cas où $G(\pi)$ ne contient pas de longs cycles. Les auteurs prouvent une modification du théorème 2.4 que nous appellerons le théorème 2.15. Ils s'assurent que chacune des deux transpositions du 0-2-coup du théorème 2.4 crée deux cycles impairs de plus. Pour renforcer le théorème 2.4 de cette manière, les auteurs doivent ajouter la condition que $G(\pi)$ ne contienne pas de longs cycles.

Théorème 2.15. *S'il n'existe pas de long cycle dans $G(\pi)$, alors il existe un bon 0-coup suivi d'un 2-coup valide.* ■

```
Algorithme TransSort(π)
Tant que π ≠ Id
    S'il existe un long cycle dans G(π)
        S'il existe un cycle orienté dans G(π)
            Faire soit un 2-coup valide soit un 0-coup
            valide suivi de deux 2-coups valides
            (théorème 2.13).
        Sinon
            Faire soit un 2-coup valide soit un 0-coup
            valide suivi de deux 2-coups valides
            (théorème 2.14).
    Sinon
        Faire un bon 0-coup suivi d'un 2-coup valide
        (théorème 2.15).
```

Théorème 2.16. $\text{TransSort}(\pi) \leq \frac{3}{4}(n + 1 - c_{odd}(\pi))$

Preuve. Le pire cas est celui où il y a le moins de cycles impairs créés par transposition. Dans le cas d'un 2-coup valide, il y a deux cycles impairs créés pour une transposition. Dans le cas d'un 0-coup valide suivi de deux 2-coups valides, il y a quatre cycles impairs créés pour trois transpositions. Cela donne en moyenne $\frac{4}{3}$ de cycles par transposition. Dans le cas d'un bon 0-coup suivi d'un 2-coup, il y a quatre cycles impairs créés en deux transpositions. Cela donne en moyenne deux cycles impairs par transposition. Le pire cas est donc $\frac{4}{3}$ cycles impairs par transposition. Le quotient du nombre de cycles impairs à créer $(n + 1 - c_{odd}(\pi))$ par le pire cas du nombre moyen de cycles impairs créés par transposition $(\frac{4}{3})$ donne la borne supérieure de l'algorithme. ■

Théorème 2.17. *La garantie de performance de* TransSort *est 1,5.*

Preuve. On divise la borne supérieure (théorème 2.16) par la borne inférieure (théorème 2.12).

$$\frac{\frac{3}{4}(n+1-c_{odd}(\pi))}{\frac{1}{2}(n+1-c_{odd}(\pi))} = 1,5$$

■

La complexité en temps de *TransSort* est dans $O(n^2)$. Les auteurs ne la prouvent pas. Il mérite d'être noté que [GHV95] et [Chr98] commentèrent qu'il s'agit d'un algorithme complexe qui n'a pas été implémenté à cause d'un manque de détails techniques. L'algorithme a été implémenté depuis mais jamais avec une complexité en temps dans $O(n^2)$ (voir la section 2.4).

2.3 Autres algorithmes

Nous présentons rapidement d'autres algorithmes que ceux de Bafna et Pevzner [BP98] dans cette section.

2.3.1 Algorithmes d'Elias et Hartman

Hartman [Har03] s'est basé sur plusieurs résultats de [BP98] pour concevoir lui aussi un algorithme qui a une garantie de performance de $1,5$ et une complexité en temps dans $O(n^2)$. Son algorithme s'applique aux permutations circulaires. Il doit d'abord prouver le théorème 2.18.

Théorème 2.18. *Le problème de trier une permutation linéaire de longueur n est équivalent au problème de trier une permutation circulaire de longueur $n+1$.* ■

Cela veut dire que l'on peut transformer une permutation linéaire de longueur n en une permutation circulaire de longueur $n+1$. Si on trouve une séquence de d transpositions pour trier cette permutation circulaire, on peut mimer ces d transpositions sur la permutation linéaire et ainsi trier cette dernière.

L'auteur construit un graphe de cycles circulaire semblable au graphe de cycles (linéaire) de [BP98]. En plus d'être circulaire, le graphe de cycles de Hartman diffère

de celui de [BP98] au niveau de quelques détails. Nous omettons de les mentionner ici.

L'auteur change certaines définitions de [BP98] s'appliquant aux cycles et aux ensembles de cycles. Son algorithme cherche de nouveaux cas concernant les cycles et leur agencement. Bien que les cas recherchés diffèrent de ceux de [BP98], l'auteur obtient un algorithme qui garantit, lui aussi, de faire soit un 2-coup valide soit un 0-coup valide suivi de deux 2-coups valides. C'est pourquoi son algorithme a, lui aussi, une garantie de performance de $1,5$.

Dans un article subséquent avec Elias, Hartman fait une analyse de cas plus exhaustive sur son graphe de cycles circulaire [EH05]. Ils utilisent une preuve par ordinateur pour analyser environ 80000 cas. Leur nouvel algorithme a une garantie de performance de $1,375$ tout en conservant une complexité dans $O(n^2)$. C'est l'algorithme qui a la meilleure garantie de performance aujourd'hui.

2.3.2 Algorithmes de Guyer, Heath et Vergara

Guyer, Heath et Vergara [GHV95] ont présenté plusieurs algorithmes simples et rapides pour trier par transpositions sans analyser leur garantie de performance. Un de leurs algorithmes utilise les LIS (*longest increasing subsequence* ou sous-séquence non contiguë croissante [1] la plus longue) (voir définition 2.2).

Définition 2.2. *Soit une permutation* $\pi = \pi_1\pi_2\ldots\pi_n$. *Une sous-séquence non contiguë croissante de* π *est une sous-séquence non contiguë* $\pi_{i_1}\pi_{i_2}\ldots\pi_{i_k}$ *telle que,* $\forall j$ *où* $1 \leq j < k$, *nous avons que* $\pi_{i_j} < \pi_{i_{j+1}}$.

Une sous-séquence non contiguë croissante la plus longue $LIS(\pi)$ *est une sous-séquence non contiguë croissante de* π *de longueur maximale. Il peut y avoir plusieurs sous-séquences non contiguës croissantes les plus longues. Dans ce cas, n'importe laquelle peut être notée par* $LIS(\pi)$.

[1] La définition de séquence, dans ce mémoire, implique que ses éléments soient contigus. Ce n'est pas le cas de la définition de *subsequence* de Guyer, Heath et Vergara. Nous traduisons donc leur terme *subsequence* par sous-séquence non contiguë.

Exemple 2.1. *Soit une permutation* $\pi = 3\ 5\ 4\ 7\ 2\ 6\ 8\ 1\ 9$. $LIS(\pi) = 3\ 4\ 6\ 8\ 9$.

L'algorithme *TSLIS* est un algorithme glouton. Il choisit, parmi toutes les transpositions possibles, celles qui crée la plus longue LIS. Sa complexité est dans $O(n^5 log(n))$.

Les auteurs présentent ensuite des algorithmes se basant sur les courses (*runs*) (voir définition 2.3).

Définition 2.3. *Une course est une séquence (contiguë) maximale d'éléments consécutifs.*

Exemple 2.2. *Soit une permutation* $\pi = 3\ 4\ 5\ 7\ 8\ 6\ 9\ 1\ 2$. *Les courses de* π *sont* 3 4 5, 7 8, 6, 9 *et* 1 2.

Soit une permutation π de [n]. Placer une course signifie transposer une course de manière à l'allonger vers la droite. Il y a une exception. Placer une course se terminant par l'élément n signifie la transposer complètement à droite de π.

Les algorithmes *TSRunLong*, *TSRunShort* et *TSRunRand* placent la plus longue course, la plus courte course et une course sélectionnée pseudo-aléatoirement respectivement. L'algorithme *TSRunBest* est un algorithme glouton qui place la course qui fait le plus diminuer le nombre de courses dans π lorsque cette course est placée. Ces quatre derniers algorithmes ont chacun une complexité dans $O(n^2)$.

Les auteurs présentent ensuite l'algorithme *TSBnB*. Cet algorithme utilise une méthode *Branch and Bound* pour calculer une séquence de transpositions optimale. C'est un algorithme exact.

Finalement, les auteurs ajoutent ensuite le concept de courses à ce dernier algorithme pour avoir moins de tests à faire. Cela rend l'algorithme plus rapide. Il devient cependant une heuristique. Ils nomment ce nouvel algorithme *TSRunBnB*. La complexité de ces deux derniers algorithmes est exponentielle.

2.3.3 Survol rapide d'autres algorithmes

D'autres algorithmes approximatifs de tri par transpositions méritent d'être mentionnés. L'algorithme de Christie a une garantie de performance de $1,5$ et une

complexité dans $O(n^4)$ [Chr98]. Walter, Dias et Meidanis utilisent une structure qu'ils nomment diagramme de points de cassure (*breakpoint diagram*) [WDM00]. Le point de cassure est défini à la définition 3.14. Leur algorithme a une garantie de performance de $2,25$. Sa complexité est dans $O(b^2)$ où b représente le nombre de points de cassure de la permutation.

L'algorithme d'Eriksson et al. trie une permutation de longueur n, pour $n \geq 9$, en au plus $\lfloor \frac{2n-2}{3} \rfloor$ transpositions

[EEK+01]. Leur algorithme n'a cependant pas de garantie de performance. Labarre présente une nouvelle borne supérieure à la distance de transposition d'une permutation sans élaborer de nouvel algorithme [Lab05].

2.4 Implémentations

Walter, Dias et Meidanis [WDM00] et Guyer, Heath et Vergara [GHV95] sont les seuls à avoir implémenté leurs algorithmes de tri par transpositions parmi les algorithmes dont nous avons parlés. Certains des autres algorithmes ont été implémentés par d'autres personnes que les auteurs des algorithmes eux-mêmes. C'est le sujet de cette section.

L'algorithme de Bafna et Pevzner *TransSort* qui a une garantie de performance de 1,5 [BP98] a d'abord été implémenté par Walter et Oliveira [Oli01, WdO02a, WdO02b]. La complexité en temps de leur implémentation est dans $O(n^5)$. Walter, Sobrinho et al. ont ensuite implémenté le même algorithme avec une complexité dans $O(n^3)$ cette fois-ci [WSO+05]. Ils présentent une version de l'algorithme de Bafna et Pevzner sans heuristique et une autre version avec des heuristiques. Ces heuristiques améliorent empiriquement la qualité des approximations.

L'algorithme de Hartman qui a une garantie de performance de 1,5 [Har03] a été implémenté par Honda [Hon04]. L'algorithme de Christie [Chr98] a été implémenté avec des heuristiques par Walter, Curado et Oliveira [WCO03].

Des résultats pour les implémentations des algorithmes de Walter, Dias et Meidanis [WDM00], de Christie [WCO03], de Hartman avec une garantie de per-

formance de 1,5 [Hon04], de Bafna et Pevzner implémenté par Walter et Oliveira [Oli01. WdO02a, WdO02b] et par Walter, Sobrinho et al. [WSO+05] et de Guyer, Heath et Vergara [GHV95] sont présentés au tableau 4.1. Des résultats supplémentaires pour les implémentations des algorithmes de [GHV95] sont présentés aux tableaux 4.3, 4.4 et 4.6.

CHAPITRE 3

NOUVEAUX ALGORITHMES

Dans ce chapitre, nous présentons nos algorithmes pour trier par transpositions. Une partie de ces résultats a été présentée dans [BGH07]. Nos algorithmes ne se basent pas sur une structure de données représentant un graphe comme le graphe de cycles de [BP98, Har03, EH05] ou le diagramme de points de cassure de [WDM00]. Plutôt que d'utiliser un graphe, nous avons essayé de travailler à partir d'un simple vecteur. Nous avons cherché à associer un nombre à chaque élément de la permutation. Nous avons choisi ce que nous avons décidé d'appeler les deux codes d'une permutation. Intuitivement, pour une position i dans une permutation π, le code gauche (respectivement droit) à cette position est simplement le nombre d'éléments plus grands (respectivement plus petits) que π_i qui sont situés à sa gauche (respectivement à sa droite).

Le code gauche et le code droit sont donc chacun une simple séquence de nombres qui a la même longueur que la permutation. Nous utiliserons certains "motifs" caractéristiques de ces séquences de nombres, que nous appellerons **objets**, pour construire un algorithme qui génère une solution sous optimale au problème du tri par transpositions. Nous avons défini différents objets. On peut choisir certains objets parmi ceux-ci et les agencer librement pour élaborer différents algorithmes de tri par transpositions. Les objets peuvent être alors vus comme des blocs de construction.

Dans la première section de ce chapitre, nous expliquons formellement comment "coder" une permutation pour obtenir ses codes gauche et droit. Nous présentons les différents objets à la section 3.2. Pour chaque objet, nous le définissons, nous expliquons comment le traiter et nous démontrons la complexité en temps de ce traitement. Nous présentons comment agencer ces objets pour élaborer des algorithmes de tri par transpositions à la section 3.3. Nous présentons différents exemples de tels algorithmes ayant différentes complexités en temps. Nous démontrons une ga-

rantie de performance de 3 pour un de ces algorithmes nommé *TriePlateau*. La section 3.4 présente les choix qu'un programmeur a à faire lorsqu'il implémente un de nos algorithmes. Ces choix ne changent pas la complexité de l'algorithme mais peuvent donner des résultats différents.

3.1 Codes d'une permutation

Les codes gauche et droit d'une permutation sont définis comme suit.

Définitions 3.1. *Soit une permutation* $\pi = \pi_1 \dots \pi_n$, *le* **code gauche** *de* π_i, *noté* $cg(\pi_i)$, *est, pour* $1 \leq i \leq n$,

$$cg(\pi_i) = |\{\pi_j \mid \pi_j > \pi_i \text{ et } 0 \leq j \leq i-1\}|.$$

Le code gauche de la permutation π *est alors défini comme la séquence des codes gauches de ses éléments.*
Similairement, le **code droit** *de* π_i, *noté* $cd(\pi_i)$, *est, pour* $1 \leq i \leq n$,

$$cd(\pi_i) = |\{\pi_j \mid \pi_j < \pi_i \text{ et } i+1 \leq j \leq n+1\}|.$$

Le code droit de la permutation π *est alors défini comme la séquence des codes droits de ses éléments.*

La figure 3.1 donne les codes gauche et droit de quatre permutations différentes de longueur 6.

Une transposition ρ agit sur quatre parties d'une permutation (voir Figure 3.2). Les parties A et D peuvent être vides.

Le nombre d'éléments à gauche plus grands que les éléments dans A ou que les éléments dans D ne change pas suite à ρ. Le nombre d'éléments à droite plus petits que les éléments dans A ou que les éléments dans D ne change pas non plus suite à ρ. C'est pourquoi les codes, gauche ou droit, ne changent jamais dans A et D suite à une transposition.

$cd(\pi^1)$	=	5	2	1	0	0	0	$cd(\pi^2)$	=	2	3	1	0	1	0
$cg(\pi^1)$	=	0	1	2	3	1	1	$cg(\pi^2)$	=	0	0	2	3	0	2
π^1	=	6	3	2	1	4	5	π^2	=	3	5	2	1	6	4
$cd(\pi^3)$	=	5	4	3	2	1	0	$cd(\mathrm{Id})$	=	0	0	0	0	0	0
$cg(\pi^3)$	=	0	1	2	3	4	5	$cg(\mathrm{Id})$	=	0	0	0	0	0	0
π^3	=	6	5	4	3	2	1	Id	=	1	2	3	4	5	6

FIG. 3.1 – Codes gauche et droit de quatre permutations de longueur 6.

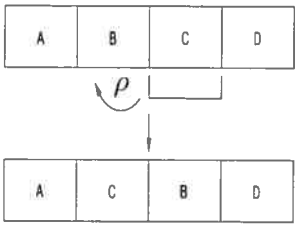


FIG. 3.2 – Les quatre parties d'une permutation liées à une transposition. Deux segments de la permutation sont interchangés lors d'une transposition. La partie A est avant le premier segment interchangé. La partie B est le premier segment interchangé. La partie C est le deuxième segment interchangé. La partie D est après le deuxième segment interchangé.

3.2 Objets

Nous appelons certaines sous-séquences de codes ayant des caractéristiques communes des objets. Nous traitons chaque objet d'une manière particulière dans le but d'augmenter le nombre de zéros dans le code gauche ou le code droit.

3.2.1 Plateaux

Deux éléments contigus qui ont le même code gauche (respectivement droit) ont les mêmes éléments plus grands qu'eux à leur gauche (respectivement à leur droite). On peut alors se demander s'il est possible d'annuler leur code en les transposant au même endroit. La réponse est oui. Nous utilisons l'objet plateau pour ce faire.

3.2.1.1 Définitions

Définition 3.2. *Soit S une séquence de nombres entiers positifs. Un* **plateau** *dans S est une sous-séquence, de longueur maximale, d'éléments contigus ayant la même valeur non nulle.*

Définitions 3.3. *Le* **nombre de plateaux** *dans un code c est noté $p(c)$. Nous dénotons par $pg(\pi) = p(cg(\pi))$ (respectivement $pd(\pi) = p(cd(\pi))$) le nombre de plateaux dans le code gauche (respectivement dans le code droit) de π. Finalement, nous dénotons $p(\pi)$ le minimum du nombre de plateaux dans les deux codes de π i.e. $p(\pi) = min\{pg(\pi), pd(\pi)\}$.*

Nous illustrons ces définitions par un exemple tiré de la figure 3.1. Pour la permutation π^1, nous avons $pg(\pi^1) = 4$ parce que nous avons quatre plateaux : $1, 2, 3$ et 1 1. Pour ce qui est du code droit, nous avons $pd(\pi^1) = 3$ avec les trois plateaux : $5, 2$ et 1. Donc, $p(\pi^1) = min\{4, 3\} = 3$.

Nous observons à la figure 3.1 que les codes gauche et droit de la permutation identité, Id, sont les seuls à n'avoir aucun plateau. Le but de nos algorithmes sera donc de trouver des transpositions qui diminuent le nombre de plateaux dans soit le code gauche soit le code droit de π.

Définitions 3.4. *La variation du nombre de plateaux dans le code gauche d'une permutation π à la suite de la transposition ρ est notée $\Delta p_{gauche}(\rho)$. On a que $\Delta p_{gauche}(\rho) = pg(\pi \cdot \rho) - pg(\pi)$. On a une définition équivalente pour le code droit. On a que $\Delta p_{droit}(\rho) = pd(\pi \cdot \rho) - pd(\pi)$.*

Nous omettons l'indice gauche ou droit quand il peut être déduit du contexte dans ce mémoire.

3.2.1.2 Traitement

Le lemme 3.1 indique comment traiter un plateau.

Lemme 3.1. *Soit une permutation $\pi = \pi_1 \dots \pi_n$, le plateau le plus à gauche de $cg(\pi)$ peut être enlevé par une transposition ρ_1 vers la gauche sans créer de nouveaux plateaux dans le code. Cela implique que $\Delta p_{gauche}(\rho_1) = -1$. Similairement, le plateau le plus à droite de $cd(\pi)$ peut être enlevé par une transposition ρ_2 vers la droite sans créer de nouveaux plateaux dans le code. On a donc $\Delta p_{droit}(\rho_2) = -1$.*

Preuve. Commençons par prouver la partie du lemme 3.1 concernant $cg(\pi)$. Elle suit directement la définition de $cg(\pi)$. Supposons que le plateau le plus à gauche dans $cg(\pi)$ s'étende de la position i à la position $j-1$ inclusivement. Alors chaque entrée dans $cg(\pi)$ est égale à 0 avant la position i. En d'autres termes, les entrées $\pi_1 \dots \pi_{i-1}$ sont en ordre croissant. Donc, si $cg(\pi_i) = v$, la transposition $\rho(i, j, k)$, où $k = i - v$, enlève le premier plateau sans en créer un nouveau.

Prouvons maintenant la partie du lemme 3.1 concernant $cd(\pi)$. Elle suit aussi directement la définition de $cd(\pi)$. Supposons que le plateau le plus à droite dans $cd(\pi)$ s'étende de la position i à la position $j-1$ inclusivement. Alors chaque entrée dans $cd(\pi)$ est égale à 0 après la position $j-1$. En d'autres termes, les entrées $\pi_j \dots \pi_n$ sont en ordre croissant. Donc, si $cd(\pi) = v$, la transposition $\rho(i, j, k)$, où $k = j + v$, enlève le plateau le plus à droite sans en créer un nouveau. ∎

Soit ρ n'importe laquelle des transpositions qui annule un plateau apparaissant dans soit $cg(\pi)$ soit $cd(\pi)$ sans en créer de nouveaux. Nous pouvons déduire du lemme 3.1 que de telles transpositions existent pour toute permutation (non nulle). Cela implique directement le théorème 3.2.

Théorème 3.2. *Pour toute permutation π de $[n]$, on a $d(\pi) \leq p(\pi)$.* ∎

La figure 3.3 montre un exemple du traitement des plateaux de $cd(\pi^1)$ (voir π^1 à la figure 3.1).

3.2.1.3 Complexité en temps

Nous pouvons élaborer l'algorithme *TriePlateau* (voir Figure 3.4) qui permet de trier complètement par transpositions n'importe quelle permutation à partir du lemme 3.1 et du théorème 3.2.

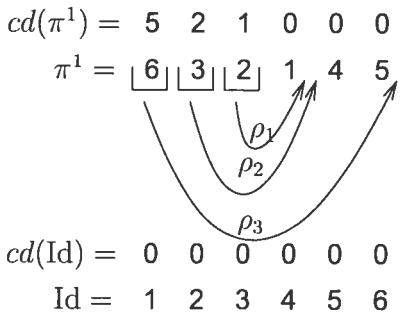


FIG. 3.3 – Exemple du traitement des plateaux de π.
Nous avons calculé le code droit de $\pi^1 = 6\ 3\ 2\ 1\ 4\ 5$. Nous avons transposé les plateaux $1, 2$ et 5 par les transpositions ρ_1, ρ_2 et ρ_3.

Algorithme TriePlateau(entrée : une permutation π de [n])
$cg(\pi)$ = code gauche de π (calculé facilement en $O(n^2)$)
$cd(\pi)$ = code droit de π (calculé facilement en $O(n^2)$)
$pg(\pi) = p(cg(\pi))$ (calculé facilement en $O(n)$)
$pd(\pi) = p(cd(\pi))$ (calculé facilement en $O(n)$)
RETOURNER $p(\pi) = min\{pg(\pi), pd(\pi)\}$

FIG. 3.4 – Algorithme *TriePlateau(π)*.

Théorème 3.3. *La complexité de* TriePlateau *est dans* $O(n^2)$.

Preuve. Voir la description de l'algorithme *TriePlateau* à la figure 3.4. ∎

Il est possible de lister les $p(\pi)$ transpositions nécessaires au tri de π. Nous devons alors enregistrer le début, la fin et la valeur du code de chaque plateau lorsqu'on calcule $pg(\pi)$ et $pd(\pi)$.

Puisque $p(\pi^1) = pd(\pi^1)$, traiter les plateaux du code droit de π^1 comme dans la figure 3.3 est un exemple de l'algorithme *TriePlateau(π)*.

3.2.2 Montagnes

Nous n'utiliserons plus que le code gauche pour définir les objets à partir de maintenant pour éviter d'alourdir le texte. Il est possible de construire une

définition équivalente avec le code droit pour chaque sorte d'objet par symétrie.

Deux plateaux qui ont la même valeur et qui sont séparés par d'autres plateaux qui ont des valeurs différentes peuvent être vus comme un seul plateau coupé en deux par ces autres plateaux. Il est alors raisonnable de penser qu'il est possible d'annuler ces deux plateaux lors d'une seule transposition dans certains cas. Nous avons créé l'objet montagne pour cela.

3.2.2.1 Définitions

Soit une permutation π de [n], notons par $P_1, P_2, \ldots, P_m$ les m plateaux de $cg(\pi)$ de gauche à droite. Notons par $v(P_x)$ la valeur de P_x, c'est-à-dire le code des éléments du plateau.

Définitions 3.5. *La séquence de plateaux $P_a\ P_{a+1} \ldots P_{a+b}$ est une* **montagne** *si $v(P_a) = v(P_{a+b})$ et $v(P_{a+1}), \ldots, v(P_{a+b-1}) \geq v(P_a)$. Les* **bases de la montagne** *sont les plateaux $P_x, a \leq x \leq a+b$, tels que $v(P_x) = v(P_a)$. Les plateaux P_a et P_{a+b} sont donc toujours des bases de la montagne mais peuvent ne pas être les seules.*

3.2.2.2 Traitement

Nous allons présenter l'algorithme *TraiteMontagne* (voir Figure 3.5) qui déplace une montagne de manière à annuler ses bases. Annuler les bases d'une montagne signifie mettre leurs codes à 0. Soit une montagne recouvrant les positions i à $j-1$ inclusivement. Soit k la position du premier élément plus grand que le premier élément de la montagne. La transposition $\rho(i, j, k)$ déplace la montagne vers la gauche. Les éléments des bases de la montagne n'ont plus d'éléments plus grands à leur gauche. Leurs codes sont annulés. La figure 3.6 illustre la transposition d'une montagne.

Théorème 3.4. *Soit ρ la transposition de* TraiteMontagne *sur π. On a que $\Delta p(\rho)$ peut ne pas être ≤ -1.*

Preuve. Voir le contre-exemple à la figure 3.6 où $pg(\pi) = 5$, $pg(\pi \cdot \rho) = 5$ et donc $\Delta p_{gauche}(\rho) = 0$. ∎

Algorithme TraiteMontagne(entrée : une permutation π de [n])
$cg(\pi)$ = code gauche de π (calculé facilement en $O(n^2)$)
Chercher une montagne (cherchée facilement en $O(n^2)$)
S'il y a une montagne
    Chercher la position d'insertion (trouvée facilement en $O(n)$).
    Transposer ($O(1)$ avec une liste chaînée).

FIG. 3.5 – Algorithme *TraiteMontagne(π)*.

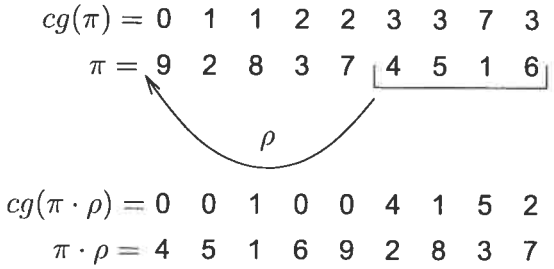


FIG. 3.6 – Exemple de *TraiteMontagne(π)*.

Nous avons calculé le code gauche de $\pi = 9\ 2\ 8\ 3\ 7\ 4\ 5\ 1\ 6$. La montagne couvre les positions 6 à 9 : 3 3 7 3. Les éléments des bases de la montagne sont 4 5 et 6. Le premier élément de la montagne est 4. Le premier élément de π plus grand que 4 est 9 à la position 1. La transposition $\rho(6, 10, 1)$ déplace la montagne vers sa gauche et annule ses bases.

Il est intéressant de noter que lorsque la sous-permutation à gauche de la montagne est triée, $\Delta p(\rho) \leq -2$ (voir Figure 3.7).

3.2.2.3 Complexité en temps

Nous tenons à rappeler que *TraiteMontagne* n'est pas un algorithme de tri complet comme *TriePlateau* puisqu'il ne fait que déplacer une montagne.

Théorème 3.5. *La complexité de* TraiteMontagne *est dans* $O(n^2)$.

Preuve. Voir la description de l'algorithme *TraiteMontagne* à la figure 3.5. ∎

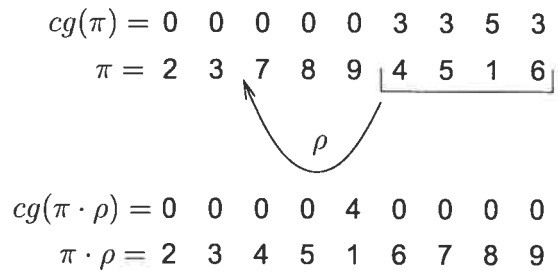


FIG. 3.7 – Exemple de *TraiteMontagne(π)* lorsque la sous-permutation à gauche de la montagne est triée.
La permutation π est celle de la figure 3.6 dont nous avons préalablement trié la sous-permutation à gauche de la montagne. Le premier élément de π plus grand que 4 est 7 à la position 3 cette fois-ci. La transposition $\rho(6, 10, 3)$ déplace la montagne et annule ses bases sans créer de nouveaux plateaux. On observe que $\Delta p(\rho) = -2$.

3.2.3 Fossés asymétriques

3.2.3.1 Définitions

La **longueur** d'une sous-permutation est le nombre d'éléments qu'elle contient. Remarquez que cette longueur n'est pas toujours égale au nombre de plateaux, étant donné que plusieurs éléments peuvent contribuer au même plateau. (Voir π^1 dans la figure 3.1 où la longueur de la sous-permutation $\pi_5\ \pi_6 = 4\ 5$ est 2 et où le nombre de plateaux de cette sous-permutation est 1.)

Définitions 3.6. *Nous appelons* **fossé** *m plateaux consécutifs $P_i, P_{i+1}, \ldots, P_{i+m-1}$ tels que $v(P_{i+1}), v(P_{i+2}), \ldots, v(P_{i+m-2}) < v(P_i)$ et $v(P_i) \leq v(P_{i+m-1})$. Les plateaux P_i et P_{i+m-1} sont appelés les* **bouts** *du fossé. Le* **fond** *d'un fossé est constitué des plateaux $P_{i+1}, P_{i+2}, \ldots, P_{i+m-2}$. Un fossé est* **asymétrique** *si $v(P_i) + l_{fond} = v(P_{i+m-1})$ où l_{fond} est la longueur du fond du fossé.*

3.2.3.2 Traitement

Nous allons présenter l'algorithme *TraiteFosséAsymétrique* (voir Figure 3.8) qui déplace le fond d'un fossé pour transformer ses deux bouts en un seul plateau. Soit

le fond d'un fossé asymétrique recouvrant les positions i à $j-1$ inclusivement. Soit k la position du plus petit élément, parmi les éléments plus grands que π_{j-1} (le dernier élément du fond du fossé), et à gauche de celui-ci. La transposition $\rho(i,j,k)$ déplace le fond du fossé vers la gauche.

Algorithme TraiteFosséAsymétrique(entrée : une permutation π de [n])
$cg(\pi)$ = code gauche de π (calculé facilement en $O(n^2)$)
Chercher un fossé asymétrique (cherché facilement en $O(n^2)$)
S'il y a un fossé asymétrique
Chercher la position d'insertion (trouvée facilement en $O(n)$).
Transposer ($O(1)$ avec une liste chaînée).

FIG. 3.8 – Algorithme *TraiteFosséAsymétrique(π).*

Les codes du fond du fossé sont inférieurs à $v(P_i)$. Les éléments du fond du fossé sont donc supérieurs aux éléments de P_i. En transposant le fond du fossé à gauche de P_i, il y aura l éléments plus grands que les éléments de P_i à leur gauche. C'est pourquoi $v(P_i)$ augmentera de l. Les éléments de P_i formeront donc un plateau avec les éléments de P_{i+m-1} par la définition de fossé asymétrique. La figure 3.9 présente un exemple.

Théorème 3.6. *Soit ρ la transposition de* TraiteFosséAsymétrique *sur π. On a que $\Delta p(\rho)$ peut ne pas être ≤ -1.*

Preuve. Voir le contre-exemple à la figure 3.9 où $pg(\pi) = 6$, $pg(\pi \cdot \rho) = 6$ et donc $\Delta p_{gauche}(\rho) = 0$. ∎

3.2.3.3 Complexité en temps

De la même manière que *TraiteMontagne*, *TraiteFosséAsymétrique* n'est pas un algorithme complet.

Théorème 3.7. *La complexité de* TraiteFosséAsymétrique *est dans $O(n^2)$.*

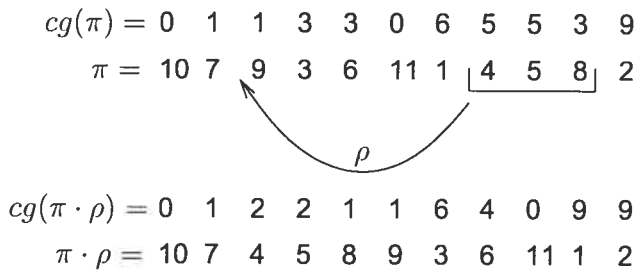


FIG. 3.9 – Exemple de *TraiteFosséAsymétrique(π)*.
Nous avons calculé le code gauche de $\pi = 10\ 7\ 9\ 3\ 6\ 11\ 1\ 4\ 5\ 8\ 2$. Le fossé asymétrique couvre les positions 7 à 11 : 6 5 5 3 9. L'élément de P_i (le bout à gauche) est 1. L'élément de P_{i+m-1} (le bout à droite) est 2. Le fond du fossé couvre les positions 8 à 10. Le dernier élément du fond du fossé est 8. Le plus petit élément parmi ceux qui sont plus grands que 8 tout en étant à sa gauche est 9 à la position 3. La transposition $\rho(8, 11, 3)$ déplace le fond du fossé à gauche de P_i. Les anciens bouts du fossé forment maintenant un plateau.

Preuve. Voir la description de l'algorithme *TraiteFosséAsymétrique* à la figure 3.8.

■

3.2.4 Ascensions

3.2.4.1 Définitions

La distance de tranposition d'une permutation inverse a été calculée indépendamment par [MWD97, Chr98, EEK+01] (voir la section 3.2.4.2). Nous avons généralisé la notion de permutation inverse ($\pi = n\ n{-}1 \ldots 2\ 1$) en créant l'objet ascension.

Soit une **plage** une suite maximale de codes égaux à 0. Soit une **plage non maximale** une suite de codes égaux à 0. Soit un **terrain plat** une plage non maximale ou un plateau. Une ascension est une suite de terrains plats ressemblant à une permutation inverse. Formellement, nous avons

Définition 3.7. *La séquence de terrains plats $T_iT_{i+1} \ldots T_{i+k}$ est une* **ascension** *si $k \geq 4$ et $v(T_{i+1}) = v(T_i) + l_i\ , \ldots , v(T_{i+k}) = v(T_{i+k-1}) + l_{i+k-1}$, où l_j est la longueur de T_j .*

3.2.4.2 Traitement

L'algorithme de [MWD97, Chr98, EEK+01] est présenté à la figure 3.10. On voit facilement que sa complexité en temps est dans $O(n)$. Une exécution de l'algorithme sur un exemple est présentée à la figure 3.11.

Algorithme TriePermutationInverse(entrée : une permutation inverse π de [n])
$i = \lceil \frac{n}{2} \rceil$
$k = 1$
Pour j de 1 à $\lfloor \frac{n}{2} \rfloor$
    $\pi = \pi \cdot \rho(i, i+2, k)$
    $i = i + 1$
    $k = k + 1$
$\pi = \pi \cdot \rho(\lfloor \frac{n}{2} \rfloor + 1, 2\lfloor \frac{n}{2} \rfloor + 1, 1)$

FIG. 3.10 – Algorithme *TriePermutationInverse*(π).

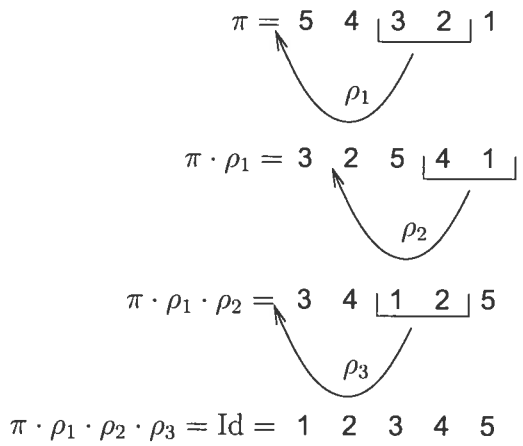


FIG. 3.11 – Exemple de *TriePermutationInverse*(π).

Soit π la permutation inverse de longueur 5. Les transpositions sont $\rho_1(3,5,1), \rho_2(4,6,2)$ et $\rho_3(3,5,1)$.

Notre algorithme *TraiteAscension* (voir Figure 3.12) est une généralisation de l'algorithme *TriePermutationInverse*. *TraiteAscension* trie les terrains plats d'une

ascension sans modifier le reste de la permutation. La figure 3.13 présente un exemple de cet algorithme.

Pour ce faire, il considère chaque terrain plat de l'ascension comme étant un élément d'une permutation inverse. Il tient compte du nombre de terrains plats dans l'ascension plutôt que de sa longueur. Soit T_x le x-ième terrain plat de l'ascension. Soit $a = T_1 \dots T_i \dots T_j \dots T_k \dots T_m$ ou $T_1 \dots T_k \dots T_i \dots T_j \dots T_m$.

Une **transposition sur des terrains plats**, notée $\rho_{terrains\ plats}(i, j, k)$, transpose les terrains plats $T_i T_{i+1} \dots T_{j-1}$ entre les terrains plats T_{k-1} et T_k.

Algorithme TraiteAscension(entrée : une permutation π de [n])
$cg(\pi)$ = code gauche de π (calculé facilement en $O(n^2)$)
Chercher une ascension (cherchée facilement en $O(n^2)$)
S'il y a une ascension a de m terrains plats
    $i = \lceil \frac{m}{2} \rceil$
    $k = 1$
    Pour j de 1 à $\lfloor \frac{m}{2} \rfloor$
        $a = a \cdot \rho_{terrains\ plats}(i, i+2, k)$
        $i = i + 1$
        $k = k + 1$
    $a = a \cdot \rho_{terrains\ plats}(\lfloor \frac{m}{2} \rfloor + 1, 2\lfloor \frac{m}{2} \rfloor + 1, 1)$
(exécuté facilement en $O(n)$ comme *TriePermutationInverse*)

FIG. 3.12 – Algorithme *TraiteAscension*(π).

Théorème 3.8. *Soit ρ_i la i-ème transposition de* TraiteAscension *sur une permutation π contenant une ascension de m terrains plats. On a que $\Delta p(\rho_i)$ sera en moyenne ≤ -1.*

Preuve. L'algorithme *TraiteAscension* (comme l'algorithme *TriePermutationInverse*) est un algorithme qui trie complètement les terrains plats faisant partie de l'ascension. La preuve est une généralisation simple des preuves de [MWD97, Chr98, EEK+01].

Il y a deux cas possibles :

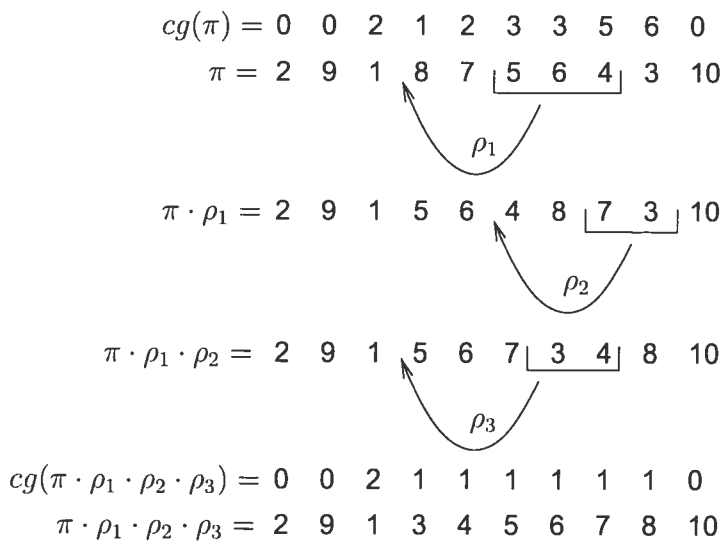


FIG. 3.13 – Exemple de *TraiteAscension*(π).

Nous avons calculé le code gauche de $\pi = 2\ 9\ 1\ 8\ 7\ 5\ 6\ 4\ 3\ 10$. Une ascension de cinq terrains plats couvre les posiitons 4 à 9 : 1 2 3 3 5 6. Les transpositions sur des terrains plats sont calculées en ignorant le reste de la permutation. On renumérote les terrains plats de l'ascension en commençant par l'indice 1. Rappelons que les indices des transpositions sur des terrains plats font référence aux terrains plats et non pas aux éléments. Nous représentons les transpositions sur des terrains plats par ρ dans la figure plutôt que par $\rho_{terrains\ plats}$ pour plus de clarté. Les transpositions sur des terrains plats sont $\rho_{(terrains\ plats)1}(3,5,1)$, $\rho_{(terrains\ plats)2}(4,6,2)$ et $\rho_{(terrains\ plats)3}(3,5,1)$.

- L'ascension débute par une plage non maximale suivie de $m-1$ plateaux. Dans ce cas, trier l'ascension implique que les $m-1$ plateaux seront annulés (leur valeur devient 0) et ce en $\lfloor \frac{m}{2} \rfloor + 1$ transpositions.
- L'ascension débute par un plateau de valeur v. Dans ce cas, trier l'ascension implique que les $m-1$ autres plateaux auront aussi la valeur v. Encore une fois, $m-1$ plateaux disparaissent en $\lfloor \frac{m}{2} \rfloor + 1$ transpositions.

Si le nouveau terrain plat créé par *TraiteAscension* fusionne avec le terrain plat à sa gauche ou à sa droite, cela fait diminuer encore plus le nombre de plateaux. Donc dans le pire cas, *TraiteAscension* élimine $m-1$ plateaux en $\lfloor \frac{m}{2} \rfloor + 1$ transpositions.

On a donc en moyenne

$$\Delta p(\rho_i) \leq \frac{-(m-1)}{\lfloor \frac{m}{2} \rfloor + 1} .$$

On peut facilement vérifier que

$$\frac{-(m-1)}{\lfloor \frac{m}{2} \rfloor + 1} \leq -1, \forall m \geq 4 .$$

Par la définition d'ascension, $m \geq 4$. Cela implique que $\Delta p(\rho_i)$ sera en moyenne ≤ -1. ■

3.2.4.3 Complexité en temps

TraiteAscension n'est pas un algorithme de tri complet. Il ne trie que les terrains plats dans l'ascension. Le reste de la permutation demeure non triée.

Théorème 3.9. *La complexité de* TraiteAscension *est dans* $O(n^2)$.

Preuve. Voir la description de l'algorithme *TraiteAscension* à la figure 3.12. ■

3.2.5 Descentes

Rappelons que $\Delta p(\rho)$ est la variation du nombre de plateaux entraînée par la transposition ρ. Nous introduisons maintenant trois objets pour lesquels on teste $\Delta p(\rho)$, où ρ est la transposition associée à l'objet en question, pour décider si on fait la transposition ou non. Le test se fait sur une copie de la permutation. On ne transpose la descente, la montée ou le fossé symétrique que si $\Delta p(\rho)$ est préférable au Δp d'une transposition de *TriePlateau*.

Nous présentons d'abord les descentes.

3.2.5.1 Définitions

Définitions 3.8. *Soient m plateaux consécutifs $P_i P_{i+1} \ldots P_{i+m-1}$. Ces plateaux forment une* **descente** *si $v(P_i) > v(P_{i+1}) > \ldots > v(P_{i+m-1})$. Ils forment une* **descente courte** *si $m = 2$.*

Définition 3.9. *Une descente est dite* **bonne** *s'il existe une position k dans la permutation telle que la transposition de la descente juste avant k enlève au moins deux plateaux.*

3.2.5.2 Traitement

L'algorithme *TraiteDescente* (voir Figure 3.14) teste chaque descente de la permutation et ne la transpose que si elle est bonne.

Algorithme TraiteDescente(entrée : une permutation π de [n])
posInsertionBonneDescente $= -1$
bonneDescente $= null$
$cg(\pi)$ = code gauche de π (calculé facilement en $O(n^2)$)
$pg(\pi)$ = nombre de plateaux dans $cg(\pi)$ (calculé facilement en $O(n)$)
Chercher toutes les descentes (cherchées facilement en $O(n^2)$)
Pour chaque descente *descente* tant que *bonneDescente* $= null$
(Le nombre de descentes $\in O(n^2)$. Voir la preuve.)
    Pour chaque position d'insertion k (leur nombre $\in O(n)$)
    tant que *posInsertionBonneDescente* $= -1$
        π^c = copie de π (en $O(n)$)
        $\pi^c \cdot \rho$ = résultat de la transposition de *descente* à gauche de k sur la copie de π ($O(1)$ avec une liste chaînée)
        $cg(\pi^c \cdot \rho)$ = code gauche de $\pi^c \cdot \rho$ (calculé facilement en $O(n^2)$)
        $pg(\pi^c \cdot \rho)$ = nombre de plateaux dans $cg(\pi^c \cdot \rho)$ (calculé facilement en $O(n)$)
        $\Delta p(\rho) = pg(\pi^c \cdot \rho) - pg(\pi)$
        Si $\Delta p(\rho) \leq -2$
            bonneDescente $=$ *descente*
            posInsertionBonneDescente $= k$
Si *bonneDescente* $\neq null$
    Transposer ($O(1)$ avec une liste chaînée)

FIG. 3.14 – Algorithme *TraiteDescente(π)*.

Il y a plusieurs positions d'insertion à tester pour chaque descente. Nous avons

conçu une heuristique pour essayer de deviner la position d'insertion la plus prometteuse. Les indices sont des positions d'éléments et non plus des positions de plateaux.

Définition 3.10. *Soit une descente* $desc = \pi_g \pi_{g+1} \dots \pi_d$. *On pose que* $\pi_0 = 0$ *et que* $\pi_{n+1} = n + 1$. *Le* **score d'insérabilité** *de la descente* desc *à la position d'insertion* k *est*

$$\begin{cases} \pi_g - \pi_{k-1} + \pi_k - \pi_d & si\ \pi_{k-1} < \pi_g\ et\ \pi_d < \pi_k \\ +\infty & sinon \end{cases}$$

L'algorithme *TraiteDescenteHeuristique* (voir Figure 3.15) ne teste que la position d'insertion pour laquelle le score d'insérabilité est minimal et différent de $+\infty$ pour chaque descente.

Algorithme TraiteDescenteHeuristique(entrée : une permutation π de [n])
scoreInserabilite
meilleurScoreInserabilite $= {}_{+}\infty$
posInsertionBonneDescente $= -1$
bonneDescente $= null$
$cg(\pi)$ = code gauche de π (calculé facilement en $O(n^2)$)
$pg(\pi)$ = nombre de plateaux dans $cg(\pi)$ (calculé facilement en $O(n)$)
Chercher toutes les descentes (cherchées facilement en $O(n^2)$)
Pour chaque descente *descente* tant que *bonneDescente* $= null$ (Le nombre de descentes $\in O(n^2)$.)
    Pour chaque position d'insertion *kTempo* (leur nombre $\in O(n)$)
        scoreInserabilite = score d'insérabilité de *kTempo* (calculé en $O(1)$)
        Si *scoreInserabilite* $<$ *meilleurScoreInserabilite*
            $k = kTempo$
            meilleurScoreInserabilite = *scoreInserabilite*
    π^c = copie de π (en $O(n)$)
    $\pi^c \cdot \rho$ = résultat de la transposition de *descente* à gauche de k sur la copie de π ($O(1)$ avec une liste chaînée)
    $cg(\pi^c \cdot \rho)$ = code gauche de $\pi^c \cdot \rho$ (calculé facilement en $O(n^2)$)
    $pg(\pi^c \cdot \rho)$ = nombre de plateaux dans $cg(\pi^c \cdot \rho)$ (calculé facilement en $O(n)$)
    $\Delta p(\rho) = pg(\pi^c \cdot \rho) - pg(\pi)$
    Si $\Delta p(\rho) \leq -2$
        bonneDescente = *descente*
        posInsertionBonneDescente $= k$
Si *bonneDescente* $\neq null$
    Transposer ($O(1)$ avec une liste chaînée)

FIG. 3.15 – Algorithme *TraiteDescenteHeuristique(π)*.

Théorème 3.10. *Le score d'insérabilité d'une position dans laquelle on insère une descente est* ≥ 2. ∎

Ce théorème permet d'arrêter de tester les positions d'insertion dès qu'on en a une dont le score d'insérabilité est 2. Un exemple de *TraiteDescenteHeuristique* est présenté à la figure 3.16.

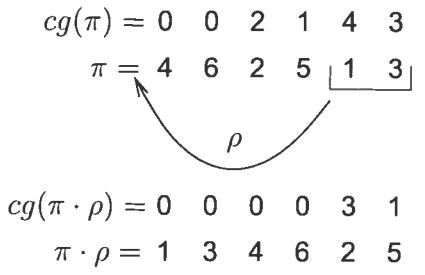


FIG. 3.16 – Exemple de *TraiteDescenteHeuristique*(π).
Nous avons calculé le code gauche de $\pi = 4\ 6\ 2\ 5\ 1\ 3$. Une descente couvre les positions 3 et 4 : 2 1. Il n'existe aucune position d'insertion k ayant un score d'insérabilité différent de $+\infty$ et donc on ne teste pas cette descente. Une deuxième descente couvre les positions 5 et 6 : 4 3. Le score d'insérabilité de la position d'insertion $k = 1$ est $\pi_5 - \pi_0 + \pi_1 - \pi_6 = 1 - 0 + 4 - 3 = 2$. C'est un score d'insérabilité minimal. La transposition $\rho(5, 7, 1)$ éliminerait deux plateaux. La descente est bonne et on choisit cette transposition.

Théorème 3.11. *Soit* ρ *la transposition de* TraiteDescente *ou* TraiteDescenteHeuristique *sur* π. *On a que* $\Delta p(\rho) \leq -2$.

Preuve. Par la définition de bonne descente. ∎

3.2.5.3 Complexité en temps

Chaque exécution de l'algorithme *TraiteDescente* ne fait que déplacer une seule descente. Ce n'est pas un algorithme de tri complet.

Théorème 3.12. *La complexité de* TraiteDescente *est dans* $O(n^5)$.

Preuve. Voir la description de l'algorithme *TraiteDescente* à la figure 3.14. Les descentes peuvent mesurer plus que deux plateaux et se chevaucher. Le nombre de positions possibles pour le début d'une descente est dans $O(n)$. Le nombre de positions possibles pour la fin d'une descente est dans $O(n)$. Le nombre de descentes possibles est donc dans $O(n^2)$. ■

On peut diminuer la complexité de l'algorithme en ne cherchant que des descentes courtes. *TraiteDescenteCourte* est une version de l'algorithme qui fait cela.

Théorème 3.13. *La complexité de* TraiteDescenteCourte *est dans* $O(n^4)$.

Preuve. La seule différence avec *TraiteDescente* est que le nombre de descentes possibles est dans $O(n)$ plutôt que dans $O(n^2)$. ■

L'heuristique utilisant le score d'insérabilité permet de réduire la complexité.

Théorème 3.14. *La complexité de* TraiteDescenteHeuristique *est dans* $O(n^4)$.

Preuve. Voir la description de l'algorithme *TraiteDescenteHeuristique* à la figure 3.15. ■

Théorème 3.15. *La complexité de* TraiteDescenteCourteHeuristique *est dans* $O(n^3)$.

Preuve. La preuve est similaire à celle du théorème 3.13. On compare *TraiteDescenteCourteHeuristique* avec *TraiteDescenteHeuristique* plutôt qu'avec *TraiteDescente.* ■

Il est intéressant de noter ici deux idées pour améliorer la complexité de l'algorithme *TraiteDescente.* Une première idée est, étant donné une descente, de ne considérer que les positions à gauche de celle-ci comme positions possibles d'insertion. C'est une idée que nous avons réellement implémentée (voir l'annexe II). Une deuxième idée que nous n'avons pas implémentée par manque de temps serait d'optimiser le calcul du code de la permutation après la transposition de la descente (voir la section 5.1.2).

3.2.6 Montées

Les montées sont comme les descentes mais les valeurs des codes augmentent de gauche à droite au lieu de diminuer. Les définitions, le traitement et l'analyse de la complexité d'une montée sont symétriques à une descente (voir la section 3.2.5).

3.2.6.1 Définitions

Définitions 3.11. *Soient m plateaux consécutifs $P_iP_{i+1}\dots P_{i+m-1}$. Ces plateaux forment une* **montée** *si $v(P_i) < v(P_{i+1}) < \dots < v(P_{i+m-1})$. Ils forment une* **montée courte** *si $m = 2$.*

La même condition que pour une descente s'applique pour dire qu'une montée est bonne (voir définition 3.9).

3.2.6.2 Traitement

L'algorithme *TraiteMontée* traite les montées comme *TraiteDescente* traite les descentes. On utilise le même score d'insérabilité dans l'heuristique (voir définition 3.10).

Théorème 3.16. *Le score d'insérabilité d'une position dans laquelle on insère une montée est ≥ 2.*

Preuve. Voir théorème 3.10. ∎

La figure 3.17 présente un exemple de *TraiteMontéeHeuristique*.

Comme pour les bonnes descentes, la définition de bonne montée garantit une certaine diminution du nombre de plateaux.

Théorème 3.17. *Soit ρ la transposition de* TraiteMontée *ou* TraiteMontéeHeuristique *sur π. On a que $\Delta p(\rho) \leq -2$.* ∎

3.2.6.3 Complexité en temps

Les algorithmes *TraiteMontée*, *TraiteMontéeCourte*, *TraiteMontéeHeuristique* et *TraiteMontéeCourteHeuristique* sont similaires à *TraiteDescente*, *TraiteDescen-*

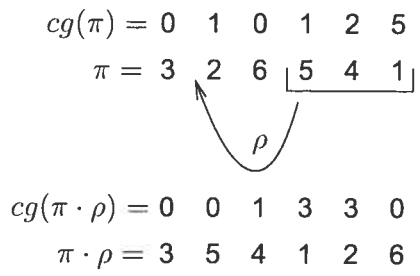


FIG. 3.17 – Exemple de *TraiteMontéeHeuristique*(π).
Nous avons calculé le code gauche de $\pi = 3\ 2\ 6\ 5\ 4\ 1$. Une montée couvre les positions 4 à 6 : 1 2 5. Le score d'insérabilité de la position d'insertion $k = 2$ est $\pi_4 - \pi_1 + \pi_2 - \pi_6 = 5-3+2-1 = 3$. C'est un score d'insérabilité minimal pour cette permutation et cette montée. La transposition $\rho(4, 6, 2)$ éliminerait deux plateaux. La montée est bonne et on choisit cette transposition.

teCourte, *TraiteDescenteHeuristique* et *TraiteDescenteCourteHeuristique* respectivement. Les algorithmes impliquant des montées ont la même complexité que les algorithmes impliquant des descentes.

Théorème 3.18. *La complexité de* TraiteMontée *est dans* $O(n^5)$. ■

Théorème 3.19. *La complexité de* TraiteMontéeCourte *est dans* $O(n^4)$. ■

Théorème 3.20. *La complexité de* TraiteMontéeHeuristique *est dans* $O(n^4)$. ■

Théorème 3.21. *La complexité de* TraiteMontéeCourteHeuristique *est dans* $O(n^3)$. ■

3.2.7 Fossés symétriques

3.2.7.1 Définitions

On conserve la définition 3.6 de fossé.

Définition 3.12. *Un fossé est* **symétrique** *si* $v(P_i) = v(P_{i+m-1})$.

Définition 3.13. *Un fossé symétrique est dit* **bon** *s'il existe une position k dans la permutation telle que la transposition du fond du fossé symétrique juste avant k crée une bonne descente (voir définition 3.9).*

3.2.7.2 Traitement

L'algorithme *TraiteFosséSymétrique* teste chaque fossé symétrique de la permutation et ne le transpose que s'il est bon. Puisqu'il y a plusieurs positions d'insertion à tester pour chaque fossé symétrique, la complexité en temps de *TraiteFosséSymétrique* est exagérément grande. Nous utiliserons plutôt la même heuristique que pour l'algorithme *TraiteFosséAsymétrique* pour essayer de deviner la position d'insertion la plus prometteuse pour concevoir l'algorithme *TraiteFosséSymétriqueHeuristique* (voir Figure 3.18). La seule position d'insertion testée est celle du plus petit élément, parmi les éléments plus grands que le dernier élément du fond du fossé, et à gauche de celui-ci. Nous utilisons ensuite l'algorithme *TraiteDescenteHeuristique* pour tester si cette transposition a créé une bonne descente. Un exemple de *TraiteFosséSymétriqueHeuristique* est présenté à la figure 3.19.

Théorème 3.22. *Soit ρ_1 et ρ_2 les deux transpositions de* TraiteFosséSymétrique-Heuristique *sur une permutation π. La moyenne de $\Delta p(\rho_1)$ et $\Delta p(\rho_2)$ peut ne pas être ≤ -1.*

Preuve. Voir le contre-exemple à la figure 3.19. On a que $\Delta p(\rho_1) = 1$ et $\Delta p(\rho_2) = -2$. ∎

3.2.7.3 Complexité en temps

Chaque exécution de l'algorithme *TraiteFosséSymétriqueHeuristique* ne fait que traiter un seul fossé symétrique. Ce n'est pas un algorithme de tri complet.

Théorème 3.23. *La complexité de* TraiteFosséSymétriqueHeuristique *est dans $O(n^3)$.*

Preuve. Voir la description de l'algorithme *TraiteFosséSymétriqueHeuristique* à la figure 3.18. Selon la définition de fossé symétrique, à une position de début, il ne correspond qu'une seule position de fin. Le nombre de fossés symétriques possibles est donc dans $O(n)$. ∎

Algorithme TraiteFosséSymétriqueHeuristique (entrée : une permutation π de [n])
bonFosseSymetrique $= null$
$cg(\pi)$ = code gauche de π (calculé facilement en $O(n^2)$).
Chercher tous les fossés symétriques (cherchés facilement en $O(n^2)$).
Pour chaque fossé symétrique *fosSym* tant que *bonFosseSymetrique* $= null$ (Le nombre de fossés symétriques est $\in O(n)$. Voir la preuve.)
    π^c = copie de π (en $O(n)$).
    Chercher la position d'insertion k_1 (trouvée facilement en $O(n)$).
    $\pi^c \cdot \rho_1$ = Résultat de la transposition du fond de *fosSym* à gauche de k_1 sur la copie de π. Une descente *descente* est créée. ($O(1)$ avec une liste chaînée.)
    $cg(\pi^c \cdot \rho_1)$ = code gauche de $\pi^c \cdot \rho_1$ (calculé facilement en $O(n^2)$).
    $pg(\pi^c \cdot \rho_1)$ = nombre de plateaux dans $cg(\pi^c \cdot \rho_1)$ (calculé facilement en $O(n)$).
    Chercher la position d'insertion k_2 qui a le score d'insérabilité minimal par rapport à la descente (Trouvée facilement en $O(n)$. Voir figure 3.15.)
    $\pi^c \cdot \rho_1 \cdot \rho_2$ = Résultat de la transposition de *descente* à gauche de k_2 sur $\pi^c \cdot \rho_1$ ($O(1)$ avec une liste chaînée).
    $cg(\pi^c \cdot \rho_1 \cdot \rho_2)$ = code gauche de $\pi^c \cdot \rho_1 \cdot \rho_2$ (calculé facilement en $O(n^2)$).
    $pg(\pi^c \cdot \rho_1 \cdot \rho_2)$ = nombre de plateaux dans $cg(\pi^c \cdot \rho_1 \cdot \rho_2)$ (calculé facilement en $O(n)$).
    $\Delta p(\rho_2) = pg(\pi^c \cdot \rho_1 \cdot \rho_2) - pg(\pi^c \cdot \rho_1)$
    Si $\Delta p(\rho_2) \leq -2$
        bonFosseSymetrique $=$ *fosSym*
Si *bonFosseSymetrique* $\neq$ *null*
    Transposer par ρ_1 et ρ_2 ($O(1)$ avec une liste chaînée).

FIG. 3.18 – Algorithme *TraiteFosséSymétriqueHeuristique(π).*

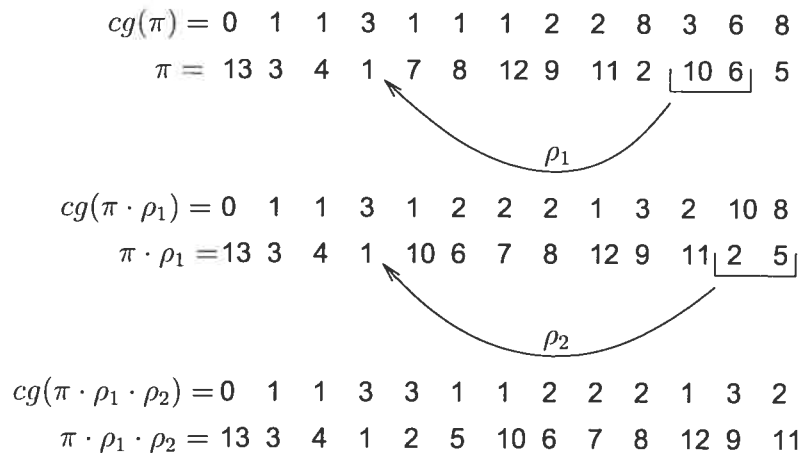


FIG. 3.19 – Exemple de *TraiteFosséSymétriqueHeuristique*(π).
Nous avons calculé le code gauche de $\pi = 13\ 3\ 4\ 1\ 7\ 8\ 12\ 9\ 11\ 2\ 10\ 6\ 5$. Un fossé symétrique couvre les positions 10 à 13 : 8 3 6 8. Le fond du fossé couvre les positions 11 et 12. Le dernier élément du fond du fossé est 6. Le plus petit élément parmi ceux qui sont plus grands que 6 tout en étant à sa gauche est 7 à la position 5. La transposition $\rho_1(11, 13, 5)$ créerait une bonne descente aux positions 12 et 13 : 10 8 parce qu'elle pourrait être transposée à la position 5 par la transposition $\rho_2(12, 14, 5)$ pour faire diminuer le nombre de plateaux de deux. Le fossé symétrique est bon et on exécute les transpositions ρ_1 et ρ_2.

3.3 Des objets aux algorithmes

Le but de ce mémoire n'est pas de présenter un seul algorithme en particulier. C'est plutôt de montrer comment on peut concevoir différents algorithmes de tri par transpositions en utilisant le code d'une permutation. Nous ne prétendons pas avoir exploité toutes les pistes prometteuses de ce concept. Le lecteur peut combiner librement entre eux les objets présentés à la section précédente comme on combine des blocs de construction pour créer différents algorithmes. Il peut même inventer de nouveaux objets ou modifier la définition d'un objet déjà existant. Nous présentons d'abord, au début de cette section, comment élaborer en général des algorithmes de tri par transpositions qui utilisent le code d'une permutation. Nous présentons et analysons des exemples de tels algorithmes ensuite.

3.3.1 Élaboration de nos algorithmes

Nous devons d'abord choisir quels objets seront considérés (voir la section 3.2). Nous pouvons prendre un ou des objets parmi les suivants : les plateaux, les montagnes, les fossés asymétriques, les ascensions, les descentes, les montées et les fossés symétriques. Le choix des objets influence la qualité de l'approximation de la distance de transposition d'une permutation et la complexité en temps de l'algorithme. Nous devons obligatoirement considérer l'objet plateau pour obtenir un algorithme qui trie au complet une permutation parce que *TriePlateau* est le seul algorithme de tri complet.

Traiter un objet (traiter **une** montagne par exemple) signifie que l'algorithme de la section 3.2 associé à cet objet est exécuté une seule fois. Traiter une sorte d'objet (traiter **les** montagnes par exemple) signifie plutôt traiter un objet répétitivement jusqu'à ce que la permutation ne contienne plus cette sorte d'objet.

Les algorithmes que nous présentons ne traitent qu'une seule sorte d'objet à la fois. Nous devons donc choisir l'ordre de priorité du traitement des objets de nature différente. Traitons-nous les montagnes ou les descentes en premier ? Un tel choix peut modifier la distance approximative de transposition obtenue pour une permutation. Puisque *TriePlateau* trie une permutation au complet, notons qu'il est inutile de traiter d'autres sortes d'objets après avoir traité les plateaux. C'est pourquoi tous nos algorithmes se terminent par *TriePlateau*.

Nous utilisons la convention suivante pour nommer un algorithme. Le nom de l'algorithme commence par le préfixe "Trie" indiquant qu'il s'agit d'un algorithme de tri complet. Le nom de chaque sorte d'objet traitée suit dans l'ordre dans lequel les sortes d'objets sont traitées. Nous utilisons toujours la version heuristique des algorithmes qui traitent les descentes, les montées et les fossés symétriques. Ainsi, *TrieDescentePlateau* signifie que l'on exécute répétitivement l'algorithme *TraiteDescenteHeuristique* tant qu'il reste des bonnes descentes avant d'exécuter l'algorithme *TriePlateau*.

3.3.2 Garantie de performance de *TriePlateau*

Le principal intérêt de *TriePlateau* (voir Figure 3.4) est sa complexité en temps. Nous pouvons cependant aussi prouver que le nombre de transpositions nécessaires pour trier la permutation π par *TriePlateau* approxime la vraie distance de transposition par un facteur entre 1 et 3 qui dépend de π. Des résultats expérimentaux obtenus avec *TriePlateau* sont présentés dans chaque tableau de la section 4.2.

3.3.2.1 Points de cassure

Nous définissons d'abord le concept de points de cassure (*breakpoints*). Ce concept est classique en génomique comparée.

Définition 3.14. *Soit une permutation π de $[n]$, nous disons que nous avons un* **point de cassure** *en π_i si $\pi_i + 1 \neq \pi_{i+1}$, pour $0 \leq i \leq n$. Le nombre de points de cassure de π est noté $b(\pi)$.*

Pour visualiser les points de cassure d'une permutation $\pi = \pi_1\pi_2 \dots \pi_n$, il peut être utile de l'étendre, comme dans [WDM00], par les deux éléments $\pi_0 = 0$ et $\pi_{n+1} = n + 1$. Cette permutation étendue est encore notée π.

Exemple 3.1. *Ainsi, la permutation $\pi^1 = 6\ 3\ 2\ 1\ 4\ 5$ des figures 3.1 et 3.3 devient $\pi^1 = \mathbf{0}\ 6\ 3\ 2\ 1\ 4\ 5\ \mathbf{7}$. Chaque élément parmi ses sept premiers a un point de cassure sauf 4 à la position 5. La permutation π^1 a donc six points de cassure.*

Le lemme 3.24 est un lemme classique concernant les points de cassure.

Lemme 3.24. *Soit une permutation π de $[n]$, nous avons $\frac{b(\pi)}{3} \leq d(\pi) \leq b(\pi)$.*

Preuve. Il y a trois endroits dans la permutation où celle-ci est coupée lors d'une transposition. C'est pourquoi il y a au plus trois points de cassure qui peuvent être enlevés lors d'une transposition. Cette observation conduit à l'inégalité $\frac{b(\pi)}{3} \leq d(\pi)$.

On observe aussi qu'une permutation peut être triée en transposant chaque élément π_i à sa "bonne position". Une telle transposition enlève au moins un point de cassure à chaque fois. Cette deuxième observation mène à l'inégalité $d(\pi) \leq b(\pi)$.

■

3.3.2.2 Garantie de performance

Nous présentons la garantie de performance (voir définition 2.1) de *TriePlateau*. Elle est de 3. Nous avons aussi trouvé une formule qui fait varier la "garantie de performance" en fonction de la permutation π sans jamais dépasser 3. Nous appelons le résultat de cette formule le **facteur d'approximation théorique** de l'algorithme.

Définition 3.15. *Le* **facteur d'approximation théorique** *d'un algorithme pour une permutation π est le facteur par lequel il faut multiplier la réponse exacte pour obtenir la pire approximation possible par l'algorithme pour π.*

C'est en fait un synonyme de garantie de performance mais ne s'appliquant qu'à une seule permutation en particulier.

Théorème 3.25. *Un facteur d'approximation théorique de $d(\pi)$, pour une permutation π, avec l'algorithme* TriePlateau *est*

$$\frac{c \cdot p(\pi)}{b(\pi)}, \quad où \quad c = \frac{3\left\lfloor \frac{b(\pi)}{3} \right\rfloor + b(\pi) \bmod 3}{\left\lceil \frac{b(\pi)}{3} \right\rceil}$$

Preuve. Le facteur d'approximation théorique est le nombre de transpositions nécessitées par l'algorithme *TriePlateau* pour trier la permutation π divisé par une borne inférieure de $d(\pi)$. *TriePlateau* trie π en $p(\pi)$ transpositions (voir Figure 3.4). Le lemme 3.24 donne une borne inférieure à $d(\pi)$, $\frac{b(\pi)}{3}$. Cette borne inférieure est obtenue quand chaque transposition réduit le nombre de points de cassure par 3. Il est possible d'augmenter un peu cette borne inférieure (et de diminuer le facteur d'approximation théorique) en tenant compte des permutations pour lesquelles $b(\pi)$ n'est pas un multiple de 3. Soit la constante c le nombre maximal de points de cassure qu'il est possible d'enlever lors d'une transposition. Si $b(\pi)$ est un multiple de 3, alors $c = 3$. Sinon, nous devons faire au moins une transposition qui réduira

le nombre de points de cassure par un terme plus petit que 3 ($b(\pi)$ mod 3 exactement). La nouvelle borne inférieure de $d(\pi)$ devient $\frac{b(\pi)}{c}$. On obtient le facteur d'approximation théorique en divisant $p(\pi)$ par $\frac{b(\pi)}{c}$. ■

Théorème 3.26. *La garantie de performance de l'algorithme* TriePlateau *est* 3 *(peu importe la permutation π).*

Preuve. On a que $c \leq 3$ (théorème 3.25). Puisque chaque début de plateau correspond à un point de cassure, on a que

$$p(\pi) \leq b(\pi)$$

Donc,

$$\frac{p(\pi)}{b(\pi)} \leq 1$$

Donc, le facteur d'approximation théorique de π (théorème 3.25) est

$$c \cdot \frac{p(\pi)}{b(\pi)} \leq 3$$

■

Exemple 3.2. *Revenons à la permutation $\pi^1 = 6\ 3\ 2\ 1\ 4\ 5$ de la figure 3.1. La figure 3.3 explique pourquoi $p(\pi^1) = 3$. L'exemple 3.1 explique que $b(\pi^1) = 6$. La définition de c au théorème 3.25 permet de calculer que $c = 3$. Le facteur d'approximation théorique est donc $\frac{3p(\pi^1)}{b(\pi^1)} = \frac{3\times3}{6} = 1,5$. Il est intéressant de noter que le facteur d'approximation réel est 1 parce que $d(\pi^1) = 3$ en fait.*

La borne supérieure du facteur d'approximation théorique du théorème 3.25 n'est pas serrée (*tight*) pour certaines permutations π de $[n]$, que nous n'avons pas classifiées pour l'instant. Par exemple, [MWD97, Chr98, EEK+01] ont prouvé que $d(\pi) = \lfloor \frac{n}{2} \rfloor + 1$ si π est la permutation inverse ($\pi = n\ n{-}1 \ldots 2\ 1$) grâce à l'algorithme *TriePermutationInverse* (voir Figure 3.10). Dans ce cas, $p(\pi) = n - 1$ et $b(\pi) = n + 1$. Nous avons le facteur d'approximation théorique, donné par le théorème 3.25, $\lim_{n\to\infty} \frac{c \cdot p(\pi)}{b(\pi)} = 3$. C'est supérieur au facteur d'approximation

réel, $\lim_{n\to\infty} \frac{p(\pi)}{d(\pi)} = 2$. Un autre exemple de cette borne non serrée est présenté à l'exemple 3.2.

3.3.3 Présentation générale de nos algorithmes

Nous présentons dans cette sous-section comment concevoir un algorithme dont on peut prouver une garantie de performance et une complexité en temps à partir des objets de la section 3.2.

Soit un algorithme *TrieObjets* (autre que *TriePlateau*) élaboré en combinant des objets à la manière décrite à la section 3.3.1. Soit l'algorithme *TraiteUnObjet* un algorithme qui cherche et traite, s'il en trouve un, un des objets de *TrieObjets*. C'est un algorithme comme ceux présentés à la section 3.2. La sorte d'objet cherchée dépend de la sorte d'objet que l'on est rendu à chercher. L'algorithme *TrieObjets* peut être décrit comme l'exécution itérée de *TraiteUnObjet* sur une permutation jusqu'à ce que celle-ci soit triée.

Soit *TraiteObjetMax* l'algorithme qui traite un des objets de *TrieObjets* et qui a une complexité en temps maximale. Posons que la complexité en temps de *TraiteObjetMax* est dans $O(n^d)$. La complexité en temps maximale de *TraiteUnObjet* sera donc dans $O(n^d)$.

Par exemple, si *TrieObjets* = *TrieMontagneDescenteCourtePlateau*, alors *TraiteObjetMax* = *TraiteDescenteCourte(Heuristique)* (rappelons que la version heuristique est utilisée par défaut) dont la complexité en temps est dans $O(n^3)$ (théorèmes 3.3, 3.5 et 3.15). La complexité maximale de *TraiteUnObjet* sera donc dans $O(n^3)$ aussi dans ce cas.

L'astuce pour prouver une garantie de performance et une complexité en temps est de faire une méthode *Branch and Bound* avec *TriePlateau* pendant que l'on exécute *TrieObjets*. Nous appelons ce nouvel algorithme *TrieObjetsBnB*. Il est décrit à la figure 3.20.

Si on désire obtenir la liste des transpositions à faire pour trier π en plus de la distance de transposition approximative, on peut lister les transpositions exécutées par *TrieObjets* au fur et à mesure qu'elles sont faites. On retourne cette liste si on

```
Algorithme TrieObjetsBnB(entrée : une permutation π de [n])
p(π) = TriePlateau(π) (voir Figure 3.4) (en O(n²))
dist = 0
Tant que π ≠ Id et que dist < p(π)
        π = TraiteUnObjet(π) (au plus en O(n^d))
        Si un objet a été trouvé et une transposition a été faite
                dist = dist + 1
Si π = Id (vérifié en O(n))
        RETOURNER dist
Sinon
        RETOURNER p(π)
```

FIG. 3.20 – Algorithme *TrieObjetsBnB(π)*.

retourne *dist* alors qu'on retourne la liste des transpositions trouvées par *TriePlateau* si on retourne $p(\pi)$.

3.3.3.1 Garantie de performance

Nous n'avons pas réussi à prouver une garantie de performance à l'algorithme *TrieObjets* (sans la méthode *Branch and Bound*). Cela devient facile avec la méthode *Branch and Bound.*

Puisque l'algorithme *TrieObjetsBnB* trie une permutation π au moins aussi rapidement que *TriePlateau*, *TrieObjetsBnB* a le même facteur d'approximation théorique sur π (théorème 3.25) et la même garantie de performance de 3 (théorème 3.26) que *TriePlateau.*

3.3.3.2 Complexités en temps

La complexité de *TrieObjets* dépend du nombre de fois que l'algorithme *TraiteUnObjet* est répété. Si on sait que le nombre de fois qu'il est répété est dans $O(n)$, on peut en déduire que la complexité en temps est dans $O(n^{d+1})$. Intuitivement, il serait surprenant que le nombre de transpositions exécutées par *TrieObjets* pour trier la permutation π de $[n]$ soit supérieur au nombre d'éléments dans π. Malheu-

reusement, nous n'avons pas réussi à prouver rigoureusement que c'est impossible dans tous les cas.

Nous avons réussi à prouver une complexité de *TrieObjets* dans $O(n^{d+1})$ seulement dans le cas suivant.

Théorème 3.27. *Soit un algorithme* TrieObjets *qui n'utilise que des objets parmi les suivants : plateau, ascension, descente et montée. Soit la complexité de* TraiteObjetMax *dans* $O(n^d)$. *La complexité de ce* TrieObjets *est dans* $O(n^{d+1})$.

Preuve. La permutation π de $[n]$ a au départ $p(\pi) < n$ plateaux. Nous savons qu'il y aura au moins un plateau de moins à chaque exécution de *TraiteUnObjet* par le lemme 3.1 et les théorèmes 3.8, 3.11 et 3.17. Puisqu'une permutation est triée si et seulement si $p(\pi) = 0$, π sera triée en moins de n transpositions. *TraiteUnObjet* sera répété moins de n fois quand il trouve un objet.

Soit la constante *cte* le nombre de sortes d'objets recherchées par *TrieObjets*. Si un objet n'est pas trouvé dans la permutation, on cherchera l'objet suivant et donc *TraiteUnObjet* sera répété au plus $cte - 1$ fois quand il ne trouve pas un objet.

TraiteUnObjet sera donc répété moins de $n + cte - 1$ fois au total. On peut alors en déduire le théorème. ∎

Nous ne pouvons pas faire le même raisonnement dans le cas où *TrieObjets* utilise les objets montagne, fossé asymétrique ou fossé symétrique à cause des théorèmes 3.4, 3.6 et 3.22 respectivement. La complexité de l'algorithme *TrieObjetsBnB* est cependant facile à prouver.

Théorème 3.28. *Soit la complexité de* TraiteObjetMax *dans* $O(n^d)$. *La complexité de* TrieObjetsBnB *est dans* $O(n^{d+1})$.

Preuve. Voir la description de l'algorithme *TrieObjetsBnB* à la figure 3.20. Soit la permutation π de $[n]$. Puisque $p(\pi) < n$, l'algorithme *TraiteUnObjet* sera répété moins de n fois quand il trouve un objet.

Soit la constante *cte* le nombre de sortes d'objets recherchées par *TrieObjetsBnB*. Comme pour la preuve du théorème 3.27, si un objet n'est pas trouvé dans

la permutation, on cherchera l'objet suivant et donc *TraiteUnObjet* sera répété au plus $cte - 1$ fois quand il ne trouve pas un objet.

TraiteUnObjet sera donc répété moins de $n + cte - 1$ fois au total. On peut alors en déduire le théorème. ∎

3.3.4 Exemples d'algorithmes

Nous présentons des exemples d'algorithmes ayant différentes complexités en temps dans cette sous-section. Revenons à l'algorithme *TriePlateau* pour commencer. Nous avons déjà vu que sa complexité est dans $O(n^2)$ (théorème 3.3). Les algorithmes suivants sont des algorithmes *Branch and Bound* de la forme de l'algorithme *TrieObjetsBnB* (voir Figure 3.20).

Théorème 3.29. *La complexité en temps de* TrieMontagnePlateauBnB *est dans* $O(n^3)$.

Preuve. Les objets plateau et montagne sont tous les deux associés à un algorithme qui a une complexité en temps dans $O(n^2)$ (théorèmes 3.3 et 3.5). La complexité de *TrieMontagnePlateauBnB* peut alors être déduite par le théorème 3.28. ∎

Théorème 3.30. *La complexité en temps de* TrieMontagneAscensionFosseAsymétriqueFosseSymétriqueDescenteCourteMontéeCourtePlateauBnB *est dans* $O(n^4)$.

Preuve. Les objets descente courte, montée courte et fossé symétrique sont les objets qui sont associés à un algorithme qui a une complexité en temps maximale. Celle-ci est dans $O(n^3)$ (théorèmes 3.15, 3.21 et 3.23). (Rappelons que la version heuristique est utilisée par défaut.) La complexité de l'algorithme au complet peut alors être déduite par le théorème 3.28. ∎

Théorème 3.31. *La complexité en temps de* TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateauBnB *est dans* $O(n^5)$.

Preuve. Les objets descente et montée sont les objets qui sont associés à une complexité en temps maximale. Elle est dans $O(n^4)$ (théorèmes 3.14 et 3.20). La com-

plexité de l'algorithme au complet peut alors être déduite par le théorème 3.28. ■

Nous n'avons pas implémenté la méthode *Branch and Bound* (voir la section 3.4). Nous avons cependant implémenté les algorithmes *TriePlateau*, *TrieMontagnePlateau* et *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescente-MontéePlateau* sans cette méthode. Nos résultats sont présentés à la section 4.2.

3.4 Implémentations

La section précédente donne les grandes lignes des algorithmes que nous pouvons concevoir avec les codes. Il reste cependant des détails à préciser. Un programmeur qui implémente ces algorithmes a des choix à faire. Nous regroupons ces choix en deux catégories. La première catégorie est justement les décisions que le programmeur a à prendre pour indiquer quelle transposition doit être faite lorsqu'il y a plusieurs transpositions permises par l'algorithme. La deuxième catégorie est composée des modifications que l'on peut apporter à l'algorithme pour le raffiner sans en changer l'idée générale.

3.4.1 Précisions apportées aux algorithmes

3.4.1.1 Code gauche et/ou code droit

Nous avons le choix entre utiliser le code gauche ou le code droit (voir définitions 3.1) dans nos algorithmes. Nous pouvons même utiliser les deux successivement (voir l'algorithme *TriePlateau* à la figure 3.4).

La preuve de la garantie de performance de *TriePlateau* (théorème 3.26) a été écrite en supposant l'utilisation des codes gauche et droit. Cependant, elle tiendrait aussi si on n'utilisait qu'un seul des deux codes. Les garanties de performance des algorithmes de type *TrieObjetsBnB* (voir la section 3.3.3.1) ne seraient pas changées puisqu'elles se basent sur la garantie de performance de *TriePlateau*.

Il est cependant évident qu'utiliser le meilleur des deux codes pour *TriePlateau* ne peut qu'améliorer la qualité de l'approximation comparativement à n'en

utiliser qu'un seul. Nous comparons des résultats obtenus avec *TriePlateau* (les codes gauche et droit) et *TriePlateauGauche* (le code gauche seulement) dans les tableaux 4.1 à 4.6.

Nous n'avons implémenté les algorithmes *TrieMontagnePlateau* et *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau* qu'avec le code gauche. Cependant, les définitions des objets peuvent être modifiées pour s'appliquer aussi au code droit et ainsi donner de meilleurs résultats.

3.4.1.2 Ordre de priorité des objets de même nature

Nous avons abordé le problème du choix de l'ordre de priorité des objets de nature différente à la section 3.3.1. Nous avons essayé différents ordres (résultats non montrés) et le meilleur semble être celui qui correspond à l'algorithme *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau*.

Nous discutons maintenant d'un problème différent. Ce problème est le choix de l'ordre de priorité des objets de même nature. Par exemple, deux montagnes peuvent exister dans la même permutation. Quelle montagne transposons-nous en premier ?

Nous avons choisi deux critères arbitraires dans notre implémentation. Le critère prioritaire est de traiter d'abord l'objet qui commence le plus à gauche. On choisit le plus long des deux objets s'ils commencent à la même position.

3.4.2 Modifications apportées aux algorithmes

Quelqu'un concevant un algorithme de tri par transpositions se basant sur les codes cherche à optimiser quatre qualités de cet algorithme. Il est souhaitable que l'algorithme donne des distances de transposition approximatives aussi petites que possible. La rapidité de l'exécution de l'algorithme (qui n'est pas déterminée que par la complexité théorique) est une autre qualité recherchée. Troisièmement, on souhaite concevoir un algorithme élégant et facile à coder. Finalement, il est avantageux d'avoir un algorithme qui s'analyse facilement. Cela permet de prouver

une complexité théorique.

Ces quatre qualités peuvent être contradictoires. Un détail de l'algorithme peut améliorer un aspect tout en nuisant à un autre aspect.

Nous pouvons considérer les algorithmes de la section 3.3 comme des algorithmes bruts pouvant être raffinés. Nous tenons à mentionner les modifications que nous avons faites aux implémentations de *TrieMontagnePlateau* et de *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau* pour donner des exemples de petites modifications aux algorithmes de base qui peuvent avoir une influence sur les quatre qualités mentionnées ci-haut. La description de ces modifications assure aussi la reproductibilité de nos résultats.

Certaines de ces modifications ont cependant le désavantage de faire en sorte que les théorèmes au sujet de la complexité théorique de certains algorithmes ne s'appliquent plus. Nous considérons cependant qu'ils deviennent des conjectures puisque les modifications sont mineures.

Par exemple, nous avons codé les algorithmes *TrieMontagnePlateau* et *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau* sans la méthode *Branch and Bound.* Les théorèmes 3.28, 3.29 et 3.31 ne sont donc plus prouvés. Ils peuvent cependant être considérés comme des conjectures puisqu'il demeure très raisonnable de penser que le nombre de fois que *TraiteUnObjet* est répété soit dans $O(n)$ (voir la preuve du théorème 3.28).

L'annexe II présente les autres modifications que nous avons apportées aux algorithmes dans notre code.

CHAPITRE 4

RÉSULTATS EXPÉRIMENTAUX

Nous avons implémenté et testé certains de nos algorithmes. Nous présentons les résultats obtenus et nous en discutons dans ce chapitre.

Nous présentons les algorithmes que nous avons choisis d'implémenter au début de la section 4.1. Nous expliquons aussi où nous sommes allés chercher, dans la littérature, les résultats des algorithmes que nous n'avons pas implémentés. La fin de la section 4.1 présente le protocole expérimental utilisé.

La section 4.2 contient les résultats expérimentaux et la discussion. Nous commençons par tester les permutations de longueur inférieure ou égale à 9. Nous connaissons la distance de transposition exacte de ces permutations. Nous discutons des résultats obtenus. Nous terminons la section 4.2 par les permutations plus longues dont nous ne connaissons pas la distance de transposition exacte.

4.1 Expérimentation

4.1.1 Implémentation de nos algorithmes

Nous avons implémenté certains de nos algorithmes dans le but de les tester et de les comparer à d'autres algorithmes existants. Rappelons que nos algorithmes calculent une séquence de transpositions qui trie une permutation. La longueur de cette séquence est la distance de transposition approximative. Nous avons implémenté *TriePlateau* (voir Figure 3.4), *TriePlateauGauche* (*TriePlateau* qui n'utilise que le code gauche), *TrieMontagnePlateau* et *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau*. Ils seront identifiés par *TP*, *TPG*, *TMP* et $TMAF_AF_SDMP$ respectivement dans le reste de ce mémoire. Les deux derniers algorithmes sont de la forme de *TrieObjets* décrit à la section 3.3.3. Le lecteur peut trouver plus d'information sur l'implémentation de nos algorithmes à la section 3.4.

4.1.2 Implémentation de *BP2*

Nous avons aussi implémenté l'algorithme *BP2* de Bafna et Pevzner [BP98] (voir la section 2.2.2). *BP2* teste beaucoup moins de cas que *TSort* et *TransSort* qui sont les deux autres algorithmes des mêmes auteurs (voir la section 2.2). Nous pouvons donc supposer que *BP2* est beaucoup plus rapide qu'eux. Nous avons comparé les résultats des temps d'exécution entre nos algorithmes *TP*, *TPG*, *TMP* et l'algorithme *BP2*. Puisque nos algorithmes sont plus rapides que *BP2*, nous pouvons supposer qu'ils sont beaucoup plus rapides que *TSort* et *TransSort* sans avoir à les implémenter.

Précisons que notre implémentation ne fait pas que calculer la borne supérieure de *BP2* (corollaire 2.6). Notre implémentation calcule plutôt une réelle séquence de transpositions. Le théorème 4.1 présente la complexité de notre implémentation. Le lecteur peut trouver la preuve de ce théorème à l'annexe III.

Théorème 4.1. *La complexité en temps de notre implémentation de* BP2 *est dans* $O(n^3)$. ■

4.1.3 Origine des résultats des autres algorithmes

Nous comparons les résultats obtenus avec nos algorithmes et *BP2* avec les résultats obtenus avec d'autres algorithmes. Nous n'avons pas implémenté ces autres algorithmes. Nous avons plutôt recueilli ces résultats dans la littérature.

Walter, Sobrinho et collaborateurs [WSO+05] présentent des résultats obtenus avec six algorithmes connus. Les auteurs vont chercher ces résultats dans la littérature pour les quatre algorithmes suivants. Le premier est l'algorithme de Walter, Dias et Meidanis [WDM00] (voir la section 2.3.3) implémenté par ces derniers (voir la section 2.4). Nous le désignerons par *WDM* dans le reste de ce mémoire. Le deuxième algorithme est celui de Christie [Chr98] (voir la section 2.3.3) implémenté avec des heuristiques par Walter, Curado et Oliveira [WCO03] (voir la section 2.4). Nous le désignerons désormais par *Cwh*. Le troisième algorithme est celui de Hartman [Har03] (voir la section 2.3.1) implémenté par Honda [Hon04] (voir la

section 2.4). Nous l'appellerons *Hartman* à partir de maintenant. Le quatrième algorithme est *TransSort* [BP98] (voir la section 2.2.4) implémenté par Walter et Oliveira [Oli01, WdO02a, WdO02b] (voir la section 2.4). Nous nommerons cette implémentation *BP-WO*.

De plus, Walter, Sobrinho et collaborateurs présentent les résultats qu'ils ont obtenus en implémentant eux-mêmes *TransSort* avec des heuristiques et sans heuristique (voir la section 2.4). Les abréviations utilisées pour les implémentations avec des heuristiques et sans heuristique sont *BPwh-WSO*$^+$ et *BP-WSO*$^+$ respectivement.

Les résultats au sujet des algorithmes *TSRunRand*, *TSRunShort*, *TSRunLong*, *TSRunBest*, *TSLIS*, *TSRunBnB* et *TSBnB* proviennent de Guyer, Heath et Vergara [GHV95] (voir les sections 2.3.2 et 2.4).

4.1.4 Protocole

Nous avons calculé la distance de transposition exacte pour toutes les permutations de longueur inférieure ou égale à 9. L'expression "**les petites permutations**" désignera ces permutations dans la suite de ce mémoire. Notons que le nombre de permutations distinctes de longueur n est $n!$.

Nous avons stocké les résultats obtenus dans des fichiers. Nous avons testé les algorithmes que nous avons implémentés sur ces fichiers. Les résultats concernant les petites permutations au sujet des algorithmes que nous n'avons pas implémentés nous-mêmes ont été obtenus de la même manière par leurs auteurs respectifs.

Nous avons généré pseudo-aléatoirement des permutations de longueur supérieure ou égale à 10. Nous appellerons ces permutations "**les grandes permutations**" désormais. Nous avons utilisé la méthode `shuffle(List)` de la classe `Collections` de `JAVA 1.5.0`.

> «This implementation traverses the list backwards, from the last element up to the second, repeatedly swapping a randomly selected element into the "current position". Elements are randomly selected from

> the portion of the list that runs from the first element to the current position, inclusive. [1] »(Copyright 2004 Sun Microsystems, Inc. Reproduit avec permission.) [SM]

Les éléments sont en fait sélectionnés pseudo-aléatoirement. Cela est dû à la source de hasard (*source of randomness*) que nous utilisons. Nous utilisons la source de hasard par défaut qui est imparfaite.

Nous avons ainsi créé un fichier de 1 000 permutations pour chacune des longueurs suivantes : 10, 11, 12, 16, 20, 30, 32, 40, 50, 64, 128, 256 et 512. Nous avons aussi créé un fichier de 100 permutations pour les longueurs 1000 et 5000.

Guyer, Heath et Vergara [GHV95] sont les seuls à présenter des résultats pour des grandes permutations. Ils ont testé leurs algorithmes sur des échantillons de taille 100 à l'exception d'un échantillon de taille 50 pour les permutations de 12 éléments.

Nos cinq algorithmes ont été implémentés par la même personne et testés sur la même machine. Ce n'est cependant pas le cas des algorithmes testés dans la littérature dont nous présentons aussi les résultats.

Nous avons codé les algorithmes en `JAVA`. Nous avons compilé les applications avec `javac 1.5.0_04`. Nous les avons testées sur un ordinateur `AMD Athlon` 64 bits 2200 MHz avec 3.9 Gb de RAM et `Fedora core`.

4.2 Résultats et discussion

Nous avons exécuté plusieurs tests pour comparer les performances, en qualité de l'approximation des distances et en temps, de notre algorithme avec plusieurs algorithmes de la littérature. Nous présentons et discutons ces résultats dans cette section.

[1] Cette implémentation traverse la liste à rebours, du dernier élément jusqu'au second, percolant répétitivement un élément sélectionné aléatoirement avec la "position courante". Les éléments sont sélectionnés aléatoirement dans la portion de la liste qui s'étend du premier élément à la position courante inclusivement. (Traduction libre.)

4.2.1 Petites permutations

Nous avons premièrement testé nos algorithmes sur les "petites permutations", c'est-à-dire les permutations de longueur 2 à 9. Ce sont les permutations pour lesquelles nous avons calculé la distance de transposition exacte.

4.2.1.1 Qualité de l'approximation des distances et temps

Le tableau 4.1 présente pour chaque n, $2 \leq n \leq 9$, le nombre de permutations pour lesquelles l'algorithme considéré ne calcule pas la bonne distance. Il présente aussi le temps en secondes nécessaire pour trier une permutation de taille 9 en utilisant les algorithmes que nous avons implémentés nous-mêmes.

Nous analysons et comparons maintenant la qualité de l'approximation de nos algorithmes et des algorithmes approximatifs provenant de la littérature. Rappelons que *TSBnB* de Guyer, Heath et Vergara [GHV95] est un algorithme exact. C'est pour cette raison qu'il ne fait aucune erreur pour $n = 6$ (Taille $= 6$) au tableau 4.1.

La colonne $n = 6$ du tableau 4.1 nous permet de classer les algorithmes selon la qualité de l'approximation en ordre décroissant. Les algorithmes *Cwh* ([Chr98] avec des heuristiques [WCO03]), *BP-WO* ([BP98] par [Oli01, WdO02a, WdO02b]), *BP-WSO*$^+$ et *BPwh-WSO*$^+$ ([BP98] par [WSO$^+$05] sans heuristique et avec des heuristiques respectivement) ne font pas d'erreurs. Les autres algorithmes en ordre décroissant, de celui qui fait le moins d'erreurs à celui qui en fait le plus, sont *Hartman* ([Har03] par [Hon04]), *WDM* ([WDM00]), $TMAF_AF_SDMP$, *TSLIS* ([GHV95]), *TSRunBnB* ([GHV95]), *TSRunBest* ([GHV95]), *BP2* ([BP98] par nous-mêmes), *TMP*, *TP*, *TSRunLong* ([GHV95]), *TSRunRand* ([GHV95]) et *TSRunShort* ([GHV95]).

Nous discutons maintenant des résultats obtenus par nos algorithmes. $TMAF_AF_SDMP$ approxime bien la distance de transposition relativement à *BP2*. Il est même meilleur que *BP2* jusqu'à $n = 8$. *TP* et *TMP* font de mauvaises approximations relativement à *BP2*. On remarque que *BPwh-WSO*$^+$ fait beaucoup moins d'erreurs que $TMAF_AF_SDMP$ ou que *BP2*. Cela est dû au fait que

Taille	2	3	4	5	6	7	8	9	Temps(s)
WDM	0	0	0	0	6	72	1167	14327	x
Cwh	0	0	0	0	0	0	40	1182	x
Hartman	0	0	0	0	2	108	1517	25425	x
BP-WO	0	0	0	0	0	1	135	4361	-
BP-WSO^+	0	0	0	0	0	0	131	4020	-
BPwh-WSO^+	0	0	0	0	0	0	34	1007	-
TSRunRand	x	x	x	x	300	x	x	x	x
TSRunShort	x	x	x	x	456	x	x	x	x
TSRunLong	x	x	x	x	217	x	x	x	x
TSRunBest	x	x	x	x	83	x	x	x	x
TSLIS	x	x	x	x	45	x	x	x	x
TSRunBnB	x	x	x	x	47	x	x	x	x
TSBnB	x	x	x	x	0	x	x	x	x
TPG	0	0	1	18	217	2326	24621	266232	$2,52E-5$
TP	0	0	0	6	108	1423	17577	211863	$2,92E-5$
TMP	0	0	0	6	103	1314	15941	190528	$2,97E-5$
TMA$F_A F_S$DMP	0	0	0	1	29	484	7416	102217	$4,14E-5$
BP2	0	0	1	7	86	792	9162	100332	$2,93E-5$

TAB. 4.1 – Nombre de permutations pour lesquelles la distance de transposition exacte n'est pas trouvée pour chaque algorithme sur les permutations de longueur 2 à 9 et le temps pour les permutations de longueur 9.
La première rangée *Taille* indique la taille des permutations. Les autres rangées sont *WDM* de Walter, Dias et Meidanis [WDM00], *Cwh* de Christie [Chr98] avec des heuristiques [WCO03], *Hartman* [Har03] par Honda [Hon04], *BP-WO* de Bafna et Pevzner [BP98] par Walter et Oliveira [Oli01, WdO02a, WdO02b], *BP-WSO*$^+$ et *BPwh-WSO*$^+$ de Bafna et Pevzner par Walter, Sobrinho et collaborateurs [WSO$^+$05] sans heuristique et avec des heuristiques respectivement, les algorithmes de *TSRunRand* à *TSBnB* de Guyer, Heath et Vergara [GHV95], *TPG* (*TriePlateauGauche*), *TP* (*TriePlateau*), *TMP* (*TrieMontagnePlateau*), $TMAF_AF_SDMP$ (*TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau*) et *BP2* [BP98]. La colonne *Temps* indique le temps moyen pris pour trier une permutation en secondes. Un × indique que le résultat est absent de la source documentaire utilisée. Un - indique que le résultat existe dans la source documentaire utilisée sous la forme d'un graphique mais qu'il n'y a pas de valeurs numériques précises.

BPwh-WSO$^+$ teste beaucoup plus de cas que ces deux derniers algorithmes.

Une limite à l'interprétation de ces résultats mérite d'être mentionnée. Ces algorithmes ont été programmés par d'autres programmeurs que nous. Or, il est possible que l'implémentation d'au moins certains de ces algorithmes modifie les résultats obtenus.

Nous discutons maintenant de la vitesse d'exécution de notre algorithme $TMAF_AF_SDMP$ relativement aux autres algorithmes implémentés par nous. Nous observons que cet algorithme est plus lent que les algorithmes *TP*, *TMP* et *BP2* pour les permutations de taille 9. Cela peut s'expliquer par la grande complexité en temps de $TMAF_AF_SDMP$ qui est dans $O(n^5)$ (théorème 3.31). Mentionnons pour comparer que la complexité en temps de *TP* est dans $O(n^2)$ (théorème 3.3). Les complexités en temps de *TMP* et de notre implémentation de *BP2* sont, quant à elles, dans $O(n^3)$ (théorèmes 3.29 et 4.1 respectivement). Nous ne présentons pas de résultats avec $TMAF_AF_SDMP$ sur les grandes permutations parce que nous n'avons pas terminé ces tests par manque de temps.

4.2.1.2 Facteur d'approximation réel maximal

Nous devons introduire le concept de facteur d'approximation réel maximal (voir définition 4.1). Intuitivement, le facteur d'approximation réel maximal nous apprend jusqu'à quel point la distance de transposition approximative calculée par l'algorithme pour la "pire" permutation de longueur n s'éloigne de la vraie distance. Nous avons calculé les facteurs d'approximation réels maximaux de chaque algorithme que nous avons implémenté. Le tableau 4.2 présente le facteur d'approximation réel maximal de chaque S_n pour $2 \leq n \leq 9$. Rappelons que les seules permutations pour lesquelles nous connaissons la distance de transposition exacte sont les petites permutations. Nous ne connaissons donc le facteur d'approximation réel maximal donné par les différents algorithmes que pour ces permutations.

Définition 4.1. *Le facteur d'approximation réel maximal de l'algorithme* algo *sur*

le groupe symétrique S_n est

$$\max_{\pi \in S_n} \frac{algo(\pi)}{d(\pi)}$$

où $algo(\pi)$ est la distance de transposition approximative calculée avec algo *sur π.*

Taille	2	3	4	5	6	7	8	9
TPG	1	1	1,5	2	2	2	2,333	2,333
TP	1	1	1	1,5	1,666	2	2	2
TMP	1	1	1	1,5	1,5	1,666	2	2
$TMAF_AF_SDMP$	1	1	1	1,5	1,5	1,666	2	2
BP2	1	1	1,5	1,5	1,5	1,666	1,75	1,75

TAB. 4.2 – Facteur d'approximation réel maximal de chaque ensemble de permutations de même longueur inférieure ou égale à 9 avec nos algorithmes et $BP2$. La première rangée *Taille* indique la taille des permutations. Les autres rangées sont *TPG* (*TriePlateauGauche*), *TP* (*TriePlateau*), *TMP* (*TrieMontagnePlateau*), $TMAF_AF_SDMP$ (*TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau*) et *BP2* [BP98].

Le facteur d'approximation réel maximal de $TMAF_AF_SDMP$ est supérieur au facteur d'approximation réel maximal de *BP2* pour les permutations de taille 8 et 9. Pour $n = 8$, cela signifie que même si $TMAF_AF_SDMP$ fait moins d'erreurs que *BP2*, sa pire approximation de la distance de transposition est plus éloignée de la distance de transposition exacte que ne l'est la pire approximation de la distance de transposition faite par *BP2*.

L'algorithme *TP* a une garantie de performance de 3 (théorème 3.26). Les algorithmes *TMP* et $TMAF_AF_SDMP$ ont aussi une garantie de performance de 3 dans leur version *Branch and Bound* (voir la section 3.3.3.1). Puisque nous n'avons pas implémenté la méthode *Branch and Bound*, cette garantie de performance devient une conjecture pour les algorithmes *TMP* et $TMAF_AF_SDMP$ du tableau 4.2.

Ce tableau nous apprend que le facteur d'approximation réel maximal ne dépasse jamais 2 quand on utilise les algorithmes *TP*, *TMP* et $TMAF_AF_SDMP$ sur les petites permutations. Le facteur d'approximation réel maximal de ces algorithmes est donc significativement inférieur à leur garantie de performance, prouvée ou

conjecturée, pour les petites permutations. Nous ne pouvons pas en dire autant des grandes permutations puisque nous n'avons pas fait les tests nécessaires.

4.2.2 Grandes permutations

4.2.2.1 Qualité de l'approximation

Nous avons testé nos algorithmes sur des permutations générées pseudo-aléatoirement (voir la section 4.1.4) de longueur supérieure ou égale à 10. Nous ne pouvons pas savoir si la distance de transposition approximative obtenue est la distance de transposition exacte puisque nous ne connaissons pas les vraies distances de transposition pour ces permutations. Au lieu de compter le nombre de permutations pour lesquelles la distance de transposition exacte n'est pas trouvée par l'algorithme testé, nous avons plutôt calculé la distance de transposition moyenne obtenue avec chaque algorithme (voir Tableaux 4.3 et 4.4).

Nous analysons et comparons maintenant la qualité de l'approximation de nos algorithmes et de ceux de Guyer, Heath et Vergara [GHV95] en utilisant les résultats des tableaux 4.3 et 4.4. Rappelons que *TSBnB* de [GHV95] est un algorithme exact. C'est pour cette raison qu'il trouve la distance de transposition moyenne la plus petite pour $n = 12$.

On remarque que les approximations de *TP* et *TMP* sont de moins en moins bonnes à mesure que n augmente. Les distances de transposition moyennes approximées par *TP* ne dépassent pas de plus de 50% celles de *BP2* pour n de 10 à 50. Ce n'est plus vrai pour $n \geq 64$. Pour $n = 5\ 000$, la distance de transposition moyenne approximée par *TP* dépasse même de plus de 75% celle de *BP2*. *TP* et *TMP* font de mauvaises approximations relativement à *BP2* encore plus pour de grandes permutations que pour de petites permutations.

La qualité des approximations de *TP* et *TMP* est comparable à la qualité des approximations des algorithmes les plus rapides de [GHV95] (*TSRunRand*, *TSRunShort*, *TSRunLong* et *TSRunBest*). Il faut cependant être prudent au sujet de cette interprétation puisque [GHV95] ont fait leurs tests sur d'autres ensembles

Taille	10	11	12	16	20	30	32	40	50	64	128
TSRunRand	x	x	7	x	13,46	21,88	x	30,59	39,46	x	x
TSRunShort	x	x	6,60	x	12,68	20,90	x	29,74	38,11	x	x
TSRunLong	x	x	6,22	x	12,08	20,49	x	28,84	37,93	x	x
TSRunBest	x	x	5,00	x	9,97	17,11	x	24,72	32,68	x	x
TSLIS	x	x	4,94	x	9,52	15,96	x	22,22	28,77	x	x
TSRunBnB	x	x	4,90	x	x	x	x	x	x	x	x
TSBnB	x	x	4,64	x	x	x	x	x	x	x	x
TPG	6,008	6,908	7,669	11,171	14,715	23,968	25,913	33,311	42,937	56,448	119,253
TP	5,587	6,426	7,189	10,625	14,037	23,191	25,02	32,475	41,925	55,451	117,963
TMP	5,459	6,222	6,925	10,11	13,284	21,816	23,601	30,615	39,588	52,502	113,29
BP2	5,063	5,595	6,176	8,501	10,782	16,561	17,757	22,324	28,037	36,092	72,571

TAB. 4.3 – Distances de transposition moyennes pour des permutations de longueur 10 à 128 générées pseudo-aléatoirement selon l'algorithme utilisé.
La première rangée *Taille* indique la taille des permutations. Les autres rangées sont les algorithmes approximatifs *TSRunRand*, *TSRunShort*, *TSRunLong*, *TSRunBest*, *TSLIS* et *TSRunBnB* de Guyer, Heath et Vergara [GHV95], l'algorithme exact *TSBnB* de [GHV95], *TPG* (*TriePlateauGauche*), *TP* (*TriePlateau*), *TMP* (*TrieMontagnePlateau*) et *BP2* [BP98]. Un x indique que le résultat est absent de [GHV95].

Taille	256	512	1 000	5 000
TSRunRand	x	x	963,4	4 929,5
TSRunShort	x	x	954,1	4 908,9
TSRunLong	x	x	955,2	4 913,9
TSRunBest	x	x	927,0	4 847,8
TSLIS	x	x	trop lent	trop lent
TSRunBnB	x	x	trop lent	trop lent
TSBnB	x	x	trop lent	trop lent
TPG	245,83	500,482	987,29	4 983,64
TP	244,382	498,912	985,63	4 982,05
TMP	237,233	488,572	971,72	4 958,75
BP2	145,181	290,562	567,53	2 840,89

TAB. 4.4 – Distances de transposition moyennes pour des permutations de longueur 256 à 5 000 générées pseudo-aléatoirement selon l'algorithme utilisé.
Voir la légende du tableau 4.3.

de permutations que nous.

De plus, comme nous l'avons mentionné à la section 4.2.1.1, le fait que leurs algorithmes aient été programmés par d'autres programmeurs que nous peut être une limite à l'interprétation des résultats.

4.2.2.2 Temps

Nous avons noté le temps d'exécution des algorithmes que nous avons implémentés. Le tableau 4.5 indique le temps moyen pris pour trier une permutation pour différentes longueurs allant de 10 à 1 000.

Taille	TPG	TP	TMP	BP2
10	3,870E-4	4,720E-4	4,67E-4	5,03E-4
11	4,17E-4	4,44E-4	5,11E-4	4,55E-4
12	4,4E-4	4,43E-4	4,59E-4	5,49E-4
16	4,71E-4	4,56E-4	4,96E-4	5,79E-4
20	5,07E-4	4,97E-4	5,24E-4	6,35E-4
30	5,200E-4	5,1E-4	5,53E-4	7,740E-4
32	5,19E-4	5,32E-4	5,29E-4	7,85E-4
40	4,88E-4	5,59E-4	5,82E-4	9,05E-4
50	5,33E-4	5,67E-4	7,14E-4	0,001242
64	6,25E-4	6,56E-4	8,68E-4	0,001667
128	8,05E-4	0,001058	0,001808	0,004838
256	0,001667	0,002547	0,005292	0,02288
512	0,004386	0,007611	0,02337	0,1160
1 000	0,01752	0,02874	0,1152	1,299

TAB. 4.5 – Temps moyen pris par l'exécution de l'algorithme pour trier une permutation générée pseudo-aléatoirement de longueur 10 à 1 000 selon l'algorithme utilisé.
La première colonne *Taille* indique la taille des permutations. Les autres colonnes sont *TPG* (*TriePlateauGauche*), *TP* (*TriePlateau*), *TMP* (*TrieMontagnePlateau*) et *BP2* [BP98]. Le temps est indiqué en secondes.

Le tableau 4.6, quant à lui, présente les temps moyens requis pour trier une permutation de longueur 5 000 en utilisant les algorithmes que nous avons implémentés

et les temps indiqués par Guyer, Heath et Vergara dans leur article [GHV95] pour leurs différents algorithmes. (Ce sont les seuls temps présentés dans leur article.)

Taille	5 000
TSRunRand	20,8
TSRunShort	14,6
TSRunLong	17,5
TSRunBest	35,6
TSLIS	trop lent
TSRunBnB	trop lent
TSBnB	trop lent
TPG	0,2591
TP	0.5824
TMP	4,448
BP2	144,8

TAB. 4.6 – Temps moyen pris par l'exécution de l'algorithme pour trier une permutation générée pseudo-aléatoirement de longueur 5 000 selon l'algorithme utilisé. La première rangée *Taille* indique la taille des permutations. Les autres rangées sont les algorithmes approximatifs *TSRunRand*, *TSRunShort*, *TSRunLong*, *TSRunBest*, *TSLIS* et *TSRunBnB* de Guyer, Heath et Vergara [GHV95], l'algorithme exact *TSBnB* de [GHV95], *TPG* (*TriePlateauGauche*), *TP* (*TriePlateau*), *TMP* (*TrieMontagnePlateau*) et *BP2* [BP98]. Le temps est indiqué en secondes. L'unité de temps n'est pas donnée dans [GHV95]. Nous avons supposé qu'il s'agissait de secondes.

Nous discutons le temps d'exécution de nos algorithmes, de *BP2* et des algorithmes de [GHV95] sur les grandes permutations en utilisant les tableaux 4.5 et 4.6.

Nous pouvons ordonner en ordre décroissant, du plus rapide au moins rapide, les algorithmes du tableau 4.6. Nous obtenons *TP*, *TMP*, les algorithmes les plus rapides de [GHV95] (*TSRunShort*, *TSRunLong*, *TSRunRand* et *TSRunBest*) et *BP2*.

La colonne $\frac{\text{temps de } TMP}{\text{temps de } TP}$ du tableau 4.7 représente combien de fois *TMP* est plus lent que *TP* pour différentes valeurs de n.

Ce ratio est grand (plus que 2) à partir de $n = 256$. Les observations pour

n	$\frac{\text{temps de } TMP}{\text{temps de } TP}$
$10 \leq n \leq 128$	< 2
256	2,1
512	3,1
1 000	4,0
5 000	7,6

TAB. 4.7 – Ratio $\frac{\text{temps de } TMP}{\text{temps de } TP}$ pour de grandes permutations de différentes longueurs générées pseudo-aléatoirement.
La colonne n représente la taille des permutations.

$n \geq 256$ sont mieux expliquées par une augmentation lente et logarithmique que par une augmentation rapide et linéaire. Par exemple, une constante (environ 1) semble être ajoutée au ratio à chaque fois que n double pour n entre 256 et 1 000. Le ratio diverge un peu de ce modèle pour $n = 5\ 000$. Puisque les complexités en temps de *TMP* et *TP* sont dans $O(n^3)$ et dans $O(n^2)$ respectivement, (théorèmes 3.29 et 3.3 respectivement), nous nous serions plutôt attendus à ce que le ratio augmente linéairement. Soit $O(TMP)$ l'ordre de complexité en temps de *TMP*. Une hypothèse pour expliquer ces observations est que $O(TMP)$ soit tel que $O(n^2 log(n)) \subseteq O(TMP) \subseteq O(n^3)$.

La colonne $\frac{\text{temps de } BP2}{\text{temps de } TP}$ du tableau 4.8 représente combien de fois *BP2* est plus lent que *TP* pour différentes valeurs de n.

Nous observons que ce ratio est grand (plus que 2) à partir de $n = 50$. Les observations pour $n \geq 50$ sont mieux expliquées par une augmentation rapide et linéaire que par une augmentation lente et logarithmique. Par exemple, une constante (environ 2) semble multiplier le ratio à chaque fois que n double pour n entre 128 et 512. Le ratio augmente encore plus rapidement que ce qui est prévu par ce modèle pour $n \geq 1\ 000$. Une augmentation linéaire du ratio est ce qui est attendu puisque les complexités de *BP2* et de *TP* sont dans $O(n^3)$ et dans $O(n^2)$ respectivement (théorèmes 4.1 et 3.3 respectivement).

Nous pouvons conclure que *TMP* est plus rapide que *BP2* et que cette différence

n	$\frac{\text{temps de } BP2}{\text{temps de } TP}$
$10 \leq n \leq 40$	< 2
$50 \leq n \leq 128$	2,1 à 4,6
256	8,9
512	15,2
1 000	45,2
5 000	248,6

TAB. 4.8 – Ratio $\frac{\text{temps de } BP2}{\text{temps de } TP}$ pour de grandes permutations de différentes longueurs générées pseudo-aléatoirement.
La colonne n représente la taille des permutations.

s'accélère à mesure que n augmente bien que les deux algorithmes aient la même complexité théorique.

CHAPITRE 5

CONCLUSION

Nous avons présenté de nouveaux algorithmes rapides pour résoudre approximativement le problème du tri par transpositions dans ce mémoire. Après avoir présenté le problème dans l'introduction, nous avons discuté de son utilité en génomique comparée. Nous avons aussi expliqué les limites de l'utilisation de la distance de transposition en génomique comparée.

Dans le chapitre 3, nous avons présenté le concept des codes (gauche et droit) d'une permutation. Nous avons ensuite présenté différents algorithmes se basant sur ce concept. Par exemple, nous avons présenté notre algorithme simple *TriePlateau* pour lequel nous avons prouvé une garantie de performance de 3. La complexité en temps de *TriePlateau* est dans $O(n^2)$. Nous avons aussi présenté d'autres algorithmes plus complexes qui sont des heuristiques pour améliorer *TriePlateau*. Leur complexité en temps peut aller jusque dans $O(n^5)$.

Nous avons testé expérimentalement *TriePlateau* et un de nos algorithmes moyennement complexe, *TrieMontagnePlateau*, sur des permutations de longueur inférieure ou égale à 5 000. Nous avons testé notre algorithme le plus complexe, *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau*, sur les permutations de longueur inférieure ou égale à 9. Les tests, leurs résultats et la discussion sont présentés au chapitre 4.

Nos algorithmes se basant sur le concept des codes d'une permutation sont beaucoup plus simples, élégants et faciles à comprendre et à implémenter que ceux se basant sur un graphe de cycles [BP98, Chr98, Har03, EH05] ou sur un diagramme de points de cassure [MWD97].

Nos deux algorithmes les plus simples, *TriePlateau* et *TrieMontagnePlateau*, sont très rapides. Ils sont beaucoup plus rapides que n'importe quel autre algorithme existant aujourd'hui pour les grandes permutations (les permutations de longueur supérieure ou égale à 10). Cependant, cette rapidité se fait au détriment

de la qualité de l'approximation. En effet, nous n'avons réussi à prouver qu'une garantie de performance de 3. Expérimentalement, ces deux algorithmes approximent plutôt mal la distance de transposition d'une permutation. Plus les permutations sont longues, plus les autres algorithmes déjà existants approximent des distances de transposition inférieures à celles approximées par *TriePlateau* et *TrieMontagnePlateau*.

Notre algorithme le plus complexe, *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau*, fait de bonnes approximations pour les petites permutations (les permutations de longueur inférieure ou égale à 9). Cependant, nous ne connaissons pas la qualité de son approximation et sa vitesse d'exécution pour les grandes permutations puisque nous ne l'avons pas testé sur ces permutations.

Jean, notre personnage fictif de la section 1.1, a maintenant de nouveaux algorithmes pour trier ses cubes par transpositions. Il a même de nouveaux aspects du problème avec lesquels il peut s'amuser.

5.1 Travail futur

Il reste des pistes prometteuses de recherche à explorer au sujet du problème du tri par transpositions. Nous en énumérons quelques unes dans cette section. Nous commençons par présenter des améliorations possibles à nos algorithmes et à nos implémentations. Nous mentionnons un test supplémentaire intéressant à faire. Nous terminons en rappelant les deux principaux problèmes généraux au sujet du tri par transpositions qui sont toujours ouverts.

5.1.1 Travail futur au sujet de nos algorithmes

Premièrement, nous pourrions essayer d'améliorer le théorème 3.26 en prouvant une garantie de performance inférieure à 3 pour *TriePlateau*.

Nous avons prouvé que *TrieObjetsBnB* a une garantie de performance de 3 (voir la section 3.3.3.1). Nous n'avons pas réussi à le prouver pour *TrieObjets* dans le

cas où cet algorithme considère les montagnes, les fossés asymétriques ou les fossés symétriques à cause des théorèmes 3.4, 3.6 et 3.22. Un projet futur pourrait être de prouver une garantie de performance à *TrieObjets* qui soit inférieure ou égale à celle de *TriePlateau* (c'est-à-dire 3). Nous présentons une idée qui pourrait permettre d'y arriver. Il s'agirait de traiter les objets dans le code gauche de la permutation π de gauche à droite par ordre d'apparition peu importe leur nature (plateau, montagne, fossé asymétrique, ascension, descente, montée, fossé symétrique). Si nous avions les codes gauche et droit de π, nous devrions alors traiter les objets des extrémités vers le centre. Soit ρ la transposition qui traite l'objet. Dans le cas des montagnes, nous savons déjà que $\Delta p(\rho) \leq -2$ lorsque la sous-permutation à gauche de celle-ci est triée (voir un exemple à la figure 3.7). Peut-être pourrions-nous prouver que $\Delta p(\rho) \leq -1$ lors du traitement d'un fossé asymétrique lorsque la sous-permutation à gauche de celui-ci est triée. Soit ρ_1 et ρ_2 les deux transpositions pour traiter un fossé symétrique. Il resterait alors à prouver que la moyenne de $\Delta p(\rho_1)$ et $\Delta p(\rho_2)$ est inférieure ou égale à -1 lorsque la sous-permutation à gauche du fossé symétrique est triée. Dans ce cas, nous pourrions prouver que *TrieObjets* ne prend jamais plus de transpositions que *TriePlateau*. Une garantie de performance de *TrieObjets* inférieure ou égale à celle de *TriePlateau* serait alors prouvée.

Finalement, nous pourrions ajouter de nouveaux objets à l'algorithme *TrieObjets*.

5.1.2 Travail futur au sujet de nos implémentations

Premièrement, nous pourrions coder *TrieObjets* pour qu'il utilise les codes gauche et droit de la permutation au lieu de n'utiliser que le code gauche. Cela permettrait d'obtenir de meilleurs résultats. Une autre manière d'obtenir de meilleurs résultats serait de tester rigoureusement les différents ordres de priorité du traitement des objets de nature différente. Finalement, nous pourrions recalculer les codes des éléments correspondants aux parties B et C de la permutation π après la transposition au lieu de recalculer tout le code de π (voir Figure 3.2). Cela améliorerait la vitesse d'exécution de nos algorithmes.

5.1.3 Expérimentation future

Un test intéressant à faire serait de tester *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau* sur de grandes permutations (des permutations de longueur supérieure ou égale à 10).

5.1.4 Problèmes ouverts généraux au sujet du tri par transpositions

La classe de complexité du problème du tri par transpositions n'est pas encore déterminée. Ce problème est-il NP-difficile ? Le diamètre de transposition en fonction de n (voir la section 1.4) n'est pas déterminé non plus.

BIBLIOGRAPHIE

[BGH07] M. Benoît-Gagné and S. Hamel. A new and faster method of sorting by transpositions. In B. Ma and K. Zhang, editors, *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 4580 of *Lecture Notes in Computer Science*, pages 131–141, 2007. À paraître.

[BP98] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2) :224–240, 1998. Une version préliminaire a paru dans les *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 614-623, 1995.

[Chr98] D. A. Christie. *Genome rearrangements problems.* PhD thesis, Glasgow University, Scotland, 1998.

[DEKM98] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis - Probabilistic models of proteins and nucleic acids.* Cambridge University Press, 1998. Chapitre 7 : Building phylogenetic trees.

[EEK+01] H. Eriksson, K. Eriksson, J. Karlander, L. Svensson, and J. Wästlund. Sorting a bridge hand. *Discrete Mathematics*, 241 :289–300, 2001.

[EH05] I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. In R. Casadio and G. Myers, editors, *Proceedings of the 5th Workshop on Algorithms in Bioinformatics (WABI)*, volume 3692 of *Lecture Notes in Computer Science (Lecture Notes in Bioinformatics)*, pages 204–215, 2005.

[GHV95] S. A. Guyer, L. S. Heath, and J. P. C. Vergara. Subsequence and run heuristics for sorting by transpositions. In *4th DIMACS International Algorithm Implementation Challenge*, 1995.

[Har03] T. Hartman. A simpler 1.5-approximation algorithm for sorting by transpositions. In R. A. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial*

Pattern Matching (CPM), volume 2676 of *Lecture Notes in Computer Science*, pages 156–169, 2003.

[Hon04] M. I. Honda. Implementation of the algorithm of Hartman for the problem of sorting by transpositions. Master's thesis, Department of Computer Science, University of Brasilia, 2004.

[HP94] S. B. Hoot and J. D. Palmer. Structural rearrangements, including parallel inversions, within the chloroplast genome of Anemone and related genera. *Journal of Molecular Evolution*, 38 :274–281, 1994.

[Lab05] A. Labarre. A new tight upper bound on the transposition distance. In R. Casadio and G. Myers, editors, *Proceedings of the 5th Workshop on Algorithms in Bioinformatics (WABI)*, volume 3692 of *Lecture Notes in Computer Science (Lecture Notes in Bioinformatics)*, pages 216–227, 2005.

[Mou01] D. W. Mount. *Bioinformatics - Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, 2001. Glossaire.

[MWD97] J. Meidanis, M. E. M. T. Walter, and Z. Dias. Transposition distance between a permutation and its reverse. In R. Baeza-Yates, editor, *Anais do IV South American Workshop on String Processing (WSP)*, pages 70–79, 1997.

[Oli01] E. T. G. Oliveira. Implementations of algorithms for the problem of sorting by transpositions. Master's thesis, Department of Computer Science, University of Brasilia, 2001.

[PH86] J. D. Palmer and L. A. Herbon. Tricircular mitochondrial genomes of Brassica and Raphanus : reversal of repeat configurations by inversion. *Nucleic Acid Research*, 14 :9755–9764, 1986.

[SM] Sun Microsystems, Inc. API Specifications : J2SE 1.5.0, 2004. <http ://java.sun.com/j2se/1.5.0/docs/api> Consulté le 14 mai 2007.

[SM97] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997. Chapitre 6 : Phylogenetic Trees.

[WCO03] M. E. M. T. Walter, L. R. A. F. Curado, and A. G. Oliveira. Working on the problem of sorting by transpositions on genome rearrangements. In R. A. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 372–383, 2003.

[WDM00] M. E. M. T. Walter, Z. Dias, and J. Meidanis. A new approach for approximating the transposition distance. In *Proceedings of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 199–208, 2000.

[WdO02a] M. E. M. T. Walter and E. T. G. de Oliveira. Extending the theory of Bafna and Pevzner for the problem of sorting by transpositions. In *Seleta do XXIV Congresso Nacional de Matemática Aplicada e Computacional (CNMAC)*, volume 3(1) of *Tendências em Matemática Aplicada e Computacional-Seleta (TEMA-Seleta)*, pages 213–222, 2002.

[WdO02b] M. E. M. T. Walter and E. T. G. de Oliveira. The problem of sorting by transpositions and the algorithm of Bafna and Pevzner. In *XXII Seminário Integrado de Software e Hardware (SEMISH)*, 2002.

[WSO+05] M. E. M. T. Walter, M. C. Sobrinho, E. T. G. Oliveira, L. S. Soares, A. G. Oliveira, T. E. S. Martins, and T. M. Fonseca. Improving the algorithm of Bafna and Pevzner for the problem of sorting by transpositions : a practical approach. *Journal of Discrete Algorithms*, 3 :342–361, 2005.

Annexe I

Pseudo-code de *BP2*

La fonction pos($elem$) retourne la position de l'élément $elem$ dans π.

Algorithme BP2 (entrée : une permutation π de [n])
Tant que $\pi \neq$ Id
    Construire $G(\pi)$.
    Si $G(\pi)$ a un cycle orienté $c(i_1, i_2, \ldots, i_k)$ de k arêtes noires
        $t \leftarrow 3 \leq t \leq k$ et $i_t > i_{t-1}$
        Faire $\rho(i_{t-1}, i_t, i_1)$ (2-coup).
    Sinon
        Soit le cycle non-orienté $c(i_1, i_2, \ldots, i_k)$ de k arêtes noires
        $r \leftarrow pos(max_{elem \in \pi,\, i_2 \leq pos(elem) \leq i_1 - 1} elem)$
        $s \leftarrow pos(\pi_r + 1)$
        Si $s > i_1$
            Faire $\rho(r+1, s, i_2)$ (0-coup).
            // *On obtient un cycle orienté.*
        Sinon
            Faire $\rho(r+1, s, i_1)$ (0-coup).
            // *On obtient un cycle orienté.*

Annexe II

Modifications apportées aux algorithmes dans notre code

1. Nous avons codé les algorithmes *TrieMontagnePlateau* et *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau* sans la méthode *Branch and Bound.* Voir la section 3.4.2.
2. Le traitement des montagnes (voir Figure 3.5) est modifié. L'application réellement codée trie complètement la sous-permutation à gauche de la montagne avant de chercher la position d'insertion de cette dernière.
3. La définition et le traitement des ascensions sont aussi un peu modifiés (voir définition 3.7 et Figure 3.12 respectivement). La nouvelle définition d'ascension ajoute la condition que le premier terrain plat de cet objet soit de longueur 1. De plus, l'application réellement codée ne transpose pas directement le dernier terrain plat. On prévoit que celui-ci soit transposé lors de l'exécution d'un algorithme suivant l'exécution de *TraiteAscension.*
4. La définition d'une descente (voir définition 3.8) est transformée de la manière suivante. Elle devient $v(P_i) \geq v(P_{i+1}), v(P_{i+2}), \ldots, v(P_{i+m-1})$. La définition de bonne descente (voir définition 3.9) est aussi modifiée. Rappelons que k est la position devant laquelle une bonne descente peut être transposée pour enlever au moins deux plateaux. La nouvelle définition de bonne descente ajoute la condition que k doit être à gauche de la descente.
5. Contrairement aux descentes, la définition des montées (voir définition 3.11) reste inchangée. Cependant, la définition de bonne montée (voir la section 3.2.6.1) est modifiée de la même manière que celle de bonne descente. La position k doit encore être à gauche de la montée.
6. La définition de score d'insérabilité (voir définition 3.10) est modifiée seulement lorsqu'elle s'applique à une montée. On pose alors que $\pi_0 = -\infty$ au lieu de 0.

7. Tout comme le traitement des ascensions, le traitement des fossés symétriques (voir Figure 3.18) est modifié de la manière suivante. L'application réellement codée ne transpose pas directement la bonne descente créée. On prévoit que celle-ci soit transposée lors de l'exécution de *TraiteDescenteHeuristique* suivant la première transposition de *TraiteFosséSymétriqueHeuristique*.
8. Finalement, l'algorithme *TrieMontagneAscensionFosséAsymétriqueFosséSymétriqueDescenteMontéePlateau* que nous avons codé cherche en fait les fossés asymétriques et symétriques en même temps.

Ces modifications ont souvent pour but d'accélérer l'exécution de l'algorithme. Cela se fait en évitant d'exécuter certains tests dont le résultat sera assurément ou presqu'assurément négatif.

Annexe III

Preuve de la complexité en temps de notre implémentation de *BP2*

L'annexe I présente le pseudo-code de *BP2*. La première étape à faire dans la boucle *Tant que* est de construire $G(\pi)$ qui est le graphe de cycles de π. Nous présentons à la figure III.1 le pseudo-code de notre implémentation. Nous n'affirmons pas que celle-ci est optimale.

```
Algorithme ConstruireG (entrée : une permutation π de [n])
posElem ← n − 1
// Créer un nouveau cycle.
Tant que posElem existe
    Créer le cycle c.
    // Allonger le cycle.
    Tant que posElem n'est pas marquée
        Ajouter posElem à c.
        Marquer posElem.
        elem ← π[posElem − 1] + 1
        posElem ← pos(elem) (cherchée en O(n))
    posElem ← max_{1≤posElem<n et posElem non marquée} posElem
    (cherchée en O(n))
```

FIG. III.1 – Pseudo-code de notre implémentation de la construction de $G(\pi)$. La fonction pos(*elem*) retourne la position de l'élément *elem* dans π.

Un élément qui commence un cycle est trouvé au moment où on commence le cycle et au moment où on s'aperçoit que le cycle est complété parce que cet élément a déjà été sélectionné. Sa position est donc cherchée deux fois.

Un élément qui ne commence pas un cycle est cherché une seule fois.

Chaque élément est donc cherché au plus deux fois. Chercher un élément est dans $O(n)$. Il y a n éléments dans π. La complexité de la construction de $G(\pi)$ est dans $O(n^2)$.

La suite de la preuve se base sur le pseudo-code de *BP2* à l'annexe I. Tester

si $G(\pi)$ contient un cycle orienté (dont les éléments ne sont pas tous en ordre décroissant) est dans $O(n)$. On peut conclure du pseudo-code de *BP2* que calculer la transposition à faire est dans $O(n)$ qu'il s'agisse d'un cycle orienté ou non.

L'étape limitante dans la boucle *Tant que* du pseudo-code de *BP2* est donc la construction de $G(\pi)$ qui est dans $O(n^2)$. Le nombre de fois que cette boucle est répétée est dans $O(n)$ (théorème 2.5). La complexité de notre implémentation de *BP2* est donc dans $O(n^3)$.